

APUNTS D'INTEL·LIGÈNCIA ARTIFICIAL

Departament de Llenguatges i Sistemes Informàtics



FIB

Facultat d'Informàtica
de Barcelona

UNIVERSITAT POLITÈCNICA DE CATALUNYA

This work is licensed under the Creative Commons
Attribution-NonCommercial-ShareAlike License. 

To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-sa/2.0/> or
send a letter to:

Creative Commons,
559 Nathan Abbott Way, Stanford,
California 94305,
USA.

Índice general

0. Introducción	1
I Resolución de problemas	3
1. Resolución de problemas	5
1.1. ¿Qué es un problema?	5
1.2. El espacio de estados	6
1.3. Algoritmos de búsqueda en el espacio de estados	10
2. Búsqueda no informada	13
2.1. Búsqueda independiente del problema	13
2.2. Búsqueda en anchura prioritaria	13
2.3. Búsqueda en profundidad prioritaria	14
2.4. Búsqueda en profundidad iterativa	17
2.5. Ejemplos	18
3. Búsqueda heurística	23
3.1. El conocimiento importa	23
3.2. El óptimo está en el camino	23
3.3. Tú primero y tú después	24
3.4. El algoritmo A^*	25
3.5. Pero, ¿encontraré el óptimo?	27
3.5.1. Admisibilidad	27
3.5.2. Consistencia	28
3.5.3. Heurístico más informado	29
3.6. Mi memoria se acaba	29
3.6.1. El algoritmo IDA*	30
3.6.2. Otras alternativas	31
4. Búsqueda local	35
4.1. El tamaño importa, a veces	35
4.2. Tu sí, vosotros no	36
4.2.1. Ejemplo: El problema de la mochila	39
4.3. Un mundo de posibilidades	42

4.4.	Demasiado calor, demasiado frío	43
4.4.1.	Ejemplo: El viajante de comercio	44
4.5.	Cada oveja con su pareja	46
4.5.1.	Codificación	47
4.5.2.	Operadores	48
4.5.3.	Combinación de individuos	49
4.5.4.	El algoritmo genético canónico	50
4.5.5.	Cuando usarlos	50
4.5.6.	Ejemplo: Las N reinas	51
5.	Búsqueda con adversario	55
5.1.	Tú contra mi o yo contra ti	55
5.2.	Una aproximación trivial	56
5.3.	Seamos un poco más inteligentes	56
5.4.	Seamos aún más inteligentes	58
6.	Satisfacción de restricciones	63
6.1.	De variables y valores	63
6.2.	Buscando de manera diferente	64
6.2.1.	Búsqueda con backtracking	65
6.2.2.	Propagación de restricciones	67
6.2.3.	Combinando búsqueda y propagación	69
6.3.	Otra vuelta de tuerca	71
6.4.	Ejemplo: El Sudoku	71
II	Representación del conocimiento	73
7.	Introducción a la representación del conocimiento	75
7.1.	Introducción	75
7.2.	Esquema de representación	76
7.3.	Propiedades de un esquema de representación	78
7.4.	Tipos de conocimiento	78
7.4.1.	Conocimiento Relacional simple	79
7.4.2.	Conocimiento Heredable	79
7.4.3.	Conocimiento Inferible	79
7.4.4.	Conocimiento Procedimental	80
8.	Lógica	81
8.1.	Introducción	81
8.2.	Lógica proposicional	81
8.3.	Lógica de predicados	83
8.4.	Lógica y representación del conocimiento	84
9.	Sistemas de reglas de producción	85
9.1.	Introducción	85
9.2.	Elementos de un sistema de producción	85
9.3.	El motor de inferencias	87
9.3.1.	Ciclo de ejecución	88
9.4.	Tipos de razonamiento	89
9.4.1.	Razonamiento guiado por los datos	90

9.4.2. Razonamiento guiado por los objetivos	91
9.5. Las reglas como lenguaje de programación	91
9.6. Las reglas como parte de aplicaciones generales	93
10.Representaciones estructuradas	95
10.1. Introducción	95
10.2. Redes semánticas	95
10.3. Frames	96
10.3.1. Una sintaxis	98
11.Ontologías	103
11.1. Introducción	103
11.2. Necesidad de las ontologías	104
11.3. Desarrollo de una ontología	105
11.4. Proyectos de ontologías	110
III Sistemas basados en el conocimiento	113
12.Introducción a los SBC	115
12.1. Introducción	115
12.2. Características de los SBC	116
12.3. Necesidad de los SBC	117
12.4. Problemas que se pueden resolver mediante SBC	118
12.5. Problemas de los SBC	119
12.6. Áreas de aplicación de los SBC	120
12.7. Breve historia de los SBC	120
13.Arquitectura de los SBC	123
13.1. Introducción	123
13.2. Arquitectura de los sistemas basados en reglas	124
13.2.1. Almacenamiento del conocimiento	124
13.2.2. Uso e interpretación del conocimiento	125
13.2.3. Almacenamiento del estado del problema	126
13.2.4. Justificación e inspección de las soluciones	126
13.2.5. Aprendizaje	126
13.3. Arquitectura de los sistemas basados en casos	127
13.3.1. Almacenamiento del conocimiento	128
13.3.2. Uso e interpretación del conocimiento	129
13.3.3. Almacenamiento del estado del problema	129
13.3.4. Justificación e inspección de las soluciones	129
13.3.5. Aprendizaje	129
13.4. Comunicación con el entorno y/o el usuario	129
13.5. Otras Metodologías	130
13.5.1. Redes neuronales	130
13.5.2. Razonamiento basado en modelos	131
13.5.3. Agentes Inteligentes/Sistemas multiagente	133

14.Desarrollo de los SBC	135
14.1. Ingeniería de sistemas basados en el conocimiento	135
14.1.1. Modelo en cascada	135
14.1.2. Modelo en espiral	136
14.1.3. Diferencias de los sistemas basados en el conocimiento	137
14.1.4. Ciclo de vida de un sistema basado en el conocimiento	138
14.1.5. Metodologías especializadas	139
14.2. Una Metodología sencilla para el desarrollo de SBC	140
14.2.1. Identificación	140
14.2.2. Conceptualización	141
14.2.3. Formalización	142
14.2.4. Implementación	142
14.2.5. Prueba	142
15.Resolución de problemas en los SBC	143
15.1. Clasificación de los SBC	143
15.2. Métodos de resolución de problemas	145
15.2.1. Clasificación Heurística	145
15.2.2. Resolución Constructiva	150
16.Razonamiento aproximado e incertidumbre	155
16.1. Incertidumbre y falta de información	155
17.Modelo Probabilista	157
17.1. Introducción	157
17.2. Teoría de probabilidades	157
17.3. Inferencia probabilística	158
17.3.1. Probabilidades condicionadas y reglas de producción	160
17.4. Independencia probabilística y la regla de Bayes	160
17.5. Redes Bayesianas	161
17.6. Inferencia probabilística mediante redes bayesianas	163
17.6.1. Inferencia por enumeración	163
17.6.2. Algoritmo de eliminación de variables	164
18.Modelo Posibilista	169
18.1. Introducción	169
18.2. Conjuntos difusos/Lógica difusa	170
18.3. Conectivas lógicas en lógica difusa	171
18.4. Condiciones difusas	174
18.5. Inferencia difusa con datos precisos	178
18.5.1. Modelo de inferencia difusa de Mamdani	178
IV Tratamiento del lenguaje natural	183
19.Tratamiento del Lenguaje Natural	185
19.1. Introducción. El Lenguaje Natural y su tratamiento	185
19.2. El problema de la comprensión del Lenguaje Natural	186
19.3. Niveles de descripción y tratamiento lingüístico	187
19.3.1. Evolución de la lingüística computacional	188
19.3.2. Los niveles de la Descripción Lingüística.	189

19.4. Aportaciones de la inteligencia artificial	190
19.4.1. Aplicaciones del lenguaje natural	191
19.4.2. Breve historia del Tratamiento del lenguaje natural	191
19.4.3. Aplicaciones basadas en Diálogos.	192
19.4.4. Aplicaciones basadas en el tratamiento masivo de la Información Textual.	192
19.5. Técnicas de comprensión del Lenguaje Natural: Su integración	193
20.El Conocimiento Léxico	195
20.1. Introducción	195
20.2. Descripción de la Información y el Conocimiento Léxicos	195
20.2.1. La segmentación de la oración	196
20.2.2. El contenido de la información léxica	196
20.2.3. Lexicones Computacionales	200
20.2.4. Lexicones Frasales	201
20.2.5. La adquisición del Conocimiento Léxico	201
20.3. Implementaciones de los Diccionarios	202
20.4. Morfología	203
21.La dimensión sintáctica	205
21.1. Introducción	205
21.2. Las redes de transición	205
21.3. Los formalismos sintácticos: Las Gramáticas	208
21.4. Gramáticas de Estructura de Frase	209
21.5. Analizadores Básicos	211
21.5.1. La técnica del Análisis Sintáctico	212
21.5.2. Las ATNs	214
21.5.3. Los Charts	216
21.6. Los formalismos de Unificación	219
21.6.1. Introducción	219
21.6.2. Referencia Histórica	222
21.6.3. El análisis gramatical como demostración de un teorema	223
21.6.4. Las Gramáticas de Cláusulas Definidas	225
22.La interpretación semántica	227
22.1. Introducción	227
V Aprendizaje Automático	233
23.Aprendizaje Automático	235
23.1. Introducción	235
23.2. Tipos de aprendizaje	236
23.3. Aprendizaje Inductivo	236
23.3.1. Aprendizaje como búsqueda	237
23.3.2. Tipos de aprendizaje inductivo	237
23.4. Árboles de Inducción	239

Introducción

Este documento contiene las notas de clase de la asignatura Intel·ligència Artificial de la ingeniería en informàtica de la Facultat d'Informàtica de Barcelona.

El documento cubre todos los temas impartidos en la asignatura salvo el tema de introducción y esta pensado como complemento a las transparencias utilizadas en las clases. El objetivo es que se lea esta documentación antes de la clase correspondiente para que se obtenga un mejor aprovechamiento de las clases y se puedan realizar preguntas y dinamizar la clase.

En ningún caso se pueden tomar estos apuntes como un sustituto de las clases de teoría.

Parte I

Resolución de problemas

Resolución de problemas

1.1 ¿Qué es un problema?

Una de las principales capacidades de la inteligencia humana es su capacidad para resolver problemas. La habilidad para analizar los elementos esenciales de cada problema, abstraéndolos, el identificar las acciones que son necesarias para resolverlos y el determinar cual es la estrategia más acertada para atacarlos, son rasgos fundamentales que debe tener cualquier entidad inteligente. Es por eso por lo que la resolución de problemas es uno de los temas básicos en inteligencia artificial.

Podemos definir la resolución de problemas como el proceso que partiendo de unos datos iniciales y utilizando un conjunto de procedimientos escogidos a priori, es capaz de determinar el conjunto de pasos o elementos que nos llevan a lo que denominaremos una solución. Esta solución puede ser, por ejemplo, el conjunto de acciones que nos llevan a cumplir cierta propiedad o como deben combinarse los elementos que forman los datos iniciales para cumplir ciertas restricciones.

Evidentemente, existen muchos tipos diferentes de problemas, pero todos ellos tienen elementos comunes que nos permiten clasificarlos y estructurarlos. Por lo tanto, no es descabellada la idea de poder resolverlos de manera automática, si somos capaces de expresarlos de la manera adecuada.

Para que podamos resolver problemas de una manera automatizada es necesario, en primer lugar, que podamos expresar las características de los problemas de una manera formal y estructurada. Eso nos obligará a encontrar un lenguaje común que nos permita definir problemas.

En segundo lugar, hemos de definir algoritmos que representen estrategias que nos permitan hallar la solución de esos problemas, que trabajen a partir de ese lenguaje de representación y que nos garanticen hasta cierta medida que la solución que buscamos será hallada.

Si abstraemos las características que describen un problema podemos identificar los siguientes elementos:

1. Un punto de partida, los elementos que definen las características del problema.
2. Un objetivo a alcanzar, qué queremos obtener con su resolución.
3. Acciones a nuestra disposición para resolver el problema, de qué herramientas disponemos para manipular los elementos que describen el problema.
4. Restricciones sobre el objetivo, qué características debe tener la solución
5. Elementos definidos por el dominio concreto que son relevantes en el problema, qué conocimiento tenemos en el dominio que nos puede ayudar a resolver el problema de una manera eficiente.

Escogiendo estos elementos como base de la representación de un problema podemos llegar a diferentes aproximaciones, algunas generales, otras más específicas. De entre los tipos de aproximaciones podemos citar:

Espacio de estados: Se trata de la aproximación más general, un problema se divide en un conjunto de pasos de resolución que enlazan los elementos iniciales con los elementos que describen la solución, donde a cada paso podemos ver como se transforma el problema.

Reducción a subproblemas: En esta aproximación suponemos que podemos descomponer el problema global en problemas más pequeños de manera recursiva hasta llegar a problemas simples. Es necesario disponer como parte del conocimiento del problema de la forma de realizar esta descomposición.

Satisfacción de restricciones: Esta aproximación es específica para problemas que se puedan plantear como un conjunto de variables a las que se han de asignar valores cumpliendo ciertas restricciones.

Juegos: También es una aproximación específica, en la que el problema se plantea como la competición entre dos o mas agentes

La elección de la forma de representación dependerá de las características del problema, en ocasiones una metodología especializada puede ser más eficiente que una general.

1.2 El espacio de estados

La aproximación más general y más sencilla de plantear un problema es la que hemos denominado *espacio de estados*. Deberemos suponer que podemos describir el problema a partir de los elementos que intervienen en él y sus relaciones. En cada instante de su resolución estos elementos tendrán unas características y relaciones específicas que seran suficientes para determinar el momento de la resolución en el que estamos.

Denominaremos **estado** a la representación de los elementos que describen el problema en un momento dado. La cuestión principal que nos deberemos plantear será qué debemos incluir en el estado. No existen unas directrices que nos resuelvan esta cuestión, pero podemos tomar como línea general que debemos incluir lo suficiente para que, cuando obtengamos la solución, ésta pueda trasladarse al problema real, ya que una simplificación excesiva nos llevará a soluciones inservibles. En el otro extremo, representar elementos de estado que sean irrelevantes o superfluos lo único que conseguirá es aumentar innecesariamente el coste computacional de la resolución.

Distinguiremos dos estados especiales, el **estado inicial** (punto de partida) y el **estado final** (objetivo del problema).

Para poder movernos entre los diferentes estados que definen el problema, necesitaremos lo que denominaremos **operadores de transformación**. Un operador es una función de transformación sobre la representación de un estado que lo convierte en otro estado. Los operadores definirán una **relación de accesibilidad** entre estados.

Los elementos que definen un operador son:

1. **Condiciones de aplicabilidad**, las condiciones que debe cumplir un estado para que podamos aplicárselo.
2. **Función de transformación**, la transformación que ha de aplicarse al estado para conseguir un estado sucesor al actual.

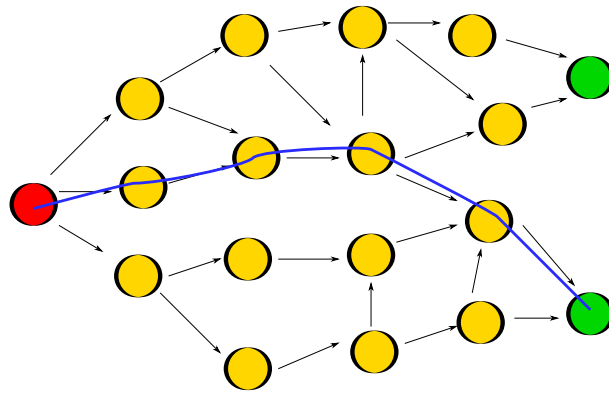


Figura 1.1: Espacio de estados

Igual que con la elección de los elementos que forman parte del estado, tampoco existen unas directrices que nos permitan decidir, de manera general, qué operadores serán necesarios para resolver el problema, cuantos serán necesarios o que nivel de granularidad han de tener (como de diferente es el estado transformado respecto al original). Se puede dar el mismo consejo que con los estados, pocos operadores pueden hacer que nuestro problema sea irresoluble o que la solución no nos sirva en el problema real, demasiados operadores pueden hacer que el coste de la resolución sea prohibitivo.

Los estados y su relación de accesibilidad conforman lo que se denomina el **espacio de estados**. Éste representa todos los caminos que hay entre todos los estados posibles de un problema. Podría asimilarse con un mapa de carreteras de un problema, la solución de nuestro problema esta dentro de ese mapa, solo nos falta hallar el camino adecuado (ver figura 1.1).

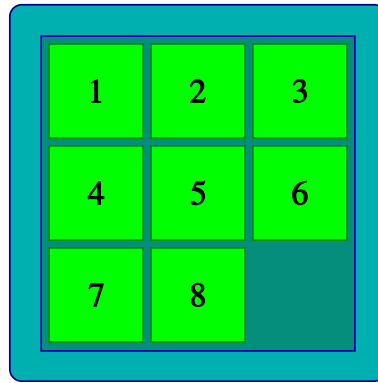
El último elemento que nos queda por definir es la **solución**. La definiremos de dos maneras, como la secuencia de pasos que llevan del estado inicial al final (secuencia de operadores) o el estado final del problema. Existen problemas en los que la secuencia de operadores es el objetivo de la solución, en éstos, por lo general, sabemos como es la solución, pero no sabemos como construirla y existen también problemas en los que queremos saber si es posible combinar los elementos del problema cumpliendo ciertas restricciones, pero la forma de averiguarlo no es importante.

También podremos catalogar los diferentes tipos problema según el tipo de solución que busquemos, dependiendo de si nos basta con una cualquiera, queremos la mejor o buscamos todas las soluciones posibles. Evidentemente el coste computacional de cada uno de estos tipos es muy diferente.

En el caso de plantearnos sobre cuantos recursos gastamos para obtener una solución, deberemos introducir otros elementos en la representación del problema. Definiremos el **coste de una solución** como la suma de los costes individuales de la aplicación de los operadores a los estados, esto quiere decir que cada operador tendrá también asociado un coste.

Vamos a ver a continuación dos ejemplos de como definir problemas como espacio de estados:

Ejemplo 1.1 *El ocho puzzle es un problema clásico que consiste en un tablero de 9 posiciones dispuestas como una matriz de 3×3 en el que hay 8 posiciones ocupadas por fichas numeradas del 1 al 8 y una posición vacía. Las fichas se pueden mover ocupando la posición vacía, si la tienen adyacente. El objetivo es partir de una disposición cualquiera de las fichas, para obtener una disposición de éstas en un orden específico. Tenemos una representación del problema en la siguiente figura:*



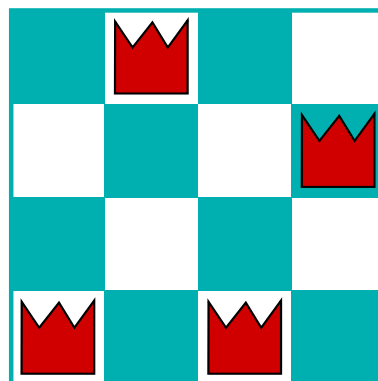
La definición de los elementos del problema para plantearlo en el espacio de estados podría ser la siguiente:

- *Espacio de estados:* Configuraciones de 8 fichas en el tablero
- *Estado inicial:* Cualquier configuración
- *Estado final:* Fichas en un orden específico
- *Operadores:* Mover el hueco
 - *Condiciones:* El movimiento está dentro del tablero
 - *Transformación:* Intercambio entre el hueco y la ficha en la posición del movimiento
- *Solución:* Que movimientos hay que hacer para llegar al estado final, posiblemente nos interesa la solución con el menor número de pasos

En esta definición hemos escogido como operador mover el hueco, evidentemente, en el problema real lo que se mueven son las fichas, pero elegir el movimiento de una ficha como operador nos habría dado 8 posibles aplicaciones con 4 direcciones posibles para cada una. Al elegir el mover hueco como operador tenemos una única aplicación con 4 direcciones posibles.

Este tipo de elecciones sobre como representar el problema pueden hacer que sea computacionalmente más o menos costoso de resolver, ya que estamos variando la forma y el tamaño del espacio de estados.

Ejemplo 1.2 El problema de las N reinas es también un problema clásico, en este caso el problema se trata de colocar N reinas en un tablero de ajedrez de $N \times N$ de manera que no se maten entre sí. En este problema no nos interesa la manera de hallar la solución, sino el estado final. En la figura tenemos representada la solución para un problema con dimensión 4:



La definición de los elementos del problema para plantearlo en el espacio de estados podría ser la siguiente:

- *Espacio de estados: Configuraciones de 0 a n reinas en el tablero con solo una por fila y columna*
- *Estado inicial: Configuración sin reinas en el tablero*
- *Estado final: Configuración en la que ninguna reina se mata entre si*
- *Operadores: Colocar una reina en una fila y columna*
 - *Condiciones: La reina no es matada por ninguna otra ya colocada*
 - *Transformación: Colocar una reina más en el tablero en una fila y columna determinada*
- *Solución: Una solución, pero no nos importan los pasos*

También podríamos haber hecho otras elecciones de representación como por ejemplo que el estado inicial tuviera las N reinas colocadas, o que el operador permitiera mover las reinas a una celda adyacente. Todas estas alternativas supondrían un coste de solución más elevado.

Un elemento que será importante a la hora de analizar los problemas que vamos a solucionar, y que nos permitirá tomar decisiones, es el *tamaño del espacio de búsqueda* y su *conectividad*. Este tamaño influirá sobre el tipo de solución que podemos esperar (si el tamaño es demasiado grande, encontrar la solución óptima puede ser irrealizable), el tipo de algoritmo que es más adecuado (habrá algoritmos más aptos para soluciones que están más lejos del estado inicial) y cuál será el coste computacional que implicará la resolución del problema.

Es esencial entonces, a la hora de plantear un problema, estimar cual es el **número de estados** que contiene. Por ejemplo, en el caso del ocho puzzle tenemos tantas combinaciones como posibles ordenes podemos hacer de las 8 piezas mas el hueco, esto significa $9!$ estados posibles (362880 estados). Probablemente no tendremos que recorrer todos los estados posibles del problema, pero nos puede dar una idea de si el problema será más o menos difícil de resolver¹.

Otro elemento a considerar es la **conectividad entre los estados**, que se puede calcular a partir del factor de ramificación de los operadores que podemos utilizar. No es lo mismo recorrer un espacio de estados en el que de un estado podemos movernos a unos pocos estados sucesores, que un espacio de estados en los que cada estado esta conectado con casi todos los estados posibles.

Esta es la razón por la que elegir operadores con un factor de ramificación no muy grande es importante. Pero debemos tener en cuenta otra cosa, un factor de ramificación pequeño puede hacer que el camino hasta la solución sea más largo. Como siempre, hemos de buscar el compromiso, tener mucha conectividad es malo, pero muy poca también puede ser malo.

En el caso del ocho puzzle, el factor de ramificación es 4 en el peor de los casos, pero en media podemos considerar que estará alrededor de 2.



Puedes tomarte unos minutos en pensar otros problemas que puedas definir según el paradigma del espacio de estados. Deberás fijarte en que los problemas representables de este modo se pueden reducir a la búsqueda de un camino.

¹Se solía comentar hace unos años que problemas que tengan menos de 2^{32} estados para los que solo nos interesa saber si existe una estado solución eran la frontera de los problemas sencillos y que se pueden resolver por fuerza bruta simplemente enumerándolos todos, con la capacidad de cálculo actual probablemente ahora ese número mínimo de estados sea algo mayor.

Algoritmo 1.1 Esquema de algoritmo de búsqueda en espacio de estados

Función: Búsqueda en espacio de estados()
Datos: El estado inicial
Resultado: Una solution
 Seleccionar el primer estado como el estado actual
mientras *estado actual* $\neq$ *estado final* **hacer**
 | Generar y guardar sucesores del estado actual (expansión)
 | Escoger el siguiente estado entre los pendientes (selección)
fin

1.3 Algoritmos de búsqueda en el espacio de estados

La resolución de un problema planteado como un espacio de estados pasa por la exploración de éste. Debemos partir del estado inicial marcado por el problema, evaluando cada paso posible y avanzando hasta encontrar un estado final. En el caso peor deberemos explorar todos los posibles caminos desde el estado inicial hasta poder llegar a un estado que cumpla las condiciones del estado final.

Para poder implementar un algoritmo capaz de explorar el espacio de estados en busca de una solución primero debemos definir una representación adecuada de éste. Las estructuras de datos más adecuadas para representar el espacio de estados son los grafos y los árboles. En estas estructuras cada nodo representará un estado del problema y los operadores de cambio de estado estarán representados por arcos. La elección de un árbol o un grafo la determinará la posibilidad de que a un estado solo se pueda llegar a través de un camino (árbol) o que haya diferentes caminos que puedan llevar al mismo estado (grafo).

Los grafos sobre los que trabajarán los algoritmos de resolución de problemas son diferentes de los que tratan los algoritmos habituales de búsqueda de caminos sobre grafos. La diferencia principal es que el grafo se construirá a medida que se haga la exploración, dado que el tamaño que puede tener hace imposible que se pueda almacenar en memoria (el grafo puede ser incluso infinito). Esto hace que, por ejemplo, los algoritmos habituales de caminos mínimos en grafos no sirvan en este tipo de problemas ya que necesitan tener todo el grafo en memoria para funcionar.

Nosotros nos vamos a centrar en los algoritmos que encuentran una solución, pero evidentemente extenderlo a todas las soluciones es bastante sencillo. En el algoritmo 1.1 se puede ver el que sería un esquema de alto nivel de un algoritmo que halla una solución de un problema en el espacio de estados.

En este algoritmo tenemos varias decisiones que nos permitirán obtener diferentes variantes que se comportarán de distinta manera y que tendrán propiedades específicas. La primera decisión es el **orden de expansión**, o lo que es lo mismo, el orden en el que generamos los sucesores de un nodo. La segunda decisión es el **orden de selección**, o sea, el orden en el que decidimos explorar los nodos que aún no hemos visitado.

Entrando un poco más en los detalles de la implementación del algoritmo, podremos distinguir dos tipos de nodos en el grafo de exploración, los **nodos abiertos**, que representarán a los estados generados pero aún no visitados y los **nodos cerrados**, que corresponderán a los estados visitados y que ya se han expandido.

Nuestro algoritmo siempre tendrá una estructura que almacenará los nodos abiertos. Las diferentes políticas de inserción de esta estructura serán las que tengan una influencia determinante sobre el tipo de búsqueda que hará el algoritmo (la estrategia de resolución).

Dado que estaremos explorando grafos, y que por lo tanto durante la exploración nos encontraremos con estados ya visitados, puede ser necesario tener una estructura para almacenar los nodos

Algoritmo 1.2 Esquema general de búsqueda**Algoritmo:** Búsqueda General

Est_abiertos.insertar(Estado inicial)

Actual ← Est_abiertos.primer()

mientras no es_final?(Actual) **y** no Est_abiertos.vacia?() **hacer**

Est_abiertos.borrar_primer()

Est_cerrados.insertar(Actual)

Hijos ← generar_sucesores(Actual)

Hijos ← tratar_repetidos(Hijos, Est_cerrados, Est_abiertos)

Est_abiertos.insertar(Hijos)

Actual ← Est_abiertos.primer()

fin

cerrados. Merecerá la pena mantenerla si el número de nodos diferentes es pequeño respecto al número de caminos, pero puede ser prohibitivo para espacios de búsqueda muy grandes.

En el algoritmo 1.2 tenemos un algoritmo algo más detallado que nos va a servir como esquema general para la mayoría de los algoritmos que iremos viendo.

Variando la estructura de nodos abiertos variaremos el comportamiento del algoritmo (orden de visita de los nodos), las funciones **primer**, **borrar_primer** e **insertar** serán diferentes según la estructura utilizada. La función **generar_sucesores** seguirá el orden de generación de sucesores definido en el problema. Este puede ser arbitrario o se puede hacer alguna ordenación usando conocimiento específico del problema. El tratamiento de repetidos dependerá de cómo se visiten los nodos, en función de la estrategia de visitas puede ser innecesario.

Para poder clasificar y analizar los algoritmos que vamos a tratar en los siguientes capítulos, escogeremos unas propiedades que nos permitirán caracterizarlos. Estas propiedades son:

- **Completitud:** Analizaremos si un algoritmo es completo en el sentido de si tenemos la garantía de que hallará una solución de existir esta. Si no lo es, existe la posibilidad de que el algoritmo no acabe o que sea incapaz de encontrarla.
- **Complejidad temporal:** Analizaremos el coste temporal de la búsqueda en función del tamaño del problema. Generalmente será una cota que es función del factor de ramificación y la profundidad a la que se encuentra la solución.
- **Complejidad espacial:** Analizaremos el espacio requerido para almacenar los nodos pendientes de explorar. Generalmente será una cota que es función del factor de ramificación y la profundidad a la que se encuentra la solución. También se puede tener en cuenta el espacio para almacenar los nodos ya explorados si es necesario para el algoritmo
- **Optimalidad:** Analizaremos si el algoritmo es capaz de encontrar la mejor solución según algún criterio de preferencia o coste.

Atendiendo a estos criterios podemos clasificar los algoritmos principales que estudiaremos en:

- **Algoritmos de búsqueda no informada:** Estos algoritmos no tienen en cuenta el coste de la solución durante la búsqueda. Su funcionamiento es sistemático, siguen un orden de visitas de nodos fijo, establecido por la estructura del espacio de búsqueda. Los principales ejemplos de estos algoritmos son el de anchura prioritaria, el de profundidad prioritaria y el de profundidad iterativa.

- **Algoritmos de búsqueda heurística y búsqueda local:** Estos algoritmos utilizan una estimación del coste o de la calidad de la solución para guiar la búsqueda, de manera que se basan en algún criterio heurístico dependiente del problema. Estos algoritmos no son sistemáticos, de manera que el orden de exploración no lo determina la estructura del espacio de búsqueda sino el criterio heurístico. Algunos de ellos tampoco son exhaustivos y basan su menor complejidad computacional en ignorar parte del espacio de búsqueda.

Existen bastantes algoritmos de este tipo con funcionamientos muy diferentes, no siempre garantizan el encontrar la solución óptima, ni tan siquiera el hallar una solución. Los principales ejemplos de estos algoritmos son A^* , IDA^* , Branch & Bound y Hill-climbing. Desarrollaremos estos algoritmos en los capítulos siguientes.



Este es un buen momento para repasar los diferentes algoritmos de recorrido de árboles y grafos y de búsqueda de caminos en grafos que se te han explicado en otras asignaturas. A pesar de que traten con estructuras finitas los principios básicos son muy parecidos.

Búsqueda no informada

2.1 Búsqueda independiente del problema

Los algoritmos de búsqueda no informada (también conocidos como algoritmos de búsqueda ciega) no dependen de información propia del problema a la hora de resolverlo. Esto quiere decir que son algoritmos generales y por lo tanto se pueden aplicar en cualquier circunstancia.

Estos algoritmos se basan en la estructura del espacio de estados y determinan estrategias sistemáticas para su exploración. Es decir, siguen una estrategia fija a la hora de visitar los nodos que representan los estados del problema. Se trata también de algoritmos exhaustivos, de manera que pueden acabar recorriendo todos los nodos del problema para hallar la solución.

Existen básicamente dos políticas de recorrido de un espacio de búsqueda, en anchura y en profundidad. Todos los algoritmos que se explicarán a continuación se basan en una de las dos.

Al ser algoritmos exhaustivos y sistemáticos su coste puede ser prohibitivo para la mayoría de los problemas reales, por lo tanto solo será aplicables en problemas pequeños. Su ventaja es que no nos hace falta obtener ningún conocimiento adicional sobre el problema, por lo que siempre son aplicables.

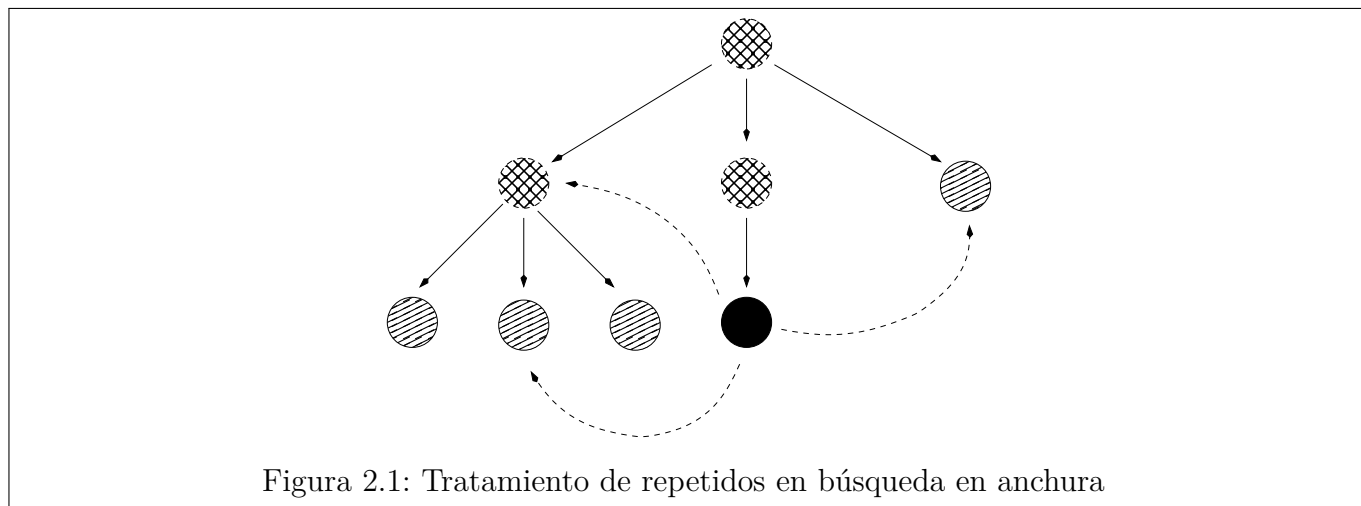
2.2 Búsqueda en anchura prioritaria

La búsqueda en anchura prioritaria intenta explorar el espacio de búsqueda haciendo un recorrido por niveles, de manera que un nodo se visita solamente cuando todos sus predecesores y sus hermanos anteriores en orden de generación ya se han visitado.

Para obtener este algoritmo solo hemos de instanciar la estructura que guarda los nodos abiertos a una cola, en el algoritmo general de búsqueda que hemos visto en el capítulo anterior. Esta estructura nos consigue que se visiten los nodos en el orden que hemos establecido.

Si nos fijamos en las propiedades que hemos escogido para clasificar los algoritmos:

- Complejidad: El algoritmo siempre encuentra una solución de haberla su sistematicidad nos garantiza llegar al nivel donde se encuentra la solución.
- Complejidad temporal: Si tomamos como medida del coste el número de nodos explorados, este crece de manera exponencial respecto al factor de ramificación y la profundidad de la solución $O(r^p)$.
- Complejidad espacial: Dado que tenemos que almacenar todos los nodos pendientes por explorar, el coste es exponencial respecto al factor de ramificación y la profundidad de la solución



$O(r^p)$. Este coste hace poco práctico a este algoritmo para problemas que no sean de tamaño pequeño.

- Optimalidad: La solución obtenida es óptima respecto al número de pasos desde la raíz, si los operadores de búsqueda tienen coste uniforme, el coste de la solución sería el óptimo.

Respecto al tratamiento de nodos repetidos, si nos fijamos en la figura 2.1 podemos ver los diferentes casos con los que nos encontramos.

El nodo de color negro sería el nodo generado actual, los nodos en cuadrícula son nodos cerrados y los nodos rayados son nodos abiertos. Si el nodo generado actual está repetido en niveles superiores (más cerca de la raíz), su coste será peor ya que su camino desde la raíz es más largo, si está al mismo nivel, su coste será el mismo. Esto quiere decir que para cualquier nodo repetido, su coste será peor o igual que algún nodo anterior visitado o no, de manera que lo podremos descartar, ya que o lo hemos expandido ya o lo haremos próximamente.

El coste espacial de guardar los nodos cerrados para el tratamiento de repetidos es también $O(r^p)$. No hacer el tratamiento de nodos repetidos no supone ninguna ganancia, ya que tanto la cola de nodos abiertos como la de nodos repetidos crecen exponencialmente.

2.3 Búsqueda en profundidad prioritaria

La búsqueda en profundidad prioritaria intenta seguir un camino hasta la mayor profundidad posible, retrocediendo cuando acaba el camino y retomando la última posibilidad de elección disponible.

Para obtener este algoritmo solo hemos de instanciar la estructura que guarda los nodos abiertos a una pila en el algoritmo genérico. Esta estructura nos consigue que se visiten los nodos en el orden que hemos establecido.

El principal problema de este algoritmo es que no acaba si existe la posibilidad de que haya caminos infinitos. Una variante de este algoritmo que evita este problema es el algoritmo de **profundidad limitada**, éste impone un límite máximo de profundidad que determina la longitud máxima de los caminos recorridos. Esta limitación garantiza que el algoritmo acaba, pero no garantiza encontrar la solución, ya que ésta puede estar a mayor profundidad que el límite impuesto.

El algoritmo de profundidad limitada se puede ver en el algoritmo 2.1. La única diferencia en la implementación de esta variante con el algoritmo general es que se dejan de generar sucesores cuando se alcanza la profundidad límite.

Si nos fijamos en las propiedades que hemos escogido para clasificar los algoritmos:

Algoritmo 2.1 Algoritmo de profundidad limitada

Procedimiento: Búsqueda en profundidad limitada (límite: entero)

```

Est_abiertos.insertar(Estado inicial)
Actual ← Est_abiertos.primer()
mientras no es_final?(Actual) y no Est_abiertos.vacia?() hacer
    Est_abiertos.borrar_primer()
    Est_cerrados.insertar(Actual)
    si profundidad(Actual) ≤ límite entonces
        Hijos ← generar_sucesores (Actual)
        Hijos ← tratar_repetidos (Hijos, Est_cerrados, Est_abiertos)
        Est_abiertos.insertar(Hijos)
    fin
    Actual ← Est_abiertos.primer()
fin

```

- Completitud: El algoritmo encuentra una solución si se impone una profundidad límite y existe una solución dentro de ese límite.
- Complejidad temporal: Si tomamos como medida del coste el número de nodos explorados, el coste es exponencial respecto al factor de ramificación y la profundidad del límite de exploración $O(r^p)$.
- Complejidad espacial: El coste es lineal respecto al factor de ramificación y el límite de profundidad $O(rp)$. Si hacemos una implementación recursiva del algoritmo el coste es $O(p)$, ya que no nos hace falta generar todos los sucesores de un nodo, pudiéndolos tratar uno a uno. Si tratamos los nodos repetidos el coste espacial es igual que en anchura ya que tendremos que guardar todos los nodos cerrados.
- Optimalidad: No se garantiza que la solución sea óptima, la solución que se retornará será la primera en el orden de exploración.



Las implementaciones recursivas de los algoritmos permiten por lo general un gran ahorro de espacio. En este caso la búsqueda en profundidad se puede realizar recursivamente de manera natural (ver algoritmo 2.2). La recursividad permite que las alternativas queden almacenadas en la pila de ejecución como puntos de continuación del algoritmo sin necesidad de almacenar ninguna información. En este caso el ahorro de espacio es proporcional al factor de ramificación que en la práctica en problemas difíciles no es despreciable. Lo único que perdemos es la capacidad de buscar repetidos en los nodos pendientes que tenemos en el camino recorrido, pero habitualmente la limitación de memoria es una restricción bastante fuerte que impide solucionar problemas con longitudes de camino muy largas.

Respecto al tratamiento de nodos repetidos, si nos fijamos en la figura 2.2 podemos ver los diferentes casos con los que nos encontramos. El nodo de color negro sería el nodo generado actual, los nodos en cuadrícula son nodos cerrados y los nodos rayados son nodos abiertos. Si el nodo generado actual está repetido en niveles superiores (más cerca de la raíz), su coste será peor ya que su camino desde la raíz es más largo, si está al mismo nivel, su coste será el mismo. En estos dos casos podemos olvidarnos de este nodo.

En el caso de que el repetido corresponda a un nodo de profundidad superior, significa que hemos llegado al mismo estado por un camino más corto, de manera que deberemos mantenerlo y continuar su exploración, ya que nos permitirá llegar a mayor profundidad que antes.

Algoritmo 2.2 Algoritmo de profundidad limitada recursivo

Función: Búsqueda en profundidad limitada recursiva (actual:nodo, limite: entero)

```

si profundidad(Actual) ≤ limite entonces
  para todo nodo ∈ generar_sucesores (Actual) hacer
    si es_final?(nodo) entonces
      retorna (nodo)
    sino
      resultado ← Búsqueda en profundidad limitada recursiva(nodo,limite)
      si es_final?(resultado) entonces
        retorna (resultado)
      fin
    fin
  fin
sino
  retorna (∅)
fin

```

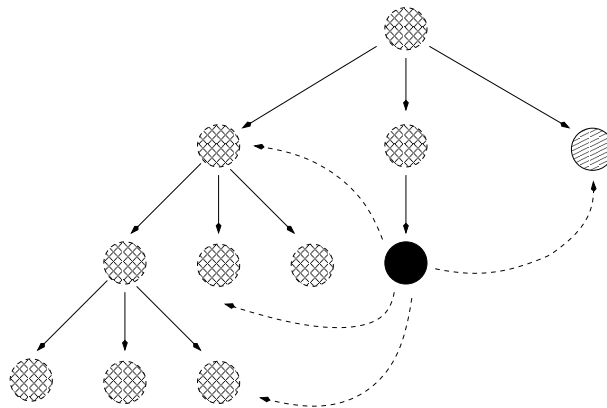


Figura 2.2: Tratamiento de repetidos en búsqueda en profundidad

En el caso de la búsqueda en profundidad, el tratamiento de nodos repetidos no es crucial ya que al tener un límite en profundidad los ciclos no llevan a caminos infinitos. No obstante, este caso se puede tratar comprobando los nodos en el camino actual ya que está completo en la estructura de nodos abiertos. Además, no tratando repetidos mantenemos el coste espacial lineal, lo cual es una gran ventaja.

El evitar tener que tratar repetidos y tener un coste espacial lineal supone una característica diferenciadora de hace muy ventajosa a la búsqueda en profundidad. Este algoritmo será capaz de obtener soluciones que se encuentren a gran profundidad.

En un problema concreto, el algoritmo de búsqueda en anchura tendrá una estructura de nodos abiertos de tamaño $O(r^p)$, lo que agotará rápidamente la memoria impidiendo continuar la búsqueda, pero solo habiendo alcanzado una profundidad de camino de p . En cambio el algoritmo en profundidad solo tendrá $O(rp)$ nodos abiertos al llegar a esa misma profundidad.

Esto quiere decir que en problemas difíciles la estrategia en profundidad será la única capaz de hallar una solución con un espacio de memoria limitado.

2.4 Búsqueda en profundidad iterativa

Algoritmo 2.3 Algoritmo de profundidad iterativa (Iterative Deepening)

```

Procedimiento: Búsqueda en profundidad iterativa (limite: entero)
prof ← 1
Actual ← Estado inicial
mientras no es_final?(Actual) y prof < limite hacer
    Est_abiertos.inicializar()
    Est_abiertos.insertar(Estado inicial)
    Actual ← Est_abiertos.primer()
    mientras no es_final?(Actual) y no Est_abiertos.vacia?() hacer
        Est_abiertos.borrar_primer()
        Est_cerrados.insertar(Actual)
        si profundidad(Actual) ≤ prof entonces
            Hijos ← generar_sucesores (Actual)
            Hijos ← tratar_repetidos (Hijos, Est_cerrados, Est_abiertos)
            Est_abiertos.insertar(Hijos)
        fin
        Actual ← Est_abiertos.primer()
    fin
    prof ← prof+1
fin

```

Este algoritmo intenta obtener las propiedades ventajosas de la búsqueda en profundidad y en anchura combinadas, es decir, un coste espacial lineal y asegurar que la solución será óptima respecto a la longitud del camino.

Para obtener esto lo que se hace es repetir sucesivamente el algoritmo de profundidad limitada, aumentando a cada iteración la profundidad máxima a la que le permitimos llegar.

Este algoritmo obtendría el mismo efecto que el recorrido en anchura en cada iteración, ya que a cada paso recorreremos un nivel más del espacio de búsqueda. El coste espacial será lineal ya que cada iteración es un recorrido en profundidad. Para que el algoritmo acabe en el caso de que no haya solución se puede imponer un límite máximo de profundidad en la búsqueda.

Aparentemente podría parecer que este algoritmo es más costoso que los anteriores al tener que repetir en cada iteración la búsqueda anterior, pero si pensamos en el número de nodos nuevos que recorreremos a cada iteración, estos son siempre tantos como todos los que hemos recorrido en todas las iteraciones anteriores, por lo que las repeticiones suponen un factor constante respecto a los que recorreríamos haciendo solo la última iteración.

El algoritmo de profundidad iterativa se puede ver en el algoritmo [2.3](#).

Si nos fijamos en las propiedades que hemos escogido para clasificar los algoritmos:

- Complejidad: El algoritmo siempre encontrará la solución de existir.
- Complejidad temporal: La misma que la búsqueda en anchura. El regenerar el árbol en cada iteración solo añade un factor constante a la función de coste $O(r^p)$.
- Complejidad espacial: Igual que en la búsqueda en profundidad ($O(r^p)$).
- Optimalidad: La solución es óptima igual que en la búsqueda en anchura.

Igual que en el caso del algoritmo en profundidad, el tratar repetidos acaba con todas las ventajas espaciales del algoritmo, por lo que es aconsejable no hacerlo. Como máximo se puede utilizar la estructura de nodos abiertos para detectar bucles en el camino actual.

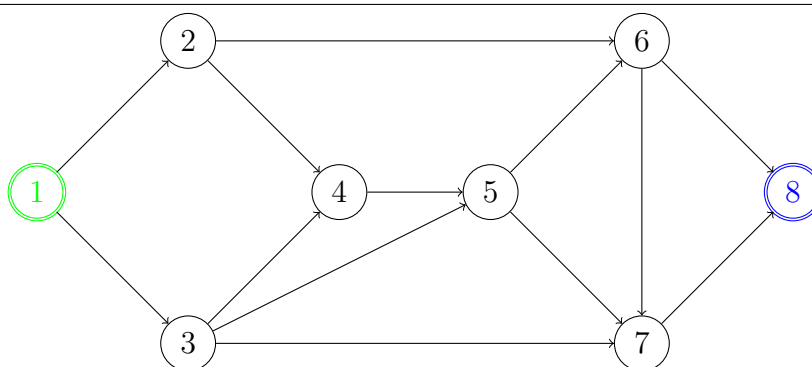


Figura 2.3: Ejemplo de grafo

Si el algoritmo en profundidad limitada es capaz de encontrar soluciones a mayor profundidad, este algoritmo además nos garantizará que la solución será óptima. Por lo tanto este es el algoritmo más ventajoso de los tres.

2.5 Ejemplos

En esta sección vamos a ver ejemplos de ejecución de los algoritmos vistos en estos capítulos. En la figura 2.3 podemos ver el grafo que vamos a utilizar en los siguientes ejemplos. Este grafo tiene como nodo inicial el 1 y como nodo final el 8, todos los caminos son dirigidos y tienen el mismo coste. Utilizaremos los algoritmos de búsqueda ciega para encontrar un camino entre estos dos nodos.

Supondremos que hacemos un tratamiento de los nodos repetidos durante la ejecución del algoritmo. Hay que tener en cuenta también que para poder recuperar el camino, los nodos deben tener un enlace al padre que los ha generado.

Búsqueda en anchura

1. Este algoritmo comenzaría encolando el nodo inicial (nodo 1).
2. Los nodos sucesores del primer nodo son los nodos 2 y 3, que pasarían a ser encolados
3. El siguiente nodo extraído de la cola sería el 2, que tiene como sucesores el 4 y el 6.
4. El siguiente nodo extraído de la cola sería el 3, que tiene como sucesores el 4, el 5 y el 7. Como el 4 está repetido no se encolaría al ser de la misma profundidad que el nodo repetido.
5. El siguiente nodo extraído de la cola sería el 4, que tiene como sucesor el 5. Como el 4 está repetido no se encolaría al tener profundidad mayor que el nodo repetido.
6. El siguiente nodo extraído de la cola sería el 6, que tiene como sucesores el 7 y el 8. Como el 7 está repetido no se encolaría al tener profundidad mayor que el nodo repetido.
7. El siguiente nodo extraído de la cola sería el 5, que tiene como sucesores el 6 y el 7. Como los dos nodos corresponden a nodos repetidos con profundidad menor, no se encolarían.
8. El siguiente nodo extraído de la cola sería el 7, que tiene como sucesor el 8. Como el 8 está repetido no se encolaría al tener la misma profundidad que el nodo repetido.
9. El siguiente nodo extraído de la cola sería el 8 que es el nodo solución y por tanto acabaría la ejecución.

En la figura 2.4 se puede ver el árbol de búsqueda que se genera. En el caso de los nodos cerrados necesitamos una estructura adicional para controlar los nodos repetidos. resultado)

Búsqueda en profundidad

1. Este algoritmo comenzaría empilando el nodo inicial (nodo 1).
2. Los nodos sucesores del primer nodo son los nodos 2 y 3, que pasarían a ser empilados
3. El siguiente nodo extraído de la pila sería el 2, que tiene como sucesores el 4 y el 6.
4. El siguiente nodo extraído de la pila sería el 4, que tiene como sucesor el 5.
5. El siguiente nodo extraído de la pila sería el 5, que tiene como sucesores el 6 y el 7. Como el nodo 6 está en la pila en con nivel superior descartaríamos este nodo.
6. El siguiente nodo extraído de la pila sería el 7, que tiene como sucesor el 8.
7. El siguiente nodo extraído de la pila sería el 8 que es el nodo solución y por tanto acabaría la ejecución.

En la figura 2.5 se puede ver el árbol de búsqueda que se genera. Fijaos que en este caso el camino es más largo que el encontrado con el algoritmo anterior. De hecho es el primer camino solución que se encuentra en la exploración según la ordenación que se hace de los nodos. Para el control de repetidos, si se quiere ahorrar espacio, pueden solo tenerse en cuenta los nodos que hay en la pila como se ha hecho en el ejemplo, esto evitará que podamos entrar en bucle y que tengamos que tener una estructura para los nodos cerrados.

Búsqueda en profundidad iterativa

1. Primera iteración (profundidad 1)
 - 1.1 Se comenzaría empilando el nodo inicial (nodo 1).
 - 1.2 Al extraer este nodo de la pila habríamos visitado todos los caminos de profundidad 1 y con eso acabaríamos la iteración.
2. Segunda Iteración (profundidad 2)
 - 2.1 Se comenzaría empilando el nodo inicial (nodo 1).
 - 2.2 Los nodos sucesores del primer nodo son los nodos 2 y 3, que pasarían a ser empilados.
 - 2.3 El siguiente nodo extraído de la pila sería el 2, los nodos sucesores estarían a mayor profundidad que el límite actual, no los empilaríamos.
 - 2.3 El siguiente nodo extraído de la pila sería el 3, los nodos sucesores estarían a mayor profundidad que el límite actual, no los empilaríamos. Al extraer este nodo de la pila habríamos visitado todos los caminos de profundidad 2 y con eso acabaríamos la iteración.
3. Tercera Iteración (profundidad 3)
 - 3.1 Se comenzaría empilando el nodo inicial (nodo 1).
 - 3.2 Los nodos sucesores del primer nodo son los nodos 2 y 3, que pasarían a ser empilados.
 - 3.3 El siguiente nodo extraído de la pila sería el 2, que tiene como sucesores el 4 y el 6.
 - 3.4 El siguiente nodo extraído de la pila sería el 4, los nodos sucesores estarían a mayor profundidad que el límite actual, no los empilaríamos.

- 3.5 El siguiente nodo extraído de la pila sería el 6, los nodos sucesores estarían a mayor profundidad que el límite actual, no los empilaríamos.
- 3.6 El siguiente nodo extraído de la pila sería el 3, que tiene como sucesores el 4, el 5 y el 7.
- 3.7 El siguiente nodo extraído de la pila sería el 4, los nodos sucesores estarían a mayor profundidad que el límite actual, no los empilaríamos.
- 3.8 El siguiente nodo extraído de la pila sería el 5, los nodos sucesores estarían a mayor profundidad que el límite actual, no los empilaríamos.
- 3.4 El siguiente nodo extraído de la pila sería el 7, los nodos sucesores estarían a mayor profundidad que el límite actual, no los empilaríamos. Al extraer este nodo de la pila habríamos visitado todos los caminos de profundidad 3 y con eso acabaríamos la iteración.

4. Cuarta Iteración (profundidad 4)

- 4.1 Se comenzaría empilando el nodo inicial (nodo 1).
- 4.2 Los nodos sucesores del primer nodo son los nodos 2 y 3, que pasarían a ser empilados.
- 4.3 El siguiente nodo extraído de la pila sería el 2, que tiene como sucesores el 4 y el 6.
- 4.4 El siguiente nodo extraído de la pila sería el 4, que tiene como sucesor el 5.
- 4.5 El siguiente nodo extraído de la pila sería el 5, los nodos sucesores estarían a mayor profundidad que el límite actual, no los empilaríamos.
- 4.6 El siguiente nodo extraído de la pila sería el 6, que tiene como sucesores el 7 y el 8.
- 4.7 El siguiente nodo extraído de la pila sería el 7, los nodos sucesores estarían a mayor profundidad que el límite actual, no los empilaríamos.
- 4.8 El siguiente nodo extraído de la pila sería el 8 que es el nodo solución y por tanto acabaría la ejecución.

En la figura 2.6 se puede ver el árbol de búsqueda que se genera. En este caso encontramos el camino mas corto, ya que este algoritmo explora los caminos en orden de longitud como la búsqueda en anchura. Para mantener la ganancia en espacio en este caso no se han guardado los nodos cerrados, es por ello que no se ha tratado en nodo 4 en el paso 3.7 como repetido.

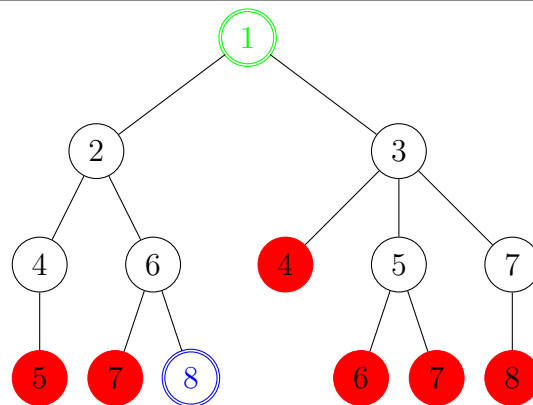


Figura 2.4: Ejecución del algoritmo de búsqueda en anchura

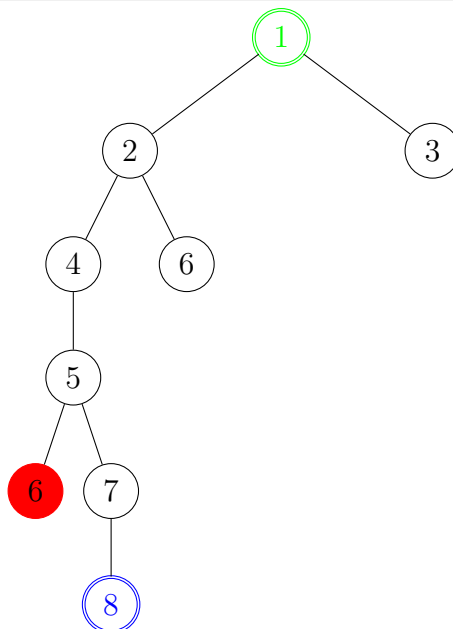


Figura 2.5: Ejecución del algoritmo de búsqueda en profundidad

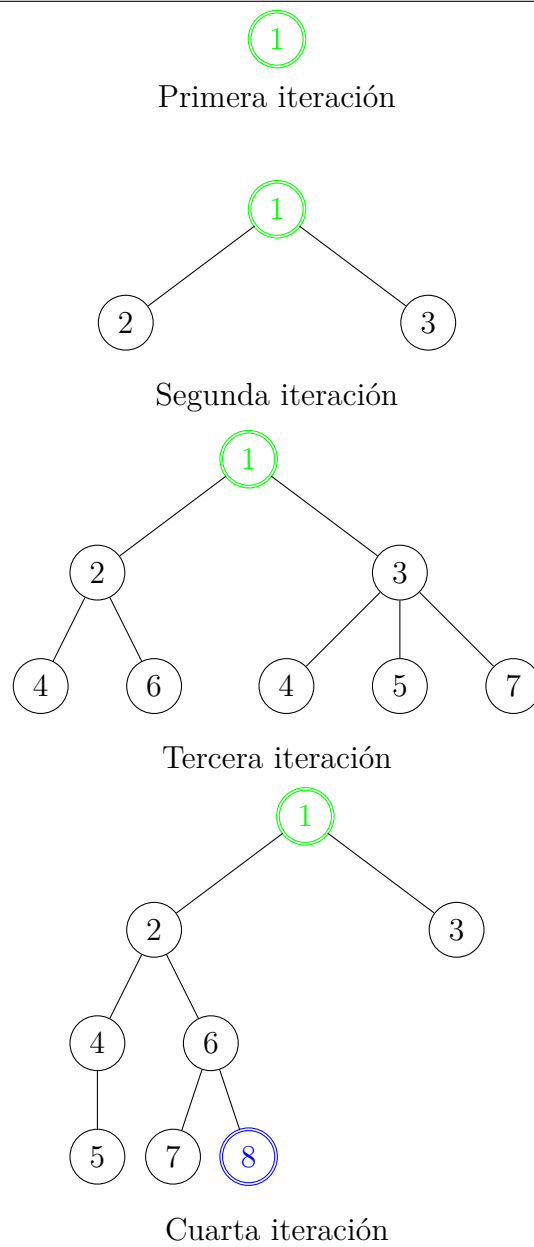


Figura 2.6: Ejecución del algoritmo de búsqueda IDA

Búsqueda heurística

3.1 El conocimiento importa

Es evidente que los algoritmos de búsqueda no informada serán incapaces de encontrar soluciones en problemas en los que el tamaño del espacio de búsqueda sea grande. Todos estos algoritmos tienen un coste temporal que es una función exponencial del tamaño de la entrada, por lo tanto el tiempo para encontrar la mejor solución a un problema no es asumible en problemas reales.

La manera de conseguir reducir este tiempo de búsqueda a algo más razonable es intentar hacer intervenir conocimiento sobre el problema que queremos resolver dentro del funcionamiento del algoritmo de búsqueda.

La dificultad que tendremos es que este conocimiento será particular para cada problema, por lo tanto no será exportable a otros. Perderemos en generalidad, pero podremos ganar en eficiencia temporal.

En este capítulo nos centraremos en los algoritmos que buscan la solución óptima. Este objetivo solo es abordable hasta cierto tamaño de problemas, existirá todo un conjunto de problemas en los que la búsqueda del óptimo será imposible por el tamaño de su espacio de búsqueda o por la imposibilidad de encontrar información que ayude en la búsqueda.

3.2 El óptimo está en el camino

Para poder plantear la búsqueda del óptimo siempre hemos que tener alguna medida del coste de obtener una solución. Este coste lo mediremos sobre el camino que nos lleva desde el estado inicial del problema hasta el estado final. No todos los problemas se pueden plantear de esta forma tal y como veremos en el próximo capítulo.

Por lo tanto, supondremos que podemos plantear nuestra búsqueda como la búsqueda de un camino y que de alguna manera somos capaces de saber o estimar cuál es la longitud o coste de éste. Por lo general, tendremos un coste asociado a los operadores que nos permiten pasar de un estado a otro, por lo que ese coste tendrá un papel fundamental en el cálculo del coste del camino.

Los algoritmos que veremos utilizarán el cálculo del coste de los caminos explorados para saber qué nodos merece la pena explorar antes. De esta manera, perderemos la sistematicidad de los algoritmos de búsqueda no informada, y el orden de visita de los estados de búsqueda no vendrá determinado por su posición en el grafo de búsqueda, sino por su coste.

Dado que el grafo va siendo generado a medida que lo exploramos, podemos ver que tenemos dos elementos que intervienen en el coste del camino hasta la solución que buscamos. En primer lugar, tendremos el coste del camino recorrido, que podremos calcular simplemente sumando los costes de

Algoritmo 3.1 Algoritmo Greedy Best First

Algoritmo: Greedy Best First

Est_abiertos.insertar(Estado inicial)

Actual ← Est_abiertos.primer()

mientras no *es_final?*(Actual) **y** no *Est_abiertos.vacía?*() **hacer**

Est_abiertos.borrar_primer()

Est_cerrados.insertar(Actual)

 hijos ← **generar_sucesores** (Actual) hijos ← **tratar_repetidos** (Hijos, Est_cerrados, Est_abiertos)

Est_abiertos.insertar(Hijos)

Actual ← Est_abiertos.primer()

fin

los operadores aplicados desde el estado inicial hasta el nodo actual. En segundo lugar, tendremos el coste más difícil de determinar, el del camino que nos queda por recorrer hasta el estado final. Dado que lo desconocemos, tendremos que utilizar el conocimiento del que disponemos del problema para obtener una aproximación.

Evidentemente, la calidad de este conocimiento que nos permite predecir el coste futuro, hará más o menos exitosa nuestra búsqueda. Si nuestro conocimiento fuera perfecto, podríamos dirigirnos rápidamente hacia el objetivo descartando todos los caminos de mayor coste, en este extremo podríamos encontrar nuestra solución en tiempo lineal. En el otro extremo, si estuviéramos en la total ignorancia, tendríamos que explorar muchos caminos antes de hallar la solución óptima, todos en el peor caso. Esta última situación es en la que se encuentra la búsqueda no informada y desgraciadamente la primera situación es rara.

Quedará pues, como labor fundamental en cada problema, obtener una función que nos haga el cálculo del coste futuro desde un estado al estado solución. Cuanto más podamos ajustarlo al coste real, mejor funcionarán los algoritmos que veremos a continuación.

3.3 Tú primero y tú después

El fundamento de los algoritmos de búsqueda heurística será el modo de elegir qué nodo explorar primero, para ello podemos utilizar diferentes estrategias que nos darán diferentes propiedades.

Una primera estrategia que se nos puede ocurrir es utilizar la estimación del coste futuro para decidir qué nodos explorar primero. De esta manera supondremos que los nodos que aparentemente están más cerca de la solución formarán parte del camino hasta la solución óptima y, por lo tanto, la encontraremos antes si los exploramos en primer lugar.

Esta estrategia se traduce en el algoritmo *primero el mejor avaricioso* (**greedy best first**). La única diferencia respecto a los algoritmos que hemos visto es el utilizar como estructura para almacenar los nodos abiertos una cola con prioridad. La prioridad de los nodos la marca la estimación del coste del camino del nodo hasta el nodo solución. El algoritmo 3.1 es su implementación.

El explorar antes los nodos más cercanos a la solución probablemente nos hará encontrarla antes, pero el no tener en cuenta el coste del camino recorrido hace que no se garantice la solución óptima, ya que a pesar de guiarnos hacia el nodo aparentemente más cercano a la solución, estamos ignorando el coste del camino completo.

3.4 El algoritmo A*

Dado que nuestro objetivo no es solo llegar lo mas rápidamente a la solución, sino encontrar la de menor coste tendremos que tener en cuenta el coste de todo el camino y no solo el camino por recorrer. Para poder introducir el siguiente algoritmo y establecer sus propiedades tenemos primero que dar una serie de definiciones.

Denominaremos el **coste de un arco** entre dos nodos n_i y n_j al coste del operador que nos permite pasar de un nodo al otro, y lo denotaremos como $c(n_i, n_j)$. Este coste siempre será positivo.

Denominaremos **coste de un camino** entre dos nodos n_i y n_j a la suma de los costes de todos los arcos que llevan desde un nodo al otro y lo denotaremos como

$$C(n_i, n_j) = \sum_{x=i}^{j-1} c(n_x, n_{x+1})$$

Denominaremos **coste del camino mínimo** entre dos nodos n_i y n_j al camino de menor coste de entre los que llevan desde un nodo al otro y lo denotaremos como

$$K(n_i, n_j) = \min_{k=1}^l C_k(n_i, n_j)$$

Si n_j es un nodo terminal, llamaremos $h^*(n_i)$ a $K(n_i, n_j)$, es decir, el coste del camino mínimo desde un estado cualquiera a un estado solución.

Si n_i es un nodo inicial, llamaremos $g^*(n_j)$ a $K(n_i, n_j)$, es decir, el coste del camino mínimo desde el estado inicial a un estado cualquiera.

Esto nos permite definir el coste del camino mínimo que pasa por cualquier nodo como una combinación del coste del camino mínimo desde el nodo inicial al nodo mas el coste del nodo hasta el nodo final:

$$f^*(n) = g^*(n) + h^*(n)$$

Dado que desde un nodo específico n el valor de $h^*(n)$ es desconocido, lo substituiremos por una función que nos lo aproximará, a esta función la denotaremos $h(n)$ y le daremos el nombre de **función heurística**. Esta función tendrá siempre un valor mayor o igual que cero. Denominaremos $g(n)$ al coste del camino desde el nodo inicial al nodo n , evidentemente ese coste es conocido ya que ese camino lo hemos recorrido en nuestra exploración. De esta manera tendremos una estimación del coste del camino mínimo que pasa por cierto nodo:

$$f(n) = g(n) + h(n)$$

Será este valor el que utilicemos para decidir en nuestro algoritmo de búsqueda cuál es el siguiente nodo a explorar de entre todos los nodos abiertos disponibles. Para realizar esa búsqueda utilizaremos el algoritmo que denominaremos A* (algoritmo 3.2).

Como se observará, el algoritmo es el mismo que el **primero el mejor avaricioso**, lo único que cambia es que ahora la ordenación de los nodos se realiza utilizando el valor de f . Por convención, consideraremos como criterio de ordenación, que a igual valor de f los nodos con h más pequeña se explorarán antes (simplemente porque si la h es más pequeña es que son nodos mas cerca de la solución), a igual h se consideraran en el orden en el que se introdujeron en la cola.

Nunca está de más el remarcar que el algoritmo solo acaba cuando se **extrae** una solución de la cola. Es posible que en cierto momento ya haya en la estructura de nodos abiertos nodos solución, pero hasta que no se hayan explorado los nodos por delante de ellos, no podemos asegurar que realmente sean soluciones buenas. Siempre hay que tener en mente que los nodos están ordenados por el coste estimado del camino total, si la estimación es menor es que podrían pertenecer a un camino con una solución mejor.

Algoritmo 3.2 Algoritmo A***Algoritmo:** A*

```

Est_abiertos.insertar(Estado inicial)
Actual ← Est_abiertos.primer()
mientras no es_final?(Actual) y no Est_abiertos.vací?() hacer
    Est_abiertos.borrar_primer()
    Est_cerrados.insertar(Actual)
    hijos ← generar_sucesores (Actual)
    hijos ← tratar_repetidos (Hijos, Est_cerrados, Est_abiertos)
    Est_abiertos.insertar(Hijos)
    Actual ← Est_abiertos.primer()
fin

```

Hay que considerar que el coste temporal de este algoritmo es también $O(r^p)$ en el caso peor. Si por ejemplo, la función $h(n)$ fuera siempre 0, el algoritmo se comportaría como una búsqueda en anchura gobernada por el coste del camino recorrido.

Lo que hace que este algoritmo pueda tener un coste temporal inferior es la bondad de la función h . De hecho, podemos interpretar las funciones g y h como las que gobiernan el comportamiento en anchura o profundidad del algoritmo.

Cuanto más cercana al coste real sea h , mayor será el comportamiento en profundidad del algoritmo, pues los nodos que aparentemente están más cerca de la solución se explorarán antes. En el momento en el que esa información deje de ser fiable, el coste del camino ya explorado hará que otros nodos menos profundos tengan un coste total mejor y, por lo tanto, se abrirá la búsqueda en anchura.

Si pensamos un poco en el coste espacial que tendrá el algoritmo, podemos observar que como éste puede comportarse como un algoritmo de búsqueda en anchura, el coste espacial en el caso peor también será $O(r^p)$. Un resultado teórico a tener en cuenta es que el coste espacial del algoritmo A* crece exponencialmente a no ser que se cumpla que:

$$|h(n) - h^*(n)| < O(\log(h^*(n)))$$

Esto pondrá el límite respecto a cuando podemos obtener una solución óptima para un problema. Si no podemos encontrar una función heurística suficientemente buena, deberemos usar algoritmos que no busquen el óptimo, pero que nos garanticen que nos acercaremos lo suficiente a él para obtener una buena solución.

El tratamiento de nodos repetidos en este algoritmo se realiza de la siguiente forma:

- Si es un nodo repetido que está en la estructura de nodos abiertos
 - Si su coste es menor substituímos el coste por el nuevo, esto podrá variar su posición en la estructura de abiertos
 - Si su coste es igual o mayor nos olvidamos del nodo
- Si es un nodo repetido que está en la estructura de nodos cerrados
 - Si su coste es menor reabrimos el nodo insertándolo en la estructura de abiertos con el nuevo coste, no hacemos nada con sus sucesores, ya se reabrirán si hace falta
 - Si su coste es mayor o igual nos olvidamos del nodo

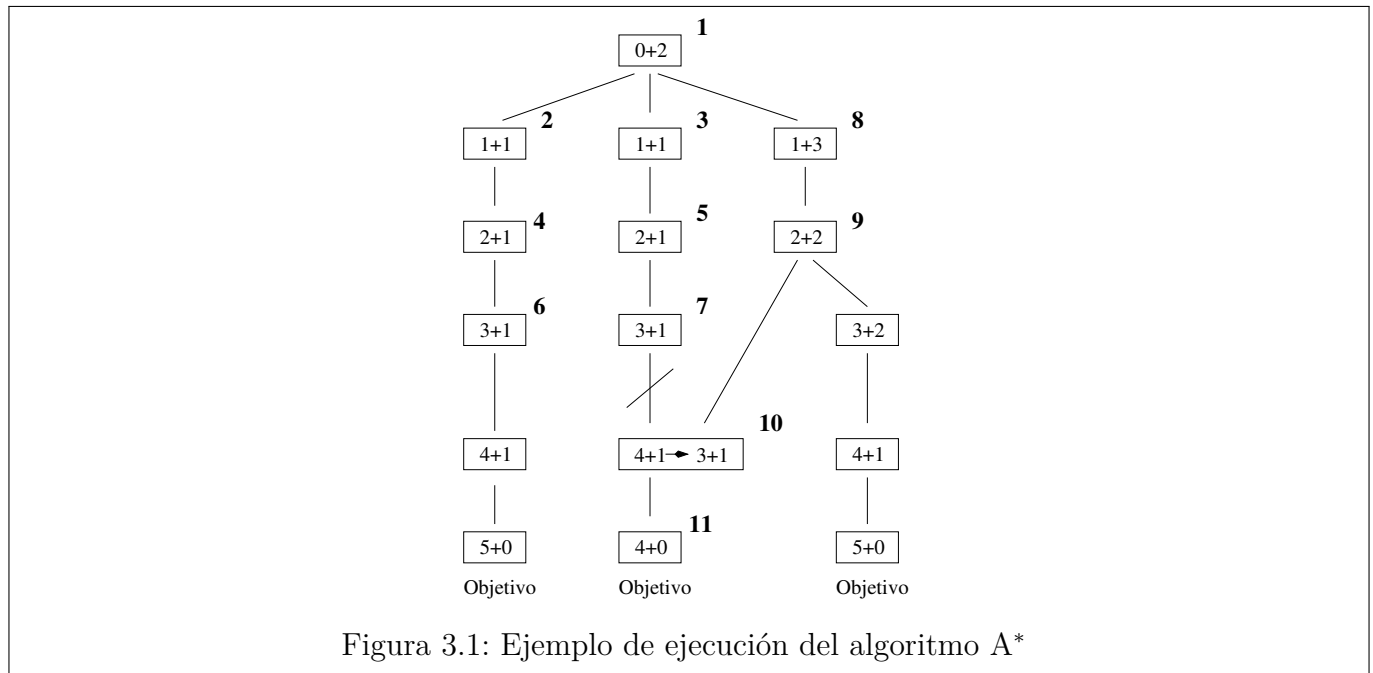


Figura 3.1: Ejemplo de ejecución del algoritmo A*

Mas adelante veremos que si la función h cumple ciertas condiciones podemos evitar el tratamiento de repetidos.

A continuación podemos ver un pequeño ejemplo de ejecución del algoritmo A*

Ejemplo 3.1 Si consideramos el árbol de búsqueda de la figura 3.1, donde en cada nodo tenemos descompuesto el coste en g y h y marcado el orden en el que el algoritmo ha visitado cada nodo.

La numeración en los nodos indica el orden en que el algoritmo A* los ha expandido (y por lo tanto el orden en el que han sido extraídos de la cola).

Podemos observar también que hay un nodo abierto que es revisitado por otro camino con un coste inferior que substituye al encontrado con anterioridad.

3.5 Pero, ¿encontraré el óptimo?

Hasta ahora no hemos hablado para nada sobre como garantizar la optimalidad de la solución. Hemos visto que en el caso degenerado tenemos una búsqueda en anchura guiada por el coste del camino explorado, eso nos debería garantizar el óptimo en este caso. Pero como hemos comentado, la función h nos permite obtener un comportamiento de búsqueda en profundidad y sabemos que en este caso no se nos garantiza el óptimo.

El saber si encontraremos o no el óptimo mediante el algoritmo A* recae totalmente en las propiedades que cumple la función heurística, si esta cumple ciertos criterios sabremos que encontraremos la solución óptima, si no los cumple, no lo podremos saber.

3.5.1 Admisibilidad

La propiedad clave que nos garantizará el hallar la solución óptima es la que denominaremos **admisibilidad**. Diremos que una función heurística h es admisible siempre que se cumpla que su valor en cada nodo sea menor o igual que el valor del coste real del camino que nos falta por recorrer hasta la solución, formalmente:

$$\forall n \quad 0 \leq h(n) \leq h^*(n)$$

Esto quiere decir que la función heurística ha de ser un estimador *optimista* del coste que falta para llegar a la solución.

Ejemplo 3.2 En la figura 3.2 podemos ver dos ejemplos de grafos de búsqueda, uno con una función admisible y otra que no lo es.

En el primer caso, la función heurística siempre es admisible, pero en la primera rama el coste es más pequeño que el real, por lo que pierde cierto tiempo explorándola (el efecto en profundidad que hemos comentado), pero al final el algoritmo pasa a la segunda rama hallando la solución.

En el segundo caso el algoritmo acabaría al encontrar la solución en G, a pesar de que haya una solución con un coste más pequeño en H. Esto es así porque el coste estimado que da en el nodo D es superior al real y eso le relega en la cola de nodos abiertos.

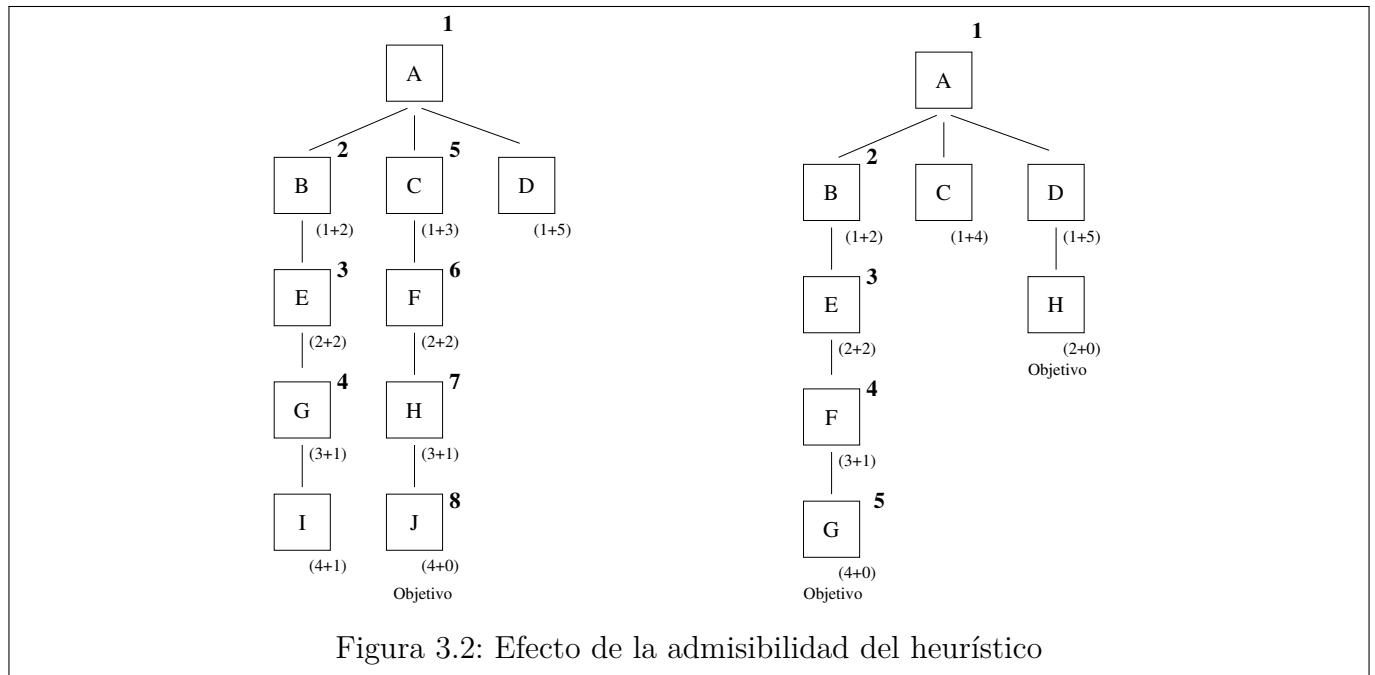


Figura 3.2: Efecto de la admisibilidad del heurístico

Esta propiedad implica que, si podemos demostrar que la función de estimación h que utilizamos en nuestro problema la cumple, siempre encontraremos una solución óptima. El problema radica en hacer esa demostración, pues cada problema es diferente. Por ello, no podemos dar un método general para demostrar la admisibilidad de una función heurística.

Lo que sí podemos establecer, es que para demostrar que una función heurística **no** es admisible basta con encontrar un nodo que incumpla la propiedad. Esto se puede observar simplemente comprobando si alguno de los nodos que están en el camino solución tiene un coste mayor que el real, pues solo disponemos del coste real para los nodos de ese camino.

Hay que remarcar también que la propiedad de admisibilidad es una propiedad de la función heurística, y es independiente del algoritmo que la utiliza, por lo que si la función es admisible cualquier algoritmo de los que veamos que la utilice nos hallará la solución óptima.

Para una discusión sobre técnicas para construir heurísticos admisibles se puede consultar el capítulo 4, sección 4.2, del libro “*Inteligencia Artificial: Un enfoque moderno*” de S. Russell y P. Norvig.

3.5.2 Consistencia

Podemos definir la propiedad de consistencia como una extensión de la propiedad de admisibilidad. Si tenemos el coste $h^*(n_i)$ y el coste $h^*(n_j)$ y el coste óptimo para ir de n_i a n_j , o sea $K(n_i, n_j)$, se ha de cumplir la desigualdad triangular:

$$h^*(n_i) \leq h^*(n_j) + K(n_i, n_j)$$

La condición de consistencia exige pedir lo mismo al estimador h :

$$h(n_i) - h(n_j) \leq K(n_i, n_j)$$

Es decir, que la diferencia de las estimaciones sea menor o igual que la distancia óptima entre nodos. Que esta propiedad se cumpla quiere decir que h es un estimador uniforme de h^* . Es decir, la estimación de la distancia que calcula h disminuye de manera uniforme.

Si h es consistente además se cumple que el valor de $g(n)$ para cualquier nodo es $g^*(n)$, por lo tanto, una vez hemos llegado a un nodo sabemos que hemos llegado a él por el camino óptimo desde el nodo inicial. Si esto es así, no es posible que nos encontremos el mismo nodo por un camino alternativo con un coste menor y esto hace que el tratamiento de nodos cerrados duplicados sea innecesario, lo que nos ahorra tener que guardarlos.

Habitualmente las funciones heurísticas admisibles suelen ser consistentes y, de hecho, hay que esforzarse bastante para conseguir que no sea así.

3.5.3 Heurístico más informado

Otra propiedad interesante es la que nos permite comparar heurísticos entre si y nos permite saber cuál de ellos hará que A^* necesite explorar más nodos para encontrar una solución óptima. Diremos que el heurístico h_1 es más informado que el heurístico h_2 si se cumple la propiedad:

$$\forall n \quad 0 \leq h_2(n) < h_1(n) \leq h^*(n)$$

Se ha de observar que esta propiedad incluye que los dos heurísticos sean admisibles.

Si un heurístico es mas informado que otro seguro que A^* expandirá menos nodos durante la búsqueda, ya que su comportamiento será más en profundidad y habrá ciertos nodos que no se explorarán.

Esto nos podría hacer pensar que siempre tenemos que escoger el heurístico más informado. Hemos de pensar también que la función heurística tiene un tiempo de cálculo que afecta al tiempo de cada iteración. Evidentemente, cuanto más informado sea el heurístico, mayor será ese coste. Nos podemos imaginar por ejemplo, que si tenemos un heurístico que necesita por ejemplo 10 iteraciones para llegar a la solución, pero tiene un coste de 100 unidades de tiempo por iteración, tardará más en llegar a la solución que otro heurístico que necesite 100 iteraciones pero que solo necesite una unidad de tiempo por iteración.

Esto nos hace pensar en que tenemos que encontrar un equilibrio entre el coste del cálculo de la función heurística y el número de expansiones que nos ahorramos al utilizarla. Es posible que una función heurística peor nos acabe dando mejor resultado. Todo esto es evidentemente dependiente del problema, incluso nos podemos encontrar con que una función no admisible nos permita hallar mas rápidamente la solución, a pesar de que no nos garantice la optimalidad.

Esto ha llevado a proponer variantes de A^* donde se utilizan heurísticos cerca de la admisibilidad (ϵ -admisibilidad) que garantizan una solución dentro de un rango del valor del coste óptimo.

3.6 Mi memoria se acaba

El algoritmo A^* tiene sus limitaciones de espacio impuestas, principalmente por poder acabar degenerando en una búsqueda en anchura si la función heurística no es demasiado buena. Además, si la solución del problema está a mucha profundidad o el tamaño del espacio de búsqueda es muy

Algoritmo 3.3 Algoritmo IDA*

Algoritmo: IDA* (limite entero)

```

prof ←  $f(\text{Estado inicial})$ 
Actual ← Estado inicial
mientras no es_final?(Actual) y  $\text{prof} < \text{limite}$  hacer
    Est_abiertos.inicializa()
    Est_abiertos.insertar(Estado inicial)
    Actual ← Est_abiertos.primer()
    mientras no es_final?(Actual) y no Est_abiertos.vacia?() hacer
        Est_abiertos.borrar_primer()
        Est_cerrados.insertar(Actual)
        Hijos ← generar_sucesores (Actual, prof)
        Hijos ← tratar_repetidos (Hijos, Est_cerrados, Est_abiertos)
        Est_abiertos.insertar(Hijos)
        Actual ← Est_abiertos.primer()
    fin
    prof ← prof+1
fin

```

grande, no hace falta mucho para llegar a necesidades de espacio prohibitivas. Esto nos lleva a plantearnos algoritmos alternativos con menores necesidades de espacio.

3.6.1 El algoritmo IDA*

La primera solución viene de la mano del algoritmo en profundidad iterativa que hemos visto en el capítulo anterior. En este caso lo replantearemos para utilizar la función f como el valor que controla la profundidad a la que llegamos en cada iteración.

Esto quiere decir que vamos a hacer la búsqueda imponiendo un límite al coste del camino que queremos hallar (en la primera iteración, la f del nodo raíz), explorando en profundidad todos los nodos con f igual o inferior a ese límite y reiniciando la búsqueda con un coste mayor si no encontramos la solución en la iteración actual. Es el algoritmo 3.3.

La función `generar_sucesores` solo retorna los nodos con un coste inferior o igual al de la iteración actual.



Este algoritmo admite varias optimizaciones, como no aumentar el coste de uno en uno, sino averiguar cual es el coste más pequeño de los nodos no expandidos en la iteración actual y usarlo en la siguiente iteración. El bucle interior es el que realiza la búsqueda en profundidad limitada y también se podría optimizar utilizando una implementación recursiva.

Este algoritmo, al necesitar solo una cantidad de espacio lineal, permite hallar soluciones a más profundidad. El precio que hemos de pagar es el de las reexpansiones de nodos ya visitados.

Este precio extra dependerá de la conectividad del grafo del espacio de estados, si existen muchos ciclos este puede ser relativamente alto ya que a cada iteración la efectividad de la exploración real se reduce con el número de nodos repetidos que aparecen.

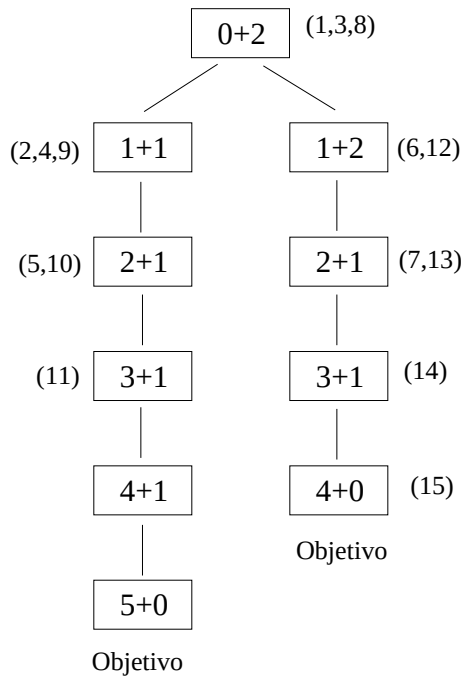


Figura 3.3: Ejemplo de ejecución del algoritmo IDA*



Si consideramos el cociente entre los nodos nuevos que se expanden en un nivel y los nodos que se han reexpandido, podemos ver que este tiende a 0 al aumentar el factor de ramificación, por lo que el trabajo de reexpandir los nodos es despreciable cuando el problema es complejo. Siendo r el factor de ramificación del problema, si consideramos que r^{n+1} será el número de nodos que visitaremos en el nivel $n + 1$ y $\sum_{i=1}^n r^i$ es la suma de nodos que revisitamos en ese nivel:

$$\frac{\sum_{i=1}^n r^i}{r^{n+1}} = \frac{r + r^2 + r^3 + \dots + r^n}{r^{n+1}} = \frac{1 + r + r^2 + r^3 + \dots + r^{n-1}}{r^n} = \frac{1}{r^n} + \frac{1}{r^{n-1}} + \dots + \frac{1}{r} = \frac{1}{r-1}$$

Ejemplo 3.3 Si consideramos el árbol de búsqueda de la figura 3.3, donde en cada nodo tenemos descompuesto el coste en g y h y marcado el orden en el que el algoritmo ha visitado cada nodo.

La numeración en los nodos indica el orden en que el algoritmo IDA* los ha expandido. Podemos observar que los nodos se reexpanden varias veces siguiendo la longitud del camino estimado impuesta en cada iteración del algoritmo.

3.6.2 Otras alternativas

Las reexpansiones del IDA* pueden llegar a ser un problema, e incrementar bastante el tiempo de búsqueda. Estas reexpansiones se deben fundamentalmente a que estamos imponiendo unas restricciones de espacio bastantes severas. Si relajáramos esta restricción manteniéndola en límites razonables podríamos guardar información suficiente para no tener que realizar tantas reexpansiones.

Una primera alternativa es el algoritmo **primero el mejor recursivo** (*recursive best first*). El elemento clave es su implementación recursiva, que permite que el coste espacial se mantenga lineal ($O(rp)$) al no tener que guardar más que los nodos que pertenecen al camino actual y sus hermanos en cada nivel.

Este algoritmo intenta avanzar siempre por la rama más prometedora (según la función heurística f), hasta que alguno de los nodos alternativos del camino tiene un coste mejor. En este momento el

Algoritmo 3.4 Algoritmo Best First Recursivo

```

Procedimiento: BFS-recursivo (nodo,c_alternativo,ref nuevo_coste,ref solucion)
si es_solucion?(nodo) entonces
    | solucion.añadir(nodo)
sino
    sucesores  $\leftarrow$  generar_sucesores (nodo)
    si sucesores.vacio?() entonces
        | nuevo_coste  $\leftarrow +\infty$ ; solucion.vacio()
    sino
        fin  $\leftarrow$  falso
        mientras no fin hacer
            | mejor  $\leftarrow$  sucesores.mejor_nodo()
            | si mejor.coste() > c_alternativo entonces
            | | fin  $\leftarrow$  cierto; solucion.vacio(); nuevo_coste  $\leftarrow$  mejor.coste()
            | sino
            | | segundo  $\leftarrow$  sucesores.segundo_mejor_nodo()
            | | BFS-recursivo(mejor,min(c_alternativo,segundo.coste()),nuevo_coste, solucion)
            | | si solucion.vacio?() entonces
            | | | mejor.coste(nuevo_coste)
            | | sino
            | | | solucion.añadir(mejor); fin  $\leftarrow$  cierto
            | fin
        fin
    fin
fin

```

camino del nodo actual es olvidado, pero modificando la estimación de los ascendientes con la del nodo mas profundo al que se ha llegado. Se podría decir que con cada reexpansión estamos haciendo el heurístico más informado. Manteniendo esta información sabremos si hemos de volver a regenerar esa rama olvidada, si pasa a ser la de mejor coste.

El efecto que tiene esta exploración es ir refinando la estimación que se tiene de la función heurística en los nodos que se mantienen abiertos, de manera que el número de reexpansiones se irá reduciendo paulatinamente ya que el coste guardado será cada vez más cercano al coste real. Se puede ver su implementación en el algoritmo 3.4.

Ejemplo 3.4 En la figura 3.4 se puede ver como evoluciona la búsqueda en un ejemplo sencillo, en cada nodo aparece el valor de la función f :

Podemos ver que cuando llega a la tercera iteración el nodo con mejor coste es el de 12 y por lo tanto la recursividad vuelve hasta ese nodo borrando la rama explorada, pero actualizando cada padre con el mejor coste encontrado, en este caso 17. En la cuarta iteración se encuentra que ahora es el nodo con coste 17 el de mejor coste, por lo que el camino vuelve a regenerarse.

Por lo general, el número de reexpansiones de este algoritmo es menor que el de IDA*, pero puede depender de las características del espacio de búsqueda como por ejemplo la longitud de los ciclos que pueda haber. Este algoritmo también impone un coste lineal al espacio, por lo que también se pueden generar bastantes reexpansiones.

Otra alternativa diferente viene de una modificación del A*, es el algoritmo llamado **A* con memoria limitada** (memory bound A*). Como su nombre indica, tenemos que establecer a

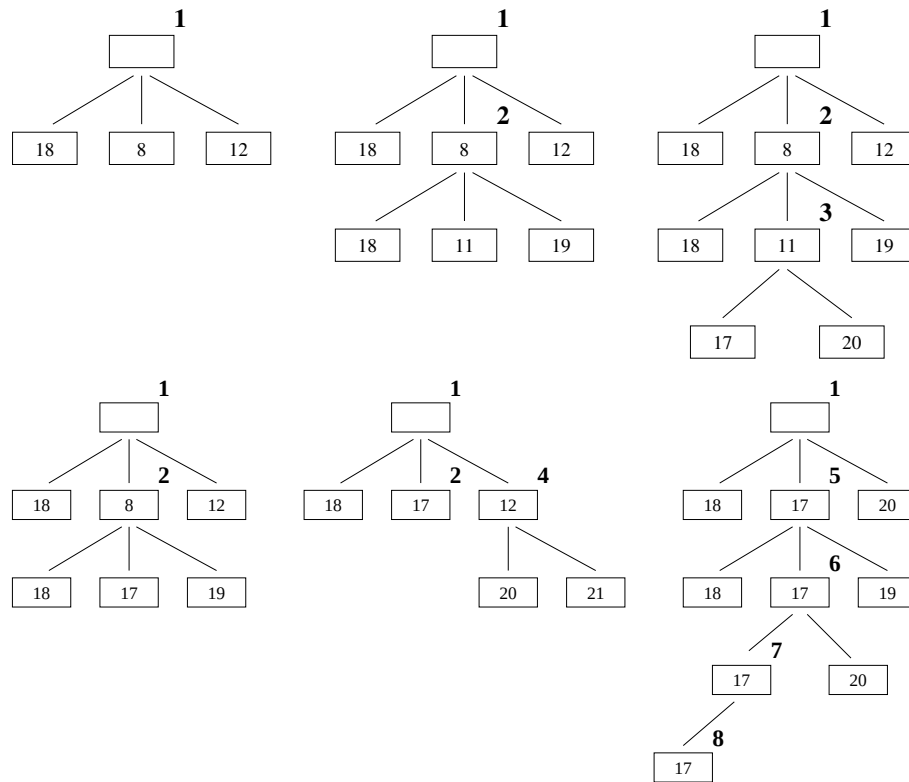


Figura 3.4: Ejemplo de ejecución del algoritmo BF-recursivo

priori la cantidad de memoria que utilizaremos. Este algoritmo utiliza la estrategia de exploración de A^* pero siguiendo una estrategia de memorización de caminos parecida al **best first recursivo**. Éste almacena todos los caminos que le caben en el tamaño de memoria que se impone y no solo el actual, de manera que se ahorra regenerar ciertos caminos. Cuanta más memoria permitamos, menos regeneraciones de caminos haremos ya que nos los encontraremos almacenados.

El algoritmo elimina los caminos peores en el momento en el que ya no caben más, guardando en los nodos la información suficiente que le permita saber cuando se han de regenerar. El aporte extra de memoria permite reducir bastante las reexpansiones, con el consiguiente ahorro de tiempo.

El algoritmo es capaz de hallar una solución, si el camino completo cabe en la cantidad de memoria que hemos impuesto. Esto quiere decir que, si tenemos una idea aproximada de la longitud de la solución podemos ajustar a priori la cantidad de memoria que permitiremos usar al algoritmo.

Este algoritmo es mucho mas complejo que los anteriores y existen diferentes versiones. Principalmente la complejidad viene de como se han de reorganizar los caminos memorizados cuando se han de borrar de la memoria y como se hace la propagación de los costes a los nodos ascendientes que se mantienen.

4.1 El tamaño importa, a veces

Los algoritmos que hemos visto en el capítulo anterior nos sirven siempre que podamos plantear el problema como la búsqueda de un camino en un espacio de estados. Pero nos podemos encontrar con problemas en los que:

- Este planteamiento es demasiado artificial, ya que no tenemos realmente operadores de cambio de estado, o no hay realmente una noción de coste asociada a los operadores del problema
- La posibilidad de hallar la solución óptima está fuera de toda posibilidad, dado que el tamaño del espacio de búsqueda es demasiado grande y nos conformamos con una solución que podamos considerar buena.

En este tipo de problemas puede ser relativamente fácil hallar una solución inicial, aunque no sea demasiado buena. Esto hace que nos podamos plantear el problema como una búsqueda dentro del espacio de soluciones, en lugar de en el de caminos. También es posible que sea muy difícil crear una solución inicial (aunque sea mala) y podamos iniciar la búsqueda desde el espacio de no soluciones (soluciones no válidas) suponiendo que la búsqueda nos llevará al final al espacio de soluciones.



El comenzar desde el espacio de soluciones o desde el de no soluciones dependerá de las características del problema. A veces el número de soluciones válidas es muy reducido debido a que lo que exigimos para ser una solución es muy estricto y es imposible generar una solución inicial sin realizar una búsqueda. Si sabemos que partiendo desde una solución inválida podremos alcanzar el espacio de soluciones, no habrá problema, el algoritmo hará esa búsqueda inicial guiado por la función heurística. Si no es así, deberemos usar otros algoritmos más exhaustivos que nos aseguren hallar una solución.

En este planteamiento lo que suponemos es que podemos navegar dentro del espacio de posibles soluciones, realizando operaciones sobre ellas que pueden ayudar a mejorarlas. El coste de estas operaciones no será relevante ya que lo que nos importará será la calidad de las soluciones¹. Deberemos disponer de una función que según algún criterio (dependiente del dominio) medirá su calidad y permitirá ordenarlas.

¹De hecho muchas veces estas operaciones no se refieren a operaciones reales en el problema, sino a formas de manipular la solución.

El planteamiento de estos problemas es parecido al que consideramos al hablar del espacio de estados, tenemos los siguientes elementos:

- Un estado inicial, que en este caso será una solución **completa**. Esto nos planteará el problema adicional de como hallarla con un coste temporal relativamente bajo y el determinar desde donde empezaremos a buscar, si desde una solución relativamente buena, desde la peor, desde una al azar, ...
- Unos operadores de cambio de la solución, que en este caso manipularán soluciones completas y que no tendrán un coste asociado.
- Una función de calidad, que nos medirá lo buena que es una solución y nos permitirá guiar la búsqueda. Hemos de tener en cuenta que este valor no nos indica cuanto nos falta para encontrar la solución que buscamos, ya que no representa un coste, es una manera de comparar la calidad entre las soluciones vecinas e indicarnos cual es la dirección que lleva a soluciones mejores.

El saber cuando hemos acabado la búsqueda dependerá totalmente del algoritmo, que tendrá que decidir cuando ya no es posible encontrar una solución mejor. Por lo tanto no tendremos un estado final definido.

Para buscar una analogía con un tema conocido, la búsqueda local se puede asimilar con la búsqueda de óptimos en funciones. En este caso la función a optimizar² será la función de calidad de la solución, la función se definirá en el espacio de soluciones generado por la conectividad que inducen los operadores entre ellas.

Desafortunadamente, las funciones con las que se trabajará no tendrán una expresión analítica global, ni tendrán las propiedades de diferenciabilidad o continuidad que nos permitirían aplicar estos algoritmos de búsqueda de óptimos en funciones. Estas funciones pueden ser de cualquier tipo, y no están limitadas de la manera que lo están las funciones usadas por los algoritmos de búsqueda heurística. Sus valores pueden ser tanto positivos, como negativos y, obviamente, su valor en la solución no tiene por que ser cero.

El planteamiento será pues bastante diferente. Ya que no tenemos una expresión analítica global, deberemos explorar la función de calidad utilizando solamente lo que podamos obtener de los vecinos de una solución. Ésta deberá permitir decidir por donde seguir buscando el óptimo. El nombre de búsqueda local viene precisamente de que solo utilizamos información local durante el proceso de hallar la mejor solución.

4.2 Tu sí, vosotros no

Enfrentados a problemas con tamaños de espacio de búsqueda inabordables de manera exhaustiva, nos hemos de plantear estrategias diferentes a las utilizadas hasta ahora.

En primer lugar, tendremos pocas posibilidades de guardar información para recuperar caminos alternativos dado el gran número de alternativas que habrá. Esto nos obligará a imponer unas restricciones de espacio limitadas, lo que nos llevará a tener que decidir que nodos debemos explorar y cuales descartar sin volver a considerarlos.

Esto nos lleva a la familia de algoritmos conocida como de **ramificación y poda (branch and bound)**. Esta familia de algoritmos limita el número de elementos que guardamos como pendientes de explotar olvidando los que no parecen prometedores. Estos algoritmos permiten mantener una

²Hablamos de optimizar, y no de minimizar o maximizar, ya que la diferencia entre ellas es simplemente un cambio de signo en la función a optimizar.

Algoritmo 4.1 Algoritmo Steepest Ascent Hill Climbing

Algoritmo: Hill ClimbingActual $\leftarrow$ Estado_inicialfin $\leftarrow$ falso**mientras** no fin **hacer** Hijos $\leftarrow$ generar_sucesores(Actual) Hijos $\leftarrow$ ordenar_y_eliminar_peores(Hijos, Actual) **si** no vacío?(Hijos) **entonces** Actual $\leftarrow$ Escoger_mejor(Hijos) **sino** fin $\leftarrow$ cierto **fin****fin**

memoria limitada, ya que desprecian parte del espacio de búsqueda, pero se arriesgan a no hallar la mejor solución, ya que esta puede estar en el espacio de búsqueda que no se explora.

De entre los algoritmos de *ramificación y poda*, los más utilizados son los denominados de **ascenso de colinas (Hill-climbing)**. Existen diferentes variantes que tienen sus ventajas e inconvenientes.

Por ejemplo, el denominado **ascenso de colinas simple**, que consiste en elegir siempre el primer operador que suponga una mejora respecto al nodo actual, de manera que no exploramos todas las posibilidades accesibles, ahorrándonos el explorar cierto número de descendientes. La ventaja es que es más rápido que explorar todas las posibilidades, la desventaja es que hay más probabilidad de no alcanzar las soluciones mejores.

El más utilizado es el **ascenso de colinas por máxima pendiente (steepest ascent hill climbing)**. Esta variante expande todos los posibles descendientes de un nodo y elige el que suponga la máxima mejora respecto al nodo actual. Con esta estrategia, suponemos que la mejor solución la encontraremos a través del sucesor que mayor diferencia tenga respecto a la solución actual, siguiendo una política avariciosa. Como veremos más adelante esta estrategia puede ser arriesgada.

El algoritmo que implementa este último es el algoritmo 4.1.

En este algoritmo, la utilización de memoria para mantener la búsqueda es nula, ya que solo tenemos en cuenta el nodo mejor³. Se pueden hacer versiones que guarden caminos alternativos que permitan una vuelta atrás en el caso de que consideremos que la solución a la que hemos llegado no es suficientemente buena, pero hemos de imponer ciertas restricciones de memoria, si no queremos tener un coste espacial demasiado grande. Comentaremos estos algoritmos más adelante.

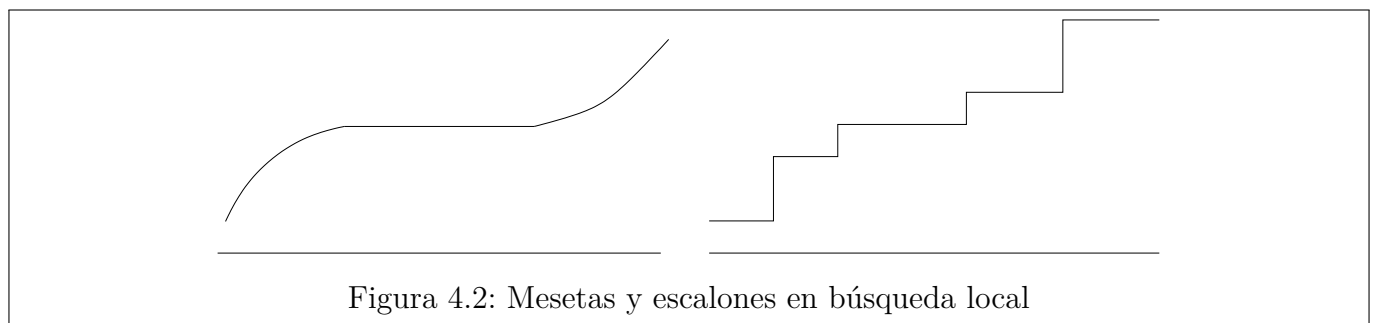
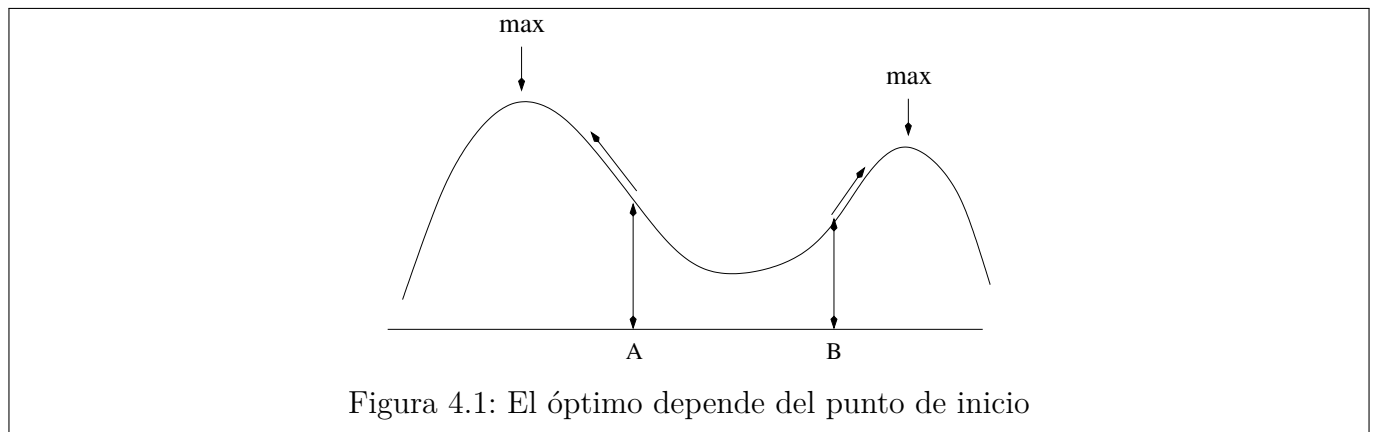
La estrategia de este algoritmo hace que sus problemas vengan principalmente derivados por las características de las funciones heurísticas que se utilizan.

Por un lado, hemos de considerar que el algoritmo parará la exploración en el momento en el que no se encuentra ningún nodo accesible mejor que el actual. Dado que no mantenemos memoria del camino recorrido nos será imposible reconsiderar nuestras decisiones, de modo que si nos hemos equivocado, la solución a la que llegaremos no será la óptima global.

Esta circunstancia es probable, ya que seguramente la función heurística que utilicemos tendrá óptimos locales y, dependiendo de por donde empecemos a buscar, acabaremos encontrando uno u otro, como se puede ver en la figura 4.1. Si iniciamos la búsqueda desde el punto A o el B, el máximo que alcanzaremos será diferente.

Esta circunstancia se puede mejorar mediante la ejecución del algoritmo cierto número de veces desde distintos puntos escogidos aleatoriamente y quedándonos con la mejor exploración. A esta

³Hay que recordar que en este tipo de búsqueda el camino no nos importa, por lo que el gasto en espacio es constante.



estrategia se la denomina búsqueda por ascenso con **reinicio aleatorio** (**random restarting hill climbing**). Muchas veces esta estrategia es más efectiva que métodos más complejos que veremos a continuación y resulta bastante barata. La única complicación está en poder generar los distintos puntos iniciales de búsqueda, ya que en ocasiones el problema no permite hallar soluciones iniciales con facilidad.

Otro problema con el que se encuentran estos algoritmos son zonas del espacio de búsqueda en las que la función heurística no es informativa. Un ejemplo son las denominadas mesetas y las funciones en escalón (figura 4.2), en las que los valores de los nodos vecinos al actual tienen valores iguales, y por lo tanto una elección local no nos permite decidir el camino a seguir.

Evitar estos problemas requiere extender la búsqueda más allá de los vecinos inmediatos para obtener información suficiente para encaminar la búsqueda. Desgraciadamente esto supone un coste adicional en cada iteración, prohibitivo cuando el factor de ramificación es grande.

Una alternativa es permitir que el algoritmo guarde parte de los nodos visitados para proseguir la búsqueda por ellos en el caso de que el algoritmo se quede atascado en un óptimo local. El algoritmo más utilizado de este tipo es el denominado **búsqueda en haces** (**beam search**). En este algoritmo se van guardando un número N de las mejores soluciones, expandiendo siempre la mejor de ellas. De esta manera no se sigue un solo camino sino N .

Las variantes del algoritmo están en cómo se escogen esas N soluciones que se mantienen. Si siempre se substituyen las peores soluciones de las N por los mejores sucesores del nodo actual, caemos en el peligro de que todas acaben estando en el mismo camino, reduciendo la capacidad de volver hacia atrás, así que el poder mantener la variedad de las soluciones guardadas es importante.

Está claro que cuantas más soluciones guardemos más posibilidad tendremos de encontrar una buena solución, pero tendremos un coste adicional tanto en memoria como en tiempo, ya que tendremos que calcular con qué sucesores nos quedamos y qué soluciones guardadas substituímos. Su implementación se puede ver en el algoritmo 4.2.

El algoritmo acaba cuando ninguno de los sucesores mejora a las soluciones guardadas, esto quiere decir que todas las soluciones son un óptimo local. Como veremos más adelante esta técnica de búsqueda tiene puntos en común con los algoritmos genéticos.

Algoritmo 4.2 Algoritmo Beam Search**Algoritmo:** Beam SearchActual $\leftarrow$ Estado_inicial

Soluciones_actuales.añadir(Estado_inicial)

fin $\leftarrow$ falso**mientras** no fin **hacer** Hijos $\leftarrow$ generar_sucesores(Actual)

soluciones_actuales.actualizar_mejores(Hijos)

si soluciones_actuales.algun_cambio?() **entonces** Actual $\leftarrow$ soluciones_actuales.escoger_mejor() **sino** fin $\leftarrow$ cierto **fin****fin****4.2.1 Ejemplo: El problema de la mochila**

Hay una extensa variedad de problemas que se puede solucionar mediante estos algoritmos, vamos a plantear como ejemplo una aplicación a un problema de sobra conocido, como es el problema de la mochila (*knapsack problem*).

El planteamiento de este problema es simple, disponemos de un contenedor de una capacidad limitada (habitualmente el peso, pero puede ser cualquier medida de capacidad) y debemos llenarlo eligiendo de entre conjunto limitado de objetos que ocupan un espacio específico en el contenedor y llevan un valor asociado. El objetivo es llenar el contenedor con un subconjunto de los objetos de manera que no se sobrepase su capacidad, y que estos objetos sumen el mayor valor posible.

Solucionar el problema de manera exacta requeriría explorar todos los subconjuntos de objetos y quedarnos con el que cumpla las restricciones. Esto requeriría explorar un numero exponencial de subconjuntos (el conjunto de partes del conjunto de objetos disponibles). Obviamente no es práctico resolver este problema de manera exacta.

El interés de este problema es que aparece en muchos problemas prácticos en múltiples variante, como por ejemplo minimizando el valor del resultado y manteniendo un peso mínimo, o usando múltiples contenedores, por ejemplo.

El planeamiento de este problema para solucionarlo mediante búsqueda local requiere primero plantear cual será nuestro espacio de búsqueda. Podemos plantear buscar solamente en el espacio de soluciones, de manera que buscaríamos dentro de los subconjuntos que cumplen las restricciones de peso, o podríamos también incluir el espacio de soluciones, buscando en dentro de todos los subconjuntos posibles. Parece más sencillo mantenernos siempre dentro del espacio de soluciones, sobre todo, porque puede ser difícil asegurar que si partimos desde una no solución el algoritmo acabe dentro del espacio de soluciones.

Una vez decidido esto podemos plantearnos cual será la solución inicial de partida. Sabemos que la solución final dependerá de esta elección, así que puede merecer la pena plantearse con calma. Existe una solución obvia, que es partir de una mochila vacía, es una solución, aunque sabemos que es la peor posible. Podemos plantearnos también partir de cualquier otra solución, solo es necesario encontrar un algoritmo que nos asegure que el peso de los objetos se encuentre dentro del límite. Hemos de pensar también que partir de una solución inicial demasiado buena, puede dejarnos cerca de un óptimo local en el que nos quedemos atascados enseguida. Seremos conservadores e iniciaremos la búsqueda desde la solución vacía.

La función heurística en ocasiones es bastante obvia, en otras se puede comprobar que la elección

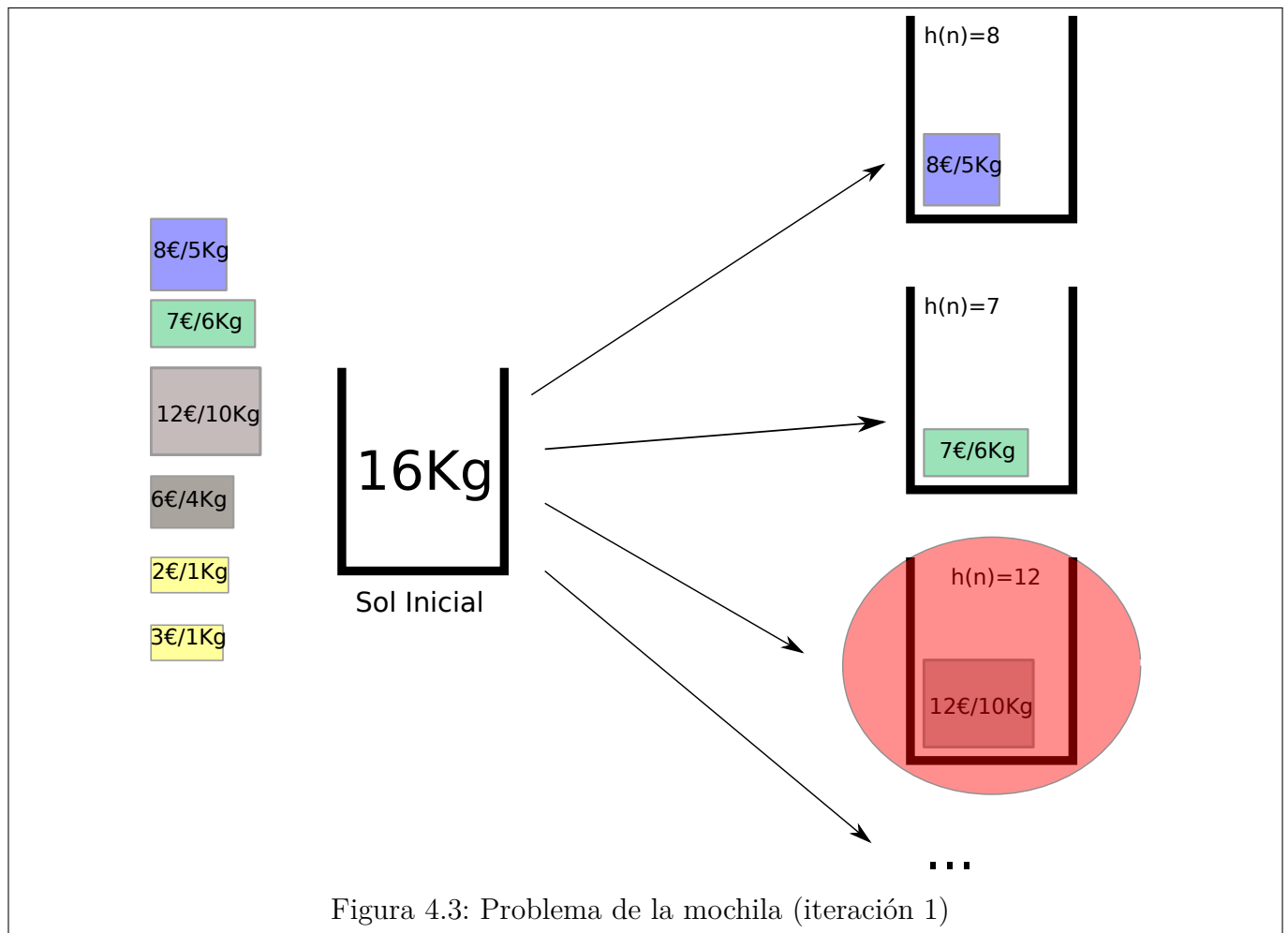


Figura 4.3: Problema de la mochila (iteración 1)

obvia quizás no es la más adecuada como comentaremos más adelante. En este caso, parece evidente que nuestro objetivo es obtener la solución con mayor valor, por lo que podríamos adoptarla como función a optimizar.

El siguiente problema a plantearnos es el conjunto de operadores. Este suele depender del espacio de búsqueda que hayamos elegido, de la solución inicial y de la función heurística. Los operadores posibles son todos aquellos que nos permitan navegar en el espacio de soluciones definido. En este caso podemos ver que nuestras posibilidades serían el añadir un objeto nuevo en la mochila, quitar un objeto de la mochila y el intercambiar un objeto de la mochila por otro que no este en la mochila.

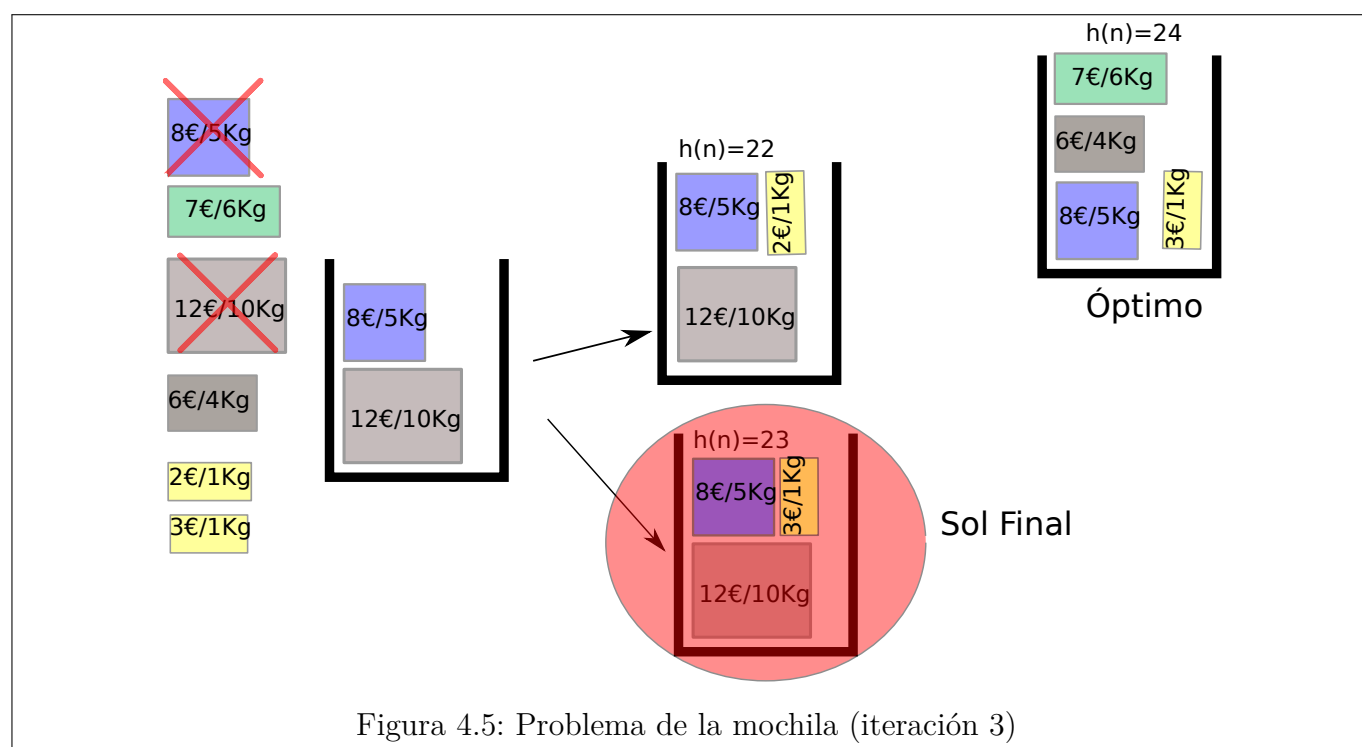
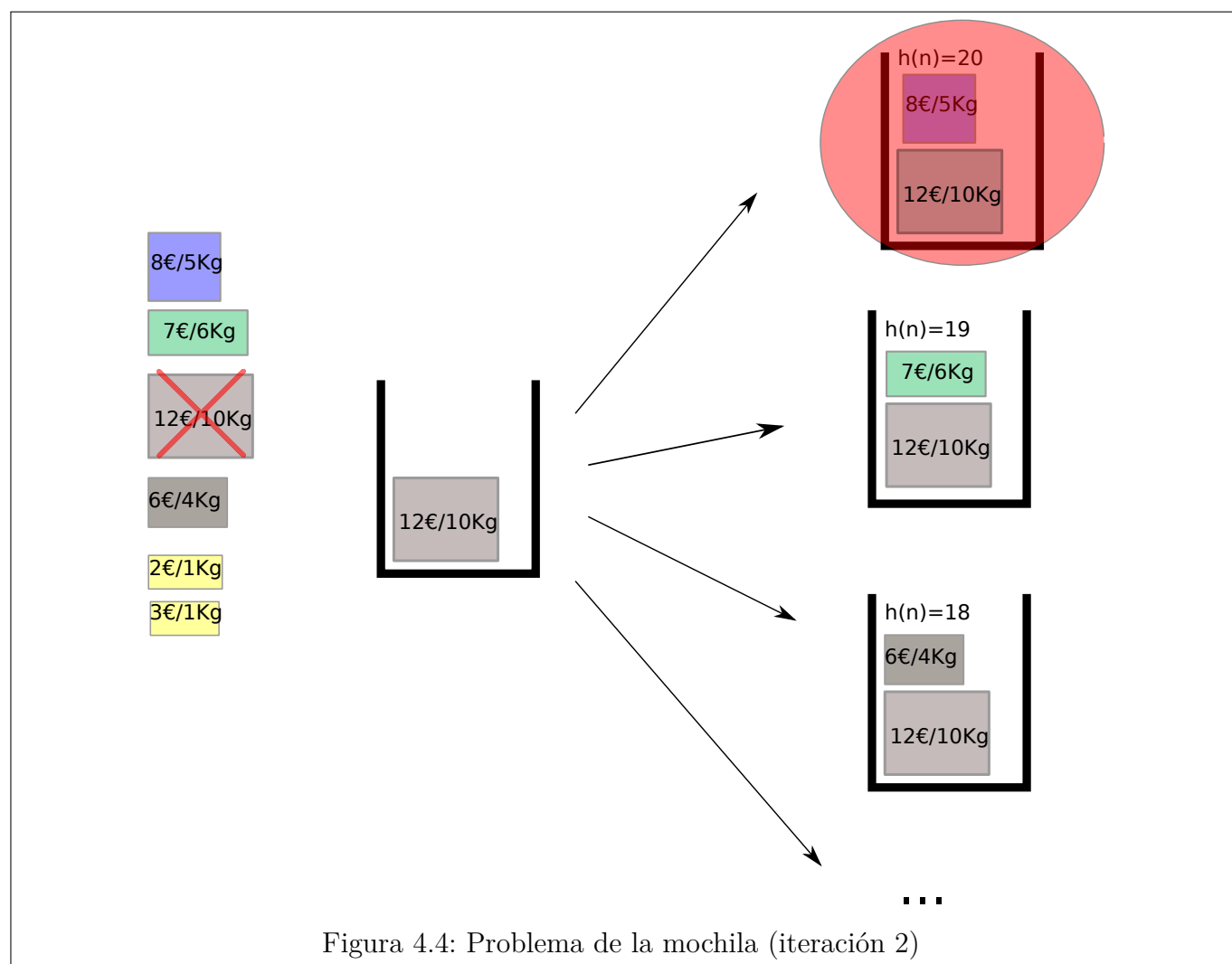
Por el comportamiento de la función heurística podemos ver que el operador de quitar objetos de la mochila siempre nos dará valores para la función heurística que serán peores que la solución actual, por lo que no tiene sentido utilizarlo. Para simplificar el ejemplo vamos a reducir nuestro conjunto de operadores a solamente añadir un objeto nuevo a la mochila.

En la figura 4.3 podemos ver para una instancia de este problema como se comporta el algoritmo de Hill Climbing en su primera iteración. El algoritmo generaría todos los estados sucesores correspondientes a añadir un objeto a la solución. El algoritmo escogería como siguiente estado el que maximiza la función heurística, en este caso añadir el objeto que tiene un valor de 12 euros.

La segunda iteración (figura 4.4) genera todas las soluciones que corresponden a añadir un objeto nuevo a la solución que contiene el objeto con 12 euros de valor. En este caso el estado que maximiza la función heurística es que añade el objeto con 8 euros de valor.

La tercera iteración (figura 4.5) ya solo deja dos posibilidades, de ellas la que obtiene un mejor valor es la añade el objeto con 3 euros de valor.

Como se ha comentado estos algoritmos no garantizan la solución óptima, como se puede ver en la figura 4.5 la solución óptima global no se corresponde con la encontrada. El óptimo alcanzado



por el algoritmo de Hill-climbing depende de múltiples factores, como la solución de partida, la información de la función heurística y su comportamiento y la conectividad entre estados que genera el conjunto de operadores elegidos.

Para este problema concreto podemos fijarnos que en la exploración del espacio de soluciones intervienen factores que no hemos incluido en la función heurística. En este caso la relación entre el valor y el peso del objeto puede ser más informativo que solamente su valor. Una función heurística que tenga en cuenta esta relación probablemente sea capaz de encontrar en ocasiones soluciones mejores para este problema.

4.3 Un mundo de posibilidades

La estrategia de búsqueda del algoritmo hill climbing y sus variantes es la más simple que se puede utilizar. El mayor problema es que es demasiado optimista y asume que hay un camino más o menos directo (siempre ascendente) a los óptimos locales. En problemas difíciles esto no tiene por que ser así, y hacen falta estrategias un poco más exhaustivas, explorando más nodos que solo el camino directo y que sepan aprovechar la información que vamos encontrando por el camino.

Varias son las estrategias que podemos añadir sobre la estrategia básica del ascenso de colinas que pueden mejorar sustancialmente sus resultados. La primera es explorar no solo los vecinos mejores, sino también otros que pueden ser peores, pero que pueden ocultar un camino que es mejor que el que podemos encontrar si seguimos directamente la pendiente ascendente de la función heurística. Para decidir qué nodos peores visitar necesitamos una estrategia de decisión, una posibilidad es la que plantea el algoritmo *simulated annealing* que veremos en la siguiente sección.

Otra posibilidad es usar información de más alto nivel que podamos ir encontrando durante la búsqueda, como por ejemplo, qué operaciones son las que consiguen mejorar más las soluciones (o no lo consiguen) de manera que podamos concentrarnos más en algunas operaciones que en otras (reduciendo así el espacio de búsqueda, o que partes de la descripción de la solución aportan más a la calidad de la solución o qué partes nunca la mejoran, de manera que nos podamos concentrar en operaciones que se centren en esos elementos de la solución. Estas y otras estrategias parecidas son las que utiliza el algoritmo *tabú search*, este algoritmo puede prohibir o permitir operaciones sobre la solución en función de la información que se vaya obteniendo durante la búsqueda.

Una última posibilidad que ha dado lugar a muchos algoritmos es el uso de una población de soluciones en lugar de una sola solución, siguiendo la idea del algoritmo *beam search*. El tener una población de soluciones nos permite ver más parte del espacio de búsqueda, siendo más exhaustivos y, además, obtener información que nos permitirá concentrarnos más en soluciones con ciertas características o el evitar parte del espacio de búsqueda que no tenga soluciones buenas.

Muchos de los métodos que siguen esta estrategia están inspirados en analogías biológicas o físicas. Los más conocidos son los *algoritmos evolutivos*, que incluyen a los **algoritmos genéticos** que veremos en la última sección, pero que también incluyen a otros algoritmos que se basan en conceptos inspirados en la evolución de las especies y la ecología.

Otra familia de algoritmos que usan también una estrategia de población es la denominada *swarm intelligence*, incluye algoritmos que están inspirados en biología como el algoritmo *Particle Swarm Optimization*, que se inspira en el comportamiento de las bandadas de pájaros y los bancos de peces, el *Ant Colony Optimization* que se inspira en la forma que usan las hormigas de solucionar problemas (son buenos para la resolución de problemas que se pueden representar como caminos en grafos), además de algoritmos inspirados en el comportamiento de abejas, luciérnagas, cucos, monos, el sistema inmunitario... También hay algoritmos inspirados en fenómenos físicos como *Intelligent Water Drops* que simula la interacción de las gotas de agua, *River Formation Dynamics*, que usa la estrategia de formación de los ríos, *Gravitational Search Algorithm*, que usa la interacción de la

fuerza de gravedad entre partículas, ...

La aplicación de todos estos algoritmos se ha ido extendiendo más allá de los problemas de inteligencia artificial y se aplican a cualquier problema que involucre la optimización de una función y que no se pueda resolver por métodos clásicos. De hecho, todos estos algoritmos se incluyen en un área de computación conocida como *metaheurísticas*

4.4 Demasiado calor, demasiado frío

Se han planteado algoritmos alternativos de búsqueda local que se han mostrado efectivos en algunos dominios en los que los algoritmos de búsqueda por ascenso se quedan atascados enseguida en óptimos no demasiado buenos.

El primero que veremos es el denominado **templado simulado** (**simulated annealing**). Este algoritmo está inspirado en un fenómeno físico que se observa en el templado de metales y en la cristalización de disoluciones.

Todo conjunto de átomos o moléculas tiene un estado de energía que depende de cierta función de la temperatura del sistema. A medida que lo vamos enfriando, el sistema va perdiendo energía hasta que se estabiliza. El fenómeno físico que se observa es que dependiendo de como se realiza el enfriamiento, el estado de energía final es muy diferente.

Por ejemplo, si una barra de metal se enfría demasiado rápido la estructura cristalina a la que se llega al final está poco cohesionada (un número bajo de enlaces entre átomos) y ésta se rompe con facilidad. Si el enfriamiento se realiza más lentamente, el resultado final es totalmente diferente, el número de enlaces entre los átomos es mayor y se obtiene una barra mucho más difícil de romper.

Durante este enfriamiento controlado, se observa que la energía del conjunto de átomos no disminuye de manera constante, sino que a veces la energía total puede ser mayor que en un momento inmediatamente anterior. Esta circunstancia hace que el estado de energía final que se alcanza sea mejor que cuando el enfriamiento se hace rápidamente. A partir de este fenómeno físico, podemos derivar un algoritmo que permitirá en ciertos problemas hallar soluciones mejores que los algoritmos de búsqueda por ascenso.

En este algoritmo el nodo siguiente a explorar no será siempre el mejor descendiente, sino que lo elegiremos aleatoriamente, en función de los valores de unos parámetros, de entre todos los descendientes (los buenos y los malos). Una ventaja de este algoritmo es que no tenemos por qué generar todos los sucesores de un nodo, basta elegir un sucesor al azar y decidir si continuamos por él o no. El algoritmo de templado simulado se puede ver como una versión estocástica del algoritmo de búsqueda por ascenso.

El algoritmo de templado simulado intenta transportar la analogía física al problema que queremos solucionar. En primer lugar denominaremos función de **energía** a la función heurística que nos mide la calidad de una solución. Tendremos un parámetro de control que denominaremos **temperatura**, que nos permitirá controlar el funcionamiento del algoritmo.

Tendremos una función que dependerá de la temperatura y de la diferencia de calidad entre el nodo actual y un sucesor. Ésta gobernará la elección de los sucesores, su comportamiento será el siguiente:

- Cuanta mayor sea la temperatura, más probabilidad habrá de que al generar como sucesor un estado peor éste sea elegido
- La elección de un estado peor estará en función de su diferencia de calidad con el estado actual, cuanto más diferencia, menos probabilidad de elegirlo.

El último elemento es la **estrategia de enfriamiento**. El algoritmo realizará un número total de iteraciones fijo y cada cierto número de ellas el valor de la temperatura disminuirá en cierta

Algoritmo 4.3 Algoritmo Simulated Annealing**Algoritmo:** Simulated Annealing

Partimos de una temperatura inicial

mientras la temperatura no sea cero **hacer**

// Paseo aleatorio por el espacio de soluciones

para un numero prefijado de iteraciones **hacer** Enuevo $\leftarrow$ Genera_sucesor_al_azar(Eactual) $\Delta E \leftarrow f(Eactual) - f(Enuevo)$ **si** $\Delta E > 0$ **entonces** Eactual $\leftarrow$ Enuevo **sino** con probabilidad $e^{\Delta E/T}$: Eactual $\leftarrow$ Enuevo **fin** **fin**

Disminuimos la temperatura

fin

cantidad, partiendo desde una **temperatura inicial** y llegando a cero en la última fase. De manera que, la elección de estos dos parámetros (número total de iteraciones y número de iteraciones entre cada bajada de temperatura) determinarán el comportamiento del algoritmo. Habrá que decidir experimentalmente cual es la temperatura inicial más adecuada y forma más adecuada de hacer que vaya disminuyendo.

Como en la analogía física, si el número de pasos es muy pequeño, la temperatura bajará muy rápidamente, y el camino explorado será relativamente aleatorio. Si el número de pasos es más grande, la bajada de temperatura será más suave y la búsqueda será mejor.

El algoritmo que implementa esta estrategia es el algoritmo 4.3. En la figura 4.6 podemos ver el comportamiento que tendría el simulated annealing comparado con el de hill climbing.

Este algoritmo se adapta muy bien a problemas de optimización combinatoria (configuración óptima de elementos⁴) y continua (punto óptimo en un espacio N-dimensional). Está indicado para problemas con un espacio de búsqueda grande en los que el óptimo está rodeado de muchos óptimos locales, ya que a este algoritmo le será más fácil escapar de ellos.

Se puede utilizar también para problemas en los que encontrar una heurística discriminante es difícil (una elección aleatoria es tan buena como otra cualquiera), ya que el algoritmo de ascenso de colinas no tendrá mucha información con la que trabajar y se quedará atascado enseguida. En cambio el templado simulado podrá explorar más espacio de soluciones y tendrá mayor probabilidad de encontrar una solución buena.

El mayor problema de este algoritmo será determinar los valores de los parámetros, y requerirá una importante labor de experimentación que dependerá de cada problema. Estos parámetros variarán con el dominio e incluso con el tamaño de la instancia del problema.

4.4.1 Ejemplo: El viajante de comercio

El problema del viajante de comercio (*travel salesman problem*) es también un problema bastante conocido y con bastantes aplicaciones prácticas que van desde el diseño de rutas de distribución hasta la optimización del diseño de chips. El planteamiento es muy sencillo, disponemos de un conjunto de N ciudades que debemos recorrer y conocemos las distancias que conectan cada pareja de ciudades,

⁴Un área en la que estos algoritmos han sido muy utilizados es la del diseño de circuitos VLSI.

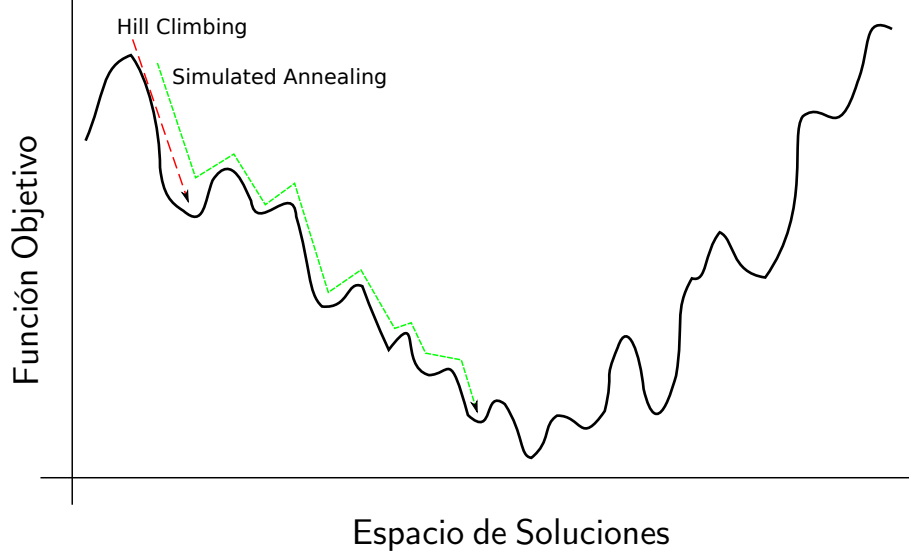


Figura 4.6: Estrategia de exploración del simulated annealing

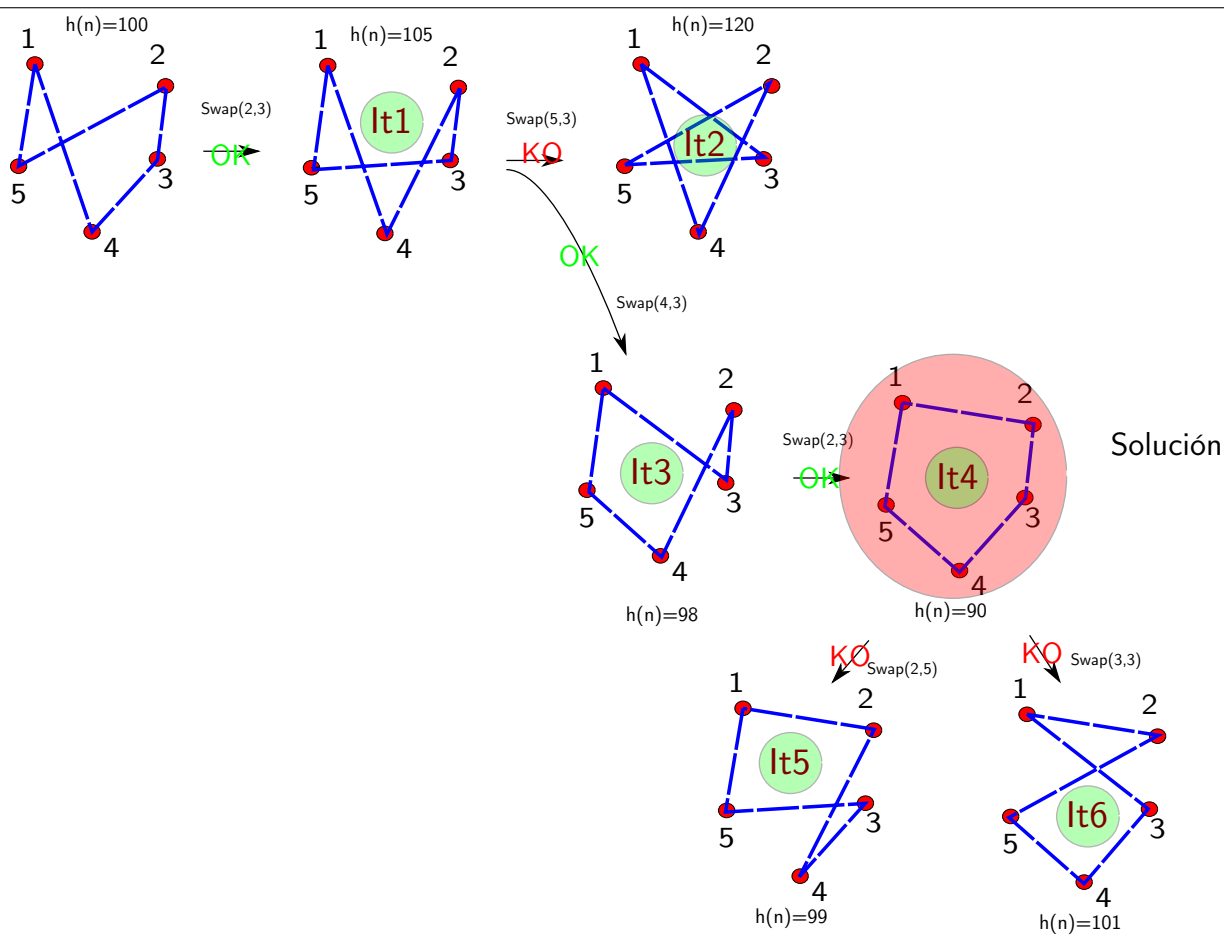


Figura 4.7: Problema del viajante de comercio

debemos encontrar un camino que recorra todas las ciudades una sola vez, comenzando y finalizando el recorrido en la misma ciudad. El objetivo es encontrar el recorrido más corto posible.

El espacio de búsqueda está formado por todos los posibles caminos que se pueden formar con las N ciudades ($N!$), por lo que es impracticable encontrar la solución óptima.

Para solucionar ese problema primero nos hemos de plantear el espacio de búsqueda. En este caso podemos considerar que está formado por todos los caminos posibles que se pueden formar con las N ciudades.

Como solución inicial podemos partir desde cualquier camino al azar o plantearnos encontrar una buena solución inicial que nos reduzca el camino a recorrer en el espacio de búsqueda antes de encontrar un óptimo local. Una heurística utilizada para encontrar una buena solución inicial en este problema consiste en usar una estrategia avariciosa, comenzando en la primera ciudad vamos eligiendo la ciudad que no hayamos visitado que se encuentre más cerca.

Como operadores de búsqueda tenemos muchas opciones, cualquier transformación de un recorrido nos permite movernos en el espacio de búsqueda. Es una elección habitual el escoger un operador que intercambia dos ciudades en el recorrido, esto supondría tener un factor de ramificación cuadrático, asegurándonos que todas las soluciones son alcanzables. También podemos elegir otros operadores como por ejemplo, mover un trozo del recorrido a una posición cualquiera o invertir el recorrido entre dos ciudades, ... El tener más operadores nos aumenta la conectividad en el espacio de soluciones, acortando los caminos hacia los óptimos, pero obviamente aumenta el factor de ramificación.

La función heurística en este problema es la obvia, hemos de minimizar la longitud del recorrido. No obstante, podemos plantear variantes del problema cambiando la función heurística, añadiendo otras restricciones que sean optimizadas por el algoritmo de búsqueda, como, por ejemplo, imponer que se recorran ciertas ciudades en un orden específico o en grupos establecidos.

En la figura 4.7 se pueden ver algunas iteraciones de este problema usando el algoritmo de simulated annealing. A cada iteración se genera un sucesor al azar y dependiendo de la función de annealing se decide moverse al sucesor o generar uno nuevo. Como se puede observar en el ejemplo, en ocasiones se avanza hacia sucesores con peor coste (iteraciones 1, 5 y 6), dependiendo de la función de annealing el algoritmo avanza a esa solución peor como en la iteración 1, o las soluciones peores son rechazadas, como en las iteraciones 5 y 6. Este comportamiento permite a este algoritmo escapar a los óptimos locales en los que queda atrapado el algoritmo de hill climbing.

4.5 Cada oveja con su pareja

Otra analogía que ha dado lugar a un conjunto de algoritmos de búsqueda local bastante efectivos es la selección natural como mecanismo de adaptación de los seres vivos. Esta analogía se fija en que los seres vivos se adaptan al entorno gracias a las características heredadas de sus progenitores, en que las posibilidades de supervivencia y reproducción son proporcionales a la bondad de esas características y en que la combinación de buenos individuos puede dar lugar a individuos mejor adaptados.

Podemos trasladar estos elementos a la búsqueda local identificando las soluciones con individuos, y donde una función de calidad indicará la bondad de la solución, es decir, su adaptación a las características del problema. Dispondremos de unos operadores de búsqueda que combinarán buenas soluciones con el objetivo de obtener soluciones mejores como resultado.

El ejemplo que veremos de algoritmos que utilizan esta analogía es el de los **algoritmos genéticos**⁵. Se trata de una familia bastante amplia de algoritmos, nosotros solo veremos el esquema básico.

⁵Existen también los denominados algoritmos evolutivos, son menos conocidos, pero suelen ser bastante efectivos en ciertas aplicaciones.

Estos algoritmos son bastante más complejos que los algoritmos que hemos visto hasta ahora y requieren más elementos y parámetros, por lo que su utilización requiere cierta experiencia y mucha experimentación. Los requisitos básicos para usarlos son:

- Dar una codificación a las características de las soluciones. Las soluciones se representan de una manera especial para que después se puedan combinar. Por lo general se utilizan cadenas binarias que codifican los elementos de la solución, por analogía a esta codificación se la denomina **gen**⁶.
- Tener una función que mida la calidad de la solución. Se tratará de la función heurística que se utiliza en todos los algoritmos que hemos visto. En el área de algoritmos genéticos a esta función se la denomina **función de adaptación (fitness)**.
- Disponer de operadores que combinen las soluciones para obtener nuevas soluciones. Los operadores que utilizan los algoritmos genéticos son bastante fijos y están *inspirados* en las formas de reproducción celular.
- Decidir el número de individuos inicial. Un algoritmo genético no trata un solo individuo, sino una población, se ha de decidir cuan numerosa ha de ser.
- Decidir una estrategia para hacer la combinación de individuos. Siguiendo la analogía de la selección natural, solo se reproducen los individuos más aptos, hemos de decidir un criterio para emparejar a los individuos de la población en función de su calidad.

Vamos a detallar un poco más cada uno de estos elementos.

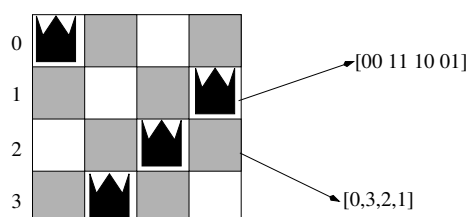


El uso de los algoritmos genéticos presenta ciertas ventajas prácticas respecto a los algoritmos anteriores. Estamos redefiniendo como se representan los problemas, todos tendrán una representación común independientemente de su naturaleza, por lo que solo nos deberemos de preocupar de como codificar el estado, su implementación viene determinada por esa codificación. Tampoco tendremos que preocuparnos de los operadores ya que esa representación común nos determina los posibles operadores que podremos usar y serán comunes también para todos los problemas.

4.5.1 Codificación

Como hemos mencionado, la forma más habitual de codificar los individuos para utilizar un algoritmo genético es transformarlos a una cadena de bits. Cada bit o grupo de bits codificará una característica de la solución. Evidentemente, no existe un criterio preestablecido sobre como realizarla.

La ventaja de esta codificación es que los operadores de modificación de la solución son muy sencillos de implementar. En la siguiente figura se puede ver un ejemplo de codificación para el problema de las N reinas siendo N igual a 4:



⁶Esta no tiene por que ser siempre la representación más adecuada y de hecho a veces una representación binaria hace que el problema no se pueda solucionar correctamente.

En la codificación binaria se ha asignado una pareja de bits a cada una de las reinas, representando la fila en la que están colocadas y se han concatenado. Alternativamente se podría utilizar una representación no binaria utilizando simplemente el número de fila y haciendo una lista con los valores.

Como se ha comentado, no siempre la codificación binaria es la más adecuada, ya que las características del problema pueden hacer que el cambio de representación haga más complicado el problema.

Una ventaja de la codificación binaria es que nos da una aproximación del tamaño del espacio de búsqueda, ya que el número de soluciones posibles corresponderá al mayor número binario que podamos representar con la codificación.



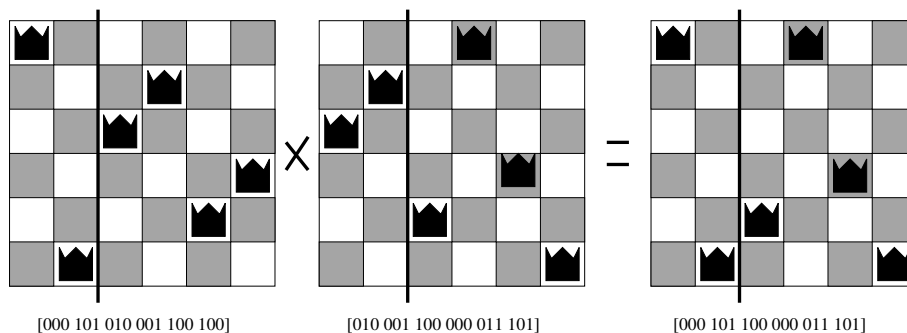
A veces es importante pensar bien como codificamos los estados, hay que tener en cuenta que la codificación binaria impone la conectividad entre los estados y que, al hacer la transformación, estados que antes eran vecinos ahora no lo serán. Puede ser interesante hacer que la codificación mantenga esa vecindad para obligar a que el algoritmo haga la exploración del espacio de búsqueda de cierta manera.

También a veces nos interesa que diferentes partes de la codificaciones correspondan a partes específicas del estado y así obtener patrones que puedan mantenerse y propagarse de manera más sencilla.

4.5.2 Operadores

Los operadores que utilizan estos algoritmos para la exploración del espacio de búsqueda se pueden agrupar en dos tipos básicos. Los operadores de **cruce** (**crossover**) y los de **mutación** (**mutation**).

Los operadores de cruce se aplican a una pareja de individuos, su efecto consiste en construir dos nuevos individuos intercambiando parte de la información de la codificación entre estos. Existe una gran variedad de formas en las que se puede realizar este intercambio. La más sencilla es la denominada **cruce por un punto** (**one point crossover**). Se aplica solo a codificaciones binarias, este operador consiste en elegir un punto al azar en la codificación e intercambiar los bits de los dos individuos a partir de ese punto. En la siguiente figura hay un ejemplo de cruce por un punto en el problema de las N reinas:



Otra variante muy utilizada es el **cruce por dos puntos** (**two points cross over**), en el que el intercambio se realiza eligiendo dos puntos en la codificación e intercambiando los bits entre estos.

Aparte de estos dos operadores existen muchos otros que tienen efectos específicos y que funcionan mejor en ciertas circunstancias (uniform crossover, random crossover, ...). Si la codificación no es binaria, los operadores se han de diseñar de manera específica para la representación escogida. También hay que pensar a veces en la eficiencia de la aplicación de los operadores, cuando tenemos que procesar muchas soluciones nos puede interesar operadores que puedan traducirse a operaciones aritméticas simples entre tiras de bits (or, and, xor, shifts, ...).

Los operadores de mutación se aplican a un solo individuo y consisten en cambiar parte de la codificación de manera aleatoria. El más sencillo es elegir un bit al azar y cambiar su signo.

Estos operadores tienen una probabilidad de aplicación asociada que será un parámetro del algoritmo. Tendremos una probabilidad de cruce que indicará cuando una pareja elegida de individuos intercambia su información y una probabilidad de mutación. Por lo general la probabilidad de mutación es mucho más pequeña que la probabilidad de cruce. La elección de estos valores es crítica para el correcto funcionamiento del algoritmo y muchas veces depende del problema.



La probabilidad de aplicación de los operadores puede ser algo muy delicado, ya que cambia la forma en la que se hace la exploración al cambiar las características de la población. Si la probabilidad de cruce es muy alta, cada generación tendrá muchos individuos nuevos, esto puede hacer que la exploración vaya más deprisa, pero también puede hacer que los patrones que llevan a soluciones mejores no puedan estabilizarse. Si la probabilidad es muy pequeña, la exploración será mas lenta y puede tardar mucho en converger.

Con la mutación pasa algo parecido. Esta es esencial para que el algoritmo pueda explorar en partes del espacio de búsqueda que no son alcanzables con la combinación de las soluciones iniciales, pero una probabilidad muy alta puede hacer que la exploración no converja al introducir demasiado ruido en la población.

4.5.3 Combinación de individuos

Estos algoritmos no tratan las soluciones de manera individual, sino en grupo. Denominaremos **población** al conjunto de soluciones que tenemos en un momento específico. El efecto de mantener un conjunto de soluciones a la vez es hacer una exploración en paralelo del espacio de búsqueda desde diferentes puntos.

Un parámetro más que deberemos decidir es el tamaño de la población de soluciones. Este es crítico respecto a la capacidad de exploración. Si el número de individuos es demasiado grande, el coste por iteración puede ser prohibitivo, pero si el tamaño es demasiado pequeño, es posible que no lleguemos a una solución demasiado buena, al no poder ver suficiente espacio de búsqueda.

Cada iteración de un algoritmo genético es denominada una **generación**, a cada generación los individuos se emparejan y dan lugar a la siguiente generación de individuos. Es clave la manera en la que decidimos como se emparejan los individuos.

La forma habitual de pasar de una generación a otra, es crear lo que se denomina una **generación intermedia** que estará compuesta por las parejas que se van a combinar. El número de individuos de esta población es igual al del tamaño original de la población. La elección de los individuos que forman parte de esta generación intermedia se puede hacer de muchas maneras, por ejemplo:

- Escoger un individuo de manera proporcional a su valor de evaluación de la función de fitness, de manera que los individuos mejores tendrán más probabilidad de ser elegidos. Esto puede tener el peligro de que unos pocos individuos pueden ser elegidos muchas veces.
- Establecer N torneos entre parejas de individuos escogidas al azar, el ganador de cada emparejamiento es el individuo con mejor valor en la función de fitness. Esta opción equilibra mejor la aparición de los mejores individuos.
- Definir un ranking lineal entre individuos según su función de calidad, estableciendo un máximo de posibles apariciones de un individuo, de manera que la probabilidad de aparición de los individuos no esté sesgada por los individuos mejores.

Con estos mecanismos de elección habrá individuos que aparezcan más de una vez e individuos que no aparezcan⁷. Cada mecanismo de selección de individuos tiene sus ventajas e inconvenientes. El primero es el más simple, pero corre el peligro de degenerar en pocas generaciones a una población homogénea, acabando en un óptimo local. Los otros dos son algo más complejos, pero aseguran cierta diversidad de individuos en la población. Se ha visto experimentalmente que el asegurar la variedad en la población permite encontrar mejores soluciones⁸.

4.5.4 El algoritmo genético canónico

Existe una gran variedad de algoritmos genéticos, pero todos parten de un funcionamiento básico que llamaremos el **algoritmo genético canónico**, los pasos que realiza este algoritmo básico son estos:

1. Se parte de una población de N individuos generada aleatoriamente
2. Se escogen N individuos de la generación actual para la generación intermedia (según el criterio escogido)
3. Se emparejan los individuos y para cada pareja
 - a) Con una probabilidad P_{cruce} se aplica el operador de cruce a cada pareja de individuos y se obtienen dos nuevos individuos que pasarán a la nueva generación. Con una probabilidad $1 - P_{cruce}$ se mantienen la pareja original en la siguiente generación
 - b) Con una probabilidad $P_{mutacion}$ se mutan los individuos cruzados
4. Se substituye la población actual con los nuevos individuos, que forman la nueva generación
5. Se itera desde el segundo paso hasta que la población converge (la función de fitness de la población no mejora) o pasa un número específico de generaciones

La probabilidad de cruce influirá en la variedad de la nueva generación, cuanto más baja sea, más parecida será a la generación anterior y harán falta más iteraciones para converger. Si ésta es muy alta, cada generación será muy diferente, por lo que puede degenerar en una búsqueda aleatoria.

La mutación es un mecanismo para forzar la variedad en la población, pero una probabilidad de mutación demasiado alta puede dificultar la convergencia.

4.5.5 Cuando usarlos

Los algoritmos genéticos han demostrado su eficacia en múltiples aplicaciones, pero es difícil decir si serán efectivos o no en una aplicación concreta. Por lo general suelen funcionar mejor que los algoritmos de búsqueda por ascenso cuando no se tiene una buena función heurística para guiar el proceso. Pero si se dispone de una buena heurística, estos algoritmos no nos obtendrán mejores resultados y el coste de configurarlos adecuadamente no hará rentable su uso.

Su mayor inconveniente es la gran cantidad de elementos que hemos de ajustar para poder aplicarlos, la codificación del problema, el tamaño de la población, la elección de los operadores, las probabilidades de cruce y mutación, la forma de construir la siguiente generación, ... Siempre nos

⁷Esto refleja lo que sucede en la selección natural, los más aptos tienen una mayor probabilidad de tener descendencia y los menos aptos pueden no llegar a reproducirse.

⁸También es algo que se observa en la naturaleza, las poblaciones aisladas o las especies muy especializadas tienen un pobre acervo genético y no pueden adaptarse a cambios en su entorno.

encontraremos con la duda de si el algoritmo no funciona porque el problema no se adapta al funcionamiento de estos algoritmos, o es que no hemos encontrado los parámetros adecuados para hacer que funcionen bien.

4.5.6 Ejemplo: Las N reinas

El problema de las N reinas es también bastante conocido, consiste en colocar en un tablero de ajedrez de $N \times N$, sin que se maten entre sí, exactamente N reinas. El espacio de búsqueda está formado por todas las formas diferentes de colocar las reinas en el tablero.

El planteamiento para solucionar este problema usando algoritmos genéticos es diferente que para los algoritmos anteriores. Primero hemos de obtener una manera de encontrar un conjunto de soluciones iniciales. La opción más sencilla es generar soluciones colocando las reinas al azar. Hemos de elegir un tamaño para la población y asegurarnos de que sea suficiente grande para tener suficiente variedad de soluciones.

El segundo elemento que nos debemos plantear es la codificación de las soluciones. Como hemos comentado, la manera habitual es transformar la solución en una cadena de bits. En este caso podemos concatenar la codificación en binario de las posiciones de las reinas.

Los operadores de búsqueda se pueden elegir de entre la gran variedad de operadores genéticos existentes, como por ejemplo el cruce por un punto, junto al operador de mutación.

La función heurística en este problema se puede obtener contando el número de reinas que se matan entre sí. En este caso buscamos obtener la solución que hace cero esta función.

Aún nos quedará elegir y ajustar el resto de los parámetros que necesitan los algoritmos genéticos, como la estrategia de selección y las probabilidades de cruce y mutación.

En la figuraa 4.8 y4.9 podemos ver dos iteraciones de un algoritmo genético aplicado en este problema con una población de cuatro individuos y un operador de cruce por un punto. A cada iteración un conjunto de individuos de la población actual es elegido para efectuar los cruces (en este caso individuos con el mejor valor para la función heurística). A estos individuos se les aplica el cruce por un punto para obtener una pareja de descendientes para cada cruce y con cierta probabilidad se aplica una mutación a algún individuo. En este ejemplo un individuo de la población resultante sufre una mutación cambiando una reina de posición. LA población resultante de la iteración 1 es la que es usada en la iteración 2.

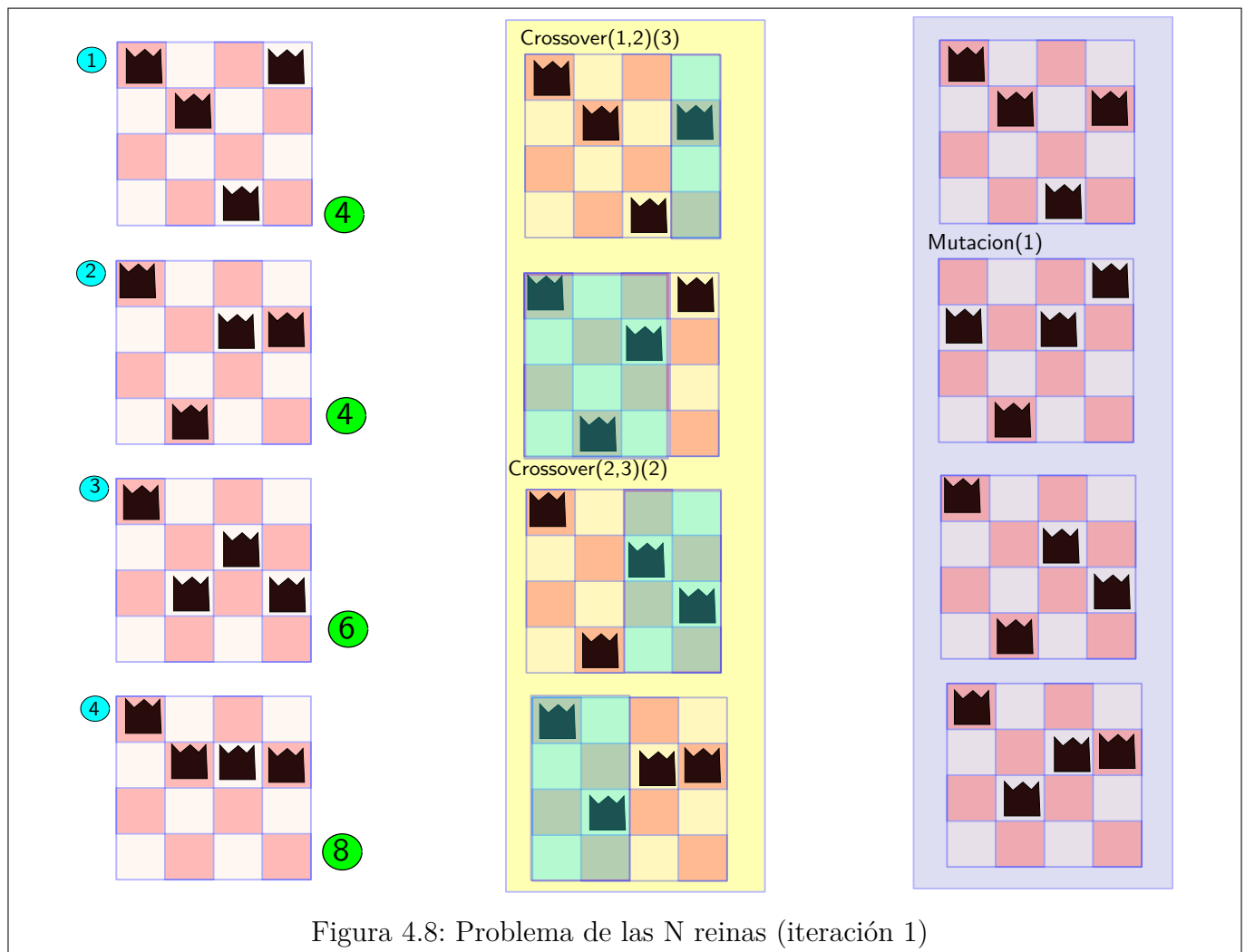
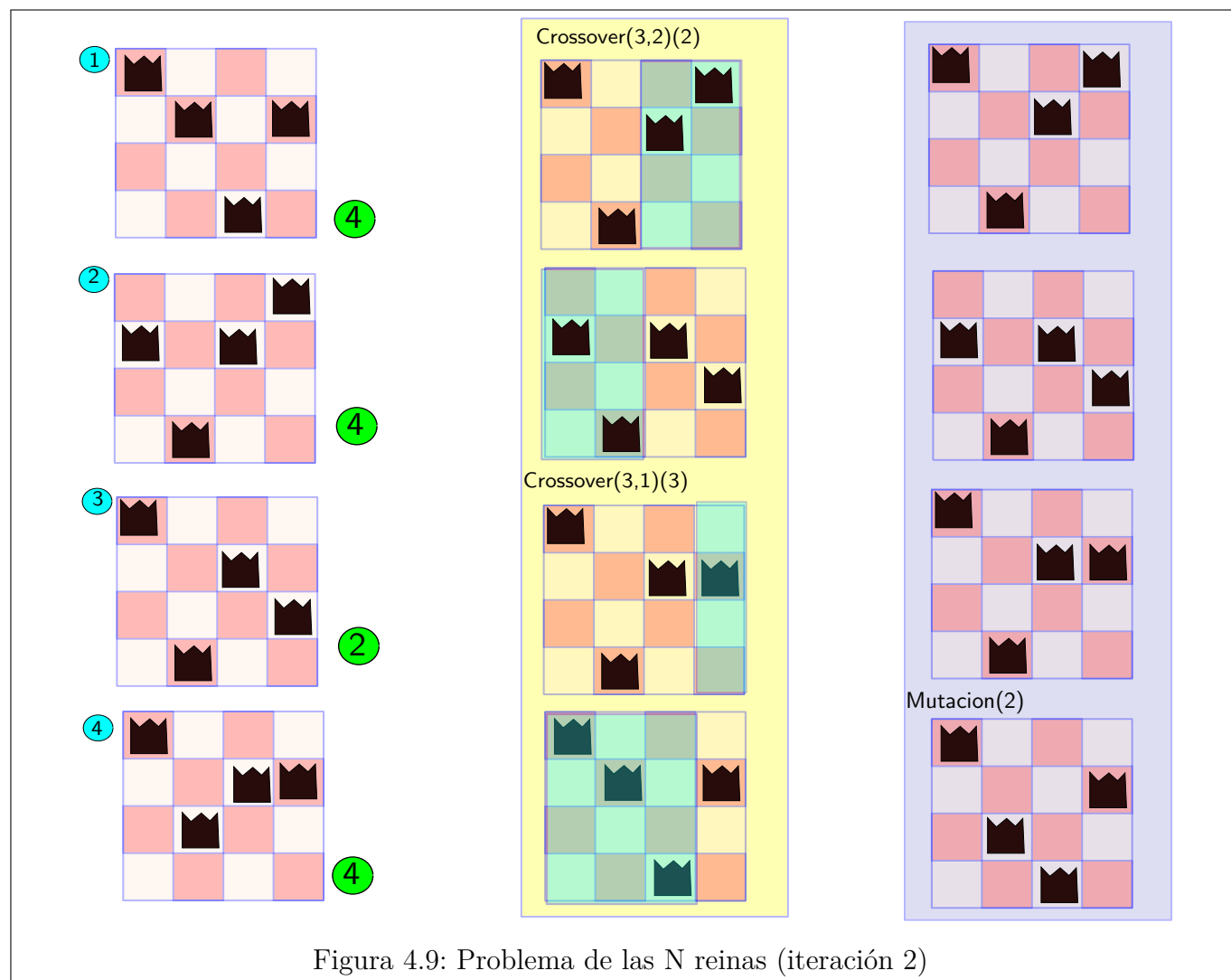


Figura 4.8: Problema de las N reinas (iteración 1)



Búsqueda con adversario

5.1 Tú contra mí o yo contra ti

Hasta ahora habíamos supuesto problemas en los que un solo *agente* intenta obtener un objetivo, en nuestro caso la resolución de un problema. Pero existen casos en los que diferentes agentes compiten por un objetivo que solamente uno de ellos puede alcanzar.

Este tipo de problemas los englobaremos dentro de la categoría de juegos¹. No veremos algoritmos para cualquier tipo de juego, sino que nos restringiremos a los que cumplen un conjunto de propiedades:

- Son bipersonales, es decir, solo hay dos agentes involucrados, y estos juegan alternativamente.
- Todos los participantes tienen conocimiento perfecto, no hay ocultación de información por parte de ninguno de ellos.
- El juego es determinista, no interviene el azar en ninguno de sus elementos.

Los elementos que intervienen en este tipo de problemas se pueden representar de manera parecida a la búsqueda en espacio de estados, tendremos:

- Un estado, que indica las características del juego en un instante específico
- Un estado inicial, que determinará desde donde se empieza a jugar y quién inicia el juego
- Un estado final, o unas características que debe cumplir, que determinarán quién es el ganador
- Unos operadores de transformación de estado que determinarán las jugadas que puede hacer un agente desde el estado actual. Supondremos que los dos jugadores disponen de los mismos operadores

Identificaremos a cada jugador como el jugador **MAX** y el jugador **MIN**. MAX será el jugador que inicia el juego, nos marcaremos como objetivo el encontrar el conjunto de movimientos que ha de hacer el jugador MAX para que llegue al objetivo (ganar) independientemente de lo que haga el jugador MIN².

¹La competición no solamente aparece en el caso de los juegos, pero los algoritmos que veremos también son aplicables.

²No es que nos caiga mal el jugador MIN, lo que pasa es que los algoritmos serán los mismos para los dos, ya que desde la perspectiva de MIN, él es MAX y su oponente es MIN.

Se puede observar que este escenario es bastante diferente del que se plantean los algoritmos de los capítulos anteriores. Nos veremos obligados a revisar el concepto de solución y de óptimo. De hecho estamos pasando de un escenario en el que poseemos todo el conocimiento necesario para resolver un problema desde el principio hasta el fin sin tener que observar la reacción del entorno, a un escenario donde nuestras decisiones se ven influidas por el entorno. En los algoritmos que veremos supondremos que podemos prever hasta cierto punto como reaccionará ese entorno, pero evidentemente no siempre tendrá por que ser ese el caso.

5.2 Una aproximación trivial

Para poder obtener el camino que permite a MAX llegar a un estado ganador, no tenemos más que explorar todas las posibles jugadas que puede hacer cada jugador, hasta encontrar un camino que lleve a un estado final. Para poder obtenerlo podemos utilizar un algoritmo de búsqueda en profundidad que explore exhaustivamente todas las posibilidades. Usamos un algoritmo de búsqueda en profundidad ya que sus restricciones de espacio son lineales.

El algoritmo explorará hasta llegar a una solución evaluándola a $+1$ si es un estado en el que gana MAX, o a -1 si es un estado en el que gana MIN, este valor se propagará hasta el nivel superior. A medida que se acaben de explorar los descendientes de un nivel el valor resultante de su exploración se seguirá propagando a los niveles superiores.

Cada nivel elegirá el valor que propaga a su ascendiente dependiendo de si corresponde a un nivel del jugador MAX o del jugador MIN. En los niveles de MAX se elegirá el valor mayor de todos los descendientes, en los niveles de MIN se elegirá el mínimo. En el momento en el que se propague un $+1$ a la raíz significará que hemos encontrado el camino que permite ganar a MAX independientemente de las jugadas de MIN. En este momento podemos parar la exploración ya que hemos cubierto su objetivo, encontrar una secuencia de pasos ganadora.

En la figura 5.1 podemos ver el espacio de búsqueda que generaría un juego hipotético (los cuadrados corresponden a niveles de MAX y los círculos a niveles de MIN), se han marcado con $+1$ las jugadas en las que gana el jugador MAX y con -1 las jugadas en la que gana el jugador MIN.

Si hacemos el recorrido en profundidad propagando los valores de las jugadas terminales, tendríamos el resultado que se muestra en la figura 5.2. En ella podemos ver marcado el camino que permite ganar a MAX.

Desgraciadamente este algoritmo solo lo podemos aplicar a juegos triviales, ya que el tiempo que necesitaríamos para hacer la exploración completa sería exponencial respecto al número de posibles jugadas en cada nivel (r) y la profundidad de la solución (p) ($O(r^p)$). En cualquier juego real este valor es una cifra astronómica, por ejemplo, el factor de ramificación del ajedrez es aproximadamente de 35 y el del go esta alrededor de 300.

5.3 Seamos un poco más inteligentes

Es evidente que no podemos explorar todo el espacio de búsqueda para obtener el mejor conjunto de jugadas, así que debemos de echar mano del conocimiento que tenemos del juego para reducir la exploración.

Tendremos que introducir una función heurística que en este caso no medirá la bondad de un estado respecto a su distancia hasta una jugada ganadora. La filosofía de esta función heurística es diferente de las que hemos visto hasta ahora, ya que no existe ninguna noción de coste, ni tampoco de optimalidad, pues todas las jugadas ganadoras son igual de buenas. Esto hará que construir esta función sea todavía mas difícil que en las ocasiones anteriores. Habrá que determinar cuál es la

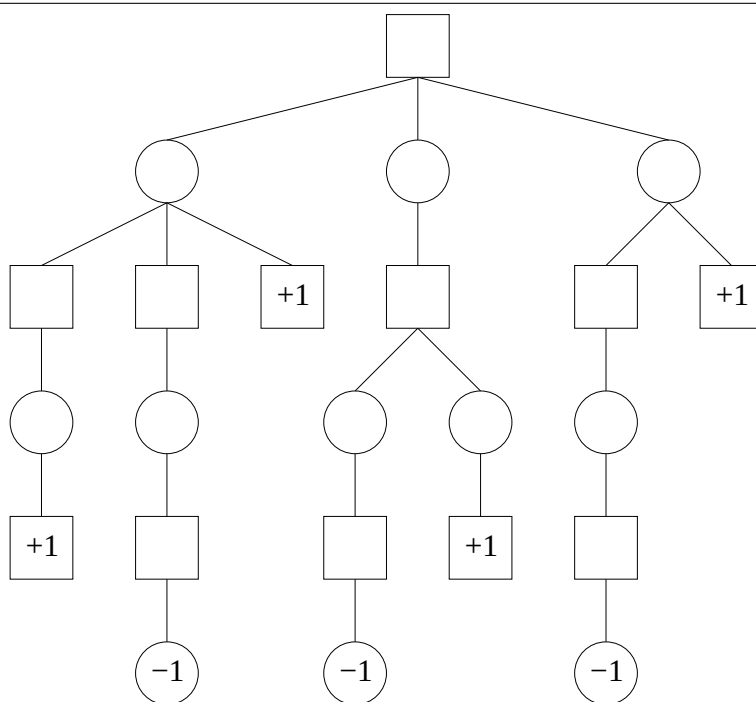


Figura 5.1: Ejemplo de árbol de jugadas

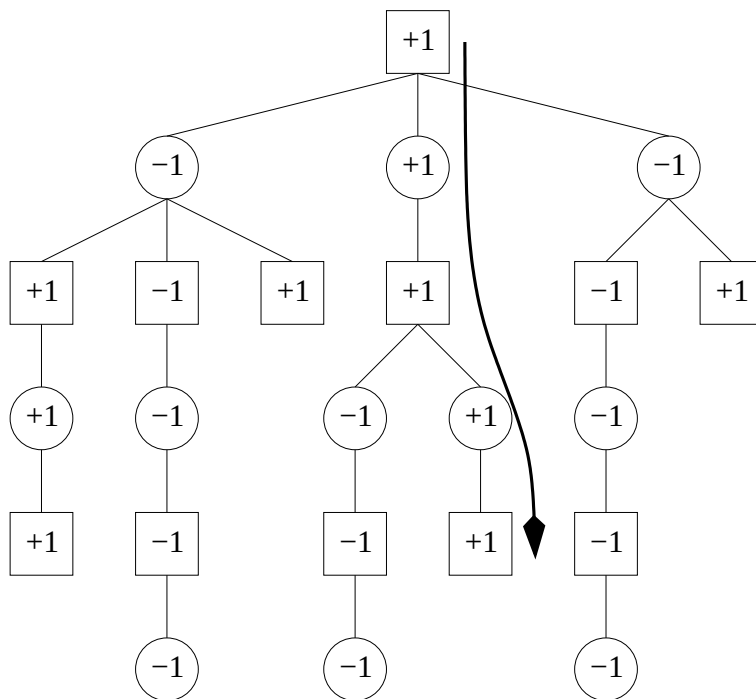


Figura 5.2: Propagación de los valores del árbol de la figura 5.1

ventaja de cada jugador en cada estado³ y esto puede ser bastante difícil de determinar. También hemos de tener en cuenta que el coste de calcular la función heurística también influirá en el coste de la exploración.

Para este tipo de problemas, esta función nos podrá dar valores positivos y negativos, indicándonos qué jugador es el que tiene la ventaja. Consideraremos que si el valor es positivo la jugada es ventajosa para el jugador MAX, y si es negativo la ventaja es para MIN. Por convenio, las jugadas ganadoras de MAX tienen un valor de $+\infty$ y las ganadoras de MIN de $-\infty$.

Además de utilizar una función heurística para guiar la búsqueda, usaremos una estrategia distinta para explorar el conjunto de jugadas. Ya que es imposible hacer una exploración exhaustiva, haremos una búsqueda en profundidad limitada, esto reducirá lo que podremos ver del espacio de búsqueda.

Decidiremos un nivel de profundidad máximo para la búsqueda, evaluaremos los estados que están en ese nivel y averiguaremos cuál es la mejor jugada a la que podemos acceder desde el estado actual. Hay que tener claro que con este procedimiento solo decidimos una jugada, y que repetimos la búsqueda para cada movimiento que tenemos que decidir. A pesar de que repetimos la búsqueda a cada paso, el número de nodos explorados es mucho menor que si exploráramos todo el árbol de búsqueda completo.

La calidad del juego vendrá determinada por la profundidad a la que llegamos en cada exploración. Cuanto más profunda sea la exploración mejor jugaremos, ya que veremos más porción del espacio de búsqueda. Evidentemente, cuanto más exploremos, más coste tendrá la búsqueda.

El algoritmo que realiza esta exploración recibe el nombre de **minimax**⁴ y es un recorrido en profundidad recursivo. El que sea un recorrido en profundidad hace que el coste espacial sea lineal, pero evidentemente el coste temporal será exponencial (dependerá de la profundidad máxima de exploración). El algoritmo 5.1 es su implementación.

El algoritmo tiene una función principal (**MiniMax**) que es la que retorna la mejor jugada que se puede realizar, y dos funciones recursivas mutuas (**valorMax** y **valorMin**) que determinan el valor de las jugadas dependiendo de si el nivel es de MAX o de MIN. La función **estado_terminal(g)** determina si el estado actual es una solución o pertenece al nivel máximo de exploración. La función **evaluacion(g)** calcula el valor de la función heurística para el estado actual.

5.4 Seamos aún más inteligentes

Un punto clave que hemos de tener en cuenta es que cuanto más profunda sea la exploración, mejor podremos tomar la decisión sobre qué jugada debemos escoger. Para juegos en los que el factor de ramificación sea muy grande, no podremos llegar a mucha profundidad, ya que el tiempo que necesitaremos para cada decisión será prohibitivo.

Para reducir este tiempo de exploración se han estudiado las propiedades que tienen estos árboles de búsqueda y se han desarrollado heurísticas que permiten no explorarlos en su totalidad y obtener el mismo resultado que obtendríamos con la exploración completa.

La heurística que veremos se denomina **poda alfa-beta** ($\alpha\beta$ **pruning**). Esta heurística aprovecha que el algoritmo del minimax hace una exploración en profundidad para guardar información sobre cuál es el valor de las mejores jugadas encontradas hasta cierto punto de la búsqueda para el jugador MAX y para el jugador MIN.

Lo que hace esta heurística es evitar seguir explorando nodos que sabemos que no van a variar la decisión que se va a tomar en niveles superiores, eliminándolos de la búsqueda y ahorrando tiempo.

Podemos ver por ejemplo el siguiente árbol de exploración:

³Nótese que la función heurística no solo evaluará los estados ganadores.

⁴El nombre minimax proviene del teorema del Minimax del área de teoría de juegos enunciado por John von Neumann en 1928.

Algoritmo 5.1 Algoritmo minimax

Función: MiniMax (*g*)

movr:movimiento; max,maxc:entero

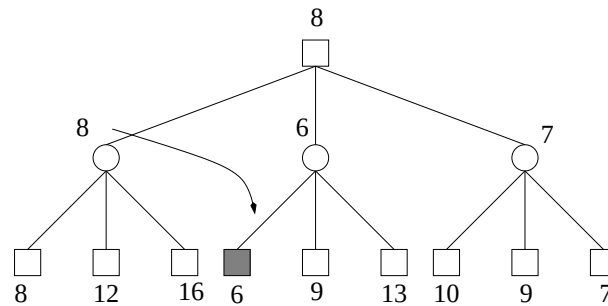
max $\leftarrow \infty$ **para cada** *mov* $\in$ *mouv_posibles*(*g*) **hacer** cmax $\leftarrow$ valor_min(aplicar(*mov*,*g*)) **si** *cmax* > *max* **entonces** max $\leftarrow$ cmax movr $\leftarrow$ *mov* **fin****fin****retorna** *movr***Función:** valorMax (*g*)

vmax:entero

si *estado_terminal*(*g*) **entonces** **retorna** *valor*(*g*)**sino** vmax $\leftarrow -\infty$ **para cada** *mov* $\in$ *mouv_posibles*(*g*) **hacer** vmax $\leftarrow$ **max**(vmax, valorMin(aplicar(*mov*,*g*))) **fin** **retorna** *vmax***fin****Función:** valorMin (*g*)

vmin:entero

si *estado_terminal*(*g*) **entonces** **retorna** *value*(*g*)**sino** vmin $\leftarrow +\infty$ **para cada** *mov* $\in$ *mouv_posibles* **hacer** vmin $\leftarrow$ **min**(vmin, valorMax(aplicar(*mov*,*g*))) **fin** **retorna** *vmin***fin**



Podemos ver que el valor que propagaría el algoritmo del minimax a nodo raíz sería 8. Pero si nos fijamos en el nodo marcado, el segundo descendiente de la raíz ya sabe que el mejor nodo del primer descendiente tiene valor 8, y su primer descendiente vale 6. Dado que la raíz es un nodo MAX, este preferirá el valor 8 al 6. Dado que el segundo descendiente es un nodo MIN, éste siempre escogerá un valor que como máximo será 6. Esto nos hace pensar que, una vez hemos visto que el valor del nodo marcado es 6, el valor que salga de esa exploración no se elegirá (ya que tenemos un valor mejor de la exploración anterior). Esto lleva a la conclusión de que seguir explorando los nodos del segundo descendiente de la raíz no merece la pena, ya que el valor resultante ya no será elegible.

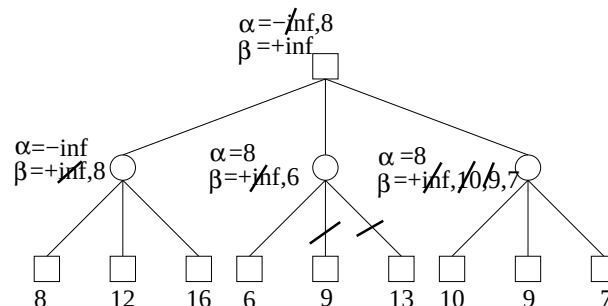
Podemos observar también que esta circunstancia se repite en el tercer descendiente al explorar su último nodo, pero al no quedar más descendientes esta heurística no nos ahorra nada. Esto quiere decir que la efectividad de esta heurística dependerá del orden de exploración de los nodos, pero desgraciadamente es algo que no se puede establecer a priori. En el caso medio, el coste asintótico del tiempo de exploración pasa de ser $O(r^p)$ a ser $O(r^{\frac{p}{2}})$.

Esta heurística modifica el algoritmo del minimax introduciendo dos parámetros en las llamadas de las funciones, α y β , y representarán el límite del valor que pueden tomar las jugadas que exploremos. Una vez superado ese límite sabremos que podemos parar la exploración, porque ya tenemos el valor de la mejor jugada accesible.

La función minimax se mantiene igual, solo varían las funciones que hacen la exploración de cada nivel, su código se puede ver en el del algoritmo 5.2. La llamada que hará la función `MiniMax` a la función `valorMax` le pasará $-\infty$ como valor de `alfa` y $+\infty$ como valor de `beta`.

En la función `valorMax`, `alfa` es el valor que se actualiza y `beta` se mantiene constante representando el valor de la mejor jugada encontrada hasta ahora. En la función `valorMin`, `beta` es el valor que se actualiza y `alfa` se mantiene constante representando el valor de la mejor jugada encontrada hasta ahora.

En la siguiente figura se ve como exploraría el algoritmo de minimax con poda alfa beta el ejemplo anterior:



La ventaja de utilizar la poda alfa beta es que podemos permitirtos ampliar el límite de profundidad de exploración, ya que nos ahorramos la exploración de parte del árbol de búsqueda, y así podemos conseguir mejores resultados.

Algoritmo 5.2 Algoritmo minimax con poda $\alpha\beta$

Función: valorMax (g, α, β)

```

si estado_terminal( $g$ ) entonces
  | retorna valor( $g$ )
sino
  | para cada  $mov \in movs\_posibles(g)$  hacer
  |   |  $\alpha \leftarrow \mathbf{max}(\alpha, \text{valorMin}(\text{aplicar}(mov, g), \alpha, \beta))$ 
  |   | si  $\alpha \geq \beta$  entonces
  |   | | retorna  $\beta$ 
  |   | fin
  | fin
  | retorna  $\alpha$ 
fin

```

Función: valorMin (g, α, β)

```

si estado_terminal( $g$ ) entonces
  | retorna valor( $g$ )
sino
  | para cada  $mov \in movs\_posibles(g)$  hacer
  |   |  $\beta \leftarrow \mathbf{min}(\beta, \text{valorMax}(\text{apply}(mov, g), \alpha, \beta))$ 
  |   | si  $\alpha \geq \beta$  entonces
  |   | | retorna  $\alpha$ 
  |   | fin
  | fin
  | retorna  $\beta$ 
fin

```



Dado que la ordenación de los descendientes es crucial para obtener una poda efectiva se suelen utilizar heurísticas adicionales que permiten aproximar esa ordenación con el objetivo de aumentar las podas. Obviamente esto no es gratis, a veces es necesario repetir expansiones y memorizar nodos para obtenerlo.

Una estrategia habitual es utilizar un algoritmo de profundidad iterativa en lugar del de profundidad limitada para hacer una exploración por niveles. A cada nivel se pueden memorizar las evaluaciones que se obtienen para cada nodo y utilizar esa evaluación en la siguiente iteración. Esto permitirá que en la siguiente iteración se puedan ordenar los descendientes conocidos y explorar primero la rama más prometedora y poder podar con mayor probabilidad el resto de descendientes.

Como ya hemos visto el ratio de reexpansiones respecto a nodos nuevos para la profundidad iterativa tiende a cero con el aumento del factor de ramificación (en estos problemas suele ser muy grande), por lo que el precio que hay que pagar en nodos extras explorados es reducido, si tenemos más posibilidades de llegar a alcanzar la complejidad media de $O(r^{\frac{n}{2}})$.

Satisfacción de restricciones

6.1 De variables y valores

Existen problemas específicos que, por sus propiedades, se pueden resolver más fácilmente utilizando algoritmos adaptados a ellos que utilizando algoritmos generales. Este es el caso de los problemas denominados de **satisfacción de restricciones** (**constraint satisfaction**¹).

Las características de un problema de satisfacción de restricciones son las siguientes:

- El problema se puede representar mediante un conjunto de variables.
- Cada variable tiene un dominio de valores (un conjunto finito de valores discretos o intervalos de valores continuos)
- Existen restricciones de consistencia entre las variables (binarias, n-arias)
- Una solución es una asignación de valores a esas variables

Esto quiere decir que todo problema de satisfacción de restricciones se puede definir a partir de una tripleta de elementos $\mathcal{P} = \langle X, D, R \rangle$, donde X es una lista de variables, D es una lista de los dominios que corresponden a las variables y R es una lista de relaciones entre conjuntos de variables que definen las restricciones que debe cumplir el problema.

A partir de esta formulación genérica se pueden definir especializaciones y generalizaciones. Por ejemplo, si restringimos los dominios de las variables a valores binarios y las relaciones entre variables son disyunciones lógicas, tenemos problemas de satisfactibilidad (si restringimos el número de variables por relación a como mucho 3, tenemos el problema 3-SAT, un ejemplo típico de problema NP-completo). También podemos generalizar el problema admitiendo, por ejemplo, la posibilidad de añadir costes a las variables de manera que tengamos como objetivo optimizar su valor.

El interés de este tipo de problemas viene de su estrecha relación con otras áreas, como satisfactibilidad, álgebra relacional, teoría de base de datos, visión o gráficos y de que muchos problemas reales se pueden plantear como problemas de satisfacción de restricciones, como por ejemplo planificación de tareas, logística (localización o asignación de recursos, personas, vehículos, ...), gestión de redes (electricidad, comunicaciones, ...), diseño/configuración, reconocimiento de formas en imágenes, razonamiento (temporal, espacial, ...), ... Estas aplicaciones hacen que sea un área bastante interesante desde un punto de vista práctico.

Todos los algoritmos de resolución de problemas de satisfacción de restricciones que veremos están pensados para el caso en el que las restricciones sean binarias ya que se pueden representar mediante grafos (**grafos de restricciones**) en los que los nodos son las variables, las restricciones son los arcos,

¹En la literatura los encontrareis abreviados como CSP (constraint satisfaction problems).

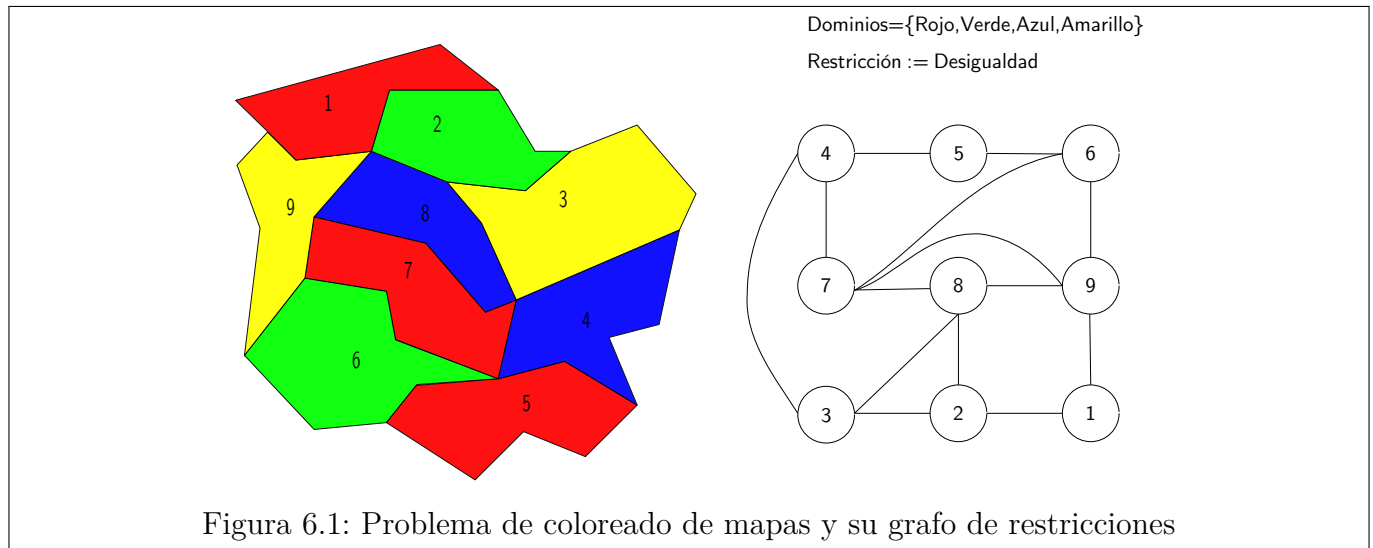


Figura 6.1: Problema de coloreado de mapas y su grafo de restricciones

y los dominios de las variables son los diferentes valores que pueden tomar los nodos. Esto no supone ninguna limitación, dado que la mayor parte de los problemas con restricciones no binarias se pueden convertir en problemas con restricciones binarias introduciendo variables artificiales o transformando el grafo en un hipergrafo.

Ejemplo 6.1 *Un ejemplo clásico de problema de satisfacción de restricciones es el de coloreado de mapas (figura 6.1). El problema consiste en asignar colores a los países de un mapa de manera que dos países adyacentes no tengan el mismo color.*

El problema se puede representar como un grafo donde cada nodo es un país y los arcos entre nodos representan las fronteras comunes. Los nodos del grafo son las variables del problema, los arcos representan las restricciones entre pares de variables (sus valores han de ser diferentes) y los colores disponibles son los dominios de las variables.

6.2 Buscando de manera diferente

Se puede plantear un problema de satisfacción de restricciones como un problema de búsqueda general. Dado que una solución es una asignación de valores a las variables del problema, solo haría falta enumerar de alguna manera todas las posibilidades hasta encontrar alguna que satisfaga las restricciones.

Evidentemente, esta aproximación es irrealizable, ya que el conjunto de posibles asignaciones a explorar es muy grande. Suponiendo n variables y m posibles valores, tendríamos un espacio de búsqueda de $O(m^n)$.

Un problema adicional es que no hay ninguna noción de calidad de solución, ya que todas las asignaciones son igual de buenas y lo único que las diferencia es si satisfacen las restricciones o no, por lo que no podemos confiar en una función heurística que nos indique que combinaciones debemos mirar primero, o como modificar una asignación para acercarla a la solución. Esto nos deja con que los únicos algoritmos que podremos utilizar son los de búsqueda no informada, lo que hará que tengamos que estudiar con más profundidad el problema para poder solucionarlo de una manera eficiente.



Esto no es completamente cierto, ya que siempre podemos utilizar el número de restricciones que no se cumplen como función heurística. Desgraciadamente este número no suele ser muy informativo respecto a lo buenos o malos que pueden ser los vecinos de una no solución, ya que habrá no soluciones con pocas restricciones incumplidas, pero que hagan falta muchas modificaciones para hacerlas solución y al revés, soluciones que a pesar de incumplir varias restricciones, con unas pocas modificaciones se conviertan en solución.

Una ventaja con la que nos encontraremos es que, las propiedades que tienen estos problemas permiten derivar heurísticas que reducen el espacio de búsqueda de manera considerable.

Veremos tres aproximaciones a la resolución de problemas de satisfacción de restricciones, la primera se basará en búsqueda no informada, la segunda se basará en la reducción de espacio de búsqueda del problema y la tercera intentará combinar ambas.

6.2.1 Búsqueda con backtracking

La primera aproximación se basará en un algoritmo de búsqueda no informada. Plantearemos el problema de manera que podamos explorar el conjunto de posibles asignaciones de valores a las variables como una búsqueda en profundidad.

Podemos observar que, dada una asignación parcial de valores a las variables de un problema, ésta podría formar parte de una solución completa si no viola ninguna restricción, por lo tanto solo tendríamos que encontrar el resto de valores necesarios para completarla.

Esto nos hace pensar que podemos plantear la búsqueda más fácilmente si exploramos en el espacio de las soluciones parciales, que si lo planteamos como una búsqueda en el espacio de las soluciones completas como habíamos comentado antes. Esto nos sugiere un algoritmo para hacer la exploración.

Podemos establecer un orden fijo para las variables y explorar de manera sistemática el espacio de soluciones parciales, asignando valores a las variables de manera consecutiva en el orden que hemos fijado. En el momento en el que una solución parcial viole alguna de las restricciones del problema, sabremos que no formará parte de ninguna solución completa, por lo que replantearemos las asignaciones que hemos realizado. A este tipo de búsqueda se la denomina búsqueda con **backtracking cronológico**. Se puede ver su implementación en el algoritmo 6.1.

Definiremos tres conjuntos con las variables del problema. Por un lado tendremos las variables pasadas, que serán las que ya tienen un valor asignado y forman parte de la solución parcial. Tendremos la variable actual, que será a la que debemos asignarle un valor. Por último, tendremos las variables futuras, que serán las que aún no tienen valor.

El algoritmo hace un recorrido en profundidad de manera recursiva por todas las asignaciones posibles de valores a cada una de las variables del problema. El parámetro `vfuturas` contiene todas las variables del problema en orden, el parámetro `solución` irá guardando las soluciones parciales. El hacer el recorrido en profundidad hace que el coste espacial de la búsqueda sea lineal.

El caso base de la recursividad es que el conjunto de variables futuras se haya agotado, el caso recursivo es iterar para todos los valores de la variable actual. El algoritmo continua las llamadas recursivas cuando consigue una asignación de valor a la variable actual que cumpla las restricciones del problema (`solucion.válida()`). En el caso de que ninguna asignación dé una solución parcial válida, el algoritmo permite que se cambie el valor de la variable anterior retornando la solución en un estado de fallo.

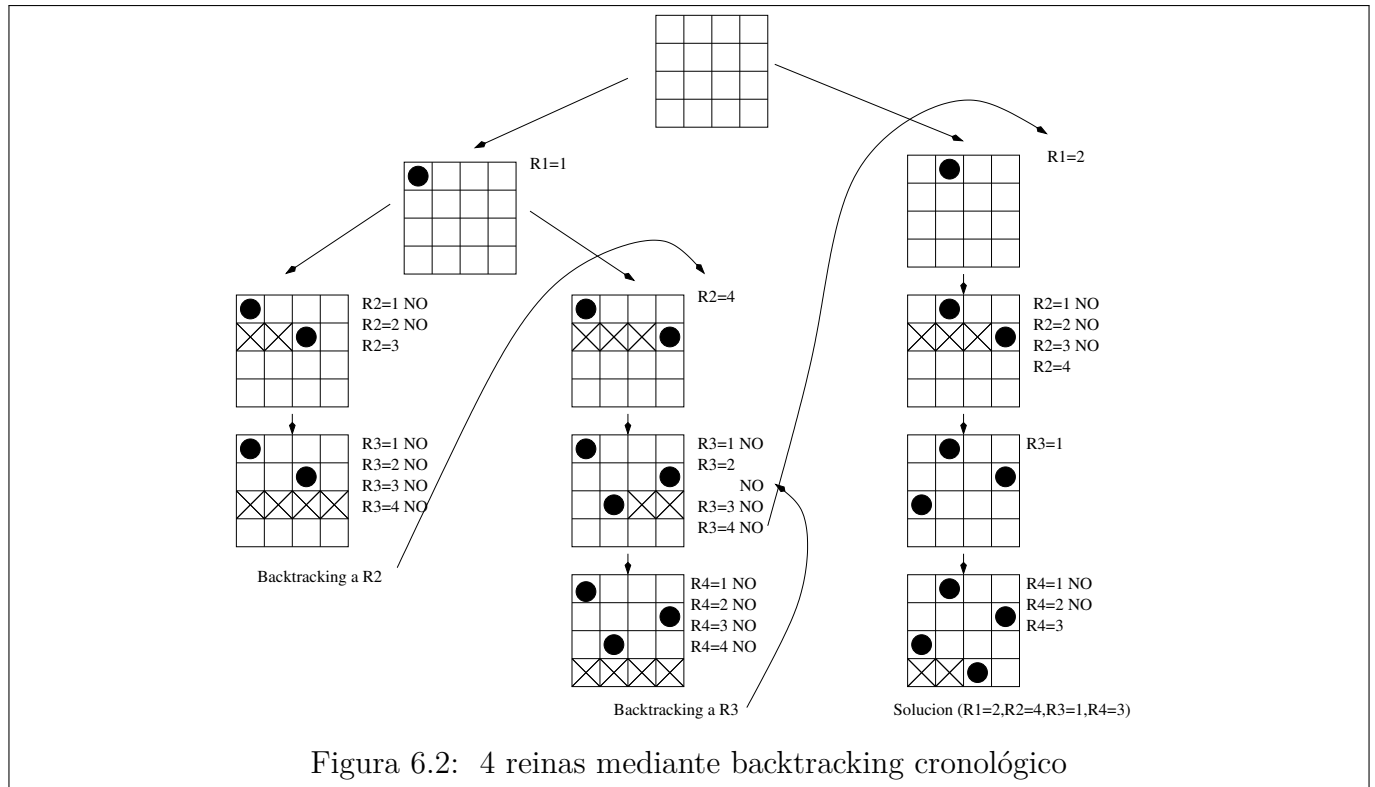
Ejemplo 6.2 Otro problema clásico de satisfacción de restricciones es el de la N reinas que ya hemos comentado en el primer capítulo. Éste se puede plantear como un problema de satisfacción de

Algoritmo 6.1 Algoritmo de backtracking cronológico

```

Función: backtracking_cronologico(vfuturas, solucion)
si vfuturas.es_vacio?() entonces
  | retorna solucion
sino
  | vactual ← vfuturas.primer()
  | vfuturas.borrar_primer()
  | para cada v ∈ vactual.valores() hacer
  |   | vactual.asignar(v)
  |   | solucion.anadir(vactual)
  |   | si solucion.valida() entonces
  |   |   | solucion ← backtracking_cronologico(vfuturas,solucion)
  |   |   | si no solucion.es_fallo?() entonces
  |   |   |   | retorna solucion
  |   |   |   | sino
  |   |   |   |   | solucion.borrar(vactual)
  |   |   |   | sino
  |   |   |   |   | solucion.borrar(vactual)
  |   |   | retorna solucion.fallo()

```



restricciones en el que las reinas serían las variables, el dominio de valores sería la columna en la que se coloca la reina y las restricciones serían que las posiciones en las que colocamos las reinas no hagan que se maten entre si.

En la figura 6.2 podemos ver como exploraría el algoritmo de backtracking cronológico el problema en el caso de 4 reinas.

En el ejemplo se puede ver como el algoritmo va probando asignaciones para cada una de las reinas, avanzando en el momento en el que encuentra una asignación compatible con las asignaciones previas. En el caso de no haber ninguna asignación compatible, el algoritmo retrocede hasta la primera variable a la que le quedan valores por probar.

El algoritmo de backtracking cronológico puede hacer bastante trabajo innecesario a veces si la variable que provoca la vuelta atrás del algoritmo no es precisamente la última asignada, haciendo que se tengan que probar todas las combinaciones posibles de las variables que hay entre el primer punto donde se encuentra la violación de la restricción y la variable que la provoca. Esto ha hecho desarrollar variantes más inteligentes del backtracking cronológico, la mas sencilla es la denominada **backjumping**, que intenta que el backtracking no sea a la variable anterior, sino que, teniendo en cuenta las restricciones involucradas, vuelva a la variable que tiene alguna restricción con la variable actual. Esto consigue evitar todas las pruebas de valores entre la variable actual y esa variable.

6.2.2 Propagación de restricciones

Otras vías a través de las que atacar este tipo de problemas son las denominadas técnicas de **propagación de restricciones**. Mediante éstas se pretende reducir el número de combinaciones que hay que probar, descartando todas aquellas que contengan asignaciones de valores que sabemos a priori que no pueden aparecer en ninguna solución.

Esta técnicas también sirven para reducir el número de vueltas atrás inútiles que realiza el algoritmo de backtracking cronológico como veremos en la siguiente sección, eliminando el mayor número posible de incompatibilidades entre los valores de las variables, de manera que no tengan que ser exploradas.

Estas técnicas se han desarrollado a partir de heurísticas derivadas de propiedades que tienen los grafos de restricciones. Las más utilizadas son las que se derivan de la propiedad denominada **k-consistencia**.

Se dice que un grafo de restricciones es **k-consistente** si para cada variable X_i del grafo, cada conjunto de k variables que la incluya y cada valor del dominio de X_i , podemos encontrar una combinación de k valores de estas variables que no viola las restricciones entre ellas. La ventaja de estas propiedades es que se pueden comprobar con algoritmos de coste polinómico.

Esta propiedad la podemos definir para cualquier valor de k , pero típicamente se utiliza el caso de k igual a 2, que en este caso se denomina **arco-consistencia**. Diremos que un grafo de restricciones es **arco-consistente**, si para cada pareja de variables del grafo, para cada uno de los valores de una de las variables, podemos encontrar otro valor de la otra variable que no viola las restricciones entre ellas.

Esta propiedad permite eliminar valores del dominio de una variable. Si dado un valor de una variable y dada otra variable con la que tenga una restricción, no podemos encontrar algún valor en la otra variable que no viole esa restricción, entonces podemos eliminar el valor, ya que no puede formar parte de una solución.

El objetivo será pues, a partir de un grafo de restricciones, eliminar todos aquellos valores que no hagan arco consistente el grafo, de manera que reduzcamos el número de posibles valores que pueden tener las variables del problema y por lo tanto el número de combinaciones a explorar.

El algoritmo 6.2 que convierte un grafo de restricciones en arco-consistente. Este algoritmo es conocido como **AC3**, su coste temporal es $O(rd^3)$ y su coste espacial es $O(r)$, siendo r el número

Algoritmo 6.2 Algoritmo de arco-consistencia**Algoritmo:** Arco consistencia

```

R ← conjunto de arcos del problema /* ambos sentidos */
mientras se modifiquen los dominios de las variables hacer
    r ← extraer_arco(R)
    /*  $r_i$  es la variable del origen del arco */
    /*  $r_j$  es la variable del destino del arco */
    para cada  $v \in$  en el dominio de  $r_i$  hacer
        si  $v$  no tiene ningún valor en el dominio de  $r_j$  que cumpla  $r$  entonces
            borrar  $v$  del dominio de  $r_i$ 
            añadir todos los arcos que tengan como destino  $r_i$  menos el  $(r_j \rightarrow r_i)$ 
        fin
    fin
fin

```

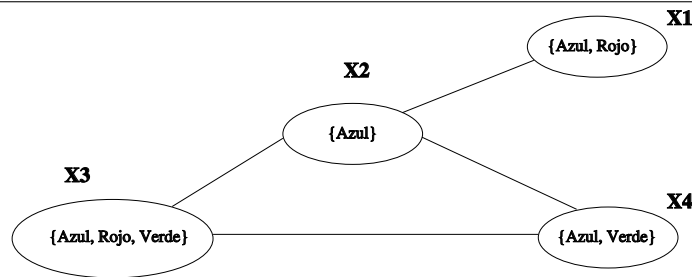


Figura 6.3: Ejemplo de coloreado de grafos

de restricciones del problema (contando los dos sentidos) y d es el tamaño de los dominios, existen algoritmos más eficientes, a costa de usar más espacio que tienen complejidad temporal $O(rd^2)$ y complejidad espacial $O(rd)$.

Ejemplo 6.3 En la figura 6.3 tenemos un problema de coloreado de grafos donde en cada uno de los nodos se indica el dominio de cada variable. El objetivo será colorear el grafo de manera que ningún nodo adyacente tenga el mismo color, por lo que cada arco es una restricción de desigualdad.

El conjunto de arcos con los que comenzaría el algoritmo son:

$$\{X_1 - X_2, X_2 - X_1, X_2 - X_3, X_3 - X_2, X_2 - X_4, X_4 - X_2, X_3 - X_4, X_4 - X_3\}$$

La ejecución del algoritmo sería la siguiente:

1. Seleccionamos $X_1 - X_2$ Eliminamos Azul del dominio de X_1
2. Seleccionamos $X_2 - X_1$ Todos los valores son consistentes
3. Seleccionamos $X_2 - X_3$ Todos los valores son consistentes
4. Seleccionamos $X_3 - X_2$ Eliminamos Azul del dominio de X_3
Tendríamos que añadir el arco $X_4 - X_3$
pero ya está
5. Seleccionamos $X_2 - X_4$ Todos los valores son consistentes
6. Seleccionamos $X_4 - X_2$ Eliminamos Azul del dominio de X_4
Tendríamos que añadir el arco $X_3 - X_4$
pero ya está
7. Seleccionamos $X_3 - X_4$ Eliminamos Verde del dominio de X_3
Añadimos el arco $X_2 - X_3$
8. Seleccionamos $X_4 - X_3$ Todos los valores son consistentes
9. Seleccionamos $X_2 - X_3$ Todos los valores son consistentes

Casualmente, al hacer arco-consistente el grafo hemos hallado la solución, pero no siempre tiene por que ser así.

La arco-consistencia no siempre nos asegura una reducción en los dominios del problema, por ejemplo, si consideramos el grafo del problema de las 4 reinas que hemos visto en el ejemplo 6.2, éste es arco-consistente, por lo que no se puede eliminar ninguno de los valores de las variables.

La k -consistencia para el caso de k igual a 3 se denomina **camino consistencia**, y permite eliminar más combinaciones, pero con un coste mayor. Existe la propiedad que se denomina **k-consistencia fuerte**, que se cumple cuando un grafo cumple la propiedad de consistencia desde 2 hasta k . Esta propiedad asegura que si el grafo tiene anchura² k y es k -consistente fuerte, encontraremos una solución sin necesidad de hacer backtracking, desgraciadamente probar esto necesita un tiempo muy elevado ($O(r^k d^k)$). Un resultado interesante es que, si el grafo de restricciones es un árbol, este se puede resolver sin backtracking si se convierte en arco consistente.

6.2.3 Combinando búsqueda y propagación

La combinación de la búsqueda con backtracking y la propagación de restricciones es la que da resultados más exitosos en estos problemas. La idea consiste en que cada vez que asignemos un valor a una variable realicemos la propagación de las restricciones que induce esa asignación, eliminando así valores de las variables que quedan por asignar. Si con esta propagación, alguna de las variables se queda sin valores, deberemos hacer backtracking.

La ventaja de este método es que podremos hacer backtracking antes de que lleguemos a la variable que acabará provocándolo. De esta manera nos ahorraremos todos los pasos intermedios.

Existen diferentes algoritmos que combinan búsqueda y propagación, dependiendo de que tipo de consistencia se evalúe en cada paso de búsqueda. El algoritmo más sencillo es el denominado de **forward checking**.

Este algoritmo añade a cada paso de asignación de variables una prueba de arco-consistencia entre las variables futuras y la variable actual. Esta prueba de arco consistencia consigue que todos los valores de las variables futuras que no sean consistentes con la asignación a la variable actual sean eliminados, de manera que si algún dominio de alguna variable futura queda vacío podemos reconsiderar la asignación actual o hacer backtracking.

El algoritmo (ver algoritmo 6.3) es muy parecido al que hemos visto para el backtracking cronológico, ahora incluiríamos la reducción de dominios mediante la prueba de arco consistencia tras hacer la asignación de valor a la variable actual (`vfuturas.propagar_restricciones(vactual)`) y

²Se puede calcular la anchura de un grafo en coste cuadrático.

Algoritmo 6.3 Algoritmo de forward checking

```

Función: forward checking (vfuturas, solucion)
si vfuturas.es_vacio?() entonces
  | retorna solucion
sino
  | vactual ← vfuturas.primer()
  | vfuturas.borrar_primer()
  | para cada v ∈ vactual.valores() hacer
  |   | vactual.asignar(v)
  |   | solucion.anadir(vactual)
  |   | vfuturas.propagar_restricciones(vactual) /* forward checking */
  |   | si no vfuturas.algun_dominio_vacio?() entonces
  |   |   | solucion ← forward_checking(vfuturas, solucion)
  |   |   | si no solucion.es_fallo?() entonces
  |   |   |   | retorna solucion
  |   |   |   | sino
  |   |   |   |   | solucion.borrar(vactual)
  |   |   | sino
  |   |   |   | solucion.borrar(vactual)
  | retorna solucion.fallo()

```

la comprobación de la validez de la solución parcial se haría mediante la comprobación de que todas las variables futuras tienen aún valores (`vfuturas.algun_dominio_vacio?()`).

Ejemplo 6.4 En la figura 6.4 podemos ver de nuevo el ejemplo 6.2 utilizando el algoritmo de forward checking. Se puede comprobar que el número de asignaciones que se realizan es menor, ya que los valores no compatibles son eliminados por las pruebas de consistencia. Ahora el coste ha pasado a las pruebas de consistencia que se realizan en cada iteración.

En este caso a cada paso el algoritmo asigna un valor a la reina actual y elimina de los dominios de las reinas futuras las asignaciones que son incompatibles con la actual. Esto nos permite hacer backtracking antes que en el ejemplo previo, reduciendo el número de asignaciones exploradas.

Hay que tener en cuenta que la prueba de arco consistencia de cada iteración incluye la comprobación de los valores que quedan en las variables futuras, por lo que dependiendo del problema el ahorro puede no existir.

Existen algoritmos que realizan pruebas de consistencia mas fuertes, como por ejemplo el algoritmo **RFL (Real Full Lookahead)** que además comprueba la arco-consistencia entre todas las variables futuras, o el algoritmo **MAC (Maintaining Arc Consistency)** que convierte el problema en arco consistente a cada paso.

Evidentemente, esta reducción de los dominios de las variables no sale gratis, el número de comprobaciones a cada paso incrementa el coste por iteración del algoritmo. Dependiendo del problema, el coste de propagación puede hacer que el coste de hallar una solución sea mayor que utilizar el algoritmo de backtracking cronológico.

6.3 Otra vuelta de tuerca

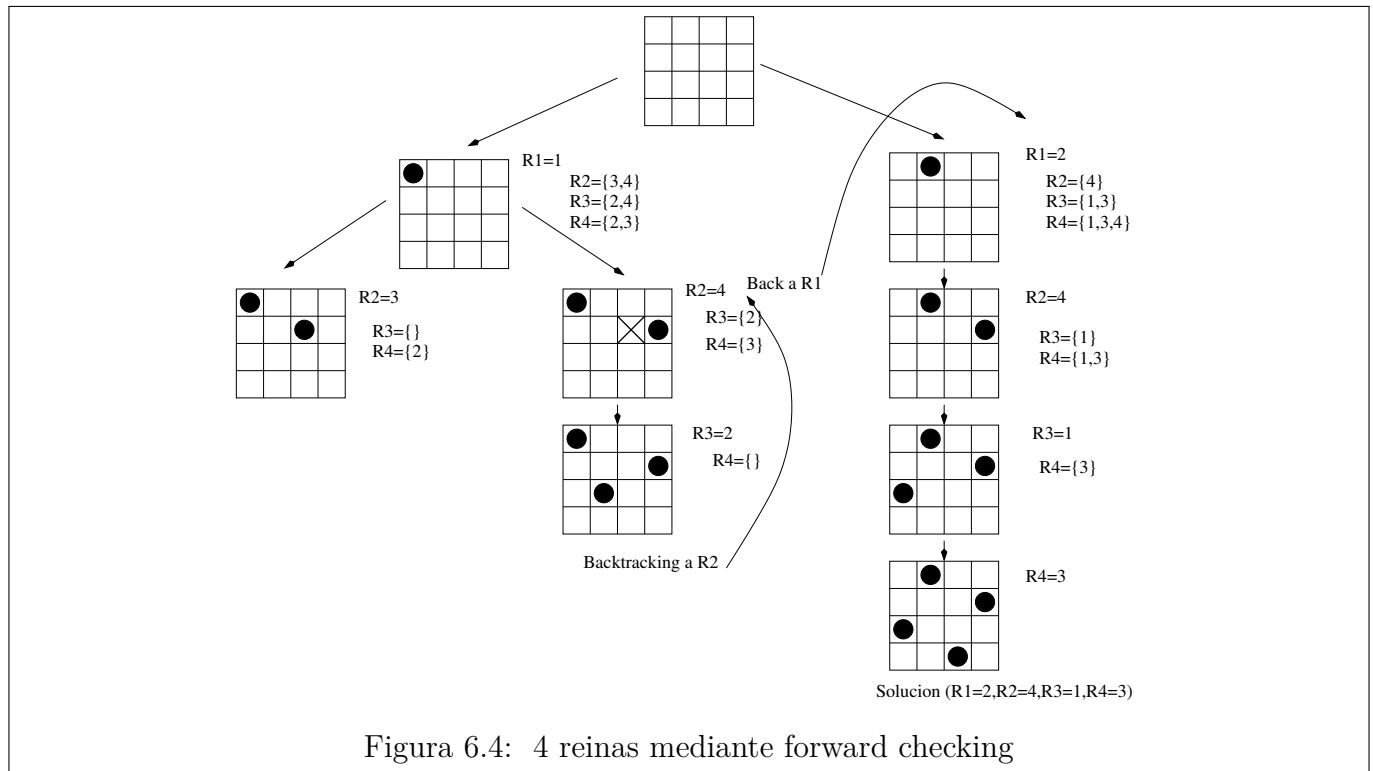


Figura 6.4: 4 reinas mediante forward checking

Muchas veces estas heurísticas no son suficientes para reducir el coste de encontrar una solución, por lo que se combinan con otras que también se han mostrado efectivas en este tipo de problemas.

Hasta ahora siempre hemos mantenido un orden fijo en la asignación de las variables, pero este puede no ser el más adecuado a la hora de realizar la exploración. Se utilizan diferentes heurísticas de ordenación dinámica de variables de manera que siempre se escoja la que tenga más probabilidad de llevarnos antes a darnos cuenta de si una solución parcial no nos llevará a una solución.

Las heurísticas más utilizadas son:

- **Dominios mínimos (Dynamic value ordering)**, escogemos la variable con el menor número de valores.
- **Min width ordering**, escogemos la variable conectada al menor número de variables no instanciadas.
- **Max degree ordering**, escogemos la variable más conectada en el grafo original.

No es despreciable el ahorro que se puede obtener mediante este tipo de heurísticas, por ejemplo, para el problema de las 20 reinas, hallar la primera solución con backtracking cronológico necesita alrededor de $2.5 \cdot 10^7$ asignaciones, utilizando forward checking se reduce a alrededor de $2.4 \cdot 10^6$ asignaciones, y si utilizamos la heurística de dominios mínimos combinada con el forward checking hacen falta alrededor de $4 \cdot 10^3$ asignaciones.

6.4 Ejemplo: El Sudoku

Muchos de los rompecabezas que aparecen habitualmente en la sección de pasatiempos son problemas de satisfacción de restricciones. Esto quiere decir que se pueden formular como un conjunto de variables a las que les corresponde un dominio de valores y un conjunto de restricciones sobre ellas.

1	2	3	4	5	6	7	8	9
4	5	6	7	8	9	1	2	3
7	8	9	1	2	3	4	5	6
2	3	1	5	6	4	8	9	7
5	6	4	8	9	7	2	3	1
8	9	7	2	3	1	5	6	4
3	1	2	9	7	8	6	4	5
6	4	5	3	1	2	9	7	8
9	7	8	6	4	5	3	1	2

Figura 6.5: Una solución al juego del sudoku

Uno de estos rompecabezas es el sudoku (figura 6.5) para este problema el espacio de búsqueda es de $9!^9 \approx 2.3 \times 10^{33}$ posibles tableros, de los que aproximadamente 6.7×10^{24} son solución. En este caso tenemos una matriz de 9×9 , lo que nos da un total de 81 variables, cada variable puede contener un número del 1 a 9. Las restricciones aparecen primero entre las variables según las filas y columnas, de manera que para una fila o columna un número puede aparecer solamente una vez y además hay una restricción adicional entre las variables de las nueve submatrices que forman el problema, dentro de cada submatriz no puede haber dos variables con el mismo número.

En el caso de este problema, el entretenimiento no viene de encontrar una solución, sino de obtener una solución dado que hemos dado ya ciertos valores a algunas de las variables. La dificultad de encontrar la solución se puede variar estableciendo el número de variables a las que damos el valor.

Parte II

Representación del conocimiento

Introducción a la representación del conocimiento

7.1 Introducción

Acabamos de ver un conjunto de algoritmos que son capaces de resolver problemas mediante la exploración del espacio de soluciones. Estos algoritmos se basan en una información mínima para plantear la resolución. No solo en lo que respecta a las características que representan al problema, sino al conocimiento involucrado en las tomas de decisión durante la exploración.

Las funciones heurísticas que se utilizan tienen en cuenta información general que evalúa el problema a cada paso de la misma manera. Evidentemente, durante la resolución de un problema no tiene por qué evaluarse su estado siempre de la misma manera, ni tiene por qué tener siempre la misma importancia toda la información de la que disponemos.

Si observamos como nosotros mismos resolvemos problemas, podemos fijarnos en que la información que vamos teniendo en cuenta a medida que lo resolvemos va cambiando. No usamos, por ejemplo, la misma información cuando iniciamos la resolución de un problema que cuando estamos en medio de la resolución, ya que nuestra incertidumbre sobre donde está la solución va cambiando, y parte de la información puede pasar a ser crucial o irrelevante. Esta capacidad para tomar mejores decisiones puede hacer que un problema se pueda resolver de manera más eficiente.

Estas decisiones no las podemos tomar si no tenemos un conocimiento profundo de las características del problema y de como nos pueden ayudar a tomar decisiones. Esto nos lleva a pensar que en todo problema complejo en IA, se debe plantear qué conocimiento nos puede ser necesario para resolverlo y como deberemos utilizarlo para hacer más eficiente la resolución.

El conocimiento que podemos usar puede ser tanto general como dependiente del dominio, pero teniendo en cuenta que el conocimiento específico del problema será el que con más probabilidad nos permitirá hacer eficiente la resolución.

Esta necesidad de introducir más conocimiento en la resolución de un problema nos lleva a plantearnos dos cuestiones. Primero, el cómo escoger el formalismo de representación que nos permita hacer una traducción fácil del mundo real a la representación. El conocimiento necesario es por lo general bastante complejo, hacer la traslación de los elementos del dominio a una representación es una tarea costosa. Segundo, el cómo ha de ser esa representación para que pueda ser utilizada de forma eficiente. Este conocimiento tendrá que utilizarse en el proceso de toma de decisiones, deberemos evaluar cuando es necesario usarlo y qué podemos obtener a partir de él. Sin un mecanismo eficiente para su uso no conseguiremos ninguna ganancia.

En este punto debemos distinguir entre *información* y *conocimiento*. Denominaremos **información** al conjunto de datos básicos, sin interpretar, que se obtienen como entrada del sistema. Por ejemplo los datos numéricos que aparecen en una analítica de sangre o los datos de los sensores de una planta química. Estos datos no nos dicen nada si no los asociamos a su significado en el problema.

Denominaremos **conocimiento** al conjunto de datos de primer orden, que modelan de forma estructurada la experiencia que se tiene sobre un cierto dominio o que surgen de interpretar los datos básicos. Por ejemplo, la interpretación de los valores de la analítica de sangre o de los sensores de la planta química para decir si son normales, altos o bajos, preocupantes, peligrosos, ..., el conjunto de estructuras de datos y métodos para diagnosticar a pacientes a partir de la interpretación del análisis de sangre, o para ayudar en la toma de decisiones de que hacer en la planta química

Los sistemas de IA necesitan diferentes tipos de conocimiento que no suelen estar disponibles en bases de datos y otras fuentes de información:

- Conocimiento sobre los objetos en un entorno y las posibles relaciones entre ellos
- Conocimiento sobre los procesos en los que interviene o que le son útiles
- Conocimiento difícil de representar como datos básicos, como la intensionalidad, la causalidad, los objetivos, información temporal, conocimiento que para los humanos es “de sentido común”, etc.

Podríamos decir para un programa de inteligencia artificial el conocimiento es la combinación entre la información y la forma en la que se debe interpretar¹.

7.2 Esquema de representación

Para poder representar algo necesitamos saber entre otras cosas sus características o su estructura, que uso le dan los seres inteligentes, como adquirirlo y traspasarlo a un sistema computacional, qué estructuras de datos son adecuadas para almacenarlo y qué algoritmos y métodos permiten manipularlo.

Una posibilidad sería fijarnos en como los seres humanos representamos y manipulamos el conocimiento que poseemos. Por desgracia no hay respuestas completas para todas estas preguntas desde el punto de vista biológico o neurofisiológico, no tenemos una idea clara de como nuestro cerebro es capaz de adquirir, manipular y usar el conocimiento que utilizamos para manejarnos en nuestro entorno y resolver los problemas que este nos plantea.

Afortunadamente podemos buscar respuestas en otras áreas, principalmente la lógica. A partir de ella construiremos modelos que simulen la adquisición, estructuración y manipulación del conocimiento y que nos permitan crear sistemas artificiales inteligentes.

Como sistema para representar el conocimiento hablaremos de **esquema de representación** que para nosotros será un instrumento para codificar la realidad en un ordenador. Desde un punto de vista informático un esquema de representación puede ser descrito como una combinación de:

- Estructuras de datos que codifican el problema en curso con el que se enfrenta el agente → **Parte estática**
- Estructuras de datos que almacenan conocimiento referente al entorno en el que se desarrolla el problema y procedimientos que manipulan las estructuras de forma consistente con una interpretación plausible de las mismas → **Parte dinámica**

La parte estática estará formada por:

- Estructura de datos que codifica el problema

¹Se suele dar la ecuación *Conocimiento = Información + Interpretación* como visión intuitiva de este concepto parafraseando el libro de N. Wirth *Algorithms + Data Structures = Programs*

- Operaciones que permiten crear, modificar y destruir elementos en la estructura
- Predicados que dan un mecanismo para consultar esta estructura de datos
- Semántica de la estructura, se definirá una función que establezca una relación entre los elementos de la realidad y los elementos de la representación. Esta función es la que nos permitirá interpretar lo que hagamos en la representación en términos del mundo real.

La parte dinámica estará formada por:

- Estructuras de datos que almacenan conocimiento referente al entorno/dominio en el que se desarrolla el problema
- Procedimientos que permiten
 - Interpretar los datos del problema (de la parte estática) a partir del conocimiento del dominio (de la parte dinámica)
 - Controlar el uso de los datos: estrategias de control, es decir, métodos que nos permiten decidir cuando usamos el conocimiento y como éste guía los procesos de decisión.
 - Adquirir nuevo conocimiento: mecanismos que nos permiten introducir nuevo conocimiento en la representación, ya sea por observación del mundo real o como deducción a partir de la manipulación del conocimiento del problema.

Es importante tener en mente que la realidad no es el esquema de representación. De hecho, podemos representar la realidad utilizando diferentes esquemas de representación. Esta es simplemente una abstracción que nos permite resolver los problemas del dominio en un entorno computacional.

Se ha de tener siempre en cuenta que nuestra representación siempre es incompleta, debido a:

- Modificaciones: el mundo es cambiante, pero nuestras representaciones son de un instante
- Volumen: mucho (demasiado) conocimiento a representar, siempre tendremos una representación parcial de la realidad
- Complejidad: La realidad tiene una gran riqueza en detalles y es imposible representarlos todos

El problema de la representación cambiante del mundo está ligado a los procedimientos de adquisición y mantenimiento de la representación. Para poder tener una representación fiel, deberíamos poder introducir en esta toda aquella información que es consecuencia del cambio de la realidad, esto requiere poder representar todo lo que observamos en la realidad y obtener todas las consecuencias lógicas de ese cambio². La eficiencia de los métodos de adquisición es clave, el problema es que no existe ningún mecanismo lo suficientemente eficiente como para que sea plausible desde el punto de vista computacional mantener una representación completa de la realidad.

Los problemas de volumen y complejidad de la realidad están relacionados con la granularidad de la representación. Cuanto mayor sea el nivel de detalle con el que queramos representar la realidad, mayor será el volumen de información que tendremos que representar y manipular. Esto hace que el coste computacional de manejar la representación crezca de manera exponencial, haciendo poco plausible llegar a ciertos niveles de detalle y obligando a que la representación sea una simplificación de la realidad.

²A este problema se le ha denominado el problema del *marco de referencia* (*frame problem*).

7.3 Propiedades de un esquema de representación

Antes de plantearnos mecanismos específicos para la representación del conocimiento, vamos a analizar qué propiedades debería tener un buen formalismo de representación:

- *Ligadas a la representación*

- **Adecuación Representacional:** Habilidad para representar todas las clases de conocimiento que son necesarias en el dominio. Se puede necesitar representar diferentes tipos de conocimiento, cada tipo necesita que el esquema de representación sea capaz de describirlo, por ejemplo conocimiento general, negaciones, relaciones estructurales, conocimiento causal, ... Cada esquema de representación tendrá un lenguaje que permitirá expresar algunos de estos tipos de conocimiento, pero probablemente no todos.
- **Adecuación Inferencial:** Habilidad para manipular estructuras de representación de tal manera que devengan o generen nuevas estructuras que correspondan a nuevos conocimientos inferidos de los anteriores. El esquema de representación debe definir los algoritmos que permitan inferir nuevo conocimiento. Estos algoritmos pueden ser específicos del esquema de representación o puede ser genéricos.

- *Ligadas al uso de la representación*

- **Eficiencia Inferencial:** Capacidad del sistema para incorporar conocimiento adicional a la estructura de representación, llamado metaconocimiento, que puede emplearse para focalizar la atención de los mecanismos de inferencia con el fin de optimizar los cálculos. El esquema de representación puede utilizar mecanismos específicos que aprovechen el conocimiento para reducir el espacio de búsqueda del problema.
- **Eficiencia en la Adquisición:** Capacidad de incorporar fácilmente nueva información. Idealmente el sistema por sí mismo deberá ser capaz de controlar la adquisición de nueva información y su posterior representación, o ,en todo caso, ser posible extender el conocimiento con facilidad, por parte del diseñador.



Desafortunadamente no existe un esquema de representación que sea óptimo en todas estas características a la vez. Esto llevará a la necesidad de escoger la representación en función de la característica que necesitemos en el dominio de aplicación específico, o a utilizar diferentes esquemas de representación a la vez.

7.4 Tipos de conocimiento

El conocimiento que podemos representar en un dominio de aplicación lo podemos dividir en dos tipos. Por un lado esta el *conocimiento declarativo* que es un conocimiento que se representa de manera independiente a su uso, es decir, no impone un mecanismo específico de razonamiento, por lo tanto podemos decidir como usarlo dependiendo del problema que vayamos a resolver con él. Los mecanismos de razonamiento empleados en este tipo de conocimiento son de tipo general (por ejemplo resolución lineal en lógica) y la eficiencia en su uso dependerá de la incorporación de conocimiento de control que permita dirigir su utilización.

Tipos de conocimiento declarativo son el *conocimiento relacional*, que nos indica como diferentes conceptos se relacionan entre si y las propiedades que se pueden obtener a través de esas relaciones,

el *conocimiento heredable*, que establece las relaciones estructurales entre conceptos, de manera que podamos definir propiedades de generalización/especialización, inclusión o herencia de propiedades y el *conocimiento inferible*, que establece como se pueden derivar propiedades y hechos de otros mediante relaciones de deducción.

El segundo tipo de conocimiento es el *conocimiento procedimental*. Este indica explícitamente la manera de usarlo, por lo que el mecanismo de razonamiento está ligado a la representación, con la ventaja de que es más eficiente de utilizar.

7.4.1 Conocimiento Relacional simple

La forma más simple de representar hechos declarativos es mediante un conjunto de relaciones expresables mediante tablas (como en una base de datos). La definición de cada tabla establece la descripción de un concepto y la tabla contiene todas sus instanciaciones. Por ejemplo, podemos tener la colección de información sobre los clientes de una empresa

Cliente	Dirección	Vol Compras	...
A. Perez	Av. Diagonal	5643832	
J. Lopez	c/ Industria	430955	
.J. García	c/ Villaroel	12734	
..			

El principal problema es que tal cual no aporta conocimiento, solo es una colección de datos que necesita de una interpretación y procedimientos que permitan utilizarlo. Podríamos obtener nuevo conocimiento a partir de esta información con procedimientos que calcularan por ejemplo la media de compras en una población o el mejor cliente o la tipología de clientes.

Las bases de datos pueden proporcionar información en un dominio, pero es necesaria una definición explícita de las propiedades que se definen, las relaciones que se pueden establecer entre las diferentes tablas y las propiedades de esas relaciones.

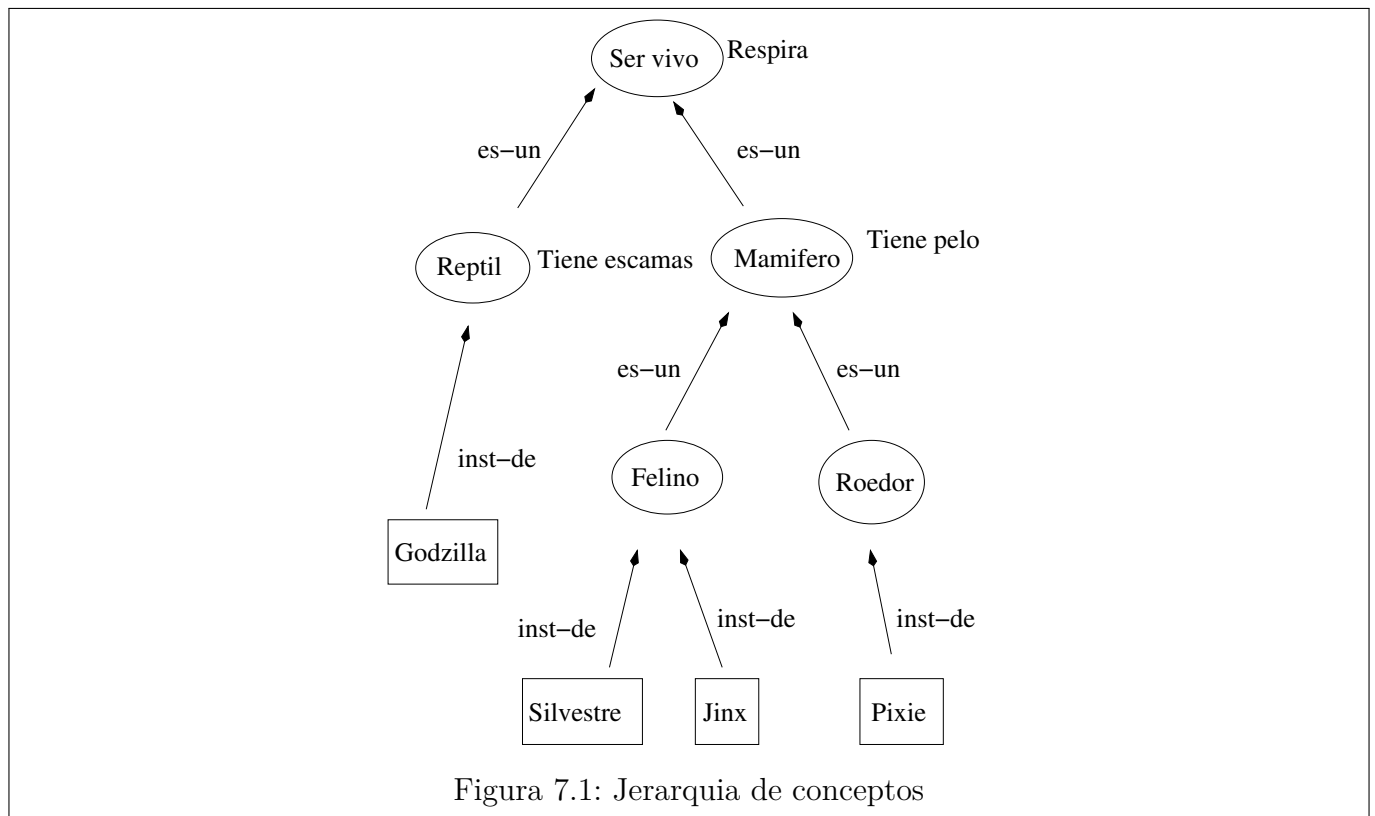
7.4.2 Conocimiento Heredable

Suele ser muy habitual que el conocimiento que debemos expresar en un dominio se pueda representar mediante una estructura jerárquica de conceptos (taxonomía jerárquica), de manera que se puedan establecer relaciones entre los diferentes conceptos. En este caso estamos representando mediante un árbol o grafo las relaciones entre conceptos que se basan en el principio de generalización y/o especialización (clase/subclase, clase/instancia, todo/parte, unión/intersección). Este conocimiento pretende representar definiciones de conceptos aprovechando las relaciones estructurales entre ellos.

En la figura 7.1 se puede ver una representación jerárquica donde los nodos son los conceptos/clases, y los arcos las relaciones jerárquicas. El mecanismo de inferencia en esta representación se basa en la herencia de propiedades y valores (herencia simple/múltiple, valores por defecto) y en la subsumción de clases (determinar si una clase está incluida en otra).

7.4.3 Conocimiento Inferible

Es conocimiento descrito mediante el lenguaje de la lógica. En este caso las expresiones describen hechos que son ciertos en el dominio. La riqueza del lenguaje de la lógica nos permite representar multitud de hechos, estableciendo por ejemplo propiedades universales sobre los predicados del dominio, la existencia de elementos que cumplan unas propiedades o relaciones de inferencia entre predicados. Podemos decir por ejemplo:



$$\forall x, y : persona(x) \wedge \neg menor(x) \wedge \neg ocupacion(x, y) \rightarrow parado(x)$$

Los métodos de inferencia en el caso de la lógica de primer orden se pueden elegir de entre cualquier método general de deducción automática que existe, como por ejemplo la resolución lineal.

7.4.4 Conocimiento Procedimental

Este conocimiento incluye la especificación de los procesos de resolución. En este tipo se incluyen los procedimientos que permiten obtener conocimiento a partir de información o conocimiento que ya se tiene. Por ejemplo, si se tiene la información Fecha_nacimiento= DD-MM-AAAA, se puede calcular la edad como una función que combine esta información con la fecha actual.

También se incluyen en este conocimiento las denominadas *reglas de producción*. Estas son expresiones condicionales que indican las condiciones en las que se deben realizar ciertas acciones del estilo **SI condición ENTONCES acción**. La combinación de estas expresiones condicionales pueden permitir resolver problemas descomponiéndolos en una cadena de acciones guiada por las condiciones de las reglas.

Este tipo de conocimiento suele ser más eficiente computacionalmente, pero hace más difícil la inferencia y la adquisición/modificación ya que se apoyan en mecanismos específicos.

8.1 Introducción

Una gran parte de los formalismos de representación del conocimiento que se utilizan en inteligencia artificial se fundamentan directamente en la lógica. De hecho la lógica es el lenguaje utilizado por todas las disciplinas científicas para describir su conocimiento de manera que pueda ser compartido sin ambigüedad y poder utilizar métodos formales para probar sus afirmaciones y obtener sus consecuencias.

Es por tanto interesante ver la lógica como lenguaje de representación. La lógica provee un lenguaje formal a través de cual podemos expresar sentencias que modelan la realidad. Es un lenguaje declarativo que establece una serie de elementos sintácticos a partir de los cuales podemos representar conceptos, propiedades, relaciones, constantes y la forma de conectar todos ellos. A partir de este lenguaje podemos relacionar la semántica de las sentencias que expresamos con la semántica de la realidad que queremos representar.

Como la lógica es un lenguaje declarativo, se establecen una serie de procedimientos que permiten ejecutar la representación y obtener nuevo conocimiento a partir de ella.

Nos centraremos en la lógica clásica y en particular en la lógica proposicional y la lógica de predicados. Los conceptos que se explican en este capítulo los conocéis de la asignatura *Introducción a la lógica*, así que podéis consultar sus apuntes para tener una descripción más detallada. Este capítulo solo servirá para refrescar la memoria y no como descripción exhaustiva del formalismo de la lógica.

8.2 Lógica proposicional

Es el lenguaje lógico más simple y nos permite definir relaciones entre frases declarativas atómicas (no se pueden descomponer en elementos más simples). El lenguaje se define a partir de átomos y conectivas, siendo estas la conjunción ($\wedge$), la disyunción ($\vee$), la negación ($\neg$) y el condicional ($\rightarrow$).

Las fórmulas válidas de lógica de enunciados son las que se pueden derivar a partir de esta gramática:

1. Si P es una sentencia atómica entonces P es una fórmula.
2. Si A y B son fórmulas entonces $(A \wedge B)$ es una fórmula
3. Si A y B son fórmulas entonces $(A \vee B)$ es una fórmula

4. Si A y B son fórmulas entonces $(A \rightarrow B)$ es una fórmula
5. Si A es una fórmula entonces $\neg A$ es una fórmula

Por convención, se consideran sentencias atómicas todas las letras mayúsculas a partir de la P y fórmulas todas las letras mayúsculas a partir de la A .

Mediante este lenguaje podemos expresar sentencias que relacionan cualquier frase declarativa atómica mediante las cuatro conectivas básicas y su semántica asociada. Esta semántica viene definida por lo que se denomina *teoría de modelos*, que establece la relación entre las fórmulas y los valores de verdad y falsedad, es lo que se denomina una interpretación. Cada conectiva tiene su tabla de verdad¹ que establece como se combinan los valores de verdad de los átomos para obtener el valor de verdad de la fórmula.

A partir de los enunciados expresados mediante este lenguaje se pueden obtener nuevos enunciados que se deduzcan de ellos o se pueden plantear enunciados y comprobar si se derivan de ellos. Para ello existen diferentes metodologías de validación de razonamientos que establecen los pasos para resolver esas preguntas.

Metodologías de validación de razonamientos hay bastantes y se pueden dividir en dos grupos. Las que se basan en la semántica, buscan demostrar que los modelos que hacen verdad un conjunto de enunciados incluyen los modelos que hacen verdad la conclusión que queremos probar. Las que se basan en sintaxis aprovechan la estructura algebraica del lenguaje de la lógica de enunciados (álgebra de boole) para demostrar que el enunciado satisfactible/insatisfactible es accesible utilizando los enunciados premisa y la conclusión.

El método más sencillo para la validación de enunciados es el *método de resolución*. Es un método que funciona por refutación, es decir, podemos saber si un enunciado se deriva de un conjunto de enunciado probando que al añadir su negación podemos obtener el enunciado insatisfactible.

Este método se basa en la aplicación de la regla de resolución:

$$A \vee B, \neg A \vee C \vdash B \vee C$$

al conjunto de cláusulas que se obtienen de transformar las premisas y la negación de la conclusión a forma normal conjuntiva. Existen diferentes variantes del algoritmo, la mas común es la que utiliza la denominada estrategia de conjunto de soporte, que inicia el algoritmo escogiendo alguna de las cláusulas de la negación de la conclusión. Este método es sólido y completo² si se demuestra la satisfactibilidad de las cláusulas de las premisas.

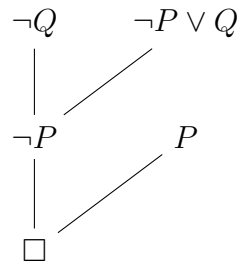
Ejemplo 8.1 Podemos plantearnos un problema de lógica proposicional donde tengamos los enunciados “Está lloviendo” y “Me mojo”. Asignamos a los enunciados las letras de átomo P y Q respectivamente. Podemos escribir la fórmula $P \rightarrow Q$, que se puede leer como “Si esta lloviendo entonces me mojo” y plantearnos el siguiente razonamiento:

$$P \rightarrow Q, P \vdash Q$$

Podemos utilizar el método de resolución para validar este razonamiento obteniendo el siguiente conjunto de cláusulas con el razonamiento $\mathcal{C} = \{\neg P \vee Q, P, \neg Q\}$. Con estas cláusulas podemos obtener el siguiente árbol de resolución validando el razonamiento:

¹Es algo que ya conocéis de la asignatura de lógica.

²Las propiedades de solidez y completitud (*soundness, completeness*) son las que se piden a cualquier método de validación. La propiedad de solidez nos garantiza que el método solo nos dará conclusiones válidas y la completitud nos garantiza que podremos obtener todas las conclusiones derivables.



8.3 Lógica de predicados

La capacidad expresiva de la lógica de enunciados no es demasiado grande, por lo que habitualmente se utiliza como método de representación la lógica de predicados. Esta amplía el vocabulario de la lógica de enunciados añadiendo parámetros a los átomos, que pasan a representar propiedades o relaciones, también añade lo que denominaremos términos que son de tres tipos: variables (elementos sobre los que podremos cuantificar o substituir por valores), constantes (elementos identificables del dominio) y funciones (expresiones que pueden aplicarse a variables y constantes que denotan el valor resultante de aplicar la función a sus parámetros). A estos elementos se les añaden los cuantificadores universal ($\forall$) y existencial ($\exists$) que son extensiones infinitas de las conectivas conjunción y disyunción.

Esto amplía la gramática de fórmulas que podemos expresar, será una fórmula válida de lógica de predicados toda aquella que cumpla:

1. Si P es un predicado atómico y $t_1, t_2, \dots, t_n$ son términos, entonces $P(t_1, t_2, \dots, t_n)$ es una fórmula.
2. Si A y B son fórmulas entonces $(A \wedge B)$ es una fórmula
3. Si A y B son fórmulas entonces $(A \vee B)$ es una fórmula
4. Si A y B son fórmulas entonces $(A \rightarrow B)$ es una fórmula
5. Si A es una fórmula y v es una variable, entonces $(\forall v A)$ y $(\exists v A)$ son una fórmula
6. Si A es una fórmula entonces $\neg A$ es una fórmula

Por convención, se consideran sentencias atómicas todas las letras mayúsculas a partir de la P y fórmulas todas las letras mayúsculas a partir de la A . Para los términos, las constantes se denotan por letras minúsculas a partir de la a , las variables son letras minúsculas a partir de la x y las funciones letras minúsculas a partir de la f .

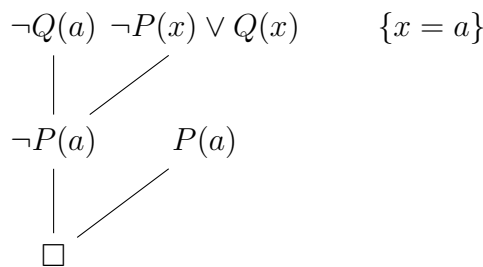
Mediante este lenguaje podemos expresar propiedades y relaciones entre objetos constantes de un dominio o establecer propiedades universales o existenciales entre los elementos del dominio. La visión que tenemos de la realidad está compuesta de elementos (las constantes) que podemos ligar mediante propiedades y relaciones. Podríamos asociar esta visión a una noción conjuntista de la realidad, los predicados unarios corresponderían a los conjuntos en los que podemos tener las constantes, los predicados con mayor aridad corresponderían a las relaciones que podemos establecer entre los elementos de esos conjuntos. En este caso la cuantificación nos permite establecer enunciados que pueden cumplir todos los elementos de un conjunto o enunciar la existencia en el conjunto de elementos que cumplen cierta propiedad o relación.

El método de resolución también se puede aplicar como método de validación en lógica de predicados, pero necesita de un conjunto de procedimientos adicionales para tener en cuenta las extensiones que hemos hecho en el lenguaje. En este caso las fórmulas son transformadas a la *forma normal de skolem*, que permite substituir todas las cuantificaciones existenciales por constantes y expresiones

funcionales³. El método exige que para la aplicación de la regla de resolución los átomos que se quieran resolver permitan la unificación de los términos de los parámetros. La unificación lo que busca es la substitución de los valores de las variables de los términos de los átomos que permitan que las dos expresiones sean iguales.

Ejemplo 8.2 Podemos plantearnos el problema de lógica de predicados con los siguientes predicados “ser un hombre”, “ser mortal” y una constante “Sócrates”. Asignamos a los predicados las letras P y Q respectivamente y a la constante la letra a . Podemos escribir la fórmula $\forall x(P(x) \rightarrow Q(x))$ (todos los hombres son mortales) y la fórmula $P(a)$ (Sócrates es un hombre) y podemos intentar validar la fórmula $Q(a)$ (Sócrates es mortal).

Podemos utilizar el método de resolución para validar este razonamiento obteniendo el siguiente conjunto de cláusulas con el razonamiento $\mathcal{C} = \{\neg P(x) \vee Q(x), P(a), \neg Q(a)\}$. Con estas cláusulas podemos obtener el siguiente árbol de resolución validando el razonamiento:



8.4 Lógica y representación del conocimiento

El lenguaje de lógica de predicados puede ser bastante incómodo para expresar cierto tipo de conocimiento y existen extensiones o modificaciones que intentan facilitar su uso. Por ejemplo, extensiones que hacen algo más cómodo el uso de lógica de predicados es incluir la igualdad como un predicado especial o permitir el utilizar tipos en las variables que se cuantifican.

Existen otros aspectos que son difíciles de tratar directamente en lógica de predicados, como por ejemplo:

- Representación del tiempo: En lógica de predicados toda fórmula se refiere al presente. Existen diferentes lógicas para el tratamiento del tiempo que permiten expresar relaciones temporales entre los elementos del dominio.
- El conocimiento entre agentes, razonamiento sobre otros: Las fórmulas que expresamos son el conocimiento propio, pero no tratamos nuestra creencia sobre el conocimiento que puedan tener otros. Las diferentes lógicas de creencias intentan establecer métodos que permiten obtener deducciones sobre lo que creemos que otros agentes creen.
- El conocimiento incierto e impreciso: La lógica de predicados supone que podemos evaluar con total certidumbre el valor de verdad de cualquier fórmula, pero en la realidad eso no es siempre así. Las lógicas probabilística y posibilística intentan trabajar con esa circunstancia y permitir obtener deducciones a partir de la certidumbre que tenemos sobre nuestro conocimiento.

³Tenéis los detalles en los apuntes de la asignatura de lógica.

Sistemas de reglas de producción

9.1 Introducción

La lógica de predicados también se puede ver como un mecanismo para describir procedimientos. Con este tipo de visión lo que hacemos es describir un problema como si fuera un proceso de razonamiento. Los predicados que utilizamos descomponen el problema en problemas más sencillos e indican un orden total o parcial que permitirá al sistema que ejecute esta descripción resolverlo.

En este tipo de descripción utilizamos un conjunto restringido de las expresiones que se pueden representar mediante el lenguaje de la lógica. La restricción fundamental que se impone es que solo se pueden utilizar fórmulas lógicas cuantificadas universalmente que se puedan transformar a cláusulas disyuntivas¹, en las que solo haya como máximo un átomo afirmado. Este tipo de cláusulas se denominan **cláusulas de Horn**.

La justificación de esta limitación es que la representación tendrá que ser ejecutada como una demostración y el coste computacional de los procedimientos para realizarla está ligado a la expresividad empleada en su descripción. A pesar de esta restricción del lenguaje, podremos describir un gran número de problemas.

Este tipo de fórmulas se corresponde con lo que denominaremos **reglas de producción**. Las reglas son fórmulas condicionales en las que puede haber cualquier número de átomos en el antecedente y solo un átomo en el consecuente, por ejemplo:

$$\forall x \forall y (Persona(x) \wedge Edad(x, y) \wedge Mayor(y, 18) \rightarrow Mayor_de_edad(x))$$

En la práctica los lenguajes de programación que permiten describir problemas mediante reglas son más flexibles en su sintaxis y permitir cosas más complejas. De hecho el consecuente de una regla puede ser una acción sobre la descripción del estado (modificación de los predicados actuales), un nuevo hecho del estado, un hecho que usamos como elemento de control de la búsqueda, una interacción con el usuario, ...

9.2 Elementos de un sistema de producción

Mediante este formalismo podremos ser capaces de describir como solucionar un problema, declarando el conjunto de reglas a partir del cual se puede obtener una solución mediante un proceso de demostración. La diferencia de las reglas respecto a otros formalismos procedimentales (como un

¹Una cláusula disyuntiva es una fórmula en la que solo se usa la conectiva disyunción.

programa implementado en un lenguaje imperativo) es que no explicitamos directamente el algoritmo a utilizar para encontrar la solución del problema, sino que se indican las condiciones que ésta ha de cumplir. Es un mecanismo de demostración el que se encarga de encontrarla combinando los pasos indicados por las reglas, ya que lo que estamos haciendo es validar un razonamiento que se fundamenta en éstas.

Esta forma de describir procedimientos se encuadra dentro de los denominados *lenguajes declarativos* (a diferencia de los lenguajes imperativos, que son los más comunes). La ventaja de declarar las condiciones que cumplen las soluciones es que no estamos limitados a una única secuencia de ejecución, un conjunto de reglas pueden describir muchos algoritmos distintos que comparten parte de su especificación. Este tipo de lenguajes permiten, además, expresar programas a un mayor nivel de abstracción, ya que no tenemos que definir exactamente como se debe conseguir la solución, solo sus condiciones. Hallar la solución que cumple las condiciones es labor del mecanismo de demostración que utilicemos.

Para poner este formalismo en perspectiva, podemos hacer la analogía con los algoritmos de búsqueda que hemos visto en los primeros capítulos. Las reglas son equivalentes a los operadores de búsqueda, y el algoritmo de búsqueda utilizado es el mecanismo de validación de demostraciones. Un número relativamente reducido de operadores puede permitir hallar soluciones a problemas muy diferentes y todos ellos se generan a partir de la combinación de éstos.

Al conjunto de reglas se le denomina la **base de conocimiento** (o de reglas). Estas reglas pueden describir la resolución de múltiples problemas, será la labor del mecanismo que resuelva el problema concreto planteado decidir qué reglas se han de utilizar.

El planteamiento del problema se realiza mediante **hechos**. Estos hechos serán la descripción de las condiciones del problema y estará formada por predicados primitivos, funciones, relaciones y constantes definidas en el problema. Por ejemplo:

```
Persona(juan)
Edad(juan,25)
```

Al conjunto de hechos que describen un problema se denomina **base de hechos**. Con una base de reglas y una base de hechos podemos plantear preguntas cuya respuesta permita obtener la solución de un problema.

Ejemplo 9.1 *Podemos crear un conjunto de reglas que nos permitan saber cuando se puede servir una bebida a una persona en un bar. Podríamos describir el problema de la siguiente manera:*

“Toda persona mayor de 18 años es mayor de edad. Toda bebida que contenga alcohol es una bebida restringida a mayores de edad. Se puede servir a una persona una bebida restringida a mayores de edad si es mayor de edad.”

Y lo podríamos describir mediante reglas de como la siguiente base de reglas:

```
si Persona(X) y Edad(X,Y) y Mayor(Y,18) entonces Mayor_de_edad(X)
si Bebida(X) y Contiene(X,alcohol) entonces Restringida_a_mayores(X)
si Mayor_de_edad(X) y Restringida_a_mayores(Y) entonces Servir(X,Y)
```

Podríamos entonces plantear un problema con un conjunto de hechos:

```
Persona(juan)
Edad(juan,25)
Bebida(cubata)
Contiene(cubata,alcohol)
```

*Y entonces preguntarnos si le podemos servir un cubata a juan **Servir(juan,cubata)**.*

La forma en la que determinamos si existe y cual es la respuesta al problema planteado dependerá del método de demostración de razonamientos que utilicemos para validar el razonamiento que describen la base de conocimiento, la base de hechos y la pregunta que realizamos.

Este mecanismo estará implementado mediante lo que denominaremos el **motor de inferencias**.

9.3 El motor de inferencias

La labor de un motor de inferencias es ejecutar la representación del problema descrita mediante la base de reglas y la base de hechos. Deberá ser por tanto capaz de demostrar razonamientos buscando la manera más eficiente de hacerlo. Para ello el motor de inferencias constará de dos elementos:

- **El intérprete de reglas:** Deberá ser capaz de ejecutar las reglas. Esto incluye interpretar el antecedente de la regla, buscar las instanciaciones adecuadas para que el antecedente se cumpla con los hechos/objetivos conocidos y generar la deducción que representa el consecuente de la regla. Una de las labores mas importantes de este componente será generar lo que se conoce como el **conjunto de conflicto**. Éste contiene todas las reglas u objetivos que se pueden utilizar en un momento dado de la resolución de un problema.

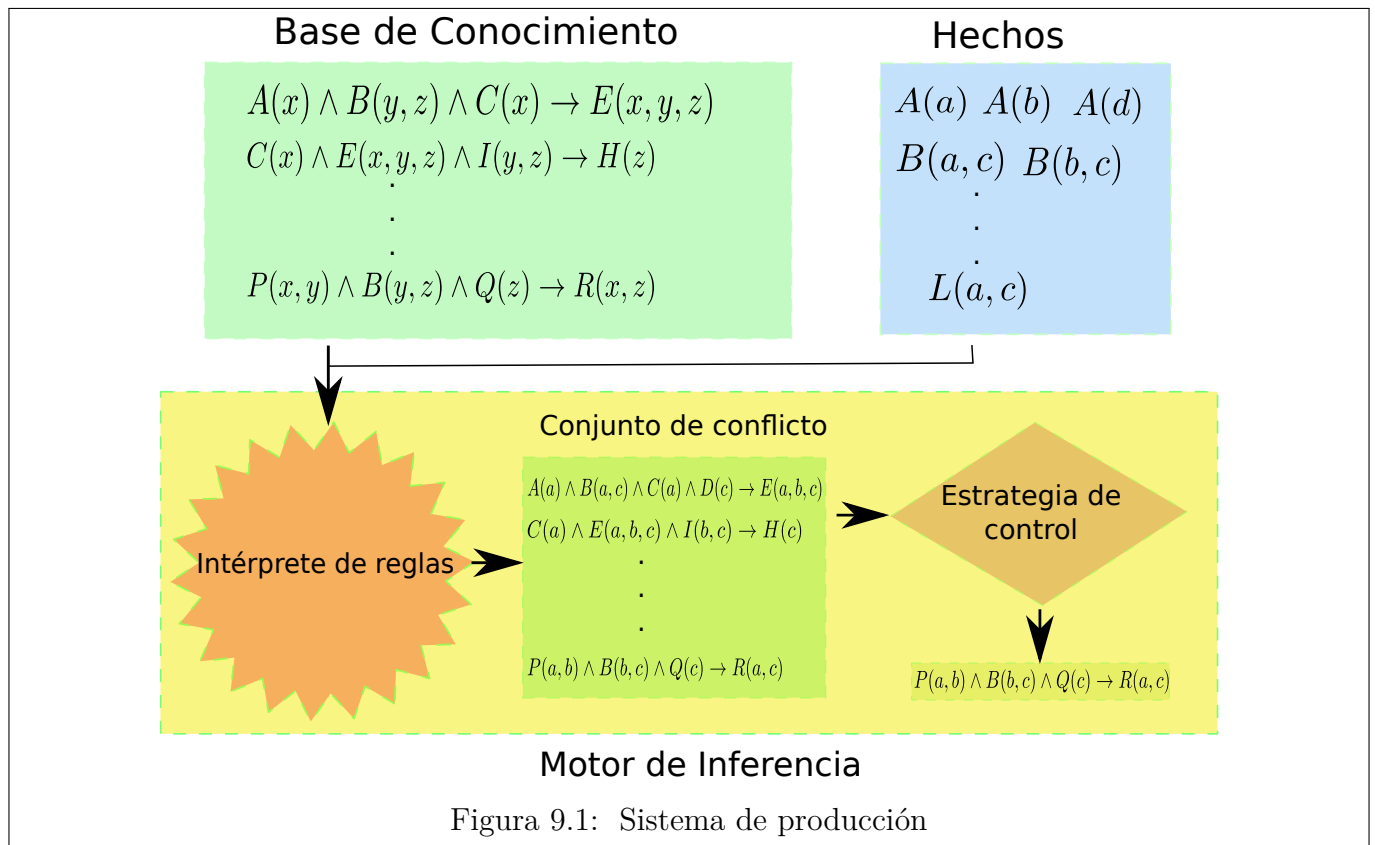
En los lenguajes de reglas, el intérprete de reglas puede ser bastante complejo, permitiendo lo que se podría encontrar en cualquier lenguaje de programación, como por ejemplo la interacción con el usuario. También podría permitir el uso de otras lógicas mas allá de la lógica de predicados como lógicas para información incompleta o para información con incertidumbre.

- **La estrategia de control:** Será el mecanismo que permitirá al motor de inferencia elegir la regla u objetivo que deberá resolver en primer lugar y se basará en lo que denominaremos **estrategia de resolución de conflictos**. Dependiendo de la eficacia de esta estrategia, el coste final en tiempo de la resolución de un problema puede variar enormemente.

En la figura 9.1 se puede ver una representación de los elementos de un sistema de producción.

Respecto a la estrategia de resolución de conflictos, esta puede ser de muy diversas naturalezas, pudiendo utilizar información sintáctica local de las reglas o estrategias mas complejas. Por ejemplo se podría utilizar como criterio de elección:

- La primera regla según el orden establecido en la base de conocimiento. El experto conoce en que orden se deben ejecutar las reglas en caso de conflicto y lo indica en la base de reglas.
- La regla mas/menos utilizada. La regla más utilizada puede ser la correcta en la mayoría de los casos y eso le daría preferencia. También se podría dar preferencia al criterio contrario si por ejemplo las reglas menos usadas tratan excepciones y precisamente estamos resolviendo un caso excepcional.
- La regla más específica/más general. Las reglas más generales suelen utilizarse al comienzo de la resolución para plantear el problema y dirigir la búsqueda. Las reglas más específicas suelen ser útiles una vez ya hemos encaminado la resolución hacia un problema concreto.
- La regla con la instanciación de hechos mas prioritarios. Podemos indicar que hechos son mas importantes en la resolución y utilizar primero las reglas que los instancien.
- La regla con la instanciación de hechos mas antiguos/mas nuevos. Es lógico pensar que los últimos hechos deducidos deberían ser los primeros en utilizarse, pero también es posible que los hechos mas nuevos sean erróneos y que la manera mejor para redirigir la búsqueda es utilizar hechos antiguos de una manera diferente.



- Ordenadamente en anchura/en profundidad
- Aleatoriamente

Ninguno de estos criterios (en ocasiones contradictorios) garantiza por si solo hallar la solución siguiendo el camino más eficiente, por lo que por lo general se suele utilizar una combinación de criterios. Habitualmente se utiliza metaconocimiento en forma de metareglas (reglas sobre el uso de las reglas) para hacer mas eficiente la resolución y para cambiar la estrategia de resolución de conflictos dinámicamente.

9.3.1 Ciclo de ejecución

El motor de inferencias ejecuta un conjunto de fases que definen su funcionamiento. Estas fases son las siguientes:

1. **Fase de detección:** En esta fase el intérprete de reglas determinará todas las instanciaciones posibles entre los hechos u objetivos del problema y las reglas de la base de reglas. El objetivo es calcular el **conjunto de conflicto**, las reglas que se podrían ejecutar en ese momento. Esta será la fase más costosa del ciclo de ejecución ya que el número de posibles instanciaciones de las reglas puede ser elevado.
2. **Fase de selección:** Se elegirá la regla a utilizar en este paso de la resolución, utilizando como criterio la *estrategia de resolución de conflictos*. La elección que se realice determinará la forma en la que se haga la exploración de las posibles soluciones del problema.
3. **Fase de aplicación:** El intérprete de reglas propagará las instanciaciones de las condiciones de la regla a la conclusión, ejecutándola y cambiando el estado del problema. Esto puede significar la obtención de nuevos hechos o el planteamiento de nuevos objetivos.

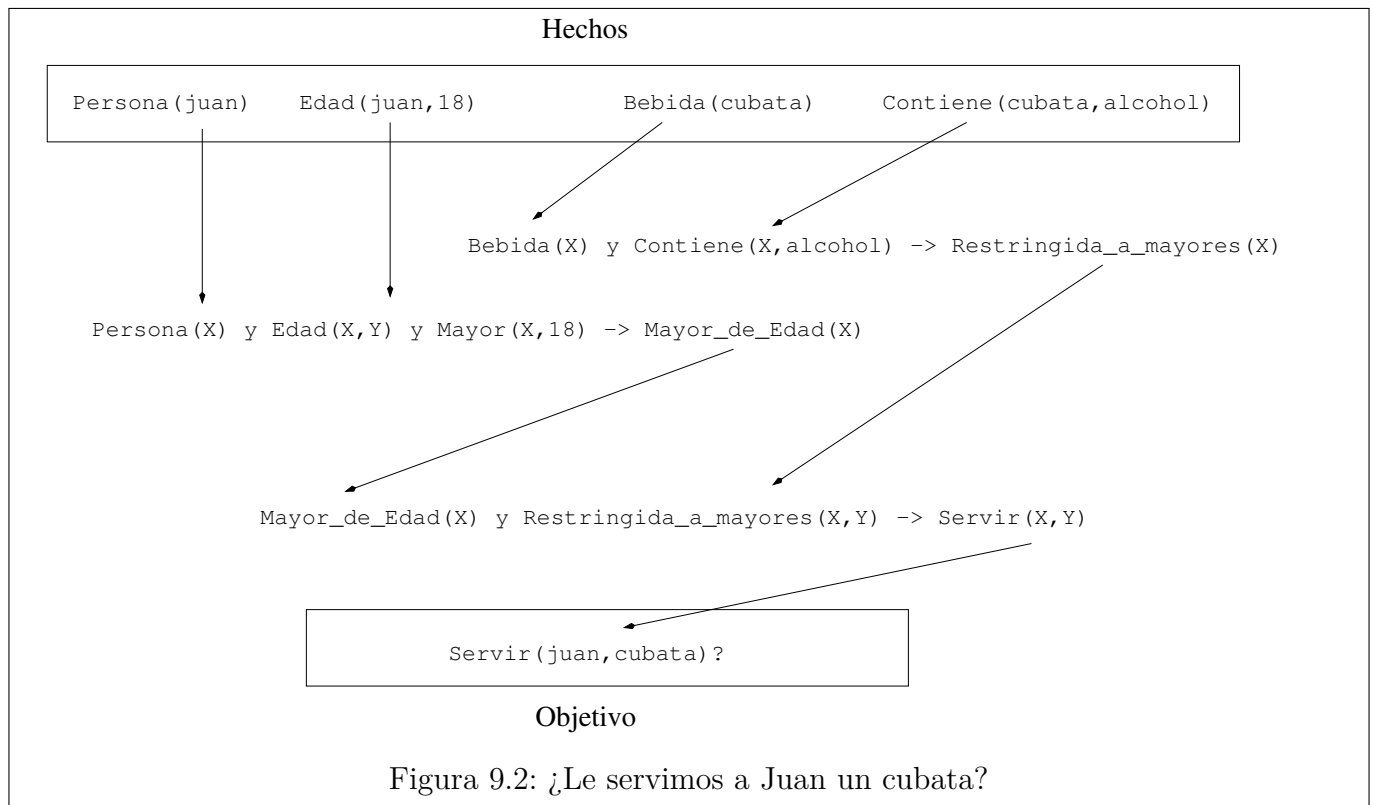


Figura 9.2: ¿Le servimos a Juan un cubata?

El ciclo de ejecución del motor de inferencias se acabará en el momento en el que se haya obtenido el objetivo deseado, ya no haya más reglas que aplicar o se hayan cubierto todos los objetivos que se hayan planteado durante la resolución del problema.

9.4 Tipos de razonamiento

La forma de resolver el razonamiento que plantea un conjunto de reglas y hechos depende del punto de vista que adoptemos. Si vemos el problema desde un punto de vista global, lo que se hace al validar el razonamiento es ligar los hechos que hay en la base de hechos con el objetivo que tenemos que probar mediante las reglas de las que disponemos.

La forma en la que ligamos hechos y objetivos es simplemente encadenando las reglas una detrás de otra adecuadamente, de manera que las instanciaciones de las condiciones del antecedente y conclusiones del consecuente cuadren. Si pensamos que estamos hablando de un método de representación procedimental, de hecho lo que estamos haciendo es ensamblar el algoritmo que resuelve el problema.

Podemos ver en la figura 9.2 el planteamiento del ejemplo que vimos anteriormente. Ligamos los hechos de la base de hechos con el objetivo planteado instanciando las reglas que tenemos. En principio no hay una dirección preferente en la que debemos ligar hechos y objetivos. El problema lo podemos resolver partiendo de los hechos y deduciendo nuevos hechos a partir de las reglas hasta llegar al objetivo, este es el sentido en el que estamos habituados a utilizar las reglas. Pero también lo podríamos resolver mirando las reglas en el sentido contrario, una regla descompondría un problema (el consecuente) en problemas más pequeños (antecedente) y podríamos iniciar la búsqueda de la solución en el objetivo final hasta descomponerlo en los hechos de la base de hechos.

Estas dos estrategias se conocen como **razonamiento guiado por los datos** o **razonamiento hacia adelante** (*forward chaining*) y **razonamiento guiado por los objetivos** o **razonamiento hacia atrás** (*backward chaining*). Los explicaremos con detalle en las siguientes secciones. Como

Algoritmo 9.1 Razonamiento hacia adelante**Procedimiento:** Razonamiento Hacia Adelante**Entrada:** Base de hechos, Base de reglas, ObjetivosAlternativas $\leftarrow$ cierto**mientras** $\exists o(o \in \text{Objetivos} \wedge o \notin \text{Base_de_hechos}) \wedge \text{Alternativas}$ **hacer** Conjunto_Conflicto $\leftarrow$ Interprete.Antecedentes_satisfactibles(Base_de_hechos, Base_de_reglas) **si** Conjunto_Conflicto $\neq \emptyset$ **entonces** Regla $\leftarrow$ Estrategia_Control.Resolucion_Conflictos(Conjunto_Conflicto)

Interprete.Aplicar(Base_de_hechos, Regla)

sino Alternativas $\leftarrow$ falso

complemento a estas estrategias existen métodos de razonamiento que utilizan una combinación de las dos (*estrategias híbridas*) para intentar obtener las ventajas de cada una y reducir sus inconvenientes.

9.4.1 Razonamiento guiado por los datos

Este tipo de razonamiento supone que son los hechos los que dirigen el ciclo de ejecución del motor de inferencias. Por lo tanto, cada vez que hacemos la fase de selección, buscamos todas aquellas reglas cuyas condiciones podamos hacer válidas con cualquier posible combinación de los hechos que tenemos en ese momento. Tras la selección de la regla más adecuada se añade a la base de hechos el consecuente de la regla. La ejecución acaba cuando el objetivo es deducido.

Este tipo de razonamiento se denomina deductivo o progresivo y se fundamenta en la aplicación de la regla denominada de la eliminación del condicional o *modus ponens*. Esta regla se puede formalizar de la siguiente manera:

$$A, A \rightarrow B \vdash B$$

Parafraseando, esta regla nos dice que cuando tenemos un conjunto de hechos y estos hechos forman parte del antecedente de una expresión condicional, podemos deducir el consecuente como un hecho mas de nuestro razonamiento. El algoritmo 9.1 es una implementación de su funcionamiento.

El mayor inconveniente de esta estrategia de razonamiento es que el razonamiento no se guía por el objetivo a conseguir, sino que se obtienen todas las deducciones posibles ya sean relevantes o no para conseguirlo. Esto implica un gran coste computacional a la hora de resolver un problema ya que se hace mas trabajo del necesario para obtener el objetivo. Esto significa que la estrategia de resolución de conflictos es bastante importante a la hora de dirigir la exploración del problema. De hecho, para hacer eficiente este método es necesario metaconocimiento y la capacidad para dirigir el flujo de razonamiento mediante las propias reglas.

Otro problema computacional es el detectar las reglas que se pueden disparar en un momento dado, pues requiere computar todas la coincidencias posibles entre hechos y antecedentes de reglas. Este problema se puede resolver de manera eficiente mediante el denominado *algoritmo de Rete*. Este algoritmo permite mantener y actualizar todas las coincidencias entre hechos y antecedentes, de manera que se puede saber eficientemente si una regla se cumple o no.

Las ventajas de este tipo de razonamiento, vienen del hecho de que, es más natural en muchos dominios expresar el razonamiento de esta manera y, por lo tanto, es más fácil plantear reglas que resuelvan problemas mediante esta estrategia.

Algoritmo 9.2 Razonamiento hacia atrás

Procedimiento: Razonamiento Hacia Atrás**Entrada:** Base de hechos, Base de reglas, ObjetivosAlternativas $\leftarrow$ cierto**mientras** *Objetivos* $\neq \emptyset \wedge$ *Alternativas* **hacer** Objetivo $\leftarrow$ Estrategia_Control.Escoger_Objetivo(Objetivos)

Objetivos.Quitar(Objetivo)

 Conjunto_Conflicto $\leftarrow$ Interprete.Consecuentes_satisfactibles(Objetivo, Base_de_reglas) **si** *Conjunto_Conflicto* $\neq \emptyset$ **entonces** Regla $\leftarrow$ Estrategia_Control.Resolucion_Conflictos(Conjunto_Conflicto)

Objetivos.Añadir(Regla.Extraer_antecedente_como_objetivos())

sino Alternativas $\leftarrow$ falso

Este tipo de razonamiento también es adecuado en problemas en los que no exista un objetivo claro a alcanzar y lo que se busque sea explorar el espacio de posibles soluciones a un problema.

9.4.2 Razonamiento guiado por los objetivos

Este tipo de razonamiento supone que es el objetivo el que dirige el ciclo del motor de inferencias. En este caso lo que hacemos es mantener una lista de objetivos pendientes de demostrar que se inicializa con el objetivo del problema. En la fase de selección miramos si entre los hechos está el objetivo actual, si es así lo eliminamos y pasamos al objetivo siguiente. Si no es así, se seleccionan todas aquellas reglas que permitan obtener el objetivo como conclusión. Una vez elegida la regla más adecuada, los predicados del antecedente de la regla pasan a añadirse a la lista de objetivos a cubrir. La ejecución acaba cuando todos los objetivos que han aparecido acaban siendo cubiertos mediante hechos del problema.

Este tipo de razonamiento se denomina inductivo o regresivo e invierte la utilización que hace el razonamiento hacia adelante de la regla del modus ponens. Esta visión puede tomarse como una forma de descomposición de problemas, en la que se parte del objetivo inicial y se va reduciendo en problemas cada vez más simples hasta llegar a problemas primitivos (los hechos). El algoritmo 9.2 es una implementación de su funcionamiento.

La ventaja principal de este método de razonamiento es el que al estar guiado por el objetivo, todo el trabajo que se realiza se encamina a la resolución del problema y no se hace trabajo extra. Esto hace que este tipo de razonamiento sea mas eficiente.

Como desventaja, nos encontramos con que plantear un conjunto de reglas que resuelva problema de esta manera es menos intuitivo. El problema debe pensarse para poder resolverse mediante una descomposición jerárquica de subproblemas. También tenemos el inconveniente de que solo se puede aplicar a problemas en los que el objetivo sea conocido, lo cual reduce los dominios en los que puede ser aplicado.

9.5 Las reglas como lenguaje de programación

El usar reglas como mecanismo de resolución de problemas ha llevado a verlas como el fundamento de diferentes lenguajes de programación generales. Estos lenguajes entran dentro de los paradigmas de

programación declarativa y programación lógica. En estos lenguajes los programas solamente definen las condiciones que debe cumplir una solución mediante un formalismo lógico y es el mecanismo de demostración el que se encarga de buscarla explorando las posibilidades.

La mayoría de estos lenguajes no son lenguajes declarativos puros e incluyen mecanismos y estructuras que pueden encontrarse en otros paradigmas, pero aún así la filosofía de programación es bastante diferente y requiere un cambio en la manera de pensar.

En este tipo de lenguajes la asignación no existe y la comunicación entre reglas se realiza mediante la unificación de las variables que se utilizan. Estas variables no se pueden ver como las que se usan en programación imperativa ya que su valor lo toman durante el proceso de prueba de deducción al comparar hechos y condiciones y realizar su unificación. Por lo tanto no son lugares donde nosotros vayamos poniendo valores, los toman cuando es adecuado para demostrar el problema. En estos lenguajes la recursividad tiene un papel fundamental a la hora de programar.

Todas estas características pueden parecer una limitación expresiva, pero en realidad no lo son y la potencia expresiva de estos lenguajes es la misma que la de los lenguajes imperativos. Existen dominios de aplicación en los que estos estilos de programación son más adecuados que los lenguajes imperativos, la inteligencia artificial es uno de ellos.

Ejemplo 9.2 *El cálculo del factorial es un ejemplo clásico, podemos definir cuales son las condiciones que debe cumplir un número para ser el factorial de otro. Sabemos que hay un caso trivial que es el factorial de 1. Podemos declarar el predicado que asocie a 1 con su factorial **fact(1,1)**.*

Para calcular que un número es el factorial de otro lo único que tenemos que hacer es especificar las condiciones para que han de cumplirse. Por ejemplo:

***si** $= (X, X1+1)$ **y** **fact**($X1, Y1$) **y** $= (Y, Y1*X)$ **entonces** **fact**(X, Y)*

*Parafraseando podríamos leer esta regla como que si hay un X que sea $X1 + 1$ y el factorial de $X1$ es $Y1$ y hay un Y que sea $Y1*X$ entonces el factorial del X será Y .*

Supondremos que el predicado $=$ es un predicado especial que unifica el primer parámetro con el segundo.

*Podemos plantearnos la pregunta de cual es el factorial de 3, o sea si existe un valor Z asociado al predicado **fact(3,Z)**. Mediante el mecanismo de razonamiento hacia atrás podremos averiguar cual es el valor de Z .*

Renombraremos las variables cada vez que usemos la regla para evitar confusiones.

***fact**(3,Z)*

según la regla se puede descomponer en tres subobjetivos

*$= (3, X1+1), \text{fact}(X1, Y1), = (Z, Y1*3)$*

Evidentemente $X1$ se unificará con 2 en la primera condición quedándonos

***fact**(2, $Y1$), $= (Z, Y1*3)$*

Volveremos a descomponer el problema según la regla

*$= (2, X1'+1), \text{fact}(X1', Y1'), = (Y1, Y1'*2), = (Z, Y1*3)$*

Ahora $X1'$ se unificará con 1

***fact**(1, $Y1'$), $= (Y', Y1'*2), = (Y, Y1*3)$*

*Sabemos por los hechos que **fact**(1,1)*

*$= (Y1, 1*2), = (Z, Y1*3)$*

$Y1$ se unificará con 2

*$= (Z, 2*3)$*

Z se unificará con 6 obteniendo la respuesta $Z=6$

*Esto puede parecer la manera habitual con la que calculamos el factorial en un lenguaje imperativo, pero el paradigma declarativo tiene mayor versatilidad, porque sin cambiar el programa podríamos preguntar por ejemplo cual es el número que tiene como factorial 6 (**fact**($X, 6$)) o incluso que nos calcule todos los factoriales que existen (**fact**(X, Y)). El mecanismo de deducción nos hallará la*

respuesta a todas estas preguntas. En un lenguaje imperativo deberíamos escribir un programa para responder a cada una de ellas.

Utilizando un mecanismo de razonamiento hacia adelante también podríamos obtener el mismo cálculo, en este caso ni siquiera necesitamos de un objetivo, si ponemos en marcha el motor de inferencias obtendremos tantos factoriales como queramos:

$=(X, X1+1), \text{ fact}(X1, Y1), =(Y, Y1*X)$

Instanciando X1 a 1 e Y1 a 1 con el hecho fact(1,1) obtendremos el nuevo hecho fact(2,2)

$=(X, X1+1), \text{ fact}(X1, Y1), =(Y, Y1*X)$

Instanciando X1 a 2 e Y1 a 2 con el hecho fact(2,2) obtendremos el nuevo hecho fact(3,6)

...

El lenguaje de programación lógica por excelencia es **PROLOG**. Es un lenguaje en el que el mecanismo de razonamiento que se sigue es hacia atrás. Esto hace que cuando se programa en este lenguaje se deba pensar que se ha de descomponer el problema mediante el formalismo de reglas en problemas mas simples que estarán declarados como hechos.

Existen otros lenguajes de reglas que utilizan motores de inferencia hacia adelante como por ejemplo **CLIPS**. En este caso la filosofía de programación es diferente, ya que estamos explorando para encontrar la solución. Las reglas se ven como elementos reactivos que se disparan en función del estado del problema y que van construyendo los pasos que llevan a la solución. Esto obliga a que, si se quiere seguir un camino específico de resolución, haya que forzarlo añadiendo hechos que lleven el control de la ejecución.

9.6 Las reglas como parte de aplicaciones generales

Los sistemas de producción y los motores de inferencia están adquiriendo en la actualidad bastante relevancia como método adicional para desarrollar aplicaciones fuera del ámbito de la inteligencia artificial.

En muchas ocasiones las decisiones que se toman dentro de una aplicación pueden variar con el tiempo (a veces bastante rápido) y no tiene mucho sentido programarlas dentro del código del resto de la aplicación, ya que eso implicaría tener que cambiarlo cada vez que se cambia la forma de tomar esas decisiones.

En la terminología de este tipo de aplicaciones, las reglas implementarían lo que se denominan **las reglas de negocio**, que no son más que algoritmos de decisión invocados a partir de un conjunto de datos, algo que concuerda de manera natural con el funcionamiento de los sistemas de producción.

Las reglas son útiles en este tipo de escenarios, ya que pueden mantenerse aparte del resto de procedimientos de la aplicación y ser ejecutadas mediante un motor de inferencia. El cambio de un mecanismo de decisión solo implicará cambiar las reglas adecuadamente, manteniendo intacta el resto de la aplicación.

Este tipo de soluciones están tomando cada vez mas relevancia sobre todo con la tendencia a hacer aplicaciones distribuidas y basadas en componentes, como por ejemplo las que entran dentro del ámbito de las arquitectura orientadas a servicio (Service Oriented Architectures, SOA). Estas últimas están bastante ligadas a las tecnologías basadas en agentes del ámbito de la inteligencia artificial.

El uso de motores de inferencia en lenguajes de programación imperativos como por ejemplo java está estandarizado (Java Rule Engine API, JSR 94). Ejemplos de sistemas que incluyen motores de inferencia como parte de sus herramientas de desarrollo son por ejemplo SAP NetWeaver, IBM WebSphere, Oracle Business Rules, JBoss Rules o Microsoft Business Rule Engine.

10.1 Introducción

El lenguaje de la lógica debería permitir representar cualquier conocimiento que quisiéramos utilizar en un problema. No obstante su capacidad para expresar conocimiento se ve entorpecida por la dificultad que supone utilizar su lenguaje para formalizar conocimiento. Esta dificultad la podríamos comparar con la diferencia que existe entre programar en lenguaje máquina o usar un lenguaje de programación de alto nivel.

El lenguaje de la lógica se encuentra a demasiado bajo nivel como para permitir representar de manera sencilla las grandes cantidades de conocimiento necesarias para una aplicación práctica. Es por ello que se crearon las representaciones estructuradas como lenguajes que se acercan más a como estamos nosotros acostumbrados a utilizar y describir el conocimiento.

Estas representaciones solucionan muchos problemas de representación y hacen mas fácil la adquisición de conocimiento. No obstante son restricciones del lenguaje de la lógica de predicados, por lo que no pueden representar cualquier conocimiento.

10.2 Redes semánticas

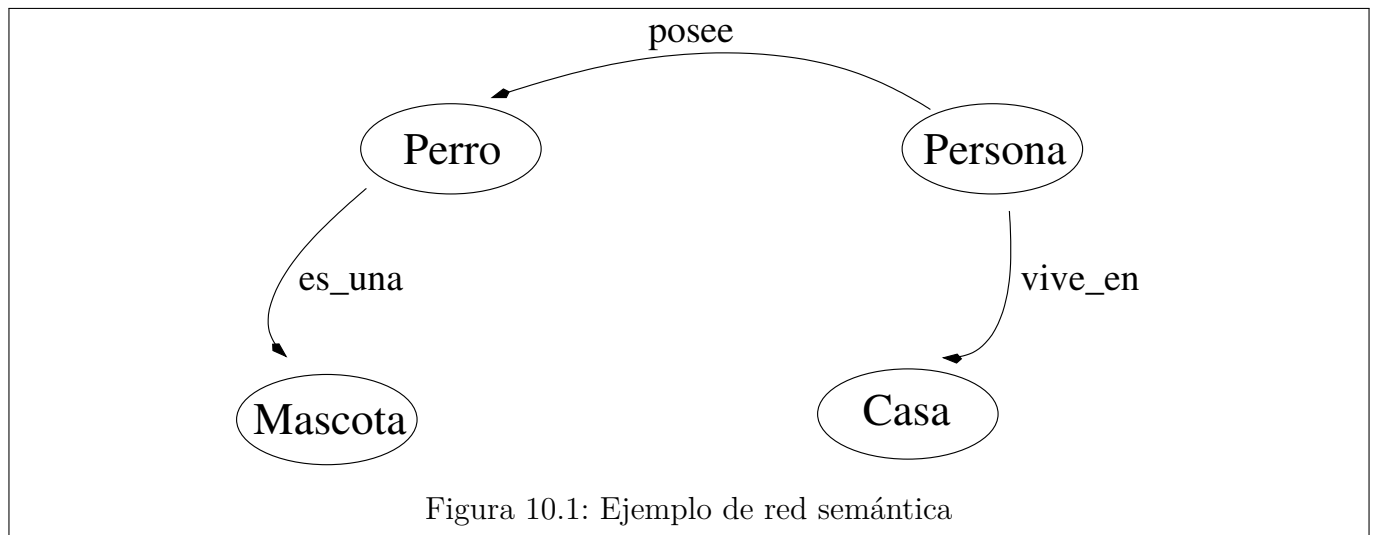
Podemos definir de forma genérica a una red semántica como un formalismo de representación del conocimiento donde éste es representado como un grafo donde los nodos corresponden a conceptos y los arcos a relaciones entre conceptos. En la figura 10.1 se puede ver un ejemplo.

La justificación del uso de este tipo de representación para formalizar conocimiento viene desde la psicología cognitiva, que es el área donde se definieron por primera vez las redes semánticas. Según esta, la representación que utilizamos las personas para almacenar y recuperar nuestro conocimiento se basa en la asociación entre conceptos mediante sus relaciones.

Este formalismo ha ido evolucionando desde su definición para permitir por un lado, una formalización de su semántica y sus capacidades (a partir de lógica de predicados), y por otro, la definición de lenguajes y formalismos que permitan su uso computacional.

A lo largo de su historia han aparecido muchos formalismos de redes semánticas, ampliando sus capacidades permitiendo formalizar diferentes aspectos del conocimiento o especializándolas para dominios concretos¹. De hecho, hay muchos lenguajes de representación que se usan habitualmente, tanto dentro del ámbito de la inteligencia artificial, como en otros ámbitos, que caen bajo la definición de las redes semánticas.

¹Un ejemplo de esta especialización son los múltiples formalismos de redes semánticas aparecidos para el tratamiento del lenguaje natural.



Por ejemplo, el UML o los diagramas de entidad relación puede verse como un notaciones de red semántica especializadas, que permiten describir entidades especializadas, ya sea qué elementos componen una aplicación y como se relaciones entre ellos, o como se describe una base de datos. Sobre estas notaciones se define de manera precisa el significado de cada uno de los elementos para permitirnros hacer diferentes inferencias a partir del diagrama.

Sobre la representación básica (grafo de conceptos y relaciones) se pueden definir mecanismos de razonamiento que permiten responder a cierto tipo de preguntas, como por ejemplo:

- ¿Cierta pareja de conceptos está relacionada entre si?
- ¿Qué relaciona dos conceptos?
- ¿Cual es el concepto más cercano que relaciona dos conceptos?

La evolución de las redes semánticas ha ido añadiéndoles nuevas características que permiten una mejor estructuración del conocimiento, distinguiendo, por ejemplo, entre conceptos/instancias/valores o entre relaciones/propiedades. También han enriquecido la semántica de las relaciones para poder hacer inferencias más complejas, como, por ejemplo, creando relaciones que indiquen taxonomía (clase/subclase/instancia). Veremos todas estas características en la siguiente sección.

10.3 Frames

Los frames evolucionaron a partir de las redes semánticas y se introdujeron a partir de 1970. Estructuran el formalismo de las redes semánticas y permiten una definición detallada del comportamiento de los elementos que se definen. Por un lado un **concepto** (*frame*) es una colección de **atributos** (*slots*) que lo definen y estos tienen una serie de características y restricciones que establecen su semántica (*facets*). Una **relación** conecta conceptos entre sí y también tiene una serie de características que definen su comportamiento.

De entre las relaciones que se pueden definir se establecen como especiales las relaciones de naturaleza taxonómica (clase/subclase, clase/instancia, parte/subparte, ...). Estas permiten establecer una estructura entre los conceptos (Clases/subclases/instancias) permitiendo utilizar relaciones de

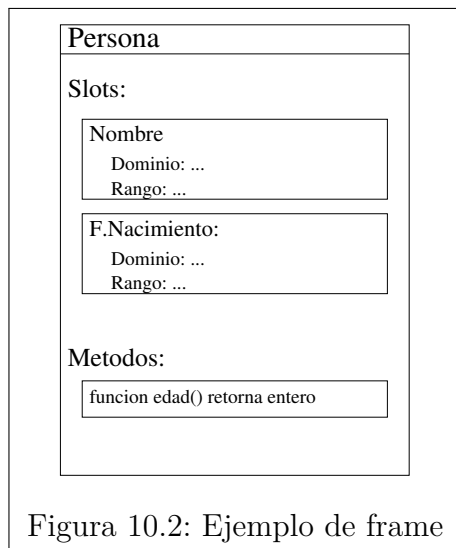


Figura 10.2: Ejemplo de frame

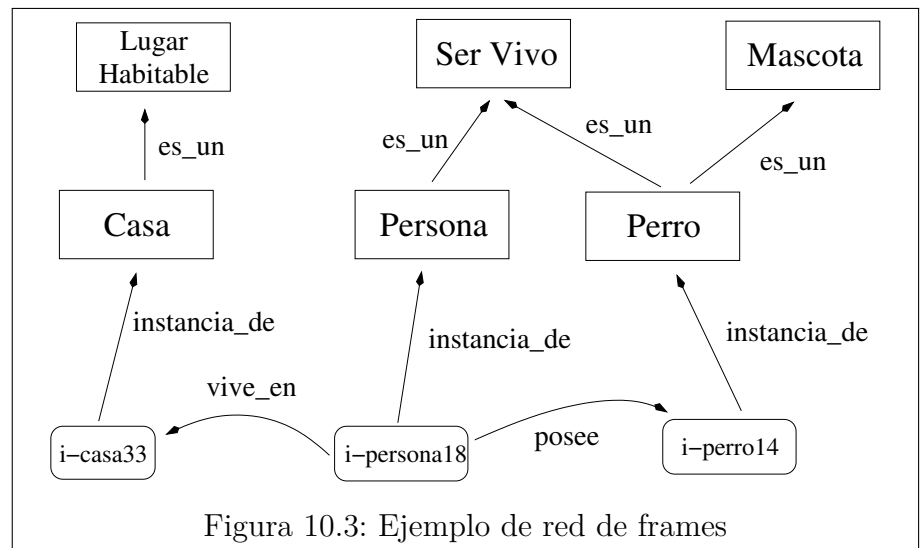


Figura 10.3: Ejemplo de red de frames

generalización/especialización y herencia de atributos como sistema básico de razonamiento².

Estos elementos definen la parte declarativa de la representación. A partir de ellos se puede razonar sobre lo representado y hacer preguntas sobre los conceptos. El mecanismo de razonamiento se basa en la denominada **lógica de la descripción** (*Description Logics*). Esta lógica³ es una restricción de la lógica de predicados que permite describir y trabajar con descripciones de conceptos. Su principal interés es que define algoritmos eficientes de razonamiento sobre definiciones y propiedades. Su principal hándicap es que esta eficiencia se obtiene a costa de no permitir formalizar ciertos tipos de conceptos como por ejemplo negaciones o conceptos existenciales.

Este hándicap hace que muchas implementaciones de este formalismo de representación incluyan mecanismos procedimentales que permitan obtener de manera eficiente deducciones que no se pueden obtener mediante razonamiento. Estos procedimientos permiten resolver problemas concretos en la representación de manera ad-hoc.

Bajo estos mecanismos procedimentales caen lo que se denominan **métodos** que son procedimientos o funciones que pueden invocar los conceptos o las instancias (al estilo de la orientación a objetos) y los **demons** que son procedimientos reactivos que se ejecutan cuando se dan ciertas circunstancias en la representación.

En la figura 10.2 se puede ver un ejemplo de frame. En la figura 10.3 se puede ver una red de frames representando conceptos, instancias y diferentes relaciones.

La **herencia** es el principal mecanismo de razonamiento sobre valores y propiedades que se utiliza en los frames. La herencia es un mecanismo que puede resultar problemático por la circunstancia de que un atributo o valor podría ser heredado por diferentes caminos si permitimos que un concepto pueda ser subclase de varias superclases. Esta *herencia múltiple* puede hacer que no sepamos que valor o definición de una propiedad es la correcta⁴.

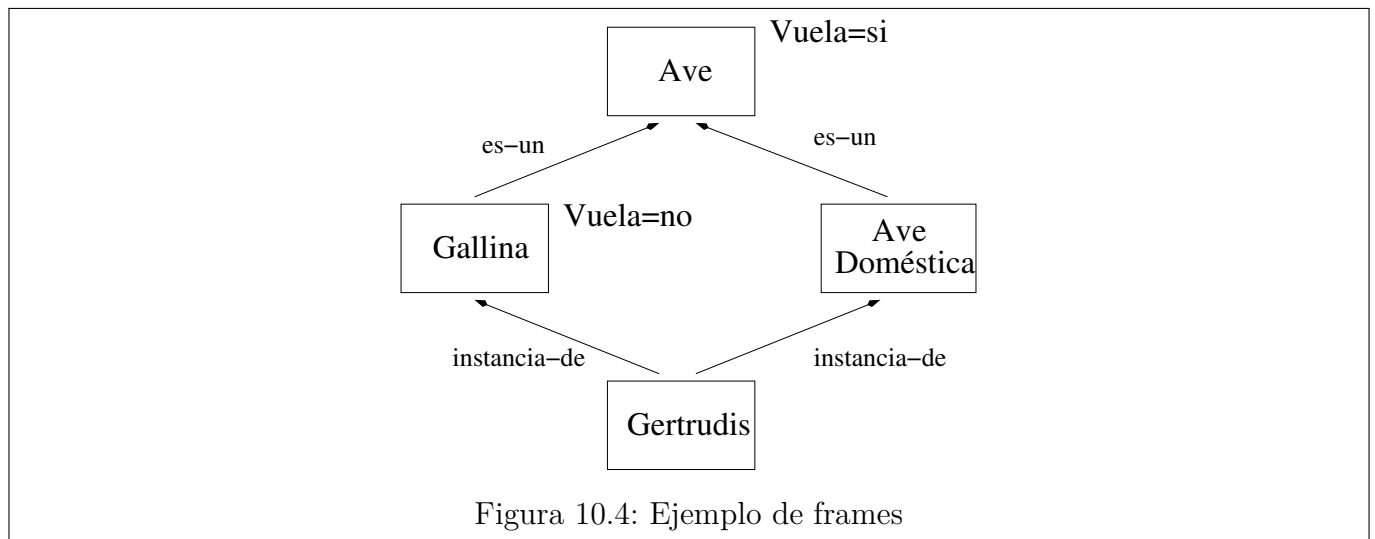
Este problema a veces se puede resolver razonado sobre la representación e identificando qué definiciones de una propiedad ocultan a otras y cuál es la más cercana a donde queremos obtener el atributo o su valor. Para ello se define el *algoritmo de distancia inferencial* que permite saber si existe o no una definición que oculta a las demás. El algoritmo es el siguiente:

1. Buscar el conjunto de frames que permiten heredar el valor del slot → **Candidatos**
2. Eliminar de Candidatos todo frame que sea padre de otro de la lista

²El sistema de representación es muy parecido al que se utiliza en orientación a objetos, de hecho se crearon en la misma época, aunque los elementos esenciales provienen de las redes semánticas, que son una década anteriores.

³De hecho son múltiples lógicas que se diferencian por el nivel de expresividad que se permite.

⁴Estos problemas hacen que por ejemplo lenguajes como java no permitan la herencia múltiple.



3. Si el número de candidatos es:

- a) 0 $\rightarrow$ No se puede heredar el slot
- b) 1 $\rightarrow$ Ese es el valor que buscamos
- c) $> 1 \rightarrow$ Problema de herencia múltiple si la cardinalidad del slot no es N

En ocasiones un problema de herencia múltiple se puede resolver ad-hoc usando los mecanismos procedimentales de la representación.

Ejemplo 10.1 Podemos observar el problema de la herencia múltiple en la figura 10.4. La instancia podría tener simultáneamente el valor **si** y **no** para el atributo **vuela** ya que puede encontrar un camino de herencia para cada uno de ellos.

En este caso el algoritmo de distancia inferencial empezaría con la lista que contendría {Gallina, Ave} como frames candidatos a dar el valor del slot. El frame Ave se eliminaría al ser ascendiente de Gallina, lo cual nos dejaría con el frame del que debemos heredar el slot.

La mayoría de los entornos para desarrollo de aplicaciones en inteligencia artificial poseen como parte del lenguaje una parte que permite representar conocimiento utilizando un mecanismo similar a los frames, aunque muchas veces aparece directamente como un lenguaje orientado a objetos con características adicionales. Veremos uno de estos lenguajes dentro del entorno de CLIPS.

También hay lenguajes de frames que no tienen parte procedimental incorporada y por lo tanto son implementaciones del lenguaje de la lógica de descripción. Un ejemplo de este tipo de lenguajes son los utilizados en la *web semántica*. Estos lenguajes se construyen sobre XML y definen los elementos básicos para crear conceptos, características y relaciones, el lenguaje que actualmente se utiliza es **OWL** (Ontology Web Language)⁵.

10.3.1 Una sintaxis

Frames

Las sintaxis de los lenguajes de frames son muy variadas, en lugar de escoger una concreta, en esta sección vamos a definir un lenguaje de frames genérico que utilizaremos en la asignatura para resolver problemas.

⁵Los lenguajes para la web semántica han tenido una larga evolución, el primero fue **RDF**, que aún se sigue usando ampliamente a pesar de sus limitaciones. Proyectos Europeos y Norteamericanos definieron sobre estos lenguajes más potentes (OIL, DAML) que acabaron fusionándose hasta llegar a OWL que es un estándar del W3C.

Los conceptos se formarán a partir de propiedades y métodos, y se definirán a partir de la construcción **frame**, que tendrá la siguiente sintaxis:

```

Frame <nombre>
  slot <nombre-slot>
  slot <nombre-slot>
  ...
  slot <nombre-slot>
  métodos
    acción <nombre-método> (parámetros) [H/noH]
    ...
    función <nombre-método> (parámetros) devuelve <tipo> [H/noH]

```

Slots

Un **slot** es un atributo que describe un concepto. Cada slot puede ir acompañado de modificadores (*facets*) que definirán las características del slot. Un slot puede ser redefinido en un subconcepto, por lo que puede volver a aparecer con características distintas que ocultan la definición anterior.

Estos modificadores permiten definir la semántica y el comportamiento del atributo e incluyen:

- *Dominio*: Conceptos que pueden poseer este slot
- *Rango*: Valores que puede tener el slot
- *Cardinalidad*: si se admite un único valor o múltiples valores
- *Valor por omisión*: Valor que tiene el slot si no se le ha asignado ninguno
- *Uso de demons*: Procedimientos que se ejecutarán si sucede un evento en el slot, estos procedimientos no tienen parámetros ya que no se pueden llamar explícitamente. Definiremos cuatro tipos de eventos que pueden suceder:
 - **If-needed** (al consultar el slot)
 - **if-added** (al asignar valor al slot),
 - **if-removed** (al borrar el valor)
 - **if-modified** (al modificar el valor)
- *Comportamiento en la herencia*: Si el slot se puede heredar a través de las relaciones.

La sintaxis que utilizaremos para definir un **slot** será la siguiente:

```

Slot <nombre>
  ++ dominio (lista de frames)
  ++ rango <tipo-simple>
  ++ cardinalidad (1 o N)
  valor (valor o lista de valores)
  demons <tipo-demon>
    accion <nombre-accion> / función<nombre-funcion> devuelve <tipo>*
  herencia (por rels. taxonómicas: SI/NO; por rels. usuario: SI/NO)

```

Los facets (propiedades) marcados con ++ son obligatorios en toda descripción de slot. Las acciones/funciones asociadas a los demons de los slots no tienen parámetros. Usan la variable **F** como referencia implícita al frame al cual pertenece el slot que activa el demon. Los demons de tipo **if-needed** solo pueden estar asociados a funciones.

Para la herencia, distinguiremos dos tipos de relaciones, las taxonómicas, que son las que definen la estructura entre los conceptos y estarán predefinidas y las relaciones de usuario que son las que podremos definir nosotros. La herencia a través de cada tipo se comporta de manera diferente. La herencia a través de las relaciones taxonómicas es de definición, de manera que si un concepto hereda un slot éste se encontrará físicamente en las instancias que creamos. La herencia a través de las relaciones de usuario es de valor, de manera que si un concepto hereda un slot solo podremos obtener su valor en una instancia del concepto, si existe una relación con una instancia que posea ese slot y la relación nos permite heredarlo.

Consideraremos que por omisión tendremos herencia a través de las relaciones taxonómicas y no la tendremos a través de las de usuario.

Métodos

Los métodos serán procedimientos o funciones que permitirán realizar un cálculo concreto a partir de una clase o una instancia. Estos métodos de clase podrán hacer cálculos a partir de todas las instancias a las que tenga acceso a través de sus relaciones. Pueden ser heredables o no, ya que algunos casos podría tener sentido la herencia del método en sus subclases o instancias como una especialización de este. Los métodos se invocan de manera explícita, por lo que podrán tener parámetros.

Las acciones/funciones que describen los métodos usarán la variable **F** como referencia implícita al frame en el que se activa el método, y por ello no se ha de pasar como parámetro⁶.

Relaciones

Las relaciones permiten conectar conceptos entre si, definiremos su comportamiento a partir de un conjunto de propiedades:

- *Dominio*: Conceptos que pueden ser origen de la relación.
- *Rango*: Conceptos que pueden ser destino de la relación.
- *Cardinalidad*: Número de instancias del rango con las que podemos relacionar una instancia del dominio.
- *Inversa*: Nombre de la relación inversa y su cardinalidad.
- *Transitividad*: Si es transitiva entre instancias.
- *Composición*: Como se puede obtener con la composición de otras relaciones.
- *Uso de demons*: Procedimientos que se ejecutarán si sucede un evento en la relación, estos procedimientos no tienen parámetros ya que no se pueden llamar explícitamente. Definiremos dos tipos de eventos que pueden suceder:
 - **If-added**: Si establecemos la relación entre instancias
 - **If-removed**: Si eliminamos la relación entre instancias

⁶Es equivalente por ejemplo a la variable de autoreferencia **this** de java o C++.

- *Slots heredables*: Slots que se pueden heredar a través de esta relación (solo el valor). Dado que las relaciones se definen bidireccionales, asumiremos que los slots se heredan en el sentido que corresponda, o sea, del frame en el que esta definido el slot al que lo debe heredar.

La sintaxis para definir relaciones es la siguiente:

Relación <nombre>

```
++ dominio (lista de frames)
++ rango (lista de frames)
++ cardinalidad (1 o N)
++ inversa <nombre> (cardinalidad: 1 o N)
transitiva SI/NO [por defecto es NO]
compuesta NO/<descripción de la composición> [por defecto es NO]
demons (<tipo-demon> acción <nombre-acción>)
herencia (lista de slots) [por defecto es lista vacía]
```

Los descriptores marcados con ++ son obligatorios en toda descripción de relación. Las acciones asociadas a los demons de las relaciones no tienen parámetros. Éstos usan las variables D y R como referencia implícita al frame origen y destino de la conexión que se está intentando añadir o eliminar entre los dos frames.

Respecto a la transitividad, para que podamos tener una relación transitiva su dominio y rango han de poder ser transportables entre instancias. Esto quiere decir que o el dominio y el rango de la relación están definidos sobre instancias de un mismo frame, o que en el rango y el dominio de la relación aparecen frames comunes. Por supuesto esto no es suficiente para que una relación sea transitiva, es necesario que la semántica de la relación lo permita.

Respecto a la composición, para que una relación sea compuesta la semántica de la relación ha de corresponder al resultado de la aplicación de dos o mas relaciones. Esto quiere decir que no es suficiente con que encontremos una cadena de relaciones entre una pareja de frames, esta cadena ha de significar lo mismo que la relación que estamos definiendo.

Como se ha comentado en el apartado sobre los slots, se distinguen dos tipos de relaciones. Por un lado están las taxonómicas, que están predefinidas y se limitan a **es-un** e **instancia-de**. La primera de ellas es una relación entre clases y la segunda entre instancias y clases. Por otro lado están las relaciones de usuario, que solo podrán ser entre instancias.

Sintaxis y elementos predefinidos

La expresión <nombre-frame>.<nombre-relación> nos dará el frame (si la cardinalidad es 1) o la lista de frames (si la cardinalidad es N) con los cuales esta conectado a través de la relación <nombre-relación>. Para consultar la cardinalidad se puede usar una función predefinida `card(<nombre-frame>.<nombre-relación>)`.

Tendremos predefinidas las siguientes relaciones taxonómicas:

- Relación **es-un** (inversa: **tiene-por-subclase**) transitiva SI
- Relación **instancia-de** (inversa: **tiene-por-instancia**) composición: **instancia-de** $\otimes$ **es-un**

Usaremos un arco para unir todas las relaciones **es-un** de un conjunto de subclases (ver figura 10.5) para indicar que las subclases son una partición completa del dominio, es decir, que no existen otras subclases que las definidas y que toda instancia debe pertenecer a una de esas subclases.

También supondremos que disponemos de las siguientes funciones booleanas predefinidas:

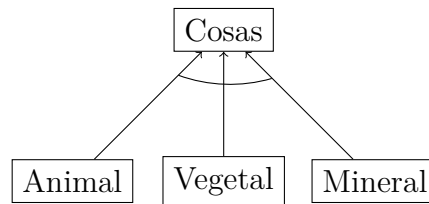


Figura 10.5: Subclasificación completa de un concepto

`<slot>?(<frame>)` Nos dice si `<frame>` posee este slot o no (activando la herencia si hace falta)

`<relación>?(<frame>)`

Nos dice si `<frame>` está conectado con algún otro frame a través de la relación indicada por la función

`<relación>?(<frame-o>, <frame-d>)`

Nos dice si existe una conexión entre `<frame-o>` y `<frame-d>` etiquetada con la relación indicada por la función

11.1 Introducción

El estudio de la representación del conocimiento no es exclusivo de la inteligencia artificial, sino que es un tema que se origina desde el primer momento en el que el hombre se planteó el entender y describir el mundo en el que se encuentra.

A este área estudio se la encuadra dentro de la filosofía y fue Aristóteles quien acuñó el término **Categoría** como la palabra para describir las diferentes clases en las que se dividían las cosas del mundo. A este área de la filosofía se la conoce actualmente como *ontología*, término relativamente moderno (s. XIX) que proviene del griego *Ontos* (Ser) y *Logos* (Palabra), literalmente “las palabras para hablar de las cosas”. Este término se empezó a utilizar para distinguir el estudio de la categorización del ser de la categorización que se hacía por ejemplo en biología. De hecho el trabajo de categorización surge en muchas áreas de la ciencia (filosofía, biología, medicina, lingüística, ...). La inteligencia artificial es una más de las áreas interesadas en este tema.

El objeto de estudio de la ontología son las entidades que existen en general o en un dominio y como se pueden agrupar de manera jerárquica en una categorías según sus diferencias y similitudes. El resultado de este estudio es lo que denominamos una **ontología**.

Una ontología se puede definir formalmente como:

“Una ontología es un catálogo de los tipos de cosas que asumimos que existen en un dominio $\mathcal{D}$ desde la perspectiva de alguien que usa un lenguaje $\mathcal{L}$ con el propósito de hablar de $\mathcal{D}$. Los elementos de una ontología representan predicados, constantes, conceptos y relaciones pertenecientes a un lenguaje $\mathcal{L}$ cuando se usa para comunicar información sobre $\mathcal{D}$.”

(Sowa, 1999)

Otras definiciones menos formales podrían ser:

“Una ontología es la definición de los términos básicos y relaciones que comprenden el vocabulario de un dominio y las reglas para poder combinarlos y definir extensiones a ese vocabulario.”

(Neches, 1991)

“Una ontología es una especificación formal de una conceptualización compartida.”

(Borst, 1997)

Una ontología es pues un vocabulario, un conjunto de términos y expresiones que escogemos para representar una realidad, a través del cual comunicarnos con otras entidades que la compartan.



Un ejemplo evidente de ontología es el diccionario de una lengua. Un diccionario es la recopilación de todas las palabras que usan los hablantes de un idioma para comunicarse. El diccionario es el conjunto de los términos que comparten los hablantes al referirse a la realidad. Cada palabra en el diccionario tiene descritas sus diferentes características de manera que los hablantes sepan como deben utilizarse y su significado. De hecho los diccionarios como descripción del conocimiento del dominio lingüístico son un elemento clave en la construcción de sistemas de tratamiento de lenguaje natural y habitualmente tienen una organización más estructurada de la que estamos acostumbrados a ver.

Lo que queda definido en una ontología se puede ver como una representación declarativa de un dominio. El uso, las maneras de combinar estos elementos y la obtención de información a partir de las expresiones formadas con los elementos de las ontologías pasa a través de la lógica.

La lógica se puede ver como un mecanismo de manipulación/ejecución que por sí misma no habla explícitamente sobre un dominio específico. Podemos ver que sus expresiones son neutras respecto al significado de sus átomos y predicados, es su combinación con una ontología lo que le da a un formalismo lógico la capacidad de expresar significados, por ejemplo:

$$\frac{P \rightarrow Q \quad P}{Q}$$

Este razonamiento no habla sobre nada en concreto salvo que asignemos significados a los átomos ($P = \text{lueve}$, $Q = \text{me mojo}$). A partir del momento en el que ligamos los átomos con los conceptos de la ontología estamos diciendo cosas y razonando sobre el dominio de esa ontología. El mismo razonamiento hablaría de algo distinto si cambiáramos la ontología que define el significado de los átomos.

11.2 Necesidad de las ontologías

Existen varios motivos por los que las ontologías son de utilidad en el ámbito de la inteligencia artificial:

1. *Permiten compartir la interpretación de la estructura de la información entre personas/agentes*

Una ontología fija un conjunto de términos que denotan elementos de la realidad, el establecer una ontología sobre un dominio específico permite que dos agentes puedan entenderse sin ambigüedad y sepan a que se refieren. Eliminamos de esta manera la ambigüedad en la comunicación (al menos en lo que se refiere a los términos que se utilizan y su significado).

Por ejemplo, un idioma es una ontología que comparten todos sus hablantes. El significado de cada término está recogido en un diccionario y cuando dos hablantes se comunican entre si, saben que tienen ese conjunto de términos y significados en común y asumen que cada uno entiende lo que el otro dice.

2. *Permiten reusar el conocimiento*

Hacer una descripción de un dominio permite que esta pueda ser usada por otras aplicaciones que necesiten tratar con ese conocimiento. La descripción del conocimiento se puede hacer independiente de su uso y una ontología suficientemente rica puede ser utilizada en múltiples aplicaciones.

3. *Hacen que nuestras suposiciones sobre el dominio se hagan explícitas*

Escribir una ontología exige la formalización al máximo detalle de todos los elementos del dominio y su significado y hacer explícitas todas las suposiciones que se van a utilizar. Esto facilita reflexionar sobre él y poder analizar las suposiciones realizadas para poder cambiarlas y actualizarlas de manera más sencilla. También ayuda a que otros puedan analizar y entender su descripción.

4. *Separan el conocimiento del dominio del conocimiento operacional*

Permite hacer independientes las técnicas y algoritmos para solucionar un problema del conocimiento concreto del problema. De hecho una ontología es una descripción declarativa del dominio, de manera que no asume una metodología específica de utilización del conocimiento.

5. *Permiten analizar el conocimiento del dominio*

Una vez tenemos una especificación del conocimiento podemos analizarlo utilizando métodos formales (para comprobar si es correcto, completo, consistente, ...)

11.3 Desarrollo de una ontología

En Inteligencia Artificial una ontología será una descripción formal explícita de los conceptos de un dominio (**Clases**). Estas clases se describirán a partir de **propiedades** que representarán las características, atributos y relaciones de las clases. Adicionalmente estas características tendrán **restricciones** (tipo, cardinalidad, ...). Finalmente tendremos **instancias** (elementos identificables) que constituirán los individuos concretos que representa la ontología.

Eso quiere decir que el desarrollo de una ontología requerirá definir las clases que forman el dominio, organizar las clases en una jerarquía taxonómica según sus afinidades, definir las propiedades de cada clase e indicar las restricciones de sus valores y asignar valores a las propiedades para crear instancias.

El cómo se realiza esto se puede encontrar en las diferentes metodologías de desarrollo de ontologías que hay descritas en la literatura de representación del conocimiento. Estas metodologías son bastante complejas, dado que en sí el desarrollo de una ontología para un dominio real es una labor compleja.

Para poder desarrollar ontologías pequeñas describiremos una metodología informal que permitirá analizar los elementos de un dominio y obtener un resultado que se puede utilizar en aplicaciones de complejidad media¹.

Antes de empezar con la metodología es necesario tener presente que:

1. No existe un modo *correcto* de modelar un dominio. La mejor solución dependerá de la aplicación/problema concreto.
2. El desarrollo de una ontología es un proceso iterativo.
3. Los objetos de la ontología deberían ser cercanos a los objetos y relaciones que se usan para describir el dominio (generalmente se corresponden a nombres y verbos que aparecen en frases que describen el dominio).

¹Una descripción algo mas detallada de esta metodología y un ejemplo sencillo lo podéis encontrar en el artículo “*Ontology Development 101: A Guide to Creating Your First Ontology*”, Noy & McGuinness, (2000) que encontrareis en la web de la asignatura.

Fases de desarrollo

1. Determinar el dominio y la cobertura de la ontología

Deberemos saber qué elementos queremos que aparezcan en la ontología que vamos a desarrollar, por lo que debemos identificar qué parte del dominio nos interesa describir. Dependiendo del objetivo de uso de la ontología los términos que deberán aparecer pueden ser muy diferentes, por lo que es importante tenerlo claro.

Una forma adecuada de saber qué elementos debemos representar es plantearnos que tipo de preguntas y respuestas deseamos de la ontología. Es lo que se denomina **preguntas de competencia**.

También es interesante plantearse quién va a usar y mantener la ontología. No será lo mismo desarrollar una ontología restringida que vamos a utilizar en nuestra aplicación, que desarrollar una ontología de un dominio amplio que pretendemos reusar o que reusen otros.

Ejemplo 11.1 *Supongamos que necesitamos una ontología para representar el funcionamiento de una facultad y todos los elementos que están relacionados con ella. Dependiendo del uso que vayamos a darle a la ontología necesitaremos unos conceptos u otros.*

Si queremos utilizar la ontología para recomendar a un alumno de qué nuevas asignaturas se puede matricular el próximo cuatrimestre, seguramente necesitaremos describir que es una asignatura, como se organizan el plan de estudios y en los ciclos del plan de estudio, que temas tratan, cual es su carga de trabajo, posiblemente también nos interese guardar información histórica sobre ella. Además necesitaremos representar que es un expediente, que es una convocatoria, que es un horario, ...

Si en cambio nos interesa hacer un programa que permita dialogar con un estudiante de bachillerato para responder sus dudas sobre como funciona la facultad tendremos que poner énfasis en otras características, como cual es la organización del plan de estudios, que órganos componen la facultad, que normativas se aplican, que temas se tratan en la carrera, que departamentos hay y cual es su función, que trámites tiene que realizar para matricularse, cual es el proceso de ese trámite, de que equipamientos dispone la facultad, ...

Podemos formular un conjunto de cuestiones de competencia que queremos que nuestra ontología sea capaz de responder, para el primer dominio de aplicación, por ejemplo:

- *¿Qué asignaturas son prerrequisito de otra?*
- *¿Cuál es la carga de laboratorio de la asignatura Criptografía?*
- *¿De qué asignaturas optativas me puedo matricular después de hacer Inteligencia Artificial?*
- *¿Que horarios de mañana tiene la asignatura Compiladores?*
- *¿De qué asignaturas de libre elección se puede matricular un alumno de fase de selección?*
- *¿Cuántas asignaturas del perfil “Tècniques avançades de programació” me quedan por hacer?*

A partir de estas preguntas podemos, por ejemplo, ver qué necesitamos definir el concepto asignatura, que tendrá diferentes especializaciones por varios criterios, estas tendrán relaciones de precedencia, estarán asociadas a perfiles, una asignatura tendrá diferentes tipos de carga de trabajo que se medirá en créditos, ...

2. Considerar la reutilización de ontologías existentes

Las ontologías se construyen para comunicar conocimiento en dominios, por lo que se construyen con la idea de compartición y reutilización. Se supone que una ontología establece un vocabulario común, por lo que no es nada extraño que ya alguien haya estudiado el dominio y creado ese vocabulario. Por lo tanto, no es necesario rehacer un trabajo que ya está hecho, si existe una ontología sobre el dominio en el que trabajamos, podemos incorporarla. En la última sección de este capítulo enumeraremos proyectos de ontologías que pueden ser el punto de partida desde el que podemos desarrollar una ontología para nuestro dominio en particular.

3. Enumerar los términos importantes en la ontología

Escribir una lista de términos que podemos usar para referirnos a nuestro dominio, elaborando frases que podríamos utilizar para preguntarnos cosas sobre él o para explicar a alguien información sobre él. Deberemos pensar en:

- ¿Qué propiedades tiene esos términos?
- ¿Qué nos gustaría decir sobre ellos?

El objetivo de esta fase es tener una visión informal de los conceptos y elementos que necesitaremos tener en cuenta para formalizar el dominio.

Ejemplo 11.2 *Siguiendo con el ejemplo de la ontología de la facultad, tendríamos que recolectar todos los términos que vamos a usar en la ontología: asignatura, horario, aula, prerrequisito, perfil, crédito, tema, departamento, convocatoria, ...*

4. Definir las clases y su jerarquía

Los conceptos no aparecen desvinculados entre sí y de hecho un conjunto desorganizado de conceptos no es útil, por lo que deberemos descubrir su estructura y sus relaciones de generalización y especialización. Podemos tomar diferentes aproximaciones

- **De arriba a abajo:** Definimos los conceptos mas generales y vamos especializándolos
- **De abajo a arriba:** Definimos las clases más específicas y vamos agrupándolas según propiedades comunes, generalizando
- **Combinación de ambas:** Definimos los conceptos mas importantes y especializamos y generalizamos para completar la ontología

Ninguno de estos métodos es esencialmente mejor y depende en gran medida del dominio. Existen dominios en los que es fácil descubrir los conceptos generales y se ha de trabajar para obtener las especializaciones más útiles o adecuadas. Hay dominios en los que es más fácil razonar a partir de conceptos específicos y hay que buscar una manera coherente y útil de organizarlos. Muchas veces es la experiencia en la construcción de ontologías la que nos hace decidarnos por una metodología u otra.

Este paso y el siguiente están estrechamente relacionados y es muy difícil hacer primero uno y luego el otro. Por lo general se realizan iterativamente en varios ciclos.

Ejemplo 11.3 *Por ejemplo podemos definir una clasificación para el concepto asignatura como el que aparece en la figura 11.1*

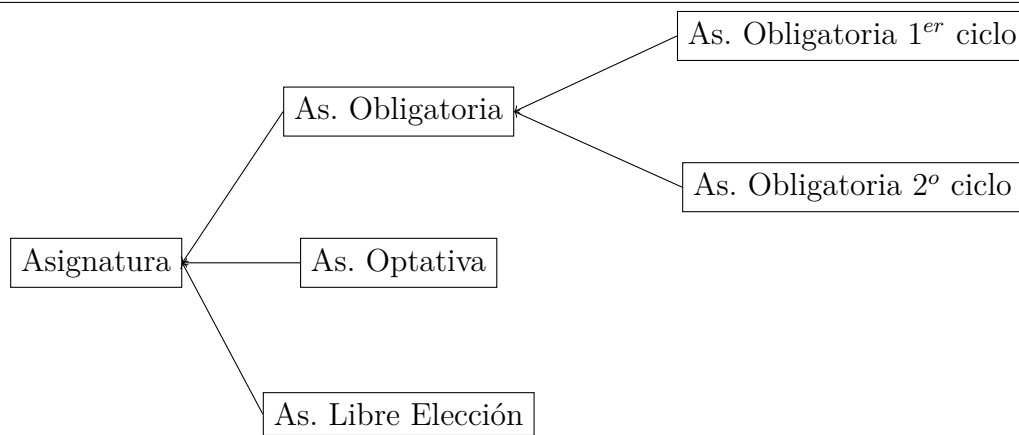


Figura 11.1: Jerarquía del concepto asignatura

5. Definir las propiedades de las clases

Debemos describir la estructura interna de las clases, esta dependerá de la semántica que queremos que tenga. Deberemos determinar una lista de características que describen esa semántica y en que clases concretas debemos poner esas características. La elección de estas propiedades puede depender del dominio de aplicación, restringiéndolas a solo las necesarias o podemos desarrollar la ontología con una visión más amplia para que esta pueda ser utilizada en otros dominios de aplicación.

En la descripción de los conceptos nos podemos encontrar muchos tipos de propiedades

- Propiedades descriptivas, cualidades
- Propiedades identificadoras, nombres
- Partes u otras relaciones taxonómicas
- Relaciones con instancias de otras clases

Desde un punto de vista de la descripción de clases, las relaciones se pueden ver al mismo nivel que las propiedades y de hecho muchos lenguajes de descripción de ontologías no hacen la distinción. Básicamente, podríamos considerar una propiedad a aquel atributo cuyo valor es un tipo predefinido (booleano, numérico, carácter, ...) y una relación sería una propiedad en la que los elementos son de la ontología.

Junto con la determinación de cuales son las propiedades necesarias, estas deberían asignarse a clases de la jerarquía de conceptos. Lo normal es asignar las propiedades a la clase mas general, dejando que el resto las obtengan vía herencia.

Ejemplo 11.4 *Siguiendo con el ejemplo, podríamos definir las características y relaciones que representan una asignatura incluyendo: Nombre, Créditos_Totales, Créditos_de_Teoría, ..., con_tema, impartida_por_departamento, perteneciente_al_perfil, ...*

6. Definir las características de las propiedades

Para describir las propiedades deberemos identificar también sus características y restricciones, entre estas podemos tener:

- Cardinalidad (número de valores permitidos)
- Tipo, valores

- Valores por defecto
- Obligatoriedad
- Si es una relación hay que definir su cardinalidad y su rango y si tiene inversa.

Ejemplo 11.5 *Podemos definir las características de las propiedades de asignatura, por ejemplo el Nombre es una cadena de caracteres y es un atributo obligatorio, los Créditos_Totales es un número real, impartida_por_departamento sería una relación entre asignatura y el departamento que la imparte con cardinalidad 1 y con una relación inversa con cardinalidad N, ...*

7. Crear instancias

La ontología supone un lenguaje de definición que utilizaremos para hablar de elementos concretos. En este caso los elementos concretos son las instancias. Es posible que una ontología tenga como parte de su definición un conjunto de instancias que aparecen en cualquier uso que podamos hacer de ella. En este caso este sería el momento de decidir cuáles son esos individuos y crearlos a partir de las definiciones que hemos determinado.

En el momento de usar la ontología tendremos que crear otras instancias específicas para el problema que vayamos a resolver. Para ello utilizaremos los términos de la ontología que hemos desarrollado para crear una representación del problema. Obviamente, la ontología nos impone los límites de lo que podremos representar, pero si hemos desarrollado correctamente la ontología tendremos el vocabulario necesario.

Consejos prácticos

Estos son unos consejos prácticos a tener en cuenta a la hora de desarrollar una ontología:

- No incluir versiones singulares y plurales de un término (la mejor política es usar solamente nombres en singular o plural). Estamos creando una terminología, por lo que es de sentido común usar un criterio uniforme a la hora de dar nombres a los conceptos que utilizaremos. Es posible que queramos representar individuos y agrupaciones de individuos en nuestro problema, tenemos que tener en cuenta que semánticamente son cosas diferentes y no deberíamos usar las mismas palabras.
- Los nombres no son las clases, debemos distinguir la clase del nombre que le damos. Podemos tener sinónimos, pero todos representan a la misma clase. Es a veces sencillo confundir el nombre de una cosa con la cosa en sí. Estamos estableciendo los conceptos que existen en nuestro dominio, los nombres no son entidades independientes.
- Asegurarnos de que la jerarquía está correctamente construida. Una jerarquía incorrecta afecta a nuestra capacidad para deducir a partir de la ontología y obtener respuestas correctas a nuestras preguntas. Hemos de pensar que la forma principal de razonamiento en esta representación es la herencia.
- Observar las relaciones de transitividad y comprobar si son correctas. La transitividad es un mecanismo importante de deducción y además ahorra explicitar en la ontología muchas relaciones que se pueden deducir vía este mecanismo, establecer relaciones transitivas que no lo son solo puede dar problemas. También es una relación que puede hacer muy ineficiente el razonamiento y puede dar lugar fácilmente a una explosión combinatoria, no conviene utilizarla más de lo necesario.

- Evitar ciclos en la jerarquía. Obviamente es algo difícil de hacer en una ontología sencilla, pero si el número de clases es grande y permitimos herencia múltiple es algo que hay que vigilar.
- Todas las subclases de una clase deben estar al mismo nivel de generalidad. Cada especialización debe llevar a conceptos que tengan el mismo nivel de descripción. Mezclar conceptos generales y específicos en un mismo nivel hace la ontología confusa y difícil de interpretar. También puede llevar a deducciones incoherentes.
- No hay un criterio respecto al número de clases que debe tener un nivel de la jerarquía, la experiencia dice que un número entre dos y doce es habitual, más clases indicaría que tenemos que estructurarlas añadiendo más niveles
- ¿Cuándo introducir nuevas clases? Suele ser incómodo navegar por jerarquías o muy planas o muy profundas, se debería elegir un punto intermedio, unas indicaciones serían:
 - Las nuevas clases tienen propiedades adicionales que no tiene la superclase
 - Tienen restricciones diferentes
 - Participan en relaciones diferentes

No obstante nos puede interesar crear clases porque existen en el dominio aunque no tengan atributos o relaciones distintas, o porque hacen más clara la comprensión de la ontología.

- Decidir si hemos de usar una propiedad o crear una clase. A veces un atributo es suficientemente importante como para considerar que sus valores diferentes corresponden a objetos diferentes.
- Decidir donde está el nivel de las instancias. Pensar cual es nivel mínimo de granularidad que necesitamos.
- Limitar el ámbito de la ontología a las necesidades de representación.
 - La ontología no necesita incluir todas las clases posibles del dominio, solo las necesarias para la aplicación que se va a desarrollar.
 - Tampoco necesitamos incluir todos los atributos/restricciones/relaciones posibles.

11.4 Proyectos de ontologías

Como se comentó al principio del capítulo, una ontología se crea con el propósito de que sea compartida y sirva como un lenguaje común para que diferentes agentes puedan intercambiar conocimiento. Esto ha hecho que se hayan desarrollado múltiples proyectos de investigación encaminados a escribir ontologías tanto en dominios específicos como generales.

Uno de los proyectos de ontologías mas ambiciosos es **CYC**. Este proyecto comenzó en los años 1980 y su pretensión es crear una base de conocimiento que describa todo el conocimiento de sentido común que puede ser necesario para escribir una aplicación de inteligencia artificial. Este proyecto ha dado lugar a una ontología de dominio público OpenCYC que contiene aproximadamente cientos de miles de conceptos y millones de aserciones sobre ellos. El lenguaje en el que esta escrita esta ontología es CYCL que esta basado en la lógica de predicados. Esta ontología se puede utilizar tal cual para escribir aplicaciones o se pueden coger subconjuntos con conceptos para dominios específicos (microteorías). Se puede ver la estructura de conceptos más generales en la figura 11.2.

Como comentamos al principio de esta capítulo, el diccionario de un idioma puede verse como una ontología. Esto ha llevado al desarrollo de múltiples ontología especializadas en el dominio del

tratamiento del lenguaje natural. Un ejemplo de este tipo de ontologías es **Wordnet** que es una ontología léxica para el idioma inglés. Esta ontología está organizada según categorías semánticas y etiquetado con las categorías sintácticas correspondientes a cada palabra. En la actualidad contiene 150.000 palabras inglesas organizadas en 115.000 sentidos. La ontología está estructurada jerárquicamente mediante la relación de hiperonimia/hiponimia. esta ontología esta pensada para aplicaciones de lenguaje natural, pero también se pueden utilizar los conceptos y la estructuración para otras aplicaciones. Se puede ver la estructura de conceptos más generales en la figura 11.3. En la actualidad existen versiones de esta ontología para muchos idiomas y se utiliza ampliamente en muchas aplicaciones que involucran la comprensión del lenguaje natural.

Dentro de los proyectos de ontologías existen algunos que intentan crear una ontología de los conceptos mas generales que se deberían utilizar en la construcción de ontologías. El objetivo de estas ontologías no es pues recolectar todos los conceptos de un dominio, sino dar los conceptos bajo los cuales estos deberían estar organizados. Este tipo de ontologías se denominan *Upper Model* y no existe de momento ninguna ontología de este tipo que sea ampliamente aceptada. De hecho desde el punto de vista teórico hay argumentos a favor y en contra de que exista o se pueda construir una ontología de este tipo.

Dentro de este tipo de proyectos cae **Generalized Upper Model**. Es una ontología léxica pensada para tratamiento del lenguaje natural que posee alrededor de 250 conceptos. Se puede ver la estructura de conceptos en la figura 11.4.

También existen multitud de ontologías para dominios específicos. Por ejemplo, en comercio electrónico existen ontologías estándar definidas por organismos internacionales para la clasificación y nomenclatura de productos y servicios que se utilizan para el intercambio de información. Otro dominio con múltiples ontologías es la medicina, área que destaca por aplicaciones donde hay grandes necesidades de almacenamiento e intercambio de información, y que necesita nomenclaturas muy específicas. Otras áreas con ontologías muy utilizadas incluyen la ingeniería, el mundo empresarial o la química.

El interés por construir ontologías se ha visto impulsado en la actualidad por el proyecto **Semantic Web**. La ambición de este proyecto es el desarrollo de un lenguaje de ontologías que permita describir el contenido de las páginas de web para que puedan ser procesadas de manera automática. Actualmente el contenido de la web está pensado para ser accedido por personas, el contenido está poco estructurado, está en lenguaje natural y el lenguaje de representación está mas pensado para el formato del contenido que para su descripción.

Los lenguajes de descripción de contenido para la web se construyen a partir del lenguaje XML. El primero de ellos fue **RDF/RDFS** (Resource Description Format/Resource Description Format Schema) que permite definir categorías y relaciones aunque está bastante limitado como lenguaje de representación. Sobre este lenguaje se construyeron **DAML** (Darpa Agent Markup Language) y **OIL** (Ontology Inference Layer), resultado de proyectos sobre web semántica en Estados Unidos y Europa respectivamente.

Estos lenguajes definen las características necesarias para tener un lenguaje de descripción de ontologías y, de hecho, se pueden ver como ontologías de dominio que definen el vocabulario necesario para representar conocimiento. Estos dos lenguajes se fusionaron y el lenguaje resultante se llama **OWL** (Ontology Web Language) que es un estándar del W3C. Este lenguaje se basa en lógica de descripción y tiene diferentes versiones: OWL lite, OWL DL y OWL full. La primera es la menos expresiva, pero garantiza que se puede razonar sobre el en tiempo finito. La segunda se permite representar expresiones en lógica de descripción sin limitaciones, pero no asegura poder resolver cualquier expresión en tiempo finito. Para la última versión no hay implementaciones de razonadores que puedan tratarla.

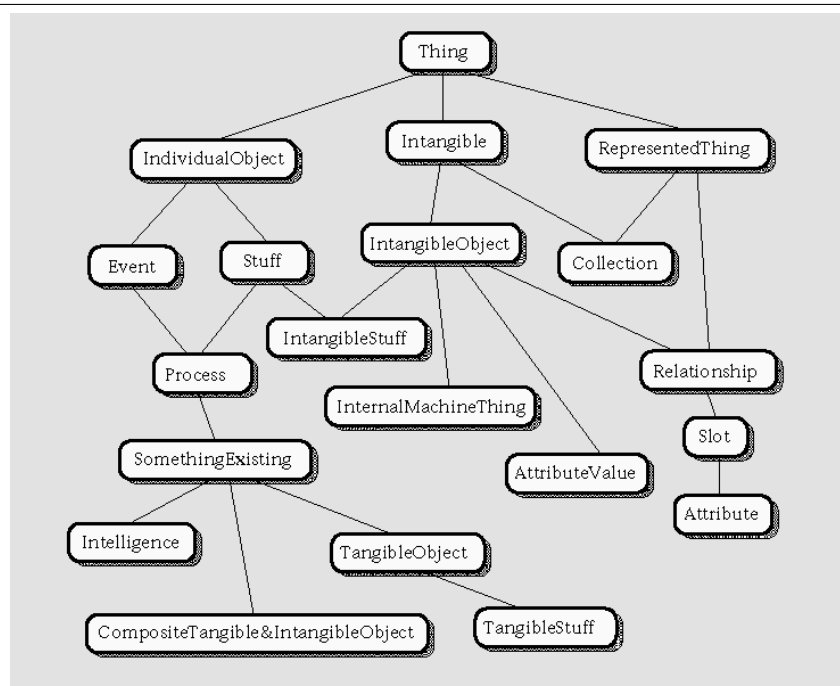


Figura 11.2: Ontología de CYC

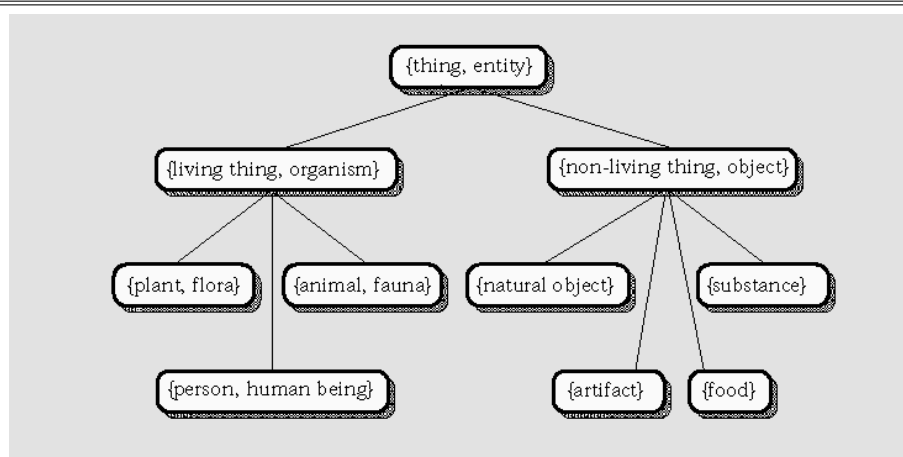


Figura 11.3: Ontología de Wordnet

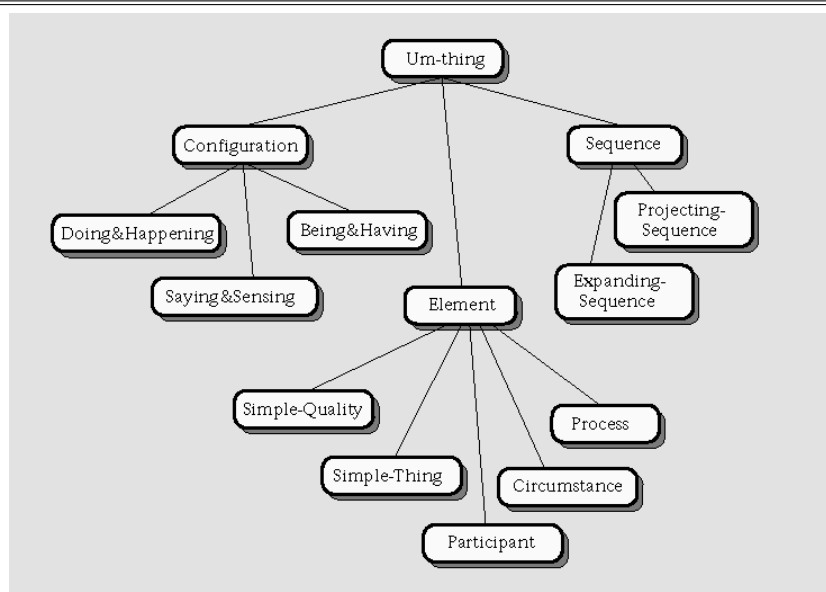


Figura 11.4: Ontología de Generalized Upper Model

Parte III

Sistemas basados en el conocimiento

12.1 Introducción

Un tema esencial en el área de Inteligencia Artificial es la resolución de problemas. Ya hemos visto en temas anteriores de la asignatura métodos generales que a partir de una representación del problema basada por ejemplo en el espacio de estados o en el de soluciones y un conjunto de operadores específicos, se exploraba este espacio en busca de una solución al problema.

Este tipo de métodos no utilizaban apenas conocimiento del dominio del problema que intentaban solucionar y se guiaban por funciones heurísticas generales que se basaban en unas pocas características del problema. Estas funciones heurísticas intentaban guiar la elección del camino solución ordenando los pasos accesibles durante la exploración. La capacidad para generar el orden adecuado en la exploración es lo que permitía reducir en la práctica su coste computacional.

Evidentemente, las decisiones que se pueden tomar a partir de los valores de una función heurística a veces no son suficientemente buenas para reducir sensiblemente el espacio de búsqueda o, simplemente, es imposible encontrar una función heurística adecuada.

Este tipo de métodos generales se utilizaron como métodos básicos de resolución de problemas en los primeros años de la Inteligencia Artificial ya que permitían su aplicación de forma relativamente sencilla a cualquier dominio. Se pudo comprobar que el coste computacional de este tipo de métodos era prohibitivo en muchos problemas reales y que hacían falta otras metodologías que mejoraran su eficiencia. El elemento a introducir para conseguir esta eficiencia es el conocimiento específico del dominio del problema a resolver. Este conocimiento puede acelerar el proceso de resolución al mejorar las decisiones que se toman.

Los métodos generales son denominados *métodos débiles* frente a los que denominaremos *métodos fuertes*, que explotarán el conocimiento del dominio. El uso de conocimiento particular hará que estos métodos necesiten más trabajo de desarrollo a la hora de aplicarlos y que la experiencia de usarlos en un problema no sea exportable directamente a otros problemas, ya que estará orientada al problema concreto que se resuelve.

El principal exponente de estos métodos que se basan en conocimiento específico del dominio son los que se denominaron en su origen *sistemas expertos* (*expert systems*) y que actualmente son conocidos también como *sistemas basados en el conocimiento* (*knowledge based systems*).

Originalmente, el término sistema experto intentaba reflejar que el objetivo que se tenía al construir este tipo de sistemas era emular la forma de resolver problemas de un experto humano en un área específica de conocimiento.

Todavía se siguen utilizando de manera indistinta las dos denominaciones, pero el término sistemas expertos se suele asociar más a sistemas contruidos a partir de conocimiento de expertos humanos, basados fundamentalmente en sistemas de reglas de producción y suelen ser sistemas cerrados donde, en el dominio del problema, el aprendizaje y la adaptación no son prioritarios. El término sistemas

basados en el conocimiento pretende ser mas general, incluyendo cualquier sistema en el que se utilice conocimiento independientemente de su procedencia (procesos de ingeniería del conocimiento, aprendizaje automático) y la metodología o arquitectura de diseño que se utilice (razonamiento basado en reglas, razonamiento basado en casos, modelos cualitativos, agentes inteligentes, redes neuronales, ...), las aplicaciones pueden incluir la necesidad de adaptación y aprendizaje.

12.2 Características de los SBC

Los sistemas basados en el conocimiento están pensados para la resolución de problemas complejos en los que las técnicas de software convencionales no son aplicables.

Dado el tipo de problemas con los que se tendrán que enfrentar, hay un conjunto características que se considera que un sistema de este tipo debería tener y que le diferencian de los sistemas de software tradicionales. Estas características son:

- Flexibilidad, ya que los tipo de problemas que se pueden presentar en un dominio pueden ser bastante variados
- Emular un comportamiento racional, ya que uno de los objetivos es utilizar los mecanismos de resolución que usan los expertos humanos tanto para elaborar la solución como para explicar sus razones.
- Operar en un entorno rico y con mucha información, ya que este tipo de problemas involucra grandes cantidades de conocimiento, con naturaleza bastante diversa y con características especiales, como la incertidumbre o la incompletitud.
- Uso de información simbólica, ya que las decisiones que toman los expertos humanos se basan principalmente en apreciaciones sobre la información descritas a partir de símbolos y no información numérica sin interpretar.
- Uso del lenguaje natural, ya que la forma de interactuar y de recibir respuesta del sistema debería ser lo mas cercana al experto que está emulando.
- Capacidad de aprender, ya que puede ser necesario incorporar nuevo conocimiento para extender las capacidades del sistema a partir de la observación y la experiencia.

Para obtener sistemas que tengan las características mencionadas se ha de tener en cuenta una serie de restricciones y características a la hora de diseñarlos, de manera que deberemos:

1. Separar el conocimiento y el control de la resolución

Ésta es una característica inspirada en la forma en la que los expertos humanos trabajan y es importante sobre todo por la gran cantidad de conocimiento que interviene en la resolución de un problema. Un experto tiene, por un lado, conocimiento específico sobre los elementos que forman parte de su dominio y, por otro, conocimiento sobre las estrategias de resolución de problemas válidas en el dominio que le permiten obtener soluciones a problemas.

Esta separación además incluye un conjunto de ventajas:

- Naturalidad para expresar el conocimiento, ya que su representación es independiente de los métodos de resolución.
- Modularización del conocimiento, ya que se puede agrupar por afinidad, permitiendo un desarrollo más sencillo y comprensible.

- Extensibilidad, ya que es más sencillo ampliar el conocimiento del sistema si podemos estructurarlo de manera independiente.
- Modificabilidad, ya que es más fácil detectar errores y modificar el conocimiento, permitiendo probar cada módulo de manera independiente del mecanismo de resolución.
- Independencia, ya que es posible utilizar diferentes estrategias de resolución para un mismo conocimiento.

2. Incorporar conocimiento heurístico

El conocimiento que se introduce en un SBC está basado en el conocimiento de un experto. Este conocimiento es de naturaleza heurística (incompleto, aproximado, no sistemático) en contraste con el conocimiento algorítmico, en el que tenemos a priori determinado el curso de la resolución y el resultado esperado. Esta resolución basada en el conocimiento experto hace que se parezca más a como los expertos humanos resuelven problemas. Ésta es también una diferencia fundamental respecto a los sistemas software convencionales en los que no existe esta componente heurística.

3. Permitir Interactividad a diferentes niveles

Este tipo de sistemas necesitarán interactuar tanto con los expertos humanos, como con otros sistemas basados en el conocimiento o como con su entorno. Esta interacción ayudará en el curso de la resolución, obteniendo la información que el sistema encuentre que es necesaria para continuar. A la vez, también deberá ser capaz de explicar su línea de razonamiento y revelar el porqué de las elecciones que realiza durante la resolución. Esto último juega un doble papel, primero el justificar las decisiones tomadas (permite dar confianza en los resultados) y segundo permitir el detectar problemas en el conocimiento que tiene el sistema.

Todas estas características hacen de los sistemas basados en el conocimiento un paradigma de la combinación de las diferentes áreas de la Inteligencia Artificial. En estos sistemas necesitaremos:

- **Representación del conocimiento:** Este conocimiento incluirá tanto la descripción de las características del dominio, como el conocimiento de resolución de problemas, la representación del control de la resolución, las heurísticas, ...
- **Razonamiento e inferencia:** La combinación de los datos del problema para llegar a la resolución se realizará como un proceso de razonamiento. Este razonamiento no solo se basará en la lógica clásica, sino que tendrá que utilizar otro tipo de lógicas que permitan, por ejemplo, el razonamiento bajo premisas incompletas, razonamiento ante incertidumbre en el conocimiento y en los pasos de resolución, razonamiento temporal, sobre otros, ...
- **Búsqueda y resolución de problemas:** Las técnicas de resolución basadas en razonamiento tendrán que complementarse con técnicas de búsqueda heurística.
- **Interacción con el usuario:** Mediante lenguaje natural para preguntar, generar explicaciones.
- **Aprendizaje:** Para adquirir el conocimiento del dominio de forma automática, adquisición de nuevo conocimiento o capacidades, aprendizaje de los errores, ...

12.3 Necesidad de los SBC

La principal razón del desarrollo de sistemas basados en el conocimiento viene directamente del interés en automatizar la resolución de problemas que requieren un conocimiento especializado. Este

interés viene del alto coste que supone acceder a personal experto ya que por lo general es escaso y su formación es cara. Detallando las razones, podemos decir que la necesidad proviene de:

- Poder disponer del conocimiento de expertos altamente cualificados. Los expertos son caros y no siempre los hay disponibles. La formación de nuevos expertos solo sirve a largo plazo. Es vital disponer de un sistema que posea el conocimiento del experto y permita usarlo.
- Poder ayudar a otros expertos/no expertos. A veces es suficiente con tener personas que puedan utilizar estos sistemas como soporte a sus decisiones o estos sistemas pueden servir para entrenar y completar la formación de otros expertos.
- Poder preservar el conocimiento de los expertos. El poder preservar de alguna manera el conocimiento que han adquirido los expertos es importante. Por un lado por el conocimiento en sí, ya que puede seguir utilizándose para tomar decisiones, por otro, por que ese conocimiento ayudará a otros expertos en su formación.
- Poder combinar el conocimiento de varios expertos. En la construcción de estos sistemas se combina el conocimiento de diferentes fuentes de información, entre ellas, grupos de expertos. De esta manera el sistema tiene una visión más amplia y se beneficia de la combinación de conocimientos.
- Poder disponer de sistemas que permitan dar soluciones rápidas y justificadas. La automatización del proceso de resolución permite que la respuesta a los problemas pueda ser más rápida. El proceso de razonamiento se puede estudiar y justificar de una manera más fiable.
- Poder tratar grandes volúmenes de información. En muchas áreas el volumen de información involucrada en los problemas a solucionar supera la capacidad de los expertos, de manera que la automatización del proceso es necesaria para su resolución efectiva.
- Poder disponer de sistemas que tomen decisiones autónomas. Ciertos problemas no necesitan constantemente una supervisión por parte de expertos y algunas decisiones se pueden dejar al sistema para reducir su dependencia.

12.4 Problemas que se pueden resolver mediante SBC

No cualquier problema es susceptible de ser resuelto mediante un SBC, antes de comenzar a trabajar nos hemos de plantear ciertas preguntas:

- ¿La necesidad de la solución justifica el desarrollo? El desarrollo de un SBC tiene un coste importante tanto económicamente, como en tiempo. Se ha de adquirir el conocimiento del experto, se ha de construir el sistema, se ha de validar. El problema ha de tener entidad suficiente.
- ¿Es posible acceder al experto necesario? Si es difícil acceder a un experto cuando es necesario, siempre es más sencillo tener disponible un sistema capaz de ayudar a otras personas no tan expertas.
- ¿El problema puede plantearse como un proceso de razonamiento simbólico? Los SBC aplicarán métodos de razonamiento simbólico para solucionar el problema, si no puede plantearse en esos términos no tiene sentido desarrollar un SBC para el problema.

- ¿El problema está bien estructurado? Para que sea posible tratar el problema hemos de poder identificar sus elementos y hemos de poder formalizar tanto el conocimiento necesario, como los procedimientos de resolución. De hecho, el proceso de construcción del SBC requerirá una formalización de todos los elementos del problema.
- ¿Puede resolverse el problema por métodos tradicionales? Si el problema tiene una solución algorítmica o matemática y no involucra ningún proceso de razonamiento cualitativo o heurístico es innecesario desarrollar un SBC para resolverlo.
- ¿Existen expertos disponibles y cooperativos? El construir este tipo de sistemas necesita la colaboración completa de expertos en el problema. Estos han de comprender la tarea y el objetivo del SBC y cooperar en la labor de formalización del conocimiento. Al ser el conocimiento el elemento fundamental del sistema, la dificultad de acceder a él hace imposible la labor de desarrollo.
- ¿Está el problema bien dimensionado? Esta restricción es clave en cualquier proyecto informático. La complejidad de la tarea no ha de ser tal que se necesiten formalizar grandes cantidades de conocimiento para resolverlo, ha de ser suficientemente restringido para ser manejable y poderse terminar la tarea con éxito.

12.5 Problemas de los SBC

Existen todo un conjunto de problemas asociados a los SBC, algunos son inherentes a sus características, otros pueden depender de las limitaciones que imponen el coste de su desarrollo. Podemos enumerar los siguientes:

Fragilidad: Su capacidad se degrada bruscamente cuando los problemas que se le plantean están fuera de su ámbito de experiencia. En el caso de un experto humano, éste podría dar respuestas mas o menos ajustadas a pesar de que los problemas vayan saliendo de su ámbito de conocimiento, estos sistemas simplemente serán incapaces de dar una respuesta.

Dificultades con el control del razonamiento: Los expertos, además del conocimiento del dominio, tienen conocimiento sobre como plantear y estructurar la resolución de un problema. Este conocimiento, denominado *metaconocimiento* es difícil de explicitar y suele ir mezclado con el conocimiento del dominio. De la posibilidad de codificar este conocimiento en el sistema dependerá la eficiencia a la hora de encontrar soluciones.

Baja reusabilidad de las bases de conocimiento: Por lo general el conocimiento de un problema depende fuertemente del dominio, sobre todo la parte de resolución. No obstante ciertas partes del conocimiento mas general puede ser reaprovechado como ontologías de dominio.

Es difícil integrar el aprendizaje: Muchas veces estos sistemas no incluyen capacidad de aprendizaje por la complejidad de integrar nuevo conocimiento con el ya existente en el sistema. No obstante hay tareas en las que la capacidad de aprendizaje es esencial para la resolución.

Problemas en la adquisición del conocimiento: Obtener tanto la descripción del dominio, como el conocimiento de resolución es el principal cuello de botella de los SBC. Esto es debido a la dificultad que tienen los expertos en explicitar su conocimiento. Se ha denominado *paradoja del experto* a la observación de que, cuanto más experiencia tiene una persona en un problema, más difícil le es explicar como lo resuelve.

Problemática de la validación: El proceso de validación de una base de conocimiento es muy costoso. El mayor problema reside en la posibilidad de que haya inconsistencias en el sistema¹. También es muy costoso probar la completitud del sistema.

12.6 Áreas de aplicación de los SBC

Los sistemas basados en el conocimiento se aplican a gran variedad de áreas. Se pueden encontrar en cualquier dominio en el que se necesite un conocimiento especializado. Áreas tan diferentes como la medicina, la biología, el diseño, la concesión de créditos o la predicción meteorológica se cuentan entre sus aplicaciones.

Su sofisticación depende del tipo de problema concreto que se desee resolver, pero básicamente se aplican en dos tareas genéricas, el análisis de un problema para identificar su naturaleza o la construcción de una solución a partir de sus elementos esenciales.

Problemas que involucran identificación pueden ser por ejemplo la interpretación de los elementos de un problema, el diagnóstico a partir de unos síntomas, la supervisión o control de un proceso para mantenerlo dentro de los parámetros correctos o la predicción de resultados a partir de evidencias.

Problemas que involucran la construcción de una solución pueden ser por ejemplo la planificación de un conjunto de tareas, el diseño a partir de un conjunto de elementos y restricciones o la configuración de un conjunto de elementos.

12.7 Breve historia de los SBC

La historia de los SBC comienza con la construcción de los primeros sistemas expertos desarrollados durante la década de 1960. Estos sistemas intentaban resolver tareas medianamente complejas dentro de un ámbito bastante específico. En esa época los diferentes lenguajes y entornos para la construcción de estos sistemas estaban poco desarrollados, por lo que estos sistemas eran esfuerzos bastante individuales y muchos de ellos tenían la intención de probar la viabilidad de la construcción de sistemas capaces de resolver problemas complejos.

El primer sistema experto fue DENDRAL, desarrollado en 1965. El problema que se quería resolver era el de la identificación de la estructura de moléculas orgánicas a través de espectrografía de masas. En esa época este tipo de problema necesitaba a una persona con bastante experiencia y llevaba un tiempo considerable. Este sistema fue capaz de reducir drásticamente el tiempo necesario para resolver el problema permitiendo su automatización. Se puede considerar que la mayoría de los sistemas que se construyeron en la siguiente década derivan de este trabajo.

Otro hito en la historia de los sistemas expertos fue MYCIN (1970). Se encuadraba en el área de la medicina y era capaz de identificar infecciones en sangre. Su principal aportación fue la introducción de tratamiento del conocimiento incierto mediante el sistema denominado de factores de certeza. De este trabajo surgió EMYCIN que es considerado el primer entorno de desarrollo de sistemas expertos.

El sistema PROSPECTOR (1974) trataba el dominio de las prospecciones minerales. Su mérito consiste en haber introducido un nuevo método para el tratamiento de la incertidumbre² que es el origen de algunos de los métodos actuales.

El primer sistema experto de un tamaño respetable que tuvo éxito comercial fue XCON (1980), con alrededor de 2500 reglas de producción. El sistema asistía a la compra de sistemas de computación VAX (de la desaparecida compañía Digital Equipment Corporation³) configurándolos según

¹Probar que un conjunto de axiomas es inconsistente es semidecidible.

²Dice la leyenda que el sistema superó a los expertos prediciendo vetas de mineral no descubiertas por los expertos.

³Digital Equipment Corporation fue una de las empresas pioneras en la fabricación de ordenadores, fue absorbida

las necesidades del cliente. Este sistema evitaba errores en los envíos de los sistemas asegurándose de que todos los elementos requeridos fueran enviados reduciendo las posibilidades de error, lo cual ahorró una buena cantidad de dinero a la compañía.

La década de 1980 marca el inicio de la carrera de la construcción de sistemas expertos en prácticamente cualquier dominio de aplicación. Su capacidad para resolver problemas tanto de diagnóstico, como de clasificación, monitorización, supervisión, predicción, diseño, ... han hecho que su construcción se cuente por miles.

Durante la década de 1990 estos sistemas se han ido diversificando también en las metodologías aplicadas para su construcción pasando de ser construidos básicamente mediante sistemas de producción a utilizar técnicas provenientes de la inteligencia artificial distribuida como los *agentes inteligentes*, usar técnicas de *razonamiento basado en casos* (Case Based Reasoning) o técnicas de aprendizaje automático como las *redes neuronales*. Esta diversificación ha generalizado su concepción y se ha rebautizado a estos sistemas como sistemas basados en el conocimiento (Knowledge Based Systems).

También se han desarrollado nuevas técnicas para el tratamiento de la incertidumbre que se han incorporado en estos sistemas como por ejemplo las redes bayesianas y el razonamiento difuso. El aprendizaje está empezando a tener gran importancia como por ejemplo en la fase de adquisición del conocimiento, de manera que se pueden construir sistemas que resuelvan problemas a partir de ejemplos presentados por los expertos y el sistema puede construir su programación a partir de ellos.

En la actualidad los sistemas basados en el conocimiento desempeñan multitud de tareas tanto especializadas, como cotidianas.

13.1 Introducción

En este capítulo vamos a describir los diferentes elementos de los que se compone la arquitectura de un sistema basado en el conocimiento. Esta arquitectura puede variar dependiendo de las técnicas de inteligencia artificial que se utilicen, pero existen un conjunto de bloques generales que aparecerán siempre. La arquitectura de estos sistemas está pensada para poder cumplir las capacidades que se esperan de un sistema basado en el conocimiento, principalmente:

- Que sea capaz de resolver problemas a partir de información simbólica
- Que aplique métodos de razonamiento y resolución heurísticos
- Que sea capaz de explicar los resultados y el proceso de razonamiento
- Que sea capaz de interactuar con el entorno y pueda ser interrogado

Para cumplir estas características necesitamos al menos los siguientes elementos en un SBC:

- Un subsistema de *almacenamiento del conocimiento*
- Un subsistema de *uso e interpretación del conocimiento*
- Un subsistema de *almacenamiento del estado del problema*
- Un subsistema de *justificación e inspección de las soluciones*
- Un interfaz de *comunicación con el entorno y/o el usuario*

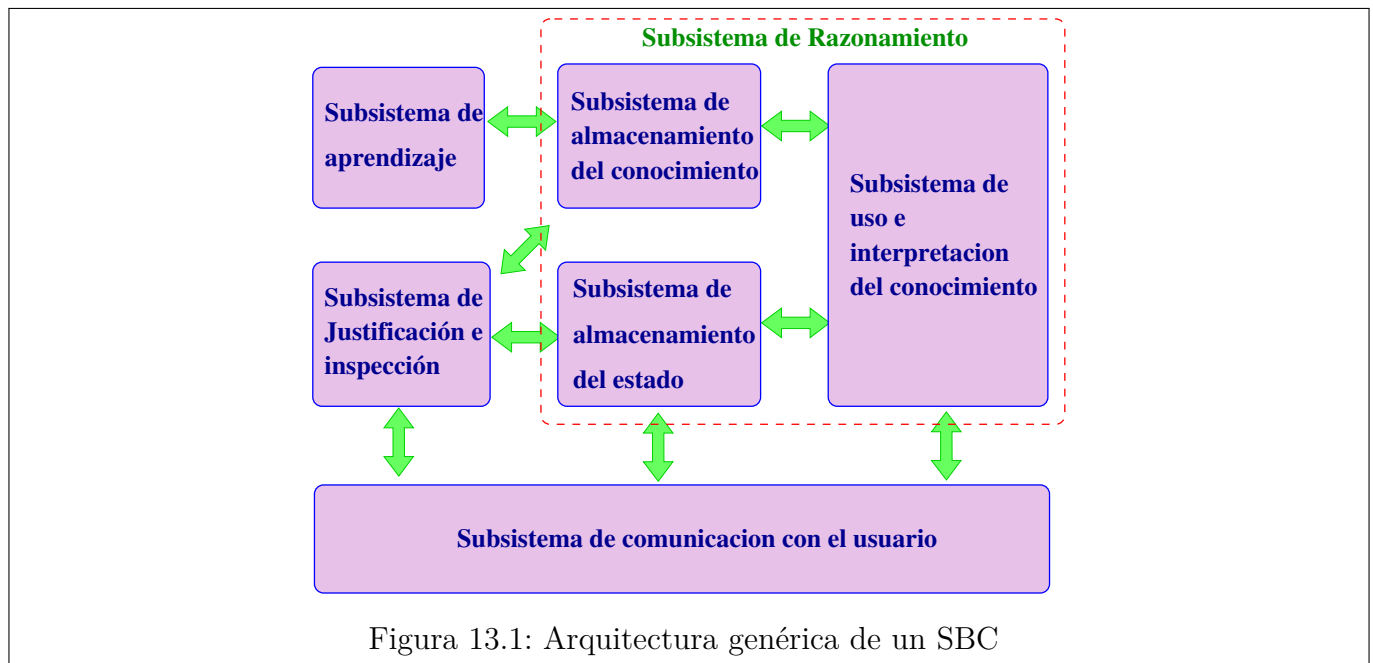
Adicionalmente se puede incorporar capacidad de aprendizaje mediante observación del entorno, que daría lugar a la necesidad de un subsistema de *aprendizaje*.

En la figura 13.1 se puede ver una representación de sus relaciones.

Los tres primeros subsistemas se pueden agrupar en el *subsistema de razonamiento*, que será el encargado de construir la solución del problema.

El conocimiento y la forma en la que funcionan estos subsistemas dependen bastante de las técnicas que se usen para la representación del conocimiento y la resolución de problemas. Nosotros nos centraremos en dos tipos:

- Sistemas basados en reglas de producción (Rule Based Reasoning)
- Sistemas basados en casos (Case Based Reasoning)



13.2 Arquitectura de los sistemas basados en reglas

Los sistemas basados en reglas utilizan como elemento principal a los sistemas de reglas de producción. La resolución de un problema se realiza mediante el proceso de razonamiento generado por el ciclo de funcionamiento de un motor de inferencia. El conocimiento del dominio está expresado como una ontología de dominio y el conocimiento de resolución está representado como un conjunto de reglas de producción.

13.2.1 Almacenamiento del conocimiento

Este subsistema contiene el conocimiento necesario para resolver problemas en el dominio del SBC y se habrá construido a partir de un proceso previo de extracción de conocimiento. También se le conoce como *base de conocimiento*.

El conocimiento almacenado es de naturalezas diferentes y puede por lo tanto estar representado utilizando distintos formalismos. Centrados en los sistemas basados en reglas de producción tendremos tres tipos de conocimiento en una base de conocimiento:

1. Conocimiento factual (objetos del dominio y sus características)
2. Conocimiento relacional (relaciones entre los objetos del dominio, por ejemplo jerárquicas)
3. Conocimiento condicional (reglas de producción que expresan conocimiento deductivo sobre el dominio)

Los dos primeros conocimientos suelen estar integrados en una ontología de dominio y habitualmente están representados mediante formalismos estructurados como frames o redes semánticas.

El tercer conocimiento es el más voluminoso y mas importante desde el punto de vista de la resolución de problemas. Está representado habitualmente mediante un formalismo de reglas de producción, aunque existen otros formalismos. La complejidad y expresividad de estas reglas dependerá del dominio, ya que puede ser necesario trabajar con formalismos lógicos que traten sus características particulares. Puede ser suficiente con un formalismo basado en lógica de primer orden, necesitar hacer tratamiento de la incertidumbre, trabajar con conocimiento incompleto, razonamiento temporal, ...

La expresividad del formalismo de reglas de producción nos permitirá representar conocimiento de diferente naturaleza que puede ser necesario para un dominio, principalmente:

- **Conocimiento deductivo (estructural):** Este conocimiento nos permitirá describir los procesos de resolución de problemas del dominio a partir de cadenas de deducción.
- **Conocimiento sobre objetivos (estratégico):** Este conocimiento orientará la resolución del problema y permitirá plantear la forma de resolver problemas a alto nivel.
- **Conocimiento causal (de soporte):** Este conocimiento permite añadir la posibilidad de realizar explicaciones sobre los procesos de deducción llevados a cabo para solucionar un problema o interrogar al sistema sobre la consecuencia de suposiciones (what if)

Habitualmente los lenguajes de reglas que se utilizan para implementar este conocimiento permiten organizarlo de manera estructurada para facilitar el desarrollo. Entre las facilidades mas comunes se encuentra la división del conocimiento en **módulos**. Estos permiten agrupar las reglas en bloques de alto nivel que tengan sentido en el dominio del problema.

Esta posibilidad de estructurar las reglas permite una mejor organización y encapsulamiento del conocimiento, facilitando el desarrollo, prueba y depuración.

Precisamente esta organización en módulos de las reglas permite la expresión del conocimiento sobre objetivos ya mencionado. Este conocimiento se describe mediante reglas denominadas **meta-reglas**. Estas reglas describen conocimiento de alto nivel que no se refiere a problemas específicos del dominio, sino a estrategias o metodologías de resolución de problemas propias del dominio. Mediante las meta-reglas se pueden desarrollar mecanismos de resolución mas potentes de lo que permiten las reglas que expresan conocimiento deductivo y reducir el coste computacional de la resolución de problemas.

El mayor problema de las meta-reglas es que son mas difíciles de adquirir a partir del experto ya que representan conocimiento de un nivel mayor de abstracción. Las meta-reglas se pueden clasificar dependiendo de su nivel de intervención en el proceso de resolución:

Meta-reglas sobre reglas: Se usan para activar o desactivar reglas dentro de un módulo o dar prioridad a las reglas que se activan

Metarreglas sobre módulos: Afectan a la forma en la que se hace la búsqueda dentro del módulo, a la forma de tratamiento de la incertidumbre, al tipo de conclusiones, ...

Metarreglas sobre estrategias: Deciden sobre la ordenación de activación de los módulos de reglas, tratamiento de excepciones, ...

Metarreglas sobre planes de actuación: Cambian las diferentes estrategias de resolución de problemas de alto nivel que pueda haber

13.2.2 Uso e interpretación del conocimiento

Si la implementación del SBC se realiza utilizando reglas de producción este subsistema es habitualmente un *motor de inferencia*. Ya conocemos el funcionamiento de un motor de inferencia del tema de sistemas de reglas de producción.

Este subsistema utilizará la base de conocimiento y la información del problema particular a resolver, para obtener nuevas inferencias que lleven a plantear los pasos necesarios para llegar a la solución.

La complejidad del motor de inferencia dependerá de la expresividad de las reglas de la base de conocimiento. Este tendrá que interpretar las condiciones de las reglas, realizar las instanciaciones

adecuadas y escoger la regla a aplicar en cada paso de la resolución. Esta elección dependerá directamente de la estrategia de resolución de conflictos que tenga implementada o del conocimiento de control expresado mediante las meta-reglas.

Al ser el motor de inferencia el mecanismo básico de resolución del sistema la capacidad y eficiencia en la resolución de problemas dependerá totalmente de él.

13.2.3 Almacenamiento del estado del problema

A este subsistema se le conoce como *memoria de trabajo*. Su complejidad dependerá de la expresividad del mecanismo de resolución. Habitualmente almacena los datos iniciales del problema descritos mediante los objetos de la ontología del dominio y las deducciones intermedias que se van obteniendo durante el proceso de resolución.

En sistemas mas complejos la memoria de trabajo puede almacenar otras cosas, como información de control que apoye al mecanismo de resolución (orden de deducción de los hechos, preferencias sobre hechos, reglas, módulos, reglas activadas recientemente, caminos alternativos, ...)

13.2.4 Justificación e inspección de las soluciones

Para que un sistema sea creíble ha de poder justificar y mostrar sus decisiones, para ello el experto ha de ser capaz de añadir a las reglas la justificación de su aplicación y de las condiciones que tienen. También puede incluir reglas que permitan interrogar al sistema sobre las deducciones que se han obtenido y las cadenas de deducción que han llevado a ellas.

Esta justificación se apoyará en la información incluida por el experto en el sistema y en la traza de la resolución que deberá almacenar el motor de inferencia en la memoria de trabajo para su posterior inspección.

Dos preguntas típicas que debería poder responder un SBC en cualquier punto de la resolución son:

- **Porqué:** Que objetivos globales se desean resolver.
- **Cómo:** Que cadena de razonamiento ha llevado a ese punto.

Podemos distinguir dos niveles de explicación:

Muestra: Traza que describe los pasos que se han realizado en la cadena de razonamiento, las reglas que se han utilizado y los hechos deducidos.

Justificación: Razones por las que se ha escogido una línea de razonamiento, porqué se preguntan ciertas cosas al usuario, tipo de subproblema en el que se ha clasificado la solución, razones de las preferencias utilizadas, ...

Las explicaciones pueden consistir en textos prefijados añadidos a las reglas o pueden generarse explicaciones en lenguaje natural que dependan del contexto.

13.2.5 Aprendizaje

Habitualmente las aplicaciones que se crean para solucionar problemas en un dominio están acotadas y delimitan el conjunto de problemas que se pueden resolver. Esto es adecuado en muchos dominios, pero otros requieren que el sistema sea capaz de aprender a resolver nuevos problemas o adaptar su comportamiento al entorno.

Hay diversas maneras de abordar el problema del aprendizaje en un SBC. Si partimos de un sistema basado en reglas, una posibilidad es el poder crear o corregir reglas del sistema cuando se detectan fallos en las soluciones. Para ello es necesario un proceso de razonamiento que detecte en qué parte de la solución aparece el problema. Otra posibilidad es que el sistema sea capaz de crear sus propias reglas de control analizando las trazas de las resoluciones que ha obtenido. En estas trazas se puede detectar en qué puntos el mecanismo de resolución perdió el objetivo y fue necesario replantear la estrategia de resolución. Creando nuevas reglas de control se puede evitar esta pérdida de tiempo en futuras resoluciones. Este tipo de aprendizaje se conoce como *aprendizaje basado en explicaciones*.

Otras posibilidades de aprendizaje aparecen en dominios en los que es difícil obtener directamente reglas de producción por la complejidad de la tarea a resolver. En estos dominios se intenta aprender un modelo de la tarea a resolver. Este modelo por lo general se construye por observación de ejemplos de soluciones de la tarea. A partir de estos ejemplos se construye una generalización que permitirá resolver los tipos de problemas que describen los ejemplos. Este modelo puede consistir en un conjunto de reglas que describe la tarea u otro formalismo equivalente. Las técnicas utilizadas caen dentro de lo que se denomina *aprendizaje inductivo*.

13.3 Arquitectura de los sistemas basados en casos

El *razonamiento basado en casos* (*Case Based Reasoning*) supone que la solución de un problema se puede obtener identificando una solución anterior similar y adaptándola a las características particulares del problema.

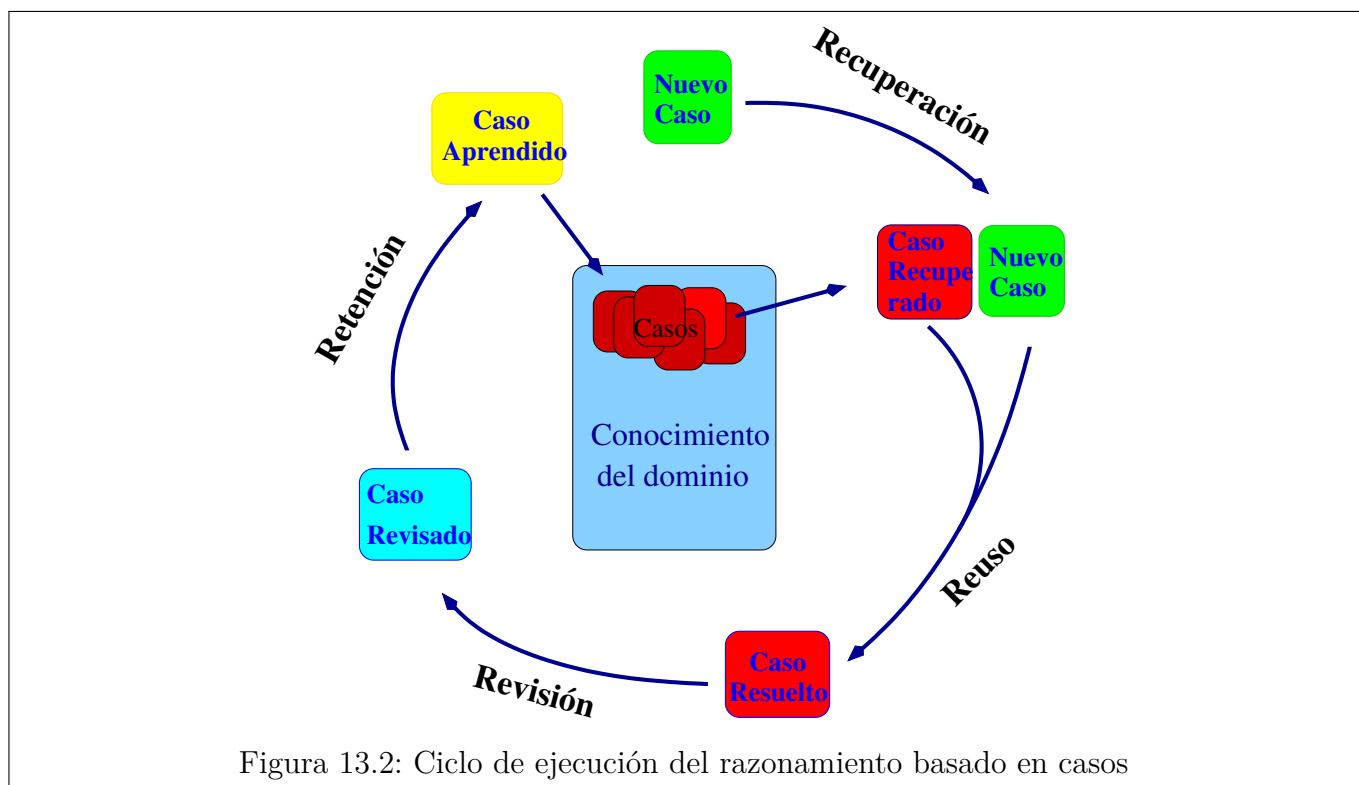
Este método de resolución está apoyado por estudios en psicología cognitiva que indican que muchas veces esa es la manera en la que personas expertas en un problema llegan a obtener sus soluciones. De este modo, la necesidad de extraer conocimiento encaminado a la resolución se reduciría en gran medida y esta se centraría en la identificación y adaptación de casos similares.

Existen muchos problemas que pueden ser resueltos mediante esta metodología, presentando algunas ventajas:

- Reducen en gran medida el proceso de explicitación y extracción del conocimiento y, sobre todo, reducen la necesidad de formalizarlo mediante un formalismo procedimental, por ejemplo, reglas de producción.
- Mejoran la capacidad de mantenimiento del conocimiento al permitir añadir o corregir fácilmente los casos
- Hacen más eficiente la resolución de problemas al reducir la necesidad de utilizar mecanismos de razonamiento
- Es cercano a la experiencia de los usuarios, ya que permite la explicación de las soluciones a partir de casos conocidos

El ciclo de resolución de los sistemas basados en casos es diferente del de los sistemas de producción. Estos se basan en un ciclo que incluye cuatro fases:

1. **Fase de recuperación:** A partir del problema se deben identificar los casos almacenados que tengan mayor similitud
2. **Fase de reuso:** Escogemos como solución inicial la que aparece en el caso recuperado



3. **Fase de revisión:** Evaluamos la solución recuperada y la adaptamos al caso particular, evaluando su resultado. Esta revisión puede necesitar un proceso de razonamiento.
4. **Fase de retención:** Evaluamos si es útil guardar la información de la nueva solución obtenida para poder solucionar nuevos problemas.

La figura 13.2 muestra las fases gráficamente. Este ciclo de resolución encaja con los elementos de la arquitectura general que hemos visto.

13.3.1 Almacenamiento del conocimiento

En esta arquitectura el sistema de almacenamiento del conocimiento tendrá dos elementos. Por un lado tendremos el sistema que almacenará los casos, denominado *base de casos*. Esta almacenará un conjunto de soluciones que se hayan escogido como representativas y que se podrán usar en resoluciones de nuevos problemas. Por otro lado estará el conocimiento específico del dominio que por lo general se expresa a partir de reglas o procedimientos. Estos representarán la información necesaria para determinar la similaridad de los casos, la adaptación de la solución recuperada al caso actual y la evaluación de la solución.

La información que describe un caso puede ser bastante compleja. Será no solo necesaria información descriptiva que permita la recuperación de los casos almacenados a partir de casos nuevos, sino también información sobre la solución o justificaciones que permitan explicar la solución.

Otro punto importante será la organización de la base de casos para la recuperación de los casos similares. La eficiencia computacional exige que exista alguna organización que ayude a la recuperación. Por lo general esta organización es jerárquica y tiene algún tipo de indexación. Hay que tener en cuenta que la recuperación no tiene por que basarse directamente en las características que describen el caso, sino que puede ser necesario cierto tipo de cálculo o razonamiento para evaluar la similaridad. También puede haber cierta noción de distancia que determine cómo de similares son los casos que se recuperan.

13.3.2 Uso e interpretación del conocimiento

El mecanismo de uso del conocimiento se basa directamente en el ciclo de funcionamiento de un sistema basado en casos.

El sistema será capaz de identificar el conjunto de casos más similares al problema actual utilizando la base de casos. Este proceso de recuperación dependerá de la información que contengan los casos y la estructura en la que esté organizada la base de casos. Se recorrerá esta estructura evaluando la similaridad de los casos con el caso actual de la manera que esté definida. Por lo general se utilizará una función de similaridad o un proceso de razonamiento sencillo para puntuar los casos.

La solución se construirá a partir del caso mas similar o la combinación de soluciones de los casos más similares. El sistema deberá instanciar la solución obtenida para el caso actual y comprobar si es válida tal como está. Esto puede requerir otro proceso de razonamiento.

Si la solución no es válida se puede utilizar el conocimiento del dominio del que se dispone para intentar adaptar la solución. Durante esta adaptación el conocimiento del dominio deberá determinar que elementos son incorrectos en la solución y proponer alternativas.

El resultado de este proceso será una solución completa.

13.3.3 Almacenamiento del estado del problema

En este caso el estado del problema está formado únicamente por el caso actual, el caso recuperado y la información que se infiera en los procesos de recuperación del caso mas similar, la evaluación de la solución y su adaptación.

13.3.4 Justificación e inspección de las soluciones

La justificación en los sistemas de razonamiento basado en casos está almacenada en los propios casos. Esta información deberá ser complementada por el proceso de razonamiento de la adaptación cuando se necesite adaptar la solución.

En estos sistemas la justificación es más natural ya que la solución se ha definido a priori a partir de un caso concreto que es mas sencillo de entender por un usuario.

13.3.5 Aprendizaje

El proceso de aprendizaje en estos sistemas es mas sencillo y natural, ya que simplemente se ha de añadir a la base de casos el nuevo caso y su solución. La ventaja respecto a los sistemas basados en reglas es que no se ha de tocar el conocimiento que ya tiene el sistema.

La decisión de añadir un nuevo caso debe ser evaluada por el sistema teniendo en cuenta lo similar que sea a casos ya existentes. El objetivo es completar la visión que se tiene del dominio, así que nuevos casos suficientemente diferentes de los conocidos son importantes.

También es posible plantearse la posibilidad de olvidar casos que tengan poca utilidad eliminándolos de la base de casos.

13.4 Comunicación con el entorno y/o el usuario

Dada la necesidad de interacción del sistema con el usuario y/o el entorno deberá existir un interfaz de comunicación. Este puede ser relativamente simple (preguntas cerradas, menús) o permitir una

interacción mas natural con el usuario (interfaz en lenguaje natural).

Debería permitir:

- Introducir los datos del problema a resolver
- Preguntar sobre el estado interno: Sobre la resolución (explicaciones, reglas utilizadas, hechos), sobre suposiciones alternativas (what if), sobre el estado de la memoria de trabajo
- Preguntar al usuario: sobre hechos, petición de confirmaciones, ...

En sistemas complejos un SBC no tiene por que interactuar solamente con usuarios humanos sino que puede colaborar con otros SBC para realizar su tarea. Esto hace necesario establecer primitivas de comunicación a través de las cuales se puedan hacer preguntas. Existen lenguajes de comunicación entre SBC estandarizados que definen las comunicaciones que se pueden realizar y su semántica. Estos lenguajes suponen que existe una ontología de dominio común entre los SBC que interactúan que permiten que el contenido de las comunicaciones sea comprendido por ambas partes.

13.5 Otras Metodologías

Los sistemas basados en reglas y el razonamiento basado en casos son arquitecturas habituales sistemas basados en el conocimiento. No obstante existen otras aproximaciones que a veces se adaptan mejor a los problemas que se pretenden resolver.

Comentaremos sucintamente algunas de ellas.

13.5.1 Redes neuronales

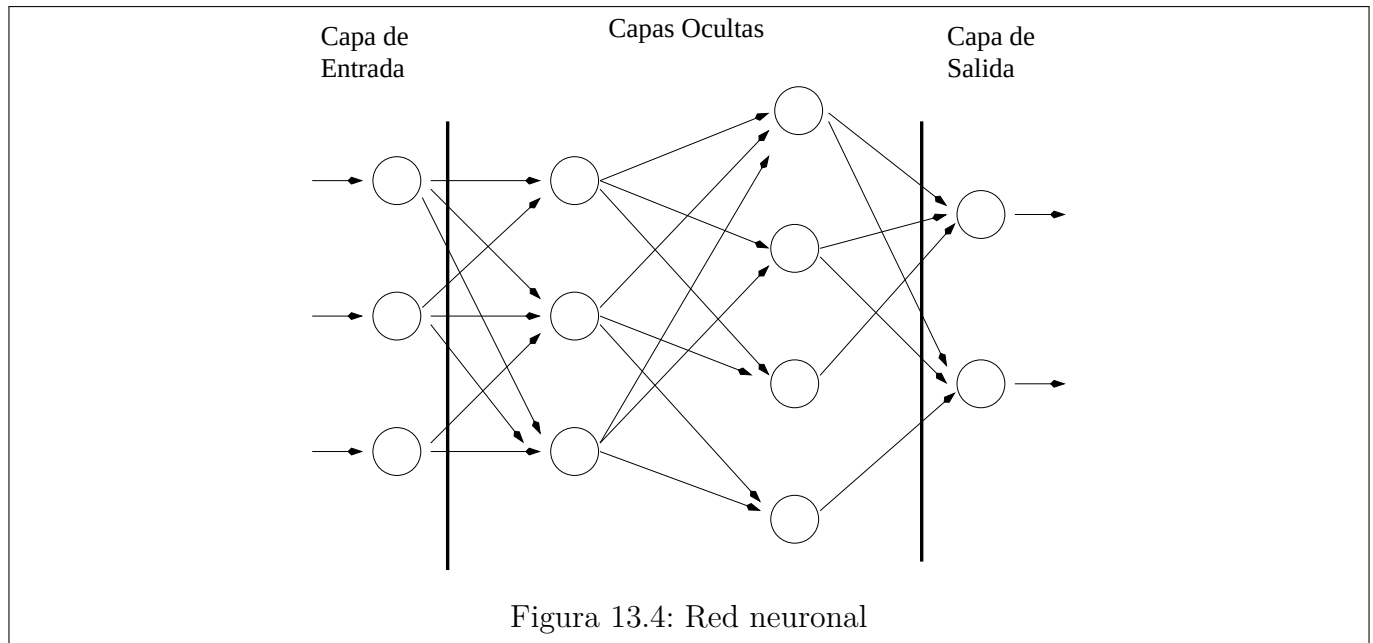
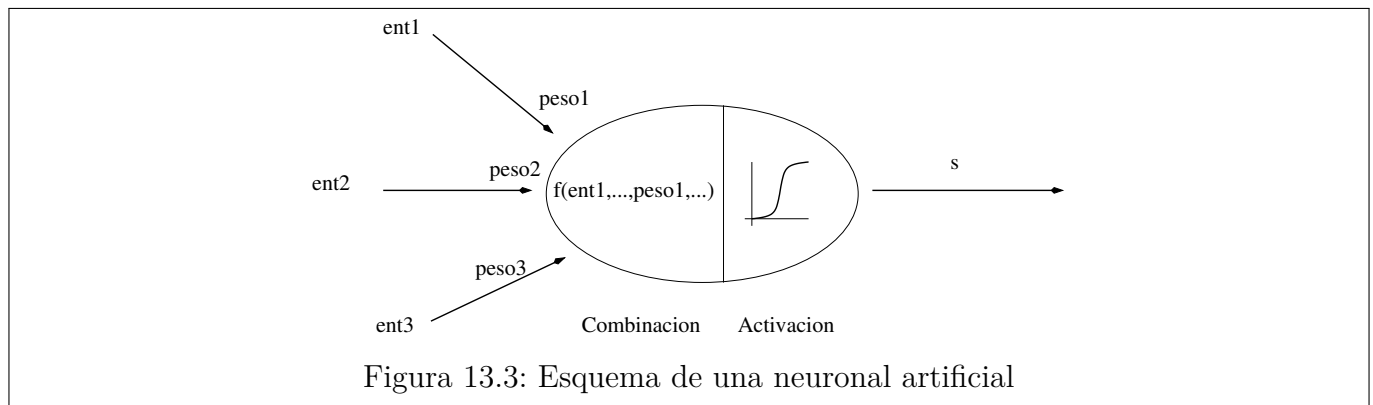
Las redes neuronales (o redes de neuronas artificiales) forman parte de una visión no simbólica de la inteligencia artificial y entran dentro del denominado modelo conexionista. Las arquitecturas descritas hasta ahora basan su funcionamiento en una descripción explícita del conocimiento y de sus mecanismos de resolución. El modelo conexionista se basa en la utilización de conjuntos de elementos simples de computación denominados neuronas (ver figura 13.3).

Las neuronas se describen a partir de unas entradas, una salida, un estado y un conjunto de funciones que se encargan de combinar las entradas, cambiar el estado de la neurona y dar el valor de salida. El funcionamiento de una neurona es muy sencillo, los valores de la entrada y el estado de la neurona se combinan y se obtiene un valor de salida, esta salida sería la respuesta de la neurona a los estímulos de entrada. La interconexión de estos elementos entre si permite realizar cálculos complejos.

Las neuronas se agrupan en lo que se denomina redes de neuronas (ver figura 13.4). Estas redes están organizadas en capas interconectadas. Por lo general se distingue **la capa de entrada**, que es la que recibe la información del exterior, **la capa de salida**, que es la que da la respuesta y **las capas ocultas**, que son las que realizan los cálculos.

Las redes neuronales asocian un conjunto de entradas con un conjunto de respuestas. Por lo general las entradas correspondientes a la capa de entrada suelen corresponder con la información del problema y los valores que corresponden a las salidas de la capa de salida codifican la respuesta al problema.

En este caso para construir un sistema basado en el conocimiento a partir de redes neuronales se debe decidir como se han de codificar los datos de entrada y la salida, el número de neuronas en cada una de las capas, el número de capas ocultas y la interconexión entre cada capa de neuronas.



La asociación entre entradas y salidas se realiza mediante técnicas de aprendizaje automático. A la red de neuronas se le van presentando ejemplos solucionados y, mediante algoritmos adecuados, las neuronas de la red van quedando en un estado que describe la asociación entre problemas y soluciones. Este periodo de aprendizaje, también llamado entrenamiento es la parte mas compleja y requiere escoger adecuadamente el conjunto de ejemplos y el número de veces que se presentan a la red.

Los principales inconvenientes de las redes neuronales son el no disponer de una descripción explícita de la resolución y en consecuencia no poder dar una justificación en términos que pueda comprender un experto. No obstante, los métodos conexionistas se usan con gran éxito en muchas aplicaciones.

13.5.2 Razonamiento basado en modelos

El *razonamiento basado en modelos*, junto al razonamiento cualitativo, pretende resolver problemas a partir de construir un modelo del sistema sobre el que se ha de trabajar. A diferencia de los modelos utilizados en simulación, que se basan fundamentalmente en información y métodos numéricos, estos modelos serían adecuados para problemas en los que estos métodos no son aplicables por la complejidad del sistema.

La descripción del sistema se realizaría a partir de información cualitativa y reglas y ecuaciones cualitativas que describirían sus relaciones. Este modelo permitiría obtener deducciones sobre las

consecuencias de las acciones que se pueden realizar en el problema y razonar sobre el comportamiento del sistema.

Este tipo de modelado se acercaría a lo que se considera razonamiento del sentido común y que permitiría obtener soluciones a problemas sin tener que entrar la complejidad del funcionamiento real del problema.

Un ejemplo de este tipo de metodología es la denominada *física naïf* (*naïve physics*), en la que se pretende incluir información del sentido común en el razonamiento sobre los fenómenos físicos.

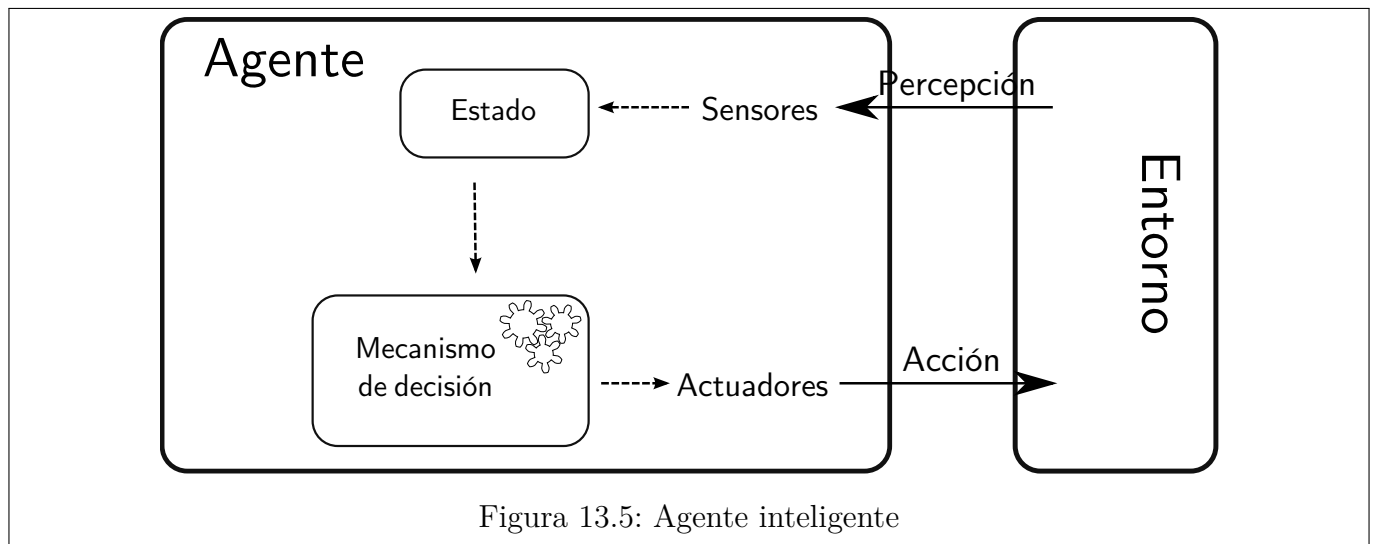


Figura 13.5: Agente inteligente

13.5.3 Agentes Inteligentes/Sistemas multiagente

La visión de todas las metodologías mencionadas de los sistemas basados en el conocimiento es una visión monolítica. La tarea a realizar es muy compleja y existe un único sistema que tiene todo el conocimiento y realiza todas las acciones.

El área de agentes inteligentes intenta ampliar esta visión, considerando que tener múltiples agentes que resuelve problemas más sencillos y que se encuentran en un entorno en colaboración/competición entre sí, permite también resolver esos problemas complejos. Podríamos decir que las otras arquitecturas tendrían una visión de agente único y el área de agentes inteligentes tendría una visión de colectividad.

Un agente sería un sistema que obtiene información del entorno (percepción) elabora una decisión basada en estas percepciones y su representación interna del estado (razonamiento) y realiza una acción mediante sus actuadores (actuación) (ver figura 13.5). El mecanismo de decisión es el elemento más complejo y se puede resolver de múltiples maneras, dando así más o menos capacidades al agente.

Los agentes inteligentes pueden considerarse más como un paradigma de desarrollo de sistemas basados en el conocimiento que una arquitectura en sí, ya que las metodologías utilizadas para construir los agentes individuales son las que ya se han descrito.

La diferencia esencial es la división de un trabajo complejo entre pequeños sistemas que tienen capacidad de razonamiento y que para resolver el problema global deben comunicarse y coordinarse. Es una visión social de la inteligencia que podemos ver en la resolución de muchos problemas.

Ejemplo 13.1 *Un ejemplo evidente de sociedad de agentes es el comportamiento de los insectos sociales. Cada individuo es relativamente simple y tiene una serie de capacidades que le permite resolver un conjunto de tareas, la organización de estos y su división del trabajo les permite llegar a resolver problemas complejos.*

Esta visión social hace necesario incluir técnicas que no están presentes en las otras metodologías, entre ellas:

- Organización y estructura de la comunidad de agentes (reglas, normas, prohibiciones)
- Asignación y compartición de recursos, resolución de conflictos y negociación
- Cooperación, división del trabajo, planificación multiagente
- Lenguajes de comunicación

- Razonamiento sobre los otros agentes, introspección

Los agentes inteligentes aportan una flexibilidad al desarrollo de sistemas basados en el conocimiento que no permiten otras metodologías. Cada uno de los elementos que forma parte de la comunidad se puede reorganizar para resolver otros problemas, de manera que se puede reaprovechar lo desarrollado a la manera de la programación a partir de componentes, pero con la ventaja de la capacidad de razonamiento que incluye cada agente. Esto se puede llevar todavía mas allá si es el propio agente el que busca y recluta a otros agentes que le permitan resolver partes de su problema que él por sí solo no puede resolver.

El desarrollo de internet está íntimamente relacionado con los agentes inteligentes y tiene áreas de interés común con por ejemplo los sistemas Grid o los servicios web.

14.1 Ingeniería de sistemas basados en el conocimiento

El punto clave del desarrollo de un sistema basado en el conocimiento es el momento de traspasar el conocimiento que posee el experto a un sistema real. En este proceso no sólo se han de captar los elementos que componen el dominio del experto, sino que también se han de adquirir las metodologías de resolución que utilizan éstos.

Este trabajo de extracción del conocimiento se realiza durante la interacción entre dos personajes, el *ingeniero del conocimiento* (persona que conoce el formalismo de representación que utilizará el SBC) y el experto (persona que posee el conocimiento, pero que no tiene por que usar un formalismo para representarlo).

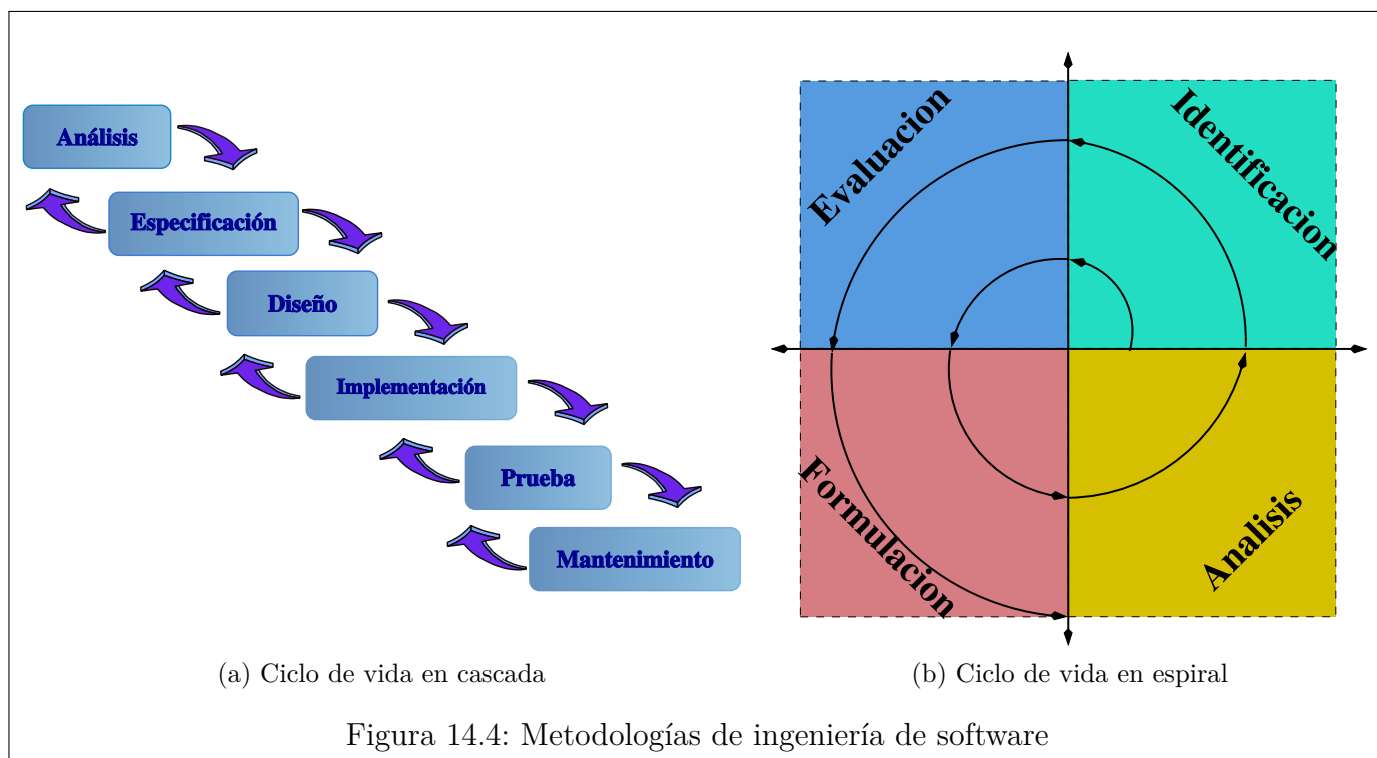
Todo este proceso ha de encajarse en las metodologías de desarrollo de software y adaptarse a las características específicas que tienen los sistemas basados en el conocimiento.

Como cualquier otro desarrollo de software el desarrollo de sistemas basados en el conocimiento puede hacerse siguiendo las metodologías existentes de ingeniería de software, adaptándolas a las diferentes particularidades que estos tienen. En las siguientes secciones describiremos brevemente diferentes metodologías de ingeniería de software que se pueden utilizar.

14.1.1 Modelo en cascada

Esta metodología es utilizable en desarrollos de tamaño pequeño o mediano. Se divide en seis fases que describiremos brevemente:

- **Análisis del problema:** Se determina la factibilidad de resolver el problema a partir de la información disponible y su viabilidad económica.
- **Especificación de requerimientos:** Se redacta un documento de especificación que describe los objetivos y los elementos que deberá poseer el sistema.
- **Diseño:** Se determina la arquitectura del sistema desde el nivel de mayor abstracción hasta el nivel más específico, describiendo las interacciones entre los diferentes elementos que formarán el sistema
- **Implementación:** Se programará cada uno de los módulos del sistema, integrándolos en un único sistema.
- **Prueba:** Se determinará si el sistema implementado cumple con las especificaciones iniciales



- **Mantenimiento:** Se corregirán los posibles errores que se descubran durante el funcionamiento del sistema y se añadirán las modificaciones necesarias para cumplir nuevas capacidades que se pidan al sistema

Las desventajas de esta metodología son que ni el desarrollador, ni el usuario final pueden hacerse una idea del resultado final hasta que el sistema sea completado y que el usuario final no puede utilizar ningún elemento del sistema hasta el final de proyecto.

14.1.2 Modelo en espiral

Este modelo combina las ideas del modelo en cascada con las de prototipado y análisis de riesgos, viendo el desarrollo como la repetición de un ciclo de pasos. Estos pasos son:

- **Identificación:** Se determinan los objetivos del ciclo, las alternativas para cumplirlos y las diferentes restricciones de estas alternativas.
- **Evaluación:** Se examinan los objetivos y restricciones para descubrir posibles incertidumbres y riesgos.
- **Formulación:** Se desarrolla una estrategia para resolver las incertidumbres y riesgos.
- **Análisis:** Se evalúa lo realizado para determinar los riesgos remanentes y se decide si se debe continuar con el elemento actual o se puede pasar al siguiente elemento o paso de desarrollo.

El punto clave de esta metodología es saber identificar lo antes posible los riesgos involucrados en el desarrollo para que estos no tengan un impacto negativo en el proceso.

14.1.3 Diferencias de los sistemas basados en el conocimiento

Al igual que otros sistemas software, los sistemas basados en el conocimiento tiene el objetivo de crear soluciones computacionales a problemas, por ello el desarrollo de estos sistemas es parecido al del software convencional. Sin embargo, hay diferencias que hay que tener en cuenta.

La primera de ellas es el tipo de conocimiento que se representa. En los sistemas software habituales se implementan procedimientos algorítmicos bien conocidos y de uso común, al contrario que en los sistemas basados en el conocimiento, que involucran conocimiento incompleto, impreciso y de naturaleza heurística que es conocido únicamente por un limitado conjunto de expertos. Este conocimiento debe ser transferido de estos expertos a una solución software, este proceso es denominado **adquisición del conocimiento** (*Knowledge Elicitation*).

Por lo general, el sistema que utiliza el ingeniero del conocimiento para obtener el conocimiento del experto es el de entrevistas. Durante estas entrevistas el ingeniero ha de ayudar al experto a sistematizarlo, consiguiendo que vaya explicitando el conocimiento del dominio y las diferentes técnicas que utiliza para resolver los problemas que deberíamos incluir en nuestro sistema. Con esto pretendemos transformar este conocimiento de manera que se puedan representar en un formalismo computable. Este proceso de extracción del conocimiento es bastante lento y costoso.

Varias son las dificultades que hacen que este proceso sea tan costoso:

- La naturaleza especializada del dominio hace que el ingeniero del conocimiento deba aprender unas nociones básicas de éste para que pueda establecerse una comunicación efectiva con el experto (vocabulario básico, elementos que intervienen en el dominio, formalismos que utilizan los expertos, etc.).
- La búsqueda de un formalismo de representación que se adapte adecuadamente al problema y que sea fácil de interpretar y adoptar por el experto. Este formalismo ha de ser susceptible de ser transformado en algo computable.
- Los expertos se encuentran más cómodos pensando en términos de ejemplos típicos que razonando en términos generales, que son de los que realmente se podría hacer una mejor abstracción.
- Por lo general, a los expertos les es muy difícil explicitar los pasos que utilizan para resolver los problemas. Es lo que se ha denominado *paradoja del experto*. Cuanta más experiencia, menos explícitos son los razonamientos del experto y más ocultos los métodos de resolución.

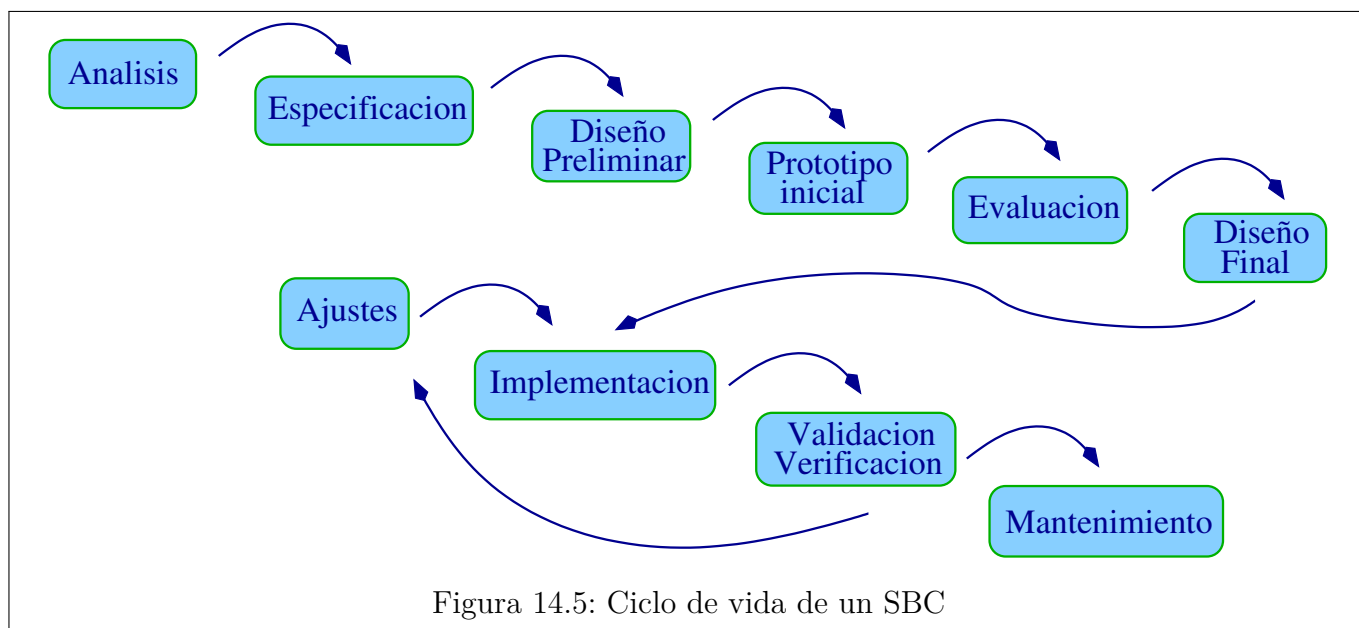
Si observamos como un experto resuelve un problema, éste omite muchas cadenas de razonamiento e información que da por supuestas, y a las que no asigna importancia dentro de la resolución, pero que son necesarias si se quiere abordar el problema de manera sistemática.

Con todas estas circunstancias, podemos observar que la auténtica dificultad de la extracción del conocimiento estriba en descubrir los métodos mediante los que se usa el conocimiento en la resolución y no tanto en la adquisición del conocimiento estático del problema (elementos del problema y relaciones)

Sobre la adquisición de los elementos básicos del dominio, existen bastantes herramientas automáticas, encuadradas dentro del área del aprendizaje automático (*Machine Learning*), que permiten reducir el esfuerzo.

La segunda diferencia es la naturaleza y la cantidad del conocimiento. En los sistemas software tradicionales es posible estimar estas dos cualidades de manera sencilla, en los sistemas basados en el conocimiento esto es mucho más difícil. Incluso para los expertos en el dominio es difícil hacer esta evaluación, lo que complica el calcular el esfuerzo necesario para desarrollar el sistema.

Esta dificultad puede hacer que sea complicado obtener un diseño adecuado en las fases iniciales del proyecto pudiendo suceder que durante el proceso de desarrollo se descubra que alguno de los elementos decididos no son realmente adecuados para resolver el problema. Esto puede llevar a tener que decidir entre continuar el desarrollo sabiendo que ese problema tendrá consecuencias en las fases



restante o empezar de nuevo el proyecto. La manera mas adecuada para minimizar este problema es el uso de técnicas de desarrollo incremental y de prototipado rápido (*rapid prototyping*).

Estas técnicas son bastante populares en el desarrollo de sistemas basados en el conocimiento y están favorecidas por los lenguajes utilizados en este tipo de problemas (PROLOG, LISP, CLIPS, ...) que permiten la construcción rápida de un prototipo funcional del sistema final. Este prototipo recogerá un subconjunto de las funcionalidades del sistema y permitirá observar la idoneidad de las decisiones.

En el ciclo de vida de este tipo de desarrollo, las fases de análisis y especificación deben realizarse teniendo en cuenta el sistema completo, pero el diseño e implementación se realizan de una manera mas preliminar y reduciendo las necesidades. Esto permite disponer de un sistema funcional que puede ser evaluado para obtener una valoración de las decisiones tomadas.

A partir del prototipo inicial se puede aplicar la metodología de desarrollo incremental. Esta metodología se basa en la división de problemas y el desarrollo iterativo, de manera que se va completando el sistema añadiendo nuevos módulos y funcionalidades. Esto permite tener un sistema funcional durante todo el desarrollo.

14.1.4 Ciclo de vida de un sistema basado en el conocimiento

Podemos modificar el ciclo de vida convencional del desarrollo de software para adaptarlo a las necesidades del desarrollo de sistemas basados en el conocimiento, estas serían las fases en las que dividiríamos el proceso (ver figura 14.5):

1. **Análisis del problema:** Se han de evaluar el problema y los recursos disponibles para determinar si es necesario y viable el desarrollo de un sistema basado en el conocimiento para resolverlo.
2. **Especificación de requerimientos:** Formalizar lo que se ha obtenido en la fase de análisis, fijando los objetivos y los medios necesarios para su cumplimiento.
3. **Diseño preliminar:** Se toman decisiones a alto nivel para crear el prototipo inicial. Estas decisiones deben incluir la forma de representar el conocimiento y las herramientas que se van a utilizar para el desarrollo del prototipo. En esta fase se hará necesario obtener información

general sobre el problema a partir de las fuentes disponibles para poder tomar las decisiones adecuadas.

4. **Prototipo inicial y evaluación:** El prototipo inicial debe tener las características de la aplicación completa pero su cobertura del problema debe estar limitada. Este prototipo debe permitirnos poder evaluar las decisiones tomadas. Esto hace necesario obtener una cantidad de conocimiento suficiente para evaluar la forma escogida de representarlo y utilizarlo. El objetivo es poder detectar los errores que hayamos podido cometer en el diseño preliminar para poder corregirlos en esta fase.
5. **Diseño final:** En esta fase debemos tomar las decisiones finales con las que vamos a continuar el desarrollo del sistema, las herramientas y recursos a usar y, sobre todo, la forma de representar el conocimiento. Se debe también hacer un diseño de la estructura de la aplicación que nos permita un desarrollo incremental.
6. **Implementación:** En esta fase debemos completar la adquisición del conocimiento necesaria para desarrollar el sistema completo. En esta fase debemos aplicar la metodología de desarrollo incremental implementando cada uno de los módulos de la aplicación.
7. **Validación y verificación:** Se ha de validar el sistema construido y verificar que cumpla las especificaciones del problema. La validación de estos sistemas es mas compleja que en los sistemas software convencionales dada la naturaleza de los problemas que se resuelven.
8. **Ajustes de diseño:** Puede ser necesaria cierta realimentación del proceso y la revisión y ajuste de algunas decisiones tomadas en sus fases anteriores, pero no debería suponer grandes cambios de diseño, ya que el prototipo inicial nos debería haber permitido eliminar los posibles errores en el diseño inicial.
9. **Mantenimiento:** Esta fase es similar a la que se realizaría en el desarrollo de un sistema de software convencional.

14.1.5 Metodologías especializadas

Se han desarrollado bastantes metodologías para el desarrollo de sistemas basados en el conocimiento, algunas genéricas, otras más especializadas, éstas son un ejemplo:

CommonKADS

Una de las metodologías mas conocidas es KADS (Knowledge Acquisition and Documentation Structuring), cuya evolución se denomina CommonKADS.

CommonKADS utiliza un ciclo de vida en espiral y aborda el desarrollo de sistemas basados en el conocimiento como un proceso de modelado utilizando técnicas y notación parecidas a UML.

El desarrollo se basa en la creación de un conjunto de seis modelos: organización, tareas, agentes, comunicación, conocimiento y diseño. Estos seis modelos están interrelacionados y se desarrollan a partir de un conjunto de plantillas que provee la metodología.

Los tres primeros modelos permiten determinar si el problema puede ser resuelto mediante sistemas basados en el conocimiento. Modelan la estructura de la aplicación, las capacidades que ha de tener y los elementos que son necesarios para la resolución. En el modelo de conocimiento se ha de representar el conocimiento que es necesario para resolver el problema, tanto la descomposición en subproblemas, el conocimiento del dominio (ontología), como los procesos de inferencia involucrados. El modelo de comunicación se encarga de describir la interacción entre los elementos de la aplicación y la información que se intercambia. El último modelo describe el diseño de la aplicación.

MIKE

MIKE (Model-Based and Incremental Knowledge Engineering) es otra metodología también basada en el ciclo de vida en espiral. Este ciclo está compuesto por cuatro fases que se repiten: adquisición del conocimiento, diseño, implementación y evaluación.

La fase de adquisición del conocimiento comienza con un proceso de extracción del conocimiento del que se obtiene un modelo semiformal del conocimiento necesario a partir de entrevistas con los expertos, este modelo es denominado *modelo de adquisición*. A partir de esta descripción semiformal del conocimiento se crea un modelo formal que describe todos los aspectos funcionales y no funcionales del sistema, este modelo es denominado *modelo de estructura*. A partir de este modelo se realiza un proceso de formalización y operacionalización que utiliza un lenguaje especializado (KARL, Knowledge acquisition representation language) que combina una descripción conceptual del sistema con una especificación formal.

Este último modelo es la entrada de la fase de diseño en la que se tienen en cuenta los requisitos no funcionales y se genera un nuevo modelo mediante otro lenguaje especializado (DesignKARL) en el que se describe el sistema de manera más detallada. Este diseño es el que pasará a la fase de implementación en el que se plasmará el sistema. Finalmente se realizará la fase de evaluación.

14.2 Una Metodología sencilla para el desarrollo de SBC

En esta sección describiremos una metodología sencilla para el desarrollo de SBC de tamaño pequeño o mediano, basada en un conjunto de fases en las que realizaremos un conjunto de tareas específicas. Está basado en un ciclo de vida en cascada, pero podríamos fácilmente encajarlo en una metodología basada en prototipado rápido y diseño incremental como el descrito en la sección 14.1.4.

Las fases que consideraremos serán las siguientes:

1. **Identificación del problema:** Determinar la viabilidad de la construcción de un SBC para solucionar el problema y la disponibilidad de fuentes de conocimiento.
2. **Conceptualización:** Descripción semiformal del conocimiento del dominio del problema, descomposición del problema en subproblemas.
3. **Formalización:** Formalización del conocimiento y construcción de las ontologías necesarias, identificación de las metodologías de resolución de problemas adecuadas. Decisión sobre las técnicas de representación y herramientas adecuadas.
4. **Implementación:** Construcción del sistema
5. **Prueba:** Validación del sistema respecto a las especificaciones.

14.2.1 Identificación

En esta fase se ha de determinar, en primer lugar, si el problema se puede o se debe abordar mediante las técnicas de los SBC. Para que un problema sea adecuado no ha de poder solucionarse de manera algorítmica, ya que si se pudiera hacer de ese modo, no tendría sentido iniciar una labor tan costosa. También ha de ser necesario tener acceso a las fuentes de conocimiento suficientes para completar la tarea. Por último, el problema a tratar ha de tener un tamaño adecuado para que no constituya una tarea inabordable por su complejidad.

El siguiente paso consiste en buscar las fuentes de conocimiento que serán necesarias para el desarrollo del sistema, las más comunes son:

- Expertos humanos en el dominio del problema.
- Libros y manuales que expliciten el problema y técnicas de resolución.
- Ejemplos de casos resueltos.

Estos últimos serán importantes sobre todo en la última fase de validación, pero se pueden usar también para utilizar técnicas de adquisición automática del conocimiento y obtener de esta manera los elementos básicos que intervienen y sus relaciones.

Con estas fuentes de información se podrán determinar los datos necesarios para la resolución del problema y los criterios que determinen la solución, tanto los pasos que permiten la resolución como su posterior evaluación.

En este momento el ingeniero del conocimiento y el experto podrán realizar una primera descripción del problema, en ésta se especificarán:

- Los objetivos
- Las motivaciones
- Las estrategias de resolución y su justificación
- Las fuentes de conocimiento
- Los tipos de tareas que son necesarias

Este esquema será el punto de partida para plantear las siguientes fases.

14.2.2 Conceptualización

Esta fase pretende obtener una visión del problema desde el punto de vista del experto. Ha de poder permitirnos decidir el conocimiento que es necesario para resolver el sistema y las tareas involucradas para obtener una idea informal del alcance del sistema.

Necesitamos conocer cuales son los conceptos que ha de manejar el sistema y sus características de cara a poder elaborar en la fase siguiente una ontología que los formalice. Hemos de identificar también sus características y necesidades de inferencia para poder elegir el método de representación del conocimiento mas adecuado. Esto significa que deberemos aplicar técnicas de desarrollo de ontologías como un proceso paralelo al desarrollo del sistema basado en el conocimiento.

Hay también que obtener una descomposición del problema en subproblemas, realizando un análisis por refinamientos sucesivos hasta que nos podamos hacer una idea de la relación jerárquica de las diferentes fases de resolución hasta los operadores de razonamiento más elementales.

Otro elemento necesario es descubrir el flujo del razonamiento en la resolución del problema y especificar cuando y como son necesarios los elementos de conocimiento.

Con esta descomposición jerárquica y el flujo del razonamiento se pueden caracterizar los bloques de razonamiento superiores y los principales conceptos que definen el problema. Hará falta distinguir entre evidencias, hipótesis y acciones necesarias en cada uno de los bloques y determinar la dificultad de cada una de las subtareas de resolución. De esta manera se conseguirá captar la estructura del dominio y las diferentes relaciones entre sus elementos.

La manera habitual de realizar estas tareas es la interacción con el experto. En particular, es necesario observar como resuelve problemas típicos y abstrae de ellos principios generales que pueden ser aplicados en diferentes contextos.

Toda esta labor debería darnos un modelo semiformal del dominio y de los problemas y métodos de resolución que se deberán incluir en el sistema.

14.2.3 Formalización

Esta fase ha de darnos una visión del problema desde el punto de vista del ingeniero del conocimiento. Se han de considerar los diferentes esquemas de razonamiento que se pueden utilizar para modelizar las diferentes necesidades de representación del conocimiento y de resolución de problemas identificadas en las fases anteriores. Se deberá decidir el mecanismo de representación adecuado y formalizar el conocimiento de manera que podamos crear la ontología que represente el conocimiento del dominio. En esta fase se detallarán los elementos que forman parte de la ontología y se iniciará el proceso de su formalización.

Deberemos decidir también la manera más adecuada de representar los procesos de resolución identificados. En este punto, se ha de poder comprender la naturaleza del espacio de búsqueda y el tipo de búsqueda que habrá que hacer. En este momento se pueden comparar los métodos de resolución que aplica el experto en un problema con diferentes mecanismos prototípicos de resolución de problemas como la clasificación, abstracción de datos, razonamiento temporal, estructuras causales, etc. De esta manera decidiremos cuales son los mas adecuados en nuestras circunstancias.

En esta etapa también tendrá que analizarse la certidumbre y completitud de la información disponible, dependencias temporales, o la fiabilidad y consistencia de la información. Se deberá descubrir qué partes del conocimiento constituyen hechos seguros y cuales no. Para estos últimos deberá elegirse alguna metodología de tratamiento de la incertidumbre, de manera que ésta pueda ser modelizada dentro del sistema.

14.2.4 Implementación

En esta fase ya se conocerán la formas más adecuadas para la representación del conocimiento y los métodos que se adaptan a los mecanismos de resolución de problemas. Ahora se podrán decidir las herramientas y lenguajes más adecuados para llevar a cabo la implementación.

Durante esta fase se implementará la ontología que ha de servir de base para la resolución del problema en el formalismo de representación del conocimiento elegido. Decidiremos también como encajar la descomposición de subproblemas de las tareas involucradas en el sistema con las metodologías de resolución de problemas genéricas que habremos identificado en la fase anterior.

Una vez validadas o corregidas las decisiones que se han tomado continuaremos con la construcción del sistema añadiendo incrementalmente nuevos módulos y nuevas funcionalidades. Si procedemos con una metodología basada en prototipado rápido y desarrollo incremental, deberemos construir un prototipo inicial a partir de una restricción de las capacidades del sistema. Con las decisiones iniciales podremos construir el conjunto de módulos necesarios para obtener un prototipo inicial capaz de resolver un subconjunto de problemas y plantearnos las fases del desarrollo incremental.

14.2.5 Prueba

Se ha de elegir un conjunto de casos resueltos representativos y se ha de comprobar el funcionamiento del sistema con estos. Estos casos deberían incluir los utilizados para el diseño del sistema, pero también casos nuevos.

La validación de un sistema basado en el conocimiento es más compleja que en otros sistemas software, dado que estamos implementado resoluciones de problemas de naturaleza heurística, por lo que es muy difícil determinar la completitud y correctitud del sistema. Esto obligará a realizar futuros mantenimientos o ampliaciones de la aplicación a medida que se descubran casos en los que el sistema no funcione correctamente. Estos fallos pueden deberse no solo a errores en la implementación o el diseño, sino también a problemas en la formalización del conocimiento y de los métodos de resolución.

Resolución de problemas en los SBC

15.1 Clasificación de los SBC

El abordar la construcción de un SBC en cualquier dominio es una tarea difícil, y sería deseable disponer de un conjunto de metodologías de resolución de problemas que permitieran aproximar soluciones a diferentes tipos de SBC según sus características.

Con esta idea en mente se han realizado clasificaciones de los SBC según las tareas que realizan, para intentar descubrir metodologías comunes y así extraer directrices de análisis en los distintos tipos de dominios.

De esta manera, dada una clase de problema dispondríamos de:

1. Un conjunto de tareas usuales para cada tipo fáciles de identificar.
2. Un conjunto de metodologías de resolución de problemas específicas para cada tipo.
3. Métodos de representación del conocimiento e inferencia adecuados para cada tipo.

Originalmente se realizó una primera clasificación de los SBC atendiendo a las tareas que realizan¹, ésta es:

Sistemas de Interpretación: Infieren descripciones de situaciones a partir de observaciones.

Sistemas de predicción: Infieren consecuencias previsibles de situaciones o eventos.

Sistemas de diagnóstico: Infieren fallos a partir de síntomas.

Sistemas de diseño: Desarrollan configuraciones de objetos que satisfacen ciertas restricciones.

Sistemas de planificación: Generan secuencias de acciones que obtienen un objetivo.

Sistemas de monitorización: Estudian el comportamiento de un sistema en el tiempo y procuran que siga unas especificaciones.

Sistemas de corrección: Genera soluciones para fallos en un sistema.

Sistemas de control: Gobiernan el comportamiento de un sistema anticipando problemas, planeando soluciones.

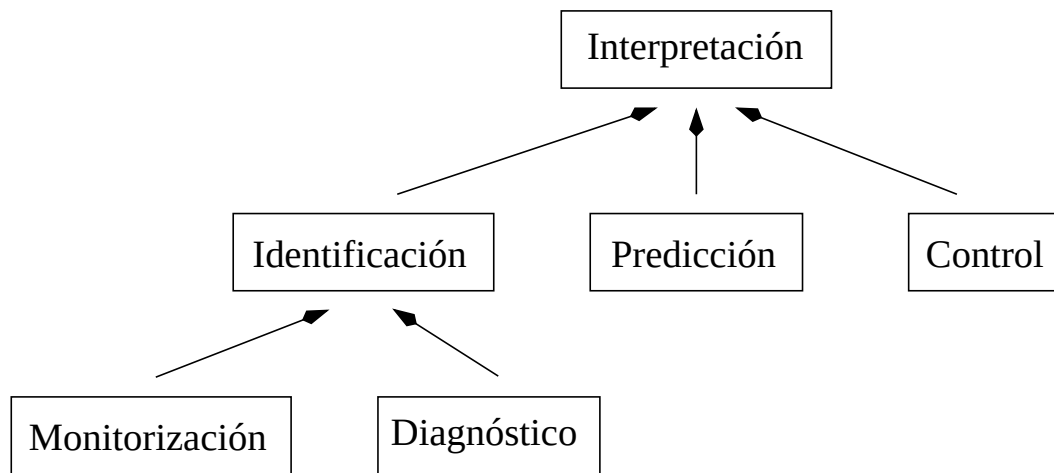
¹Esta clasificación apareció en: F. Hayes-Roth, D. A. Waterman, D. B. Lenat, *Building Expert Systems*, Addison Wesley, Reading, MA, 1983.

Esta primera clasificación, que es la que se utilizó como punto de partida para la identificación de necesidades para el desarrollo de SBC, plantea varios problemas ya que varias categorías se superponen o están incluidas en otras. No obstante, da una idea inicial de los rasgos comunes que aparecen entre los distintos dominios en los que tratan los sistemas.

Un análisis alternativo, posterior a éste, permite un tratamiento más sistemático de las necesidades de un SBC, éste se basa en las operaciones genéricas que puede hacer un SBC respecto al entorno. Se distinguen dos operaciones genéricas:

- Operaciones de análisis, que interpretan un sistema.
- Operaciones de síntesis, que construyen un sistema.

Estas operaciones se pueden especializar en otras más concretas dando lugar a una jerarquía de operaciones. Para el caso del **análisis** tenemos:



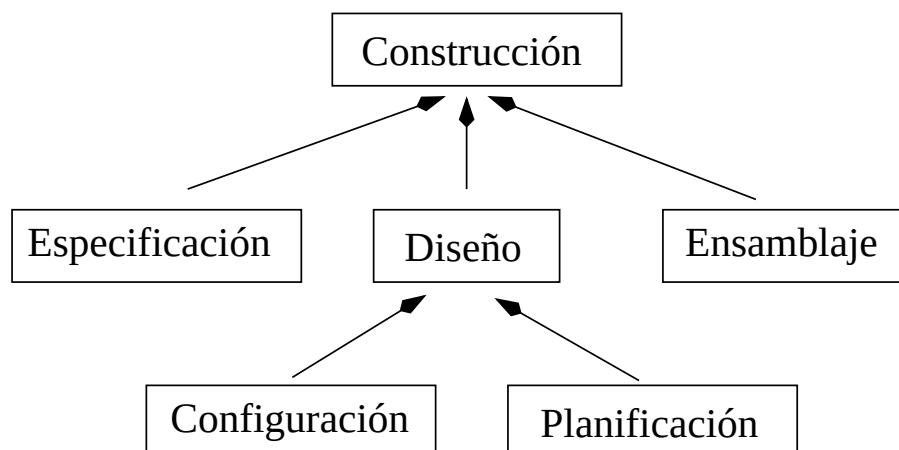
En este caso, la *interpretación* se podría especializar según la relación entre los elementos de entrada/salida de un sistema:

- Identificación, nos dice que tipo de sistema tenemos.
- Predicción, nos dice que tipo de resultado podemos esperar.
- Control, determina que entradas permiten conseguir la salida deseada.

La *identificación* se puede especializar para sistemas con fallos en:

- Monitorización, detecta discrepancias de comportamiento.
- Diagnóstico, explica discrepancias.

Para el caso de las operaciones de *síntesis* tenemos:



La especialización de la *construcción* se puede realizar en:

- Especificación, busca que restricciones debe satisfacer un sistema.
- Diseño, genera una configuración de elementos que satisfacen las restricciones.
 - Configuración, como es la estructura actual del sistema.
 - Planificación, pasos a realizar para ensamblar la estructura.
- Ensamblaje, construye un sistema juntando las diferentes piezas.

Obteniendo una clasificación de las diferentes tareas y operaciones que realiza un SBC podemos establecer una correspondencia entre estos y los métodos de resolución, y de esta manera facilitar la tarea de análisis de los dominios.

15.2 Métodos de resolución de problemas

Diferentes son las técnicas de resolución de problemas que se pueden utilizar para las tareas que debe realizar un SBC. Existen ciertas técnicas generales que se pueden aplicar a diferentes tipos de dominios y tareas. De ellas destacaremos las dos más utilizadas:

- Clasificación Heurística (*Heuristic Classification*)
- Resolución Constructiva (*Constructive Problem Solving*)

15.2.1 Clasificación Heurística

La clasificación es un método utilizado en muchos dominios. El elemento esencial de ésta consiste en que el experto escoge una categoría de un conjunto de soluciones previamente enumerado.

En dominios simples, el disponer de las características esenciales de cada una de las categorías es suficiente para establecer la clase del problema y su solución. Esto no ocurre así cuando la complejidad del problema aumenta, pues las características esenciales son cada vez más difíciles de identificar. El objetivo de la técnica de clasificación heurística será obtener y representar el conocimiento necesario para que la asociación problema-solución se pueda realizar.

Se define como *clasificación heurística* a toda asociación no jerárquica entre datos y categorías que requiere de inferencias intermedias. Es decir, el establecer la clase de un problema requiere realizar inferencias y transformaciones sobre éste, para poder asociarlo con la descripción de la clase. El esquema de razonamiento para hacer estas inferencias se ha de adquirir del experto.

La clasificación heurística se divide en tres etapas:

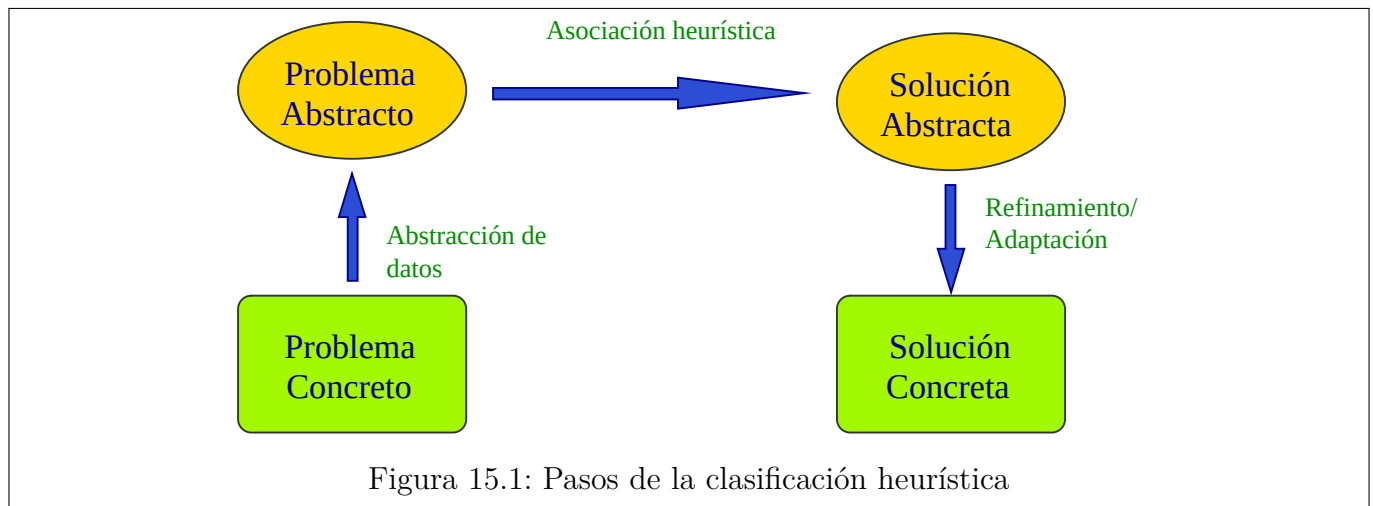
1. Abstracción de los datos

Por lo general, se hace una abstracción del caso concreto para acercarlo a las soluciones que se poseen.

2. Asociación heurística

Se busca la mayor coincidencia entre el caso abstraído y las soluciones. Esta asociación es de naturaleza heurística, es decir, depende de conocimiento basado en la experiencia, y, por lo general, la correspondencia entre caso y soluciones no será uno a uno, existirán excepciones, y las coincidencias no serán exactas.

La solución corresponderá con la que mejor coincida con la abstracción de los datos.



3. Refinamiento de la solución

Haber identificado la abstracción de la solución reducirá el espacio de búsqueda, ahora será necesario buscar la mejor solución determinada por la solución abstracta. Esto puede necesitar de más deducciones, o de la utilización de más información. De esta manera se debe reducir el espacio de búsqueda hasta encontrar la mejor solución.

En la figura 15.1 se puede ver un esquema del proceso.

Dentro de este proceso, un punto importante es la abstracción de los datos. Tres son las más utilizadas:

Abstracción definicional: Se deben extraer las características definitorias del problema y focalizar la búsqueda con éstas. Le corresponde al experto decidir cuales son esas características.

Cualitativa: Supone abstraer sobre valores cuantitativos, convirtiéndolos en cualitativos (e.g.: Fiebre = 39°C $\Rightarrow$ Fiebre = alta).

Generalización: Se realiza abstracción sobre una jerarquía de conceptos (e.g.: forma = pentágono $\Rightarrow$ forma = polígono).

Se puede ver que esta metodología de resolución de problemas capta una gran cantidad de dominios, siendo adecuada para cualquier problema en el que se pueda hacer una enumeración del espacio de soluciones. Es válida para todas las tareas de análisis.

Clasificación heurística en los sistemas de reglas

Por lo general, la construcción de un sistema mediante clasificación heurística basado en reglas es una labor iterativa. A los expertos a veces les es difícil dar las reglas que son capaces de realizar la labor de clasificación, y además encuentran difícil el formalismo de las reglas.

El proceso de refinamiento del sistema ha de hacerse paso a paso, añadiendo nuevas reglas que cubran nuevos casos y vigilando las interacciones. La metodología que se suele seguir es la siguiente:

1. El experto da las nuevas reglas al IC.
2. El IC cambia la base de conocimiento.
3. El IC prueba casos ya resueltos para comprobar inconsistencias.
4. Si aparecen errores, se comprueba el nuevo conocimiento con el experto y se empieza de nuevo.

5. Se prueban nuevos casos.
6. Si no hay problemas se para, sino se retorna al principio.

Esta labor iterativa se puede realizar de manera independiente para cada uno de los módulos que componen el sistema, reduciendo de esta manera las interacciones entre diferentes partes del conocimiento.

Estrategias de adquisición del conocimiento con clasificación heurística

La aplicación de la clasificación heurística a diferentes problemas ha llevado a métodos que permiten dirigir la explicitación del conocimiento por parte del experto de una manera más sistemática, enfocando la labor de extracción en cada uno de los elementos que componen las reglas (hipótesis, síntomas, causas, cadenas de inferencia, hechos intermedios, confianza en las evidencias y las asociaciones evidencia-conclusión).

El conjunto de conceptos del problema se puede dividir en tres:

- **Las hipótesis:** Soluciones posibles a nuestro problema.
- **Los síntomas:** Características que describen las hipótesis
- **Las causas originales:** Información del problema que lleva a los síntomas

Las causas y los síntomas se relacionarán mediante las *reglas de abstracción*. De los datos originales obtendremos los valores para el conjunto de características que describen a los problemas. Los síntomas y las hipótesis se relacionarán mediante las *reglas de asociación heurística*. De las características que tiene el problema a solucionar deberemos identificar las hipótesis más probables.

En cada conjunto de reglas (reglas de abstracción, reglas de asociación heurística) deberemos observar qué antecedentes están asociados con que consecuentes. El objetivo es escoger aquellos antecedentes que permitan distinguir mejor los consecuentes, ya que estos pueden tener características comunes. El objetivo es escribir las reglas que permitan diferenciar mejor a los consecuentes.

En este proceso es posible que sea necesario introducir conceptos nuevos (**conceptos intermedios**) que pueden determinar una cadena de deducciones entre las premisas originales y las conclusiones. Estas cadenas de deducciones pueden ser complejas, dependiendo del problema.

Durante el proceso de construcción de las reglas podemos observar también la certidumbre de esas asociaciones (confianza con la que los antecedentes permiten deducir los consecuentes), de manera que podamos hacer el tratamiento adecuado.

En el caso de las soluciones, si estas corresponden a abstracciones será necesario determinar las reglas que permiten especializarlas en soluciones concretas.

Aplicación de la clasificación heurística

Como ejemplo de la técnica de clasificación heurística, vamos a plantear un pequeño SBC para la concesión de créditos bancarios para creación de empresas. El propósito de este sistema será examinar las solicitudes de créditos de clientes con pretensiones de crear una empresa, para determinar si se les debe conceder y que cuantía es la recomendable respecto a la que solicitan.

El problema que se nos plantea tiene por lo tanto una labor de análisis que nos ha de predecir la fiabilidad de si cierta persona, en ciertas condiciones, será capaz de devolver un crédito si se lo concedemos. El número de soluciones a las que podemos llegar es evidentemente finito, el crédito se concede, o no se concede, y en el caso de que se conceda, se decidirá si la cuantía solicitada es adecuada o si sólo se puede llegar hasta cierto límite.

Todas estas características indican que la metodología de resolución que mejor encaja es la clasificación heurística, por lo tanto dirigiremos el planteamiento con las fases que necesita.

Deberemos plantear cuatro tipos de elementos que definen el proceso de clasificación heurística y los mecanismos para transformar unos en otros. Primero definiremos como se plantearán los problemas al sistema, es decir, qué elementos se corresponderán con los datos específicos, las solicitudes de crédito.

Esta información ha de definir el estado financiero del solicitante, el motivo por el que pide el crédito, cuanto dinero solicita, etc. Supongamos que una solicitud contiene la siguiente información:

- Si tiene avales bancarios.
- Si tiene familiares que puedan responder por él.
- Si tiene cuentas corrientes, casas, coches, fincas, etc. y su valoración.
- Si tiene antecedentes de morosidad.
- Si ha firmado cheques sin fondos.
- Si tiene créditos anteriores concedidos.
- Tipo de empresa que quiere crear.
- Cantidad de dinero que solicita.

Esta información deberá convertirse mediante el proceso de abstracción de datos en los problemas abstractos a partir de los cuales se hará el razonamiento. Podríamos decidir que nuestras soluciones abstractas quedan definidas por los siguientes atributos:

- Apoyo financiero: Valoración de la capacidad económica para responder al valor del crédito que solicita. Este apoyo se puede evaluar con la información sobre avales y personas allegadas que puedan responder por él.
- Bienes: Dinero o propiedades que puedan usarse para responder por el crédito o que se puedan embargar en caso de no devolución.
- Fiabilidad de devolución: Información sobre si el cliente tiene antecedentes económicos positivos o negativos.
- Compromiso: Información sobre si ya se tienen compromisos económicos con esa persona o si se tienen intereses especiales con ella.
- Viabilidad de la empresa: Tipo de empresa que se quiere crear y su posible futuro.

Supondremos que estos cinco atributos pueden tomar valores cualitativos que estarán dentro de este conjunto: muy bueno, bueno, normal, regular, malo, muy malo.

Para realizar la abstracción de datos se podrían dar un conjunto de reglas que harían la transformación, como por ejemplo:

- si *avales* > un millón euros o tío rico entonces *apoyo financiero* bueno
- si *avales* entre 100000 euros y un millón entonces *apoyo financiero* normal
- si *avales* < 100000 euros entonces *apoyo financiero* malo
- si suma *bienes* < un millón entonces *bienes* malo
- si suma *bienes* entre uno y dos millones entonces *bienes* normal

- si suma *bienes* > dos millones entonces *bienes* bien
- si *cheques sin fondos* o *moroso* entonces *fiabilidad* muy mala
- si *fábrica de agujeros* entonces *viabilidad* muy mala
- si *hamburguesería* o *heladería* entonces *viabilidad* normal
- si *grandes almacenes* o *proveedor de internet* entonces *viabilidad* muy buena
- si *concedido crédito* < 100000 euros entonces *compromiso* regular
- si *concedido crédito* > un millón o *hermano del director* entonces *compromiso* bueno

El conjunto de soluciones abstractas a las que podría dar el análisis de las solicitudes podría ser el siguiente:

- Denegación, no hay crédito para el cliente.
- Aceptación, se acepta el crédito tal como se solicita.
- Aceptación con rebaja, se acepta el crédito, pero se rebaja la cantidad solicitada, harán falta reglas para crear la solución concreta indicando la cantidad final que se concede.
- Aceptación con interés preferente, se concede la cantidad solicitada, pero además se rebaja el interés que normalmente se pone al crédito, en este caso también hará falta generar una solución concreta.

Ahora nos faltan las reglas que nos harán la asociación heurística entre los problemas abstractos y las soluciones abstractas. Un conjunto de reglas que cubre una pequeña parte del espacio de soluciones podría ser:

- si *apoyo financiero*=regular y *bienes*=malo entonces *denegar*
- si *fiabilidad*={mala, muy mala} entonces *denegar*
- si *apoyo financiero*=normal y *bienes*=normal y *viabilidad*=buena entonces *aceptar con rebaja*
- si *apoyo financiero*=bueno y *bienes*=normal y *compromiso*=normal y *viabilidad*=buena entonces *aceptar*
- si *apoyo financiero*=bueno y *bienes*=bueno y *compromiso*=muy bueno y *viabilidad*=muy buena entonces *aceptar con interés preferente*

Por último, nos hacen falta reglas para poder generar soluciones concretas en los casos que son necesarias, algunas reglas podrían ser:

- si *aceptación con rebaja* y *petición* > 500000 euros y *bienes* = 500000 euros entonces *rebaja* a 500000 euros
- si *aceptación con interés preferente* y *petición* > un millón y *bienes* > un millón entonces *rebaja* de un 1 % de interés
- si *aceptación con interés preferente* y *hermano del director* entonces *rebaja* de un 2 % de interés

15.2.2 Resolución Constructiva

En contraste con la clasificación heurística, hay dominios en los que las soluciones no se pueden enumerar a priori, sino que la solución ha de construirse. Por ejemplo, en problemas de diseño, de planificación, y, por lo general, en todos los problemas que incluyen tareas de síntesis.

Este tipo de problemas se pueden atacar mediante métodos no guiados por conocimiento, pero obtener una solución satisfactoria es computacionalmente prohibitivo.

Construir una solución necesita que exista un modelo de la estructura y el comportamiento del objeto que se desea construir, modelo que debe contener conocimiento acerca de las restricciones que se deben satisfacer. Este conocimiento debe incluir:

1. Restricciones en la configuración de los componentes (físicas, temporales, ...).
2. Restricciones respecto a las entradas y salidas/precondiciones y postcondiciones de los operadores de construcción.
3. Interacciones entre estos dos tipos de restricciones.

Es necesario también conocimiento que permita evaluar las decisiones que se toman y la solución actual (total o parcial). Este conocimiento es más elaborado que el que se utiliza en los algoritmos de búsqueda heurística o búsqueda local. En estos algoritmos la información heurística es de carácter más general. En la resolución constructiva se debe poder evaluar la bondad de cada paso en las circunstancias particulares de su aplicación.

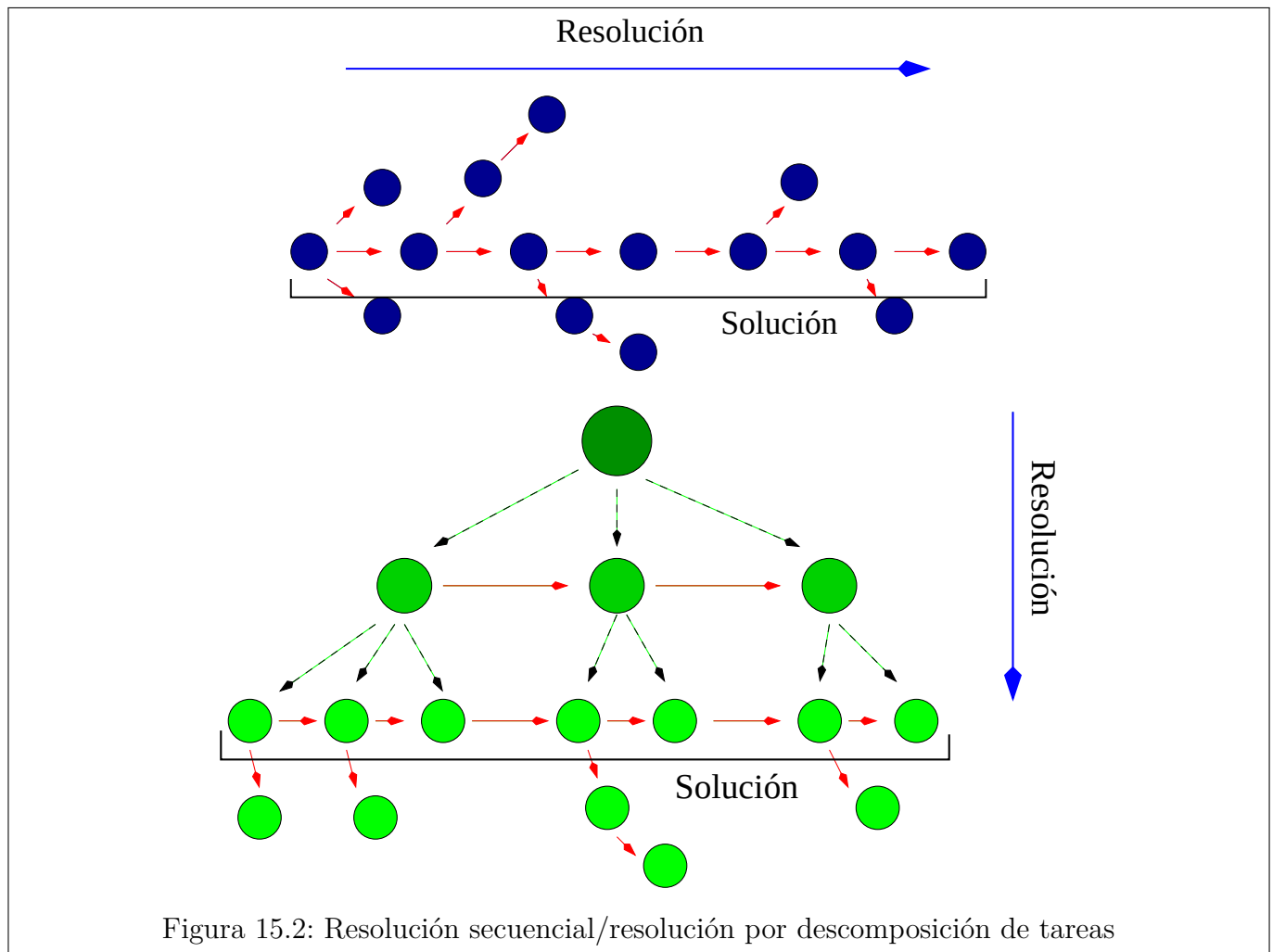
Dos son las estrategias generales que se siguen para la resolución de este tipo de problemas:

- Proponer y aplicar (*Propose and apply*).
- Mínimo compromiso (*Least commitment*).

Proponer y aplicar

En principio, el experto debe tener una idea clara de la descomposición en tareas del problema y de las relaciones espacio temporales entre éstas para, de esta manera, plantear las restricciones que se tienen que cumplir. Se han de definir también las operaciones que se pueden efectuar en cada estado de la resolución, cuándo se pueden aplicar y cuáles son sus efectos. Los pasos que se siguen en esta metodología son los siguientes:

1. Inicializar el objetivo: Se crea el elemento que define la solución actual
2. Proponer un operador: Se seleccionan operaciones plausibles sobre la solución actual.
3. Podar operadores: Se eliminan operadores de acuerdo con criterios globales. Estos criterios globales consistirán en criterios de consistencia generales que permiten descartar operadores que, aun siendo aplicables, se ve claramente que no mejorarán la solución (e.g.: no tiene sentido escoger el operador que deshaga el efecto del último operador aplicado).
4. Evaluar operadores: Se comparan los efectos de los operadores sobre la solución y se evalúa su resultado. Es en este punto donde interviene el conocimiento del experto para realizar la evaluación de los operadores. Si ninguno de los operadores es considerado adecuado se pasa a reconsiderar pasos anteriores.
5. Seleccionar un operador: Se escoge el operador mejor evaluado.
6. Aplicar el operador: Se aplica el operador al estado actual.



7. Evaluar el objetivo: Se para si se ha llegado al objetivo final o se reinicia el proceso.

La forma de plantear la solución del problema puede ser esencial para la eficiencia del proceso de resolución. El planteamiento más sencillo es definir el problema como una búsqueda en el espacio de soluciones parciales construyendo la solución paso a paso (secuencialmente). Esta aproximación puede ser viable si se dispone de un conocimiento exhaustivo sobre como evaluar cada uno de los pasos de la resolución y se puede dirigir la exploración rápidamente hacia el camino solución.

Esta aproximación puede ser poco eficiente si el conocimiento para la evaluación de los pasos de resolución no es suficientemente buena. Una alternativa es plantear el problema como una descomposición jerárquica de tareas. El conjunto de operadores permite dividir el problema inicial en problemas de complejidad decreciente hasta llegar al nivel de operaciones primitivas. Esto significa que el conjunto de operadores que permiten solucionar el problema no se ha de limitar a las acciones primitivas, sino que debe incluir aquellos que permiten hacer la descomposición del problema. Evidentemente esto aumenta el trabajo de adquisición del conocimiento, pero redundará en un aumento considerable de la eficiencia.

Mínimo compromiso

Un planteamiento alternativo consiste en partir de soluciones completas no óptimas e ir mejorándolas hasta llegar a una solución mejor, aunque veces se puede partir de una no solución y corregirla hasta llegar al espacio de soluciones. La búsqueda la realizaríamos en el espacio de soluciones completas. Para poder aplicar esta estrategia el problema nos debería permitir hallar fácilmente una

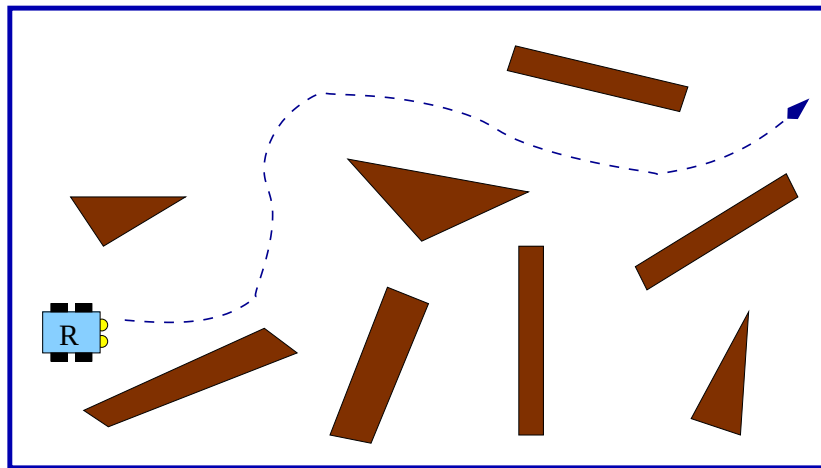


Figura 15.3: Planificación de la trayectoria de un robot

solución inicial completa. La elección del operador a aplicar en cada paso de la resolución la define la estrategia de mínimo compromiso: mínima modificación que imponga menos restricciones futuras.

La estrategia de resolución sería la siguiente:

1. Partir de una solución inicial no óptima, usualmente que satisface las restricciones.
2. Hacer una modificación sobre la solución. Esta modificación ha de hacerse de acuerdo con la heurística de mínimo compromiso, es decir, escoger la acción que menos restricciones imponga sobre la solución y, por lo tanto, menos restricciones imponga sobre el próximo paso.
3. Si la modificación viola alguna de las restricciones, se intenta deshacer alguno de los pasos anteriores, procurando que las modificaciones sean las mínimas. Esta modificación no tiene por que ser precisamente deshacer el último paso que se realizó.

El conocimiento del experto ha de aparecer en la evaluación de los efectos de los operadores sobre las restricciones, de manera que se pueda escoger siempre el operador con menos efecto sobre éstas y que permita más libertad de movimientos.

Aplicación de la resolución constructiva

Queremos planificar la mejor trayectoria de un robot en una habitación de manera que de un punto de salida llegue a la puerta de la habitación sin colisionar con ninguno de los obstáculos que hay en la habitación. Supondremos que el robot puede ver lo que tiene delante de él y con ello calcular la distancia a los objetos que tiene a su alrededor.

Disponemos de un conjunto de operadores que le permiten:

- Moverse hacia adelante o hacia atrás cierta distancia a cierta velocidad
- Girar cierto número de grados

En este caso las restricciones globales son que se debe llegar a la puerta de salida y que el recorrido de la trayectoria y el tiempo que se tarde debe ser el menor posible. Las restricciones que se aplican a la elección de los operadores serán que el robot no choque con ninguno de los obstáculos o la pared, esto incluye que la distancia para pasar entre dos obstáculos debe ser la adecuada para que el robot pueda maniobrar y no se quede atascado.

La evaluación de la bondad de los operadores dependerá de cada movimiento. El operador mover será mejor cuando su aplicación nos acerque más al objetivo y más rápidamente. El operador girar

será mejor cuando la trayectoria en que nos deje el giro esté más libre y por lo tanto más lejos tenga los obstáculos más próximos.

Con estos operadores y restricciones, podemos resolver el problema utilizando los pasos de resolución de la estrategia proponer y aplicar para encontrar la ruta.

Razonamiento aproximado e incertidumbre

16.1 Incertidumbre y falta de información

Por lo general, el conocimiento que se debe manejar dentro de la mayoría de los dominios tratados por los sistemas basados en el conocimiento (SBC) no es de naturaleza exacta. En la práctica nos encontramos con problemas como:

- Representar el conocimiento para cubrir todos los hechos que son relevantes para un problema es difícil
- Existen dominios en los que se desconocen todos los hechos y reglas necesarias para resolver el problema
- Existen problemas en los que aún teniendo las reglas para resolverlos no disponemos de toda la información necesaria para aplicarlas

Esto significa que para poder razonar dentro de estos sistemas tendremos que utilizar herramientas más potentes que las que nos brinda la lógica clásica, que sólo nos permitiría trabajar con conocimiento del que pudiéramos establecer de manera efectiva su veracidad o falsedad.

De hecho, este objetivo no es descabellado ya que podemos observar como toda persona esta acostumbrada a tomar decisiones ante información incompleta o imprecisa (Invertimos en bolsa, diagnosticamos enfermedades, ...) y esa imprecisión o falta de conocimiento no impide la toma de decisiones. Esta claro que si deseamos que los SBC emulen la capacidad de los expertos hemos de dotarlos de mecanismos que sean capaces de abordar este problema.

La imprecisión o la falta de certeza en la información proviene de muchas fuentes, de entre ellas podemos citar:

1. *Incompletitud de los datos* debida a la no disponibilidad de éstos.
2. *Incertidumbre de los datos* debida a las limitaciones de los aparatos de medida, o a apreciaciones subjetivas del observador.
3. *Incertidumbre en las asociaciones* realizadas entre datos y conclusiones.
4. *Imprecisión en el lenguaje de descripción* debida al uso del lenguaje natural, ya que se presta a ambigüedades y malas interpretaciones.

El tratar con este problema ha llevado a desarrollar un conjunto de lógicas y modelos que intentan tratar el problema de la incompletitud e imprecisión del conocimiento desde diferentes perspectivas

y modelizar de esta manera los procesos de razonamiento que aplican las personas. Muchas son las propuestas que se han desarrollado a lo largo de la evolución de los SBC, nos centraremos únicamente en dos formalismos que provienen de dos visiones distintas de la incertidumbre:

- Modelo probabilista (Redes Bayesianas)
- Modelo posibilista (Lógica difusa)

17.1 Introducción

Los modelos probabilistas se fundamentan en la teoría de la probabilidad. Las probabilidades se utilizan para modelizar nuestra creencia sobre la veracidad o falsedad de los hechos, de manera que podamos asignar valores de probabilidad a los diferentes hechos con los que tratamos y utilizar esas probabilidades para razonar sobre su certidumbre.

Cada hecho tendrá una probabilidad asociada de por sí, o derivada de la probabilidad de aparición de otros hechos. Estas probabilidades serán las que nos permitirán tomar decisiones. Esta toma de decisiones no es estática, la probabilidad de un hecho podrá ser modificada por la observación y la modificación de la creencia en otros hechos que estén relacionados.

Podemos por ejemplo suponer que el hecho de decidir llevar o no un paraguas al salir de casa por la mañana puede verse afectada por múltiples circunstancias. En principio tendremos una decisión a priori que dependerá del clima donde nos encontremos. Si estamos en una zona donde no llueve apenas nuestra decisión por omisión será no cogerlo.

Si por ejemplo vemos la predicción del tiempo el día anterior esta decisión puede variar dependiendo de lo que nos cuenten, si nos dicen que al día siguiente habrá nubosidad la decisión se inclinará más hacia coger el paraguas. Si además observamos al día siguiente que el suelo está mojado esta decisión se reforzará aún más.

17.2 Teoría de probabilidades

Antes de comenzar a hablar de cómo modelizar el razonamiento mediante probabilidades, debemos repasar algunos conceptos básicos de la teoría de probabilidades.

El elemento fundamental de teoría de probabilidades es la **variable aleatoria**. Una variable aleatoria tiene un dominio de valores (valores posibles que puede tomar y sobre los que establecemos una distribución de probabilidad), podemos tener variables aleatorias **booleanas**, **discretas** o **continuas**.

Para poder trasladar la teoría de la probabilidad a un sistema basado en el conocimiento, deberemos crear una relación entre la representación del conocimiento que utilizamos y los elementos sobre los que establecemos las distribuciones de probabilidad.

En la práctica, toda representación del conocimiento que utilizamos se fundamenta en la lógica, de manera que la utilizaremos como lenguaje representación y utilizaremos las fórmulas lógicas como elemento básico. De esta forma, definiremos una **proposición lógica** como cualquier fórmula en lógica de enunciados o predicados, siendo éstos elementos primitivos de nuestra representación. Una

proposición lógica tendrá asociada una variable aleatoria que indicará nuestro grado de creencia en ella.

Una variable aleatoria tendrá asociada una **distribución de probabilidad**. La forma de expresar esta distribución de probabilidad dependerá del tipo de variable aleatoria (Discretas: Binomial, Multinomial, ...; Continuas: Normal, χ^2 , ...). El elegir un tipo de variable aleatoria u otro depende de como creamos que la información correspondiente a la proposición lógica debe modelarse. Para simplificar, sólo trabajaremos con variables aleatorias discretas, de manera que toda proposición lógica tendrá un conjunto enumerado de posibles respuestas.

En cualquier problema, tendremos distintas proposiciones lógicas que intervendrán en una decisión, por lo tanto, tendremos que describir como son las variables aleatorias que describen estas proposiciones y como se debe calcular su influencia sobre las deducciones que permiten realizar las proposiciones.

La unión de variables aleatorias se puede describir mediante una **distribución de probabilidad conjunta**. Este será el mecanismo que nos va a permitir describir como se puede razonar mediante probabilidades.

Denotaremos como $P(a)$ la probabilidad de que la proposición A tenga el valor a . Por ejemplo, la proposición *Fumar* puede tener los valores $\{fumar, \neg fumar\}$, $P(\neg fumar)$ es la probabilidad de la proposición $Fumar = \neg fumar$. Denotaremos como $P(A)$ al **vector de probabilidades** de todos los posibles valores de la proposición A .

Definiremos como **probabilidad a priori** ($P(a)$) asociada a una proposición como el grado de creencia en ella a falta de otra información. Del conjunto de proposiciones que tengamos, algunas no tienen por que estar influidas por otras, de estas dispondremos de una distribución de probabilidad a priori que representará la probabilidad de que tomen cualquiera de sus valores.

Definiremos como **probabilidad a posteriori** o **condicional** ($P(a|b)$) como el grado de creencia en una proposición tras la observación de proposiciones asociadas a ella. Esta probabilidad estará asociada a las proposiciones que se ven influidas por la observación de otras proposiciones, por lo que nuestra creencia en ellas variará según la observación de éstas.

La probabilidad a posteriori se puede definir a partir de probabilidades a priori como:

$$P(a|b) = \frac{P(a \wedge b)}{P(b)}$$

Esta fórmula se puede transformar en lo que denominaremos la **regla del producto**:

$$P(a \wedge b) = P(a|b)P(b) = P(b|a)P(a)$$

Podemos observar por esta regla que la teoría de la probabilidad no asigna a priori una dirección a la causalidad y que se puede calcular la influencia de una proposición en otra y viceversa.

17.3 Inferencia probabilística

El usar como base de un mecanismo de inferencia la teoría de la probabilidad, restringe las cosas que podemos creer y deducir al marco de los axiomas en los que se fundamenta la probabilidad. Estos axiomas son:

- Toda probabilidad está en el intervalo $[0, 1]$

$$0 \leq P(a) \leq 1$$

- La proposición *cierto* tiene probabilidad 1 y la proposición *falso* tiene probabilidad 0

$$P(\text{cierto}) = 1 \quad P(\text{falso}) = 0$$

- La probabilidad de la disyunción se obtiene mediante la fórmula

$$P(a \vee b) = P(a) + P(b) - P(a \wedge b)$$

Dadas estas reglas básicas, podemos establecer una serie de mecanismos de inferencia, como por ejemplo:

- **Marginalización:** Probabilidad de una proposición atómica con independencia de los valores del resto de proposiciones

$$P(Y) = \sum_z P(Y, z)$$

- **Probabilidades condicionadas:** Probabilidad de una proposición dados unos valores para algunas proposiciones e independiente del resto de proposiciones (a partir de la regla del producto)

$$P(X|e) = \alpha \sum_y P(X, e, y)$$

El valor α es un factor de normalización que corresponde a factores comunes que hacen que las probabilidades sumen 1.

Ejemplo 17.1 Consideremos un problema en el que intervengan las proposiciones $Fumador = \{fumador, \neg fumador\}$, $Sexo = \{varon, mujer\}$, $Enfisema = \{enfisema, \neg enfisema\}$

La siguiente tabla nos describe las distribuciones de probabilidad conjunta de estas proposiciones

	enfisema		$\neg enfisema$	
	varon	mujer	varon	mujer
fumador	0.2	0.1	0.05	0.05
$\neg fumador$	0.02	0.02	0.23	0.33

A partir de ella podemos hacer ciertas inferencias probabilísticas respecto a la combinación de las diferentes proposiciones y su influencia entre ellas

$$\begin{aligned}
 P(enfisema \wedge varon) &= 0.2 + 0.02 \\
 P(fumador \vee mujer) &= 0.2 + 0.1 + 0.05 + 0.05 + 0.02 + 0.33 \\
 P(Fumador|enfisema) &= \langle P(fumador, enfisema, varon) \\
 &\quad + P(fumador, enfisema, mujer), \\
 &\quad P(\neg fumador, enfisema, varon) \\
 &\quad + P(\neg fumador, enfisema, mujer) \rangle \\
 &= \alpha \langle 0.3, 0.04 \rangle \\
 &= \langle 0.88, 0.12 \rangle
 \end{aligned}$$

Para poder realizar todos estos procesos de inferencia se requiere almacenar y recorrer la distribución de probabilidad conjunta de todas las proposiciones. Esto supone un gasto en tiempo y espacio impracticable. Suponiendo proposiciones binarias el coste en espacio y tiempo es $O(2^n)$ siendo n el número de proposiciones.

Cualquier problema real tiene un número de proposiciones suficiente para hacer que estos mecanismos de inferencia no sean útiles por su coste computacional. Se hace pues necesario crear mecanismos que nos simplifiquen el coste del razonamiento

17.3.1 Probabilidades condicionadas y reglas de producción

Hasta ahora podría parecer que los métodos de inferencia probabilística son mecanismos alejados de los sistemas de producción, ya que no parece que existan los conceptos de premisas y conclusiones, ni de encadenamiento.

En el mundo probabilístico la realidad está compuesta por un conjunto de eventos (variables aleatorias) que están ligados a través de las distribuciones de probabilidad conjunta. Un suceso es la observación de una combinación de valores de esas variables.

Al utilizar la formula de probabilidades condicionadas estamos imponiendo una relación de causalidad entre las variables que componen esos sucesos, cuando escribimos:

$$P(X|e) = \alpha \sum_y P(X, e, y)$$

estamos preguntando como influyen el conjunto de variables conocidas e sobre el valor de verdad de la variable en la que estamos interesados X (¿es mas probable x o $\neg x$?). Asumimos que X es consecuencia de e .

La ventaja de la formalización probabilística es que esta fórmula nos permite evaluar todas las reglas posibles que podamos crear como combinación de todas las variables de nuestro problema.

Por ejemplo, supongamos que en un problema intervienen un conjunto de variables $\mathcal{V} = \{A, B, C, D, E, F\}$, todas las variables están ligadas mediante una distribución de probabilidad conjunta $P(A, B, C, D, E, F)$.

En el formalismo de reglas de producción si consideramos que F se ve influida por los valores del resto de variables escribiríamos reglas como por ejemplo:

$$\begin{aligned} A \wedge \neg B \wedge C \wedge \neg D \wedge E &\rightarrow F \\ \neg A \wedge B \wedge \neg C &\rightarrow \neg F \end{aligned}$$

Pero si nos fijamos en el significado de la distribución de probabilidad conjunta, ésta precisamente nos representa todas las posibles reglas que podemos construir con esas variables. Esta distribución nos permite calcular la influencia que tiene cualquier subconjunto de variables respecto al valor de verdad de cualquier otro subconjunto. La forma de calcularla es aplicar fórmula de probabilidades condicionadas.

Evidentemente, supone una ventaja poder codificar todas estas reglas como una distribución de probabilidad y poseer un mecanismo de razonamiento simple.

Al ser este planteamiento exhaustivo (representamos todas las reglas posibles) el formalismo probabilístico también nos permite realizar inferencia ante la falta de premisas (en nuestro caso serán variables ocultas). La distribución de probabilidad conjunta y la formula de probabilidades condicionadas nos permiten incluir la influencia de estas variables a pesar de que no conozcamos su valor.

Los problemas que tendremos serán obtener esa distribución de probabilidad conjunta y que el mecanismo de inferencia es computacionalmente prohibitivo tal como está planteado hasta ahora.

17.4 Independencia probabilística y la regla de Bayes

Por lo general, no todas las proposiciones que aparecen en un problema están relacionadas entre si. De hecho para cada proposición dependiente podemos identificar sólo un subconjunto de proposicio-

nes que las influyen, siendo el resto irrelevantes para la inferencia de sus probabilidades. Llamaremos a esta propiedad **independencia probabilística**

Suponiendo que dos proposiciones X e Y no se influyen entre si, podemos reescribir sus probabilidades como:

$$P(X|Y) = P(X); \quad P(Y|X) = P(Y); \quad P(X, Y) = P(X)P(Y)$$

Dadas estas propiedades podremos reescribir las probabilidades conjuntas de manera más compacta reduciendo la complejidad

Anteriormente hemos enunciado la regla del producto como:

$$P(X, Y) = P(X|Y)P(Y) = P(Y|X)P(X)$$

Esta regla nos lleva a lo que denominaremos la **regla de Bayes**

$$P(Y|X) = \frac{P(X|Y)P(Y)}{P(X)}$$

Esta regla y la propiedad de independencia serán el fundamento del razonamiento probabilístico y nos permitirá relacionar las probabilidades de unas evidencias con otras.

Suponiendo que podemos estimar exhaustivamente todas las probabilidades que involucran la variable Y podemos prescindir de las probabilidades a priori de la variable X y reescribir la formula de Bayes como:

$$P(Y|X) = \alpha P(X|Y)P(Y)$$

Esto es así porque las probabilidades $P(Y = y_1|X) \dots P(Y = y_n|X)$ han de sumar uno, α será un factor de normalización.

Suponiendo independencia condicional entre dos variables X e Y podremos escribir la probabilidad condicional de otra variable Z respecto a estas como:

$$P(X, Y|Z) = P(X|Z)P(Y|Z)$$

De manera que si sustituimos en la regla de Bayes:

$$P(Z|X, Y) = \alpha P(X|Z)P(Y|Z)P(Z)$$

17.5 Redes Bayesianas

Si determinamos la independencia entre variables podemos simplificar el cálculo de la combinación de sus probabilidades y su representación, de manera que podremos razonar sobre la influencia de las probabilidades de unas proposiciones lógicas sobre otras de una manera más eficiente

Las **redes bayesianas** son un formalismo que permite la representación de las relaciones de independencia entre un conjunto de variables aleatorias. Una red bayesiana es un **grafo dirigido acíclico** que contiene información probabilística en sus nodos, indicando cual es la influencia que tienen sobre un nodo X_i sus padres en el grafo ($P(X_i|padres(X_i))$). El significado intuitivo de un enlace entre dos nodos X e Y es que la variable X tiene influencia directa sobre Y .

En cada uno de los nodos de la red aparece la distribución de probabilidad del nodo respecto a sus padres, es decir, como estos influyen la probabilidad del hijo. Esta forma de representar las influencias entre variables permite factorizar la distribución de probabilidad conjunta, convirtiéndose en el producto de probabilidades condicionales independientes:

$$P(x_1, x_2, \dots, x_n) = \prod_{i=1}^n P(x_i | \text{padres}(x_i))$$

El conjunto de probabilidades representadas en la red bayesiana describe la distribución de probabilidad conjunta de todas las variables, esto hace que no sea necesaria una tabla completa que describa la influencia entre todas ellas.

Intuitivamente podríamos decir que con la red bayesiana estamos añadiendo suposiciones adicionales al modelo probabilístico puro (todo está relacionado con todo) estableciendo que solamente algunas influencias son posibles y que existe una dirección específica para la causalidad, que es la que indican los arcos de la red.

Las propiedades que tienen las redes bayesianas nos dan ciertas ideas sobre como debemos construirlas a partir de un conjunto de proposiciones. Si consideramos que (regla del producto):

$$P(x_1, x_2, \dots, x_n) = P(x_n | x_{n-1}, \dots, x_1) P(x_{n-1}, \dots, x_1)$$

Iterando el proceso tenemos que:

$$\begin{aligned} P(x_1, \dots, x_n) &= P(x_n | x_{n-1}, \dots, x_1) P(x_{n-1} | x_{n-2}, \dots, x_1) \\ &\quad \dots P(x_2 | x_1) P(x_1) \\ &= \prod_{i=1}^n P(x_i | x_{i-1}, \dots, x_1) \end{aligned}$$

Esta es la llamada **regla de la cadena**

Dadas estas propiedades, podemos afirmar que si $\text{padres}(X_i) \subseteq \{X_{i-1}, \dots, X_1\}$, entonces:

$$P(X_i | X_{i-1}, \dots, X_1) = P(X_i | \text{padres}(X_i))$$

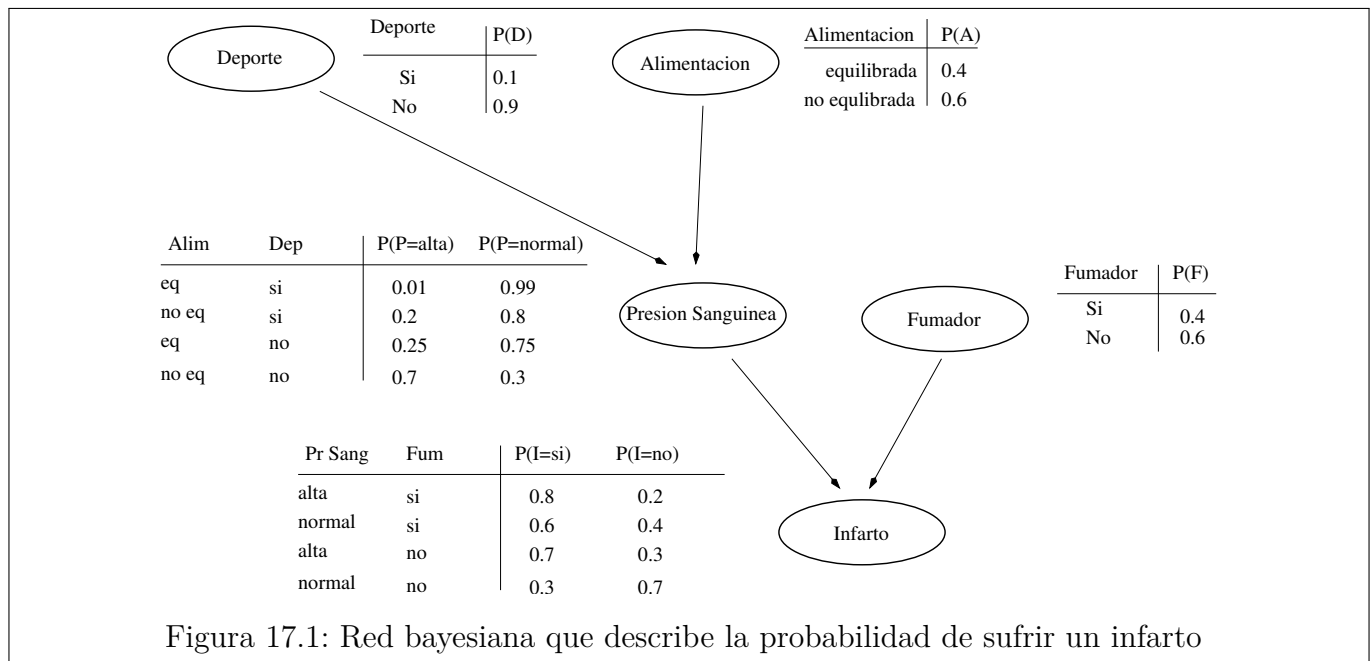
Esto quiere decir que una red bayesiana es una representación correcta de un dominio sólo si cada nodo es condicionalmente independiente de sus predecesores en orden, dados sus padres.

Para lograr esto, se han de escoger como padres de una variable X_i aquellas de entre las variables $X_1, \dots, X_{i-1}$ que influyan directamente en X_i . Es decir, para describir la influencia que recibe una proposición del resto de proposiciones de las que disponemos, sólo es necesario utilizar las que influyen más directamente. La influencia del resto de proposiciones (si es que existe) estará descrita por las relaciones que puedan tener estas con los padres inmediatos de la proposición.

El utilizar una red bayesiana como representación de la distribución de probabilidad conjunta de un grupo de proposiciones supone una gran reducción en coste espacial. Como comentamos, el coste de representar la distribución de probabilidad conjunta de n variables binarias es $O(2^n)$. La representación de redes bayesianas nos permite una representación más compacta gracias a la factorización de la distribución conjunta. Suponiendo que cada nodo de la red tenga como máximo k padres ($k \ll n$), un nodo necesitará 2^k para representar la influencia de sus padres, por lo tanto el espacio necesario será $O(n2^k)$. Por ejemplo, con 10 variables y suponiendo 3 padres como máximo tenemos 80 frente a 1024, con 100 variables y suponiendo 5 padres tenemos 3200 frente a aproximadamente 10^{30} .

Ejemplo 17.2 *La red bayesiana de la figura 17.1 muestra las relaciones de dependencia entre un conjunto de proposiciones lógicas y la distribución de probabilidad que sigue cada una de esas influencias.*

A partir de la red podemos calcular la probabilidad de una proposición lógica utilizando las relaciones de dependencia entre las variables



$$\begin{aligned}
 &P(\text{Infarto} = \text{si} \wedge \text{Presion} = \text{alta} \wedge \text{Fumador} = \text{si} \\
 &\wedge \text{Deporte} = \text{si} \wedge \text{Alimentacion} = \text{equil}) \\
 &= \\
 &P(\text{Infarto} = \text{si} | \text{Presion} = \text{alta}, \text{Fumador} = \text{si}) \\
 &P(\text{Presion} = \text{alta} | \text{Deporte} = \text{si}, \text{Alimentacion} = \text{equil}) \\
 &P(\text{Fumador} = \text{si})P(\text{Deporte} = \text{si})P(\text{Alimentacion} = \text{equil}) \\
 &= 0.8 \times 0.01 \times 0.4 \times 0.1 \times 0.4 \\
 &= 0.000128
 \end{aligned}$$

17.6 Inferencia probabilística mediante redes bayesianas

El objetivo de la inferencia probabilística es calcular la distribución de probabilidad a posteriori de un conjunto de variables dada la observación de un evento (valores observados para un subconjunto de variables).

Denotaremos como X la variable sobre la que queremos conocer la distribución, $\mathbf{E}$ será el conjunto de variables de las que conocemos su valor $\{E_1, \dots, E_n\}$, e $\mathbf{Y}$ será el conjunto de variables que no hemos observado $\{Y_1, \dots, Y_n\}$ (variables ocultas). De esta manera $\mathbf{X} = \{X\} \cup \mathbf{E} \cup \mathbf{Y}$ será el conjunto completo de variables. Nos plantearemos el cálculo de $P(X|\mathbf{e})$, es decir la distribución de probabilidad de los valores de X a partir de la influencia de los valores observados de las variables de $\mathbf{E}$.

Nosotros nos plantearemos lo que denominaremos la inferencia exacta, que es la que se realiza utilizando directamente la distribución de probabilidad que describe la red bayesiana. Como veremos mas adelante ésta sólo es tratable computacionalmente si la topología de la red tiene ciertas propiedades.

17.6.1 Inferencia por enumeración

El primer algoritmo de inferencia exacta que veremos es el denominado de **Inferencia por enumeración**. Éste se basa en que cualquier probabilidad condicionada se puede calcular como la suma de todos los posibles casos a partir de la distribución de probabilidad conjunta.

$$P(X|\mathbf{e}) = \alpha P(X, \mathbf{e}) = \alpha \sum_y P(X, \mathbf{e}, y)$$

La red bayesiana nos permite factorizar la distribución de probabilidad conjunta y obtener una expresión mas fácil de evaluar.

Eso no quita que el número de operaciones crezca de manera exponencial con el número de variables que tiene la expresión, además de realizar parte de las operaciones múltiples veces dependiendo de la estructura de la red. Por esta causa este método no es utilizado en la práctica.

Ejemplo 17.3 Usando la red bayesiana ejemplo podemos calcular la probabilidad de ser fumador si se ha tenido un infarto y no se hace deporte

$$P(\text{Fumador} | \text{Infarto} = \text{si}, \text{Deporte} = \text{no})$$

La distribución de probabilidad conjunta de la red sería:

$$P(D, A, S, F, I) = P(I|S, F)P(F)P(S|D, A)P(D)P(A)$$

Debemos calcular $P(F|I = \text{si}, D = \text{no})$, por lo tanto tenemos

$$\begin{aligned} P(F|I = s, D = n) &= \alpha P(F, I = s, D = n) \\ &= \alpha \sum_{A \in \{e, \neg e\}} \sum_{S \in \{a, n\}} P(D = n, A, S, F, I = s) \\ &= \alpha P(D = n)P(F) \sum_{A \in \{e, \neg e\}} P(A) \sum_{S \in \{a, n\}} P(S|D = n, A)P(I = s|S, F) \end{aligned}$$

Si enumeramos todas las posibilidades y las sumamos de acuerdo con la distribución de probabilidad conjunta tenemos que:

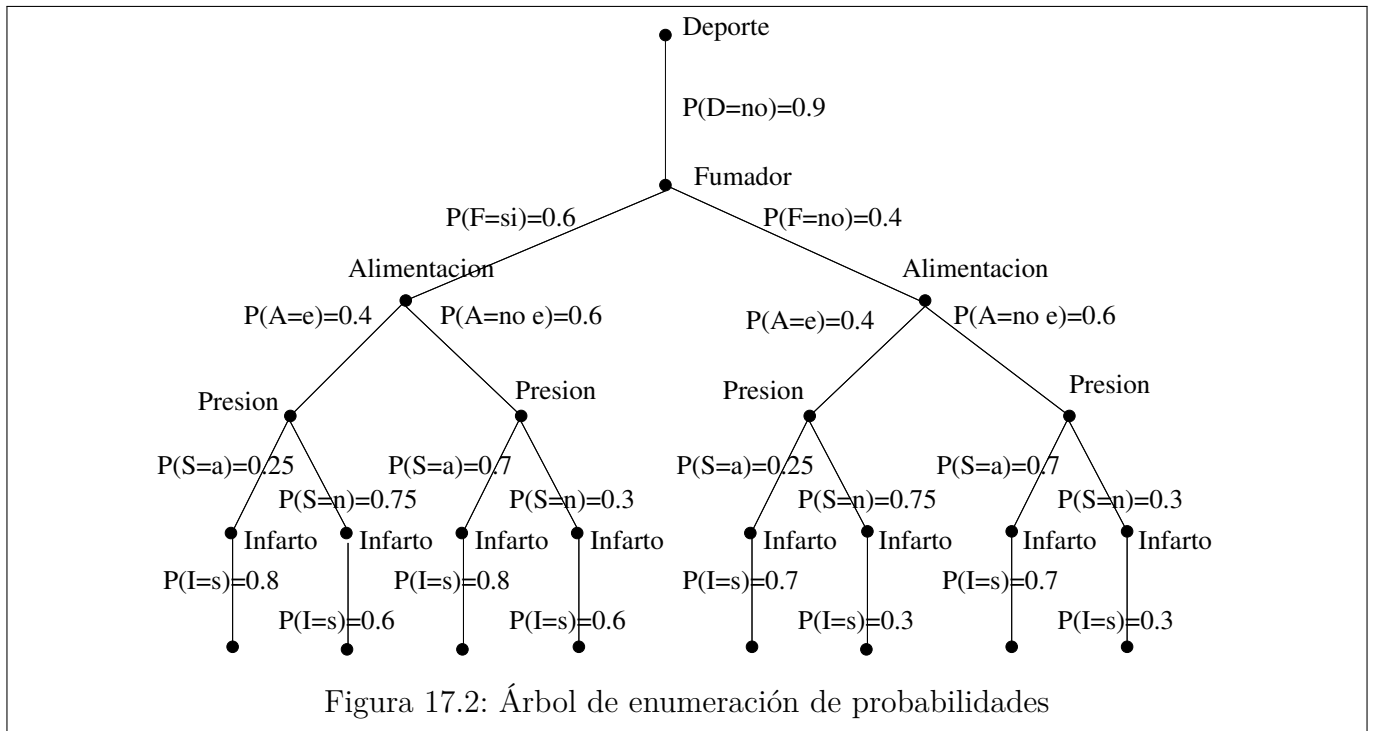
$$\begin{aligned} &P(\text{Fumador} | \text{Infarto} = \text{si}, \text{Deporte} = \text{no}) \\ &= \alpha (0.9 \cdot 0.4 \cdot (0.4 \cdot (0.25 \cdot 0.8 + 0.75 \cdot 0.6) + 0.6 \cdot (0.7 \cdot 0.8 + 0.3 \cdot 0.6)), \\ &\quad 0.9 \cdot 0.6 \cdot (0.4 \cdot (0.25 \cdot 0.7 + 0.75 \cdot 0.3) + 0.6 \cdot (0.7 \cdot 0.7 + 0.3 \cdot 0.3)) \\ &= \alpha (0.253, 0.274) \\ &= \langle 0.48, 0.52 \rangle \end{aligned}$$

Podemos ver las operaciones que se realizan en el árbol de probabilidades que se calcula (figura 17.2). Cada una de las ramas del árbol corresponde a cada uno de los eventos posibles.

17.6.2 Algoritmo de eliminación de variables

La inferencia por enumeración puede ser bastante ineficiente dependiendo de la estructura de la red y dar lugar a muchos cálculos repetidos, por lo que se han intentado hacer algoritmos más eficientes. El **algoritmo de eliminación de variables** intenta evitar esta repetición de cálculos. El algoritmo utiliza técnicas de programación dinámica (memorización) de manera que se guardan cálculos intermedios para cada variable para poder reutilizarlos. A estos cálculos intermedios los denominaremos **factores**

El cálculo de la probabilidad se realiza evaluando la expresión de la distribución de probabilidad conjunta de izquierda a derecha, aprovechando ese orden para obtener los factores. Esta estrategia hace que los cálculos que impliquen una variable se realicen una sola vez. Los *factores* correspondientes a cada variable se van acumulando y utilizándose según se necesitan.



Algoritmo 17.1 Algoritmo de eliminación de variables

Función: Elimination de Variables (X, e, rb)

```

factores  $\leftarrow []$ 
vars  $\leftarrow$  REVERSE(VARS( $rb$ ))
para cada  $var \in vars$  hacer
    factores  $\leftarrow$  concatena(factores, CALCULA-FACTOR( $var, e$ ))
    si  $var$  es variable oculta entonces
        factores  $\leftarrow$  PRODUCTO-Y-SUMA( $var, factores$ )
    fin
fin
retorna NORMALIZA(PRODUCTO(factores))

```

Una ventaja adicional de este algoritmo es que las variables no relevantes desaparecen al ser factores constantes en las operaciones y por lo tanto permite eliminarlas del cálculo (de ahí el nombre de algoritmo de eliminación de variables).

Su implementación se puede ver en el algoritmo 17.1. CALCULA_FACTOR genera el factor correspondiente a la variable en la función de distribución de probabilidad conjunta, PRODUCTO_Y_SUMA multiplica los factores y suma respecto a la variable oculta, PRODUCTO multiplica un conjunto de factores.

Un factor corresponde a la probabilidad de un conjunto de variables dadas las variables ocultas. Se representa por una tabla que para cada combinación de variables ocultas da la probabilidad de las variables del factor, por ejemplo:

	Y	Z	
$f_X(Y, Z) =$	C	C	0.2
	C	F	0.4
	F	C	0.8
	F	F	0.6

Los factores tienen dos operaciones, la suma y producto de factores.

La suma se aplica a un factor y sobre una variable oculta del factor. Como resultado obtenemos una matriz reducida en la que las filas del mismo valor se han acumulado, por ejemplo

$$f_{X\bar{Z}}(Y) = \sum_Z f_X(Y, Z) = \begin{array}{c|c} Y & \\ \hline C & 0.6 \\ F & 1.4 \end{array}$$

Es igual que una operación de agregación sobre una columna en bases de datos

El producto de factores permite juntar varios factores entre ellos utilizando las variables ocultas comunes, por ejemplo:

$$f_{X_1 X_2}(Y, W, Z) = f_{X_1}(Y, Z) \times f_{X_2}(Z, W) =$$

Y	Z		Z	W		Y	Z	W	
C	C	0.2	C	C	0.3	C	C	C	0.2×0.3
C	F	0.8	C	F	0.7	C	C	F	0.2×0.7
F	C	0.4	F	C	0.1	C	F	C	0.8×0.1
F	F	0.6	F	F	0.9	C	F	F	0.8×0.9
						F	C	C	0.4×0.3
						F	C	F	0.4×0.7
						F	F	C	0.6×0.1
						F	F	F	0.6×0.9

Es igual que una operación de join en una base de datos multiplicando los valores de las columnas de datos.

Ejemplo 17.4 Volveremos a calcular $P(\text{Fumador} | \text{Infarto} = \text{si}, \text{Deporte} = \text{no})$ a partir de la distribución de probabilidad conjunta:

$$P(D, A, S, F, I) = P(I | S, F) P(F) P(S | D, A) P(D) P(A)$$

Debemos calcular $P(F | I = \text{si}, D = \text{no})$, por lo tanto tenemos

$$\begin{aligned} P(F | I = s, D = n) &= \alpha P(I = s, F, D = n) \\ &= \alpha \sum_{A \in \{e, \neg e\}} \sum_{S \in \{a, n\}} P(D = n, A, S, F, I = s) \end{aligned}$$

En esta ocasión no sacamos factores comunes para seguir el algoritmo

$$\alpha P(D = n) \sum_{A \in \{e, \neg e\}} P(A) \sum_{S \in \{a, n\}} P(S | D = n, A) P(F) P(I = s | S, F)$$

El algoritmo empieza calculando el factor para la variable Infarto ($P(I = s | S, F)$), esta tiene fijo su valor a **si**, depende de las variables Presión Sanguínea y Fumador

$$f_I(S, F) = \begin{array}{cc|c} S & F & \\ \hline a & s & 0.8 \\ a & n & 0.7 \\ n & s & 0.6 \\ n & n & 0.3 \end{array}$$

La variable fumador ($P(F)$) no depende de ninguna otra variable, al ser la variable que preguntamos el factor incluye todos los valores

$$f_F(F) = \begin{array}{c|c} F & \\ \hline s & 0.4 \\ n & 0.6 \end{array}$$

La variable Presión Sanguínea ($P(S|D = n, A)$), depende de las variable Deporte que tiene fijo su valor a **no** y Alimentación. Esta es una variable oculta, por lo que se debe calcular para todos sus valores

$$f_S(S, A) = \begin{array}{cc|c} S & A & \\ \hline a & e & 0.25 \\ a & \neg e & 0.7 \\ n & e & 0.75 \\ n & \neg e & 0.3 \end{array}$$

Al ser la variable Presión Sanguínea una variable oculta debemos acumular todos los factores que hemos calculado

$$f_S(S, A) \times f_F(F) \times f_I(S, F)$$

$$f_{FI}(S, F) = f_F(F) \times f_I(S, F) = \begin{array}{cc|c} S & F & \\ \hline a & s & 0.8 \times 0.4 \\ a & n & 0.7 \times 0.6 \\ n & s & 0.6 \times 0.4 \\ n & n & 0.3 \times 0.6 \end{array}$$

$$f_{FIS}(S, F, A) = f_{FI}(S, F) \times f_S(S, A) = \begin{array}{ccc|c} S & F & A & \\ \hline a & s & e & 0.8 \times 0.4 \times 0.25 \\ a & s & \neg e & 0.8 \times 0.4 \times 0.7 \\ a & n & e & 0.7 \times 0.6 \times 0.25 \\ a & n & \neg e & 0.7 \times 0.6 \times 0.7 \\ n & s & e & 0.6 \times 0.4 \times 0.75 \\ n & s & \neg e & 0.6 \times 0.4 \times 0.3 \\ n & n & e & 0.3 \times 0.6 \times 0.75 \\ n & n & \neg e & 0.3 \times 0.6 \times 0.3 \end{array}$$

Y ahora sumamos sobre todos los valores de la variable S para obtener el factor correspondiente a la variable Presión Sanguínea

$$f_{FIS}(F, A) = \sum_{S \in \{a, n\}} f_{FIS}(S, F, A) =$$

F	A	
s	e	$0.8 \times 0.4 \times 0.25 + 0.6 \times 0.4 \times 0.75 = 0.26$
s	$\neg e$	$0.8 \times 0.4 \times 0.7 + 0.6 \times 0.4 \times 0.3 = 0.296$
n	e	$0.7 \times 0.6 \times 0.25 + 0.3 \times 0.6 \times 0.75 = 0.24$
n	$\neg e$	$0.7 \times 0.6 \times 0.7 + 0.3 \times 0.6 \times 0.3 = 0.348$

El factor de la variable Alimentación ($P(A)$) no depende de ninguna variable, al ser una variable oculta generamos todas las posibilidades

$$f_A(A) = \begin{array}{c|c} A & \\ \hline e & 0.4 \\ \neg e & 0.6 \end{array}$$

Ahora debemos acumular todos los factores calculados

$$f_{AFI\bar{S}}(F, A) = f_A(A) \times f_{FI\bar{S}}(F, A) =$$

F	A	
s	e	$0.26 \times 0.4 = 0.104$
s	$\neg e$	$0.296 \times 0.6 = 0.177$
n	e	$0.24 \times 0.4 = 0.096$
n	$\neg e$	$0.348 \times 0.6 = 0.208$

Y ahora sumamos sobre todos los valores de la variable A para obtener el factor correspondiente a la variable Alimentación

$$f_{\bar{A}FI\bar{S}}(F) = \sum_{A \in \{e, \neg e\}} f_{AFI\bar{S}}(F, A) =$$

F	
S	$0.104 + 0.177 = 0.281$
n	$0.096 + 0.208 = 0.304$

Y por último la variable Deporte ($P(D = n)$) tiene el valor fijado a **no** y dado que no depende de la variable fumador se puede obviar, ya que es un factor constante.

Ahora, si normalizamos a 1

$$P(F|I = s, D = n) =$$

F	
S	0.48
n	0.52

La complejidad del algoritmo de eliminación de variables depende del tamaño del mayor factor, que depende del orden en el que se evalúan las variables y la topología de la red. El orden de evaluación que escogeremos será el topológico según el grafo, pero podríamos utilizar cualquier orden. De hecho se podría escoger el orden que más variables eliminara para hacer que el cálculo sea más eficiente, el problema es que encontrar el orden óptimo es NP.

La complejidad de la inferencia exacta es NP-hard en el caso general. En el caso particular en que la red bayesiana cumple que para cada par de nodos hay un único camino no dirigido (**poliárbol**), entonces se puede calcular en tiempo lineal. Por eso es interesante que cuando se construya una red bayesiana para un problema se construyan poliárboles. Si podemos construir una red que cumpla esta propiedad podemos utilizar este algoritmo sin ningún problema.

Para obtener resultados en el caso general se recurre a algoritmos aproximados basados en técnicas de muestreo. Evidentemente el valor obtenido con estos algoritmos es una aproximación del valor real, pero el coste temporal es razonable.

18.1 Introducción

Un método alternativo para representar la imprecisión es el que presenta el modelo posibilista. Este modelo surge de la llamada lógica posibilista, esta es una lógica no clásica especialmente diseñada para el razonamiento con evidencias incompletas y conocimiento parcialmente inconsistente.

Muchas veces los expertos utilizan términos para hablar de los valores de las variables involucradas en su conocimiento en lugar de valores específicos. Estos términos no representan un valor concreto, sino un conjunto de valores que no tiene por que coincidir con la idea clásica de conjunto.

Habitualmente un conjunto tiene una definición suficiente y necesaria que permite establecer claramente la pertenencia de los diferentes elementos al conjunto. Pero podemos observar que el tipo de información que expresan los términos de los expertos permite que ciertos valores puedan pertenecer a diferentes conjuntos con diferente grado.

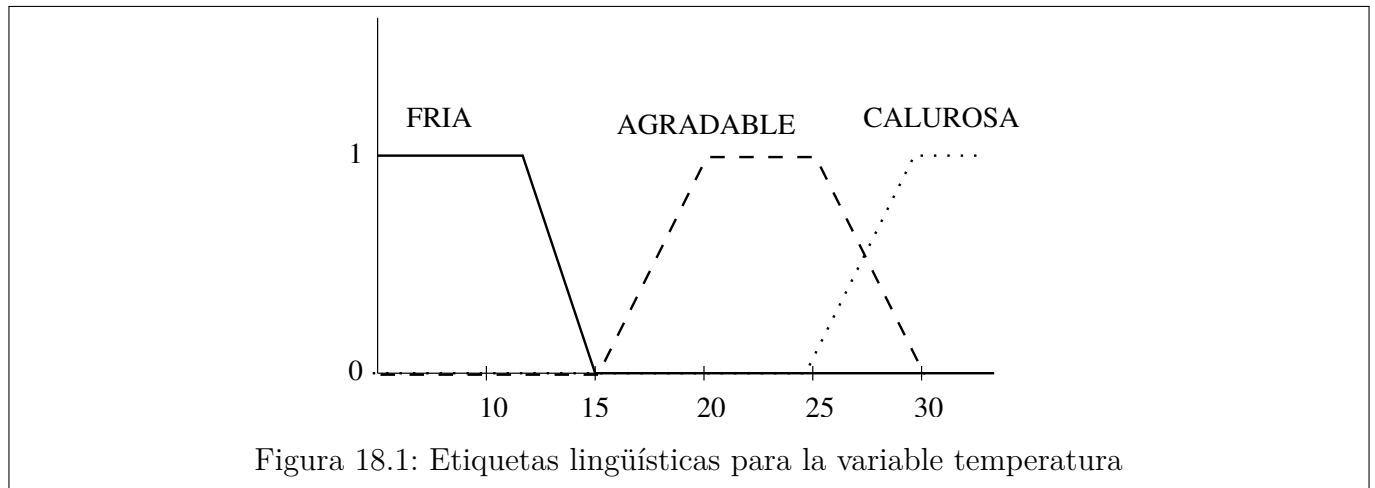
Por ejemplo, un experto puede referirse a la **temperatura** diciendo que es *alta*. La variable *temperatura* estará definida sobre un rango (por lo general continuo) de valores y el término *alta* no tiene por que tener unas fronteras definidas, ya que en un dominio real sería difícil pensar que a partir de un valor específico la temperatura se considera alta, pero que el valor inmediatamente anterior no lo es.

Podríamos decir que la expresión *la temperatura es alta* tiene diferentes grados de verdad dependiendo del valor exacto sobre el que la evaluáramos. Desde el punto de vista del experto esta interpretación es más intuitiva que la interpretación clásica y es la que usa habitualmente para describir ciertos dominios. Este tipo de imprecisión en el lenguaje es la que queremos modelar mediante la lógica posibilista.

De hecho este modelo intenta acercarse más a la forma que tiene el experto de modelar y expresar conocimiento sobre su dominio cuando la información de la que dispone no tiene por que evaluarse directamente a un valor concreto, por lo tanto tenemos no solo incertidumbre sobre la asociación entre hechos y conclusiones, sino también sobre los propios hechos. Esto difiere a lo que hemos visto en el modelo probabilístico en el que para las evidencias que observamos siempre lo hacemos con total seguridad.

En este tipo de representación trabajaremos con variables que están definidas en un *universo del discurso* (conjunto de valores posibles) y que se definirán a partir de *etiquetas lingüísticas* (términos que podemos usar para referirnos a las variables).

18.2 Conjuntos difusos/Lógica difusa



La lógica posibilista se basa en la teoría de los conjuntos difusos y, por extensión, en la lógica difusa. Los conjuntos difusos se han tomado como punto de partida para la representación de la vaguedad del lenguaje, de esta manera, proposiciones como por ejemplo “*la temperatura de hoy es agradable*” supondría que existe un conjunto de temperaturas asignadas al término lingüístico *agradable* que correspondería a un conjunto difuso, y el razonamiento se haría basándose en éste. Esto permitiría utilizar estos conjuntos para representar los razonamientos cualitativos que suelen utilizar los expertos.

Los conjuntos difusos son una generalización de los conjuntos clásicos en los que la pertenencia no está restringida a sí o no, sino que puede haber una gradación de pertenencia en el intervalo $[0, 1]$, determinada por una función de distribución asignada al conjunto.

Los hechos representables mediante conjuntos difusos siguen el esquema $[X \text{ es } A]$ donde X es una variable definida sobre un *universo* U de valores posibles y A es un *término lingüístico* aplicable a X , que restringe los valores que puede tomar. Estos valores corresponderán al conjunto difuso representado por A sobre el universo U . En el ejemplo anterior, X sería la *temperatura*, el universo U serían todos los posibles valores que puede tomar la temperatura y A sería *agradable*, este término indicaría los valores posibles para X . En la figura 18.1 se pueden ver diferentes etiquetas aplicables al concepto temperatura y sus conjuntos difusos correspondientes.

Todo conjunto difuso está representado por una función característica μ_A que define la pertenencia de un elemento al conjunto. A través de ésta se puede definir la función de posibilidad de los valores de X al conjunto A . De este modo, la posibilidad de que X tenga un valor $u \in U$ sabiendo que $[X \text{ es } A]$ viene definida por la función:

$$\pi_A : U \rightarrow [0, 1] \text{ tal que } \pi_A(u) = \mu_A(u)$$

Es decir, la posibilidad para un valor se corresponde con su grado de pertenencia al conjunto difuso representado por la etiqueta lingüística. Un valor de posibilidad se puede interpretar como el grado con el que una proposición es compatible con el conocimiento que se describe con la proposición $[X \text{ es } A]$.

18.3 Conectivas lógicas en lógica difusa

Al igual que se puede interpretar la representación de las proposiciones con una visión conjuntista, ésta se puede también transformar en una visión lógica¹, que es en definitiva lo que se va a utilizar en la deducción con las reglas de los expertos.

Se puede ver la pertenencia total o la no pertenencia de un valor a un conjunto difuso como los valores de verdad *cierto* y *falso* para una proposición (esta sería la noción clásica de valores de verdad), a esto le añadimos el poder tener una gradación de valor de verdad entre estos valores extremos, determinada por la función de posibilidad del conjunto difuso. Con esto tenemos una lógica que permite cualquier valor de verdad entre cierto y falso.

Siguiendo con esta visión de la lógica, podemos establecer las combinaciones que permiten las conectivas de la lógica clásica sobre sus dos valores de verdad se extiendan sobre la gradación de valores de verdad que permite la función de posibilidad de un conjunto difuso.

En lógica de proposiciones disponemos de tres conectivas para la combinación de enunciados: $\wedge$, $\vee$ y $\neg$. Sobre estas, la teoría de modelos establece tres tablas de verdad que indican como se combinan los únicos dos valores de verdad permitidos. La lógica difusa establece tres tipos de funciones equivalentes a estas conectivas que dan la extensión continua de estas combinaciones.

Conjunción difusa

La función que permite combinar en conjunción dos funciones de posibilidad es denominada T-norma y se define como $T(x, y): [0, 1] \times [0, 1] \rightarrow [0, 1]$ y ha de satisfacer las propiedades:

1. Conmutativa, $T(a, b) = T(b, a)$
2. Asociativa, $T(a, T(b, c)) = T(T(a, b), c)$
3. $T(0, a) = 0$
4. $T(1, a) = a$
5. Es una función creciente, $T(a, b) \leq T(c, d)$ si $a \leq c$ y $b \leq d$

Se puede observar que son las mismas propiedades que cumple la conectiva $\wedge$ en lógica de proposiciones, lo mismo pasará con el resto de conectivas. Además, estas propiedades imponen que el comportamiento de la función en sus extremos sea el mismo que los valores de la tabla de verdad de la conectiva $\wedge$.

Ejemplos de funciones que cumplen estas propiedades son:

- $T(x, y) = \min(x, y)$
- $T(x, y) = x \cdot y$
- $T(x, y) = \max(0, x + y - 1)$

Se puede ver la forma de estas funciones en la figura 18.2.

¹Los conjuntos difusos y la lógica difusa se pueden ver como una extensión continua de la teoría de conjuntos y la lógica clásica. Por lo tanto si la teoría de conjuntos se puede ver como una variante notacional de la lógica de predicados, su extensión continua también.

Disyunción difusa

La función que permite combinar en disyunción dos funciones de posibilidad es denominada T-conorma y se define como $S(x, y): [0, 1] \times [0, 1] \rightarrow [0, 1]$ y ha de satisfacer las propiedades:

1. Conmutativa, $S(a, b) = S(b, a)$
2. Asociativa, $S(a, S(b, c)) = S(S(a, b), c)$
3. $S(0, a) = a$
4. $S(1, a) = 1$
5. Es una función creciente, $S(a, b) \leq S(c, d)$ si $a \leq c$ y $b \leq d$

Ejemplos de funciones que cumplen estas propiedades son:

- $S(x, y) = \max(x, y)$
- $S(x, y) = x + y - x \cdot y$
- $S(x, y) = \min(x + y, 1)$

Se puede ver la forma de estas funciones en la figura 18.2.

Estas tres funciones de T-conorma, junto a las tres anteriores T-normas, corresponden a tres pares de funciones duales respecto a las leyes de DeMorgan y son las más utilizadas. El ser dual respecto a las leyes de DeMorgan es una característica importante ya que permite preservar propiedades de la lógica de proposiciones en el comportamiento de las conectivas difusas.

Negación difusa

La función que permite negar una función de posibilidad es denominada *negación fuerte* y se define como $N(x): [0, 1] \rightarrow [0, 1]$ y ha de satisfacer las propiedades:

1. $N((N(a))) = a$
2. Es una función decreciente, $N(a) \geq N(b)$ si $a \leq b$

Ejemplos de funciones que cumplen estas propiedades son:

- $N(x) = 1 - x$
- $N(x) = \sqrt{1 - x^2}$
- $N_t(x) = \frac{1-x}{1+t \cdot x} \quad t > 1$

Se puede ver la forma de estas funciones en la figura 18.2.

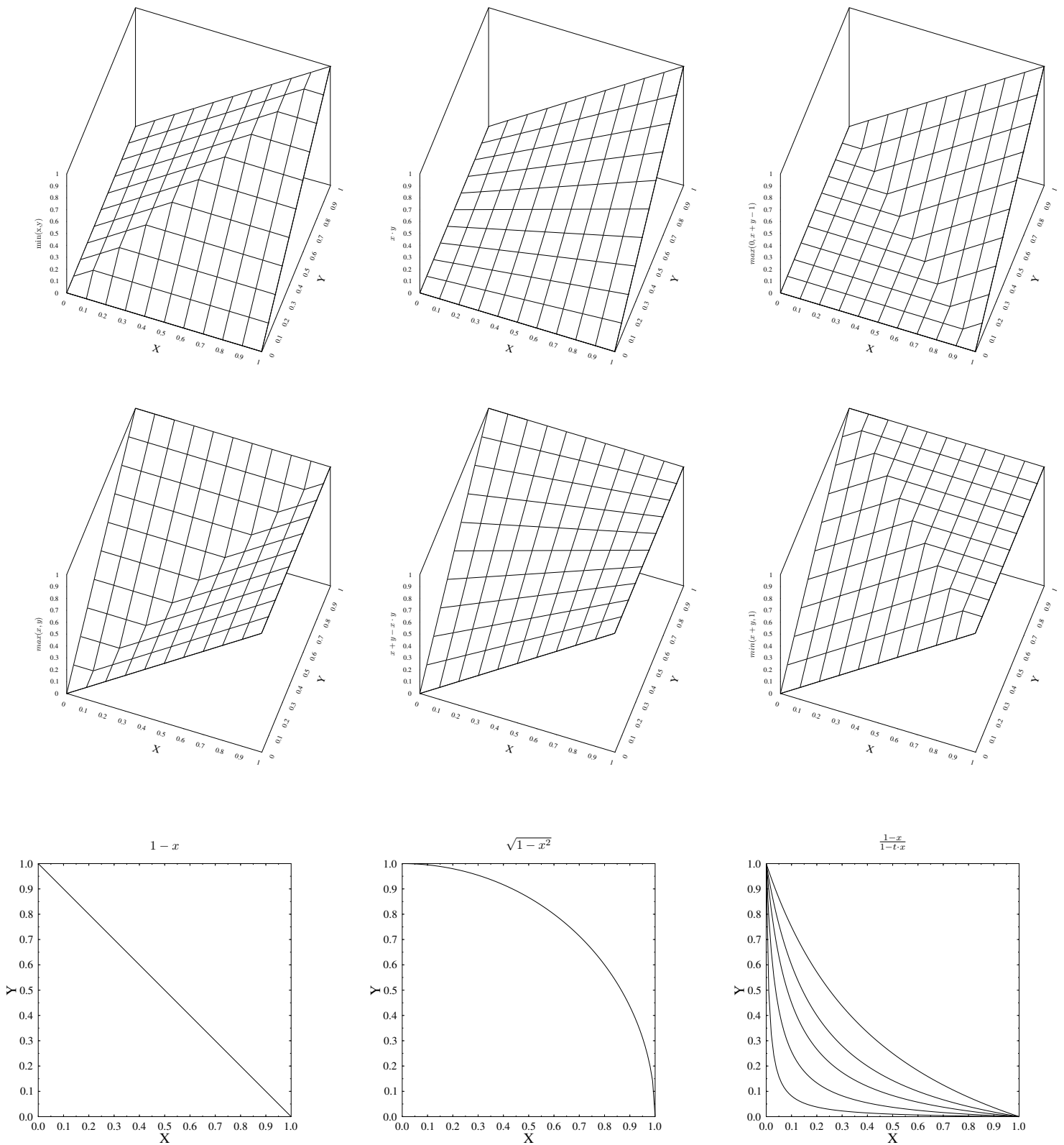
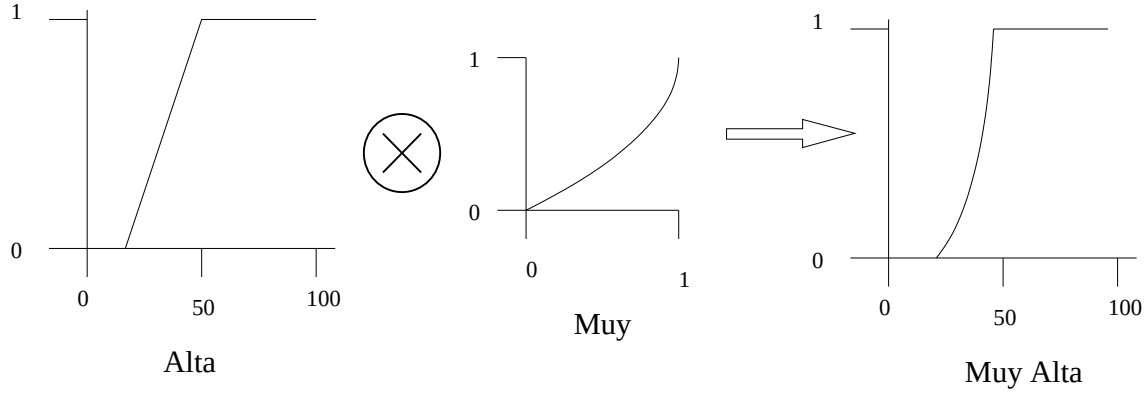


Figura 18.2: Funciones T-norma, T-conorma y negación

Modificadores difusos

Algo que también nos permite la lógica difusa es añadir modificadores a los términos lingüísticos que utilizamos de la misma manera en que lo haría el experto. Por ejemplo, podríamos decir que *la temperatura es muy alta* y utilizar una función que represente la semántica del modificador *muy* como una función que se combina con la etiqueta, donde el conjunto difuso *muy alta* sería el resultado de aplicar el modificador **muy** al conjunto difuso *alta*. Los modificadores están definidos como funciones $[0, 1] \rightarrow [0, 1]$ y varían el valor de verdad del conjunto difuso original.



18.4 Condiciones difusas

Mediante las funciones de posibilidad podemos construir combinaciones de proposiciones que permiten crear condiciones a partir de proposiciones difusas simples. El conjunto difuso resultante de la condición dependerá de los conjuntos que estemos combinando y su función característica estará definida sobre uno o más universos del discurso.

De esta manera, si tenemos las proposiciones $F=[X \text{ es } A]$ y $G=[X \text{ es } B]$ y π_A y π_B están definidas sobre el mismo universo U , entonces:

$$\begin{aligned} F \wedge G &= [X \text{ es } A \text{ y } B] \text{ con } \pi_{F \wedge G}(u) = T(\pi_A(u), \pi_B(u)) \\ F \vee G &= [X \text{ es } A \text{ o } B] \text{ con } \pi_{F \vee G}(u) = S(\pi_A(u), \pi_B(u)) \end{aligned}$$

Donde la T-norma y T-conorma se definen sobre una sola dimensión.

En cambio, si tenemos $F=[X \text{ es } A]$ y $G=[Y \text{ es } B]$ con π_A definida sobre U y π_B definida sobre V , con $U \neq V$, tenemos:

$$\begin{aligned} F \wedge G &= [X \text{ es } A] \text{ y } [Y \text{ es } B] \text{ con } \pi_{F \wedge G}(u, v) = T(\pi_A(u), \pi_B(v)) \\ F \vee G &= [X \text{ es } A] \text{ o } [Y \text{ es } B] \text{ con } \pi_{F \vee G}(u, v) = S(\pi_A(u), \pi_B(v)) \end{aligned}$$

Donde la T-norma y T-conorma se definen sobre dos dimensiones.

Podemos ver que las funciones de combinación de funciones de posibilidad generan una nueva función de posibilidad. En definitiva, estamos definiendo un álgebra sobre funciones en las que las conectivas difusas nos sirven de operaciones de combinación.

Veamos dos ejemplos de combinación de proposiciones en ambos casos.

Ejemplo 18.1 Supongamos que tenemos las proposiciones $\{[La \text{ temperatura de hoy}] \text{ es } [agradable]\}$ y $\{[La \text{ temperatura de hoy}] \text{ es } [calurosa]\}$, proposiciones definidas sobre el mismo universo U (el de

las temperaturas posibles) donde las funciones de posibilidad de las etiquetas agradable y calurosa se pueden ver en la figura 18.3.

Podemos utilizar como función de T-norma $T(x,y)=\min(x,y)$ y de T-conorma $S(x,y)=\max(x,y)$ y construir la función de posibilidad que define la conjunción y la disyunción de ambas proposiciones, el resultado se puede ver en la figura 18.4.

También podemos definir la negación de agradable mediante la función de negación fuerte $N(x)=1-x$, el resultado se puede ver en la figura 18.5.

Ejemplo 18.2 Supongamos que tenemos las proposiciones $[\{\text{Juan}\} \text{ es } \{\text{alto}\}]$ y $[\{\text{Juan}\} \text{ es } \{\text{joven}\}]$, proposiciones definidas sobre dos universos diferentes, el de las alturas, y el de las edades. Las funciones de posibilidad de las etiquetas lingüísticas están representadas en la figura 18.6

Podemos utilizar como función de T-norma $T(x,y)=\min(x,y)$ y de T-conorma $S(x,y)=\max(x,y)$ y construir la función de posibilidad que define la conjunción y la disyunción de ambas proposiciones, en este caso, definidas sobre dos dimensiones, ya que las proposiciones corresponden a universos diferentes. La figura 18.7 y 18.8 muestran un mapa de niveles de las etiquetas resultantes y una representación tridimensional.

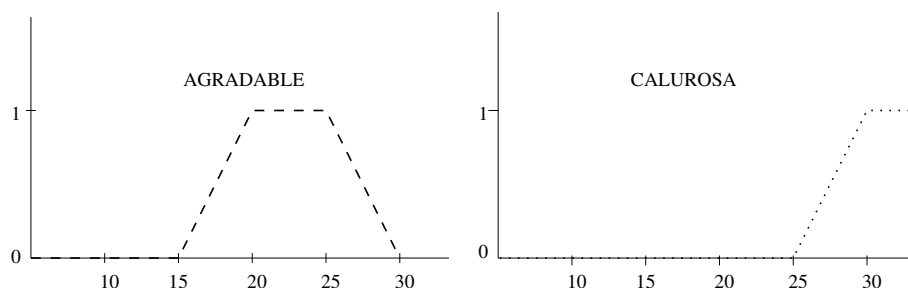


Figura 18.3: Conjuntos difusos agradable y calurosa

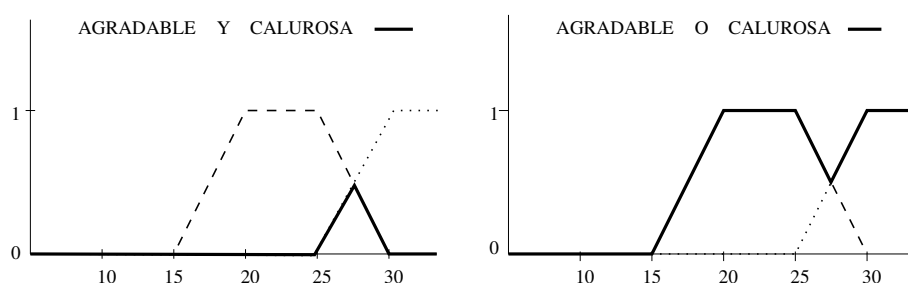


Figura 18.4: Combinación de las etiquetas agradable y calurosa (conjunción y disyunción)

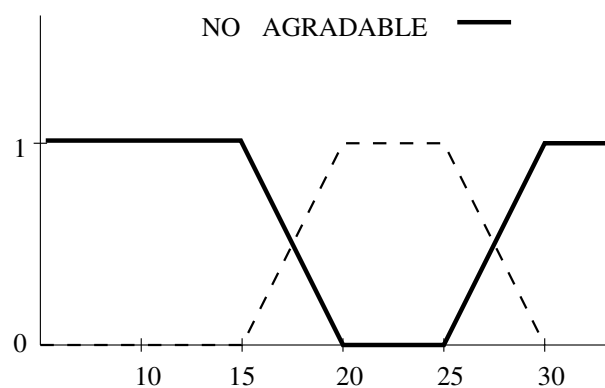


Figura 18.5: Negación de la etiqueta agradable

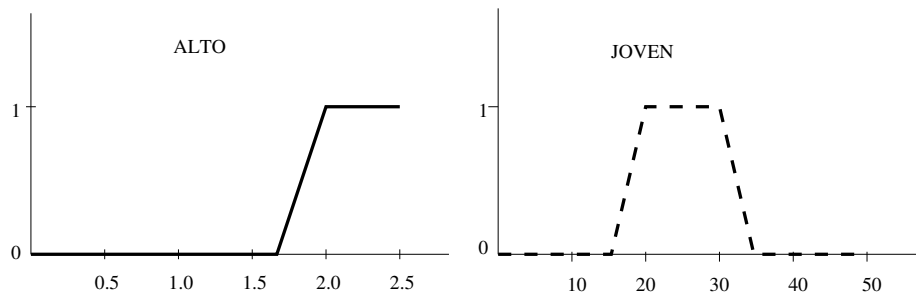


Figura 18.6: Etiquetas difusas alto y joven

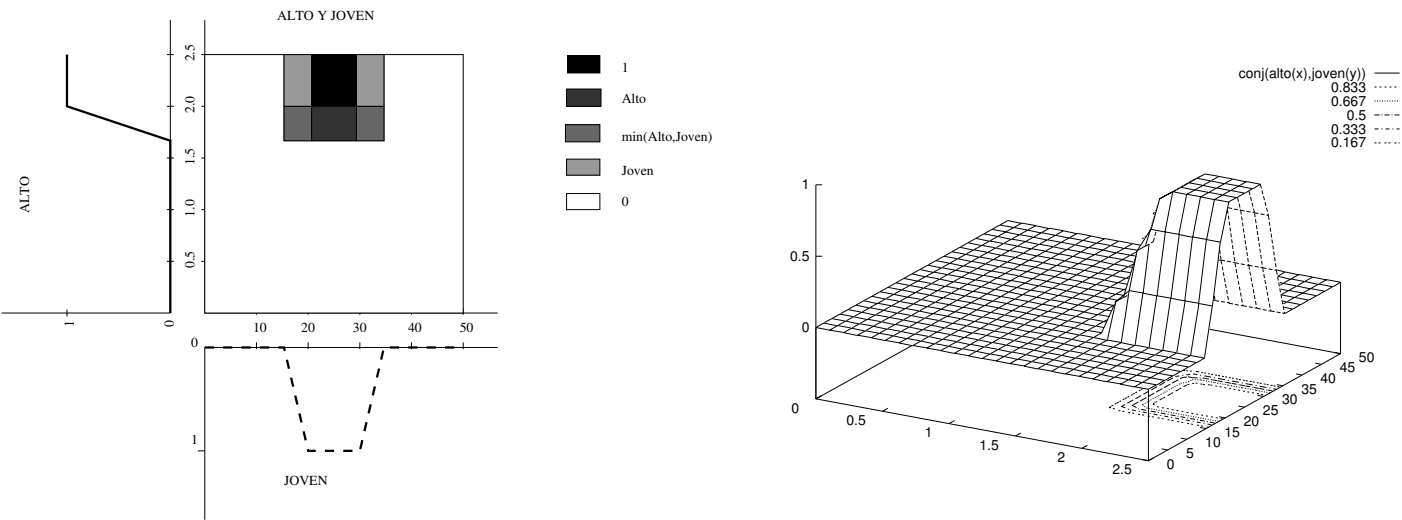


Figura 18.7: Representación de la combinación de las etiquetas alto y joven

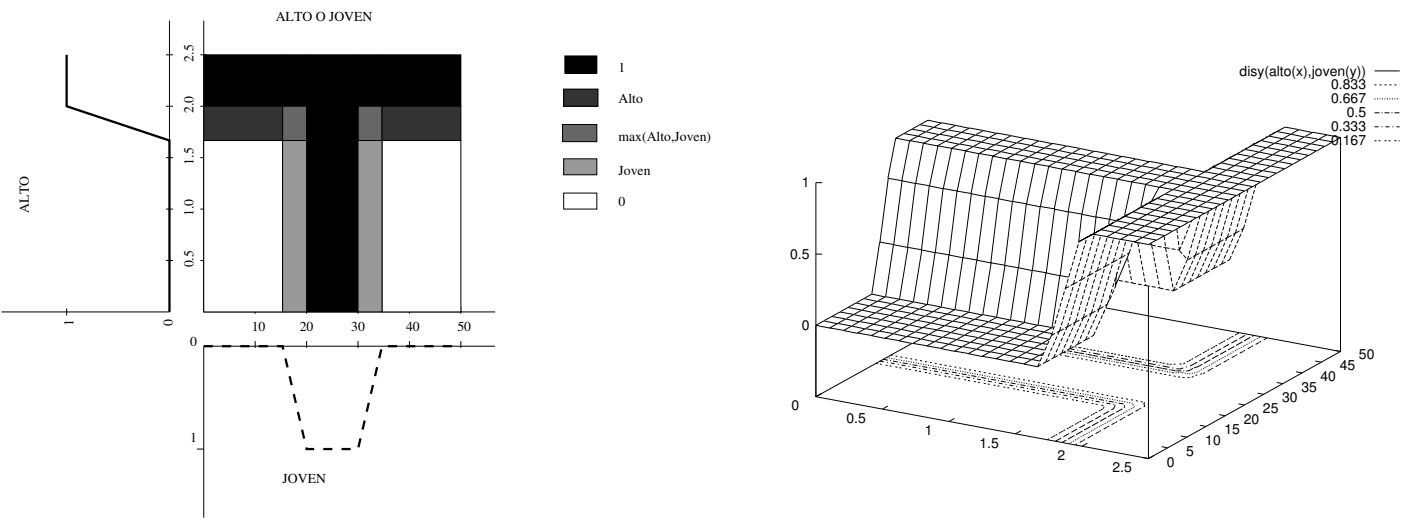


Figura 18.8: Representación de la combinación de las etiquetas alto o joven

18.5 Inferencia difusa con datos precisos

A partir del formalismo de la lógica difusa podemos establecer un modelo de razonamiento incluyendo el resto de operadores de la lógica clásica. El operador que nos permitirá un razonamiento equivalente al que obtenemos con los sistemas de reglas clásicos será el operador de implicación.

Con todos estos operadores podemos escribir reglas utilizando como elementos atómicos expresiones difusas, por ejemplo:

$$[\text{altura}(X)=\text{alto}] \wedge [\text{edad}(X)=\text{joven}] \rightarrow [\text{peso}(X)=\text{medio}]$$

En este escenario nos podemos plantear dos tipos de inferencias. Una primera posibilidad sería que nuestros hechos estén expresados también como hechos difusos, de manera que podríamos utilizar las reglas como el mecanismo de razonamiento habitual. El resultado del razonamiento sería un conjunto difuso que combinara adecuadamente los conjuntos difusos que provienen de los hechos, los antecedentes de las reglas y sus consecuentes. Para ello, se definen las funciones de combinación que corresponden a la implicación y a la regla de deducción del **modus ponens**².

Otra posibilidad es que dispongamos de valores precisos para las variables de nuestro dominio, nuestro objetivo será obtener el valor preciso que se obtiene a partir de las relaciones que nos indican las reglas difusas que tenemos.

Este segundo caso es mas sencillo, supone que tenemos una variable respuesta y un conjunto de variables que influyen directamente sobre esta variable respuesta. El conjunto de reglas difusas nos indica cual es la relación entre los valores de unas variables con otras. Esta variable respuesta puede a su vez influir en otras variables.

Si observamos el formalismo que estamos utilizando, éste nos permite simplificar en gran manera como expresamos las relaciones entre evidencias y conclusiones. Dada una regla difusa, esta tiene en cuenta siempre todos los valores posibles de los antecedentes y se puede utilizar el valor de verdad de cada uno de ellos como un factor de influencia sobre la conclusión, lo que nos permite obtener una respuesta gradual a todos los posibles casos que caen bajo el antecedente de la regla.

Hay dos modelos de inferencia difusa que se utilizan habitualmente, el primero es el modelo de *Mamdani*. Éste asume que la conclusión de la regla es un conjunto difuso y los antecedentes permiten obtener un valor que modifica el conjunto según el valor de verdad del antecedente. El segundo es el modelo de *Sugeno*, éste considera que la conclusión de la regla es una función (por lo general lineal) de los valores del antecedente modificada por el valor de verdad de éste. Nosotros solo nos preocuparemos del primer modelo.

18.5.1 Modelo de inferencia difusa de Mamdani

Dado que los antecedentes de las reglas corresponden a conjuntos difusos, un valor concreto puede disparar varias reglas a la vez, por lo que deberemos evaluarlas todas. Cada una de ellas nos dará un valor de posibilidad que será la fuerza con la que cada regla implica el consecuente.

El proceso de inferencia con datos precisos tendrá los siguientes pasos:

1. **Evaluación de los antecedentes de las reglas:** Esta evaluación pasará por obtener el valor de posibilidad que nos indican los valores precisos para cada elemento del antecedente y su combinación mediante los operadores difusos correspondientes
2. **Evaluación de los consecuentes de las reglas:** Cada uno de los consecuentes se ponderará según el valor de posibilidad indicado por su antecedente. El resultado de esta ponderación será la etiqueta de la conclusión recortada al valor de posibilidad del antecedente.

²La conoceréis de la asignatura de lógica como la eliminación de la implicación $A, A \rightarrow B \vdash B$

3. **Combinación de las conclusiones:** Todas las conclusiones de las reglas se combinarán en una etiqueta que representa la conclusión conjunta de las reglas
4. **Obtención del valor preciso (Nitidificación³):** Se calcula el valor preciso correspondiente a la etiqueta obtenida. Este cálculo se puede hacer de diferentes maneras, una posibilidad es obtener el centro de gravedad de la etiqueta.

El centro de gravedad se calcula como una integral definida sobre la función que describe la etiqueta resultante:

$$CDG(f(x)) = \frac{\int_a^b f(x) \cdot x dx}{\int_a^b f(x) dx}$$

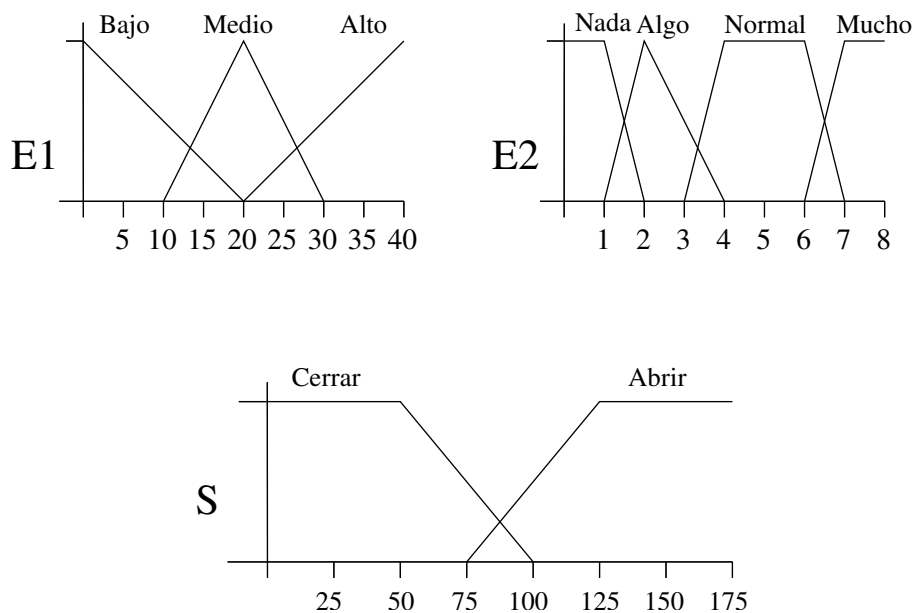
donde a y b son los extremos de la etiqueta. Ésto no es mas que una versión continua de una media ponderada.

Existen otras posibilidades, con un coste computacional menor, para obtener un valor preciso de la etiqueta resultante de la composición de reglas, como por ejemplo, escoger la media de los valores donde la etiqueta sea máxima.

Ejemplo 18.3 Supongamos que tenemos dos variables de entrada $E1$ y $E2$ y una variable de salida S que pueden tomar valores en los conjuntos siguientes de etiquetas lingüísticas:

- $E1 = \{\text{bajo}, \text{medio}, \text{alto}\}$
- $E2 = \{\text{nada}, \text{algo}, \text{normal}, \text{mucho}\}$
- $S = \{\text{cerrar}, \text{abrir}\}$

Estas etiquetas están descritas mediante los siguientes conjuntos difusos



Supongamos que tenemos el siguiente conjunto de reglas:

R1. si $([E1=\text{medio}] \text{ o } [E1=\text{alto}])$ y $[E2=\text{mucho}]$ entonces $[S=\text{cerrar}]$

³Esta es una adaptación del término inglés *defuzzification*

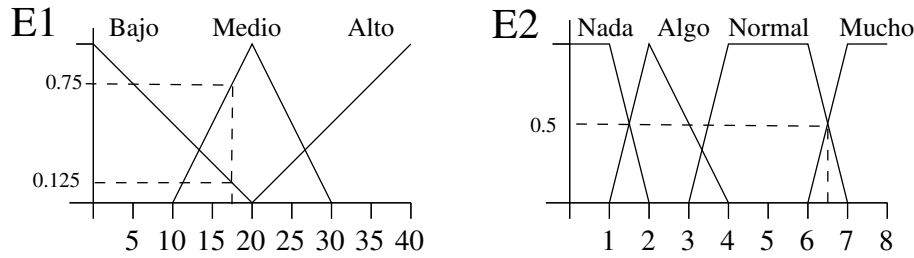
R2. si $[E1=alto]$ y $[E2=normal]$ entonces $[S=cerrar]$

R3. si $[E1=bajo]$ y $no([E2=mucho])$ entonces $[S=abrir]$

R4. si $([E1=bajo] \text{ o } [E1=medio])$ y $[E2=algo]$ entonces $[S=abrir]$

Las funciones de combinación son el mínimo para la conjunción, máximo para la disyunción y $1 - x$ para la negación y los valores concretos que tenemos para las variables $E1$ y $E2$ son 17.5 y 6.5 respectivamente.

Si trasladamos esos valores a los conjuntos difusos de las variables $E1$ y $E2$ obtenemos los siguientes valores de posibilidad



Si evaluamos la regla R1 tendríamos:

$$[E1=medio] = 0.75, [E1=alto] = 0, [E2=mucho] = 0.5 \Rightarrow \min(\max(0.75, 0), 0.5) = 0.5$$

Por lo que tenemos $0.5 \cdot [S=cerrar]$

Si evaluamos la regla R2 tendríamos:

$$[E1=alto] = 0, [E2=normal] = 0.5 \Rightarrow \min(0, 0.5) = 0$$

Por lo que tenemos $0 \cdot [S=cerrar]$

Si evaluamos la regla R3 tendríamos:

$$[E1=bajo] = 0.125, [E1=mucho] = 0.5, \Rightarrow \min(0.125, 1-0.5) = 0.125$$

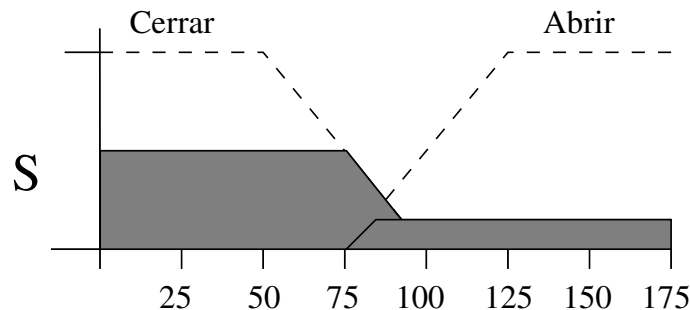
Por lo que tenemos $0.125 \cdot [S=abrir]$

Si evaluamos la regla R4 tendríamos:

$$[E1=bajo] = 0.125, [E1=medio] = 0.75, [E2=algo] = 0 \Rightarrow \min(\max(0.125, 0.75), 0) = 0$$

Por lo que tenemos $0 \cdot [S=abrir]$

La etiqueta resultante de la combinación de las conclusiones sería:



Ahora debemos hallar el centro de gravedad de la etiqueta, ésta se puede describir mediante la función:

$$f(x) = \begin{cases} 0.5 & x \in [0 - 75] \\ (100 - x)/50 & x \in (75 - 93.75] \\ 0.125 & x \in (93.75 - 100] \end{cases}$$

Por lo que podemos calcular el CDG como:

$$\frac{\int_0^{75} 0.5 \cdot x \cdot dx + \int_{75}^{93.75} (100 - x)/50 \cdot x \cdot dx + \int_{93.75}^{175} 0.125 \cdot x \cdot dx}{\int_0^{75} 0.5 \cdot dx + \int_{75}^{93.75} (100 - x)/50 \cdot dx + \int_{93.75}^{175} 0.125 \cdot dx} =$$

$$\frac{0.25 \cdot x^2|_0^{75} + (x^2 - x^3/150)|_{75}^{93.75} + 0.0625 \cdot x^2|_{93.75}^{175}}{0.5 \cdot x|_0^{75} + (2 \cdot x - x^2/100)|_{75}^{93.75} + 0.125 \cdot x|_{93.75}^{175}} =$$

$$\frac{(0.25 \cdot 75^2 - 0) + ((93.75^2 - 93.75^3/150) - (75^2 - 75^3/150)) + (0.0625 \cdot 175^2 - 0.0625 \cdot 93.75^2)}{(0.5 \cdot 75 - 0) + ((2 \cdot 93.75 - 93.75^2/100) - (2 \cdot 75 - 75^2/100)) + (0.125 \cdot 175 - 0.125 \cdot 93.75)} =$$

$$\frac{3254.39}{53.53} = 60.79$$

Parte IV

Tratamiento del lenguaje natural

19.1 Introducción. El Lenguaje Natural y su tratamiento

La capacidad de comunicarse a través de un lenguaje es uno de los rasgos distintivos de la especie humana. Es por ello que el Tratamiento del Lenguaje Natural (TLN) ha constituido desde sus mismos orígenes una parte importante de la Inteligencia Artificial e, incluso, de la Informática en general.

Es absurdo pretender que el TLN esté englobado en la inteligencia artificial lo mismo que pretender que se pueda calificar de una parte de la Lingüística o, incluso, de la Lingüística Computacional. En realidad el TLN utiliza técnicas y formalismos tomados de éstas y otras disciplinas para conseguir sus fines que no son otros que la construcción de sistemas computacionales para la comprensión y la generación de textos en lenguaje natural. Vamos a precisar estos conceptos:

La primera consideración que hemos de hacer es que al hablar de lenguaje natural nos estamos refiriendo al Lenguaje Humano. Es decir, lenguaje natural se opone a Lenguaje Artificial o a Lenguaje Formal. De ahí proviene la primera, importante, dependencia. El lenguaje natural es, si no inabarcable, sí mucho más difícil de abarcar que cualquier Lenguaje Formal.

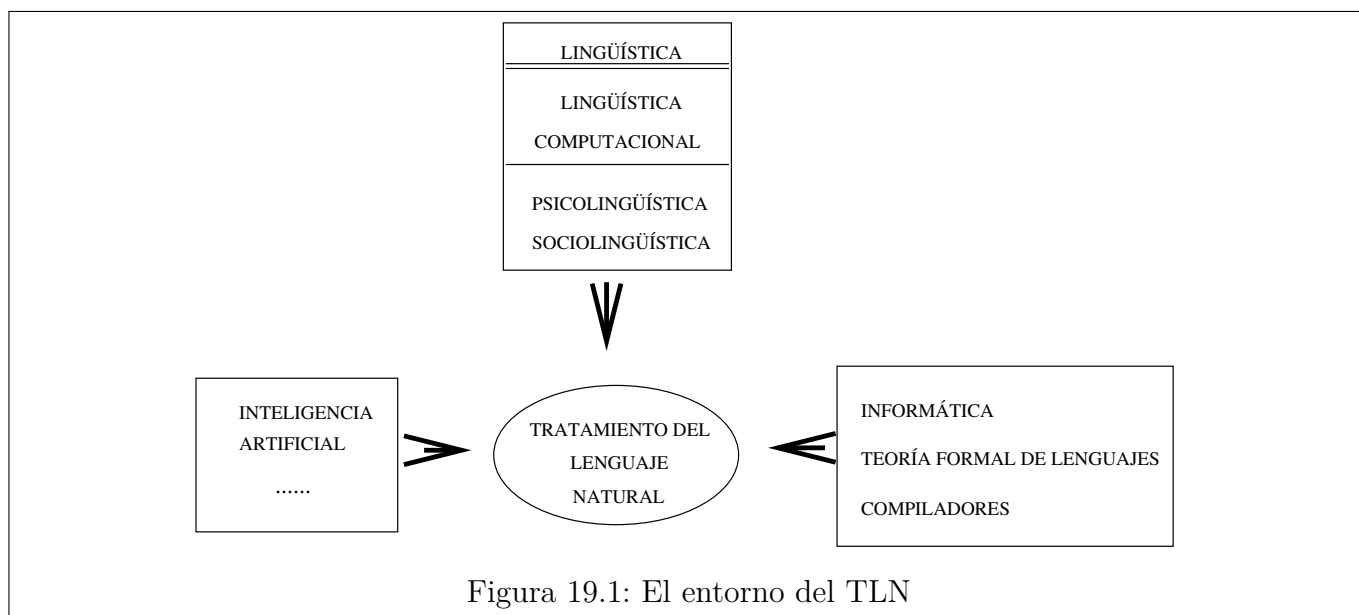
El TLN se ha alimentado durante décadas de las técnicas diseñadas para el tratamiento de los lenguajes artificiales (especialmente de los Lenguajes de Programación) provenientes de la Teoría Formal de Lenguajes o de la Teoría de la Compilación. Por supuesto, el lenguaje natural no es un Lenguaje de programación y de ahí que el TLN se haya convertido en una disciplina diferenciada.

Si el TLN se refiere al Lenguaje Humano, otra disciplina con la que debe tener afinidades es la Lingüística. La Lingüística es, quizás, la más formalizada de las Ciencias Humanísticas. La razón de ello es que el Lenguaje, objeto de su estudio, presenta unas características altamente estructuradas y, por lo tanto, es más susceptible de formalización que otros aspectos de la actividad humana. La Lingüística trata del estudio del Lenguaje tanto en general como de las diferentes lenguas particulares que los seres humanos hablamos. E, incluso, trata de los Sublenguajes: Subconjuntos más o menos cerrados de un lenguaje que permiten un tratamiento formal o computacional más eficientes.

Durante años, el desarrollo de la Lingüística (evidentemente anterior) y el del TLN siguieron caminos casi paralelos con apenas transferencia de resultados de uno a otro.

Los lingüistas construían sus teorías, básicamente sintácticas, sin tener en cuenta la posible realización computacional de las mismas y los informáticos trataban las aplicaciones informáticas del lenguaje natural como un problema más de Ingeniería del Software. No se llegó muy lejos en ninguna de las disciplinas (a pesar de éxitos inicialmente notables en ambas) y finalmente se impuso la sensatez. En la actualidad cualquier propuesta de Teoría Lingüística lleva aparejado el estudio de su componente computacional (básicamente expresada en cómo aplicarla al análisis y/o a la generación del Lenguaje y con qué coste computacional hacerlo). Cualquier aplicación sería del Lenguaje Natural, descansa, por otra parte, en teorías o modelos lingüísticos previamente establecidos.

En cuanto el TLN adquirió un cierto nivel de utilización y las exigencias de calidad aumentaron,



se hizo patente que una adaptación pura y simple de los métodos y técnicas provenientes del Tratamiento de los Lenguajes Artificiales no conducía a nada. Se vio que el proceso de Comprensión (y, más adelante, de Generación) del lenguaje natural implicaba el manejo de una cantidad ingente de Conocimiento de índole diversa y la utilización de procesos inteligentes. La inteligencia artificial proporcionó la base para elevar decisivamente el nivel y las prestaciones de los sistemas de TLN

Un último punto a mencionar, a caballo entre la Psicología y la Lingüística, es el de la Psicolingüística, que estudia los mecanismos humanos de comprensión y generación del lenguaje.

Aquí se reproduce el eterno dilema de la inteligencia artificial, ¿Debe modelizar formas humanas de tratamiento de los problemas o debe limitarse a obtener resultados similares a los que los seres humanos, enfrentados a los mismos problemas, obtendrían? Ciñéndonos a la comprensión o la generación del lenguaje natural, ¿deben las máquinas intentar reproducir las formas humanas de tratamiento, de acuerdo a lo que la Psicolingüística proponga, o deben explorar sus propios métodos de procesamiento, más adecuados a sus peculiaridades? Como es lógico podemos encontrar abundantes ejemplos de uno y otro signo. La figura 19.1 expresa gráficamente estas consideraciones.

19.2 El problema de la comprensión del Lenguaje Natural

El principal problema que plantea la comprensión del lenguaje natural, es decir, el proceso por el cual la máquina es capaz de comprender lo que su interlocutor le dice (oralmente o por escrito), es precisamente su característica de *natural*. El lenguaje natural es el hablado por las personas y, como tal, se rebela a ser apresado en términos de una gramática.

Se han hecho esfuerzos considerables por construir gramáticas de cobertura extensa que describan fragmentos importantes de lenguas naturales (casi todos los esfuerzos que se han llevado a cabo lo son sobre la lengua inglesa) y aunque hay resultados notables como LSP, Diagram, o ALVEY, el lenguaje natural acaba por desbordarlos: surgen, o se descubren, nuevas construcciones sintácticas no previstas inicialmente, se utilizan palabras nuevas y aparece una retahíla de excepciones en cuanto se intenta tratar un nuevo dominio: que si los nombres propios y su tipología, que si las siglas, fórmulas o abreviaturas, que si las jergas, que si las diferencias entre lengua reglamentada y lengua real: faltas ortográficas y gramaticales, usos restringidos de la lengua, barbarismos, *etc*

Queda fuera de este trabajo hacer un sumario de los problemas que el TLN debe resolver para lograr un nivel aceptable de calidad pero es interesante presentar unos cuantos ejemplos:

Ambigüedad léxica:

1. *Se sentó en el banco*
2. *Entró en el banco y fue a la ventanilla*
3. *El avión localizó el banco y comunicó su posición.* ¿Qué tipo de conocimiento hay que utilizar y cómo debemos utilizarlo para conjeturar que (probablemente) la aparición de “*banco*” en (1) se refiere a un mueble que eventualmente sirve para sentarse, mientras que la aparición en (2) se refiere a una oficina en la que una entidad financiera realiza operaciones a través de una ventanilla y la aparición en (3) se refiere, una vez examinado el contexto, a un banco de pesca...?

Ambigüedad sintáctica:

4. *La vendedora de periódicos del barrio*
¿Queremos indicar aquí que la vendedora es del barrio o bien son los periódicos los que son del barrio?
5. *Pedro vió al hombre en lo alto de la montaña con unos prismáticos*
¿Era el hombre o Pedro (o ambos) quien estaba en la montaña?
¿Quién llevaba (o usaba) los prismáticos?
...

Ambigüedad semántica:

6. *Pedro dió un pastel a los niños*
¿Uno para todos?
¿Uno a cada uno?
A lo mejor depende del tamaño (del pastel, no de Pedro ni de los niños)
...

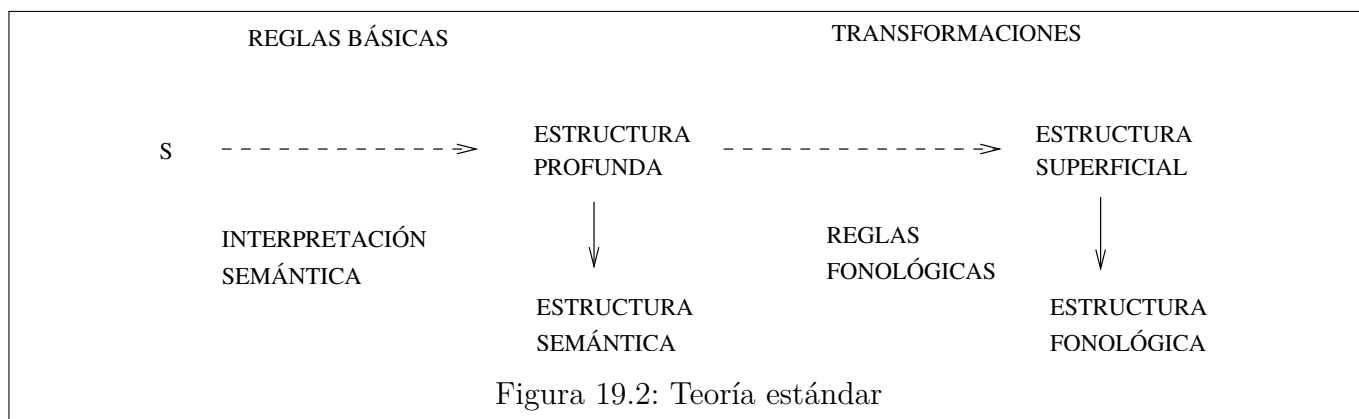
Referencia:

7. *le dijo, después, que lo pusiera encima*
¿Quién dijo?
¿A quién?
¿Cuándo, después de qué?
¿Que pusiera qué?
¿Encima de dónde?
...

Todos estos ejemplos son gramaticales, usan un vocabulario sencillo y no recurren a ninguna licencia especial.

19.3 Niveles de descripción y tratamiento lingüístico

Los ejemplos que hemos propuesto en la sección anterior plantean problemas que aparecen en niveles diferentes de la descripción lingüística y cuya solución requiere conocimientos diversos, a



menudo expresados en otros niveles. Conviene detallar un poco más las características de los diferentes tipos de Conocimiento implicados.

El reconocimiento y utilización de estos niveles lingüísticos corre parejo con el desarrollo de la lingüística computacional como disciplina. Vamos, a continuación, a hacer un breve recorrido por la evolución de la lingüística computacional en los últimos años para, después, definir con más precisión esos niveles y los tratamientos que en cada uno de ellos se llevan a cabo.

19.3.1 Evolución de la lingüística computacional

Hasta comienzos de este siglo el contenido de la Lingüística (que no recibía tal nombre) era básicamente descriptivo o normativo (“en tal lengua, las cosas se dicen de tal manera”, “en tal lengua, la forma correcta de decir las cosas es ésta”).

Se suele considerar el Estructuralismo (Saussure, 1916) como el primer intento serio de introducir la Lingüística como disciplina con un contenido científico. El Estructuralismo, por otra parte, trascendía de lo puramente lingüístico para incorporar contenidos filosóficos y antropológicos.

El hito fundamental, sin embargo, en el nacimiento de la lingüística computacional fue la aportación de Chomsky que en 1965 (*Aspects*) sentó las bases de la Gramática Transformacional Generativa. La Gramática Generativa especifica, en forma precisa, qué combinación de los elementos básicos son permisibles (*gramaticales*) para cada uno de los niveles de descripción lingüística.

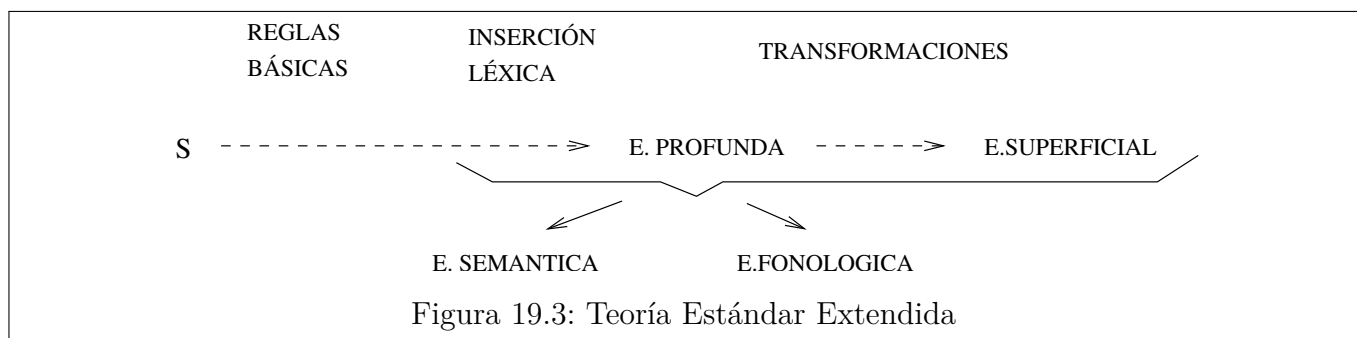
La teoría inicial de Chomsky (la llamada *teoría estándar*) fue propuesta en 1965. Se basaba en la existencia de dos estructuras sintácticas: una profunda (*universal* en la nomenclatura chomskiana) y otra superficial. El paso de una a otra se lograba mediante la aplicación de una serie de transformaciones. A partir de la estructura profunda, una serie de reglas de interpretación semántica producirían la Estructura Semántica mientras que, a otro nivel, las reglas fonológicas traducirían la estructura sintáctica superficial a la Estructura Fonológica, es decir a aquella en que se producían las expresiones del lenguaje (ver la figura 19.2).

El punto fundamental del formalismo residía en la capacidad de, a través de reescritura de términos, generar los elementos de la estructura profunda a partir de un símbolo inicial (axioma) de la gramática. El tipo de cobertura de estas reglas básicas dió lugar a la bien conocida clasificación de las gramáticas (y, a través de ellas, de los lenguajes que aquellas eran capaces de reconocer o generar) en **tipo 0**, **tipo 1** (gramáticas sensitivas), **tipo 2** (gramáticas de contexto libre)¹ y **tipo 3** (gramáticas regulares).

Las limitaciones de la Teoría Estándar en su aplicación práctica, especialmente el limitar la interpretación semántica a la estructura profunda, condujeron a Chomsky a una reformulación de su teoría y a proponer en 1970 la llamada *Teoría Estándar Extendida* (ver figura 19.3).

También de principios de los 70 es la Semántica Generativa (Lakoff, Fillmore). El propio Fill-

¹También llamadas gramáticas incontextuales (Context-free grammars).



morecreó las Gramáticas de Casos, origen de las Redes Semánticas, que tanto éxito han tenido en inteligencia artificial como mecanismo de Representación del Conocimiento.

Aunque la semántica no ha tenido en ningún momento el mismo éxito que la sintaxis en cuanto a proponer formalizaciones lingüísticas (seguramente por la mayor dificultad que supone su manejo) hay que citar aquí la obra de Montague² que se considera en la base de la mayoría de los sistemas de interpretación semántica de tipo compositivo.

A partir de los años 80 se producen nuevos avances, ya claramente identificados con los objetivos de la lingüística computacional

En el campo de la Sintaxis hay que acudir nuevamente a Chomsky que en 1983 propuso su teoría de la *Rección y Ligadura* (*Government and Binding*).

Ésta teoría ha supuesto el punto de partida de una polémica entre los partidarios de las formulaciones sintácticas basadas en la gramática y los que dan más importancia al léxico, reduciendo el papel de la gramática a una serie de principios de buena formación. Parece que los últimos resultados en Teoría Lingüística se inclinan hacia esta segunda opción lexicalista.

Las *Gramáticas de Estructura de Frase Generalizadas* (GPSG), las "Gramáticas Léxico Funcionales" (LFG) o, más recientemente, las *Gramáticas de Estructura de Frase Regidas por el Núcleo* (HPSG, *Head Driven Phrase Structure Grammars*) siguen esta línea.

Paralelamente se han desarrollado formalismos basados en Unificación, a partir, básicamente, de las *Gramáticas de Cláusulas Definidas* (DCG) y se han recuperado otros que habían quedado arrinconados, como las *Gramáticas Catoriales* (CG).

En el campo de la semántica cabe reseñar los estudios sobre Semántica y Ontología (Katz, Fodor), la Semántica Léxica, la Semántica de Prototipos y la Semántica de Situaciones (Perry, Barwise, 1983).

Por último cabe citar que se ha ampliado el campo de cobertura de la Lingüística para incluir, aunque en forma aún incipiente, al menos desde la perspectiva del TLN campos como la Pragmática, el estudio del nivel ilocutivo, la Lingüística textual y discursiva, la Teoría de los diálogos, *etc*

19.3.2 Los niveles de la Descripción Lingüística.

Depende del tipo de aplicación el que deban utilizarse más o menos niveles de descripción y tratamiento lingüísticos. En alguno de los niveles hay amplia coincidencia mientras que otros producen cierta controversia.

Se suelen presentar los niveles de descripción en forma estratificada, comenzando por los más próximos a la realización superficial (voz, frases escritas) y acabando por los más próximos a las capacidades cognitivas de quien produce el lenguaje. Veamos muy brevemente una posible clasificación:

- **Nivel fonético:** Trata de los sonidos a nivel de sus características físicas (frecuencia, intensidad, modulación, *etc*)

² PTQ: "The proper treatment of quantifiers in ordinary English")

- **Nivel fonológico:** Trata de los sonidos desde el punto de vista de la realización de las palabras en forma de voz. Sus unidades son los fonemas.
- **Nivel léxico:** Trata de las palabras en cuanto depositarias de unidades de significado. Sus unidades serían los lexemas. A este nivel se plantean los problemas de segmentación del texto en palabras, la diferencia entre palabra ortográfica y lexema, las lexías, *etc*
- **Nivel morfológico:** Trata de la formación de las palabras a partir de fenómenos como la flexión, la derivación o la composición. Sus unidades son los morfemas.
- **Nivel sintáctico:** Trata de la forma en que las palabras se agrupan para formar frases. Las relaciones estructurales (sintácticas) entre los componentes de la frase son variadas. En los formalismos de tipo sintagmático la estructura sintáctica refleja la propia estructura de constituyentes de la frase (que se denominan sintagmas). Esta estructura se corresponde con el árbol de análisis. Los sintagmas terminales (las hojas del árbol) se corresponden con los elementos de los niveles inferiores de la descripción lingüística de la oración (morfemas, palabras, cadenas de palabras, *etc*).
- **Nivel lógico:** Trata del significado literal de la frase (sin tener en cuenta el contexto). A este nivel se introduce el concepto de forma lógica. Las unidades dependen del formalismo lógico empleado: predicados, funciones, constantes, variables, conectivos lógicos, cuantificadores, *etc*
- **Nivel semántico:** Se dota a la forma lógica de una interpretación semántica, es decir, se conectan los elementos de la forma lógica con los elementos del mundo sobre el que la aplicación informática debe trabajar.
- **Nivel pragmático:** Trata de la forma en que las oraciones se usan (y se interpretan) dentro del contexto en el que se expresan.
- **Nivel ilocutivo:** Trata de las intenciones que persigue quien produce la frase. Se pueden estudiar los actos de habla directos e indirectos. Otros fenómenos ligados a los objetivos, intenciones, planificación de los diálogos, *etc* también se explican a este nivel.

19.4 Aportaciones de la inteligencia artificial

Ya hemos indicado anteriormente que la razón básica para incluir el TLN dentro de la inteligencia artificial es el que la comprensión del lenguaje natural se considera una forma clara de comportamiento inteligente. Hemos de admitir, sin embargo, que buena parte de las técnicas y métodos que se aplican para tratar el lenguaje natural poco o nada tienen que ver con las propias de la inteligencia artificial sino que se importan de disciplinas ligadas al tratamiento de los lenguajes formales.

Sin embargo, y a medida que las exigencias de calidad en el TLN han aumentado, la necesidad de utilizar Fuentes de Conocimiento extensas y complejas y de emplear tratamientos no estrictamente algorítmicos ha dado lugar a una utilización creciente de la inteligencia artificial.

Entre las herramientas de la inteligencia artificial que se emplean extensamente en el TLN encontramos los siguientes:

1. Sistemas de Representación del Conocimiento tanto basados en **Lógica** (usados en los niveles sintáctico y lógico), como en **Redes Semánticas** (Modelos de actantes, Gramáticas de casos) o en Modelos de Objetos Estructurados, **Frames** (Modelos de Representación conceptual y léxica, formas complejas de inferencia, herencia, *etc*).

2. Sistemas de Planificación (Planificación de diálogos, generación en lenguaje natural a partir de planes, generación de explicaciones, *etc*).
3. Sistemas de Búsqueda Heurística (Estrategias de análisis sintáctico, cooperación sintaxis/semántica, *etc*).
4. Sistemas de Razonamiento (Búsqueda de referentes, resolución de anáforas, determinación del ámbito de los cuantificadores, *etc*).
5. Sistemas de representación y razonamiento aproximado e incierto (cuantificadores difusos, información incierta, lógica de modalidades, analizadores probabilísticos, *etc*).

Hemos de tener en cuenta, por otra parte, que buena parte de las aplicaciones del TLN actúan como interfaz de Sistemas Inteligentes en los cuales se integran. Buena parte de los sistemas de representación de conocimiento e información que se utilizan al tratar el lenguaje natural deben tener en cuenta esta doble función.

19.4.1 Aplicaciones del lenguaje natural

Dos son las grandes áreas de aplicación de los Sistemas de TLN: las aplicaciones basadas en diálogos y las basadas en el tratamiento masivo de información textual.

La razón de ello es, obviamente, económica. La interacción persona/máquina se ha convertido en el punto fundamental de la mayoría de las aplicaciones informáticas. El desarrollo de formas cada vez más sofisticadas de dispositivos de interacción (los llamados *multimedia*) y el acceso a la informática de una gama cada vez mayor de usuarios abundan en este interés.

Los volúmenes de información textual que se manejan están creciendo en forma exponencial planteando problemas cada vez mayores de tratamiento. Las necesidades de traducción, formateo, resumen, indicación, corrección, *etc* de cantidades ingentes de textos, con niveles de exigencia de calidad crecientes, ha hecho que se haya vuelto la vista hacia el lenguaje natural como fuente (parcial) de soluciones.

19.4.2 Breve historia del Tratamiento del lenguaje natural

La historia del TLN se remonta a los orígenes de las aplicaciones informáticas a nivel comercial. En 1946, Weaver y Booth presentaron el primer sistema de Traducción Automática, seguido poco después por el GAT ("Georgetown Automatic Translator") y, ya en 1961, por CETA ("Centre d'études pour la Traduction Automatique") en Grenoble.

En 1964 el informe ALPAC, cancelando los fondos para los proyectos de traducción automática en los U.S.A. y negando su viabilidad, supuso un freno para estos primeros intentos pero no impidió el nacimiento de programas tan importantes como METAL o SYSTRAN. Era la época en que el TLN se apoyaba en métodos bastante rudimentarios como los analizadores de pattern matching, utilizados en sistemas como ELIZA, SIR, STUDENT o BASEBALL.

En los años 70 continúan los esfuerzos en traducción automática como SUSY, en Saarbrücken, GETA en Grenoble o TAUM-METEO en Montreal. Paralelamente comienzan a construirse las primeras interfaces en LN, primero con Bases de Datos y más adelante con otras aplicaciones. LUNAR, Rendezvous o LADDER son algunos ejemplos.

En paralelo aparecen sistemas más evolucionados de TLN como las ATN, las Gramáticas Procedurales (Winograd, SHRDLU, 1971) y una larga lista de analizadores de muy diversa índole (GUS, MARGIE, SOPHIE, RUS, SAM, PLANNER, JETS, *etc*).

Ya en los 80 asistimos a la incorporación de los nuevos formalismos sintácticos a los que antes nos referíamos, a la aparición de las gramáticas lógicas y los formalismos de unificación y a la construcción de sistemas cada vez más evolucionados.

Sistemas como Ariane-78, EUROTRA o ATLAS, en el campo de la traducción automática, Interfaces como TEAM, CHAT-80, FRED o DATALOG en el campo de las Interfaces con bases de datos y otras como XCALIBUR, XTRA, INKA en otras aplicaciones completan el panorama de estos años.

¿Qué cabe decir de los 90?

En primer lugar que se han ido incorporando a los sistemas de TLN algunos de los formalismos, sobre todo sintácticos, que se introdujeron como novedad en los 80.

Que, igualmente, han seguido desarrollándose algunos de estos formalismos (las HPSG, por ejemplo, se han convertido en una referencia ineludible y las gramáticas de cláusulas definidas han evolucionado para dar lugar a formalismos más adecuados a la descripción lingüística como las MLG, las DCTG o las gramáticas de rasgos (PATR-II, ALE).

Se ha detectado la necesidad de adquisición masiva de información léxica y gramatical y se han construido sistemas para el aprovechamiento de fuentes de información léxica ya existentes, como los diccionarios para uso humano en soporte informático o los corpórea textuales.

Se han perfeccionado las interfaces en las líneas de mejora de la calidad del diálogo, de las cualidades de transportabilidad de los productos, construcción de sistemas integrados (NAT, FRED) e interfaces multimodales (MMI2, XTRA).

Se han establecido criterios de separación clara entre la descripción de la estructura de la lengua (*Competence*) y el tratamiento, usando esa descripción, de los casos concretos (*Performance*). La descripción debe estar orientada al usuario: lingüista, lexicógrafo, usuario final, *etc* y favorecer, por lo tanto las características de claridad, amplia cobertura, cohesión, reutilización, *etc*. El tratamiento estará orientado a la máquina y deberán, por lo tanto, privar los criterios de eficiencia, tanto espacial como temporal. Los sistemas deberán proveer mecanismos de transferencia entre uno y otro formalismo.

19.4.3 Aplicaciones basadas en Diálogos.

Quizás el sector del TLN que ha adquirido más importancia es el de las Interfaces en lenguaje natural a diversos tipos de aplicación.

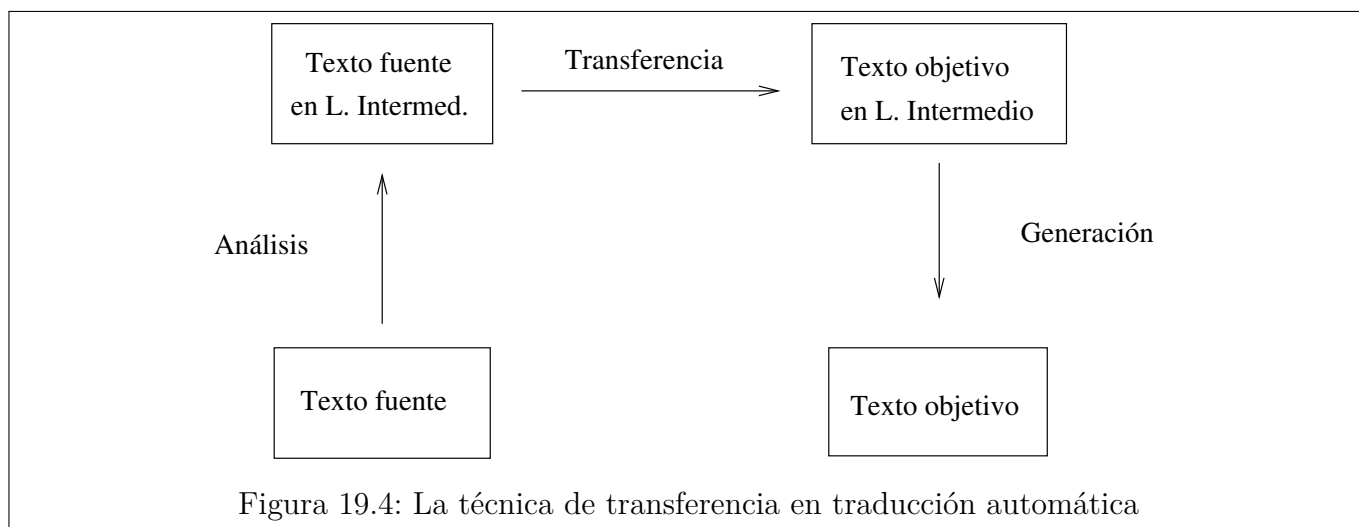
Lo que en principio se reducía a interfaces capaces de consultar en lenguaje natural a una base de datos se ha extendido, tanto en calidad: diálogos cooperativos, admisión (y recuperación) de errores por parte del usuario, interfaces ergonómicas, *etc* como en cobertura: interacción compleja con bases de datos, incluyendo su actualización, interfaz con Sistemas Expertos, tanto a nivel de la adquisición del Conocimiento como de la Explotación del Sistema, acceso a Sistemas Operativos, Sistemas tutores o de asesoramiento, *etc*

Los máximos logros se están produciendo últimamente en el tema de la cooperación inteligente entre formas diversas de interacción (lenguaje natural con forma gráfica, por ejemplo) llegando a la construcción de interfaces multimodales que permiten una comunicación multimedia (lenguaje natural escrito, voz, gráficos, gesto, *etc*) en un entorno cooperativo.

19.4.4 Aplicaciones basadas en el tratamiento masivo de la Información Textual.

La otra gran línea de desarrollo del TLN se refiere al tratamiento de la información textual.

La necesidad de tratamiento de ingentes cantidades de información textual, a la que antes aludíamos, ha llevado a intentar aplicar métodos de tratamiento lingüístico a lo que hasta el momento se consideraban simplemente textos formados por cadenas de caracteres.



La traducción automática es, desde luego, la primera aplicación en este campo (y la más espectacular) pero junto a ella hemos de colocar los sistemas de formateo, resumen e indicación (semi) automáticos de textos, los sistemas de corrección ortográfica, los de partición silábica, los de compaginación, *etc.*

El tratamiento de diccionarios, de corpóra o, en general, la adquisición (semi) automática de información a partir de textos escritos (o hablados) constituyen aplicaciones en las que se trabaja intensamente.

19.5 Técnicas de comprensión del Lenguaje Natural: Su integración

El TLN tiene dos aspectos: la Comprensión del texto y la Generación.

Un diálogo en lenguaje natural entre dos interlocutores adopta, generalmente, la forma de un proceso de intervenciones alternadas de los interlocutores. En cada intervención, el hablante *genera* una expresión en lenguaje natural, de acuerdo con las intenciones que persiga. El oyente, por su parte deberá tratar de *comprender* la expresión para reaccionar de acuerdo a ella.

La arquitectura más empleada en traducción automática es la de *Transferencia*. En ella el texto expresado en la Lengua fuente (u origen) es *analizado* hasta un nivel de representación interna predefinido. En ese nivel la expresión es *transferida* al nivel correspondiente de la lengua objetivo (o destino). Desde allí se producirá, posteriormente la *generación* de la expresión superficial en la lengua objetivo (ver la figura 19.4).

Es decir, en dos aplicaciones paradigmáticas del lenguaje natural encontramos con arquitecturas semejantes, basadas en los dos componentes que antes mencionábamos. La atención que normalmente se presta a estos dos componentes es tremendamente desigual³. Las razones son varias: una mayor *visibilidad* de la comprensión del lenguaje, formas alternativas a la generación igualmente aceptables (partiendo de la base de la mayor capacidad de adaptación del interlocutor humano que de la máquina), mayor integración de la Generación con la aplicación informática subyacente, *etc.* El resto del capítulo se centrará en la comprensión del lenguaje.

¿Qué implica comprender una expresión en lenguaje natural?

Lo primero que debemos plantear es la extensión de lo que tratamos de comprender y el ámbito (el contexto) en el que hacerlo. En los sistemas basados en diálogos la unidad suele ser la intervención

³De hecho la mayoría de los libros sobre TLN omiten el problema de la Generación o le dedican una atención marginal.

del interlocutor humano, una frase más o menos compleja. En los sistemas de tratamiento textual la unidad es a veces la oración y en otras ocasiones el párrafo.

El contexto depende, asimismo, del tipo de aplicación. En un diálogo, la propia historia del proceso de la conversación, las diferentes intervenciones de los interlocutores va proporcionando la estructura en la que deben interpretarse las oraciones. Es imposible interpretar adecuadamente la expresión “*dímelo*” si antes no se ha especificado, directa o indirectamente, algo susceptible de ser dicho por nuestro interlocutor.

Por supuesto el establecimiento y organización de este contexto no es tarea sencilla. Temas como el *Conocimiento de sentido común*, el *Conocimiento general del mundo* o el *Conocimiento del dominio* tienen aquí importancia decisiva.

Una vez delimitada la unidad a analizar y el contexto en el que llevar a cabo el análisis, tiene lugar el proceso de comprensión. La manera más simple de llevarlo a cabo es en cascada, mediante una serie de transductores que van pasando el texto a través de los diferentes niveles de descripción lingüística: análisis morfológico, análisis léxico, análisis sintáctico, interpretación semántica, *etc*

La manera de representar estos niveles y los mecanismos de transducción entre ellos puede estar ligada al propio nivel o ser homogénea (es frecuente, por ejemplo, en los formalismos lógicos el empleo de notaciones homogéneas). La ventaja de tal proceder es una mayor modularidad e independencia en los procesos. El inconveniente básico es que, a menudo, la solución de los problemas lingüísticos de un nivel no se puede realizar en dicho nivel o sólo con el conocimiento que dicho nivel posee sino que precisa de otros (el caso más claro es la resolución de ambigüedades sintácticas mediante la utilización de información semántica).

Una alternativa es el uso de mecanismos de cooperación entre los diferentes niveles. Otra es el empleo de procedimientos de comprensión globales (no estratificados).

Especial interés presentan los sistemas en los que se mantiene una diferencia formal en la expresión del Conocimiento Lingüístico en los diferentes niveles, pero en los que toda esta información es *compilada* y tratada internamente de un modo uniforme.

20.1 Introducción

El componente léxico es fundamental en cualquier sistema de TLN. En última instancia las frases que el sistema debe comprender están formadas por palabras y en ellas se deposita la información (morfológica, sintáctica, semántica) que se precisará en procesos posteriores.

El tratamiento léxico no es sencillo, a pesar de lo que en una primera aproximación pudiera parecer (de hecho, y en esto el TLN no se diferencia de otras disciplinas, construir un pequeño lexicón para una aplicación de juguete es trivial, construir uno real para una aplicación real no lo es). La razón de ello es doble:

1. Por una parte tenemos los problemas intrínsecos al propio léxico: segmentación (e identificación) de las palabras, homonimia (multiplicidad de acepciones), polisemia, desplazamientos de significado (por ejemplo, los usos metafóricos de las palabras), *lexías*, locuciones, *etc*
2. Por otra parte los problemas ligados al volumen de la información léxica necesaria: representación, compacidad, redundancia, acceso eficiente, adquisición, *etc*

Dividiremos esta sección en tres subsecciones que tratarán, respectivamente, de la información léxica, los diccionarios (principales depositarios de dicha información) y la morfología.

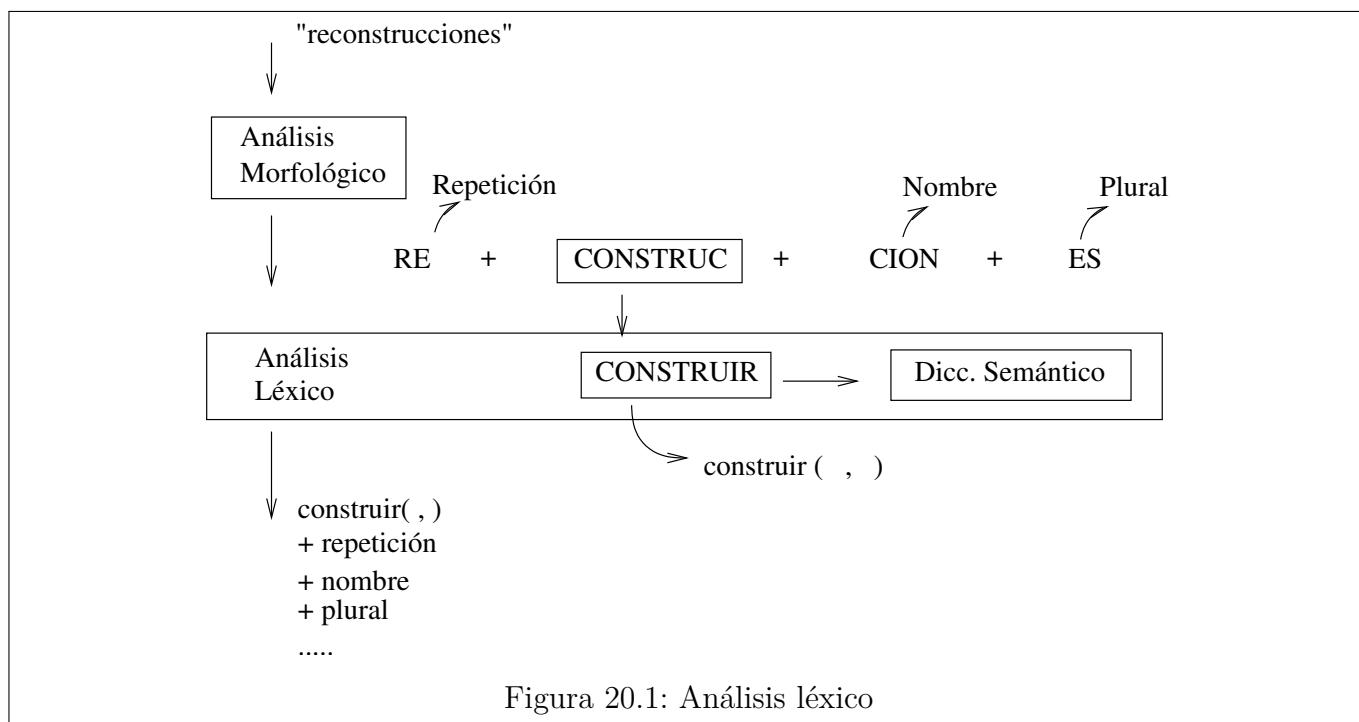
20.2 Descripción de la Información y el Conocimiento Léxicos

Si la función del nivel léxico en la descripción lingüística de una oración (de una secuencia de palabras) es la de asignar información a las unidades elementales de la frase, la Fuente de Conocimiento básica será el Lexicón.

Un Lexicón es, simplemente, un repositorio de información léxica. El modelo de representación más sencillo es una tabla cuyo dominio está constituido por las unidades léxicas (*lexemas*) y cuyo rango consiste en la información necesaria asociada a cada entrada léxica.

El lexicón actúa en:

1. Desplegado de la flexión, composición y derivación morfológicas.
2. La asignación de la categoría (simple o compleja) sintáctica.
3. La asignación de propiedades de diverso tipo (morfológico, sintáctico, semántico).
4. La invocación del análisis semántico de nivel léxico (el asociado a la semántica léxica).



20.2.1 La segmentación de la oración

El proceso de análisis léxico comienza por la segmentación del texto en unidades (palabras). Si identificáramos las palabras ortográficas con las unidades léxicas el proceso no plantearía dificultad alguna: Podemos definir, simplemente, la palabra ortográfica como una secuencia de caracteres delimitada por espacios o signos de puntuación. Construir un segmentador, en este supuesto, es, pues, trivial.

Ahora bien, esta simplificación no es aceptable: Existen palabras gramaticales que se realizan ortográficamente en más de una palabra (por ejemplo, “*sin embargo*”, “*no obstante*”). Existen palabras ortográficas que contienen más de una palabra gramatical (por ejemplo, “*del*” = “*de el*”, “*dámelo*” = “*da me lo*”).

Podemos discutir si “*amó*”, “*amar*”, “*amaría*” son palabras diferentes o formas léxicas diferentes que corresponden a la misma palabra (o *lema*, o *lexema*), “*amar*”.

Se plantea el problema de la polisemia (palabras con varios significados), el de la homonimia (palabras diferentes con la misma grafía): “*separado*” puede ser nombre, adjetivo o participio, “*banco*”, nombre, admite, como vimos en la sección 19.1 de este capítulo, varias acepciones, *etc*

Una aproximación aceptable es la de identificar el lexema con la unidad de información léxica no sintáctica (es decir, básicamente semántica) en tanto que los posibles afijos que modificarían al lexema para construir la forma léxica aportarían el resto de la información. La figura 20.1 nos muestra el proceso.

20.2.2 El contenido de la información léxica

¿Qué cabe incorporar a las unidades léxicas como resultado del Análisis Léxico y, por lo tanto, qué información debe incluirse en el lexicon?

1. Categorización Sintáctica. Se trata, por supuesto de un etiquetado dependiente del tipo de formalismo sintáctico utilizado. Existen categorías cerradas (preposición, determinante, *etc*) y otras abiertas (nombre, adjetivo, *etc*). A veces es importante subcategorizar (verbo transitivo, verbo pronominal, *etc*).

2. Propiedades sintácticas de concordancia, como el género, el número, la persona, el caso, *etc*
3. Otras propiedades sintácticas, como las restricciones selectivas (tipo de argumentos que una palabra, normalmente un verbo, admite), el tipo de complementos que una palabra rige, las preposiciones que acepta, *etc*
4. La información morfológica, como el patrón de formación de la palabra.
5. La información semántica, como la categoría semántica, la forma lógica asociada, los rasgos semánticos, *etc*

Por supuesto no todas las combinaciones ni el tipo de información ni los valores posibles son admisibles. Si, por ejemplo, la categoría es un verbo, la concordancia no incluirá seguramente el género (excepto para algunos tiempos); si la categoría es una preposición, no existirán restricciones selectivas, *etc*

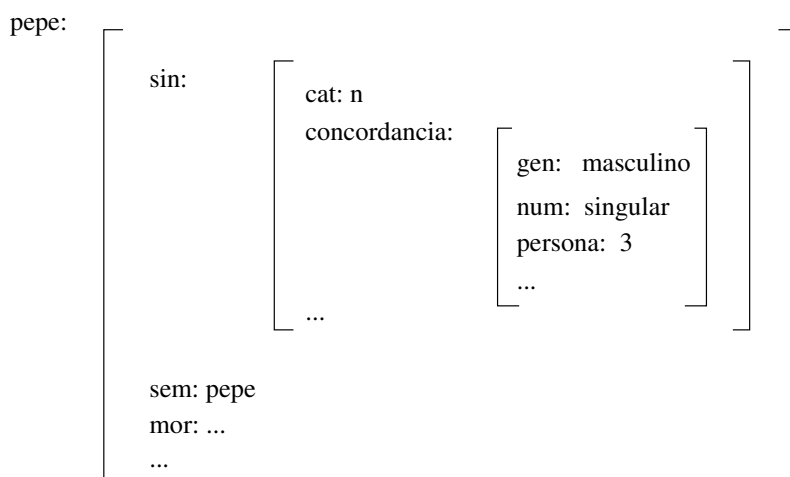
Por ejemplo, para los verbos podemos tener la siguiente información:

1. Aridad: Número de argumentos.
2. Argumentos internos y externos.
3. Opcionalidad de cada uno de ellos.
4. Restricciones selectivas sobre los argumentos:
 - a) Sintácticas (p. ej. preposiciones regidas).
 - b) Semánticas (características exigibles al argumento: animado, humano, contable, medible, *etc*
5. Casos de los argumentos.
6. Tipo aspectual del verbo (suceso, estado, proceso).
7. Categoría (tipo) de la información.

Una manera bastante expresiva de representar la información léxica la constituyen las matrices de rasgos.

Las matrices de rasgos (*Feature Structures (FSs)*) son esencialmente listas de atributos a los que se asocian valores. Los valores pueden ser atómicos o nuevamente FSs. Normalmente la estructura resultante no es un árbol ya que podemos identificar los valores correspondientes a dos o más atributos distintos. En este sentido se suele hablar de estructuras reentrantes o simplemente DAGs (*Directed Acyclic Graphs*).

Veamos algún ejemplo:



en un formalismo del estilo de PATR-II (Schieber) esta estructura se podría expresar como:

word pepe:

```
<sin cat> = n
<sin concordancia gen> = masculino
<sin concordancia num> = singular
<sin concordancia persona> = 3
<sem> = pepe
...
```

Organización del Lexicón

Un formalismo de representación léxica como las FSs proporciona un nivel de expresividad adecuado pero presenta limitaciones graves cuando se trata de aplicarlo a un lexicón real, y por lo tanto voluminoso). Conviene, pues, incorporar a los lexicones mecanismos para:

1. Reducir la redundancia
2. Capturar y expresar generalizaciones
3. Realizar con facilidad la interfaz con las aplicaciones

Veamos, brevemente, alguno de estos mecanismos:

■ Convenciones abreviatorias

Suponen el uso de símbolos que se expanden, a la manera de las macros de algunos lenguajes de programación, permitiendo una descripción más concisa.

La mayoría de los lexicones admiten mecanismos de este estilo. Podemos citar, por ejemplo, los clusters de rasgos del LSP de Sager (ver), los *alias* de Gazdar en GPSG o las *plantillas* de Schieber en PATR-II. Veamos un ejemplo de estas últimas:

let nombre-masc-sing **be**:

```
<sin cat> = n
<sin concordancia gen> = masculino
<sin concordancia num> = singular
<sin concordancia persona> = 3.
```

Con esta declaración previa la definición de "pepe" se reduciría a:

word pepe:

```
nombre-masc-sing
<sem> = pepe.
```

■ Herencia

Diferentes mecanismos de herencia se han venido utilizando en la construcción de lexicones: herencia simple, herencia múltiple, herencia monotónica, herencia por omisión e incluso combinaciones entre ellas o formas más complejas de herencia. Schieber, en PATR-II, permite la herencia simple de propiedades:

let verbo **be**:

```
<sin cat> = v
<sin suj cat> = gn
<sin suj caso> = nominativo.
```

let vt be:

verbo
 <sin obj1 cat> = gn
 <sin obj1 caso> = acusativo.

let vdat be:

vt
 <sin obj2 cat> = gprep
 <sin obj2 prep> = a.

De forma que **vdat** heredaría las propiedades de **vt**, que, a su vez, lo habría hecho de **verbo**.

Haciendo uso de estas declaraciones previas podríamos definir, en forma más concisa, las entradas léxicas de tipo verbal:

word reir:

verbo
 <sem pred> = reir
 <sem arg1> = humano.
 (alguien ríe)

word dar:

vdat
 <sem pred> = dar
 <sem arg1> = humano
 <sem arg2> = cosa
 <sem arg3> = humano.
 (alguien da algo a alguien).

Un sistema parecido al anterior es el de los *Feature Networks* de Flickinger, del que presentamos un ejemplo. En este caso la herencia es múltiple.

\$VERB	
\$FINITE	ISA \$VERB
\$THIRD-SING	ISA \$FINITE
	ISA \$SINGULAR
	ISA \$THIRDPERSON
\$MAIN-VERB	ISA \$VERB
\$TRANSITIVE	ISA \$MAIN-VERB
eats	ISA \$TRANSITIVE
	ISA \$THIRD-SING

■ Transformaciones Léxicas

Se trata de reglas que permiten derivar entradas léxicas no presentes explícitamente en el lexicón a partir de las si existentes mediante la aplicación de determinados procedimientos (normalmente reglas de producción). La aplicación de la regla permite modificar parte del contenido de la entrada léxica fuente para incorporarlo al resultado.

La utilización de reglas léxicas permite obviar la presencia de un módulo específico de Análisis Morfológico para las lenguas (como el inglés) en las que la flexión o derivación morfológicas no plantean problemas graves.

Son numerosos los sistemas de tratamiento léxico que incorporan reglas léxicas (Kaplan, Bresnan, Ritchie, Jackendoff, Copestake, *etc*). Más adelante se describe brevemente uno de tales sistemas.

■ Mecanismos ligados al formalismo sintáctico utilizado

Dependiendo del formalismo sintáctico utilizado se pueden incorporar al Tratamiento Léxico mecanismos de simplificación específicos. Suele tratarse de mecanismos ligados a formalismos sintácticos fuertemente lexicalistas.

En el caso de las GPSG, por ejemplo, algunos de los principios de buena formación se aplican sobre la información léxica restringiéndola o ampliándola. El FSD (Feature Specification Default), que asigna valores por defecto a algunas construcciones, o el FCR (Feature Cooccurrence Restriction), que establece restricciones de coocurrencia de determinados atributos son alguno de ellos.

■ Otros mecanismos

Existen, finalmente, mecanismos ligados a la propia aplicación informática de la que el sistema de TLN es componente.

20.2.3 Lexicones Computacionales

Toda aplicación de TLN incorpora el uso de algún lexicón. En la mayoría de los casos el lexicón se desarrolla expresamente para la aplicación que lo utiliza. Se han desarrollado, sin embargo, algunos lexicones computacionales de uso general. Entre otros podemos citar el BBN-CFG de R.Ingria, el IRUS de M. Bates o el incorporado al programa ALVEY. Uno de los más conocidos es debido a Ritchie. Vamos a describirlo brevemente:

El sistema de Ritchie se basa en la existencia de un **lexicón** básico y una colección de **reglas**. El efecto de la aplicación de las reglas es el de combinar la información (posiblemente nula) que el lexicón asocia a la entrada léxica con la aportada por las reglas aplicadas. El efecto es incremental de forma que las sucesivas reglas que se aplican con éxito van enriqueciendo la información asociada a la entrada¹.

El lexicón contiene entradas diferentes para los diferentes morfemas (raíz, afijos) que constituyen la palabra. La representación utilizada es la de listas de pares <atributo-valor>.

El sistema soporta dos tipos de reglas: las Léxicas y las Morfológicas. Las primeras, como vimos arriba, derivan entradas léxicas a partir de otras. Las reglas morfológicas establecen combinaciones válidas de morfemas.

La forma de las reglas léxicas es la siguiente:

<precondición> <operador> <acción>

donde la <precondición> establece el patrón a comparar con la información de que se dispone en un momento dado para permitir la ejecución de la regla. La <acción> determina las operaciones que permitirán derivar la información de salida a partir de la de entrada y de la proporcionada por la propia regla. El <operador> establece el tipo de relación entre estructura fuente (de entrada) y objeto (de salida). Ritchie incorpora los siguientes tipos de reglas léxicas:

1. Reglas de compleción. Se trata de reglas que modifican, generalmente enriqueciéndolas, las estructuras de rasgos de entrada.
2. Reglas de multiplicación. Se trata de reglas que permiten la creación de nuevas estructuras de rasgos a partir de las ya existentes.

¹Este mecanismo recuerda el funcionamiento de los sistemas de producción.

3. Reglas de consistencia. Se trata de reglas que permiten verificar la consistencia de los rasgos presentes en una estructura. Es decir no crean ni añaden nada, simplemente validan que lo existente hasta el momento es consistente.

La siguiente, por ejemplo, es una regla de completación que permite añadir, a una estructura de rasgos que carezca de él, el atributo “BAR”, asignándole el valor 1. En el ejemplo, “fix” y “bar” son constantes (literales), “_fix” y “_rest” son variables, “_” es la variable anónima y “~” indica la negación (la no aparición de determinado símbolo).

$$\begin{aligned} &(_ ((\text{fix } _ \text{fix}) \sim (\text{bar } _) _ \text{rest}) _) \\ \Rightarrow &(\&\& ((\text{fix } _ \text{fix}) (\text{bar } 1) _ \text{rest}) \&\&) \end{aligned}$$

La siguiente es una regla de multiplicación:

$$\begin{aligned} &(_ ((V +) (N -) (\text{BAR } 0) (\text{VFORM BSE}) (\text{INFL } +) _ \text{rest}) _) \\ \Rightarrow &(_ ((V +) (N P) (\text{BAR } 0) (\text{PN PER1}) (\text{INFL } P) _ \text{rest}) _) \\ &(_ ((V +) (N P) (\text{BAR } 0) (\text{PN PER2}) (\text{INFL } P) _ \text{rest}) _) \\ &(_ ((V +) (N P) (\text{BAR } 0) (\text{PN PLUR}) (\text{INFL } P) _ \text{rest}) _) \end{aligned}$$

La regla indica que (en inglés) una forma verbal, caracterizada por los rasgos (V +) (N -) (BAR 0), básica y que admite flexión, da lugar a tres otras formas que corresponden a la primera y segunda persona del singular y a cualquiera del plural.

Las reglas morfológicas, por su parte, pueden ser de deletreo (spelling), de asignación de valores por defecto o de formación (word grammar).

La siguiente regla, por ejemplo, permite describir que “boxes”, como plural de “box” se forma a partir de esta última forma y del morfema “s” en un proceso que implica la inclusión léxica de la letra “e”.

$$+:e \iff \langle s:s \ c:c \ h:h \rangle \ s:s \ x:x \ z:z \ _s:s$$

20.2.4 Lexicones Frasales

No es corriente el uso de lexicones cuya entrada sean frases, es decir, cadenas de palabras, y no palabras aisladas. La razón de ello es que aunque el empleo de tal tipo de lexicones resolvería alguno de los problemas planteados por algunas lexías o construcciones léxicas que implican a más de una palabra ortográfica, dejaría otros sin resolver, por ejemplo, la inclusión dentro de la lexía de palabras ajenas a ella o la existencia de flexión interna en la misma. Por otra parte, el empleo de cadenas de palabras puede complicar la forma de representación y acceso al lexicon.

Ello no obstante podemos encontrar algunos ejemplos de uso de lexicones frasales. Carbonell-Hayes constituyen un ejemplo notable.

20.2.5 La adquisición del Conocimiento Léxico

Uno de los puntos clave del tratamiento léxico es el problema de su adquisición. Se han seguido básicamente dos enfoques para abordar este problema: la adquisición manual y la adquisición (semi) automática.

La adquisición manual se basa en la existencia de entornos interactivos de ayuda al lexicógrafo que permiten a éste incorporar y estructurar la información léxica. Se trata del sistema más utilizado ya que posibilita, en forma relativamente sencilla, crear lexicones de tamaño medio/bajo, suficientes para muchas aplicaciones.

Un ejemplo notable de este tipo de entornos de adquisición lo proporciona S. Nirenburg. Nirenburg parte del lema “*World first, words later*” para construir primero una organización ontológica de los conceptos a representar (en forma de frames), con sus propiedades y relaciones mutuas, para, más adelante incorporar las palabras en que estos conceptos se realizan léxicamente. El sistema, en cuanto el volumen aumenta, se vuelve bastante complejo y difícil de manejar.

Un enfoque similar, aunque a una escala bastante mayor es el usado por EDR para la construcción de lexicones del japonés y el inglés (incluyendo la traducción entre lenguas). El volumen previsto por este sistema es de unas 50.000 entradas léxicas por lengua.

También podemos citar aquí, aunque el sistema tiene más que ver con la información conceptual que con la léxica, a CYC, de D.Lenat. Se trata de un proyecto, de construcción de una Base de Conocimiento con un contenido equivalente al de una enciclopedia de uso general.

La alternativa es la utilización de recursos léxicos ya existentes (diccionarios para uso humano, corpus textuales, *etc*) para extraer de ellos de forma automática la información léxica que se precisa.

Especial interés han tenido en los últimos años las experiencias de extracción de información léxica a partir de diccionarios de uso humano en soporte magnético (MRD, Machine Readable Dictionaries). A partir de los trabajos iniciales de Amsler, se han desarrollado varios proyectos (Chodorov, Calzolari, Briscoe, Boguraev, Byrd, *etc*).

Uno de los proyectos más ambiciosos es Acquilex. Este proyecto condujo, en una primera etapa a la construcción de Bases de Datos Léxicas (BDL) multilingües a partir de diccionarios del inglés, italiano, holandés y español. La BDL contenía, estructurada, la información presente en los distintos diccionarios. En una segunda etapa se desarrollaron métodos de extracción de la información semántica contenida en la BDL (especialmente a partir de las definiciones de las diferentes entradas). Ello condujo a la construcción de una Base de Conocimientos Léxica Multilingüe (BCL).

20.3 Implementaciones de los Diccionarios

Un problema importante, en relación con el tratamiento léxico, es el de la implementación de los diccionarios o lexicones. En cuanto el volumen se hace importante las exigencias de eficiencia en el acceso obligan a una implementación cuidadosa.

Es importante precisar las características funcionales y operativas en cada caso: volumen, acceso sólo para consultas o también para actualización, diccionario de palabras o de lexemas (o incluso de frases), acceso incremental o global, tipo de información asociada, mecanismos de expansión presentes (herencia, reglas, morfología, *etc*).

En cuanto a organizaciones en memoria, se han utilizado listas (para lexicones de juguete) de varios tipos, árboles, tablas, árboles de búsqueda, *etc* ... Especial interés tienen la utilización de **TRIES**, árboles binarios con búsqueda digital o **BTREES**.

En cuanto a implementaciones en disco, se han utilizado toda la gama de ficheros indexados o Bases de datos.

Un punto importante se refiere a la implementación del rango, es decir, de las propiedades o de la información correspondientes a cada entrada.

Suelen utilizarse notaciones externas, de tipo simbólico, accesibles al usuario, y representaciones internas, más compactas. Se han utilizado listas, árboles de propiedades y representaciones compactas del tipo de los mapas de bits.

Un interés especial tienen los problemas de implementación de las estructuras de rasgos. La

utilización de FSs en formalismos de unificación hace que, a menudo, las FS se implementen en forma de términos de PROLOG (se suele hablar de compilación de FSs a términos).

La presencia de estructuras reentrantes hace que surjan problemas tanto de representación como de unificación de las FSs.

20.4 Morfología

Se ha prestado poca importancia a la Morfología en el TLN debido, básicamente, a la poca complejidad de los problemas morfológicos en la lengua inglesa. Otras lenguas, sin embargo, con gran capacidad aglutinante, flexiva o derivativa, presentan graves problemas en cuanto a la identificación de las formas léxicas y por ello es importante el empleo de Analizadores Morfológicos.

El tratamiento morfológico puede afectar a la adquisición del Conocimiento Léxico, al almacenamiento del mismo, o a ambos. El tratamiento morfológico se basa en los mecanismos de formación de las palabras. En cierto modo un analizador morfológico trata de realizar el proceso inverso al que condujo a la formación de la palabra ortográfica que se examina.

Las palabras pueden formarse por concatenación o composición de formas más simples (“*racacielos*”) o por adjunción. La adjunción supone que a una raíz se le adjunten uno o varios afijos. Los afijos, de acuerdo con el punto de adjunción, pueden ser prefijos, infijos o sufijos. Estos últimos pueden ser flexivos o derivativos. Así por ejemplo, “*adormecedoras*” presenta la adjunción del prefijo “a”, de la raíz “dorm”, del sufijo derivativo “ecedor” y de los sufijos flexivos “a” y “s”.

No todos estos fenómenos tienen la misma importancia en la formación de las palabras en las distintas lenguas y, por lo tanto no todos se tratan en un analizador morfológico concreto. En castellano, por ejemplo, es bastante complicada la casuística de la flexión verbal y la de los derivados (nominalizaciones verbales, por ejemplo), mientras que no tiene demasiada importancia la composición.

El punto básico en un analizador morfológico es la obtención de la descomposición (pattern) de la palabra en una cadena de morfemas. A menudo se trata, además, de obtener el lexema asociado a la forma léxica para, a través de él, acceder a la información semántica. Existen dos tipos básicos de analizadores morfológico, *los analizadores de dos niveles* y *los analizadores de un nivel*.

Los **analizadores de dos niveles** (Ritchie, Koskeniemi) funcionan como transductores (normalmente de estados finitos) de forma que existe un nivel de entrada (superficial) y otro de salida (léxico). La entrada correspondería a la palabra que se analiza y la salida al lexema correspondiente. Las reglas morfológicas establecen las condiciones sobre las letras que se van consumiendo y las que se van produciendo.

Los analizadores de un solo nivel trabajan solamente a nivel superficial. Las reglas, en este caso, establecen condiciones válidas de concatenación entre los diferentes morfemas. Existen abundantes ejemplos que utilizan técnicas muy variadas. MARS, por ejemplo, utiliza una red de transición (TN) para definir las transiciones posibles. SIPA utiliza autómatas de estados finitos con condiciones ligadas a las transiciones entre estados. Ben Hamadou, por su parte, utiliza matrices para definir las combinaciones válidas de afijos.

Un sistema interesante es SEGWORD. En este sistema, también de un solo nivel, se permite no sólo la concatenación de afijos sino también la eliminación de afijos de la forma léxica. El resultado es que el sistema no consta de raíz y afijos concatenados a ella sino de forma base y de afijos concatenados o eliminados de la misma. El interés reside en que entonces es posible sustituir el posible diccionario de raíces por un diccionario de formas base (o formas canónicas) que puede ser un diccionario de uso humano.

21.1 Introducción

El análisis sintáctico constituye el núcleo de cualquier sistema de TLN. El objetivo del Analizador Sintáctico es doble:

1. Determinar si una frase es correcta (es gramatical)
2. Proporcionar una estructura de la frase que refleje sus relaciones sintácticas y que pueda ser usada como base de los tratamientos posteriores.

En general, lo que pretende el análisis sintáctico es determinar si una frase pertenece o no al lenguaje que se trata de analizar. Si los elementos de la frase son palabras, podemos decir que una frase está constituida por una cadena de palabras, es decir, $w \in V^*$, donde V nota al vocabulario terminal de la gramática o lo que es lo mismo, al conjunto de palabras válidas. Por supuesto, no toda concatenación de elementos de V se puede considerar como una frase correcta (es decir, perteneciente al lenguaje). En general tendremos que el lenguaje es un subconjunto estricto de las cadenas posibles, $L \subset V^*$.

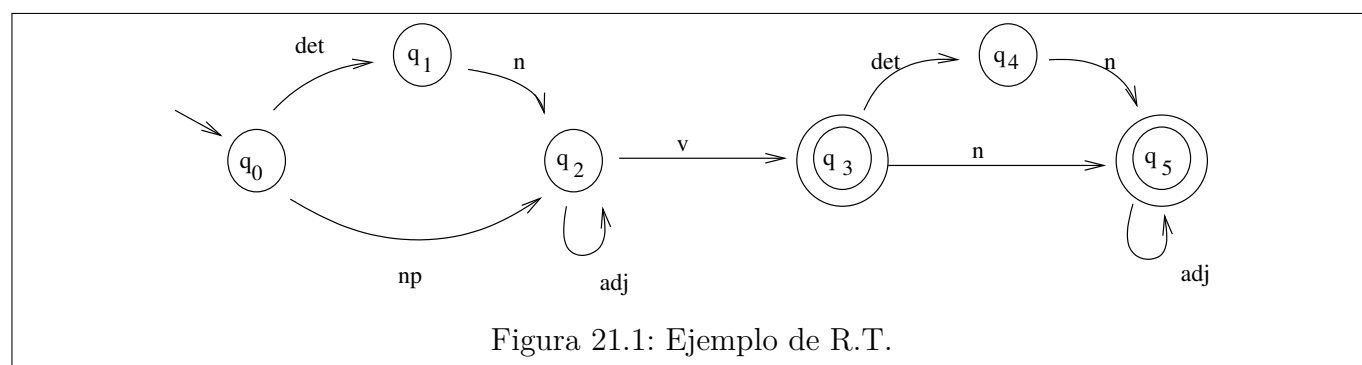
La manera de definir el lenguaje L puede ser a través de una lista de frases correctas, a través de procedimientos interpretativos, o constructivos, o a través de una gramática.

21.2 Las redes de transición

Los primeros procedimientos que se emplearon para realizar el análisis (o mejor dicho, el reconocimiento) sintáctico fueron las **Redes de Transición** RT (o Autómatas de Estados Finitos).

Consideremos la red de la figura 21.1. La red consta de una serie de nodos y una serie de arcos. El nodo origen (algún formalismo admite más de uno) aparece señalado con una flecha mientras que los nodos finales aparecen señalados con un doble círculo. Los nodos representan estados y los arcos transiciones entre estados. Los arcos están etiquetados con nombres de categorías (elementos del vocabulario terminal de la gramática).

Cada transición es aceptable si la palabra de la cadena de entrada tiene una categoría igual a la etiqueta del arco. El tránsito de un estado a otro hace que paralelamente se consuma una palabra en la cadena a analizar (es frecuente utilizar el símil de una cadena de palabras y una ventana abierta sobre la palabra actual que se desplaza al realizarse la transición). El proceso de reconocimiento comienza situándose la ventana sobre la primera palabra y arrancando en el estado de inicio. El proceso continúa, realizándose transiciones válidas entre estados y desplazándose paralelamente la



ventana sobre la cadena de entrada. Si al consumir completamente la cadena a analizar estamos en un estado final, entonces la frase es correcta. Si no fuera así, o bien si en el curso del recorrido nos encontramos en una situación en que ninguna transición es posible, entonces la frase es incorrecta. Podemos admitir no-determinismo, bien porque exista más de un estado de inicio, bien porque de un estado a otro salgan varios arcos con la misma etiqueta o bien por que alguna de las palabras de la cadena de entrada sea ambigua (tenga más de una categoría). En este caso el algoritmo de análisis debería incluir un sencillo mecanismo de backtracking que gestionara el no-determinismo.

Supongamos que, con el autómata de la figura 21.1, tratamos de analizar la frase “*el gato come pescado*”. Un análisis léxico previo nos ha proporcionado la información siguiente:

```

el: cat: det
gato: cat: n
come: cat: v
pescado: cat: n

```

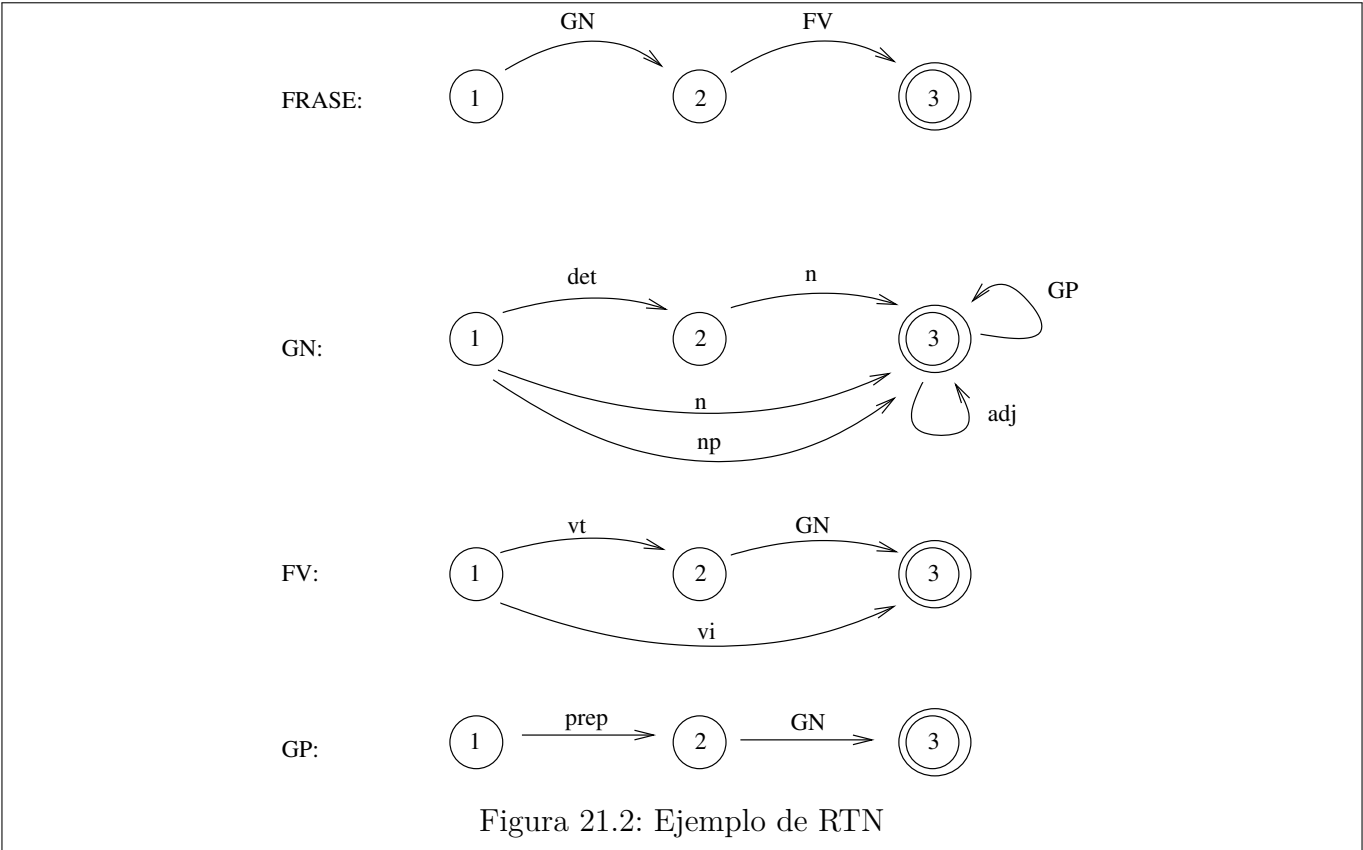
La aplicación del algoritmo nos conduciría desde el estado q_0 (inicio) al q_1 consumiendo “el”, al q_2 consumiendo “gato”, al q_3 consumiendo “come” y al q_4 consumiendo “pescado”. Hemos acabado la frase y estamos en un estado final, por lo tanto la frase es correcta.

Es fácil comprobar que la capacidad expresiva de las RT no iba mucho más allá de lo conseguido mediante analizadores de pattern-matching u otros artilugios.

El siguiente hito lo supusieron las *textbfRedes de Transición Recursivas* (RTN). Una RTN consiste en una colección de RT etiquetadas con un nombre. Los arcos de cualquiera de las RT pueden estar etiquetados con categorías (como en el caso precedente) o con identificadores de las RT. El procedimiento de reconocimiento se complica algo. Cuando la etiqueta del arco es de tipo categoría se procede como hasta ahora; cuando es de tipo RT entonces se continúa el reconocimiento situándose en el estado de inicio de la red referenciada. Cuando se alcanza un estado final, si la red que estamos recorriendo no era la activada inicialmente lo que se hace es continuar el recorrido en el estado destino del arco cuya transición motivó la llamada. En el fondo la transición es una llamada recursiva a una nueva RT (de ahí la denominación) y el llegar a un estado final implica el final de la llamada recursiva y la continuación dentro de la RT llamadora. El mecanismo de backtracking también se complica algo ya que se ha de gestionar una pila de estados de retorno o dejar que la recursividad se ocupe de ello.

La figura 21.2 nos presenta una RTN compuesta de 4 redes: FRASE, GN, FV y GP. La primera analiza frases, la segunda grupos nominales, la tercera frases verbales y la cuarta grupos preposicionales. Fijémonos en la figura 21.3 en las llamadas recursivas cruzadas que se producen a través del ejemplo siguiente: “*el perro de Pepe come pescado*”.

Dado que no hay estado de retorno, que estamos en un estado final y que hemos agotado la cadena de entrada, podemos afirmar que la frase es correcta.



Posicion	Red	Estado	Retorno	arco
el perro de ...	FRASE	1	-	-
el perro de ...	GN	1	FRASE:2	GN
perro de pepe...	GN	2	FRASE:2	det
de pepe come ...	GN	3	FRASE:2	n
de pepe come ...	GP	1	GN:3	GP
pepe come pescado	GP	2	GN:3	prep
pepe come pescado	GN	1	GP:3	GN
come pescado	GN	3	GP:3	np
come pescado	GP	3	GN:3	fin GN
come pescado	GN	3	FRASE:2	fin GP
come pescado	FRASE	2	-	fin GN
come pescado	FV	1	FRASE:3	FV
pescado	FV	2	FRASE:3	vt
pescado	GN	1	FV:3	GN
-	GN	3	FV:3	n
-	FV	3	FRASE:3	fin GN
-	FRASE	3	-	fin FV

Figura 21.3: Ejemplo de análisis mediante una RTN

Las RTN constituyeron una herramienta seria de TLN pero tenían varias limitaciones importantes. La primera era que al hacer depender la posibilidad de una transición exclusivamente de la categoría esperada el sistema era en exceso “local” y se dejaba de utilizar información no categorial. Por otra parte, las RTN eran reconocedoras y no analizadoras del lenguaje.

La siguiente extensión la constituyeron las Redes de Transición Aumentadas (ATN, propuestas por Woods en 1970). Las ATNs son, básicamente, RTNs a las que se añaden operaciones en los arcos. Las operaciones son de dos tipos:

1. Condiciones, que permiten filtrar la transición entre estados.
2. Acciones, que permiten construir estructuras de salida y convertir, de esta forma, el reconocedor en un verdadero analizador. Volveremos más adelante con el tema de las ATNs.

Los formalismos presentados, especialmente las ATNs, han sido y son ampliamente utilizados en el análisis sintáctico pero todos ellos presentan graves dificultades de expresividad notacional. Incluso incorporando mecanismos más legibles para expresar las características de las redes: formas de definir las transiciones válidas, formas de nombrar estados y arcos, *etc* e incluso facilitando medios interactivos, por ejemplo, editores sintácticos y/o gráficos, difícilmente podemos considerar cómoda la forma de definir las condiciones de pertenencia al Lenguaje o las estructuras de salida.

Por ello, la mayoría de los sistemas actuales de análisis sintáctico se apoyan en dos componentes diferentes y diferenciados: la Gramática y el Analizador, facilitando al lingüista la tarea de escribir la gramática como componente separada. De ello nos ocuparemos en la sección siguiente.

21.3 Los formalismos sintácticos: Las Gramáticas

El concepto de Gramática, o más precisamente el de Gramática de Estructura Sintagmática, es bien conocido. Una gramática G es una tupla de 4 elementos:

$$G = \langle N, T, P, S \rangle$$

donde

- N es el vocabulario No Terminal, es decir, el conjunto de elementos no terminales de la Gramática.
- T es el vocabulario Terminal, es decir, el conjunto de elementos terminales de la Gramática.
La intersección de T y N ha de ser nula.
La reunión de T y N es lo que llamamos Vocabulario (V) de la Gramática.
- S , perteneciente a N es el axioma o símbolo de inicio.
- P es el conjunto de reglas de producción de la gramática.

Según las características de P , Chomsky propuso una clasificación ya clásica de las gramáticas en 4 tipos:

tipo 0: En las cuales los elementos de P son reglas de reescritura del tipo

$$u \rightarrow w$$

donde u, w pertenecen a V^* .

tipo 1 (gramáticas sensitivas): en las cuales se establece la limitación de ser la longitud de la cadena de u menor o igual que la de w .

tipo 2 (gramáticas de contexto libre, GCL): que establecen la limitación adicional de permitir sólo reglas del tipo

$$A \rightarrow w$$

donde A es un no terminal y w , como en el caso anterior pertenece a V^* (de hecho la aceptación o no de la cadena nula a la derecha de las producciones, es decir, la aceptación de elementos opcionales, es un punto importante que conviene precisar en los diferentes formalismos).

tipo 3 (gramáticas regulares, GR): son aquellas que admiten únicamente reglas del tipo:

$$A \rightarrow a$$

$$A \rightarrow aB$$

donde “ A ” y “ B ” son elementos de N y “ a ” lo es de T .

Se define el concepto de *derivación directa* $\xRightarrow{G}$ como la operación de reescritura.

$uxw \xRightarrow{G} uyw$ (que podemos leer como “ uxw ” deriva directamente a “ uyw ”) si la cadena “ uxw ” se reescribe en forma “ uyw ” a través de la aplicación de la producción $x \rightarrow y$, que debe pertenecer a P .

Podemos definir la relación $\xRightarrow{*}_G$ como la clausura transitiva de la relación $\xRightarrow{G}$. Llamaremos *derivación* a esta relación.

Diremos que una cadena de elementos de T (es decir, una frase), w , pertenece al lenguaje generado por la gramática G ($w \in L(G)$) si es posible encontrar una derivación de w a partir de S , es decir si $S \xRightarrow{*}_G w$.

Se puede demostrar que las RTNs son equivalentes a las GCL, es decir que a cada RTN le podemos asociar una gramáticas de contexto libre y viceversa, de forma que generen –reconozcan– el mismo lenguaje.

Es preciso establecer un compromiso entre las capacidades expresivas de cada tipo de gramática, las necesidades de expresión necesarias para el tratamiento del lenguaje natural y la existencia de mecanismos computacionales que hagan tratable el formalismo. En general se considera que el lenguaje natural es demasiado complejo para ser expresado mediante gramáticas de contexto libre. Sin embargo, dado que éstas son aquellas que permiten un tratamiento más eficiente (dejamos de lado las GR cuya cobertura es claramente insuficiente) la aproximación más extendida en el TLN, a nivel sintáctico, es la de proporcionar una gramática de contexto libre nuclear y tratar los fenómenos sensitivos en forma de añadidos locales, filtros, *etc* (a menudo implementados *ad-hoc*).

21.4 Gramáticas de Estructura de Frase

Hemos dicho anteriormente que el modelo más extendido en el análisis sintáctico para el TLN consiste en un núcleo de gramáticas de contexto libre y un conjunto de restricciones que incluyen de alguna forma la sensibilidad. Precisemos un poco esta idea.

Supongamos que deseamos construir una gramática que cubra el mismo lenguaje que la RTN de la figura 21.2 (ya hemos dicho que las gramáticas de contexto libre y las RTNs son equivalentes). Una posibilidad es la siguiente:

(G1) $G1 = \langle N1, T1, P1, FRASE \rangle$

donde

$N1 = FRASE, GN, FV, RGN, GP$

$T1 = \text{det}, n, np, \text{adj}, vi, vt, \text{prep}$

$P1 = \{$
 1 $FRASE \rightarrow GN FV.$
 2 $GN \rightarrow \text{det } n RGN.$
 3 $GN \rightarrow n RGN.$
 4 $GN \rightarrow np RGN.$
 5 $RGN \rightarrow \epsilon.$
 6 $RGN \rightarrow GP RGN.$
 7 $RGN \rightarrow \text{adj } RGN.$
 8 $FV \rightarrow vi.$
 9 $FV \rightarrow vt GN.$
 10 $GP \rightarrow \text{prep } GN. \}$

La gramática G1 es obviamente una gramáticas de contexto libre que incluye categorías opcionales (RGN). Es fácil ver que la cobertura es la misma que la de la RTN de la figura 21.2.

La frase “*el gato come pescado*” será, pues, correctamente analizada (es fácil comprobarlo). Ahora bien, ¿qué ocurre si tratamos de analizar “*la gato come pescado*” o “*las gatas come pescado*” o “*el gato comen pescado*”? Todas estas frases, obviamente incorrectas, serán reconocidas como pertenecientes a $L(G1)$.

¿Cabe una solución a este problema dentro del formalismo de las gramáticas de contexto libre? Por supuesto podríamos multiplicar los elementos de V incorporando a las categorías la información de género, número, persona, etc ... (nmassing, nmaspl , etc ... para indicar, respectivamente, un nombre, masculino, singular o un nombre masculino plural). Deberíamos también aumentar el número de reglas de P ($GN \rightarrow \text{detmassing } n\text{massing}$) pero esta multiplicación se haría totalmente insoportable en cuanto el número de fenómenos a tratar, concordancia, dependencias a larga distancia, movimiento de constituyentes, conjunción, etc y la información requerida para el tratamiento se incorporarán a los elementos de V para producir nuevas particiones.

La solución que antes sugeríamos es la del enriquecimiento de las gramáticas con componentes (a menudo de tipo procedural) que traten estos problemas. Así, la segunda producción de G1 podría sustituirse por la siguiente:

$GN \rightarrow \text{det } n RGN \{ \text{concordancia_1} \}$

donde concordancia_1 sería algún filtro capaz de verificar la concordancia.

Formalismos que siguen esta filosofía son DIAGRAM y LSP. Veamos un ejemplo de este último.

*BNF

<SENTENCE>	% %=	<ENUNCIACION> ``.
<ENUNCIACION>	% %=	<SUJETO><VERBO><OD>.
<SUJETO>	% %=	<LNR> / <*NULL>.
<LNR>	% %=	<LN><*N><RN> / <*NULL>.
<LN>	% %=	<*ART> / <*NULL>.
<RN>	% %=	<*NULL>.
<VERBO>	% %=	<*TV>.
<OD>	% %=	<LNR>.

*RESTR

```

WCONC1  =  IN LNR %
           BOTH $SING AND $PLUR.
$SING    =  IF CORE OF LNR HAS ATTRIBUTE SG
           THEN CORE OF LN DOES NOT HAVE ATTRIBUTE PL.
$PLUR    =  IF CORE OF LNR HAS ATTRIBUTE PL
           THEN CORE OF LN DOES NOT HAVE ATTRIBUTE SG.

```

En este ejemplo simplificado vemos que la gramática incluye dos partes, identificadas por las claves *BNF y *RESTR, la primera correspondiente a la gramática de contexto libre y la segunda a las restricciones.

La primera parte admite, en el formalismo de LSP, algunas facilidades expresivas (del estilo del formato BNF). La gramática del ejemplo se autoexplica.

La parte de restricciones está escrita en un lenguaje, el RL (Restriction Language), próximo a Lisp. El RL trabaja sobre los fragmentos de árbol de análisis que se están construyendo. Las primitivas del lenguaje permiten hacer navegaciones sobre el árbol de análisis, consultas sobre la información asociada a los nodos del árbol y anotaciones sobre los mismos. El ejemplo que presentamos establece que a todos los nodos LNR (grupo nominal) se les debe asociar un chequeo de la concordancia en número. Este chequeo se lleva a cabo a través de dos operaciones \$SING y \$PLUR. La primera de ellas exige que, de existir el atributo “SG” en el núcleo de la componente, no debe aparecer el atributo “PL” en el adjunto izquierdo. La otra operación es simétrica.

21.5 Analizadores Básicos

Una vez definido un formalismo gramatical hemos de ocuparnos ahora del Analizador. Lo primero que cabe precisar es lo que esperamos del Analizador. Por supuesto no nos conformaremos con una variable booleana que nos informe sobre la gramaticalidad de la frase. Existen dos resultados básicos a obtener: la estructura sintáctica y la estructura lógica o semántica básica.

La forma habitual de estructura sintáctica es el Árbol de Derivación, o Árbol de Análisis (a menudo decorado con fragmentos de estructuras semánticas). El árbol de derivación nos refleja la estructura sintagmática o de componentes de la oración que analizamos. Una alternativa son las Estructuras Funcionales que permiten expresar relaciones sintácticas no estrictamente ligadas a la estructura de la frase. Los modelos que representan la frase en términos de acciones, estados, situaciones y actantes, como las estructuras de casos, son un ejemplo conocido. Respecto a las estructuras semánticas producidas por el Análisis Sintáctico, las fórmulas lógicas o las redes conceptuales son los ejemplos más corrientes.

El segundo punto de interés es el de la interacción sintaxis/semántica. Ya indicamos anteriormente que existen dos modos básicos de TLN: el secuencial o estratificado y el cooperativo. El nivel de

tratamiento en el que más se aprecian las diferencias entre estos dos modelos es, precisamente, en el sintáctico/semántico.

Se han seguido básicamente las siguientes aproximaciones:

1. Ausencia de Semántica.

Modelos exclusivamente sintácticos que producen un árbol de análisis a partir del cual actúa la aplicación informática. Muchos sistemas de traducción automática siguen este modelo.

2. Ausencia de Sintaxis.

Modelos básicamente lexicalistas que prescinden de la sintaxis obteniendo toda la información necesaria para el tratamiento de las palabras que forman la oración. Modelos aplicados a dominios muy restringidos siguen esta aproximación.

3. Síntesis de las dos aproximaciones anteriores.

La categorización es sintáctico-semántica y el árbol de análisis tiene en sí mismo un etiquetado que incluye la semántica. Las gramáticas semánticas son un ejemplo relevante.

4. Actuación en secuencia.

El análisis sintáctico produce un árbol de análisis y, a partir de él, el Intérprete Semántico (IS) produce la Estructura Semántica. Se trata del modelo más sencillo. Permite una modularización del Conocimiento y de los Tratamientos implicados sin graves complicaciones.

5. Actuación en paralelo.

Modelos cooperativos. Existen formas diversas de realizar la cooperación:

- Uso de interpretaciones semánticas parciales para validar algunas construcciones sintácticas, igualmente parciales. Normalmente estas interpretaciones se incorporan al árbol de análisis (en lo que se suelen llamar decoraciones del árbol) para evitar la redundancia en los cálculos.
- Uso de restricciones selectivas (información léxica de tipo semántico que determinadas componentes sintácticas deben poseer: el verbo “reir”, por ejemplo, exige un sujeto de tipo humano).
- Valencia sintáctica adecuada a determinadas relaciones semánticas.

El tercer punto se refiere a las Fuentes de Conocimiento implicadas en el análisis. La primera, por supuesto, es la Gramática. No es la única, sin embargo. El Lexicón juega, asimismo, un papel predominante no sólo como fuente de categorización sintáctica sino como fuente de asignación de las diferentes propiedades que pueden ser utilizadas por la “extensión” de la gramática de contexto libre.

Son igualmente frecuentes sistemas que incorporan diferentes tipos de reglas de formación al margen de las expresadas en la Gramática: Sistemas basados en Principios, Transformaciones, Metarreglas, *etc*

21.5.1 La técnica del Análisis Sintáctico

La técnica básica del Análisis Sintáctico es la habitual de cualquier sistema de reescritura. Se parte de un objetivo (por ejemplo, FRASE en la gramática G1) y de unos hechos (la frase a analizar, por ejemplo, “*el gato come pescado*”). El objetivo del analizador es lograr una derivación que conduzca del objetivo a los hechos. Varios puntos conviene mencionar en este proceso:

- Estrategia del Análisis

Las estrategias básicas son la Descendente (dirigida por objetivos, *top-down*) y la Ascendente (dirigida por hechos, *bottom-up*). Ambas tienen ventajas e inconvenientes y la elección de una u otra dependerá de las características de la Gramática, del grado de ambigüedad léxica y de la técnica de análisis a emplear. Existen métodos que combinan ambas estrategias, estática o dinámicamente. En ellos la dificultad estriba en dotar al mecanismo de control de suficiente conocimiento para que la mejora de las prestaciones compense el overhead.

- Dirección del Análisis

La mayoría de los analizadores operan de izquierda a derecha. Existen, sin embargo, excepciones. La más conocida es la de los analizadores activados por *islas* en los cuales la localización de determinadas palabras (no ambiguas, muy denotativas, *etc*) activa el proceso ascendente en forma de capas concéntricas alrededor de dichas islas. Otro ejemplo notable de bidireccionalidad lo constituyen los analizadores regidos por el núcleo (*Head Driven*) en los cuales la activación se lleva a cabo, para cada regla, a partir del núcleo (*head*) o componente principal del sintagma correspondiente (el verbo para la frase, el nombre para el grupo nominal, *etc*).

- Orden de aplicación de las reglas.

El azar o el orden de escritura de las reglas de la gramática suele ser el criterio que se sigue al seleccionar las reglas a aplicar en cada caso. Existen, sin embargo, sistemas más sofisticados en los que las reglas se ponderan o en los que se incluyen formas de selección inteligentes.

- La ambigüedad

Dos tipos de ambigüedad tienen importancia durante el análisis sintáctico, la ambigüedad léxica, que hace que una palabra pueda poseer más de una categoría sintáctica, y la ambigüedad sintáctica que hace que puedan producirse más de un árbol de análisis correcto para alguna de las componentes (incluyendo, por supuesto, el propio axioma).

Lo único que debemos señalar es que cualquier sistema de AS debe poseer mecanismos de gestión de estos tipos de ambigüedad. Existen sistemas que asignan probabilidades a las categorizaciones y a las reglas de forma que los árboles de análisis obtenidos pueden dotarse de alguna medida de probabilidad relativa y ordenarse de acuerdo ella. En la mayoría de los casos la selección del árbol adecuado se deja a procesos posteriores.

- No determinismo

El analizador sintáctico presenta, como hemos visto, varias fuentes de indeterminismo. En cualquier caso, e incluso para analizadores calificados de deterministas, es necesario gestionar un mayor o menor grado de indeterminismo. Las herramientas que se emplean para ello son el backtracking y el (pseudo) paralelismo.

Veamos, para acabar, como procedería un analizador sintáctico para analizar la frase "*el gato come pescado*" usando la gramática G1, con una estrategia descendente, unidireccional de izquierda a derecha y aplicando las reglas en el orden en el que están escritas:

Objetivos	Hechos	Regla aplicada
1 FRASE	el gato come pescado	1
2 GN FV	el gato come pescado	2
3 det n RGN FV	el gato come pescado	det = el
4 n RGN FV	gato come pescado	n = gato
5 RGN FV	come pescado	5
6 FV	come pescado	8
7 vi	come pescado	fallo (vuelta a 6)
8 vt GN	come pescado	vt = come
9 GN	pescado	2
10 det n RGN	pescado	fallo (vuelta a 9)
11 n RGN	pescado	n = pescado
12 RGN	-	5
13 -	-	éxito

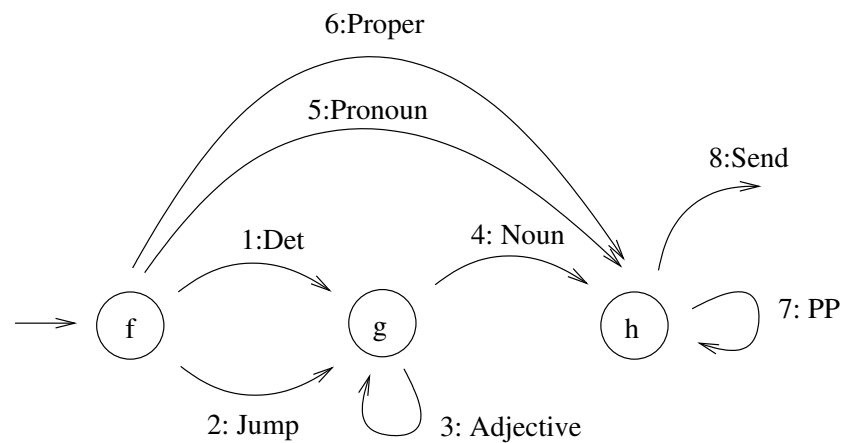
El enfoque alternativo, ascendente, daría lugar a:

Hechos	Regla aplicada
1 el gato come pescado	el = <det>
2 <det> gato come pescado	gato = <n>
3 <det> <n> come pescado	<e>
4 <det> <n> <e> come pescado	5
5 <det> <n> <RGN> come pescado	2
6 <GN> come pescado	come = <vt>
7 <GN> <vt> pescado	pescado = <n>
8 <GN> <vt> <n>	<e>
9 <GN> <vt> <n> <e>	5
10 <GN> <vt> <n> <RGN>	3
11 <GN> <vt> <GN>	9
12 <GN> <FV>	1
13 <FRASE>	éxito

21.5.2 Las ATNs

Como antes señalábamos, las RTNs son equivalentes a las gramáticas de contexto libre. Una de las maneras de analizar una gramática de contexto libre es transformarla en una RTN e implementar esta última. El procedimiento es sencillo pero el método presenta algún problema:

- El mecanismo de análisis de las RTNs es puramente descendente. Intentar implementar estrategias ascendentes o híbridas es antinatural.
- Cualquier transformación de una gramática hace que la estructura resultante, sobre todo si se trata del árbol de análisis, refleje no la estructura de la frase de acuerdo a la gramática original (que es la escrita y prevista por el lingüista) sino de la transformada. Ello puede producir problemas no sólo de depuración de la gramática sino a la hora de describir y llevar a cabo los procesos posteriores al AS. Por supuesto una alternativa es escribir directamente la RTN, con los inconvenientes que en su momento se señalaron.



Feature Dimensions: Number: Singular, Plural: default –empty–

Initializations, Conditions and Actions:

NP-1: f Determiner g

A: Set Number to the Number of *

NP-4: g Noun h

C: Number is empty or Number is the Number of *

A: Set Number to the Number of *

NP-5: f Pronoun h

A: Set Number to the Number of *

NP-6: f Proper h

A: Set Number to the Number of *

Figura 21.4: Ejemplo de ATN

El ejemplo presentado en la figura 21.4 está tomado de Winograd. Se reproduce un fragmento de una gramática de ATNs. Concretamente se trata de una red, llamada NP, que trata de reconocer grupos nominales. Existen 3 nodos, identificados con f , g y h . f es inicial y h es final (de él sale un arco SEND, en la nomenclatura de Winograd). La red es capaz de reconocer pronombres, nombres propios y nombres, opcionalmente precedidos de un determinante. Un número indeterminado de grupos preposicionales se puede postponer al núcleo. Un número indeterminado de adjetivos se podrían anteponer al nombre.

A algunos de los arcos se les asocian Condiciones (notadas con C) y Acciones (notadas con A). Las acciones asocian valores a determinadas variables, llamadas aquí rasgos, mientras que las condiciones chequean el valor de dichos rasgos.

21.5.3 Los Charts

Un problema que presentan las ATNs (y las RTNs y, en general, todos los sistemas que utilizan el backtracking) es el de la redundancia en la ejecución de determinadas operaciones.

En general, el mecanismo del backtracking hace que al ejecutarlo se retorne al estado anterior y, por lo tanto, se pierdan todos los efectos producidos en la opción ahora rechazada. Por supuesto, alguno de estos efectos eran perniciosos (si no, no tendría sentido el volver atrás), pero otros no. Si los procesos realizados incluyen operaciones constructoras: fragmentos de árboles de análisis o de estructuras semánticas, *etc* el problema es aún más grave. La técnica de los Charts intenta solucionar estas insuficiencias.

La idea de los Charts proviene de las llamadas Tablas de Cadenas Bien Formadas (WFST: Well Formed String Tables). Se trata de tablas, construídas dinámicamente, que almacenan los resultados de los elementos no terminales reconocidos durante el proceso de análisis. Estos resultados son accesibles globalmente, no se destruyen durante el proceso de backtracking y pueden ser consultados y utilizados por el analizador.

Para utilizar una WFST debemos modificar el analizador de forma que, antes de abordar el reconocimiento y la construcción de cualquier componente, se compruebe que tal componente no hubiere ya sido construído e incorporado a la tabla. El mecanismo no afecta a la estrategia del análisis.

El inconveniente, como es lógico, es que la tabla ocupa espacio, que construirla requiere tiempo y que una parte (potencialmente considerable) del almacenamiento puede corresponder a material inútil. Una limitación adicional de las WFST es que sólo los componentes bien formados (completados) tienen acomodo. No se memorizan ni objetivos ni hipótesis a medio confirmar. Los Charts pretenden solucionar estos problemas.

Un Chart es un grafo dirigido que se construye en forma dinámica e incremental a medida que se realiza el análisis.

Los nodos del grafo corresponden al inicio y fin de la frase y a las separaciones entre palabras. Si la frase a analizar tiene n palabras tendremos $n+1$ nodos.

Los arcos se crean en forma dinámica. Un arco que va desde el nodo i al j (siempre $j \geq i$) se entiende que subsume a todas las palabras comprendidas entre las posiciones i y j . Los arcos pueden ser de dos tipos, activos e inactivos. Los arcos inactivos corresponden a componentes completamente analizadas (las entradas de las WFSTs) mientras que los arcos activos son objetivos o hipótesis, es decir componentes aún no completamente analizadas.

Una manera clásica de notar y etiquetar los arcos es mediante las llamadas *reglas punteadas* (DR, *dotted rules*). Una DR es, simplemente, una de las reglas de P que contiene en algún lugar de su parte derecha un punto que separa la parte ya analizada de la pendiente.

La regla $A \rightarrow B \ C \ D$ podría, en varias etapas de su proceso, dar lugar a las siguientes DRs:

$$\begin{aligned}
 A &\rightarrow . B C D \\
 A &\rightarrow B . C D \\
 A &\rightarrow B C . D \\
 A &\rightarrow B C D .
 \end{aligned}$$

Esta última correspondería a un arco inactivo y las anteriores a arcos activos. La primera es una mera hipótesis sin ninguna confirmación, ni aún parcial.

Notaremos los arcos del Chart de la siguiente manera:

$$\langle i, j, A \rightarrow \alpha.\beta \rangle$$

donde:

- i es el nodo origen
- j es el nodo destino
- A es un no terminal
- α y β son elementos de V^*
- $A \rightarrow \alpha.\beta$ es una DR
- $A \rightarrow \alpha\beta$ pertenece a P

La regla básica que permite combinar dos arcos es que si el Chart contiene dos arcos contiguos, el primero activo y el segundo inactivo, tales que el segundo es una continuación válida del primero, se crea un nuevo arco con origen en el origen del primero, con destino en el destino del segundo y con una etiqueta igual a la del primer arco con el punto desplazado a la derecha. Será un arco activo o inactivo dependiendo de si quedan o no elementos a la derecha del punto

$$\begin{aligned}
 &\langle i, j, A \rightarrow \alpha.B\beta \rangle \text{ (activo)} \\
 &\langle j, k, B \rightarrow \gamma. \rangle \text{ (inactivo)}
 \end{aligned}$$

producirían:

$$\langle i, k, A \rightarrow \alpha B.\beta \rangle$$

El Chart debe inicializarse añadiendo los arcos (inactivos) que corresponden a las categorías léxicas (terminales).

Las dos estrategias básicas de análisis se pueden implementar de la siguiente forma:

- Estrategia Ascendente:

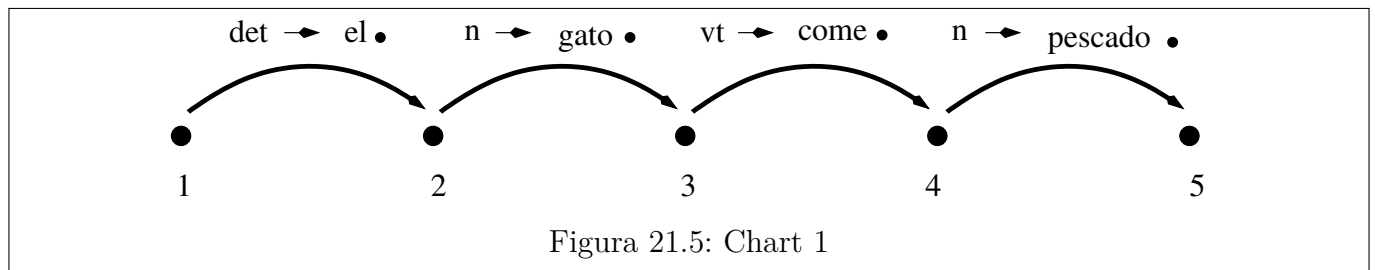
Cada vez que se añade un arco inactivo al Chart, $\langle i, j, A \rightarrow \beta. \rangle$ entonces se debe añadir a su extremo izquierdo un arco activo $\langle i, i, B \rightarrow .A\beta \rangle$ por cada regla $B \rightarrow A\beta$ de P .

- Estrategia Descendente:

Cada vez que se añade un arco activo al Chart, $\langle i, j, A \rightarrow \alpha.B\gamma \rangle$ entonces, para cada regla $B \rightarrow \beta$ de P se deberá añadir un arco activo a su extremo derecho: $\langle j, j, B \rightarrow \beta. \rangle$.

La combinación de la regla básica con la estrategia ascendente o descendente (o cualquier astuta combinación de ambas) nos proporciona el método de análisis. Encontraremos un análisis correcto si en cualquier momento del proceso logramos un arco inactivo que vaya desde el primer nodo al último y esté etiquetado con el axioma de la Gramática.

Consideremos la gramática siguiente (G_2): $G_2 = \langle N_2, T_2, P_2, \text{FRASE} \rangle$



donde

N1 = FRASE,GN,FV

T1 = det,n,vi,vt

P1 = {
 1 FRASE → GN FV.
 2 GN → det n.
 3 GN → n.
 4 FV → vi.
 5 FV → vt GN.
 }

Si deseamos analizar la frase “*el gato come pescado*” deberíamos en primer lugar inicializar el Chart produciendo (ver la figura 21.5):

1 <1, 2, det → el.>
 2 <2, 3, n → gato.>
 3 <3, 4, vt → come.>
 4 <4, 5, n → pescado.>

Veamos el funcionamiento del analizador con una estrategia ascendente: Al añadir el arco inactivo 1 se crearía un nuevo arco, correspondiente a la regla 2 de G2:

5 <1, 1, GN → .det n>

lo mismo ocurriría con los arcos 2, 3 y 4 dando lugar a la creación de los siguientes arcos:

6 <2, 2, GN → .n>
 7 <3, 3, FV → .vt GN>
 8 <4, 4, GN → .n>

La aplicación de la regla básica entre los arcos 5 y 1 daría lugar a:

9 <1, 2, GN → det . n>

Entre 9 y 2:

10 <1, 3, GN → det n .>

De forma similar, los arcos 6 y 2, 7 y 3, 8 y 4 darían, respectivamente lugar a la creación de:

11 <2, 3, GN → n .>
 12 <3, 4, FV → vt . GN>
 13 <4, 5, GN → n .>

Los arcos 10,11 y 13 son inactivos y dan, por lo tanto, lugar a la aplicación de la regla del Análisis Ascendente.

14 <1, 1, FRASE → .GN FV>
 15 <2, 2, FRASE → .GN FV>
 16 <4, 4, FRASE → .GN FV>

Por otra parte, 12 y 13 pueden componerse para producir

17 <3, 5, FV $\rightarrow$ vt GN .>

Igualmente, 14 y 10, 15 y 11, 16 y 13 se pueden componer para dar lugar a

18 <1, 3, FRASE $\rightarrow$ GN . FV>

19 <2, 3, FRASE $\rightarrow$ GN . FV>

20 <4, 5, FRASE $\rightarrow$ GN . FV>

Finalmente 18 y 17 se compondrían para producir

21 <1, 5, FRASE $\rightarrow$ GN FV .>

que es precisamente la condición de gramaticalidad de la frase.

Veamos ahora, en forma gráfica el análisis siguiendo la estrategia descendente (simplificamos para facilitar la lectura de las figuras).

Es preciso iniciar el análisis con un arco activo que corresponde al objetivo de obtener una FRASE. Ello desencadena la regla descendente con el objetivo de un GN y luego de un <n> y un <det>, tal como presenta la figura 21.6.

Sólo el tercero de estos arcos puede ser extendido, primero incorporando <det> y luego <n> hasta completarse. En este punto también el primer arco podrá extenderse un paso para llegar a la figura 21.7.

La creación del arco

<1, 3, FRASE $\rightarrow$ GN . FV>

activa, de nuevo la regla de análisis descendente, como presentamos en la figura 21.8.

La reiteración de estos procesos conduce, finalmente, a la figura 21.9 en la que la aparición de un arco

<1, 5, FRASE $\rightarrow$ GN FV .>

nos indica la gramaticalidad de la frase analizada.

21.6 Los formalismos de Unificación

21.6.1 Introducción

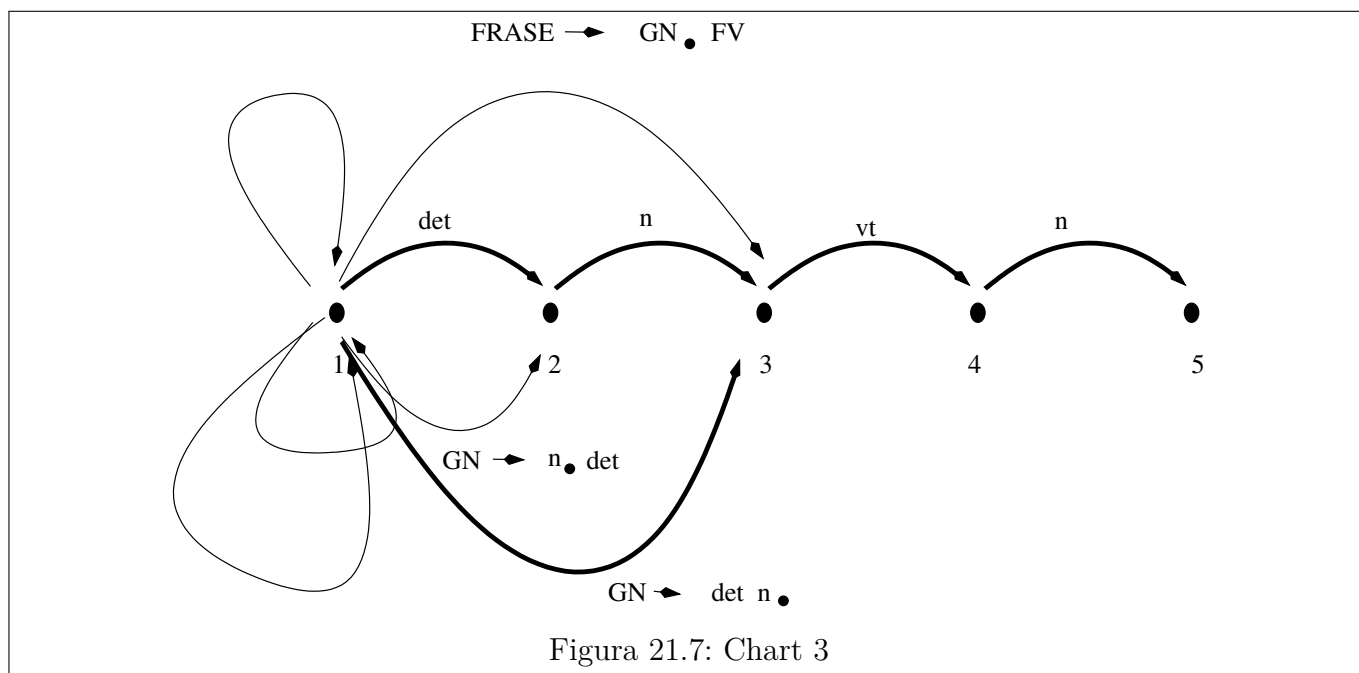
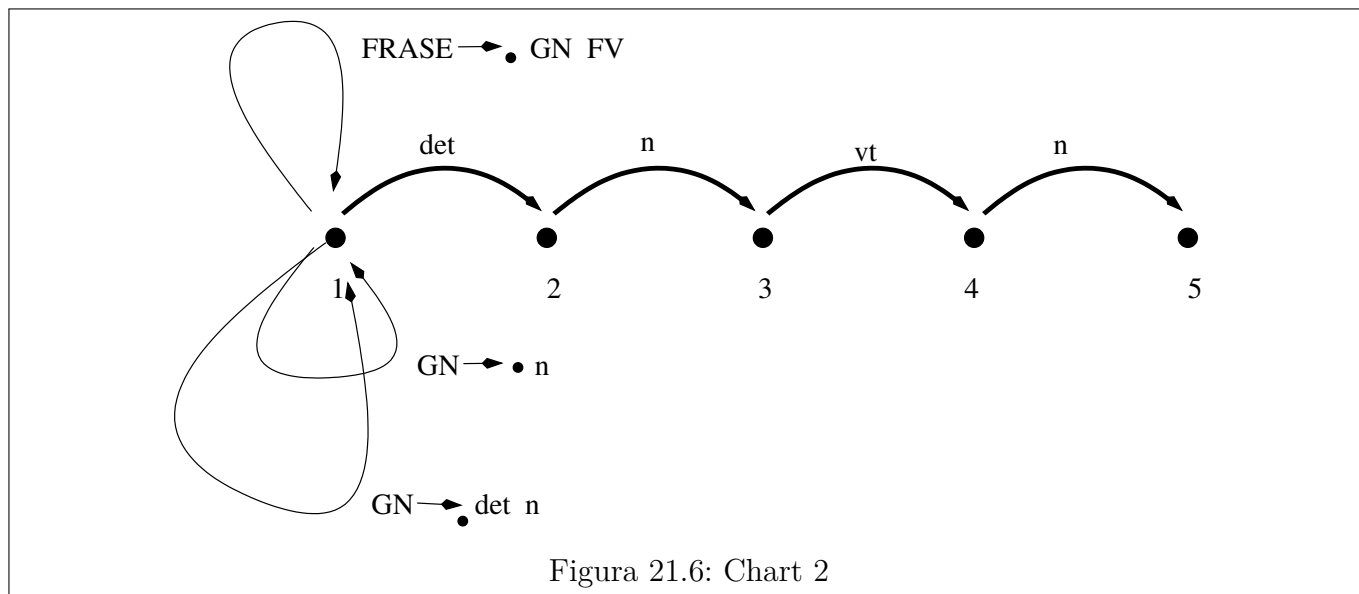
Una especial importancia ha adquirido en los últimos tiempos la utilización de formalismos basados en unificación o, más ampliamente, el empleo de las Gramáticas Lógicas.

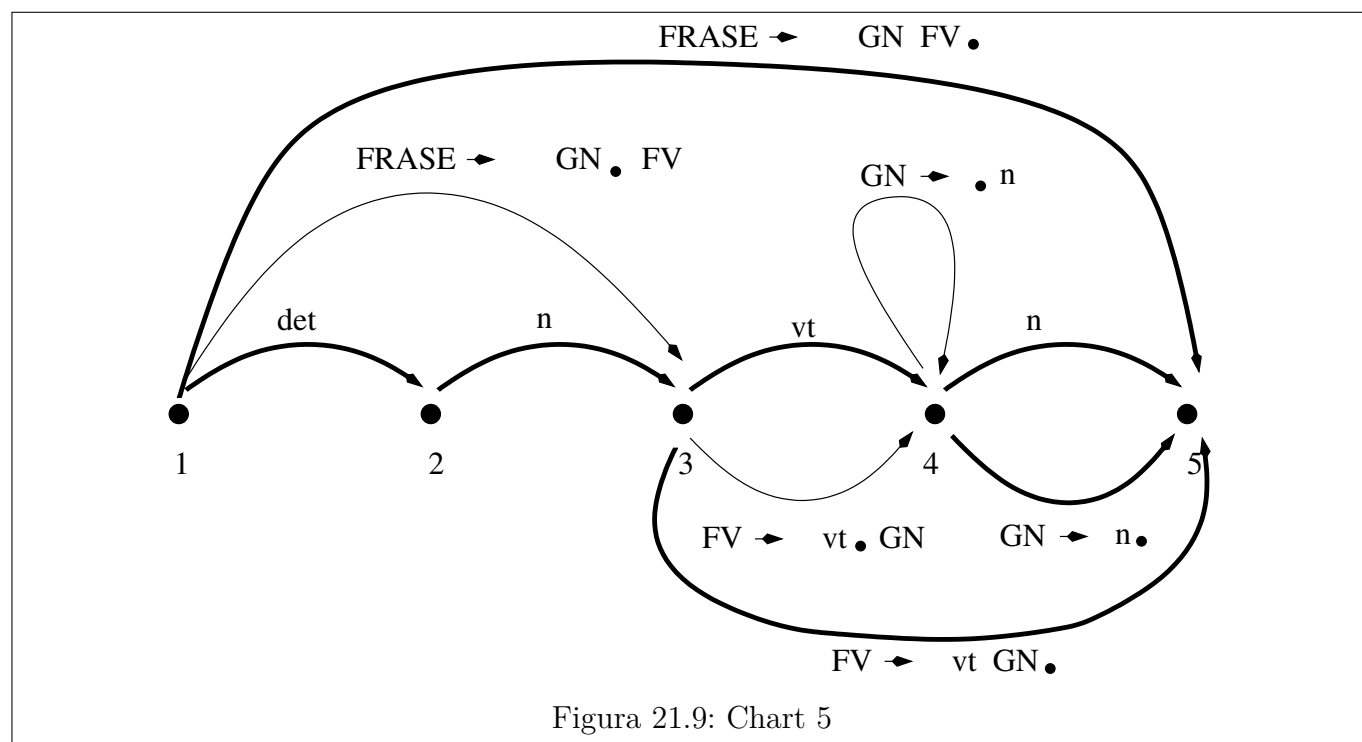
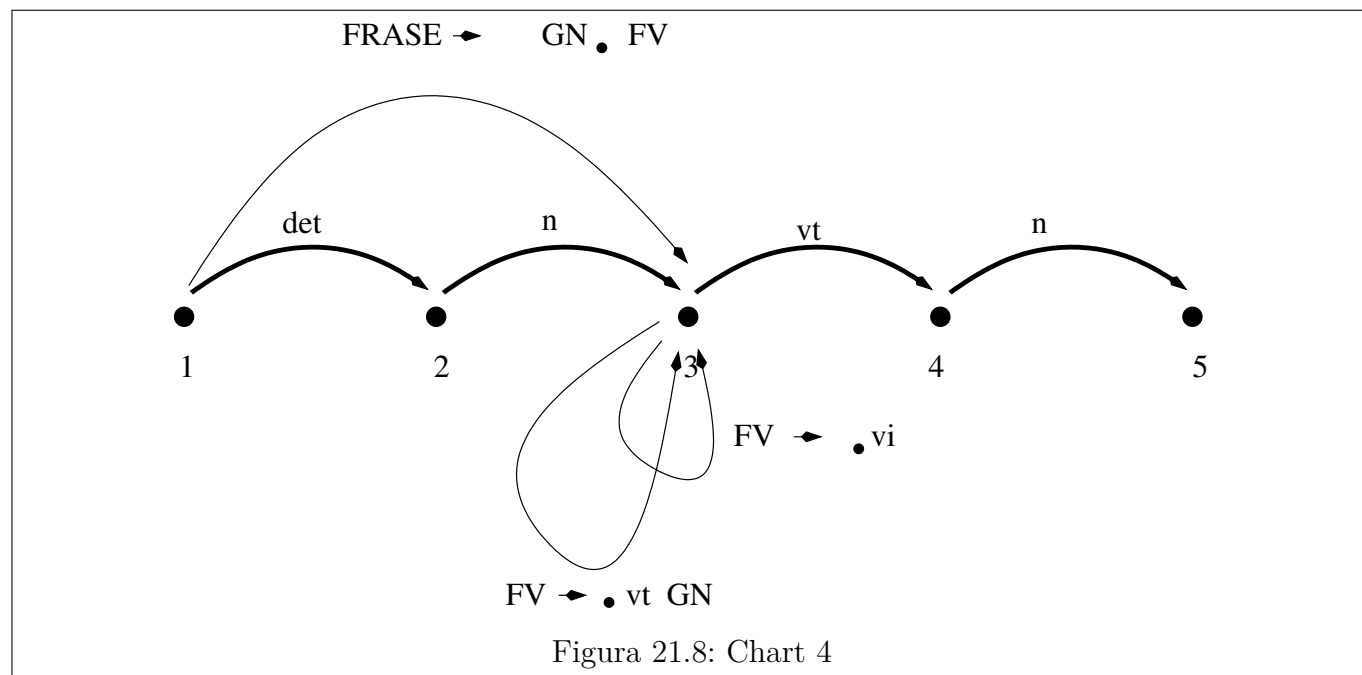
Bajo la etiqueta genérica de *Gramáticas Lógicas* (GL) podemos agrupar a toda una serie de formalismos de descripción lingüística, tanto sintáctica como semántica, que cubren un espectro muy amplio de fenómenos lingüísticos y abordan el TLN desde muy diversas perspectivas.

Existen formalismos que sirven de soporte a teorías lingüísticas muy concretas (como las GPSG, ya mencionadas), otros encuentran su base en aproximaciones lógicas o algebraicas (como las CG) y otros, en fin, son neutrales respecto a la teoría y admiten su utilización para descripciones lingüísticas con base muy diversa (las DCG pertenecerían a este último apartado).

El nivel utilizado para la descripción es asimismo muy variado, existen formalismos en los cuales la descripción se realiza a un nivel de abstracción alto (PATR-II puede ser un ejemplo) mientras que en otros el lenguaje es muy próximo al de implementación, que habitualmente es Prolog. Igualmente son variados los aspectos de implementación que se ocultan al lingüista facilitando su tarea.

En algunos formalismos la descripción lingüística va ligada en forma rígida a una forma o estrategia de funcionamiento (es el caso de las MLG, Gramáticas Lógicas Modulares) mientras que en otros la utilización es bastante más flexible.





Es habitual el uso de las gramáticas lógicas para la tarea de comprensión del lenguaje natural aunque también podemos encontrar ejemplos interesantes de utilización para Generación o Transferencia.

Algunos de los formalismos de las gramáticas lógicas permiten la descripción separada de la información sintáctica y semántica (como las DCTG, Definite Clause Translation Grammars) mientras que en otros el mecanismo de descripción es uniforme.

Si quisiéramos obtener las características básicas de las gramáticas lógicas a partir de la intersección de las características de los diversos sistemas seguramente obtendríamos el conjunto vacío. Existen, sin embargo, dos factores que podemos considerar fundamentales: el uso de la unificación como mecanismo básico (no único) de composición entre constituyentes y el uso de una aproximación de estructuración sintagmática como forma básica de descripción gramatical.

La utilización de la Unificación permite entroncar a las gramáticas lógicas en el paradigma lógico de programación (o de resolución de problemas) mientras la utilización de gramáticas de estructura sintagmática sigue la tradición de descripción sintáctica de la gramática generativa y concretamente la práctica habitual en el TLN de describir el núcleo del lenguaje mediante gramáticas de contexto libre y modificar éstas (extenderlas o restringirlas) para hacer frente a los aspectos sensitivos.

En los últimos tiempos y atendiendo a las tendencias más recientes en descripción sintáctica (especialmente las basadas en la teoría GB) que refuerzan el papel del léxico como fuente de información gramatical reduciendo relativamente el de la gramática y substituyéndolo (parcialmente) por principios generales de buena formación, se han producido esfuerzos destinados a incluir estos elementos en los formalismos lógicos.

21.6.2 Referencia Histórica

El desarrollo histórico de las gramáticas lógicas ha supuesto la incorporación de elementos de índole muy diversa provenientes de diferentes campos: Lingüística, Lógica, Teoría de Lenguajes, Deducción *etc*. Veremos a continuación una breve panorámica de las aportaciones más importantes.

El origen de las gramáticas lógicas suele asociar a los Q-Systems, desarrollados por Colmerauer en 1972 y que dieron lugar algo más tarde a la aparición del lenguaje Prolog. El propio Colmerauer presentó en 1975 las Gramáticas de Metamorfosis (MG, Metamorphosis Grammars) que pueden considerarse como el primer formalismo expresamente creado para la descripción de gramáticas lógicas.

Por esa época el TLN se solía llevar a cabo mediante el uso de las ATNs. Intentos posteriores de remediar los problemas que el uso incontrolado del backtracking, inherente por otra parte a las ATNs, producía dieron lugar al desarrollo de las Tablas de Cadenas Bien Formadas y posteriormente a la técnica de los Charts que ya hemos descrito.

Precisamente las propuestas de Colmerauer y Kowalsky de tratar el análisis de una frase en lenguaje natural como un problema de demostración de un teorema no eran sino implementaciones de los Charts en el nuevo paradigma lógico.

La siguiente aportación, fundamental, fue la de las Gramáticas de Cláusulas Definidas. Las DCG suponían una implementación de las gramáticas lógicas a un nivel de abstracción levemente por encima de Prolog y enteramente incorporadas al mismo. De hecho las DCG se han convertido en un auténtico estándar de expresión de las gramáticas lógicas de forma que buena parte de los formalismos posteriores no son sino refinamientos sintácticos de Gramáticas de Cláusulas Definidas. En paralelo a estos esfuerzos de tipo informático fueron apareciendo las primeras propuestas para dar un soporte lingüístico teórico a las GL. Podemos considerar tres grandes líneas de actuación:

1. La asociada a las Gramáticas Léxico Funcionales (LFG).
2. La asociada a las Gramáticas Funcionales (FG, Functional Grammars, Dik), a las Gramáticas

de Unificación (UG, Unification Grammars, Kay) y, finalmente a las Gramáticas Funcionales de Unificación (FUG, Functional Unification Grammars, Kay).

3. La asociada a las Gramáticas de Estructura de Frase Generalizadas (GPSG) y sus derivadas (HG, Head Grammars, Pollard), (HPSG, Pollard, Sag).

Hemos mencionar también aquí la incorporación de la familia de las Gramáticas Catoriales (GC, Steedman) y posteriormente de las Gramáticas Catoriales de Unificación (UCG, Zeevat).

Desde el punto de vista del formalismo, aparecieron en los años 80 varios enriquecimientos de las gramáticas de cláusulas definidas que aportaban soluciones a problemas puntuales que las gramáticas de cláusulas definidas presentaban. Las Gramáticas de Extraposición (XP, Pereira), las SG ("Slot Grammars", Mc Cord), las GG ("Gap Grammars", Dahl), las DC ("Discontinuous Grammars", Dahl), las SDG ("Static Discontinuous Grammars", Dahl) o el sistema DISLOG de St.Dizier pueden considerarse así. Se realiza un estudio de la gramáticas en la sección 21.3.

Más interés presentan sistemas desarrollados posteriormente que intentan proporcionar medios de separar los diferentes tipos de información a describir, especialmente la información sintáctica y la semántica. Las MSG (Modifier Structure Grammars, Dahl y McCord) y las muy conocidas y utilizadas MLG (Modular Logic Grammars, McCord) son ejemplos significativos.

Mención aparte merecen las DCTG ("Definite Clause Translation Grammars", Abramson) que se basan en las conocidas Gramáticas de Atributos, ampliamente usadas en compilación.

Aportaciones como las de Hirshman o Sedogbo (RG, "Restriction Grammars") que intentan proporcionar medios de describir las restricciones sobre el núcleo gramatical básico en forma procedural, a partir de formalizaciones anteriores, son también reseñables. Las últimas aportaciones de Dahl y Saint Dizier en la línea de incorporar elementos de "Sistemas basados en Principios" son también dignas de mención.

Un último punto a mencionar es el de las Gramáticas de Rasgos. Se trata de formalismos en los cuales la descripción de las restricciones se realiza a través de ecuaciones que ligán los valores de determinados rasgos de los constituyentes. La influencia de las FUG (Kay) o de las LFG (Kaplan, Bresnan) es patente. El ejemplo paradigmático es PATR-II (Shieber), que se ha convertido de hecho en un estándar de este tipo de formalismos. Propuestas posteriores como ALE (Carpenter) son también interesantes.

21.6.3 El análisis gramatical como demostración de un teorema

La expresión de la gramática y el lexicón en forma de cláusulas de Horn permite aplicar la resolución y el razonamiento por refutación como procedimiento de análisis y reducir de esta manera el análisis a la demostración de un teorema.

Consideremos la siguiente gramática:

(G3)

- (1) frase(X,Y) $\leftarrow$ gnom(X,Z),gver(Z,Y)
- (2) gnom(X,Y) $\leftarrow$ art(X,Z),nom(Z,Y)
- (3) gver(X,Y) $\leftarrow$ ver(X,Y)

Consideremos el siguiente lexicón (no establecemos diferencia alguna por el momento entre gramática y lexicón):

- (4) art(X,Y) $\leftarrow$ el(X,Y)
- (5) nom(X,Y) $\leftarrow$ perro(X,Y)
- (6) ver(X,Y) $\leftarrow$ ladra(X,Y)

Tanto gramática como lexicón han sido expresados mediante implicaciones.

Incorporemos, a continuación, la frase a analizar (“*el perro ladra*”):

- (7) $\text{el}(1,2) \leftarrow$
- (8) $\text{perro}(2,3) \leftarrow$
- (9) $\text{ladra}(3,4) \leftarrow$

Y planteemos el teorema a demostrar: que “*el perro ladra*” sea una frase gramatical respecto a la gramática anterior. Estableceremos el teorema como:

$\text{frase}(1,4) \leftarrow$

es decir, “hay una frase entre las posiciones 1 y 4” (fijémonos en la similitud con la técnica de los Charts).

Razonando por refutación deberemos negar el teorema:

$\leftarrow \text{frase}(1,4)$

y vía derivación descendente tratar de demostrar []. Supongamos que los criterios de selección de término a expandir y expansión a realizar son los triviales: recorrido de izquierda a derecha de la negación a convertir y búsqueda secuencial en la gramática.

La única regla aplicable es (1). Se produce para ello la unificación de $(X \leftrightarrow 1, Y \leftrightarrow 4)$ dando lugar a la negación siguiente:

$\leftarrow \text{gnom}(1,Z), \text{gver}(Z,4)$

Se aplican, a continuación las reglas (2) y (4) sobre los términos “ $\text{gnom}(1,Z)$ ” y “ $\text{art}(1,U)$ ” dando lugar a las siguientes negaciones de la secuencia:

- $\leftarrow \text{art}(1,U), \text{nom}(U,Z), \text{gver}(Z,4)$
- $\leftarrow \text{el}(1,U), \text{nom}(U,Z), \text{gver}(Z,4)$

La aplicación de la regla (7) incorpora a la lista de substituciones $(U \leftrightarrow 2)$ y produce la siguiente negación de la secuencia:

$\leftarrow \text{nom}(2,Z), \text{gver}(Z,4)$

La aplicación de las reglas (5) y (8) incorpora a la lista de substituciones $(Z \leftrightarrow 3)$ y produce la siguientes negaciones de la secuencia:

- $\leftarrow \text{perro}(2,Z), \text{gver}(Z,4)$
- $\leftarrow \text{gver}(3,4)$

Se pueden aplicar a continuación las reglas (3) (6) y (9) para producir:

- $\leftarrow \text{ver}(3,4)$
- $\leftarrow \text{ladra}(3,4)$
- $\leftarrow []$

Hemos derivado, pues, la contradicción y por lo tanto la negación de $\text{frase}(1,4)$ no es cierta y por lo tanto es cierta la afirmación $\text{frase}(1,4)$, es decir, “*el perro ladra*” es una frase gramatical.

Resumimos, a continuación todo el proceso deductivo:

- $\leftarrow \text{frase}(1,4)$
- $\leftarrow \text{gnom}(1,Z), \text{gver}(Z,4)$
- $\leftarrow \text{art}(1,U), \text{nom}(U,Z), \text{gver}(Z,4)$
- $\leftarrow \text{el}(1,U), \text{nom}(U,Z), \text{gver}(Z,4)$
- $\leftarrow \text{nom}(2,Z), \text{gver}(Z,4)$
- $\leftarrow \text{perro}(2,Z), \text{gver}(Z,4)$
- $\leftarrow \text{gver}(3,4)$
- $\leftarrow \text{ver}(3,4)$
- $\leftarrow \text{ladra}(3,4)$
- $\leftarrow []$

Una implementación obvia de este formalismo lo proporciona el lenguaje Prolog. El mecanismo deductivo, así como los criterios de selección de términos y reglas, forman parte del propio lenguaje.

21.6.4 Las Gramáticas de Cláusulas Definidas

Ya hemos indicado anteriormente que el desarrollo de los formalismos lógicos ha seguido una triple dirección: la mejora de la capacidad expresiva, la solución de determinados problemas que los anteriores formalismos no permitían y la ocultación de algunos argumentos para permitir trabajar a un nivel de abstracción más adecuado. Las Gramáticas de Cláusulas Definidas constituyen el primer paso importante en esa dirección.

Las Gramáticas de cláusulas definidas incorporan cuatro novedades importantes sobre el formalismo descrito en la sección anterior:

1. Los símbolos gramaticales ya no tienen por qué reducirse a meras etiquetas. Podemos, de acuerdo a los formalismos lógicos, utilizar constantes pero también funtores que admiten una lista de argumentos que a su vez pueden ser constantes u otros funtores.

Así, en la gramática G3, teníamos la categoría “nom” y una cláusula nos permitía decir que “perro” tenía esa categoría. Podemos, en las DCG, enriquecer la categorización y definir que los nombres pueden tener género y número y asociar a “perro” la categoría “nom(gen(masc),num(sing))”. Por supuesto la argumentación no se limita a aspectos sintácticos (categorización, concordancia, rección, *etc*) sino que puede ampliarse a aspectos semánticos: Así en G4 podemos ver como se ha utilizado la etiqueta “adj([arg(con)])” asignada a la palabra “enfadado” para indicar que dicha palabra es un adjetivo, y que rige la preposición “con”.

2. Una segunda característica es la posibilidad de utilizar variables (de hecho el ejemplo anterior contenía las variables “X” y “Y”). Las variables pueden ser instanciadas durante el proceso unificación y proporcionan el medio natural de comunicación entre los componentes.
3. Las Gramáticas de cláusulas definidas ocultan los argumentos ligados a la posición de los componentes en la frase.
4. Las Gramáticas de cláusulas definidas permiten incluir código Prolog en las reglas para efectuar acciones o comprobaciones no ligadas estrictamente al proceso de análisis. Cuestiones como la concordancia, la interpretación semántica o la interacción de la gramática con otras partes de la aplicación informática pueden ser de este modo abordadas de un modo sencillo.

Sintaxis de las gramáticas de cláusulas definidas.

Una gramáticas de cláusulas definidas se compone de una o varias reglas DCG.

Cada regla DCG es de la forma “<izda> → <dcha>.” o “<izda>.” (conviene prestar atención al uso del operador “→” que viene declarado en la mayoría de las versiones de Prolog) dónde <izda> puede ser cualquier átomo, <dcha> puede ser una lista de uno o más elementos separados por comas. Cada uno de los elementos puede ser un átomo, una lista de una o varias constantes o variables encerradas entre corchetes (“[”, “]”) y separadas por comas o una lista de uno o varios predicados Prolog entre llaves (“{”, “}”) y separados por comas.

Las constantes, como es habitual en Prolog, se notan con símbolos que comienzan por minúscula, las variables por mayúscula.

Los átomos pueden ser constantes o funtores. En este último caso se notan con la etiqueta del functor y encerrados entre paréntesis la lista de argumentos que a su vez pueden ser átomos.

Como se ha dicho anteriormente, la comunicación entre los elementos de la parte derecha entre sí y con los de la parte izquierda se realiza, vía unificación, mediante el uso de variables.

Los átomos son los reponsables de la parte sintáctica de la gramática. El código entre llaves implica simplemente una llamada a Prolog. La lista de variables o constantes entre corchetes se unifica con los elementos correspondientes de la lista de entrada (de hecho es lo que implementa la diferencia entre la lista de palabras de entrada y salida).

Veamos un ejemplo: `gver → iver, [Palabra], verbo(Palabra), dver`.

“gver”, “iver” y “dver” son constantes. “[Palabra]” consume una palabra de la cadena de entrada y la unifica con la variable Palabra. “verbo(Palabra)” llama a un predicado Prolog y le pasa la palabra anterior como argumento.

La principal limitación de las gramáticas de cláusulas definidas es la de limitar la longitud de la cadena izquierda a un solo elemento. Ello recuerda el formato de las gramáticas de contexto libre aunque obviamente las DCG no lo son (podemos fácilmente implementar la contextualidad mediante comunicación vía argumentos de los funtores). La siguiente gramática es un ejemplo sencillo de DCG.

(G4)

```

asercion → npr, verb_ser, adj (X), compl(X).
asercion → npr, verb(X), compl(X).
compl([ ]) → [ ].
compl([arg(X)|Y]) → prep(X), npr, compl(Y).
npr → [clara].
npr → [maria].
npr → [juan].
verb([arg(en)]) → [piensa].
verb([arg(en)]) → [esta].
verb_ser([ ]) → [es].
verb_ser([ ]) → [esta].
verb([ ]) → [rie].
verb([arg(con)]) → [habla].
prep(en) → [en].
prep(con) → [con].
prep(de) → [de].
verb([arg(con),arg(de)]) → [habla].
adj([arg (con)]) → [enfadado].
adj([arg (a)]) → [parecido].
prep(a) → [a].

```

22.1 Introducción

La Semántica se refiere al significado de las frases. La Interpretación Semántica (IS) es el proceso de extracción de dicho significado. No existe unanimidad en cuanto a los límites de la IS. En nuestra presentación consideraremos que la IS se refiere a la representación del significado literal de la frase considerada ésta en forma aislada, es decir, sin tener en cuenta el contexto en que ha sido enunciada. El conseguir a partir de aquí la interpretación final de la oración será una cuestión que resolveremos a nivel de la Pragmática.

Ya indicamos en su momento que existen diferentes modelos semánticos, aunque ni de lejos se ha llegado en Semántica al mismo grado de formalización que en Sintaxis. Desde el punto de vista del TLN se suelen exigir una serie de cualidades al modelo semántico utilizado y, por lo tanto, al proceso de la interpretación. Allen, por ejemplo, propone:

- La Semántica ha de ser compositiva. Es decir, la representación semántica de un objeto debe poder realizarse a partir de la de sus componentes.
- Debe utilizarse una teoría, no basarse en una Semántica “ad-hoc”.
- Se deben utilizar objetos semánticos. Se debe definir un Sistema de Representación Semántico.
- Debe existir un mecanismo de interfaz entre Sintaxis y Semántica.
- La IS debe ser capaz de tratar fenómenos complejos como la cuantificación, la predicación en sus diversas formas, modalidades, negación, *etc*
- La IS debe ser robusta frente a las ambigüedades léxica (polisemia) y sintáctica. El sistema de representación ha de ser no ambiguo.
- El sistema de representación debe soportar inferencias (herencia, conocimiento no explícito, *etc*).

En cuanto al mecanismo de la representación semántica, se han utilizado formas variadas. Quizás lo más corriente sea el uso de formas de representación basadas en lógica, especialmente formas diversas del Cálculo de Predicados de primer orden. También las Redes Semánticas han sido ampliamente utilizadas. Representaciones más profundas, como los grafos de dependencia conceptual de Schank (u otros de más alto nivel, como los MOPs o SCRIPTs) tienen también sus partidarios. Los modelos más basados en Semántica (con poca o nula incidencia de la Sintaxis) encuentran especial acomodo en representaciones estructuradas (como los Frames). Por último, los enfoques más lexicalistas y

próximos a las gramáticas de rasgos suelen acomodar el formalismo de las FSs para incorporar a él los rasgos semánticos.

Es habitual que la compositividad de la Semántica se realice a nivel sintáctico. Es decir, que la IS de un objeto sea función de las IS de sus componentes sintácticas. Ésta será la hipótesis que adoptaremos en lo que sigue.

Si hablamos de semántica compositiva hemos de definir el nivel atómico, es decir el que no admite mayor descomposición. En la hipótesis adoptada éste no puede ser sino el nivel léxico (es decir, consideraremos al lexema como unidad de significado). En este sentido cabría asignar a las palabras, a través del Lexicón, estas unidades de significado.

Consideremos, por ejemplo, un sistema de representación basado en lógica. En él parece razonable que los nombres propios se interpreten como las constantes del formalismo (“Pedro” $\rightarrow$ pedro). Los verbos intransitivos, por su parte, se podrían interpretar como predicados unarios (“rie” $\rightarrow$ (λx), reir(x)), *etc*

La composición de significados de estas unidades en la frase “*Pedro ríe*” podría, si la función de composición fuera la aplicación lambda, dar lugar a:

$$(\lambda x, \text{reir}(x)) \text{ pedro} \rightarrow \text{reir}(\text{pedro}).$$

¿Cómo llevar a cabo la IS? Hay que decidir dos factores, la función (o funciones) de composición y el momento de llevar a cabo la interpretación. Respecto a la función de composición, el enfoque más elegante sería el de admitir una sola función de composición de forma que las diferencias de comportamiento quedaran reflejadas en los parámetros de tal función y, en último extremo, en el léxico.

Si siguiéramos este enfoque no tendría demasiada importancia el momento de llevar a cabo la IS. Si lo hacemos con posterioridad al análisis sintáctico resultará que la IS dará lugar a un recorrido en postorden del árbol de análisis (del que ya dispondremos) llamando recursivamente al IS que aplicaría la misma función de interpretación sobre las componentes sintácticas del nodo (es decir, sobre sus hijos en el árbol de análisis).

El enfoque opuesto sería el de la aplicación regla a regla. Cada vez que se aplique una regla con éxito, es decir, cada vez que se construya un nuevo nodo en el árbol de análisis podemos calcular la IS del constituyente.

La ventaja del primer caso es que no se presentan problemas con el backtracking (el árbol de análisis ya ha sido construido) y, por lo tanto, las acciones del IS serán ya definitivas. El inconveniente es que muy probablemente no tengamos un árbol de análisis sino varios y que alguno de ellos pudiera haber sido eliminado por consideraciones de tipo semántico.

El inconveniente de la aplicación regla a regla es que se puede llevar a cabo procesamiento semántico inútil (que el backtracking deshará). La ventaja es que la IS constituye una forma más de restricción de la regla (además de las posibles restricciones de tipo sintáctico).

Elegir una u otra aproximación se puede basar en consideraciones sobre el coste relativo de los procesos sintáctico y semántico.

Una forma extendida de implementar la función de composición es limitarla a una lambda evaluación. En este caso, la única información semántica que se debería asociar a las reglas de la gramática sería el orden de aplicación de los componentes. Consideremos la siguiente gramática:

(G5)

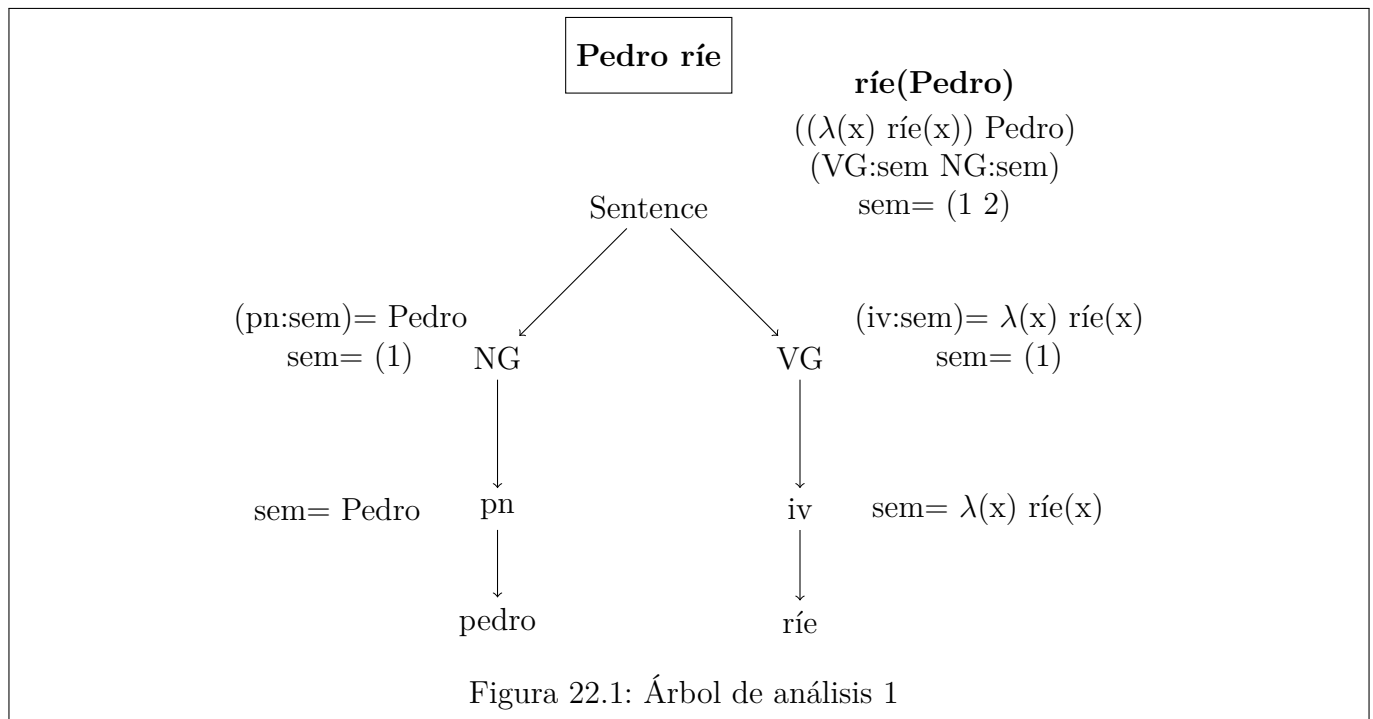
FRASE $\rightarrow$ GN FV, (2 1)

GN $\rightarrow$ np, (1)

FV $\rightarrow$ vi, (1)

FV $\rightarrow$ vt GN (1 2)

y el siguiente lexicón:



Pedro $\rightarrow$ np, pedro
 María $\rightarrow$ np, maria
 ríe $\rightarrow$ vi, $(\lambda(x), \text{reir}(x))$
 ama a $\rightarrow$ vt, $((\lambda(x), (\text{lambda}(y), \text{ama}(y,x))))$

La lista (2 1) que acompaña a la regla $\text{FRASE} \rightarrow \text{GN FV}$, indica que la IS de FRASE se obtendrá aplicando la IS de la segunda componente (FV), que deberá, por lo tanto ser una función con un solo parámetro, a la primera componente (GN). La lista (1) que acompaña a la regla $\text{GN} \rightarrow \text{np}$, indica que la IS de GN es la misma que la de np.

Fijémonos en que “Pedro” y “María” se interpretan como objetos individuales, “ríe” (verbo intransitivo) como función que aplicada a un objeto produce un enunciado y “ama a” como una función que aplicada a un objeto produce una función como la anterior.

Si aplicáramos la IS regla a regla al análisis de la frase “Pedro ríe” obtendríamos el árbol de análisis de la figura 22.1.

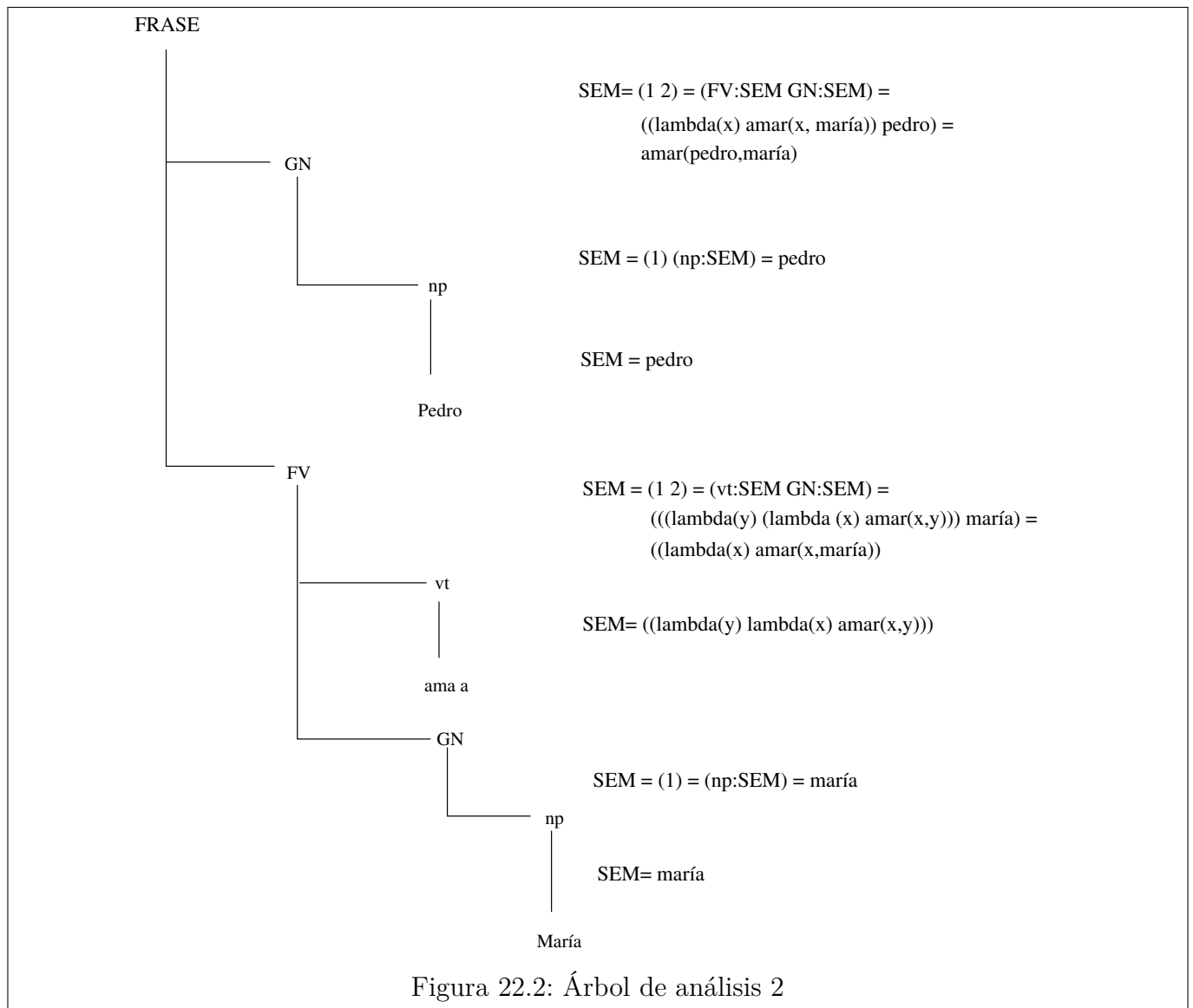
mientras que al analizar “Pedro ama a María” obtendríamos el árbol de análisis de la figura 22.2.

Es habitual, si se emplea la forma de interpretación regla a regla, el almacenamiento de las interpretaciones parciales como etiquetas del árbol de análisis (en nuestro ejemplo como valores del atributo SEM).

Si empleamos el proceso a posteriori, lo único que necesitamos almacenar es el orden de aplicación de los argumentos de la composición.

No siempre es adecuado un sistema de interpretación como el aquí presentado. A veces conviene utilizar más de una función de composición para atenuar la rigidez del método. En esos casos la información asociada no es únicamente el orden de aplicación sino también el tipo de operación a efectuar. Allen, por ejemplo, propone un método similar al presentado aquí, en el que se utilizan algunas operaciones más de composición. En cualquier caso, si el número de operaciones diferentes no es grande las mismas consideraciones que hemos hecho serían aplicables.

En todos los casos anteriores, la información básica de tipo semántico o es aportada por el léxico o se reduce a un mínimo (operación y orden de aplicación) que se debe incorporar a cada regla. Independientemente de que la IS se realice en paralelo o al final del análisis sintáctico se produce una interacción muy fuerte Sintaxis-Semántica. Difícilmente podrá escribir la parte semántica de las reglas quien no haya escrito la parte sintáctica. Existe, por supuesto, la aproximación contraria: la



que se basa en una descripción independiente de las reglas sintácticas y semánticas.

En esta otra aproximación el AS produce un árbol de análisis y sobre él se aplicará la IS. No existe una correspondencia regla a regla.

¿Cómo serán las reglas de IS en este caso? Suelen ser reglas que producen interpretaciones semánticas parciales que luego se combinarán entre sí (recordemos la compositividad) para formar IS más complejas.

El mecanismo de activación y actuación de estas reglas suele ser del tipo de los sistemas de producción. Las reglas constan de una condición y una acción. La condición se aplica sobre un fragmento del árbol de análisis y cuando tiene éxito se ejecuta la acción. Ésta suele incorporar nueva información al árbol de análisis. La condición acostumbra a implicar la actuación de un mecanismo de pattern-matching.

Así expuesto el sistema parece sencillo. No lo es. En primer lugar, un mecanismo de pattern-matching sobre los nodos de un árbol (potencialmente a diferente profundidad) que suponga el examen de todos los posibles cortes del árbol para intentar asociarlos a los patrones de todas las reglas de interpretación es a todas luces computacionalmente impracticable.

Es necesario limitar el espacio de búsqueda en ambos sentidos: no todos los cortes, no todas las reglas. Ello obliga, en primer lugar, a establecer algún tipo de indicación de las reglas (agrupación en paquetes, acceso eficiente, posiblemente a través de multiíndices, *etc*) y, en segundo lugar, a incluir algún tipo de mecanismo de control que establezca cuándo deben aplicarse qué reglas.

Consideremos el siguiente ejemplo, tomado de Allen.

Supongamos que la frase “una manzana verde” ha sido analizada sintácticamente dando lugar al siguiente árbol de análisis:

```
(NP  DET una
      HEAD manzana
      ADJ verde)
```

Supongamos que indexadas por lexemas se encuentran las siguientes reglas de interpretación semántica:

```
(verde 1)      (NP  ADJ verde
                  HEAD +objeto-fisico) → (? * T(HEAD) (COLOR * verde))
(verde 2)      (NP  ADJ verde
                  HEAD +fruta) → (? * T(HEAD) (verde *))
(una 1)        (NP  DET una
                  NUM sg) → (INDEF/SG * ? ?)
(manzana 1)    (NP  HEAD manzana) → (? * manzana)
```

Las reglas que Allen propone constan de una condición, que debe emparejarse con un fragmento del árbol de análisis, y una acción, que consiste en una forma lógica completada parcialmente. Las formas lógicas que Allen utiliza suelen ser tuplas de 4 elementos. El primero es el operador, que depende del tipo de objeto, frase, grupo nominal, nombre propio, *etc*, que se trata de describir. El segundo es el identificador (único) del objeto, el tercero, el tipo o clase genérica a que pertenece, el cuarto, la lista de posibles modificadores.

Las dos primeras reglas de interpretación corresponden a dos interpretaciones de "verde": color, si se trata de un objeto físico, grado de maduración si se trata de una fruta. La interpretación semántica produciría los intentos de emparejamiento de las reglas anteriores con el resultado del árbol de análisis y la posterior unificación de los resultados. El resultado final sería en este caso:

```
(INDEF/SG M1 manzana {(color M1 verde) (verde M1)})
```

Este sistema separa completamente la IS del AS (sólo el conocimiento de las formas posibles de árbol de análisis es necesario). Ahora bien, esto no quiere decir que la IS se deba llevar a cabo obligatoriamente después del AS. Podemos perfectamente interpretar semánticamente cualquier fragmento de árbol de análisis, no necesariamente el completo, una vez esté construido, y almacenar las interpretaciones parciales en el árbol. De esta manera es posible implementar una estrategia cooperativa entre sintaxis y semántica sin vernos obligados a desarrollar en paralelo las dos fuentes de conocimiento. Si usáramos esta aproximación, iríamos construyendo y decorando semánticamente el árbol de análisis en forma incremental. Los mismos argumentos que se adujeron al hablar de la IS regla a regla serían aplicables aquí.

Parte V

Aprendizaje Automático

23.1 Introducción

Todas las técnicas que hemos visto hasta ahora están encaminadas a desarrollar aplicaciones para problemas que necesitan inteligencia en su resolución. Fundamentalmente, éstas están orientadas a incluir conocimiento del dominio para facilitar su resolución, intentando evitar el coste computacional que implican los algoritmos generales.

Una de las limitaciones que podemos percibir es que estas aplicaciones no podrán resolver problemas para las que no haya sido programadas. Es decir, sus límites están impuestos por el conocimiento que hemos integrado en ellas.

Si queremos tener aplicaciones que podamos considerar inteligentes en un sentido amplio, estas han de ser capaces de ir mas allá de este límite. De hecho nos parecería mas inteligente una aplicación capaz de adaptarse y poder integrar nuevo conocimiento, de manera que pueda resolver nuevos problemas.

El área de aprendizaje automático dentro de la inteligencia artificial es la que se encarga de estas técnicas. La idea es buscar métodos capaces de aumentar las capacidades de las aplicaciones habituales (sistemas basados en el conocimiento, tratamiento del lenguaje natural, robótica, ...) de manera que puedan ser mas flexibles y eficaces.

Hay diversas utilidades que podemos dar al aprendizaje en los programas de inteligencia artificial (y en el resto también) podemos citar tres esenciales:

- **Tareas difíciles de programar:** Existen muchas tareas excesivamente complejas en las que construir un programa capaz de resolverlas es prácticamente imposible. Por ejemplo, si queremos crear un sistema de visión capaz de reconocer un conjunto de caras sería imposible programar a mano ese reconocimiento. El aprendizaje automático nos permitiría construir un modelo a partir de un conjunto de ejemplos que nos haría la tarea de reconocimiento. Otras tareas de este tipo lo constituirían ciertos tipos de sistemas basados en el conocimiento (sobre todo los de análisis), en los que a partir de ejemplos dados por expertos podríamos crear un modelo que realizara su tarea
- **Aplicaciones auto adaptables:** Muchos sistemas realizan mejor su labor si son capaces de adaptarse a las circunstancias. Por ejemplo, podemos tener una aplicación que adapte su interfaz a la experiencia del usuario. Un ejemplo bastante cercano de aplicaciones auto adaptables son los gestores de correo electrónico, que son capaces de aprender a distinguir entre el correo no deseado y el correo normal.
- **Minería de datos/Descubrimiento de conocimiento:** El aprendizaje puede servir para ayudar a analizar información, extrayendo de manera automática conocimiento a partir de

conjuntos de ejemplos (usualmente millones) y descubriendo patrones complejos.

23.2 Tipos de aprendizaje

El área de aprendizaje automático es relativamente amplia y ha dado lugar a muchas técnicas diferentes de aprendizaje, podemos citar las siguientes:

- **Aprendizaje inductivo:** Se pretenden crear modelos de conceptos a partir de la *generalización* de conjuntos de ejemplos. Buscamos descripciones simples que expliquen las características comunes de esos ejemplos.
- **Aprendizaje analítico o deductivo:** Aplicamos la deducción para obtener descripciones generales a partir de un ejemplo de concepto y su explicación. Esta generalización puede ser memorizada para ser utilizada en ocasiones en las que nos encontremos con una situación parecida a la del ejemplo.
- **Aprendizaje genético:** Aplica algoritmos inspirados en la teoría de la evolución para encontrar descripciones generales a conjuntos de ejemplos. La exploración que realizan los algoritmos genéticos permiten encontrar la descripción mas ajustada a un conjunto de ejemplos.
- **Aprendizaje conexionista:** Busca descripciones generales mediante el uso de la capacidad de adaptación de redes de neuronas artificiales. Una red neuronal esta compuesta de elementos simples interconectados que poseen estado. Tras un proceso de entrenamiento, el estado en el que quedan las neuronas de la red representa el concepto aprendido.

23.3 Aprendizaje Inductivo

Se trata del área de aprendizaje automático mas extensa y mas estudiada (de hecho los aprendizajes genético y conexionista también son inductivos). Su objetivo es descubrir descripciones generales que permitan capturar las características comunes de un grupo de ejemplos. El fundamento de estos métodos es la observación de las similitudes entre los ejemplos y la suposición de que se puede generalizar a partir de un número limitado de observaciones.

Esta generalización a partir de un conjunto limitado de ejemplos cae dentro de lo que denominaremos razonamiento inductivo, en contraste con el razonamiento deductivo. El razonamiento inductivo permite obtener conclusiones generales a partir de información específica. Estas conclusiones evidentemente son conocimiento nuevo, ya que no estaba presente en los datos iniciales. Este tipo de razonamiento tiene la característica de no preservar la verdad, en el sentido de que nuevo conocimiento puede invalidar lo que hemos dado por cierto mediante el proceso inductivo. Por ejemplo, si observamos un conjunto de instancias de aves y vemos que todas vuelan, podemos sacar la conclusión de que todas las aves vuelan (inducción), pero cuando nos presentan como nuevo ejemplo un pingüino, veremos que la conclusión a la que habíamos llegado no era cierta, ya que no cubre la nueva observación. Por esta razón se dice que el razonamiento inductivo (y en consecuencia el aprendizaje inductivo) no preserva la verdad (*non truth preserving*). El mayor problema del razonamiento inductivo es que no tiene una base teórica sólida como el razonamiento deductivo que, al contrario que el primero, sí preserva la verdad (ningún nuevo ejemplo puede invalidar lo que deducimos) y está soportado por la lógica matemática.

Los métodos inductivos son de naturaleza heurística, no existe una teoría o procedimientos que los fundamenten de manera sólida. Se basan en la suposición de que la observación de un número

limitado de ejemplos es suficiente para sacar conclusiones generales, a pesar de que un solo ejemplo puede invalidarlas¹. Pese a esta falta de fundamento en el aprendizaje inductivo, podemos observar que gran parte del aprendizaje humano es de este tipo, de hecho la mayor parte de las ciencias se basan en él.

23.3.1 Aprendizaje como búsqueda

El aprendizaje inductivo se suele plantear como una búsqueda heurística. El objetivo de la búsqueda es descubrir la representación que resuma las características de un conjunto de ejemplos. Esta representación puede ser una función, una fórmula lógica o cualquier otra forma de representación que permita generalizar.

En esta búsqueda, consideraremos como *espacio de búsqueda* todas las posibles representaciones que podemos construir con nuestro lenguaje de representación. El objetivo de la búsqueda es encontrar la representación que describa los ejemplos. El tamaño de este espacio de búsqueda dependerá del lenguaje de representación que escojamos. Evidentemente, cuanto más expresivo sea el lenguaje más grande será este espacio de búsqueda. Por ejemplo, si queremos poder expresar nuestro concepto mediante una fórmula conjuntiva pura tenemos $O(3^n)$ posibles conceptos, si queremos expresarlo mediante una fórmula en FND tenemos $O(2^{2^n})$ posibles conceptos.

Para describir un problema de búsqueda necesitamos dos elementos más aparte del espacio de búsqueda, un conjunto de *operadores de cambio de estado* y una *función heurística*. Los operadores de cambio de estado dependerán del método de aprendizaje que empleemos y el lenguaje de representación, pero han de ser capaces de modificar la representación que utilizamos para describir el concepto que queremos aprender. La función heurística también depende del método, ésta nos ha de guiar en el proceso de encontrar la descripción que incluye a los ejemplos de que disponemos. La mayoría de métodos utilizan como criterio de preferencia una función que permita obtener la descripción mas pequeña consistente con los ejemplos.

La razón de utilizar este sesgo en la función heurística es simple. Por lo general deseamos que la descripción que obtengamos sea capaz de prever ejemplos no vistos. Si la descripción que obtenemos es la más pequeña, existe menos riesgo de que tengamos características que sean específicas de los ejemplos que hemos utilizado para construirla, de manera que probablemente será suficientemente general para capturar el concepto real. Esta preferencia por los conceptos mínimos proviene del principio denominado **navaja de Occam**².

23.3.2 Tipos de aprendizaje inductivo

Podemos distinguir dos tipos de aprendizaje inductivo, el **supervisado** y el **no supervisado**.

El aprendizaje supervisado es aquel en el que para cada ejemplo que tenemos, conocemos el nombre del concepto al que pertenece, lo que denominaremos la **clase** del ejemplo. Por lo tanto la entrada de este tipo de aprendizaje es un conjunto de ejemplos clasificados. El aprendizaje se realiza por contraste, pretendemos distinguir los ejemplos de una clase de los del resto. Dispondremos de un conjunto de operadores capaces de generar diferentes hipótesis sobre el concepto a aprender y tendremos una función heurística que nos permitirá elegir la opción más adecuada. El resultado del proceso de aprendizaje es una representación de las clases que describen los ejemplos.

El aprendizaje no supervisado es más complejo, no existe una clasificación de los ejemplos y debemos encontrar la mejor manera de estructurarlos, obteniendo por lo general una partición en grupos. El proceso de aprendizaje se guía por la similaridad/disimilaridad de los ejemplos, construyendo

¹Citando a Albert Einstein: "Ninguna cantidad de experimentos pueden probar que estoy en lo cierto, un solo experimento puede demostrar que estoy equivocado"

²William of Occam (1285-1349) "Dadas dos teorías igualmente predictivas, es preferible la más simple"

grupos en los que los ejemplos similares están juntos y separados de otros ejemplos menos similares. El resultado de este proceso de aprendizaje es una partición de los ejemplos y una descripción de los grupos de la partición.

23.4 Árboles de Inducción

Podemos plantear el aprendizaje de un concepto como el averiguar qué conjunto mínimo de preguntas hacen falta para distinguirlo de otros (algo así como el juego de las 20 preguntas). Este conjunto de preguntas nos servirá como una caracterización del concepto.

Nuestro objetivo en este método de aprendizaje es conseguir distinguir conjuntos de ejemplos pertenecientes a diferentes conceptos (clases). Para representar la descripción de los conceptos de cada clase usaremos como lenguaje de representación un árbol de preguntas (*árbol de inducción*) que almacenará las preguntas que nos permitirán distinguir entre los ejemplos de los conceptos que queramos aprender. Cada nodo del árbol será una pregunta y tendremos para cada nodo tantas ramas como respuestas pueda tener esa pregunta. Las hojas de este árbol corresponderán con la clase a la que pertenecen los ejemplos que tienen como respuestas las correspondientes al camino entre la raíz y la hoja.

El lenguaje de descripción de los árboles de inducción corresponde a las fórmulas lógicas en FND, tenemos por lo tanto $O(2^{2^n})$ descripciones posibles. Evidentemente, no podemos plantearlos explorar todos para encontrar el más adecuado, por lo que tendremos que utilizar búsqueda heurística.

Los algoritmos de construcción de árboles de inducción siguen una estrategia Hill-Climbing. Se parte de un árbol vacío y se va particionando el conjunto de ejemplos eligiendo a cada paso el atributo que mejor discrimina entre las clases. El operador de cambio de estado es la elección de este atributo. La función heurística será la que determine qué atributo es el mejor, esta elección es irrevocable, por lo que no tenemos garantía de que sea la óptima. La ventaja de utilizar esta estrategia es que el coste computacional es bastante reducido.

La función heurística ha de poder garantizarnos que el atributo elegido minimiza el tamaño del árbol. Si la función heurística es buena el árbol resultante será cercano al óptimo.

El algoritmo básico de árboles de inducción fue desarrollado por Quinlan³ y es conocido como algoritmo **ID3**. Este algoritmo basa su función heurística para la selección del mejor atributo en cada uno de los niveles del árbol en la teoría de la información. En concreto el algoritmo utiliza la noción de entropía de Shannon, que se puede usar como una medida de la aleatoriedad de la distribución estadística de un conjunto de ejemplos sobre las clases a las que pertenecen.

La teoría de la información estudia entre otras cosas los mecanismos de codificación de mensajes y el coste de su transmisión. Si definimos un conjunto de mensajes $M = \{m_1, m_2, \dots, m_n\}$, cada uno de ellos con una probabilidad $P(m_i)$, podemos definir la cantidad de información (I) contenida en un mensaje de M como:

$$I(M) = \sum_{i=1}^n -P(m_i) \log(P(m_i))$$

Este valor se puede interpretar como la información necesaria para distinguir entre los mensajes de M (Cuántos bits de información son necesarios para codificarlos). Dado un conjunto de mensajes podemos obtener la mejor manera de codificarlos para que el coste de su transmisión sea mínimo⁴.

Podemos hacer la analogía con la codificación de mensajes suponiendo que las clases son los mensajes y la proporción de ejemplos de cada clase su probabilidad. Podemos ver un árbol de decisión como la codificación que permite distinguir entre las diferentes clases. El objetivo es encontrar el mínimo árbol (codificación) que nos permite distinguir los ejemplos de cada clase. Cada atributo se deberá evaluar para decidir si se le incluye en el código (árbol). Un atributo será mejor cuanto mas permita discriminar entre las diferentes clases.

Como se ha comentado, el árbol se construye a partir de cero de manera recursiva, en cada nodo del árbol debemos evaluar que atributo permite minimizar el código (minimizamos el tamaño del

³J. R. Quinlan, *Induction of Decision Trees*, Machine Learning, 1(1) 81-106, 1986.

⁴En estos mismos principios se basan también, por ejemplo, los algoritmos de compresión de datos.

Algoritmo 23.1 Algoritmo de aprendizaje de árboles de decisión

Algoritmo: ID3 ($\mathcal{X}$: Ejemplos, $\mathcal{C}$: Clasificación, $\mathcal{A}$: Atributos)

si todos los ejemplos son de la misma clase

entonces

| **retorna** una hoja con el nombre de la clase

sino

| Calcular la función de cantidad de información de los ejemplos (**I**)

| **para cada** atributo en $\mathcal{A}$ **hacer**

| | Calcular la función de entropía (**E**) y la ganancia de información (**G**)

| **fin**

| Escoger el atributo que maximiza **G** (**a**)

| Eliminar **a** de la lista de atributos ($\mathcal{A}$)

| Generar un nodo raíz para el atributo **a**

| **para cada** Partición generada por los valores v_i del atributo **a** **hacer**

| | $\text{Árbol}_i = \text{ID3}(\text{ejemplos de } \mathcal{X} \text{ con } \mathbf{a} = \mathbf{v}_i, \text{ Clasificación de los ejemplos, Atributos restante})$

| | Generar una nueva rama con $\mathbf{a} = \mathbf{v}_i$ y Árbol_i

| **fin**

| **retorna** El nodo raíz para **a**

fin

árbol). Este atributo será el que haga que la cantidad de información que quede por cubrir sea la menor posible, es decir, el que minimice el número de atributos que hace falta añadir para discriminar totalmente los ejemplos del nodo. La elección de un atributo debería hacer que los subconjuntos de ejemplos que genera el atributo sean mayoritariamente de una clase. Para medir esto necesitamos una medida de la cantidad de información que no cubre un atributo (medida de Entropía, E)

La formulación es la siguiente, dado un conjunto de ejemplos X clasificados en un conjunto de clases $\mathcal{C} = \{c_1, c_2, \dots, c_n\}$, siendo $\#c_i$ la cardinalidad de la clase c_i y $\#\mathcal{X}$ el numero total de ejemplos, se define la función *cantidad de información* como:

$$I(\mathcal{X}, \mathcal{C}) = - \sum_{c_i \in \mathcal{C}} \frac{\#c_i}{\#\mathcal{X}} \cdot \log\left(\frac{\#c_i}{\#\mathcal{X}}\right)$$

Para cada uno de los atributos A_i , siendo $\{v_{i1}, \dots, v_{in}\}$ el conjunto de modalidades del atributo A_i y $\#[A_i(\mathcal{C}) = v_{ij}]$ el numero de ejemplos que tienen el valor v_{ij} en su atributo A_i , se define la función de *entropía* como:

$$E(\mathcal{X}, \mathcal{C}, A_i) = \sum_{v_{ij} \in A_i} \frac{\#[A_i(\mathcal{C}) = v_{ij}]}{\#\mathcal{X}} \cdot I([A_i(\mathcal{C}) = v_{ij}], \mathcal{C})$$

Con estas dos medidas se define la función de *ganancia de información* para cada atributo como:

$$G(\#\mathcal{X}, \mathcal{C}, A_i) = I(X, \mathcal{C}) - E(X, A_i)$$

El atributo que maximice este valor se considera como la mejor elección para expandir el siguiente nivel del árbol (es el atributo que menor cantidad de información deja por cubrir).

El algoritmo que se sigue para la generación de árboles de decisión es que se puede ver en el algoritmo 23.1.

Este algoritmo de generación de árboles de decisión no es el único que existe, por ejemplo se

pueden escoger otras heurísticas para hacer la elección de atributos en cada nodo⁵ o se puede hacer más de una pregunta en cada nodo del árbol.

Un árbol de decisión se puede transformar fácilmente en otras representaciones. Por ejemplo, para generar un conjunto de reglas a partir de un árbol es suficiente con recorrer todos los caminos desde la raíz hasta las hojas generando una regla con los atributos y valores que aparecen en los nodos de cada camino.

Ejemplo 23.1 *Tomemos la siguiente clasificación:*

<i>Ej.</i>	<i>Ojos</i>	<i>Cabello</i>	<i>Estatura</i>	<i>Clase</i>
1	Azules	Rubio	Alto	+
2	Azules	Moreno	Medio	+
3	Marrones	Moreno	Medio	−
4	Verdes	Moreno	Medio	−
5	Verdes	Moreno	Alto	+
6	Marrones	Moreno	Bajo	−
7	Verdes	Rubio	Bajo	−
8	Azules	Moreno	Medio	+

Si calculamos los valores para las funciones I y E :

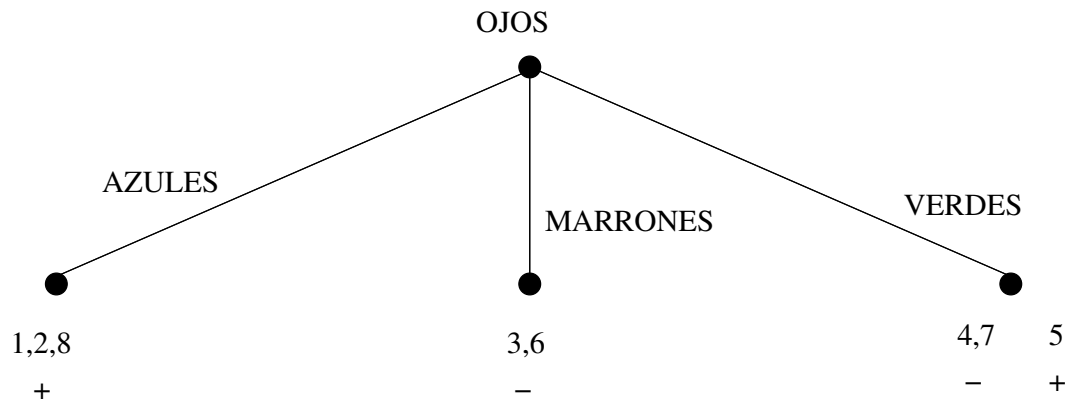
$$\begin{aligned}
 I(X, C) &= -1/2 \cdot \log(1/2) - 1/2 \cdot \log(1/2) = 1 \\
 E(X, ojos) &= (azul) \, 3/8 \cdot (-1 \cdot \log(1) - 0 \cdot \log(0)) \\
 &\quad + (marrones) \, 2/8 \cdot (-1 \cdot \log(1) - 0 \cdot \log(0)) \\
 &\quad + (verde) \, 3/8 \cdot (-1/3 \cdot \log(1/3) - 2/3 \cdot \log(2/3)) \\
 &= 0.344 \\
 E(X, cabello) &= (rubio) \, 2/8 \cdot (-1/2 \cdot \log(1/2) - 1/2 \cdot \log(1/2)) \\
 &\quad + (moreno) \, 6/8 \cdot (-1/2 \cdot \log(1/2) - 1/2 \cdot \log(1/2)) \\
 &= 1 \\
 E(X, estatura) &= (alto) \, 2/8 \cdot (-1 \cdot \log(1) - 0 \cdot \log(0)) \\
 &\quad + (medio) \, 4/8 \cdot (-1/2 \cdot \log(1/2) - 1/2 \cdot \log(1/2)) \\
 &\quad + (bajo) \, 2/8 \cdot (0 \cdot \log(0) - 1 \cdot \log(1)) \\
 &= 0.5
 \end{aligned}$$

*Como podemos comprobar, es el atributo **ojos** el que maximiza la función.*

$$\begin{aligned}
 G(X, ojos) &= 1 - 0.344 = 0.656 * \\
 G(X, cabello) &= 1 - 1 = 0 \\
 G(X, estatura) &= 1 - 0.5 = 0.5
 \end{aligned}$$

Este atributo nos genera una partición que forma el primer nivel del árbol.

⁵La entropía no es el único criterio, de hecho no existe ninguna heurística que sea mejor que el resto y el resultado depende muchas veces del dominio de aplicación



Ahora solo en el nodo correspondiente al valor **verdes** tenemos mezclados objetos de las dos clases, por lo que repetimos el proceso con esos objetos.

Ej.	Cabello	Estatura	Clase
4	Moreno	Medio	-
5	Moreno	Alto	+
7	Rubio	Bajo	-

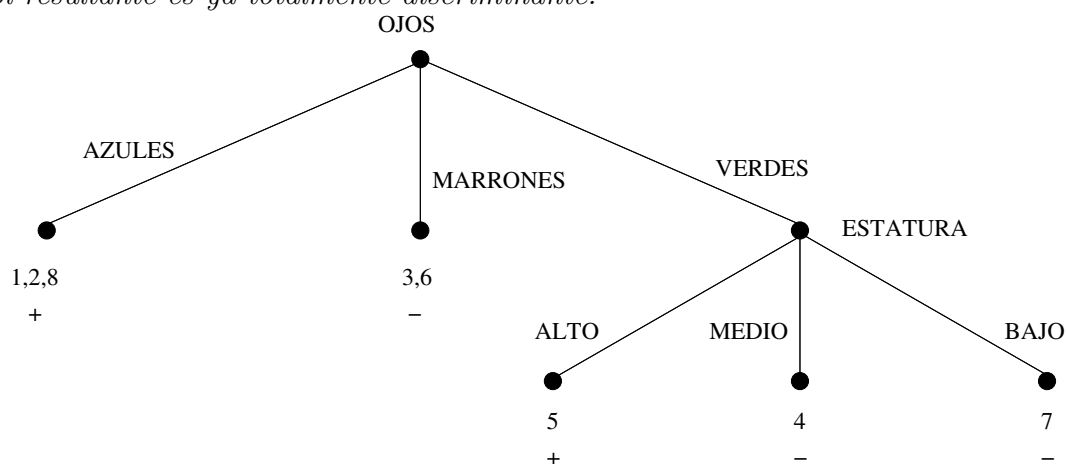
Si calculamos los valores para las funciones I y E :

$$\begin{aligned}
 I(X, C) &= -1/3 \cdot \log(1/3) - 2/3 \cdot \log(2/3) = 0.918 \\
 E(X, \text{cabello}) &= (\text{rubio}) \, 1/3 \cdot (0 \cdot \log(0) - 1 \cdot \log(1)) \\
 &\quad + (\text{moreno}) \, 2/3 \cdot (-1/2 \cdot \log(1/2) - 1/2 \cdot \log(1/2)) \\
 &= 0.666 \\
 E(X, \text{estatura}) &= (\text{alto}) \, 1/3 \cdot (0 \log(0) - 1 \cdot \log(1)) \\
 &\quad + (\text{medio}) \, 1/3 \cdot (-1 \cdot \log(1) - 0 \cdot \log(0)) \\
 &\quad + (\text{bajo}) \, 1/3 \cdot (0 \cdot \log(0) - 1 \cdot \log(1)) \\
 &= 0
 \end{aligned}$$

Ahora el atributo que maximiza la función es **ojos**.

$$\begin{aligned}
 G(X, \text{cabello}) &= 0.918 - 0.666 = 0.252 \\
 G(X, \text{estatura}) &= 0.918 - 0 = 0.918*
 \end{aligned}$$

El árbol resultante es ya totalmente discriminante.



Este árbol se podría reescribir por ejemplo como un conjunto de reglas:

(Ojos = Azul $\rightarrow$ C+)

(Ojos = Marrones $\rightarrow$ C-)

(Ojos = Verdes y Altura = Medio $\rightarrow$ C-)

(Ojos = Verdes y Altura = Bajo $\rightarrow$ C-)

(Ojos = Verdes y Altura = Alto $\rightarrow$ C+)

Bibliografía

- [1] James Allen *Natural language understanding*, Benjamin /Cummings, 1995.
- [2] Joseph Giarratano, Gary Riley *Expert systems: principles and programming*, Thomson Course Technology, 2005.
- [3] Avelino J. Gonzlez, Douglas D. Dankel *The Engineering of knowledge-based systems : theory and practice*, Prentice Hall, 1993.
- [4] Jackson, P. *Introduction to Expert Systems*, Addison-Wesley, 1990.
- [5] Noy & McGuinness *Ontology Development 101: A Guide to Creating Your First Ontology*, (2000)
- [6] Stuart J. Russell and Peter Norvig *Artificial intelligence: a modern approach*, Prentice Hall, 2003.

Índice alfabético

- A*, 25
- Abstracción de datos, 145
- Adecuación Inferencial, 78
- Adecuación Representacional, 78
- Adquisición de conocimiento, 137
- Agentes Inteligente, 133
- Algoritmo de distancia inferencial, 97
- Algoritmo de unificación, 83
- Algoritmo genético canónico, 50
- Algoritmo minimax, 58
- Algoritmos genéticos, 46
- ALVEY, 186, 200
- Análisis, 144
- análisis léxico, 196
- analizador, 208
- analizadores activados por islas, 213
- analizadores de dos niveles, 203
- analizadores de un nivel, 203
- analizadores regidos por el núcleo, 213
- aplicaciones del lenguaje natural, 191
- arco-consistencia, 67
- Asociación heurística, 145
- ATN, 208

- Búsqueda en anchura, 13
- Búsqueda en haces, 38
- Búsqueda en profundidad, 14
- Búsqueda en profundidad iterativa, 17
- Backtracking cronológico, 65
- Backward chaining, 91
- Base de casos, 128
- Base de conocimiento, 86, 124
- Base de Hechos, 86
- Beam search, 38
- Branch and bound, 37
- Case Based Reasoning, 127

- CG, 189, 219
- Chomsky, N., 188
- Ciclo de vida, 138
- Cláusulas de Horn, 85
- Clasificación de los SBC, 143
- Clasificación Heurística, 145
- Colmerauer, A., 222
- CommonKADS, 139
- Conectivas difusas, 171
- Conjunto de conflicto, 88
- conjunto de conflicto, 87
- Conjunto difuso, 170
- Conocimiento, 75
- conocimiento de sentido común, 194
- Conocimiento declarativo, 78
- Conocimiento heredable, 79
- Conocimiento inferible, 79
- Conocimiento procedimental, 79
- Conocimiento relacional, 79
- Constraint satisfaction, 63
- Constructive Problem Solving, 150
- CYC, 202

- DAG, 197
- DCG, 189, 219
- DCTG, 222
- Demons, 97
- Description Logics, 97
- Diagram, 186
- Discontinuous Grammars, 223
- DISLOG, 223
- dotted rules, 216

- Eficiencia en la Adquisición, 78
- Eficiencia Inferencial, 78
- Eliminación de variables, 164
- Espacio de estados, 6

- Esquema de representación, 76
Estado, 6
Estrategia de control, 87
Estrategia de enfriamiento, 43
Estrategia de resolución de conflictos, 87
Etiqueta lingüística, 169
Expert Systems, 115

Facet, 96
Factores, 164
Feature Structures, 197
FG, 222
Forma normal de skolem, 83
formalismos de unificación, 219
Forward chaining, 90
Forward checking, 69
frame, 190
Frame problem, 77
Frames, 96
FUG, 223
Función de adaptación, 47
Funcion de fitness, 47

Gap Grammars, 223
GPSG, 219
gramática, 208
Gramática Generativa, 188
Gramática Transformacional Generativa, 188
gramáticas categoriales de unificación, 223
gramáticas de casos, 189
gramáticas de contexto libre, 209
gramáticas de extraposición, 223
gramáticas de metamorfosis, 222
gramáticas de rasgos, 223
gramáticas de unificación, 223
gramáticas funcionales, 222
gramáticas funcionales de unificación, 223
gramáticas léxico funcionales, 222
gramáticas lógicas, 219
gramáticas regulares, 209
gramáticas sensitivas, 209
greedy best first, 24

Hechos, 86
Herência, 97
Herência múltiple, 97
Heuristic Classification, 145
Hill-climbing, 37
HPSG, 223

IDA*, 30
Independencia condicional, 161

Inferencia difusa, 178
Inferencia difusa de Sugeno, 178
Inferencia probabilística, 163
Información, 75
Ingeniero del conocimiento, 135
Intérprete de reglas, 87
interfaces en lenguaje natural, 192
interfaces multimodales, 192
interpretación semántica, 227
Iterative Deepening, 17

Justificación de la solución, 126

k-consistencia, 67
Knowledge Based Systems, 115
Knowledge Elicitation, 137
Koskeniemi, K., 203

Lógica de la descripción, 97
Lógica difusa, 170
Lógica Posibilista, 169
Lógica proposicional, 81
Lakoff, G., 188
Least Commitment, 151
lenguaje natural, 185
lexicón, 195
LFG, 222
lingüística computacional, 185

Método de resolución, 82
Métodos, 97
Métodos débiles, 115
Métodos fuertes, 115
Mínimo Compromiso, 151
Marcos, 96
matrices de rasgos, 197
Memoria de trabajo, 126
Memory bound A*, 33
Meta-reglas, 125
Metaconocimiento, 119
MIKE, 140
MLG, 219
Modelo en cascada, 135
Modelo en espiral, 136
Modelo probabilista, 157
Montague, R., 189
Motor de inferencia, 125
Motor de inferencias, 87

Negación fuerte, 172
Nitidificación, 179
no-determinismo, 206

- One point crossover, [48](#)
Operador, [6](#)
Operador genético de cruce, [48](#)
Operador genético de mutación, [48](#)

Paradoja del experto, [119](#)
PATR-II, [219](#)
Población, [49](#)
poda alfa-beta, [58](#)
Poliárbol, [168](#)
Preguntas de competencia, [106](#)
Propagación de restricciones, [67](#)
Proponer y Aplicar, [150](#)
Propose and Apply, [150](#)
Prototipado Rápido, [138](#)
psicolingüística, [186](#)

Q-Systems, [222](#)

Random restarting hill climbing, [38](#)
Rapid Prototyping, [138](#)
Razonamiento Basado en Casos, [127](#)
Razonamiento Basado en Modelos, [131](#)
Razonamiento guiado por los datos, [90](#)
Razonamiento guiado por los objetivos, [91](#)
Razonamiento hacia adelante, [90](#)
Razonamiento hacia atrás, [91](#)
Recuperación de casos, [127](#)
Recursive best first, [31](#)
Red bayesiana, [161](#)
redes de transición, [205](#)
redes de transición aumentadas, [208](#)
Redes Neuronales, [130](#)
Redes semánticas, [95](#)
redes semánticas, [190](#)
Refinamiento de la solución, [146](#)
Regla de Bayes, [161](#)
Regla del producto, [158](#)
Reglas de producción, [85](#)
Relación de accesibilidad, [6](#)
Resolución Constructiva, [150](#)
Resolución de problemas, [145](#)
Restriction Grammars, [223](#)
Ritchie, G.D., [200](#), [203](#)

Síntesis, [144](#)
Satisfacción de restricciones, [63](#)
semántica, [227](#)
semántica compositiva, [228](#)
SHRDLU, [191](#)
Simulated annealing, [43](#)
Sistemas Basados en el Conocimiento, [115](#)

Sistemas Expertos, [115](#)
Sistemas multiagente, [133](#)
Slot, [96](#)
Slot Grammars, [223](#)
Solución, [7](#)
Static Discontinuous Grammars, [223](#)
Steedman, M.J., [223](#)
Steepest ascent hill climbing, [37](#)
Sublenguajes, [185](#)

T-conorma, [172](#)
T-norma, [171](#)
tablas de cadenas bien formadas, [216](#)
traducción automática, [191](#)

UCG, [223](#)
UG, [223](#)
unificación, [222](#)
Universo del discurso, [169](#)

Variables observadas, [163](#)
Variables ocultas, [163](#)

Woods, W.A., [208](#)