Inteligencia Artificial (IA)
Grado en Ingeniería Informática

Profesor: Carlos Ansótegui

Área: Ciencias de la Computación e Inteligencia Artificial (CCIA)
Departamento de Informática e Ingeniería Industrial
Escuela Universitaria Politécnica
Universidad de Lleida

12 de septiembre del 2012

Índice

Definiciones de la IA

- John McCarthy acuño el término de Inteligencia Artificial en 1956, definiéndolo como “la ciencia e ingeniería de construir máquinas inteligentes”.
- IA es la inteligencia de las máquinas y la rama de las ciencias de la computación que cuyo objetivo es crearla.
- Inteligencia en la parte computacional de la habilidad para alcanzar objetivos en el mundo. Diferentes tipos y grados de inteligencia se dan en las personas, animales y algunas máquinas.
- IA es el estudio de las facultades mentales a través del uso de modelos computacionales.
- IA es el estudio de cómo hacer que la máquinas realicen tareas que, por el momento, los humanos realizan mejor.
- IA es el estudio y diseño de agentes inteligentes, donde un agente inteligente es un sistema que percibe su entorno y toma acciones que maximizan sus oportunidades de éxito.

Inteligencia

- Concepto relativo a tareas que involucran procesos mentales superiores: creatividad, resolución de problemas, reconocimiento de patrones, clasificación, aprendizaje, procesamiento, conocimiento, etc.
- Inteligencia es la parte computacional de la habilidad para conseguir objetivos.

Comportamiento inteligente

- Percepción del entorno
- Actuar en entornos complejos
- Aprender y entender de la experiencia
- Razonar para resolver problemas y descubrir nuevo conocimiento
- Aplicación del conocimiento en nuevas situaciones.
- Pensar de forma abstracta, utilizando analogías
- Comunicarse con otros
- Creatividad, ingenio, expresividad, curiosidad, etc.

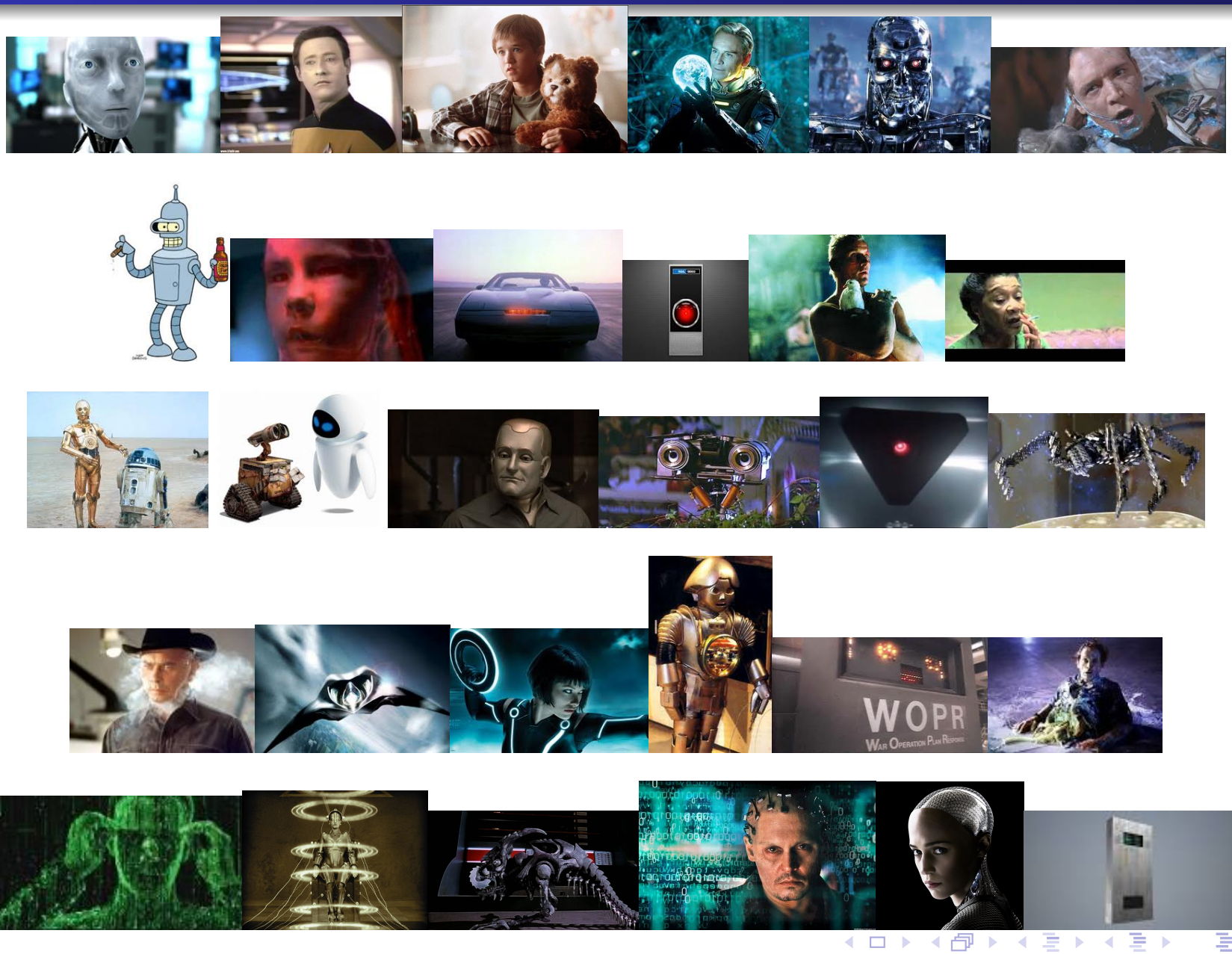
Compresión de la IA

- Cómo se adquiere el conocimiento y se almacena
- Cómo el comportamiento inteligente se genera y aprende
- Cómo las motivaciones, emociones y prioridades se desarrollan y usan
- Cómo las señales sensoriales se transforman en símbolos
- Cómo los símbolos se manipulan para aplicar la lógica, razonar sobre el pasado y planear sobre el futuro.
- Cómo los mecanismos de la inteligencia producen el fenómeno de la ilusión, creencia, esperanza, miedo, sueños, bondad, amor, etc.

IA con mayúsculas (Strong AI)

- Generalmente, la investigación en IA tiene como objetivo último replicar la inteligencia humana completamente.
- Strong AI es un término que se aplica a una máquina, que se apróxima o supera la inteligencia humana
  - si puede realizar la tareas típicamente humanas
  - si puede aplicar una amplia gama de conocimientos básicos
  - si tiene cierto grado de autoconciencia
- Strong AI pretende crear máquinas cuya habilidad intelectual es indistinguible de la de un humano.

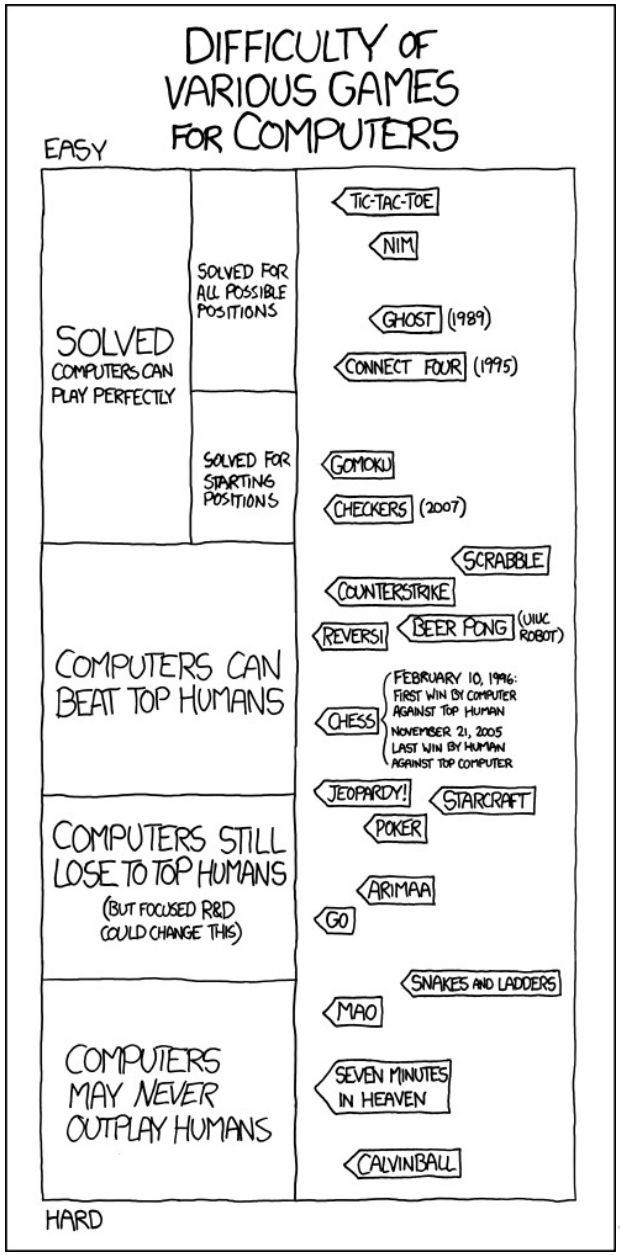
IA en el cine



ia con minúsculas (Weak AI)

- El término Weak AI se refiere al uso de software para estudiar o lograr la resolución de problemas o tareas de razonamiento que no abarcan la totalidad de las habilidades cognitivas de los humanos. Ej: IBM: deep Blue para el ajedrez o Watson para Jeopardy.

ia con minúsculas (Weak AI)



Objetivos de la IA

Las definiciones de la IA nos llevan a 4 posibles objetivos:

1. Sistemas que piensan como los humanos
2. Sistemas que piensan racionalmente
3. Sistemas que actúan como los humanos
4. Sistemas que actúan racionalmente

La mayoría del trabajo en IA recae en los objetivos (2) y (4).

Definiciones en libros de texto

	como humanos	racionalmente
pensar	*Enfoque de la ciencia cognitiva* 'The exciting new effort to make computers think ... machines with minds, in the full and literal sense' (Haugeland, 1985) 'The automation of activities that we associate with human thinking, activities such as decision-making, problem solving, learning ...' (Bellman, 1978)	*Enfoque de "leyes del pensamiento"* 'The study of mental faculties through the use of computational models' (Charniak and McDermott, 1985) 'The study of the computations that make it possible to perceive, reason, and act' (Winston, 1992)
actuar	*Enfoque del test de turing* 'The art of creating machines that perform functions that require intelligence when performed by people' (Kurzweil, 1990) 'The study of how to make computers do things at which, at the moment, people are better' (Rich and Knight, 1991)	*Enfoque del agente racional* 'A field of study that seeks to explain and emulate intelligent behavior in terms of computational processes' (Schalkoff, 1990) 'The branch of computer science that is concerned with the automation of intelligent behavior' (Luger and Stubblefield, 1993)

Prehistoria de la IA

Filosofía	lógica, métodos de razonamiento la mente como un sistema físico fundamentos del aprendizaje, lenguaje, racionalidad
Matemáticas	representación formal y demostraciones algoritmos, computación, (in)decibilidad, (in)tractabilidad probabilidad
Psicología	adaptatción fenómeno de la percepción y control motor técnicas experimentales (psicofísica, etc.)
Económicas	teoría formal de decisones racionales
Linguística	representación del conocimiento gramáticas
Neurosciencia	física de la actividad mental
Teoría de control	sistemas homeostáticos, estabilidad diseños simples de agentes óptimos

Historia de la IA

1943	McCulloch & Pitts: circuito Booleano modelando neuronas
1950	Turing's "Computing Machinery and Intelligence"
1952–69	Mira Mamá, sin manos!
1950s	Primeros programas en IA, Samuel's checkers,
1956	conferenciad de Dartmouth: "Intelligencia Artificial" acuñado
1965	algoritmo completo de Robinson para el razonamiento lógico
1966–74	la IA descubre la complejidad computacional
	La investigación en redes neuronales casi desaparece
1969–79	Primer desarrollo de sistemas expertos
1980–88	Boom de la industria de los sistemas expertos
1988–93	La industria de los sistemas expertos decae: "Invierno de la IA"
1985–95	Las redes neuronales vuelven a ser populares
1988–	Resurge la probabilidad
	"Nouvelle AI": ALife, GAs, soft computing
1995–	Agentes, agentes, por doquier . . .
2003–	Strong AI vuelve a la agenda

La IA en la actualidad

En la conferencia AAAI del 2012 encontramos las siguientes áreas temáticas:

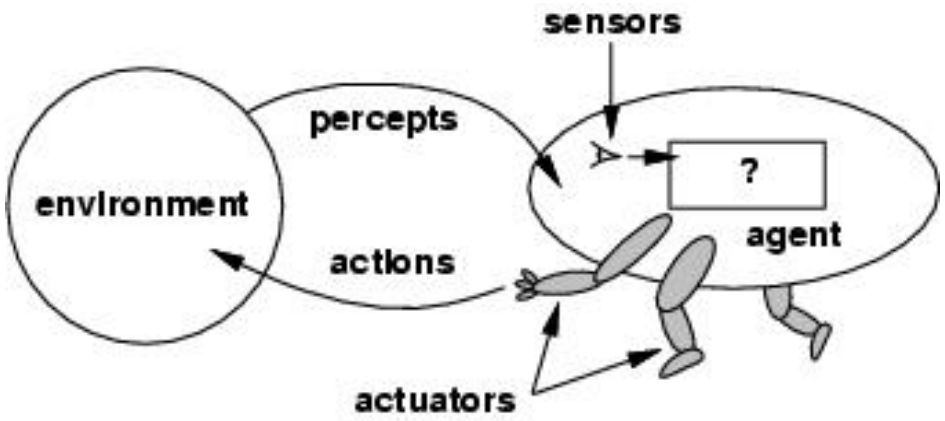
Agent-based and multiagent systems
Cognitive modeling and human interaction
Commonsense reasoning
Computer vision
Constraint satisfaction, search, and optimization
Evolutionary computation
Game playing and interactive entertainment
Information retrieval, integration, and extraction
Knowledge acquisition and ontologies
Knowledge representation and reasoning
Machine learning and data mining
Model-based systems
Multidisciplinary artificial intelligence
Natural language processing
Planning and scheduling
Probabilistic reasoning
Robotics
Web and information systems

Agentes y entornos

Un **agente** es cualquier cosa capaz de percibir su entorno a través de sensores y de actuar sobre él a través de actuadores.
Ej: agente humano, agente robótico, agente de software, ...

Un **agente** racional es aquel que realiza la acción correcta.
Entendemos por acción correcta aquella que le conduce al mayor éxito a la hora de realizar una tarea.

Los **entornos** son aquellos problemas para los que los agentes racionales son las soluciones.



Propiedades de los entornos

Podemos identificar algunas propiedades de los entornos:

- Totalmente observable vs. parcialmente observable:
  Decimos que un entorno es totalmente observable para un agente si sus sensores detectan todos los aspectos necesarios para elegir una acción.
- Determinista vs. estocástico:
  Decimos que un entorno es determinista si el siguiente estado es determinado por el estado actual y l a acción que toma el agente. Si el entorno es determinista excepto por la acción de otros agentes, decimos que es **estratégico**.
- Episódico vs. secuencial:
  Decimos que un entorno es episódico si la experiencia del agente puede dividirse en episodios atómicos. En estos entornos, la elección de una acción depende exclusivamente de las condiciones del episodio.

Propiedades de los entornos

- Estático vs. dinámico:
  Decimos que el entorno es dinámico si puede variar mientras el agente toma una decisión. Si el entorno no cambia, pero si lo hace el rendimiento del agente entonces es **semidinámico**.
- Discreto vs. contínuo:
  La distinción entre discreto y contínuo se aplica al estado del entorno, a cómo se maneja el tiempo, a las percepciones y alas acciones del agente.
- Agente único vs. multiagente:
  Decimos que un entorno es multiagente si interviene más de una agente. Si loas agentes compiten decimos que es **competitivo**, y si los agentes pueden cooperar para mejorar su rendimiento respectivo decimos que es **cooperativo**.

Índice

Introducción

La mayoría de problemas en IA requiere, normalmente, determinar (**buscar**) una secuencia de acciones o decisiones, que nos lleven de una situación inicial a un objetivo o situación final.

Esta secuencia se ejecuta posteriormente para alcanzar el objetivo a partir de una situación inicial dada.

Por **búsqueda** nos referimos a un método computacional para resolver problemas.

El primer paso en la resolución de un problema consiste en **formular un objetivo** con una medida de rendimiento asociada en base a una situación inicial.

La **formulación de un problema** es el proceso de decidir qué estados y acciones deben considerarse dado un objetivo.

Introducción

Un **estado** es la representación de la configuración (física) del problema en un instante del proceso de resolución.

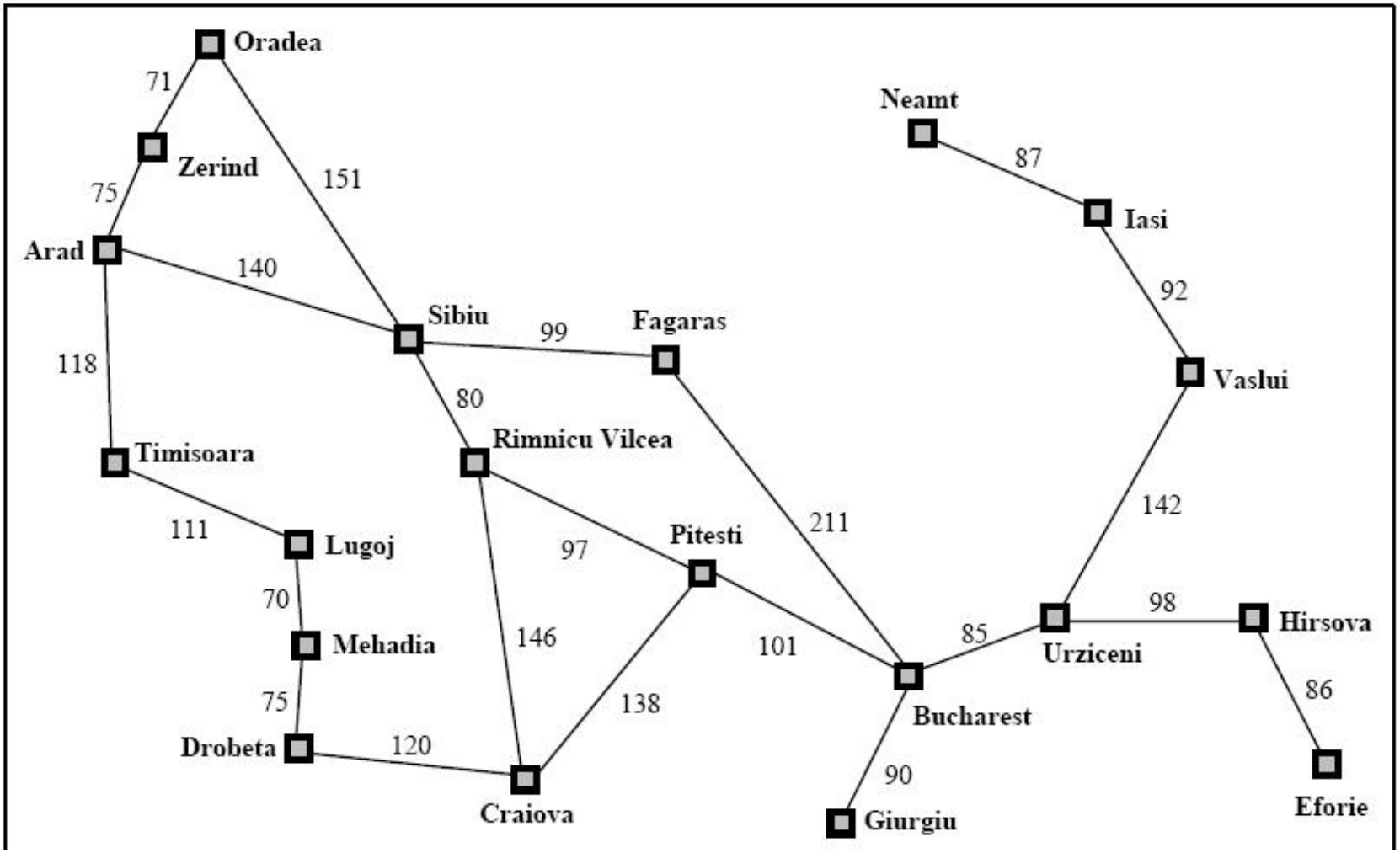
La ejecución de una **acción** nos permite transitar de un estado a otro.

Las acciones tienen un **coste** asociado que se trata de minimizar o un beneficio que se trata de maximizar.

En las técnicas de resolución que presentaremos, supondremos que el entorno es observable, discreto y determinista, y la solución es una secuencia fija de acciones.

Introducción

Pensemos en el problema de encontrar una ruta óptima de Arat a Bucarest dado el siguiente mapa de Rumania:



Problemas bien definidos

Definimos formalmente un problema p en base a cinco componentes:

1. El **estado inicial**. El estado inicial en que se encuentra el agente, ej: $in(Arad)$. Potencialmente, podemos considerar varios estados en los que se puede encontrar inicialmente el agente. La función $p.initials()$ retorna los estados iniciales.
2. Las **acciones** que puede ejecutar el agente en un determinado estado s, representado por la función $p.actions(s)$. Ej: desde $in(Arad)$ las acciones posibles son $\{go(Sibiu), go(Timisoara), go(Zerind)\}$.
3. Una descripción del efecto de las acciones. Formalmente se denomina **modelo de transición**. Dado un estado s y una acción a, la función $p.result(s, a)$, retorna el estado que resulta de aplicar la acción a sobre el estado s, ej: $result(in(Arad), go(Zerind)) = in(Zerind)$.

   Denominamos **sucesor** a cualquier estado alcanzable a partir de otro a través de una acción aplicable, y **función sucesor** a aquella función que nos retorna todos los sucesores de un determinado estado.

Problemas bien definidos

El estado inicial, las acciones y el modelo de transición definen implícitamente el espacio de estados. Diremos que el **espacio de estados** es el conjunto de estados que se pueden llegar a alcanzar partiendo del estado inicial aplicando recursivamente la función sucesor.

Podemos conceptualizar el espacio de estados como un grafo dirigido donde los nodos son estados y las acciones los arcos. Diremos que un **camino en el espacio de estados** es una secuencia de estados conectados por una secuencia de acciones. Ej: Podemos interpretar el mapa como el grafo del espacio de estados si consideramos cada ciudad como un nodo y cada conexión entre ciudades como un arco bidireccional.

4 El **test de objetivo**. Determina si un estado es el **estado objetivo o final**. Se puede definir explícitamente o implícitamente. Ej: explícito: $in(Bucharest)$. Puede existir más de un estado objetivo. Ej: implícito: estados de jaque mate en el ajedrez. La función $p.goal(s)$ devuelve cierto si el estado s es un estado objetivo.

Problemas bien definidos

5 El **coste de un cámino** es una función que asigna un coste numérico a cada camino. Asumimos que el coste de un camino se puede calcular como la suma de las acciones individuales a lo largo de un camino. El **coste de un paso** asociado a la acción a para ir del estado s al estado $p.result(s, a)$, viene dado ro la función $p.c(s, a)$. Asumiremos que los costes son ≥ 1. Ej: el coste de un paso en el mapa de Rumania se muestran como las distancias entre ciudades.

La **solución** a un problema es el camino (secuencia de acciones) desde el estado inicial a un estado objetivo. Una **solución optimal** es aquella que tiene el menor coste de camino entre todas las soluciones.

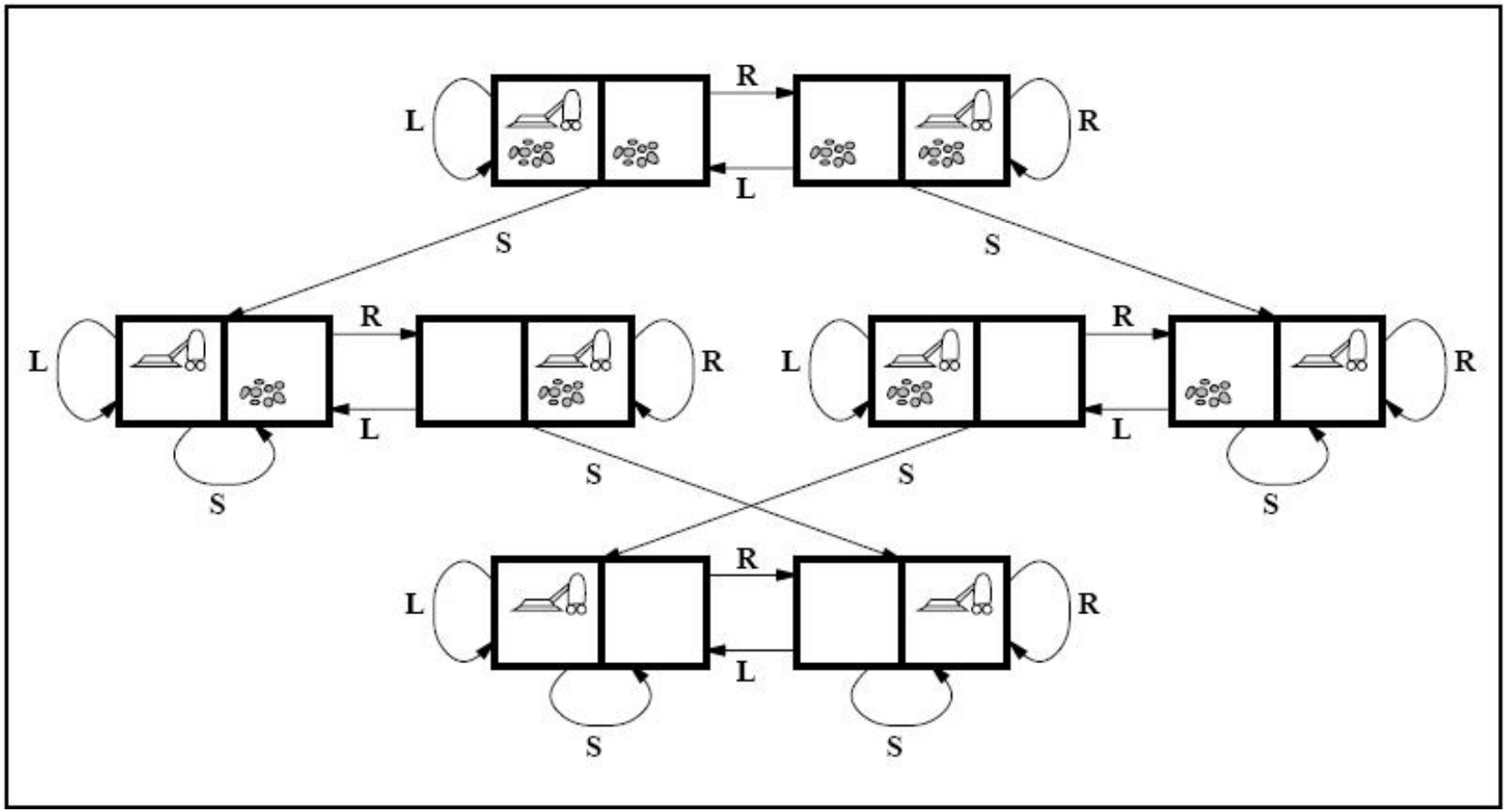
Formulación de problemas: *problemas académicos*

vacuum world problem: disponemos de un robot aspirador y dos celdas que pueden contener basura. Queremos eliminar toda la basura existente.

- Estados: el estado lo determinan la localización del robot y de la basura. El agente se encuentra en una de dos celdas, que pueden o no contener basura. Por lo tanto disponemos, de $2 \cdot 2^2 = 8$ posibles estados.
- Estado inicial: cualquier estado puede ser designado como inicial.
- Acciones: disponemos de tres acciones posibles (`left`, `right` o `suck`).
- Modelo de transición: todas las acciones tienen sus efectos esperados excepto `left` en la celda de la izquierda, `right` en la celda de la derecha, y `suck` en una celda vacía.
- Test de objetivo: comprueba si ambas celdas limpias.
- Coste del camino: Cada paso cuesta 1, por lo tanto el coste del camino es la longitud o número de pasos del camino.

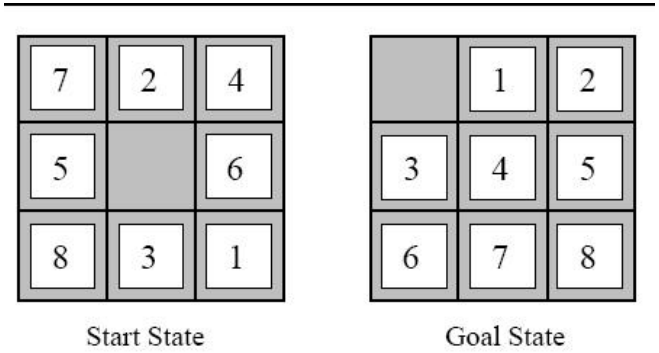
Formulación de problemas: *problemas académicos*

Grafo del espacio de estados del *vacuum world problem*:



Formulación de problemas: *problemas académicos*

8-puzzle problem: disponemos de un tablero 3×3 con ocho piezas numeradas y un espacio en blanco. Queremos llegar a una determinada configuración objetivo.



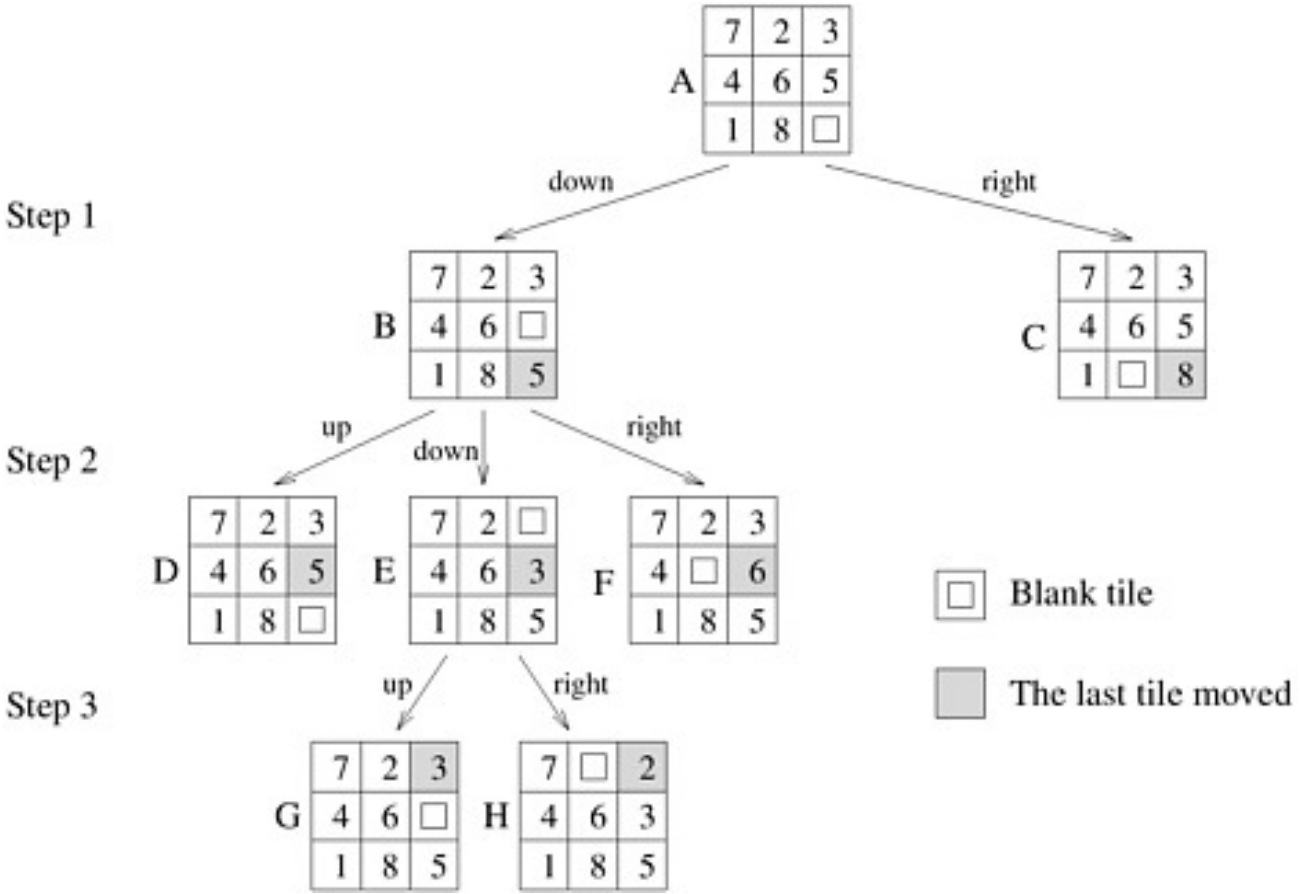
- Estados: La descripción de un estado especifica la localización de cada una de las ocho piezas numeradas y el espacio en blanco en el tablero.
- Estado inicial: cualquier estado puede ser designado como inicial.
- Acciones: disponemos de cuatro posibles acciones: `left`, `right`, `up` o `down`.

Formulación de problemas: *problemas académicos*

- Modelo de transición: todas las acciones tienen sus efectos esperados. Ej: si aplicamos la acción `right`, es decir, desplazamos una ficha a la derecha, intercambiamos las posiciones del espacio en blanco y la pieza situada a su izquierda. Las acciones son aplicables si existe una pieza que pueda intercambiar su posición con el espacio en blanco en la dirección determinada.
- Test de objetivo: comprobar si un estado encaja con la configuración objetivo.
- Coste del camino: cada paso cuesta 1, por lo tanto el coste del camino es el número de pasos en el camino.

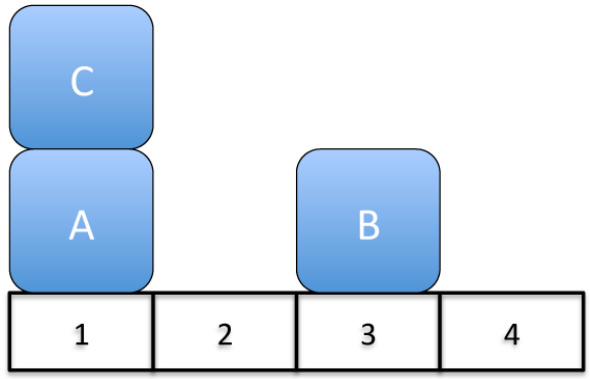
Formulación de problemas: *problemas de juguete*

Grafo del espacio de estados del *8-puzzle problem*:



Formulación de problemas: *problemas académicos*

blocks world problem: disponemos de un robot con un brazo articulado y de tres bloques distinguibles sobre una mesa con unas posiciones:



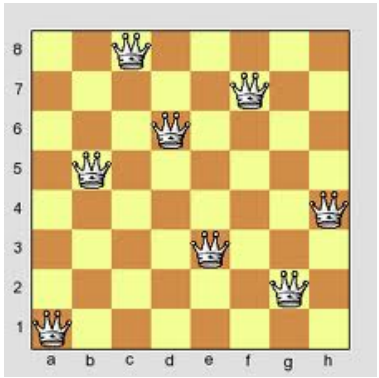
- Estados: La descripción de un estado especifica la localización de cada uno de los bloques. Los bloques pueden estar situados encima de otro bloque o de la mesa en una posición determinada.
- Estado inicial: cualquier estado puede ser designado como inicial.
- Acciones: disponemos de la acción `move(x,y)`, donde `x` es un bloque e `y` es un bloque o un posición de la mesa.

Formulación de problemas: *problemas académicos*

- Modelo de transición: el resultado de ejecutar la acción `move(x,y)` es colocar el bloque `x` encima del bloque o posición *y*. Para poder ejecutar esta acción es necesario que el bloque `x` y el bloque o posición `y` no tengan ningún bloque encima.
- Test de objetivo: comprueba si un estado encaja con la configuración objetivo.
- Coste del camino: cada paso cuesta 1, por lo tanto el coste del camino es el número de pasos en el camino.

Formulación de problemas: *problemas académicos*

8-queens problem: disponemos de un tablero de ajedrez y ocho reinas.



- Estados: cualquier disposición de 0 a 8 reinas en el tablero.
- Estado inicial: ninguna reina inicialmente en el tablero.
- Acciones: colocar una reina en el tablero.
- Modelo de transición: el resultado de la acción es el esperado. Podemos colocar una reina en una casilla vacía.
- Test de objetivo: comprueba si hay 8 reinas colocadas en el tablero sin que se ataquen. .

Formulación de problemas: *problemas académicos*

The Knuth Sequence: conjetura de Donald Knuth in 1964:

"empezando en el número 4 y aplicando una secuencia de operaciones de factorial, raíz cuadrada y parte entera, se puede llegar a cualquier entero positivo."

$$\left\lfloor \sqrt{\sqrt{\sqrt{\sqrt{\sqrt{(4!)!}}}}} \right\rfloor = 5$$

Formulación del problema:

- Estados: números positivos.
- Estado inicial: el número 4.
- Acciones: disponemos de tres acciones que corresponden a las operaciones matemáticas: `factorial`, `square-root` y `floor`.

Formulación de problemas: *problemas académicos*

- Modelo de transición: el resultado de cada acción es el número obtenido tras aplicar la operación matemática. La acción `factorial` sólo se aplica a números enteros.
- Test de objetivo: comprueba si un estado encaja con la configuración objetivo.
- Estado objetivo: el entero deseado.

No se conoce ninguna cota sobre el tamaño de los números generados. Por lo tanto, el espacio de estados es infinito. Este tipo de estados surgen frecuentemente en tareas que involucran expresiones matemáticas, circuitos, demostraciones, programas y otros objetos definidos recursivamente.

Formulación de problemas: *problemas reales*

route-finding problem. Disponemos de localizaciones y transiciones entre ellas. Los algoritmos para resolver este tipo de problema se aplican en sitios web o sistemas de abordo para proveer direcciones de conducción. También, en el routing de video streamming en redes, operaciones militares de logística, sistemas de planificación de viajes de líneas aéreas, etc. Consideremos un ejemplo simplificado del problema de un sitio web para la planificación de viajes:

Formulación de problemas: *problemas reales*

- Estados: cada estado contiene una localización y la hora actual. Puede incluir información histórica dependiendo de la función de coste.
- Estado inicial: especificado por la consulta del usuario.
- Acciones: tomar un vuelo desde la localización actual, en un determinada clase, que salga despúes de la hora actual con suficiente tiempo para transferencias en el aeropuerto si le precede otro vuelo.
- Modelo de transición: el estado resultante de tomar un vuelo tendrá como localización el destino del vuelo y como hora actual la llegada del vuelo.

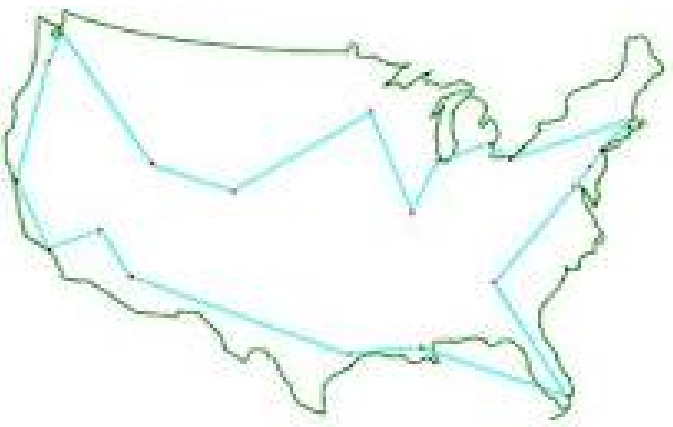
Formulación de problemas: *problemas reales*

- Test de objetivo: comprueba si se ha llegado al destino especificado por el usuario.
- Coste del camino: depende del coste del vuelo, el tiempo de espera, las horas de vuelo, la aduana y los procesos de inmigración, la calidad de los asientos, la hora del día, tipo de avión, tarjetas bonus de vuelo, etc.

Formulación de problemas: *problemas reales*

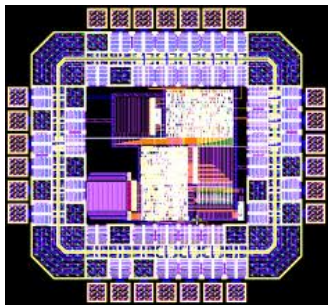
touring problems: similares a los *route-finding* pero cada estado contiene no sólo la localización actual sino el conjunto de ciudades que se han visitado previamente.

Consideremos el *Travelling Salesman Problem (TSP)*: dado un conjunto de ciudades de un territorio, el objetivo es encontrar una ruta que, comenzando y terminando en una ciudad concreta, pase una sola vez por cada una de las ciudades y minimice la distancia recorrida por el viajante.



Formulación de problemas: *problemas reales*

VLSI Layout problem: se necesitan colocar millones de componentes y conexiones sobre un chip minimizando el área, los retrasos en el circuito, las capacitancias parásitas y maximizando el rendimiento de fabricación. Se realiza tras la fase de diseño lógico y se divide en dos partes: *cell layout* y *channel routing*.



cell layout: los componentes primitivos del circuito se agrupan en celdas. Cada celda tiene una forma y tamaño fijados, y requiere un cierto número de conexiones hacia otras celdas. El objetivo es colocar las celdas en el chip de manera que no se solapen y que haya espacio para las conexiones entre celdas.

channel routing: consiste en encontrar una ruta especifica para cada conexión entre las celdas.

Formulación de problemas: *problemas reales*

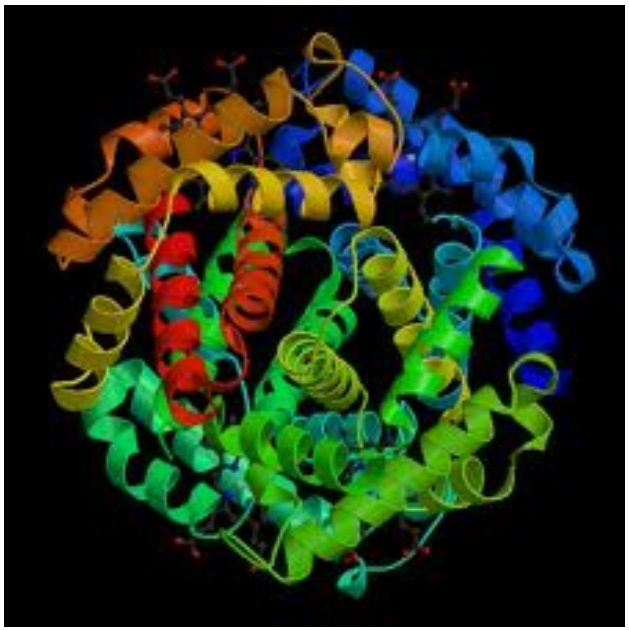
robot navigation problem: es una generalización del route-finding problem. Sin embargo, el robot puede moverse (en principio) es un espacio continuo dando lugar a un conjunto infinito de acciones y estados. Para un robot aspirador de movimiento circular, el espacio se puede considerar bidimensional aunque si dispone de brazos articulados o ruedas que deben ser controladas el espacio es multidimensional. Se utilizan técnicas avanzadas para discretizar el espacio.

Formulación de problemas: *problemas reales*

automatic assembly sequencing problem: el objetivo es encontrar el orden en que se deben ensamblar las partes de un objeto, de manera que se minimice el tiempo empleado y el coste de las operaciones. Ej: cadenas de montaje de coches, aviones, etc.

Formulación de problemas: *problemas reales*

protein design problem: el objetivo es encontrar una secuencia de aminoácidos que se plieguen en el espacio formando una proteína con las propiedades adecuadas para realizar alguna función, como curar una enfermedad.



Búsqueda

Una vez hemos formulado el problema, necesitamos resolverlo. Lo haremos buscando a través del espacio de estados.

Necesitamos una estructura que nos permita navegar (buscar) en el espacio de estados. La estructura que utilizaremos para representar el proceso de búsqueda será un grafo simple dirigido (**búsqueda en grafo (*graph search*)**).

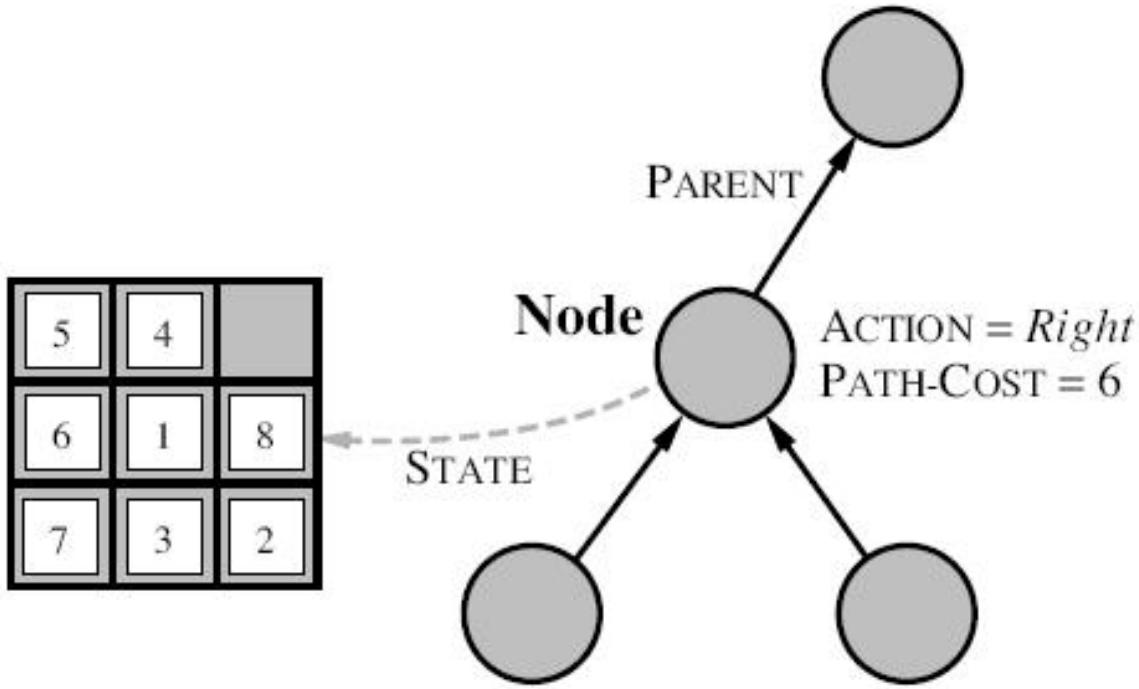
Los nodos corresponden a los estados y los arcos a las acciones.

Un posible diseño consiste en considerar un nodo n como una estructura con cuatro miembros:

- $n.state$: el estado vinculado al nodo n
- $n.parent$: el nodo a partir del que se creó n
- $n.action$: la acción que permite transitar de $n.parent.state$ a $n.state$
- $n.path\text{-}cost$: el coste del camino desde el estado inicial a $n.state$, denotado por $g(n)$.

El atributo $n.parent$ nos permite identificar los arcos del grafo.

Búsqueda



Búsqueda

Comenzaremos la búsqueda con la creación de los **nodos iniciales** que contienen un estado inicial. Denominamos **nodo objetivo** a aquel nodo que contiene un estado objetivo. Para avanzar, necesitamos escoger y expandir nodos.

Dado un problema p, un nodo n y una acción $a \in p.actions(n.state)$, la función $successor(p, n, a)$ retorna un nodo n_s tal que:

- $n_s.parent := n$
- $n_s.state := p.result(n.state, a)$
- $n_s.action := a$
- $n_s.path\text{-}cost := g(n) + p.c(n.state, a)$

Diremos que n_s es un **hijo o sucesor** de n.

Expandir un nodo consiste en crear sus sucesores.

Búsqueda

Denominamos **frontera (*fringe*)** al conjunto de nodos no expandidos (algunos autores utilizan el término *lista abierta*).

Denominamos **expandidos (*expanded*)** al conjunto de nodos expandidos (algunos autores utilizan el término *lista cerrada*).

El proceso de escoger y expandir nodos de la frontera continua hasta que, o bien se encuentra una solución, o bien no hay más nodos que expandir.

Obviamente, el proceso también finalizará si agotamos los recursos disponibles (tiempo o memoria).

Los esquemas más básicos de búsqueda utilizan un estructura de grafo más restrictiva, concretamente, un grafo simple dirigido *acíclico* o árbol (**búsqueda en árbol (*tree search*)**). Nos referimos como **raíz** del árbol a aquel nodo que contiene el estado inicial.

Búsqueda en árbol

Tree-search

input : a problem definition p
output: a solution or failure

```
1 initialize fringe to the set of nodes corresponding to p.initials()
2 while true do
3 |   if fringe.empty() then return failure
4 |   n := fringe.pop()
5 |   if p.goal(n.state) then return solution
6 |   foreach a ∈ p.actions(n.state) do
7 |   |   fringe.push(successor(p, n, a))
8 |   end
9 end
```

Búsqueda en árbol

Dependiendo de cómo extraigamos (l.4) y añadamos (l.7) nodos a *fringe* definiremos diferentes estrategías de búsqueda.

Se debe escoger la estructura de datos adecuada para implementar *fringe*, que permita extraer y añadir nodos eficientemente en función de la estrategia escogida. Podemos pensar en tres tipos de colas comunes que se caracterizan por el orden en que se almacenan los elementos:

- **Cola FIFO** (First Input First Output), habitualmente llamada cola.
- **Cola LIFO** (Last Input First Output), habitualmente llamada pila.
- **Cola con prioridad** (se extrae el elemento con mayor prioridad en base a una función de ordenación).

en función del tipo de *fringe* algunas funciones variarán su comportamiento:

- *fringe.empty*(): retorna cierto si la cola esta vacía.
- *fringe.pop*(): extrae el primer elemento.
- *fringe.push*(*n*): inserta el nodo *n*.

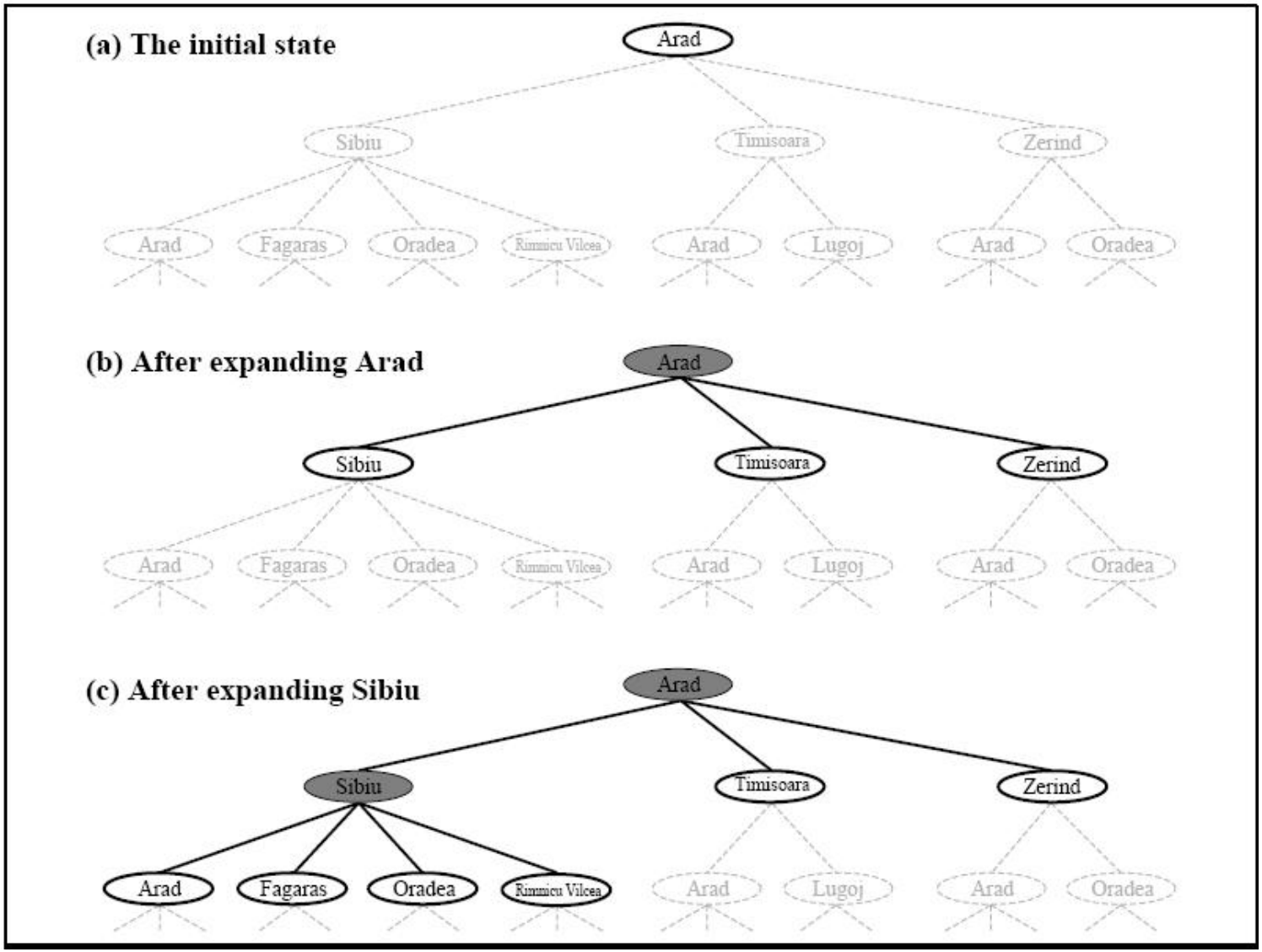
Búsqueda en árbol

La ubicación del test de objetivo (l.5) varia también la estrategia de búsqueda. Si ubicamos el test entre las líneas 6 y 7, realizándolo sobre los nodos sucesores, no variamos la eficacia de nuestro algoritmo pero sí la eficiencia. Si adoptamos esta última opción debemos comprobar si los nodos iniciales contienen un estado objetivo.

También podemos decidir no crear todos los sucesores de un determinado nodo (l.5).

Búsqueda en árbol

Búsqueda en árbol para el problema del mapa de Rumania:



Búsqueda en árbol

Como podemos comprobar en el ejemplo anterior la búsqueda en árbol nos puede llevar a la creación de nodos con el mismo estado, ej: Arad.

Esta situación puede ser debida a la repetición de un camino, por la existencia de un **ciclo**. Un caso más general es la existencia de **caminos redundantes**, que se dan cuando existe más de una manera de llegar al mismo estado, ej: Arad-Sibiu y Arad-Zerind-Oradea-Sibiu.

Los caminos redundantes pueden causar que un problema *tratable* se convierta en *intratable*. Esto es cierto incluso para los algoritmos capaces de evita ciclos infinitos.

“Quien olvida su historia está condenado a repetirla”.
Jorge Agustín Nicolás Ruiz de Santayana y Borrás

Búsqueda en árbol

Una forma de evitar los caminos redundantes es recordar dónde se ha estado. Podemos extender nuestro algoritmo de búsqueda en árbol con una nueva estructura, **expanded**, que recuerde los nodos expandidos.

De esta forma, obtenemos el algoritmo de búsqueda en grafo.

Búsqueda en grafo

Graph-search

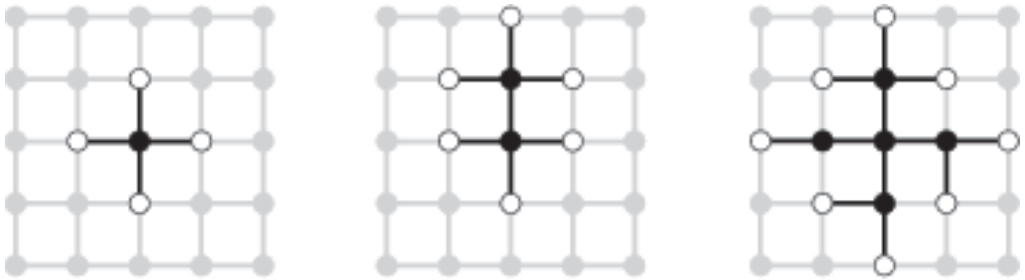
input : a problem definition p
output: a solution or failure

```
1  initialize fringe to the set of nodes corresponding to p.initials()
2  initialize expanded to be empty
3  while true do
4      if fringe.empty() then return failure
5      n := fringe.pop()
6      if p.goal(n.state) then return solution
7      expanded.push(n)
8      foreach a ∈ p.actions(n.state) do
9          n_s := successor(p, n, a)
10         if n_s ∉ fringe and n_s ∉ expanded then
11             fringe.push(n_s)
12         end
13     end
14 end
```

Búsqueda en grafo

Puesto que como mucho siempre habrá una copia de cualquier estado durante la búsqueda, podemos pensar que este algoritmo describe un árbol en el grafo del espacio de estados.

Este algoritmo, tiene una propiedad interesante. La frontera separa el grafo de espacio de estados en dos regiones: la explorada y la inexplorada, de tal forma que todo camino desde el estado inicial a un estado inexplorado debe pasar por un estado de la frontera. Ej: imaginemos un problema de *routing* sobre un mapa en forma de rejilla.



Búsqueda en grafo

Podemos utilizar estructuras de datos hash que nos permitan consultar rápidamente si un estado está repetido.

Es importante evitar repeticiones de estados a nivel lógico. Las estructuras de datos deben estar en alguna forma canónica tal que los estados lógicamente equivalentes se mapeen en la misma estructura.

Cabe resaltar que las operaciones $\notin$ de la línea 10 están sobrecargadas, y comprueban si el estado de n_s (no el nodo) aparece ya en alguno de los nodos de *fringe* o *expanded*.

Finalmente, la sentencia de la línea 7 parece que lógicamente debe ubicarse tras la 13. Es un detalle técnico que nos permite no tener que considerar *lazos* en el grafo del espacio de estados.

Medidas de calidad de la búsqueda

Comparamos los algoritmos de búsqueda en base a cuatro criterios:

- **Completitud**: ¿garantiza el algoritmo que se encuentre un solución si esta existe.? También nos puede interesar que el algoritmo informe de la inexistencia de solución, en caso de no existir.
- **Optimalidad**: ¿garantiza el algoritmo que se encuentre la solución óptima?
- **Complejidad en tiempo**: ¿cuánto tiempo invierte el algoritmo en dar una repuesta?
- **Complejidad en espacio**: ¿cuánto espacio (memoria) necesita el algoritmo?

Medidas de calidad de la búsqueda

Habitualmente la complejidad en tiempo se describe en función del número de nodos generados durante la búsqueda, y la complejidad en espacio en función de los nodos que simultáneamente se han de mantener en memoria. Utilizaremos tres medidas que nos permitirán computar dichas cantidades:

- b, el **factor de ramificación (*branching factor*)**: máximo número de sucesores de un nodo.
- d, la **profundidad (*depth*)**: longitud del camino más corto entre el nodo inicial y un nodo objetivo, y
- m, la **máxima longitud**: longitud del camino más largo.

Medidas de calidad de la búsqueda

Decimos que un algoritmo es **eficiente** cuando logra los objetivos planteados utilizando la menor cantidad de recursos posibles.

Para medir la eficiencia de nuestros algoritmos, utilizaremos el **coste de la búsqueda**, que típicamente depende de la complejidad en tiempo pero puede incluir algún término relativo al consumo de memoria.

También podemos utilizar el **coste total** que combina el coste de la búsqueda y el coste del camino a la solución.

Ej de coste total: para el problema del mapa de Rumania el coste de la búsqueda es el tiempo empleado (proporcional al número de nodos generados) y el coste del camino a la solución es la longitud del trayecto en kilómetros. Para combinar ambos costes podemos traducir los kilómetros en tiempo en función del la velocidad media del vehículo.

Estrategias de búsqueda

En las próximas secciones veremos diferentes estrategias de búsqueda. Nos centraremos, en primer lugar, en las estrategias de **búsqueda no informada o búsqueda ciega**, denominadas así porque la única información de que disponen es de la definición del problema.

Por el contrario, las estrategias de **búsqueda informada o búsqueda heurística**, que veremos más adelante, tienen cierta noción de dónde deben buscar las soluciones, gracias al uso de heurísticas.

Búsqueda no informada

Se distinguen por no tener información adicional sobre los estados salvo la proporcionada por la definición del problema.

Veremos las siguientes estrategias:

- Búsqueda en anchura prioritaria (breadth-first search)
- Búsqueda de coste uniforme (uniform-cost search)
- Búsqueda en profundidad prioritaria (depth-first search)
- Búsqueda en profundidad prioritaria limitada (depth-first limited search)
- Búsqueda en profundidad (prioritaria) iterativa (iterative deepening (depth-first) search)
- Búsqueda bidireccional (bidirectional search)

Estas estrategias se distinguen por el orden en que se visitan los nodos. Las presentaremos extendiendo la búsqueda en grafo.

Búsqueda en anchura

Breadth-First Search (BFS)

input : a problem definition p
output: a solution or failure
/* *fringe*: FIFO queue */

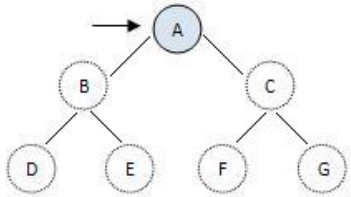
```
1  foreach s ∈ p.initials() do
2  |   if p.goal(s) then return solution(node(s, _, _, 0))
3  |   fringe.push(n(s, _, _, 0))
4  end
5  expanded := ∅
6  while true do
7  |   if fringe.empty() then return failure
8  |   n := fringe.pop()
9  |   expanded.push(n)
10 |   foreach a ∈ p.actions(n.state) do
11 |   |   n_s := successor(p, n, a)
12 |   |   if n_s ∉ fringe and n_s ∉ expanded then
13 |   |   |   if p.goal(n_s.state) then return solution(n_s)
14 |   |   |   fringe.push(n_s)
15 |   |   end
16 |   end
17 end
```

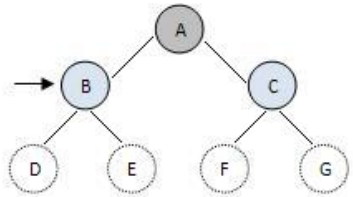
Búsqueda en anchura

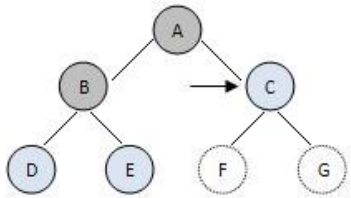
Al ser *fringe* un cola FIFO, se realiza una búsqueda en grafo expandiendo en primer lugar los nodos más superficiales (más próximos a su nodo inicial respectivo). De esta forma se describe una búsqueda en anchura o por niveles de profundidad.

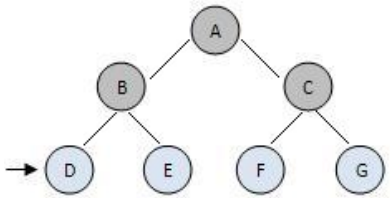
Ya hemos mencionado anteriormente la posibilidad de realizar el test de objetivo cuando se generan los sucesores, esto nos permite finalizar la búsqueda en el preciso momento en que se crea un nodo que contiene un estado objetivo.

Búsqueda en anchura









Búsqueda en anchura

Medidas de calidad del algoritmo:

- **Completitud**: si el nodo objetivo más superficial está en una profundidad finita, el algoritmo lo encontrará tras generar todos los nodos de igual o menor profundidad (siempre y cuando b sea finito).

  Si no existe solución y utilizamos un esquema de búsqueda en árbol (grafo) no (sí) es completo.

- **Optimalidad**: el algoritmo es optimal si el coste del camino es una función no decreciente sobre la profundidad del nodo.

- **Complejidad en tiempo**: el número de nodos generados hasta la profundidad d en el caso peor es:

$$1 + b + b^2 + b^3 + \ldots + b^d$$

  Por lo tanto, si la solución se encuentra a profundidad d, la complejidad es del $O(b^d)$. Si el test de objetivo se realizase tras extraer el nodo de *fringe* entonces la complejidad sería del $O(b^{d+1})$.

- **Complejidad en espacio**: el número de nodos almacenados en una búsqueda en grafo es del $O(b^{d-1})$ en *expanded* y del orden de $O(b^d)$ en *fringe*. El tamaño de *fringe* domina la complejidad por lo que la búsqueda en grafo y en árbol tienen la misma complejidad espacial, i.e., $O(b^d)$.

Búsqueda en anchura

Asumamos $b = 10$ y que se pueden generar 100000 nodos por segundo necesitando 1000 bytes de memoria.

profundidad	nodos	tiempo		memoria	
2	110	1.1	milisegundos	107	kilobytes
4	11110	111	milisegundos	10.6	megabytes
6	10^6	11	segundos	1	gigabytes
8	10^8	19	minutos	103	gigabytes
10	10^{10}	31	horas	10	terabytes
12	10^{12}	129	días	1	petabytes
14	10^{14}	35	años	99	petabytes
16	10^{16}	3.500	años	10	exabytes

Búsqueda de coste uniforme

Hemos comentando anteriormente que la búsqueda en anchura garantiza la optimalidad cuando el coste del camino es una función no decreciente sobre la profundidad del nodo. Podemos superar esta limitación si en vez de escoger el nodo el nodo más superficial, escogemos aquel con menor coste desde su estado inicial respectivo.

Hablaremos en este caso de **búsqueda de coste uniforme (uniform-cost search)**. Es un caso especial de *búsqueda el primero mejor*. El término uniforme hace referencia a que el coste del camino no decrece.

Búsqueda de coste uniforme

Uniform-Cost Search (UCS)

input : a problem definition p
output: a solution or failure
/* *fringe*: Priority queue ordered by path-cost */

```
1  foreach s ∈ p.initials() do
2  |   fringe.push(n(s, _, _, 0))
3  end
4  expanded := ∅
5  while true do
6  |   if fringe.empty() then return failure
7  |   n := fringe.pop()
8  |   if p.goal(n.state) then return solution(n)
9  |   expanded.push(n)
10 |   foreach a ∈ p.actions(n.state) do
11 |   |   n_s := successor(p, n, a)
12 |   |   if n_s ∉ fringe and n_s ∉ expanded then
13 |   |   |   fringe.push(n_s)
14 |   |   else if n_s is in fringe with higher cost then
15 |   |   |   replace that fringe node with n_s
16 |   |   end
17 |   end
18 end
```

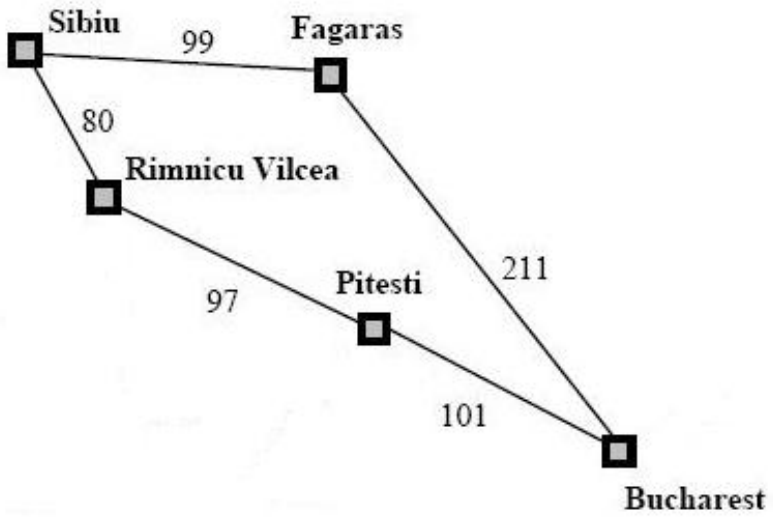
Búsqueda de coste uniforme

Como podemos observar el test de objetivo se realiza ahora en la línea 9. La nueva ubicación impide que demos por solución un camino que tenga un coste mayor a los caminos *parciales* que representan los nodos en la frontera.

También hemos añadido las líneas 15 y 16. Esto es necesario por si llegamos, por un camino más corto, a un estado que ya se encuentra asociado a un nodo de la frontera.

Es importante resaltar que no es necesario realizar la misma comprobación con el conjunto de nodos expandidos, ya que podemos garantizar que tienen asociado un camino mínimo.

Búsqueda de coste uniforme



Si empezamos nuestra búsqueda en Sibiu, siendo el objetivo llegar a Bucharest, la búsqueda en anchura nos retorna como solución $\langle go(Fagaras), go(Bucharest)\rangle$, sin embargo, la búsqueda de coste uniforme, pese a explorar con anterioridad ese camino, escoge como solución $\langle go(RimnicuVilcea), go(Pitesti), go(Bucharest)\rangle$

Búsqueda de coste uniforme

Medidas de calidad del algoritmo:

- **Completitud**: a diferencia del BFS, al UCS no le preocupan el número de pasos (acciones) aplicados, sino su coste agregado. Por lo tanto, para asegurar su completitud es necesario que el coste de un paso sea superior o igual a un cierto ϵ.
- **Optimalidad**: el algoritmo es optimal. Si el algoritmo decide expandir un nodo es porque se ha llegado a él por un camino de coste mínimo. Dado que los costes de las acciones son no negativos, los caminos nunca se acortan. Por lo tanto, el primer nodo con un estado objetivo seleccionado para expansión debe pertenecer a un camino que representa una solución óptima.
- **Complejidad en tiempo y espacio**: sea C^* el coste de una solución óptima y ϵ el coste mínimo de una acción. El máximo número de nodos en un camino óptimo es C^*/ϵ. Por lo tanto, teniendo en cuenta dónde está situado el test del objetivo, la complejidad en tiempo y espacio, es del orden de $O(b^{1+\lfloor C^*/\epsilon \rfloor})$, que es peor que b^d ya que puede ser que UCS invierta tiempo y espacio en dar pasos con costes pequeños, en vez de tomar otros mayores y útiles.

Búsqueda en profundidad

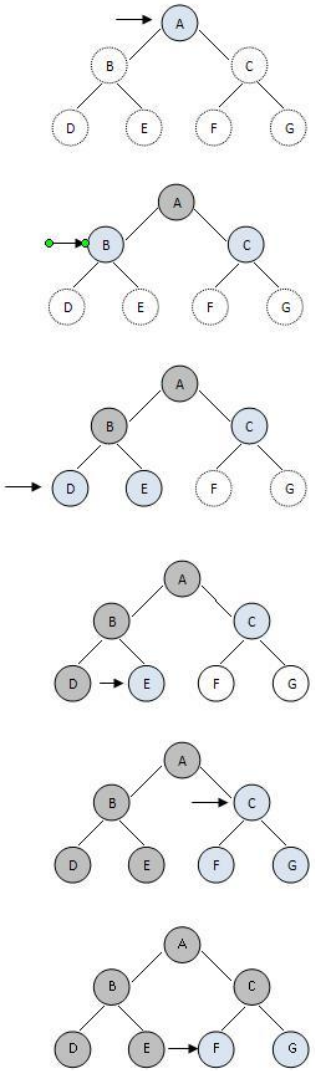
Depth-First Search (DFS)

input : a problem definition p
output: a solution or failure
/* *fringe*: LIFO queue */

```
1  foreach s ∈ p.initials() do
2  |   if p.goal(s) then return solution(node(s, _, _, 0))
3  |   fringe.push(n(s, _, _, 0))
4  end
5  expanded := ∅
6  while true do
7  |   if fringe.empty() then return failure
8  |   n := fringe.pop()
9  |   expanded.push(n)
10 |   foreach a ∈ p.actions(n.state) do
11 |   |   n_s := successor(p, n, a)
12 |   |   if n_s ∉ fringe and n_s ∉ expanded then
13 |   |   |   if p.goal(n_s.state) then return solution(n_s)
14 |   |   |   fringe.push(n_s)
15 |   |   end
16 |   end
17 end
```

Búsqueda en profundidad

El algoritmo DFS sólo se diferencia del BFS en que *fringe* se comporta como una cola LIFO, es decir, una pila. Por lo tanto, DFS siempre expande el nodo más profundo de la frontera actual, es decir, aquel que se ha añadido más recientemente a *fringe*.



Búsqueda en profundidad

Medidas de calidad del algoritmo:

- **Completitud**: la versión de búsqueda en grafo es completa en espacios de estados finitos ya que eventualmente expanderá todos los nodos. Sin embargo, la versión de búsqueda en árbol no es completa.

  Se puede modificar el DFS para que compruebe la existencia de un ciclo en el camino actual, de esta forma evitamos los ciclos, pero no los caminos redundantes.

  Si el espacio de estados es infinito, ambas versiones (árbol y grafo) fallarán si entramos un camino infinito que no lleva al objetivo.

- **Optimalidad**: no es optimal, en caso de encontrar solución no es necesariamente la solución óptima.
- **Complejidad en tiempo**: La complejidad en tiempo de la versión de búsqueda en grafo está acotada por el tamaño del espacio de estados. Sin embargo, DFS puede generar del $O(b^m)$ nodos si la búsqueda es en árbol. Recordemos que m puede ser mucho más grande que d.

Búsqueda en profundidad

Medidas de calidad del algoritmo:

- **Complejidad en espacio**: la ventaja del DFS se da en la complejidad en espacio cuando la búsqueda es en árbol. Sólo necesita almacenar un único camino desde el nodo inicial al más profundo, más los hermanos de los nodos en el camino. Un nodo expandido puede ser eliminado cuando sus hijos han sido explorados. Por lo tanto, DFS con búsqueda en árbol sólo requiere del $O(b \cdot m)$ nodos.

  Una variante del DFS es la **búsqueda con retroceso (backtracking search)** que utiliza aún menos memoria, del $O(m)$. La idea es tener nodos parcialmente expandidos que recuerdan cuál es el siguiente sucesor a generar.

  Si utilizamos el ejemplo del BFS y asumimos que los nodos a la misma profundidad que el objetivo no tienen sucesores, DFS requiere 156 kilobytes en vez de 10 exabytes a profundidad $d = 16$.

  Otra forma de ahorrar memoria, que incorpora la búsqueda con retroceso, es generar un sucesor modificando la descripción del estado actual, no realizando una copia. De esta forma tan sólo tenemos una descripción de estado y $O(m)$ acciones. Es necesario poder deshacer las modificaciones cuando retrocedemos. En entornos con descripciones de gran envergadura, como el ensamblaje robótico, es esencial.

Búsqueda en profundidad limitada

Para mitigar la inoperancia del DFS en espacios de estados infinitos podemos limitar la búsqueda a profundidad k. De tal forma que, los nodos a profundidad k se tratan como si no tuviesen sucesores. Nos referiremos al nuevo esquema de búsqueda como **búsqueda en profundidad limitada (depth-limited search)**.

Introducir el límite k es otra fuente de no optimalidad. Sin embargo, en algunos problemas podemos preveer la máxima profundidad a la que se encontrará una solución, ej: computando el diámetro del grafo del espacio de estados, si esto es posible.

DFS puede ser visto como un caso especial de búsqueda en profundidad limitada donde $k = \infty$.

Búsqueda en profundidad limitada

Depth-Limited Search (DLS)

input : a problem definition p, a limit k
output: a solution, failure or cutoff
/* *fringe*: LIFO queue */

```
1  foreach s ∈ p.initials() do
2      if p.goal(s) then return solution(node(s, _, _, 0, 0))
3      fringe.push(n(s, _, _, 0, 0))
4  end
5  expanded := ∅
6  cut := false
7  while true do
8      if fringe.empty() then return cut? cutoff : failure
9      n := fringe.pop()
10     if n.depth = k then cut := true
11     else
12         expanded.push(n)
13         foreach a ∈ p.actions(n.state) do
14             n_s := successor(p, n, a)
15             if n_s ∉ fringe and n_s ∉ expanded then
16                 if p.goal(n_s.state) then return solution(n_s)
17                 fringe.push(n_s)
18             end
19         end
20     end
21 end
```

Búsqueda en profundidad limitada

Hemos extendido la estructura de datos que representa un nodo con un nuevo atributo *n.depth*, que indica la profundidad del nodo *n*, i.e., la longitud del camino desde el nodo inicial. Por lo tanto, debemos modificar la función sucesor, de tal forma que $n_s.depth = n.depth + 1$. Dependiendo del problema es posible utilizar directamente el atributo *n.path-cost*.

El uso de la variable Booleana *cut* nos permite conocer si la profundidad limite *k* ha impedido que algún nodo se expanda. Si la frontera está vacía y *cut* es falsa podemos concluir que hemos explorado todo el espacio de estados y que el problema no tiene solución.

Búsqueda en profundidad iterativa

Iterative Deepening Search (IDS)

input : a problem definition p
output: a solution or failure

```
1 for depth = 0 to ∞ do
2 |   result := DLS(p, depth)
3 |   if result ≠cutoff then return result
4 end
```

La **búsqueda en profundidad iterativa (iterative deepening search)** nos permite encontrar el límite idóneo. En esencia, se combinan las buenas propiedades del DFS y el BFS.

Búsqueda en profundidad iterativa

Medidas de calidad del algoritmo:

- **Completitud y optimalidad**: como el BFS, es completo cuando el factor de ramificación b es finito y optimal cuando el coste de un camino es una función no decreciente sobre la profundidad del nodo.
- **Complejidad en tiempo y espacio**: como el DFS, su complejidad en espacio es del $O(b \cdot m)$, ya que en cada iteración se aplica un DLS.

  Podemos pensar que este algoritmo realiza un gran trabajo adicional por repetir en cada iteración los nodos a un determinada profundidad. Sin embargo la mayoría de los nodos corresponden a la última profundidad d y sólo se generan una vez. Por lo tanto el número total de nodos generados en el peor caso es (asumiendo test de solución al expandir):

$$N(IDS) = (d+1) + (d) \cdot b + (d-1) \cdot b^2 + \cdots + (1)b^d$$

  es decir, la complejidad en tiempo es del $O(b^d)$ como el BFS. Ej: sea $b = 10$ y $d = 5$.

$$N(IDS) = 6 + 50 + 400 + 3000 + 20000 + 100000 = 123456$$

$$N(BFS) = 1 + 10 + 100 + 1000 + 10000 + 100000 = 111111$$

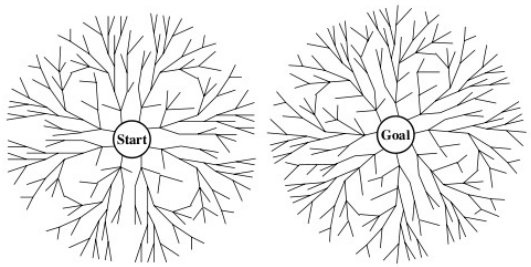
Búsqueda en profundidad iterativa

En general, IDS es el esquema de búsqueda no informada preferido cuando el espacio de estados es grande y la profundidad de la solución es desconocida.

Búsqueda bidireccional

Nuestro último esquema de búsqueda no informada, es la **búsqueda bidireccional (bidirectional search)**. La idea es iniciar dos búsquedas simultáneamente. Una hacia adelante desde el estado inicial y otra hacia atras desde el final. La motivación es que $b^{d/2} + b^{d/2}$ es mucho menor que b^d.

Implementaremos esta búsqueda reemplazando el test de objetivo con una comprobación de si las dos fronteras intersectan, que realizaremos cuando un nodo es generado o expandido.



Ej: Sea $d = 6$ y $b = 10$, y lanzamos en cada dirección un BFS. En el peor caso ambas búsquedas se encontrarán cuando hayan generado todos los nodos a profundidad 3. Para $b = 10$ esto se traduce en 2220 nodos generados en vez de 1111110 del BFS standard.

Búsqueda bidireccional

Debemos ser capaces de calcular los **predecesores** de un nodo. Pero el principal problema es que el *objetivo* puede estar declarado implícitamente. Ej: 8-queens

Medidas de calidad del algoritmo:

- **Completitud y optimalidad**: si ambas búsquedas son BFS y los costes idénticos, es completo y optimal. Otras combinaciones sacrifican completitud, optimalidad o ambas.
- **Complejidad en tiempo y espacio**: la complejidad en tiempo y espacio utilizando BFS en ambas direcciones es del $O(b^{d/2})$. Podemos reducirla a la mitad utilizando IDS, pero manteniendo una de las fronteras para poder aplicar la intersección.

Comparación de búsquedas no informadas

Esta comparación tiene en cuenta las versiones de búsqueda en árbol.

Criterio	BFS	UCS	DFS	DLS	IDS	BDS
Completo	Sí[1]	Sí[1,2]	No	No	Sí[1]	Sí[1,4]
Tiempo	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^k)$	$O(b^d)$	$O(b^{d/2})$
Espacio	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b \cdot m)$	$O(b \cdot k)$	$O(b \cdot d)$	$O(b^{d/2})$
Óptimo	Sí[3]	Sí	No	No	Sí[3]	Sí[3,4]

[1] si b es finito

[2] completo si el coste de un paso es $\geq \epsilon$

[3] óptimo si los costes de un paso son idénticos

[4] ambas direcciones usan BFS

Para las versiones de búsqueda en grafo, el DFS es completo si el espacio de estados es finito, y las complejidades de espacio y tiempo están acotadas por el tamaño del espacio de estados.

La pregunta de completitud responde únicamente a si el algoritmo es capaz de encontrar una solución en caso de que ésta exista.

Búsqueda informada

Ahora nos ocuparemos de las estrategias de búsqueda informada o búsqueda heurística. El algoritmo general que seguiremos se llama **búsqueda "primero el mejor" (best-first search)**.

Este algoritmo es una instancia de la búsqueda en árbol o en grafo, donde el nodo a seleccionar se escoge en base a una función de coste $f(n)$. De tal manera, que aquel nodo n con menor $f(n)$ es el escogido para ser expandido.

De hecho, estamos hablando del UCS pero donde $g(n)$ se sustituye por $f(n)$.

Muchos algoritmos *best-first* incluyen como componente de $f(n)$, una función heurística, $h(n)$.

$h(n)$ **= estimación del coste mínimo desde** n **a un nodo objetivo**

A diferencia de $g(n)$, $h(n)$ sólo depende del estado de n.

Ej: mapa de Rumania, h(n) = distancia en línea recta desde la ciudad n a Bucarest.

Búsqueda informada

Por ahora, consideraremos que las heurísticas son arbitrarias, no negativas y funciones específicas del problema, pero si n es un nodo objetivo, entonces $h(n) = 0$.

Dependiendo de la $f(n)$ tenemos:

$f(n) = g(n)$	Búsqueda de coste uniforme (uniform cost search)
$f(n) = h(n)$	Búsqueda "primero el mejor" voraz (greedy best-first search)
$f(n) = g(n) + h(n)$	Búsqueda A* (A* search)

$f(n)$: estimación del coste mínimo de una solución a través de n

$g(n)$: coste de llegar a n desde el nodo inicial

$h(n)$: estimación del coste mínimo desde n a un nodo objetivo

Búsqueda "primero el mejor" voraz

Volvamos al mapa de Rumania

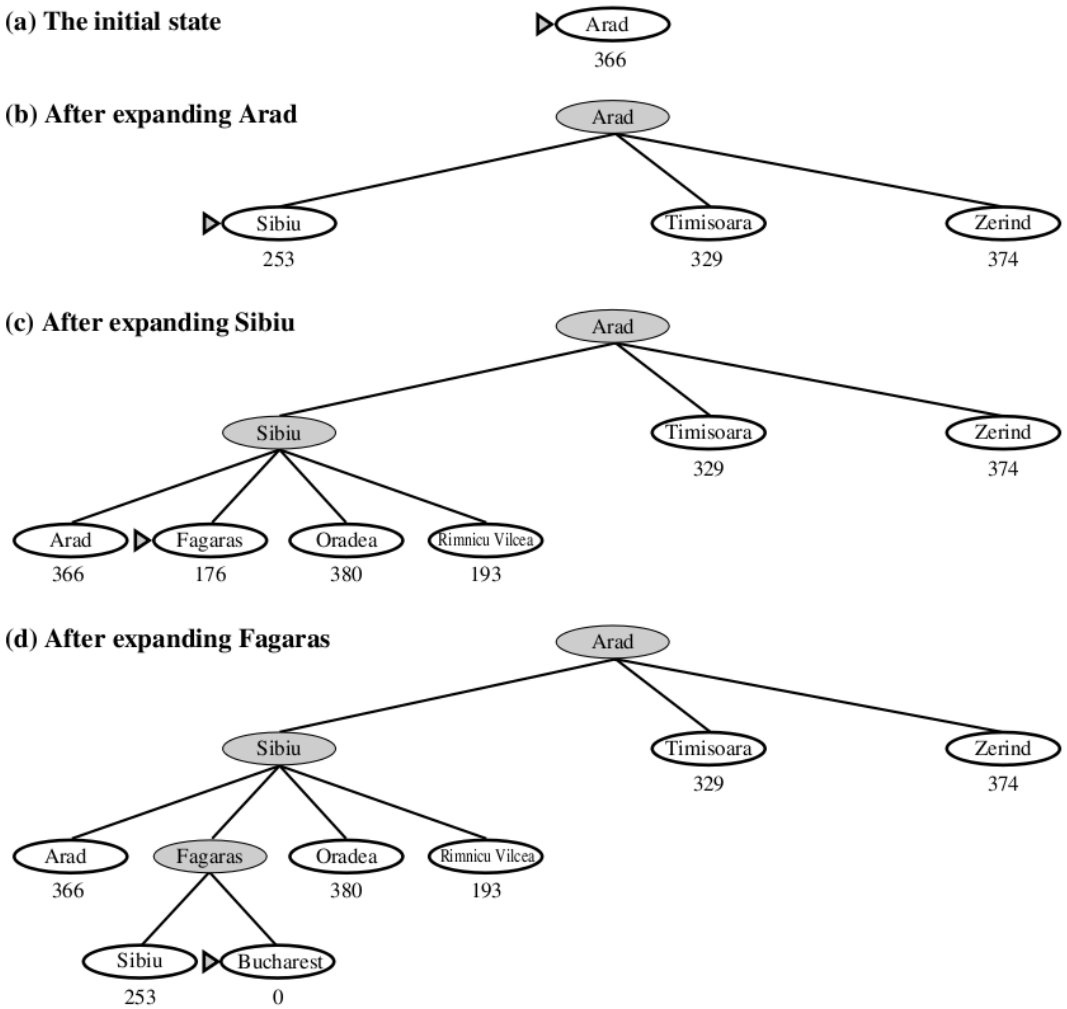
Utilizamos como $h(n)$ la distancia en línea recta a Bucarest.

Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Aplicamos greedy best-first tree search.

La solución no es optimal.

El algoritmo es voraz porque trata de acercarse en cada paso tanto como puede al objetivo.



Búsqueda "primero el mejor" voraz

Medidas de calidad:

- **Completitud**: La búsqueda en árbol es incompleta. ej: considerar el problema de ir de Iasi a Fagaras. La heurística sugiere ir a Neamt, pero es una vía muerta.
- **Optimalidad**: No es óptimo. Ver ejemplo página anterior.
- **Complejidad en tiempo y en espacio**: La complejidad tanto en tiempo como en espacio es del $O(b^m)$. Sin embargo, con una buena heurística, se puede reducir significativamente.

Búsqueda A*

El algoritmo A* es idéntico al UCS pero utiliza como función de coste, $f(n) = g(n) + h(n)$.

Veremos que si $h(n)$ cumple ciertas condiciones entonces A^* es completo y optimal.

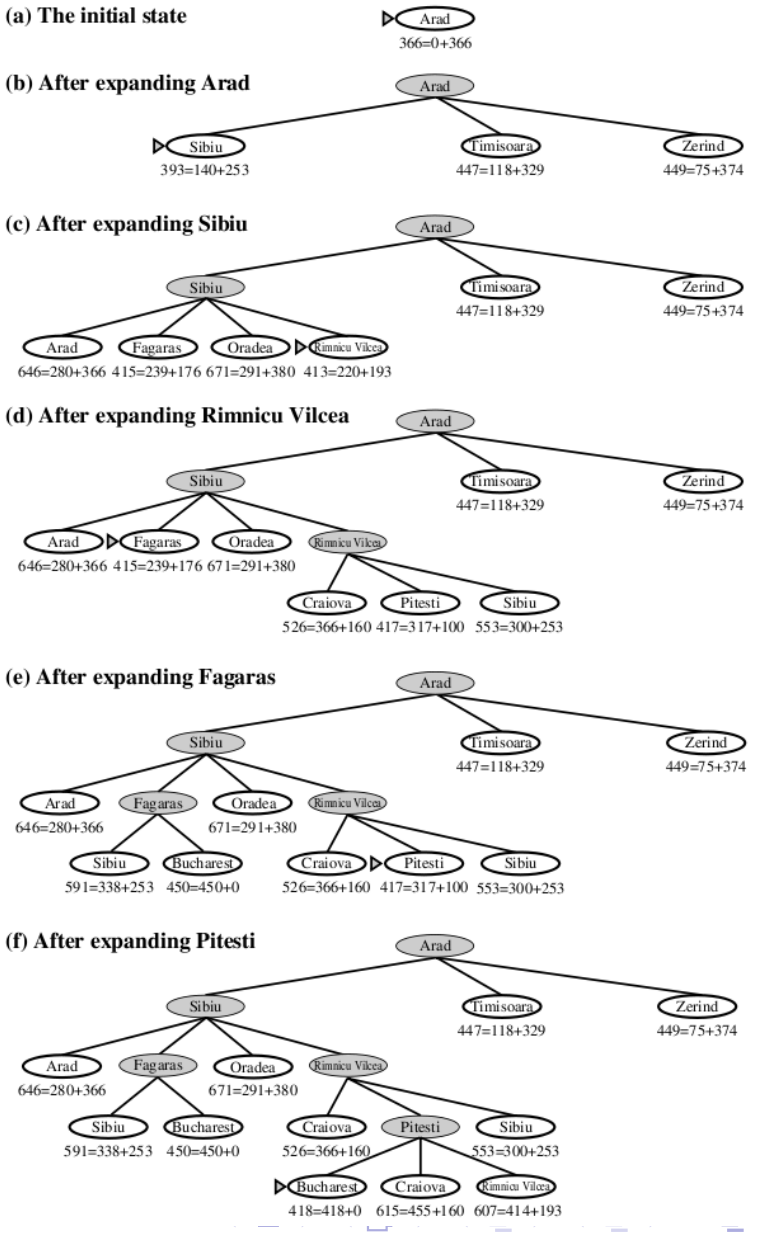
Si A^* utiliza una búsqueda en árbol, será óptimo si $h(n)$ es **admisible**.

Si A^* utiliza una búsqueda en grafo, será óptimo si $h(n)$ es **consistente**.

Aplicamos A* tree search.

$h(n)$ es la distancia en línea recta.

La solución es optimal.



Heurísticas admisibles

Una heurística es **admisible** si nunca sobreestima el coste de alcanzar el nodo objetivo.

Sea $h^*(n)$ el coste mínimo desde n hasta una solución.

Diremos que $h(n)$ es admisible ssi $\forall n\ h(n) \leq h^*(n)$.

Si $f(n) = g(n) + h(n)$, y $h(n)$ es admisible, $f(n)$ nunca sobreestima el coste de alcanzar la solución a través de n.

Estas heurísticas son optimistas porque pueden decir que el coste es menor que el real. Ej: distancia en línea recta en el mapa de Rumania.

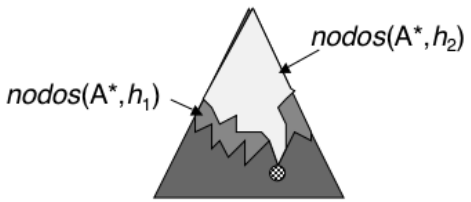
Heurísticas más informadas

$h_2(n)$ es más informada que $h_1(n)$ ssi $\forall n\ h_2(n) > h_1(n)$.

Sea $nodos(A^*, h(n))$, los nodos expandidos por A* con la heurística $h(n)$.
Si $h_1(n)$ y $h_2(n)$ son admisibles, y
$h_2(n)$ es más informada que $h_1(n)$ entonces,

$$nodos(A^*, h_2(n)) \subset nodos(A^*, h_1(n))$$

Consideremos un nodo expandido por $(A^*, h_2(n))$. Sea $f^*(s)$ el coste de la solución óptima. $f^*(s) \geq f_2(n) = g(n) + h_2(n) > g(n) + h_1(n) = f_1(n)$. A^* expande todos los nodos n t.q. $f(n) < f^*(s)$. Por lo tanto, $(A^*, h_1(n))$ expandirá n.



Heurísticas monótonas y consistentes

Una heurística es **monótona** ssi, por cada nodo n y cada sucesor n' de n, el coste estimado de alcanzar el objetivo desde n no es mayor que el coste de paso a n' más el coste estimado de alcanzar el objetivo desde n', i.e., $h(n) \leq c(n, n') + h(n')$.

Es una desigualdad triangular donde se estipula que cada lado del triángulo no puede ser mayor que la suma de los restantes. El triángulo lo forman los nodos n, n' y el nodo objetivo más cercano a n.

Una heurística es **consistente** ssi por cada par de nodos (n, n'), $h(n) \leq k(n, n') + h(n')$, donde $k(n, n')$ es el coste mínimo entre n y n'

Heurísticas monótonas y consistentes

$h(n)$ es monótona ssi $h(n)$ es consistente.

Claramente, consistencia implica monotonía, veamos el sentido inverso. Sólo necesitamos considerar los pares (n, n') tal que exista un camino entre ambos. Sea $n_1, n_2, n_3, \ldots, n_m$, con $n = n_1$ y $n' = n_m$, el camino óptimo entre n y n'. Tenemos,

$$h(n) = h(n_1) \leq c(n_1, n_2) + h(n_2) \leq c(n_1, n_2) + c(n_2, n_3) + h(n_3) \leq c(n_1, n_2) + c(n_2, n_3) + \cdots$$

$$\cdots + c(n_{m-1}, n_m) + h(n_m) = k(n, n') + h(n_m)$$

Admisibilidad y consistencia

Toda heurística consistente es admisible.

Necesitamos demostrar que si $h(n)$ es consistente, entonces $\forall n\ h(n) \leq h^*(n)$. Consideremos que n_s es el nodo óptimo alcanzable desde n. Gracias a la consistencia, $h(n) \leq k(n, n_s) + h(n_s) = k(n, n_s) = h^*(n)$

No toda heurística admisible es monótona.

Imaginemos dos nodos intermedios de un camino tal que el nodo padre n tiene $h(n) = h^*(n)$ y n' un sucesor de n, tiene $h(n') = 0$. Respecto a estos nodos la heurítica sigue siendo admisible, sin embargo no es consistente. Podemos diseñar $h(n)$ para el resto de nodos de forma que sea admisible.

En la práctica, muchas heurísticas admisibles suelen ser consistentes.

Sin embargo, no hay que menospreciar las heurísticas inconsistentes, ya que en ciertos casos pueden ser más eficientes.

Inconsistent Heuristics. Uzi Zahavi, Ariel Felner, Jonathan Schaeffer, Nathan R. Sturtevant. AAAI 2007: 1211-1216

Admisibilidad y consistencia

Podemos convertir una heurística admisible pero no consistente, en consistente (monótona) utilizando la técnica **path-max**.

Supongamos que $h(n)$ es admisible pero no es monótona.

Redefinimos $f(n')$ como:

$$f(n') = max(f(n), g(n') + h(n')) \; \forall n' \in sucesores(n)$$

Optimalidad del A*

Demostraremos que A* es óptimo para búsqueda en grafo si $h(n)$ es consistente.

Primera observación: si $h(n)$ es consistente, entonces los valores de $f(n)$ a lo largo de cualquier camino son no decrecientes.

Supongamos que n' es un sucesor de n, entonces $g(n') = g(n) + c(n, n')$. Tenemos que,

$$f(n') = g(n') + h(n') = g(n) + c(n, n') + h(n') \geq g(n) + h(n) = f(n)$$

Segunda observación: siempre que A* selecciona un nodo n para expansión, $g(n)$ es óptimo.

Supongamos que no es así. Entonces, debería haber otro nodo en la frontera n' con un camino óptimo desde el nodo inicial hasta n. Dado que $f(n)$ es no decreciente a lo largo de cualquier camino, n' debería tener un coste $f(n')$ inferior a $f(n)$. Por lo tanto, A* hubiera expandido n' previamente!!!

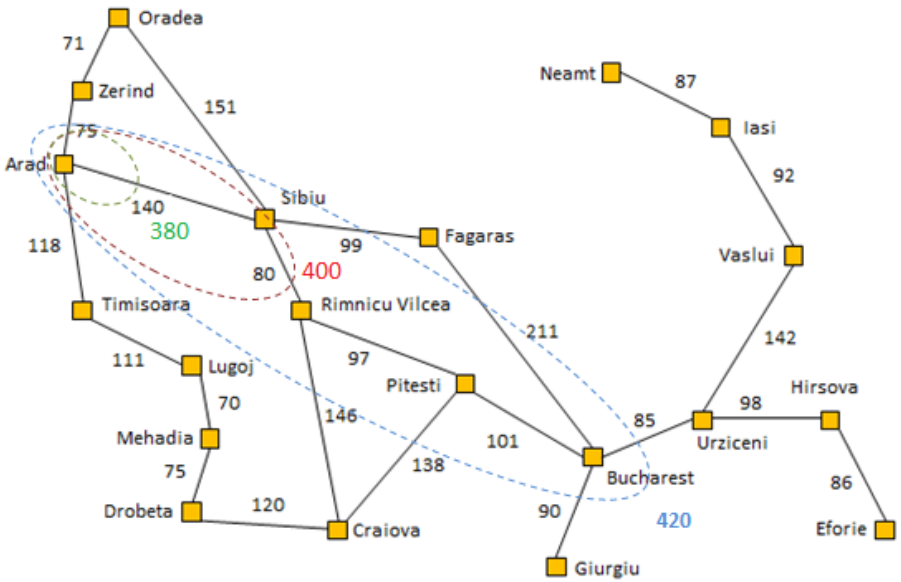
Si la búsqueda es en grafo, y $h(n)$ es admisible pero inconsistente, podemos llegar a encontrar un camino más corto a un nodo ya expandido. Para asegurar la optimalidad, trasladamos ese nodo a la frontera actualizado con el camino más corto.

Optimalidad del A*

De las dos observaciones anteriores se desprende que la secuencia de nodos expandidos por A* cuando la heurística es consistente, se realiza en orden no decreciente de $f(n)$. Por lo tanto, el primer nodo objetivo n_s expandido debe corresponder a la solución óptima, ya que $f(n_s) = g(n_s)$ y todos los nodos objetivo posteriores que pudiésemos expandir tendrían un coste igual o superior.

Contornos en A*

El hecho de que $f(n)$ sea no decreciente para heurísticas consistentes a lo largo de un camino nos permite dibujar **contornos**, como en un mapa topográfico.



Cuando $h(n) = 0$ los contornos son circulares con centro en el nodo inicial. A medida que $h(n)$ es más informada se estrechan sobre el camino óptimo hacia el objetivo.

Completitud del A*

Si C^* es el coste de un camino a la solución optimal, entonces:

A* expande todos los nodos con $f(n) < C^*$

A* puede expandir algunos nodos con $f(n) = C^*$ antes de expandir un nodo objetivo.

La completitud sólo requiere que haya un número finito de nodos con $f(n)$ menor o igual que C^*. Esto se cumple si todos los costes de paso son $\geq$ que un cierto ϵ y b es finito.

A* es optimalmente eficiente

De entre todos los algoritmos que extienden caminos desde el nodo inicial y utilizan la misma heurística, A* es optimalmente eficiente. Es decir, no existe otro algoritmo optimal que pueda garantizar una expansión menor de nodos que el A* (excepto por el desempate entre nodos con $f(n) = C^*$).

Esto se debe a que cualquier algoritmo que no expanda todos los nodos con $f(n) < C^*$ corre el riesgo de perder la solución optimal.

Complejidad en tiempo y espacio del A*

Pese a todas las buenas propiedades anteriores, en muchos problemas, el número de nodos dentro del espacio de búsqueda del contorno objetivo es exponencial en la longitud de la solución.

Por lo tanto, la complejidad temporal y espacial es exponencial.

Se pueden utilizar variantes que encuentran soluciones suboptimales rápidamente, o diseñar heurísticas más precisas pero no estrictamente admisibles.

En cualquier caso, el uso de una buena heurística puede reportar un gran ahorro en comparación con la búsqueda no informada.

Búsqueda heurística con memoria acotada

La manera más simple de reducir los requerimientos de memoria para el A* es adaptar la idea del Iterative Deepening, obteniendo el **Iterative Deepening A* (IDA*)**.

Utlizaremos como *cutoff*, la función de coste, $f(n)$, en lugar de la profundidad.

IDA* es útil para problemas con costes unitarios.

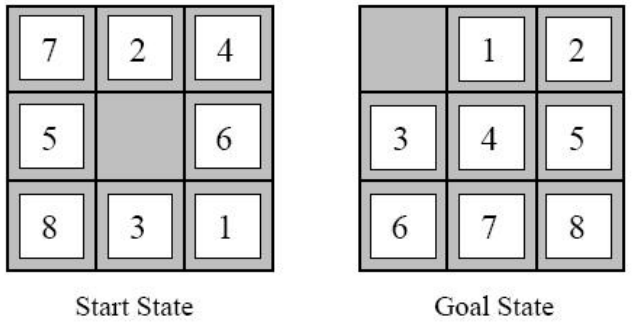
Sin embargo, no lo es cuando los costes se definen sobre los reales, mostrando los mismos problemas que el Iterative Lengthening.

Existen otras alternativas:

- Recursive Best-First Search (RBFS)
- Memory-Bounded A* (MA*)

Funciones heurísticas

Nos concentramos en el *8puzzle problem*.



El coste medio de un *8puzzle* generado aleatóriamente es de 22 pasos.

El factor de ramificación es aproximadamente 3.

Una búsqueda en árbol exhaustiva exploraría sobre los 3^{22} estados.

Una búsqueda en grafo podría suponer una reducción considerable ya que sólo hay $9!/2 = 181440$ estados.

Razonable para el *8puzzle*, pero para el *15puzzle* hablamos aún de 10^{13} ...

Necesitamos una buena heurística.

Funciones heurísticas

Necesitamos incorporar heurísticas admisibles en el A*.

Dos candidatos comunes:

$h_1 =$ "el número de piezas mal colocadas".

Es admisible, ya que al menos cada pieza mal colocada deberá moverse una vez.

$h_2 =$ "la suma de la distancias de las piezas a su posición correcta".

Nos referiremos a esta distancia como distancia *Manhattan*. Las piezas sólo pueden moverse horizontal o verticalmente.

Es admisible, ya que es el número mínimo de movimientos que una pieza ha de realizar.

Efecto de la precisión de la heurística

Una manera de caracterizar la calidad de una heurística es calculando el **factor de ramificación efectivo** b^*. Si el número total de nodos generados por A^* es N, y la profundidad de la solución es d, entonces b^* es el factor de ramificación que debe tener un árbol completo de profundidad d para contener $N+1$ nodos.

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

Ej: Si A^* encuentra una solución a profundidad 5 generando 52 nodos, entonces $b^* = 1.92$

Efecto de la precisión de la heurística

Ej: Comprobemos la eficacia de h_1 y h_2. Generamos 1200 problemas aleatorios con longitud de soluciones de 2 a 24 (100 por cada número par). Los resolvemos con IDS y A* sobre búsqueda en árbol utilizando h_1 y h_2.

	Nodos			b^*		
d	IDS	A$^*(h_1)$	A$^*(h_2)$	IDS	A$^*(h_1)$	A$^*(h_2)$
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
14		539	113		1.44	1.23
16		1301	211		1.45	1.25
18		3056	363		1.46	1.26
20		7276	676		1.47	1.27
22		18094	1219		1.48	1.28
24		39135	1641		1.48	1.26

Ambas heurísticas son admisibles, pero h_2 es más informada que h_1.

Generación de heurísticas admisibles

Nos preguntamos si es posible generar automáticamente heurísticas admisibles.

h_1 y h_2 son estimaciones del camino por recorrer en el 8puzzle, pero también son heurísticas perfectas en relajaciones de este problema.

Para h_1, imaginemos el problema donde al relajar las restricciones permitimos no ir simplemente a la casilla adyacente, sino a cualquiera.

Para h_2, relajamos el problema de forma que podemos movernos en cualquier dirección, incluso si hay otra pieza.

El grafo del espacio de estados de un problema relajado es un **supergrafo** del espacio de estados original, ya que la eliminación de restricciones provoca la adición de nuevos arcos.

Generación de heurísticas admisibles

Cualquier solución al problema original, es una solución del problema relajado.

Sin embargo, el problema relajado puede tener mejores soluciones debido a que los nuevos arcos pueden ser vistos como atajos.

Por lo tanto, el coste de una solución optimal al problema relajado, es una heurística admisible.

Dado que la heurística derivada es un coste exacto para el problema relajado, debe obedecer la *desigualdad triangular* y, por lo tanto, también es consistente.

Generación de heurísticas admisibles

Podemos automatizar la construcción de problemas relajados, si se describen en un lenguaje formal.

Ej: Consideremos la siguiente descripción formal del *8puzzle*:

Una pieza puede moverse de la celda A a la B si
A es horizontal o verticalmente adyacente a B **y** B está vacía.

Podemos generar tres problemas relajados eliminando una o ambas condiciones:

- r_1 Una pieza puede moverse de la celda A a la B si A es adyacente a B.
- r_2 Una pieza puede moverse de la celda A a la B si B esta vacía.
- r_3 Una pieza puede moverse de la celda A a la B.

r_1 corresponde a h_2.
r_3 corresponde a h_1.

Generación de heurísticas admisibles

Obviamente, nos interesa que el problema relajado se pueda resolver sin búsqueda, en caso contrario, el cómputo de la heurística sería demasiado costoso.

El programa *ABSOLVER* genera heurísticas automáticamente a partir de la relajación de problemas definidos formalmente.

ABSOLVER generó una nueva heurística para el *8puzzle* mejor que cualquiera hasta la fecha, y encontró la primera heurística útil para el cubo de Rubik.

Generación de heurísticas admisibles

Dado que es difícil encontrar la mejor heurística, si disponemos de un conjunto de heurísticas admisibles $h_1, \ldots, h_m$ y ninguna de ellas es más informada que las demás, podemos escoger,

$$h(n) = max\{h_1, \ldots, h_m\}$$

Aprendizaje de heurísticas basado en la experiencia

Podemos aprender heurísticas a partir de la experiencia en la resolución del problema.

Cada solución optimal al *8puzzle* provee ejemplos a partir de los cuales $h(n)$ puede ser aprendida.

Cada ejemplo lo componen, un estado de la solución y el coste desde éste hasta el estado objetivo siguiendo el camino de la solución.

Utilizando algoritmos de aprendizaje automático, como redes neuronales, árboles de decisión, etc., podemos construir $h(n)$ que prediga el coste para otros estados.

Ej: Asumamos que $h(n)$ se puede describir como una combinación lineal de un conjunto de características asociadas al nodo. Sea $x_1(n)$ el número de piezas mal colocadas, y $x_2(n)$ el número de pares de piezas adyacentes que no lo son en la configuración objetivo. Buscamos aprender los coeficientes c_1 y c_2 de la siguiente ecuación:

$$h(n) = c_1 x_1(n) + c_2 x_2(n)$$

Índice