INTELIGENCIA ARTIFICIAL:
INTRODUCCIÓN Y TAREAS DE BÚSQUEDA

VERSIÓN 20100619

PRESENTACIÓN

El presente documento surge de los apuntes tomados de diversas fuentes durante mi paso por la Ingeniería Técnica en Informática de Sistemas de la Universidad Nacional de Educación a Distancia.

Tiene pocos ejemplos y ningún ejercicio propuesto: solo es un tratado teórico. Este inconveniente se puede solventar ampliamente si se consulta [Fernández et al 1998], pues se trata de un libro de problemas enfocado, entre otros temas, a las tareas de búsqueda.

Por último, aunque el documento está enfocado hacia la asignatura de la UNED antes mencionada, el lector no tendrá problemas en su comprensión si tiene conocimientos previos de programación, complejidad algorítmica y grafos (de este último se incluye un apéndice).

AVISO DE DERECHOS DE AUTOR

Roberto J. de la Fuente López

INDICE

CAPÍTULO 1.- FUNDAMENTOS DE INTELIGENCIA ARTIFICIAL

1.1.- JERARQUÍA DE NIVELES

Dado un sistema, sea de la naturaleza que sea, podemos verlo desde dos perspectivas:

- **Análisis**.- Escudriñamos (estudiamos) el sistema para comprender como funciona y así fijar sus especificaciones funcionales, qué hace el sistema, de una manera formal. Este punto de vista es propio de cualquier ciencia como la Física o la Biología.

- **Síntesis**.- Obtención del sistema a partir de unas especificaciones funcionales (creamos el sistema). Este punto de vista es propio de la las ingenierías.

Vamos a utilizar como ejemplo un sistema electrónico: Nos lo pueden mostrar descrito con esquemas en papel. Con los conocimientos de electrónica que tenemos, *analizamos* el circuito en varios pasos, (por ejemplo, primero por partes y luego de forma global) deduciendo qué es lo que hace y cómo lo hace (por ejemplo, extrayendo las ecuaciones que lo describen). Por el contrario, podemos tener que diseñar un circuito para resolver un problema concreto; para ello tenemos que *sintetizar* el problema (extraer sus características relevantes, que pueden ser las ecuaciones que lo definen) para posteriormente, y después de una serie de pasos, implementarlo con alguna tecnología.

Un sistema, por regla general, es muy complejo por lo que se hace necesario segmentarlo en unidades funcionales más pequeñas. Esta segmentación la realizaremos mediante una **jerarquía de niveles**. Cada uno de estos (los pasos a los que hacíamos referencia en el ejemplo anterior) se caracteriza por:

- Fenomenología propia del nivel.
- Entidades (componentes) y relaciones entre ellos (interconexiones).

- Organización y estructura propias (sintaxis propia, "lenguaje"[1] que se habla a este nivel).
- Restricciones de capacidad.

No hay que confundir la jerarquía de niveles con la abstracción: en esta última lo que hacemos es segmentar un sistema en unidades funcionales más pequeñas, de manera que el diseño se puede hacer por bloques, para posteriormente desarrollar cada uno de estos. Sin embargo estos bloques funcionales pueden estar al mismo nivel jerárquico. Mediante la abstracción podríamos no encontrar un enlace (relación entre dos niveles) de la descripción de un sistema a un nivel con la descripción en un nivel superior (esta puede no existir).

Un nivel está parcialmente encapsulado: cada uno se define y funciona de manera independiente. Para ello podemos describir un nivel con:

- **Espacio de entradas**.- Es un espacio multidimensional de características relevantes. Este espacio ha de ser medible.

- **Reglas de transformación**.- Descripción precisa de transformaciones sobre el espacio de entradas.

- **Espacio de salidas**.- También es un espacio multidimensional, que es el resultado de la aplicación en el tiempo, de las reglas de transformación sobre el espacio de entradas. Este espacio también ha de ser medible.

La fenomenología del nivel se puede describir mediante la partición de este en **medio** y **sistema.** Definimos el *medio* como el conjunto de características del espacio de entrada más el conjunto de resultados (espacio de salidas) y que son entendibles por el *sistema* (el que realiza las transformaciones en este nivel). El sistema "entiende y habla" el lenguaje en el que están escritos los espacios de entrada y salida.

Aunque los niveles son independientes, están relacionados los unos con los otros por medio de dos enlaces:

[1] Aquí se utiliza el término "lenguaje" en un sentido general

- **Reducción**.- Cada nivel se enlaza con el inferior por medio de una serie de especificaciones precisas de traducción:

    - El medio del nivel se utiliza para describir el sistema del nivel inferior.

    - En este proceso se pierde información, pues el resultado es más simple.

    - El resultado de un proceso de reducción no es único, pues puede dar lugar a varios niveles inferiores, todos ellos equivalentes entre sí.

    - El proceso de *reducción* es el que se produce durante la *síntesis* de un sistema.

- **Emergencia**.- Cada nivel se enlaza con el superior por medio de unas especificaciones:

    - Se *inyecta el conocimiento* perdido durante el proceso de reducción. Se añaden al sistema unas estructuras adicionales, *emergiendo* así entidades del nivel superior (el medio del nivel superior) y el sistema que las procesa.

    - El proceso de *emergencia* es el que se produce durante el *análisis* de un sistema.

Esta encapsulación parcial conlleva que la acción del sistema en un nivel sea invariante ante cambios en los niveles inferiores a los que han sido reducidos. Desde el punto de vista del sistema son totalmente independientes.

En la descripción realizada para la fenomenología no hemos hecho mención a la posible semántica de los espacios de representación (entradas-salidas). Esto es porque el proceso que realiza el sistema es causal, determinado por las leyes de transformación. El significado de las entidades se las da el experto humano que está analizando/sintetizando el sistema. Por ejemplo, un sumador aritmético digital hace que si la entrada tiene una cadena 010 y una cadena 001 a su entrada, a su salida tendrá una cadena 0011. Es el experto el que identifica que se está realizando la operación 2+1=3.

Hemos dicho que, dado un nivel n, en la reducción al nivel n-1 perdemos información y que en el proceso de emergencia al nivel n+1 tenemos que inyectar conocimiento para subir de nivel: para esto necesitamos conocer la semántica en este nivel. Por lo tanto, los espacios de entrada y salida estarán compuestos por pares (X_i, S_i) y (Y_j, S_i) respectivamente, donde X_i e Y_j serán características del medio y S_i y S_j sus correspondientes significados en este nivel. Al conjunto de estos pares lo vamos a llamar **tablas de semántica** en este nivel. Hay que hacer hincapié en que la semántica solo tiene sentido para el experto que está modelando el sistema (esta distinción se verá cuando definamos los dominios de descripción).

Una característica importante de cada nivel es la robustez frente a alteraciones sintácticas y semánticas. Los niveles más bajos, al ser más formales, son menos robustos (cualquier cambio en las entradas provoca un cambio en las salidas) que los superiores, que permiten ambigüedades y siguen siendo válidos.

Volvamos a nuestro ejemplo de sistema electrónico y describámoslo en el nivel inferior: la fenomenología será la Física (corrientes de electrones y huecos), sus entidades serán los dispositivos pasivos (resistencias, condensadores…) y/o activos (diodos, transistores….) interconectadas entre sí, la sintaxis (el lenguaje utilizado) serán las matemáticas (cálculo integral, diferencial…), y las restricciones estarán impuestas por la tecnología de implementación.

¿Hay algo raro en esta descripción? A simple vista parecería que no, pero adolece de un problema muy extendido en I.A.: Se están mezclando conceptos del nivel de la física de componentes con el nivel de símbolos de dispositivos. En este caso deberíamos definir las entidades como física de materiales. La mezcla de descripciones de un nivel en otro es una fuente frecuente de errores.

1.2.- JERARQUIA DE NIVELES EN COMPUTACIÓN

El primero en identificar niveles en computación fue Chomsky en su estudio de la jerarquía de lenguajes, en especial en la distinción que hizo, para los lenguajes naturales, entre la **competencia lingüística** y la **ejecución del lenguaje.**

- Competencia lingüística.- Conocimiento de las reglas gramaticales para la formación de frases.

- Ejecución del lenguaje.- Dado que un lenguaje natural no es formal, la manera en que se aplican las reglas gramaticales es ambigua; para su buena ejecución hay que tener en cuenta también la semántica y el contexto de la frase. Por ejemplo, si nos dicen: Hoy paseamos con Chispas. ¿paseamos con destellos? (si por fuese escrito, desecharíamos esta opción, pues empieza por mayúscula) ¿Chispas es el perro? ¿Chispas es el mote de una persona? ¿hay chispas en el ambiente?

La teoría de niveles en los modelos de computación fue introducida de forma independiente por David Marr y por Allen Newell. El primero buscaba una teoría computacional en percepción visual[2] y el segundo se percató de la dificultad de pasar directamente de la descripción del problema en lenguaje natural a la implementación del programa. Los dos determinaron la segmentación de la descripción de un modelo computacional en tres niveles. En la tabla se visualiza la equivalencia entre niveles de uno y otro.

<table>
<tr><th>D. Marr</th><th>A. Newell</th></tr>
<tr><td>Teoría computacional</td><td rowspan="2">Nivel de conocimiento</td></tr>
<tr><td rowspan="2">Representación
y algoritmo</td></tr>
<tr><td>Nivel simbólico</td></tr>
<tr><td>Implementación</td><td>Nivel físico</td></tr>
</table>

En la teoría computacional de Marr, se dice que hay que tener una clara compresión de qué hay que calcular planteándolo en lenguaje natural, analizando los posibles esquemas de solución en términos del conocimiento del dominio (del experto). En el nivel de conocimiento de Newell, que engloba a la teoría computacional de Marr, se indica que en muchas ocasiones no se dispone de una teoría de cálculo, sino especificaciones ambiguas quizás válidas en lenguaje natural, pero difíciles o imposibles de plantear a nivel simbólico

[2] Ver la definición de agente más adelante

(una teoría de cálculo es un procedimiento tangible, por tanto más específico que el conocimiento como tal).

En el nivel de representación y algoritmo de Marr, se dice que hay que definir un lenguaje que describa el medio (espacios de representación) y el sistema (los algoritmos) que los enlazan para su implementación simbólica. Este nivel se solapa con los de conocimiento y simbólico de Newell. Para este, el nivel simbólico es el formado por las estructuras de datos en lenguaje de alto nivel, gobernados por las gramáticas formales; es la representación del programa. Los algoritmos y la representación del medio pertenecen al nivel del conocimiento.

El nivel de implementación de Marr es equivalente al nivel físico de Newell, descrito en términos de procesadores físicos (circuitos lógicos combinacionales, secuenciales y de transferencia de registros) gobernados por el álgebra de Boole.

1.2.1. Los agentes

Antes de continuar vamos a definir que es un ***agente***. Un agente es un "ente" capaz de "percibir" estímulos y "actuar" en consecuencia. En computación, este "ente" puede ser un sistema electrónico (el medio son señales), un sistema lógico (el medio son los lenguajes de programación) o un experto humano (el medio es el conocimiento en estado puro). Como vemos, los tres agentes propuestos se diferencian por el nivel al que actúan. Los dos primeros actúan respectivamente en el nivel físico y simbólico; sus conjuntos de entradas, su forma de actuar y su conjunto de salidas, son totalmente distintas estando, además, perfecta y claramente definidas. Estos dos agentes son causales por definición.

Por otro lado, tenemos al experto humano, al que denominaremos **agente inteligente**. No obstante, vamos a definir este en términos abstractos diciendo que va a tener una serie de sensores que perciben estímulos del entorno, va a tener un conocimiento previo del entorno en el que se mueve, el cual está estructurado en algún tipo de representación simbólica desconocida para nosotros (pueden ser hechos, creencias, intenciones…). También tiene unos objetivos o metas que cumplir. Con arreglo a este conocimiento, a los estímulos exteriores que percibe del entorno y a sus objetivos, actúa en consecuencia,

afectando al entorno utilizando sus actuadores[3]. Después de cumplir estos objetivos el agente inteligente aprende del resultado obtenido, añadiendo este nuevo hecho (el objetivo) a su estructura de conocimiento. Así, el agente inteligente es capaz de *predecir,* con arreglo a su conocimiento, el comportamiento que va a tener si se encuentra en un entorno similar a uno que ya haya aprendido, es decir, son entornos distintos pero de semántica equivalente.

La predicción se ha producido como consecuencia de un razonamiento, que puede ser de tipo:

- **Deductivo**.- Se parte de un conjunto de premisas o hipótesis, y todas ellas se cumplen, se saca una conclusión. Se va de lo general a lo específico

- **Inductivo**.- Dado un caso particular, podemos generalizarlo para un caso general. En este razonamiento siempre va a tener una validez con un grado de incertidumbre. Se va de lo específico a lo general

- **Abductivo**.- Dado una conclusión se trata de deducir las premisas o hipótesis que han dado lugar a ella.

El agente inteligente puede utilizar uno solo de estos **métodos** de razonamiento**, que en I.A.** normalmente **llamaremos de inferencia,** o una mezcla de ellos**.**

Un ejemplo: Un niño que estando en su casa mete el dedo en un enchufe por primera vez en su vida. Aunque muchas veces sus padres le advirtieran sobre el peligro que ello supone, finalmente la curiosidad puede con él y lo hace. Aunque se queda en un susto, le da un buen calambrazo: acaba de aprender, por deducción, que meter el dedo en el enchufe es muy desagradable. Cuando este niño va a jugar a casa de un amigo e identifica un enchufe, predice, inductivamente, que si mete el dedo en un agujero de ese enchufe se produce un efecto desagradable. El entorno es distinto aunque la semántica de la situación es la misma: es otra casa y otro enchufe (más grande, de otro color, con luz de neón…).

1.2.2. Nivel de conocimiento de Newell

¿ En base a qué actúa un agente inteligente? En base a lo que Newell llamó el **principio de racionalidad (**también llamado de causalidad semántica**),** cuya traducción es**:**

[3] También llamados efectores.

"*Si un agente tiene el conocimiento de que una de sus acciones conducirá a una de sus metas, entonces el agente seleccionará esa acción*" [4]

Según este principio, se toman las decisiones más óptimas de forma instantánea y automática, en base a TODA la información del entorno disponible en ese momento. Esto significa que el agente puede tomar decisiones erróneas si el conocimiento que tiene del entorno es insuficiente.

¿Cómo describimos el conocimiento? Según la teoría de niveles antes descrita, hemos de describir el medio de este nivel con unos espacios de representación, que por definición son finitos, es decir, tiene una sintaxis formal. El conocimiento del mundo no se puede modelar completamente con estructuras finitas, luego tenemos un problema, ya que esto nos está diciendo que solo podremos modelar parcelas del conocimiento. Otro problema, derivado del anterior, es que, si no podemos describir el conocimiento con estructuras finitas, ¿cómo podemos describir, entonces, las percepciones del agente?

Esto es consecuencia de que los problemas se expresan en lenguaje natural, el cual no es estricto en cuanto a estructuras, y normalmente incompleto en cuanto a lo que describen; se complementan con la semántica que sólo el agente inteligente es capaz de interpretar. Un ejemplo muy parecido al anterior sobre ejecución del lenguaje para Chomsky: "Hoy voy a comer con UNIX"; la frase está bien formada sintácticamente, pero semánticamente ¿quién come con UNIX?, ¿es un sistema operativo? ¿O es una persona con el mote UNIX? El contexto solo lo identifica el agente inteligente en su entorno.

Estas dos cuestiones que nos hemos planteado (encontrar una sintaxis que describa el conocimiento y un medio racional para identificar el contexto) son las que trata de solucionar la I.A. y que, además, son las que llevan a Newell a definir el nivel de conocimiento: No existe un mecanismo claro para reducir el conocimiento de manera que se pueda representar mediante estructuras finitas (sintaxis formal) y tampoco hay un mecanismo claro de cómo computar el comportamiento no analítico del experto, que es también conocimiento (razonamiento en base a la semántica). El nivel de conocimiento de Newell es equivalente a la competencia lingüística definida por Chomsky en su jerarquía de

[4] Traducción de la definición en [Newell 1980].

lenguajes, es decir, describe como se debe pasar de una estructura ambigua a una sintaxis formal.

En la definición dada por Newell para sus niveles, estos los describe en base a cinco aspectos:

Nivel / Aspecto	Nivel físico	Nivel simbólico	Nivel de Conocimiento
Sistema	Sistemas digitales y Arquitectura de Computadores	Agente programador	Agente inteligente
Medio	Bits y vectores de bits	Símbolos y expresiones	Conocimiento
Componentes	Puertas, registros, UAL,s	Operadores y ubicaciones en memoria (primitivas del lenguaje)	Metas u objetivos Creencias Intenciones (semántico)
Composición	Algebra de Boole	Asignación y asociación	Estructura desconocida
Comportamiento	Autómatas finitos	Autómatas de pila	Principio de racionalidad

El nivel de conocimiento se caracteriza por ser:

- **Abstracto y genérico**.- Lo que se intenta es encontrar **tareas genéricas** que sirvan para *manipular* la representación del conocimiento, llegando a ser estas tan simples que permitan un proceso de reducción al nivel simbólico. Así mismo se intentan encontrar lenguajes de *representación del conocimiento* (aplicación de la lógica matemática, reglas, marcos y redes) y unos *métodos de inferencia* (inducción, abducción, deducción, *resolución y herencia*) que permitan el aprendizaje.

- **Es independiente del dominio**.- Intenta definir una **arquitectura general** y a ser posible **reutilizable**, que modele el conocimiento y que enlace fácilmente con el nivel simbólico.

1.2.3 Dominios de descripción

En cada nivel se van a tener dos sistemas de referencia para representar las magnitudes y su semántica (recordar que el medio hay que definirlo para que sea medible):

- **Dominio Propio (DP) o auto contenido**.- Describe la fenomenología propia del nivel. Es propio de los niveles físico y simbólico, en los que las descripciones son operacionales y los resultados son causales por definición. No se puede salir de la sintaxis del nivel, dado el carácter formal de este (en el nivel físico gobernado por una función lógica y en el simbólico por una gramática independiente del contexto).

- **Dominio del observador externo (DO)**.- Interpretación de la fenomenología desde el punto de vista del experto, luego sólo existe en el nivel de conocimiento. En este dominio se utiliza el lenguaje natural para definir y dar significado a los procesos del DP.

El DP solo tiene significado en el DO, luego son las tablas de semántica de los niveles físico y simbólico las que relacionan estos dos dominios. Así, para el nivel físico, en el DP son ceros y unos, y en el DO se le asocia un valor, por ejemplo numérico, que es el que tiene sentido para el observador externo. Para el nivel simbólico, en el DP son símbolos cuya sintaxis está definida por una gramática independiente del contexto y en el DO se trata de una secuencia de operaciones que manipulan una estructura de datos, que para el observador externo tienen un significado dentro del dominio.

Anteriormente se ha dicho que hay que tener cuidado en mezclar aspectos de distintos niveles. Pues también deberemos tener cuidado con mezclar descripciones de los dos dominios dentro del mismo nivel. Si esto se produce, el modelo probablemente será erróneo o cuando menos ambiguo.

¿Y el DP en el nivel de conocimiento? Se considera que la descripción de este dominio en este nivel no existe (aun cuando en la tabla que dibujamos a continuación así parezca), ya que es el mismo observador externo el que modela el conocimiento en lenguaje natural: realiza la descripción detallada de la arquitectura general definida en el nivel de conocimiento de Newell, estructurada en tareas genéricas, lenguajes de representación del

conocimiento y métodos de inferencia. Además esta arquitectura general también determina las tablas de semántica en este nivel.

En su teoría del nivel de conocimiento, Newell sólo definió la representación del conocimiento y no una teoría sobre como representarlo. Para esto se están haciendo esfuerzos en un nivel intermedio entre el de conocimiento y el simbólico desarrollándose:

- Teoría de agentes cooperativos (descomposición, segmentación y especialización) para la arquitectura global.

- Metodología KADS (o KADS-I, ya que ha evolucionado hacia commonKADS, o KADS-II, que se generaliza para la administración y análisis del conocimiento) con su estructura de tareas genéricas predefinidas (metodología utilizada para creación de **sistemas basados en el conocimiento, SBC, también llamados sistemas expertos**[5]).

Los niveles de computación junto con los dominios de descripción del modelo computable, hacen que podamos sintetizarlos/analizarlos, tal como lo haríamos en cualquier otra ingeniería. Esto se muestra en el siguiente recuadro.

En el proceso de análisis, partimos de la implementación del hardware para obtener los circuitos lógicos y las funciones que en él se dan. Con el DO del nivel físico (la semántica), junto con el DP del nivel simbólico (el programa), emergen las estructuras de datos que se manipulan con él (DO del nivel simbólico), y con estas y la tabla de semántica del DP del nivel de conocimiento termina por emerger qué hace el modelo. En el DO siempre habrá más conocimiento que el que se pueda deducir del DP.

En el proceso de síntesis, partimos del problema, capturamos los procesos no analíticos del experto humano junto con el conocimiento del dominio, y lo reducimos al nivel simbólico, implementamos el programa, el cual traduciremos a una secuencia de fórmulas lógicas que finalmente implementaremos en el hardware.

[5] Sistemas Expertos, SE, también llamados "sistemas basados en el conocimiento", SBC, es una aplicación informática que implementa un modelo de conocimiento circunscrito a un campo específico del conocimiento humano (por ejemplo, medicina), funcionando como un agente inteligente.

DOMINIO / NIVEL	DO	DP
Nivel de conocimiento	Caracterizado por las metas, creencias e intenciones y gobernado por el principio de racionalidad.	Tareas genéricas, teorías sobre la representación del conocimiento (Lógica formal, marcos, reglas y redes) e inferencia (inducción, deducción, abducción, resolución y herencia), para su reescritura computable y construcción de las tablas de semántica.
Nivel simbólico	Caracterizado por un lenguaje formal y estructuras de datos. Semántica del DP	Implementación del programa en el lenguaje formal
Nivel físico	Diseño de los circuitos lógicos, cuyo comportamiento está gobernado por el álgebra de Boole. Semántica del DP	Implementación del hardware

1.2.4. SBC, Un agente ¿inteligente?

Según un estudio realizado en 1986 por Dietterich basándose en la teoría de niveles, muchos de los sistemas que se llamaban de aprendizaje se quedaban en optimizadores de prestaciones, pero no aprendiendo nada. Llegó a la conclusión de que el comportamiento de un programa de aprendizaje no se puede describir ni predecir al nivel de conocimiento, pues realizan saltos inductivos no justificados.

Recordemos que un agente inteligente es aquel que percibe unos estímulos y actúa en base a ellos y al conocimiento que tiene para conseguir los objetivos (metas, creencias, intenciones). Lo importante del agente inteligente es que ha de ser capaz de reaccionar en situaciones nuevas para él, tomando sus propias decisiones (realmente no se pueden predecir de manera formal el comportamiento). Como ya se dijo, el resultado de las acciones se inyecta al agente inteligente como conocimiento adicional.

1.3.- HIPÓTESIS FUERTE DE INTELIGENCIA ARTIFICIAL

Como ya se expuso, en los enlaces de reducción entre niveles, se produce una pérdida de conocimiento que se almacena en su tabla de semántica, y en la emergencia hay que inyectar conocimiento para reconstruir el medio. La hipótesis fuerte de la IA es que, a pesar de las pérdidas de semántica en el enlace de reducción del nivel de conocimiento al simbólico y luego al físico, todavía es posible hacer computable la inteligencia humana.

1.4.- MODELADO DEL CONOCIMIENTO

Con lo dicho hasta ahora, podemos decir que tenemos dos tipos de conocimiento para modelar:

- Procesos no analíticos del experto humano, que definen la forma de razonar y actuar (sus métodos de inferencia); es el **conocimiento estratégico y es independiente del dominio**.

- Conocimiento del dominio, que define el **dominio propio** en el que vamos a sintetizar el SBC (identificación del vocabulario específico del dominio y del contenido de las estructuras de datos).

Para abordar el modelado del conocimiento, el DO se puede estructurar en tres capas:

- Una capa interna, que estará compuesta por una estructura de **Tareas Genéricas** que, interconectadas, forman el conocimiento estratégico (por ejemplo: clasificación, diagnóstico, planificación…).

- Una capa intermedia, formada por las **herramientas** (formalismos de representación del conocimiento: lógica, reglas, marcos, redes o híbridos de los anteriores) y los **métodos** (Inducción, deducción, abducción, resolución y herencia) **que utilizarán las tareas genéricas**. Para una misma tarea genérica puede haber varias soluciones con distintas herramientas y métodos.

- Una capa externa, que es la descripción global de la solución al problema planteado, donde se incluye la organización de la adquisición del conocimiento y del modelo, la segmentación en tareas genéricas y sus estrategias de control.

Un primer paso en el modelado es el **análisis del problema de forma estratégica**, identificando los objetivos, las funciones necesarias y como se estructuran (jerarquía de objetos). Este conocimiento estratégico es el conjunto de procesos no analíticos del experto humano. La forma más típica de este análisis es la entrevista con el experto, aunque debido a que puede ser incompleta y ambigua, se han desarrollado otros métodos basados en la psicología que garanticen estas dos características. El ingeniero de conocimiento ha de conocer el domino, pero sin llegar a ser un experto en él. Al principio de los SBC,s, lo que se pretendía era introducir todo el conocimiento del dominio en el sistema para que obtuviese conclusiones (eran estáticos a nivel de conocimiento, además de tener que convertir en experto al ingeniero, lo que es un tiempo prohibitivo en el desarrollo del sistema). Por eso lo que se pretende es modelar "como" piensa el experto, más que en "qué" piensa.

Con los objetivos obtenidos de este análisis, se modela el conocimiento en una secuencia limitada de bloques funcionales de alto nivel que llamamos **tareas genéricas.** Estas todavía son demasiado complejas para reducirlas al nivel simbólico, por que se determinan también unos **métodos** de resolución de problemas que desarrolla las anteriores, es decir, los métodos son las subtareas en las que se dividirán las Tareas Genéricas para llegar a su objetivo. Estos métodos serán de **naturaleza heurística**[6]**.** Estas subtareas, a su vez, podrán desglosarse en otras subtareas de manera recursiva hasta que se obtengan unos métodos elementales ya reductibles directamente al nivel simbólico. Al ser conocimiento estratégico, las tareas y los métodos son independientes del dominio, por lo que se podrán reutilizar en otros SBC,s. Las tareas genéricas más importantes son la clasificación y el diagnóstico.

El método estará íntimamente ligado a las herramientas de representación que se utilicen. Estas, que también son genéricas, son las que van a enlazar con el conocimiento

[6] Etimológicamente, heurística es el estudio del descubrimiento y la invención debido a la reflexión y no al azar. En I.A. la usaremos en el sentido de estrategias que conducen a un descubrimiento razonado de la solución -dirigen las inferencias con la información disponible, la cual normalmente será incompleta.

del dominio. Este último puede hacer que nos decantemos por una u otra herramienta (incluso híbridas), y determinará a su vez los métodos a utilizar.

Una vez que tenemos esta estructura de tareas genéricas y métodos, decimos que tenemos un **modelo del conocimiento**, y lo que tenemos que hacer ahora es codificarlo a nivel simbólico.

1.4.1. Metodología KADS-I

Acrónimo de Knowledge Analysis and Design Support, fue un proyecto inicial en el que se diseñó una metodología para la adquisición de conocimiento con herramientas similares a las de la teoría de sistemas (ciclo de vida de desarrollo, estructuración en módulos funcionales….). KADS-I no era lo suficientemente precisa y eficiente; es en el proyecto KADS-II donde se alcanza un grado de formalización comercialmente viable, denominándose actualmente commonKADS. Esta metodología es un estándar de facto en el desarrollo de sistemas basados en el conocimiento. Además sentó las bases para UML[7], utilizado en ingeniería del software y que tiene una sintaxis muy parecida.

Veremos las bases de KADS introduciendo KADS-I. Esta metodología se basa en la estructura de tareas genéricas antes descrita, para la que dispone de una biblioteca de estas ya predefinida, así como los métodos de inferencia que utilizan. El ciclo temporal de desarrollo consta de fases alternativas de análisis y síntesis.

El análisis comienza con la identificación del problema global, su descomposición en bloques funcionales y la descripción de como va a interaccionar con el entorno. Todo ellos se documenta. Este análisis derivará en una síntesis global del sistema en varios subsistemas cooperantes (módulo de conocimiento, interfaz y resto de aplicación-base de datos,…-).

Otra parte importante del análisis es la adquisición del conocimiento del experto, tal como la hemos descrito anteriormente. Para ello esta metodología estructura este conocimiento en cuatro capas estructuradas jerárquicamente:

[7] Unified Modeling Language.- Lenguaje de modelado uniforme, orientado a objetos.

- **Conocimiento del dominio.-** Esta es la capa más profunda en la que se adquiere el conocimiento estático: entidades del dominio y sus relaciones y se determinan las herramientas de representación (lógica, reglas, marcos, redes).

- **Inferencia**.- Son los métodos que utilizaremos (inducción, deducción, abducción, resolución y herencia) y que estarán relacionados íntimamente con la o las herramientas de representación elegidas (aunque no con su contenido).

- **Estructura de tareas genéricas**.- Seleccionadas o construidas a partir de los métodos utilizados.

- **Estrategia de modelado.-** En la que se determina la estructura de tareas genéricas y si hay que cambiar alguna.

La suma de estas cuatro capas es lo que forma el modelo de conocimiento. Una vez obtenido este, hay que compararlo con la biblioteca de tareas genéricas predefinida para escoger la que más se parezca, modificando, si es necesario, la estructura obtenida en el análisis. Así se encontrará la estructura de tareas genéricas más adecuada al problema que se está resolviendo. Una vez que se ha obtenido la estructura del modelo de conocimiento, se procede a sus síntesis.

1.5.- ALTERNATIVAS: IA SIMBÓLICA E IA CONEXIONISTA

Actualmente, y debido a las limitaciones de ambas, estas dos alternativas cooperan en la resolución de un problema (antes estaban enfrentados los que defendían una u otra alternativa), dependiendo de la naturaleza de este. Tenemos:

- **I.A. Simbólica.-** Se caracteriza por pasar del nivel de conocimiento al físico, programando los procesadores, es decir, pasando por el nivel simbólico.

- **I.A. Conexionista.-** Se caracteriza por pasar del nivel de conocimiento al físico directamente, sustituyendo la programación por el aprendizaje de su "red neuronal" (siempre se va a estructurar en una red multicapa de procesadores elementales, donde parte de la computación es intrínseca a la tipología de la misma). El

autoaprendizaje se produce por el ajuste en el valor de un conjunto de parámetros o por asociación de estímulos. Para poder aplicar esta alternativa el problema debe ser segmentable y modular (paralelismo).

Comparando las dos alternativas:

- En la conexionista, para reducir del nivel de conocimiento directamente al nivel físico, el análisis debe ser más pormenorizado. En la simbólica se facilita enormemente la labor de análisis, ya que tiene, como herramienta, los lenguajes de alto nivel.

- La conexionista está limitada a problemas que admitan esta alternativa (segmentable y modular). La simbólica ofrece varias tareas genéricas y métodos de aplicación, mientras que la conexionista va a ser siempre una red multicapa, lo que limita el tipo de problemas a resolver.

- En la conexionista, la bajada de dos niveles de manera directa es problemática pues la red multicapa es de procesadores muy elementales, luego la función de aprendizaje no puede ser muy complicada. Por ello se necesita tener la posibilidad de obtener un conjunto de entrenamiento.

- La conexionista es adecuado para un medio cambiante y poco conocido.

1.6.- UNA TAREA GENÉRICA: CLASIFICACION

Sea cual sea la alternativa que se elija para solucionar el problema, vamos a tener una tarea genérica común: la clasificación (ya se dijo antes que era una de las más importantes).

La tarea de clasificación agrupa configuraciones de salida en clases y lo que hace es asociar configuraciones del espacio de entradas a estas clases de salida. Ambos espacios precisan una definición previa en extenso. Según el dominio podríamos tener:

- Entradas
    - Datos.

  - Vectores de características.
  - Hipótesis iniciales.

- Salidas
  - Diagnósticos.
  - Fallos del sistema.
  - Conceptos asociados.

El método que utiliza es distinto según el tipo de clasificación que se realice:

- Clasificación jerárquica.- Establecer y refinar (redes semánticas).

- Clasificación heurística.- Comparación abstracta.

- Clasificación conexionista.- función de aproximación de coste.

1.6.1. Clasificación heurística

En 1985 Clancey encontró similitudes en todos los sistemas basados en reglas (herramienta para la representación de datos), y las aplicó en MYCIN (diagnóstico médico), denominándolo clasificación heurística. Se divide en tres subtareas:

- **Abstracción de los datos**.- El valor real de una variable pertenecerá implícitamente a una categoría abstracta de datos (clase de datos de entrada). Por ejemplo, las temperaturas corporales superiores a 37ºC son de la categoría "fiebre").

- **Equiparación heurística.**- Busca una asociación entre la categoría abstracta de entrada con clases abstractas de soluciones (por ejemplo, la fiebre se puede asociar con la clase de patologías que son "infección").

- **Refinamiento.**- Especialización de las clases obtenidas para llegar a la solución.

Clancey analizó varios métodos de inferencia de varios sistemas expertos y mostró que la clasificación heurística se podía aplicar a todos ellos (mostró que era independiente del dominio).

1.6.2. Clasificación conexionista

La tarea a solucionar en el enfoque conexionista se puede plantear en términos de un clasificador multicapa, donde cada capa realiza una inferencia. Se descompone en 4 subtareas:

- **Extracción de propiedades.-** Se preparan las señales del espacio de entradas donde cada variable será una línea etiquetada con su tabla de semántica (la red maneja números, dejando la semántica siempre en el dominio del observador). Se puede hacer por métodos analíticos (filtros), algorítmicos (mínimos, máximos o histogramas) y simbólicos (condicionales).
  La salida de cada procesador elemental de la última capa es una línea etiquetada, asociada a una tabla de semántica, y cada una representa una clase de resultados.
  Las entradas y salidas así obtenidas son estáticas. Otra parte de las entradas y salidas son programables por autoaprendizaje.

- **Métricas.-** Determinar la distancia de las salidas a los valores representativos de las clases.

- **Selección de máximos.-** Una capa decide sobre la clase a la que "creemos" que pertenece la configuración. Esto se puede realizar de varias formas:

  - Selección del valor máximo: procedimientos competitivos.

  - Procesos cooperativos.

  - Interpretación borrosa o probabilística del resultado final.

- **Aprendizaje por realimentación.**

--

CAPÍTULO 2.- MÉTODOS DE RESOLUCIÓN: LA BÚSQUEDA

2.1. TÉCNICAS DE REPRESENTACIÓN DEL CONOCIMIENTO

Dependiendo del problema que se nos plantee, se ha de elegir una forma de representar el conocimiento del dominio; así tenemos dos opciones:

- **Técnicas declarativas**.- Se describen los aspectos conocidos del problema. Se especifica la información pero sin decir cómo usarla. Describen el conocimiento del dominio como tal. Se caracterizan por:
    - Claridad y uso modular.- Permiten añadir nuevos hechos, los cuales se almacenan una sola vez.
    - Conllevan un tratamiento heurístico.
- **Técnicas procedimentales.-** Describe el proceso a realizar para encontrar la solución. Declaran como se manipulan las entidades. Se caracterizan por:
    - Son más eficientes que las anteriores.
    - Conllevan un tratamiento algorítmico. Por ello son más fáciles de mantener.
    - Se utilizan para guiar las *líneas de razonamiento* (que la evolución del razonamiento sean coherente).

Ningún sistema experto es completamente declarativo o procedimental (salvo que el problema que soluciona sea muy sencillo), ya que la especificación del conocimiento

(declarativo) necesita de algoritmos para su tratamiento (procedimental). No obstante, el uso de unas u otras técnicas determinan como se representa el conocimiento.

Estas dos técnicas son intercambiables, siempre y cuando, para las declarativas, haya un procedimiento de interpretación algorítmico.

Ya se dijo que los métodos estarán ligados a las herramientas de representación. Para problemas cuyo conocimiento se representa por técnicas procedimentales, los métodos son de **resolución** y se llevan a cabo mediante **tareas de búsqueda**, que son de naturaleza algorítmica (procedimental): son ***subtareas genéricas*** **para la resolución de problemas.** Los métodos clásicos de resolución, que no son exclusivos entre sí, sino que, para problemas complejos, se pueden utilizar conjuntamente, son:

- Generar – Probar
- Medios – Fines
    - STRIPPS
- Reducción del problema

Estos métodos se describirán después de formalizar los problemas en términos de tareas de búsqueda.

2.2.- PROBLEMAS EN LOS QUE SE APLICAN TÉCNICAS DE IA

Ante un problema se nos pueden plantear dos situaciones:

- Que se tenga el conocimiento sobre lo que hay que hacer.
- Que haya que indagar como llegar a una solución.

Desde el punto de vista del nivel simbólico (Dominio Propio), y para los **problemas para los que existe algún algoritmo** que los soluciona, sus implementaciones se pueden clasificar según el coste computacional que vaya a tener. Se pueden presentar dos tipos:

- **Problemas tipo P**.- Son aquellos en los que el coste tiene una complejidad polinómica (además, en problemas reales, el exponente no puede ser muy grande).

- **Problemas tipo NP**.- Son aquellos en los que los algoritmos que los solucionan tienen una complejidad exponencial. Esto nos lleva a que, con tamaños muy pequeños del problema se consuman todos los recursos disponibles.

Desde el punto de vista del nivel de conocimiento (Dominio del Observador), los problemas pueden ser:

- Problemas P.- Se conoce el algoritmo y es computable. Se implementa mediante las técnicas normales de estructura de datos y algoritmos.

- Problemas NP.- Se conoce el algoritmo que lo soluciona, pero es de complejidad intratable.

- Problemas con solución parcialmente conocida.- En el campo del conocimiento humano, la incógnita está en como se formaliza el razonamiento para llegar a alguna solución, ya sea conocida totalmente o con un grado de incertidumbre.

La I.A. estudia precisamente como pasar del nivel de conocimiento al simbólico y conseguir un algoritmo computable para alguno de estas dos últimas categorías. Esto se hace con herramientas y métodos genéricos, que al ser independientes del dominio son:

- **Aplicables a muchas clases de problemas**. Cada uno de ellos se pueden caracterizar encontrando unas "entidades" y "procedimientos de manipulación" comunes. Entonces, es cuando podemos crear procedimientos de resolución genéricos, independientes del dominio del problema.

- **Son débiles**, pues no utilizan la información relevante del domino para su resolución (pueden ser problemas de una misma clase, pero de dominios totalmente dispares).

- Su principal desventaja es que **son poco eficientes**, debido, precisamente, a su carácter genérico.

2.3.- PLANTEAMIENTO DEL PROBLEMA

De manera general, para plantear el problema, lo primero que se hace es fijar una *meta*, y en base a esta formular el problema. Para ello se parte de una descripción en lenguaje natural y a partir de esta se *abstraen* los elementos y acciones más relevantes (se eliminan los detalles superfluos). De esta manera definimos el problema en términos representables por "*entidades*", que el sistema reconoce y puede *"manipular"*. Para que un sistema reconozca estas "entidades", deberemos poder definirlas de manera formal y estructurada, mediante un *lenguaje de representación.* El proceso de definición de entidades (abstracción) es muy importante, ya que depende de la habilidad que tenga el diseñador del sistema (es poco sistemático y muy subjetivo). El problema, por regla general, se representará con un grafo donde aparecerán todas las relaciones entre unas entidades y otras (también hay otras formas de mostrar estas relaciones).

Uno de los procedimientos para "manipular" estas "entidades" es la **búsqueda,** la cual será **ejecutada por un agente** (el sistema, el programa) que es el que tiene unas metas a alcanzar. A este agente se le denomina **solucionador.**

Para buscar las metas se necesita un conjunto de *acciones*, y estas ser aplicadas en el orden fijado por algún *mecanismo de selección que guía la búsqueda*. A este mecanismo se le llama **estrategia de control o estrategia de búsqueda**. La *solución* obtenida (puede no haber o ser más de una), si la hay, se expondrá como una secuencia de acciones determinada; habrá casos en los que solo se necesita el valor final de la resolución, pero por lo general, en IA, se quieren saber los pasos seguidos para llegar a una solución válida.

La debilidad y eficiencia de los métodos genéricos se puede mejorar, como se verá más tarde, dirigiendo la búsqueda mediante el conocimiento heurístico8 aplicado a la estrategia de control, siendo este de dos tipos:

1.- Dependiente del dominio. Para determinar la proximidad a la meta.

2.- Dependiente de cómo funciona internamente la tarea de búsqueda.

El proceso de búsqueda de la solución se representará con un grafo dirigido y acíclico (GDA)[9]. Los algoritmos de búsqueda se van a ver desde el punto de vista del nivel de conocimiento; la implementación detallada y eficiente de estos (nivel simbólico) se deja para las disciplinas de programación de estructura de datos y algoritmos.

2.4.- ENTIDADES EN LAS TAREAS DE BÚSQUEDA

Se ha dicho que, en el proceso de abstracción del problema, hemos de identificar correctamente las "entidades". Para hacerlo hemos de tener en cuenta el enfoque que necesitamos; en este caso enfocados a la búsqueda, por tanto en un marco procedimental.

Cualquier tarea de búsqueda responde al esquema de **grafo de exploración** (luego veremos como se representa) de la tabla 1.

2.4.1. ¿Qué incluimos en un estado?

Este es un punto difícil: hay que definir qué incluimos en un estado y qué no. Si abstraemos demasiado la descripción del problema, puede que las entidades extraídas del mismo sean demasiado generales; el resultado obtenido sería inútil porque este no se puede trasladar al nivel simbólico. Y viceversa, si se detalla demasiado, el número de operadores

[8] Ya se dijo que etimológicamente, *heurística* es el estudio del descubrimiento y la invención debido a la reflexión y no al azar. Aquí la usaremos en el sentido de estrategias que conducen a un descubrimiento razonado de la solución (dirigen el proceso de búsqueda con la información disponible, la cual normalmente será incompleta -lo que se hace en estos problemas es simplificarlos, para extraer algún criterio que nos sirva para dirigir la búsqueda-)

[9] Ver anexo A: Repaso de grafos

puede ser muy grande y el coste de llegar a la solución puede hacer no computable la búsqueda de una solución del problema con esos operadores.

CON ESTOS ELEMENTOS	**ENCONTRAR**
• **Conjunto (finito o infinito) de estados.-** Son TODAS las configuraciones posibles (todas las situaciones posibles) dentro del dominio del problema. • **Estados iniciales.-** Uno o varios de los anteriores (un subconjunto finito no vacío del conjunto de estados). A partir de estos, el solucionador empieza la búsqueda. Representan los datos iniciales desde los que parte el problema. • **Estados finales**.- Uno o varios de los primeros (un subconjunto finito no vacío del conjunto de estados), que son aceptados como meta válida. • **Conjunto finito de operadores o reglas**.- Describen las acciones que manipulan los estados. Son una función de transformación de un estado que lo convierte en otro u otros f(nk)={(nk+1)}.	• **Una solución,** es decir, una secuencia de *acciones* desde un estado inicial (datos de entrada) del problema hasta un estado final (una de sus metas). Si se identifican elementos relevantes del problema en el dominio de aplicación, estos nos pueden ayudar a *guiar la búsqueda* de la secuencia de manera más eficiente (conocimiento heurístico).

Tabla 1.- Esquema general de las tareas de búsqueda

2.4.2. ¿Cómo se define un operador?

Hemos dicho que un operador es una función de transformación de un estado que lo convierte en otro u otros. En este esquema, para cada operador se especificarán:

- **Condiciones de aplicabilidad**.- Cada uno de los operadores va a definir lo que se llama *"espacio de aplicabilidad" o "dominio de estados",* que serán, para cada operador, todos los estados a los que se les puede aplicar. Son las condiciones que

debe cumplir cada estado para poder aplicar el operador. Podemos tener varios casos de aplicabilidad:

- Casos en los que se pueden aplicar todos los operadores a todos los estados en cada situación (propiedad de conmutatividad).

- Según la situación, solo pueden ser aplicables unos cuantos.

- Se puede aplicar un mismo operador de diferentes formas (lo que se llama "*instanciación*" de un operador).

- **Función de transformación.**- Es la que determina que cambios se aplican al estado actual. El resultado de la transformación definirá un conjunto de estados resultantes (*rango del operador*).

En cuanto al número de operadores que se han de definir para cada tipo de estado, hay que llegar a un compromiso entre el número de operadores a aplicar (si se eligen pocos, el problema puede ser irresoluble y si se eligen muchos el coste puede ser prohibitivo) y el grado de transformación que estos apliquen (si la transformación es mínima, el coste también puede ser prohibitivo). Cuanto menor transformación haya, será necesario un mayor número de operadores lo que redunda en un mayor coste.

2.5.- REPRESENTACIÓN DE LAS TAREAS DE BÚSQUEDA

Hemos dicho que vamos a representar el proceso de la búsqueda mediante GDA. Esto es porque necesitamos la relación de orden parcial de estos (concepto de más profundo o menos profundo) que no se da en grafos dirigidos con ciclos. Se construirán a medida que avancemos hacia una meta. Antes de continuar[10], vamos a definir como representamos el esquema general con un GDA:

- **Nodo (vértice).-** Uno de los elementos del conjunto de estados.

[10] Para recordar más detalles sobre grafos, se puede consultar el apéndice A

- **Arco (arista).-** En la expansión de un estado n, denota cada uno de los operadores aplicados a n.

- **Expansión de un nodo.-** Dado un estado n, consiste en generar todos los posibles nodos sucesores de este, por aplicación de todos los operadores posibles y en todas sus formas de aplicación (todas sus instanciaciones).

- **Nodo meta.**- Es un nodo terminal u hoja que cumple los objetivos del problema (que constituye una meta del problema).

- **Nodo frontera.**- Nodo que está en espera de ser expandido.

- **Frontera de expansión.**- Conjunto de nodos frontera.

- **Nodo (estado) cerrado.**- Nodo que ya se ha expandido completamente.

- **Nodo (estado) abierto.**- Nodo al que todavía le quedan sucesores que obtener.

- **Coste (computacional) de un arco.**- Es un valor numérico positivo. Se define como el tiempo consumido al aplicar un operador dado a un estado (a un nodo). Si no se dice nada, su coste implícito será 1.

- **Coste (computacional) de un nodo.**- Es un valor numérico positivo. Se define como el tiempo consumido en alcanzar ese nodo desde la raíz, por el mejor camino encontrado.

- **Coste (computacional) real de un nodo.**- Es un valor numérico positivo. Se define como el tiempo consumido en alcanzar este nodo desde la raíz.

- **Coste (computacional) entre dos nodos.-** Es un valor numérico positivo. Se define como la suma de los costes de todos los arcos que van desde un nodo hasta otro. Si los nodos son n_a y n_b, entonces el coste se representa como $C(n_a, n_b)$

- **Factor de ramificación.**- Número "*medio"* de sucesores (operadores) de un nodo (aplicables a un estado). Este valor es la media de operadores de todos los estados expandibles, pues cada uno de estos puede tener distinto espacio de aplicabilidad.

- **Longitud de una trayectoria.**- Número de nodos generados a lo largo de un camino, o lo que es lo mismo, número de operadores aplicados a lo largo de ese camino.

- **Profundidad del grafo.**- Longitud (número de niveles del grafo) desde el nodo inicial hasta el nodo meta por el camino más corto, o lo que es lo mismo, número de operadores (número de arcos) aplicados a la solución desde el estado inicial al estado meta.

  La profundidad del estado inicial (raíz en los árboles) es 0 y la profundidad de cualquier otro nodo será la de su antecesor menos profundo más 1 (podemos tener un GDA en el que, en un nodo dado, puede tener más de un antecesor o padre; entonces su profundidad es la del antecesor menos profundo -el que menos ha costado de calcular- más 1).

- **Recorrido de un grafo.**- Se trata de visitar todos los nodos a partir del nodo raíz. Existen dos tipos de recorrido (se tratan detalladamente en el siguiente capítulo):

  - **En amplitud o anchura.**- Se recorre el grafo en niveles: desde la raíz, se visitan todos los nodos sucesores de este. Una vez terminados todos estos sucesores, se recorren los descendientes de estos últimos, y así sucesivamente, hasta encontrar una solución, si existe.

  - **En profundidad.-** Se visita la raíz, se expande el nodo y se visita uno de sus sucesores. Se expande este último y se visita uno de sus sucesores. Si se llega a un *callejón sin salida*[11], se retrocede un nivel[12] . Si en este nivel no quedan nodos para expandir, se sube otro nivel (así hasta que no queden nodos o se encuentre un nivel con nodos para expandir). Si hay, se expande el nodo de ese nivel para seguir el mismo proceso. De este modo se avanza una vez en cada nivel, hasta encontrar una solución, si esta existe.

[11] Ver la definición en la búsqueda por tentativas del siguiente punto.
[12] Este retroceso (backtracking en inglés) se conoce como cronológico, pues se retrocede al nivel inmediato superior. Existen otros retrocesos que dependen del estado del problema y no solo del anterior estado.

El *grafo de exploración* será el GDA resultante de la aplicación del método. Este grafo contendrá todos los subgrafos resultantes del seguimiento de todos los caminos seguidos, incluidos los infructuosos.

¿Por qué no almacenar un grafo con todas las combinaciones de estados y operadores y trazar un camino? Aunque el número de estados y operadores a aplicar sean finitos, el número de combinaciones posibles de estos será una cantidad muy grande en problemas medianos o grandes(normalmente de orden exponencial), por lo que nos quedaríamos sin recursos (memoria insuficiente). Aunque ya se han descrito por separado, vamos a distinguir entre dos tipos de grafos: un grafo *implícito* que estará formado por todas las posibles combinaciones de estados y operadores (que es la representación de las entidades y sus relaciones), y un grafo *explícito* que será el que se va formando conforme va avanzando la búsqueda, y que es el que hemos llamado grafo de exploración.

2.6.- CLASIFICACIONES DE LAS TAREAS DE BÚSQUEDA

Hemos dicho anteriormente que la búsqueda es una secuencia de acciones en un orden, determinado este por la estrategia de control. Según la información utilizada para avanzar hacia una meta, las tareas de búsqueda se pueden clasificar:

- **Búsqueda ciega o exhaustiva.-** En esta estrategia se generan estados para luego comprobar si estos cumplen con los objetivos para ser meta; si no son meta, se siguen generando otros estados. Al no tener en cuenta el conocimiento del dominio disponible (de ahí el nombre de ciega), no puede dejar ningún nodo de todos los posibles sin examinar (de ahí el nombre de exhaustiva). Es por ello que su complejidad será la del problema, exponencial en la mayor parte de los casos, lo que deriva en que estos procedimientos solo sirvan para problemas pequeños.

- **Búsqueda heurística o informada.-** La estrategia de control utiliza conocimiento del dominio para estimar cual es siguiente mejor estado. Dado que la búsqueda es un proceso dinámico, la estrategia de control utiliza toda la información disponible hasta ese momento. El objetivo de esta dirección informada es que el número de operadores a aplicar a un estado sea bastante menor que en la exhaustiva (menos

caminos inútiles), y por lo tanto mejore apreciablemente la eficiencia promedio del algoritmo (en el caso peor podría ser la misma que la búsqueda ciega). Hay que tener muy claro que las técnicas heurísticas no eliminan estados u operadores, sino que intentan mejorar el coste del camino a una meta.

Otra clasificación que se puede hacer es por la forma en la que se avanza en la búsqueda:

- **Búsqueda de tentativas**.- Se avanza en una dirección y si se llega a un punto en el que se supone que no se llega a alguna meta, se abandona este camino para retomar alguno anterior que también prometía. Esta situación se puede dar:

    - Cuando a un estado no se le pueden aplicar operadores y además no es meta. A esto lo llamamos **callejón sin salida, vía muerta o punto sin retorno**.

    - Cuando las técnicas heurísticas estimen que hay un camino mejor que el que se está siguiendo.

- **Búsqueda irrevocable o sin vuelta atrás**.- Una vez que se ha tomado un camino, este no se puede dejar.

La búsqueda de tentativas se puede dar tanto en exhaustivas como informadas, mientras que la irrevocable solo en las informadas. Si no se dice lo contrario, las tareas de búsqueda que se verán lo serán por tentativas.

Otra clasificación será según la naturaleza del problema. La búsqueda podrá ser:

- **Búsqueda dirigida por los datos o encadenada hacia delante.**- Consiste en seguir algún procedimiento para encontrar alguna meta. El problema se plantea como una situación inicial a partir de la cual se realiza una secuencia de acciones (aplicación de operadores) para llegar a una situación que cumpla ser una meta. También se la llama búsqueda de arriba - abajo.

- **Búsqueda dirigida por las metas o encadenada hacia atrás.**- Consiste en, dada una solución conocida, encontrar el procedimiento para llegar a esa solución. En estas se parte de una meta, a la que se le aplica algún operador que la **transforma en**

una o más submetas de un menor tamaño o dificultad. Cada submeta se puede transformar en una o más submetas de manera recursiva, hasta que estas sean lo suficientemente triviales para ser solucionadas en el nivel simbólico. También se la llama búsqueda de abajo-arriba.

La mayor parte de los problemas de I.A. son planteados en términos de búsquedas dirigidas por las metas.

2.7. CLASIFICACIÓN Y APLICABILIDAD DE LOS OPERADORES

Vamos a clasificar los operadores por su capacidad de **reversibilidad:**

- **Operadores irreversibles**.- Tras su ejecución no se puede volver al estado anterior.

- **Operadores semi-reversibles**.- Tras su ejecución se puede volver al estado anterior, pero si se dispone de recursos para ello. Si no los hay, entonces se consideran operadores irreversibles.

- **Operadores reversibles**.- Se puede obtener el estado anterior sin consumir recursos. Estos a su vez pueden ser reversibles con solo una operación o requerir la ejecución de varias operaciones para llegar al estado padre (complica la reversibilidad).

2.8. ESQUEMAS DE REPRESENTACIÓN

Antes de describir los dos esquemas de representación posibles, vamos a definir unos términos generales:

- **Espacio de representación o espacio del problema.-** Es el entorno dentro del cual se realiza la búsqueda. Este espacio estará formado por:

    - El conjunto de todos los estados.

    - El conjunto de todos los operadores.

El espacio de representación dará una idea de la complejidad del problema (la combinación de ambos puede llegar a ser intratable, computacionalmente hablando).

- **Instancia de un problema.-** Es un conjunto de estados en el que se dice cual es el inicial y uno de las posibles metas (si hay más de una).

De manera general, en todas las tareas de búsqueda vamos a tener una base de datos de trabajo en la que almacenaremos el estado inicial y todos los que se vayan obteniendo sucesivamente a partir del primero. Se mantendrán dos estructuras de datos que llamaremos ABIERTA y CERRADA:

- **ABIERTA**.- Contendrá todos los estados abiertos. Es decir, contiene la *frontera de expansión*.

- **CERRADA**.- Contendrá todos los estados cerrados, y por tanto no se van a volver a visitar (esta no existirá en alguno de los métodos que veremos).

Vamos a tener dos esquemas generales de funcionamiento, distintos según el problema a solucionar:

- **Esquema de producción**.- También llamado de búsqueda en el *espacio de estados*.

- **Esquema de reducción.-** Responde a la posibilidad de que el problema se pueda descomponer en subproblemas más pequeños. También llamado de búsqueda en el *espacio de reducción*.

2.8.1. Esquema de producción

Al espacio del problema se denomina "espacio de estados", y en este planteamiento, la búsqueda consiste en la aplicación de un único operador a un único estado, y se obtiene como resultado un único estado. Este estado es un nuevo *"paso"* en el proceso de búsqueda. Este esquema es propio de las búsquedas guiadas por los datos o encadenado hacia delante. La **equiparación** es el proceso por el cual se comprueba la aplicabilidad de un operador.

La estrategia de control procederá de la siguiente manera:

1.- Selecciona un nodo.

2.- Equiparación: Determinar los espacios de aplicabilidad a los que pertenece (operadores aplicables al nodo seleccionado en el punto anterior).

3.- Conforme a algún criterio, elige uno de los operadores aplicables al nodo seleccionado y lo aplica.

4.- Se determina si el estado generado es una meta. Si no lo es, y tampoco se han consumido todos los recursos, se pueden hacer dos cosas:

4.1.- Intentar aplicar otro operador, o el mismo si tiene varias formas de aplicación, de los que se puedan aplicar al nodo seleccionado (esto se verá que corresponde al recorrido en amplitud).

4.2.- Intentar aplicar el procedimiento al nuevo estado obtenido (corresponderá al recorrido en profundidad).

El **avance** en la búsqueda **será sistemático** para que el algoritmo sea lo más eficiente posible (es decir, los estados y sus operadores se elegirán de manera ordenada). La estrategia de control avanzará en la misma hacia la solución **teniendo en cuenta toda la información obtenida hasta ese momento**, es decir, considerando el proceso de forma global.

El conjunto de estados obtenidos en el proceso de búsqueda descrito, lo representaremos con el "*grafo de exploración o de búsqueda".* Este es un **GDA explícito** donde cada nodo expandido estará enlazado con un arco dirigido a su nodo predecesor y contendrá todos los subgrafos resultantes del seguimiento de todos los caminos seguidos, incluidos los infructuosos. **Implícito en este habrá un árbol**, que contendrá la solución si se llegó a esta (cada estado va a recordar cual fue su antecesor, con lo cual, para dar una solución, podemos recorrer la trayectoria desde la meta hasta el estado inicial).

Este esquema se utiliza en problemas en los que un solo agente intenta encontrar una solución, tales como:

- Enrutamiento de paquetes en redes de propósito general.
- Desplazamiento de robots.
- Búsqueda de la ruta de viaje óptima.
- Diseño de circuitos integrados.

2.8.2. Esquema de reducción

Al espacio del problema se llama "espacio de reducción" y es el propio de las búsquedas dirigidas por las metas o encadenadas hacia atrás. Como ya se dijo, se pueden dar dos situaciones:

- Que al aplicar un operador, se produzca una única submeta, siendo esta más pequeña o simple. En este caso se actúa igual que en el esquema de producción.
- Que al aplicar un operador se produzca un conjunto de submetas, siendo cada una de ellas más sencilla que su predecesora y además tratable independientemente (y por tanto simultáneamente)**.** A este proceso de descomposición se le llama reducción de una meta.

En consecuencia, en este esquema, vamos a tener dos tipos de operadores:

- Los que conocemos hasta ahora, con sus condiciones de aplicabilidad y función de transformación, y que son con los que avanzamos hacia las metas.
- Operadores que se encargan propiamente de descomponer la meta en submetas más sencillas.

Se dijo que con este planteamiento se llegaría a una situación en la cual todas las submetas obtenidas serán lo suficientemente triviales para ser solucionadas en el nivel simbólico. Para dar la solución al problema se pueden dar dos situaciones:

- Que la solución sea la unión de todos esos subproblemas, en cuyo caso **estará resuelto cuando lo así están todos los estados terminales del grafo resultante** (resueltas todas las submetas).

- Que la resolución de los subproblemas triviales, permita la resolución de su subproblema padre y así sucesivamente hasta llegar a la resolución completa del problema (dirigido por las metas). Para poder realizar este cometido los operadores han de ser **reversibles**.

El conjunto de metas y submetas obtenidas en este esquema (proceso de búsqueda y de subdivisión), lo representaremos con "*grafos o árboles Y/O*"[13]. En estos, el estado raíz corresponde a la meta global y la conexión entre nodos se produce mediante **enlaces.** Un enlace va a tener k arcos dirigidos desde un nodo hacia sus sucesores (subproblemas más sencillos) y estos k arcos estarán unidos por una línea curva (ver figura a continuación). Podemos diferenciar dos tipos de enlaces:

- **Enlace O (OR)**.- El enlace tiene un arco dirigido hacia un sucesor, tal como lo hacía el esquema de producción. Este enlace ocurre cuando el operador es de transformación.

- **Enlace Y (AND)**.- El enlace tiene dos o más arcos dirigidos desde el nodo a sus sucesores. Este enlace ocurre cuando el operador es de subdivisión del problema.

Los nodos terminales de este grafo (los que ya no se pueden subdividir más) serán resolubles si se les puede aplicar algún operador de transformación, o irresolubles si no es así. Un nodo estará resuelto cuando lo estén los nodos de alguno de sus enlaces. **El problema estará resuelto cuando lo estén todos los nodos terminales de uno de sus enlaces**, que hace que sus nodos antecesores estén resueltos (incluido en raíz). Para su representación, se dibujará:

[13] Del inglés, "**A**ND/**O**R graphs", también conocidos como grafos A/O.

- Los enlaces tal como se han descrito.

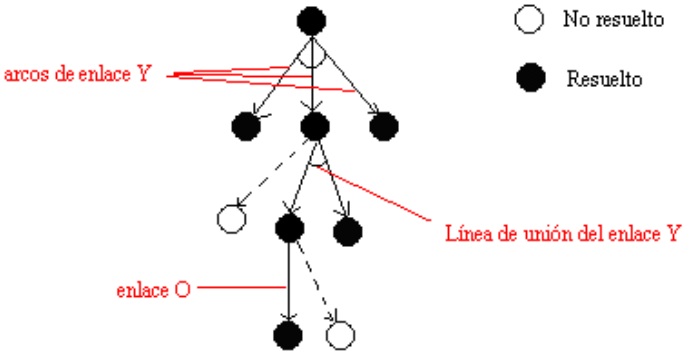
- Los nodos no resueltos se representan mediante circunferencias.

- Los nodos resueltos se representan mediante círculos.

- Tenemos dos situaciones distintas para mostrar la solución:

    - Que el grafo solo tenga enlaces O.- En este caso tenemos el mismo grafo que en el esquema de producción.

    - Que el grafo tenga enlaces Y.- Se mostrará la solución de cada uno de los subproblemas (nodos terminales) derivados del original (la solución es el conjunto de ellos).



En este ejemplo vemos el grafo de exploración final de una búsqueda con este esquema. Implícito en él tenemos un **subgrafo solución**, que son los arcos continuos con los nodos resueltos. Uno nodo está resuelto si lo están todos los nodos de uno de sus enlaces. Este proceso se verá en detalle más adelante.

Este esquema se utiliza para la resolución de problemas en varios campos:

- Para resolver problemas en los que la solución es un conjunto no ordenado de subproblemas resueltos. Un ejemplo es el caso matemático de la integración por partes.

- Para resolver problemas en que dos agentes compiten por un mismo objetivo. Estos problemas pertenecen a la teoría matemática de juegos y se verán mas adelante.

- Problemas de razonamiento lógico y planificación por etapas. Se pasa a cada etapa cuando se cumplan todas las premisas de la etapa anterior.

2.9.- MÉTODOS DE RESOLUCIÓN

Ya se ha comentado que todos los métodos de resolución de problemas van a tener a la búsqueda como subtarea genérica.

2.9.1. Generar – probar

Estos métodos tienen dos módulos: el módulo generador de soluciones posibles y el módulo verificador, que comprueba la validez de estas posibles soluciones generadas.

Una primera aproximación sería generar todas las soluciones posibles, para luego comprobar cuales son las válidas. Sin embargo esta solo sería apropiada para problemas pequeños, ya que para medianos o grandes, el número de estados a almacenar puede ser inmenso (de orden exponencial), por lo que nos quedaríamos si recursos para su almacenamiento. Por ello lo normal es, en este método, alternar fases de generación y de verificación.

Estos métodos utilizan un esquema de producción para las tareas de búsqueda, donde los operadores son conocidos, por tanto la equiparación es trivial. La función de transformación será inmediata, obteniendo los sucesores del estado actual.

Son propios de los problemas en los que se tiene el conocimiento sobre lo que hay que hacer. Un ejemplo es la búsqueda de una contraseña por fuerza bruta (se generan todas las posibles contraseñas hasta encontrar la que es válida).

2.9.2. Medios – fines

Este método se desarrolló a través del GPS (General Problema Solving) Solucionador General de Problemas, inicialmente descrito por Newell, Simon y Shaw (1960).

El objetivo de este método es identificar el **medio** (los operadores) para transformar la descripción de un estado con el objetivo de llegar a algún **fin** (alguna meta), o un estado intermedio (descripción derivada de la anterior) más cercano al fin. El procedimiento para hacer esto es **reduciendo la diferencia** entre la descripción del problema y una meta.

Este método también utiliza el esquema de producción, donde la equiparación es de naturaleza heurística (depende del dominio) y la función de transformación es la que reduce las diferencias con la meta. Así pues, necesitamos dos tablas:

- Tabla de operadores: Va a tener dos columnas bien diferenciadas :
    - Precondiciones.- Que establece el espacio de aplicabilidad de cada operador.
    - Resultados.- Que establece el rango del operador.
- Tabla de diferencias: Las diferencias están ordenadas de mayor a menor importancia, en función del análisis del dominio. Para cada diferencia se establecen los operadores que reducen esta diferencia. Así la equiparación heurística es intrínseca a esta tabla.

Así el proceso consiste en hacer una búsqueda con retroceso (variante del recorrido en profundidad que veremos mas adelante) para resolver las diferencias: determinar los operadores y su espacio de aplicabilidad y después aplicar el operador elegido para obtener una descripción derivada o una descripción meta. Sin embargo, la equiparación NO consiste en comprobar la pertenencia del estado al espacio de aplicabilidad (almacenar el espacio puede ser, de nuevo, prohibitivo), sino en verificar si el estado tiene una parte semejante con el operador, para poder aplicarlo.

Hay que reseñar que, aunque el método es genérico, es muy dependiente del dominio, ya que es la tabla de diferencias la que determina la aplicación de operadores.

2.9.3. STRIPPS

Es una variante del método de medios-fines aplicado en robótica en la que se elimina la tabla de diferencias, estando intrínseca en la tabla de operadores, que ahora tiene dos columnas:

- Adiciones.- Elementos introducidos después de aplicar un operador
- Eliminaciones.- Condiciones que se dejan de cumplir después de aplicar el operador.

Las diferencias ahora son los elementos de la meta y las condiciones de aplicabilidad de los operadores.

2.9.4. Reducción del problema

En estos métodos se parte de una meta difícil de encontrar, que se convierte en una o varias submetas más fáciles. Este procedimiento es recursivo hasta que se encuentran un conjunto de submetas lo suficientemente fáciles para resolver trivialmente.

Estos métodos utilizan un esquema de reducción y son propios de los problemas en los que hay que indagar como llegar a la solución. Un ejemplo de estos métodos es la programación, en la que se parte de una meta y esta se implementa a base de los refinamientos sucesivos de sus funciones y procedimientos.

2.10.- FORMALIZACIÓN DEL PROBLEMA

Como ya se dijo antes, el problema hay que describirlo en términos formales. Se va a partir de la descripción en lenguaje natural y se va a modelar el conocimiento para las técnicas procedimentales, identificándose las entidades con el esquema general de la tabla 1. Se deben tener en cuenta los siguientes elementos:

- Representar adecuadamente los elementos que participan en el proceso de búsqueda.

- Estrategia de búsqueda más eficiente, con la identificación del conocimiento del dominio relevante.

- Limitaciones del sistema: tiempo de ejecución y espacio de almacenamiento.

- Como se eliminan los nodos y subgrafos inútiles (que no aportan información y ocupan espacio de almacenamiento inútilmente).

También se ha dicho que la manera de describir estas entidades, es mediante un lenguaje de representación y un procedimiento para llegar a una posible meta, que es el que controla la estrategia de búsqueda. Para poder llevar a cabo este procedimiento, necesitamos definir dos lenguajes formales, uno para cada uno de los elementos del espacio del problema:

- Lenguaje de descripción de estados.

- Lenguaje de operadores.

2.10.1. Lenguaje de descripción de estados

Se utiliza una gramática formal que define el lenguaje. Recorriendo todas las reglas de reescritura de la gramática se obtienen todos los estados válidos. Tanto las sentencias terminales (expresiones válidas) como no terminales (sustituciones posibles) sirven como descripciones.

2.10.2. Lenguaje de operadores

Cada operador definido por este lenguaje, que también se hará con una gramática formal, va a constar de:

- Un espacio de aplicabilidad o dominio de estados.

- Una regla de reescritura (función de transformación).

- Un conjunto de estados, que son los que se generan por aplicación del citado operador.

También se va a utilizar una gramática formal, donde cada regla de reescritura para un operador va a tener la siguiente estructura:

- Parte Izquierda (PI).- Describe el dominio de estados del operador. Es una expresión general que describe los estados a los que se puede aplicar.

- Parte Derecha (PD).- Describe el *rango* del operador.- Expresión que queda después de aplicar el operador. Puede sustituir al estado o solo una parte del mismo.

- Parte avanzar (PA).- Se asignan los valores del estado a los símbolos de la parte izquierda, que no van a estar en la parte derecha, para aplicar la transformación.

- Parte comentario (PC).- Describe en lenguaje natural (o en el formal del domino de definición) la acción del operador.

Un operador se aplica de la siguiente manera: Se extrae la descripción de la parte izquierda y se comprueba su semejanza con el estado o porción de este (equiparación); si es un buen candidato se transforma en la parte derecha, teniendo en cuenta los valores de la parte avanzar. Finalmente se reescribe el estado resultante con la transformación realizada junto con la parte no transformada (la que no ha cambiado), si la hay.

2.10.3 Equiparación de descripciones

Para hacer la equiparación, hemos definido dos lenguajes de descripción. Al haberlo hecho, implícitamente se está definiendo una relación de orden entre expresiones que pertenecen a ese lenguaje: de más específicas a más generales. Este orden de generalidad nos permite asociar el operador más parecido al estado, teniendo en cuenta que la descripción del operador ha de ser *más general* que la descripción del estado.

2.11.- SOLUCIONADOR

Hemos dicho que el *solucionador* es el agente que va a ejecutar el proceso de búsqueda. Este estará gobernado por la **estrategia de control**, que es la que determina el estado y el operador que le aplicará, generando el o los sucesores correspondientes (guía el proceso de búsqueda del grafo que contiene la solución) utilizando el conocimiento del dominio disponible hasta ese momento *para optimizar el proceso de búsqueda*.

Para evaluar un solucionador que realice una tarea de búsqueda, se tendrán en cuenta las siguientes características:

- Eficacia.- Obtención de la solución:
    - **Completo.**- Lo será si se tiene la certeza de que siempre llegue a la solución del problema, si esta existe.
    - **Óptimo.**- Si de las soluciones que es capaz de encontrar, siempre encuentra la de menor coste.
- Eficiencia.- Coste computacional para llegar a la solución. Ya se ha comentado que depende mucho del factor de ramificación (muchos operadores es sinónimo de mucho coste) y de la profundidad a la que se encuentre la solución:
    - **Complejidad temporal.**- Medida asintótica que se refiere al tiempo necesario para llegar a la meta.
    - **Complejidad espacial.**- Medida asintótica que se refiere al espacio de almacenamiento necesario para llegar a la meta.

De acuerdo con estas características, buscamos un solucionador que sea eficaz y eficiente, es decir, un solucionador que siempre encuentre una solución que, además, sea la de menor coste, consumiendo unos recursos aceptables de tiempo y espacio. En ocasiones se sacrifica la solución de menor coste por la que consuma menos recursos (sobre todo espacio de almacenamiento).

De estas características deducimos que la estrategia de control (que gobierna en solucionador), tendrá en cuenta el coste, y la mejor estrategia de búsqueda será la más eficaz y eficiente para el problema a solucionar. En las tareas exhaustivas la estrategia de control hará una selección no informada (sistemática), mientras que la búsqueda heurística hará una selección informada (selectiva).

Para llevar a cabo su trabajo, los métodos de búsqueda van a tener información de dos clases:

- **Dependiente del funcionamiento interno del solucionador.-** Esta no depende de la información del dominio, sino del grado de transformación de los operadores (**profundidad** del grafo) y del número medio de operadores a aplicar (**factor de ramificación**). Está influenciada por el compromiso entre el número de estados y el número de operadores elegido durante el análisis del problema y también depende de cómo aplica las restricciones (**callejón sin salida**).

- **Dependiente del dominio**.- La que utiliza para estimar lo bueno que es la aplicación de un operador para llegar a la meta.

Vamos a hacer hincapié en un detalle sobre la eficiencia del solucionador. En las técnicas exhaustivas el coste de llegar a una solución será el del problema, normalmente exponencial. El coste de las búsquedas heurísticas en el caso peor realmente es el mismo, ya que se basan en los anteriores, pero el introducir conocimiento del dominio hace que el comportamiento del solucionador tenga, en promedio, una eficiencia mejor que sin información.

Por último, la estrategia de control puede utilizar dos tipos de técnicas heurísticas, y que dependerán de qué herramientas de representación se utilicen:

- **Reglas**.- de naturaleza declarativa (sistemas de representación basados en reglas).

- **Funciones de evaluación heurística**.- Son de naturaleza procedimental (como lo son los métodos de búsqueda que las utilizan).

--

CAPÍTULO 3. BÚSQUEDA SIN INFORMACIÓN DEL DOMINIO

3.1.- INTRODUCCIÓN

La búsqueda sin información del dominio, también llamada ciega o exhaustiva es:

- **Sistemática.-** No deja sin explorar ningún nodo y lo explora sólo una vez.

- **Objetiva.-** Pues no depende del dominio del problema.

Estas técnicas son poco eficientes, aunque, en coste promedio, son mejores que las que utilizan el conocimiento del dominio del problema. Se caracterizan, entre otras cosas, porque la aplicación de los operadores a los estados se realiza de manera sistemática, uno detrás de otro (estrategia de control no informada). Además todas ellas son búsquedas por tentativas y utilizan un esquema de producción (búsqueda en el espacio de estados).

Como criterio para diferenciarlas, se utilizará el orden en que se recorren los estados. Así, para el espacio de estados, tendremos:

- Búsqueda en amplitud

- Búsqueda en profundidad

- Búsqueda con retroceso

- Otras estrategias derivadas de las anteriores

    - Búsqueda en profundidad progresiva

    - Búsqueda bidireccional

Se ha dicho que para representar el proceso de búsqueda se van a utilizar grafos. Para explicar estas técnicas no se va a entrar en temas específicos de implementación. De momento vamos a limitarnos a utilizar árboles de exploración: no vamos a tener en cuenta los casos en los que, cuando se *expande* un estado, se produce un estado que ya estaba anteriormente en el grafo de búsqueda (el caso general lo veremos al final, con el algoritmo general de búsqueda en grafos). Así mismo, para el análisis de coste, se va a considerar que cada operador tiene un coste unitario. Esto, en un problema real, es poco probable ya que unos operadores supondrán más esfuerzo que otros. También vamos a considerar que el número máximo de hijos de un nodo es finito.

Ya dijimos que, en algún momento, podíamos encontrarnos con un *punto sin retorno* (estado que no se puede expandir). Para la determinación de estos habrá que definir unas restricciones que han de cumplir todos los estados. Si en una tarea de búsqueda no informada nos encontramos con uno de estos estados, lo que se hace es que no se añade a la lista de estados para expansionar ABIERTA(veremos estas situaciones cuando hablemos de las tareas de búsqueda informadas).

Un detalle importante en estos métodos es el orden en que se aplican los operadores. Al ser una búsqueda sistemática, este orden está determinado por la forma de implementar el algoritmo: puede ser por orden alfabético o un orden aleatorio; es indiferente, lo importante es que es un orden no informado.

3.2.- BÚSQUEDA EN AMPLITUD

También denominada en *anchura*, es la aplicación del recorrido en anchura visto en el capítulo anterior. Dado un nivel de profundidad N, se revisan todas las trayectorias (se aplican todos los operadores e instanciaciones de los mismos) antes de revisar trayectorias más profundas. Es decir, no se expanden nodos del nivel N+1 hasta que no se hayan expandido todos los estados del nivel N. El proceso termina con error si no hay más nodos que expansionar sin encontrar una meta.

Para almacenar la evolución de la búsqueda, vamos a definir una estructura de datos del TDA[14] cola (estructura FIFO), que llamaremos ABIERTA, donde va a almacenar todos los estados abiertos (frontera de expansión). Cada celda de ABIERTA almacenará el estado y un apuntador a su precedesor[15] (padre).

Según vamos aplicando el algoritmo, vamos dibujando el grafo de exploración. Dependiendo de la solución requerida tendremos dos variantes: Que la solución sea la meta, cuyo caso la solución es inmediata o que sea la secuencia de operaciones hasta una meta, en cuyo caso seguimos el camino más corto de la meta a la raíz en el grafo de exploración obtenido (árbol implícito con la solución).

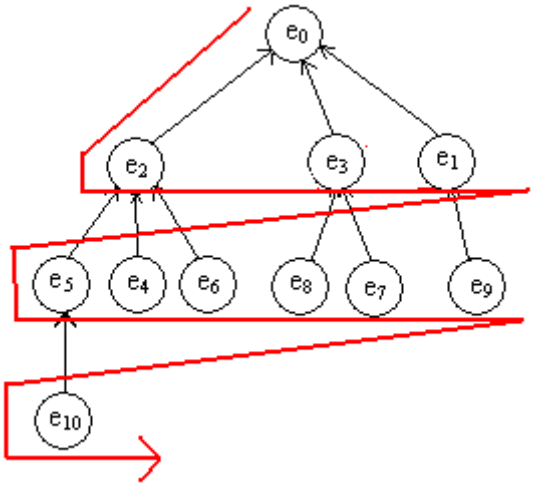
Ejemplo 1:

Supongamos que la descripción del problema es el estado e_0 (raíz del grafo) y se le pueden aplicar dos operadores y uno de ellos con dos instanciaciones, lo que nos da tres estados de nivel 1: e_1 e_2 e_3. A e_1 se puede aplicar un operador, a e_2 tres operadores e_3 dos operadores, que nos da seis estados de nivel 2: e_4, e_5, e_6, e_7, e_8, e_9. En nivel 3, a e_5 se puede aplicar 1 operador**….**

En el grafo explícito (de exploración) de la secuencia de acciones en la tarea de búsqueda que tenemos a continuación, vemos en rojo el orden en el que se generan los estados a partir del padre (de izquierda a derecha, primero el nivel 0, luego el 1, luego el 2,….. y así sería hasta que se encuentre una solución), y que es el mismo orden de expansión. Podemos notar que los hijos de e_0 y e_3 no se generan correlativos: esto es por el orden no informado de aplicación de los operadores (otro orden también es válido). En el momento de terminar el nivel 2, mostrado en el grafo, el estado de ABIERTA y CERRADA sería:

- ABIERTA : e_5, e_4, e_6, e_8, e_7 e_9
- CERRADA: e_1, e_3, e_2, e_0

[14] Tipo de Datos Abstracto.
[15] Predecesor es sinónimo de antecesor inmediato y antepasado es sinónimo de antecesor



Y el siguiente paso sería expandir e_5:

- ABIERTA : e_4, e_6, e_8, e_7 e_9, e_{10}
- CERRADA: e_5, e_1, e_3, e_2, e_0

--

3.2.1. Algoritmo para búsqueda en amplitud. Nivel de conocimiento

- Crear la lista ABIERTA y poner en su primer elemento el nodo raíz (descripción del problema).
- HASTA que ABIERTA esté vacía (error) o el estado sea META
  - Extraer (eliminar) de ABIERTA el primer nodo de la lista, y asignarlo a 't'(temporal) //expandimos 't'[16]
  - PARA cada operador y cada instanciación de operador aplicable a 't':
    - Aplicar el operador a 't'. Obtendremos un nuevo estado 's' (sucesor). A este le asignamos como padre a 't' (apuntador).
    - SI 's' es meta:
      - Terminar con la expansión de nodos
    - SINO
      - Incluir 's' al final de ABIERTA. **(1)**
  - //fin PARA cada operador
- //fin HASTA que ABIERTA está vacía (error) o el estado es META
- SI es META
  - Devolver la solución **(2)**

[16] "//" indica que es un comentario

- SINO //ABIERTA está vacía = error
    - Mensaje de error. No se ha encontrado una solución posible.

(1) Se podría pensar en este punto en no añadir 's' si este es un callejón sin salida. Pero en una búsqueda ciega ¿Cómo comprobamos que es un punto sin retorno? Este estado se eliminará cuando se vaya a expansionar y no haya operadores para aplicar. Complicarlo para no añadirlo a ABIERTA según se genera, no merece la pena porque el número de estos estados es muy pequeño con respecto al número total de estos, por lo que no afecta a la eficiencia espacial, pero la comprobación adicional sí puede afectarla.

(2) Si la solución es la meta, se devuelve 's' y si es la secuencia de operaciones damos el camino más corto de la meta 's' a la raíz en el grafo de exploración obtenido.

3.2.2. Análisis del algoritmo. Nivel de conocimiento

- **Completo**.- Es completo[17], pues si hay una solución, esta estrategia la encuentra (siempre con los recursos disponibles).

- **Óptimo.-** Hemos dicho que si hay solución, esta estrategia la encuentra; pero además va a encontrar la de menor coste, si consideramos que los operadores tienen el mismo coste unitario, y en todo caso la solución con el menor número de operaciones (la de menor nivel).

- **Complejidad temporal**.- El tiempo de ejecución dependerá del factor de ramificación y de la profundidad. Si el número de operadores es *n* y la solución está en la profundidad *p*, el tiempo necesario será $1+n+n^2+n^3\ldots.+n^p$; el coste temporal está en $O(n^p)$.

- **Complejidad espacial.**- Al terminar la expansión de los estados de un nivel de profundidad (p-1) tendremos almacenados en ABIERTA todos los estados del nivel p. Si el número de operadores es *n* y estamos a profundidad *p*, tendremos

[17] Esto solo es cierto si el número máximo de hijos es finito. Si este fuera infinito, y se diera el caso en el nivel n, no se terminaría nunca este nivel, y por tanto nunca llegaríamos a la solución, salvo que esta se encuentre en ese mismo nivel n.

almacenados n^p estados. El caso peor será cuando la meta esté en el último estado generable de un nivel p. Así el coste espacial está en $O(n^p)$.

3.2.3. Algoritmo para búsqueda en amplitud. Nivel simbólico

Si la solución requerida es el estado meta, el algoritmo descrito anteriormente funciona al nivel simbólico. Pero si la solución buscada es el camino desde la meta hasta la raíz ¿Cómo representamos el grafo explícito en el nivel simbólico? Porque hasta ahora lo hemos dibujado. Para dar una solución (no la mejor) se puede definir otra estructura de datos del TDA pila, que llamaremos CERRADA, donde va a almacenar los estados cerrados. Como en ABIERTA, cada celda de las estructuras de datos almacenará el estado y un apuntador a su predecesor (padre). Si se encuentra la meta, la solución estará en CERRADA, en el árbol implícito que resulta de seguir los punteros desde la meta hasta la raíz.

- Crear la lista ABIERTA y poner en su primer elemento el nodo raíz (descripción del problema).
- Crear la lista CERRADA, que inicialmente estará vacía. **(1)**
- HASTA que ABIERTA esté vacía o el estado sea META
    - Extraer (eliminar) de ABIERTA el primer nodo de la lista, y asignarlo a 't'(temporal), y a su vez lo copiaremos en la siguiente posición de CERRADA.
    - //expandimos 't'
    - PARA cada operador y cada instanciación de operador aplicable a 't':
        - Aplicar el operador a 't'. Obtendremos un nuevo estado 's' (sucesor). A este le asignamos como padre a 't' (apuntador).
        - SI 's' es meta:
            - Asignar 's' a CERRADA y terminar con la expansión de nodos
        - SINO
            - Incluir 's' al final de ABIERTA. **(2)**
    - //fin PARA cada operador
- //fin HASTA que ABIERTA está vacía o el estado es META
- SI es META
    - La solución estará en CERRADA. Se visita la pila desde el estado meta hasta el raíz en el orden dado por el campo de puntero al estado padre, siendo la profundidad del árbol solución el número de estados en la secuencia y el coste la suma de las aristas del árbol.
- SINO
    - Mensaje de error. No se ha encontrado una solución posible.

(1) Si solo se pide la meta como solución, no se necesitaría la estructura CERRADA.

(2) Se podría pensar en este punto en no añadir 's' si este es un callejón sin salida. Pero en una búsqueda ciega ¿Cómo comprobamos que es un punto sin retorno? Este estado se eliminará cuando en su expansión no haya operadores para aplicar.

3.2.4. Análisis del algoritmo. Nivel simbólico

- **Completo**.- Es completo[18], pues si hay una solución, esta estrategia la encuentra (siempre con los recursos disponibles).

- **Óptimo.-** Hemos dicho que si hay solución, esta estrategia la encuentra; pero además va a encontrar la de menor coste, si consideramos que los operadores tienen el mismo coste unitario, y en todo caso la solución con el menor número de operaciones (la de menor nivel). Veremos formalmente esta propiedad cuando veamos, en el siguiente capítulo, el algoritmo heurístico A*.

- **Complejidad temporal**.- El tiempo de ejecución dependerá del factor de ramificación (nº medio de operadores) y de la profundidad. Si el número de operadores es *n* y la solución está en la profundidad *p*, el tiempo necesario será $1+n+n^2+n^3\ldots.+n^p$; el coste temporal está en $O(n^p)$.

- **Complejidad espacial.**- Si se juntan los estados que almacena ABIERTA (los que están pendientes de expandir) con los que almacena CERRADA (los que ya están expandidos), resulta que necesitamos almacenamiento para todos los estados que se vayan a generar en la búsqueda de la solución. Este número será la suma de cada nivel que será de $1+n+n^2+n^3\ldots.+n^p$; el coste espacial está en $O(n^p)$.

3.2.5. Conclusiones

Podemos decir que tiene como ventaja la de que si existe una solución, siempre la encuentra y que además es la de menor coste (si son costes uniformes) y en todo caso la

[18] Esto solo es cierto si el número máximo de hijos es finito. Si este fuera infinito, y se diera el caso en el nivel n, no se terminaría nunca este nivel, y por tanto nunca llegaríamos a la solución, salvo que esta se encuentre en ese mismo nivel n.

de menor profundidad. Como desventajas están la complejidad espacial y temporal que son de orden exponencial (si el factor de ramificación es muy grande, se pueden almacenar innecesariamente muchos estados).

3.3.- BUSQUEDA EN PROFUNDIDAD

Esta estrategia, que es la aplicación del recorrido en profundidad descrito en el capítulo anterior, la podemos definir diciendo que dado un estado cualquiera en el nivel de profundidad N-1, este se expande; se coge uno de sus hijos de nivel N y se expande solo este; se coge uno de los hijos de nivel N+1 y se expande solo este…., es decir, se va a expandir siempre uno de los nodos más profundos presentes. Este proceso sigue hasta que se llega a un callejón sin salida, en cuyo punto se retrocede un nivel y se procede con alguno de los otros estados abiertos de ese nivel, si quedan; si no quedan se retrocede otra vez al siguiente nivel superior. El proceso terminará con error si no quedan nodos para expandir antes de encontrar una meta.

Durante este proceso nos podemos encontrar con que no se llega nunca a un callejón sin salida. En este caso, la rama en la que hemos entrado puede ser muy larga, incluso infinita, que finalmente consumirá los recursos disponibles sin haber llegado a una meta, la cual podría estar a una profundidad menor en otra de las posibles ramas. Para evitar esto hay que establecer un **límite de profundidad (lp) o de exploración**, es decir, la profundidad máxima del grafo a partir de la cual consideramos que no se encontrará una solución (o que una de estas no estará lo suficientemente cerca de la raíz sin consumir todos los recursos del sistema). Pero ¿Cuál es el límite óptimo? Depende del problema y su elección es una solución de compromiso: si se elige un límite poco profundo, no se llegará nunca a la solución y si es muy grande, se puede llegar a consumir todos los recursos disponibles.

Para almacenar la evolución de la búsqueda vamos a utilizar una estructura de datos del TDA pila (LIFO), que llamaremos ABIERTA, donde van a almacenar los nodos abiertos (frontera de expansión). Cada una de las celdas va a almacenar el estado, un apuntador al estado padre y el nivel al que se encuentra.

Según vamos aplicando el algoritmo, vamos dibujando el grafo de exploración. Dependiendo de la solución requerida tendremos dos variantes: Que la solución sea la meta,

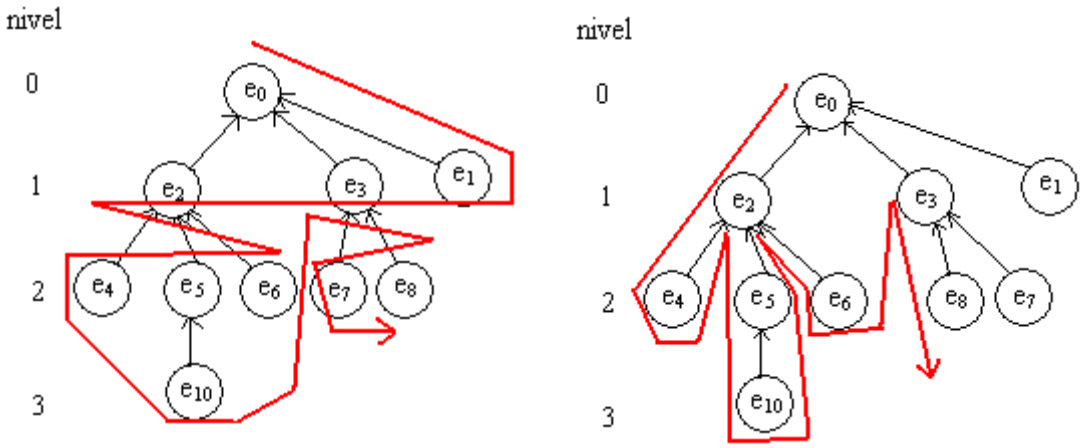
cuyo caso la solución es inmediata, o que sea la secuencia de operaciones hasta la meta, en cuyo caso seguimos el camino más corto desde la meta a la raíz en el grafo de exploración obtenido (árbol implícito con la solución).

Ejemplo 2:

Supongamos que la descripción del problema es el estado e_0 (raíz del grafo) y se le pueden aplicar dos operadores y uno de ellos con dos instanciaciones, lo que nos da tres estados de nivel 1: e_1, e_2, e_3. A e_1 se puede aplicar un operador, a e_2 tres operadores y a e_3 dos operadores, que nos da seis estados de nivel 2: e_4, e_5, e_6, e_7, e_8, e_9. En nivel 3, a e_5 se puede aplicar 1 operador, a e_4 ningún operador….. Fijamos lp=3.

En el grafo de exploración de la secuencia de acciones que tenemos a continuación, vemos, señalado en rojo, a la izquierda el orden en el que se generan los estados a partir del padre: primero los hijos del estado inicial, luego los sucesores de (e_2), luego los sucesores de (e_5)… y a la derecha vemos, señalado en rojo, el orden en el que se expanden los estados a partir del padre (justo al revés por el efecto de la pila, aunque podría haber sido de otra forma por el orden no informado de aplicación de operadores): primero el estado inicial, luego un estado del nivel 1(e_2), luego un estado del nivel 2 (e_4), donde, como es un callejón sin salida se pasa a otro de los sucesores de (e_2), que es (e_5). El sucesor de (e_5) es (e_{10}): como este último está a nivel lp y no es meta se pasa al siguiente sucesor de (e_2). ,….. y así sería hasta que se encuentre una solución, si la hay, en un nivel menor o igual a lp. Podemos observar que los hijos de e_0 no se generan correlativos: esto es por el carácter no informado de la estrategia de control (ha dado la casualidad que ha generado los sucesores de e_0 en el mismo orden que el ejemplo 1, pero podía no haber sido así). En el momento mostrado en el grafo, el estado de ABIERTA, sería (el primero es la cima de la pila):

ABIERTA : e_8, e_7, e_1



--

3.3.1. Algoritmo para búsqueda en profundidad. Nivel de conocimiento

- Crear la pila ABIERTA poner en su primer elemento el nodo raíz (descripción del problema), asignándole profundidad 0.
- HASTA que ABIERTA esté vacía (error) o el estado sea META
    - Extraer (eliminar) de ABIERTA el primer nodo de la lista, y asignarlo a 't' (temporal).
    - SI la profundidad de 't', es menor que 'lp' (límite de profundidad)
        //expandimos 't'
        - PARA cada operador y cada instanciación de operador aplicable a 't'
            - Aplicar el operador a 't'. Obtendremos un nuevo estado 's' (sucesor). A este le asignamos como padre a 't' y su nivel la de su padre más 1.
            - SI 's' es meta
                - Terminar con la expansión de nodos
            - SINO
                - incluirlo en la cima de ABIERTA.
        - //fin PARA cada operador
    - //fin SI se ha llegado al límite de exploración
- //fin HASTA que ABIERTA está vacía (error) o el estado es META
- SI es META
    - Devolver la solución **(1)**
- SINO //ABIERTA está vacía = error
    - Mensaje de error. No se ha encontrado una solución posible.

(1) Si la solución es la meta, se devuelve 's' y si es la secuencia de operaciones damos el camino más corto de la meta a la raíz en el grafo de exploración obtenido.

3.3.2. Análisis del algoritmo. Nivel de conocimiento

- **Completo.-** No es completo, pues no encontrará una solución si está a un nivel superior a lp (más profundo). Si no hubiese límite de exploración tampoco sería completo, pues podemos entrar en una rama que no tenga metas.

- **Óptimo**.- No es óptimo, pues la primera meta que encuentre, si está en esa rama, puede no ser la de menor coste y también puede que no sea la de menor profundidad (podría estar en otra rama a una profundidad menor).

- **Complejidad temporal**.- El caso peor será si se encuentra la meta a la profundidad lp en la última rama posible. En este caso se van a expandir todos los estados posibles, así que la complejidad dependerá del factor de ramificación y de la profundidad; esta complejidad está en $O(n^p)$.

- **Complejidad espacial**.- El caso peor será cuando tengamos, en cada nivel que se expande, un estado sin eliminar ninguno de sus hijos. Así, si r es el factor de ramificación, vamos a almacenar r estados a la profundidad p, luego vamos a almacenar p veces r, luego el coste estará en O(np).

3.3.3. Algoritmo para búsqueda en profundidad. Nivel simbólico

Al igual que en la búsqueda en profundidad, si la solución es solo el estado meta, el algoritmo descrito funciona al nivel simbólico. También, si la solución buscada es el camino desde la meta hasta la raíz tenemos que representar el grafo de exploración en el nivel simbólico. Para dar una solución (no la mejor) se puede definir otra estructura de datos del TDA pila, que llamaremos CERRADA, donde va a almacenar los estados que ya han sido expandidos.

- Crear la pila ABIERTA poner en su primer elemento el nodo raíz (descripción del problema).
- Crear la lista CERRADA, que inicialmente estará vacía. **(1)**
- HASTA que ABIERTA esté vacía (error) o el estado sea META
    - Extraer (eliminar) de ABIERTA el primer nodo de la lista, y asignarlo a 't' (temporal), y a su vez lo copiaremos en la siguiente posición de CERRADA.
    - SI la profundidad de 't' es menor que 'lp' (límite de profundidad)

//expandimos 't'
- PARA cada operador y cada instanciación de operador aplicable a 't'
  - Aplicar el operador a 't'. Obtendremos un nuevo estado 's' (sucesor). A este le asignamos como padre a 't' y su nivel el de su padre más 1.
  - SI 's' es meta
    - Añadir 's' a CERRADA y terminar con la expansión de nodos
  - SINO
    - incluirlo al final de ABIERTA.
- //fin PARA cada operador

- //fin SI se ha llegado al límite de exploración

- //fin HASTA que ABIERTA está vacía (error) o el estado es META
- SI es META
  - La solución estará en CERRADA: se visitará desde el estado meta hasta el estado raíz, en el orden dado por el puntero al estado padre.
- SINO //ABIERTA está vacía = error
  - Mensaje de error. No se ha encontrado una solución posible.

(1) Si solo se pide la meta como solución, no se necesitaría la estructura CERRADA

3.3.4. Análisis del algoritmo. Nivel simbólico

- **Completo.-** No es completo, pues no encontrará una solución si está a un nivel superior a lp (más profundo). Si no hubiese el límite de exploración tampoco sería completo, pues podemos entrar en una rama que no tenga metas.

- **Óptimo**.- No es óptimo, pues la primera meta que encuentre, si está en esa rama, puede no ser la de menor coste y también puede que no sea la de menor profundidad (podría estar en otra rama a una profundidad menor).

- **Complejidad temporal**.- El caso peor será si se encuentra la meta a la profundidad lp en la última rama posible. En este caso se van a expandir todos los estados posibles, así que la complejidad dependerá del factor de ramificación y de la profundidad; ya calculamos que está en $O(n^p)$.

- **Complejidad espacial**.- Dado que los estados almacenados en ABIERTA y en CERRADA constituyen todos los estados visitados, significa que el caso peor, en el

que si se encuentra la solución en el último operador aplicado a un estado de profundidad lp y en la última rama posible, tendremos que almacenar todos los estados, por lo que estamos en el mismo caso que en la búsqueda en amplitud. Su coste estará en O (n^p).

3.3.5. Conclusiones

El seudo-código mostrado no es la mejor solución, ya que el recorrido en profundidad es de naturaleza recursiva (ya dijimos que no entraríamos en temas de implementación).

Podemos decir que, como ventaja tenemos que necesita menos espacio de almacenamiento que la búsqueda en anchura (si la solución es la meta) y como desventajas tenemos que su complejidad temporal es de orden exponencial, tener que elegir un límite de profundidad adecuado, y la solución, si la encuentra (puede no encontrarla aun habiéndola), puede no ser la de menor coste y tampoco la de menor profundidad.

3.4.- BUSQUEDA CON RETROCESO CRONOLÓGICO

Se trata de una derivación de la búsqueda en profundidad anterior. La diferencia está en que, en lugar de expandir el más profundo, lo que se hace es generar sólo uno de los estados hijos, marcando el padre con los operadores que quedan por aplicar[19]. Si se llega a un callejón sin salida o al límite de exploración, se retrocede al primer estado que le queden operadores por aplicar.

La solución se obtendrá de la misma forma que el recorrido en profundidad, es decir, desde el grafo de exploración dibujado.

3.4.1. Algoritmo para búsqueda con retroceso cronológico. Nivel de conocimiento

- Crear la pila ABIERTA poner en su primer elemento el nodo raíz (descripción del problema).

[19] En el estado está la descripción de los operadores y condiciones de aplicabilidad del mismo; la marca de aplicación ocupa menos espacio que volver a almacenar el estado completo.

- HASTA que ABIERTA esté vacía (error) o el estado sea META
    - Examinar en ABIERTA el primer nodo de la lista, y asignarlo a 't' (temporal).
    - SI la profundidad de 't', (p), es menor que 'lp' (límite de profundidad)
        - //generación de hijo de 't'
        - SI a 't' le quedan operadores y/o instanciaciones por aplicar
            - Elegir el siguiente operador y aplicarlo a 't'. Obtendremos un nuevo estado 's' (sucesor). A este le asignamos como padre a 't' y su nivel la de su padre más 1.
            - incluirlo en la cima de ABIERTA
            - SI 's' es meta
                - Terminar con la generación de nodos
        - //fin SI le quedan operadores
        - SINO // ya se ha terminado de expandir 't'
            - Eliminar 't' de ABIERTA.
    - //fin SI se ha llegado al límite de exploración
- //fin HASTA que ABIERTA está vacía o el estado es META
- SI es META
    - Devolver la solución **(1)**
- SINO //ABIERTA está vacía = error
    - Mensaje de error. No se ha encontrado una solución posible.

(1) Si la solución es la meta, se devuelve 's' y si es la secuencia de operaciones damos el camino más corto de la meta a la raíz en el grafo de exploración obtenido.

3.4.2. Análisis del algoritmo

- **Completo.-** No es completo, pues no encontrará una solución si está a un nivel superior a lp. Si no hubiese límite de exploración, tampoco sería completo, pues podemos entrar en una rama que no tenga metas.

- **Óptimo**.- No es óptimo, pues puede no darnos la solución de menor coste y también puede que no sea la de menor profundidad (podría estar en una rama a una profundidad menor).

- **Complejidad temporal**.- El caso peor será si se encuentra la meta a la profundidad lp en la última rama posible. En este caso se van a expandir todos los estados posibles, así que la complejidad dependerá del factor de ramificación y de la profundidad; ya calculamos que está en $O(n^p)$

- **Complejidad espacial**.- En este caso es menor, pues en cada nivel se genera exactamente un estado, luego, en todo momento, en el grafo de exploración, tendremos una secuencia de estados que constituye una de las soluciones parciales candidatas (denominadas en conjunto *solución parcial del problema*), con una profundidad p, es decir p estados. En el caso peor, en ABIERTA tendremos este conjunto, luego el coste estará en el orden de O(p).

3.4.3. Conclusiones

Al ser también un recorrido en profundidad (naturaleza recursiva), el seudo-código mostrado no es la mejor solución.

Este método solo aventaja al anterior en que necesita menos espacio de almacenamiento que la búsqueda en profundidad; como desventajas, las mismas que el método del que se deriva: su complejidad temporal es de orden exponencial, hay que elegir un límite de profundidad adecuado y la solución puede no ser la de menor coste y también puede que no sea la de menor profundidad.

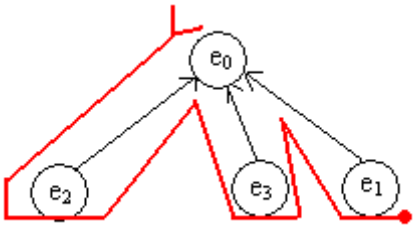
3.5.- BUSQUEDA EN PROFUNDIDAD PROGRESIVA

También denominada como *descenso iterativo o profundidad iterativa*, esta es una técnica también derivada de la búsqueda en profundidad. Esta consiste en realizar la tarea de búsqueda en sucesivos niveles, esto es, primero realiza una búsqueda con lp=1, si no se llega a la meta, se vuelve a realizar la búsqueda con lp=2, si no se llega a la meta, volver a realizar la búsqueda con lp=3, y así hasta que se encuentre la meta o consuma todos los recursos.

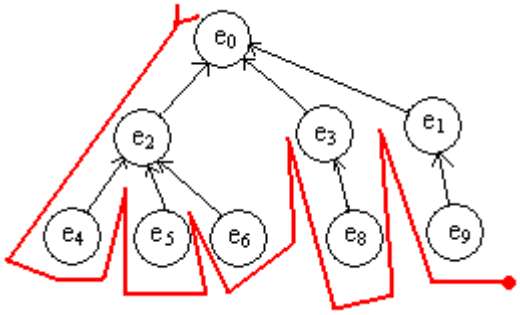
Ejemplo 3:

Si aplicamos este algoritmo al problema descrito en el ejemplo 2, tendremos el siguiente grafo (conjunto de subgrafos de nivel):

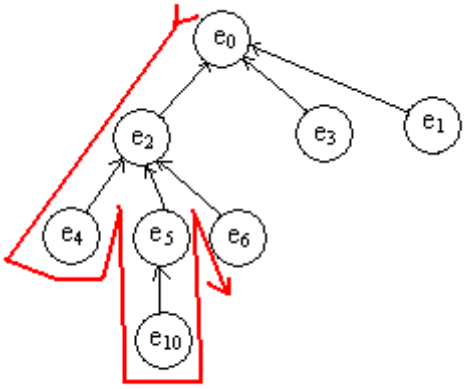
- Para nivel 1:



- Para nivel 2:



- Para nivel 3: Este está incompleto: está en el instante en el que hay que expansionar el estado e_6:



--

3.5.1. Algoritmo para búsqueda en profundidad progresiva

- Inicializar el límite de exploración lp=1 (primer nivel)
- HASTA que se encuentre una META
  - Algoritmo de búsqueda en profundidad (o con retroceso cronológico).
  - Aumentar el límite de exploración (pasar al siguiente nivel)
- FIN HASTA

3.5.2. Análisis del algoritmo

- **Completo.-** Es completo[20], pues si hay una solución, esta estrategia la encuentra (siempre con los recursos disponibles). Esto es así porque no se expande un nivel sin haber visitado completamente el nivel anterior.

- **Óptimo.**- Este método sí lo es, pues de haber solución, nos ofrece la de menor coste (si el coste de los operadores es uniforme) y en todo caso la de menor profundidad.

- **Complejidad temporal.-** A primera vista, vemos que en cada nivel se vuelven a expandir los estados de los niveles superiores, y que esto podría aumentar el coste temporal del algoritmo, pero esto, en términos asintóticos, no es así. Si observamos, dado un nivel p, el primer nivel se expandirá p veces el factor de ramificación n, el segundo nivel p-1 veces,… el nivel p una vez. Así nos queda que

$$pn + (p-1)n^2 + (p-2)n^3 + \ldots + 2n^{p-2} + n^{p-1} + n^p \equiv O(p/2)(n^p) \equiv O(n^p)$$

  De donde deducimos que la complejidad temporal solo empeora en una constante (p/2) la complejidad del algoritmo original; por tanto está en el mismo orden que los anteriores algoritmos.

- **Complejidad espacial.**- en ABIERTA vamos a tener en todo momento los mismo estados que en los algoritmos originales de búsqueda utilizados, por lo que su complejidad espacial será la misma:
  - Búsqueda en profundidad O(np)
  - Búsqueda con retroceso O(p).

3.5.3. Conclusiones

Esta técnica está aunando las ventajas de la búsqueda en profundidad (necesita menos almacenamiento) y las de la búsqueda en amplitud (si hay una solución encuentra la menos profunda). Esto último es cierto dado que no se empieza a explorar un nivel sin haber terminado el anterior.

[20] Esto es cierto si el número de hijos de los estados es finito.

Como vemos en el análisis de las características, vemos que la incidencia de volver a expandir los nodos no es tan grande como a priori se pudiera pensar. Por otro lado, si comparamos la complejidad temporal con la de la búsqueda en anchura, veremos que en el último nivel (que es donde más estados se expanden), en promedio, con la búsqueda en anchura se habría expandido parte del último nivel mientras que en la búsqueda en profundidad progresiva no lo hace.

En conclusión, es el método más eficiente de todos los vistos hasta ahora, siendo además completo y óptimo (esto es cierto si no se añade una limitación de lp máximo, cuyo caso tendríamos las mismas desventajas que la búsqueda en profundidad).

3.6.- BUSQUEDA BIDIRECCIONAL

Esta se puede aplicar a un problema en el que conocemos la situación inicial (descripción) del mismo y del que, además, conocemos una meta de manera explícita. En este caso podemos utilizar dos procesos de búsqueda: uno desde la descripción del problema hacia la solución (es decir, encadenado hacia delante) y otra desde la meta hacia la descripción del problema (es decir, encadenado hacia atrás), de manera que el proceso termina cuando las dos búsquedas confluyen en algún estado.

Para la representación con grafos, tenemos uno para la búsqueda desde el nodo raíz al nodo meta y otro para la búsqueda desde la meta hasta el nodo raíz. Durante el proceso se alterna entre uno y otro grafo. Si la alternancia es de nivel en nivel (pueden ser varios niveles), dado un nivel n-1, en la búsqueda encadenada hacia delante se generan todos los estados del nivel n comprobando si alguno de ellos está en la frontera de exploración de la otra búsqueda; si es así, las dos búsquedas se han encontrado, y la solución será la composición del resultado de las dos búsquedas: el camino desde la situación inicial a la meta pasando por el estado común. La pertenencia a la frontera de exploración se puede comprobar en solo uno de los grafos o en los dos.

Para que se pueda realizar este tipo de búsqueda se tienen que cumplir dos condiciones:

- **Los operadores han de ser reversibles.**- Lo fácil es que se determine el nodo padre en una sola operación (podría darse el caso de requerir más de una para ello, en cuyo caso se complica la equiparación de manera notable).

- **Que la meta sea explícita.** Si para un problema hay varias metas distintas, habrá que hacer una búsqueda para cada una de ellas.

3.6.1. Algoritmo para la búsqueda bidireccional

Para este algoritmo, vamos a considerar que las dos búsquedas son en amplitud y que se va a comprobar en las dos si el nodo a generar en un grafo está en la frontera de exploración del otro. El algoritmo sería el siguiente:

- Inicializar un grafo de búsqueda, que llamamos P, cuyo nodo raíz es la descripción del problema, que añadimos a ABIERTA(P)
- Inicializar un grafo de búsqueda, que llamamos M, cuyo nodo raíz es la meta del problema, que añadimos a ABIERTA (M).
- Inicializar 'p' al número de niveles que se van generar consecutivamente **(1)**
- HASTA que se encuentre el camino de la solución o ABIERTA(P) vacía
    - PARA cada nodo de P en los 'p' niveles a expansionar, estos se eliminan de ABIERTA(P) y se generan sus sucesores de manera que
        - SI el nodo siguiente a generar está en ABIERTA(M)
            - Incluirlo en P y en ABIERTA(P) y devolver solución.
        - SINO
            - Incluirlo en ABIERTA(P) y en P.
    - //fin para
    - Si no se encontraron
        - PARA cada nodo de M en los 'p' niveles a expansionar, estos se eliminan de ABIERTA(M) y se generan sus sucesores de manera que
            - SI el nodo siguiente a generar está en ABIERTA(P)
                - Incluirlo en M y en ABIERTA (M) y devolver solución
            - SINO
                - Incluirlo en ABIERTA(M) Y EN M
        - //fin PARA cada nodo de M
    - //fin si no se encontraron
- //fin HASTA que se encuentre el nodo solución
- Devolver el camino desde la raíz hasta la meta, pasando por un estado en común de las dos búsquedas (puede haber más de una solución).

(1) La opción más eficiente es que en cada nivel se cambie de grafo de búsqueda (p=1). Aunque se podría aumentar el número de niveles antes de cambiar de grafo (este es un parámetro crítico).

3.6.2. Análisis del algoritmo

- **Completo.**- Lo es, ya que las dos estrategias son de búsqueda en amplitud. Si se conoce la meta y el problema, siempre se encuentra la solución.

- **Óptimo**.- Si las dos estrategias son en amplitud, el algoritmo es óptimo por serlo también las anteriores.

- **Complejidad temporal**.- El caso peor ocurre cuando las dos búsquedas se encuentran en la mitad del grafo, es decir a profundidad p/2. Si el factor de ramificación es *n* y la solución está en la profundidad *p*, el tiempo necesario para llegar a la solución será $2(1+n+n^2+n^3….+n^{p/2})$, ya que es el mismo tiempo para los dos. El coste temporal está en O ($n^{p/2}$). No se tiene en cuenta el tiempo que tarda en comprobar la frontera de exploración, pues su coste es muy pequeño en comparación con el resto.

- **Complejidad espacial**.- El caso peor también ocurre cuando se cruzan a profundidad p/2. En la estrategia en amplitud, también vamos a almacenar $2(n^{p/2})$ estados. Así el coste espacial también está en O ($n^{p/2}$).

3.6.3. Otras variaciones para este algoritmo

- Para que el algoritmo funcione, una de las dos estrategias ha de serlo en amplitud. Esto es así porque es necesario conocer en todo momento todos los nodos frontera en un nivel determinado. Si las dos fuesen en profundidad, podría resultar que cada una fuese por una rama distinta y que no se encontrasen nunca (no sería completo).

- Otras combinaciones que resulten con una de las dos estrategias en amplitud no afectan a los costes. Por ejemplo, si el algoritmo fuese una combinación de amplitud-profundidad:

- Si el factor de ramificación es n (para amplitud) y la solución está en profundidad p, el tiempo necesario seguiría siendo el mismo, $2(1+n+n^2+n^3\ldots.+n^{p/2})$, luego esta en O $(n^{p/2})$.

- El coste de almacenamiento cambia un poco: para la búsqueda en amplitud tendremos que almacenar $n^{p/2}$ y en la de profundidad pn; dado que $n^{p/2} >> pn$, el coste de almacenamiento también estará en O $(n^{p/2})$.

- El algoritmo será completo, pero no óptimo, ya que la rama de la búsqueda en profundidad puede no ser la de menor coste ni la del camino más corto.

- Otra variación posibles es que, en lugar de cambiar de dirección de búsqueda en cada nivel, se cambiara cada vez que se genera un estado. Esta alternativa tampoco es la más eficiente, pues lo que se gana con la posible no expansión de nodos en la frontera se pierde con los cambios de una a otra búsqueda, que en este caso sí pueden influyen en la complejidad temporal.

3.6.4. Conclusiones

Si una de las dos estrategias es en amplitud, el algoritmo es completo. La ventaja de esta búsqueda es que reduce a la mitad el factor exponencial del coste computacional con respecto a la búsqueda en amplitud, siendo a su vez, la principal desventaja desde el punto de vista del coste de almacenamiento (aunque se reduce drásticamente el número de nodos a almacenar, este es todavía muy grande).

Se puede utilizar en problemas en los que se conoce la meta y el estado inicial.

3.7. ALGORITMO GENERAL DE BÚSQUEDA EN GRAFOS

Anteriormente se dijo que durante la búsqueda se genera un grafo de estados explicito, que contiene todos los estados generados, y que implícito en este había un árbol desde la meta al estado inicial, recorriendo sucesivamente los punteros al estado predecesor.

Hasta ahora se ha considerado que en la expansión de estados, estos no se repetían. La realidad es que hay problemas que se pueden describir de manera que nunca se repita un estado anterior, pero hay otros muchos para los que esto es imposible; siempre se puede generar un estado que ya lo estaba anteriormente. En los primeros no hay problema, pero en los segundos el problema está en que se puede llegar a un nodo por distintos caminos; esto se traduce en la formación de ciclos (el grafo de exploración ya no sería un GDA); estos hay que evitarlos, ya que la búsqueda en este supuesto podría entrar en un bucle sin fin. El algoritmo que veremos es importante por ser una generalización para todos los vistos y la mayoría de los que veremos a continuación.

El coste de comprobar si un nuevo estado ya generado estaba ya en el grafo es muy grande, sobre todo en problemas complejos, en los que el número de estados puede ser inmenso. Pero ¿qué cuesta más, expandir nodos repetidos o comprobar si ya se habían generado? Para llegar a un equilibrio entre el coste de comprobar estados repetidos y el coste de generarlos, lo que se hace es **comprobar la duplicación en el árbol implícito,** a lo largo del camino desde el estado al que se le está aplicando el operador hasta la raíz. Si no está en este árbol, el nodo se genera, esté o no repetido.

En el caso de estar generando un estado repetido que esté en el árbol implícito, y antes de desechar uno de los dos, **tenemos que evaluar cual es el que nos ofrece el camino menos costoso (con menos pasos si el coste de los operadores es uniforme)**. Así, si el nuevo estado generado ya existía y es menos costoso (menos profundo si el coste de los operadores es uniforme), lo que se hace es cambiar el puntero del antiguo, para que apunte al padre del nuevo. En caso contrario se desecha el nuevo. Uno de los motivos de mantener la estructura ABIERTA y CERRADA es poder realizar esta comprobación. Cuando se genera un nodo se pueden dar tres situaciones:

- Que el estado no esté en ABIERTA ni en CERRADA.- En este caso se enlaza el nodo generado con su padre y se añade a ABIERTA.

- Que el nodo generado ya esté en ABIERTA.- En este momento se abre un nuevo camino al nodo generado. Se evalúa si el nuevo camino es menos costoso que el anterior, y si es así se redirige el puntero del nodo que ya existía al padre del nuevo nodo generado. En caso contrario se ignora el estado.

- Que el nodo generado ya esté en CERRADA.- Significa que este ya se había expandido anteriormente. Si se da la situación de antes, entonces habrá que actualizar el puntero de todos los sucesores al padre del estado repetido menos costoso, y si procede, cambiar también los punteros de sus descendientes.

3.7.1. Algoritmo general de búsqueda en grafos de Nilsson

La estructura ABIERTA será un TDA cola, CERRADA será una estructura de TDA pila, y cada celda contendrá el estado, un puntero a su padre y la profundidad a la que se genera (se consideran costes uniformes).

- Crear un grafo de búsqueda G, que solo va a tener la raíz con el estado inicial del problema.
- Crear una lista ABIERTA e incluir el nodo raíz.
- Crear una lista CERRADA que en el momento inicial estará vacía.
- HASTA que ABIERTA esté vacía (fallo) o se llegue a la META
    - Extraer (eliminar) de ABIERTA el primer nodo de la lista, y asignarlo a 't', y a su vez lo copiaremos en la siguiente posición de CERRADA.
    - Expandir 't':
        - Generar el conjunto S de todos los sucesores de 't' **que a su vez no sean antepasados de 's' en G**, e introducirlos en el grafo de exploración G como sucesores de 't' **(1)**
        - SI algún miembro de S es META
            - terminar con la expansión
        - SINO
            - Para los miembros de S que no están en ABIERTA ni CERRADA, añadirlos a ABIERTA, y asignarle 't' como su predecesor **(2)**
            - Para los miembros de S que estén en ABIERTA o en CERRADA:
                - Si el miembro de S está menos profundo, se redirige el apuntador del estado en la estructura de datos al padre del miembro de S.
                - Para los miembros de S que estén en CERRADA, decidir además si para los sucesores del estado repetido hay que redirigir sus punteros, así como sus descendientes.
    - Reordenar la lista ABIERTA, según un esquema arbitrario **(3)**.
- //fin HASTA que ABIERTA esté vacía o sea META
- SI es META
    - Dar la solución siguiendo los punteros desde la meta hasta el estado inicial
- SINO // fallo
    - Dar mensaje de error.

(1) Esta restricción evita que se cree un ciclo en el grafo de exploración.

(2) El adaptar este algoritmo a una búsqueda ciega en amplitud o en profundidad, es tan fácil como

a. En amplitud, ABIERTA es una cola (FIFO).

b. En profundidad ABIERTA es una pila (LIFO). Para esta estrategia, se determina como nodo más profundo el último estado que entró en la cola o la pila. (el coste de comprobar la profundidad de cada nodo del grafo de exploración puede ser prohibitivo, aparte de lo que ya es de por sí el algoritmo).

c. Eliminar la reordenación de ABIERTA (ver a continuación).

En el caso de búsqueda exhaustiva, la reordenación no tiene sentido. La tiene en el caso de los métodos informados (heurísticos) en los que se puede elegir el mejor de los candidatos a ser expansionado cada vez (camino más prometedor). Además, dado un estado a nivel n, no vamos a generar nunca ese mismo estado con un nivel menos profundo, por lo tanto, si los costes son uniformes, no necesitamos comprobar todo lo relativo a si está en ABIERTA o en CERRADA. Así, el algoritmo sería el que se ha visto originalmente, pero con la única modificación de que el estado a generar no sea antepasado de sí mismo, es decir, eliminando simplemente el nuevo camino.

3.7.2. Análisis del algoritmo

Las características de completo y óptimo así como la complejidad temporal y espacial dependerán, en las búsquedas ciegas, de si son en amplitud o en profundidad y en las informadas de las heurísticas utilizadas (se verán más adelante). El coste temporal, no estará influenciado por el procedimiento de análisis de estados repetidos, ya que esta comprobación tiene un coste bastante menor que el de generar estados nuevos.

3.7.3. Conclusiones

Podemos ver que este algoritmo es la generalización a la búsqueda en amplitud antes vista, pero tiendo en cuenta la repetición de estados en el grafo de exploración. Como veremos en el siguiente capítulo también va a ser la base para los algoritmos de búsqueda heurística más eficientes.

3.8. REFERENCIAS

Para un análisis de costes más exhaustivo de los algoritmos en amplitud, profundidad y profundidad iterativa, se puede consultar [Ginsberg 1993].

--

CAPÍTULO 4.- TAREAS DE BÚSQUEDA HEURÍSTICA

4.1.- INTRODUCCIÓN

Ya se dijo que el objetivo de los métodos de búsqueda heurística es reducir el número de estados a generar durante la búsqueda. Utilizan en su estrategia de control las **funciones de evaluación heurística**, que llamaremos **fev**. Estas son una aplicación del conjunto de todos los estados posibles en $\mathbb{R}$

$$f:\{estados\} \rightarrow \mathbb{R} \quad \text{tal que} \quad f(estado_j)=n_j$$

De esto se deduce que e**l valor** de esta función **depende exclusivamente del estado que se está evaluando en un instante dado**, es decir, para ese estado, el valor es función de la información disponible hasta ese momento sobre la búsqueda. Por regla general, el sistema experto almacenará esta información, refinándola mediante el aprendizaje.

Este valor numérico lo que hace es una **ESTIMACIÓN de lo bueno que puede ser este estado para llegar a la meta**. Esta estimación, que suele ser de coste pero que también puede ser de otro parámetro, cumple dos condiciones:

- El valor mínimo (máximo) de esta función se ha de dar cuando se llega a la meta.

- Se pretende que el valor calculado por esta función sea óptimo durante todo el camino. En este caso se dice que la fev es óptima, y por tanto nos va a dar la mejor solución (la óptima): la búsqueda sería la simple ejecución de una secuencia de operadores. En un caso real no sabemos cual es el camino óptimo (de ahí la búsqueda) y lo que se hace es buscar, a lo largo del camino, el valor que más se acerque a este valor óptimo.

Un detalle que hay que tener en cuenta es que la fev, aunque utilice el conocimiento del dominio para el cálculo de su valor, forma parte del solucionador y por tanto forma parte de las tareas de búsqueda; estas son, recordemos, las que usan los métodos de resolución para definir las tareas genéricas (no dependientes del dominio).

Encontrar una fev óptima podría ser fácil si no fuese por un detalle: su cálculo también tiene un coste asociado: puede darse el caso de que el coste del algoritmo con esa fev óptima sea mayor que el coste del algoritmo sin ella. Así pues, volvemos a encontrarnos en una situación en la que debemos llegar al compromiso entre el coste del algoritmo con fev y el coste sin ella.

Para identificar una fev, lo que hacemos es identificar todas las restricciones que ha de cumplir el problema. Luego simplificamos el modelo, es decir, relajamos estas restricciones (quitamos una o varias), de manera que se convierte en el mismo problema, pero menos estricto (subproblema más sencillo). Con esto lo que conseguimos es que la fev sea más simple, pero lo suficientemente útil para que nos estime lo bueno que sea el estado para llegar a la meta en función de las restantes restricciones. Para poder simplificar el modelo, el problema ha de poderse **descomponer en subproblemas** más sencillos.

Dado un problema, el valor calculado por una fev, *que estima lo bueno que es el estado para llegar a la meta teniendo en cuenta todas las restricciones*, va a ser mejor que el calculado por todas las fev que estimen lo mismo, pero cumpliendo solo algunas de estas restricciones. Cuanto más simple sea el modelo relajado, menor coste tendrá la fev, ya que esta es más simple, pero, en contrapartida, también dirigirá peor la búsqueda.

Una fev que solo mida un parámetro del problema, además de ser demasiado simple para estimar la distancia a la meta, también puede producir otros problemas:

- **Máximos (mínimos) locales**.- Es un estado que es el estimado como mejor que todos los que le rodean, pero que se estima peor que alguno que está más alejado.

- **Altiplanicies o mesetas.-** Son un conjunto de estados que tienen el mismo valor de fev, por lo que no se tiene información de por donde seguir.

Estos inconvenientes se pueden solventar, aunque no definitivamente, con fev,s que midan más de un parámetro y eligiendo adecuadamente el método de búsqueda.

4.1.1. Algoritmo general de búsqueda en grafos

Un ejemplo inmediato de uso de la fev es el algoritmo general de búsqueda en grafos (se completará más adelante). En él, una de las sentencias decía "Reordenar la lista ABIERTA" y decíamos que en los métodos ciegos no tenía razón de ser. En este caso se aplica la fev a todos los estados de ABIERTA, ordenándolos en función de ese valor[21] (de mayor a menor si se está estimando un máximo de fev y de menor a mayor si se está estimando un mínimo de fev). De esta manera, como en ABIERTA tenemos los estados a expandir, elegiremos el de la primera posición, que es el que tiene el mejor valor de fev.

4.1.2. Problema del 8 puzzle

Es un recuadro dividido en 9 celdas, cada una de las cuales, excepto una, contiene una ficha cuadrada. Entre todas las fichas componen un dibujo. El objetivo del juego es, dada cualquier situación en que las fichas están desordenadas, buscar como volver a colocarlas; gráficamente

Dado

5	4	7
	1	6
3	8	2

Encontrar

1	2	3
8		4
7	6	5

Este problema tiene dos restricciones:

- Solo podemos mover una ficha a una casilla adyacente (distancia 1)
- Solo podemos mover si la casilla está vacía.

Podríamos elegir como fev el número de piezas descolocadas en cada momento: es una mala elección, pues es demasiado simple. Una fev más elaborada sería mediante la suma de

[21] Recordar que estamos hablando a nivel de conocimiento: para que la ordenación no influya en el coste de los algoritmos, lo que se hace es implementar un montículo (cola de prioridad) o con tablas hash, ambas de coste unitario.

las distancias de Manhattan[22] de cada una de las piezas, intentando minimizar esta (fev = 0 en la meta).

4.1.3. Mapa de carreteras

Se trata de encontrar la ruta de menor distancia entre dos ciudades. Una primera aproximación es elegir la siguiente ciudad que esté más cerca, y así hasta llegar a la meta. Esta es demasiado simple. Una segunda aproximación sería utilizar la distancia aérea entre una ciudad adyacente y destino y sumársela a la distancia a la ciudad adyacente. Por ejemplo, podemos ir desde Madrid a Barcelona pasando por Toledo o pasando por Zaragoza; con la primera aproximación, primero elegiríamos Toledo (está más cerca de Madrid) y luego Barcelona. Con la segunda aproximación primero elegiríamos Zaragoza, pues la suma de la distancia aérea entre Zaragoza y Barcelona más la distancia de Madrid a Zaragoza es menor que la suma de la distancia aérea de Toledo-Barcelona más la distancia de Madrid a Toledo. De esta manera el viaje es menos costoso.

4.1.4. Optimizar el tiempo de una ruta

Se trata de encontrar la ruta por la que tardemos menos entre dos ciudades. Una primera aproximación puede ser tener en cuenta solo la distancia entre las dos ciudades. Una segunda aproximación es tener en cuenta qué tipo de carretera es: si es un llano o si hay puertos: en un llano la velocidad media es superior a si hay puertos, por tanto se estima que por un llano se tardaría menos. Una tercera aproximación sería tener en cuenta las condiciones de tráfico de cada camino. Esta información conllevaría la toma de datos durante un largo período de tiempo, y calcular las estimaciones estadísticas pertinentes. Esta tercera aproximación es inviable, porque es mayor el coste del cálculo teniendo en cuenta todas las variables, que si hacemos el camino teniendo en cuenta solo algunas de estas.

[22] Distancia de Manhattan: suma de la distancia horizontal y vertical a su posición correcta. Es cero si está en el sitio correcto.

4.2. PLANTEAMIENTO DEL PROBLEMA

El problema se plantea igual que en los métodos exhaustivos, es decir, con el esquema general para los métodos de búsqueda (tabla 1 del capítulo anterior). La diferencia entre estas estrategias y las anteriores está en que, para dirigir la búsqueda, se define una fev que calcula cual es el **estado más prometedor de los abiertos** para llegar a la meta. Cuanta mayor sea la calidad de la fev, menor será el grafo de búsqueda explícito.

Las estrategias heurísticas de búsqueda clásicas son:

- Método del gradiente (voraz)
- Búsqueda primero el mejor
- Búsqueda en haz
- Búsqueda A*
- Búsqueda en profundidad progresiva (A*-P)
- Búsqueda AO*
- Búsqueda con adversario:
    - Principio Minimax y poda α-β

4.3. MÉTODO DEL GRADIENTE O BÚSQUEDA EN ESCALADA

Es un método de búsqueda sin vuelta atrás (se convierte en un algoritmo voraz)[23]. Corresponde a la *búsqueda en profundidad guiada por una fev*. Al igual que su homólogo exhaustivo, se puede fijar un límite de exploración.

[23] Es el único que se verá que no es por tentativas.

La característica importante de este método es que, una vez tomado un camino, este no se puede dejar (no se evalúan alternativas, se toma una decisión *óptima*). La fev normalmente será creciente (puede haber casos en los que interese que sea decreciente) y lo que se busca es maximizar (minimizar) su valor en la meta. El nombre de este método viene precisamente de su similitud con la escalada de una montaña: la meta es la cima y en cada paso lo que hacemos es ascender unos metros, no pudiendo retroceder en el camino.

En la expansión de cada nodo **se elegirá siempre el que tenga asociado el de mayor (menor) valor de fev**, que además **deberá tener asociado un valor mayor (menor) que el asociado a su nodo predecesor**. Si el valor fuese menor (mayor) o igual que el asociado a su precedesor, entonces estamos en un callejón sin salida. Por tanto, elegir una buena fev es primordial: ha de ser monótona creciente (decreciente), teniendo el máximo (mínimo) valor en la meta (se evitarían los máximos/mínimos locales a los que este método es muy sensible).

Este método es de aplicación a aquellos problemas en los que los espacios de estados son muy grandes, donde los métodos exhaustivos serían inaplicables (explosión combinatoria). También es de aplicación a problemas en los que los operadores cumplan la conmutatividad (estos problemas siempre se resuelven, son completos).

Ejemplos típicos de este métodos los tenemos en la teoría de grafos (algoritmos de Kruscal y Djikstra -minimizar el coste-) y la planificación de tareas (minimizar coste de tiempo). Otras aplicaciones son la búsqueda de mínimos y máximos locales en cálculo numérico, laberintos y situaciones tipo en el ajedrez (mate con torre y rey,….).

4.3.1. Algoritmo para el método del gradiente o búsqueda en escalada

No necesitamos ninguna estructura de datos, pues no necesitamos recordar más que el estado más prometedor, que es el siguiente a expandir. Para ello, se coge el nodo a expansionar, se generan todos sus sucesores y nos quedamos con el que tenga asociado el mayor (menor) valor de la fev que además ha de ser mayor (menor) que el valor asociado al estado padre; si no se cumple esto último, se termina con error. Este algoritmo contempla el caso de que la fev es máxima en la meta:

- Coger el estado raíz (descripción del problema) y asignarlo a 'n' (nodo) y a 't' (temporal) **(1)**
  //f(n) será el menor valor posible = 0
- HASTA que 't' sea META o ERROR
  - PARA cada operador e instanciación de 'n': //todos los sucesores de 'n'
    - Generar un sucesor y asignarlo a 's' (sucesor)
    - SI 's' es META
      - 'n' = 's' y terminar
    - SINO
      - SI f('s') > f('t') ENTONCES 't' = 's' **(2)**
    - // fin SINO es META
  - //fin para cada operador
  - SI f('t') > f ('n') **(3)**
    - 'n' = 't'
  - SINO terminar con ERROR
- //fin hasta que 't' sea META o ERROR
- SI META devolver 'n'
- SINO devolver ERROR

(1) 't' va a contener el estado que se está expandiendo o el estado sucesor que tenga asociado un valor de fev mayor que el asociado a su padre 'n'.

(2) Si algún sucesor tiene asociado un valor de fev superior al asociado al padre, pasa a ser este el candidato elegido para expandir en el siguiente nivel. Si la fev es monótona decreciente, solo hay que invertir la inecuación.

(3) Si el sucesor con mayor valor de fev no es superior al asociado al padre, entonces devuelve ERROR (si se supone una fev monótona creciente la sentencia sería válida con f('t') ≠ f ('n'), dado que nunca va a ser menor).

Si solo se busca la meta, esta es la solución y si se busca el camino seguido, la solución será la secuencia de nodos expandidos, que podemos extraer del grafo de exploración.

4.3.2. Análisis del algoritmo

- **Completo**.- Tiene una gran dependencia respecto de la fev; si está bien informada, se encuentra la solución antes de llegar a profundidad lp, si es que existe; excepto los

problemas que permitan la conmutatividad de operadores, no es completo (en un momento dado puede desechar el camino a la solución).

- **Óptimo.-** Aún cuando la fev sea la adecuada y encuentre una solución, es posible que no encuentre la solución de menor coste (en un momento dado puede desechar el camino a la solución óptima).

- **Complejidad temporal.-** En cada nivel solo se va a expandir un estado, luego la complejidad está en $O(pn)$[24].

- **Complejidad espacial.-** Al solo requerir almacenar el nodo a expandir, esta es $O(1)$.

4.3.3. Conclusiones

La principal ventaja de este método es su poca necesidad de almacenamiento y su coste temporal, de orden lineal en el caso peor. Su principal desventaja es su dependencia con respecto a la definición correcta de la fev: necesita estar muy bien informada para intentar solventar los máximos locales y las mesetas (conjunto de estados con el mismo valor de fev) a los que es muy sensible y que desembocan en error. Aun así puede no encontrar la solución, aunque exista. Tampoco es óptimo.

Para solucionar los máximos locales, la fev debe ser monótona creciente (decreciente) y para solventar las mesetas, se puede elegir aleatoriamente el siguiente estado a expansionar de los que tienen el mismo valor de fev asociado (esta solución, llamada de movimiento lateral, tiene el problema de que se podrían dejar nodos sin explorar o repetir nodos, llevando pareja la posible formación de bucles sin fin).

4.4. BUSQUEDA PRIMERO EL MEJOR[25] (PM)

En la búsqueda PM, que es una derivación del algoritmo general de búsqueda en grafos, la fev mide la **distancia (cercanía) estimada hasta la meta**; como consecuencia su

[24] Recordar que n es el número de operadores (factor de ramificación) y p la profundidad del grafo.
[25] En [Russel Norvig 2004] llaman a este método "búsqueda voraz primero el mejor" o "búsqueda avara (voraz)". Incluir la palabra voraz es un error, porque un algoritmo voraz no evalúa alternativas (que es lo que hace este), se toma una decisión óptima [Brassard 1997].

valor mínimo será 0. Esta distancia podrá ser el coste estimado a la meta, aunque también puede medir cualquier otro parámetro.

En esta búsqueda toma sentido la reordenación de ABIERTA, ya que se hace en función del valor calculado con la fev. **Mientras el valor de la fev sea óptimo, la exploración será en profundidad**, dado que tras la ordenación de ABIERTA esta va a tener siempre en la cima el mejor sucesor en la rama que estamos siguiendo. Si la rama que se está siguiendo se aleja de la meta y tenemos en ABIERTA otro estado con mejor estimación, la ordenación pondrá este último en la cima, tomándose entonces esta nueva rama. En la búsqueda en profundidad exhaustiva si llegábamos a un callejón sin salida o al límite de profundidad se retomaba la exploración en profundidad desde otro estado, hermano del predecesor al callejón sin salida. Ahora se toma el estado más prometedor de los que hay en ABIERTA, y como consecuencia podemos expandir un nodo en cualquier nivel y rama del grafo de exploración. Por ello aquí ya es necesario comprobar si el nuevo estado generado estaba ya en ABIERTA o CERRADA para quedarnos con el estado que se supone más cercano a la meta.

4.4.1. Algoritmo de búsqueda primero el mejor (PM)

La fev, que llamamos h(n), va a medir el coste estimado hasta la meta y su valor calculado para cada estado se almacenará junto con él y el apuntador a su nodo padre dentro de la estructura ABIERTA o CERRADA, la que corresponda. Aquí CERRADA es imprescindible para comprobar que no se repitan estados en el grafo de exploración.

- Crear un grafo de búsqueda G, que solo va a tener la raíz con el estado inicial del problema.
- Crear ABIERTA e incluir el nodo raíz, con su valor de fev.
- Crear CERRADA que en el momento inicial estará vacía.
- HASTA que ABIERTA esté vacía o se llegue a la META
    - Extraer (eliminar) de ABIERTA el primer nodo de la lista, y asignarlo a 't', y a su vez lo copiaremos en la siguiente posición de CERRADA.
    - Expandir 't':
        - Generar el conjunto S de todos los sucesores 's' de 't', que a su vez no sean antepasados de sí mismo en G. Asociarle a cada uno su h(s) e introducirlos en G como sucesores de 't'.
        - Si algún miembro de S es META, terminar con la expansión

- Para los miembros de S que no están en ABIERTA ni CERRADA, añadirlos a ABIERTA junto con el valor h(s), y señalar a 't' como su predecesor.
- Para los miembros s de S que estén en ABIERTA o en CERRADA (está repetido):
  - SI h(s) es mayor que el que ya tenía el estado en ABIERTA o CERRADA **(1)**
    - No hacer nada.
  - SINO //menor o igual
    - SI la estimación de 's' es mejor (menor o igual) que la de 't' (el antiguo) en ABIERTA o CERRADA (en el grafo):
      - Sustituir el nuevo h(s) por el antiguo h(t)
      - Cambiar el puntero. Hacer que su padre sea 't'
      - SI el estado está en CERRADA
        - **a.** Actualizar el valor de la fev para sus descendientes **(2)**
    - //Fin si la estimación de 's' es mejor que el de 't'
  - //fin SINO es mayor al h(s) que tenía
- //FIN para los miembros repetidos

- //FIN expandir 't'
- Reordenar la lista ABIERTA, según el valor de fev

- //FIN hasta que ABIERTA vacía o meta
- Si se llega a la meta, dar la solución siguiendo los punteros hasta el estado inicial, y si no hay meta dar mensaje de error.

(1) Se mejora el rendimiento si se evalúa h(s) como mayor o igual que el valor que ya tenía el estado en ABIERTA o CERRADA.

(2) El coste de actualizar la fev de los descendientes siempre será menor que volver a generar el estado. Así, como el grafo de exploración es finito, propagar los nuevos valores de fev es también finito. Una vez propagados, el árbol implícito hay que ajustarlo para ver si los caminos son los correctos. Hay autores que discuten la conveniencia de propagar y comprobar, proponiendo volver a pasar el nodo a ABIERTA y generar, si procede en pasos sucesivos, sus descendientes con los nuevos datos.

4.4.2. Análisis del algoritmo

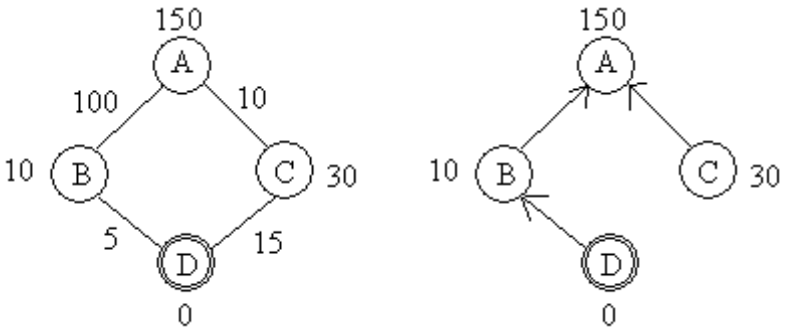
Este algoritmo soluciona el problema de los mínimos locales y de las mesetas. Si se llega a una de estas situaciones se permite que se continúe con la búsqueda, y como no se

permite la repetición de estados (ausencia de bucles), finalmente saldrá de ellas. En el mínimo local se permite seguir por el camino más prometedor siguiente y en las mesetas se sale después de generar todos los estados de esta (esto supone un coste importante).

- **Completo.-** Si el estado es meta, su fev es cero. Luego, por muy larga que sea la rama, siempre encontrará otra rama que tenga mejor fev y finalmente encontrará la meta, si existe; por lo tanto es completo.

- **Óptimo.-** No es óptimo, pues la fev solo estima la cercanía a la meta y no se tiene en cuenta el coste del camino ya recorrido. Se puede abandonar una rama cuya fev estima que está más lejos de la meta que otra, pero que el coste total del camino de la raíz a la meta (el recorrido más el estimado) sea inferior a esta segunda.

  Supongamos el problema representado por el grafo de la derecha, donde los arcos y los nodos se etiquetan con sus costes asociados. Como podemos ver, el camino menos costoso de la raíz a la meta es A-C-D. En el grafo de exploración (derecha) al expandir la raíz A se generan los nodos B y C. Por este algoritmo el siguiente nodo a expandir es B, pues el que tiene estimado un camino menos costoso a la meta. Si lo que buscamos es el camino solución menos costoso, el siguiente nodo a expandir debería ser C. Esto se podría hacer si el algoritmo tuviese en cuenta el camino recorrido hasta B y C (luego vemos como solucionarlo).



- **Coste computacional.-** En el caso peor tendrá el mismo coste que la búsqueda en profundidad, ya que la solución podrá estar en el último estado de la última rama. Está en O (n^p). En promedio, el comportamiento, gobernado por la fev, puede ser mucho mejor.

- **Coste espacial.**- Dado que entre ABIERTA y CERRADA tenemos almacenado todo el grafo de exploración, en el último estado de la última rama tendremos todos los estados almacenados; luego el coste[26] estará en O (n^p). Al igual que el coste computacional, en promedio, su comportamiento puede ser mucho mejor dependiendo de la calidad de la fev seleccionada.

4.4.3. Conclusiones

Su principal ventaja es que es completo, a pesar de ser una exploración en profundidad, superando el problema de los mínimos locales y las mesetas que tenía el método en escalada. Su principal inconveniente está en la fev, ya que tiene en cuenta lo cerca que está el estado de la meta, pero sin tener en cuenta coste del camino ya recorrido, con lo que la solución encontrada puede no ser la más óptima.

4.5. BÚSQUEDA EN HAZ

Variación del método PM, en el que vamos a seguir teniendo una fev en el mismo sentido que en PM (distancia a la meta). En lugar de considerar un solo estado como candidato, se amplía a varios estados los que se consideran simultáneamente candidatos al mejor camino; a este conjunto se le denomina FRONTERA del haz.

Para determinar la frontera del haz tenemos una segunda función, que evalúa y determina, normalmente según una estimación heurística, el número de estados que van a formarlo. Se puede fijar de dos maneras:

- Estableciendo un número fijo 'k' de estados (no es heurística).

- Estableciendo otra función que determine un valor umbral f_0 de la fev, por debajo del cual los estados no pueden pertenecer a la frontera del haz. Este f_0 determina en cada paso los 'k' estados más prometedores.

[26] Ver algoritmo de búsqueda en profundidad para el nivel simbólico

Además de CERRADA, vamos a tener dos estructuras más, que son el resultado de dividir ABIERTA en dos:

- FRONTERA.- Contendrá los estados que pertenecen al haz actual.

- SUCESORES.- Contendrá los posibles sucesores de la frontera. Son todos los generados a partir de los estados del haz.

Al haz, nodos de FRONTERA, se le aplica el algoritmo PM. Una vez obtenidos los sucesores del haz (de todos los nodos de FRONTERA), se eligen los más prometedores, según el criterio elegido (número fijo o umbral), y se eliminan los demás.

4.5.1. Algoritmo de búsqueda en haz

- Crear un grafo de búsqueda G, que solo va a tener la raíz con el estado inicial del problema.
- Crear FRONTERA e incluir el nodo raíz, con su valor de fev.
- Crear SUCESORES y CERRADA, que en el momento inicial estarán vacías.
- HASTA que SUCESORES Y FRONTERA estén vacías o se llegue a la META
    - HASTA que FRONTERA esté vacía o META
        - Extraer (eliminar) de FRONTERA el primer nodo de la lista, y asignarlo a 't', y a su vez lo copiaremos en la siguiente posición de CERRADA.
        - Expandir 't':
            - Generar el conjunto S de todos los sucesores 's' de 't', que a su vez no sean antepasados de sí mismo en G y asociarle a cada uno su h(s) e introducirlos en G como sucesores de 't'.
            - Si algún miembro de S es META terminar con la expansión.
            - Para los miembros de S que no están en FRONTERA, SUCESORES ni CERRADA, añadirlos a SUCESORES junto con el valor h(s), y asignarle 't' como su predecesor.
            - Para los miembros 's' de S que estén en FRONTERA, SUCESORES o CERRADA (repetido):
                - SI h(s) es mayor que el que ya tenía el estado en G
                    - No hacer nada.
                - SINO //menor o igual
                    - SI h(s) es mejor (menor o igual) que la estimación en el estado del grafo G:
                        a. Sustituir el nuevo h(s) por el antiguo h(t)
                        b. Cambiar el puntero. Hacer que su padre sea 't'

c. SI el estado está en CERRADA
    - Actualizar el valor de fev para sus descendientes
        - //Fin si el camino de 's' es mejor que el de 't'
    - //fin SINO es mayor al h(s) que tenía
- //FIN para estados repetidos
- //FIN expandir 't'
- Reordenar la lista SUCESORES, según el valor de fev
- //fin HASTA que SUCESORES vacía o META
- Copiar en FRONTERA los estados de sucesores que son más prometedores **(1)**

- //FIN hasta que SUCESORES Y FRONTERA vacías o META
- Si se llega a la meta, dar la solución siguiendo los punteros hasta el estado inicial, y si no hay meta dar mensaje de error.

(1) Según el criterio de poda elegido, se pueden elegir los k nodos con mejor valor de fev (si k es fijo), o elegir los k nodos con valores de fev que estén por encima del umbral calculado (función de criterios heurísticos).

4.5.2. Análisis del algoritmo

- **Completo.-** Este algoritmo no es completo, ya que con la eliminación definitiva de caminos que en un momento dado no son prometedores, se puede estar eliminando la o las metas.

- **Óptimo.-** No es óptimo por los mismos motivos que no lo era el PM original (no tiene en cuenta el camino recorrido).

- **Complejidad temporal.-** A costa de no ser completo del algoritmo PM, este coste es mucho mejor, ya que no depende del factor de ramificación, sino del valor medio de 'k', que es bastante menor que p. Así el coste estará en $O(n^k)$.

- **Complejidad espacial.-** Dado que k será bastante menor que p, se reduce el número de estados necesario a almacenar, que serán 'k' en cada nivel, luego está en $O (n^k)$.

4.5.3. Conclusiones

La principal ventaja de este algoritmo es que se reduce apreciablemente el coste, aunque sea en detrimento de no ser completo. El problema de este algoritmo radica en la buena elección del valor de k. Si es demasiado restrictivo (k=1), resulta que la búsqueda se convierte en irrevocable, y si es demasiado general, se pueden llegar a expandir todos los estados a la vez, convirtiéndose en una búsqueda ciega en amplitud. Si la función para el valor de umbral está lo suficientemente informada, se reducirá la posibilidad de eliminar una posible meta, además de reducir apreciablemente el grafo de expansión.

Hay autores que llaman a este algoritmo búsqueda de haz en anchura (bread first beam search). Una variación a este es que en cada paso, solo se expansiona el primer estado de FRONTERA, y luego se eligen los 'k' estados más prometedores de los no expansionados en SUCESORES y FRONTERA, pasándolos todos a FRONTERA. A esta variación se la llaman búsqueda en haz primero el mejor (best first beam search).

Se aplica con éxito en problemas en los que el espacio de estados es muy grande: reconocimiento de voz, visión artificial y aprendizaje.

4.6. BÚSQUEDA A*

Este algoritmo es también otra mejora del PM, en el que se soluciona el problema que tiene el no considerar el coste del camino ya recorrido desde el estado inicial hasta el que se va a expansionar. En este, la fev no va a ser un estimador de cercanía a la meta sino un estimador del coste total del camino hasta la meta. Así, el **objetivo de A*** es **encontrar el camino óptimo** (más corto) desde el estado inicial a una META, **expandiendo el menor número de estados posible** (para ello la fev ha de cumplir unas propiedades). Para un estado n cualquiera, la fev, que llamamos f(n), se calcula sobre el coste del camino real recorrido desde la raíz hasta n más la estimación del coste desde n hasta la meta.

Recordar que el coste de un camino entre dos nodos es la suma de los costes de todos los arcos que llevan desde un nodo a otro. Si n_a y n_b son dos nodos, el coste de n_a a n_b lo denotamos como $C(n_a , n_b)$; este va a ser siempre positivo.

Llamemos h(n) a la estimación del coste de llegar a una meta desde n (distancia a la meta) por el camino óptimo (menos costoso). La función h(n) se calcula con la información que se tiene hasta ese momento en la búsqueda. Para cada uno de los caminos reales que hay desde n a un nodo meta y para todos los nodos meta, habrá uno que será el camino real de menor coste que llamaremos h* (n). Llamemos g(n) al coste del camino real llevado hasta n en el proceso de búsqueda. Para cada uno de los caminos existentes desde el nodo inicial al nodo n, uno de ellos será el de menor coste, que denominaremos g* (n). Ahora ya podemos componer nuestra fev como

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

donde f(n) es el estimador del camino de menor coste desde el estado inicial hasta un nodo meta pasando por n. También podemos componer

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

donde f*(n) es el coste real del camino óptimo entre el nodo inicial y la meta de mejor coste, pasando por n. En este caso f*(n) es constante para cualquier estado. Podemos decir que f(n) es un estimador de f*(n), ya que cualquier estado que se salga del camino óptimo lo que hace es aumentar el coste.

Si a lo largo de todo el camino se diera que f(n) = f*(n), es decir, que en todo el camino g(n) = g*(n) y h(n)= h*(n), entonces estamos ante una búsqueda perfecta (el caso ideal), es decir, se habría dado una ejecución secuencial de operadores. Esto es precisamente lo que estamos buscando, conocer f*(n). Dado que, en un momento de la búsqueda, g(n) es el mejor camino conocido hasta ese momento, se deduce que lo difícil es encontrar un h(n) que estime lo más exactamente posible h*(n).

Se dice que h(n) es **admisible,** y por ende que **A* es admisible**, si la estimación del coste es optimista, esto es, que su valor estimado nunca supera al coste real[27] (estimado mejor que el real). Así

$$\text{Para todo estado n,} \quad h(n) \leq h^*(n)$$

De la definición de la fev, se deduce que, cuanto más cerca esté h(n) de h*(n), más eficiente será la búsqueda, ya que se generarán menos estados. Así, si tenemos, para el mismo problema, dos funciones $h_1(n)$ y $h_2(n)$, decimos que $h_1(n)$ está más informada que $h_2(n)$ si ambas son admisibles y $h_1(n)$ está más cerca de h*(n). Formalmente, se debe cumplir

$$\text{Para todo estado n, } h_2(n) \leq h_1(n) \leq h^*(n)$$

Una consecuencia directa de esto es que el número de nodos expandidos por $f_1(n)$ será menor o igual que el número de nodos expandidos por $f_2(n)$; es decir, $f_2(n)$ expandirá los nodos de $f_1(n)$ y algunos más. Por asociación de lo anterior, se dice que la búsqueda A* está más informada si así lo está su fev. No obstante, no hay que perder de vista que hay que llegar a un compromiso entre el coste de la fev más informada y el beneficio que nos puede aportar.

Volvamos al caso ideal. Si i es el estado inicial, en este se cumple que

$$g^*(n) + h^*(n) = h^*(i) \text{ , ya que } g(i) = 0$$

También en el caso ideal, para cualquier nodo n se cumple que

$$g^*(n) + h^*(n) = f^*(n) = h^*(i) \text{ , ya que es una constante}$$

En un caso real, el camino encontrado para un nodo va a tener un coste superior que el ideal, así, suponiendo que h(n) es admisible y que h(i) = h*(i),

$$\mathbf{h(i) \leq \ g(n) + h(n)} \quad \text{(la inecuación se cumple igualmente para } h^*(n))$$

[27] Nilsson y otros fijaron tres condiciones: Además de la expuesta, otras dos que hemos dado por supuestas: cada estado tiene un número finito de sucesores y el coste entre nodos es siempre positivo.

Por la definición de la fev, podemos decir que

$$h(i) \leq [g(n)- g(i)] + h(n) \text{ ya que } g(i) \text{ siempre es cero}$$

Se dice que h(n) es **monótona si cumple la anterior inecuación para todos los nodos del grafo**. Esta se puede particularizar para un nodo n con su sucesor s, resultando

$$h(n) \leq [g(s)- g(n)] + h(s) \text{ siendo } [g(s)- g(n)] \text{ el coste } C(n,s), \text{ luego}$$

$$\mathbf{h(n) \leq C(n,s) + h(s)}$$

que significa que la estimación del coste a la meta de un nodo siempre será menor o igual que el coste estimado a través de alguno de sus sucesores. Se dice que h(n) es consistente si se cumple esta última ecuación para cualquier estado del grafo.

Como vemos, decir que una h(n) es consistente es equivalente a decir que es monótona (se puede demostrar por inducción) y que, dado que hemos partido de la premisa de que h(n) es admisible, una fev que sea consistente y monótona, también es admisible. Una consecuencia inmediata de que h(n) sea consistente es que f(n) va a ser no decreciente a lo largo de toda la búsqueda.

*4.6.1. Algoritmo A**

El algoritmo es el mismo que el PM con las modificaciones propias de la fev de coste óptimo. Con esta fev tenemos varias situaciones. Si suponemos i el estado inicial y m un elemento del conjunto M, de METAS posibles:

- En el estado inicial: $f^*(i) = h^*(i) \rightarrow f(i) = h(i)$, ya que $g(i) = 0$
- En cualquier nodo intermedio: $f^*(n) = g^*(n) + h^*(n) \rightarrow f(n) = g(n) + h(n)$
- En alguna meta: $f^*(n) = g^*(n) \rightarrow f(n) = g(n)$, ya que h(n) es cero 0 en cualquier meta.

Dentro de las estructuras ABIERTA y CERRADA se almacenará cada estado junto con los valores de f(n) y de g(n) y el apuntador a su nodo padre. CERRADA es imprescindible para comprobar que no se repitan estados en el grafo de exploración.

- Crear un grafo de búsqueda G, que solo va a tener la raíz con el estado inicial del problema.
- Crear ABIERTA e incluir el nodo raíz, con su valor de fev **(1)**
- Crear CERRADA que en el momento inicial estará vacía.
- HASTA que ABIERTA esté vacía o se llegue a la META
    - Extraer (eliminar) de ABIERTA el primer nodo de la lista (valor mínimo de fev), y asignarlo a 't', y a su vez lo copiaremos en la siguiente posición de CERRADA.
    - Si 't' es META, terminar con la expansión. **(2)**
    - Expandir 't':
        - Generar el conjunto S de todos los sucesores 's' de 't' que a su vez no sean antepasados de sí mismo en G y asociarle a cada uno su fev e introducirlos en G como sucesores de 't'. **(3)**
        - PARA los miembros de S que no están en ABIERTA ni CERRADA, añadirlos a ABIERTA junto con el valor de fev, y asignarle 't' como su predecesor.
        - PARA los miembros s de S que estén en ABIERTA o en CERRADA (repetido en G, que llamamos 'r'):
            - SI el coste del camino en el grafo, g(r), es menor o igual que el coste del nuevo camino, g(s) **(4)**
                - No hacer nada: ignorar 's'.
            - SINO //el nuevo camino es mejor
                - En el estado del grafo 'r':
                    - Sustituir los costes de 'r' por los de 's'. **(5)**
                    - Cambiar el puntero. Hacer que su padre sea 't'
                - SI el estado está en CERRADA, propagar los nuevos costes a sus descendientes en G: Tomando 'r' como raíz, se realiza un recorrido en profundidad sobre sus descendientes, calculando sobre los que estén en G **(6)**
                    - HASTA que g('d') sea igual al nuevo coste de 'd' o no haya más descendientes en G: **(7)**
                        a. SI 'd' tiene un camino (punteros a los sucesivos predecesores) hasta 'r' en G y g('d') es mayor del nuevo coste de 'd', actualizar el coste de 'd' a partir del coste de su predecesor en G.
                        b. Si 'd' no tiene un camino (punteros a los sucesivos predecesores) hasta 'r' en G y el nuevo coste de 'd' es menor que el coste g('d'), redireccionar el puntero al predecesor de d en G para que tenga un

camino a 'r', y actualizar el coste de 'd' a partir del coste de su predecesor en G.

- //fin HASTA
- //Fin SINO //el nuevo camino es mejor
- Fin PARA los estados repetidos
- //FIN expandir 't'
- Reordenar la lista ABIERTA, según el valor de fev

- //FIN hasta que ABIERTA vacía o meta
- Si se llega a la meta, dar la solución siguiendo los punteros hasta el estado inicial, y si no hay meta dar mensaje de error.

(1) Asignar al nodo raíz, i, el valor de fev asociado: en este caso g(i)=0 → f(i) = h(i)

(2) Modifica el PM en el sentido de que este finaliza cuando un sucesor es meta, mientras que A* finaliza cuando el estado a expansionar es meta. Esto garantiza que la solución es la óptima, ya que se expansiona la meta de mejor coste total (si hubiese más de una en ABIERTA), y no la primera que aparezca en ese mismo nivel.

(3) Sucesores de 't' nuevos en el grafo. Para un estado 's' sucesor de 't':

a. g(s) = g(t) + C (t,s)
b. f(s) = g(s) + h(s)

(4) Estamos comprobando si el camino seguido hasta ahora es mejor que el que se siguió anteriormente.

(5) Si el nuevo camino encontrado es mejor, se coge este. Para ello, si 'r' es el estado en G y 's' es el sucesor repetido

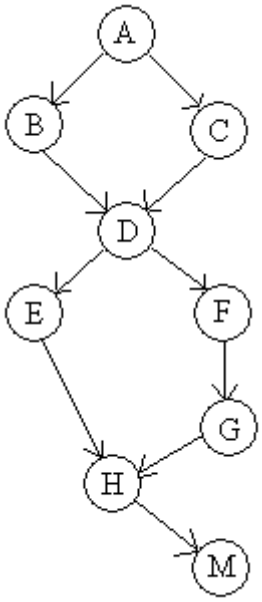
a. g(r) = g(s)
b. f(r) = g(s) + h(r) (h(r) depende solo del estado)

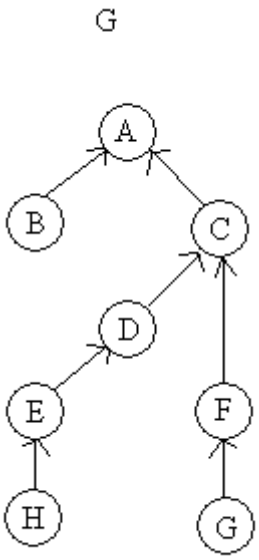
(6) Si resulta que se ha encontrado un nuevo camino menos costoso y el estado tenía descendientes, es necesario propagar el coste de este nuevo camino a sus descendientes. Existe variantes en las que, para un estado repetido en G con mejor camino que en CERRADA este simplemente se pasa de nuevo a ABIERTA, para volver a expandir sus descendientes de nuevo, cuando corresponda. Esta variante es mucho más costosa (generación de estados) que comprobar los costes de todos los

descendientes del estado repetido (estados ya generados). No obstante, toda esta comprobación no hará falta si h(n) es consistente y monótona: hemos visto que su consecuencia inmediata es que f(n) va a ser no decreciente, lo que deriva en que siempre vamos tener en G el estado con el camino óptimo a ese nodo repetido.

(7) Esta comprobación garantiza que la propagación de costes a los descendientes termina.

Veamos el punto 6 con un ejemplo: Supongamos que ciertos nodos de un problema tienen las relaciones indicadas en el grafo de la izquierda. En un momento dado el grafo de exploración G está en la situación del grafo de la derecha.





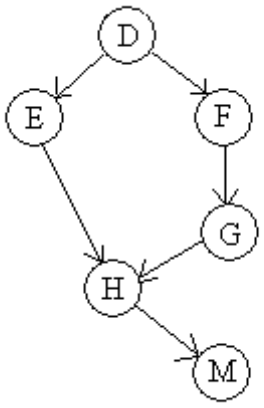
En este instante en las estructuras de datos tenemos:

- ABIERTA: B, E, H (en orden de menor a mayor, por lo que el siguiente nodo a expandir es B)

- CERRADA: A, C, D, F, G

Sucesores de B solo es D. Resulta que para D el nuevo camino abierto a través de B tiene un camino menos costoso que a través de C, por lo que cambia el puntero de C a D. Dado que D ya estaba en CERRADA, debemos comprobar si hay que propagar el coste del

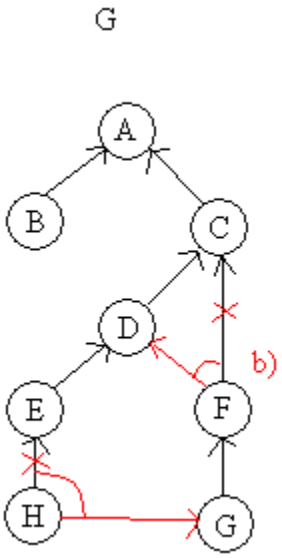
camino. Para ello cogemos D y hacemos un grafo basado en las relaciones de los nodos, cuya raíz es D. Así nos queda el siguiente subgrafo:



Supongamos que empezamos por la rama de E y H. En este caso estamos en el caso a), es decir, tenemos un puntero a la raíz. Procedemos a actualizar los costes sucesivamente, si procede.

Por la rama de F la cosa cambia. Estamos en el caso b), ya que F no tiene un puntero a D en G. Pero resulta que el camino de F a través de D es mejor, por lo que cambiaremos el puntero de C a D y luego propagar el coste de D a F. Con G procederemos como en el caso a). Hemos llegado a H. Si resulta que el camino a través de G es mejor que a través de E deberemos cambiar el puntero de E a H y propagar el coste de G a H.

Por último, en el nodo M ya hemos terminado el recorrido en profundidad, ya que no está en G. Así el grafo de exploración quedaría:



4.6.2. Análisis del algoritmo

- **Completo.-** Al ser una derivación del PM, si el problema tiene una solución, este algoritmo la encuentra siempre. La mejora introducida en la fev hace que se abandonen caminos que no están en el camino óptimo, además de estimar el coste a la meta.

- **Óptimo.-** Si h(n) es admisible, se garantiza que la solución encontrada es la óptima (la de menor coste).

- **Complejidad temporal.-** Si h(n) es consistente, se garantiza que este algoritmo es el que menos nodos expande de todos los derivados del PM. Sin embargo, si la fev no está muy informada, la búsqueda se podrá convertir en una búsqueda ciega en amplitud (ver estrategias derivadas inmediatas, a continuación).

- **Complejidad espacial.-** Al igual que sus hermanos, este algoritmo necesita almacenar todos los estados del grafo de exploración en el caso peor(exponencial con la profundidad y el factor de ramificación).

4.6.3. Conclusiones

Como ventajas tiene que, de los derivados de PM, es el algoritmo más eficiente, que además calcula la solución óptima si su fev es consistente y monótona (por tanto admisible).

Su principal desventaja es que necesita almacenar todo el grafo de exploración, lo que para problemas medianos o grandes este algoritmo no tendrá suficientes recursos (agotamiento de la memoria).

4.6.4. Estrategias derivadas inmediatas

Son consecuencia de un caso especial, que es que para todos los nodos, h(n) es 0. Podemos ver que es admisible, luego cualquier método derivado de A* con h(n) = 0 encuentra la solución óptima.

Si, además de ser h(n) = 0 para todos los nodos, el coste es uniforme entre todos ellos y sus sucesores, la búsqueda se convierte en una estrategia en amplitud vista en el capítulo anterior.

Si para todos los nodos h(n) = 0, la búsqueda se convierte en una estrategia de **coste uniforme**, que consiste en expandir primero el estado con el menor coste.

4.6.5. A con funciones de error*

Nos podemos encontrar con h(n) que sean estimadores pesimistas (estimación por encima del valor real). Si esta estimación no se desvía de un margen absoluto ε, es decir, para todos los nodos

$$h(n) - h^*(n) \leq \varepsilon \rightarrow h(n) \leq \varepsilon + h^*(n)$$

se puede decir que el coste de la solución no sobrepasa más de 1+ ε el coste óptimo de la misma. En este caso se dice que el algoritmo se llama $A^{e}*$.

Si el error admitido es relativo, α h*(n), entonces el margen de error es exponencial.

$$h(n) \leq h^*(n)\ \alpha\ h^*(n) \rightarrow h(n) \leq (1+\alpha)\ h^*(n)$$

podemos decir que el coste de la solución no sobrepasa más de (1 + α) veces h*(n). En este caso se dice que el algoritmo se llama $A^{\alpha}*$.

Se puede establecer una jerarquía de algoritmos de la forma:

$$A^* \subset A^{e}* \subset A^{\alpha}*$$

ya que las h(n) admisibles y consistentes están incluidas en h*(n) + ε y a su vez estas están incluidas en (1+α) h*(n).

4.7. A* EN PROFUNDIDAD ITERATIVA (A* -P)[28]

Dijimos que el problema principal del algoritmo A* era que necesita mucha memoria para realizar la búsqueda (coste exponencial). Para intentar solucionarlo, se realiza un planteamiento semejante a la búsqueda en haz, en el sentido de utilizar la fev para limitar el número de estados a expansionar.

Cuando vimos la búsqueda en profundidad progresiva ciega, dijimos que era óptima y completa. En él utilizábamos lp como límite de exploración en cada iteración. En A*-P vamos a utilizar la misma estrategia (no la de primero el mejor modificada para A*) pero utilizando la fev de A* para calcular el límite de profundidad en cada iteración. Hay que tener en cuenta que la fev solo se utilizará para calcular lp y no para dirigir el orden en que se generan los nodos.

Para la primera iteración, calculamos el valor de fev:

$$f(i) = g(i) + h(i), \text{ como } g(i) = 0, \text{ nos queda que } f(i) = h(i)$$

así, aplicaremos la búsqueda en profundidad hasta el valor lp = f(i) o hasta que se encuentre una meta. Si se llega al último nodo de esta iteración y no se encontró la meta, calculamos el nuevo f(n) de corte para la siguiente iteración. Para ello tenemos que saber cual es el estado m con menor valor f(m) que superó el corte f(i), es decir, el estado m con coste más pequeño de los que no se generaron. Con este nuevo límite de exploración (f(m)= lp) se vuelve a realizar la búsqueda en profundidad.

4.7.1. Algoritmo A*-P

- Inicializar el límite de exploración lp=f(i) (primer nivel) y lpSiguiente = ∞
- HASTA que se encuentre una META
  //Algoritmo de búsqueda en profundidad.
  - Crear la pila ABIERTA poner en su primer elemento el nodo raíz (descripción del problema) HASTA que ABIERTA esté vacía (error) o el estado sea META
    - Extraer (eliminar) de ABIERTA el primer nodo de la lista, y asignarlo a 't' (temporal).
      //expandimos 't'

[28] Acrónimo en inglés IDA* (iterative deepening A*)

- PARA cada operador y cada instanciación de operador aplicable a 't'
  - Aplicar el operador a 't'. Obtendremos un nuevo estado 's' (sucesor).
  - SI 'S' no es antepasado de 's' en G
    - SI f(s) no es mayor que lp:
      - A este le asignamos como padre a 't'
      - SI 's' es meta
        a. Terminar con la expansión de nodos
      - SINO
        a. incluirlo en la cima de ABIERTA.
    - SINO // f(s) está por encima: nos quedamos con el menor
      - SI f(s) < lpSiguiente hacer lpSiguiente = f(s)
  - //FIN si no está repetido en el árbol solución.
- //fin PARA cada operador

o //fin HASTA que ABIERTA está vacía (error) o el estado es META

o SI es META
- Devolver la solución

o SINO //ABIERTA está vacía = error

// Aumentar el límite de exploración (pasar al siguiente nivel)
- lp = lpSiguiente
- lpSiguiente= ∞

- FIN HASTA

4.7.2. Análisis del algoritmo

Si la fev es admisible y consistente, este algoritmo es completo y óptimo. También tiene la misma ventaja que la búsqueda en profundidad progresiva, que era una complejidad espacial de orden lineal O (np), ya que solo se necesita mantener en ABIERTA los sucesores de un estado en cada nivel.

Si la fev no es admisible, la complejidad temporal del algoritmo es muy sensible a la repetición de estados a lo largo de la búsqueda, ya que no se tiene una estructura CERRADA para comprobar este hecho.

4.8. BÚSQUEDA AO*

Esta búsqueda se utiliza en problemas encadenados hacia atrás o dirigido por las metas. Recordemos que *el grafo de exploración en este caso es del tipo Y/O* y que el problema estará resuelto cuando lo estén todos los nodos terminales. Consecuencia de esto, es que ahora no buscamos un camino óptimo, sino que **buscamos el grafo óptimo que nos de la solución** (uno o varios estados meta). Este es precisamente el objetivo del algoritmo AO*.

Esta búsqueda es una generalización del algoritmo A* para esquemas de reducción, en el sentido de que también utilizaremos un estimador del coste total del camino hasta la meta. Para ello necesitamos que la fev también sea admisible y consistente. Remarcar que ahora no buscamos un camino óptimo, sino un grafo óptimo. Al igual que en A* decíamos que si h(n) era admisible y monótona, entonces el algoritmo encontraba el camino óptimo a la meta, en AO* decimos que si h(n) es admisible y monótona, entonces el algoritmo encontrará el grafo óptimo solución. La fev h(n) es admisible si

$$\text{Para todo estado n, } h(n) \leq h^*(n)$$

Para la monotonía, si un enlace desde n tiene k sucesores y C es la suma de los costes de los arcos del enlace, se cumple que

$$h(n) \leq C + h(n_1) + \ldots. + h(n_k)$$

lo que significa que a la hora de actualizar el valor h(n) de un nodo, el valor previo no supera nunca al calculado.

En este algoritmo, h(n) va a representar la estimación del coste del grafo con la solución óptima desde ese nodo n hasta un conjunto de nodos terminales, que llamaremos T.

Se va a proporcionar una definición recursiva para el grafo solución. Para ello se va a suponer que los grafos Y/O que se van a generar no tienen ciclos, es decir, ningún estado va a tener un sucesor que a su vez sea antecesor del mismo. Dada esta restricción, será un GDA, del que sabemos que define un orden parcial entre sus nodos, y esto nos permite afirmar que la función recursiva finaliza en algún momento.

Dado un grafo de exploraciónY/O que llamamos G y que va a tener un conjunto T de nodos terminales, va a tener implícito un subgrafo Gs, que *es una solución parcial* desde la raíz hasta T. Dado un nodo n, el subgrafo Gs se define recursivamente:

- Si n es un elemento de T, entonces Gs es un grafo con exactamente un nodo, que es n. Si n está resuelto, entonces Gs es el grafo solución.

- Si n tiene un enlace con k arcos, de manera que para cada sucesor en ese enlace existe un subgrafo hasta T, entonces Gs es el nodo n, todos los sucesores del enlace y todos los subgrafos hasta T desde cada sucesor de ese enlace. El nodo n estará resuelto si lo están todos los sucesores del enlace. Si n es el nodo raíz, Gs es el grafo solución.

- Si no se cumple una de las dos anteriores condiciones, entonces no existe una solución para G.

Para el cálculo de h(n) en cada nodo, también utilizaremos una definición recursiva. Aquí el coste de un nodo será el del grafo, C(n,T):

- Si n es un elemento de T, entonces C(n,T) = 0

- Si n tiene un enlace con k arcos, el coste de cada uno de los k sucesores será el coste del subgrafo solución hasta T desde ese sucesor, luego, si C es el coste de los k arcos del enlace, el coste de n será:

$$h(n) = C + h(n_1) + \ldots. + h(n_k)$$

Como en este cálculo se suman los costes hasta la solución *para cada sucesor* y dado que es un grafo, se puede dar el caso de que un mismo nodo tenga distintos antepasados; por tanto nos podemos encontrar que, para este cálculo, consideremos el coste de un mismo nodo en dos sucesores de n distintos: se está sumando más de una vez.

En esta estrategia vamos a tener un grafo de exploración G, y como ya se ha dicho, implícito en él habrá un grafo Gs que llamamos *solución parcial*. Lo que va a hacer es

explorar Gs hasta llegar a algún nodo terminal[29] que no esté solucionado (no importa la profundidad a la que esté); expande uno de ellos, generando todos sus sucesores con los enlaces correspondientes. Finalmente, calculará el coste de ese nodo a partir de los costes obtenidos en sus sucesores. Si todos los nodos sucesores de un enlace están solucionados, el nodo también lo estará. El algoritmo terminará cuando el nodo raíz esté solucionado, siendo Gs el grafo solución.

Según vayamos creando el grafo de exploración G, vamos a ir marcando, en cada paso, el grafo solución parcial, Gs (estas marcas van a cumplir el mismo cometido que hacía la lista ABIERTA en los otros métodos). Vamos a tener también un valor *coste_maximo*, por encima del cual se supone que el grafo no tendrá solución. Las marcas en Gs se pueden hacer, dibujando los arcos de G con trazos discontinuos y los de Gs con trazos continuos o dibujando los arcos de G con trazos continuos y los de Gs con trazos gruesos.

*4.8.1. Algoritmo AO**

- Crear un grafo de exploración G, que solo va a tener la raíz con la meta global (el problema) que llamamos p. Asignarle su fev h(p).
- SI p no tiene sucesores
  - Marcar p como RESUELTO.
- SINO
  - HASTA que p no esté RESUELTO o tenga un valor igual o superior a coste_maximo
    - Recorrer G', esto es, seguir los enlaces marcados en G, hasta los nodos terminales que todavía no está marcado como RESUELTO. **(1)**
    - Escoger uno de ellos y llamarlo 't'

    //expandir 't' = aplicar todos los enlaces posibles para 't'
    - SI 't' no tiene sucesores
      - h(t) = coste_maximo
    - SINO
      - Añadir a G todos los sucesores de 't' con sus respectivos enlaces.
      - Para cada sucesor 's' de 't'
        - SI 's' se considera RESUELTO, marcarlo como tal y asignarle h(s)=0;
        - SINO asignarle el h(s) que corresponda.
    - //fin si tiene sucesores

[29] Por claridad se ha tomado el convenio de tomar como sinónimos hoja, extremo y terminal, pudiendo estar estos resueltos o no resueltos. En [Nilsson 1980] y [Fernández el al 1998], donde se describe este algoritmo, se distingue entre nodo hoja y nodo terminal, diciendo que un nodo terminal es un nodo resuelto, luego hay "nodos hoja terminales".

//propagar costes **(2)**

- Crear un conjunto R que inicialmente contendrá a 't'
- HASTA que R esté vacío
  - Extraer (eliminar) de R un nodo 'r' que no tenga descendientes en G que pertenezcan a R **(3)**
  - Calcular el coste de cada enlace que sale de 'r'. **(4)**
  - Asignar a 'r', de entre sus enlaces, el coste estimado de aquel con menor valor estimado.
  - SI alguno de los enlaces de 'r' que no se han elegido estuviera marcado como mejor solución parcial en G', eliminar la marca.
  - Marcar el enlace de 'r' con menor valor estimado como mejor solución parcial en G'.
  - SI todos los nodos del enlace elegido están marcados como RESUELTO, marcar 'r' como RESUELTO.
  - SI h('r') ha cambiado o 'r' se marcó como RESUELTO
    - Añadir a R todos los predecesores de 'r'
- //FIN hasta que R esté vacío

- //FIN hasta que p (RESUELTO) o h(p) ≥ a coste_maximo.

- //FIN si raíz no tiene sucesores
- SI h(p) ≥ coste_maximo
  - Error: no hay solución
- SINO
  - Gs= G', grafo solución óptimo **(5)**

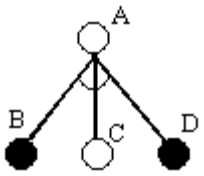
(1) El nivel al que se encuentre es indiferente. Se podría elegir el nodo terminal con mayor estimación de h(n), ya que será el que más cambios produzca en G', y por tanto sea más fácil encontrar el grafo solución óptimo.

(2) Se van a propagar hacia arriba los costes de los nodos expandidos.

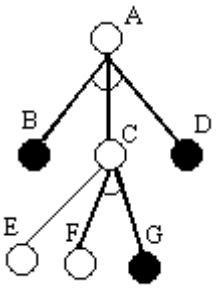
(3) Con esto se evita propagar el coste a un nodo sin haber propagado los costes de todos sus descendientes.

(4) Si un enlace tiene k arcos, su coste será $Ce= C_1+h(1) + \ldots. + C_k+h(k)$, donde C_i (para i entre 1 y k) es el coste de la arista dirigida al sucesor i, y h(i) es el coste estimado del sucesor i.

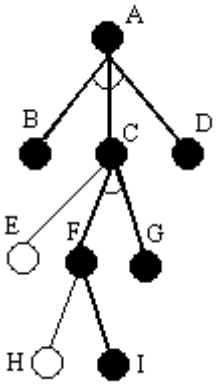
Para aclarar conceptos, se muestra un ejemplo: Supongamos que tenemos el grafo de exploración Y/O en la siguiente situación:



El nodo raíz solo tiene un enlace, luego el problema estará resuelto cuando lo estén todos los sucesores de este enlace. En este caso solo se puede expandir C. Al expandir C se obtienen dos enlaces, uno con un solo arco que genera E, y el otro con dos arcos que generan F y G. Dado que el enlace con F y G es menos costoso que el de E, este se convierte en solución parcial, y su coste se propaga a C, que a su vez se propaga a A.



En el siguiente paso, recorremos Gs hasta los nodos terminales y nos encontramos con F y G. Como G está resuelto, solo nos falta resolver F, así que es el que expandimos. Este va a tener dos enlaces cada uno con un arco que generan respectivamente los nodos H e I. Como es I el que tiene un menor coste, este se propaga a F, y como además está resuelto, F también estará resuelto. El coste de F y su resolución se propaga a C, que hace que el raíz A esté resuelto también.



4.8.2. Análisis del algoritmo

- **Completo.-** Este algoritmo encontrará la solución si esta existe.

- **Óptimo.-** El grafo solución obtenido es óptimo siempre y cuando h(n) será admisible y monótona. En este caso, en (4), el valor de h(r) antes de propagar, nunca será superior al que se ha calculado.

- **Complejidad temporal.-** En el caso peor de que h(n)=0, la búsqueda se convierte en amplitud, luego en el caso peor su complejidad será exponencial.

- **Complejidad espacial**.- La necesidad de almacenar el grafo de exploración completo, determina que, en el caso peor, esta también sea exponencial.

4.9. BÚSQUEDA CON ADVERSARIOS

Hasta ahora se han visto estrategias en las que un solo agente busca la solución a un problema. Existen otro tipo de problemas en los que dos agentes compiten por un mismo objetivo. Las búsquedas con adversarios son parte de la teoría matemática de juegos, y para la I.A. nos vamos a centrar en los de un tipo determinado:

- Hay dos contrincantes, que juegan alternadamente, uno cada vez.

- Es simétrico, esto es, los contrincantes conocen todas las reglas del juego, que son iguales para los dos.

- Es determinista, esto es, son movimientos razonados, el azar no interviene.

- Perfectamente informado, esto es, la situación es observable por ambos jugadores, no hay ocultación de movimientos.

- Cada uno de los dos jugadores puede perder, ganar o quedar en empate.

Ejemplos de juegos que no son deterministas son los de naipes; juegos no simétricos son los de policías y ladrones; juego no informado completamente es el Stratego.

Como ejemplos de juegos que utilizan estos métodos tenemos el ajedrez, tres en raya, Otello, nim (montón de palillos). Una característica común a todos estos es que la calidad de una acción individual para llegar a la victoria es medible de manera objetiva. Para estos juegos vamos a considerar un **movimiento** como un par de acciones individuales, una por cada contrincante. A estas acciones individuales, la denominaremos **medio movimiento o jugada**[30].

4.9.1. Procedimiento MINIMAX

Lo que se trata es de saber cuál es el mejor movimiento a efectuar, pero teniendo en cuenta toda la dinámica del juego (todo lo que ocurrirá si se hace ese movimiento).

Vamos a considerar que tenemos dos jugadores MAX y MIN (MAXimalista y MINimalista) y que cada uno de ellos, en su turno, obviamente va a realizar siempre la mejor jugada posible. En una primera aproximación podemos pensar en generar un grafo completo del juego, el explícito, y buscar en él un grafo solución, como hacíamos en el algoritmo AO*, y así poder elegir el mejor movimiento siguiente.

Para construir el grafo del juego, cada estado es una configuración del juego (en el ajedrez sería cada una de las posiciones válidas de las piezas en el tablero) y cada arco es la aplicación de cada acción individual (jugada), con arreglo a las reglas del juego, que es la que determina la transformación de cada configuración (en ajedrez, una jugada tiene como resultado una posición distinta de las piezas del tablero). Dado que juegan alternadamente, cada profundidad del grafo corresponderá con uno de los dos jugadores, es decir, a profundidad n son estados desde los que mueve MAX y a profundidad n+1 son estados desde los que mueve MIN. Cada uno de los nodos terminales representa una configuración en la que se gana, pierde o empata (no tienen porqué estar al mismo nivel).

Vamos a tener dos tipos de enlaces: Los enlaces O que representan cada uno de las posibles jugadas de MAX y los enlaces Y que representan todas las jugadas posibles de MIN a partir de la configuración obtenida después de una jugada de MAX.

[30] En ajedrez una jugada es lo que aquí llamamos movimiento (un adversario y la reacción del contrario)

En este grafo completo lo que *buscamos es una estrategia ganadora* para uno de los dos jugadores, es decir, un grafo solución en el que, haga lo que haga un jugador, el otro siempre gane. Por convenio vamos a determinar que buscamos una estrategia ganadora para MAX, que además será el primero en jugar; la raíz corresponde a este jugador y los nodos terminales del grafo solución representan configuraciones con las que MIN pierde la partida (ya no puede mover). Durante la búsqueda de esta estrategia ganadora para MAX tenemos dos situaciones: Si le toca mover a MAX, una jugada ganadora será cualquier enlace O que le conduzca a un nodo terminal ganador y si le toca mover a MIN, una jugada ganadora para MAX será aquella en la que un enlace Y le conduce a un terminal ganador (todos los nodos del enlace conducen a un terminal ganador).

Esta primera aproximación solo es válida para juegos con un número de configuraciones pequeño. Por ejemplo, las tres en raya tiene solo 9! configuraciones posibles. Para otros muchos juegos, el número de configuraciones posibles es intratable, con lo que la búsqueda en grafos Y/O no es aplicable; para juegos normales como el ajedrez, con una estimación entre 10^{120} y 10^{160} combinaciones posibles, la generación del grafo explícito tardaría siglos en formarse.

El procedimiento que se va a describir, que utiliza los conceptos antes descritos, es válido tanto para juegos en los que es posible generar el grafo del juego, como los que no. De momento vamos a considerar que podemos generar el grafo del juego.

Se va a *evaluar el juego de forma global* y para ello vamos a utilizar una *fev*, que llamamos h(n), que va a evaluar de manera *estática* la configuración actual. Ahora no se estiman costes, sino lo buena que es esta configuración para llegar a la victoria. Dado que lo bueno que sea una jugada depende de la evolución de la partida, resulta que el valor de esta *evaluación estática dependerá del valor de h(n) asociado a algunos de sus sucesores*. Por lo tanto, para poder evaluar una jugada, tendremos que **evaluar los nodos terminales y propagar hacia arriba sus valores asociados de h(n),** siempre considerando que cada jugador va a realizar la mejor jugada posible.

Al ser h(n) obtenida por la evaluación global, va a estar influenciada por los dos jugadores. Para encontrar la estrategia ganadora lo que se hace es que, si es el turno de MAX, se elige la siguiente mejor jugada, de todas las posibles, con arreglo al mejor valor de h(n) y si el turno es de MIN, se elige la mejor jugada para este, de todas las posibles, con

arreglo al menor valor de h(n); es decir, la jugada de uno intenta contrarrestar la del otro. Debido a esto se puede demostrar que, cuanto peor lo haga uno de sus contrincantes, mejor lo hará el otro.

Vamos a considerar que h(n)=1 si la jugada es ganadora para MAX y h(n) = -1 si la jugada es ganadora para MIN. Se considera este método como de suma 0, ya que si los dos son infalibles (la estrategia de ambos es óptima) el valor final será 0 (empate). Aunque se utilicen para esta explicación los valores 1 y -1, lo normal será un rango de valores más amplio, o incluso un rango solo positivo con un valor prefijado de empate, ya que en los juegos se pueden dar múltiples situaciones, y por tanto, es difícil reducir las estimaciones a tan solo dos valores.

Hemos dicho que se va a propagar de abajo a arriba los valores h(n) para saber cuál es la mejor jugada. Esto se hará por niveles:

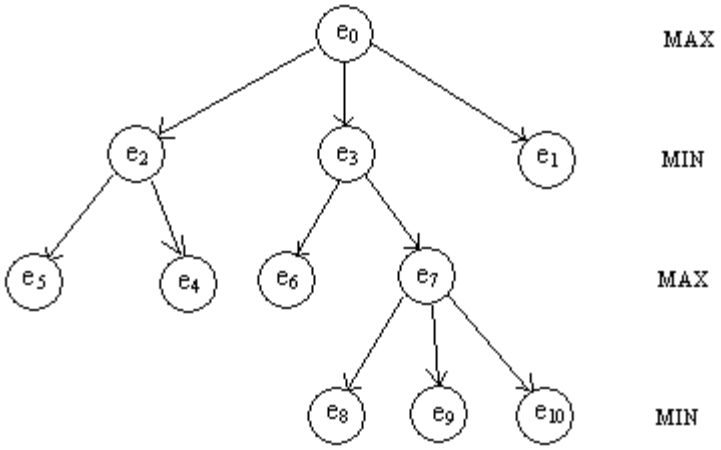
- Si el nodo es terminal, asociarle h(n).

- Si el nodo no es terminal se pueden dar dos situaciones:

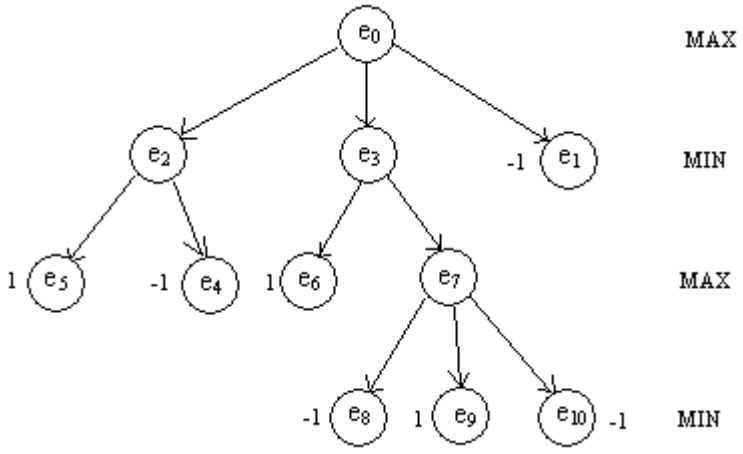
    - Que sea una configuración desde la que mueve MAX, entonces se le asocia el valor máximo de todos sus sucesores.

    - Que sea una configuración desde la que mueve MIN, entonces se le asocia el valor mínimo de todos sus sucesores.

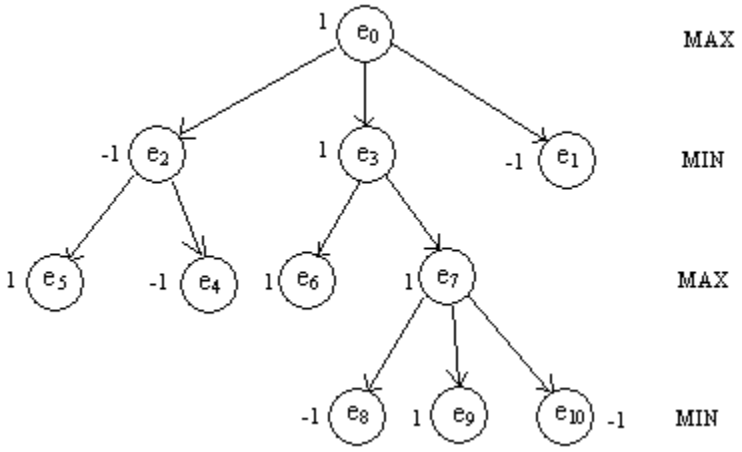
Ejemplo 1 : Supongamos que el grafo del juego es el de la figura siguiente, y lo que buscamos es una estrategia ganadora para MAX.



Lo primero que hacemos es etiquetar los nodos terminales, con sus valores h(n).



A profundidad 2 solo queda e_7 para etiquetar. Como es una jugada MAX, y uno de sus sucesores está etiquetado con un 1, este también se etiqueta con 1 (el máximo). Ahora también podemos etiquetar e_3. Es un movimiento MIN, pero como sus dos sucesores están etiquetados con un 1, se etiqueta con 1. En el nivel de e_3 tenemos que etiquetar e_2: en este caso, como es una jugada MIN, este se etiqueta con el menor de los dos, -1. Ahora ya tenemos etiquetados los sucesores de e_0, que, al ser un movimiento MAX, sabemos que el movimiento hacia e_3 es ganador.



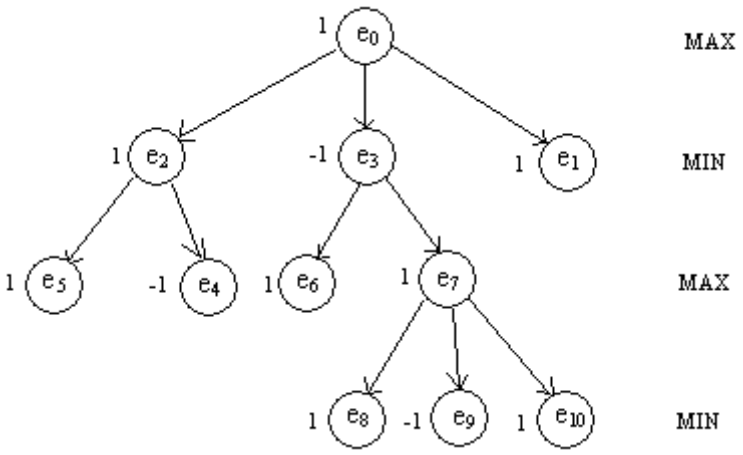
Como se ha dicho antes, generar todo el grafo del juego es imposible por tener, por regla general, un conjunto de configuraciones intratable. En estos casos lo que se hace es prever el mayor número de jugadas posibles para decidir cuál es la mejor a realizar. En estos casos, lo que se hace es generar el grafo de juego hasta una profundidad para determinar lo bueno que es el siguiente movimiento. Esta profundidad estará calculada en base a múltiples factores y que se considerará adecuada a los recursos disponibles y al nivel de juego requerido. Al conjunto de nodos terminales de esa profundidad máxima lo llamamos **frontera de exploración** o simplemente **frontera**. Para obtener la mejor jugada se propaga hacia arriba el valor de h(n) para cada nodo terminal de este grafo parcial, considerando este resultado como si fuese el de un nodo terminal del grafo completo.

Para simplificar el etiquetado de los nodos y no tener que saber si hay que propagar el menor o el mayor valor de h(n) dependiendo de si es MAX o MIN, podemos aplicar una **variación**, que llamaremos **MMvalor**, en la que el valor propagado indica lo buena que es la situación para el jugador que corresponde al nodo que se está evaluando. Se procederá al propagado de los valores de la siguiente manera:

- Si el nodo es terminal:
    - Si es un nodo MAX, se le etiqueta con h(n)
    - Si es un nodo MIN, se le etiqueta con –h(n)

- Si el nodo no es terminal, se le asigna el menor valor asociado a sus sucesores, que es, en consecuencia, la situación más desfavorable del contrario. Después se cambia de signo, pues es la más favorable para el jugador en turno.

Ejemplo 2: Aplicando este método al grafo del ejemplo anterior, vemos que los nodos terminales e_1, e_8, e_9 y e_{10} se etiquetan con –h(n), mientras que e_4, e_5 y e_6 se etiquetan con h(n). Podemos ver que e_9 no es bueno para MIN (-1), mientras que e_8 y e_{10} sí lo son (1).



Para el nodo e_7 se elegirá el menor de los valores de sus sucesores, -1, cambiado de signo, luego se etiqueta con 1. Para el nodo e_3 solo hay el valor 1, que, cambiado de signo, es la etiqueta -1. Igualmente para e_2 se elegirá el menor de los valores de sus sucesores, -1 y se cambiará de signo, luego se etiqueta con 1. Finalmente, para el nodo e_0, se elige el valor mínimo, -1, y se cambia de signo, luego se etiqueta con 1.

La estrategia ganadora elegirá siempre el sucesor que tenga el peor valor para el contrincante, en este caso e_3.

--

4.9.2. Algoritmo del método MINIMAX. Etiquetado MAX-MIN

Este método realiza una exploración exhaustiva del grafo del juego hasta el final o hasta el límite de exploración Se va a utilizar una búsqueda con retroceso (efectivamente, el método **MINIMAX es un método exhaustivo**), ya que es la que menos complejidad espacial tiene, lineal con la profundidad. En el ejemplo hemos visto que h(n) oscilaba entre 1 y -1. Estos algoritmos son más generales, oscilando entre un valor MAX_FEV y

MIN_FEV, que es el rango de h(n). Por sencillez, normalmente se inicializan con los valores +∞ y -∞ respectivamente. El procedimiento MAXIMALISTA nos va a dar el valor asociado de un nodo MAX (desde un nodo MIN), y el procedimiento MINIMALISTA, nos va a dar el valor asociado de un nodo MIN (desde un nodo MAX).

La llamada inicial se realizará a MINIMAX (raíz) y devolverá la etiqueta de la rama con la mejor jugada, que será la mayor de todos sus sucesores MIN.

- Procedimiento MINIMAX (raíz) devuelve ganador
    - ganador = MIN_FEV
    - Generar el conjunto S de sucesores 's' de 'raíz' que no sea 'raíz'.
      PARA cada elemento 's' de S
        - etiquetaHijo = MINIMALISTA ('s',profundidad+1)
        - SI ganador < etiquetaHijo //nos quedamos con el mayor valor**(2)**
            - ganador=etiquetaHijo
    - //FIN para cada sucesor
    - Devolver ganador
- //FIN procedimiento MINIMAX

//Valor de nodo sucesor MIN

- Procedimiento MINIMALISTA (nodo, profundidad) devuelve etiquetaHijo
    - SI (nodo no tiene sucesores) o (nodo está al límite de exploración)
        - Devolver etiqueta= h(nodo)
    - SINO //hay sucesores
        - minimo = MAX_FEV
        - Generar el conjunto S de sucesores 's' de 'nodo' que no sean antepasados de 's' en el grafo del juego. **(3)**
          PARA cada elemento 's' de S
            - etiquetaHijo= MAXIMALISTA('s', profundidad+1)
            - SI minimo > etiquetaHijo //nos quedamos con el menor valor**(1)**
                - minimo = etiquetaHijo
        - //FIN para cada sucesor
        - Devolver etiquetaHijo = minimo
    - //fin SINO (hay sucesores
- //fin procedimiento MINIMALISTA

//Valor del nodo sucesor MAX

- Procedimiento MAXIMALISTA (nodo, profundidad) devuelve etiquetaHijo
    - SI (nodo no tiene sucesores) o (nodo está al límite de exploración)
        - Devolver etiqueta= h(nodo)
    - SINO //hay sucesores
        - maximo = MIN_FEV

- Generar el conjunto S de sucesores 's' de 'nodo' que no sean antepasados de 's' en el grafo del juego.
  PARA cada elemento 's' de S
  - etiquetaHijo= MINIMALISTA('s', profundidad+1)
  - SI maximo < etiquetaHijo //nos quedamos con el mayor valor**(2)**
    - maximo= etiquetaHijo
- //FIN para cada sucesor
- Devolver etiquetaHijo = maximo

o //fin SINO (hay sucesores)

- //fin procedimiento MAXIMALISTA

Dada la similitud de los tres procedimientos, se puede realizar una versión más elegante, esta recursiva, del mismo algoritmo, que devuelve la etiqueta de la rama con la mejor jugada.

- Procedimiento MINIMAX (nodo, maximo, minimo, profundidad, jugador) devuelve etiquetaHijo
  - SI (nodo no tiene sucesores) O (nodo está al límite de exploración)
    - Devolver h(n)
  - SINO
    - Generar el conjunto S de sucesores 's' de 'nodo' que no sea antepasado de 's' en el grafo solución. **(3)**
    - PARA cada sucesor 's' de 'nodo'
      - SI jugador = MAX
        - etiquetaHijoMIN =
          MINIMAX (s, maximo, minimo, profundidad+1,MIN))
        - SI maximo < etiquetaHijoMIN //nos quedamos con el mayor
          - maximo = etiquetaHijoMIN
      - SINO //jugador=MIN
        - etiquetaHijoMAX=
          MINIMAX (s,maximo, minimo, profundidad+1,MAX))
        - SI minimo > etiquetaHijoMAX //nos quedamos con el menor
          - Minimo = etiquetaHijoMax
      - //fin SI jugador=max
    - //FIN para cada sucesor
    - SI jugador = MAX
      - Devolver maximo
    - SINO //jugador=MIN
      - Devolver minimo
  - //FIN si no es terminal

- //fin procedimiento MINIMAX

Llamada inicial: MINIMAX (raíz, MIN_FEV ,MAX_FEV,0,MAX):

(1) En la literatura sobre este tema, este condicional se suele resumir en
minimo = min (minimo, MAXIMALISTA('s', profundidad+1))

(2) Se suele resumir en
maximo = max (maximo, MINIMALISTA ('s', profunidad +1))

(3) Evita la formación de ciclos en el grafo.

4.9.3. Algoritmo del método MINIMAX. Etiquetado MMvalor

Igual que el anterior se va a realizar un recorrido exhaustivo en profundidad con retroceso, aunque por claridad se va a implementar solo el procedimiento recursivo.

- Procedimiento MINIMAX (nodo, profundidad, jugador) devuelve MMvalor
    - SI (nodo no tiene sucesores) O (nodo está al límite de exploración)
        - SI jugador = MAX
            - Devolver MMvalor = h(nodo)
        - SINO //jugador MIN
            - Devolver MMvalor = -h(nodo)
    - SINO //tiene sucesores a profundidad menor del límite
        - Generar el conjunto S de sucesores 's' de 'nodo' que no es antepasado de 's' en el grafo solución, e incluirlo en el grafo de exploración. **(1)**
        - Mejor = MIN_FEV
        - SI jugador = MAX
            - Jugador = MIN
        - SINO
            - Jugador = MAX
        - PARA cada sucesor 's' de 'nodo' **(2)**
            - MMvalor ('s') = MINIMAX ('s', profundidad+1, jugador) **(3)**
            - SI mejor < -MMvalor **(4)**
                - Mejor = -MMvalor
        - //FIN PARA cada sucesor
        - Devolver MMvalor = mejor
    - //fin SINO sucesores
- //FIN procedimiento MINIMAX

Llamada inicial MINIMAX (raíz, 0, MAX). Devuelve la etiqueta de la peor jugada para el contrario.

(1) Evita la formación de ciclos en el grafo.

(2) Se va a quedar con el valor más pequeño de los asociados a sus sucesores, cambiado de signo.

(3) Se cambia el jugador, ya que lo que se obtiene es el valor asociado al sucesor.

(4) El peor valor de los sucesores cambiado de signo es el mejor valor para el jugador.

4.9.4. Análisis del algoritmo

El concepto de completo y óptimo cambia en las estrategias entre adversarios, ya que, hasta ahora, estas características lo eran del método en base a como afrontaban la resolución del problema. Ahora lo que se busca es la mejor jugada que nos lleve a la victoria (o al empate), al margen del valor que tengan asociado; es una situación dinámica, pues la situación puede cambiar con el tiempo. En este contexto, estas dos propiedades pierden significado.

- **Complejidad temporal**.- Dado que se trata de una exploración exhaustiva en profundidad la complejidad será exponencial. Si suponemos que el coste de cada jugada es uniforme, el coste estará en O (n^p). Aunque sea en una constante, el procedimiento MMvalor es más eficiente que el etiquetado MAX-MIN.
  Dado que, durante el juego, se están buscando dos estrategias ganadoras, una para MAX y otra para MIN, que confluyan en un nodo terminal común, puede haber situaciones en las que las dos estrategias sean compatibles (sigan los mismos caminos); esto es, la estrategia ganadora para cada jugador se realiza en profundidades alternas, por lo que expandirá cada una $n^{p/2}$ nodos, que, al ser los mismos, resultará en complejidad Ω ($n^{p/2}$). Encontrar dos estrategias totalmente compatibles es muy raro, por lo que este límite solo se alcanza en raras ocasiones.

- **Complejidad espacial**.- Dado que es una exploración en profundidad con retroceso, su complejidad espacial será lineal con la profundidad, O (p).

4.9.5. Conclusiones

Su principal ventaja, la de su sencillez, es a su vez su principal desventaja, pues al ser una búsqueda exhaustiva, su complejidad será exponencial. Si su factor de ramificación es muy grande, se podrá profundizar muy pocos niveles para evaluar la hipotética evolución del juego (el ajedrez es uno de ellos). Cuanto más profundo se pueda llegar, mayor calidad tendrá la respuesta obtenida, y por tanto, mayor la posibilidad de realizar una partida mejor.

Un problema que nos puede surgir en un juego es que se debe realizar una jugada en un tiempo máximo determinado. Dado que la profundidad a la que podamos llegar depende del número de nodos que hay que generar, podríamos encontrarnos con que para algunas jugadas no se tuviera una respuesta, y para otra sí, en el tiempo estipulado. Para solventar este problema lo que se hace es una exploración en profundidad progresiva (ya se vio que su complejidad temporal se diferenciaba en una constante con la búsqueda con retroceso cronológico).

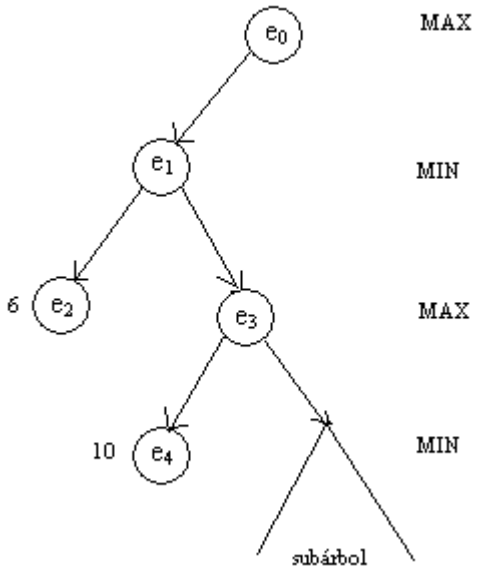
Otro problema que nos podemos encontrar con este método es el "efecto frontera", el cual es difícil de detectar. Supongamos que el algoritmo tiene fijado el límite de profundidad en 4 movimientos (8 jugadas). En condiciones normales se evalúa el grafo para encontrar la mejor jugada. Puede resultar que en alguno de los 4 movimientos siguientes, el contrario haga una jugada extraña (incoherente), que hace que en el quinto movimiento convierta la jugada elegida como ganadora se convierta en perdedora. Una posible solución es una búsqueda alternativa a partir de esa jugada inesperada.

4.9.6. Poda α-β

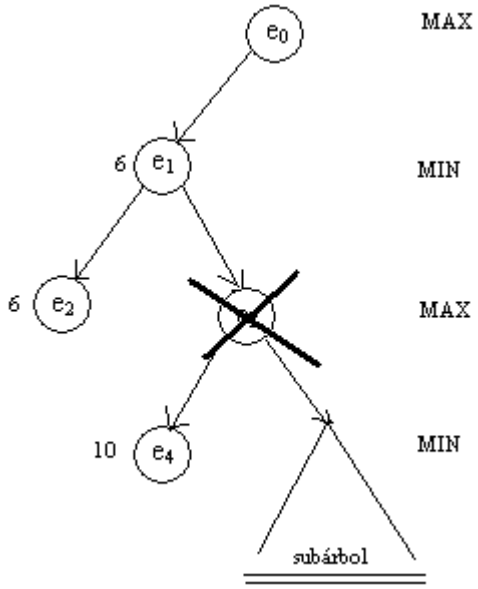
Como podemos ver, en MINIMAX, se realiza una búsqueda en todo el grafo parcial del juego, por lo tanto, si queremos mejorar la eficiencia del algoritmo, necesitamos reducir el espacio de búsqueda. Esto va a ser posible, dado que hay muchas ramas que no van a influir en la evolución del juego. Si observamos el primer ejemplo de MINIMAX, se puede ver que nos podríamos haber ahorrado expandir la rama del estado e_2, ya que ya sabíamos de antemano que teníamos un valor máximo para el movimiento en otra rama, que era e_3 y por tanto ya sabíamos que ese era el valor que se propagaría.

El procedimiento α-β es un método heurístico que modifica MINIMAX para reducir tanto el número de ramas que hay generar como el número de nodos terminales que hay que evaluar estáticamente. Para ello lo que se hace es, durante la evaluación dinámica, recordar cuál es el mayor valor encontrado para un nodo MAX y cuál es el menor encontrado para MIN, para proceder a ignorar las ramas que se sabe de antemano que no va a ser útiles y por tanto reducir el grafo de exploración.

Para entender este método, vamos a analizar el comportamiento de MINIMAX con un ejemplo. Supongamos el grafo de la figura, en el que sabemos la evaluación de los nodos terminales $h(e_2) = 6$ y $h(e_4) = 10$.

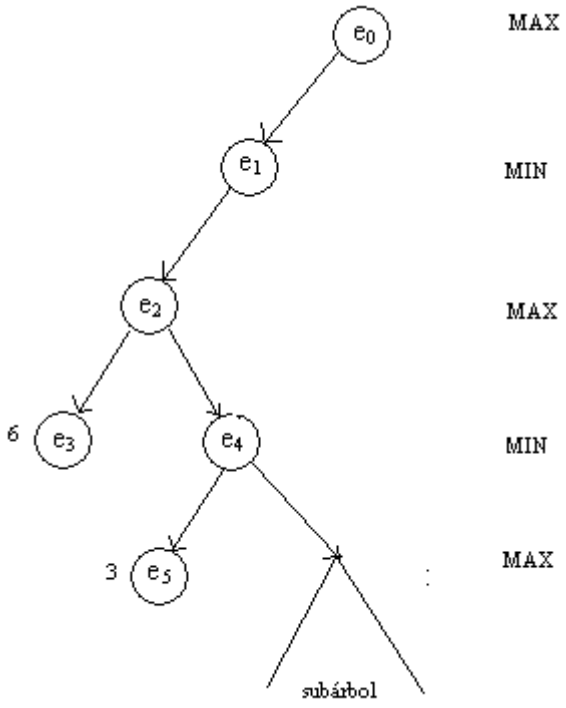


Si aplicamos MINIMAX, etiquetaremos e_1, que es un nodo MIN, como máximo con un 6 ya que por ser este jugador, e_1 se etiquetará con el menor de sus sucesores. El nodo e_3 es MAX, luego se tiene que etiquetar con el mayor de sus sucesores. Como podemos observar, si alguno de estos últimos es mayor que 6 (en este caso 10), como mínimo, estará etiquetado con este valor. ¿De qué nos ha servido evaluar esta rama? Pues de nada, ya que en el momento que e_3 tenga una etiqueta superior a 6 (la de su nodo hermano), esta siempre se desechará al etiquetar e_1, ya que es superior a 6.



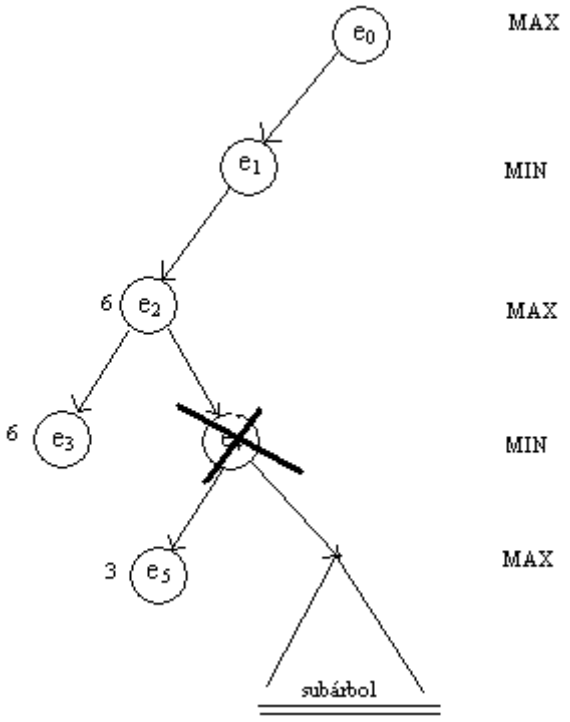
Para nuestro procedimiento vamos a tener una variable **β** para recordar cuál es el menor valor encontrado para un nodo MIN y que se va a ir modificando según profundizamos en el grafo. Va a ejercer como cota superior de valores de MIN, es decir, es el valor máximo que debe tener un nodo MAX descendiente para no desechar este subárbol. Los podados del árbol realizados en virtud de esta variable se llaman poda-β, y se señalará (en este caso, las dos líneas paralelas indican que se ha podado esta rama).

Supongamos ahora el siguiente grafo, en el que sabemos la evaluación de los nodos terminales $h(e_3) = 6$ y $h(e_5) = 3$.



Aplicando MINIMAX, etiquetaremos e_2, que es un nodo MAX, con el mayor valor de sus nodos hijos: e_2 nunca va a ser inferior a 6. El nodo e_4 es MIN, luego se va a etiquetar con el menor de sus hijos, que son MAX. En el momento que cualquier de estos sea inferior o igual a 6, como es el caso, nos será indiferente el valor de la etiqueta de e_4, ya se ignorará a favor del 6. En este caso se va a ignorar todo el subárbol que parte de e_4 por ser una tarea inútil.

Para nuestro procedimiento vamos a tener una variable **α** para recordar cuál es el mayor valor encontrado para un nodo MAX y que se va a ir modificando según profundizamos en el grafo. Va a ejercer como cota inferior de valores de MAX, es decir, es el valor mínimo que debe tener un nodo MIN descendiente para no desechar el subárbol. A estos podados se les llama poda- α (también en este caso, las dos líneas paralelas indican que se ha podado esta rama).



4.9.7. Algoritmo de poda α-β. Etiquetado MAX-MIN

Tomando como base el algoritmo MINIMAX, se modificará en lo referente al acotamiento. Para ello se sustituyen, en las llamadas, las variables minimo y maximo por α y β. Para determinar cuándo se debe desechar una rama, tenemos que, mientras se realiza el

procedimiento, α es menor que β y que el intervalo entre ellos se va estrechando. Cuando se llega a que α sea mayor que β, se pueden producir estas dos situaciones:

- Dado un nodo MAX_1, tenemos que uno de sus sucesores le ha dado un valor α. Tenemos un descendiente MAX_2, cuyo valor será inferior a ese α. Dado que este valor será un valor β, por el cálculo del nodo MIN antecesor de MAX_2, resulta que esta rama nunca se va a tener en cuenta.

- Dado un nodo MIN_1, tenemos que uno de sus sucesores le ha dado un valor β. Tenemos un descendiente MIN_2, cuyo valor será superior a ese β. Dado que este valor será un valor α, por el cálculo del nodo MAX antecesor de MIN_2, resulta que esa rama nunca se va a tener en cuenta.

Luego, en cualquiera de los dos casos, desechamos la rama del nodo en el que se cruzaron las cotas.

Se van a utilizar como base los algoritmos vistos para MINIMAX. En el caso del primero, el no recursivo, la función MINIMAX no se modifica, excepto la llamada a MINIMALISTA, en la que se le pasa MIN_FEVy MAX_FEV como parámetros, que son α y β respectivamente.

- Procedimiento poda_αβ (raíz) devuelve ganador
    - ganador = MIN_FEV
    - Generar el conjunto S de sucesores 's' de 'raíz' que no sea 'raíz'.
      PARA cada elemento 's' de S
        - etiquetaHijo = MINIMALISTA ('s',profundidad+1,MIN_FEV,MAX_FEV)
        - SI ganador < etiquetaHijo //nos quedamos con el mayor valor
            - ganador=etiquetaHijo
    - //FIN para cada sucesor
    - Devolver ganador
- //FIN procedimiento MINIMAX

  //Valor de nodo sucesor MIN
- Procedimiento MINIMALISTA (nodo, profundidad, **α, β**) devuelve etiquetaHijo
    - SI (nodo no tiene sucesores) o (nodo está al límite de exploración)
        - Devolver etiqueta= h(nodo)
    - SINO //hay sucesores

- Generar el conjunto S de sucesores 's' de 'nodo' que no sean antepasados de 's' en el grafo del juego.
  PARA cada elemento 's' de S
  - etiquetaHijo= MAXIMALISTA('s', profundidad+1)
  - SI **β** > etiquetaHijo //nos quedamos con el menor valor**(1)**
    - **β** = etiquetaHijo
  - **SI α ≥ β**
    - **Devolver α**
- //FIN para cada sucesor
- Devolver etiquetaHijo = **β**

o //fin SINO (hay sucesores

- //fin procedimiento MINIMALISTA

//Valor del nodo sucesor MAX

- Procedimiento MAXIMALISTA (nodo, profundidad**, α, β**) devuelve etiquetaHijo
  - SI (nodo no tiene sucesores) o (nodo está al límite de exploración)
    - Devolver etiqueta= h(nodo)
  - SINO //hay sucesores
    - Generar el conjunto S de sucesores 's' de 'nodo' que no sean antepasados de 's' en el grafo del juego.
      PARA cada elemento 's' de S
      - etiquetaHijo = MINIMALISTA('s', profundidad+1)
      - SI α < etiquetaHijo //nos quedamos con el mayor valor
        - α = etiquetaHijo
      - **SI α ≥ β**
        - **Devolver β**
    - //FIN para cada sucesor
    - Devolver etiquetaHijo = α
  - //fin SINO (hay sucesores)
- //fin procedimiento MAXIMALISTA

La modificación para la versión recursiva también es la misma:

- Procedimiento poda_αβ (nodo, profundidad, jugador**, α, β**) devuelve etiquetaHijo
  - SI (nodo no tiene sucesores) O (nodo está al límite de exploración)
    - Devolver h(n)
  - SINO
    - Generar el conjunto S de sucesores 's' de 'nodo' que no sea antepasado de 's' en el grafo solución.
    - PARA cada sucesor 's' de 'nodo'

- SI jugador = MAX
  - etiquetaHijoMIN = poda_αβ (s, profundidad+1,MIN **, α, β**)
  - SI **α** < etiquetaHijoMIN //nos quedamos con el mayor
    - **α** = etiquetaHijoMIN
  - **SI α ≥ β**
    - **Devolver β** //finaliza la rama
- SINO //jugador=MIN
  - etiquetaHijoMAX= poda_αβ (s, profundidad+1,MAX**, α, β**)
  - SI **β** > etiquetaHijoMAX //nos quedamos con el menor
    - **β** = etiquetaHijoMAX
  - **SI α ≥ β**
    - **Devolver α** //finaliza la rama
- //fin SI jugador=max

▪ //FIN para cada sucesor
▪ SI jugador = MAX
- Devolver **α**

▪ SINO //jugador=MIN
- Devolver **β**

o //FIN si no es terminal

- //fin procedimiento poda_αβ

Llamada inicial: poda_αβ (raíz, MIN_FEV ,MAX_FEV,0,MAX):

4.9.8. Algoritmo de poda α-β. Etiquetado MMvalor

Como ya se dijo, el procedimiento MMvalor elimina la necesidad de saber si estábamos en un nodo MAX o MIN a la hora de etiquetar los antecesores. Por este motivo se sustituyen las variables α y β por limiteSiguientes (ls) y limiteActual (la) respectivamente, y que siguen siendo cotas máxima y mínima para los sucesores. El valor αβvalor nos va a dar el valor h(n) para la mejor jugada para el nodo actual de entre sus sucesores.(el mayor si es un nodo MAX o el menor si es un nodo MIN).

- Procedimiento poda_αβm (nodo, profundidad, jugador,**ls,la**) devuelve αβvalor
  - SI (nodo no tiene sucesores) O (nodo está al límite de exploración)
    - SI jugador = MAX
      - αβvalor = h(nodo)
    - SINO //jugador MIN

- αβvalor = -h(nodo)

o SINO //tiene sucesores a profundidad menor del límite
  - Generar el conjunto S de sucesores 's' de 'nodo' que no es antepasado de 's' en el grafo solución. Añadir cada 's' al grafo de exploración
  - SI jugador = MAX
    - Jugador = MIN
  - SINO
    - Jugador = MAX
  - PARA cada sucesor 's' de 'nodo'
    - αβvalor = poda_αβm ('s', profundidad+1, jugador,-la,-ls)
    - SI ls < -αβvalor //nos quedamos con el mayor
      - ls = -αβvalor
    - SI ls ≥ la
      - Terminar de analizar sucesores
    - //FIN PARA cada sucesor
  - αβvalor = ls
o //fin SINO sucesores
o Devolver αβvalor

- //FIN procedimiento poda_αβm

Llamada inicial poda_αβm (raíz, 0, MAX, MIN_FEV, MAX_FEV). Devuelve la etiqueta de la peor jugada para el contrario.

4.9.9. Análisis del algoritmo

- **Complejidad temporal**.- Depende en gran medida de orden en que se generen los nodos del grafo. Para un grafo ordenado de manera que se obtiene la máxima ventaja de aplicar la poda α-β, el teorema de M. Levin enuncia que el número máximo de nodos que se van a evaluar es :

o Si la profundidad p del grafo es par

$$2n^{\frac{p}{2}} - 1$$

o Si la profundidad p del grafo es impar

$$2n^{\frac{p+1}{2}} + 2n^{\frac{p-1}{2}} - 1$$

Donde n es el factor de ramificación.

Es decir, se puede reducir casi a la mitad el factor exponencial del caso peor, en el que haya que evaluar todos los nodos terminales del grafo, donde la complejidad será la misma que MINIMAX, $O(n^p)$.

- **Complejidad espacial**.- Al estar basado en el recorrido en profundidad con retroceso, esta será lineal con la profundidad O(p).

4.9.10 Conclusiones

Su principal ventaja es que puede reducir considerablemente el número de nodos a evaluar, lo que hace que se pueda profundizar más en el grafo del juego, y por lo tanto hacer mejores movimientos.

Su principal desventaja es la dependencia con respecto al orden en que se generen los nodos. Una posible mejora sería sustituir la exploración en profundidad progresiva por la profundidad iterativa, de manera que se va obteniendo información para, en un momento dado, ordenar los nodos según convenga. Hay que ser cautelosos con esta modificación, ya que puede aumentar el coste del algoritmo.

--

APENDICE A.- TEORÍA DE GRAFOS

A.1.- DEFINICIONES

• **Vértice o nodo.**- Se representará por una circunferencia, que podrá estar etiquetada. Vértice es el nombre formal en matemáticas. Para las técnicas que utilizaremos, normalmente se usará el término *nodo*.

• **Arista y arco.**- Si A y B son dos nodos distintos,

- una arista se define como un conjunto de dos nodos de la forma {A,B}. Su representación será con una línea que une lo dos nodos.

- Un arco se define como un par ordenado de la forma (A,B). Su representación será con una línea con dirección hacia el segundo nodo del par.

Se dice que dos nodos unidos por un arco son **adyacentes** y también que son los **extremos** del arco**.**

• **Grado de un vértice (nodo).-** Número de aristas (arcos) que inciden en el vértice (nodo). Un grafo es **regular** si todos sus vértices tienen el mismo grado. Un grafo regular se dice que es **completo** si en cada nodo existe una arista que sale a cada uno de los nodos restantes del grafo.

• **Grafo (no dirigido).-** conjunto de '***vértices'*** distintos unidos por '***aristas'***. Formalmente un grafo es G=(N,A), donde

- N es un *conjunto* finito *no vacío* de nodos.

- A es un conjunto de aristas, es decir, de conjuntos de dos elementos, que son los vértices que se unen.

- **Grafo dirigido.**- o digrafo: Conjunto de ***'vértices'*** distintos unidos por ***'arcos'.*** Formalmente, un grafo dirigido es G = (N,A) donde:

  - N es un conjunto finito no vacío de nodos.

  - A es un conjunto de pares *ordenados*, llamados *arcos directores*, que describen la unión desde el primer hacia el segundo nodo. Así (A,B) ≠ (B,A).

Desde el punto de vista de la teoría de conjuntos, un grafo dirigido es una relación sobre el conjunto N de los nodos del grafo.

- **Subgrafo.-** *Es también un grafo* (dirigido o no), extraído del grafo original. Formalmente el grafo G´=(N´,A´) es subgrafo de G=(N,A) si

  - N´ es un subconjunto no vacío de nodos de N.

  - A´ es un subconjunto de A.

- **Pseudografo**.- Es un grafo (o grafo dirigido) en el que se permiten aristas (arcos) cuyos extremos salen y entran al mismo nodo, es decir, cada arista (arco) forma un **bucle o lazo** sobre sí mismo, siendo un conjunto de la forma {A,A} o un par ordenado de la forma (A,A), para grafo no dirigido y dirigido respectivamente..

- **Multigrafo**.- Es un grafo (o grafo dirigido) en el que puede haber más de una arista (arcos en la misma dirección) para dos nodos cualesquiera. Formalmente, en el conjunto A de aristas (arcos), puede haber dos con los mismos extremos pero distinta aristas (arco). Luego se verá como diferenciarlos.

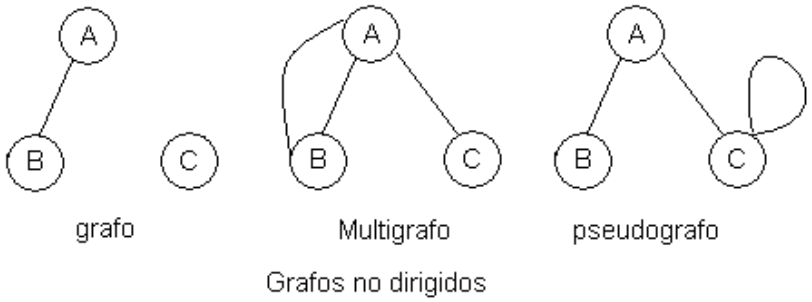


Figura A.1. Grafos diversos

En teoría de computación (autómatas) podremos tener cualquier de los tres tipos de grafo dirigido, mientras que para las técnicas de programación solo se van a considerar aquellos grafos y grafos dirigidos en los que las aristas y los arcos, respectivamente, unen nodos distintos (no habrá bucles ni varios arcos para dos mismos nodos, o si el grafo es dirigido, los arcos tienen que ir en direcciones opuestas).

- **Camino.**- Es una sucesión finita de nodos y aristas (arcos) alternativamente, de manera que entre dos nodos consecutivos hay una arista (arco).
  Para grafos (no así para los multigrafos ni pseudografos), la sucesión puede simplificarse a sólo nodos, dado que en un grafo solo hay un posible camino entre dos nodos.
  En la definición de camino no se tiene en cuenta la dirección de las flechas. Es decir, es indiferente que el grafo sea dirigido o no, lo que se tiene en cuenta es la unión entre nodos.

- **Longitud de un camino**.- Número de aristas (arcos) que tiene. Si la arista (arco) está etiquetada con números, a esta etiqueta se le suele llamar *longitud* de la arista (arco), y la longitud total del camino será la suma de las etiquetas de las aristas (arcos).

- **Camino simple o ruta.**- Son aquellos que no pasan dos veces por el mismo nodo.

- **Camino Cerrado.**- si el último nodo del camino es el mismo que el primero. Formalmente, dados los nodos de un camino $(n_1 \ldots n_k)$, donde $n_1=n_k$ y $k \geq 1$, y todos los nodos son distintos excepto n_1 y n_k.

- **Circuito.**- Camino cerrado en el que no se repite ninguna arista (arco).

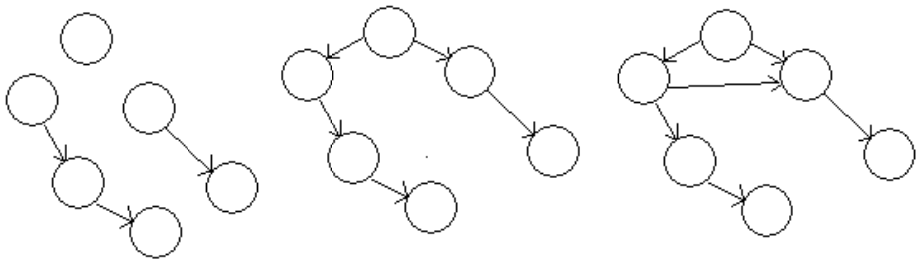
- **Camino abierto.**- Cuando en un camino, el último nodo no es el mismo que el primero.

- **Grafo conexo.-** Cuando entre dos nodos cualesquiera (todos) hay al menos un camino. Dicho de otra forma, hay un camino para cada dos nodos cualesquiera del grafo. Si no es conexo, es **no conexo, inconexo o desconexo**. Estos se pueden clasificar en:

  - **Grafo simplemente conexo o poliárbol.-** Cuando para cada dos nodos cualesquiera (todos) hay solo un camino y solo uno.

  - **Grafo múltiplemente conexo.-** Para algún par de nodos (alguno) hay 2 o más caminos entre ellos (como consecuencia de la definición, si hay un camino cerrado, es múltiplemente conexo).

  - **Grafo fuertemente conexo.**- Es un grafo dirigido en el que se puede pasar de entre dos nodos cualesquiera (todos) por un camino siguiendo las flechas de sus arcos.



a) Inconexo b) Simplemente conexo c) Múltiplemente conexo

Estos son grafos dirigidos. Para clasificarlos como conexos se tiene en cuenta el camino, por tanto obviamos las flechas de dirección.

Fig A.2.- Grafos conexos

- **Circuito euleriano**.- Es un circuito que contiene todas las aristas que aparecen una y solo una vez en el camino. Si un grafo tiene algún circuito euleriano, se dice que es un grafo euleriano.
  Un circuito euleriano se dará siempre que todos y cada uno de los vértices tengan un grado par.

- **Camino hamiltoniano**.- Es un camino simple que contiene todos los vértices del grafo una y solo una vez. Un ciclo (camino cerrado) hamiltoniano es aquel camino hamiltoniano que empieza y acaba en el mismo vértice. Para resolver un camino hamiltoniano todavía no se ha encontrado un criterio eficaz.

A.1.1. Grafos no dirigidos

- **Ciclo.-** Es cualquier camino cerrado.

- **Longitud del ciclo.-** Número de aristas distintas que tiene un camino cerrado.

- **Árbol o árbol libre.-** Un grafo conexo que no tiene ciclos.
  Un árbol tiene las siguientes características:

  - Para n nodos, hay exactamente n-1 aristas.

  - Si se añade una arista, entonces se produce un ciclo

  - Si se elimina una arista, entonces deja de ser conexo.

  Un caso especial de estos, los árboles con raiz se verá después de los grafos dirigidos.

- **Bosque**.- Es un grafo que no tiene ciclos pero que no es conexo. Para cumplir con la definición, cada subgrafo conexo que pertenece al bosque es un árbol (de ahí el nombre de bosque).

A.1.2. Grafos dirigidos

Dado el grafo dirigido de la figura 3 vamos a definir la relación entre sus nodos:

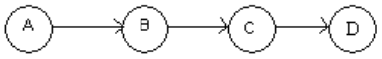


Fig A.3. Grafo dirigido simple

- A es **precedesor** de B y viceversa, B es **sucesor** de A. (distancia 1)
- A es **antepasado** de D si se da una de las siguientes condiciones:
    - A es predecesor de D
    - Existen un nodo B que es sucesor de A y antepasado de D (definición recursiva, para distancia >1).
- D es **descendiente** de A si se da una de las siguientes condiciones:
    - D es sucesor de A
    - Existen un nodo C que es predecesor de D y descendiente de A (definición recursiva para distancia >1).
- A y B son **familia.** Una familia es el conjunto de un nodo y de todos sus padres (si tuviera más de uno).
- **Ciclo (dirigido).-** Cuando, habiendo un camino cerrado, este se puede recorrer en la dirección indicada por los arcos.
- **Bucle.-** Cuando, habiendo un camino cerrado, este NO se puede recorrer en la dirección indicada por los arcos.
- **Grafo Dirigido Acíclico (GDA -DAG en inglés-).-** son aquellos grafos dirigidos en los que no hay ciclos (aunque sí bucles). En estos, **PADRE** es sinónimo de

predecesor e **HIJO** es sinónimo de sucesor. Los nodos que no tienen hijos (no tienen descendientes) se llaman **TERMINALES, HOJAS O EXTREMOS.** A su vez todos los nodos que no son extremos, se denominan **NO TERMINALES O INTERNOS**.

Un GDA *conexo* define una relación de orden (normalmente parcial) entre los nodos (además de la propiedad reflexiva y antisimétrica, pueden cumplir la propiedad transitiva; si tuviese ciclos, ya no cumpliría esta última, y por lo tanto no podríamos determinar un primero y un último). Algunos detalles:

- Un hijo puede tener varios padres

- Como mínimo tiene un nodo del que solo salen arcos y como mínimo tiene un nodo al que solo le llegan los arcos (dirigidos).

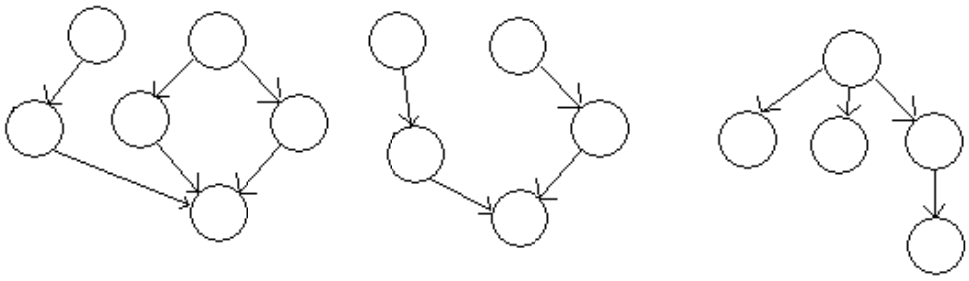
- **Poliárboles.-** Son **GDA,s conexos** que **no tienen bucles** (es decir, no tiene caminos cerrados). Al igual que en los anteriores, un nodo puede tener más de un padre.

- **Árboles (dirigidos)-** Son poliárboles (GDA,s conexos sin bucles) en los que cada nodo tiene exactamente un padre. Sus propiedades son:

  - Existe un único nodo que no tiene antepasados y que además es antepasado de todos los demás. Se denomina **RAÍZ.**

  - Esta raíz no tiene ningún padre. El resto de nodos tienen exactamente un padre. Los nodos que tienen el mismo padre son **HERMANOS.**

    A partir de la definición se deduce que, para cualquier nodo del árbol, existe un camino (y solo uno) desde este nodo hasta el nodo raíz. A este camino se le llama **rama** del árbol (recordar que es antepasado de todos los nodos, y por tanto ese camino tiene que existir).



a) GDA b) Poliárbol c) Árbol

Fig A.4. Grafos dirigidos

A.1.3. Árboles con raíz

Hemos visto que en los árboles dirigidos, el nodo raíz es único y viene dado por la definición del mismo, y que además también podemos utilizar todos los términos heredados de los GDA,s: padre, hijo, terminal u hoja, junto con los propios de los árboles dirigidos: hermano y rama.

Un árbol con raíz es un árbol (no dirigido), donde uno de los nodos (uno cualquiera) se considera como raíz. Como es un árbol especial en el que estamos definiendo un orden parcial para los nodos, podemos utilizar los términos padre, hijo, hermano y terminal en el mismo sentido que en los GDA,s y árboles dirigidos. De la misma forma, podremos utilizar el término bosque, para un grafo dirigido compuesto por árboles dirigidos.

En la representación gráfica de un árbol dirigido y un árbol con raíz (no dirigido), la única diferencia serán las flechas de sus arcos. Así se dibujará el nodo raíz en la parte superior. Por debajo de él se dibujarán todos sus hijos, unidos por una arista (arco) dirigido o no según el árbol que se trate. Esto es lo que forma un **NIVEL** del árbol. Esto se repetirá hasta que se llegue a todos los nodos terminales (ver la figura A.4.c en la que se dibuja un árbol dirigido de tres niveles).

A.2. ÁRBOLES Y GDA,S DE UNA RAÍZ

Dado un nodo cualquiera del árbol con raíz, o un nodo cualquiera en un GDA con solo una raíz, este tiene unas características:

- **Altura de un nodo**.- número de arcos, POR EL CAMINO MÁS LARGO, desde ese nodo hasta un nodo terminal. La altura de un nodo terminal es cero.

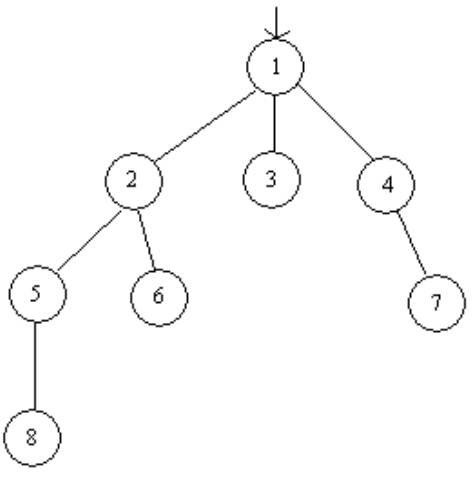
- **Altura del árbol o GDA con una raíz**.- Es la altura del nodo raíz.

- **Profundidad de un nodo**.- Número de arcos que hay desde el nodo raíz a ese nodo POR EL CAMINO MÁS CORTO (esto es obvio en el caso de árboles, en el que solo hay un camino posible). La profundidad de la raíz es cero y la de cualquier otro nodo es la de su antecesor (padre) menos profundo más 1 (recordar que en un GDA –con una raíz en este caso-, se puede dar el caso de tener un nodo que tenga más de un padre).

- **Profundidad del árbol o GDA con una raíz**.- El número de arcos que hay desde el nodo raíz hasta el nodo terminal menos profundo.

- N**ivel de un nodo en un árbol.-** En la mayor parte de la bibliografía consultada, el concepto se define como nivel de profundidad del nodo, siendo por tanto sinónimo de profundidad. Así la raíz está en el nivel 0, sus hijos en el nivel 1, ….
  Otros autores definen el nivel como la diferencia de la altura de la raíz menos la profundidad del nodo. Esta definición, que es poco usual, y es equivalente a decir que los nodos terminales más profundos están en nivel 0, sus padres en el nivel 1….

Ejemplo A.1. Dado el árbol de la figura, identificar cada parámetro del mismo.



Se trata de un árbol con raíz (nodo 1) no dirigido.

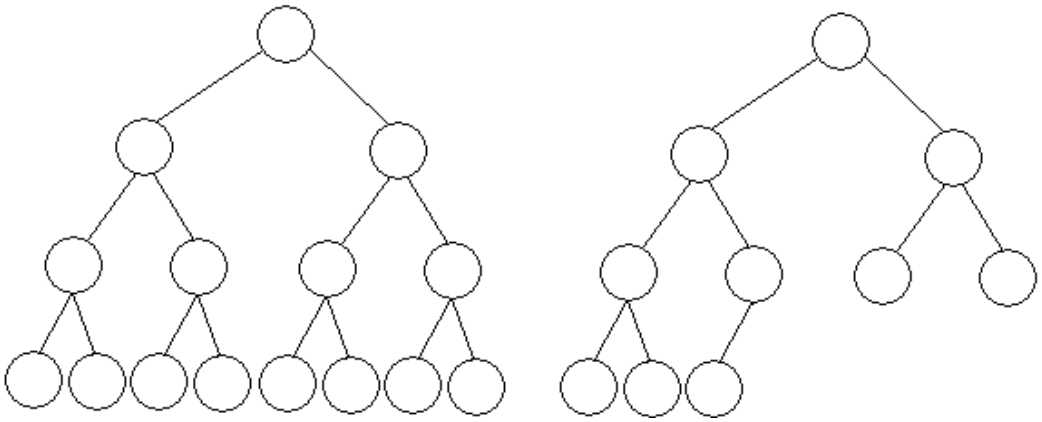
- Profundidad del árbol.- El nodo terminal menos extremo es el 3, luego la profundidad del árbol es 1.
- El nodo 8, 6, 3 y 7 tienen una altura 0 (son terminales)
- El nodo 5 y 4 tienen una altura 1.
- El nodo 2 tiene una altura 2. (recordar, camino más largo a un nodo terminal, el 8 en este caso)
- El nodo 1, que es el raíz tiene una altura de 3, que es la altura del árbol (el camino más largo a un nodo terminal es al nodo 8).
- La profundidad del nodo 1, el raíz, es 0. Dicho de otra forma, el nivel del nodo 1 es 0.
- Los nodos 2,3 y 4 están a profundidad 1. Dicho de otra forma, los nodos 2,3 y 4 están en el nivel 1 (con la segunda definición en el nivel 2).
- Los nodos 5, 6 y 7 están a profundidad 2. Dicho de otra forma, los nodos 5,6 y 7 están en el nivel 2(con la segunda definición en el nivel 1).
- La profundidad del nodo 8 es 3. Dicho de otra forma el nodo 3 está en el nivel 3 (con la segunda definición en el nivel 0).

--

A.2.1 Árboles binarios

Es un árbol tal que el grado de todos sus nodos no es superior a tres. De aquí se deduce que cada nodo solo puede tener dos hijos como máximo, conocidos como *hijo izquierdo e hijo derecho* (una arista es la que viene del padre y las otras dos de sus posibles hijos).

Se dice que un árbol *binario* es **completo** (no confundir con la definición de completo en los grafos) si todos sus nodos tienen cero o dos hijos; los que tienen cero hijos son nodos terminales y se caracterizan por estar a la misma profundidad. Es **casi-completo o esencialmente completo** si, dado un árbol de profundidad n, todos los nodos de profundidad n-2 tienen dos hijos y los de profundidad n-1 tienen 0, 1 (que se representa a la izquierda) ó 2 hijos. Además se representan de izquierda a derecha, de manera que los nodos del nivel n-1 que son no terminales, primero se representan los que tienen 2 hijos, luego el que tiene un hijo y luego los que son terminales.



a) Arbol binario completo b) Arbol binario casi-completo

Fig A.5. Árboles binarios

A.3.MATRIZ DE ADYACENCIA

Es la manera analítica de representar un grafo o un grafo dirigido (pensar que la manera gráfica está limitada a grafos o digrafos sencillos, ya que, en caso contrario, serían ilegibles). Según se trate de un grafo o un digrafo tenemos dos matrices distintas:

- Dado un grafo G=(N,A) , donde hay n nodos, etiquetados de 1 a n, la matriz M_n de adyacencia se define como la matriz cuadrada de dimensión n, donde cada elemento m_{ij} (para los valores $1 \leq i \leq n$ y $1 \leq j \leq n$) toma el valor 1 ó 0:
  - Si la arista $\{i,j\} \in A$, entonces los elementos m_{ij} y m_{ji} toman el valor 1.
  - Si la arista $\{i,j\} \notin A$, entonces los elementos m_{ij} y m_{ji} toman el valor 0.

- Dado un grafo dirigido G=(N,A), donde hay n nodos, etiquetados de 1 a n, la matriz M_n de adyacencia se define como una matriz cuadrada de dimensión n, donde cada elemento m_{ij} (para los valores $1 \leq i \leq n$ y $1 \leq j \leq n$) :

  - Toma el valor 1 si el arco $(i,j) \in A$

  - Toma el valor 0 si el arco $(i,j) \notin A$

  recordar que en el grafo dirigido, (i,j) es un par ordenado y por tanto $(i,j) \neq (j,i)$

La matriz de adyacencia va a representar, para un nodo concreto, los nodos que están conectados a este (caminos de longitud 1):

- Para grafos no dirigidos, los que están en la fila (columna) del nodo (es una matriz simétrica).

- Para grafos dirigidos, son los nodos que están en la fila y la columna del nodo en cuestión.

Ejemplo A.2. Matriz de adyacencia para el árbol del ejemplo A.1.

$$G = \begin{vmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{vmatrix}$$

--

BIBLIOGRAFÍA

[Alonso et al 2007] Alonso Jiménez, J., A. , *Temas de inteligencia artificial I* [en línea] Universidad de Sevilla. Escuela Superior de Ingeniería Informática. Departamento de Ciencias de la Computación e Inteligencia Artificial. Asignatura del curso 2006/2007 [consulta 27-8-2007]. Disponible en: http://www.cs.us.es/cursos/ia1/temas/

[Bejar 2009]] Bejar, J. *Apunts d'intel.ligència artificial*. [en línea]. Departament de Llenguatges i Sistemes Informátics. Universidad Politécnica de Barcelona. Curso 2009/2010. [consulta 15-03-2010]. Disponible en : http://www.lsi.upc.es/~bejar/ia/teoria.html

[Brassard et al 1997] Brassard, G., Bratley, P. *Fundamentos de Algoritmia.* 1ª Edición. Madrid (España): Pearson Educación S.A, impresión de 2006. ISBN 9788489660007

[Bujalance et al, 1993] Bujalance, E., Bujalance, J.A., Costa, A.F., Martínez. E. *Elementos de Matemática Discreta*. 1º edición. Madrid (España): Editorial Sanz y Torres, S.L, 2003. ISBN 84-88667-00-0

[Diez 1998] Diez, F., J. Introducción al razonamiento aproximado [en línea]. Universidad Nacional de Educación a Distancia (UNED). Departamento de Inteligencia artificial. [consulta 11/11/2008] Revisión 2005. Referencia: http://www.ia.uned.es/~fjdiez/libros/razaprox.pdf

[Eduards et al 1961] Eduards, D.J., Hart, T.P. The *α-β heuristic.* [en línea] Artificial Intelligence project. RLE and MIT computation center. [consulta 22-04-2010] disponible en http://dspace.mit.edu/handle/1721.1/6098

[Escolano et al 1993] Escolano Ruiz, F., Alfonso Galipienso, M. A., Cazorla Quevedo, M., A., Colomina Pardo, O., Lozano Ortega, M.A. *Inteligencia artificial. Modelos, técnicas y áreas de aplicación*. Madrid (España): Thomson Paraninfo, 2003 ISBN 8497321839

[Fernández et al 1998] Fernández Galán, S., González Boticario, J., Mira Mira, J. *Problemas resueltos de inteligencia artificial aplicada. Búsqueda y representación*. 1ª edición. Madrid (España): Addison-Wesley Iberoamericana España. S.A. 1998. ISBN 84-7829-017-6

[Gardner 1980] Gardner, A., V.,D.,L. *Search: an overview.* [en línea] AI Magazine. American Association of Artificial Intelligence Winter 1980. Vol 2, núm 1, pags 2-6,23 [consulta 14-3-2010]. Disponible en http://www.aaai.org/AITopics/assets/PDF/AIMag02-01-002.pdf

[Ginsberg 1993]. Ginsberg, M. *Essentials of Artificial Inteligence.* 1ª edición. San Francisco (USA): Morgan Kaufmann Publishers. 1993 ISBN 9781558602212

[Maldonado et al 2005] Hernández García, I. (coordinadora), Maldonado, C., E. Estética Ciencia y Tecnología: Creaciones electrónicas y numéricas. 1ª Edición. Bogotá (Colombia): Pontificia Universidad Javeriana. 2005 ISBN 958-683-791-2

[Maldonado 2007] Maldonado C., E. *Origen del concepto y origen de un problema.* [en línea] Universidad Externado de Colombia. [consulta 20-8-2007] Disponible en http://www.complexsites.com/gpage18.html

[Mira et al 1995] Mira, J., Delgado, A.E., Boticario, J.G., Diez, F.J. *Aspectos básicos de la inteligencia artificial.* 1º Edición. Madrid (España): Editoria Sanz y torres, S.L., 2005. ISBN 84-88667-13-2

[Mira 2001]. Mira, J. *El viejo sueño griego de mecanizar el pensamiento y la emoción*. [en línea restringido]. Universidad Nacional de Educación a Distancia (UNED). Departamento de Inteligencia artificial. Curso 2009-2010. Tema 1 de la asignatura Percepción y control basado en el conocimiento. Referencia en: http://www.ia.uned.es/asignaturas/pcbc/

[Mira et al 2006] Mira, J., *Aspectos metodológicos del desarrollo de SBC,s.* [en línea] Universidad Nacional de Educación a Distancia (UNED). Departamento de Inteligencia artificial. Curso 2009-2010. Inteligencia Artificial e Ingeniería del Conocimiento. [consulta 23-03-2010]. Disponible en http://www.ii.uned.es/superior/cuarto/IAIngCon/docsIA-IC/Aspectos_Basicos.pdf

[Newell 1980] Newell, A. *The Knowledge Level*. [en línea] Presidential address, American Association of Artificial Intelligence, 19 Aug 1980.[consulta 15-3-2007] disponible en: http://www.aaai.org/AITopics/assets/PDF/AIMag02-02-001.pdf

[Nillson 1980] Nilsson, N., J. *Principles of artificial intelligence*. Reimpresión del original 1980. Morgan Kauffman. ISBN 0-934613-10-9. Berlín (Alemania):. Springer-Verlag 1993. ISBN 978-3-540-11340-9

[Nilsson 2001] Nilsson, N., J. *Inteligencia artificial. Una nueva Síntesis.* 1ª edición. Madrid (España): McGRAW-HILL/INTERAMERICANA DE ESPAÑA, S.A.U. 2001. ISBN 8448128249

[Pazos et al 1997] Pazos Sierra, J., Borrajo Millán, D. *Inteligencia artificial. Métodos y técnicas.* 1º Edición. Madrid (España): Editorial Centro de Estudios Ramón Areces (CERASA) 1997. ISBN 8480040904

[Russel Norvig 2004] Russell, S., Norvig, P. *Inteligencia Artificial, Un enfoque moderno* 2ª Edición. Madrid (España): Pearson Educación S.A. 2004 ISBN 84-205-40003-X

[Thorton et al 1998] Thorton, C., du Boulay, B. Artificial intelligence. Strategies, applications and models Throught search. 2ª edición. Nueva York (USA): AMACOM 1998 ISBN 0-8144-0470-7

[Winston 1994]. Winston, P., H. *Inteligencia artificial*. 3º Edición. Madrid (España): Addison Wesley Iberoamericana. 1994 ISBN 9780201518764

[Zhang 1998] Zhang, W. *Complete Anytime Beam Search*. [en línea restringido] Proceedings American Association of Artificial Intelligence, 1998 [consulta 29-3-2010]. Disponible en : http://www.aaai.org/Papers/AAAI/1998/AAAI98-060.pdf

--