# Sistemas Inteligentes

Jose Antonio Lorencio Abril

Curso 20/21

## Contents

# 1 Introducción a los Sistemas Inteligentes Artificiales

## 1.1 ¿Qué es la inteligencia artificial?

Es un área de la ciencia e ingeniería bastante reciente. Su objetivo son las capacidades que consideramos inteligentes.

La inteligencia artificial (IA) puede ser entendida desde cuatro enfoques:

- **Sistemas que actúan como humanos:** el estudio de cómo hacer computadores que hagan cosas que, de momento, la gente hacer mejor.

  El modelo es el hombre, y el objetivo es construir un sistema que pase por humano.

  **Test de Turing**: test ideado por Alan Turing, consistente en que un humano se comunica con una máquina. Si no es capaz de distinguir a la máquina de un humano, se considera que esa máquina pasa el test. Se supone que una máquina que pasa el test de Turing es inteligente, aunque hoy en día no se cree que esto sea suficiente para calificar un sistema de inteligente (hay chatbots muy avanzados que no son inteligentes)

  Para actuar como un humano, la máquina debe ser capaz de procesar el lenguaje natural (NLP), representar el conocimiento, razonar, aprender,...

  Es necesaria, además, la interacción programas-personas, por lo que los programas deben actuar como humanos.

- **Sistemas que piensan como humanos:** el esfuerzo por hacer a las computadores pensar, conseguir máquinas con mentes en el sentido amplio y literal. Ahora el modelo es la mente humana. Intentamos establecer una teoría sobre el funcionamiento de la mente (mediante experimentación psicológica), y, a partir de esta, establecer modelos computacionales, de lo que se encargan las ciencias cognitivas.
- **Sistemas que actúan racionalmente:** campo de estudio que busca explicar y emular el comportamiento inteligente en términos de procesos computacionales.

  Actuar racionalmente significa conseguir unos objetivos dadas unas creencias.

  El paradigma es el **agente racional**, que es un agente capaz de percibir su entorno y que aplica su conocimiento racional para actuar y conseguir sus objetivos.

  Requiere de ciertas capacidades, como NLP, representación del conocimiento, aprendizaje, adaptación al entorno,...

  Presenta una visión general, no centrada en el modelo humano, utiliza fuentes de conocimiento adicionales a la lógica

- **Sistemas que piensan racionalmente:** estudio de las facultades mentales a través del estudio de modelos computacionales. Se presupone que las leyes del pensamiento racional se fundamentan en la lógica formal, que es la base de los programas inteligentes. Este enfoque se encuentra con dos obstáculos:
  - Es muy difícil formalizar el conocimiento
  - Hay un gran salto entre la capacidad teórica de la lógica y su realización práctica

## 1.2 Bases de la IA

La historia de la IA va ligada al proceso humano de comprensión de la naturaleza, concretamente al problema del funcionamiento de la mente.

La **filosofía**, desde hace milenios, se ha preguntado sobre la inteligencia, desde diversos puntos de vista. Es destacable el desarrollo de la lógica formal, buscando formalizar los razonamientos humanos.

A través de la lógica, se hace posible la formalización de la búsqueda de patrones, las **matemáticas**, que pasan de ser métodos de contabilidad, a ser una marco lógico mediante el que se pueden establecer reglas, relaciones y verdades irrefutables (bajo los axiomas acordados como ciertos). Las matemáticas permiten abordar la IA desde varias perspectivas:

- Lógica: búsqueda de las reglas del razonamiento
- Complejidad: intento de establecer qué problemas puede resolver un ordenador en un tiempo razonable, y cuáles no
- Probabilidad: razonamiento en entornos inciertos

La **economía** ha profundizado bastante en el proceso de toma de decisiones con diferentes objetivos:

- decisiones beneficiosas
- decisiones contra competidores
- corto plazo VS largo plazo

dando lugar a la teoría de la decisión (estudio de la toma de decisiones), teoría de juegos (estudio de la toma de decisiones en entornos multiagente), investigación operativa (optimización).

La **neurociencia** y la **psicología** son dos grandes colaboradoras de la IA. Proponen modelos de funcionamiento del cerebro, modos de actuación, de aprendizaje y de desarrollo mental

La **computación** estudia las mejoras hardware y software para poder implementar potentes modelos de IA.

La **teoría de control** o **cibernética** colabora mediante la construcción de sistemas autónomos.

La **lingüística** ha resultado ser un importante campo para la IA, ya que estudia la representación del conocimiento, así como el funcionamiento gramatical de la lengua.

## 1.3 Aplicaciones de la IA

- Sistemas de recomendación o filtrado
- Interfaces hombre-máquina
- Robótica
- Tareas específicas en medicina, educación, finanzas,...

## 1.4 Método de la IA

Un **método** es un modo de realizar con orden la actividad científica y académica según su propia naturaleza.

La IA dispone de varias herramientas para resolver los problemas que se le plantean:

- Heurística y algoritmia:
  - Un **algoritmo** es un método general de resolución de algún problema, de forma que su validez sea incuestionable, por basarse en principios formalizados. Puede cuestionarse, eso sí, su eficiencia
  - Un **método heurístico** es un método de resolución de problemas que se basa en la intuición o en la experiencia, y sirve para resolver problemas de forma más rápida, pero correcta
- Computación simbólica y numérica
  - Un **símbolo** es una entidad cuyo valor no pertenece dominio en el que existe el símbolo. La inteligencia se asocia a la posibilidad de usar símbolos para representar la realidad y las ideas. Una aproximación metodológica correcta a un problema debe considerar el problema como un sistema de símbolos que poseen un álgebra específica y nunca considerar a la aritmética como el álgebra principal
- Procedimentalismo y declarativismo
  - la **programación procedimental** utiliza una descripción del problema basada en la especificación de un conjunto de órdenes o instrucciones (funciones) que, ejecutadas en un orden, conducen a una solución
  - la **programación declarativa** realiza una descripción de los problemas en forma de las relaciones lógico-funcionales de los componentes y los datos del mismo. No especifica la forma de alcanzar la solución, sino la relación que debe existir entre esta y los datos, así que una máquina que incorpore este tipo de programación debe realizar una inferencia de la solución a partir de los datos y de las relaciones especificadas

## 1.5 Distintas definiciones de Sistemas Inteligentes

Un **sistema** es un conjunto de elementos conectados que se organizan para un propósito común. Los **sistemas inteligentes** incluyen no solo dispositivos inteligentes, sino también conjuntos interconectados de tales dispositivos.

- Un sistema inteligente es un programa de ordenador que reúne características y comportamientos asimilables al de la inteligencia humana o animal. Se usa a veces para sistemas inteligentes incompletos, ya que un sistema inteligente completo debe incluir "sentidos" que le permitan recibir información de su entorno, actuar, y disponer de memoria para almacenar el resultado de sus acciones. Además, debe aprender de su experiencia para lograr mejorar su rendimiento y eficiencia.

  Wikipedia

- Un sistema controlado por ordenador y es capaz de responder a cambios del entorno para establecer las condiciones óptimas de funcionamiento sin intervención humana.

  RAE

- Un sistema basado en procedimientos, metodologías o técnicas de la IA, para llevar a cabo de forma más precisa y efectiva operaciones para la resolución de problemas.

  Handbook of Research on Novel Soft Computing Intelligent Algorithms: Theory and Practical Applications

- Sistema que puede imitar y automatizar algunos comportamientos inteligentes humanos. Es una disciplica que estudia el comportamiento inteligente y sus implementaciones, así como su impacto en la sociedad humana.

  Handbook - Research on E-Business Standards and Protocols: Documents, Data and Advanced Web Technologies

- Sistema que incorpora inteligencia en aplicaciones desarrolladas por máquinas. Efectúan búsqueda y optimización, junto con habilidades de aprendizaje. También ejecutan tareas complejas automatizadas, que no son posibles en el paradigma de computación tradicional

  Handbook - Research on Artificial Intelligence Techniques and Algorithms

- Sistema con un conjunto coherente de componentes y subsistemas que trabajan conjuntamente para llevar a cabo actividades dirigidas por un objetivo

  Encyclopedia of Artificial Intelligence

- Sistema que da respuestas apropiadas de resolución de problemas a problemas recibidos como inputs, aunque esos inputs sean nuevos e inesperados. Los trabajos de estos sistemas se suelen describir con analogias con sistemas biológicos. Deben usar la intuición, experiencia pasada, suposiciones, reglas de oro, así como reconocer y correlacionar patrones generales instantáneamente, en lugar de secuencialmente. Estos sistemas pueden aprender y generalizar a partir de situaciones particulares observando el comportamiento actual del sistema. Operadores humanos expertos pueden traducir la incertidumbre de sus proceso mental de razonamiento a acciones de control efectivas y pueden explicar sus acciones, aunque sea de forma imprecisa y mediante características cualitativas. Para que un sistema sea más inteligente debe poder razonar de forma imprecisa e incierta, tal y como hacemos los humanos

  Fuzzy Control of Industrial Systems, theory and applications

# 2 Los problemas y los procesos de búsqueda

No existe un algoritmo determinista general para resolver problemas porque los problemas son complejos, el mundo es cambiante y no totalmente conocido.

Hay técnicas específicas para la resolución de problemas concretos, pero estamos interesados en analizar el conjunto de métodos de resolución de problemas que consideramos más generales.

Para aprender a diseñar **sistemas de resolución de problemas** es necesario analizar los elementos generales de cualquier sistema que permita encontrar soluciones a problemas. Esto es útil para abordar nuevos problemas de manera formalizada y sistemática.

Los elementos principales de los sistemas de resolución de problemas son:

- **Representación**: describe el dominio del problema, el objetivo final que se desea alcanzar y la situación inicial
- **Operadores**: transforman la situación del problema, o dividen un problema en varios subproblemas
- **Estrategia de control**: selecciona el operador a aplicar, en cada situación del problema
- **Solución de un problema**: será una secuencia adecuada de operadores que conduce a su resolución

## 2.1 Representación

Los métodos de representación de problemas pueden estar basados en formulación en el espacio de estados o formulación mediante reducción.

### 2.1.1 Formulación en el Espacio de Estados

Un **estado** es una representación completa de la situación del problema en un momento dado. Los **operadores** son acciones que pueden transformar un estado en otro.

Encontramos distintos tipos de estado:

- **estado inicial**: es la situación inicial del problema
- **estado final (meta, objetivo)**: configuración determinada que representa el objetivo deseado
- **estados intermedios**: son los que se obtienen al aplicar los operadores, partiendo del estado inicial

Un problema representado mediante estados se especifica como una terna $(S, O, G)$ donde $S$ y $G$ son conjuntos de estados iniciales y finales, y $O$ es un conjunto de operadores.

Una **estrategia** define una solución o trayectoria en el espacio de estados. Equivale a la secuencia de operadores utilizadas.

### 2.1.2 Formulación mediante Reducción

Este esquema responde a la posibilidad de que el problema se puede descomponer en subproblemas más pequeños. El principal elemento es la descripción del problema a resolver y la dinámica de resolución consiste en su descomposición en subproblemas más simples.

- **Situación inicial**: definida por la formulación del problema y un conjunto de operadores o transformadores del problema
- **Descripción inicial**: se resuelve por aplicación de una secuencia de transformaciones que dividen el problema en subproblemas, hasta obtener subproblemas sencillos de resolución inmediata

Un operador puede transformar un problema en:

- varios problemas secundarios, que deberán ser todos resueltos y ensamblados para dar solución al problema original. Por ejemplo, A se resuelve si se resuelven B y C.
- varios sub-problemas alternativos, que con encontrar la solución a uno de ellos nos dan la solución al problema original. Por ejemplo, A se resuelve si se resuelve B o se resuelve C.

## 2.2 Características de los problemas

Para escoger la combinación más apropiada de métodos para resolver un problema, debemos analizar el problema desde varias perspectivas:

- **Problemas descomponibles**: son aquellos que pueden ser descompuestos en subproblemas más simples
- **Problemas recuperables**: son aquellos en los que, a partir de la solución, podemos inferir el estado inicial mediante procedimientos de razonamiento
- **Problemas de cualquier o mejor camino**: puede ser que busquemos una solución cualquiera o que busquemos la mejor entre todas las posibles
- **Consistencia y papel del conocimiento**: es útil preguntarse si el conocimiento del que disponemos sobre el problema es consistente, así como determinar si necesitamos una gran cantidad de conocimiento para resolverlo o bien solo es importante para restringir la búsqueda

### 2.2.1 Características de los procesos de búsqueda

Existen muchos problemas para los que no se conoce (o ni siquiera se puede conocer) un algoritmo de resolución.

Un **espacio de estados** puede tratarse como un árbol o un grafo dirigido, donde los **nodos** contienen los estados y los **arcos** representan a los operadores del problema. La **solución del problema** es un camino desde un nodo inicial a uno final.

El número de operadores determina el grado de ramifiación del árbol que representa el problema.

El **nodo raíz** se corresponde con el estado inicial, los **nodos finales** irán asociados con los objetivos o metas. Según la presencia de nodos finales podemos distinguir tres casos:

- No hay nodos finales
- Solo hay un nodo final

- Hay varios nodos finales. En este caso es posible que se deba definir un mecanismo de comparación para elegir la mejor solución

La diferencia entre un grafo y un árbol es que en los grafos diversos caminos pueden llegar a un mismo nodo. El **problema de los árboles** es que, frecuentemente se genera el mismo nodo en caminos distintos, procesándolo más de una vez. El **problema de los grafos** es que pueden aparecer ciclos en la búsqueda, por lo que es más difícil probar que el método termina.

Al buscar en un grafo en lugar de un árbol reducimos el esfuerzo que se invierte explorando el mismo camino varias veces, pero aumenta el esfuerzo cada vez que se genera un nuevo nodo, para ver si ya había sido generado.

La elección depende del problema: si es muy probable que se genere el mismo nodo en varios caminos, es mejor utilizar un grafo. Si los nodos no suelen duplicarse es mejor utilizar un árbol.

La representación por **reducción** también puede tratarse como un árbol o grafo. En esta ocasión se introduce la estructura de **árbol Y/O**, que se caracteriza por:

- cada nodo contiene un problema simple o un conjunto de problemas
- un nodo que no se descompone es un **nodo terminal**
- un nodo terminal con solución inmediata se denomina **primitiva**
- si por cada posible operador se genera un conjunto de subproblemas de solución alternativa, entonces se genera un **nodo O**
- si la aplicación de un operador genera diversos subproblemas, siendo necesaria la resolución de todos ellos, entonces se produce un **nodo Y**

La resolución de un problema representado por un árbol Y/O está asociada con la resolución del nodo raíz. Un nodo será **resoluble** si:

- es un nodo primitiva, o
- es un nodo Y y sus sucesores son todos resolubles, o
- es un nodo O y al menos uno de sus sucesores es resoluble

Si no se verifica alguna de estas condiciones, el nodo será **irresoluble**, o sea, si:

- es un nodo terminal pero no es primitiva, o
- es un nodo Y, y alguno de sus sucesores es irresoluble, o
- es un nodo O, y todos sus sucesores son irresolubles

## 2.3 Razonamiento hacia delante y hacia atrás

- En el **razonamiento hacia delante** se aplican los operadores a la estructura inicial, provocando una transformación de su representación, y se repite el proceso. El problema tendrá solución si es posible alcanzar el objetivo
- En el **razonamiento hacia atrás** se parte de la configuración final y se aplican los operadores inversos al objetivo. El problema tendrá solución si es posible alcanzar la configuración inicial.
- Existe la posibilidad de mezclar ambos enfoques: **búsqueda bidireccional**

## 2.4 Selección de operadores

Es importante determinar qué operador se debe escoger entre los distintos posibles. **Emparejar** es determinar los operadores aplicables a la situación actual. Para ello, se comprueba si se verifican las precondiciones de los operadores.

A menudo, el emparejamiento entre una situación particular y las precondiciones de un operador involucra un proceso complejo de búsqueda.

Una clase sencilla de emparejamiento que puede requerir una búsqueda extensiva surge cuando las precondiciones contienen variables.

## 2.5 Heurísticas. Funciones heurísticas de evaluación

Una **heurística** es una estrategia, método o criterio usado para hacer más sencilla la resolución de problemas difíciles.

El **conocimiento heurístico** es aquel usado por los humanos para resolver problemas complejos.

Una **técnica heurística** es un conjunto de pasos que deben realizarse para identificar una solución de alta calidad con pocos recursos, aunque no podamos garantizar encontrarla. Existen técnicas heurísticas de aplicación muy general y otras que representan conocimientos específicos que son relevantes para la solución de un problema.

La **función heurística** estima lo próximo que se encuentra un estado de un estado objetivo.

## 2.6 La exploración como paradigma de resolución y búsqueda

La **exploración** es un método de resolución tentativo (basado en el criterio de prueba y error), en el que se exige la selección de alguna opción entre un conjunto de posibilidades y no existe un principio determinista para definir esta elección. Es útil ante problemas con las siguientes propiedades:

- se desconoce la posible trayectoria que puede conducir a la solución
- tales trayectorias no se pueden hallar de forma sistemática o, en muchos problemas, su tiempo de cálculo puede exceder lo razonable
- en el mundo real son resueltos por los seres humanos usando principio heurísticos, que por su naturaleza son de difícil justificación

**Inconvenientes:**

- La complejidad, que en algunos métodos crece exponencialmente con el tamaño del problema

**Ventajas**

- Es un método universal de resolución de problemas
- Se han desarrollado métodos para incorporar conocimiento heurístico adecuado, lo que tiene como consecuencia que aumente su eficacia

# 3 Estrategias básicas de búsqueda

Para resolver un problema debemos:

- **establecer un objetivo**: estados que satisfacen los criterios para ser meta o subproblemas establecidos como primitivos
- **aplicar un método de búsqueda**: proceso para examinar y encontrar qué secuencia de accioner permite obtener un estado objetivo o dar como resuelto el problema

En general, un diseño simple de este proceso debe "formular, buscar, ejecutar":

```
función RESOLVER-PROBLEMAS(percepción) devuelve una acción

entradas: percepción
estático: problema: una formulación del problema
                sec: una secuencia de acciones
                estado-actual
                objetivo

repetir
        problema <- FORMULAR-PROBLEMA()
        inicio <- FORMULAR-INICIO()
        objetivo <- FORMULAR-OBJETIVO()
        sec <- BUSQUEDA(problema)
        accion <- PRIMERO(sec)
        devolver accion
        estado-actual <- ACTUALIZAR(estado-actual, percepción)
```

Se presupone un entorno estático porque la formulación y la búsqueda del problema se hace sin prestar atención a los cambios que pueden ocurrir en el entorno. También que se conoce el estado inicial. Que el entorno es discreto, o sea, que se pueden enumerar las líneas de acción alternativas, y que el entorno es determinista, es decir, que las soluciones son secuencias de acciones y se ejecutan sin prestar atención a las percepciones.

Independientemente de la representación utilizada (por espacio de estados o reducción) la estructura subyacente será un árbol, un grafo dirigido o un Y/O, descrito por los nodos y los arcos. Los nodos serán estructuras de 5 componentes, como mínimo:

- estado o subproblema: identifica al nodo
- nodo padre: nodo en el árbol de búsqueda que ha generado este nodo
- acción: el operador que se aplicará al padre para generar el nodo
- costo del camino: el costo $g(n)$ de un camino, indicado por los punteros a los padres
- profundidad: el número de pasos a lo largo del camino

En el proceso de búsqueda, definimos dos conjuntos de nodos relevantes:

- **Frontera:** colección de nodos generados pero que aún no han sido expandidos
- **Cerrados:** colección de nodos generados y expandidos

La estrategia de búsqueda será un función que selecciona de la frontera el siguiente nodo a expandir.

La colección de nodos se implementará como una cola con las siguientes operaciones:

- VACIA(cola): true si cola está vacía
- BORRAR(cola,criterio): elimina de la cola los elementos que verifican el criterio
- BORRAR-PRIMERO(cola): pop
- INSERTA(elemento,cola)
- INSERTA-TODO(elementos,cola)

Se denomina **abstracción** al proceso de eliminar detalles innecesarios de una representación. Es importante para resolver problemas porque debemos simplificar un problema lo máximo posible, hasta quedarnos únicamente con el problema en sí mismo.

#### Midiendo el rendimiento de la resolución del problema

Un método de resolución de problemas devuelve bien una solución, bien un fallo. Para evaluar su rendimiento, atendemos a cuatro dimensiones:

- **completitud**: se pregunta sobre las garantías de que, existiendo solución, se encuentre
- **optimalidad**: sobre la obtención de la solución óptima
- **complejidad en tiempo**: cuánto tarda en encontrar la solución
- **complejidad en espacio**: cuánto espacio necesita el método

La complejidad se suele considerar con respecto a alguna medida de la dificultad del problema. Debido a la estructura de árbol de búsqueda, normalmente expresaremos la complejidad en términos del **factor de ramificación**, $b$, la **profundidad del nodo objetivo más superficial**, $d$, y la **longitud máxima**, $m$, de cualquier camino en el espacio de búsqueda.

## 3.1 Estrategia sobre una representación por espacio de estados

### 3.1.1 Búsqueda no informada o a ciegas

No se hace uso de información específica del problema o del dominio concreto de este. En esta situación se adopta algún criterio arbitrario, pero fijo.

Un problema de búsqueda consta de cuatro componentes:

1. El estado inicial
2. Las posibles acciones: la formulación más común es
    - SUCESOR(x): devuelve conjuntos de pares ordenados <acción, sucesor>, que indican que si realizamos "acción", obtendremos "sucesor". Implícitamente, el estado inicial y esta función definen el espacio de estados.

Un **camino** es un conjunto de estados conectados por una secuencia de acciones.

3. El test objetivo, que determinará si un estado es un estado objetivo o no
4. Una **función costo**, que asigna un coste numérico a cada camino:
   (a) **coste individual**, coste no negativo de una acción $a$ de $x$ a $y$, $c(x, y) = c(x, a, y)$
   (b) **coste del camino**, suma de los costes individuales de los nodos que forman el camino

   La calidad de la solución se mide por la función costo del camino. Una solución óptima será aquella con el costo más pequeño entre todas las soluciones (minimización).

**Búsqueda sobre árboles**

```
función BUSQUEDA_ARBOLES(problema, frontera) devuelve {solución o fallo}
{
frontera <- INSERTA(HACER_NODO(ESTADO_INICIAL(problema)), frontera)

bucle hacer
        si VACIA(frontera) entonces devolver fallo
        nodo <- BORRAR_PRIMERO(frontera)
        si TEST_OBJETIVO(problema, ESTADO(nodo)) entonces devolver
           SOLUCION(nodo)
        frontera <- INSERTAR_NODO(EXPANDIR(nodo, problema), frontera)
}

función EXPANDIR(nodo, problema) devuelve Conjunto<nodos>
{
sucesores <- conjunto_vacío

para cada (accion, resultado) en SUCESOR(problema, ESTADO(nodo)) hacer
        s <- nuevo nodo
        s.estado <- resultado
        s.padre <- nodo
        s.accion <- accion
        s.costo <- nodo.costo + COSTO_INDIVIDUAL(nodo, accion, s)
        s.profundidad <- nodo.profundidad + 1
        sucesores <- ADD(s, sucesores)

devolver sucesores
}
```

Las estrategias se distinguen por el orden de expansión de los nodos.

**Búsqueda primero en anchura**

Se implementa llamando a BUSQUEDA-ARBOLES(problema, COLA-FIFO()). Se expanden todos los nodos de un nivel, antes de expandir los del siguiente.

- completa si $b$ es finita
- el nodo objetivo más superficial no es necesariamente el óptimo, es óptima si la función coste es creciente con el nivel de profundidad
- suponiendo un factor de ramificación constante, se generan

$$b + b^2 + ... + b^d + \left(b^{b+1} - b\right) = O\left(b^{d+1}\right)$$

nodos. Esta es tanto la complejidad de tiempo como de espacio

**Búsqueda de costo uniforme**

Se implementa llamando a BUSQUEDA-ARBOLES(problema, frontera_O()), donde frontera_O() está ordenado por el costo $g$, de menor a mayor.

- completa si $g(x, y) \geq \varepsilon > 0$, $\forall x, y$ con $y$ descendiente de $x$. Ya que, si hubiera alguna acción con coste 0, que deriva en el mismo camino del que partimos, obtendríamos un bucle infinito. Cuando todos los costes son mayores que una cierta cantidad, los costes siempre aumentan y esto no puede suceder.
- en tal caso también es óptima
- la complejidad en tiempo y espacio es $O\left(b^{\left\lceil \frac{C^*}{\varepsilon} \right\rceil}\right)$, donde $C^*$ es el costo de la solución óptima

**Búsqueda primero en profundidad**

Se implementa llamando a BUSQUEDA-ARBOLES(problema, COLA_LIFO()). Ahora expandimos el nodo más profundo en la frontera, antes de proceder con los demás.

- no es completa en general
- por tanto tampoco puede ser óptima
- la complejidad en tiempo es $O(b^m)$
- la complejidad en espacio es $O(bm) \sim O(m)$

**Búsqueda de profundidad limitada**

Es una búsqueda primero en profundidad, pero con un límite de profundidad $l$ predeterminado. Este límite resuelve un problema de la BPP, la generación de un camino infinito que no lleva a ninguna solución, pero introduce una fuente adicional de incompletitud si $l < d$, pues en tal caso no vamos a llegar suficientemente profundo como para encontrar la solución.

- completa si $l \geq d$
- no es óptima, podría haber mejores soluciones por debajo de $l$
- complejidad en tiempo $O\left(b^l\right)$
- complejidad en espacio $O(bl) \sim O(b)$

**Búsqueda primero en profundidad con profundidad iterativa**

Es una búsqueda en profundidad limitada, pero en la que aumentamos el límite hasta encontrar el objetivo:

```
función BUSQUEDA-PROFUNDIDAD-ITERATIVA(problema) devuelve {solución o
    fallo}

entradas: problema

para profundidad <- 0 a infinito hacer
        resultado <- BUSQUEDA-PROFUNDIDAD-LIMITADA(problema, profundidad)
        si TEST-OBJETIVO(resultado, problema) entonces devolver
            resultado
```

- completa
- óptima
- complejidad en tiempo $O\left(b^d\right)$
- complejidad en espacio $O\left(bd\right) \sim O\left(b\right)$

Si $b$ es constante, generar nodos repetidamente no es muy costoso.

**Comparación de las estrategias de búsqueda no informada**

| Criterio | BPA | Costo Uniforme | BPP | BPPL | BPPIt |
|---|---|---|---|---|---|
| **Completa** | Sí* | Sí* | No | No | Sí* |
| **Optima** | Sí* | Sí | No | No | Sí* |
| **Tiempo** | $O\left(b^{d+1}\right)$ | $O\left(b^{\left\lceil \frac{C^*}{\varepsilon} \right\rceil}\right)$ | $O\left(b^m\right)$ | $O\left(b^l\right)$ | $O\left(b^d\right)$ |
| **Espacio** | $O\left(b^{d+1}\right)$ | $O\left(b^{\left\lceil \frac{C^*}{\varepsilon} \right\rceil}\right)$ | $O\left(bm\right)$ | $O\left(bl\right)$ | $O\left(bd\right)$ |

**Evitar estados repetidos → Búsqueda sobre grafos**

Para algunos problemas, los árboles de búsqueda son infinitos. Si no se detectan los estados repetidos se puede provocar que un problema resoluble llegue a ser irresoluble. Para ello:

- creamos una nueva estructura de datos que almacene loos nodos expandidos y que no acepta repetidos
- en problemas con muchos estados repetidos, la búsqueda en grafos es más eficiente que la búsqueda en árboles

El pseudocódigo sería:

```
función BUSQUEDA_GRAFOS(problema, frontera) devuelve {solución o fallo}
{
cerrado <- conjunto-vacio
frontera <- INSERTAR(HACER_NODO(ESTADO_INICIAL(problema)),frontera)

bucle hacer
        si VACIA(frontera) entonces devolver fallo
        nodo <- BORRAR_PRIMERO(frontera)
        si TEST_OBJETIVO(ESTADO(nodo), problema) entonces devolver
           SOLUCION(nodo)
        su ESTADO(nodo) \notin cerrado entonces
                cerrado <- ADD(cerrado,ESTADO(nodo))
                frontera <- INSERTAR_TODO(EXPANDIR(nodo,problema),
                   frontera)
}
```

Podemos simplificar, evitando introducir en la frontera nodos ya expandidos o ya presentes, con menor coste. Es más eficiente en memoria pero necesita más comprobaciones.

### 3.1.2 Búsqueda Heurística o informada

Las estrategias de búsqueda no informada encuentran soluciones generando sistemáticamente nuevos estados y probándolos con el test objetivo. Esto es muy ineficiente en problemas complejos. Como alternativa, podríamos intentar diseñar algún mecanismo que permita dirigir la búsqueda hacia las zonas más prometedoras, y encontrar soluciones de una forma más eficiente.

Las **heurísticas** son criterios, reglas, puntuaciones o métodos que ayudan a decidir cuál es la mejor alternativa o la más prometedora entre varias posibles, para alcanzar un determinado objetivo. Este mecanismo requiere disponer de conocimiento sobre el problema. Se utilizan para decidir qué nodo expandir a continuación, qué nodos sucesores son generadores y qué nodos no deben tenerse en cuenta.

Las **estrategias de búsqueda heurística**:

- utilizan algún mecanismo heurístico para dirigir la búsqueda hacia las zonas más prometedoras y encontrar soluciones de una manera más eficiente
- existen diversas estrategias, basadas en los métodos básicos y subyacentes para la búsqueda no informada
- pueden encontrar soluciones de una forma más eficiente

#### Búsqueda Primero el Mejor

Es la estrategia general y más básica de búsqueda informada. Corresponde a un caso particular del método general de BUSQUEDA-ARBOLES/GRAFOS.

Hace uso de una **función de evaluación** $f(n)$ que, a cada nodo $n$, le asigna un valor numérico que indique lo prometedor que es el nodo para ser expandido. Este valor se utiliza para ordenar la lista

frontera, de tal forma que los nodos más prometedores se encuentren al principio. Se suele interpretar esta función como una distancia al objetivo, por lo que se considera que

$$n \text{ es más prometedor que } m \iff f(n) \leq f(m)$$

Puede implementarse con una cola con prioridad, de orden ascendente con $f$-valores (las evaluaciones mediante $f$).

Un componente clave es la creación de una **función heurística** $h(n)$, que indicará el coste estimado del camino más barato desde el nodo $n$ a un nodo objetivo y, si $n$ es un nodo objetivo, entonces verificará $h(n) = 0$. Este tipo de funciones representan la manera más común de transmitir el conocimiento adicional del problema al método de búsqueda.

### Búsqueda primero el mejor greedy (avara o voraz)

Expande el nodo que estima más cercano al objetivo. Para ello, se establece

$$f(n) = h(n)$$

- Complejidad en tiempo y espacio: $O(b^m)$. Aunque con una buena $h$ se reduce la complejidad considerablemente.

### Búsqueda A*

En esta ocasión consideramos que está bien tener en cuenta tanto la distancia estimada al objetivo, como el coste del camino recorrido hasta ahora, y trataremos de minimizar la suma de ambas cantidades, es decir, usaremos

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

donde $g(n)$ es el coste del camino desde el inicio hasta $n$ y $h(n)$ el coste estimado desde $n$ hasta el objetivo. Por tanto, $f(n)$ representa el coste estimado desde el inicio hasta el objetivo pasando por $n$.

Vamos a ver ahora que si $h$ satisface ciertas propiedades, A* será completa y óptima.

**Definición**
Una heurística $h(n)$ es **admisible** si nunca sobrestima el coste real de alcanzar el objetivo desde un nodo $n$, $h^*(n)$, o sea

$$h \text{ admisible} \iff h(n) \leq h^*(n), \ \forall n$$

Esta propiedad implica que, al estimar el coste del camino verdadero $C^*(n) = g(n) + h^*(n)$ mediante $f(n) = g(n) + h(n)$, se tendrá

$$f(n) \leq C^*(n)$$

es decir, el coste estimado de alcanzar una solución pasando por $n$, nunca será mayor que el coste real de una solución que pasa por $n$.

Las heurísticas no admisibles son pesimistas, puesto que pueden descartar un sucesor porque piensan que es peor de lo que realmente es, lo que impide la optimalidad.

Por otro lado, las heurísticas admisibles son optimistas, suponen que el coste de resolver un problema es menor del coste real. Esto permite la optimalidad, un nodo del camino óptimo no puede parecer tan malo como para descartarlo. Así, tenemos el siguiente teorema, que asegura la **optimalidad de A* para BUSQUEDA-ARBOLES**:

**Teorema**
Si $h$ es una heurística admisible, entonces A* guiada por $h$ es óptima.

PROOF Sea $C^*$ el coste de la solución óptima y supongamos que tenemos en la frontera un nodo objetivo subóptimo, $G$.

Entonces, como $G$ es subóptimo y $h(G) = 0$, por ser este objetivo, se verifica

$$f(G) = g(G) + h(G) = g(G) > C^*$$

Consideremos, pues, un nodo $n$ en la frontera sobre un camino solución óptimo, es decir, que conduce a una solución con coste $C^*$. Entonces, como $h$ es admisible

$$f(n) = g(n) + h(n) \leq g(n) + h^*(n) = C^*$$

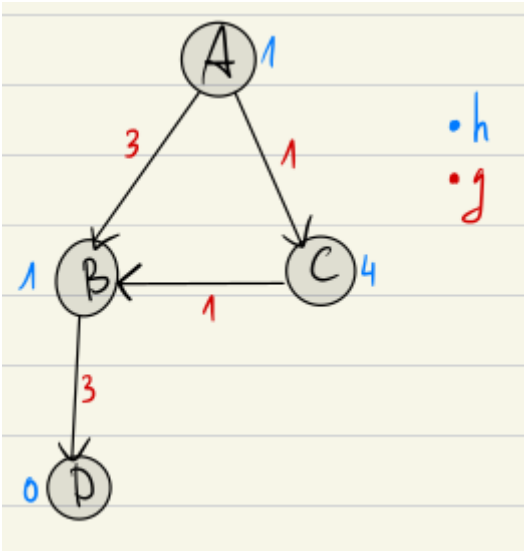
Por tanto, lo que obtenemos es que

$$f(n) \leq C^* < f(G)$$

y A* escogerá $n$ en lugar de $G$. Por lo que, en última instancia, devolverá la solución óptima.

¿Por qué es necesario que sea un árbol?

La respuesta es que los árboles generan una sola vez cada nodo, mientras que los grafos pueden crearlos varias veces y, si la función heurística solo verifica admisibilidad, podemos obtener soluciones subóptimas. Veamos un contraejemplo:




| $n$ | $h(n)$ | $h^*(n)$ |
|---|---|---|
| A | 1 | 5 |
| B | 1 | 3 |
| C | 4 | 4 |
| D | 0 | 0 |

En esta tabla vemos como $h$ es admisible, pues $h(n) \leq h^*(n)\,,\ \forall n$, y se ve claro que la solución óptima es A-C-B-D, sin embargo, A* seguiría el siguiente proceso:

| Frontera | Cerrados |
|---|---|
| $A(1)$ | - |
| $B_A(4), C_A(5)$ | $A(1)$ |
| $C_A(5), D_B(6)$ | $A(1), B_A(4)$ |
| $D_B(6)$ | $A(1), B_A(4), C_A(5)$ |
| - | $A(1), B_A(4), C_A(5), D_B(6)$ |

Y nos da el camino subóptimo A-B-D. La clave se enceuntra en que, al cerrar $C$, detecta que $B$ ya está cerrado y no lo expande.

Tenemos dos posibilidades para abordar este obstáculo:

- Extender la BUSQUEDA-GRAFOS para que deseche el camino más caro cuando vuelve a generar un nodo ya cerrado. Esto añade cálculos complementarios pero garantiza optimalidad
- Exigir alguna propiedad, más estricta que la admisibilidad, a la heurística, de forma que podamos asegurar que el camino óptimo a cualquier estado repetido es siempre el primero que seguimos

**Definición**
Una heurística $h$ es **consistente** si, y solo si,

$$h(n) \leq K\left(n, a_s, n'\right) + h\left(n'\right)\,,\ \forall n, n'$$

donde $K(n, a_s, n')$ es el coste del camino de $n$ a $n'$ a través de la secuencia de acciones $a_s$.
Es decir, es consistente si la estimación dada por $h$ no es mayor que el coste de ir de $n$ a otro nodo cualquiera más la estimación del coste desde ese otro nodo a la solución. Es una suerte de desigualdad triangular.

Nótese que esta condición es más estricta que la admisibilidad. De hecho, fijémonos en el ejemplo anterior: hemos visto que $h$ era admisible, no obstante, nótese que $h(C) = 4$ y que
$K(C, B) + h(B) = 1 + 1 = 2$, luego no es consistente

**Definición**
Una heurística $h$ es **monótona** si, y solo si,

$$h(n) \leq c\left(n, a, n'\right) + h\left(n'\right)\,,\ \forall n, n',\ n' \text{ sucesor de } n$$

donde $c(n, a, n')$ es el coste del arco que une $n$ con $n'$.
O sea, es monótona si la estimación dada por $h$ no es mayor que el coste de ir a cualquier vecino más la estimación del coste desde ese vecino a la solución.

Notemos ahora que la monotonía parece una condición más laxa que la consistencia, pues debemos comprobar la misma condición, pero para menos nodos. No obstante, se puede demostrar que

$$h \textit{ consistente} \iff h \textit{ monótona}$$

lo cual facilita comprobar la consistencia, al solo necesitar comprobar la monotonía.

PROOF [ $\Longrightarrow$ ] Obvio, si se verifica para cualquier nodo $n'$ del grafo, en particular se verifica para los vecinos de $n$

[ $\Longleftarrow$ ] Si el camino para llegar desde $n$ hasta $n'$ es

$$n = n_0, n_1, n_2, ..., n_{k-1}, n_k = n'$$

entonces, $K(n, a_s, n') = \sum_{i=1}^{k} c(n_{i-1}, a_{i-1,i}, n_i)$ y, por ser $h$ monótona, se tiene que

$$h(n) = h(n_0) \leq c(n_0, a_{0,1}, n_1) + h(n_1) \leq c(n_0, a_{0,1}, n_1) + c(n_1, a_{1,2}, n_2) + h(n_2) \leq ... \leq$$

$$\leq c(n_0, a_{0,1}, n_1) + c(n_1, a_{1,2}, n_2) + ... + c(n_{k-1}, a_{k-1,k}, n_k) + h(n_k) =$$

$$= \sum_{i=1}^{k} c(n_{i-1}, a_{i-1,i}, n_i) + h(n') = K(n, a_s, n') + h(n')$$

por lo que $h$ es consistente.

**Teorema**
Si $h$ es una heurística consistente, entonces $h$ es admisible

PROOF Sea un nodo $n$ y $G$ el objetivo más cercano.

Como suponemos que $h$ es consistente, entonces

$$h(n) \leq K(n, a_s, G) + h(G)$$

ahora bien, como $G$ es objetivo, entonces $h(G) = 0$ y como es el más cercano a $n$, entonces $K(n, a_s, G) = h^*(n)$, por tanto

$$h(n) \leq h^*(n) + 0 = h^*(n)$$

y $h$ es admisible

Observemos que lo contrario no es cierto, como ya hemos visto en el ejemplo anterior.

De esta forma, podemos interpretar la búsqueda como una exploración en un mapa con curvas de nivel, donde el estado inicial está en el centro, y se van recorriendo los nodos concéntricamente hasta agotar todos los de una misma zona de nivel. Entonces se pasa a la siguiente, y así hasta alcanzar el objetivo más cercano.

Tenemos, por tanto, que:

- Si $h$ es consistente, entonces los valores de $f(n)$ a lo largo de cualquier camino no disminuyen. Eso quiere decir que la secuencia de nodos expandidos por A* utilizando BUSQUEDA-GRAFOS está en orden no decreciente de $f(n)$, por lo que no será posible encontrar un camino de coste menor para un nodo ya cerrado.
- Por tanto, si $h$ es consistente, entonces A* es óptimo para BUSQUEDA-GRAFOS:

PROOF Supongamos que, en la frontera, se encuentra un nodo objetivo subóptimo, $G$, con $f(G) = g(G) + h(G) = g(G) > C^*$, donde $C^*$ es el coste óptimo. Sea $n$ un nodo de la frontera, tal que pertenece al camino hacia la solución óptima, $G^*$. Entonces, como $h$ es consistente, tendremos

$$f(n) = g(n) + h(n) \leq g(n) + K(n, a_s, G^*) + h(G^*)$$

pero $g(n) + K(n, a_s, G^*) = C^*$, pues estamos suponiendo $G^*$ el nodo solución óptimo, y que $n$ está en el camino óptimo hacia $G^*$. Por tanto

$$f(n) \leq C^* + h(G^*) = C^* < g(G) = f(G)$$

luego elegiremos $n$, antes de elegir $G$.

¿Existe siempre tal nodo $n$?

Si $G^*$ es alcanzable desde el nodo inicial, entonces hay un camino óptimo desde el nodo inicial hasta $G^*$, luego sí.

¿$g(n)$ es el coste sobre el camino óptimo? Para que la demostración fuese completa, deberíamos ver que también se verifica esto.

De esta forma, vemos que A* es

- completa
- óptima
- óptimamente eficiente entre todos los métodos

No obstante, A* no es ideal. La dificultad se encuentra en que, para la mayoría de los problemas, la cantidad de nodos en cada banda delimitada por las curvas de nivel es aún exponencial en la longitud de la solución. Y este no es el principal problema. Como A* mantiene todos los nodos generados en memoria, por lo general se queda sin espacio mucho antes que sin tiempo.

Se han desarrollado otros métodos que sacrifican un poco de tiempo para solventar los problemas de espacio, sin sacrificar la optimalidad ni la completitud.

### Búsqueda Primero el Mejor Recursiva (BPMR)

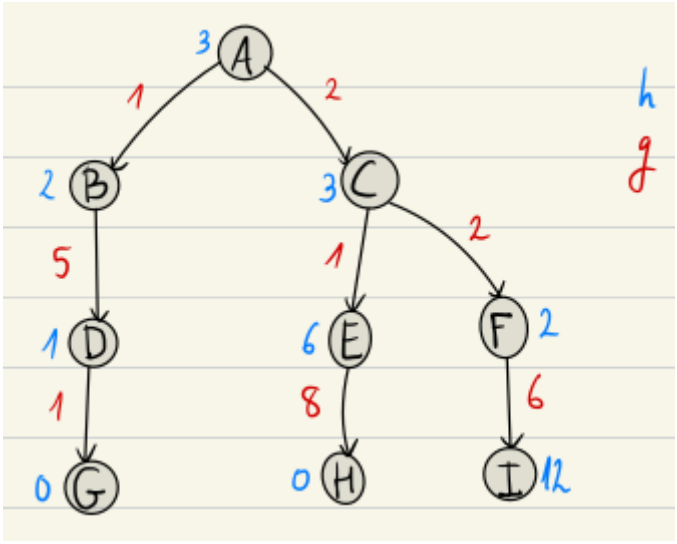
Su objetivo es limitar el consumo de memoria en la búsqueda heurística. Para ello, imita a Primero el Mejor, pero genera menos nodos y el consumo de memoria es lineal. Evita profundizar sin límite por un camino que podría empezar a empeorar.

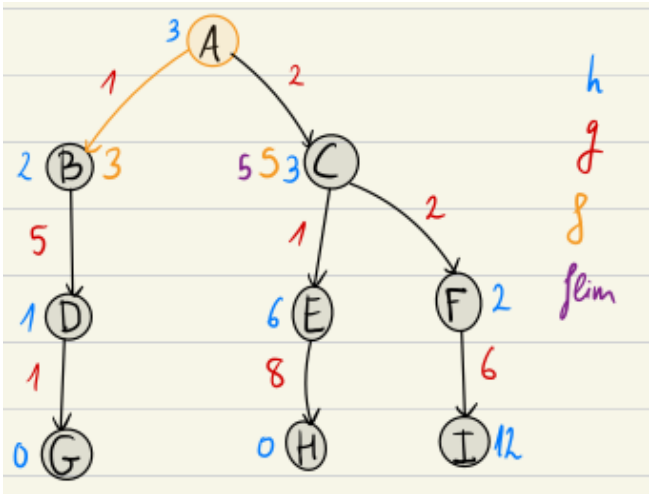
- Utiliza una variable $f_{limite}$ para recordar el $f - valor$ del mejor camino alternativo desde cualquier ancestro del nodo actual
- si el nodo actual excede este límite, la recursión retrocede al camino alternativo, descargando el subárbol actual
- al retroceder, el método reemplaza el $f - valor$ de cada nodo a lo largo del camino con un valor de respaldo: el mejor $f - valor$ de sus hijos

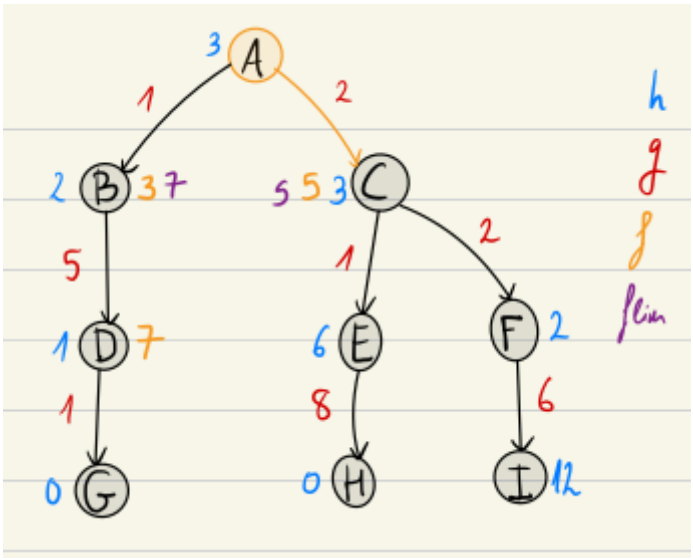
El pseudocódigo queda:

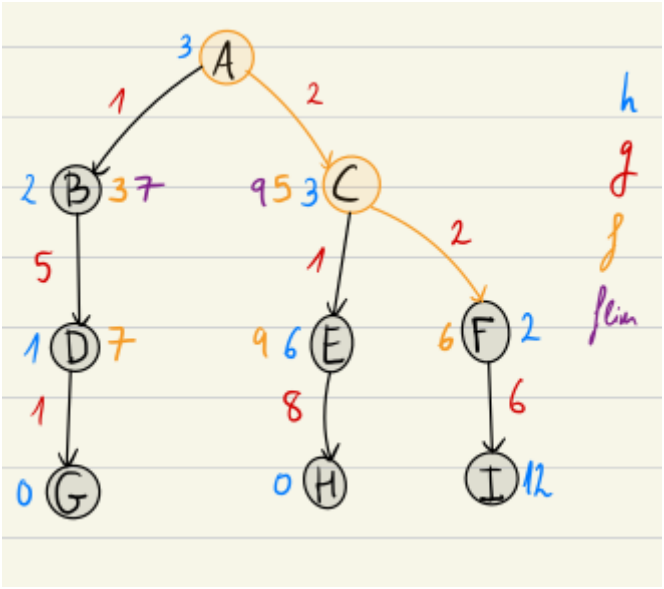
```
función BPMR(problema) devuelve {solución o fallo}
{
BPMR-f(problema, HACER-NODO(ESTADO-INICIAL(problema)),infinito)
}

función BMPR-f(problema, nodo, flimite) devuelve {solucion o fallo o
    flimite_nuevo}
{
si TEST-OBJETIVO(problema, nodo) entonces devolver nodo
sucesores <- EXPANDIR(nodo, problema)

si VACIO(sucesores) entonces devolver (fallo,infinito)
para cada s en sucesores hacer s.f <- max(g(s)+h(s), nodo.f)

repetir
        mejor <- nodo tal que nodo.f = min(s.f : s en sucesores)
        si mejor.f > flimite entonces devuelve (fallo,mejor.f)
        alernativa <- nodo tal que nodo.f es el segundo más bajo entre
            los sucesores
        resultado,mejor.f <- BMPR-f(prolema,mejor,min(flimite,
            alternativa))

        si resultado != fallo entonces devolver resultado
}
```

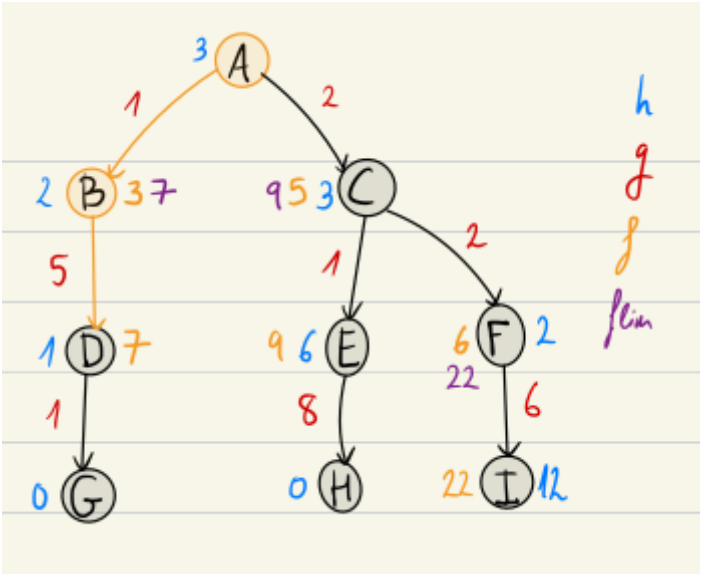
Y como ejemplo:

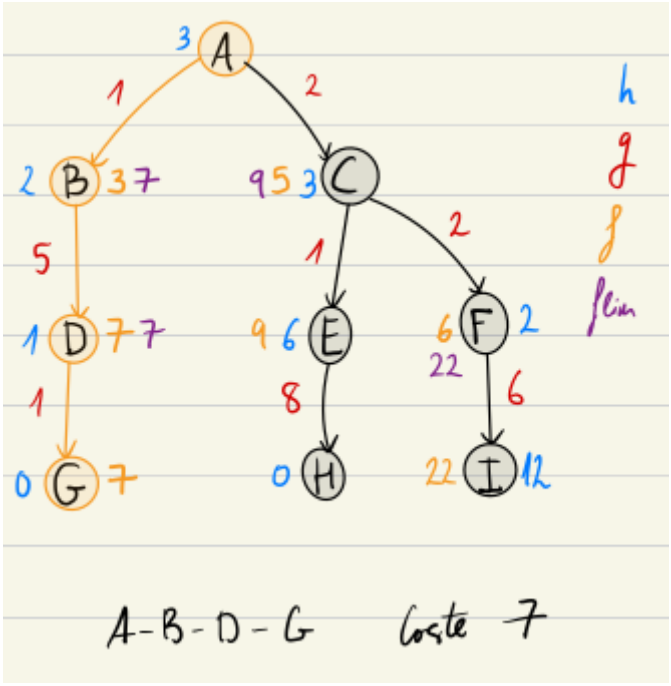



A
B
C
D
E
F
G
H
I
h
g
g
f=g+h


A
B
C
D
E
F
G
H
I
h
g
g
f=g+h


A
B
C
D
E
F
G
H
I
h
g
g
f=g+h




BMPR es óptimo si $h$ verifica las mismas condiciones que en A* (admisibilidad en árboles y consistencia/monotonía en grafos).

Su complejidad en espacio es $O(bd)$ y en tiempo es complicado de caracterizar, depende de la función heurística, y de cómo de a menudo cambia el mejor camino mientras se expanden los nodos.

Así, si hubiera más memoria disponible, sería sensato aplicar un método que use toda la memoria disponible, a cambio de disminuir la complejidad en tiempo.

Uno de estos métodos es **A* con memoria acotada simplificada (A*MS)** que avanza como A* hasta que se llene la memoria. En este momento, no se puede añadir un nuevo nodo al árbol de búsqueda sin retirar uno viejo. A*MS retira el peor nodo hoja, y, como BPMR, devuelve hacia atrás, a su padre, el valor del nodo olvidado. Así, A*MS vuelve a generar ese subárbol solo cuando todos los demás caminos parecen peores que el olvidado.

A*MS es completo si hay alguna solución alcanzable ($d$ menor que el tamaño de memoria). Es óptimo si cualquier solución óptima es alcanzable.

En términos prácticos, A*MS podría ser el mejor método de uso general para encontrar soluciones óptimas.

#### Suboptimalidad controlada: heurísticas e-admisibles

Una forma de perder la optimalidad, pero de una forma controlada es permitir incrementar en una cantidad $e$ el salto de la frontera de optimalidad, de tal forma que, si

$$h^*(n) < h(n) \implies h(n) \leq h^*(n) + e$$

Este tipo de heurísticas se denominan ***e*-admisibles**. Son importantes porque dan lugar a **soluciones *e*-óptimas**, es decir, soluciones que no son óptimas pero por tan solo una cantidad $e$.

## 3.2 Estrategias sobre una representación por Reducción

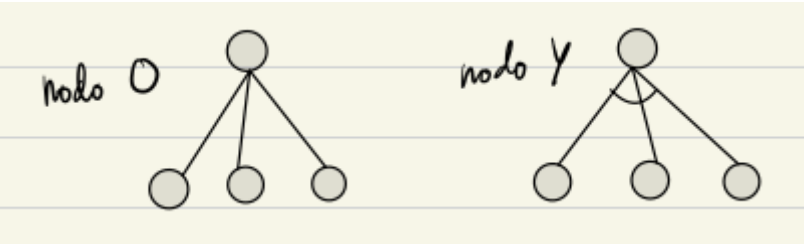
### 3.2.1 Búsqueda no informada o a ciegas

Utilizaremos un árbol de búsqueda Y/O. Para ello, especificaremos:

- un nodo inicial
- un conjunto de nodos terminales resolubles (primitivas)
- el conjunto de operadores para reducir el problem en subproblemas

Los **nodos O** representan operadores que dividen un problema en subproblemas alternativos, representados por arcos dirigidos desde el nodo padre a cada nodo sucesor. Para resolver el nodo padre, basta resolver cualquiera de los sucesores.

Los **nodos Y** representan operadores que dividen un problema en un conjunto de subproblemas, queda representado por un conjunto de arcos que uno el nodo padre con el conjunto de sucesores. Para resolver el nodo padre, es necesario resolver todos los nodos sucesores.




Una **solución al problema** original está dada por un subárbol del árbol Y/O suficiente para demostrar que el nodo comienzo es resoluble.

Los métodos de búsqueda en problemas de reducción tienen en común con los métodos de búsqueda en espacio de estados que la operación de expandir sigue presente, y que los distintos métodos difieren principalmente en el orden de expansión.

Vamos a hacer dos suposiciones simplificadoras:

- el espacio de búsqueda será un árbol Y/O, no un grafo Y/O general

- cuando un problema se transforma en un conjunto de subproblemas, estos pueden ser resueltos en cualquier orden (los subproblemas son independientes entre sí)

El pseudocódigo queda:

```
función BUSQUEDA-ARBOLES-YO(problema, frontera) devuelve {solucion o
   fallo}
{
frontera <- INSERTA(HACER-NODO(PROBLEMA-INICIAL(problema)),frontera)
bucle hacer
        si VACIA(frontera) entonces devolver fallo
        nodo <- BORRAR-PRIMERO(frontera)
        frontera <- INSERTAR-TODO(EXPANDIR-YO(nodo, problema), frontera)
        si VACIO(EXPANDIR-YO(nodo, problema)) entonces
                nodo <- irresoluble
                si (nodo irresoluble => algún ascentro irresoluble)
                   entonces ancestros <- irresolubles
                si (PROBLEMA-INICIAL(problema) irresoluble) entonces
                   devolver fallo
                BORRAR(frontera, {nodos con ascentros irresolubles})
        en otro caso si (se generan nodos resolubles) entonces
                etiquetar esos nodos como resolubles
                si (esos nodos resolubles => algún ascentro resoluble)
                   entonces ancestros <- resolubles
                si (PROBLEMA-INICIAL(problema) resoluble) entonces
                   devolver SOLUCION
                BORRAR(frontera, {nodos etiquetados resolubles o con
                   ascentros resolubles})
}

funcion EXPANDIR-YO(nodo, problema) devuelve set<nodo>
{
SUC <- {}
para cada sucesor m de nodo
        si m es un conjunto de subproblemas entonces
                SUC <- INSERTAR-TODO(sucesores de m, SUC)
        en otro caso SUC <- INSERTAR(m, SUC)
devolver SUC
}
```

**Búsqueda Primero en Anchura en Árboles Y/O**

Se implementa llamando a BUSQUEDA-ARBOLES-YO(problema, COLA-FIFO()).

**Búsqueda Primero en Profundidad en Árboles Y/O**

Se implementa llamando a BUSQUEDA-ARBOLES-YO(problema, COLA-LIFO())

### Búsqueda de Profundidad Limitada en Árboles Y/O

Se puede aliviar el problema de los árboles ilimitados aplicando la búsqueda primero en profundidad con un límite de profundidad predeterminado. Al igual que ocurría en espacio de estados, el límite introduce una fuente de incompletitud, puede que no profundicemos suficiente para resolver el problema principal.

### 3.2.2 Búsqueda Heurística o Informada

La presencia de nodos Y añade complicaciones conceptuales. Es útil ver los grafos YO como hipergrafos. Pasamos de tener arcos que conectan pares de nodos $(a, b)$ a tener hiperarcos, que conectan un nodo antecesor con un conjunto de nodos sucesores $(a, \{b_1, ..., b_k\})$, o ***k*-conectores**, con $k$ el cardinal del conjunto de sucesores.

Podemos definir un grafo solución de forma precisa si suponemos que nuestros grafos YO no poseen ciclos (son árboles), de forma recursiva:

**Definición**
$G'$ es un **grafo solución** desde el nodo $n$ a un conjunto $N$ de nodos de un grafo YO, $G$ si:

- $G'$ es un subgrafo de $G$
- Si $n \in N \implies G' = \{n\}$
- Si $n \notin N$, entonces
  - si $n$ tiene un $k$-conector, $(n, \{n_1, ..., n_k\})$ tales que hay un grafo solución hasta $N$ desde cada $n_i,\ i = 1, ..., k$, se verifica que $G'$ consta de:
    - nodo $n$
    - el $k$-conector
    - los nodos $n_1, ..., n_k$
    - los grafos solución a $N$ desde cada nodo $n_1, ..., n_k$
  - si no, no hay grafo solución desde $n$ a $N$

Como hicimos anteriormente, es útil asignar costes a los conectores de los grafos Y/O, que serán utilizados para calcular el coste de un grafo solución.

Sea $K(n, N)$ el coste de un grafo solución desde el nodo $n$ hasta $N$. Entonces:

- si $n \in N \implies K(n, N) = 0$
- si $n \notin N$, entonces, si $n$ tiene un $k$-conector $(n, \{n_1, ..., n_k\})$ en el grafo solución, sea $c_n$ el coste de ese conector, entonces

$$K(n, N) = c_n + \sum_{i=1}^{k} K(n_i, N)$$

### Características de las funciones de evaluación en grafos Y/O

Estaremos interesados, normalmente, en encontrar el grafo solución que tenga costo mínimo, o sea, el **grafo solución óptimo**. Para ello, dispondremos de una función heurística $h$ para estimar el costo.

Para buscar en un grafo Y/O, es necesario realizar cuatro acciones en cada paso:

- atravesar el grafo empezando por el nodo inicial y siguiendo el mejor camino actual, acumulando el conjunto de nodos que van en ese camino y no se han expandido
- elegir uno de esos nodos no expandidos y expandirlo. Añadir sus sucesores al grafo y calcular $h$ para cada uno de ellos
- cambiar la $h$ estimada del nodo recientemente expandido para reflejar la nueva información proporcionada por sus sucesores. Propagar este cambio hacia atrás a través del grafo
- para cada nodo que se visita mientras se avanza en el grafo, debemos decidir cuál de sus conectores es más prometedor y marcarlo como parte del mejor grafo solución parcial actual

Así, en el nodo raíz de la estructura aparecerá el coste del grafo solución actual, y una marca indicando el operador que genera ese grafo solución.

**Estrategia Y/O***

No utiliza las dos listas frontera y cerrados, sino una única estructura $G$, que representa la parte del grafo de búsqueda que se ha generado explícitamente hasta el momento, denominado **grafo de exploración**.

También se emplea un **valor futilidad**, que sirve de valor tope para descartar un camino. Si el costo estimado de una posible solución se vuelve mayor que el valor futilidad, entonces abandonaremos la búsqueda.

El pseudocódigo queda:

```
función YO*(problema, futilidad) devuelve grafo-solucion
        locales: G,G' grafos
{
G grafo vacío
G <- G+{inicio}
inicio.costo <- h(inicio)
si (ES-TERMINAL(inicio)) entonces MARCAR(inicio, resuelto)
repetir hasta (ES-RESUELTO(inicio) o inicio.costo > futilidad)
        construir un subgrafo de G, G', con los conectores marcados
        nodo <- elegir nodo de FRONTERA(G')
        si (VACIO(EXPANDIR(nodo))) entonces nodo.costo=futilidad
        en otro caso (para cada sucesor en EXPANDIR(nodo)) hacer
                G <- G+{sucesor}
                si (ES-TERMINAL(sucesor) entonces MARCAR(sucesor,
                   resuelto) y sucesor.costo=0
                si (!ES-TERMINAL(sucesor) y (no estaba en G)) entonces
                   sucesor.costo=h(sucesor)
        S={nodo : nodo ha sido marcado resuelto o su costo ha cambiado}
        repetir hasta VACIO(S)
                actual <- actual en S tal que no hay descendientes de
                   actual en S
                S <- S-{actual}
                para cada (k-conector de actual {ni1,...,nik}) hacer
                        actual.costoi=c+ni1.costo+...+nik.costo
                actual.costo <- min(actual.costoi)
                conector <- (el que nos da el valor anterior)
                MARCAR(conector, marca)
                si (todos los sucesores a través de conector están
                   marcados como resueltos) entonces MARCAR(actual,
                   resuelto)
                si (actual ha sido marcado resuelto o su costo ha
                   cambiado) entonces
                        propagar la información hacia el principio del
                           grafo
                        S <- S+{actual.antecesores}
}
```

**Definición**
La **condición de monotonía en una estructura Y/O** de una función heurística $h$ es que, para cualquier conector dirigido desde el nodo $n$ a los sucesores $n_1, ..., n_k$ se verifica

$$h(n) \leq c + \sum_{i=1}^{k} h(n_i)$$

donde $c$ es el costo del $k$-conector.

Así, si hay un grafo solución desde el nodo inicial a un conjunto de nodos terminales y $h$ es admisible

y satisface la condición de monotononía para estructuras Y/O, entonces se dice que el **método YO* es admisible** y proporciona un grafo solución óptimo.

Cuando se verifica la condición de monotonía de $h$, las revisiones de costos que se hacen en YO* solo pueden consistir en incrementos.

Se puede modificar la última parte del método para que se incluyan en S solo los antecesores de actual tales que actual sea uno de sus sucesores a través de conectores marcados.

Notemos, por último, que:

- se puede ampliar el método para trabajar con grafos Y/O generales, que pueden contener ciclos
- la definición del coste de un grafo solución supone que un subproblema puede resolverse dos o más veces, lo que implica contar el costo de resolverlo varias veces. Si el problema implica algún proceso físico, entonces las soluciones proporcionadas por YO* son óptimas. Pero si el proceso de resolución no es de ese tipo quizás no deberían contarse los costos más de una vez

## 3.3 El efecto de la precisión heurística en el rendimiento

Una manera de caracterizar la calidad de una heurística es el **factor de ramificación eficaz**, $b^*$.

Si el número total de nodos generados por un método para un problema es $N$ y la profundidad de la solución es $d$, entonces el factor de ramificación eficaz, $b^*$, es el factor de ramificación que un árbol uniforme de profundidad $d$ debería tener para tener $N+1$ nodos. O sea, que verifica

$$N+1 = 1 + b^* + (b^*)^2 + ... + (b^*)^d$$

Una buena heurística debe tener $b^*$ lo más próximo posible a 1, lo que permite resolver problemas complejos en un tiempo razonable.

**Definición**
Una heurística $h_2$ domina a otra $h_1$ si, y solo si,

$$h_2(n) \geq h_1(n), \ \forall n$$

La dominancia se traslada directamente a la eficiencia

### 3.3.1 Diseñando funciones heurísticas admisibles

**Definición**
Un problema con menos restricciones que el original se denomina **problema relajado**. El costo de una solución óptima en un problema relajado es una heurística admisible para el problema original, porque la solución óptima en el problema original es, por definición, también una solución del problema relajado, por lo que debe ser al menos tan cara como su solución óptima.

Un problema con la generación de nuevas funciones heurísticas es que a menudo no es fácil o incluso posible conseguir una heurística claramente mejor (una es mejor en unos casos, y otra es mejor en otros). Pero, si disponemos de un conjunto de heurísticas admisibles $h_1, ..., h_k$ para un problema, y ninguna domina a las demás, podemos escoger la heurística

$$h(n) = \max\{h_1(n), ..., h_k(n)\}$$

que es admisible:

$$h(n) = h_i(n) \leq h^*(n)$$

consistente si las componentes lo son

$$h(n) = h_i(n) \leq K(n, a_s, m) + h_i(m) \leq K(n, a_s, m) + \max_i \{h_i(m)\} = K(n, a_s, m) + h(m)$$

además, $h$ domina a todas sus componentes.

# 4 Estrategias avanzadas de búsqueda

Los métodos de búsqueda vistos hasta ahora se diseñan para explorar sistemáticamente espacios de búsqueda. No obstante, en muchos problemas el camino al objetivo es irrelevante, solo nos interesa llegar hasta él. Además, en los procesos de búsqueda que hemos visto hasta ahora, todas las decisiones tomadas pueden ser recuperables, pero hay muchos problemas no recuperables, en los que una decisión de un agente en un momento dado tiene que considerar posibles acciones que pueden realizar otros agentes, planteando distintos escenarios posibles de forma imprevisible. Esto puede introducir dificultades en el proceso de resolución de problemas.

Los entornos competitivos, en los que los objetivos de varios agentes están en conflicto, dan lugar a problemas de búsqueda entre adversarios o **juegos**.

## 4.1 Métodos de búsqueda local y problemas de optimización

Si no importa el camino al objetivo, podemos considerar una nueva clase de métodos, los **métodos de búsqueda local**, que:

- operan con un único estado actual
- generalmente se mueven solo a los vecinos del estado actual
- los caminos seguidos no suelen ser recordados

Estos métodos presentan dos grandes ventajas:

- usan muy poca memoria (generalmente $O(n)$
- a menudo pueden encontrar soluciones rezonables en espacios de estados grandes o infinitos, donde los algoritmos sistemáticos son inadecuados

Por tanto, son útiles para resolver problemas de optimización puros, en los que se busca el mejor estado según una función objetivo.

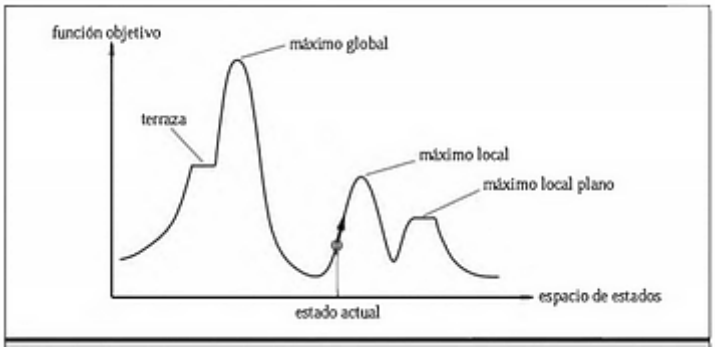
Es útil conceptualmente considerar que el espacio de estados conforma un paisaje que tiene:

- posición: definida por el estado
- elevación: definida por la función objetivo en cada punto

Así, si la elevación corresponde al coste, el objetivo es encontrar un mínimo global (el valle más bajo). Si corresponde a una función de evaluación, entonces buscamos un máximo global (el pico más alto).

Los métodos de búsqued local exploran este paisaje.

Un método de búsqueda local siempre encuentra un objetivo, si este existe. Y un método óptimo siempre encuentra el máximo global.



Podemos encontrar algunas zonas diferenciadas del paisaje:

- **máximo local**: pico que es más alto que cada uno de sus estados vecinos, pero más bajo que el máximo global
- **cresta**: secuencia de máximos locales que hace muy difícil la navegación para los métodos avaros
- **meseta o terraza**: área del paisaje del espacio de estados donde la función objetivo es plana

### 4.1.1 Búsqueda por Ascensión de Colinas

Simplemente es un bucle que continuamente se mueve en dirección del valor creciente. Termina cuando alcanza un pico en donde ningún vecino tiene un valor más alto (un máximo local).

No mantiene un árbol de búsqueda, sino una estructura de datos del nodo actual que necesita solo el registro del estado y su valor de función objetivo. No mira más allá de los vecinos inmediatos del estado actual.

En pseudocódigo:

```
función ASCENSION–COLINAS(problema) devuelve {estado máximo local}
entradas: problema
locales: actual, vecino
{
actual <– HACER–NODO(ESTADO–INICIAL(problema))
bucle hacer
        vecino <– sucesor(actual) con valor máximo
        si vecino.valor<=actual.valor entonces devolver ESTADO(actual)
        actual <– vecino
}
```

Ascensión de colinas a menudo se atasca, cuando encuentra máximos locales que no son globales, crestas o mesetas. En cada caso, el método alcanza un punto en el cual no se puede hacer ningún progreso.

Una posible solución es permitir **movimiento lateral**, es decir, permitir al método moverse a estados vecinos aunque no mejoren el valor actual. Sin embargo, si permitimos que siempre se haga movimiento lateral cuando no hay movimiento ascendente, entraremos en un bucle infinito siempre que el método alcance un máximo local plano. Esto puede solucionarse poniendo un límite sobre el número de movimientos laterales consecutivos laterales permitidos.

Los métodos de ascensión de colinas son incompletos, pueden dejar de encontrar un objetivo aunque este exista.

### 4.1.2 Ascensión de colinas de reinicio aleatorio

Consiste en realizar una serie de búsquedas sucesivas desde un estado inicial aleatorio. Es completa con probabilidad cercana a 1.

Si la búsqueda por ascensión de colinas tiene una probabilidad $p$ de éxito, entonces el número esperado de reinicios es $\frac{1}{p}$, que se reparten entre 1 acierto y $\frac{1}{p} - 1$ fallos, por tanto, el número total de pasos es el coste de una iteración con éxito más $\frac{1-p}{p}$ veces el coste de los fallos.

### 4.1.3 Búsqueda por temple simulado

Intenta combinar la eficacia de la ascensión de caminas con la completitud de la elección aleatoria de caminos. La idea se basa en un proceso metalúrgico para templar metal: se calienta a alta temperatura y se enfría gradualmente, facilitando que alcance una estructura cristalina de baja energía.

Así, de forma intuitiva, podemos pensar que, dado un estado:

- si se agita con suficiente fuerza puede escapar de mínimos locales. La fuerza de agitación la determina una función que simula la temperatura
- si se agita demasiado fuerte puede escapar de un mínimo global
- la fuerza de agitación disminuye con el tiempo, se "enfría" el paisaje

```
función TEMPLE-SIMULADO(problema, esquema) devuelve {estado solución}
entradas: problema
                esquema, función temperatura respecto tiempo
locales: actual y siguiente: nodos
                T: temperatura
{
actual <- HACER-NODO(ESTADO-INICIAL(problema))

para t <- 1 a infinito hacer
        T <- esquema(t)
        si (T=0) entonces devolver actual
        para i <- 1 to Lt hacer
                siguiente <- sucesor aleatorio de actual
                dE <- siguiente.valor - actual.valor
                si dE>0 entonces
                        actual <- siguiente
                si no
                        actual <- siguiente con probabilidad e^(dE/T)
}
```

El bucle interno es similar a ascensión de colinas, pero el sucesor se elige aleatoriamente. Si mejora al actual, se elige seguro. Si lo empeora, se elige con probabilidad $e^{\frac{\Delta E}{T}}$, valor que decrece exponencialmente con la falta de calidad del sucesor y con la temperatura $T$, que disminuye con el tiempo.

### 4.1.4 Búsqueda por haz local

Este método guarda la pista de $k$ estados, y hace lo siguiente:

1. comienza con $k$ estados generados aleatoriamente
2. se generan los sucesores de cada uno de los $k$ estados
3. si alguno es objetivo, fin

4. si no, se seleccionan los $k$ mejores sucesores de la lista completa y volvemos a 2

Podría parecer que es equivalente a realizar $k$ reinicios aleatorios en paralelo en lugar de secuencialmente, pero los dos métodos son distintos. En la búsqueda con reinicio aleatorio, cada proceso de búsqueda es independiente de los demás, mientras que en la búsqueda por haz local, la información útil es pasada entre los $k$ hilos paralelos de búsqueda.

En su forma más simple, la búsqueda de haz local es carente de diversidad entre los $k$ estados.

Una variante muy utilizada son los llamados **algoritmos genéticos**, una variante de la búsqueda de haz estocástica.

#### Algoritmos genéticos

Variante de los métodos de búsqueda por haz estocástica. Esto quiere decir que son un método de búsqueda local: no suele importar cómo llegar a la solución, tan solo la solución en sí misma; y no se recorre el espacio de soluciones al completo, sino parte de él. Por tanto, no siempre obtendremos una solución óptima, pues podemos obtener óptimos locales. Los AGs constan de las siguientes partes:

- **Población:** Llevaremos la pista de $k$ estados, generados al comienzo aleatoriamente.
- **Individuos o cromosomas:** cada estado de la población.
- **Codificación:** forma de representación de los estados. La codificación será una cadena sobre un alfabeto finito. La codificación puede ser binaria, entera, de orden, o real.
- **Gen:** cada elemento de un estado se denomina gen. Por tanto, un gen será un elemento del alfabeto de codificación. El conjunto de genes de un individuo a veces se denomina genotipo, y a la traducción del significado del genotipo al problema que tratamos de resolver se le denomina fenotipo.
- **Función Fitness:** es la función usada para cuantificar lo bueno que es cada individuo en relación al problema. Esta será la función que buscamos optimizar.
- **Operadores genéticos:** los algoritmos genéticos dan uso a 3 operadores básicos para simular el proceso de evolución:
  - **Selección:** utilizado para seleccionar qué individuos pasan a reproducirse, y cuales no. Para la selección es muy normal tener en cuenta los valores de fitness de los individuos, ya que es este valor el que indica la calidad de los mismos.
  - **Cruce:** simula la reproducción de los individuos seleccionados. Define la forma en la que los genes de pares de individuos se entremezclan para dar lugar a nuevos individuos. Por supuesto, los individuos generados mediante cruce deben representar estados válidos del espacio de búsqueda.
  - **Mutación:** este operador provoca variaciones aleatorias en algunos genes de un individuo. Se utiliza para añadir variabilidad a la población.

Así, el esquema básico de un AG queda como sigue:

```
funcion AG(poblacion, fitness) devuelve individuo
entradas: población, función fitness

{
repetir
    mi_poblacion := SELECCION(poblacion, fitness)
    para x=1 hasta tamaño(mi_población) hacer
        si(r<pc) entonces
            poblacion_cruzar = poblacion_cruzar + mi_poblacion[x]
    para i=1 hasta tamaño(poblacion_cruzar) hacer
        {x,y} = dos individuos aleatorios de poblacion_cruzar
        h1, h2 = CRUCE(x,y)
        {x,y} = {h1, h2}
    para x=1 hasta tamaño(mi_poblacion)*longitud(individuo) hacer
        si(r<pm) entonces
            gen = MUTAR(gen)
            modificar el gen del individuo correspondiente
    poblacion = mi_poblacion
hasta condición de fin (fitness mínimo, o suficientemente pequeño, o ha
   pasado mucho tiempo)
devolver individuo de poblacion tal que fitness(individuo)=max{fitness(i
   ): i en poblacion}
}
```

Donde $r$ es un número generado aleatoriamente y$p_c$ es la probabilidad de que un individuo sea seleccionado para cruzarse, $p_m$ es la probabilidad de que se produzca una mutación en un gen determinado, y SELECCION, CRUCE y MUTAR son implementaciones concretas de los operadores que hemos explicado anteriormente. Algunas implementaciones comúnmente utilizadas son:

- Selección:
  - Ruleta: se eligen los individuos de forma directamente proporcional a la función de idoneidad, de modo que individuos con mejor valor fitness tienen más posibilidades de ser elegidos
  - Torneo: se hacen $k$ emparejamientos entre individuos de la población y se eligen los que tienen mejor fitness en cada emparejamiento
- Cruce:
  - Cruce por un punto: se intercambian todos los genes de los dos individuos de un punto en adelante
  - Cruce por dos puntos: se intercambian todos los genes de los dos individuos entre dos puntos seleccionados
- Mutación:
  - Cambio de un gen aleatorio: un gen cambia su valor aleatoriamente
  - Intercambio entre dos genes: dos genes aleatorios intercambian sus valores

## 4.2 Búsqueda entre adversarios: Decisiones óptimas en Juegos

En los entornos multiagente, cualquier agente tiene que considerar las acciones de los otros agentes y cómo afectan a su propio bienestar. La imprevisibilidad de estos otros agentes puede introducir muchas posibles contingencias en el proceso de resolución de problemas del agente.

Los entornos competitivos, en los cuales los objetivos de diferentes agentes están en conflicto dan ocasión a problemas de búsqueda entre adversarios o **juegos**.

En IA, normalmente se consideran **juegos de suma cero**, es decir, juegos tales que se desarrollan en entornos deterministas, totalmente observables con dos agentes cuyas acciones deben alternar y en los que los valores utilidad, al final de juego, son siempre iguales y opuestos.

Consideraremos juegos con dos jugadores llamados MAX y MIN. MAX mueve primero, y mueven por turnos hasta que el juego se termina. Al final del juego se concederán puntos al jugador ganador y penalizaciones al perdedor, en la misma cantidad (juegos de suma cero).

**Definición**
Un **juego** es una clase de problemas de búsqueda, donde:

- el estado inicial es la posición del tablero inicial e identifica al jugador que mueve primero
- la función sucesor devuelve una lista de pares (movimientos, estado)
- un test terminal que determina cuándo se termina el juego
- una función utilidad que asigna un valor numérico a los estados terminales

El estado inicial y los movimientos legales para cada lado definen el árbol de juego. El número sobre cada nodo hoja indica el valor de utilidad del estado terminal desde el punto de vista de un jugador: cuanto mayor sea un valor, mejor para MAX y peor para MIN. Así, MAX usará el árbol de búsqueda para determinar el mejor movimiento.

### 4.2.1 Estrategia óptima. Minimax

MAX debe encontrar una estrategia contingente, que especifica el movimiento de MAX en el estado inicial, después los movimientos de MAX en los estados que resultan de cada respuesta posible de MIN, después los movimientos de MAX en los estados que resultan de cada respuesta posible de MIN de los anteriores movimientos, y así.

Así, el valor de minimax en cada nodo $n$, será:

$$VALOR\_MINIMAX\,(n) = \begin{cases} utilidad\,(n) & si\ n\ terminal \\ \max_{s\in SUC(n)} VALOR\_MINIMAX\,(s) & si\ n\ es\ de\ MAX \\ \min_{s\in SUC(n)} VALOR\_MINIMAX\,(s) & si\ n\ es\ de\ MIN \end{cases}$$

Esto sería un juego óptimo para MAX, suponiendo que MIN también juega de forma óptima. Pero si MIN no juega de forma óptima, entonces MAX debe seguir la misma estrategia, ya que lo hará aún mejor.

El pseudocódigo es:

```
función DECISION–MINIMAX(estado) devuelve acción
entrada: estado
{
        v <– MAX–VALOR(estado)
        devolver {accion de SUCESORES(estado) con valor v}
}

función MAX–VALOR(estado) devuelve {valor utilidad}
{
        si TEST–TERMINAL(estado) entonces devolver estado.utilidad
        v <– –infinito
        para {un s en SUCESORES(estado)} hacer
                v <– MAX(v,MIN–VALOR(s))
        devolver v
}

función MIN–VALOR(estado) devuelve {valor utilidad}
{
        si TEST–TERMINAL(estado) entonces devolver estado.utilidad
        v <– infinito
        para {un s en SUCESORES(estado)} hacer
                v <– MIN(v,MAX–VALOR(s))
        devolver v
}
```

El método minimax, por tanto, realiza una exploración primero en profundidad. Suponiendo que $m$ es la profundidad máxima y $b$ el número de movimientos legales:

- la complejidad en tiempo es $O(b^m)$
- la complejidad en espacio es

$$\begin{cases} O(bm) & \text{si se generan todos los sucesores a la vez} \\ O(m) & \text{si se generan los sucesores uno por uno} \end{cases}$$

Un árbol MINIMAX es análogo a un árbol Y/O, donde los nodos MAX son nodos O y los nodos MIN son nodos Y (si MIN actúa de forma óptima).

### 4.2.2 Decisiones óptimas en juegos multijugador

Minimax se puede extender a juegos multijugador. Para ello, sustituimos el valor para cada nodo con un vector de valores. Para los estados terminales, este vector dará la utilidad del estado desde el punto de vista de cada jugador. Nótese que en juegos de suma cero, el vector de dos elementos puede reducirse a un valor porque el otro será el opuesto.

Por tanto, debemos hacer que la función utilidad devuelve un vector de utilidades. El valor hacia atrás de un nodo $n$ es el vector de utilidad de cualquier sucesor que tiene el valor más alto para el jugador que elige en ese nodo.

### 4.2.3 MINIMAX ALFA-BETA

La búsqueda minimax provoca un crecimiento exponencial en el número de movimientos. Aunque no se puede eliminar el exponente, se puede dividir. Este es el objetivo de la poda alfa-beta.

La **poda alfa-beta** devuelve el mismo movimiento que devuelve minimax, pero podando nodos o subárboles que pueden descartarse por seguridad de no empeorar la solución actual.

Llamamos $\alpha$ a la mejor opción (valor más alto) encontrada a lo largo del camino para MAX, y $\beta$ al mejor valor (el más bajo) encontrado a lo largo del camino para MIN.

```
1  función BUSQUEDA-ALFA-BETA(estado) devuelve acción
   entrada: estado
   {
           v <- MAX-VALOR(estado,-infinito,infinito)
           devolver {acción de SUCESORES(estado) con valor v}
6  }

   función MAX-VALOR(estado, alpha, beta) devuelve {valor utilidad}
   entrada: estado
                   alpha
11                 beta
   {
           si TEST-TERMINAL(estado) entonces devolver estado.utilidad
           v <- -infinito
           para {un s en SUCESORES(estado)} hacer
16                 v <- MAX(v,MIN-VALOR(s,alpha,beta))
                   si v>=beta entonces devolver v
                   alpha <- MAX(alpha,v)
           devolver v
   }
21
   función MIN-VALOR(estado, alpha, beta) devuelve {valor utilidad}
   entrada: estado
                   alpha
                   beta
26 {
           si TEST-TERMINAL(estado) entonces devolver estado.utilidad
           v <- infinito
           para {un s en SUCESORES(estado)} hacer
                   v <- MIN(v,MAX-VALOR(estado))
31                 si v <= alpha entonces devolver v
                   beta <- MIN(beta,v)
           devolver v
   }
```

La eficacia de la poda alfa-beta es muy dependiente del orden en el que se examinan los sucesores.

- si podemos elegir el mejor orden, alfa-beta tiene que examinar $O\left(b^{\frac{d}{2}}\right)$ nodos para escoger el mejor movimiento, en lugar de $O\left(b^{d}\right)$
- si los sucesores se examinan en orden aleatorio, el número total de nodos examinados será aprox-

imadamente $O\left(b^{\frac{3d}{4}}\right)$

### 4.2.4 Decisiones en tiempo real imperfectas. Heurísticas

Tanto minimax como alfa-beta tienen que buscar en todos los caminos, o una parte de ellos, hasta los estados terminales. Esta profundidad no es, en general, práctica, porque los movimientos deben hacerse en una cantidad razonable de tiempo.

Se propuso que los programas cortasen la búsqueda antes y entonces aplicar una función de evaluación heurística a los estados, convirtiendo los nodos no terminales en hojas terminales.

Para este propósito, se sustituye la función de utilidad por una función de evaluación heurística que estime la utilidad de la posición, y sustituir el test-terminal por un test-límite que decide cuando aplicar la heurística.

Se debe llevar el cálculo de la profundidad y poner un límite de profundidad fijo, de modo que TEST-LÍMITE(estado, profundidad) devuelva verdadero para toda profundidad mayor que una profundidad prefijada $d$, que se elige de modo que la cantidad de tiempo usado no exceda lo permitido por las reglas del juego.

Un test límite más sofisticado tiene en cuenta que la función de evaluación solo debería aplicarse a posiciones estables (es decir, que es improbable que sufran grandes variaciones en su valor en un futuro próximo), por lo que las no estables se extienden hasta alcanzar posiciones estables.

### 4.2.5 Juegos que incluyen un elemento de probabilidad

Muchos juegos incluyen elementos aleatorios. Aún así, queremos escoger el movimiento que conduzca a la mejor posición. Como no hay valores minimax definidos, debemos calcular el valor esperado (esperanza matemática).

Para ello, generalizamos el valor minimax para juegos deterministas a un valor minimax esperado para juegos con nodos probabilísticos.

Los nodos terminales y los MAX y los MIN trabajan igual que antes, pero los nodos probabilísticos se evalúan tomando el promedio ponderado de los valores que se obtienen de todos los resultados posibles:

$$MINIMAXESP\,(n) = \begin{cases} n.utilidad & \text{si } n \text{ es terminal} \\ \max_{s \in SUC(n)} MINIMAXESP\,(s) & \text{si } n \text{ es un nodo MAX} \\ \min_{s \in SUC(n)} MINIMAXESP\,(s) & \text{si } n \text{ es un nodo MIN} \\ \sum_{s \in SUC(n)} P\,(s) \cdot MINIMAXESP(s) & \text{si } n \text{ es un nodo probabilístico} \end{cases}$$

donde $P\,(s)$ es la probabilidad de que ocurra $s$.

# 5 Representación del conocimiento. Razonamiento.

Como venimos viendo, el objetivo básico de un sistema inteligente no es más que resolver tareas para las que se necesita una gran cantidad de información o solo se dispone de descripciones poco estructuradas, incompletas o imprecisas.

Muchas veces buscamos la consecución de tareas genéricas, que para ser resueltas necesitamos tanto de información sobre el dominio como un método genérico de resolución independiente, en muchos casos, del dominio de aplicación.

Según la información que necesitan, los sistemas inteligentes pueden ser:

- **Intensivos en datos:** minan grandes cantidades de datos para obtener conocimiento, que luego puede ser usado para resolver una tarea
- **Intensivos en conocimiento:** aplican gran cantidad de conocimiento a los datos para obtener conclusiones útiles

**Definición**
Un **sistema basado en conocimiento (SBC)** es un sistema inteligente intensivo en conocimiento.
El **conocimiento** no son más que descripciones declarativas y explícitas formadas por:
- conceptos y relaciones entre los conceptos, específicos del dominio de aplicación
- **métodos de resolución de problemas (PSM):** métodos genéricos que especifiquen paso a paso el proceso para resolver la tarea concreta

Así, para resolver una tarea, necesitamos:

- una representación computacional del conocimiento: conceptos, relaciones y PSM
- una técnica de razonamiento: que utilice el las relaciones entre conceptos para inferir conclusiones, mediante la estructura de control especificada por el PSM

**Representación del conocimiento y razonamiento**

Es una disciplina de la IA centrada en el diseño de estructuras de datos para almacenar conocimiento y el diseño de técnicas de razonamiento capaces de manipular esas estructuras para realizar inferencias.

**Tipos de conocimiento:**

- Conocimiento de entidades: propiedades y estructura física de objetos reales
- Conocimiento de conducta: comportamiento o modo de proceder de los entes
- Conocimiento de eventos: secuencia y distribución temporal de sucesos, así como relaciones causales entre los mismos
- Conocimiento procedimental: cómo deben realizarse determinados procesos o transformaciones
- Conocimiento sobre incertidumbre: sobre la certeza de los hechos

- Meta-Conocimiento: conocimiento sobre el conocimiento

**Fases de Utilización**

Se refieren a la etapas en la obtención y uso del conocimiento establecidas en distintos paradigmas de la Ingeniería del Conocimiento:

1. Adquisición: se extrae el conocimiento de alguna fuente (experto/procesos de prueba-error)
2. Conceptualización: definición y organización de los distintos componentes del conocimiento. Constituye el proceso central de la ingeniería del conocimiento
3. Representación: diseño de las estructuras de datos que se usarán como soporte computacional del conocimiento. Por tanto, la representación del conocimiento es declarativa
4. Implementación: desarrollo del software que implementa el conocimiento y las técnicas de inferencia, que suele ser procedimental
5. Acceso: relacionado con la forma de extraer el conocimiento desde una representación estructurada. Factores como la velocidad de acceso son muy relevantes (grandes cantidades de datos o funcionamiento en tiempo real)
6. Selección o recuperación: fase en la que hay que seleccionar un elemento del conocimiento concreto, adecuado al problema y al estado del proceso de resolución del mismo. A menudo se utiliza meta-conocimiento

**Propiedades de las Representaciones del Conocimiento**

- Ámbito: extensión del dominio en el que se va a aplicar el conocimiento. Se considera imposible definir conocimiento que abarque cualquier contexto de aplicación.
- Granularidad: tamaño de la unidad mínima de representación.
  - Grano de gran tamaño: permite representar fácilmente conceptos de alto nivel, pero presenta un difícil tratamiento y procesamiento
  - Grano pequeño: permite representar conceptos de bajo nivel, construir estructuras jerárquicas y su tratamiento es más sencillo, pero complica la tarea de representar conceptos de alto nivel
- Redundancia: responde a si es posible o tiene sentido representar un mismo conocimiento de múltiples formas. Como inconveniente presenta que abandonamos la unicidad del conocimiento, pero tiene como ventaja que genera una diversidad que permite una aplicación más efectiva del conocimiento, ganando en generalidad
- Modularidad: posibilidad de agrupar gránulos de conocimiento en módulos, asociados a distintos pasos de un PSM. Esto facilita su recuperación, así como el diseño
- Comprensibilidad: capacidad de interpretar las estructuras de representación

## 5.1 Sistemas Basados en Reglas (SBR)

La lógica, si bien no describe de forma precisa ni completa el proceso de razonamiento humano en general, es ciertamente una buena herramienta de descripción de algunos razonamientos concretos. Concretamente, la lógica la utilizamos cuando de una serie de sucesos, extraemos la causa y la consecuencia. Como ejemplo, podemos pensar en que estamos plantados frente a un interruptor cuya función desconocemos, lo pulsamos, y se enciende una luz azul. Concluiremos entonces que, a causa de pulsar el botón, la luz azul se ha encendido. De esta forma, la lógica trata de formalizar este proceso de relacionar causas y efectos.

En este sentido, podríamos plantear la posibilidad de dotar a los ordenadores de sistemas lógicos de razonamiento, de forma que proporcionándoles una serie de inputs (que en analogía con los humanos serían lo que detectamos mediante nuestros sentidos), usen unas reglas que relacionan y distinguen las causas de los efectos (para nosotros esto es la experiencia, lo que hemos aprendido) y extraigan las consecuencias que derivan de esos inputs. Y esto precisamente son los **Sistemas Basados en Reglas (SBR)**, la implementación en un ordenador de un sistema lógico de deducción formal. En ocasiones se denominan Sistemas Expertos, porque las reglas de inferencia son proporcionadas por expertos en la materia en la que estamos interesados en hacer predicciones.

### 5.1.1 Componentes de un SBR

- **Base de Conocimiento (BC)**: contiene las **reglas** que codifican el conocimiento sobre la materia de estudio. Las reglas, a su vez, tienen dos partes:
  - Condición, consecuente o Left Hand Side (LHS): codifica la causa que precede a un efecto.
  - Consecuente o Right Hand Side (RHS): codifica el efecto producido por la causa.

  Es decir, las reglas son un par causa-efecto, normalmente expresado como **Si** causa **Entonces** efecto.

- **Base de Hechos (BH)**: representa el estado actual de resolución de un problema concreto. Contiene los inputs del problema, las consecuencias inferidas a lo largo del razonamiento, y las metas a alcanzar o consecuencias finales que buscamos conocer.

- **Motor de Inferencias (MI)**: es el mecanismo algorítmico que utiliza la BC y la BH para obtener conclusiones. Los motores de inferencia pueden trabajar de dos formas:
  - Razonamiento hacia delante: parte de los hechos conocidos, y usando las reglas obtiene las consecuencias derivables de estos. Responde a la cuestión: ¿Qué efectos tendrán estos hechos conocidos?
  - Razonamiento hacia atrás: parte de las consecuencias que deseamos saber si se verifican, y usa las reglas, hacia atrás, para ver si, con los hechos conocidos, se verifican las consecuencias buscadas. Responde a la cuestión: ¿Ocurrirán estas consecuencias, dados estos hechos?

- **Red de inferencia**: es un grafo dirigido en el que los nodos son las reglas, representadas mediante puertas lógicas. Las condiciones del antecedente son las entradas a los nodos y los efectos son las salidas, que pueden ser las condiciones de otros nodos. Los antecedentes que no son consecuentes de ninguna otra regla son los posibles hechos de partida; y el consecuente que no es antecedente de ninguna otra regla se convierte en la meta a alcanzar por el sistema. La red de inferencia resulta de utilidad cuando no hay una gran cantidad de reglas.

Es importante notar que la Base de Conocimiento describe todo nuestro conocimiento acerca del problema general que estamos tratando, mientras que la base de hechos es dependiente de las circunstancias específicas de cada situación. Por ejemplo, un SBR que diferencia sillas de mesas, debería tener una BC en la que se proporcione la información necesaria para diferenciar una mesa de una silla, y en la BH tendremos, por ejemplo, fotos de las sillas y mesas concretas que buscamos diferenciar.

Además, la BC y la BH trabajan en conjunción, unidas por la MI, una sola de ellas es inútil si no se dispone de la otra. Las reglas de la BC operan sobre la BH, la condición de una regla debe tratar de verificarse con los datos de la BH; y si uno de estos test se verifica, entonces obtendremos una consecuencia, que puede propiciar cambios en el contenido de la BH.

## 5.2 Representación mediante Lógicas no Clásicas

Los primeros sistemas de razonamiento estaban basados en lógicas clásicas (proposicional y de primer orden), pero esto presenta varios inconvenientes:

- los algoritmos para manipular conocimiento lógico son ineficientes
- las lógicas clásicas no pueden tratar conocimiento incompleto, incierto, impreciso o inconsistente
- la expresividad es limitada

Pero hay muchas otras lógicas, la mayoría diseñadas específicamente para superar ciertas deficiencias de la lógica clásica. En general, un sistema para la manipulación del conocimiento puede ser considerado como una lógica si contiene:

- Un **lenguaje** bien definido para representar conocimiento
- Una teoría de modelos o **semántica** bien definida que se ocupe del significado de declaraciones expresadas en el lenguaje
- Una **teoría de demostración** que se ocupe de la manipulación sintáctica y de la obtención de declaraciones a partir de otras declaraciones

### 5.2.1 Lógicas no monótonas

**Definición**
Una **lógica es monótona** si cuando se añade un axioma o un teorema propio a una teoría $T$ para obtener una teoría $T'$, entonces todos los teoremas de $T$ son también teoremas en $T'$.
Por contra, en una **lógica no monótona** la incorporación de un teorema a una teoría puede invalidar conclusiones que podrían haberse hecho previamente.

Hay al menos tres circunstancias en las que el razonamiento no monótono puede ser adecuado:

- cuando el conocimiento es incompleto deben hacerse suposiciones por defecto, que puedan invalidarse cuando se disponga de más conocimiento
- cuando el universo es cambiante
- en resolución de problemas donde deban hacerse suposiciones temporales

Las lógicas clásicas son **estáticas**, es decir, que una vez que se establece que una afirmación es cierta, siempre será así (si no el sistema no sería consistente y no serviría de mucho). Nótese que esto restringe mucho los problemas atacables con lógica clásica, por los motivos que ya hemos comentado.

### Lógica de Situaciones

En un mundo cambiante, los hechos pueden ser ciertos en determinadas situaciones y pasar a ser falsos en otras, o al revés. La lógica de situaciones es una forma de modelar un mundo cambiante. En ella, los predicados tienen un argumento extra que denota en qué situación la afirmación es verdadera.

Las situaciones y eventos se relacionan mediante una relación, $R$, de forma que $R(e, s)$ denota la situación que se obtiene cuando ocurre $e$ en la situación $s$.

### Lógica Difusa: representación de la vaguedad

Históricamente, las ciencias formales han restringido el uso del lenguaje natural a aquellos predicados que, aplicados a un dominio, lo dividen en subconjuntos claramente delimitados (por ejemplo, decir los *números pares*, divide a los naturales en dos subconjuntos sin lugar a confusión sobre si un elemento está en un conjunto u otro). En el mundo real, en cambio, es muy frecuente encontrarse con afirmaciones vagas, que no permiten realizar una división satisfactoria del dominio (por ejemplo, los *altos* no son un subconjunto de la población claramente diferenciado. Otra cosa sería decir, la *gente que mide más de* $1.80m$, *recién levantados*, que sí que proporciona una división precisa de la población).

Estos **predicados vagos** están presentes en el lenguaje ordinario y son un elemento fundamental en el razonamiento humano. Pero la falta de nitidez de estos predicados supone una fuente de incosistencias (ya que un individuo podría ser al mismo tiempo alto y no alto, dependiendo de la interpretación que hagamos de ser alto). Así, un predicado vaho puede definirse como aquel en que la frontera del conjunto de objetos que cumplen la propiedad representada por el predicado no está bien definida. O sea "$x$ es $A$" es vago si $A$ no tiene la frontera bien marcada. Nótese que la incertidumbre no se encuentra en el mundo externo (la altura es cuantificable y en un momento concreto es fija), sino que la incertidumbre se encuentra en el término linguístico que trata de describir la propiedad (la palabra *alto* no especifica a partir de qué altura ni en qué circunstancias se realiza la medición).

Podemos, entonces, razonar a partir de afirmaciones vagas, usando **razonamiento aproximado**. Por ejemplo, si estamos ante la regla

$$x \textit{ pequeño} \implies x \textit{ ligero}$$

y nos proporcionan la siguiente información o hecho

$$x \textit{ ocupa } 1cm^3$$

entonces, en lógica clásica no podemos equiparar el hecho con el antecedente de la regla, ya que *pequeño* es una expresión vaga. No obstante, en el mundo real, se produce un emparejamiento parcial entre el hecho y la condición de la regla, y es posible inferir alguna conclusión. Por ejemplo, si estamos en un contexto de paquetes de envíos, $1cm^3$ se puede considerar pequeño, por lo que se puede concluir que $x$ es ligero y, por ejemplo, aplicar la tarifa "paquetes ligeros".

A partir de estos conceptos, se definen los **conjuntos difusos**, que suponen un salto cualitativo en la relación entre la ciencia y el lenguaje ante la necesidad de herramientas formales para su análisis. Estas herramientas las proporciona la **teoría de los conjuntos difusos** y la ciencia que aporta los principios formales del razonamiento aproximado es la **lógica difusa**.

**Conjuntos precisos o clásicos VS Conjuntos difusos**

En lógica clásica, la noción de predicado está íntimamente ligada a la de conjunto. En general, todo predicado preciso $P(x)$ aplicado a un dominio $U$ tiene asociado un conjunto preciso, $P \subset U$, formado por $P = \{x \in U : P(x) = true\}$. Este conjunto puede definirse, entonces, mediante una función de pertenencia (función característica) que describe la pertenencia de cada elemento de $U$ a $P$ como un valor en $\{0, 1\}$

$$\begin{array}{rcl} \mu_P : U & \rightarrow & \{0,1\} \\ x & \mapsto & \begin{cases} 0 & x \notin P \\ 1 & x \in P \end{cases} \end{array}$$

de tal forma que $\mu_P(x) = 0 \implies P(x) = false$ y $\mu_P(x) = 1 \implies P(x) = true$.

Sin embargo, al aplicar un predicado vago, $A(x)$, a un dominio $U$, obtenemos un conjunto difuso $A \subset U$, definido mediante una función de pertenencia de cada elemento de $U$ a $A$ coo un grado en $[0, 1]$, donde el 0 indica que seguro que $x$ no verifica $A$, y el 1 que seguro que sí (es como una probabilidad de que $x$ verifique la propiedad $A$).

$$\begin{array}{rcl} \mu_A : U & \rightarrow & [0,1] \\ x & \mapsto & \mu_A(x) \in [0,1] \end{array}$$

## 5.3 Representación y razonamiento con incertidumbre

En muchos sistemas inteligentes es preciso considerar hechos cuya fiabilidad y precisión es limitada, y conocimiento del que no se tiene total certeza. Por tanto, debemos resolver dos problemas:

- **Representación**: necesitamos formalismos que capturan y soportan incertidumbre, así como establecer medidas formales de la incertidumbre
- **Razonamiento**: gestión de la incertidumbre en los procesos de inferencia. Técnicas formales para la propagación y combinación de afirmaciones inciertas

Es frecuente incorporar la incertidumbre sobre una representación que originalmente no la incluye, para aumentar su ámbito de aplicación. Se hace necesario diferenciar entre:

- **impreciso**: ambiguo, no concreto, no detallado
- **incierto**: hechos, afirmaciones o sucesos carentes de verdad absoluta o seguridad de ocurrencia

### 5.3.1 Teoría de la certidumbre o de los factores de certeza

No todo (de hecho, casi nada) son hechos factuales que conocemos con seguridad, y cuyas consecuencias conocemos con certeza. Normalmente, trabajamos con incertidumbre y las conclusiones obtenidas a partir de una causa suelen tener un cierto grado de fiabilidad.

Para este propósito incorporamos el concepto de **Factor de Certeza**, que se entiende como la credibilidad del consecuente o hipótesis en función de la conjunción de antecedentes o evidencias. Los factores de certeza, en última instancia, son valoraciones subjetivas proporcionadas por los expertos en la materia a estudio.

Formalmente, se define en términos de dos componentes subjetivos:

- MC(h,e): medida de la creencia en la hipótesis h, dada la evidencia e.
  $MC(h, e) \in [0, 1]$ *y* $MC(h, e) = 0$ si la evidencia no soporta a h.

- MI(h,e): medida de la incredulidad en la hipótesis h, dada la evidencia e.
  $MI(h,e) \in [0,1]$ *y* $MC(h,e) = 0$ si la evidencia soporta a h.

Es de notar que una misma evidencia no puede, al mismo tiempo, apoyar la creencia y la incredulidad de una hipótesis, es decir:

$$MC(h,e) > 0 \implies MI(h,e) = 0$$
$$MI(h,e) > 0 \implies MC(h,e) = 0$$

Y el factor de certeza se define como

$$FC(h,e) = MC(h,e) - MI(h,e)$$

de forma que $FC(h,e) \in [-1,1]$

Por último, debemos combinar nuestro método de razonamiento explicado anteriormente con la existencia de incertidumbre, para ello debemos definir la forma en la que los factores de certeza se combinan al usar distintas reglas, obtener nueva evidencia a favor o en contra de algunos hechos, etc.

Para ello, debemos distinguir los posibles casos con los que nos podemos encontrar:

- **Combinación de antecedentes:** combinamos las piezas de evidencia $e_1$, *y* $e_2$, que afectan al factor de certeza de h:

$$FC(h, e_1 \wedge e_2) = \min\{FC(h,e_1), FC(h,e_2)\}$$

$$FC(h, e_1 \vee e_2) = \max\{FC(h,e_1), FC(h,e_2)\}$$

- **Adquisición incremental de evidencia:** cuando dos evidencias afectan de forma distinta (mediante reglas diferentes) a una hipótesis:

- $$FC(h, e_1 \wedge e_2) = \begin{cases} FC(h,e_1) + FC(h,e_2) \cdot (1 - FC(h,e_1)), & \text{si } FC(h,e_1), FC(h,e_2) \geq 0 \\ FC(h,e_1) + FC(h,e_2) \cdot (1 + FC(h,e_1)), & \text{si } FC(h,e_1), FC(h,e_2) \leq 0 \\ \frac{FC(h,e_1)+FC(h,e_2)}{1-\min\{|FC(h,e_1)|,|FC(h,e_2)|\}}, & \text{si } FC(h,e_1) \cdot FC(h,e_2) < 0 \end{cases}$$

- **Encadenamiento de evidencia:** se combinan dos reglas, tales que el resultado de una es la entrada de la otra:

  $$FC(h,e) = FC(h,s) * \max\{0, FC(s,e)\}$$

  donde las reglas han sido de la forma R1: Si e Entonces s; y R2: Si s Entonces h.

**¿Qué mide el factor de certeza asociado a un hecho?**

El factor de certeza es la probabilidad subjetiva que un experto da a la veracidad de un hecho a partir de la evidencia disponible. Aunque no siempre es directamente proporcionado por el experto, sino que puede ser inferido a partir de los hechos y los factores de certeza de otros hechos anteriores. En este caso, el significado del FC es el mismo, pero ahora no ha sido asignado directamente por el experto, sino que ha sido derivado mediante el proceso de inferencia.

## 5.4 Representaciones estructuradas del conocimiento

No existe actualmente una forma general de representación del conocimiento capaz de ser usada con éxito en todo tipo de aplicación; las formas disponibles están limitadas a dominios específicos. Entre las representaciones más usadas encontramos:

- **Redes semánticas:** técnica basada en grafos que pretende modelar los mecanismos de memoria humanos, haciendo énfasis en nuestra capacidad de recuperar conceptos a través de las relaciones que los enlazan
- **Marcos o Frames:** técnica basada en estructuras parecidas a formularios, que es preciso rellenar e inferencia basada en herencia

Más adelante, se adoptó el término **ontología** para los esquemas de representación del conocimiento basadas en redes semánticas o marcos. Es una escificación formal y explícita de una conceptualización compartida, que puede ser léida por un ordenador. Se utiliza en el ámbito de la ingeniería del conocimiento para referirse a un conjunto de conceptos organizados jerárquicamente, representados en algún sistema informático cuya utilidad es la de servir de soporte a diversas aplicaciones que requieren de conocimiento específico sobre la materia que la ontología representa.

# 6 Planificar para la resolución de problemas

El **problema de la planificación** consiste en la dificultad de llevar un sistema inteligente a resolver problemas del mundo real, en el que se pueden llevar a cabo infinidad de acciones distintas, aunque queremos que el sistema solo ejecute un subconjunto muy reducido de todas las posibles acciones. Por tanto, se debe identificar ese conjunto de acciones que debe ser capaz de realizar el sistema inteligente, y cómo cambia el mundo cada acción.

**Definición**
Se denomina **planificar** a la tarea de obtener una secuencia de acciones que permita llegar a un estado objetivo.
La secuencia de acciones obtenida se denomina **plan**

En principio, un problema de planificación podría resolverse aplicando técnicas de búsqueda, pero esto no es factible en la práctica, porque el espacio de búsqueda, como adelantábamos, suele ser extraordinariamente complejo, los problemas reales de planificación no suelen ser descomponibles en problemas independientes, y los estados de búsqueda suelen requerir descripciones complejas.

## 6.1 Descomposición y Problemas Interactivos

Un **sistema planificador** es esencialmente un sistema de descomposición de problemas. Usa técnicas de búsqueda diferentes a las estudiadas. Para generar un plan, un sistema planificador necesita de:

- una descripción del estado inicial del problema
- un objetivo
- un conjunto de acciones, operadores o reglas

La eficacia del sistema depende de la complejidad de las reglas disponibles. Así, el diseño de reglas debería ser modular y completo, es decir, que permita alcanzar el objetivo.

Nos vamos a encontrar con los siguientes problemas:

- **problema del marco o estructura**: cuando se ejecuta una regla, ¿qué permanece sin cambios?
- **problema de la cualificación**: ¿qué necesita una regla en su entorno para ejecutarse?
- **problema de la ramifiación**: ¿qué elementos relevantes del entorno se modifican?

### 6.1.1 Métodos de planificación

- **Lineal:** presenta una lista ordenada de todas las acciones a realizar, ordenadas con un orden total. Es decir, se realiza una acción tras de otra en el orden especificado.
- **No Lineal**: un plan no es necesariamente una lista ordenada, sino que puede ser una ordenación parcial de las acciones a realizar para conseguir un objetivo. La ordenación parcial debe respetar os prerrequisitos de cada acción.
- **Jerárquico**: para establecer un plan se pueden establecer aproximaciones sucesivas. Tras un primer plan general se va refinando jerárquicamente hasta el nivel de detalle que requiera el problema.

### 6.1.2 Componentes de un sistema de planificación

Recordemos que utilizamos la lógica proposicional de primer orden (LPPO), por lo que las descripciones serán fórmulas bien formadas de la lógica (fbfs). Consideraremos el lenguaje de planificación STRIPS, que hereda muchos aspectos de la lógica de situaciones, que especificaba cómo las situaciones, descritas en LPPO, se veían afectadas por als acciones de un agente.

Así, un **problema STRIPS**, se define como una tupla:

- **estado inicial**: conjunto de fbfs. En general, tanto el estado inicial como los estados intermedios se describen mediante conjunciones de literales básicos
- **estado objetivo**: condición descriptiva del objetivo. Se expresarán como conjunciones de literales y todas las variables se cuantificarán existencialmente
- **conjunto de acciones**: u operadores útiles para la elaboración del plan. Estas acciones generan nuevos estados, por lo que pueden verse como reglas que cambian la descripción de un estado a otra. Constan de:
  - Antecedente: la fórmula de precondición ($P$), es una conjunción de literales cuyas variables tienen cuantificación existencial. Para poder aplicar una regla, la fórmula debe ser cierta en el contexto actual del mundo. Si se halla algún emparejamiento, se dice que la precondición empareja con los hechos. La composición unificadore se denomina sustitución de emparejamiento. Cada sustitución es una posible aplicación de la regla.
  - Consecuente: formado por:
    * los elementos que dejan de ser ciertos (lista de supresión ($S$)): al aplicar una regla, el emparejamiento se aplica a los literales de la lista de supresión, y las particularizaciones se suprimen de la descripción del estado
    * aquellos que pasan a ser ciertos (fórmula de adición($A$)): al aplicar una regla, el emparejamiento se aplica a la lista de adición, y las particularizaciones se añaden a la descripción del estado

## 6.2 Planificación de Orden Total

### 6.2.1 Planificación como Búsqueda en un Espacio de Estados

El enfoque más simple en planificación consiste en considerarla como un problema de búsqueda en el que aplicamos las técnicas de búsqueda que ya conocemos. El **objetivo de la planificación** equivale a un estado final, las **acciones disponibles** equivalen a los operadores de búsquedas y se generará un árbol de búsqueda cuyos nodos equivalen a estados.

Como las descripciones de acciones especifican tanto precondiciones como efectos, es posible realizar la búsqueda en ambas direcciones, hacia delante y hacia atrás.

#### Sistema de resolución hacia delante

- Usa la descripción del estado actual como base de datos global
- Los operadores son reglas tipo STRIPS
- Seleccionamos reglas aplicables hasta que se produzca una descripción del estado que satisfaga la expresión objetivo

**Sistema de resolución hacia atrás**

1. Partimos de la expresión objetivo usada como base de datos global
2. Llegamos al estado inicial mediante aplicación de reglas inversas, que transforman expresiones de objetivos en expresiones de subobjetivos

//CAUTION ZONE//

//Esta parte está tan mal explicada que no soy capaz de condensarla de forma clara//

Así, sea una regla concreta $D$, con precondición $P$ y adición $A$, y el objetivo $\{L \wedge G_1 \wedge ... \wedge G_N\}$. Entonces, aplicamos $D$ de forma inversa para generar un subobjetivo $S$ a partir del objetivo:

- para llegar al objetivo concreto se hace a través de la operación de adición
- supongamos que en $A$ tenemos un $L'$ que se unifica con $L$ mediante un unificador más general $u$, y la particularización de las precondiciones es $Pu$. Entonces, los literales de $Pu$ son un subconjunto del subobjetivo buscado, pues serán necesarios para aplicar $D$
- en el subobjetivo debemos incluir, también, $G'_1 \wedge ... \wedge G'_N$, que deben ser tales que al aplicar la particularización de la regla $D$ a una descripción de estado que satisfaga esas expresiones, produzca una descripción que satisfaga $G_1 \wedge ... \wedge G_N$. A este proceso se le denomina **regresión**

Entonces, el **proceso de regresión** de un literal $G$ es el siguiente:

- denotamos $R\,[G, D_u]$ a la regresión de $G$ (o sea, $G'$) a través de la regla $D$ con precondición $P$, lista de supresión $S$ y de adición $A$
- sean $S_u, A_u$ las particularizaciones de los literales de $A$ y $S$ para $D_u$, entonces:
  - si $G_u$ es un literal de $S_u$ entonces $R\,[G, D_u] = False$ y $D$ no es aplicable
  - si $G_u$ es un literal de $A_u$ entonces $R\,[G, D_u] = True$ y no se añade
  - en otro caso, $R\,[G, D_u] = G_u$

//END CAUTION ZONE//

Entonces, aplicando todas las reglas inversas el espacio de subobjetivos es más amplio que el espacio de estados que se obtiene al aplicar todas las reglas en el sistema hacia delante, pero muchas de las descripciones de subobjetivos son imposibles, por lo que se pueden podar, reduciendo sustancialmente el espacio de búsqueda.

En la mayoría de problemas del mundo real, la aplicación de cualquiera de los dos métodos no es práctica, debido a la cantidad de tiempo y espacio requerida para encontrar la solución. Dado que no es viable una búsqueda exhaustiva del espacio, se deberán emplear heurísticas.

### 6.2.2 Planificación secuencial usando una pila de objetivos: STRIPS

STRIPS es un sistema de planificación lineal, o sea, no se pasa al objetivo siguiente hasta que no se ha resuelto completamente el actual. Para ello, utiliza una pila de subobjetivos, en lugar de un conjunto, y funciona bien cuando no interdepencias o estas son débiles.

El pseudocódigo del proceso es:

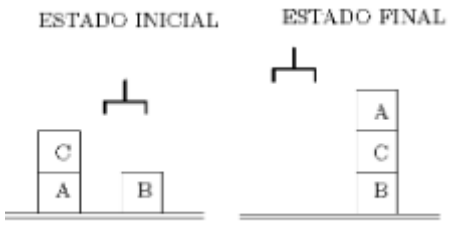
```
función STRIPS(estado-inicial, objetivo)
{
        estado <- estado-inicial; plan <- {}; pila <- {};
        pila.push(objetivo)
        repetir hasta VACIO(pila)
                obj <- pila.head
                si (MATCH(obj, estado.descripcion)) entonces
                        pila.pop
                        para cada expresión en pila hacer {aplicar la
                           sustitución}
                sino, si (obj es conjunción de objetivos) entonces
                        objs <- ordenarlos
                        para cada o en objs hacer pila.push(o)
                sino, si (todos los subobjetivos resueltos Y !(objetivo
                   resuelto)) entonces
                        {reconsiderar objetivo compuesto y apilar}
                sino, si (obj es un objetivo simple) entonces
                        elegir operador OP tal que EMPAREJA(OP.
                           listaAdicion,obj)
                        REEMPLAZAR(obj, OP)
                        para cada precondicion en OP.listaPrecondiciones
                            hacer pila.push(precondicion)
                sino, si (obj es un operador OP) entonces
                        estado <- APLICAR(OP, estado)
                        plan <- plan + {OP}
}
```

La componente de control del sistema de resolución asociado a STRIPS debe tomar diversas decisiones, entre las que se incluyen:

1. Ordenación de las componentes de un objetivo compuesto. Se pueden obtener planes subóptimos por una ordenación deficiente de los objetivos a satisfacer
2. Elección entre las distintas particularizaciones posibles
3. Selección de operadores relevantes cuando hay más de uno

**Ejemplo STRIPS**

La formulación del problema, visualmente, es




Y esto podemos expresarlo como

| Estado | Objetivo |
|---|---|
| Libre(B) | Sobre(C,B)∧Sobre(A,C) |
| Libre(C) | |
| Sobre(C,A) | |
| Sobremesa(A) | |
| Sobremesa(B) | |
| Manovacia | |

Los operadores disponibles son:

| | |
|---|---|
| coger(x) | P y S: Sobremesa(x), Manovacia, Libre(x) |
| | A: Cogido(x) |
| dejar(x) | P y S: Cogido(x) |
| | A: Sobremesa(x), Libre(x), Manovacia |
| apilar(x,y) | P y S: Cogido(x), Libre(y) |
| | A: Manovacia, Sobre(x,y), Libre(x) |
| desapilar(x) | P y S: Manovacia, Libre(x), Sobre(x,y) |
| | A: Cogido(x), Libre(y) |

Como el objetivo es compuesto, se apilan los objetivos simples en un orden determinado

| Estado | Objetivo |
|---|---|
| Libre(B) | Sobre(C,B |
| Libre(C) | Sobre(A,C) |
| Sobre(C,A) | Sobre(C,B)∧Sobre(A,C) |
| Sobremesa(A) | |
| Sobremesa(B) | |
| Manovacia | |

En la cima de la pila hay un objetivo simple, luego se escoge un operador y se añaden sus precondiciones

| Estado | Objetivo |
|---|---|
| Libre(B) | Libre(B)∧Cogido(C) |
| Libre(C) | apilar(C,B) |
| Sobre(C,A) | Sobre(C,B |
| Sobremesa(A) | Sobre(A,C) |
| Sobremesa(B) | Sobre(C,B)∧Sobre(A,C) |
| Manovacia | |

En la cima vuelve a haber un objetivo compuesto, se apilan sus objetivos simples en un orden determinado

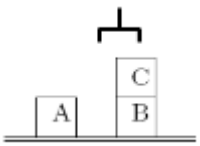
| Estado | Objetivo |
|---|---|
| Libre(B) | Cogido(C) |
| Libre(C) | Libre(B) |
| Sobre(C,A) | Libre(B)∧Cogido(C) |
| Sobremesa(A) | apilar(C,B) |
| Sobremesa(B) | Sobre(C,B |
| Manovacia | Sobre(A,C) |
| | Sobre(C,B)∧Sobre(A,C) |

A partir del objetivo simple Cogido(c), se escoge un operador y se apilan sus precondiciones

| Estado | Objetivo |
|---|---|
| Libre(B) | Manovacia∧Libre∧Sobre(C,y) |
| Libre(C) | Desapilar(C,y) |
| Sobre(C,A) | Cogido(C) |
| Sobremesa(A) | Libre(B) |
| Sobremesa(B) | Libre(B)∧Cogido(C) |
| Manovacia | apilar(C,B) |
| | Sobre(C,B |
| | Sobre(A,C) |
| | Sobre(C,B)∧Sobre(A,C) |

Tras sucesimos emparejamientos de objetivos simples y aplicación de los dos operadores que serán añadidos al plan, que ahora mismo será PLAN={Desapilar(C,A),Apilar(C,B)}, obtenemos

| Estado | Objetivo |
|---|---|
| Libre(C) | Sobre(A,C) |
| Libre(A) | Sobre(C,B)∧Sobre(A,C) |
| Sobre(C,B) | |
| Sobremesa(A) | |
| Sobremesa(B) | |
| Manovacia | |




Se escoge ahora el operador apilar y se insertan sus precondiciones en la pila

| Estado | Objetivo |
|---|---|
| Libre(C) | Libre(C)∧Cogido(A) |
| Libre(A) | Apilar(A,C) |
| Sobre(C,B) | Sobre(A,C) |
| Sobremesa(A) | Sobre(C,B)∧Sobre(A,C) |
| Sobremesa(B) | |
| Manovacia | |

Estas precondiciones se apilan una a una

| Estado | Objetivo |
|---|---|
| Libre(C) | Libre(C) |
| Libre(A) | Cogido(A) |
| Sobre(C,B) | Libre(C)∧Cogido(A) |
| Sobremesa(A) | Apilar(A,C) |
| Sobremesa(B) | Sobre(A,C) |
| Manovacia | Sobre(C,B)∧Sobre(A,C) |

Como Libre(C) empareja con el estado, se desapila y continuamos

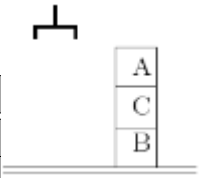
| Estado | Objetivo |
|---|---|
| Libre(C) | |
| Libre(A) | Cogido(A) |
| Sobre(C,B) | Libre(C)∧Cogido(A) |
| Sobremesa(A) | Apilar(A,C) |
| Sobremesa(B) | Sobre(A,C) |
| Manovacia | Sobre(C,B)∧Sobre(A,C) |

Se selecciona el operador coger y se apilan sus precondiciones

| Estado | Objetivo |
|---|---|
| Libre(C) | Sobremesa(A)∧Libre(A)∧Manovacia |
| Libre(A) | Coger(A) |
| Sobre(C,B) | Cogido(A) |
| Sobremesa(A) | Libre(C)∧Cogido(A) |
| Sobremesa(B) | Apilar(A,C) |
| Manovacia | Sobre(A,C) |
| | Sobre(C,B)∧Sobre(A,C) |

Esas precondiciones se apilan una por una, vemos que las tres están en el estado luego se desapilarán. Estaremos con Coger en el tope de la pila, por lo tanto lo desapilamos y los añadimos al plan, PLAN={Desapilar(C,A), Apilar(C,B), Coger(A)} y tenemos:

| Estado | Objetivo |
|---|---|
| Libre(A) | Cogido(A) |
| Sobre(C,B) | Libre(C)∧Cogido(A) |
| Cogido(A) | Apilar(A,C) |
| Sobremesa(B) | Sobre(A,C) |
| Manovacia | Sobre(C,B)∧Sobre(A,C) |




Como Cogido(A) se encuentra en el estado, se desapila y habremos completado el siguiente objetivo, nos quedaremos con Apilar(A,C) en el tope, que lo despilaremos y añadiremos al PLAN={Desapilar(C,A), Apilar(C,B),Coger(A),Apilar(A,C)}, obtenemos

| Estado | Objetivo |
|---|---|
| Libre(A) | Sobre(A,C) |
| Sobre(C,B) | Sobre(C,B)∧Sobre(A,C) |
| Sobre(A,C) | |
| Sobremesa(B) | |
| Manovacia | |

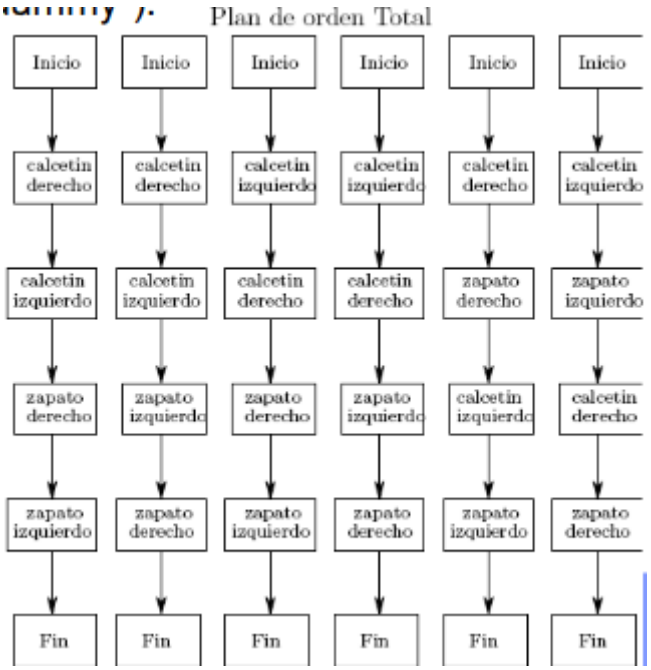
Y el último subobjetivo queda cumplido, luego hemos terminado, queda PLAN={Desapilar(C,A), Apilar(C,B),Coger(A),Apilar(A,C)}.

### 6.2.3 Planificación Ordenada Parcialmente

Un **planificador no lineal** construye un plan cuyo orden no es total. Se basan en la posible descomposición del problema, trabajando en varios subobjetivos de manera independiente, solucionados con subplanes y creando un plan a partir de ellos.

La idea básica es trabajar con secuencias de dos acciones, sin distinguir el orden dentro de la secuencia.

Un **plan de orden parcial (POP)** es un grafo de acciones, en el cada una de las soluciones de orden total se denomina linealizaciñón del POP. Por ejemplo, supongamos que nos queremos calzar, entonces, los distintos planes de orden total son




Mientras que el POP quedaría


Y, por lo general, los POP generan un espacio más pequeño, por lo que la búsqueda será menos costosa.

Las ideas básicas son:

- Planificación de compromiso mínimo: escoger solo acciones y particularizaciones estrictamente necesarias
- Uso de restricciones temporales

Los **arcos amenaza** representarán situaciones en las que el resultado de aplicar una acción influye en la precondición de otra acción, por lo que nos ayudarán a determinar el orden en el que deben ocurrir las acciones.

### 6.2.4 Planificación Jerárquica

La idea de este tipo de planificación es jerarquizar el problema en diferentes niveles de complejidad, y resolver gradualmente aumentando progresivamente la complejidad. Es un enfoque **de arriba a abajo** o **top-down**. El sistema de referencia es ABSTRIPS.

El principal argumento a su favor es que el resolver un problema medianamente coplejo no es posible con un planteamiento de búsqueda completa. Así, las **jerarquías** se crean dependiendo de valores críticos de las precondiciones de los oepradores: a más dificultad de satisfacer la precondición, mayor valor crítico, y en cada nivel de la jerarquía se tratan precondiciones con el mismo valor crítico, comenzando con las que tienen el mayor, asumiendo las demás satisfechas.

# 7 Introducción al Aprendizaje Computacional

Con el término **aprendizaje** nos referimos al proceso por el que una entidad acrecienta su conocimiento y/o habilidad. Es un aspecto esencial de la inteligencia. Y uno de sus elementos imprescindibles es la experiencia.

Un programa no puede denominarse inteligente si no tiene capacidades de adaptación a cambios. Por tanto, es necesario conseguir programas que tengan la habilidad de aprender.

Algunas razones que justifican el aprendizaje computacional son:

1. Algunas tareas solo pueden describirse bien mediante ejemplos
2. Es posible que escondidos en una gran cantidad de datos se encuentren relaciones y correlaciones valiosas
3. Algunas máquinas no trabajan tan bien inicialmente como se desearía a largo plazo
4. En ciertos dominios, la cantidad de conocimiento disponible puede sobrepasar la capacidad de codificación humana
5. Los entornos cambian con el tiempo, por lo que los programas deberían adaptarse a esos cambios
6. Constantemente se genera nuevo conocimiento

## 7.1 Conceptos básicos

### 7.1.1 Tipos de aprendizaje

- **Aprendizaje supervisado:** la experiencia viene en forma de ejemplos etiquetados
- **Aprendizaje no supervisado:** la experiencia viene en forma de observaciones del entorno

### 7.1.2 Fases del aprendizaje y Proceso de inferencia

Las fases del aprendizaje son el entrenamiento y las pruebas.

Debemos tener en cuenta tanto parámetros externos (aprendidos) como internos (no aprendidos).

La **base del aprendizaje** es la optimización de una función de rendimiento.

En la **fase de entrenamiento** se distingue entre supervisión o no supervisión.

En la **fase de prueba** no se hace tal distinción.

Por último, queda el proceso de inferencia, que consiste en utilizar el sistema entrenado para realizar predicciones a futuro.

### 7.1.3 Características deseables de un sistema de aprendizaje

- **Precisión**: fiabilidad del método aprendido, representada normalmente por la proporción de ejemplos aprendidos correctamente
- **Velocidad**: rapidez del proceso de predicción, puede que sea preferible reducir la precisión si se mejora considerablemente la velocidad
- **Comprensión**: si es un operador humano el que usa el modelo aprendido, debe ser fácil de entender para evitar errors

- **Tiempo en aprender**: de especial importancia en aprendizaje online aplicado a sistemas que cambian rápidamente. Puede implicar la necesidad de un número pequeño de observaciones para obtener un modelo adecuado

### 7.1.4 Estimación del error

La **matriz de confusión** sirve para determinar la precisión de un sistema de clasificación binario (en el que solo hay dos categorías) y tiene la siguiente forma

| | | Predicción | |
|---|---|---|---|
| | | False | True |
| Realidad | False | Verdaderos negativos | Falsos positivos |
| | True | Falsos negativos | Verdaderos positivos |

La matriz de confusión puede generalizarse para una cantidad arbitraria de categorías, donde la posición $x_{ij}$ indica que el objeto es realmente de categoría $i$ y ha sido clasificado como de categoría $j$ (obviamente puede ser $i = j$).

Algunos estimadores del error son:

- **Estimador del error de los ejemplos**: sea $N$ el número de ejemplos y sea $E$ el número de veces que el modelo se equivoca. El estimador del error es $\frac{E}{N}$.

  Es un buen estimador si los ejemplos no fueron utilizados en el proceso de aprendizaje.

- **Estimando el error de resustitución**: usamos para estimar el error el mismo conjunto de datos que para el entrenamiento. Por tanto, será una aproximación optimista al error real, tiende a subestimarlo.

- **Estimación del error por validación cruzada (cross validation)**: se usa para estimar el error de un método, no de un modelo concreto. Es muy útil en casos con escasos datos. Se sigue el siguiente procedimiento:

  1. Dividir el conjunto de ejemplos $S$ en $k$ partes disjuntas de tamaño similar
  2. Para $i = 1...k$
     (a) Se construye el modelo usando el método sobre el conjunto $S \setminus S_i$
     (b) Se determina el estimador del error de los ejemplos $R_i$ usando el conjunto de prueba $S_i$
  3. Se calcula el estimador del error por validación cruzada como

  $$\sum_{i=1}^{k} \frac{|S_i|}{|S|} R_i$$

  con $|S_i|, |S|$ los cardinales de $S_i$ y $S$.

  Así, este método realiza varias estimaciones a partir de diferentes combinaciones de conjuntos de entrenamiento y prueba, y las agrega. Si seleccionamos $k = 10$, estamos ante **10-fold cross-validation**. Otro caso particular es el **leave-one-out** consistente en hacer $k = |S|$, es decir, se toman los conjuntos unipuntuales.

Es bastante costoso, por lo que se usa con conjuntos pequeños, donde estimar simplemente el error de los ejemplos puede no resultar significativo.

En algunos dominios, algunos errores son más costosos que otros (por ejemplo, en diagnóstico médico un falso negativo puede costar la vida al paciente).

Así, tiene sentido poner costes a los fallos. La expresión del coste de cada error se especifica con una matriz análoga a la matriz de confusión, denominada **matriz de costos**, y donde $c_{ij}$ es el coste de clasificar un objeto de la clase $i$ como si fuese de la $j$, por ejemplo, podría ser

| | | Predicción | |
|---|---|---|---|
| | | False | True |
| Realidad | False | 0 | 1 |
| | True | 5 | 0 |

Si $E_{ij}$ es el número de ejemplos de clase $i$ clasificados como de la clase $j$, el **estimador del coste de la mala clasificación** es

$$c = \sum_{i=1}^{N} \sum_{j=1}^{N} c_{ij} E_{ij}$$

Cuando el modelo no es de clasificación, sino que es numérico, un buen estimador es el **error cuadrático medio (MSE)**:

$$MSE = \frac{\sum_{i=1}^{N} (y_i^* - y_i)^2}{N}$$

donde $y_i^*$ es el valor real e $y_i$ es el inferido por el modelo.

## 7.2 Algunos Sistemas Básicos de Aprendizaje e Inferencia

### 7.2.1 Aprendizaje Memorístico

La idea básica es almacenar todo el nuevo conocimiento para utilizarlo cuando sea preciso. Todo sistema de aprendizaje debe recordar lo aprendido para poder utilizarlo en el futuro, por lo que esta parte puede estar integrada en un sistema aprendiz más complejo.

Lo que hará será tomar problemas ya resueltos por el sistema, y memorizarlos junto con su solución.

En el aprendizaje memorístico podemos destacar algunos elementos:

- **Organización de la memoria**: se ha de utilizar memorización siempre que sea más conveniente que recalcular
- **Estabilidad del entorno**: la memoria solo es útil si los recuerdos sirven para resolver problemas futuros. Por tanto, en entornos rápidamente cambiantes, lo almacenado puede quedar desfasado rápidamente, y no es conveniente usarlo en este caso
- **Almacenamiento frente a recálculo**: el aprendizaje por memorización debe ser eficiente. Se pueden seguir dos enfoques para conseguir esta eficiencia:
  - Análisis coste-beneficio: cada vez que llega nueva información, se decide si se almacena o no
  - Olvido selectivo: se almacena todo y se olvida lo que se usa poco frecuentemente

### 7.2.2 Aprendizaje en la resolución de problemas

Un programa para resolver problemas puede aprender a partir de la generalización de sus propias experiencias:

- al resolver un problema, puede recordarse su estructura, lo métodos que llevaron a su solución, y esta solución
- más adelante, podemos utilziar esta experiencia si nos encontramos con el mismo problema, o generalizarla para que pueda ser usada ante problemas similares

STRIPS, para llevar a cabo esta función, da uso a **macro-operadores**, que consisten en que, tras cada episodio de planificación, un componente de STRIPS toma el plan calculado o una parte del mismo, y lo transforma en un macro-operador, que consiste en:

- precondiciones: las condiciones iniciales del problema resuelto
- postcondiciones: el objetivo alcanzado mediante el plan utilizado

O sea, que es un nuevo operador, que abstrae un conjunto de operadores básicos del sistema ejecutados en un cierto orden. Se trata como un operador normal.

No obstante, raramente se presenta el mismo problema dos veces, por lo que es preciso generalizar, transformando el macro-operador en uno más genérico que pueda ser equiparado con una familia de problemas, y no únicamente con el problema que lo generó. Para este propósito, se sustituyen las constantes por variables.

Pero, a menudo, la generalización no es tan directa, y algunas constantes deben retener su valor. Esto plantea algunas dificultades, pues debemos determinar qué constantes podemos cambiar por variables, y cuáles deben permanecer como están.

Así, el proceso de generalización que aplica STRIPS es un proceso complejo que hace los siguientes pasos:

1. Sustituye todas las constantes por variables
2. Reevalúa todas las precondiciones de cada operador del plan parametrizado. Aquellas precondiciones que requieran que alguna variable tome siempre un valor concreto, obligan al sistema a hacer que ese parámetro no sea una variable, sino ese valor específico

Los macro-operadores, bien utilizados, resultan muy útiles. Un macro-operador puede dar lugar a un pequeño cambio global en el mundo, aunque los operadores individuales que lo forman produzcan muchos cambios locales no buscados.

### 7.2.3 Aprendizaje de Reglas de Asociación

Una **regla de asociación** relaciona los distintos elementos que pueden aparecer en una base de datos relacional o transaccional. A diferencia de las reglas de un SBR, no expresa causalidad. Se obtienen a partir de observaciones:

- aquellas cuyo antecedente y consecuente son ciertos en muchas de las observaciones se dice que tienen una **cobertura alta**

- aquellas que, en las observaciones, cuando se cumple el antecedente también suele cumplirse el consecuente, se dice que tienen una **confianza alta**

Nuestro objetivo es, por tanto, encontrar reglas interesantes, con valores buenos de cobertura y confianza.

Encontrar todos los subconjuntos frecuentes de la base de datos es difícil ya que esto implica considerar todos los posibles subconjuntos de items (combinaciones de items). El conjunto de posibles conjuntos de items es el conjunto potencia de $I$ y su tamaño es de $2^n - 1$ (excluyendo el conjunto vacío que no es válido como conjunto de items). Aunque el tamaño del conjunto potencia crece exponencialmente con el número de ítems, $n$, de $I$, es posible hacer una búsqueda eficiente utilizando la propiedad "downward-closure" del soporte (también llamada anti-monótona) que garantiza que para un conjunto de items frecuente, todos sus subconjuntos también son frecuentes, y del mismo modo, para un conjunto de items infrecuente, todos sus superconjuntos deben ser infrecuentes. Explotando esta propiedad se han diseñado algoritmos eficientes para encontrar los items frecuentes.

Notación:

- $I = \{I_1, ..., I_n\}$ es un conjunto de $n$ atributos arbitrarios, llamados ítems
- $D = \{t_1, ..., t_m\}$ es un conjunto de $m$ ejemplares o transacciones de un problema
- $t_i$ también se denomina *itemset*, es un conjunto de items de $I$

Una **regla de asociación** será una expresión de la forma

$$X \implies Y$$

donde $X, Y \subset I$ y $X \cap Y = \emptyset$. Y, definimos las medidas anteriores de forma precisa:

- **Confianza**: la confianza de una regla $X \implies Y$ se interpreta como la probabilidad de que una trasacción que contiene los ítems de $X$ también contenga los ítems de $Y$:

$$c\left(X \implies Y\right) = P\left(Y|X\right) = \frac{|\{X \cup Y \subset t_k\}|}{|\{X \subset t_k\}|}$$

- **Soporte**: el soporte de una regla $X \implies Y$ se interpreta como la probabilidad de que una transacción contenga a todos los ítems de $X$ y de $Y$

$$s\left(X \implies Y\right) = P\left(X \cup Y\right) = \frac{|\{X \cup Y \subset t_k\}|}{|D|}$$

Nos interesan reglas con mucho soporte, por lo que buscamos pares $(atributo, valor)$ que cubran una gran cantidad de transacciones o itemsets. Una vez tenemos los itemsets, los transformamos en reglas.

Para obtener las reglas, usaremos un proceso de dos fases:

1. Generar todas las posibles combinaciones de ítems, entendiendo estas como itemsets, con un soporte por encima de un mínimo
2. Dado un itemset frecuente $Y = \{i_1, ..., i_k : k \geq 2\}$, generamos todas las reglas que contengan todos los items de ese itemset, generando todos los

   $$X \implies (Y \setminus X)$$

   donde $\emptyset \neq X \subset Y$ que verifique que su confianza es mayor que la mínima requerida:

   $$c\left(X \implies (Y \setminus X)\right) = \frac{s\left(Y\right)}{s\left(X\right)} \geq c_{min}$$

**Algoritmo A PRIORI**

Definimos los itemsets frecuentes como aquellos itemsets con soporte mínimo. Como hemos mencionado antes, se verifica:

- Un subconjunto de un itemset frecuente es también un itemset frecuente
- Si un itemset no satisface el soporte mínimo, ningún superconjunto suyo lo satisface

Así, iterativamente, se encuentran itemsets frecuentes con cardinalidad 1 hasta $k$.

**Fase 1**

```
función APRIORI
{
        Ck: itemsets candidatos de tamaño k
        Lk: itemsets frecuentes de tamaño k
        L1 = {itemsets frecuentes de tamaño 1}
        para (k=1; !VACIO(Lk); k++) hacer
                C_{k+1}=APRIORI-CANDIDATOS(Lk)
                para cada transaccion t en D hacer
                        Ct=subset(C_{k+1},t)
                        para cada candidato c en Ct hacer c.count++
                L_{k+1}={c en C_{k+1}|c.count > minSoporte}
        devolver {la unión de todos los Lk}
}

función APRIORI-CANDIDATOS(Lk)
{
        FASE DE FORMACIÓN
                insertar en C_{k+1} todas las combinaciones de los
                   itemsets desde Lk para formar itemsets de tamaño k+1

        FASE DE PODA
                para cada itemset c en C_{k+1} hacer
                        para cada k-subconjunto s de c hacer
                                si s notin Lk entonces
                                        borrar c de C_{k+1}

        devolver C_{k+1}
}
```

**FASE 2**

```
función GENERAR_REGLAS_APRIORI
{
        para cada (itemset-frecuente f) hacer
                generar todos los k-subconjuntos no vacíos de f con k>=2

        para cada (k-subconjunto no vacío, S, de f) hacer
                si s(f)/s(S) >= minConfianza entonces
                        regla de salida (s -> (f-s))
}
```