



Universidad
Francisco de Vitoria
UFV Madrid

Ingeniería del Conocimiento

Tema 0:
***El Conocimiento y su
Representación***

Objetivos del tema



- Ubicación
 - Unidad: **INGENIERÍA DEL CONOCIMIENTO**
 - *Tema 0: El Conocimiento y su Representación*
- Objetivos generales
 - Entender la *definición de conocimiento* y su diferencia con información y dato
 - Comprender el concepto de “*ciclo de conocimiento*” y sus fases
 - Diferencias entre *Lenguaje Natural* y *Lenguaje Formal*
 - *Representación del conocimiento*: ¿Por qué la información/conocimiento es siempre incompleta?



1. Introducción
2. ¿Qué es Conocimiento?
3. Ciclo del Conocimiento
4. Tipos de Conocimiento
 1. Conocimiento declarativo
 2. Conocimiento procedimental
5. Formas de representación
 1. Lenguaje Natural
 2. Lenguaje Formal
 3. Problema de la representación



- 1. Introducción**
- 2. ¿Qué es Conocimiento?**
- 3. Ciclo del Conocimiento**
- 4. Tipos de Conocimiento**
 - 1. Conocimiento declarativo**
 - 2. Conocimiento procedimental**
- 5. Formas de representación**
 - 1. Lenguaje Natural**
 - 2. Lenguaje Formal**
 - 3. Problema de la representación**



1. Introducción

Resolución de problemas en Inteligencia Artificial simbólica:

- **Búsqueda Informada:**

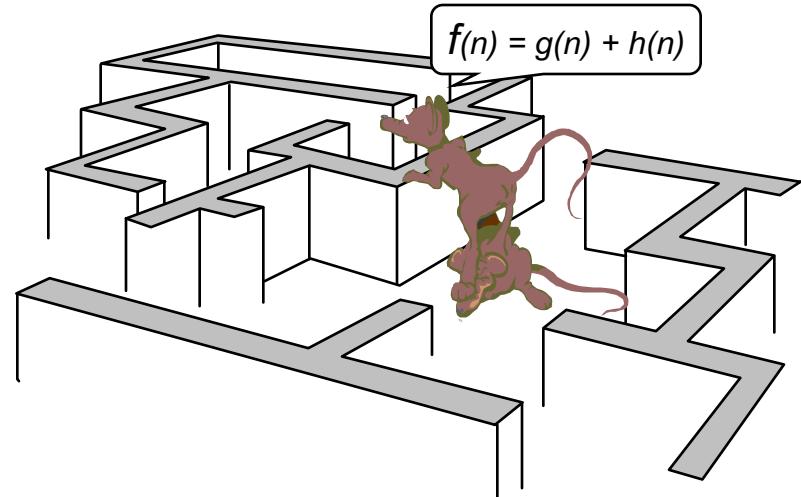
Uso de algoritmos para buscar la solución en el espacio de los posibles estados (grafo) en que se puede encontrar un problema

- Representación del problema

Usando el paradigma del Espacio de Estados

- Búsqueda de la solución

Búsqueda entre todos los estados posibles mediante una estrategia eficiente sobre el grafo/árbol que representa al problema





1. Introducción

■ *Razonamiento e Inferencia*

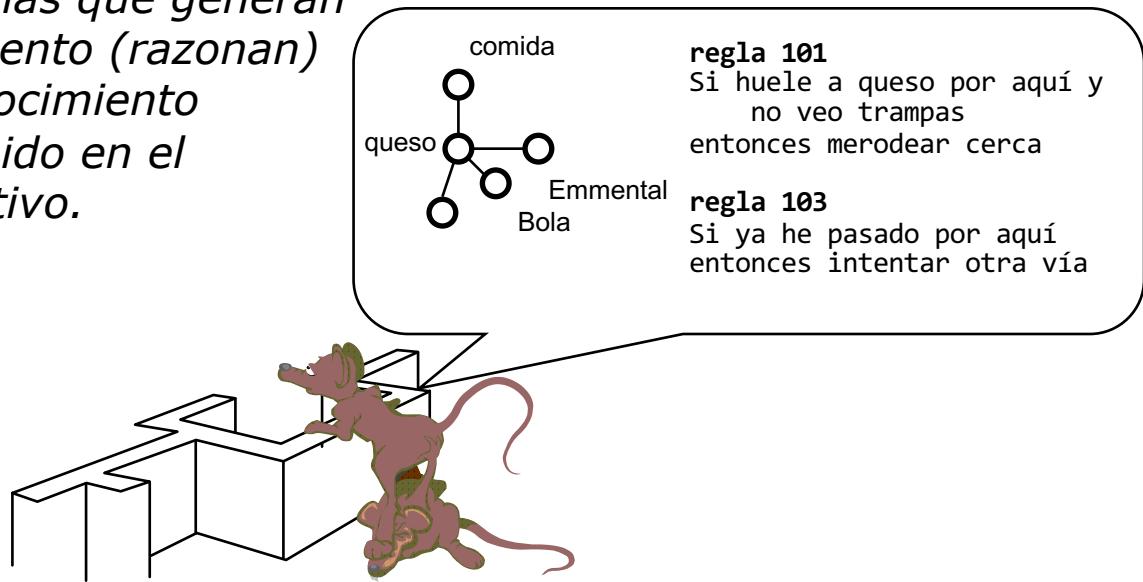
Uso de formalismos para representar la realidad en forma de conocimiento sobre el que se puedan realizar inferencias.

- Representación del problema

Usando un modelo descriptivo de las características del problema

- Obtención de la solución

Mediante sistemas que generan nuevo conocimiento (razonan) a partir del conocimiento explícito contenido en el modelo descriptivo.





1. Introducción
2. ¿Qué es Conocimiento?
3. Ciclo del Conocimiento
4. Tipos de Conocimiento
 1. Conocimiento declarativo
 2. Conocimiento procedimental
5. Formas de representación
 1. Lenguaje Natural
 2. Lenguaje Formal
 3. Problema de la representación



2. ¿Qué es Conocimiento?

- Conocimiento (del griego γνώσις, *conocer*) es
 - Comprensión teórica o práctica de un tema o dominio
 - Descripción o modelo simbólico de un dominio
 - La suma de lo que es actualmente conocido
 - ¡¡ Poder !!
- Diferencia entre DATO/INFORMACION/CONOCIMIENTO
DATOS → INFORMACION → CONOCIMIENTO
- *DATOS: letras o números que representan una cantidad, una medida, una palabra o una descripción*
 - Información pasiva, de estructura y formato simple
 - No contienen ninguna información por sí mismos.



2. ¿Qué es Conocimiento?

- **INFORMACION:** *conjunto organizado de datos relacionados en un contexto que constituyen un mensaje sobre un determinado ente o fenómeno.*
 - Resultado de organizar datos estableciendo relaciones entre ellos, lo que proporciona contexto y significado
 - *Información = Datos + Contexto (añadir valor)*
- **CONOCIMIENTO:** *Información interpretada, de estructura y formato complejo, y que modela la experiencia que se tiene sobre un dominio o campo concreto.*
 - Capacidad de relacionar la información que poseemos para resolver una determinada situación
 - Permite predecir o imaginar con un grado aceptable de certeza acerca de acontecimientos futuros
 - *Conocimiento = Datos + Contexto (añadir valor) + Utilidad*



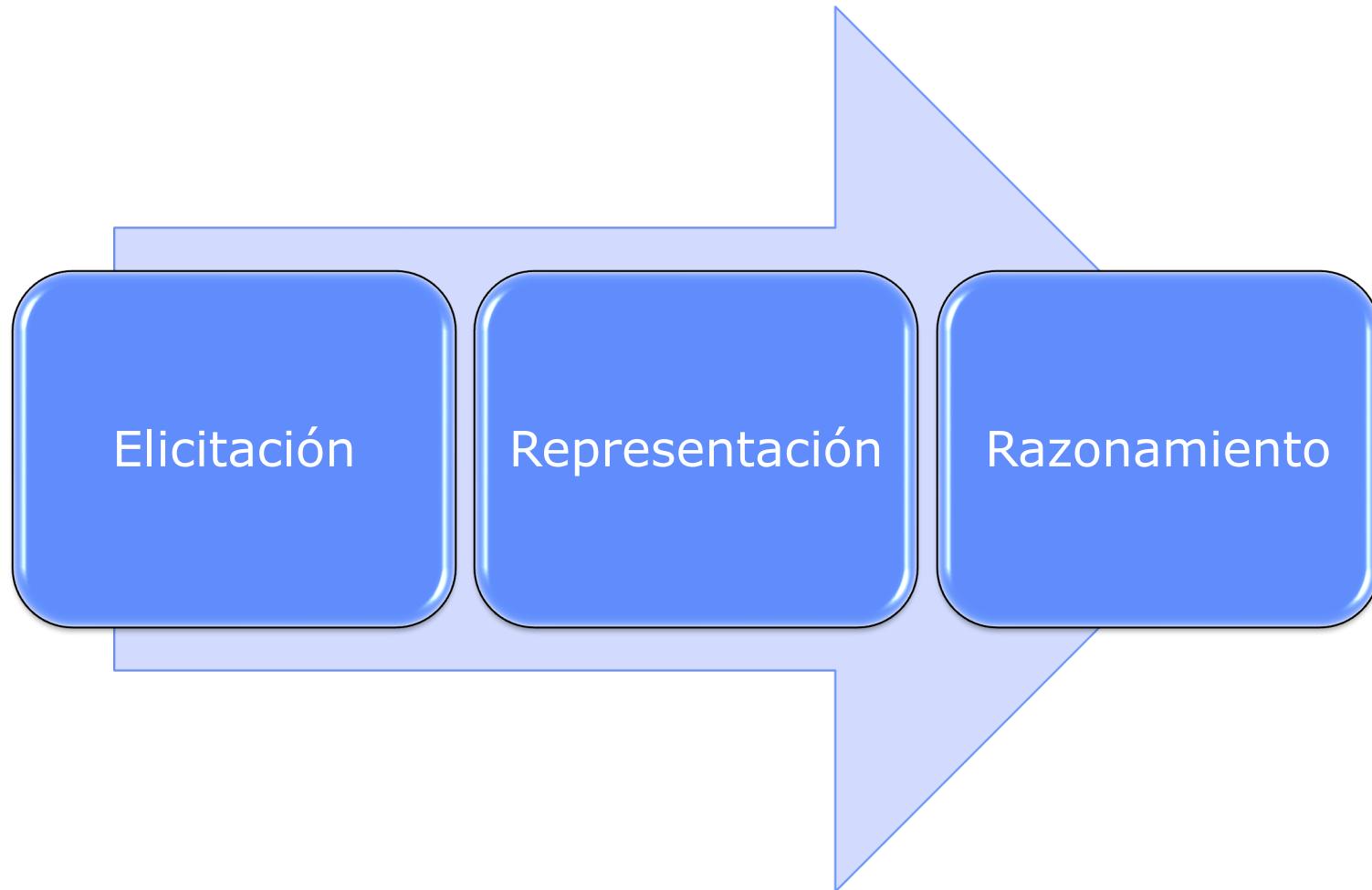
2. ¿Qué es Conocimiento?

- Los problemas “*reales*” son complicados
 - Incluyen “*aspectos humanos*” como parte del conocimiento necesario para su resolución:
 - incertidumbre, incompletitud, suposiciones, intencionalidad, información temporal, conocimiento “de sentido común”, etc.
 - La representación de este conocimiento debe de ser más amplio y flexible que una heurística
 - Las heurísticas son un tipo de representación de conocimiento demasiado simple y “escaso”
 - Las funciones heurísticas tienen en cuenta información general que evalúa el problema siempre de la misma manera.
 - Pero durante la resolución de un problema
 - No tiene por que evaluarse su estado siempre igual
 - No tiene por que tener siempre la misma importancia toda la información de la que disponemos.



1. Introducción
2. ¿Qué es Conocimiento?
- 3. Ciclo del Conocimiento**
4. Tipos de Conocimiento
 1. Conocimiento declarativo
 2. Conocimiento procedimental
5. Formas de representación
 1. Lenguaje Natural
 2. Lenguaje Formal
 3. Problema de la representación

3. Ciclo del Conocimiento



3. Ciclo del Conocimiento



Elicitación

- Para poder representar el conocimiento, hay que obtenerlo a partir de las fuentes que lo poseen (los llamados “*expertos*”)
- *Adquisición, obtención, extracción del conocimiento a partir de expertos* mediante
 - Entrevistas, cuestionarios y análisis de documentación
 - Observación (múltiples puntos de vista)
 - Uso de herramientas específicas (Grids...)
- Similar a obtener los **requisitos de usuario** en Ingeniería del SW

3. Ciclo del Conocimiento



Representación

- *Construir enunciados del mundo usando un lenguaje formal y más limitado que el lenguaje natural*
 - *Mediante el cual podamos escribir enunciados que denotan hechos del mundo*
 - *Que nos permita razonar*
- El conocimiento puede ser representado de varias maneras
 - *Icónica*
 - Se simulan aspectos estructurales del mundo
 - Es eficiente, pero específico e inflexible
 - *Descriptiva*
 - En base a características binarias cierto/falso
 - es genérico y flexible, aunque menos eficiente
- Hay que elegir un formalismo que nos permita representar de forma adecuada unos ciertos hechos

3. Ciclo del Conocimiento



Razonamiento

- *Uso del conocimiento para derivar nuevo conocimiento*
- Cada formalismo de representación usa un método de razonamiento específico:
 - Razonamiento hacia adelante/hacia atrás
 - Herencia
 - Resolución...
- Los principales métodos de razonamiento son:
 - Proyección
 - Simulación
 - Cálculo de estados futuros correspondientes al resultado de las acciones (búsqueda, etc.)
 - Inferencia
 - Inducción de información sobre el estado presente

3. Ciclo del Conocimiento



La IA resuelve problemas empleando la INFERENCIA sobre un modelo DESCRIPTIVO

- Representa un dominio de conocimiento (modelo del mundo)
- Utiliza un proceso de inferencia para derivar nuevas representaciones del mundo
- Emplea éstas para deducir qué hacer



1. Introducción
2. ¿Qué es Conocimiento?
3. Ciclo del Conocimiento
- 4. Tipos de Conocimiento**
 - 1. Conocimiento declarativo**
 - 2. Conocimiento procedimental**
5. Formas de representación
 1. Lenguaje Natural
 2. Lenguaje Formal
 3. Problema de la representación



4. Tipos de Conocimiento

- **Conocimiento Declarativo o Factual:** El conocimiento se representa de forma independiente a su uso posterior
 - Explícito o Relacional
 - Implícito: se obtiene a partir del conocimiento explícito mediante
 - Herencia
 - Inferencia
- **Conocimiento Procedimental:** El conocimiento representado implica la inclusión de información sobre como usarlo.
- **Meta-conocimiento:** Conocimiento sobre el propio conocimiento, que permite controlarlo, gestionarlo y garantizar la consistencia

Fecha_nacimiento < Fecha_actual



4.1 Conocimiento declarativo

- **Conocimiento Declarativo Explícito o Relacional:**
 - La forma más simple de representar hechos declarativos
 - Conjunto de relaciones expresables mediante tablas (como en una Base de Datos)

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

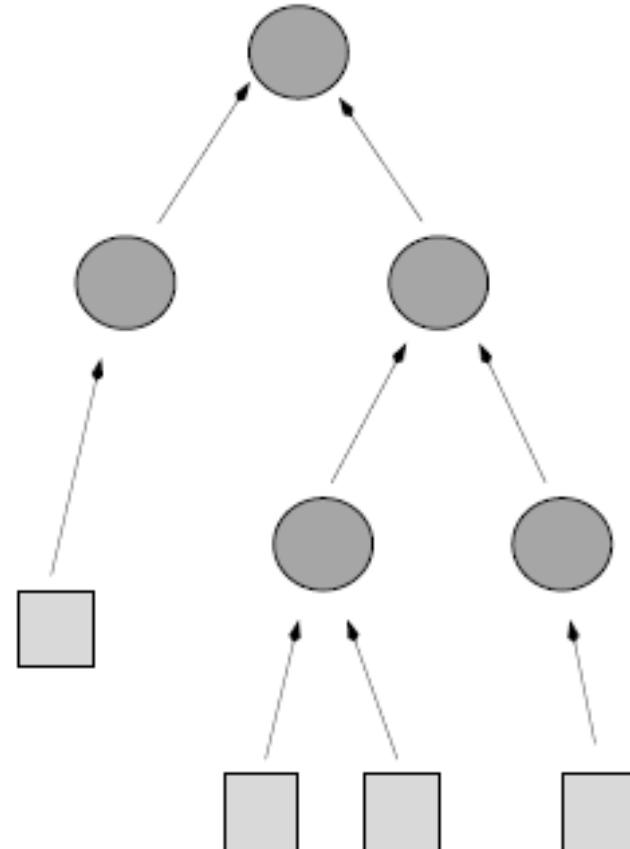
- Problema: tal cual no aporta mucha información
 - Hemos de aportar procedimientos que lo enriquezcan, como son los mecanismos de inferencia que generan conocimiento a partir de información
 - media de compras en una población, mejor cliente, tipología de clientes ...

4.1 Conocimiento declarativo



■ Conocimiento Declarativo Implícito/heredable

- Estructuración jerárquica del conocimiento (taxonomía jerárquica)
- Árbol o grafo de conceptos basado en la generalización y/o especialización
 - Los nodos son los conceptos/clases.
 - Los arcos las relaciones
 - is-a (es-un): relación clase-clase
 - Instance-of (instancia-de, ejemplar-de)
- El mecanismo de inferencia son las reglas de herencia de propiedades y valores
 - Herencia simple/múltiple
 - Valores por defecto





4.1 Conocimiento declarativo

- **Conocimiento Declarativo Implícito/inferible**
 - Conocimiento descrito mediante lógica
 - Forma general de obtener conocimiento implícito en un formalismo de representación
 - A partir de las reglas de inferencia (modus ponens, resolución, etc.)

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

4.2 Conocimiento procedural



- Incluye la especificación de los procesos de uso del conocimiento:
 - Programas: utilizan funciones para obtener el conocimiento a partir de información o de otro conocimiento que ya se tiene
 - Ej: Fecha_nacimiento= DD-MM-AAAA; función Edad (Fecha_nacimiento:entero)
 - Reglas de producción: si se cumplen unas condiciones entonces se realizan unas acciones u otras.
 - Ej: SI condición ENTONCES acción
- Este tipo de conocimiento suele ser más eficiente computacionalmente, pero hace más difícil la inferencia y la adquisición/modificación.



1. Introducción
2. ¿Qué es Conocimiento?
3. Ciclo del Conocimiento
4. Tipos de Conocimiento
 1. Conocimiento declarativo
 2. Conocimiento procedimental
5. Formas de representación
 1. Lenguaje Natural
 2. Lenguaje Formal
 3. Problema de la representación

5.1 Lenguaje Natural



- Se articula en las **dos dimensiones**:

- **Sintáctica:**

- relaciones entre los signos (palabras) para construir unidades con sentido completo, es decir, las oraciones
 - "*Cómo se escribe*"
 - Podemos utilizar la estructura de nuestro lenguaje con sus mecanismos sintácticos, para obtener conclusiones válidas sobre nuestro entorno.

- **Semántica:**

- las relaciones de los signos con las cosas significadas.
 - "*Qué significa*"
 - Podemos establecer relaciones entre los símbolos de nuestro lenguaje y los referentes de su entorno para establecer el valor de verdad de sus enunciados.



5.1 Lenguaje Natural

- Con el lenguaje construimos **oraciones**, en las que las palabras significan las cosas mediante los conceptos mentales que nos construimos de nuestro mundo (*conocimiento*).
- El conocimiento se recoge/construye durante toda la vida de un ser humano y se almacena en el cerebro
- Pensamos y actuamos de acuerdo a este conocimiento pero a menudo somos incapaces de expresarlo de una forma entendible al 100% mediante el uso del *Lenguaje Natural*

5.1 Lenguaje Natural



- Los humanos empleamos el **lenguaje natural** con diferentes funcionalidades
 - Comunicar
 - Declarar
 - *Razonar*
 - ...
- El *lenguaje natural* usado para el intercambio de conocimiento es bastante ineficaz.
 - El lenguaje natural es muy rico,
 - Pero es **ambiguo**,
 - Mismo significado, distinta representación
 - Distinto significado, misma representación
 - Causas de ambigüedad
 - por la **interpretación emisor-receptor**
 - por el **propio mensaje**.



5.1 Lenguaje Natural

Entre lo que pienso,
Lo que quiero decir,
Lo que creo decir,
Lo que digo,
Lo que quieras oír,
Lo que oyes,
Lo que crees entender,
Lo que quieras entender,
Lo que entiendes,
Existen nueve posibilidades
de no entenderse.

5.1 Lenguaje Natural



- Experimentos (Anderson, Wanner, Sachs 1967)
 - Los sujetos recuerdan las palabras un corto intervalo (segundos a decenas de segundos), pero acababan por olvidarlas y recordar sólo el significado.
 - Sugiere que la gente procesa las palabras para formar alguna representación no verbal que se almacena como recuerdo.
- Distintos lenguajes dividen el mundo de distinta forma:

Spanish has two words for "fish," one for the live animal and one for the food. English does not make this distinction, but it does have the cow/beef distinction.

El inglés tiene dos palabras para la vaca, una para el animal vivo y otra cuando es comida. En cambio, no tiene la distinción entre pez y pescado.

 - Sin embargo no hay evidencia de que los angloparlantes y los hispanohablantes piensen sobre el mundo de una manera fundamentalmente distinta.



5.2 Lenguaje Formal

- Se necesita una esquematización del lenguaje:
 - Debe ser formal
 - Acotado
 - Más limitado que el lenguaje natural
 - Mediante el cual podamos escribir enunciados (simples o compuestos) con que denotaremos hechos (proposiciones) del mundo
 - Que nos permita realizar inferencias y razonar
 - *Lenguaje formal es un lenguaje cuyos símbolos primitivos y reglas para unir esos símbolos están formalmente especificados*
- ↓
- lenguajes formales**

5.2 Lenguaje Formal



- Los Lenguajes Formales se definen mediante
 - Sintaxis
 - especifica la **estructura** de las sentencias
 - ¿que expresiones son correctas?
 - Semántica
 - define el **valor de verdad** de cada sentencia (**valor** que indica en qué medida una declaración es **verdad**)
 - ¿que significan las expresiones? (Las expresiones son ciertas o falsas en cada “mundo posible” o modelo)
 - Sistema de inferencia/Axiomas
 - Reglas para manipular las expresiones y poder derivar una cosa a partir de otra



5.2 Lenguaje Formal

- Un ejemplo de lenguaje formal: **la Aritmética**
 - Ejemplos de “frases” aritméticas:
 - $x+2 \geq y$ es una frase
 - $x2+y > \{\}$ no es una frase
- Otro ejemplo: **!! LA LÓGICA !!**
- Además de la lógica, existen otras descripciones (no lenguajes) formales del conocimiento.
 - Reglas
 - La representación en forma de reglas es intuitiva para muchas personas
 - Redes semánticas
 - Marcos
 - Modelos no simbólicos
 - Combinaciones de los anteriores



5.2 Lenguaje Formal

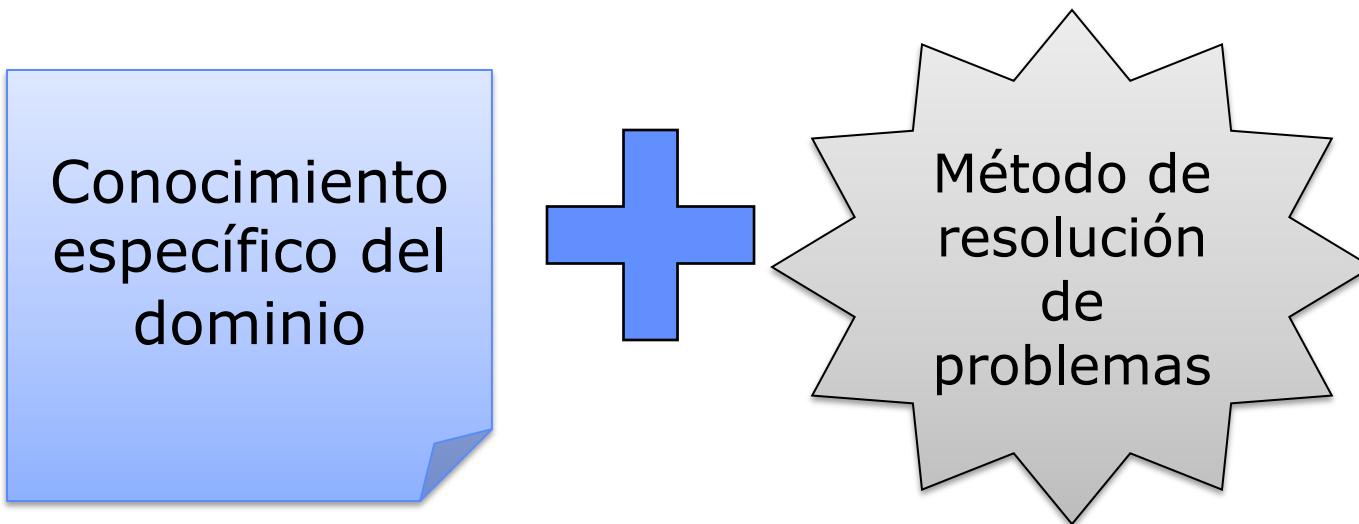
■ **Métodos de Representación del Conocimiento usados en IA**

- De acuerdo al **formalismo** utilizado
 - Conocimiento declarativo
 - Lógicas (proposicional, de predicados, temporal, difusa...)
 - Reglas (*rules*)
 - Conocimiento estructurado
 - Redes Semánticas
 - Marcos (*frames*)
- De acuerdo al **tipo de representación**
 - Representación Basada en Relaciones
 - Lógicas
 - Redes Semánticas
 - Representación Basada en Objetos
 - Marcos
 - Representación Basada en Acciones
 - Reglas



5.3 Problema de la representación

- El conocimiento dependiente del dominio se combina con el conocimiento general sobre cómo resolver problemas.



- Problemas:
 - ¿Cómo escoger el formalismo de representación?
 - ¿Cómo ha de ser esa representación para que pueda ser utilizada de forma eficiente?



5.3 Problema de la representación

- Para representar algo necesitamos saber:
 1. Su forma o estructura
 2. Qué uso le dan los seres inteligentes
 3. Qué uso le dará una inteligencia artificial
 4. Como adquirir el conocimiento
 5. Como almacenarlo y manipularlo
- Por desgracia *no hay respuestas completas para todas estas preguntas desde el punto de vista biológico o neurofisiológico.*
- Los procesos mentales humanos son internos y demasiado complejos para ser representados por un algoritmo.
- Se construyen modelos que intentan *simular* la elicitation, representación y manipulación del conocimiento (IA).

5.3 Problema de la representación



Problemas al representar del conocimiento

■ Representación incompleta:

- Falta conocimiento. Suele ser imposible representar todo el conocimiento debido a:
 - **Modificaciones:** el mundo es cambiante, pero nuestras representaciones son de un instante determinado.
 - Relacionados con los *procedimientos de adquisición y mantenimiento* de la representación (*Frame Problem*):
 - Una representación fiel requiere poder representar todo lo que observamos en la realidad y obtener todas las consecuencias lógicas de cada cambio.
 - **Volumen:** mucho (demasiado) conocimiento a representar
 - **Complejidad:** La realidad tiene una gran riqueza en detalles.
 - Ambos relacionados con la *granularidad de la representación*.

5.3 Problema de la representación



- **Conocimiento inseguro** (incierto, dudoso):
 - “muchos chavales son rebeldes”
 - Conocimiento acompañado de grado de certeza
 - Uso de lógica difusa, cuantificadores especiales, etc.
- **Conocimiento por omisión** (by default):
 - Conocimiento que se asume implícitamente mientras no se niegue explícitamente. Requieren garantizar la consistencia (TMS)
 - Sistemas monótonos
 - Lo verdadero no puede dejar de serlo
 - Sistemas no monótonos
 - Las conclusiones establecidas en un cierto momento pueden dejar de ser ciertas si llega nueva información

5.3 Problema de la representación



Problemas al usar esquema de representación

- *Ligados a la representación*
 - Adecuación Representacional: Habilidad para representar todas las clases de conocimiento que son necesarias en el dominio.
 - Adecuación Inferencial: Habilidad para manipular estructuras de representación de tal manera que generen nuevo conocimiento inferidos del anterior.
- *Ligados al uso de la representación*
 - Eficiencia Inferencial: Capacidad del sistema para incorporar conocimiento adicional a la estructura de representación (metaconocimiento) que puede emplearse para optimizar el cómputo.
 - Eficiencia en la Adquisición: Capacidad de incorporar fácilmente nueva información.



5.3 Problema de la representación

- No existe un esquema de representación que sea óptimo en todas estas características a la vez.
- Soluciones:
 - Escoger la representación en función de la característica que necesitemos en el dominio de aplicación específico
 - Utilizar diferentes esquemas de representación a la vez.



Universidad
Francisco de Vitoria
UFV Madrid

Ingeniería del Conocimiento

Tema 1:
***Introducción al Razonamiento
Artificial***

Objetivos del tema



- Ubicación
 - Unidad 1: **ASPECTOS BASICOS DE LA IA SIMBOLICA**
 - *Tema 1: Introducción al Razonamiento Artificial*
- Objetivos generales
 - Definir la **IA simbólica** (razonamiento artificial) y establecer sus áreas y técnicas de trabajo.
 - Presentar los **modelos de Procesamiento Simbólico** que se estudiarán en el cuatrimestre
 - Comprender lo que es un **espacio de estados**, como se crea y como se usa para **buscar** la solución de problemas.
 - Entender la importancia de la **representación del conocimiento** como forma de generar nuevo conocimiento mediante mecanismos de inferencia



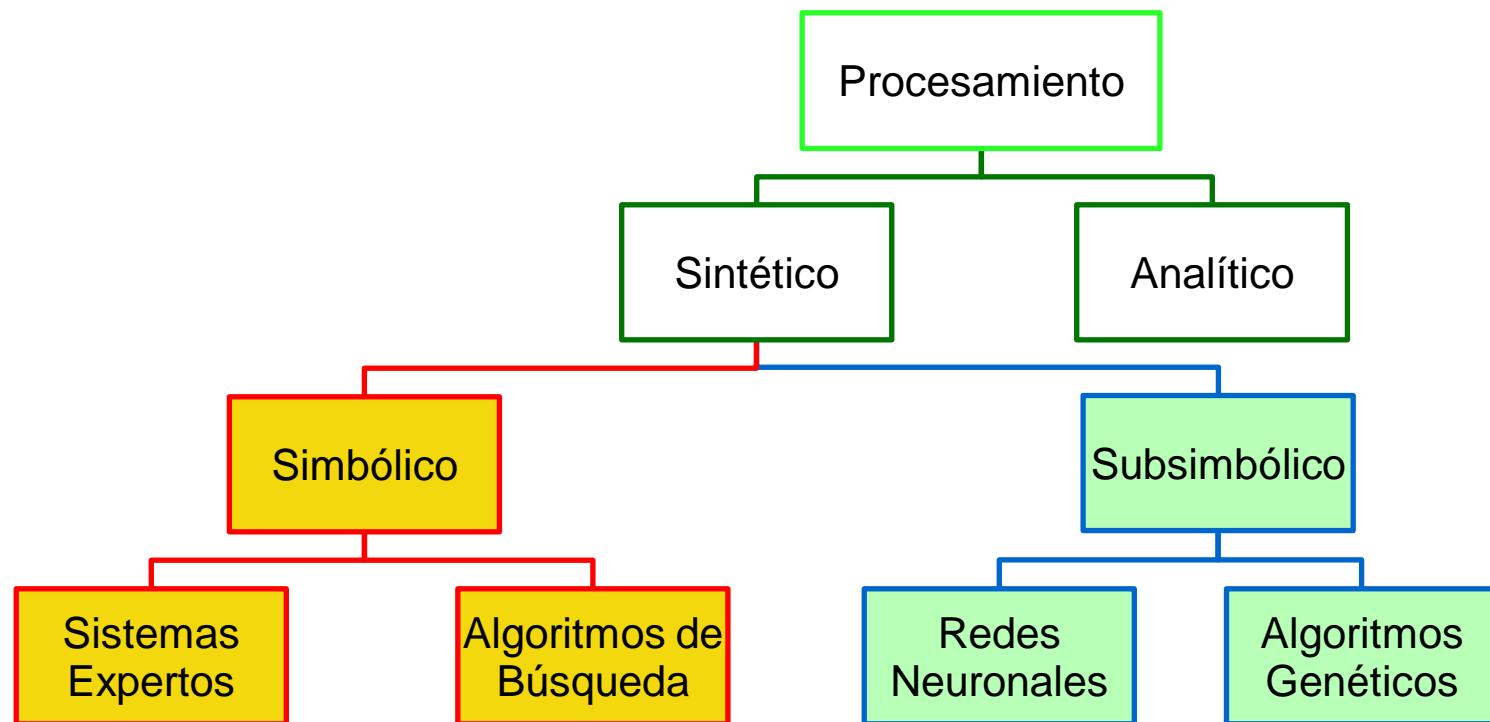
1. Aproximación simbólica
2. Técnicas simbólicas
3. Áreas
 1. Búsqueda en espacio de estados
 2. Representación/Ingeniería del Conocimiento
 3. Otras áreas



1. Aproximación simbólica
2. Técnicas simbólicas
3. Áreas
 1. Búsqueda en espacio de estados
 2. Representación/Ingeniería del Conocimiento
 3. Otras áreas



1. Aproximación simbólica



IA simbólica
(razonamiento artificial)
Aproximaciones basadas en procesamiento de símbolos

IA subsimbólica
(procesamiento biológico)
Aproximaciones basadas en procesamiento de datos



1. Aproximación simbólica

- Hipótesis del Sistema de Símbolos Físicos SSF (Newell y Simon, 1976)
 - “Un SSF tiene los medios necesarios y suficientes para producir un comportamiento inteligente”
 - Cualquier sistema (humano, animal o máquina) que exhiba inteligencia debe operar manipulando estructuras compuestas por símbolos. Procesamiento de la **información**.
- La noción de símbolo establece un vínculo entre la IA y los sistemas formales (lógica, matemáticas)
 - Computación simbólica: Un símbolo es algo que representa a otra cosa (objeto físico o concepto)
 - El símbolo “7” representa al concepto 7
 - Un símbolo es algo físico
 - EL PERRO PERSIGUE AL GATO

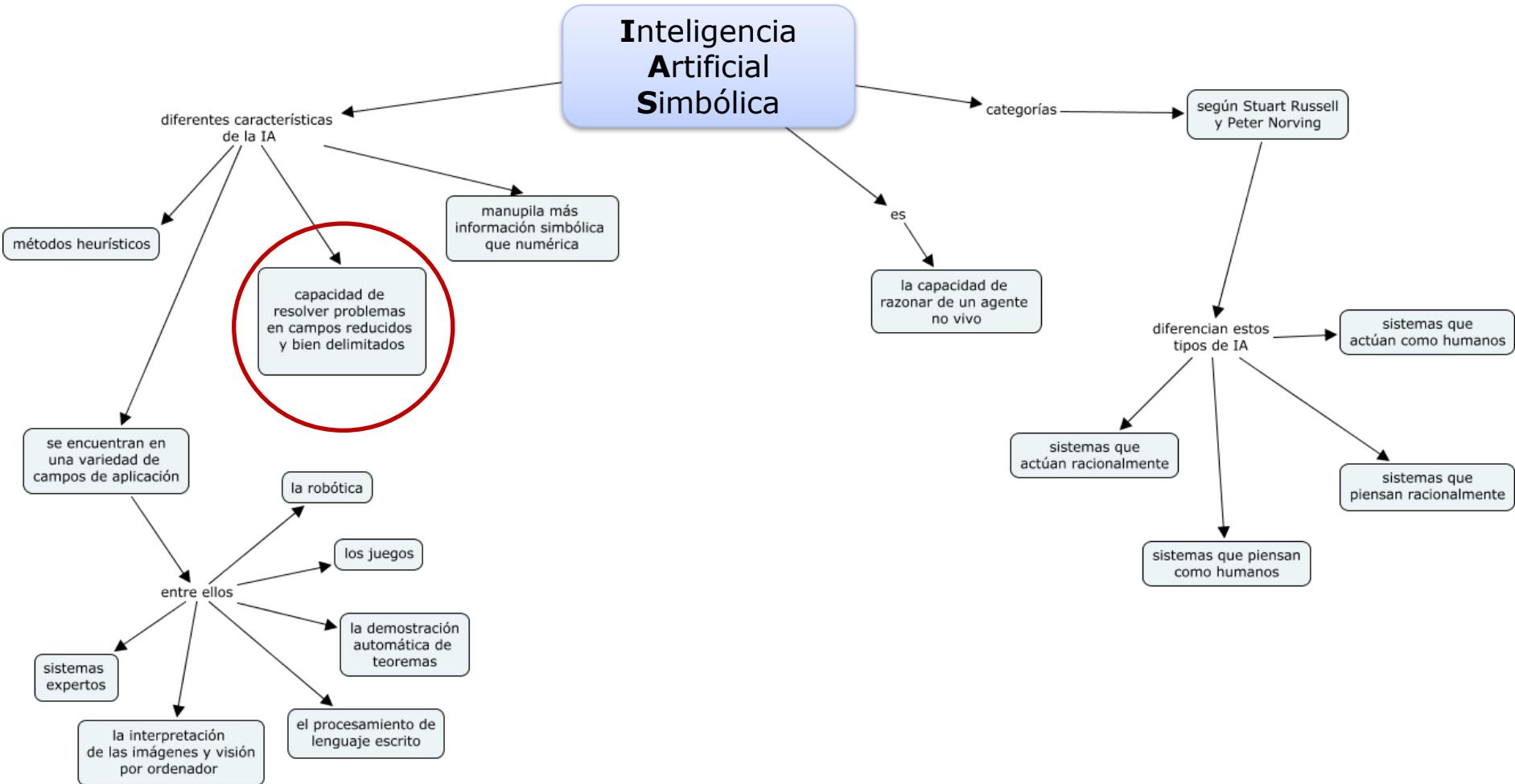


1. Aproximación simbólica

- Tres niveles en la representación del mundo real:
 - *Nivel de conocimiento* (nivel conceptual) ➔
 - Se modela la realidad mediante un modelo formal
 - *Nivel simbólico* (nivel lógico) ➔
 - El conocimiento se representa en un SSF
 - *Nivel de implementación* (nivel físico)
 - El SSF se implementa en un Lenguaje de Programación
- También funciona al revés (*es lo interesante*):
 - *Nivel de implementación* ➔
 - A partir de las expresiones simbólicas implementadas
 - *Nivel simbólico* ➔
 - se infieren nuevas estructuras simbólicas
 - *Nivel de conocimiento*
 - que pueden ser interpretadas para obtener nuevo conocimiento



1. Aproximación simbólica





1. Aproximación Simbólica
2. Técnicas simbólicas
3. Áreas
 1. Búsqueda en espacio de estados
 2. Representación/Ingeniería del Conocimiento
 3. Otras áreas

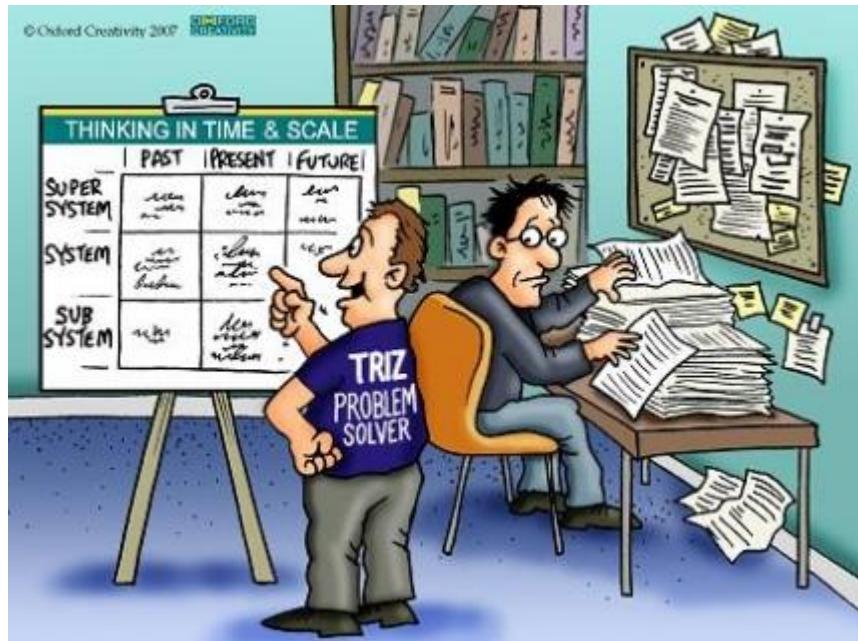
2. Técnicas simbólicas



- **Técnicas de la Computación clásica** (enfoque algorítmico): se tiene toda la información necesaria para una solución óptima del problema
- **Técnicas específicas de la IA**: no se asume conocimiento absoluto. Decisiones basadas en conocimiento parcial que no garantiza encontrar el óptimo
 - Estas técnicas se denominan **heurísticas**: estrategias de resolución de problemas que los humanos usamos y donde reside parte de la inteligencia
- **Inteligencia**: saber sacar el máximo provecho a la información disponible para obtener el resultado deseado
 - Compromiso entre exhaustividad del análisis y calidad del resultado
 - Se sacrifica la seguridad de obtener soluciones óptimas por la ventaja de poder operar con información incompleta



2. Técnicas simbólicas



- **Modelo algorítmico:**
 - La resolución de un problema se obtiene en un proceso secuencial lineal de pasos elementales a partir de unas premisas.
- **Modelo basado en el conocimiento (heurística):**
 - Hay problemas de los que no se conoce un algoritmo o no vale
 - Resolución mediante *BUSQUEDA INFORMADA*
 - Resolución mediante *SISTEMAS BASADOS EN EL CONOCIMIENTO*



2. Técnicas simbólicas

- Diferencias entre la solución algorítmica y heurística (para un constructor que recibe un encargo)
 - Calcular el precio de una casa mediante un análisis detallado:
 - Calcular materiales, llamar proveedores y subcontratistas para obtener precios, estimar contingencias razonables, etc.
 - Ventaja: el presupuesto es correcto
 - Desventaja: tiempo hasta dar una respuesta al comprador
 - Estimar el precio comparando con otras obras parecidas buscando diferencias que podrían subir o bajar el precio
 - Añadir una piscina, muebles de la cocina de pino en vez de roble, un baño menos...
 - Ventaja: rapidez en la estimación
 - Desventaja: inexactitud (a lo mejor no importa)



1. Aproximación simbólica
2. Técnicas simbólicas
3. Áreas
 1. Búsqueda en espacio de estados
 2. Representación/Ingeniería del Conocimiento
 3. Otras áreas

3.1 Búsqueda en espacio de estados



Uso de algoritmos para buscar la solución en el espacio de los posibles estados (grafo) en que se puede encontrar un problema

- Los dos elementos básicos para resolver el problema son
 - Representación del problema (Específico)
Usando el paradigma del Espacio de Estados
 - Búsqueda de la solución (General)
Búsqueda entre todos los estados posibles mediante una estrategia eficiente sobre el grafo/árbol que representa al problema
- Este paradigma es totalmente general
 - principal ventaja
 - principal inconveniente.



3.1 Búsqueda en espacio de estados

- La investigación inicial en búsquedas en espacios de estados se hizo con juegos de tablero
 - Ajedrez, tres en raya, damas...
 - Muy fácil medir el éxito o el fracaso
 - En comparación con otras aplicaciones de IA (comprensión del lenguaje, etc.) los juegos no necesitan mucho conocimiento
 - Conjunto de reglas de juego bien definido que facilita la generación del espacio de búsqueda
 - Las configuraciones de tablero se representan fácilmente en una máquina
 - No hay implicaciones éticas, económicas...
- Primer intento: búsqueda exhaustiva en el árbol de estados del juego



3.1 Búsqueda en espacio de estados

- *Pero los juegos pueden generar espacios de búsqueda inmensos. Se precisan técnicas (heurísticas) para determinar qué alternativas se exploran*

Heurística ≈ Inteligencia

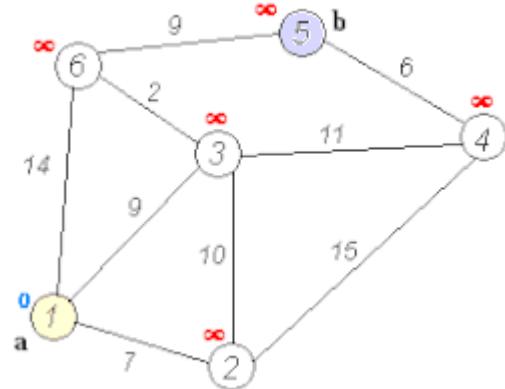
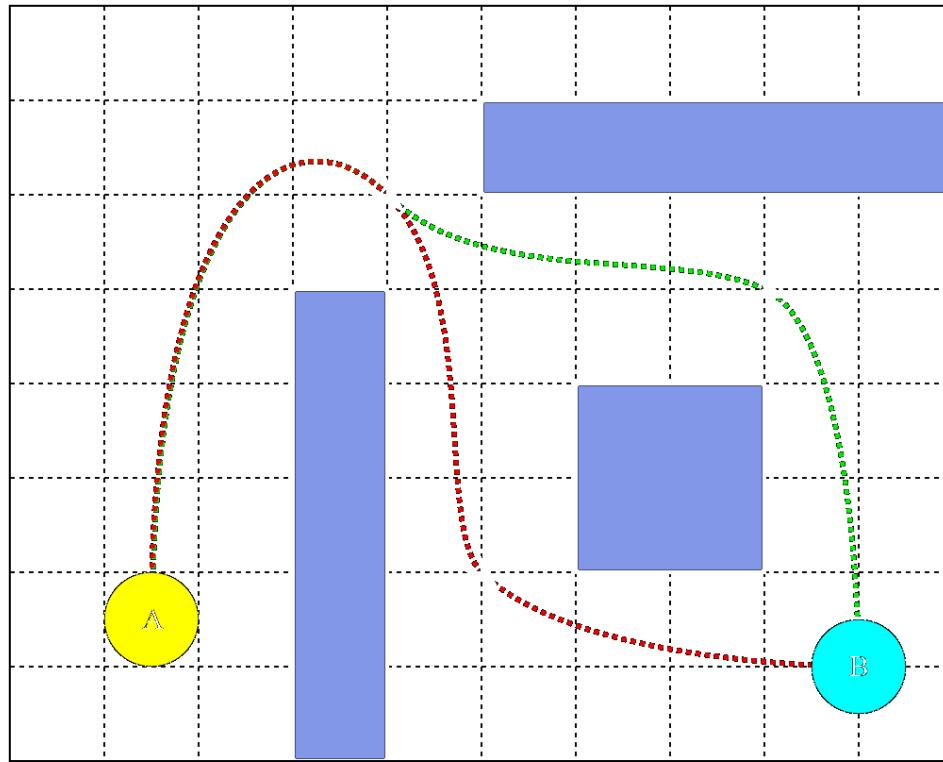


- Tipos de búsqueda
 - Búsqueda no informada o ciega
 - Búsqueda informada o heurística: *Para problemas de tamaño real es necesario dirigir esta búsqueda usando conocimiento heurístico*
 - Algoritmos genéricos de búsqueda en grafos: A* (**puzzles**)
 - Búsqueda con adversarios: minimax (**juegos de dos jugadores**)
 - Búsqueda con restricciones (**8 reinas**)



3.1 Búsqueda en espacio de estados

- Un caso particular: encontrar el camino entre dos puntos (**pathfinding**)
 - Búsqueda de la ruta más corta entre dos puntos
 - Uso del Algoritmo de Dijkstra en grafos



3.2 Representación/Ingeniería del conocimiento



Uso de formalismos para representar la realidad de forma que se puedan realizar inferencias a partir de dicho conocimiento.

Sistemas que permiten generar nuevo conocimiento (inferencia) a partir del conocimiento explícito almacenado en las bases de conocimiento.

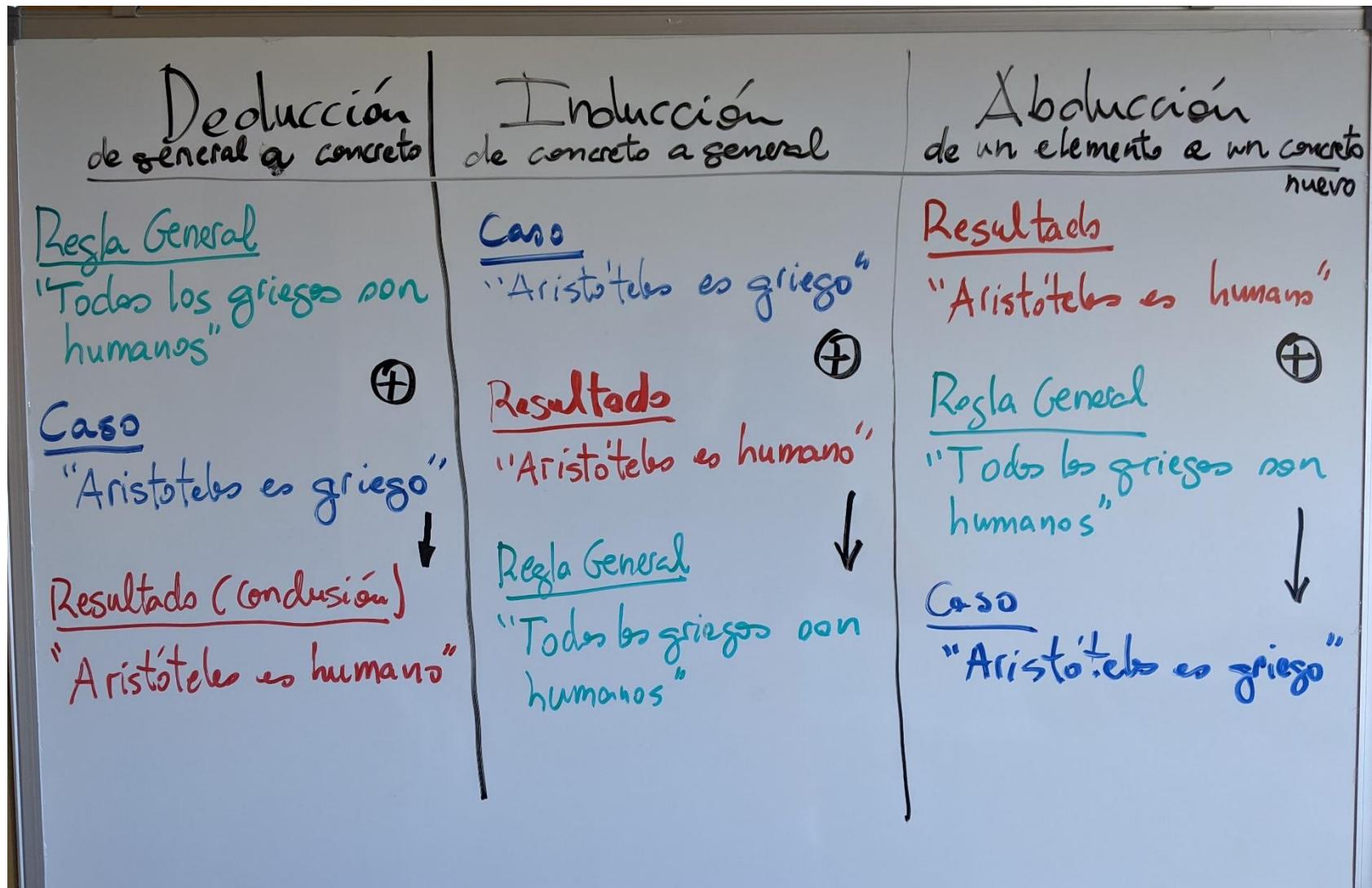
- El conocimiento se construye durante toda la vida de un ser humano y se almacena en el cerebro
- A menudo somos incapaces de expresarlo de una forma entendible al 100%
 - El lenguaje natural es **ambiguo** e **impreciso**
 - Esto obliga a usar *Esquematizaciones del lenguaje* → **lenguajes formales** (¿ejemplos?)

3.2 Representación/Ingeniería del conocimiento



- Cada formalismo de representación del conocimiento usa un método de inferencia (**Razonamiento**) específico
 - Lógica → *Inferencia*: obtención de nuevo conocimiento a partir del conocimiento de partida.
 - *Deducción*: A partir de leyes generales obtenemos conocimiento particular.
 - *Inducción*: Es la generalización de la información extraída de casos particulares. No se puede garantizar la validez de la inferencia. Es la base del aprendizaje
 - *Abducción*: Es la capacidad de generar explicaciones plausibles para un cierto hecho que ha ocurrido.
 - Reglas → Razonamiento hacia adelante, Razonamiento hacia atrás
 - Etc...

3.2 Representación/Ingeniería del conocimiento



3.2 Representación/Ingeniería del conocimiento



- Los dos elementos básicos para resolver el problema son
 - Representación del problema (Específico)
Usando un modelo descriptivo de las características del problema
 - Obtención de la solución (General)
Mediante sistemas que generan nuevo conocimiento (razonan) a partir del conocimiento explícito contenido en el modelo descriptivo.
- Los sistemas así creados se denominan **Sistemas Basados en el Conocimiento**
 - Usan razonamiento Lógico, probabilístico, temporal, incierto...
 - Normalmente el razonamiento se lleva a cabo mediante un motor de inferencia

3.3 Otras áreas



- **Aprendizaje automático (Machine Learning)**

Generalización de comportamientos a partir de información no estructurada en forma de ejemplos (inducción del conocimiento)

- Inductivo ([algoritmo ID3](#))
- Deductivo
- Árboles y Redes de Decisión

- **Procesamiento de Lenguaje Natural (NLP)**

Área de la IA que estudia la capacidad de entender y generar lenguaje humano (hablado/escrito)

- Subyace a la mayoría de las aplicaciones: interfaces de programas, comprensión de noticias, filtrado de información
- Depende de conocimiento implícito del dominio y aplicación de conocimiento contextual para resolver omisiones y ambigüedades

3.3 Otras áreas



Ejemplo de Machine Learning + NLP

- Softbots o knowbots para automatizar tareas y facilitar el uso de internet
 - Papel de asistente personal o mayordomo
 - Inicialmente, observan las tareas del usuario. Posteriormente, intentan automatizar aquéllas que el usuario realiza rutinariamente
 - Detectan a otros agentes en la red y colaboran con ellos

3.3 Otras áreas



■ Planificación

Proceso de generar secuencias de acciones para conseguir un objetivo dado (plan) a partir de una descripción de la situación actual

- Basada en estados (un caso particular de búsqueda)
- Basada en lógica ([PDDL](#))
- Cuestiones a abordar:
 - Representación del mundo y de las acciones que lo transforman
 - Algoritmos de búsqueda de planes
 - Minimizar los recursos consumidos por el plan
 - Tiempo en el que se realiza cada acción



Universidad
Francisco de Vitoria
UFV Madrid

Ingeniería del Conocimiento

Tema 2:
**Representación mediante Espacio
de Estados**

Objetivos del tema



- Ubicación
 - Unidad 2: **BUSQUEDA EN ESPACIO DE ESTADOS**
 - *Tema 2: Representación mediante Espacio de Estados*
- Objetivos generales
 - *Definir Espacio de Estados* y sus *componentes*. Estudiar su Complejidad
 - Comprender la importancia de la adecuada *representación de problemas* en espacio de estados
 - Desarrollar la *capacidad de representar* problemas simples empleando esta formulación
 - Acercamiento al concepto *búsqueda* mediante ejemplos
 - Distinguir los *distintos tipos de búsquedas* de la solución que podremos definir



1. Introducción
2. Problemas y resolución
3. Espacio de Estados
 1. Definición
 2. Búsqueda
4. Aplicación
5. Ejemplos



1. Introducción
2. Problemas y resolución
3. Espacio de Estados
 1. Definición
 2. Búsqueda
4. Aplicación
5. Ejemplos



1. Introducción

- Los dos elementos básicos para resolver un problema son
 - Representación del problema
 - Primer paso
 - Consiste en especificar el problema usando el paradigma del Espacio de Estados
 - Búsqueda de la solución
 - Buscar entre todos los estados posibles aquel que es solución al problema
 - Mediante una estrategia de búsqueda potente y eficiente
 - Es un mecanismo genérico
- *Ventaja:* se pueden aplicar procedimientos generales de búsqueda de soluciones *independientes* del problema



1. Introducción
2. Problemas y resolución
3. Espacio de Estados
 1. Definición
 2. Búsqueda
4. Aplicación
5. Ejemplos

2. Problemas y resolución



Problema

Abstracción

Expresión como Espacio de Estados

El mundo real es muy complejo es necesario realizar una **abstracción** para omitir los detalles irrelevantes

Implementación en un Lenguaje de Programación

Aplicación de Algoritmos de Búsqueda

Interpretación

Solución

2. Problemas y resolución



- Tipos de problemas según el conocimiento (abstracción) del problema:
 - Si se conoce las acciones y el estado actual
 - Entorno determinista y accesible →
 - Problema de un solo estado inicial (*single-state*)
 - Si conoce las acciones pero no el estado actual
 - Entorno determinista e inaccesible →
 - Problema de conjuntos de estados iniciales (*multiple-state*)
 - Si el conocimiento sobre acciones y estado actual es incompleto
 - Entorno no determinista e inaccesible →
 - Problema de contingencia: Durante la resolución se calcula un árbol de acciones cuyas ramas tratan distintos casos
 - Si se desconoce completamente las acciones
 - Espacio de estados desconocido →
 - Problema de exploración: experimentar y descubrir información sobre acciones y estados



1. Introducción
2. Problemas y resolución
3. Espacio de Estados
 1. Definición
 2. Búsqueda
4. Aplicación
5. Ejemplos



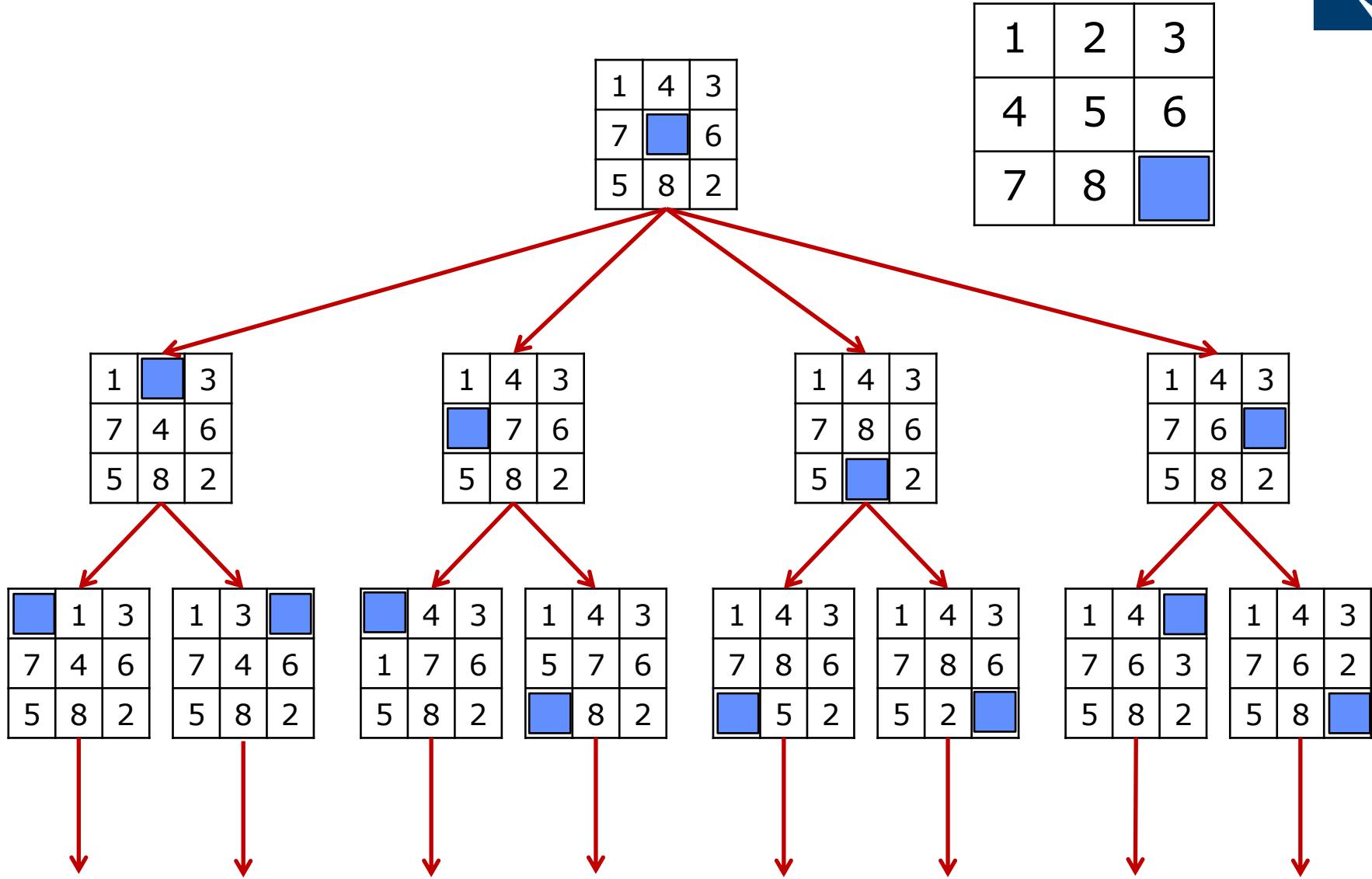
3.1 Definición de Espacio de Estados

- *Espacio de Estados* de un problema: forma de representar un problema para facilitar su resolución.

Modelo matemático de un sistema físico consistente en un grafo en el que se representan todos y cada uno de los posibles estados en los que se puede encontrar el sistema y que debe de ser representable mediante un árbol

- *Especificar un problema* como espacio de estados es describir cada uno de los componentes de ese espacio (es decir, del grafo que lo representa)

3.1 Definición de Espacio de Estados





3.1 Definición de Espacio de Estados

- **Grafo:** estructura de información compuesta de **Nodos** (*piezas de información*) + **Arcos** (*uniones entre ellos*)
 - Hojas: nodos sin descendientes (los últimos)
 - Camino: sucesión de nodos siguiendo los arcos
 - Ciclo: camino cerrado (bucle)
- **Árbol:**
 - Es un **grafo dirigido acíclico conexo**
 - *Grafo dirigido*: los arcos indican el sentido de la relación
 - *Grafo acíclico*: no tiene ciclos
 - *Grafo conexo*: entre dos nodos siempre hay un camino
 - En el que
 - Hay un único nodo raíz
 - Cada nodo tiene un único parente
 - Para cada nodo existe un único camino que lo conecta con el nodo raíz



3.1 Definición de Espacio de Estados

- Elementos del Espacio de Estados para problemas single-state

Elemento	Pregunta
Conjunto de estados del problema	¿Cuántos estados hay? ¿Cómo se representan? ¿Cuál es el <i>árbol</i> de estados?
Estado(s) inicial(es)	¿cuál es la <i>situación inicial</i> de la que se parte?
Estado(s) final(es) o test de finalización	¿cuál es el <i>objetivo final</i> ?
Conjunto de operadores permitidos para cambiar de estado	¿Qué <i>acciones</i> se pueden llevar a cabo en cada momento para cambiar las situaciones y cómo cambian?
Función de coste de la solución $g(x)$	¿Cuánto <i>cuesta alcanzar</i> esa situación en concreto? Suma del coste de las <i>acciones del camino</i> desde el estado inicial hasta ese estado

- Un camino es una *secuencia de operadores*.

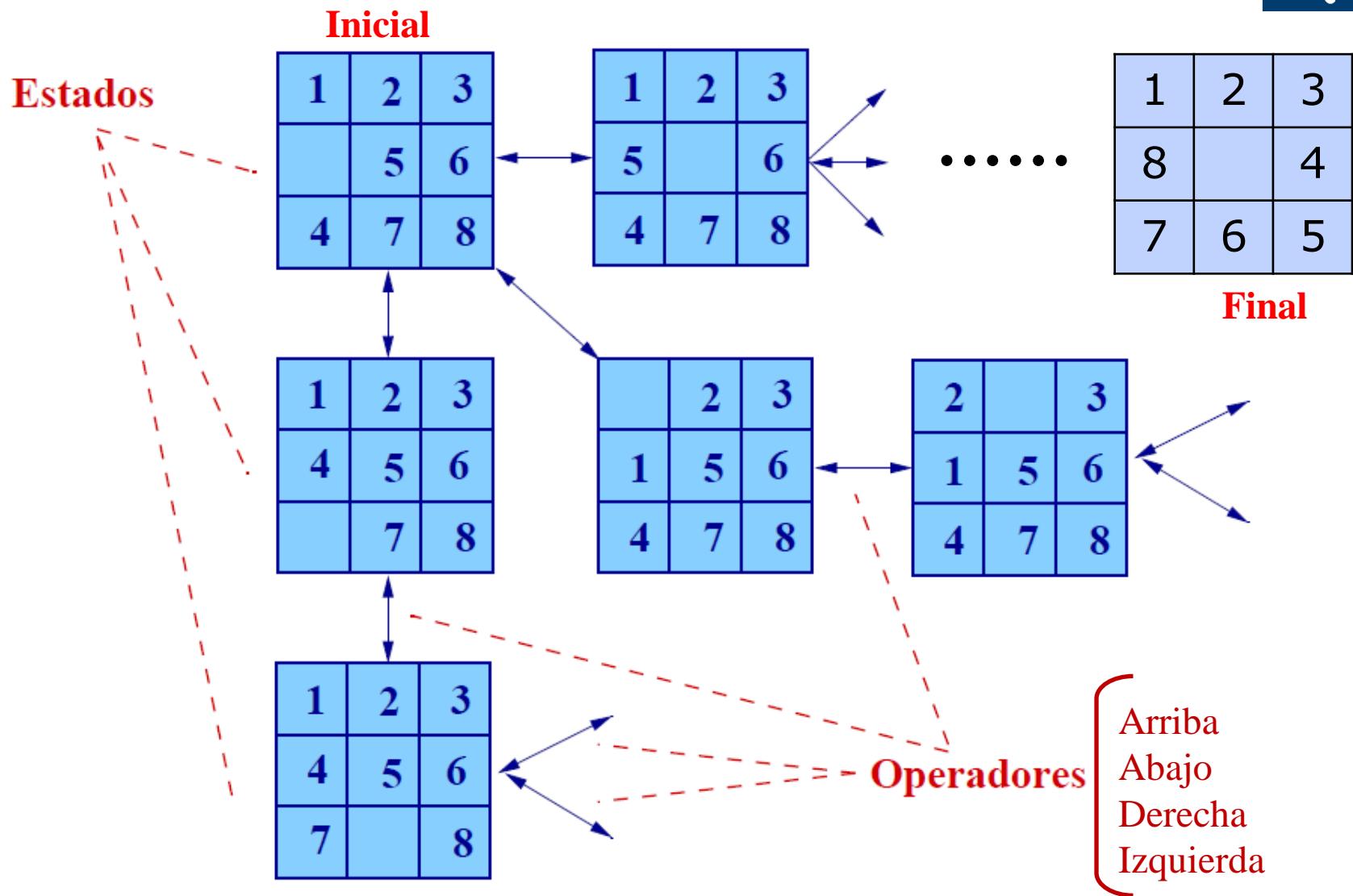


3.1 Definición de Espacio de Estados

1. Conjunto de Estados
 1. Abstracción
 2. Nivel Conceptual
 3. Nivel Lógico
 4. Nº de Estados
2. Estado Inicial
3. Estado Final
4. Operadores
 1. Definición/Nombre
 2. Precondición
 3. Estado resultante
 4. Poscondición
 5. Precedencia
5. Coste
6. Grafo



3.1 Definición de Espacio de Estados



3.1 Definición de Espacio de Estados



- Representación de Estados
 - Abstracción de propiedades
 - Niveles de representación
 - **Nivel conceptual:** se especifican estados y operadores, sin hacer referencia a estructuras de datos o algoritmos que vayan a usarse
 - Descripción de todas las posibles situaciones en el problema
 - Hay descripciones válidas e inválidas (violan el enunciado) ➔ no son estados
 - Enumeración de estados (solo los válidos)
 - Formas de describir los estados:
 - Enumerativa.
 - Declarativa.
 - **Nivel lógico:** se elige una estructura de datos para los estados y se determina el formato de codificación de los operadores
 - Importancia de una buena representación de los estados
 - Solo considerar información relevante para el problema
 - Representación suficiente y necesaria
 - La representación escogida influye en el numero de estados y éste en los procedimientos de búsqueda de soluciones

3.1 Definición de Espacio de Estados



- Operadores:

- Representan un conjunto finito de acciones básicas que transforman unos estados en otros
- Elementos que describen un operador
 - Aplicabilidad: precondición y postcondición
 - Estado resultante de la aplicación de un operador (aplicable) a un estado
 - Hay estados válidos pero inalcanzables (espacios **no** conexos)
- Criterio para elegir operadores.
 - Depende de la representación de los estados
 - Preferencia por representaciones con menor número de operadores (lo más generales y aplicables posible)
 - Ejemplo: en el 8-puzzle
 - 32 operadores si consideramos el movimiento de los bloques
 - 4 operadores si consideramos el movimiento del hueco

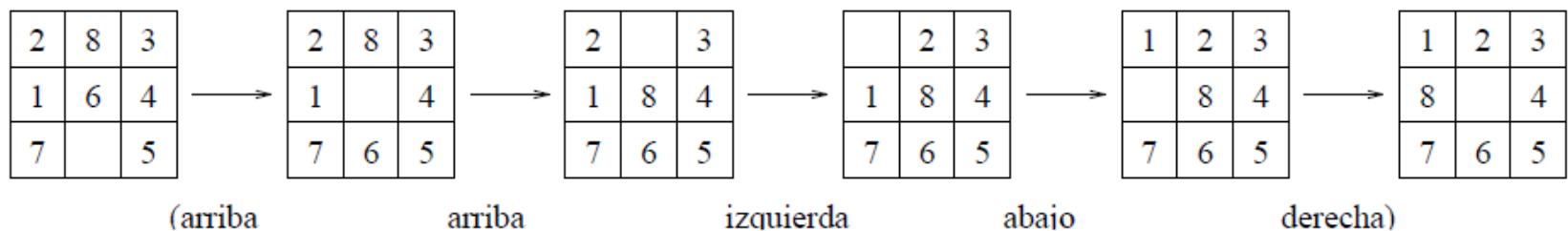


3.1 Definición de Espacio de Estados

- Solución

Resolución del problemas = búsqueda de la solución en el espacio de los posibles estados (grafo) en que se puede encontrar un problema

Una solución es un camino que conduce del estado inicial a un estado que satisface el test de objetivo



- Solución óptima: la que minimiza la función de coste



3.1 Definición de Espacio de Estados

Dominio	Número de estados	Tiempo (10^7 nodos/s)
8-puzzle	$\left(\frac{N^2!}{2}\right) \Big _{N=3} = 181,440$	0.01 segundos
15-puzzle	$\left(\frac{N^2!}{2}\right) \Big _{N=4} = 10^{13}$	11,5 días
24-puzzle	$\left(\frac{N^2!}{2}\right) \Big _{N=5} = 10^{25}$	$31,7 \times 10^9$ años
Hanoi (3,2)	$(3^n) \Big _{n=2} = 9$	9×10^{-7} segundos
Hanoi (3,4)	$(3^n) \Big _{n=4} = 81$	$8,1 \times 10^{-6}$ segundos
Hanoi (3,8)	$(3^n) \Big _{n=8} = 6561$	$6,5 \times 10^{-4}$ segundos
Hanoi (3,16)	$(3^n) \Big _{n=16} = 4,3 \times 10^7$	4,3 segundos
Hanoi (3,24)	$(3^n) \Big _{n=24} = 2,824 \times 10^{11}$	0,32 días
Cubo de Rubik $2 \times 2 \times 2$	10^6	0,1 segundos
Cubo de Rubik $3 \times 3 \times 3$	$4,32 \times 10^{19}$	31.000 años

Hanoi: $T(n) = 2T(n-1) + 1; T(1) = 1; T(n) = 2^n - 1$

3.2 Búsqueda en Espacio de Estados

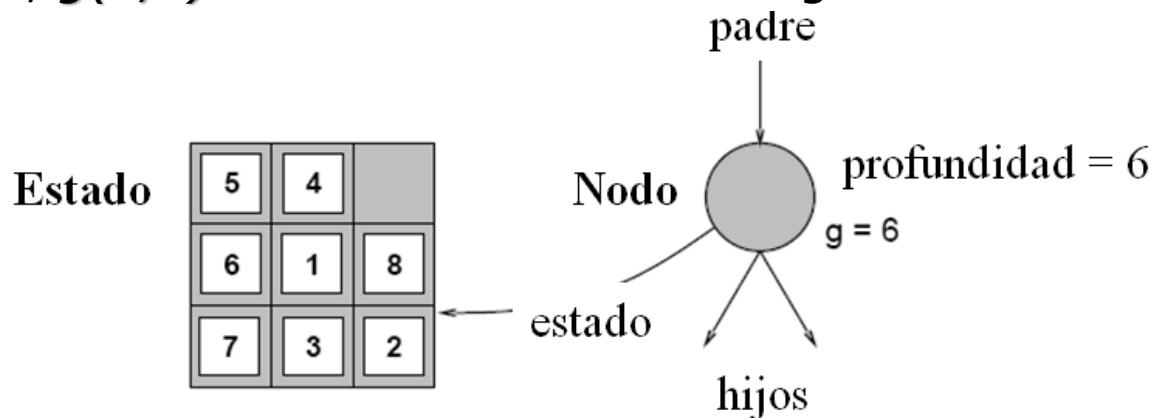


- La búsqueda es la exploración simulada del grafo del espacio de estados por medio de la generación de sucesores de los estados ya explorados
 - Genera un árbol de soluciones a partir del estado inicial del Espacio de Estados y los operadores que generan estados
 - El árbol generado depende del algoritmo de búsqueda utilizado
- **Nodo**: estructura de datos que forma parte de un árbol de búsqueda
 - Frontera: conjunto de nodos pendientes de expandir
- **Objetivo**: encontrar una secuencia de operadores que, partiendo del estado inicial, obtenga un estado final



3.2 Búsqueda en Espacio de Estados

- Parámetros de un nodo:
 - Estado: basta con poder diferenciarlo de otros
 - Padre: nodo del que es sucesor
 - Hijos: nodos sucesores
 - Acción: acción que nos llevó del padre a hijo
 - Factor de ramificación, b : número de sucesores de un nodo (propiedad del grafo de estados)
 - Profundidad del árbol de búsqueda, d : número de pasos desde el origen (propiedad del problema concreto a resolver)
 - Coste, $g(o,n)$: coste de ir desde el origen al nodo n ($\sum g_i$)



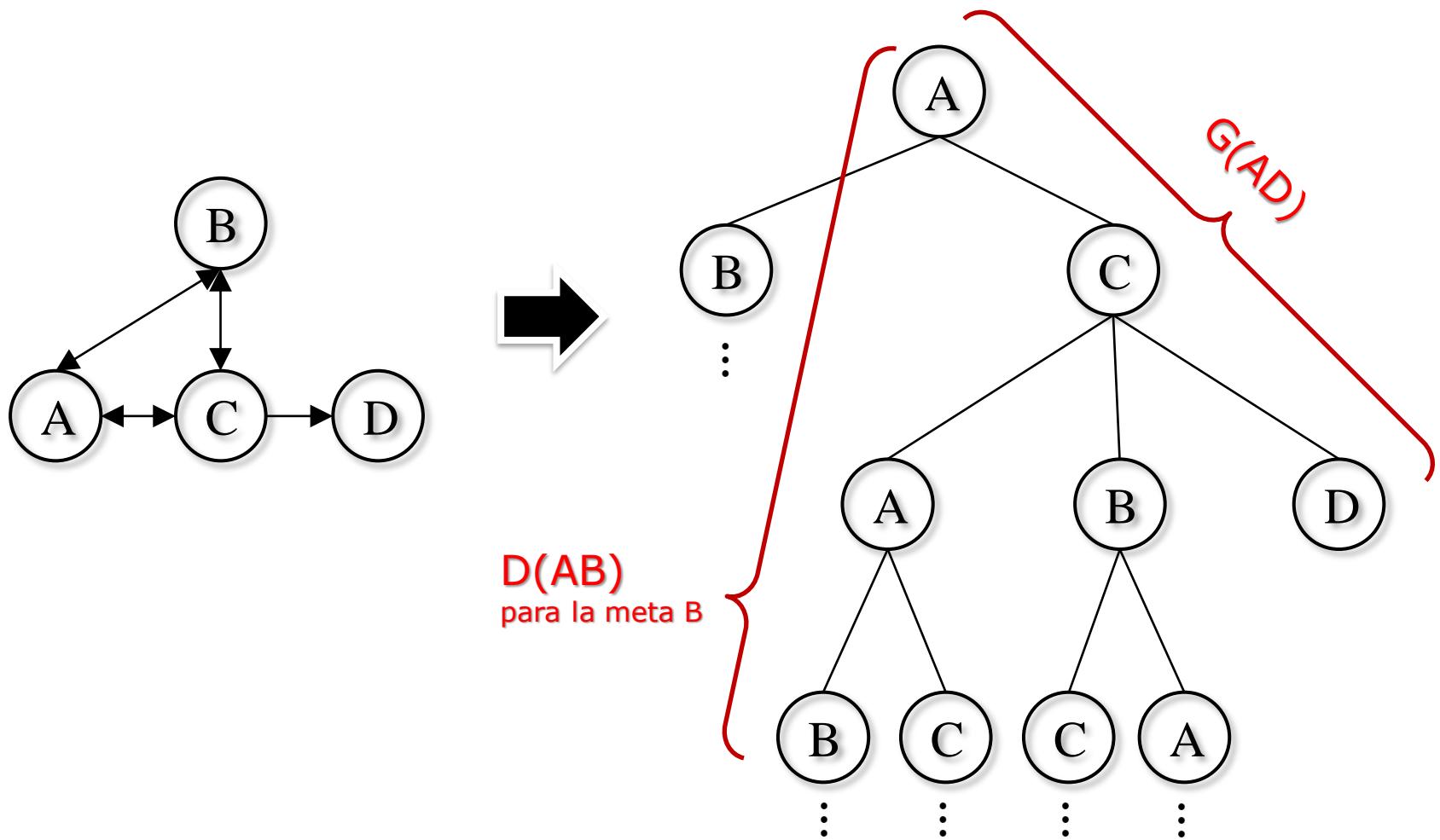
3.2 Búsqueda en Espacio de Estados



- El espacio de búsqueda (*árbol*) se construye incrementalmente sobre el espacio de estados (*grafo*)
- La elección del nodo a analizar en cada momento determina una *estrategia* de búsqueda
- Aunque el grafo sea finito, el árbol puede ser infinito (ciclos del grafo)
 - Nodos distintos del árbol pueden corresponderse con el mismo estado del espacio de estados



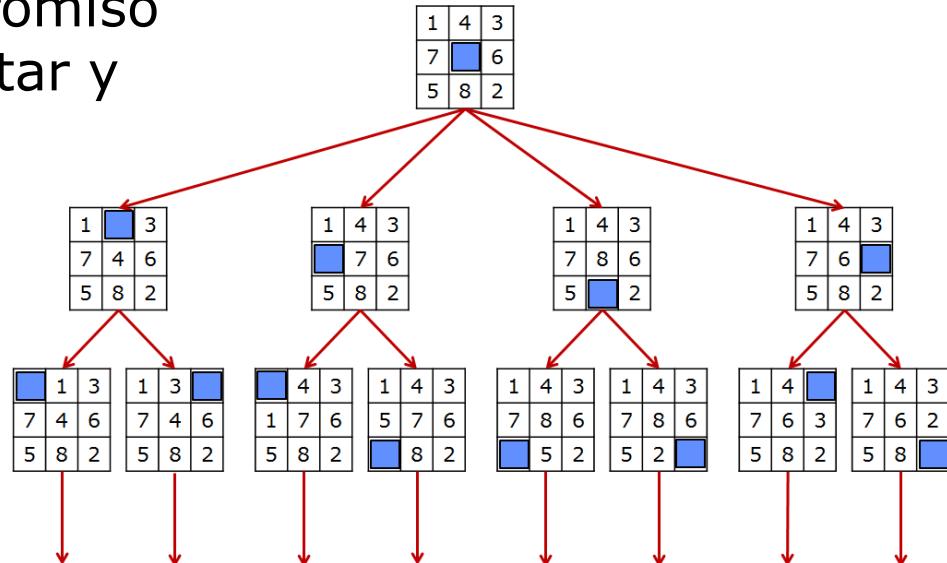
3.2 Búsqueda en Espacio de Estados





3.2 Búsqueda en Espacio de Estados

- Es imprescindible evitar la repetición de estados (ciclos del grafo) pero tiene un coste:
 - Evitar aplicación sucesiva de operadores inversos (barato)
 - Guardar o marcar estados del camino actual
 - Marcar todos los estados generados para evitar la repetición de cualquier estado (costoso)
- Hay que llegar a un compromiso entre lo que se intenta evitar y el coste de evitarlo
- Esto es un árbol (no un grafo) y ya está “optimizado”





3.2 Búsqueda en Espacio de Estados

- Los problemas
 - de un solo estado (*single-state*)
 - de conjuntos de estados (*multiple-state*)se pueden resolver mediante estrategias de
búsqueda no informada o ciega
(no hay información adicional disponible)
- El resto de problemas **Y** los problemas
 - *single-state*
 - *multiple-state*demasiado complejos (con espacios de estados imposibles) requieren del uso de
búsqueda informada o heurística
(las heurísticas ayudan a disminuir la complejidad del problema)



1. Introducción
2. Problemas y resolución
3. Espacio de Estados
 1. Definición
 2. Búsqueda
4. Aplicación
5. Ejemplos

4. Aplicación



- Tipos de problemas:
 - Determinar si existe solución y encontrar un estado final.
 - Buscar cualquier solución lo más rápidamente posible.
 - Buscar todas las soluciones.
 - Buscar la solución más corta.
 - Buscar la solución menos costosa.



4. Aplicación

- Casos reales
 - Buscar rutas
 - Redes de ordenadores
 - Sistemas automáticos de guiado en viajes
 - Planificación de viajes
 - Problema del viajante: cada ciudad exactamente una vez
 - Diseño del layout de VLSI
 - Navegación de Robots
 - Aplicaciones espaciales (Curiosity, etc.)
 - Videojuegos
 - Ensamblaje automático: el orden importa, búsqueda geométrica difícil
 - Diseño de proteínas: plegado en 3D de fragmentos
 - Búsqueda en internet: respuestas, precios, ...



1. Introducción
2. Problemas y resolución
3. Espacio de Estados
 1. Definición
 2. Búsqueda
4. Aplicación
5. Ejemplos



Universidad
Francisco de Vitoria
UFV Madrid

Ingeniería del Conocimiento

Tema 3:
Búsqueda *NO* Informada



Objetivos del tema

- Ubicación
 - Unidad 2: **BUSQUEDA EN ESPACIO DE ESTADOS**
 - *Tema 3: Búsqueda NO Informada*
- Objetivos generales
 - *Definir búsqueda NO informada* y entender su ámbito de aplicación
 - Comprender los *distintos métodos* de búsqueda no informada en función del algoritmo de expansión de nodos
 - Saber *aplicar cada método* en función de la completitud y *complejidad* espacial y temporal
 - *Resolver problemas* de búsqueda no informada de forma teórica y práctica,
 - Analizando *grafos*
 - *Implementando* los correspondientes algoritmos



1. Introducción
2. Implementación
3. Métodos no informados
 1. Búsqueda en anchura
 2. Búsqueda de coste uniforme
 3. Búsqueda en profundidad
 4. Búsqueda en profundidad limitada
 5. Búsqueda en profundidad iterativa
 6. Búsqueda bidireccional
4. Complejidad



- 1. Introducción**
- 2. Implementación**
- 3. Métodos no informados**
 - 1. Búsqueda en anchura**
 - 2. Búsqueda de coste uniforme**
 - 3. Búsqueda en profundidad**
 - 4. Búsqueda en profundidad limitada**
 - 5. Búsqueda en profundidad iterativa**
 - 6. Búsqueda bidireccional**
- 4. Complejidad**

1. Introducción



- Búsqueda: exploración del espacio de estados por medio de la generación de sucesores de los estados explorados
- Cuando cualquier nodo del árbol de búsqueda es igualmente prometedor para alcanzar la meta, hablamos de estrategias de búsqueda
 - NO INFORMADAS
 - CIEGAS

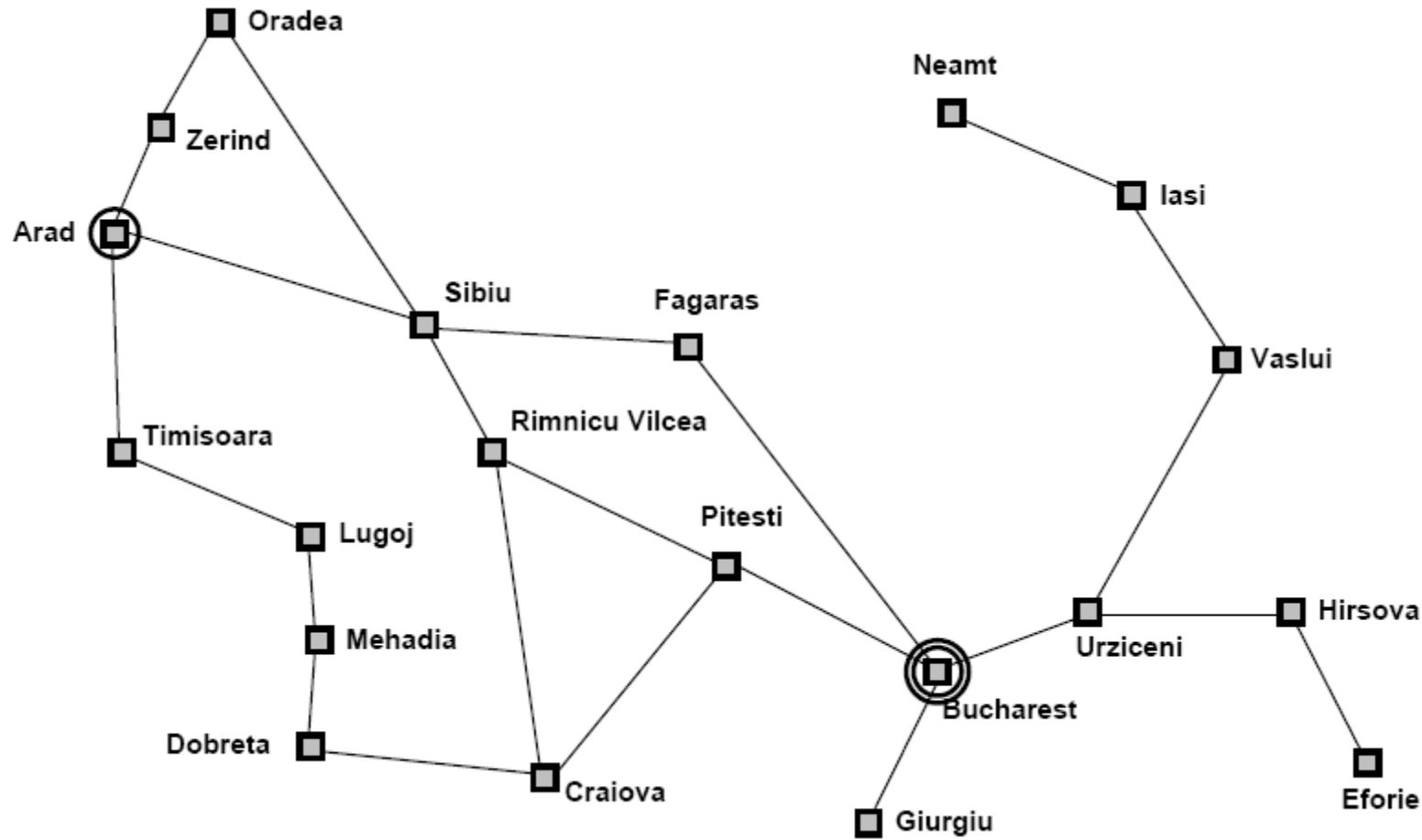
(solo usan información disponible en la definición del problema)
- Una búsqueda a ciegas requiere visitar un número de nodos mucho mayor que una búsqueda inteligente. Por ello solo se aplica a problemas simples
 - de un solo estado (*single-state*)
 - de conjuntos de estados (*multiple-state*)

1. Introducción



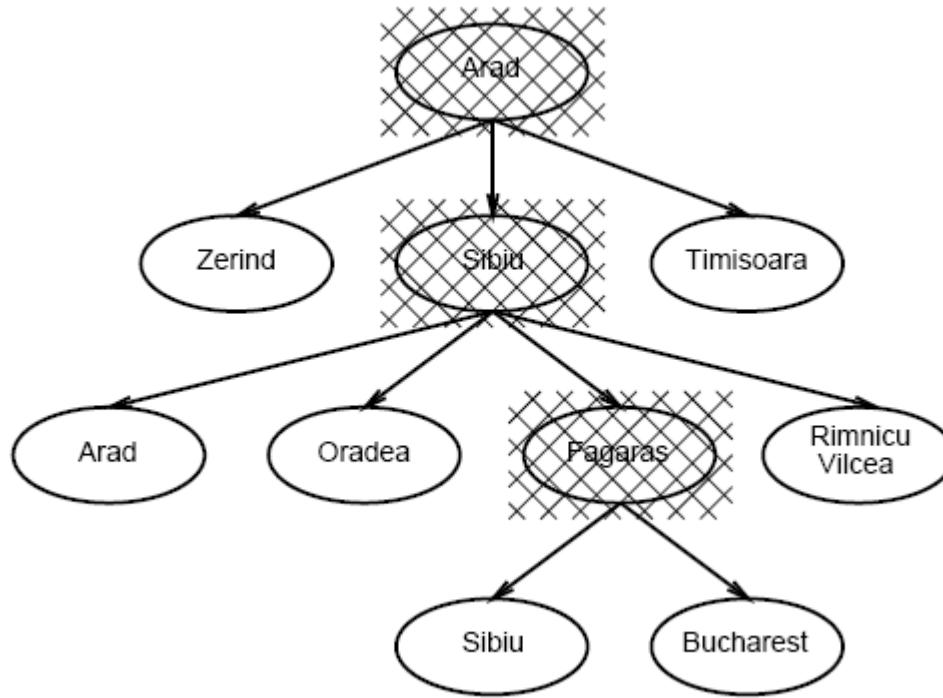
- Conceptos (*nodo*):
 - Expansión: se añaden los posibles sucesores al nodo
 - Acción: acción que nos llevó del nodo padre a “éste”
 - Frontera: conjunto de nodos pendientes de expandir
 - Función de Coste: $g(n)$ desde el nodo origen hasta “éste”
- Definición de problema (*problema bien definido*):
 - Estado inicial
 - Descripción de acciones posibles
 - Test Objetivo, que determina si el estado es objetivo
 - Función de Coste del camino
- Solución: Camino desde el estado inicial al estado objetivo
- Solución óptima: La que minimiza la función de coste

1. Introducción





1. Introducción





1. Introducción
2. Implementación
3. Métodos no informados
 1. Búsqueda en anchura
 2. Búsqueda de coste uniforme
 3. Búsqueda en profundidad
 4. Búsqueda en profundidad limitada
 5. Búsqueda en profundidad iterativa
 6. Búsqueda bidireccional
4. Complejidad

2. Implementación



- Implementación de un problema como búsqueda no informada en espacio de estados
 - Elección de una representación (estructura de datos):
 - para los estados
 - para los operadores
 - Elementos de la representación:
 - Variables: *ESTADO-INICIAL*, *ESTADO-FINAL*, ESTADO-SUCESOR, NODO-ACTUAL, *ABIERTO*, *CERRADO*
 - Lista de operadores: *OPERADORES*
 - Funciones de acceso: ESTADO(NODO), ES-ESTADO-FINAL(ESTADO(NODO)), EXTRAE-PRIMERO(ABIERTO), CAMINO(NODO), COSTE(NODO)
 - Funciones operadores: SUCESORES(NODO), FUNCION-SUCESOR(NODO,OPERADOR), GESTIONA-COLA(ABIERTO,SUCESORES)

2. Implementación



FUNCION SUCESOR(NODO,OPERADOR)

1. Si el OPERADOR no es aplicable a ESTADO(NODO),
devolver NO-APLICABLE (*precondición*)

en caso contrario,

ESTADO-SUCESOR = resultado de aplicar OPERADOR a ESTADO(NODO)

Si ESTADO-SUCESOR = NO-APLICABLE

devolver NO-APLICABLE (*poscondición*)

2. Devolver SUCESOR, un nodo

cuyo estado es ESTADO-SUCESOR

cuyo camino es añadir OPERADOR a CAMINO(NODO)

cuyo coste es añadir COSTE a COSTE(NODO)



2. Implementación

FUNCION SUCESORES(NODO)

1. Hacer *SUCESSORES* vacío
2. Para cada OPERADOR en *OPERADORES*,
 si SUCESSOR(NODO,OPERADOR) ≠ NO-APLICABLE,
 incluir SUCESSOR(NODO,OPERADOR) en *SUCESSORES*
3. Devolver *SUCESSORES*

GESTIONA-COLA(ABIERTO,SUCESSORES)

 Insertar *SUCESSORES* en *ABIERTO*

 Elegir un nuevo estado actual, dejando los restantes para analizarlos posteriormente (*frontera*)



2. Implementación

■ Exploración del grafo del espacio de estados

ABIERTO = *ESTADO-INICIAL* (cola formada por nodo inicial y CAMINO vacío)

CERRADO = vacío

BUCLE (Repetir el proceso mientras ABIERTO ≠ vacío)

1. NODO-ACTUAL = EXTRAE-PRIMERO(ABIERTO) → (expandir)
2. Poner NODO-ACTUAL en *CERRADO*
3. Si ES-ESTADO-FINAL(ESTADO(NODO-ACTUAL))
devolver CAMINO(NODO-ACTUAL) → (acabar)
4. Si no, FUNCION-SUCESORES(NODO-ACTUAL)
GESTIONA-COLA(ABIERTO, SUCESORES) → (generar sucesores)

FIN DE BUCLE

Devuelve FALLO

Madre
del
cordero

2. Implementación



- La implementación anterior es *independiente del problema*
 - ABIERTO es la *frontera*, una “cola” en la que esperan los nodos generados pendientes de ser analizados (y expandidos)
 - CERRADO contiene los nodos ya analizados:
 - Permite no iniciar la búsqueda en estados analizados
 - En particular, permite evitar ciclos en el proceso de búsqueda
 - En determinados problemas es prescindible
- GESTIONA-COLA(ABIERTO,SUCESORES): Estrategia de control responsable de elegir el **orden** en que se van explorando los estados
 - A qué nodo se aplican los operadores
 - Qué nodo se expande

2. Implementación



- **Rendimiento** de una estrategia de búsqueda se evalúa por:
 - **Completitud**
 - ¿encuentra siempre una solución si existe ésta?
 - **Optimización**
 - ¿se encuentra siempre la solución de mínimo coste?
 - **Complejidad en tiempo**
 - número de nodos generados/expandidos durante la búsqueda
 - **Complejidad en espacio**
 - máximo número de nodos en memoria (visitados o no)
- Las complejidades temporal y espacial se miden en términos de
 - b : máximo factor de ramificación del árbol de búsqueda
 - d : profundidad de la solución de menor coste
 - m : profundidad máxima del espacio de estados (puede ser ∞)

2. Implementación



- Hipótesis: el tiempo de generación de nodos sucesores es constante y no existen otros factores
- Coste total
 - Coste de la solución: coste del camino encontrado
 - Coste de la búsqueda de la solución: complejidad del algoritmo utilizado
- Hay que llegar a un compromiso entre ambos costes
 - Obtener la mejor solución posible con los recursos computacionales disponibles



1. Introducción
2. Implementación
3. Métodos no informados
 1. Búsqueda en anchura
 2. Búsqueda de coste uniforme
 3. Búsqueda en profundidad
 4. Búsqueda en profundidad limitada
 5. Búsqueda en profundidad iterativa
 6. Búsqueda bidireccional
4. Complejidad

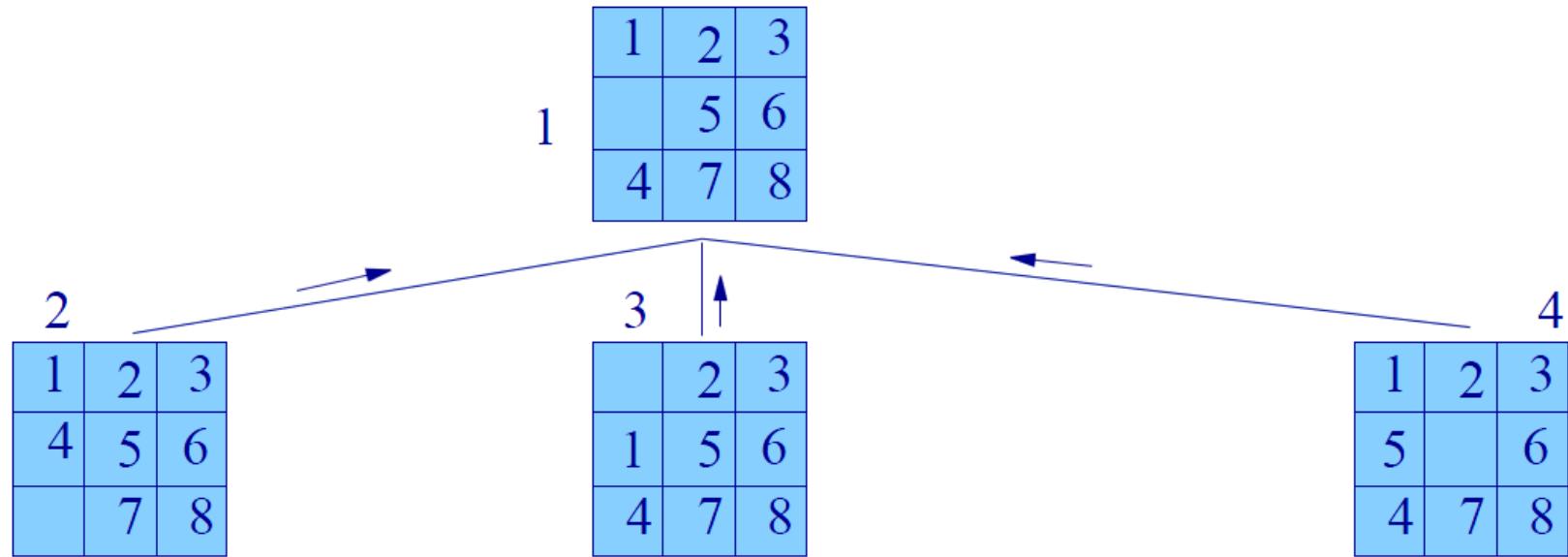


3. Métodos no informados

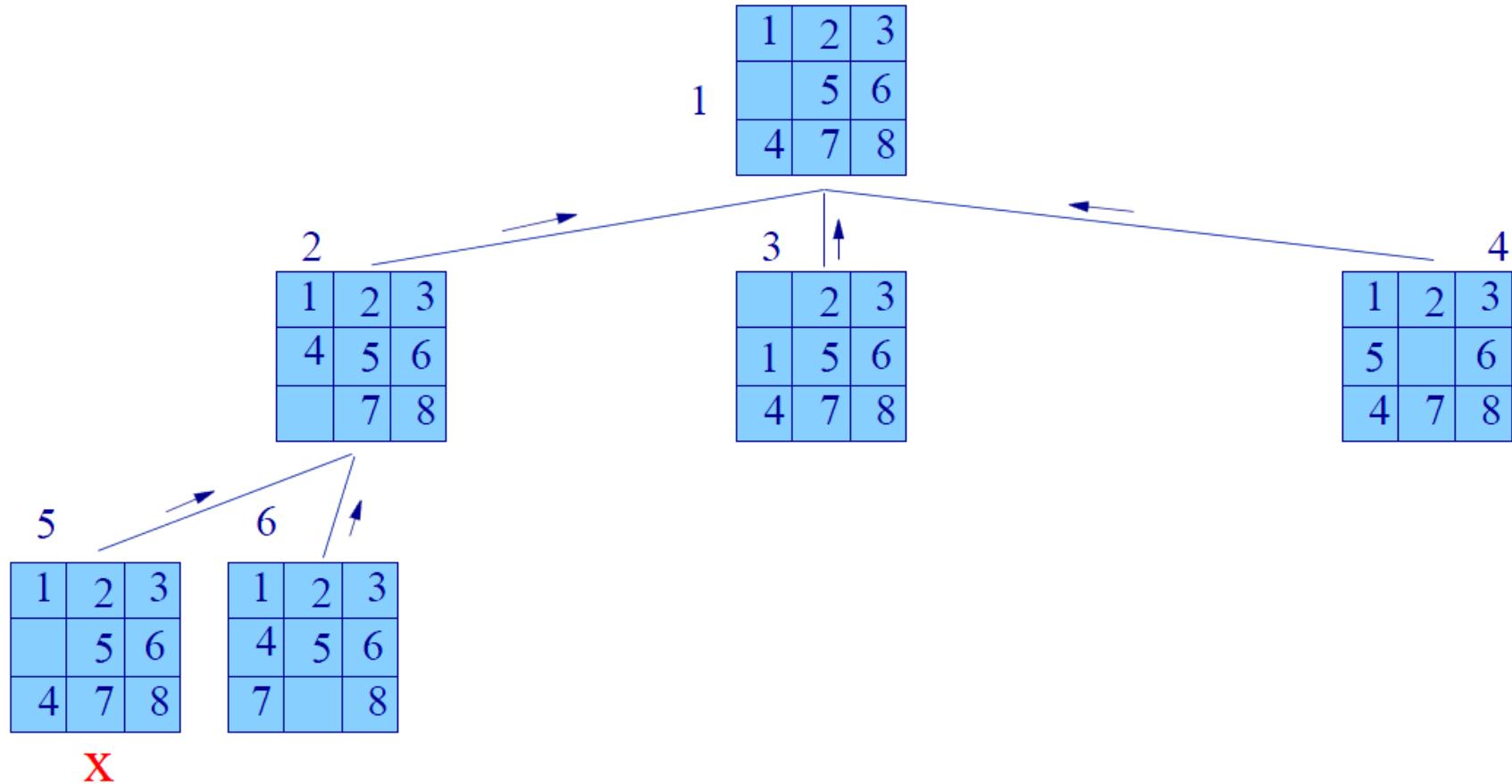
- Para problemas ciegos, se pueden usar varias estrategias según se genere la frontera y en qué orden se exploran los nodos en ella
- Tipos de Búsqueda sin Información
 - Primero en Anchura (*Breadth-first search BFS*)
 - De Coste Uniforme (*Uniform cost search UCS*)
 - Primero en Profundidad (*Depth-first search DFS*)
 - En Profundidad Limitada (*Depth-limited search DLS*)
 - En Profundidad Iterativa (*Iterative deepening search IDS*)
 - Bidireccional (*Bi-directional search BS*)



3.1 Búsqueda en anchura

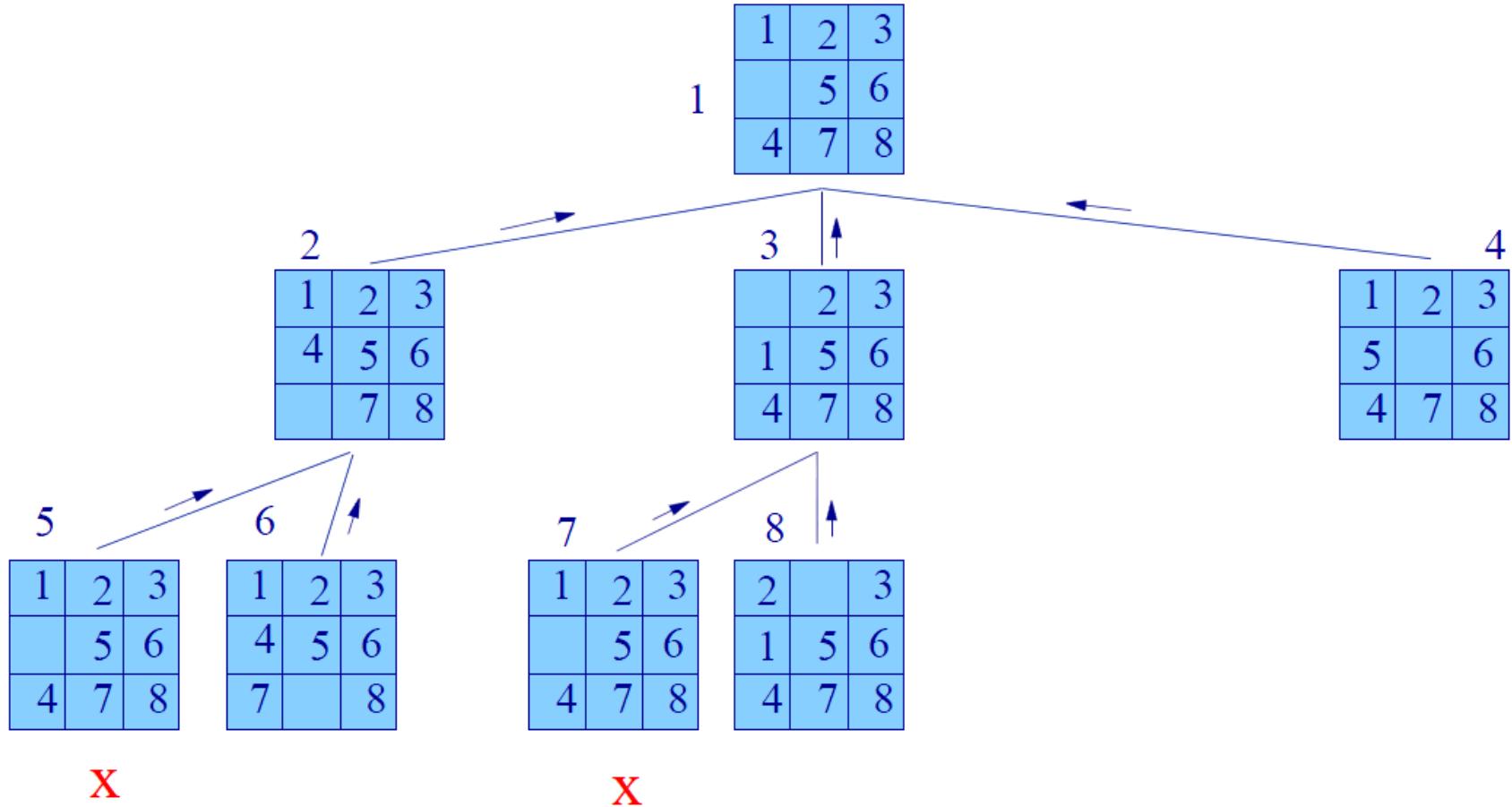


3.1 Búsqueda en anchura



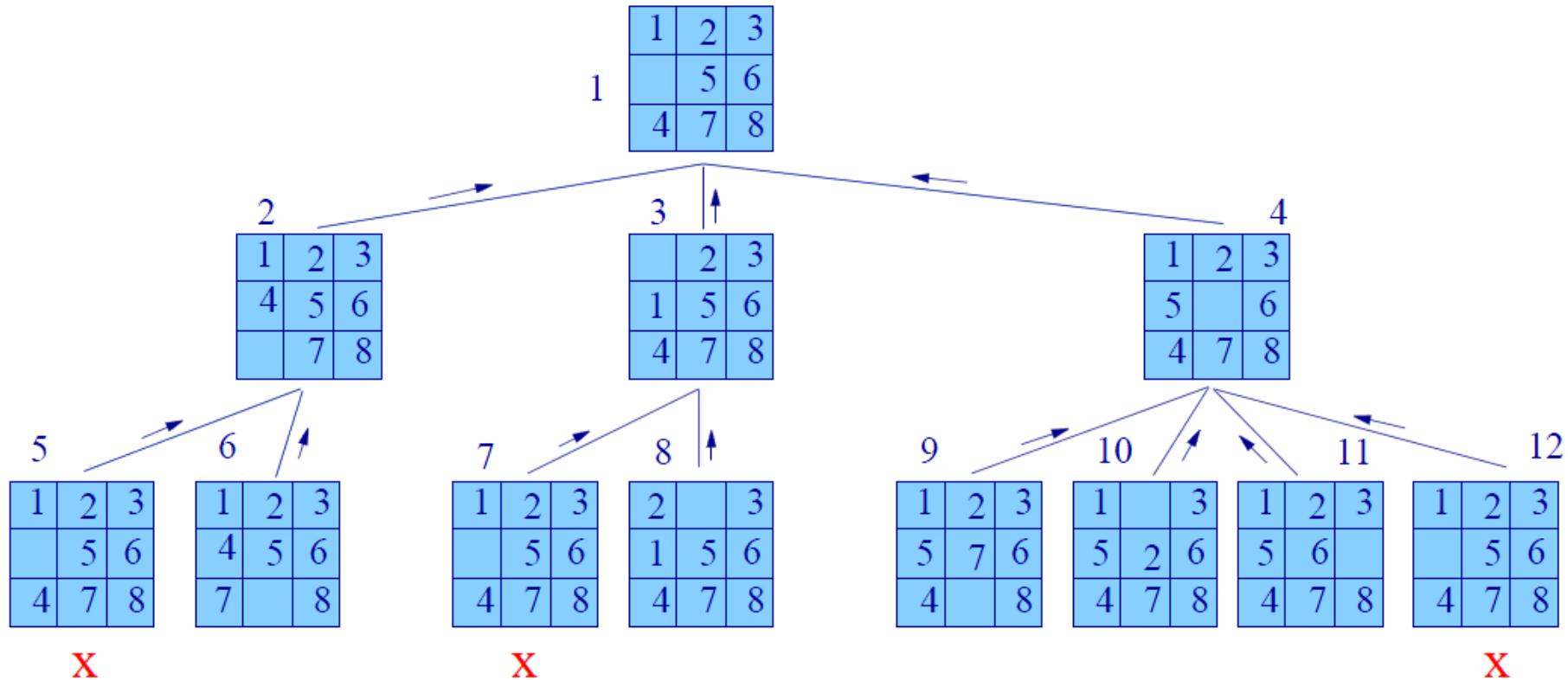


3.1 Búsqueda en anchura



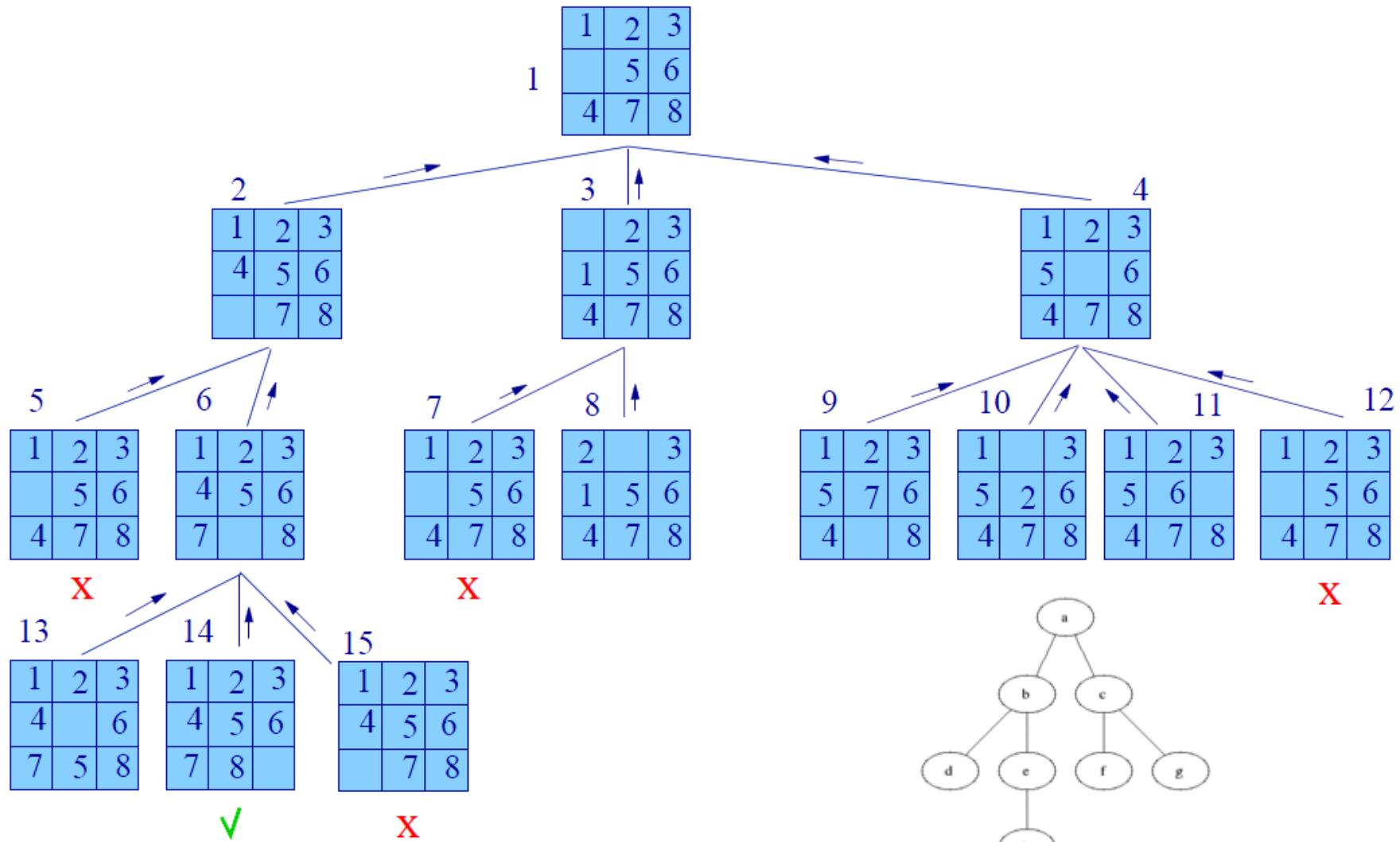


3.1 Búsqueda en anchura





3.1 Búsqueda en anchura





3.1 Búsqueda en anchura

- **Moore, 1959**
- Los nodos se expanden por orden creciente de profundidad
 - Los nodos de profundidad d se expanden antes que los nodos de profundidad $d+1$ (*por lo que no hay vuelta atrás*)
 - Expande primero los nodos no expandidos menos profundos
- ABIERTO se implementa con una **cola FIFO**
 - GESTIONA-COLA: Añadir al final de ABIERTO
- Propiedades:
 - Completa: **Si**, encuentra solución si existe y b es finito
 - Complejidad tiempo: $1 + b + b^2 + b^3 + \dots + b^d = O(b^{d+1})$
 - Complejidad espacio: $O(b^{d+1})$
 - Optima: **Si**, si todos los operadores tienen el mismo coste y (coste acción = 1)



3.1 Búsqueda en anchura

- Resumen
 - Eficiencia: buena si las metas están cercanas
 - Problema: consume memoria exponencial y hay que guardar todos los nodos en memoria. Solo viable en casos pequeños
- Para
 - Ramificación 10
 - 1000 nodos por seg.
 - 100 bytes por nodo:

Profundidad	Nodos	Tiempo	Espacio
0	1	1 ms.	100 b
2	111	0.1 seg.	11 Kb
4	11111	11 seg.	1 Mb
6	10^6	18 min.	11 Mb
8	10^8	31 horas	11 Gb
10	10^{10}	128 días	1 Tb
12	10^{12}	33 años	11 Tb
14	10^{14}	3500 años	11.111 Tb



3.1 Búsqueda en anchura

PROCEDIMIENTO ANCHURA(Estado-inicial, Estado-Final)

ABIERTO = *ESTADO-INICIAL*

Hacer *CERRADO* vacío

BUCLE (Repetir el proceso mientras ABIERTO ≠ vacío)

1. NODO-ACTUAL = EXTRAE-PRIMERO(ABIERTO)
2. Poner NODO-ACTUAL en *CERRADO*
3. Si ES-ESTADO-FINAL(ESTADO(NODO-ACTUAL))
devolver CAMINO(NODO-ACTUAL)
4. Si no, FUNCION SUCESORES(NODO-ACTUAL)
GESTIONA-COLA(ABIERTO,SUCESORES)
Añadir *SUCESORES* al final de ABIERTO (cola FIFO)

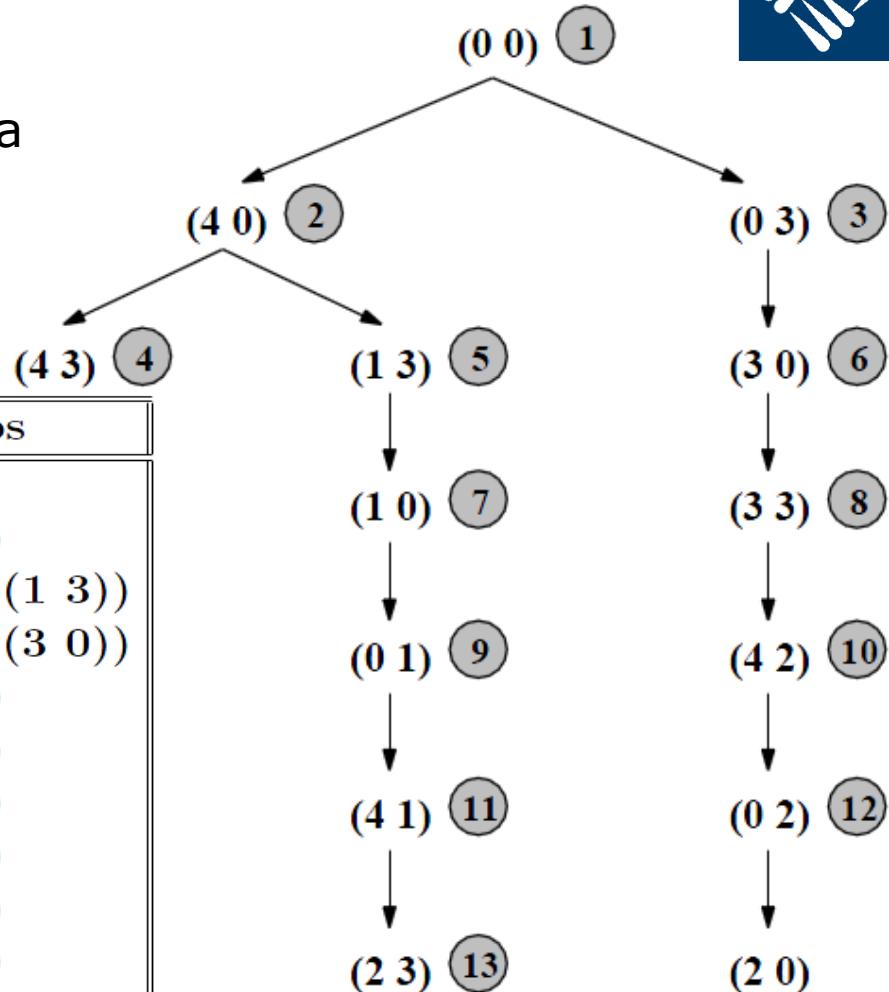
FIN DE BUCLE

Devuelve FALLO 😞



3.1 Búsqueda en anchura

Árbol y Tabla de búsqueda en anchura para el problema de las jarras:



Nodo	Actual	Sucesores	Abiertos
1	(0 0)	((4 0) (0 3))	((0 0)) ((4 0) (0 3))
2	(4 0)	((4 3) (1 3))	((0 3) (4 3) (1 3))
3	(0 3)	((3 0))	((4 3) (1 3) (3 0))
4	(4 3)	()	((1 3) (3 0))
5	(1 3)	((1 0))	((3 0) (1 0))
6	(3 0)	((3 3))	((1 0) (3 3))
7	(1 0)	((0 1))	((3 3) (0 1))
8	(3 3)	((4 2))	((0 1) (4 2))
9	(0 1)	((4 1))	((4 2) (4 1))
10	(4 2)	((0 2))	((4 1) (0 2))
11	(4 1)	((2 3))	((0 2) (2 3))
12	(0 2)	((2 0))	((2 3) (2 0))
13	(2 3)		



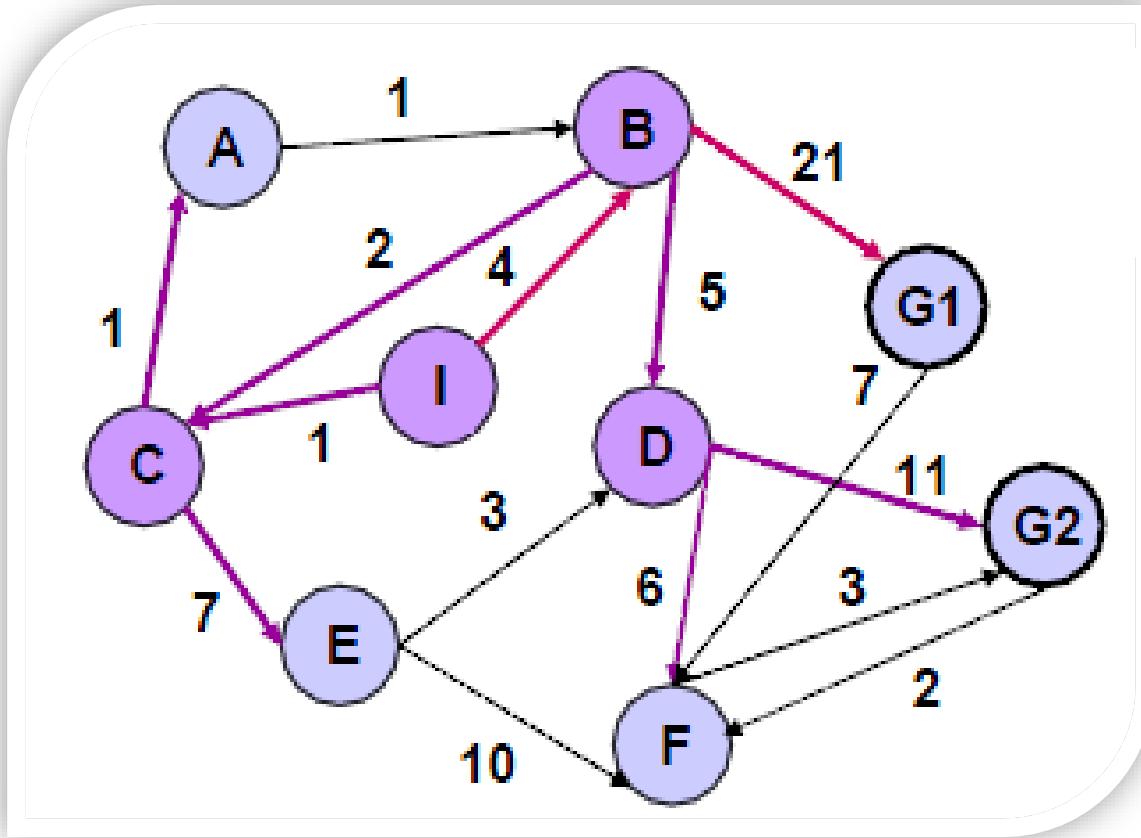
3.1 Búsqueda en anchura

- Estadísticas de casos conocidos

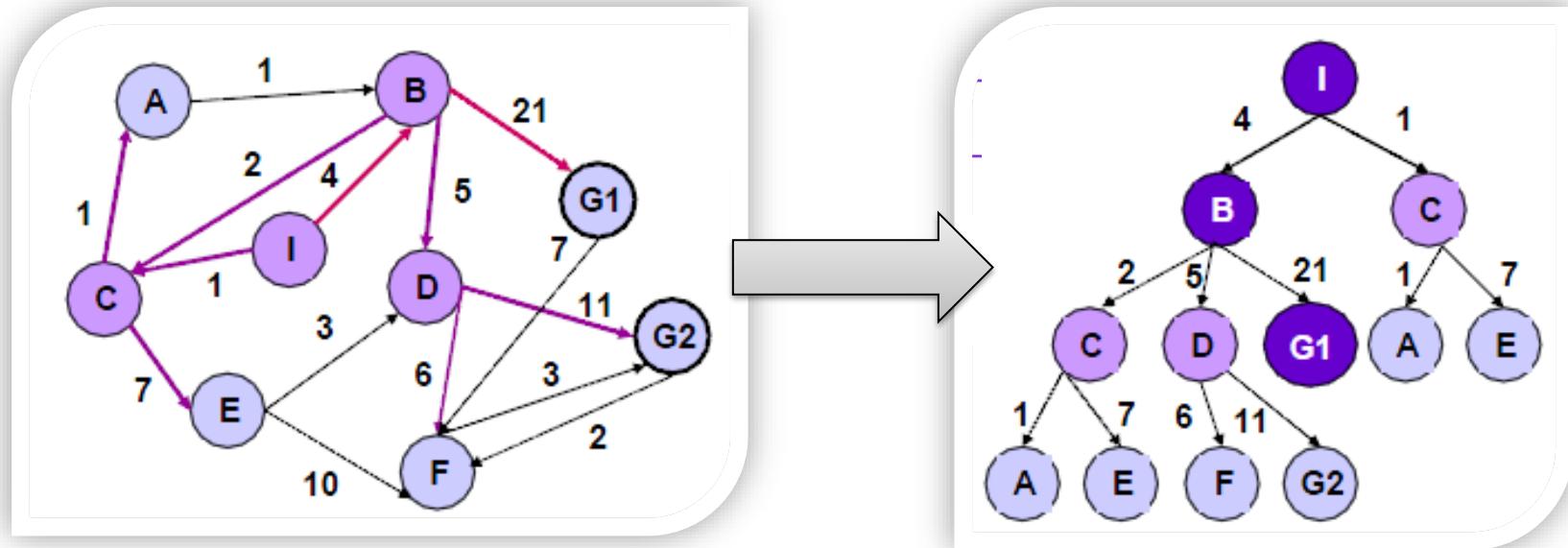
	Tiempo (seg.)	Espacio (bytes)	Nodos cerrados	Máximo en abierto	Max. Depth
Viaje	0,18	3.260	8	4	3
Granjero	0,18	3.432	10	2	7
Jarras	0,41	7.236	13	3	6
8-puzzle	4,51	68.292	46	22	5



3.1 Búsqueda en anchura



3.1 Búsqueda en anchura

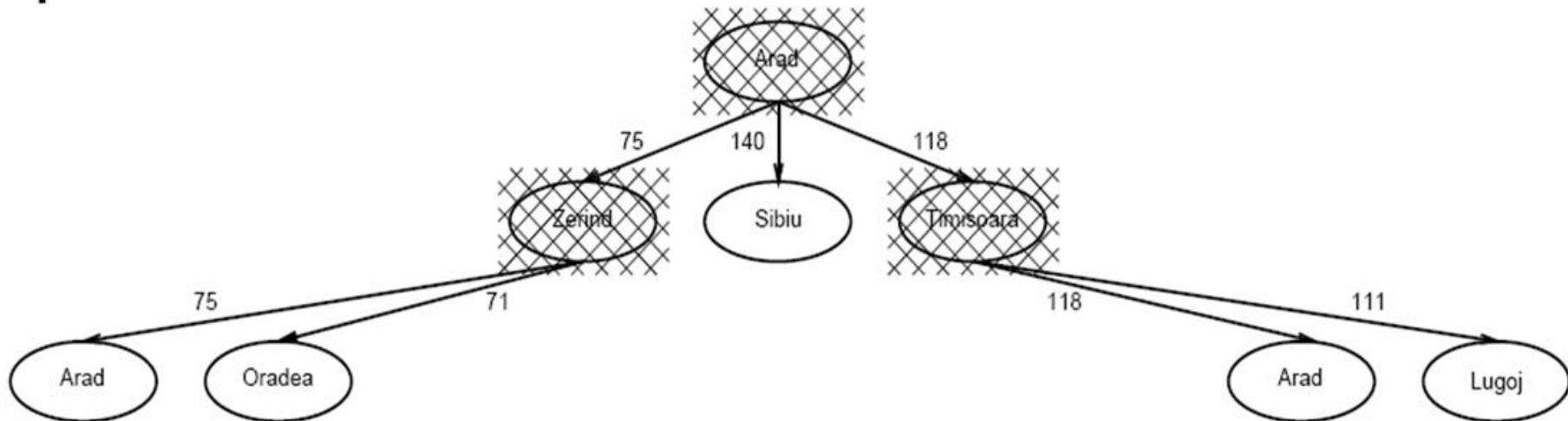


- Nodos cerrados (por orden): I B C C D G1
- Nodos abiertos (por orden): I B C C D G1 A E A E F G2
- Camino a la solución: I B G1
- Coste: $4+21 = 25$



3.2 Búsqueda de coste uniforme

- **Dijkstra, 1959**
- Parecida a la búsqueda en anchura, pero se expande el nodo de menor coste (camino de coste más pequeño)
- ABIERTO se implementa con una lista ordenada (cola de prioridad)
 - GESTIONA-COLA: Añadir ordenadamente a ABIERTO en orden creciente del coste del camino desde nodo inicial a cada nodo





3.2 Búsqueda de coste uniforme

- Propiedades:
 - Completa: **Si**, encuentra la solución de menor coste
 - Complejidad tiempo: $O(b^{C^*/\varepsilon})$ del peor caso
 - Complejidad espacio: $O(b^{C^*/\varepsilon})$ del peor caso
 - Óptima: **Si**, si se cumple que $g(\text{SUCESSOR}(N)) \geq g(N)$
 - C^* es el coste de la solución óptima
 - ε es el mínimo coste de acción
 - Problema si $\varepsilon=0$
- La búsqueda de coste uniforme no se preocupa por el número de pasos que tiene el camino si no por coste total del camino
 - Mejor que la búsqueda en anchura si los costes son muy diferentes y están bien estimados
 - Si todos los costes son iguales, idéntica a la búsqueda en anchura



3.2 Búsqueda de coste uniforme

PROCEDIMIENTO COSTE UNIFORME-DIJKSTRA(Estado-inicial, Estado-Final)

ABIERTO = *ESTADO-INICIAL*

Hacer *CERRADO* vacío

BUCLE (Repetir el proceso mientras ABIERTO ≠ vacío)

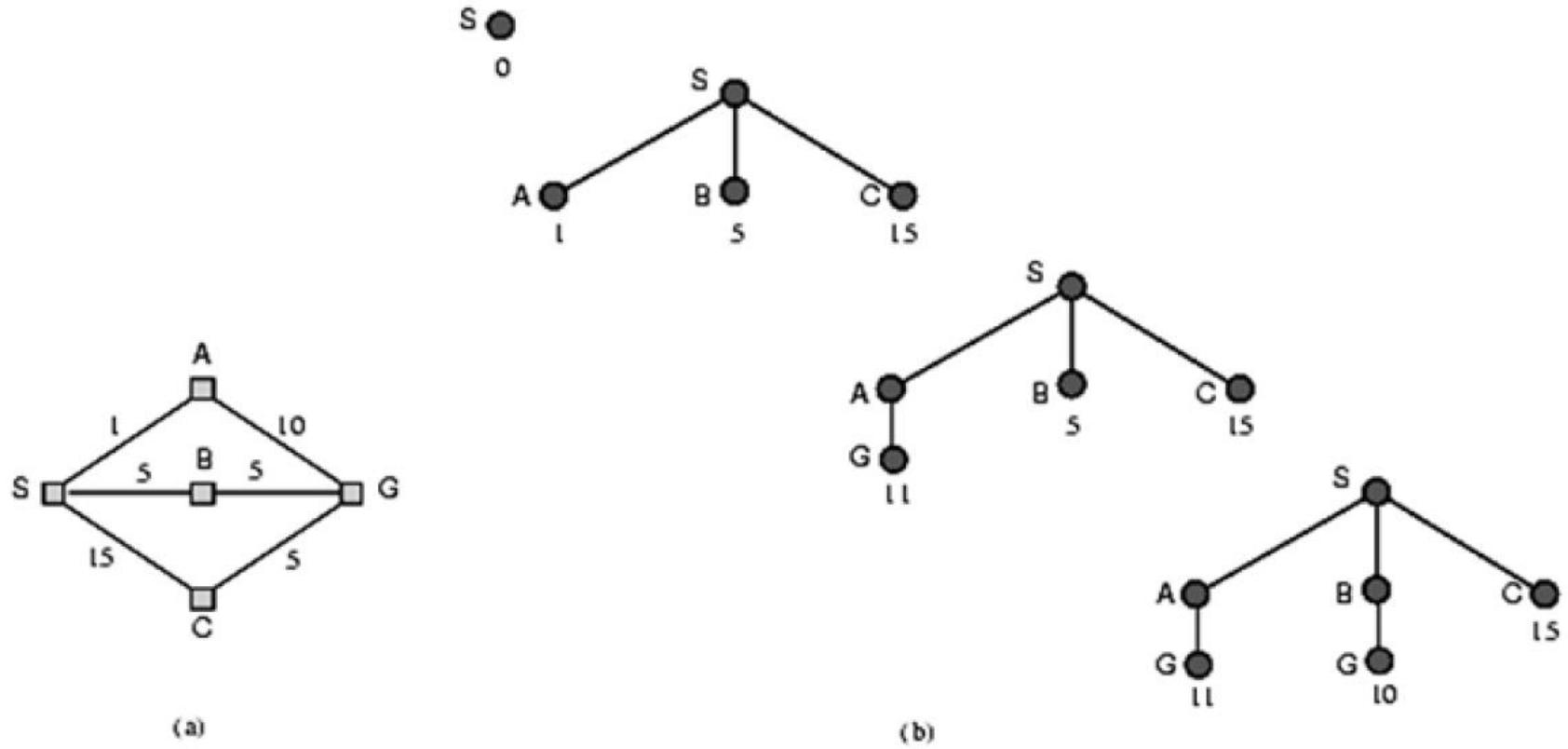
1. NODO-ACTUAL = EXTRAE-PRIMERO(ABIERTO)
2. Poner NODO-ACTUAL en *CERRADO*
3. Si ES-ESTADO-FINAL(ESTADO(NODO-ACTUAL))
devolver CAMINO(NODO-ACTUAL)
4. Si no, FUNCION SUCESORES(NODO-ACTUAL)
 5. Si SUCESOR ya está en *CERRADO*,
Si coste es menor, insertar ordenadamente en *ABIERTO*
Actualizar coste y camino
 6. Si SUCESOR ya está en *ABIERTO*,
Si coste es menor, actualizar coste, posición y camino
 7. GESTIONA-COLA(ABIERTO, SUCESORES) Añadir *SUCESORES* a *ABIERTO*
en orden creciente de g(n)

FIN DE BUCLE

Devuelve FALLO ☹

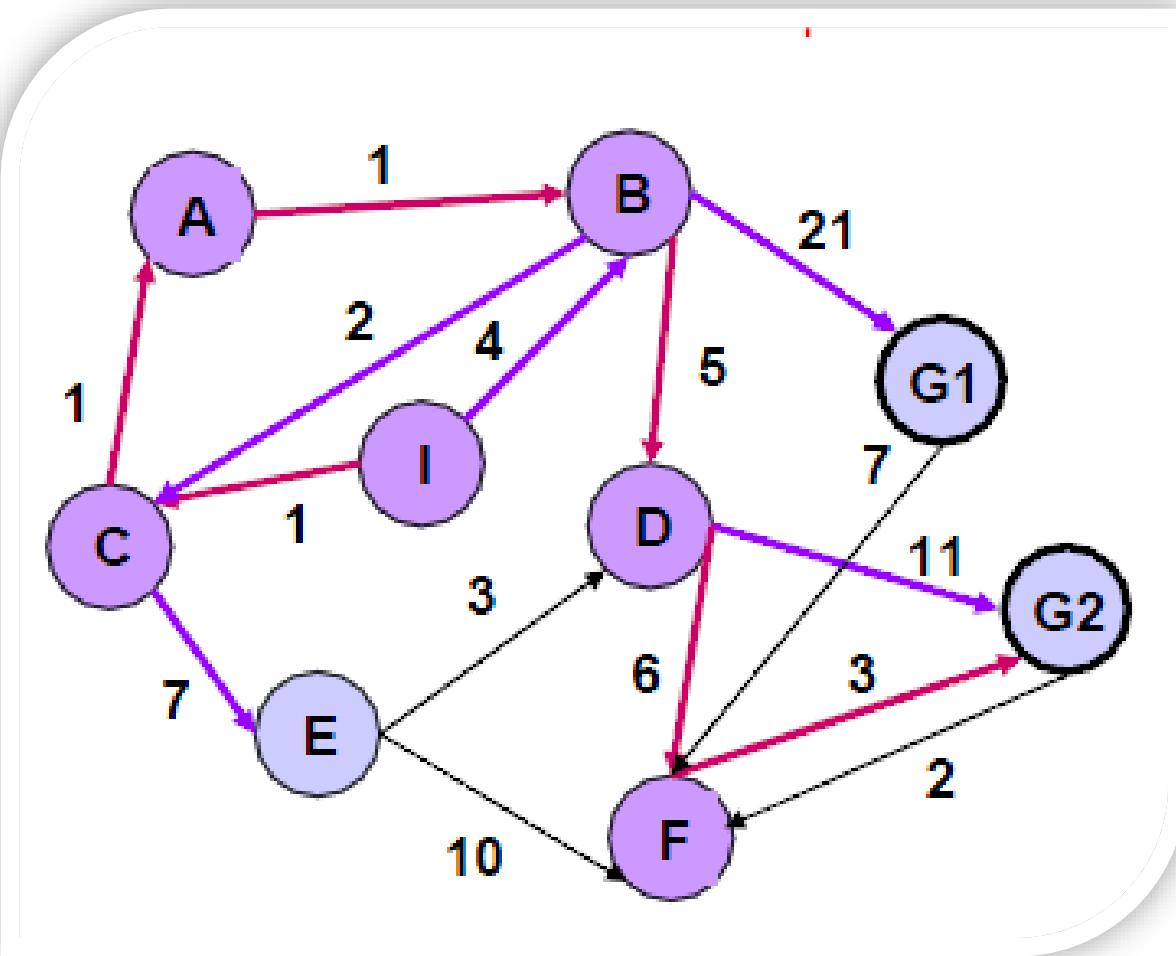


3.2 Búsqueda de coste uniforme





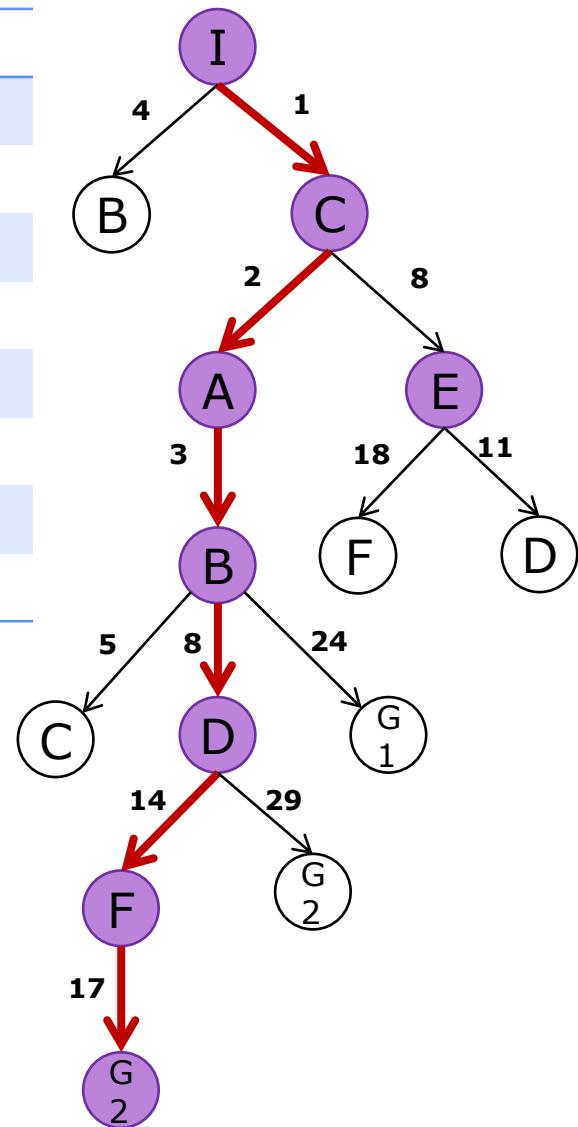
3.2 Búsqueda de coste uniforme





3.2 Búsqueda de coste uniforme

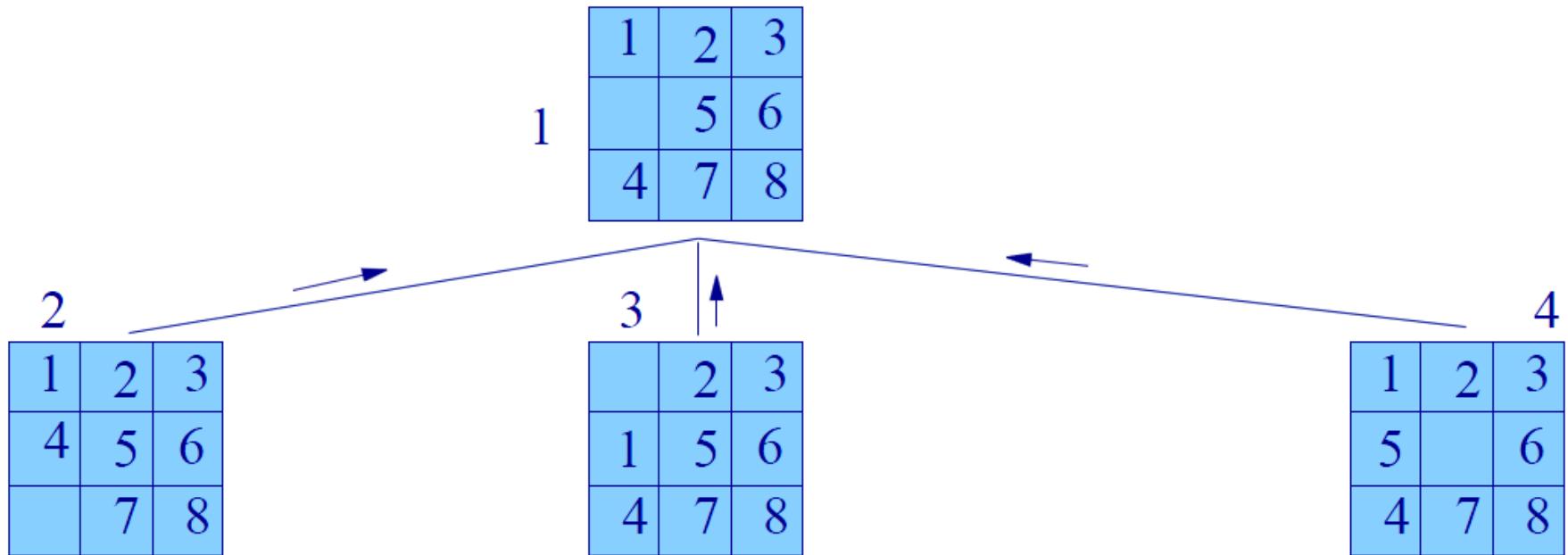
Nodo	Actual	Sucesores	Abiertos (cola de prioridad)
		I	
1	I	B(4), C(1)	C(1) B(4)
2	C	A(2), E(8)	A(2) B(4) E(8)
3	A	B(3)	B(3) B(4) E(8)
4	B	C(5), D(8), G1(24)	B(4) C(5) D(8) E(8) G1(24)
5	B C D	F(14), G2(19)	E(8) F(14) G2(19) G1(24)
6	E	F(18), D(11)	D(11) F(14) F(18) G2(19) G1(24)
7	D F	G2(17)	



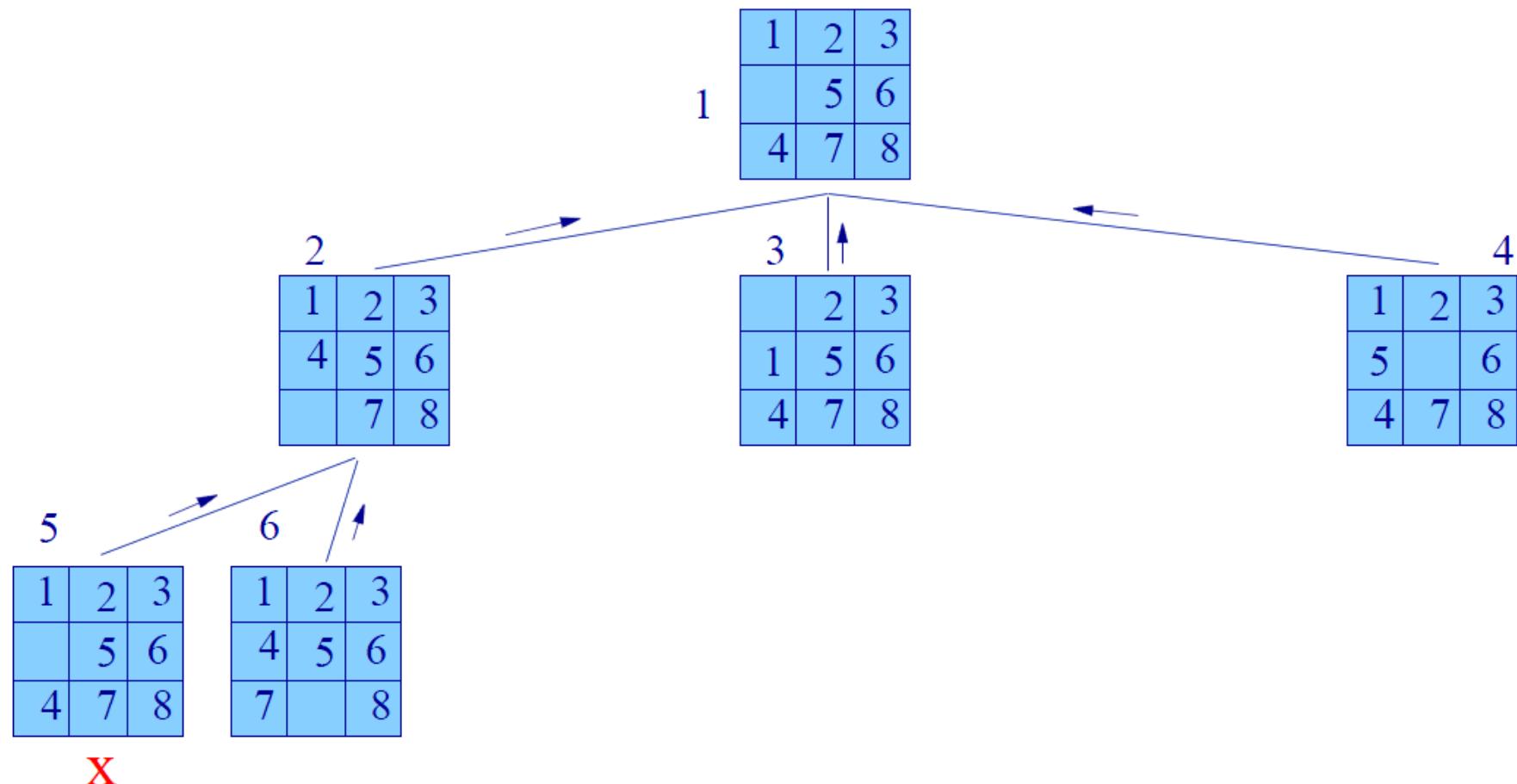
- Camino a la solución: I C A B D F G2
- Coste: $1+1+1+5+6+3 = 17$



3.3 Búsqueda en profundidad

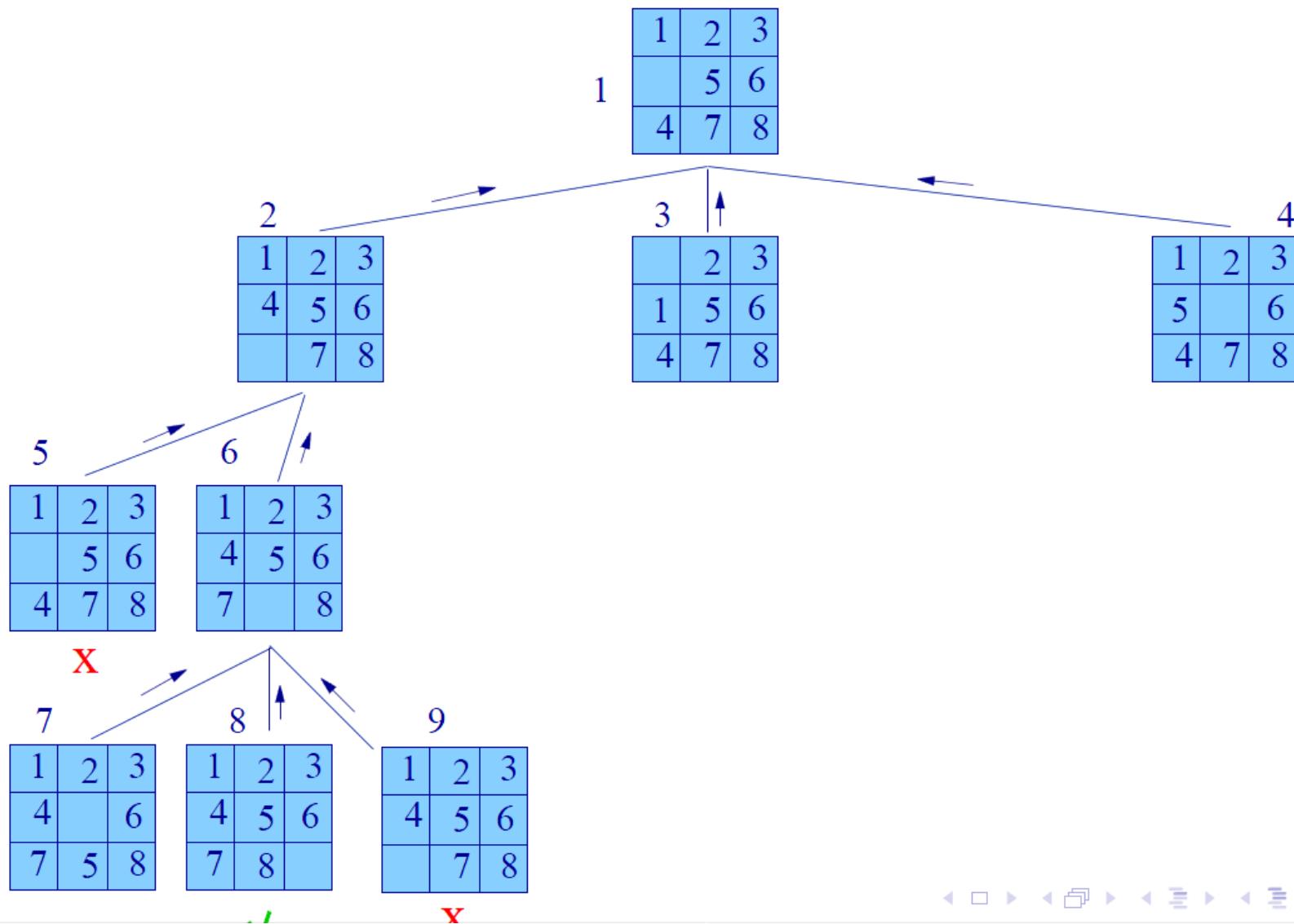


3.3 Búsqueda en profundidad





3.3 Búsqueda en profundidad



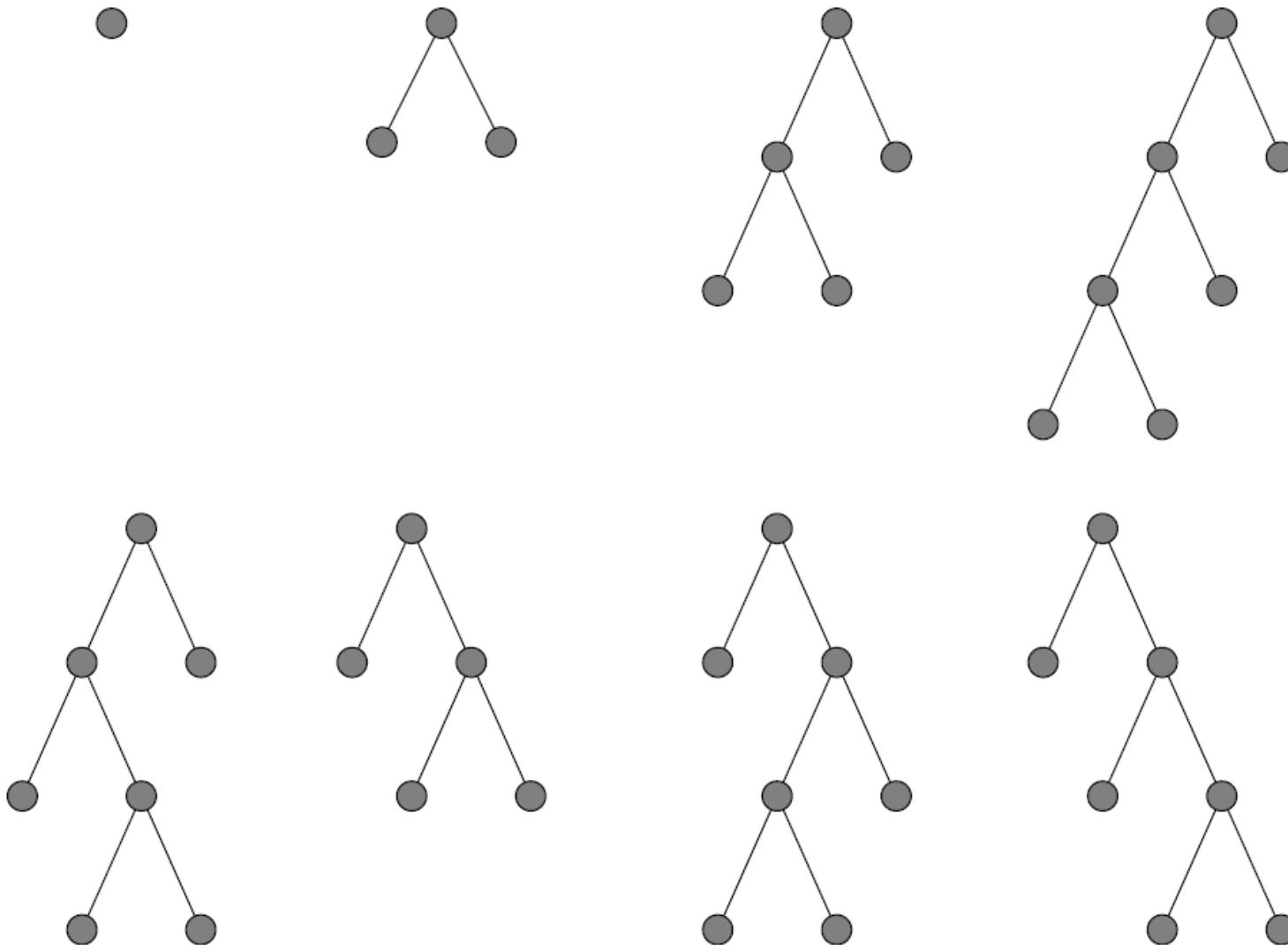
3.3 Búsqueda en profundidad



- Los sucesores van por delante. Se expanden antes los nodos no expandidos más profundos hasta el final
 - Se expande primero el último nodo que se insertó visitando un sucesor del nodo actual en cada paso
- ABIERTO se implementa con una pila LIFO
 - GESTIONA-COLA: Añadir al principio de ABIERTO
- Propiedades
 - **Completa:** **No**, falla en espacios de profundidad ∞ y en espacios cíclicos.
Si se evitan estados repetidos → completa en espacios finitos
 - **Complejidad tiempo:** $O(b^m)$ pero para espacios densos es más rápido que la de en anchura primero
 - m es la máxima profundidad del árbol de búsqueda: $\max(d)$
 - **Complejidad espacio:** $O(b*m+1)$ lineal, barata en espacio
 - **Óptima:** **No**



3.3 Búsqueda en profundidad



3.3 Búsqueda en profundidad



- Resumen
 - Eficiencia: bueno con metas alejadas de estado inicial o problemas de memoria
 - Problema: No es bueno cuando hay ciclos
- Características
 - Requiere técnica de retroceso ("*backtracking*")
 - Razones para retroceso:
 - Se ha llegado al límite de profundidad
 - Se han estudiado todos los sucesores de un nodo y no se ha llegado a la solución
 - Se sabe que el estado no conduce a la solución
 - Se genera un estado repetido
- Puede caer en bucles infinitos. Precisa de un espacio finito no cíclico de búsqueda (o una función de testeado de estados repetidos).



3.3 Búsqueda en profundidad

PROCEDIMIENTO PROFUNDIDAD(Estado-inicial, Estado-Final, DEPTH-MAX)

ABIERTO = *ESTADO-INICIAL*

Hacer *CERRADO* vacío

BUCLE (Repetir el proceso mientras ABIERTO ≠ vacío)

1. NODO-ACTUAL = EXTRAE-PRIMERO(ABIERTO)
2. Poner NODO-ACTUAL en *CERRADO*
3. Si ES-ESTADO-FINAL(ESTADO(NODO-ACTUAL))
devolver CAMINO(NODO-ACTUAL)
4. Si no,
 - 4.1 Generar SUCESORES(NODO-ACTUAL) que no están ni en ABIERTO ni en CERRADOS
 - 4.2 GESTIONA-COLA(ABIERTO, SUCESORES)
Añadir *SUCESORES* al comienzo de ABIERTO (*pila LIFO*)

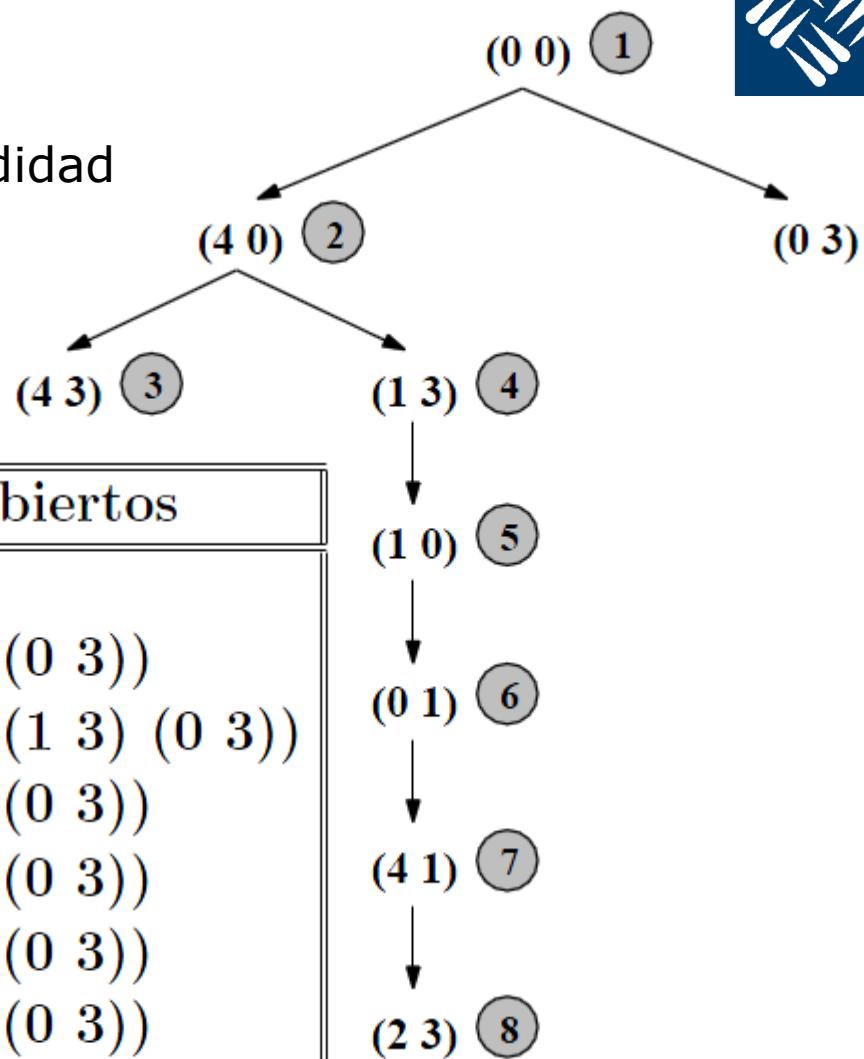
FIN DE BUCLE

Devuelve FALLO ☹

3.3 Búsqueda en profundidad

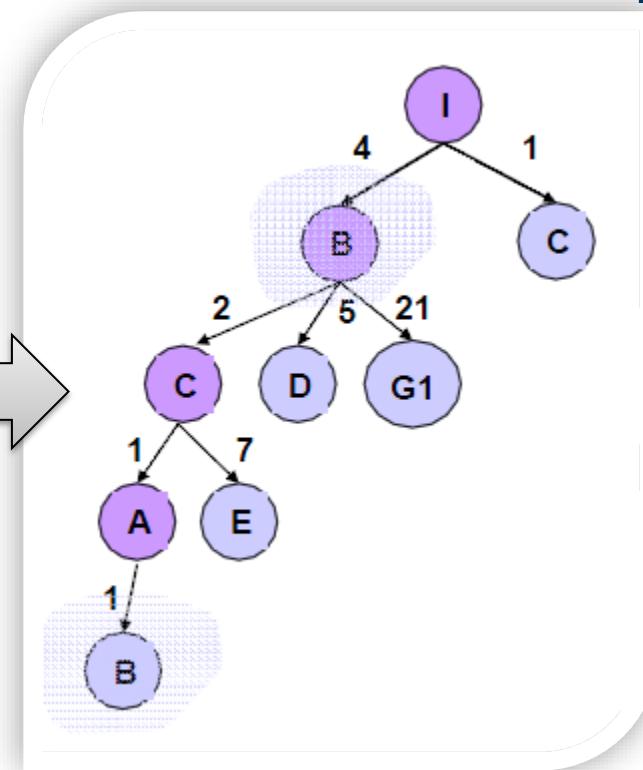
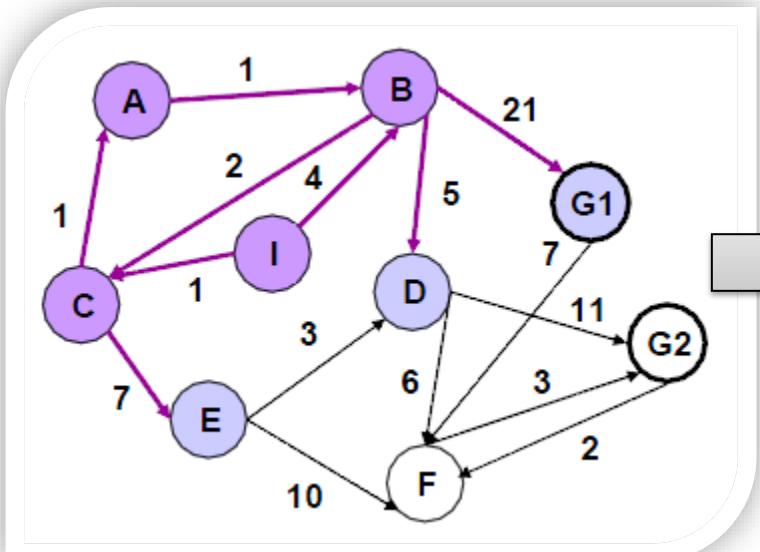


Árbol y Tabla de búsqueda en profundidad para el problema de las jarras:



Nodo	Actual	Sucesores	Abiertos
1	(0 0)	((4 0) (0 3))	((0 0))
2	(4 0)	((4 3) (1 3))	((4 0) (0 3))
3	(4 3)	()	((4 3) (1 3) (0 3))
4	(1 3)	((1 0))	((1 3) (0 3))
5	(1 0)	((0 1))	((1 0) (0 3))
6	(0 1)	((4 1))	((0 1) (0 3))
7	(4 1)	((2 3))	((4 1) (0 3))
8	(2 3)		((2 3) (0 3))

3.3 Búsqueda en profundidad



- Sin control de ciclos no encuentra solución (rama infinita)
- Nodos abiertos: B E D G1 C
- Nodos expandidos: I B C A



3.3 Búsqueda en profundidad

- Cuando el espacio de estados tiene ciclos es necesario tener cuidado con ellos
 - por eficiencia
 - por terminación
- "Los algoritmos que olvidan su historia están condenados a repetirla"*
- El control de ciclos empeora la complejidad de los algoritmos utilizados
 - Detección de ciclos en la búsqueda:
 - En algunos espacios de estados no hay posibilidad de caminos cílicos y no se necesita CERRADO para detectar ciclos
 - Para detectar ciclos solo es necesario almacenar los nodos de la rama que se está explorando en cada momento



3.3 Búsqueda en profundidad

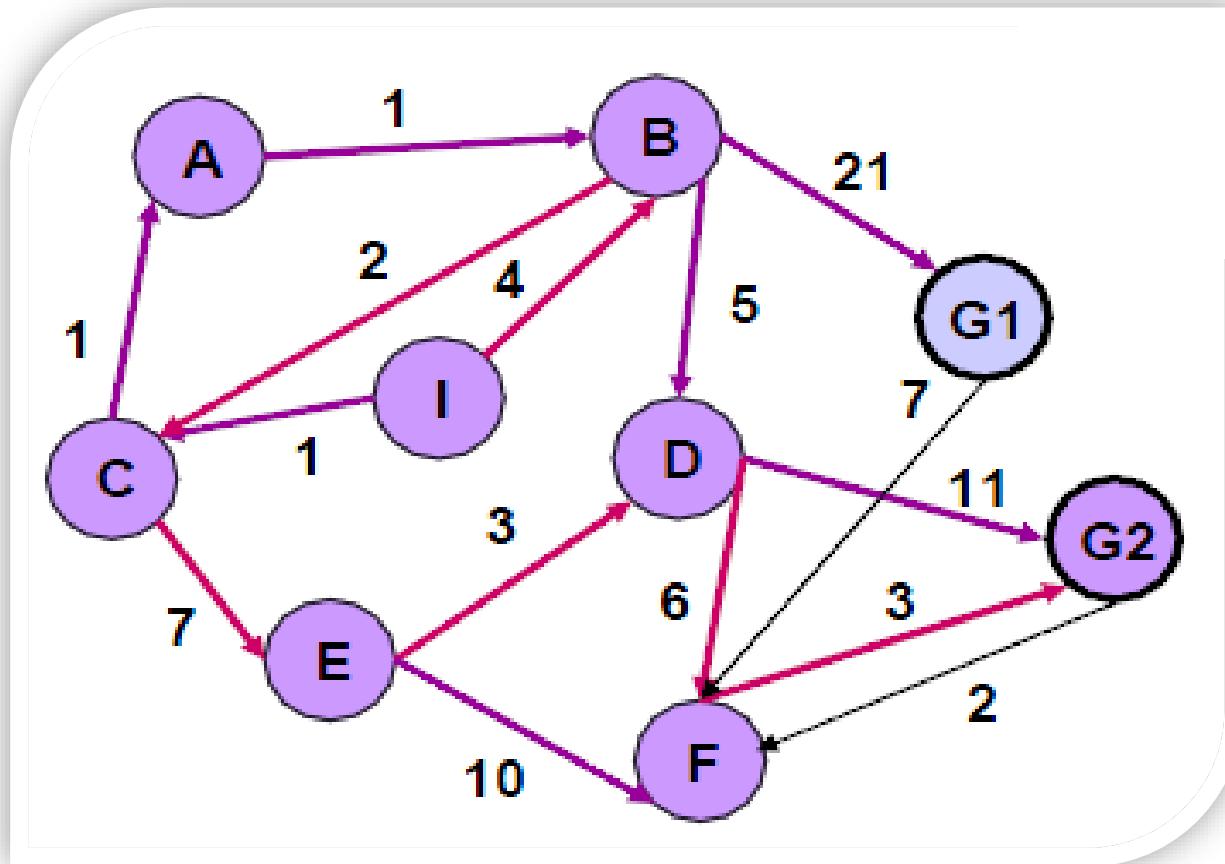
- Mecanismos de control de ciclos
 - Mirar los nodos del **camino actual** es lo más sencillo y lo menos costoso (cada nodo tiene acceso a su padre)
 - Mirar los **nodos de ABIERTO** (generados sin expandir)
 - Mirar los **nodos de CERRADO** (estados ya expandidos)
- Un control completo se consigue combinando las 2 últimas comprobaciones
 - Puede suponer una sobrecarga inaceptable si no es necesario
 - Hay que implementar algoritmos de búsqueda en **ABIERTO** y **CERRADO**
 - Gestión de **CERRADO** (siempre aumenta; nunca se eliminan elementos)



3.3 Búsqueda en profundidad

Búsqueda en profundidad + control de ciclos:

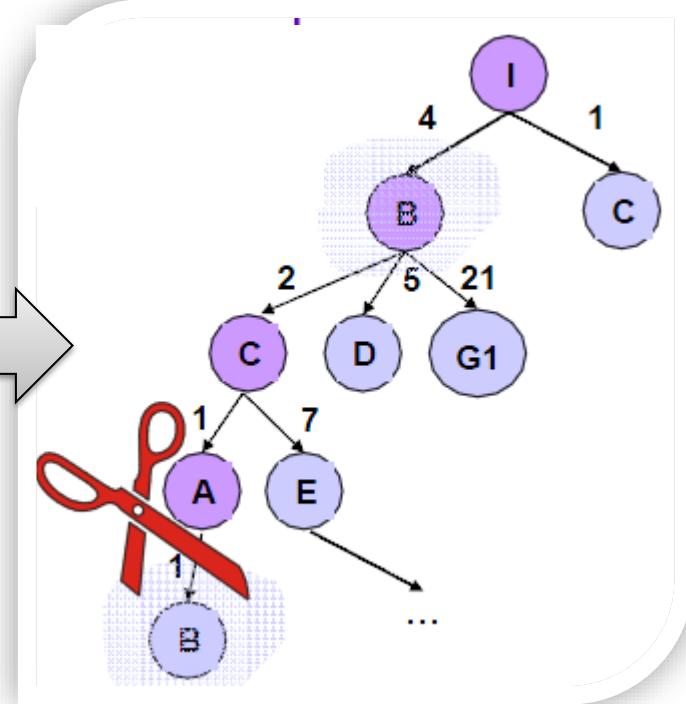
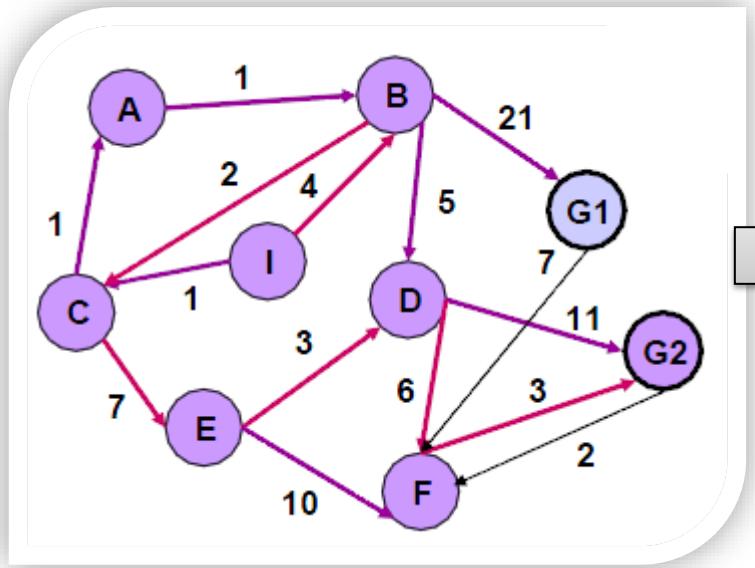
Mirar en la rama actual





3.3 Búsqueda en profundidad

Búsqueda en profundidad + control de ciclos



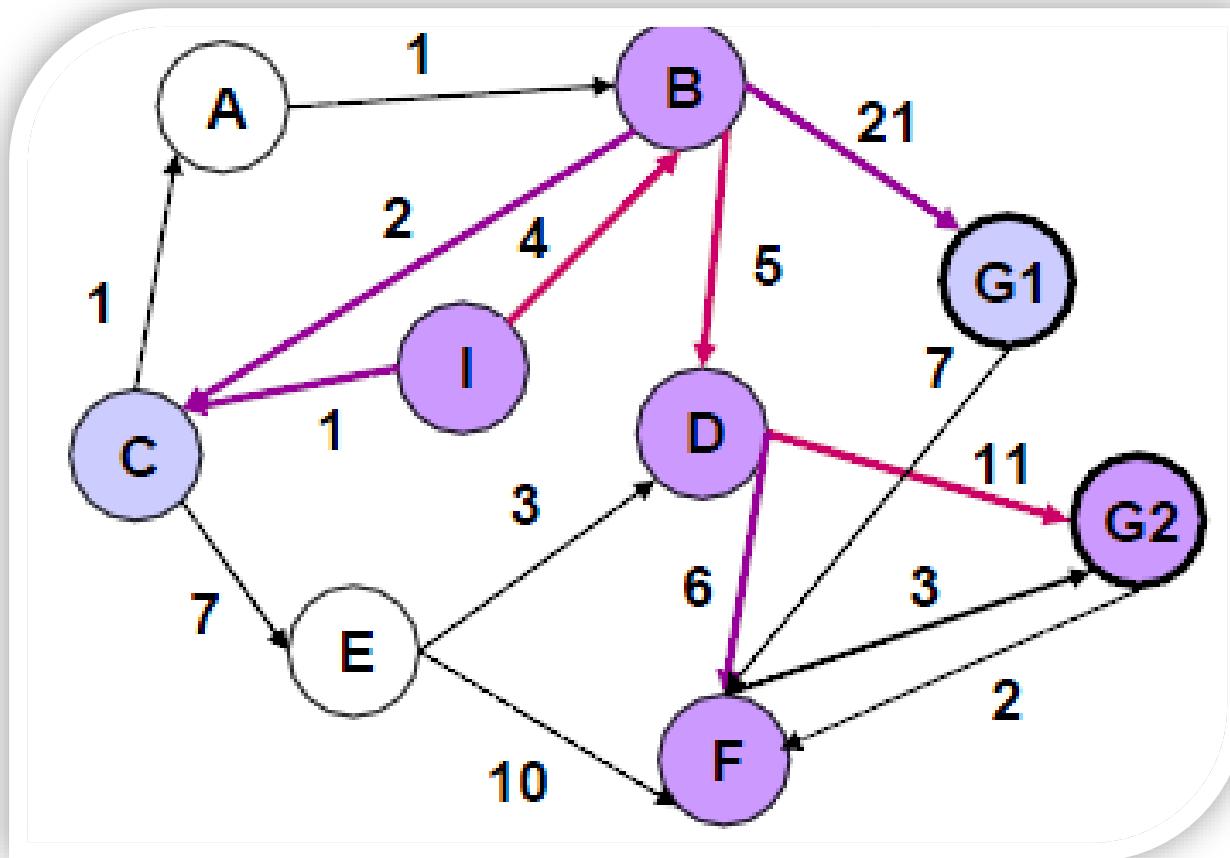
Mirar en la rama actual

- Nodos cerrados: I B C A E D F G2
- Camino a la solución: I B C E D F G2
- Coste: $4+2+7+3+6+3 = 25$



3.3 Búsqueda en profundidad

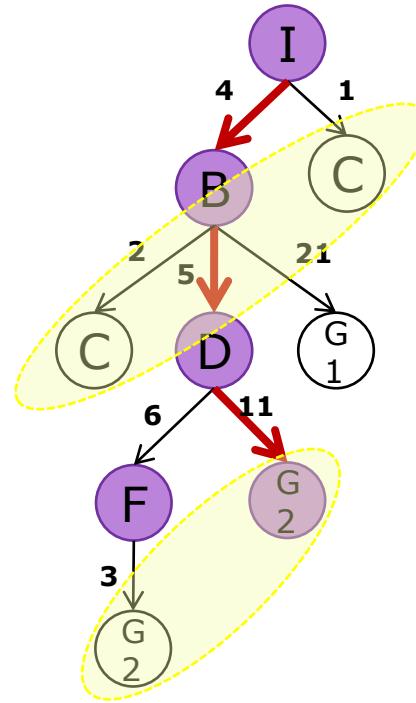
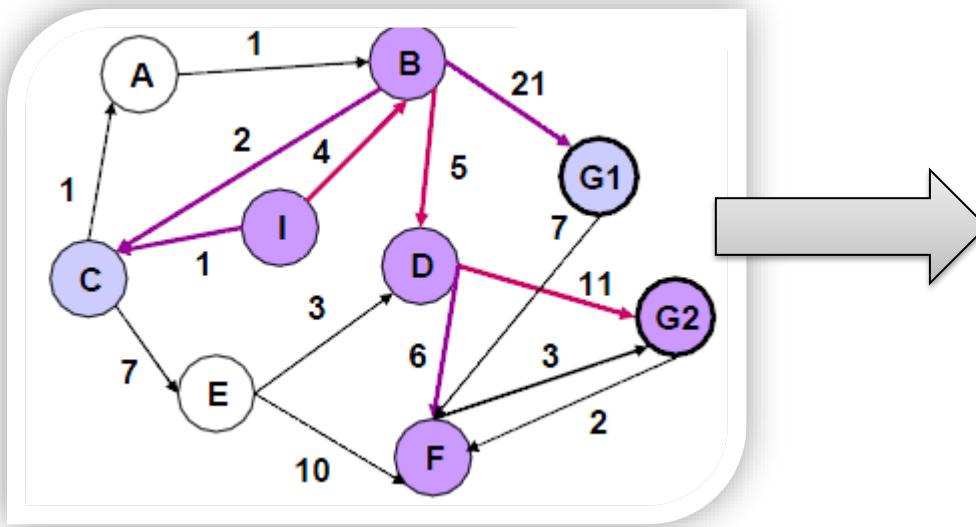
Búsqueda en profundidad + control de ciclos:
Mirar en los nodos abiertos + cerrados





3.3 Búsqueda en profundidad

Búsqueda en profundidad + control de ciclos



Mirar en los nodos abiertos + cerrados

- Nodos abiertos (por orden): I B C D G1 F G2
- Nodos cerrados (por orden): I B D F G2
- Camino a la solución: I B D G2
- Coste: $4+5+11 = 20$

3.4 Búsqueda en profundidad limitada



- Como la **búsqueda en profundidad**, pero se fija un *límite ℓ* de profundidad en la búsqueda para evitar descender indefinidamente por el mismo camino
 - El límite permite desechar caminos en los que se supone que no encontraremos un nodo objetivo lo suficientemente cercano al nodo inicial
- ABIERTO se implementa con una pila (LIFO)
 - GESTIONA-COLA: Añadir al principio de ABIERTO excepto que los nodos de profundidad ℓ no tienen sucesores
- Se expande primero el último nodo que se insertó hasta una profundidad ℓ
 - Si $\ell=\infty$ es idéntica a la búsqueda en profundidad
 - A veces se conoce el “diámetro” del espacio de estados a priori



3.4 Búsqueda en profundidad limitada

- Propiedades:
 - **Completa:** **Si, si $l \geq d$** (profundidad mínima de “la solución”)
 - Si d es desconocido, la elección de l es una incógnita
 - Hay problemas en los que este dato (diámetro del espacio de estados) es conocido de antemano, pero en general no se conoce a priori
 - **Complejidad tiempo:** $O(b^l)$
 - **Complejidad espacio:** $O(b^l)$
 - **Optima:** **No.** No puede garantizarse que la primera solución encontrada sea la mejor



3.4 Búsqueda en profundidad limitada

PROCEDIMIENTO PROFUNDIDAD_LIMITADA(Estado-Inicial, Estado-Final, LIMITE)

ABIERTO = *ESTADO-INICIAL*

Hacer *CERRADO* vacío

BUCLE (Repetir el proceso mientras ABIERTO ≠ vacío)

1. NODO-ACTUAL = EXTRAE-PRIMERO(ABIERTO)
2. Poner NODO-ACTUAL en *CERRADO*
3. Si ES-ESTADO-FINAL(ESTADO(NODO-ACTUAL))
devolver CAMINO(NODO-ACTUAL)
4. Si no,
 Si DEPTH(NODO-ACTUAL) <= LIMITE entonces
 4.1 Generar SUCESORES(NODO-ACTUAL) que no están ni en ABIERTO
 ni en CERRADOS
 4.2 GESTIONA-COLA(ABIERTO,SUCESORES)
 Añadir *SUCESORES* al comienzo de ABIERTO (*pila LIFO*)
 Asignarles DEPTH = DEPTH(NODO-ACTUAL) + 1

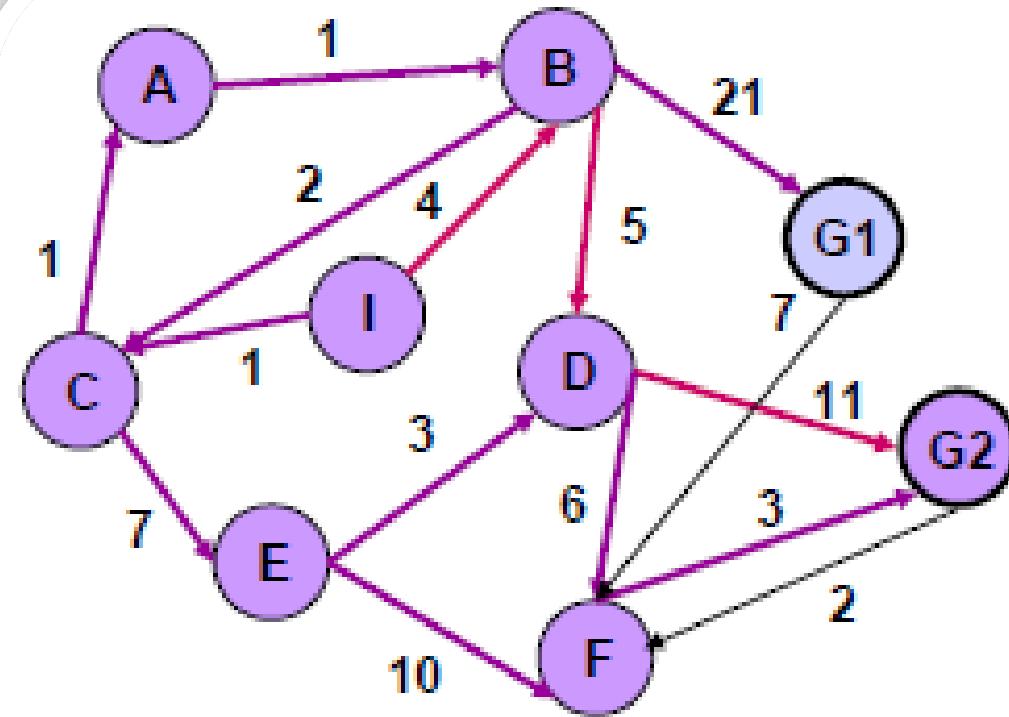
FIN DE BUCLE

Devuelve FALLO ☹

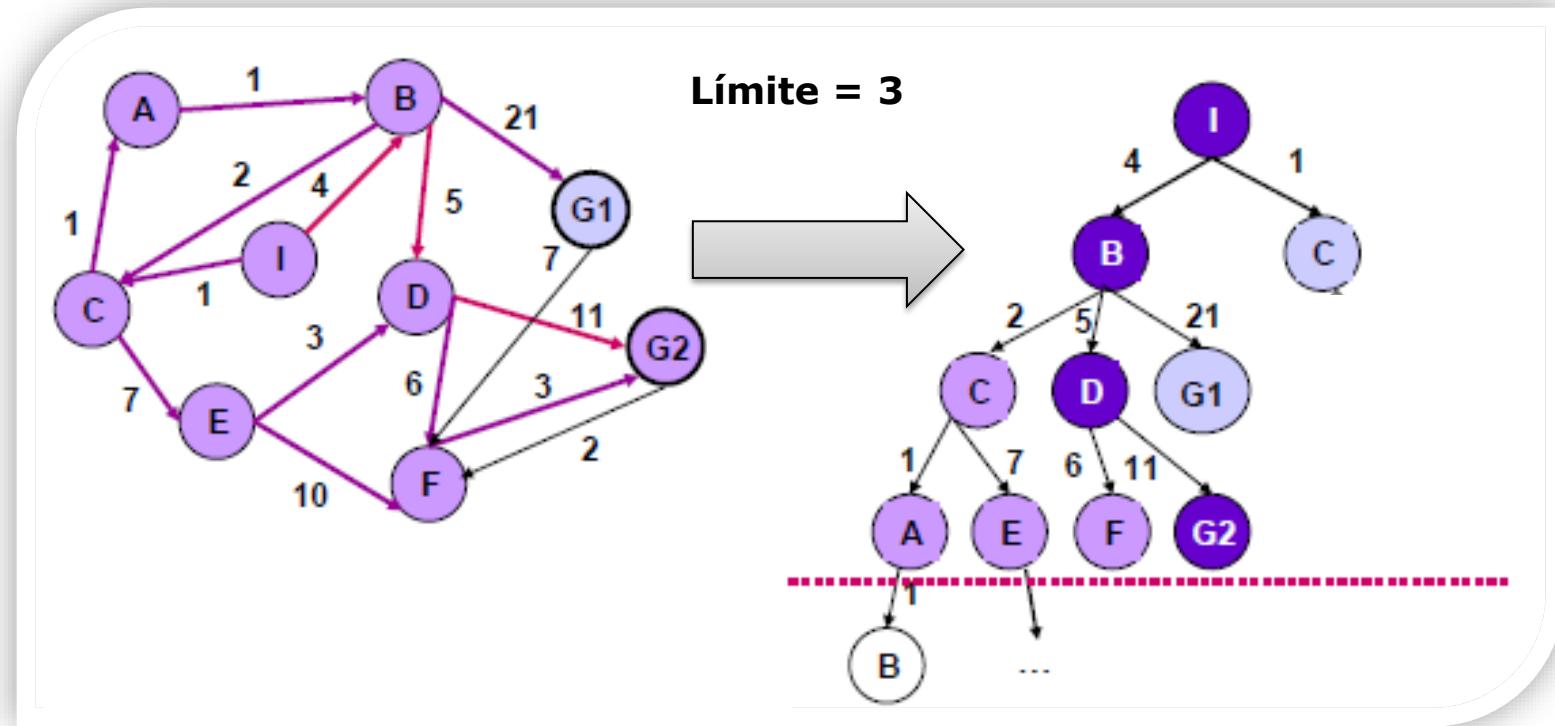


3.4 Búsqueda en profundidad limitada

Límite = 3



3.4 Búsqueda en profundidad limitada



- Nodos cerrados: I B C A E D F G2
- Camino a la solución: I B D G2
- Coste: $4+5+11 = 20$

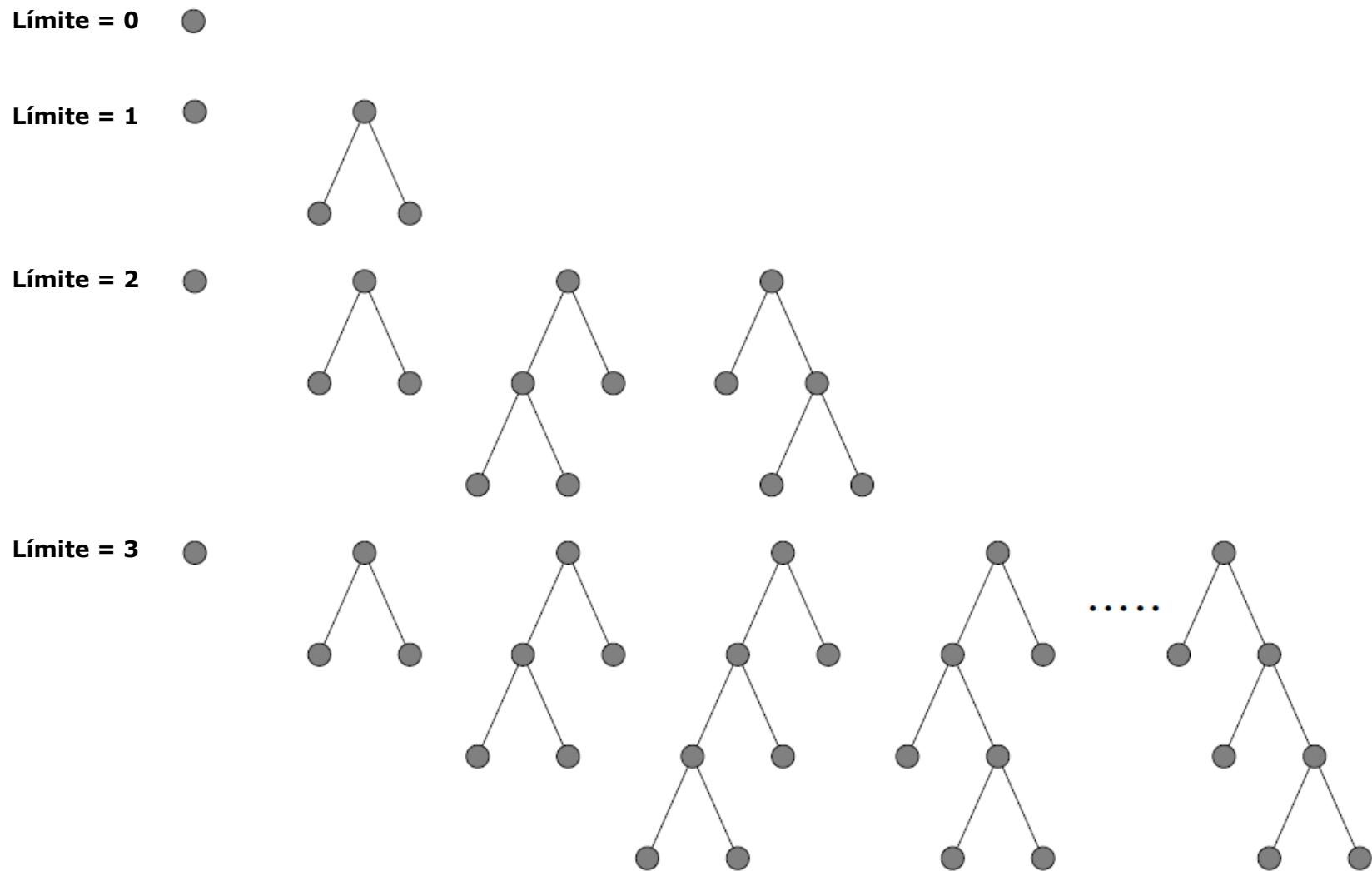
3.5 Búsqueda en profundidad iterativa



- Aplicación **iterativa** del algoritmo de **búsqueda en profundidad limitada**: límite de profundidad varia de forma creciente primero 1, luego 2, ...
- Combina las ventajas de los algoritmos primero en profundidad y anchura
 - como la búsqueda en anchura, es completa
 - como la búsqueda primero en profundidad, requiere poca memoria
- Algoritmo
 1. Se fija profundidad máxima d_{max}
 2. Se busca en profundidad primero
 3. Si no se encuentra solución, se hace $d_{max} = d_{max} + k$
 4. Se vuelve al paso 2



3.5 Búsqueda en profundidad iterativa





3.5 Búsqueda en profundidad iterativa

- Propiedades
 - Completa: **Si**, encuentra la solución, si ésta existe
 - Complejidad tiempo: $O(b^d)$.
 - Complejidad espacio: $O(b*d+1)$.
 - Óptima: **Si**, encuentra la solución óptima si los costes son uniformes y el incremento de profundidad $k = 1$
- Problema: puede generar muchos nodos duplicados pero aunque repite la expansión de los nodos cercanos a la raíz, su número habitualmente no es muy grande
- *Método de búsqueda no informada preferido cuando hay un espacio grande de búsqueda y no se conoce la profundidad de la solución*



3.5 Búsqueda en profundidad iterativa

- Ejemplo, $b = 10$ y $d = 5$:
 - Búsqueda acotada, nodos analizados:
$$1 + 10 + 100 + 1000 + 10000 + 100000 = 111111$$
 - Búsqueda iterativa, nodos analizados:
$$1 + 11 + 111 + 1111 + 11111 + 111111 = 123456$$
- Tan solo un 10% más. Razón:
La mayoría de los nodos están en el último nivel del árbol

3.5 Búsqueda en profundidad iterativa



PROCEDIMIENTO PROFUNDIDAD_ITERATIVA(Estado-inicial, Estado-Final, COTA-INICIAL)

Hacer N = COTA-INICIAL

BUCLE (Repetir el proceso mientras $N < \text{PROFUNDIDAD-MAXIMA}$ si la hay)

Si PROFUNDIDAD_LIMITADA(N) ≠ FALLO

devolver CAMINO(NODO-ACTUAL)

Si no hacer N = N+1

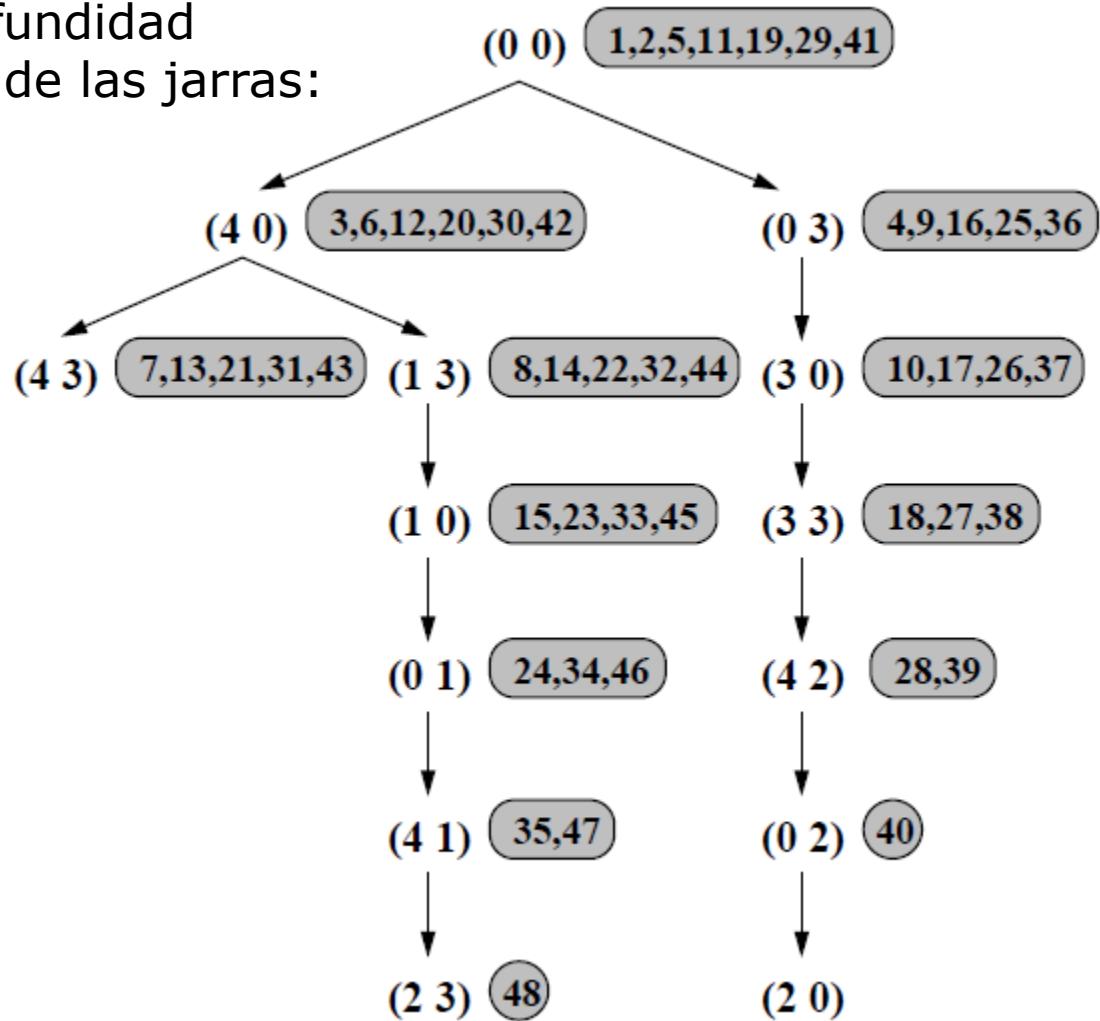
FIN DE BUCLE

Devuelve FALLO ☹



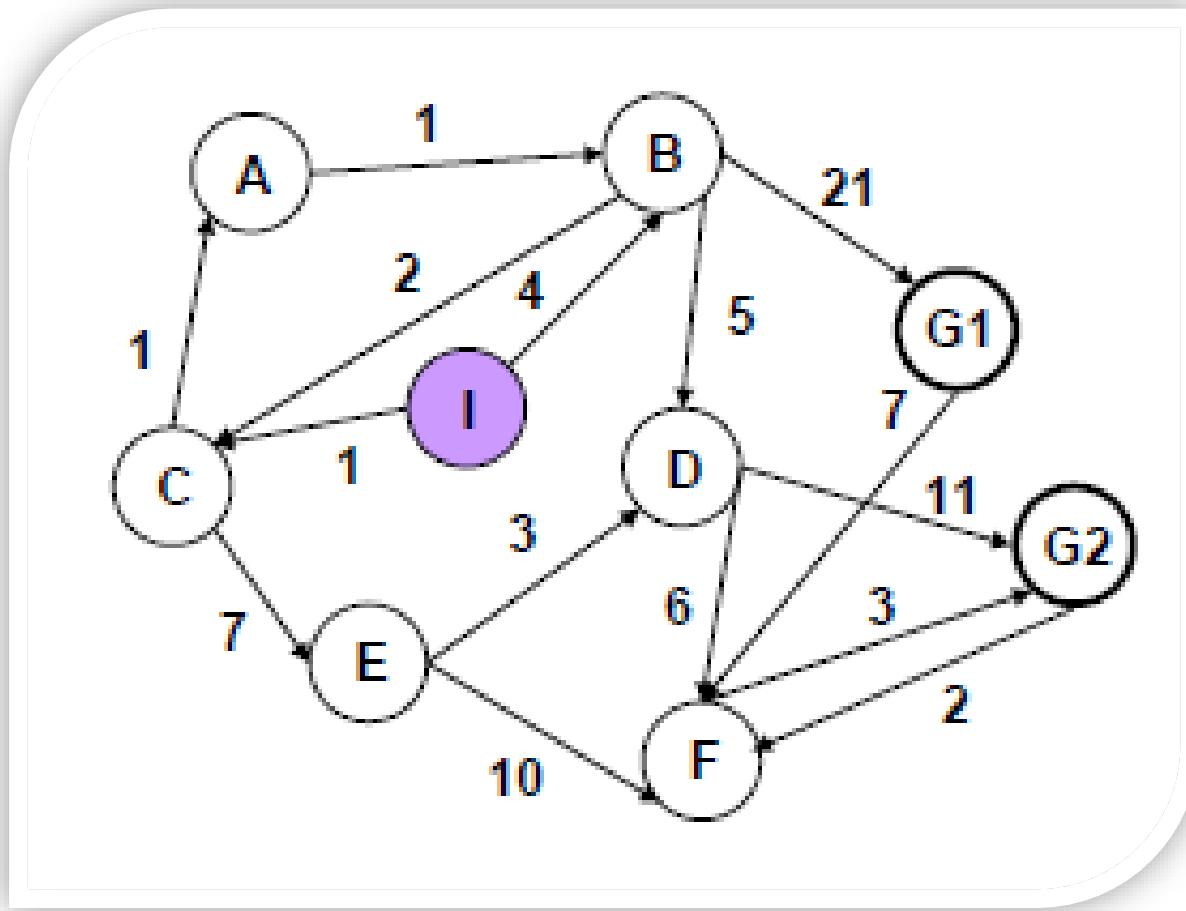
3.5 Búsqueda en profundidad iterativa

Árbol de búsqueda en profundidad iterativa para el problema de las jarras:

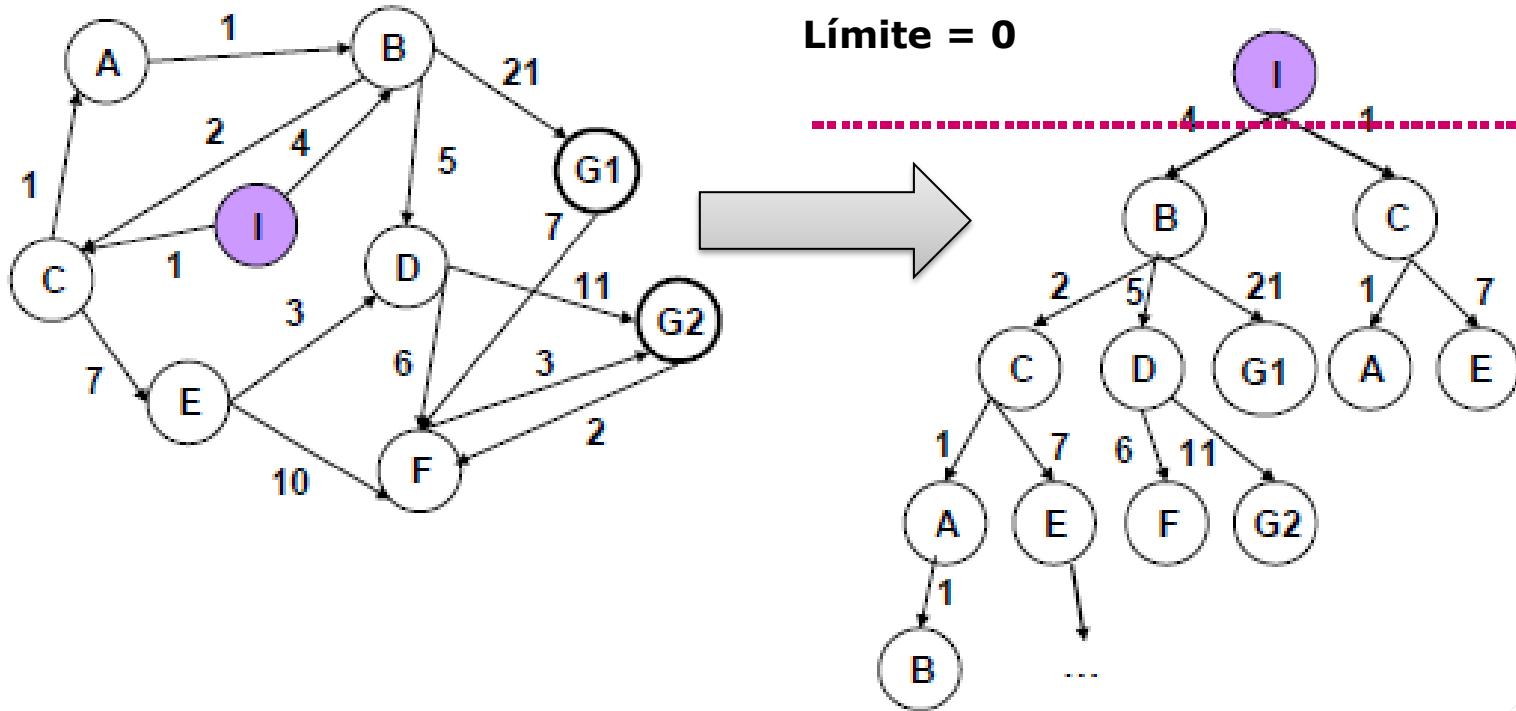




3.5 Búsqueda en profundidad iterativa

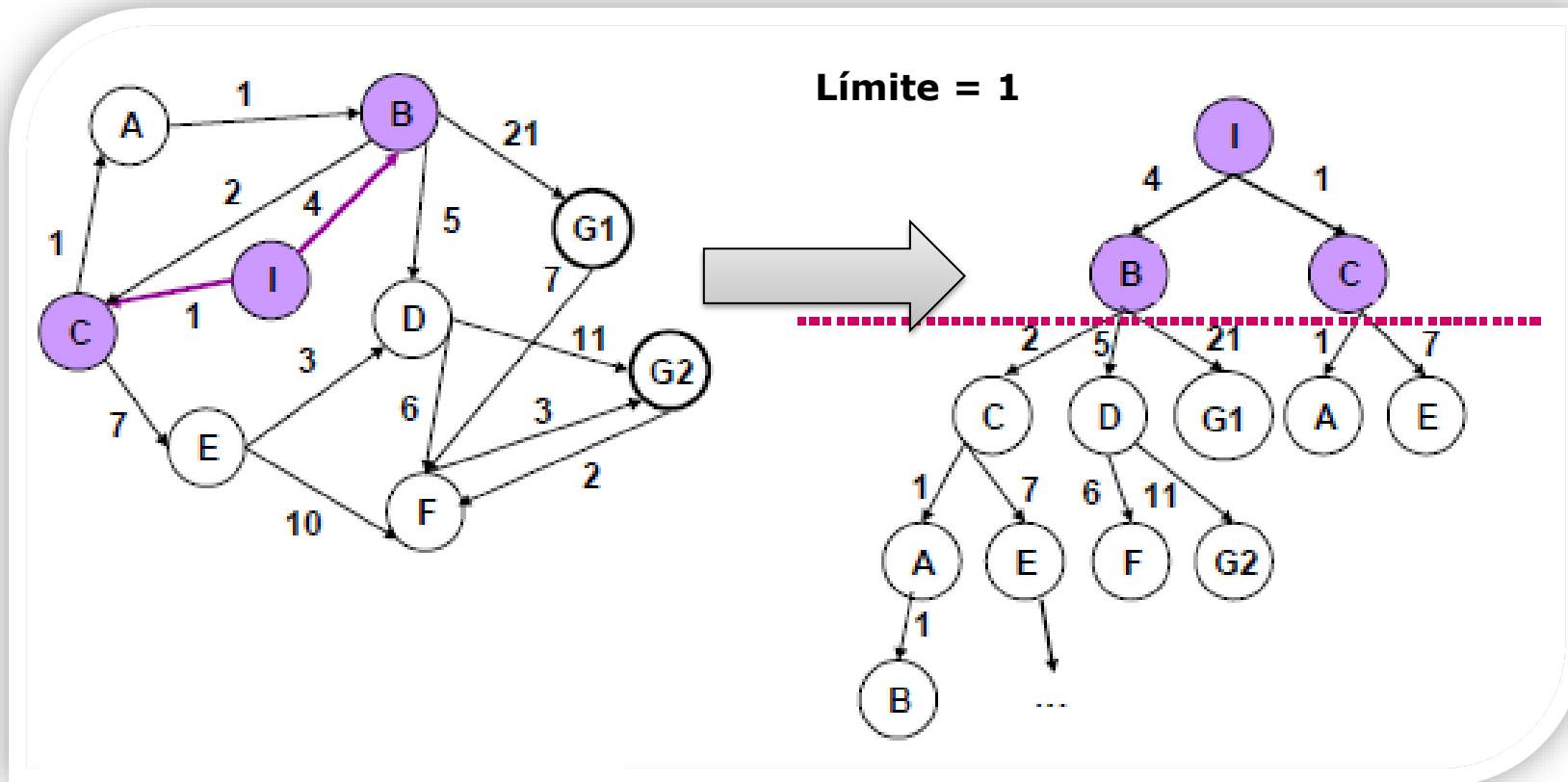


3.5 Búsqueda en profundidad iterativa



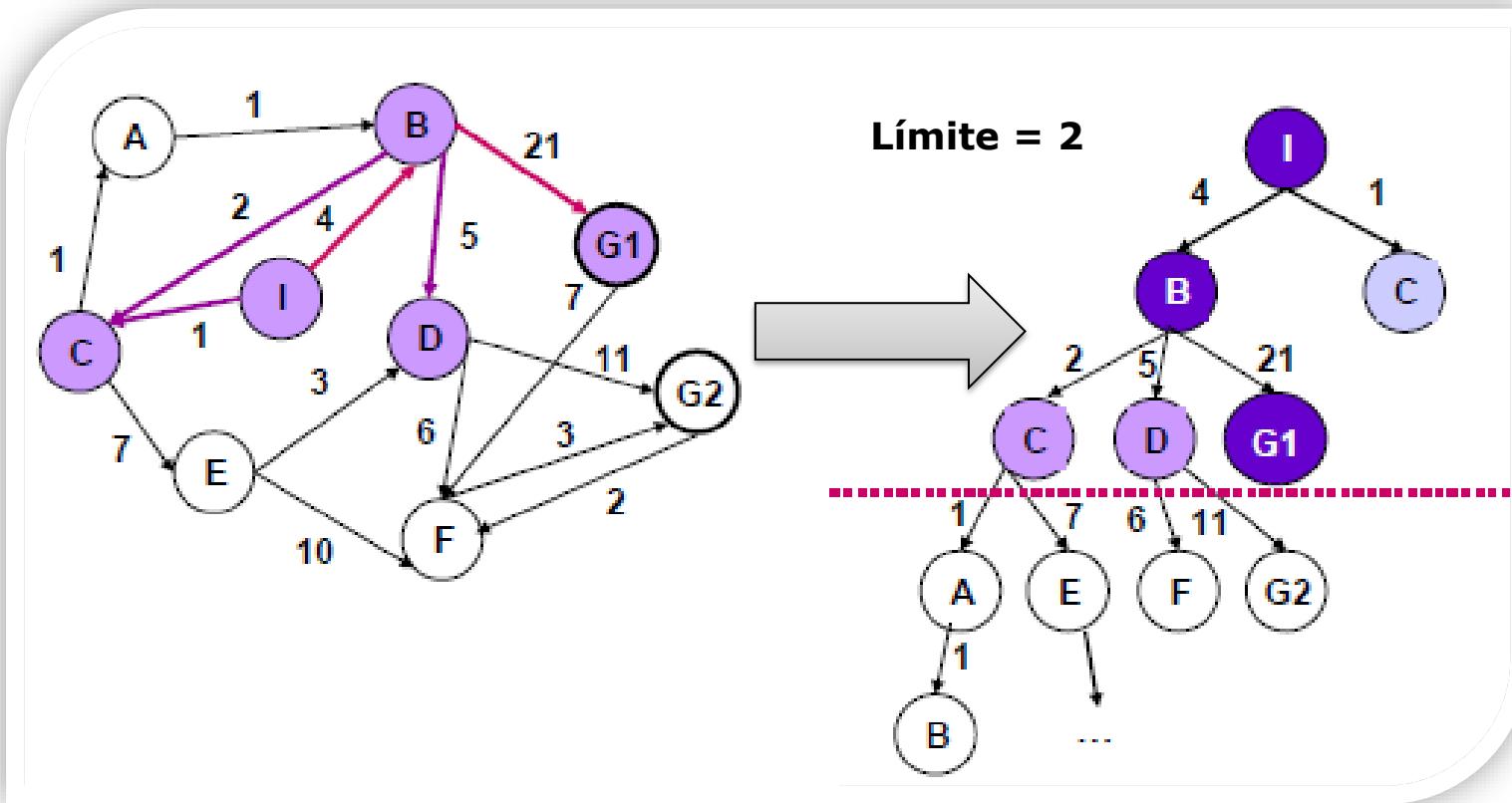
- Nodos cerrados: I

3.5 Búsqueda en profundidad iterativa



- Nodos cerrados: I I B C

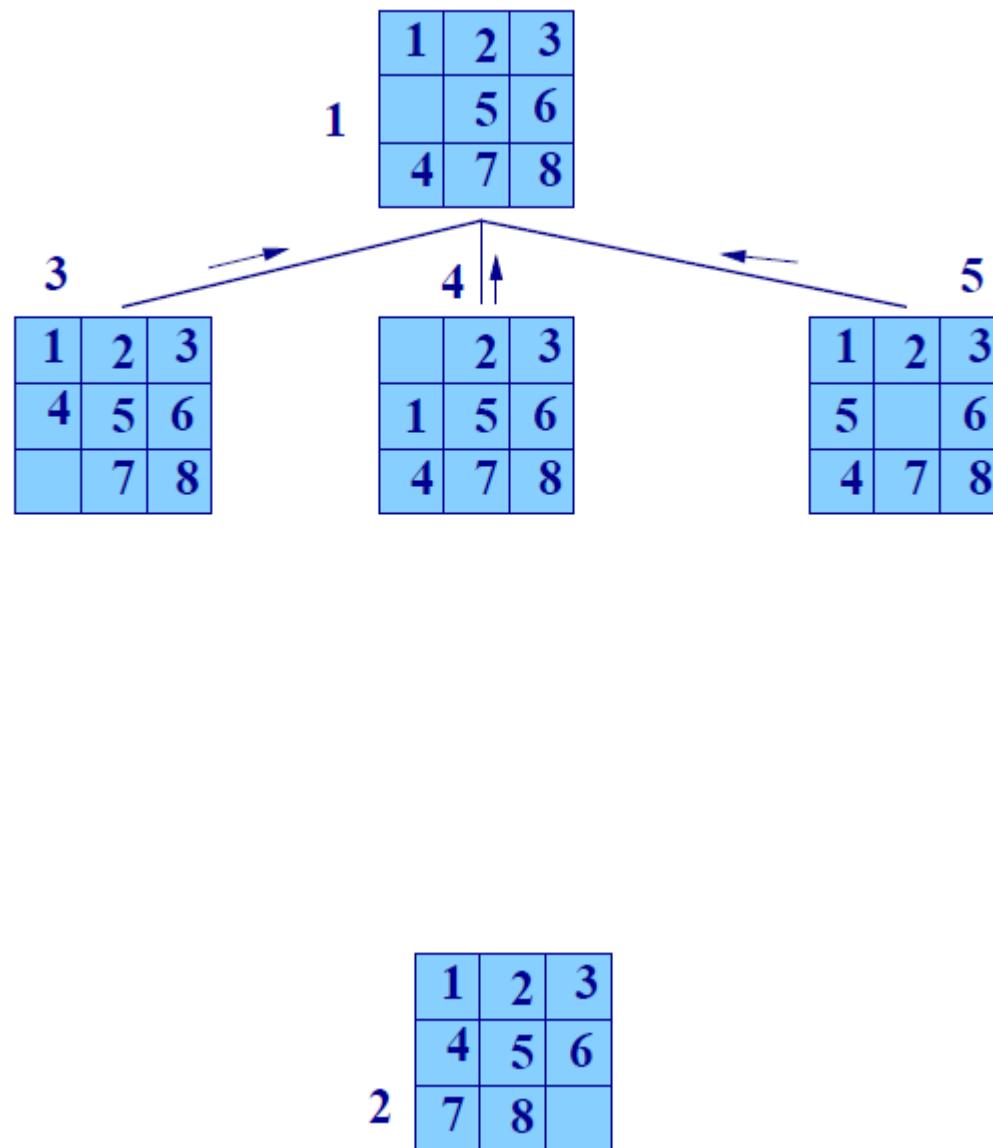
3.5 Búsqueda en profundidad iterativa



- Nodos cerrados: I I B C I B C D G1
- Camino a la solución: I B G1
- Coste: $4+21 = 25$

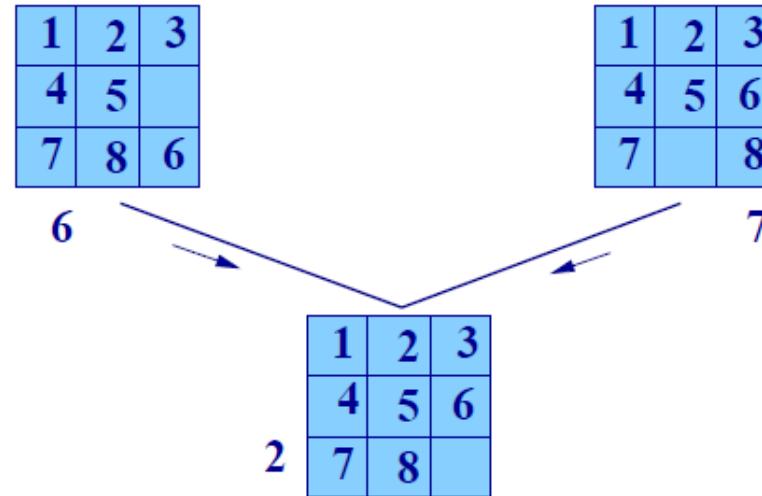
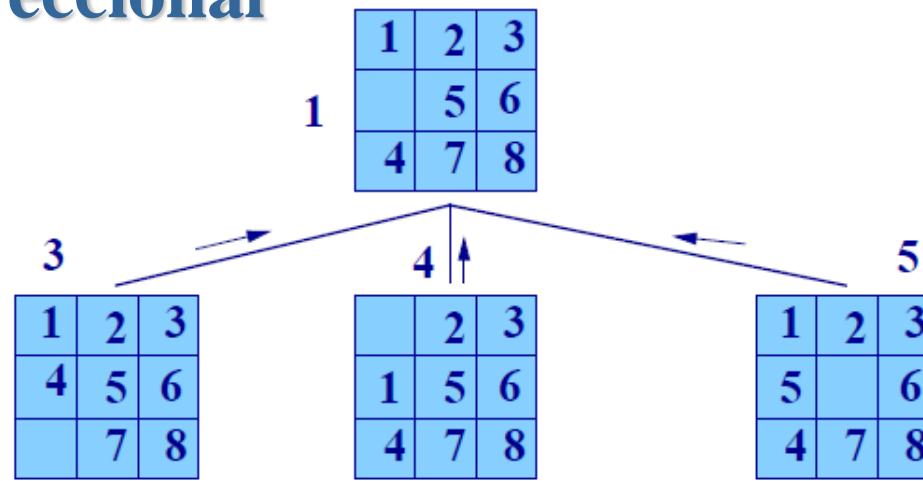


3.6 Búsqueda Bidireccional



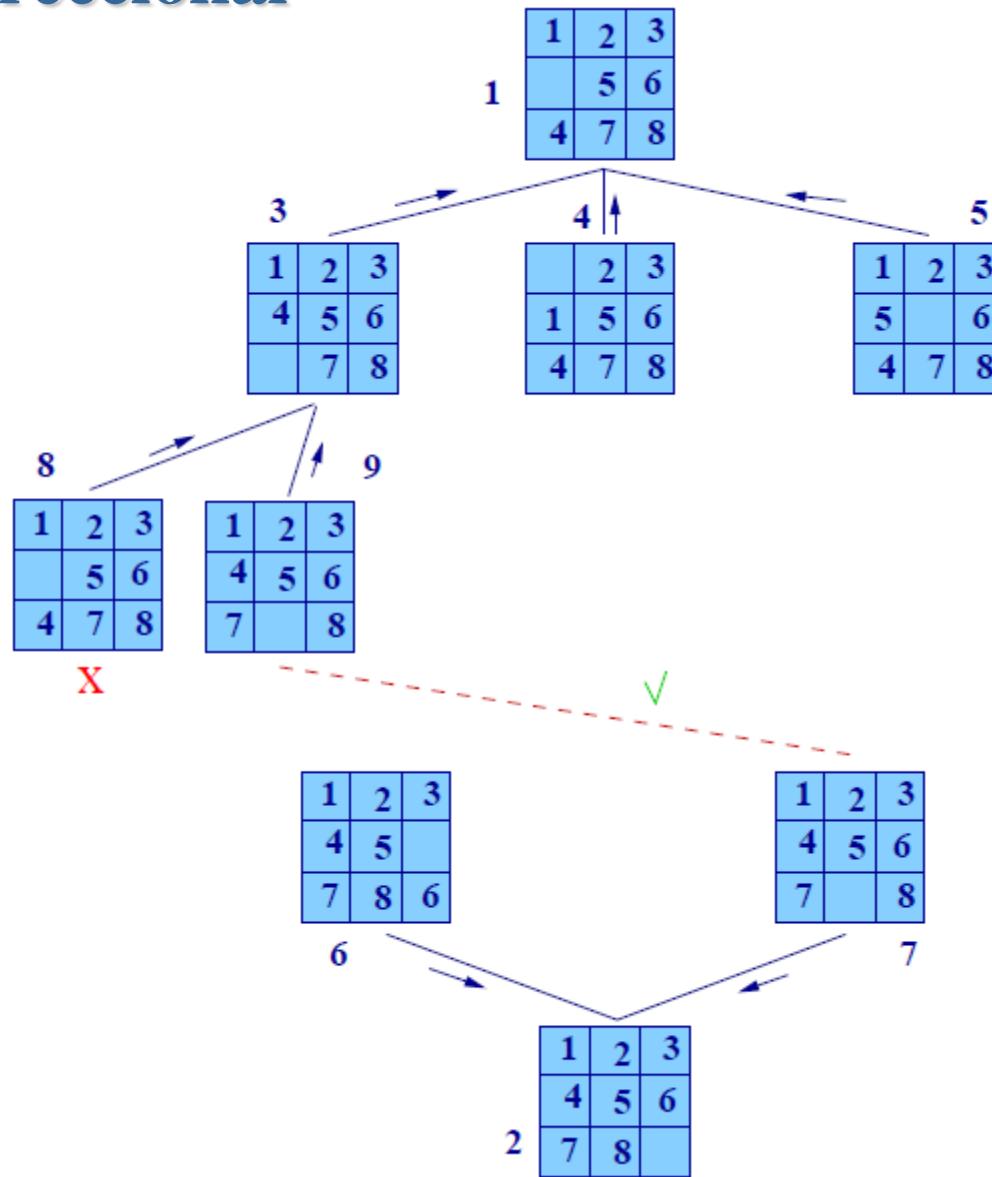


3.6 Búsqueda Bidireccional

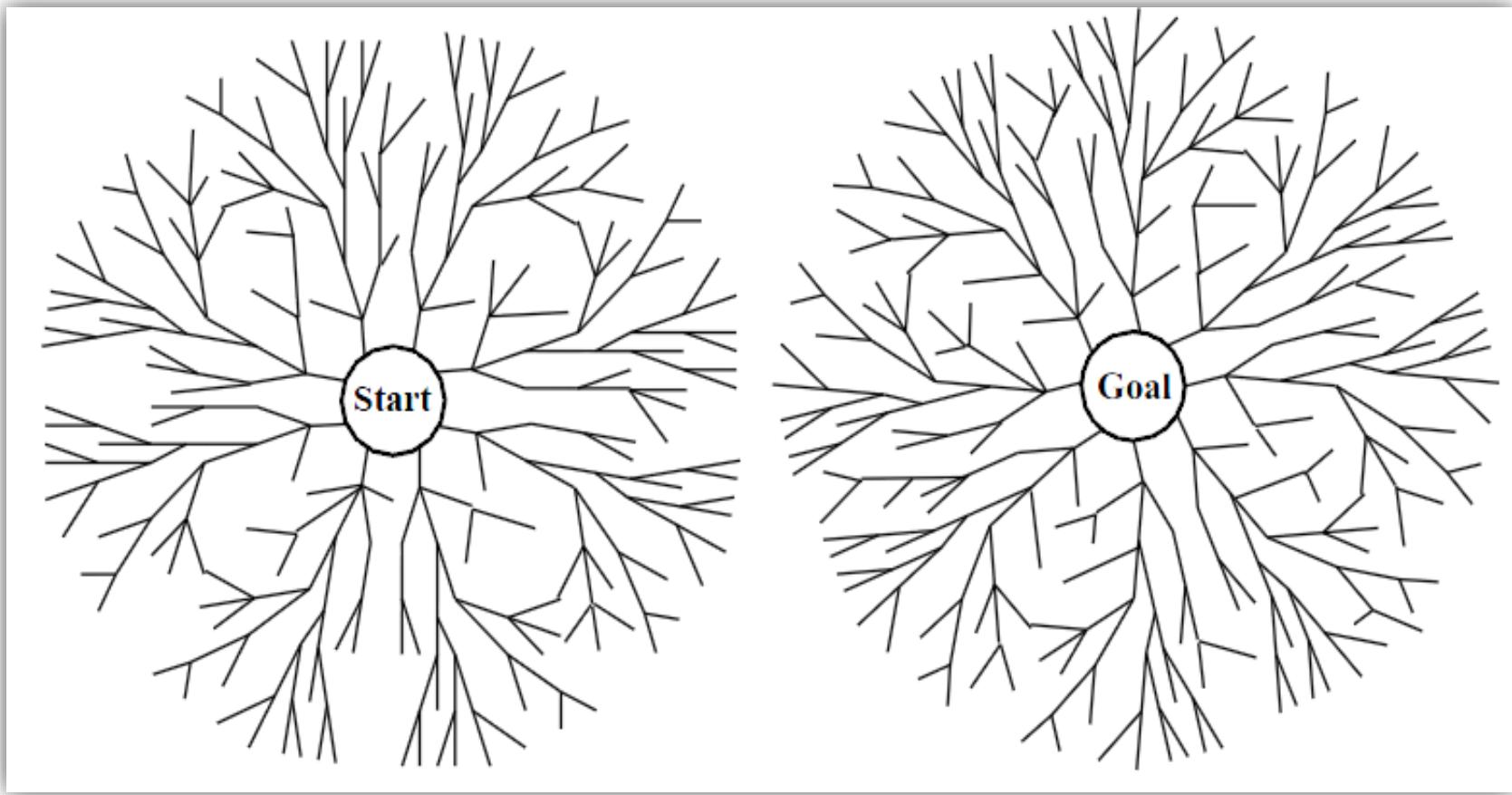




3.6 Búsqueda Bidireccional



3.6 Búsqueda Bidireccional



3.6 Búsqueda Bidireccional



- Ejecución de dos búsquedas simultáneas: una hacia delante desde el estado inicial y la otra hacia atrás desde el estado objetivo, parando cuando las dos búsquedas se encuentren
 - Motivación: $b^{d/2} + b^{d/2}$ es mucho menor que b^d
- Propiedades:
 - **Completa:** *Si, si las búsquedas son en anchura y los costes uniformes.* Otras combinaciones no garantizan encontrarse
 - **Complejidad tiempo:** $O(2 * b^{d/2}) = O(b^{d/2})$ Si la comprobación de la coincidencia puede hacerse en tiempo constante
 - **Complejidad espacio:** $O(b^{d/2})$ (su mayor debilidad) Al menos los nodos de una de las dos partes se deben mantener en memoria para la comparación
 - **Óptima:** *Si, si las búsquedas son en anchura y los costes son uniformes.*

3.6 Búsqueda Bidireccional



- Es necesario
 - Conocer explícitamente el estado objetivo (si hay varios, puede ser problemático)
 - Poder obtener los predecesores de un estado usando **operadores inversos**
 - No siempre la función predecesora es viable
 - ejemplo (jaque mate)
- Antes de expandir cada nodo hay que comprobar si está en la frontera del otro árbol
- Resultados
 - Eficiencia: reduce el tamaño total del árbol de búsqueda
 - Problema: la implementación de la comprobación de colisión debe ser muy eficiente

3.6 Búsqueda Bidireccional



PROCEDIMIENTO BIDIRECCIONAL(Estado-inicial, Estado-Final)

ABIERTO-I = ESTADO-INICIAL, *ABIERTO-F* = ESTADO-FINAL

Hacer *CERRADO* vacío

BUCLE (Repetir el proceso mientras ABIERTO-I o ABIERTO-F ≠ vacío)

1. NODO-ACTUAL = EXTRAE-PRIMERO(ABIERTO-I)
 - 1.1 Poner NODO-ACTUAL en *CERRADO*
 - 1.2 FUNCION SUCESORES(NODO-ACTUAL)
 - 1.3 GESTIONA-COLA(ABIERTO-I,SUCESORES)
 - 1.4 Si algún sucesor de NODO-ACTUAL coincide con algún nodo de ABIERTO-F devolver CAMINO desde ESTADO-INICIAL a ESTADO-FINAL por los punteros
2. Si no NODO-ACTUAL = EXTRAE-PRIMERO(ABIERTO-F)
 - 2.1 Poner NODO-ACTUAL en *CERRADO*
 - 2.2 FUNCION SUCESORES(NODO-ACTUAL)
 - 2.3 GESTIONA-COLA(ABIERTO-F,SUCESORES)
 - 2.4 Si algún sucesor de NODO-ACTUAL coincide con algún nodo de ABIERTO-I devolver CAMINO desde ESTADO-FINAL a ESTADO-INICIAL por los punteros

FIN DE BUCLE

Devuelve FALLO ☹



1. Introducción
2. Implementación
3. Métodos no informados
 1. Búsqueda en anchura
 2. Búsqueda de coste uniforme
 3. Búsqueda en profundidad
 4. Búsqueda en profundidad limitada
 5. Búsqueda en profundidad iterativa
 6. Búsqueda bidireccional
4. Complejidad

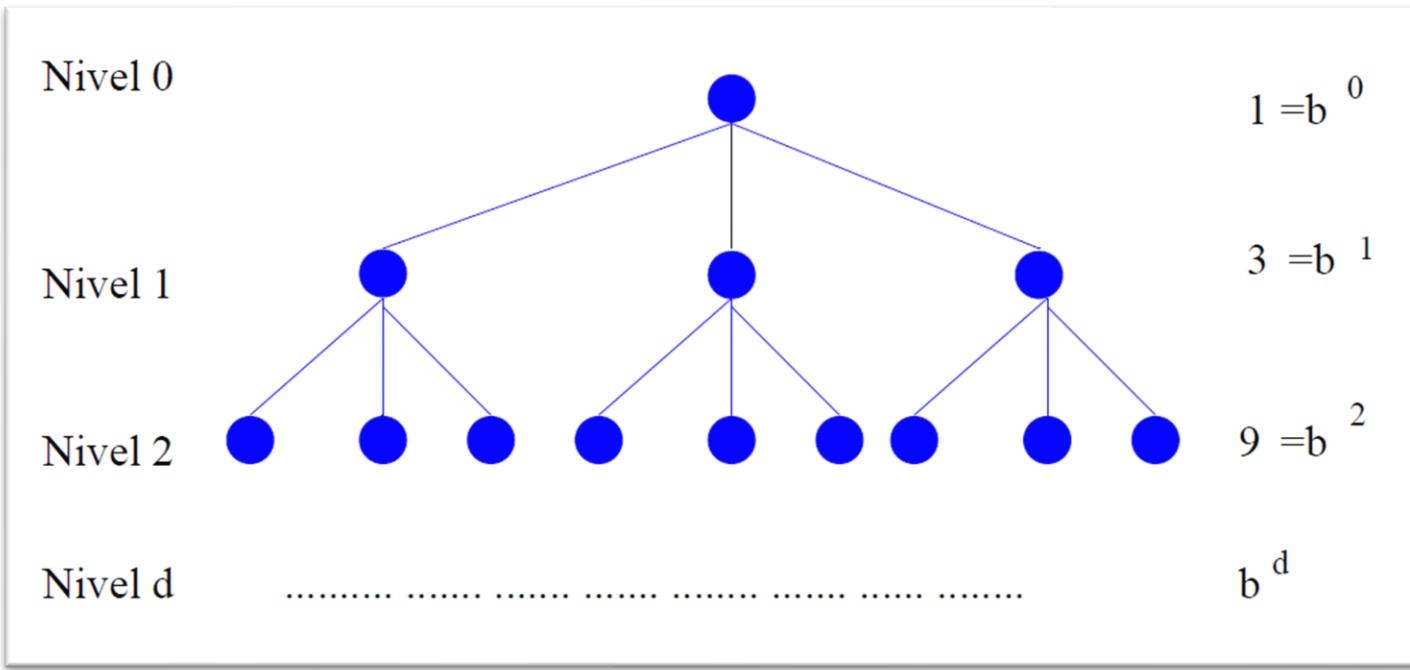


4. Complejidad

- Comparación de procedimientos de búsqueda
- Parámetros
 - b : factor de ramificación.
 - d : profundidad de la solución.
 - m : máxima profundidad de la búsqueda.
 - l : cota de la profundidad (profundidad límite)
- ¿Cuál sería, en el peor de los casos, el número de nodos que examinaría cada técnica?
- Ejemplo:
 - Supongamos $b = 3$, y vamos incrementando d
 - Calcular de forma inductiva el número de nodos



4. Complejidad



Técnica de Búsqueda	Número máximo de nodos
Primero en amplitud	$\sum_{i=0}^d b^i$
Primero en profundidad	$\sum_{i=0}^d b^i$
Primero en profundidad iterativo	$\sum_{i=0}^d (d - i + 1)b^i$
Bidireccional	$2 \sum_{i=0}^{\frac{d}{2}} b^i$

4. Complejidad



	Completa	Óptima	Eficiencia Tiempo	Eficiencia Espacio
Anchura	Si	Si coste \approx profundidad	$O(b^d)$	$O(b^d)$
Coste uniforme	Si no hay bucles de coste ∞	Si coste operadores ≥ 0	$O(b^d)$	$O(b^d)$
Profundidad	No	No	$O(b^m)$	$O(b*m)$
Profundidad limitada	Si $l \geq d$	No	$O(b^l)$	$O(b*l)$
Profundidad iterativa	Si	Si coste \approx profundidad	$O(b^d)$	$O(b*d)$
Bidireccional	Si (anchura)	Si	$O(b^{d/2})$	$O(b^{d/2})$



Universidad
Francisco de Vitoria
UFV Madrid

Ingeniería del Conocimiento

Tema 4:
Búsqueda Informada (I)

Objetivos del tema



- Ubicación
 - Unidad 2: **BUSQUEDA EN ESPACIO DE ESTADOS**
 - *Tema 4: Búsqueda Informada: Heurísticas (I)*
- Objetivos generales
 - *Definir búsqueda informada* y entender su ámbito de aplicación en contraposición a la búsqueda ciega (no informada)
 - Definir las *funciones heurísticas* y entender su uso en las búsquedas informadas evaluando estados en vez de explorar todos los caminos desde el estado inicial
 - Entender los algoritmos de búsqueda *Primero El Mejor (Avara y el Algoritmo A*)*, su uso y sus limitaciones
 - Saber *aplicar de cada método* en función de la completitud y *complejidad* espacial y temporal



1. Introducción
2. Funciones heurísticas
3. Búsquedas “primero el mejor”
 1. Búsqueda avara
 2. Búsqueda A*
 3. Variaciones de A*
4. Búsquedas iterativas
 1. Hill Climbing
 2. Simulated Annealing



- 1. Introducción**
- 2. Funciones heurísticas**
- 3. Búsquedas “primero el mejor”**
 - 1. Búsqueda avara**
 - 2. Búsqueda A***
 - 3. Variaciones de A***
- 4. Búsquedas iterativas**
 - 1. Hill Climbing**
 - 2. Simulated Annealing**

1. Introducción



- Búsqueda: exploración del espacio de estados por medio de la generación de sucesores de los estados explorados
 - Si se tiene conocimiento perfecto → algoritmo exacto
 - Si no se tiene conocimiento → búsqueda sin información
 - Los problemas reales están en posiciones intermedias (búsqueda con alguna información).
- Cuando se emplea información del espacio de búsqueda para evaluar el proceso y elegir que nodo del árbol de búsqueda es más prometedor para alcanzar la meta hablamos de estrategias de búsqueda
 - INFORMADAS ó HEURÍSTICAS
(usan información disponible del problema)
- La idea es utilizar una función de evaluación (heurístico) de cada nodo (del coste de llegar de él al estado final).
Estimamos el futuro...



1. Introducción

- Todos los problemas que trata la IA son NP-difíciles
 - Resolverlos de forma exacta requiere búsqueda en un espacio de estados de tamaño exponencial.
 - No se sabe cómo evitar esa búsqueda.
 - No se espera que nadie lo consiga nunca.
 - Si existe un algoritmo rápido para resolver un problema, no consideramos que el problema requiera inteligencia
- Si un problema es NP-difícil y tenemos un algoritmo que encuentra la solución de forma rápida y casi siempre correcta, podemos considerar que el algoritmo es “inteligente”.
 - Que lo resuelva “casi siempre de forma correcta” implica que la búsqueda está sujeta a error.



1. Introducción
2. Funciones heurísticas
3. Búsquedas “primero el mejor”
 1. Búsqueda avara
 2. Búsqueda A*
 3. Variaciones de A*
4. Búsquedas iterativas
 1. Hill Climbing
 2. Simulated Annealing



2. Funciones heurísticas

- Heurística (¡Eureka!):

heurístico, ca.

Artículo enmendado

(Del gr. *εὑρίσκειν*, hallar, inventar, y *-tico*).

1. adj. Perteneciente o relativo a la **heurística**.
2. f. Técnica de la indagación y del descubrimiento.
3. f. Busca o investigación de documentos o fuentes históricas.
4. f. En algunas ciencias, manera de buscar la solución de un problema mediante métodos no rigurosos, como por tanteo, reglas empíricas, etc.

Real Academia Espaola © Todos los derechos reservados

- *Técnica o regla empírica que ayuda a encontrar la solución de un problema (pero que no garantiza que se encuentre)*
- *Criterios, métodos o principios para decidir cuál de entre varias acciones promete ser la mejor para alcanzar una determinada meta.*



2. Funciones heurísticas

- Características de los métodos heurísticos:
 - **No garantizan** que se encuentre una solución, aunque existan soluciones (sacrifica la completitud).
 - Si encuentran una solución, **no se asegura que sea la mejor** (longitud mínima o de coste óptimo).
 - En **algunas ocasiones** (que, en general, no se podrán determinar a priori) encontrarán una solución aceptablemente buena en un tiempo razonable.
- Se representan mediante
 - Funciones $h(n)$
 - Metareglas
- Las heurísticas se descubren resolviendo modelos simplificados (*relajados*) del problema real

2. Funciones heurísticas



- Asocian a cada estado del espacio de estados **una cierta cantidad numérica que evalúa de algún modo lo prometedor que es ese estado** para alcanzar un estado objetivo
- ¿Qué es “mejor” valor heurístico?. Dos opciones:
 - Si estimamos la "calidad" de un estado
 - Los estados de mayor valor heurístico son los preferidos
 - Si estimamos lo “próximo” que se encuentra de un estado objetivo (coste estimado del camino más barato)
 - Los estados de menor valor son los preferidos
- Ambos puntos de vista son complementarios
- Convenio: asumiremos la 2^a interpretación. Implica:
 - Valores no negativos
 - El mejor es el menor
 - Los objetivos tienen valor heurístico 0

2. Funciones heurísticas . Heurística Admisible



- **Heurística admisible** (optimista): Una heurística $h(n)$ es admisible si

$$\forall n, h(n) \leq h^*(n),$$

donde $h^*(n)$ es el coste real de alcanzar el objetivo desde el estado n .

- Una heurística admisible no subestima la calidad de un buen plan. Por el contrario, una heurística inadmisible (pesimista) impiden que el algoritmo sea óptimo al descartar buenos planes.
- Una heurística **admisible** se puede derivar de una versión simplificada del problema.

2. Funciones heurísticas . Heurística Admisible



- Por ejemplo, del 15-puzzle visto anteriormente, podemos definir tres problemas más simples.
La regla original del juego es:
"Una ficha se puede mover desde la casilla A hasta la casilla B si A es adyacente a B y B está vacía (en blanco)".
- Se pueden generar tres problemas quitando una o ambas condiciones:
 1. Una ficha se puede mover desde la casilla A hasta la casilla B si A es adyacente a B.
 2. Una ficha se puede mover desde la casilla A hasta la casilla B si B está vacía (en blanco).
 3. Una ficha se puede mover desde la casilla A hasta la casilla B.
- de 1) podemos derivar la **Distancia Manhattan**,
- de 2) podemos derivar la Distancia **de Gaschnig** (movimiento del rey) y
- de 3) podemos derivar la **Distancia de Hamming**.

2. Funciones heurísticas



Ejemplo: 8-puzzle

- Restricciones:
 1. Solo se puede mover el blanco
 2. Solo se puede mover a casillas adyacentes horizontal o vertical
 3. En cada paso, se intercambian los contenidos de dos casillas
- Relajaciones:
 - Si quitamos 1 y 2, heurística h_1 :
número de casillas mal colocadas respecto al objetivo
excluyendo la vacía
 - Es la heurística más sencilla y parece bastante intuitiva
 - No usa información relativa al esfuerzo (nº de movimientos) necesario para llevar una ficha a su sitio

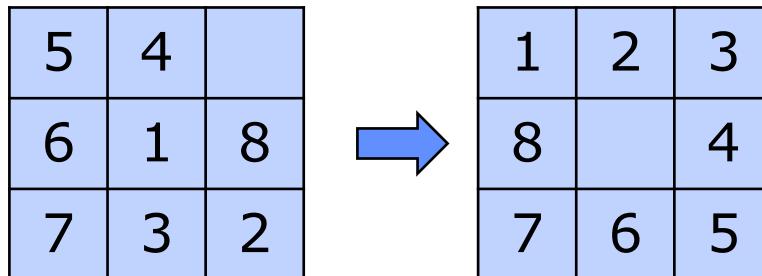


2. Funciones heurísticas

- Si quitamos 1, heurística h_2 :
suma de las distancias de las fichas a sus posiciones en el objetivo excluyendo la vacía
 - Como no hay movimientos en diagonal, se sumarán las distancias horizontales y verticales
 - Distancia de Manhattan (distancia taxi): número de cuadros desde el sitio correcto de cada cuadro

$$|(x_f - x_i)| + |(y_f - y_i)|$$

- Ejemplo



- Solución

- h_1 : 7
- h_2 : $2+3+3+2+4+2+0+2 = 18$

(1 2 3 4 5 6 7 8)



2. Funciones heurísticas

- Sin embargo, estas dos heurísticas no dan importancia a la dificultad de la inversión de fichas
 - Si 2 fichas están dispuestas de forma contigua y han de intercambiar sus posiciones, ponerlas en su sitio supone (bastante) más de 2 movimientos
 - Heurística h_3 : doble del nº de pares de fichas a "invertir entre sí"
 - Tampoco es buena porque se centra solo en un cierto tipo de dificultad sin considerar el problema general
 - En particular, tendrán valor 0 muchos estados que no son el objetivo
- Se suelen usar heurísticas compuestas
$$h_4 = h_2 + h_3$$
 - Mejor heurística, pero requiere más cálculo

2. Funciones heurísticas. Dominante



Heurística dominante

- $h_i(n)$ es dominante si para un problema dado

$$h_i(n) \geq h_h(n) \quad \forall h \quad \forall n$$

- Si $h_2 \geq h_1 \quad \forall n \rightarrow h_2$ (domina a/está más informada que) h_1

- La dominación se traduce en eficiencia: una heurística dominante expande menos nodos

Por ejemplo: La distancia de Manhattan está más informada que la heurística de número de casillas mal colocadas

2. Funciones heurísticas



1	3	
8	2	4
7	6	5

1	2	3
8		4
7	6	5

2	1	3
8		4
7	6	5

Estado objetivo

$$h_1 = 2$$

$$h_2 = 2$$

$$h_3 = 0$$

$$h_1 = 2$$

$$h_2 = 2$$

$$h_3 = 2$$

$$h_4 = 2$$

$$h_4 = 4$$

Dominante

Reales = 2

Reales > 10



2. Funciones heurísticas

1	2	3
8		4
7	6	5

Estado objetivo

	H1	H2	H3	H4
	5	6	0	6
	3	4	0	4
	5	6	0	6



2. Funciones heurísticas

- En general, los métodos heurísticos son preferibles a los métodos no informados en la solución de problemas difíciles para los que una búsqueda exhaustiva necesitaría un tiempo demasiado grande.
- *Es necesario un compromiso entre el coste de la función heurística y la mejora que supone en la búsqueda*
- *Es preferible usar una función heurística dominante siempre y cuando sea admisible*



2. Funciones heurísticas

- Ejemplo: el coste de resolver el puzzle de las 8 piezas mediante las estrategias de búsqueda por profundización iterativa y A* con heurísticos h_1 (casillas mal colocadas) y h_2 (*distancia de Manhattan*)
 - $d = 14$
 - profundización iterativa → 3.473.941 nodos
 - A* con h_1 → 539 nodos
 - A* con h_2 → 113 nodos
 - $d = 24$
 - profundización iterativa → !demasiados nodos!
 - A* con h_1 → 39.135 nodos
 - A* con h_2 → 1.641 nodos



2. Funciones heurísticas

- Búsquedas heurísticas o informadas

- Primero el mejor (PEM o *best-first*)
 - Búsqueda avara/voraz (*greedy search*)
 - *Búsqueda A**
 - Variaciones de A*
- Mejora iterativa
 - Métodos de gradiente (*hill-climbing*)
 - *Simulated annealing*
- Búsqueda con adversarios
 - *Búsqueda MiniMax con decisiones imperfectas*
 - *Poda Alfa-Beta*
- Búsqueda con restricciones



1. Introducción
2. Funciones heurísticas
3. Búsquedas “primero el mejor”
 1. Búsqueda avara
 2. Búsqueda A*
 3. Variaciones de A*
4. Búsquedas iterativas
 1. Hill Climbing
 2. Simulated Annealing



3. Búsquedas “primero el mejor”

- Se utiliza una función de evaluación $f(n)$ para cada nodo y se expande el nodo mejor evaluado no expandido
 - Misma idea que en la búsqueda de coste uniforme:
 - Cola con prioridad, mantiene **continuamente** la frontera con orden creciente de $f(n)$
 - Se expande el nodo que *parece* mejor según $f(n)$ (*aunque es una función inexacta*)



- En general $f(n) = g(n) + h(n)$
- La función heurística $h(n)$ es el coste estimado del camino más barato desde n al objetivo (el futuro)
 - Condición: Si n es un nodo objetivo entonces $h(n) = 0$
- Válido cuando lo que interesa es encontrar el camino completo, no la solución en sí.



3. Búsquedas “primero el mejor”

- Tipos de búsquedas PEM:

- Búsqueda en anchura
 - $f(n) = \text{profundidad}(n)$ mínima
- Búsqueda de coste uniforme (Dijkstra)
 - $f(n) = g(n)$ mínima

Ya hemos visto estas búsquedas NO HEURISTICAS al ver los métodos NO INFORMADOS

Ahora nos centramos en:

- Búsqueda voraz o avara (*greedy search*)
 - $f(n) = h(n)$ mínima
- Búsqueda A*
- Variaciones del A* que funcionan acotando el uso de memoria



3.1 Búsqueda avara

- $f(n)$ estima el coste del nodo n hasta la meta, con lo que se expande el nodo **NO EXPANDIDO** que *parece* estar más cerca de la meta
- $f(n)$ es directamente la función heurística $h(n)$ del estado

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

- h puede ser cualquier función siempre y cuando $h(n) = 0$ en los nodos que representan estados objetivo
- Propiedades:
 - Completa: **No**, en general. **Si**, si se aplican políticas de poda de bucles.
 - Complejidad tiempo: $O(b^d)$ hay que recorrer todos los nodos
 - Complejidad espacio: $O(b^d)$ hay que recorrer todos los nodos
 - Optima: **No**



3.1 Búsqueda avara

- La búsqueda voraz:
 - es propensa a comienzos erróneos
 - como la búsqueda primero en profundidad, no es completa ni óptima
 - prefiere seguir un camino hasta el final (cabezota)
 - puede atascarse en bucles infinitos
- Las complejidades temporal y espacial pueden reducirse sustancialmente con un buen heurístico.

¡¡Una mala heurística es peor que una mala búsqueda!!



3.1 Búsqueda avara

PROCEDIMIENTO VORAZ(Estado-inicial, Estado-Final)

ABIERTO = *ESTADO-INICIAL*

Hacer *CERRADO* vacío

BUCLE (Repetir el proceso mientras ABIERTO ≠ vacío)

1. NODO-ACTUAL = EXTRAE-PRIMERO(ABIERTO)
2. Poner NODO-ACTUAL en *CERRADO*
3. Si ES-ESTADO-FINAL(ESTADO(NODO-ACTUAL))
devolver CAMINO(NODO-ACTUAL)
4. Si no, FUNCION SUCESORES(NODO-ACTUAL)
GESTIONA-COLA(ABIERTO,SUCESORES)
Si no están en *ABIERTO* o *CERRADO* añadir *SUCESORES* ordenadamente a ABIERTO en orden creciente de $h(n)$

FIN DE BUCLE

Devuelve FALLO 😞

3.1 Búsqueda avara

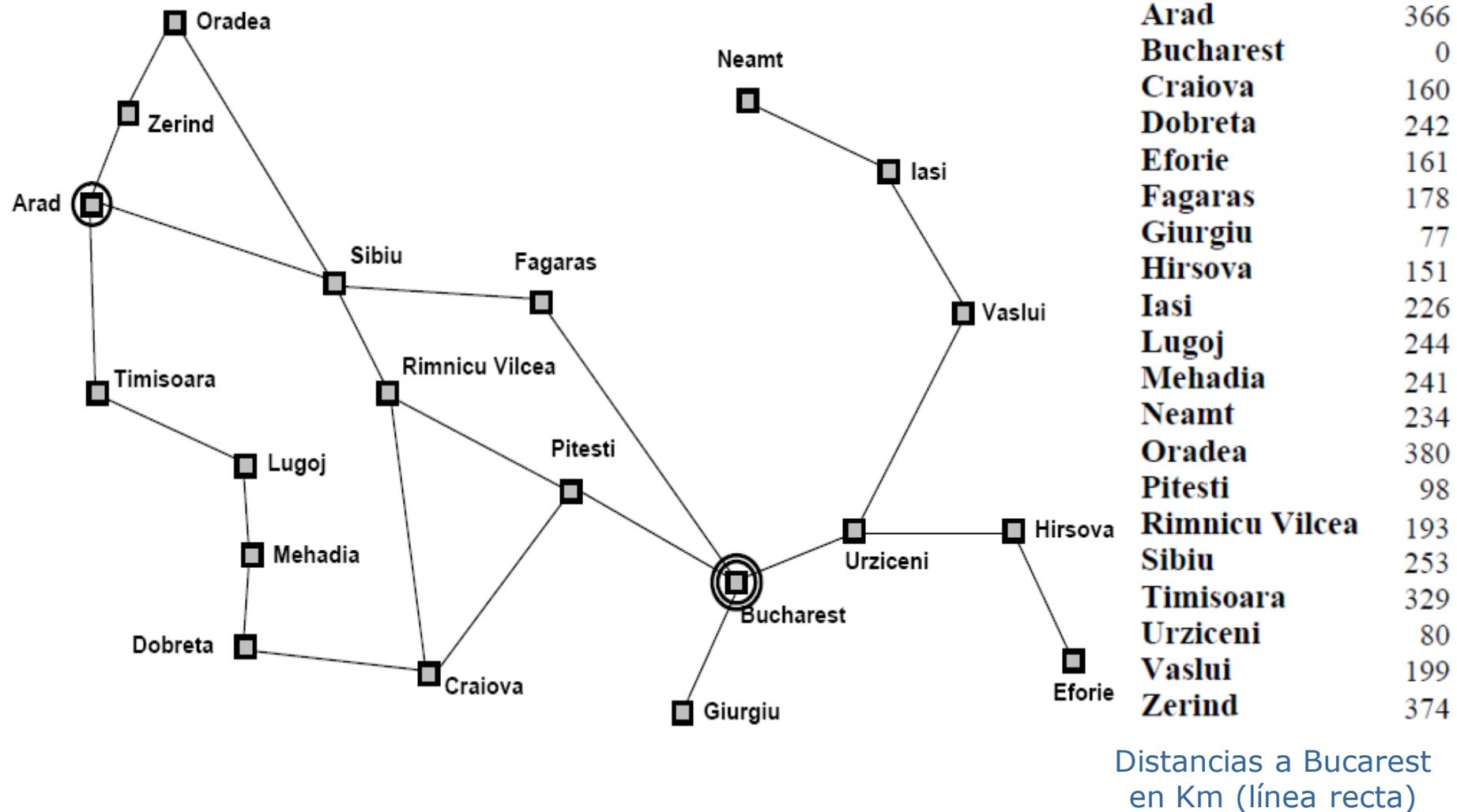


- El problema del viaje por Rumanía.
 - Estado inicial: estamos en una ciudad.
 - Estado meta: quiere viajar a otra ciudad por la mejor ruta posible (la más corta)
 - Medios: Las ciudades colindantes están unidas por carreteras; se dispone de un mapa con la disposición de las provincias y sus "coordenadas" en kilómetros respecto al "centro"
- $F(n)$: asignar a cada nodo la distancia aérea (en línea recta) con el estado objetivo (distancia euclídea entre las coordenadas de dos ciudades).
 - Se elige una ciudad como siguiente en el camino cuando la distancia aérea a la meta sea la menor.

$$f(n) = h(n) \text{ mínima}$$



3.1 Búsqueda avara



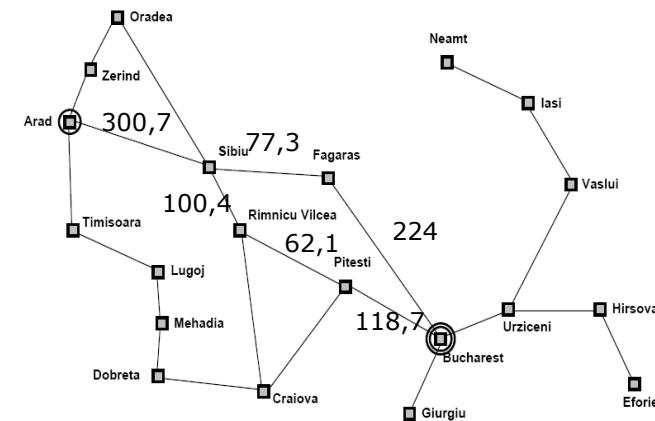
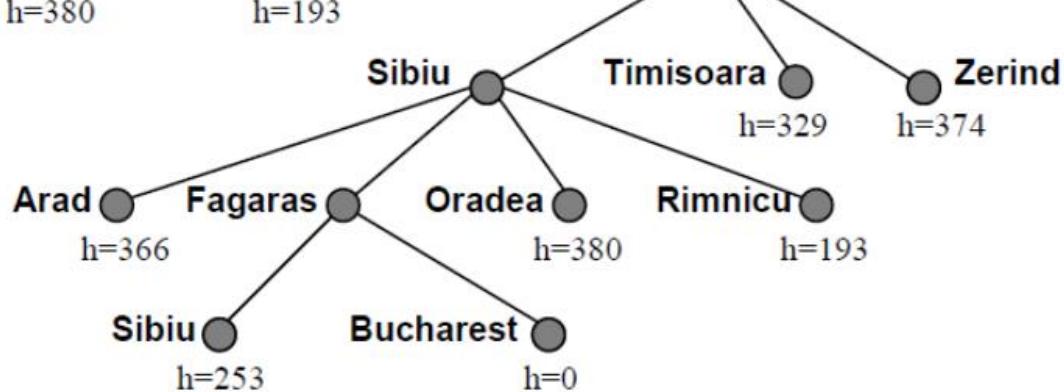
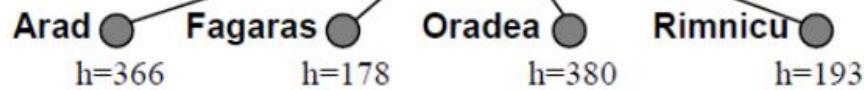
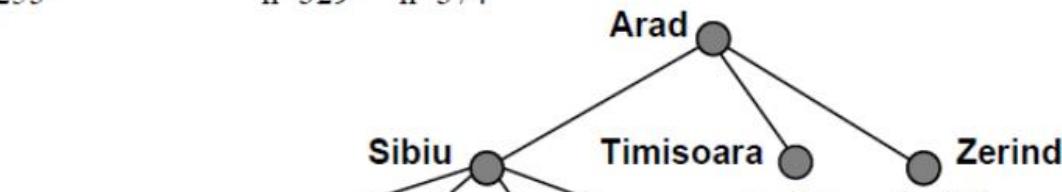
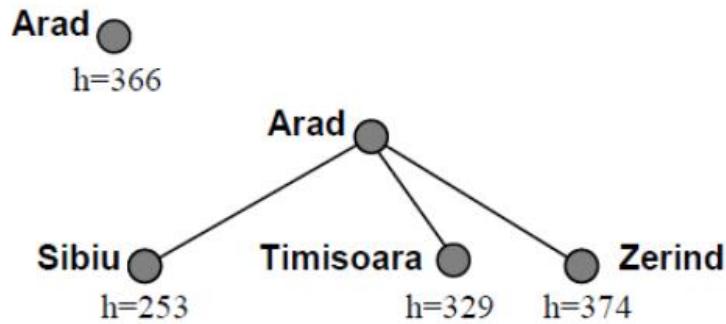


3.1 Búsqueda avara

- Para el problema de hallar una ruta entre Arad y Bucarest, la búsqueda voraz con el heurístico $h_{DLR}(n)$:
 - encuentra una solución sin expandir ningún nodo que no esté incluido en la misma (coste de búsqueda mínimo),
 - aunque la solución no es óptima
- $h_{DLR}(n)$: (Distancia en Línea Recta)
 - necesita de las coordenadas de las ciudades del mapa
 - es útil porque sabemos que las carreteras entre dos ciudades tienden a ser rectas (conocimiento específico del problema)



3.1 Búsqueda avara



Por Fagaras: 622 km
Por Rimnicu: 581,9



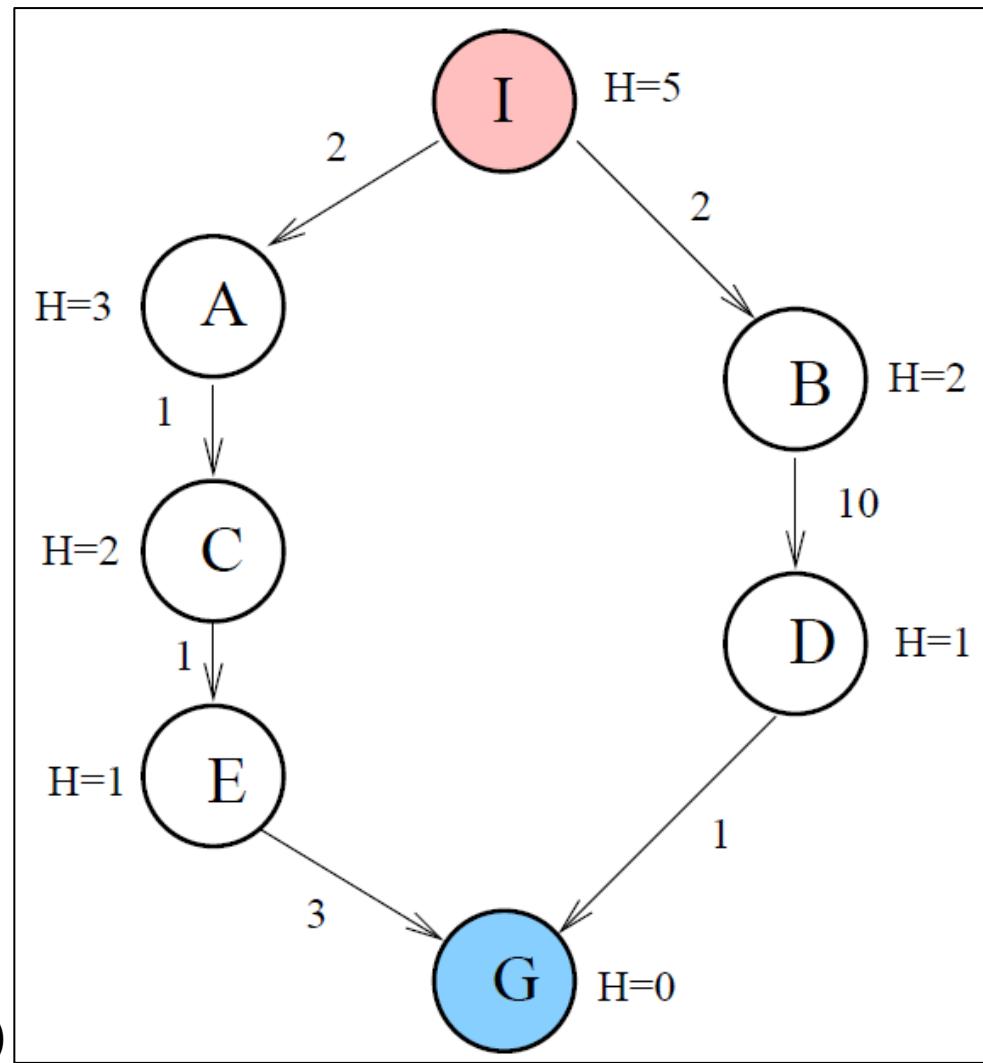
3.1 Búsqueda avara

- EJEMPLO:
NO se encuentra
la solución óptima

- Solución encontrada
por búsqueda voraz:
I-B-D-G
Coste: 13

Pero existe una solución
mejor: I-A-C-E-G
con un coste menor: 7

- Causa: no se han tenido
en cuenta *los costes* de
los caminos ya recorridos
(no ha utilizado el pasado)



3.2 Búsqueda A*



- Hart, Nilsson y Raphael, 1968
- Búsqueda A*
 - Evita expandir caminos que ya son muy costosos minimizando el costo estimado total de la solución.
 - Combina:
 - la búsqueda voraz, que minimiza el coste al objetivo $h(n)$
 - Búsqueda en profundidad
 - la búsqueda de coste uniforme, que minimiza el coste acumulado $g(n)$
 - Búsqueda en anchura
- Expande primero el nodo no expandido más prometedor hasta ese momento según la Función de Evaluación:

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

3.2 Búsqueda A*



- Donde
 - $f(n)$: función de evaluación
 - Coste estimado total hasta la meta pasando por n .
 - Expresa el mínimo estimado de la solución que pasa por el nodo n
 - $g(n)$: función de coste para ir desde el nodo inicial al actual
 - $h(n)$: función heurística que mide la distancia (o coste) estimada desde n a algún nodo meta
- Los valores reales (*) solo se conocen al final de la búsqueda
 - $h^*(n)$: coste real para ir del nodo n a algún *nodo meta*
 - $f^*(n)$: coste real para ir del *nodo inicial* a algún *nodo meta* a través de n
 - $g(n)$ si se conoce y se calcula como la suma de los costes de los arcos recorridos, $k(n_i, n_j)$

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

3.2 Búsqueda A*



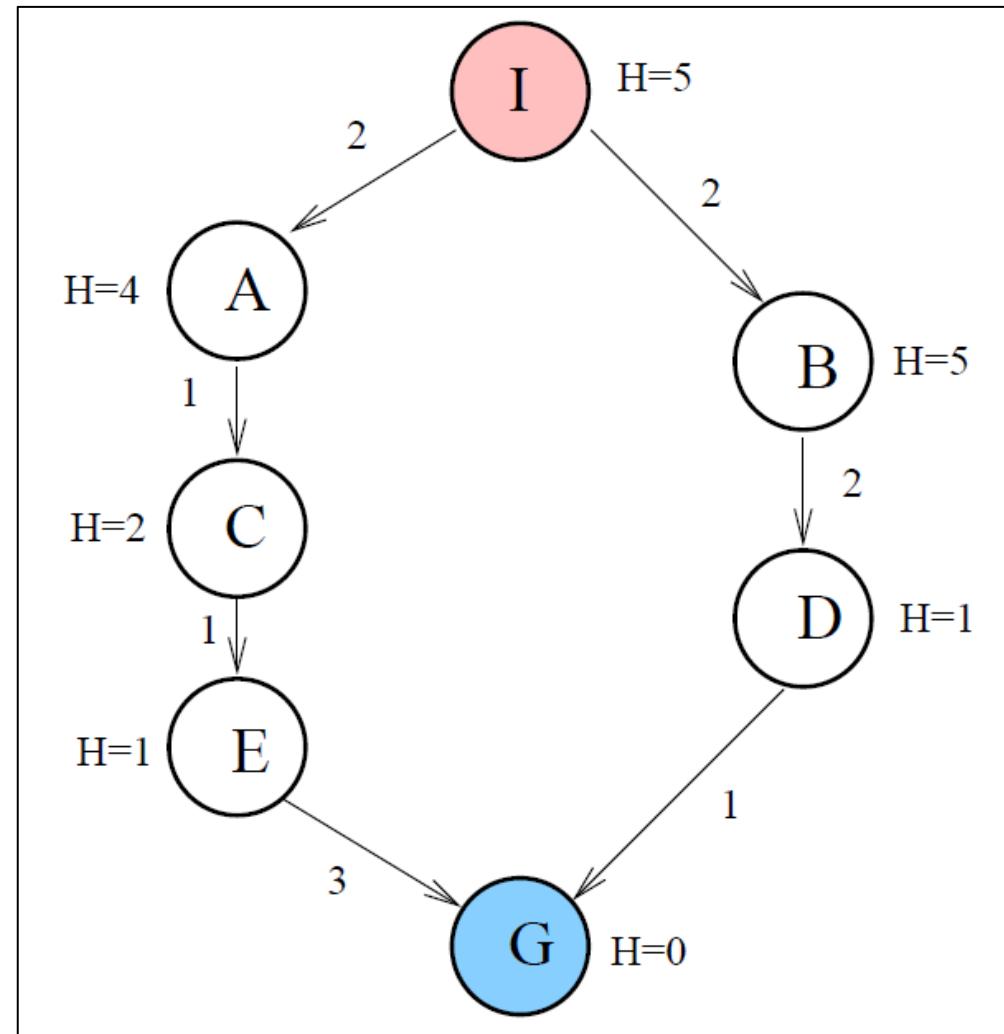
Heurística admisible

- Como no conocemos el valor h^* usamos una función de estimación a la que llamamos $h_i(n)$ que será admisible si
$$h^*(n) \geq h(n) \quad \forall n$$
 - *Optimista*: Subestima el coste real de llegar al objetivo
 - *El coste de una solución óptima en un problema relajado es una heurística admisible para el problema original*
- Posibilidades
 - $h(n) \leq h^*(n) \quad \forall n$: *$h(n)$ admisible. Búsqueda A**
 - $h(n) = 0 \quad \forall n$: no se tiene información. Búsqueda Dijkstra
 - $h(n) = h^*(n) \quad \forall n$: estimación perfecta. No hay búsqueda
 - $h(n) > h^*(n)$ para algún n : *$h(n)$ no admisible. Búsqueda A.* No se puede garantizar que la solución sea óptima



3.2 Búsqueda A*

- EJEMPLO:
NO se encuentra la solución óptima
 - Solución encontrada por A*: I-A-C-E-G
 - Causa: la heurística *sobreestima* el coste real en B



3.2 Búsqueda A*



Heurística consistente

- Una heurística $h(n)$ es consistente si
$$h(\text{padre}) \leq h(\text{hijo}) + k(\text{padre}, \text{hijo}) \quad \forall n$$

consistente → admisible

- Si $h(n)$ es **admisible** y **consistente** entonces $f(n)$, a lo largo de cualquier camino, no disminuye (es *monótona no decreciente*)



3.2 Búsqueda A*

- En el nodo inicial,
 - $g(\text{inicial})=0$
 $f(\text{inicial}) = h(\text{inicial})$
 $f^*(\text{inicial}) = h^*(\text{inicial})$
 - Como además $h(\text{inicial}) \leq h^*(\text{inicial})$
 $f(\text{inicial}) \leq f^*(\text{inicial})$
- En el nodo final,
 - $h(\text{final})=0$
 $f(\text{final}) = g(\text{final})$
 - Como además $h^*(\text{inicial}) = g(\text{final})$
 $f(\text{final}) = f^*(\text{inicial})$
- Luego
 $f(\text{inicial}) \leq f(\text{final})$

3.2 Búsqueda A*



- La secuencia de nodos expandidos por A* estará en orden no decreciente de $f(n)$
 - El primer nodo expandido cada vez debe ser una solución óptima, ya que todos los posteriores serán al menos tan costosos
 - No revisita nodos. La primera expansión es la mejor
- Si f^* es el coste de la solución óptima, entonces
 - A* expande todos los nodos con $f(n) < f^*$
 - A* puede expandir algunos nodos situados sobre "*la curva de nivel objetivo*" (donde $f(n) = f^*$) antes de seleccionar un nodo objetivo
 - A* no expande ningún nodo con $f(n) > f^*$ (poda)

3.2 Búsqueda A*

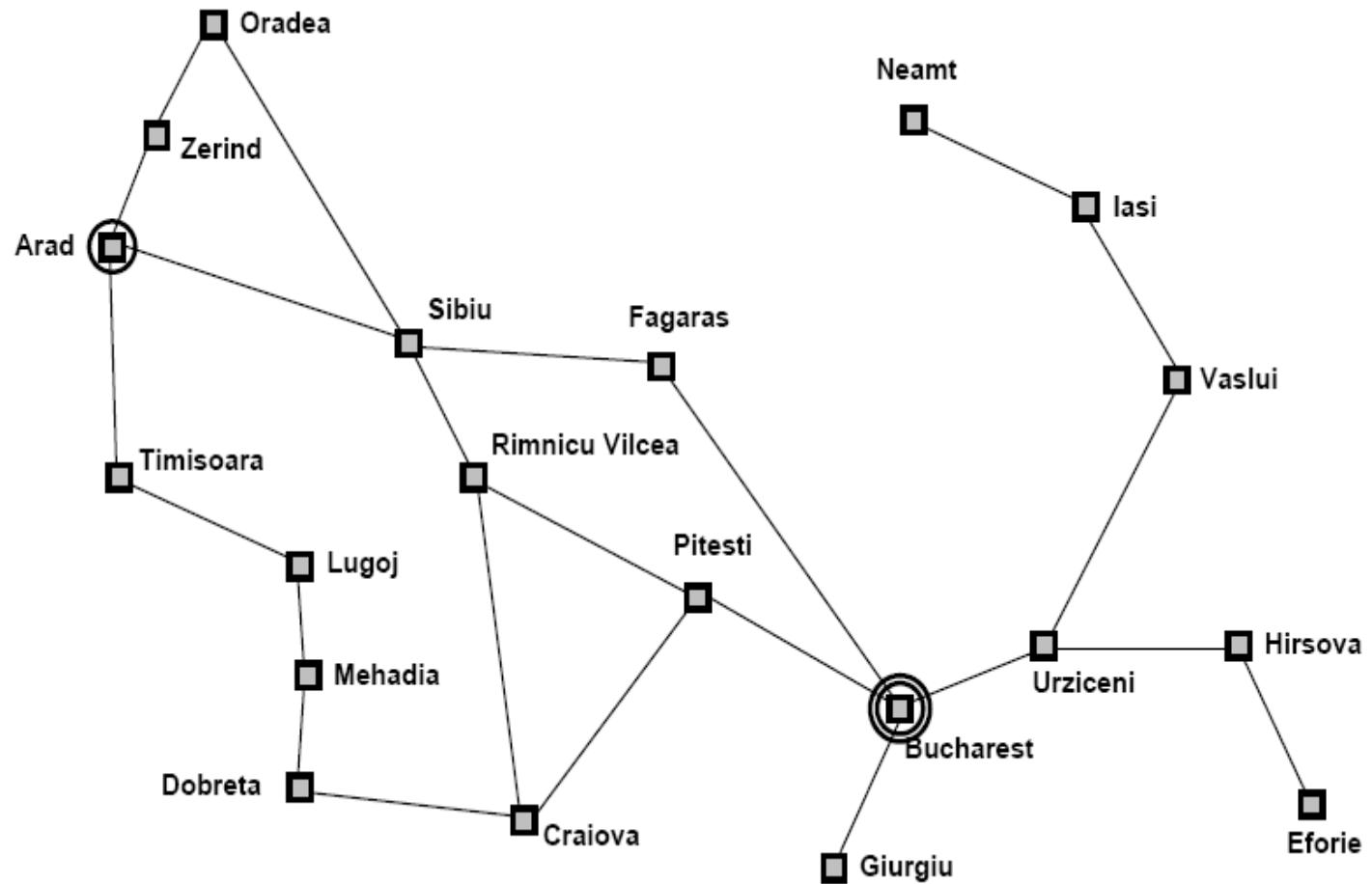


El problema del viaje por Rumanía:

- $F(n)$: asignar a cada nodo la distancia desde el origen + la distancia en línea recta al estado objetivo.
 - Se elige una ciudad como siguiente en el camino cuando la suma de la **distancia por carretera a la ciudad actual** más la **distancia aérea a la meta** sea la menor.

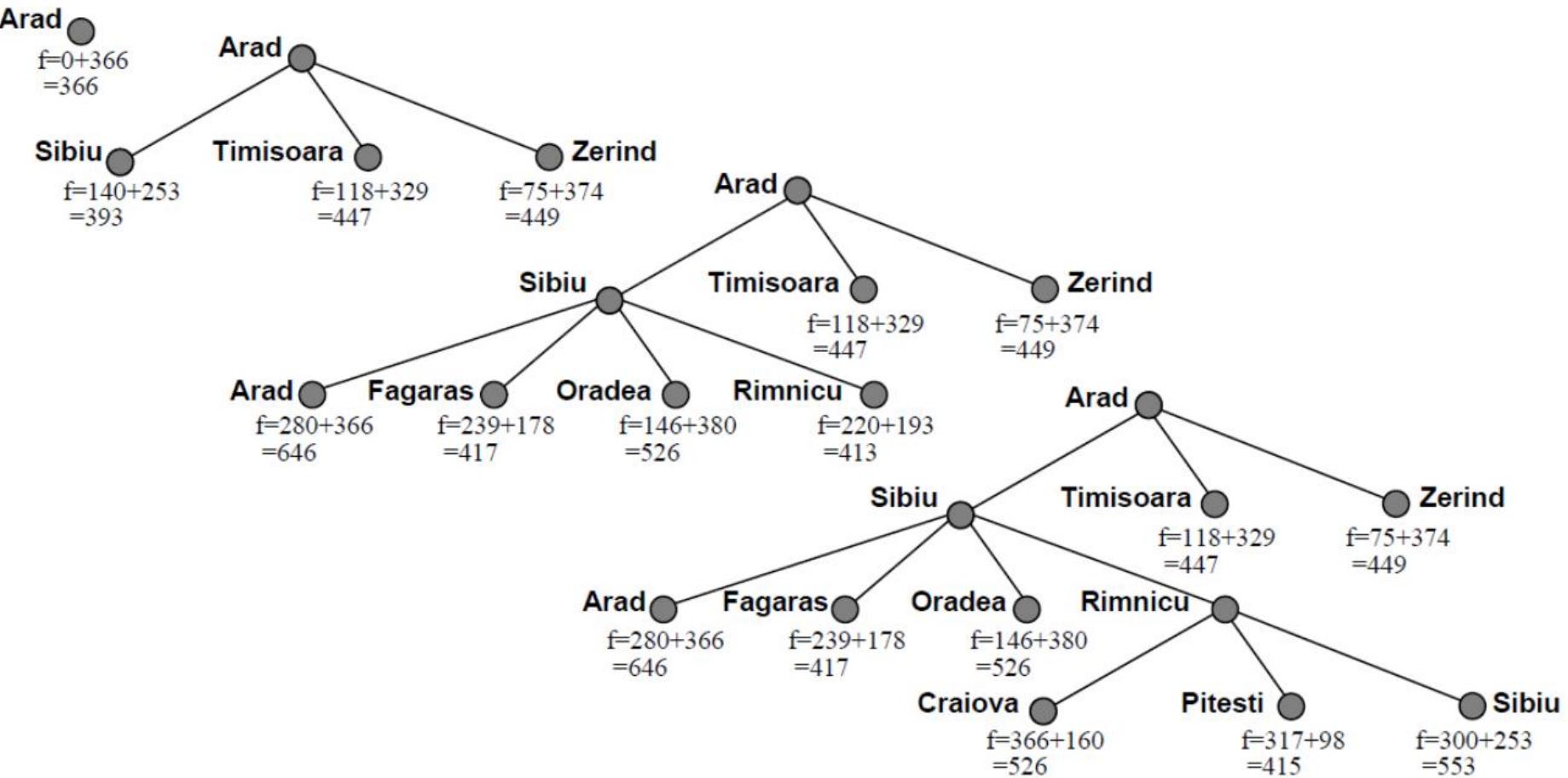
$$f(n) = g(n)+h(n) \text{ mínima}$$

3.2 Búsqueda A*



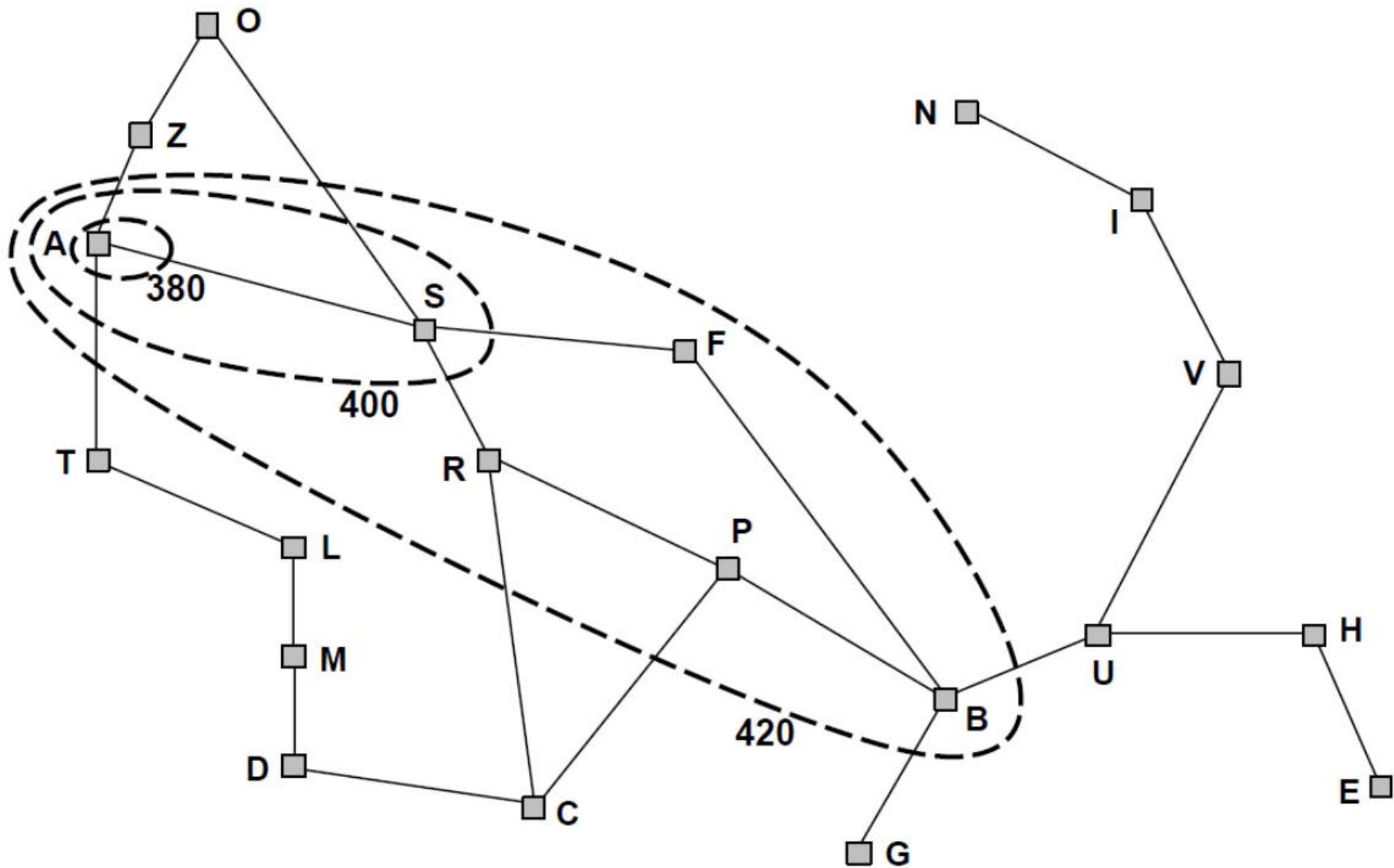
Distancias a Bucarest
en Km (línea recta)

3.2 Búsqueda A*





3.2 Búsqueda A*





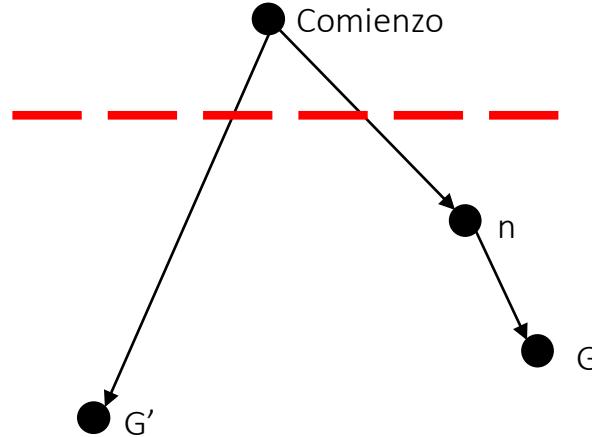
3.2 Búsqueda A*

- Si ...
 - h es admisible
 - El número de sucesores de n (b) es finito $\forall n$
 - $k(n_i, n_j) \geq 0$ en todo arco
- entonces A* es:
 - Completa: **Si**, si existe solución la encuentra
 - Complejidad tiempo: $O(b^d)$
 - Complejidad espacio: $O(b^d)$
 - Optima: **Si**
 - A* es además óptimamente eficiente
 - Ningún otro algoritmo óptimo expande menos nodos para cualquier heurístico
 - Lo hemos visto al discutir la consistencia de las heurísticas



3.2 Búsqueda A*

- Supongamos que se ha generado un objetivo subóptimo (G') y que está en la cola
- Sea n un nodo no expandido en el camino más corto al objetivo óptimo G



$f(G') = g(G')$ ya que $h(G')= 0$

$f(G) = g(G)$ ya que $h(G)= 0$

$f(G') > f(G)$ ya que G' no es óptimo

$f(G') \geq f(n)$ ya que h es consistente

Por lo que A* no expandirá G' antes de alcanzar G



3.2 Búsqueda A*

PROCEDIMIENTO A-STAR(Estado-inicial, Estado-Final)

ABIERTO = *ESTADO-INICIAL*

Hacer *CERRADO* vacío

BUCLE (Repetir el proceso mientras ABIERTO ≠ vacío)

1. NODO-ACTUAL = EXTRAE-PRIMERO(ABIERTO)
2. Poner NODO-ACTUAL en *CERRADO*
3. Si ES-ESTADO-FINAL(ESTADO(NODO-ACTUAL))
devolver CAMINO(NODO-ACTUAL)
4. Si no, FUNCION SUCESORES(NODO-ACTUAL)

4.1 Si SUCESOR ya está en *CERRADO*,

Si $g(\text{SUCESOR})$ es menor, insertar ordenadamente en *ABIERTO*

Actualizar coste y camino

4.2 Si SUCESOR ya está en *ABIERTO*,

Si $g(\text{SUCESOR})$ es menor, actualizar coste, posición y camino

4.3 GESTIONA-COLA(ABIERTO,SUCESORES) Añadir *SUCESORES* a *ABIERTO*
en orden creciente de $f(n)$

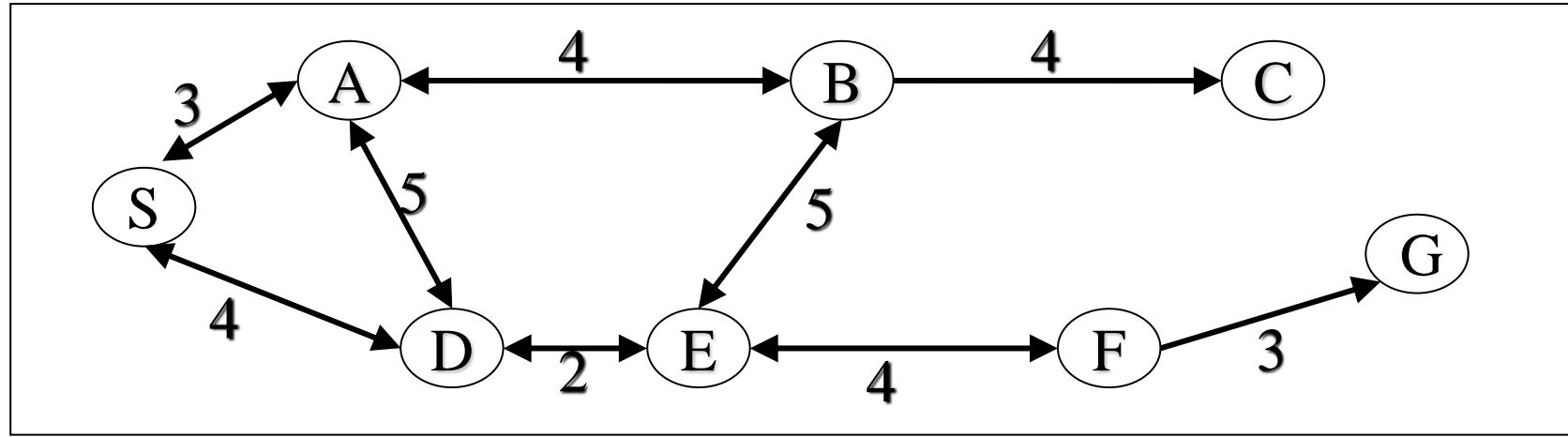
FIN DE BUCLE

Devuelve FALLO ☹



3.2 Búsqueda A*

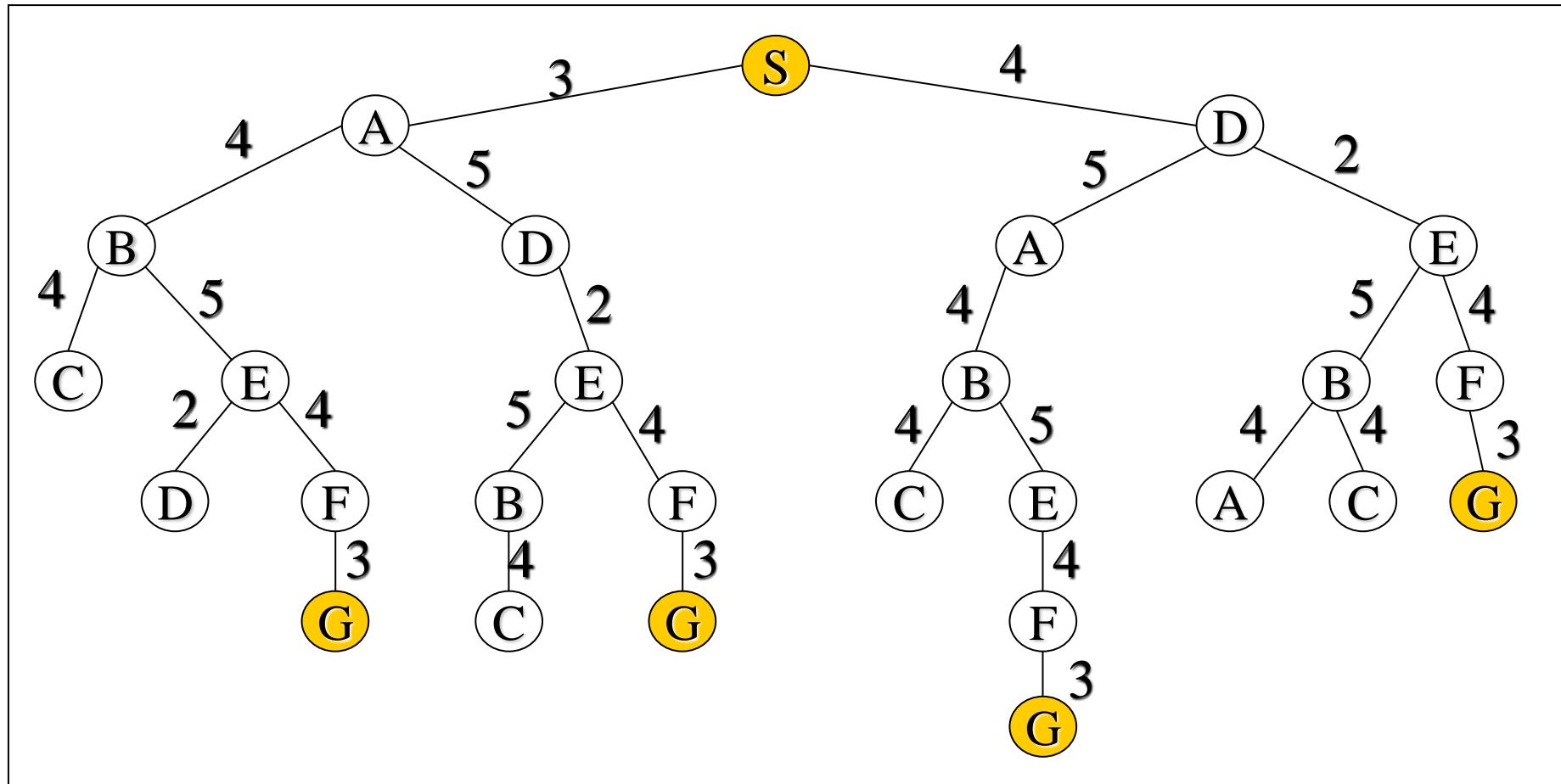
- Ejemplo: sea el siguiente grafo





3.2 Búsqueda A*

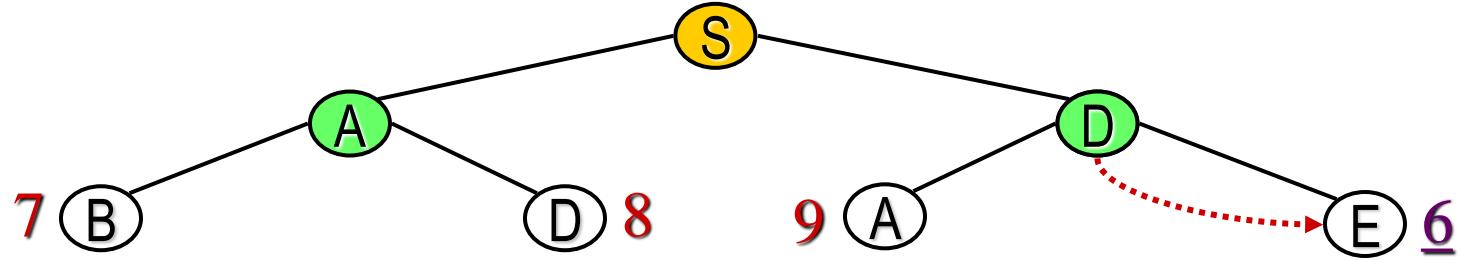
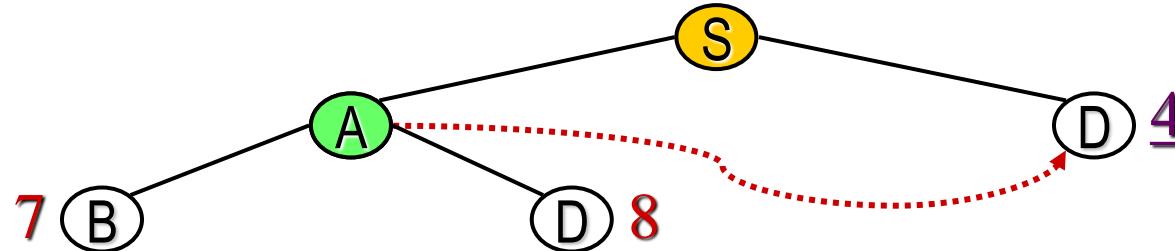
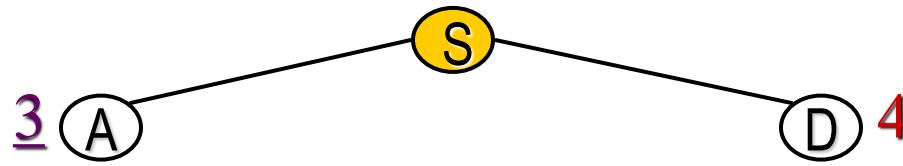
- Que genera este árbol



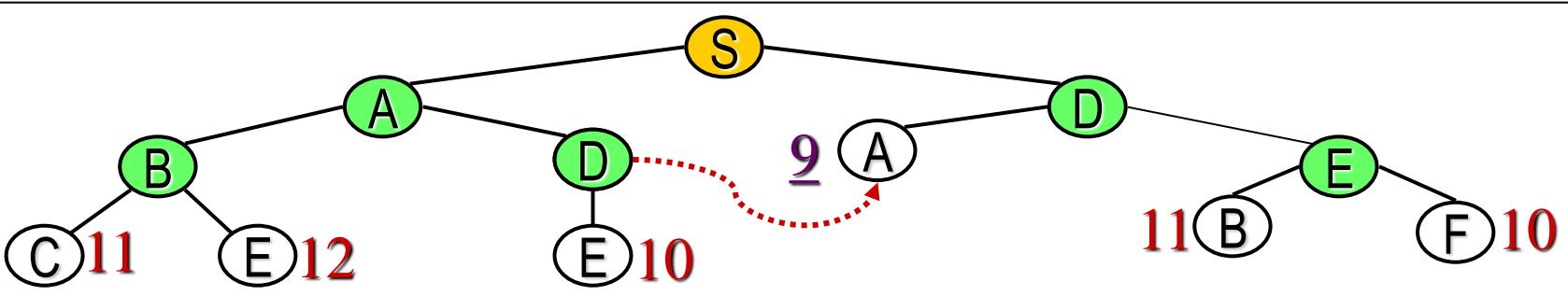
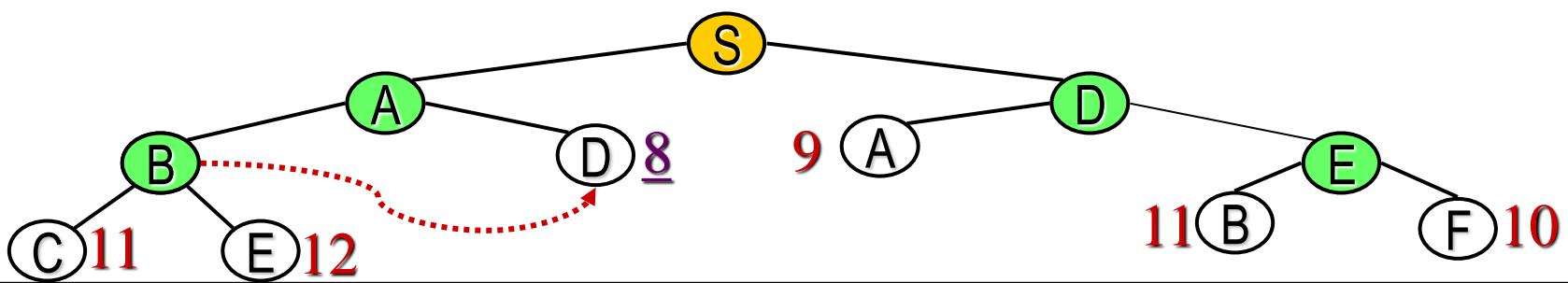
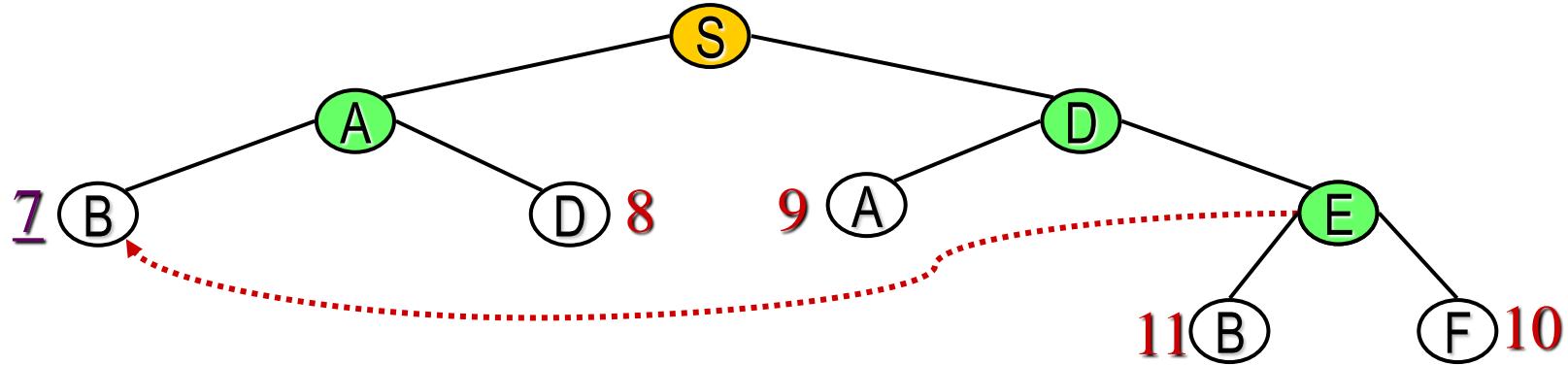


3.2 Búsqueda A*

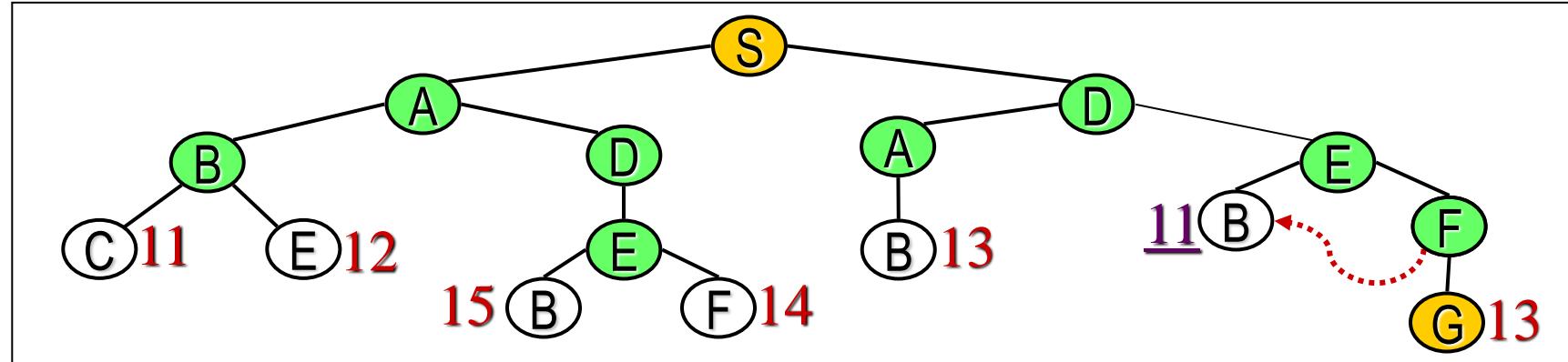
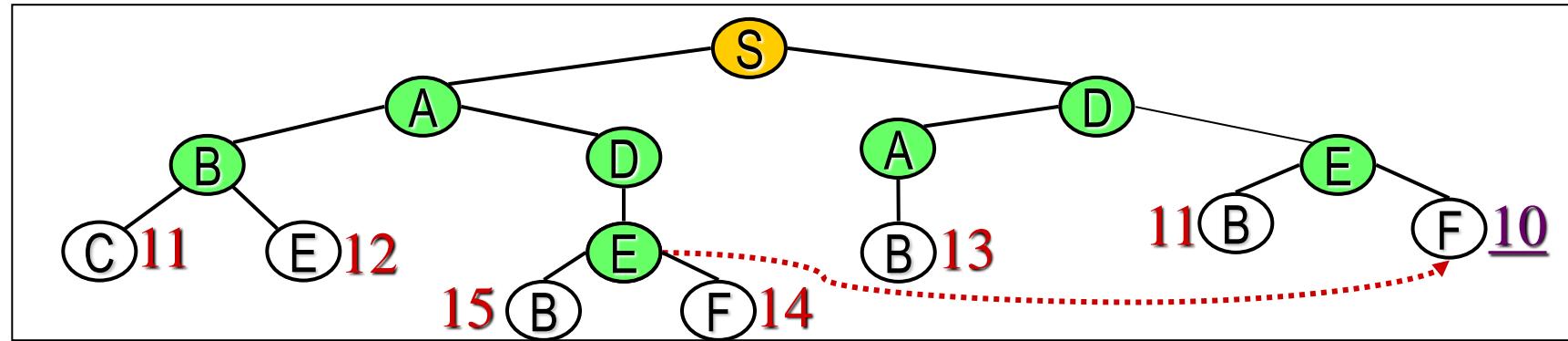
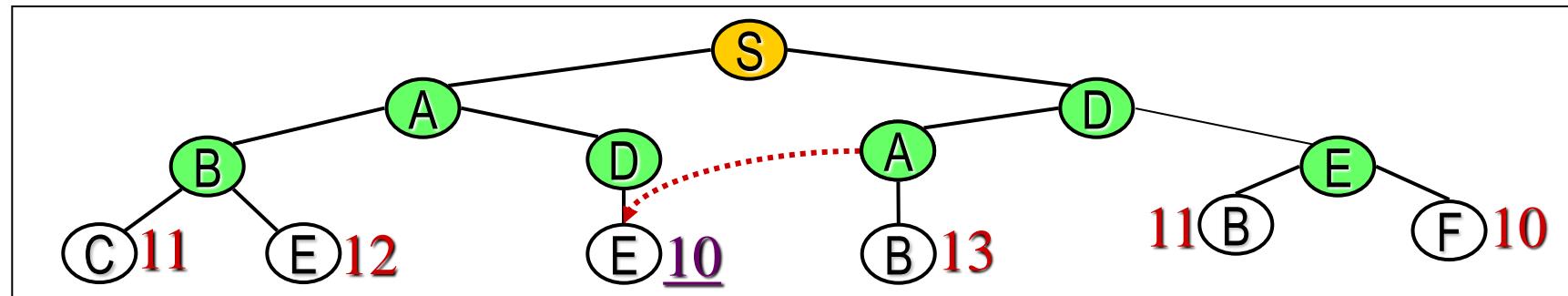
- Resolución por **Búsqueda de Coste Uniforme** (**SOLO $g(n)$**)



3.2 Búsqueda A*

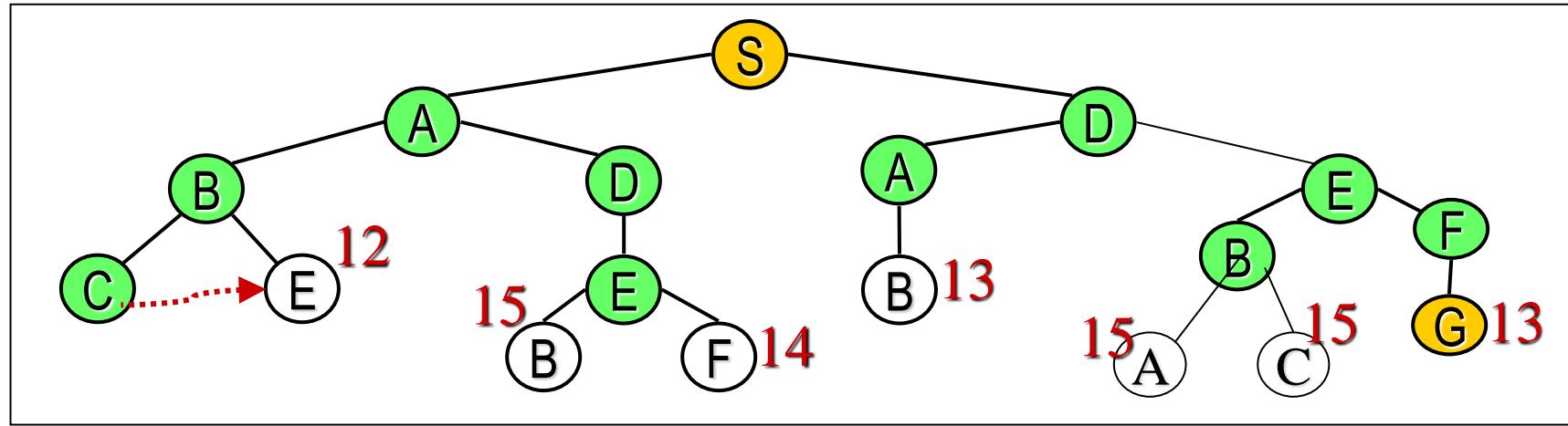
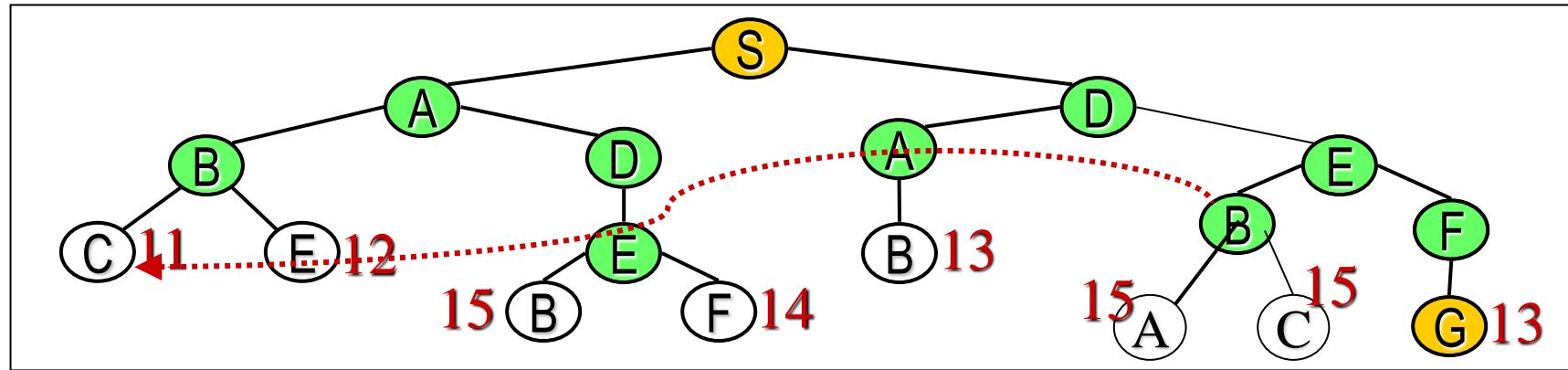


3.2 Búsqueda A*

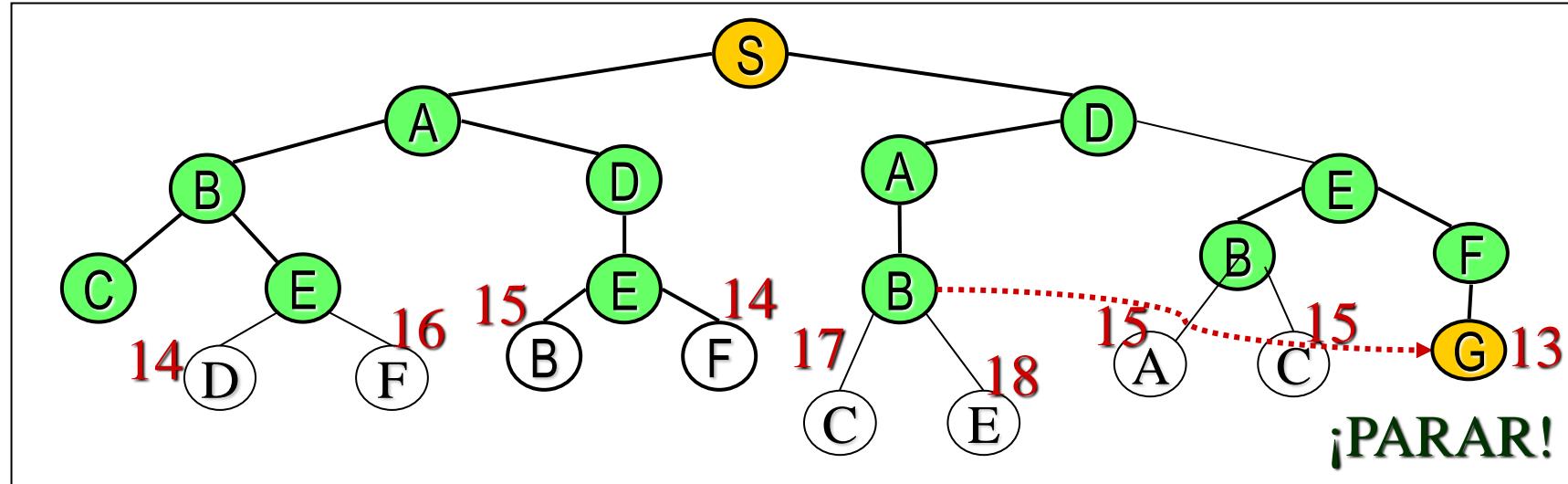
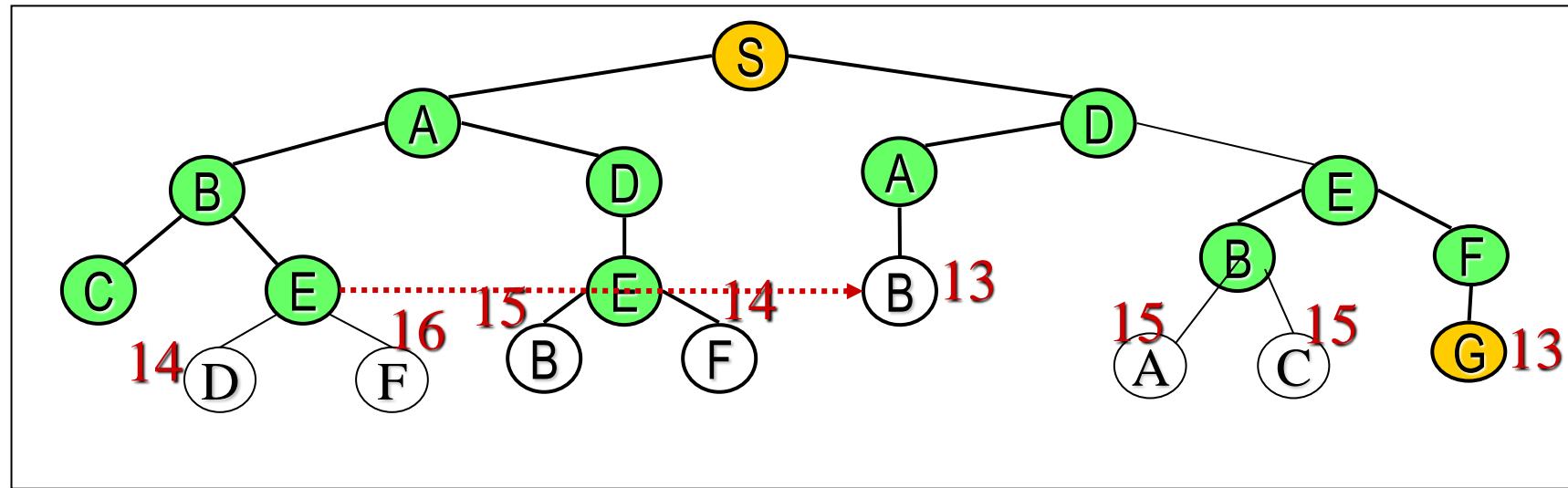




3.2 Búsqueda A*



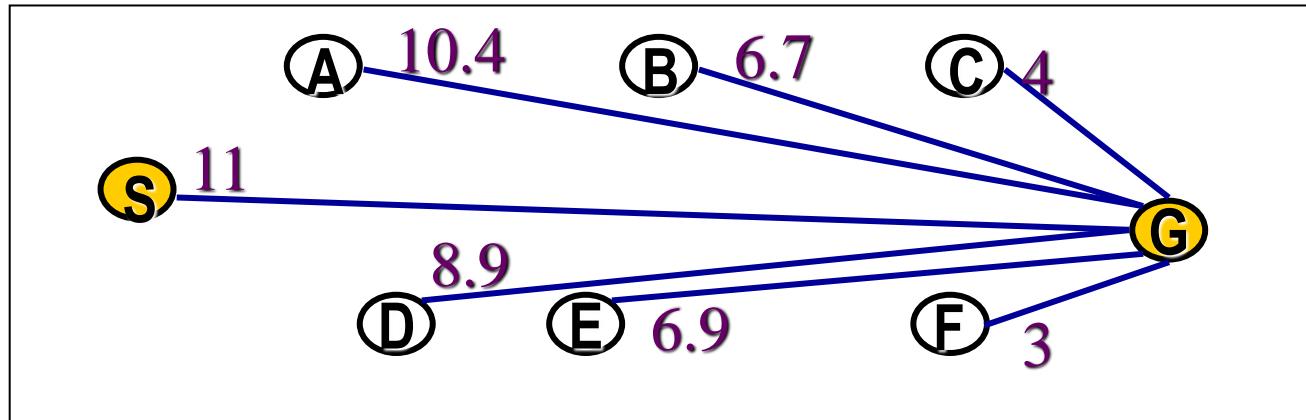
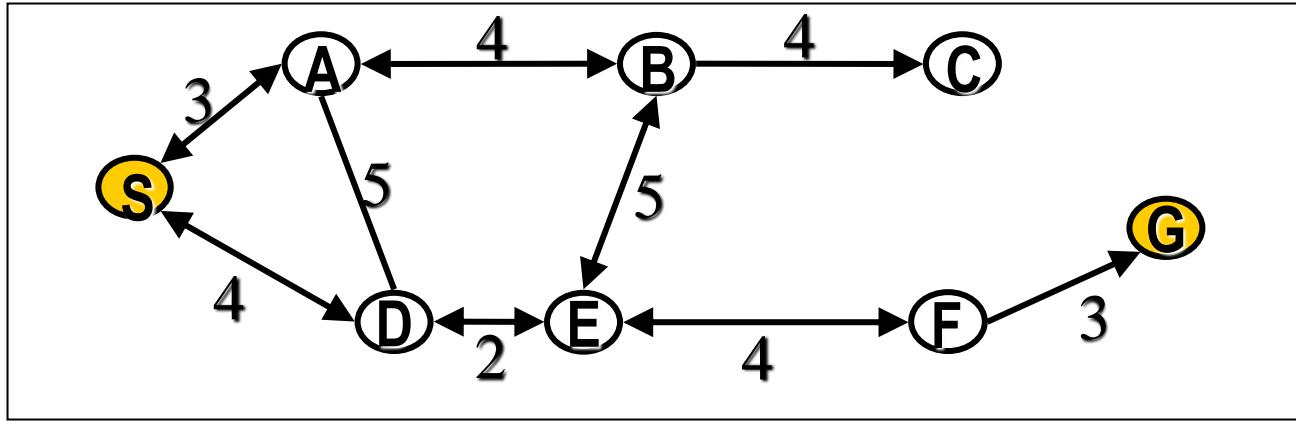
3.2 Búsqueda A*



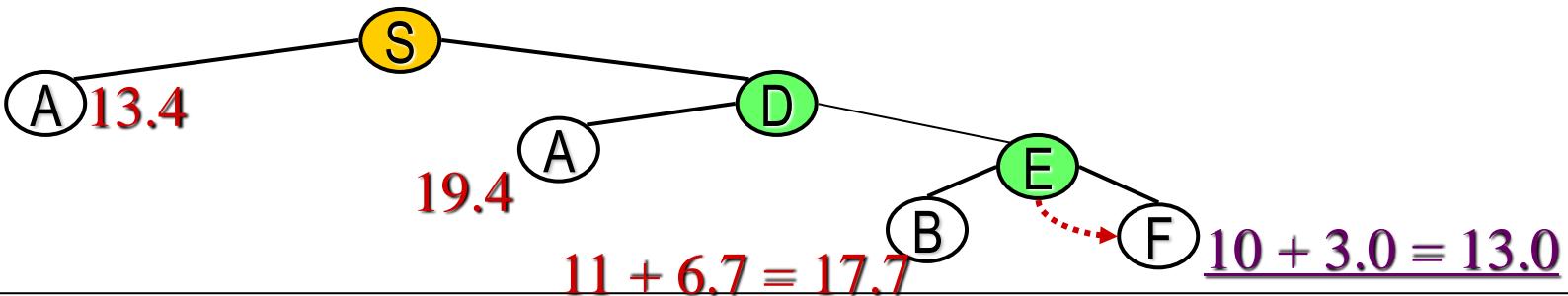
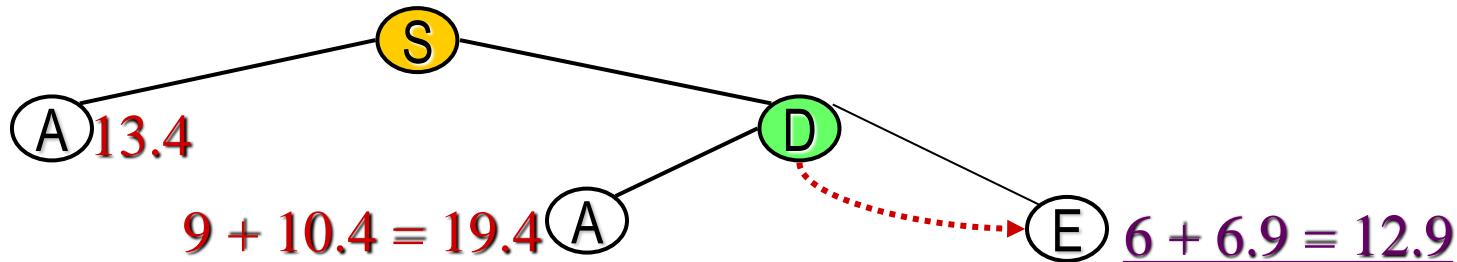
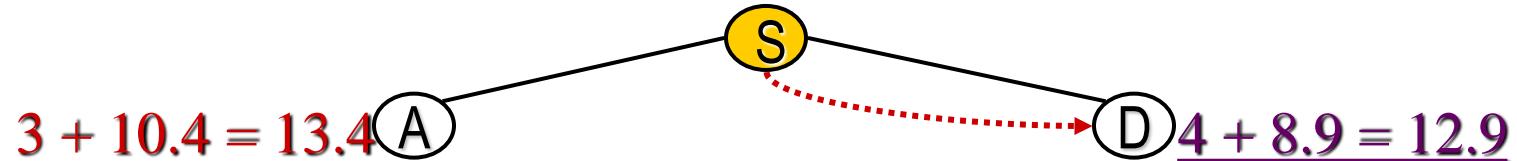


3.2 Búsqueda A*

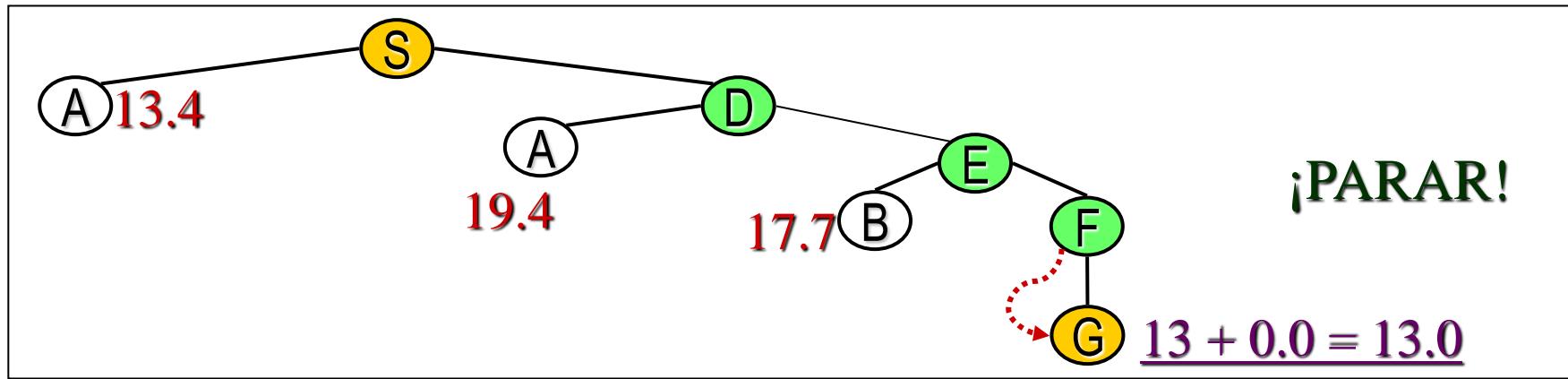
- Reconsiderar el problema incluyendo la heurística h_{DLR} (tenemos en cuenta $g(n) + h(n)$)



3.2 Búsqueda A*



3.2 Búsqueda A*



Ver video: "Comparando Algoritmos: A* vs Dijkstra, en el mapa de la ciudad." en YouTube

<https://youtu.be/oMgfGkFSgI0?si=7GfuIuhVC1pMdAji>

3.3 Variaciones de A*



- Algoritmos que mejoran A* acotando la memoria:
 - IDA*: A* con Profundidad Iterativa (*Iterative Deepening A**)
 - MA*: A* con Memoria acotada (*Memory Bounded A**)
 - SMA*: A*M Simplificada (*Simplified Memory Bounded A**)
 - Si al generar un sucesor falta memoria, se libera el espacio de los nodos de *abiertos* menos prometedores
 - RTA*: A* en Tiempo Real (Korf, 1988)

3.3 Variaciones de A*

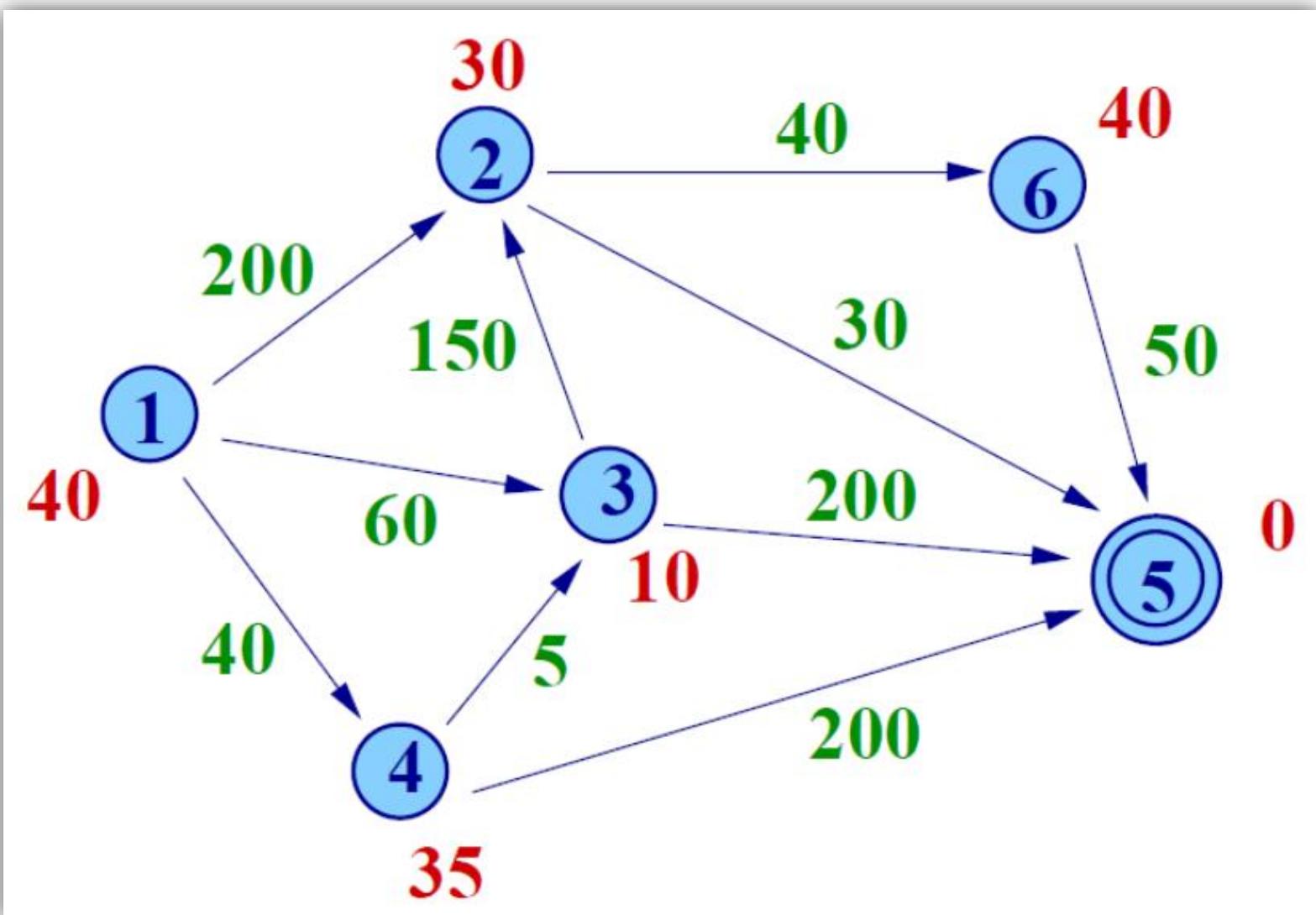


IDA*: A* con Profundidad Iterativa (Korf, 1985)

- Similar a la búsqueda ciega en Profundidad Iterativa
- Expande todos los nodos cuyo coste $f(n)$ no excede un determinado valor
- η (*coste de corte*): valor mínimo de la función de coste en todos los nodos visitados pero NO expandidos
 - Iteración 1: $\eta_1 = h_0(\text{nodo inicial})$
 - Expandir nodos según A* hasta que $f(\text{nodos sucesores}) > \eta$
 - Si no es META,
 - Nueva iteración
 - Nuevo η sobre el conjunto de nodos todavía no expandidos
- Repetir iteración

$$\eta = \min_{i=1,n} \{f(i)\} = \min_{i=1,n} \{g(i) + h(i)\}$$

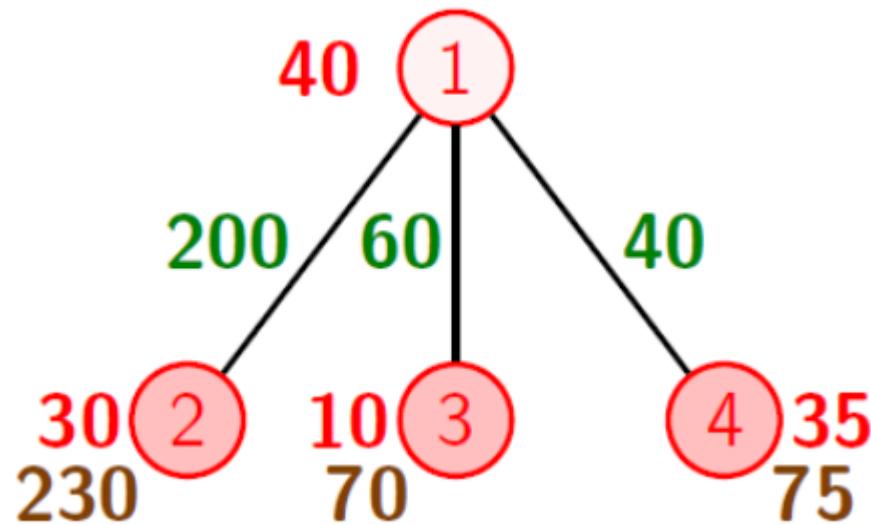
3.3 Variaciones de A*





3.3 Variaciones de A*

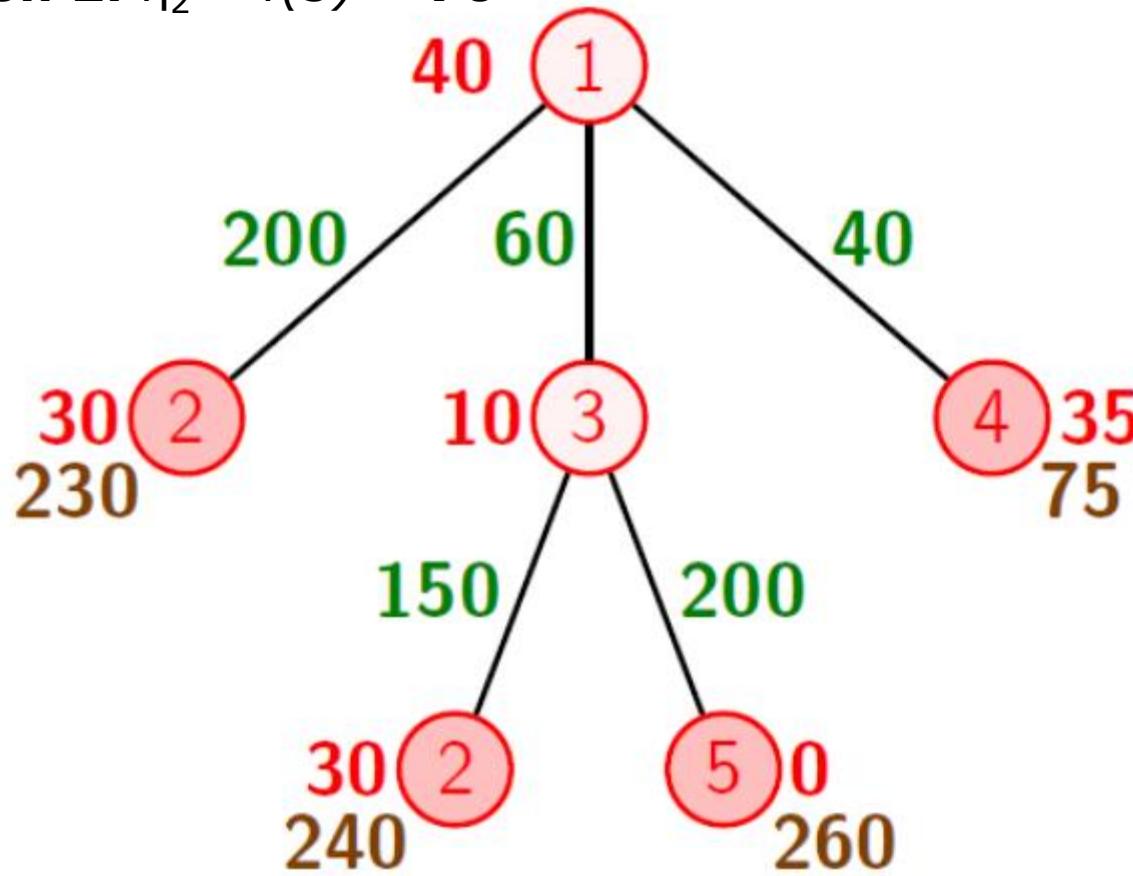
Iteración 1: $\eta_1 = h(1) = 40$





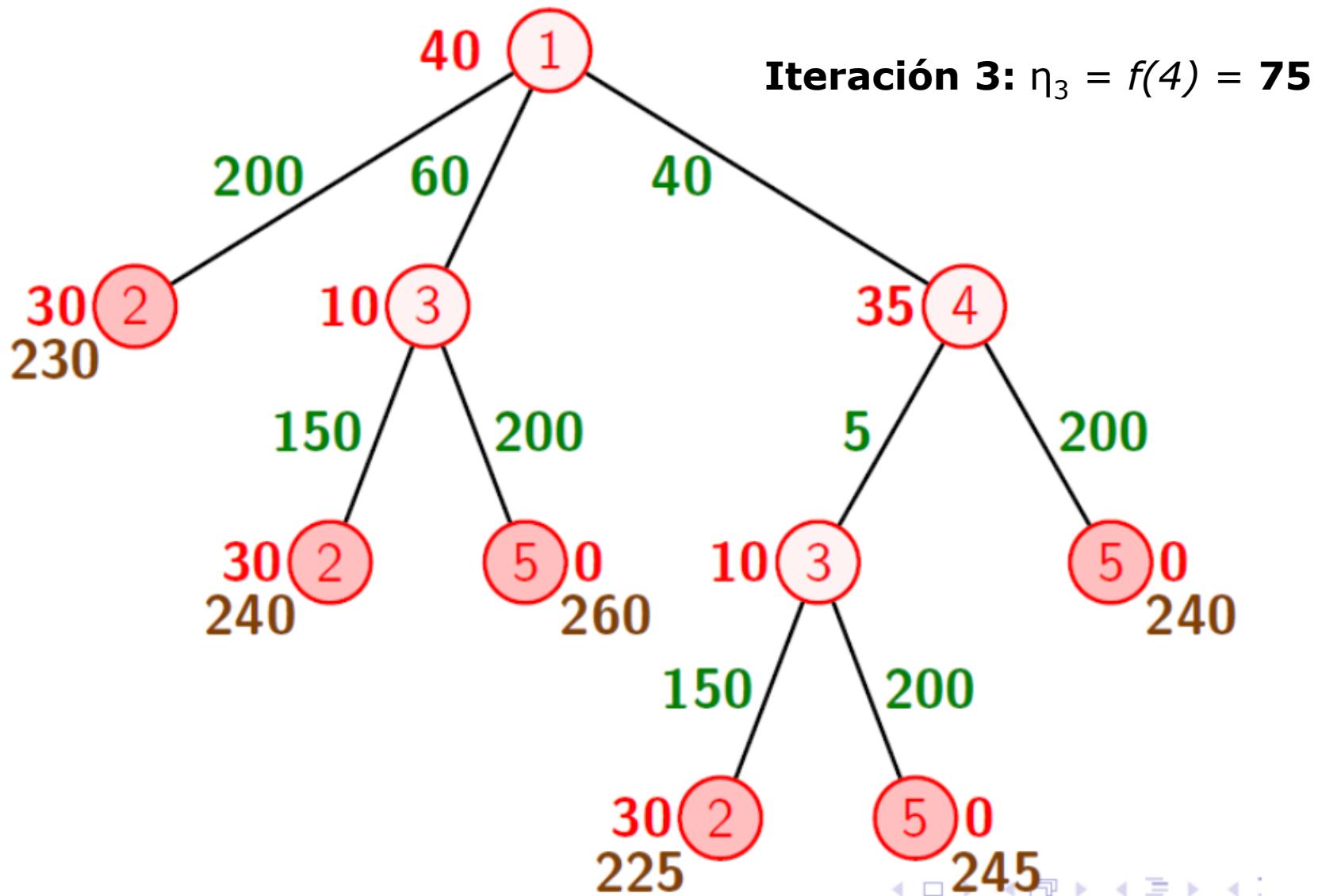
3.3 Variaciones de A*

Iteración 2: $\eta_2 = f(3) = 70$



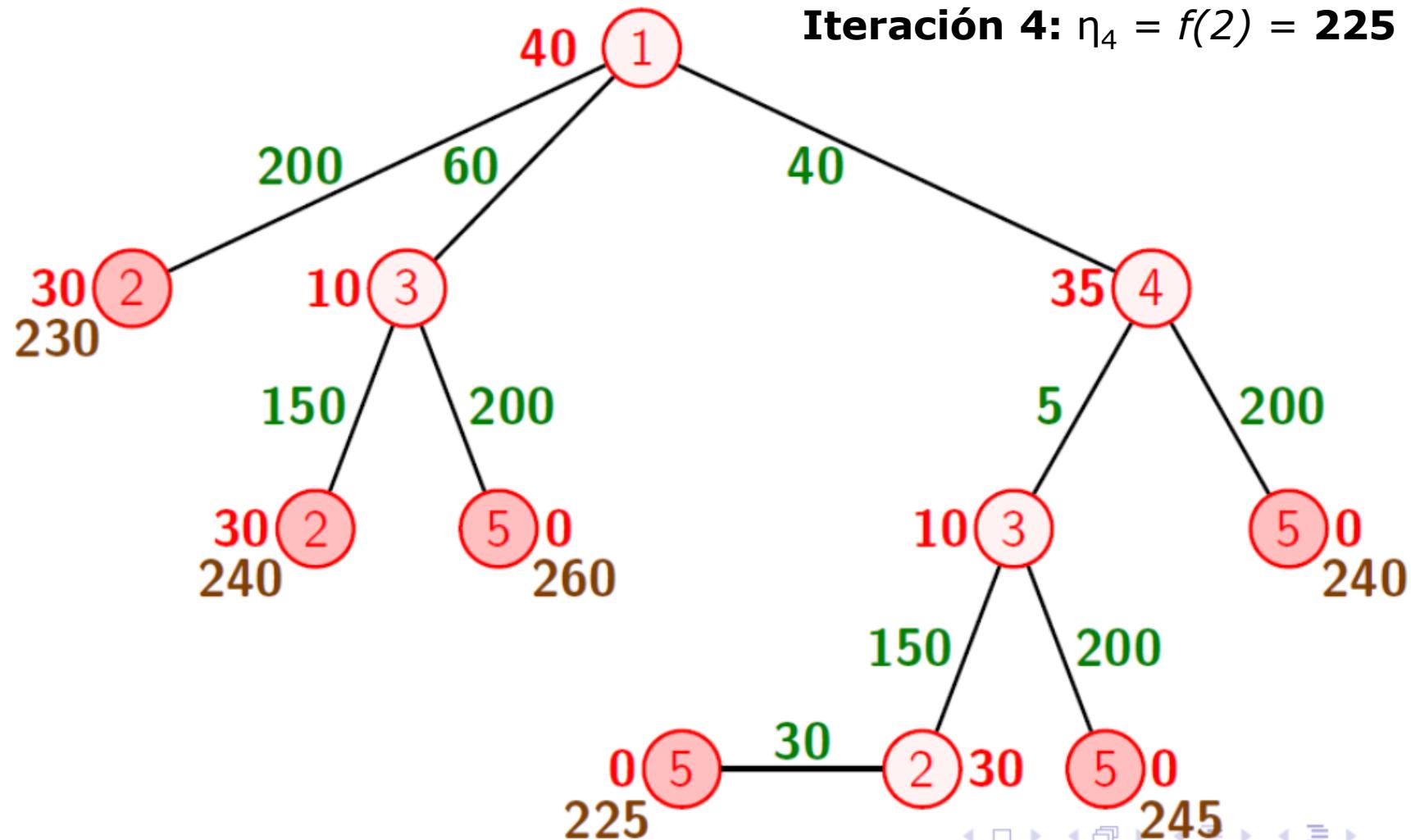


3.3 Variaciones de A*





3.3 Variaciones de A*



3.3 Variaciones de A*



- Propiedades
 - **Compleitud:** es completo (encuentra solución si existe)
 - **Complejidad tiempo:** $O(b^d)$ Exponencial
 - **Complejidad espacio:** $O(b*d)$ lineal en la profundidad del árbol de búsqueda
 - **Optima:** **Si.** Es admisible y, por lo tanto, encuentra la solución óptima
 - Aunque pudiera parecer lo contrario, el número de re-expansiones es solo mayor en un pequeño factor que el número de expansiones de los algoritmos PEM
- Fue el primer algoritmo que resolvió óptimamente 100 casos generados aleatoriamente en el 15-puzzle



1. Introducción
2. Funciones heurísticas
3. Búsquedas “primero el mejor”
 1. Búsqueda avara
 2. Búsqueda A*
 3. Variaciones de A*
4. Búsquedas iterativas
 1. Hill Climbing
 2. Simulated Annealing

4. Búsquedas iterativas



- Se usan cuando solo interesa el objetivo final
 - NO importa el camino ni su coste (ni se calcula!!)
 - 8-reinas, planificación, rutas...
- Reemplazan a las técnicas de búsqueda exhaustiva de forma eficiente
 - Comienzan con la configuración completa y hacen modificaciones para mejorar la calidad de la solución
 - No suelen almacenar caminos y buscan desde el estado actual hacia vecinos → *algoritmos de búsqueda local*
 - *Lo típico es que no encuentran la mejor solución, pero pueden encontrar una solución aceptable.*
- Ventajas
 - usan poca memoria
 - funcionan en problemas continuos inmensamente grandes

4. Búsquedas iterativas



- Uso en problemas de optimización

Búsqueda de los valores óptimos para los parámetros de un sistema que minimicen la función de coste

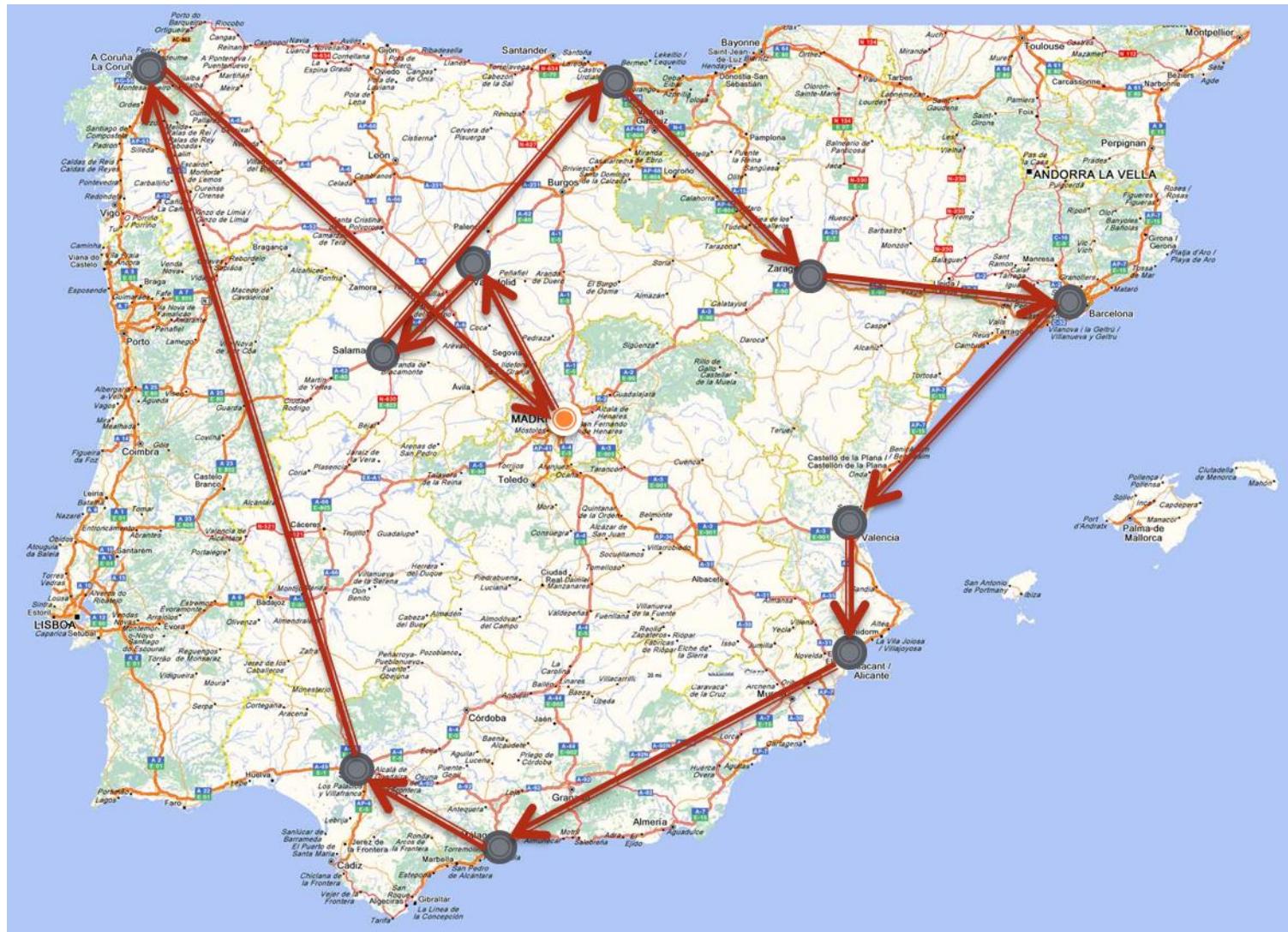
- Buscan valles o picos en el paisaje formado por la función de coste

- Ejemplo: TSP

- Un comerciante debe recorrer N ciudades, sin repetir ninguna, y volver a la ciudad de partida, en la mínima distancia.
- El número de posibles rutas viene dado por permutaciones sin repetición ($N!$)
- Para 2 ciudades (A, B), podemos hacer 2 recorridos:
 - (1) A → B → A
 - (2) B → A → B
- El resultado depende de la ciudad de partida



4. Búsquedas iterativas



4. Búsquedas iterativas

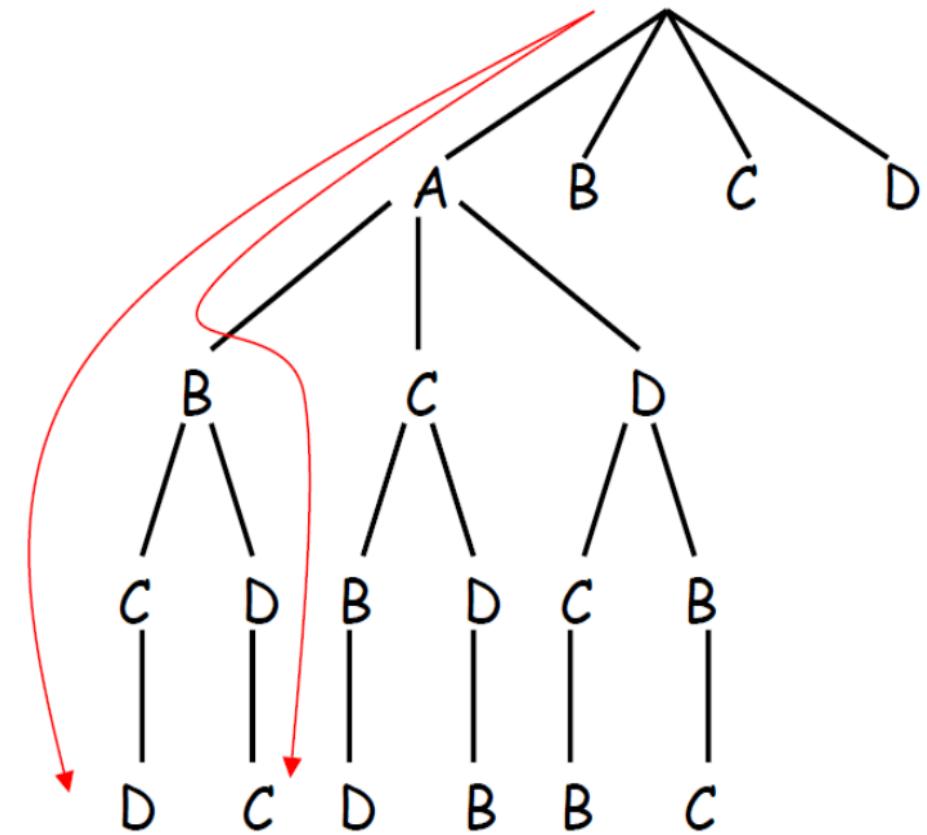
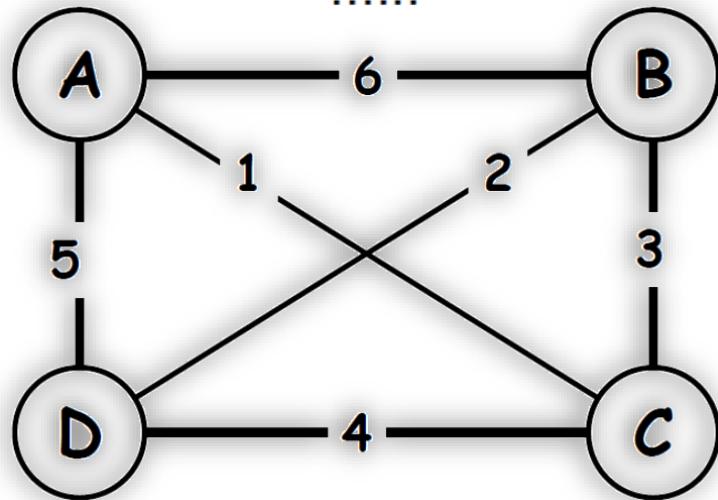




4. Búsquedas iterativas

- La generación de las posibles soluciones se lleva a cabo por orden alfabético de ciudades

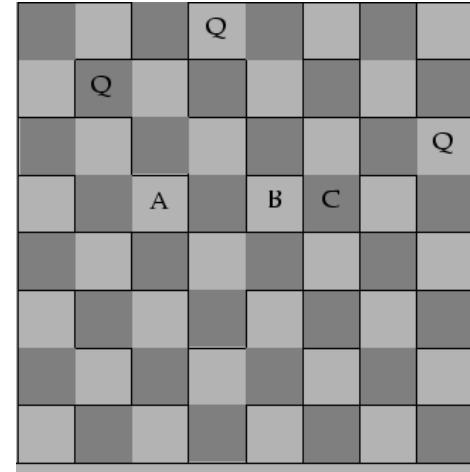
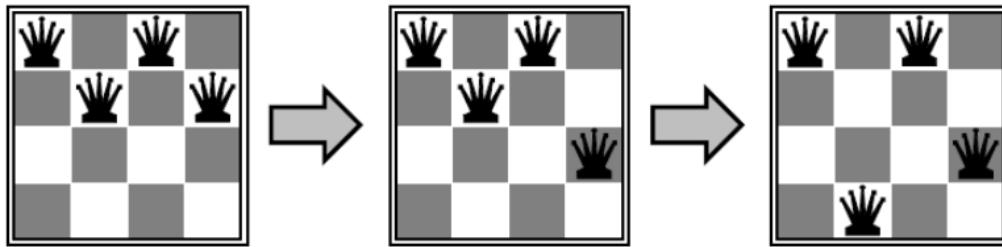
1. A - B - C - D
 2. A - B - D - C
 3. A - C - B - D
 4. A - C - D - B
-



4. Búsquedas iterativas



- Otro ejemplo: Problema de las n damas



Posibles candidatos de heurísticas:

1. Preferir colocar damas que dejen el mayor número de celdas sin atacar.

En el ejemplo: $h_1(A)=12$, $h_1(B)=13$, $h_1(C)=13$.

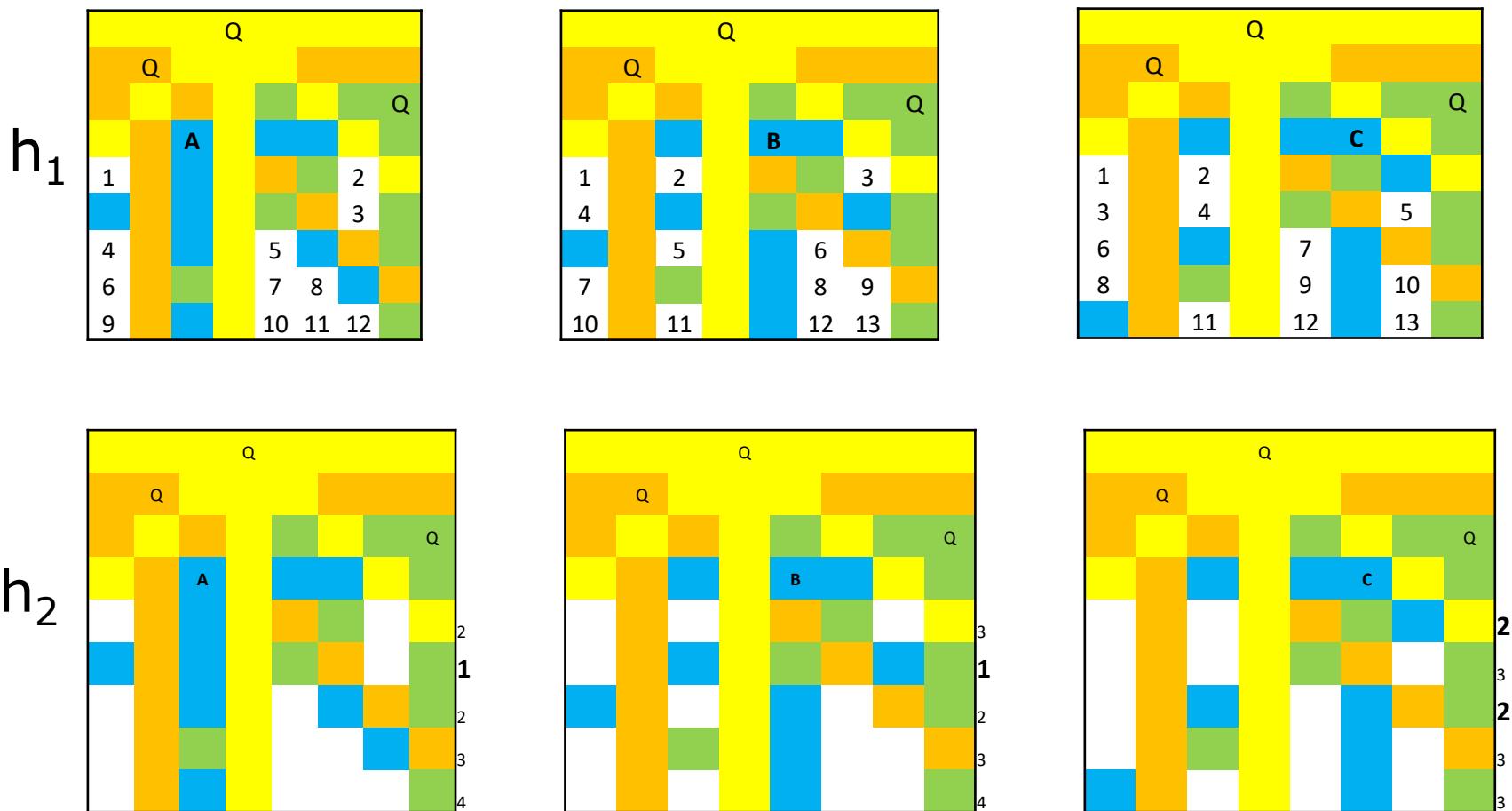
2. Ver cuál es el menor número de celdas no atacadas en cada renglón y escoger la que su número menor sea mayor.

En el ejemplo: $h_2(A)=1$, $h_2(B)=1$, $h_2(C)=2$



4. Búsquedas iterativas

- Heurísticas n-damas



La h_2 permite detectar *caminos sin salida*



4. Búsquedas iterativas

■ Tipos de búsquedas iterativas

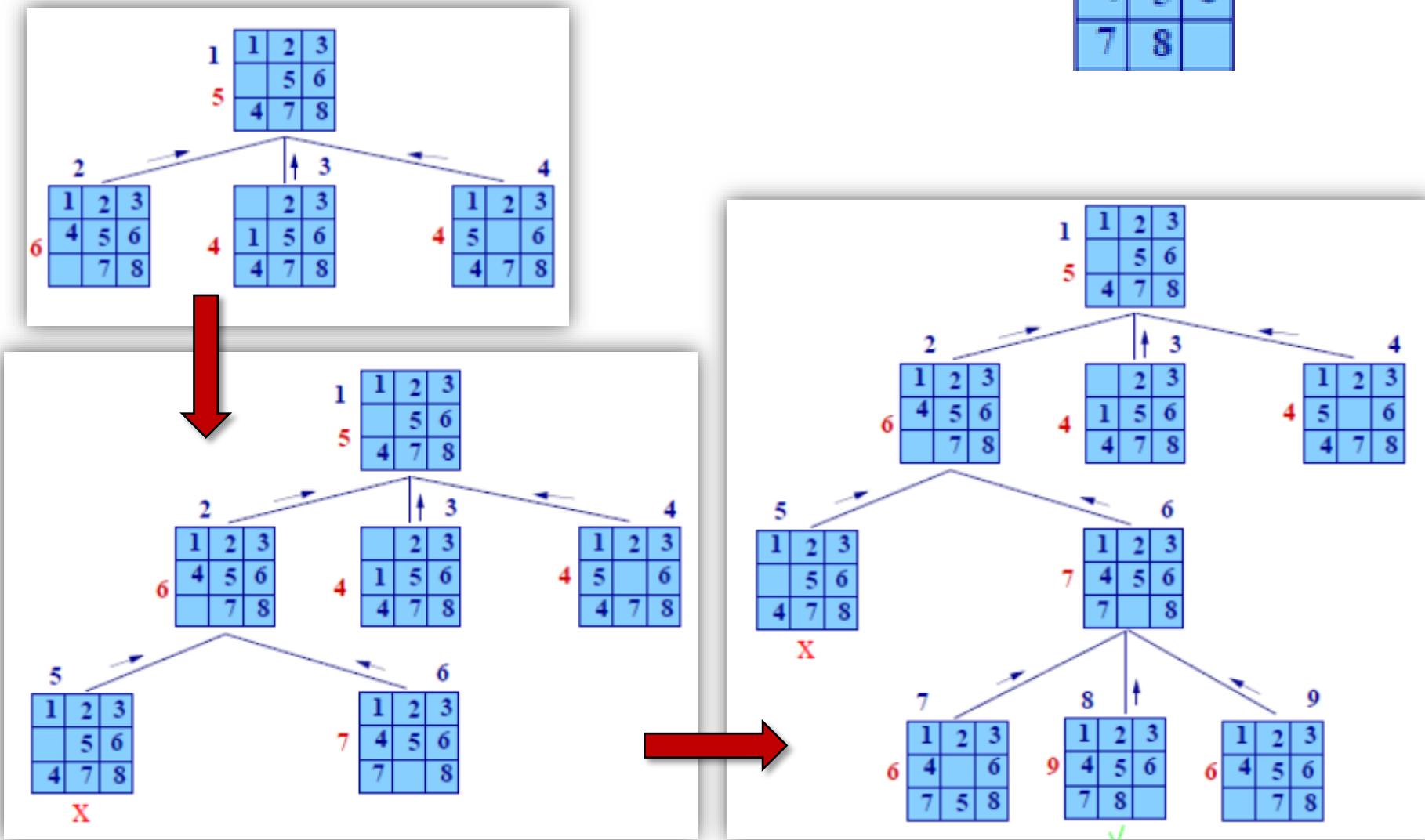
- Optimización evolutiva
 - Algoritmos genéticos
 - Redes neuronales
 - Redes de Hopfield
 - Ya hemos visto estas búsquedas NO HEURISTICAS al ver la IA SUBSIMBÓLICA
-
- Ascenso de gradiente
 - Hill climbing
 - Simulated Annealing
 - Métodos de MonteCarlo
 - Máquina de Boltzmann



4.1 Hill Climbing

- Se llaman de escalada (o de ascensión a la colina) porque tratan de elegir en cada paso un estado cuyo valor heurístico sea mejor que el del estado activo en ese momento
 - Avanza al sucesor que maximice la función sucesor
 - “**muévete siempre a un estado mejor si es posible**”
- Búsqueda “avara” local (no respecto al estado final)
- Suele funcionar bien, pero tiene problemas con la terminación
 - El proceso termina cuando en un estado dado, al aplicar todos los operadores ninguno de los estados resultantes es mejor
 - Puede atascarse en **máximos locales**, según el estado inicial

4.1 Hill Climbing

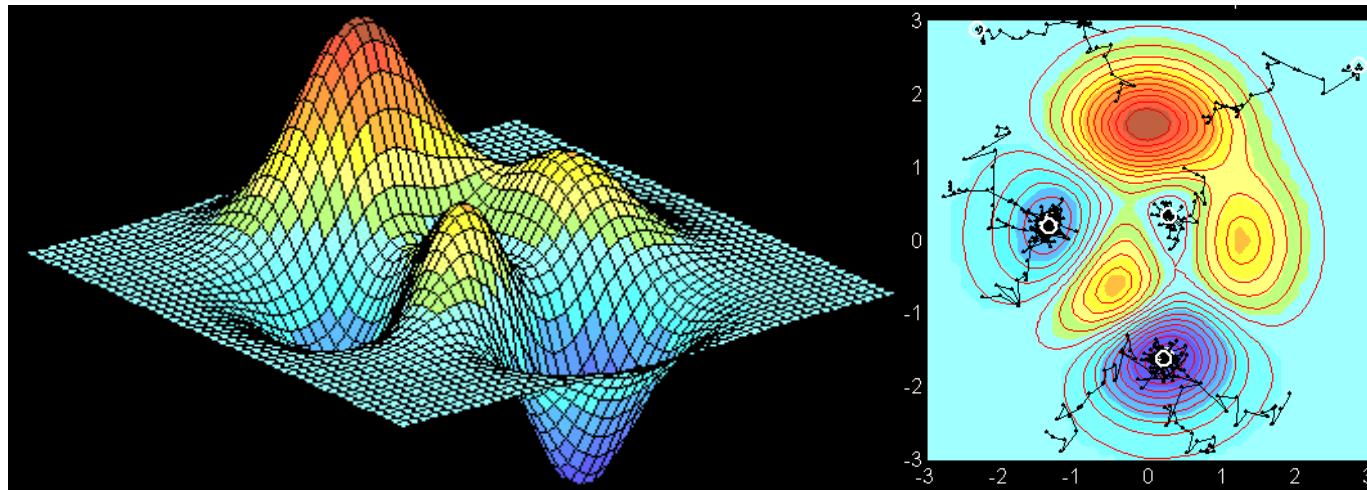
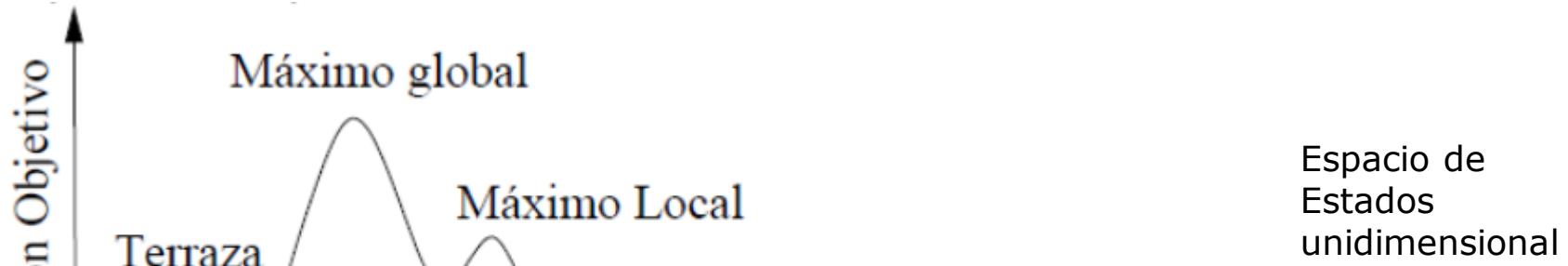




4.1 Hill Climbing

- Propiedades
 - **Compleitud:** no tiene por qué encontrar la solución
 - **Admisibilidad:** no siendo completo, aún menos será admisible
 - **Eficiencia:** rápido y útil si la función es monótona (de)creciente
- Problemas
 - **Máximos (o mínimos) locales:** pico que es más alto que cada uno de sus estados vecinos, pero más bajo que el máximo global
 - **Mesetas:** zona del espacio de estados con función de evaluación plana
 - **Crestas:** zona del espacio de estados con varios máximos (mínimos) locales

4.1 Hill Climbing



4.1 Hill Climbing



1	2	3
8		4
7	6	5

objetivo

3	2	1
8		4
7	6	5

E_{máximo local}

Máximo local

Todos los movimientos empeoran el valor de la función heurística.

1	2	3
6	7	4
	8	5

E_{meseta}

Meseta

Todos los movimientos dejan igual el valor de la función heurística.

4.1 Hill Climbing



- Variantes (mejoran el problema del óptimo local)
 - Estocástica (aleatorio entre ascendentes → retroceso)
 - De primera opción (al azar hasta encontrar uno mejor)
 - Reinicio aleatorio
- Alternativas
 - Algoritmos Genéticos
 - Simulated Annealing



4.2 Simulated Annealing

- Desarrollado en 1983 para modelado de procesos físicos
- Basado en el proceso metalúrgico de recalentamiento o “templado” (como las espadas)
- Busca alcanzar el óptimo absoluto
 - La idea es escapar de los máximos locales permitiendo **movimientos “incorrectos”** (saltos)
 - Pero reduciendo gradualmente su tamaño y frecuencia
 - Acabando en un hill climbing normal.
- Combina ascensión de colinas con movimientos aleatorios
- Busca unir eficacia y completitud

4.2 Simulated Annealing



- Se utiliza como parámetro la **temperatura** T del proceso
 - “calienta” → aumenta el número y amplitud de los saltos aleatorios
 - “enfría” → ascensión
 - Si la temperatura se reduce suficientemente despacio, se alcanza la solución óptima
- Probabilidad de salto: función de aceptación $h(\Delta E, T)$

$$h = \frac{1}{1 + e^{\frac{\Delta E}{cT}}}$$

- *Distribución de Boltzmann* donde $\Delta E = f(n_{\text{próximo}}) - f(n)$
 - Si $\Delta E < 0$ se acepta el movimiento ya que conduce a un estado de menos energía
 - Si $\Delta E > 0$ se puede aceptar el movimiento con más probabilidad cuanto mayor sea T



Universidad
Francisco de Vitoria
UFV Madrid

Ingeniería del Conocimiento

Tema 5:
Búsqueda Informada (II)



Objetivos del tema

- Ubicación
 - Unidad 2: **BUSQUEDA EN ESPACIO DE ESTADOS**
 - *Tema 5: Búsqueda Informada: Heurísticas (II)*
- Objetivos generales
 - Entender el concepto de *árbol de búsqueda* cuando se trata de un juego con adversarios
 - Conocer los *distintos tipos de juegos* y su facilidad de resolución
 - Entender los algoritmos de búsqueda *minimax* con sus variaciones y mejoras: *Decisiones imperfectas* y *poda alfa-beta*
 - Saber *aplicar de cada método* en función de la completitud y *complejidad* espacial y temporal



1. Búsqueda con adversarios
2. Búsqueda Minimax
 1. Minimax con decisiones imperfectas
 2. Poda Alfa-Beta
3. Otros tipos de juegos
4. Estado del arte en problemas de juegos



- 1. Búsqueda con adversarios**
- 2. Búsqueda Minimax**
 1. Minimax con decisiones imperfectas
 2. Poda Alfa-Beta
- 3. Otros tipos de juegos**
- 4. Estado del arte en problemas de juegos**



1. Búsqueda con adversarios

- Búsqueda en un entorno hostil, competitivo e impredecible donde interviene al menos un adversario contrario a nuestros objetivos
 - Los movimientos de un jugador, por sí solos no aseguran la victoria: es necesaria una estrategia de oposición
 - El tiempo disponible para cada movimiento (reglas de juego) impone soluciones aproximadas (no sirve la fuerza bruta)
 - No se puede calcular exactamente las consecuencias de cada movimiento → hay que tomar una decisión sin la certeza de que sea la correcta
- Los problemas de búsqueda con adversario (o de *conflicto de intereses*) se denominan **juegos**. La resolución del problema es la estrategia para ganar el juego
 - Estrategia: búsqueda en un espacio de billones de nodos con profundización selectiva



1. Búsqueda con adversarios

- ¿Por qué gustan tanto en IA?
 - Divertidos
 - Difíciles (el ajedrez tiene un factor de ramificación medio de 35 y pueden sucederse 50 movimientos de cada jugador)
 - Fáciles de formalizar y con un número pequeño de acciones
- Los juegos son para la IA como la F1 para la ingeniería del automóvil*
- Programa con muy buen rendimiento: ajedrez
 - Importante: desde 1956, el ajedrez era un objetivo para IA
 - Deep Blue gano a Kasparov en 1997



1. Búsqueda con adversarios

- Tipos de juegos:
 - Información completa
 - **Deterministas**: sin azar (4-en-rayo, ajedrez, damas)
 - **No deterministas**: con azar (backgammon, parchís, monopoly...)
 - Información incompleta con azar (juegos de cartas, dominó)
- De cara a simplificar, consideraremos los juegos con estas características:
 - **Información completa**: se conoce en cada momento el estado completo del juego
 - **Deterministas**: no entra en juego el azar (se puede generalizar)
 - **2 jugadores** cuyas jugadas se alternan
 - **Juegos de suma cero**: Lo que “gana” uno lo “pierde” el otro. Al acabar, cada jugador pierde, gana o empata



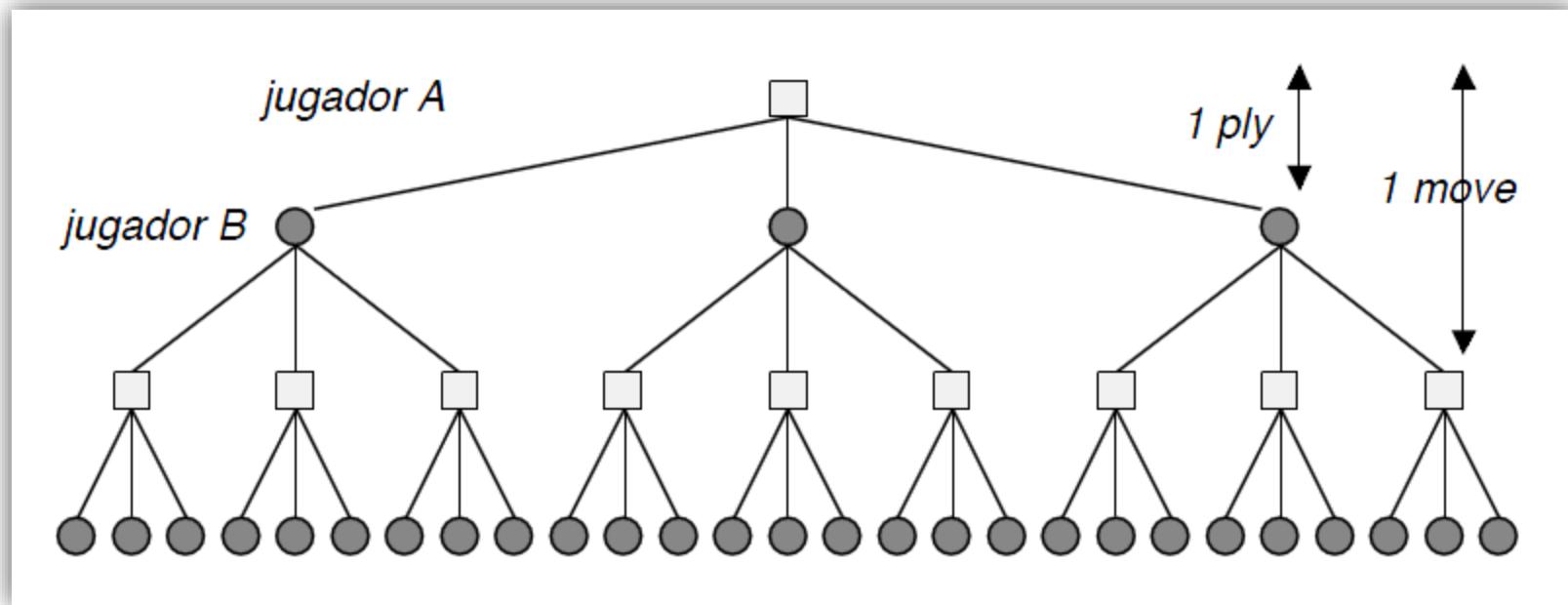
1. Búsqueda con adversarios

- Un juego puede definirse como un *problema* de búsqueda:
 - *Estado inicial*: posición inicial del tablero con una indicación de quien debe mover
 - *Operadores*: movimientos legales posibles
 - *Test terminal*: determina cuando ha finalizado el juego
 - *Función de utilidad $f(n)$* : asigna un valor numérico al resultado del juego
- **Árbol de juego:** representación explícita de todas las secuencias de jugadas posibles en una partida
 - Cada nivel representa, alternativamente, los movimientos posibles de cada jugador
 - Las hojas corresponden a estados GANAR ($+\infty$), PERDER ($-\infty$) o EMPATAR (0)
 - Cada camino desde la raíz (el estado inicial del juego) hasta una hoja representa una partida completa



1. Búsqueda con adversarios

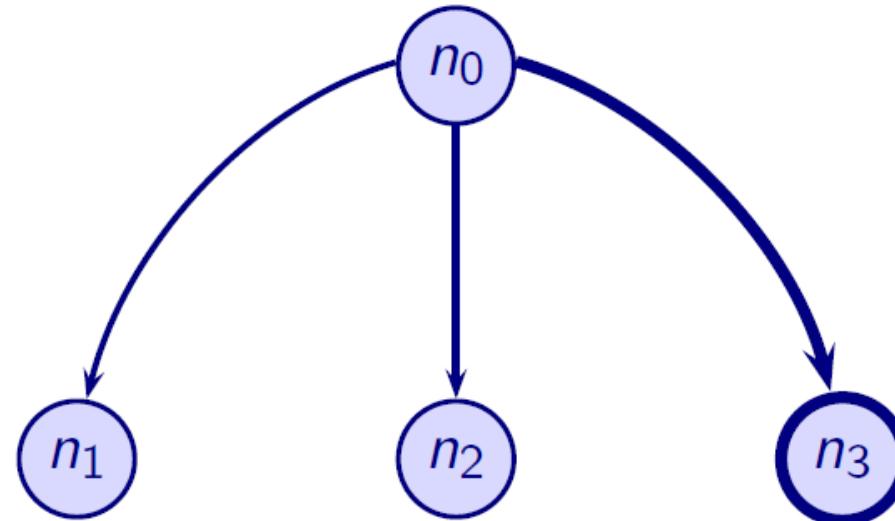
- Los algoritmos de búsqueda vistos hasta ahora no sirven:
El problema ya no es encontrar un camino en el árbol de juego (esto depende de los movimientos futuros del oponente), sino decidir el mejor movimiento dado el estado actual del juego





1. Búsqueda con adversarios

- Resolución con función de utilidad perfecta

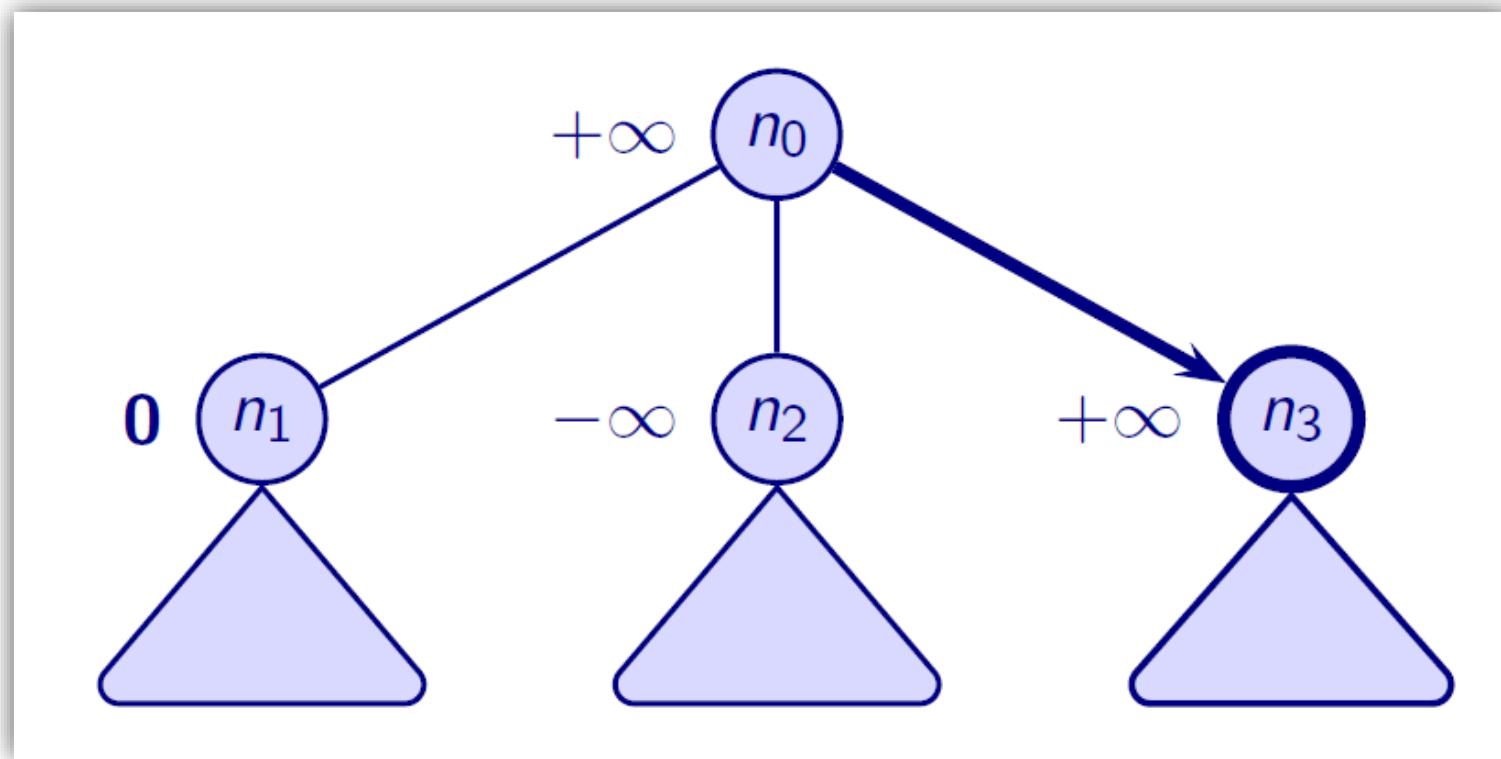


PROBLEMA: Normalmente no se conoce esa función



1. Búsqueda con adversarios

- Resolución con búsqueda completa



PROBLEMA: Es intratable, no se puede realizar en un tiempo razonable



1. Búsqueda con adversarios
2. Búsqueda Minimax
 1. Minimax con decisiones imperfectas
 2. Poda Alfa-Beta
3. Otros tipos de juegos
4. Estado del arte en problemas de juegos

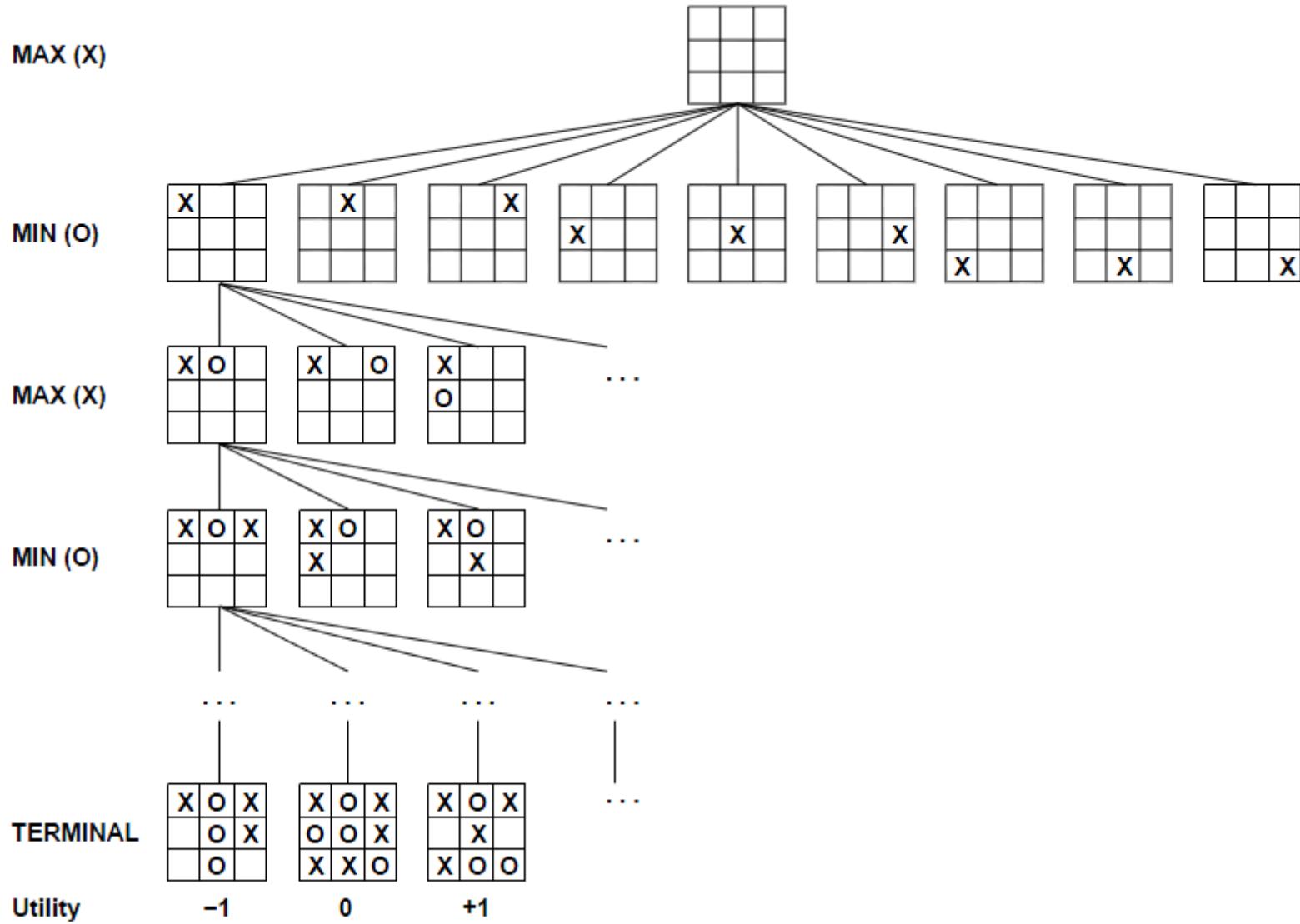


2. Búsqueda Minimax

- Algoritmo iterativo para un juego con dos jugadores: *MAX* y *MIN*:
 - Primero juega *MAX*, después *MIN*, y así hasta terminar
 - Son nodos *MAX/MIN* aquéllos en los que juega *MAX/MIN*
 - Si llamamos nivel 0 a la raíz, los nodos de nivel par son nodos *MAX* y los de nivel impar son nodos *MIN*
- $f(n)$ es la **función de utilidad** y asigna un valor numérico al resultado del juego desde el punto de vista de *MAX*.
 - Es la misma para los nodos de ambos jugadores
 - En los nodos terminales el valor de $f(n)$ es
 - $f(n) = +\infty$: gana *MAX* (G)
 - $f(n) = -\infty$: gana *MIN* (P)
 - $f(n) = 0$: *MAX* y *MIN* empatan (E)



2. Búsqueda Minimax





2. Búsqueda Minimax

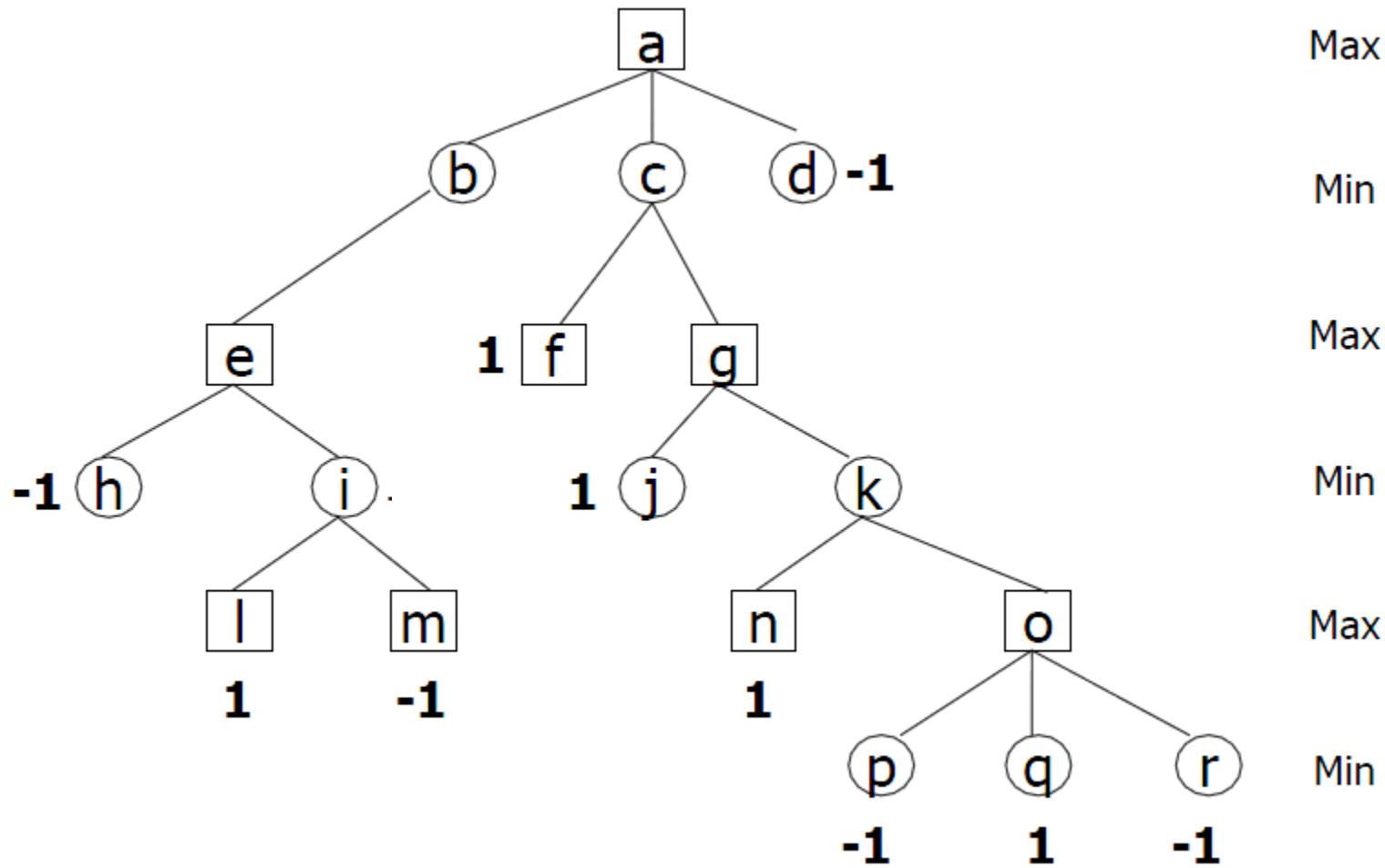
- A partir de los valores asignados a los nodos terminales, podemos ascender hasta la raíz (*propagar hacia arriba*):
 - A cada nodo *MAX* se le asigna el máximo de los valores de sus hijos (nodos *MIN*)
 - *MAX* intenta maximizar su ventaja
 - Buscamos el mejor movimiento para *MAX*
 - A cada nodo *MIN* se le asigna el mínimo de los valores de sus hijos (nodos *MAX*)
 - *MIN* procura minimizar la puntuación de *MAX*
 - Asumimos que siempre elegirá el peor movimiento para *MAX*, es decir, el mejor para sus intereses (*siempre jugará óptimamente*)
- *MAX* tiene que encontrar una estrategia que le permita **maximizar** $f(n)$ (ganar) independientemente de lo que haga *MIN*, que juega a **minimizar** $f(n)$ (ganar él)
 - Elige como siguiente movimiento el del sucesor del nodo inicial con mayor valor



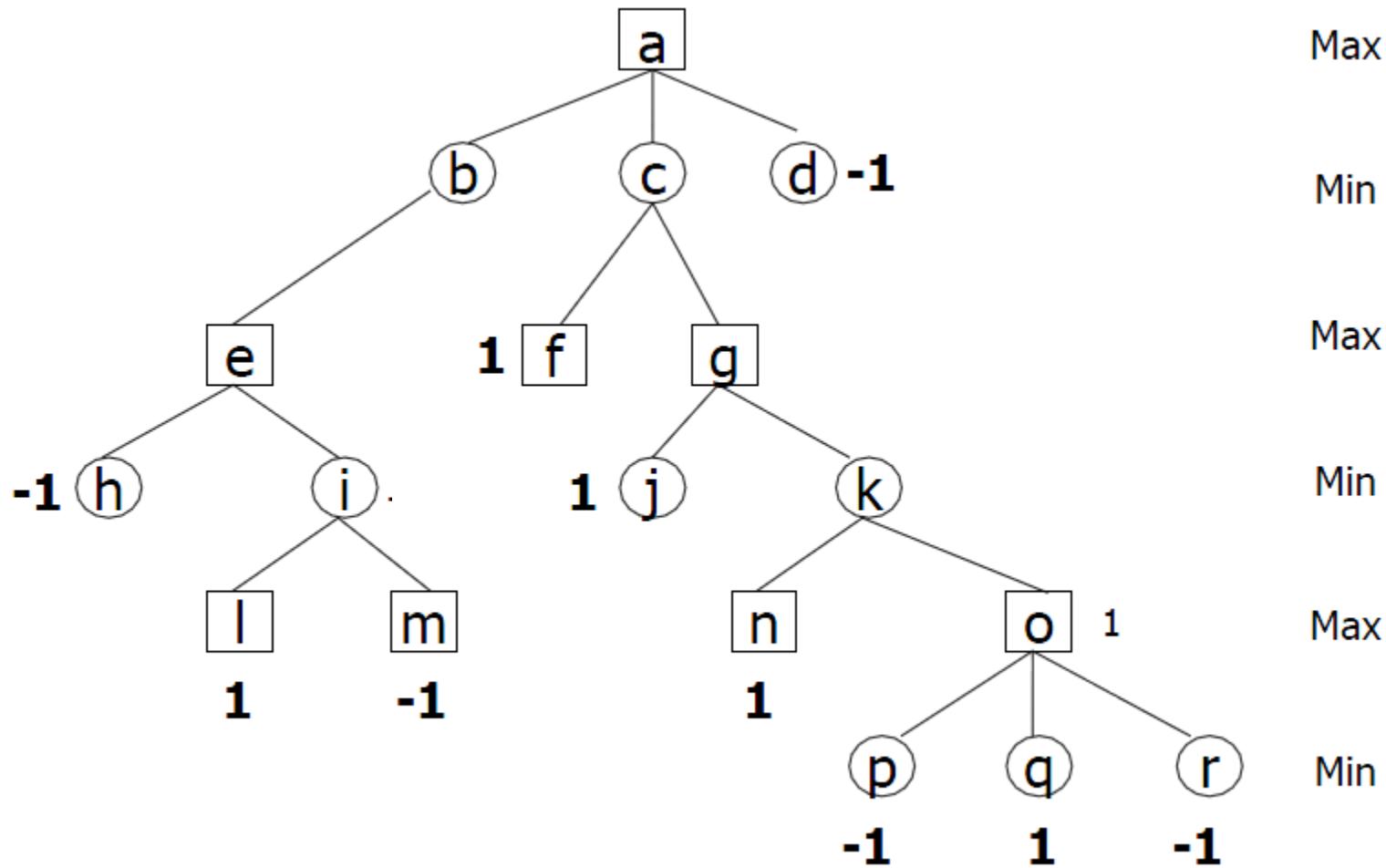
2. Búsqueda Minimax

- Pasos del algoritmo minimax:
 1. Generar el árbol de búsqueda completo, hasta los estados terminales.
 2. Aplicar la función de utilidad a los estados terminales.
 3. Usar la utilidad de los estados terminales para determinar la de los nodos de un nivel superior
 4. Continuar propagando los valores hacia la raíz.
 5. Al llegar a la raíz, escoger la acción que lleva a una mayor utilidad.

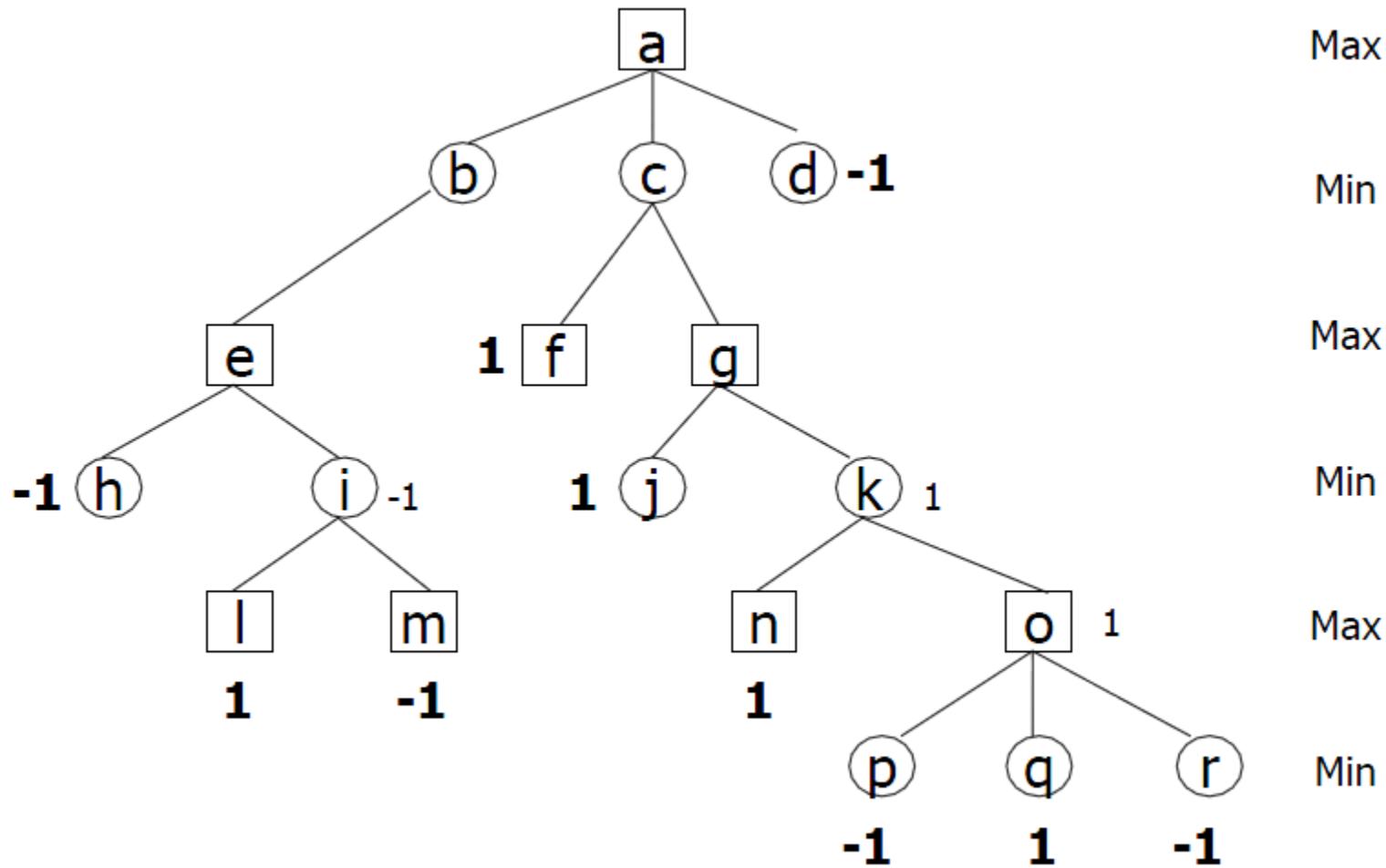
2. Búsqueda Minimax



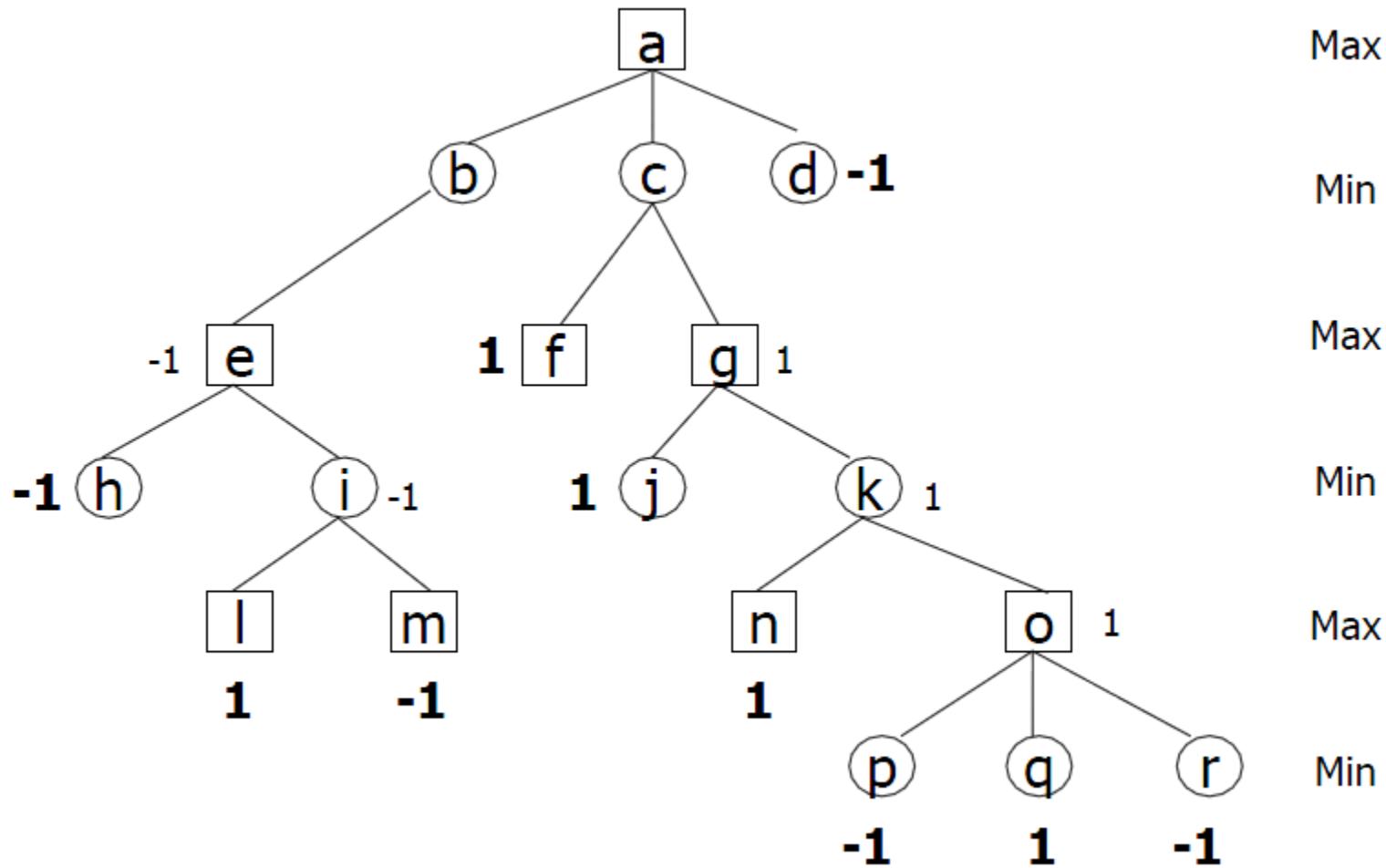
2. Búsqueda Minimax



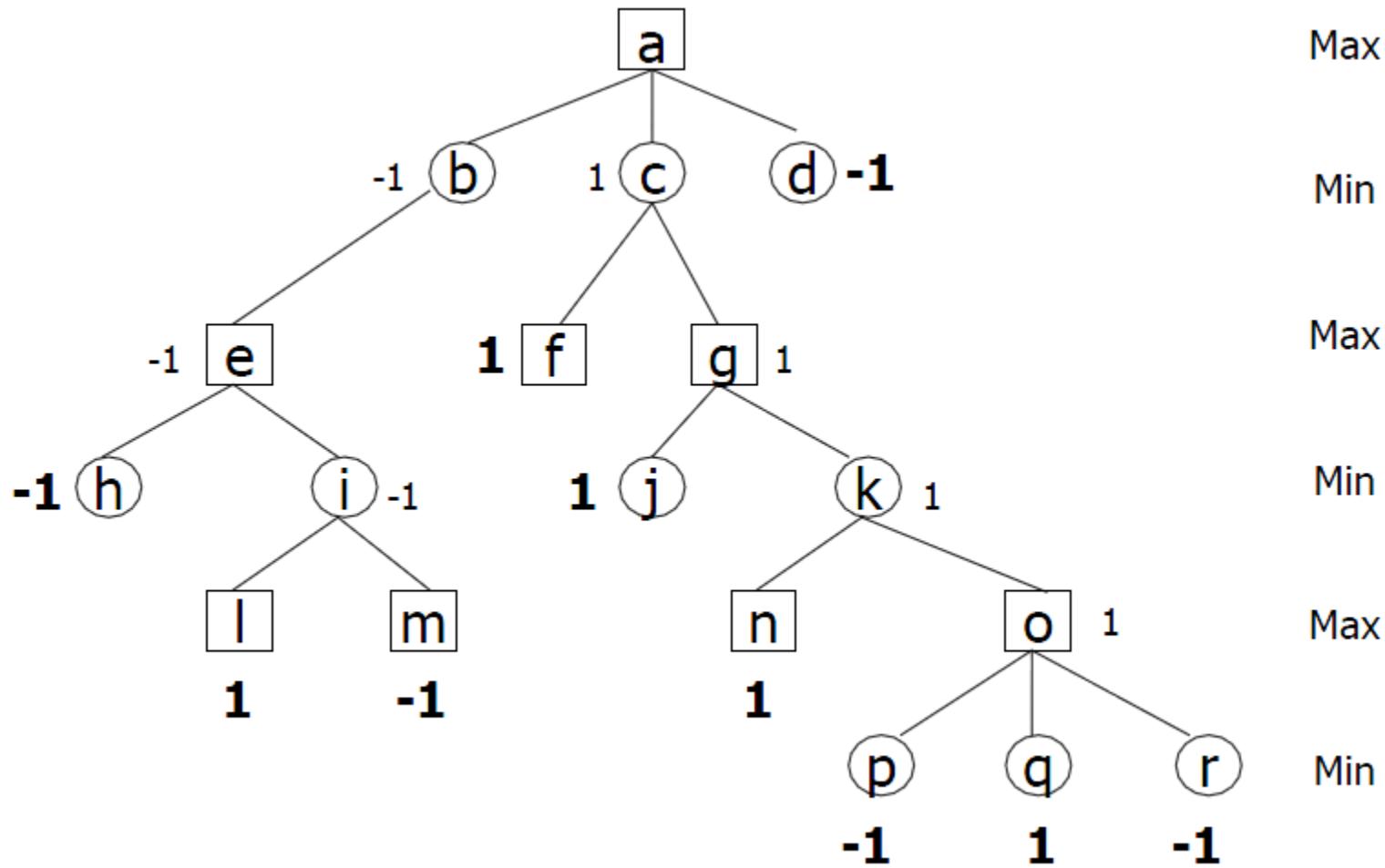
2. Búsqueda Minimax



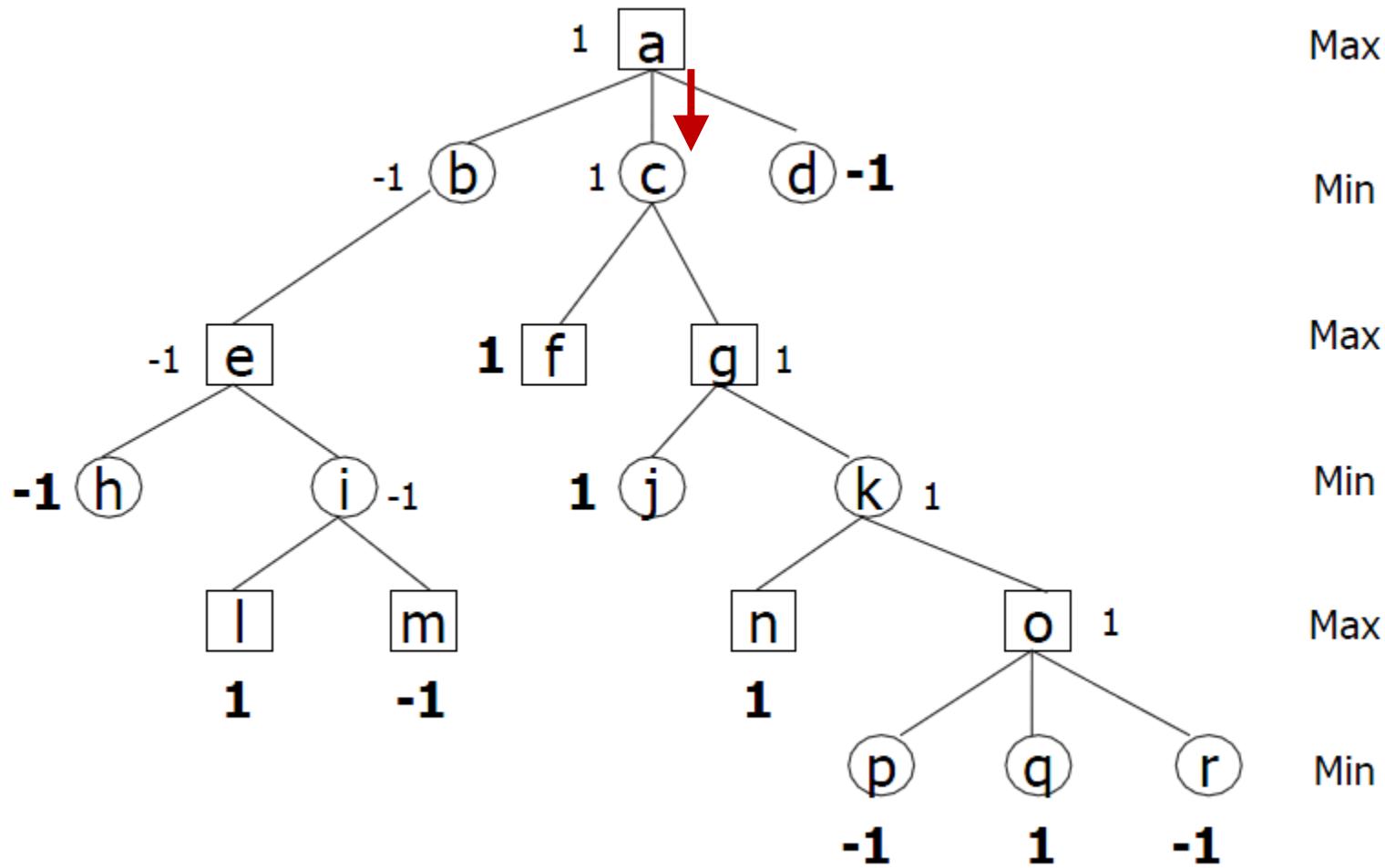
2. Búsqueda Minimax



2. Búsqueda Minimax

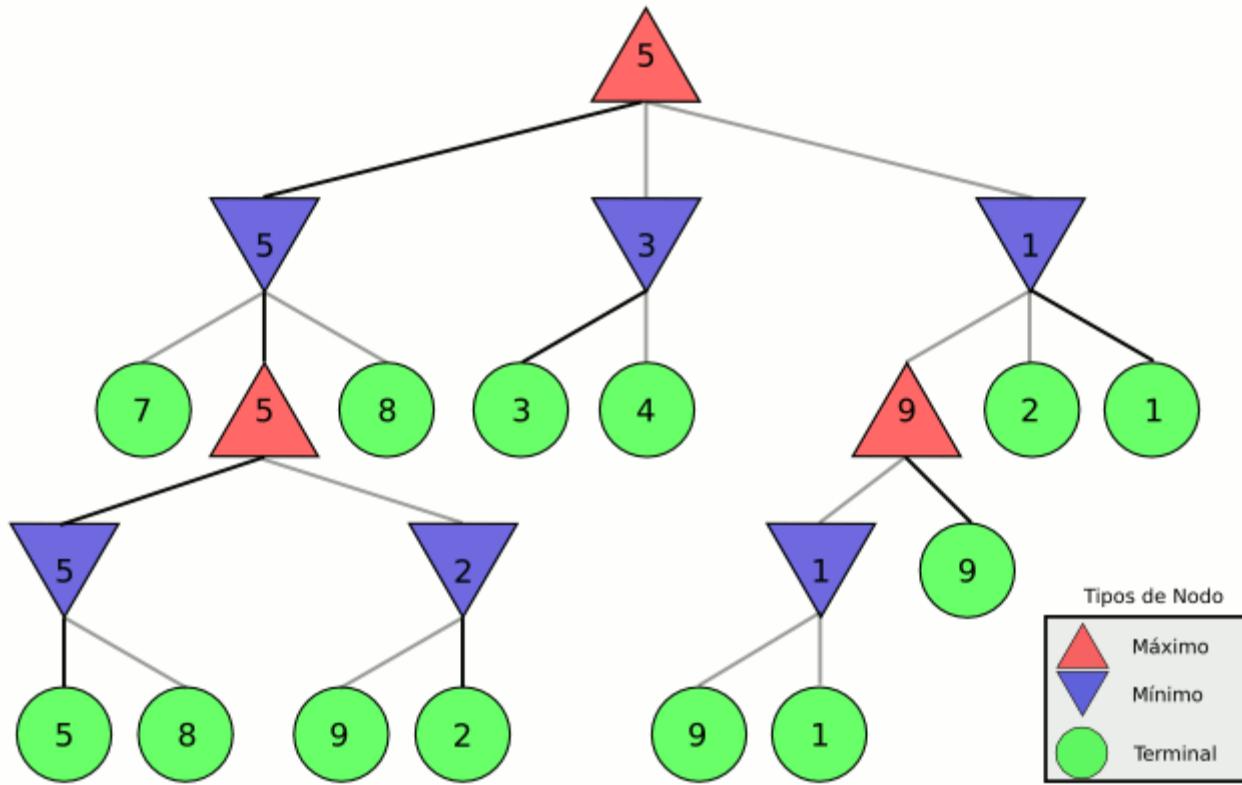


2. Búsqueda Minimax





2. Búsqueda Minimax





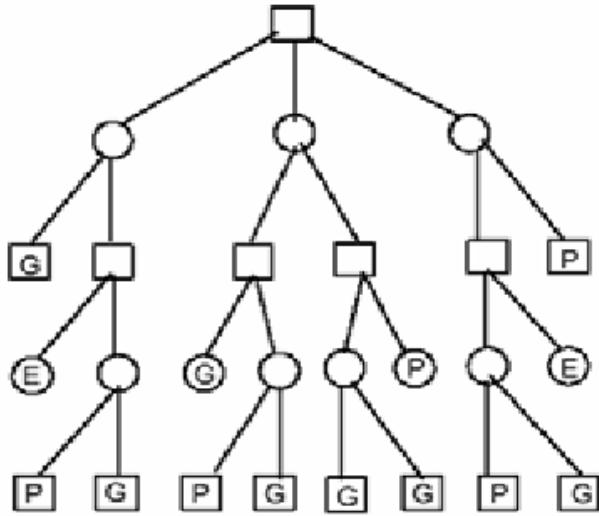
2. Búsqueda Minimax

- El valor ascendido a la raíz indica el valor del mejor estado que el jugador *MAX* puede aspirar a alcanzar
 - La etiqueta de un nodo nos indica lo mejor que podría jugar *MAX* en el caso de que se enfrentase a un oponente perfecto
- **Árbol solución (o estrategia de juego)** para *MAX*
 - Subárbol del árbol de juego con los movimientos que debe realizar *MAX* ante cualquier movimiento posible de *MIN*
 - Contiene
 - La raíz
 - Un sucesor de cada nodo *MAX* no terminal que aparezca en él
 - Todos los sucesores de cada nodo *MIN* que aparezca en él
- **Árbol ganador** para *MAX*: árbol solución en el que todos los nodos terminales dan a *MAX* ganador
 - Asegura que el jugador *MAX* ganará, haga lo que haga *MIN*



2. Búsqueda Minimax

Árbol de juego sin resolver

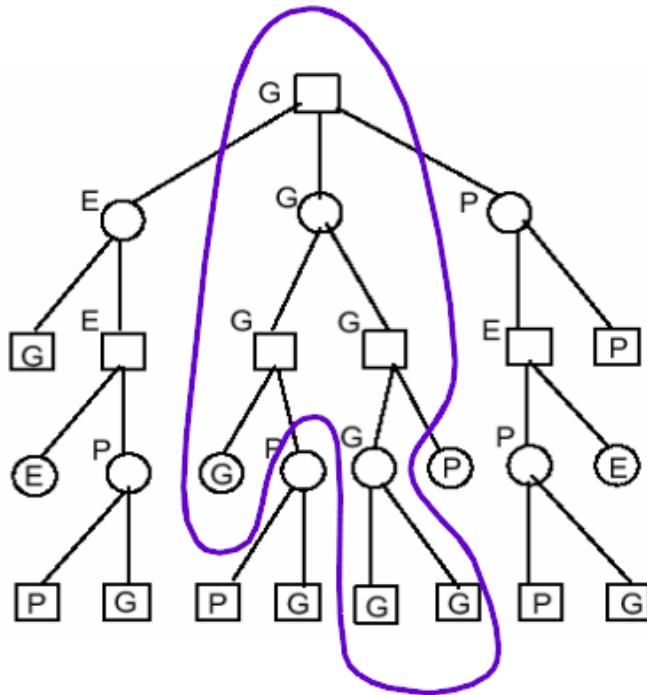


$G = 1, E = 0, P = -1$

Nodos MAX: cuadrados

Nodos MIN: círculos

Árbol de juego resuelto



Árbol ganador para MAX

2. Búsqueda Minimax

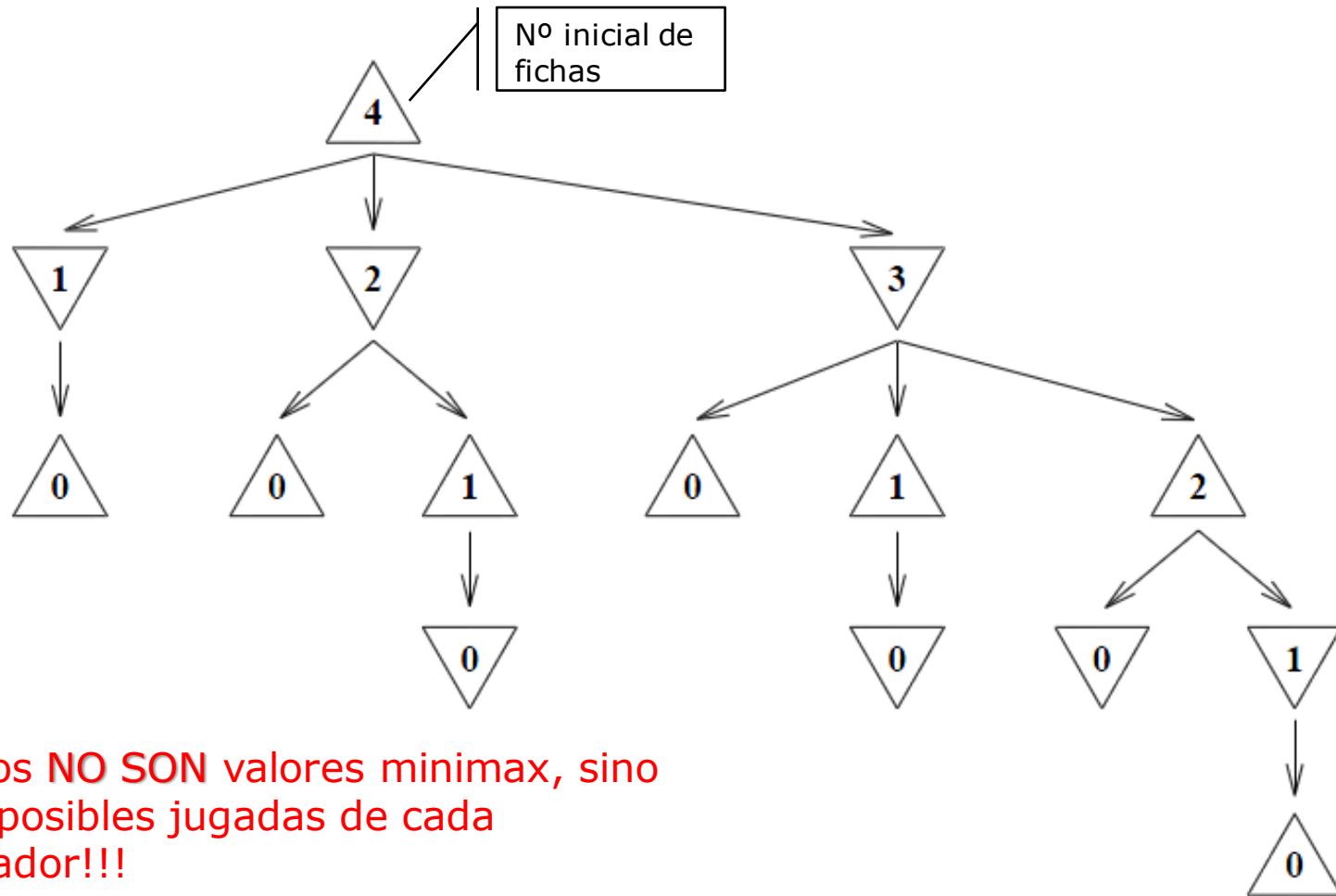


Nim

- Descripción
 - Una pila con N fichas de la que cada jugador puede tomar alternativamente 1, 2 o 3 fichas.
 - Objetivo: obligar al adversario a coger la última ficha.
- Elementos del juego
 - Estado inicial: numero de fichas al comienzo (N)
 - Estado final: 0 (único)
 - Estados: numero fichas que quedan en la mesa
 - Un jugador gana si en su turno el estado del sistema es el estado final
 - Función de utilidad (para el estado final):
 - 1 si le toca a MAX
 - -1 si le toca a MIN



2. Búsqueda Minimax



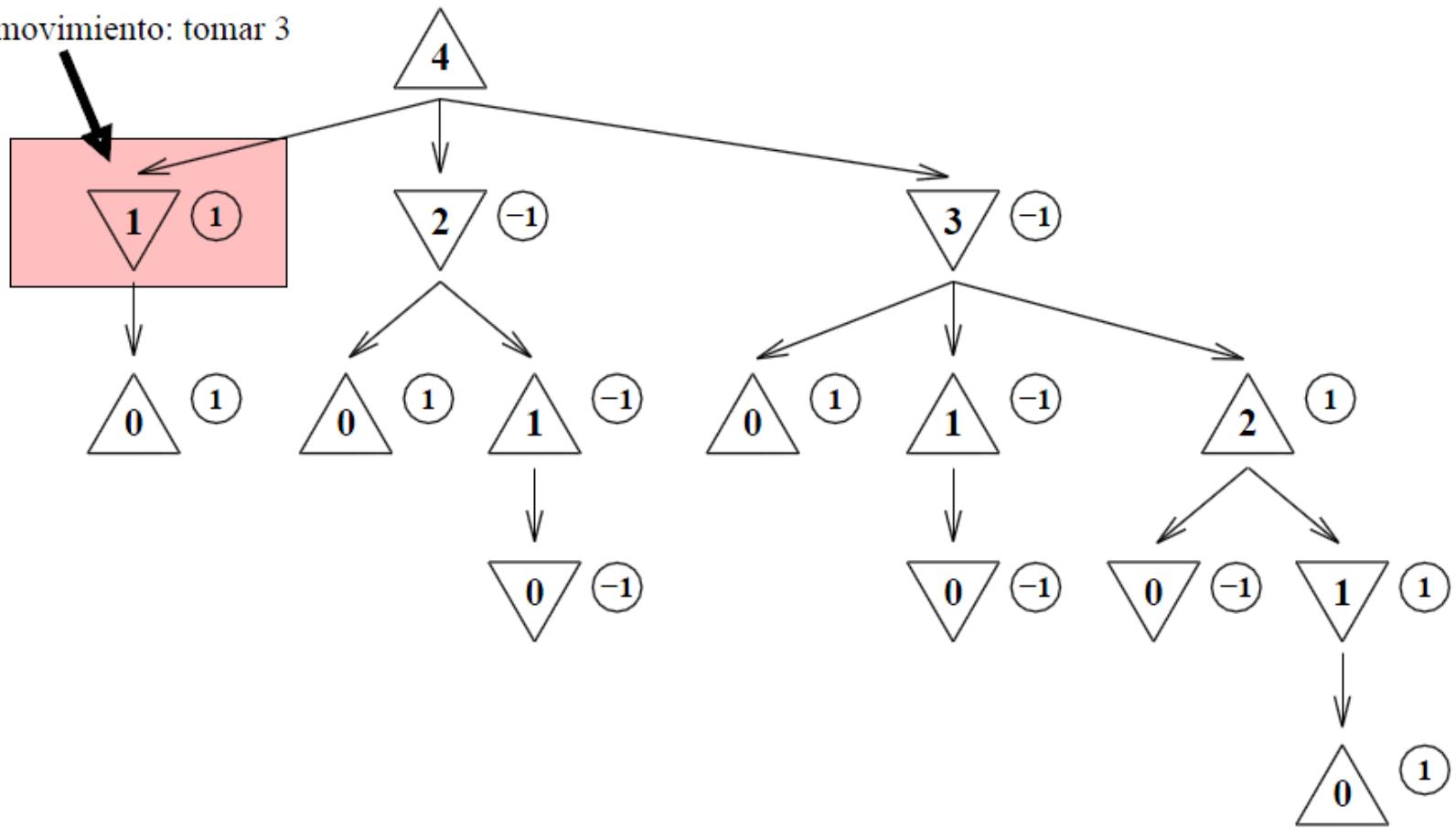
Estos NO SON valores minimax, sino
las posibles jugadas de cada
jugador!!!

Resolver el árbol minimax



2. Búsqueda Minimax

Mejor movimiento: tomar 3





2. Búsqueda Minimax

- Propiedades de la búsqueda minimax:
 - Completa: **Si**, si el árbol es finito
 - Complejidad tiempo: $O(b^d)$
 - Complejidad espacio: $O(b^d)$ (igual que la búsqueda primero en profundidad)
 - Optima: **Si**, si el contrincante juega perfectamente
- Para etiquetar la raíz es necesario un árbol de juego completo, pero...
- *Los costes temporal y espacial son excesivos e invalidan la aplicación de minimax a programas de juegos reales*
 - $\sim 10^{21}$ siglos para generar el árbol completo de búsqueda de las damas (10^{40} nodos, 3×10^{12} nodos/seg)
 - $\sim 10^{100}$ siglos para generar el árbol completo de búsqueda del ajedrez (10^{120} nodos)



2. Búsqueda Minimax

- Para solventar este problema se han propuesto dos aproximaciones distintas:
 - Minimax con decisiones imperfectas: Búsqueda con horizonte limitado y estimación en nodos límite
 - Poda alfa-beta

2.1 Minimax con decisiones imperfectas



- Interrumpir la búsqueda antes de llegar a los estados terminales → *horizonte limitado*
 - Profundidad k : número de pares de movimientos alternativos que se exploran antes de decidir el movimiento que hará MAX
 - La profundidad del conjunto de nodos es $2k$ para MAX y $2k-1$ para MIN
 - Las “hojas” de este árbol no necesariamente son finales de partida
 - No siempre se puede asignar valor de $f(n)$
 - Se añade una **Función estática de evaluación $h(n)$** :
 - Heurística de la expectativa de ganar de MAX
 - Se sustituye el test terminal por un *test de interrupción*
 - Nodos terminales (finales de partida): $f(n)$
 - Nodos profundidad k (terminales o no): $h(n)$
 - Los valores se propagan (ascienden) usando el principio minimax

2.1 Minimax con decisiones imperfectas

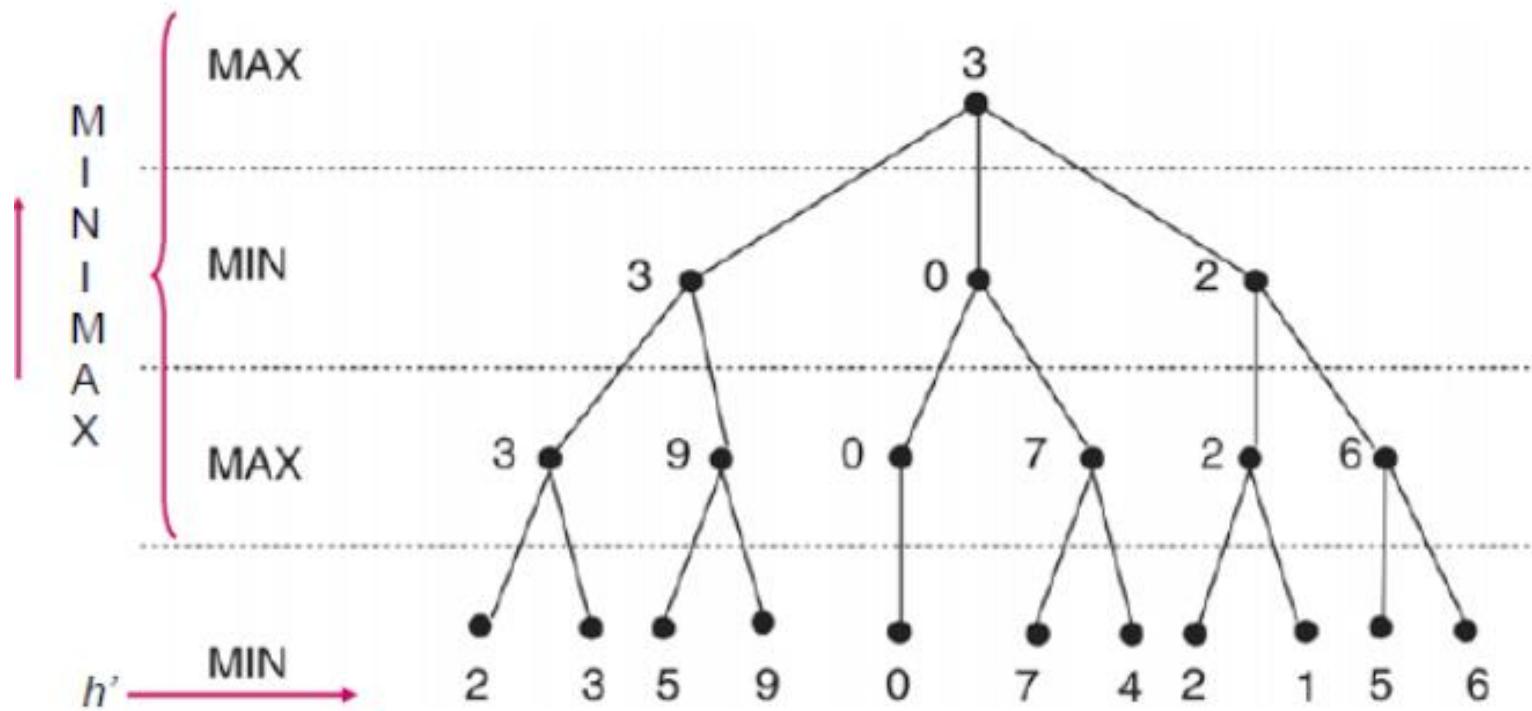


Valor de la Función de Utilidad en el modelo
Minimax con decisiones imperfectas

$$f(n) = \begin{cases} +\infty & \text{si } n \text{ es una situación ganadora} \\ -\infty & \text{si } n \text{ es una situación perdedora} \\ 0 & \text{si } n \text{ es una situación de empate} \\ h(n) & \text{si } p = \text{Profundidad-máxima} \\ \max_{S_i \in S(n)} f(S_i) & \text{si } n \text{ es nodo MAX y } p < p_{max} \\ \min_{S_i \in S(n)} f(S_i) & \text{si } n \text{ es nodo MIN y } p < p_{max} \end{cases}$$



2.1 Minimax con decisiones imperfectas



2.1 Minimax con decisiones imperfectas



■ Función Estática de Evaluación

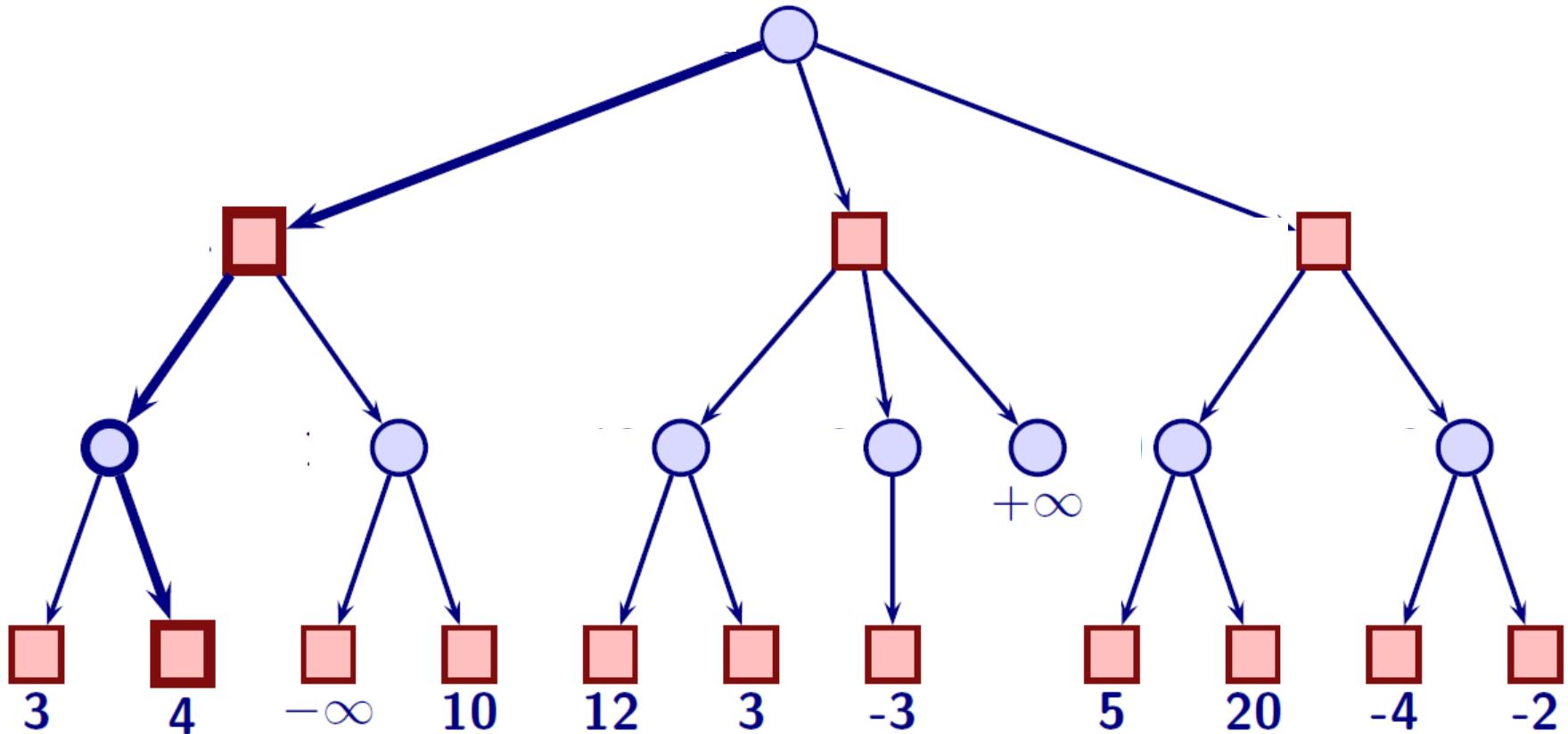
- Heurística que codifica todo el *conocimiento* que poseemos acerca del juego
 - en el ajedrez se puede dar un *valor material* a cada pieza: 1 al peón, 3 al caballo y al alfil, . . .
 - Dado un estado del juego y un turno, es una *estimación* de las posibilidades de ganar de *MAX* en tal situación
 - Están basadas normalmente en la experiencia previa
-
- Características de una función estática de evaluación:
 - Debe acercarse a la función de utilidad en los estados terminales
 - Su cálculo no debe ser muy costoso
 - Debe reflejar de forma precisa la probabilidad de resultar ganador
 - $h(n)$ grande y positivo es bueno para *MAX* y malo para *MIN*

2.1 Minimax con decisiones imperfectas

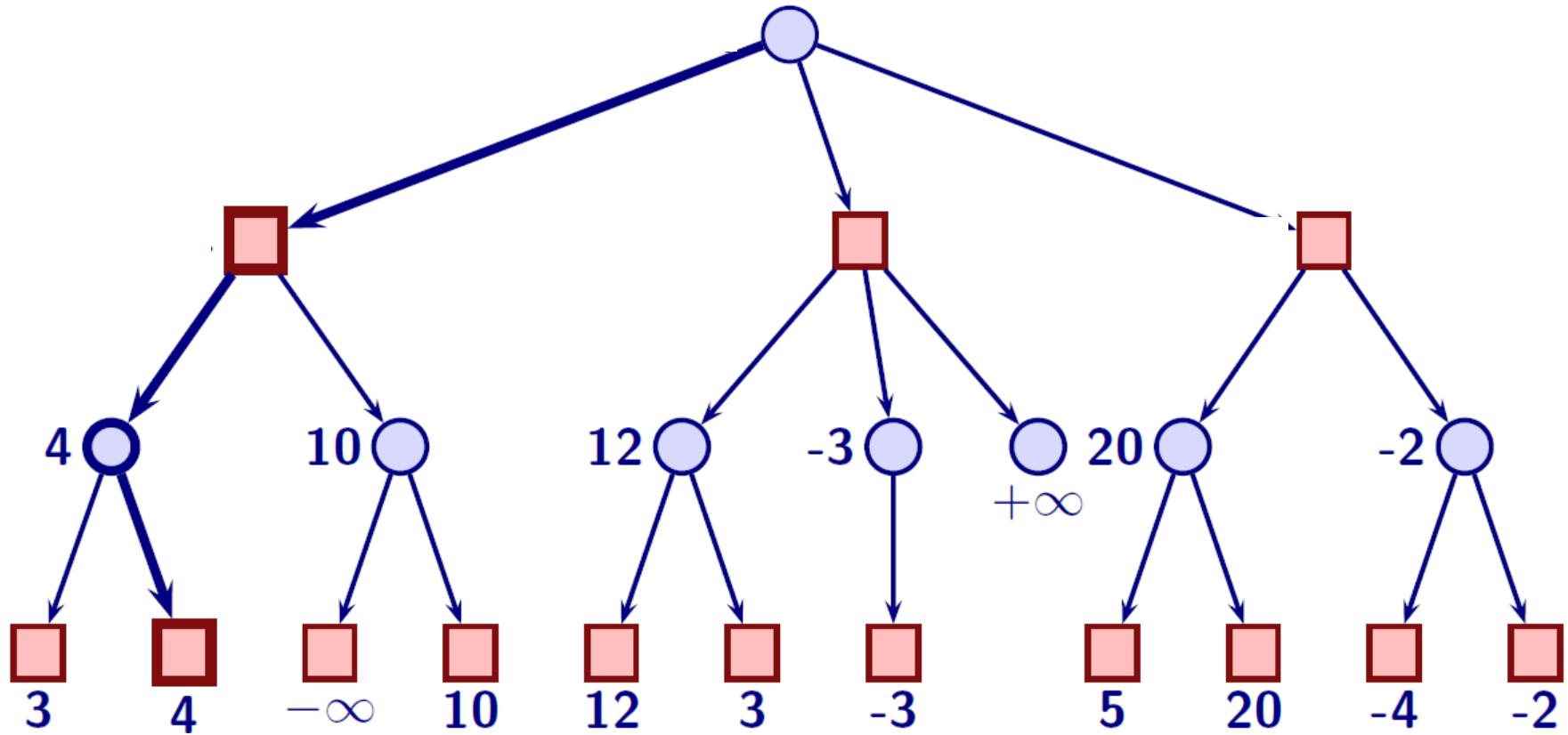


- Se asume que el valor ascendido hasta la raíz, obtenido en una profundidad k , es una estimación mejor que aplicar directamente la función de estimación a los sucesores del nodo raíz
- A la raíz le llega la medida heurística del mejor estado alcanzable en k movimientos
 - Cuanto mayor sea el horizonte, más seguros son los elementos de decisión para elegir la mejor jugada
 - Así se prevén las consecuencias de la jugada a más largo plazo
 - Pero no hay garantías
- Lo importante es la comparación del valor entre los estados

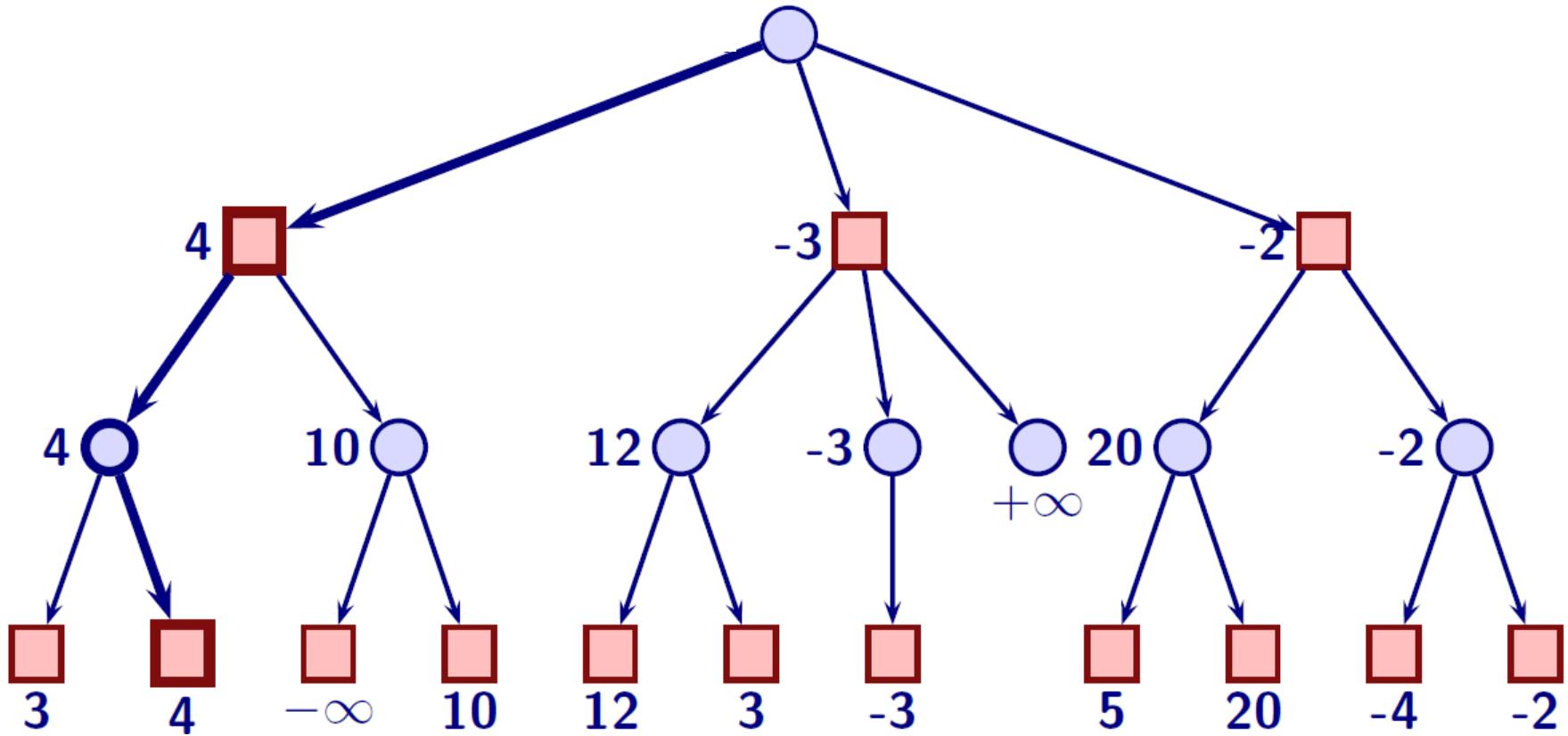
2.1 Minimax con decisiones imperfectas



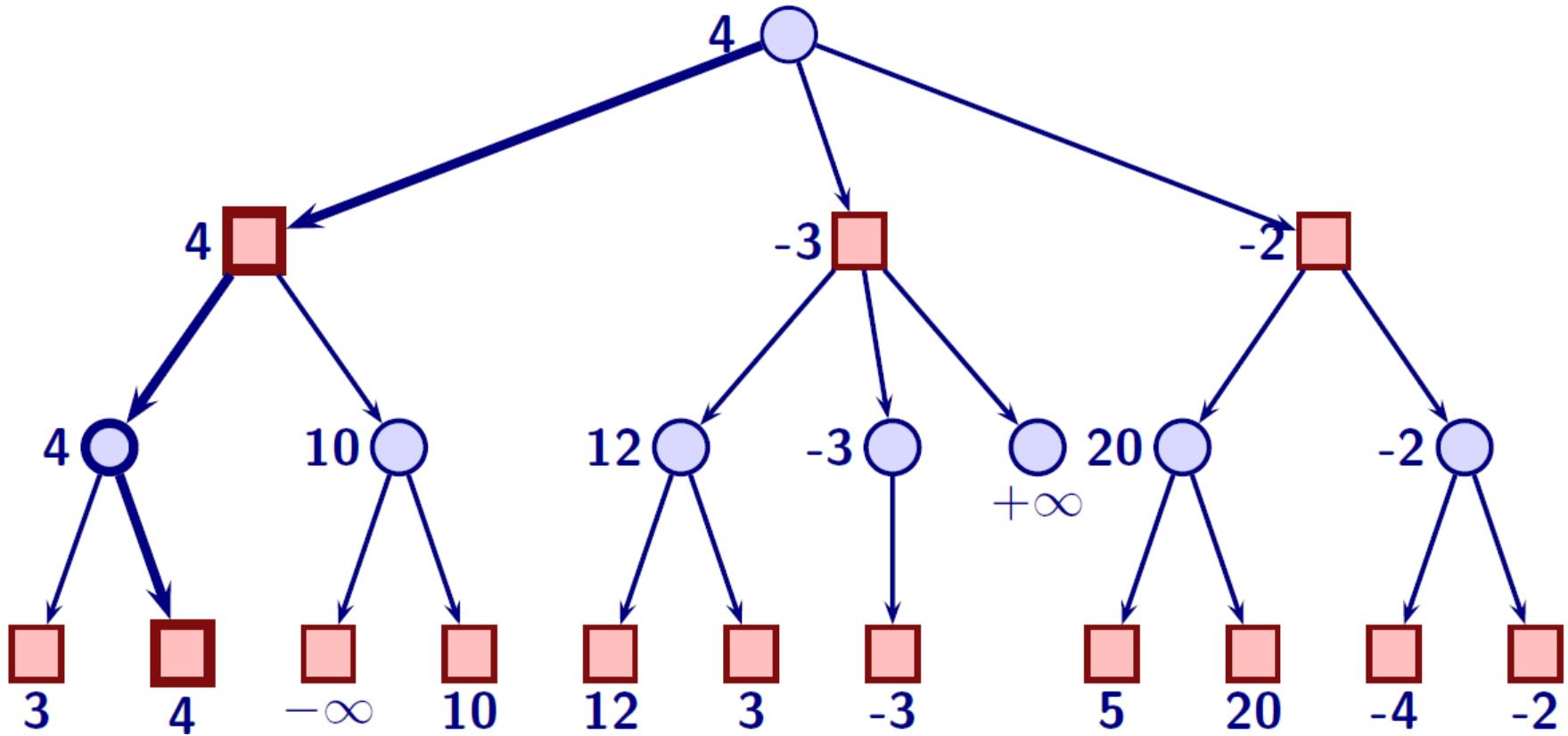
2.1 Minimax con decisiones imperfectas



2.1 Minimax con decisiones imperfectas



2.1 Minimax con decisiones imperfectas



2.1 Minimax con decisiones imperfectas



Tres en raya

- Descripción
 - En un tablero de 3x3 alternativamente un jugador pone una ficha **X** y el otro jugador pone una ficha **O**
 - Gana el que consigue colocar tres de sus fichas en línea (vertical, horizontal o diagonal)
- Elementos del juego
 - Estado inicial: tablero vacío + ficha de salida
 - Estados finales: tableros completos o con línea ganadora
 - Estados: tablero + ficha que se pondrá a continuación
 - Estados ganadores para un jugador: estados finales en los que no le toca poner
- Movimientos:
 - 9 movimientos posibles, uno por casilla
 - Aplicable si la casilla no esta ocupada

2.1 Minimax con decisiones imperfectas



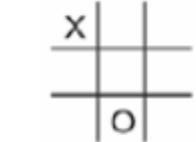
- Función de utilidad:
 - $+\infty$ si es ganador para *MAX*
 - 0 si es tablas
 - $-\infty$ si es ganador para *MIN*
- Función Estática de Evaluación (heurística):
 - Diferencia entre el numero de posibles líneas ganadoras para *MAX* y el numero de posibles líneas ganadoras para *MIN*
 - Líneas ganadoras: Número de filas, columnas y diagonales que se pueden completar, incluyendo las vacías.

$$h(n) = \text{Líneas}(MAX) - \text{Líneas}(MIN)$$



2.1 Minimax con decisiones imperfectas

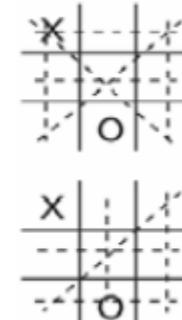
- Si MAX juega con X



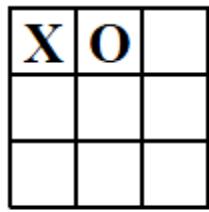
$$h'(n) = 6 - 5 = 1$$

X tiene 6 posibles líneas ganadoras

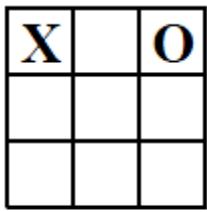
O tiene 5 posibles líneas ganadoras



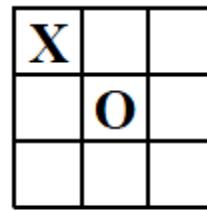
- Para otros casos:



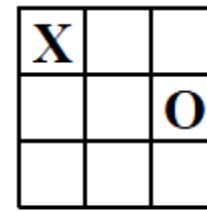
$$6-5=1$$



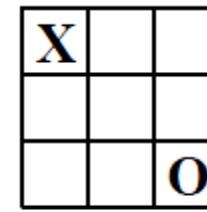
$$5-5=0$$



$$4-5=-1$$



$$6-5=1$$

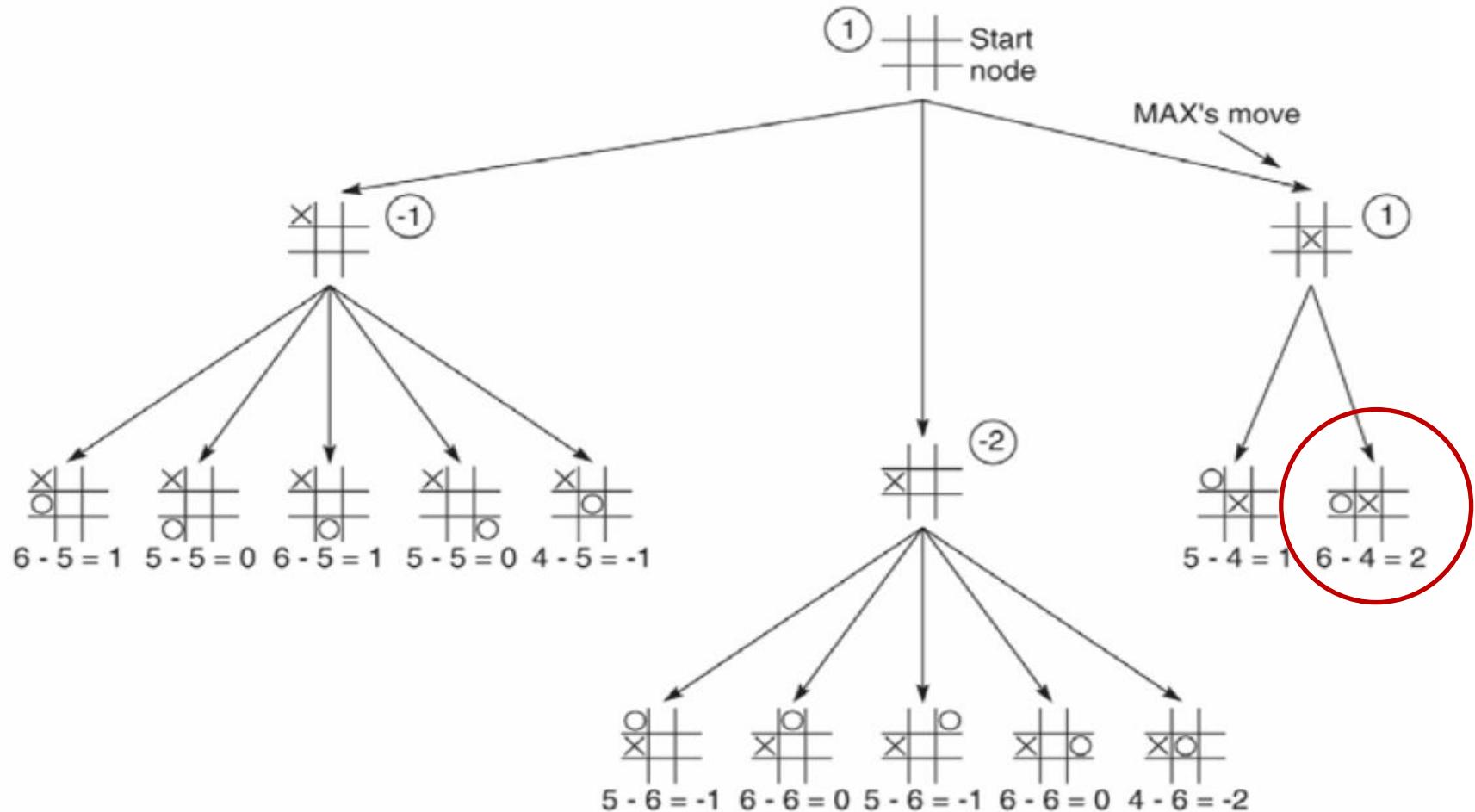


$$5-5=0$$



2.1 Minimax con decisiones imperfectas

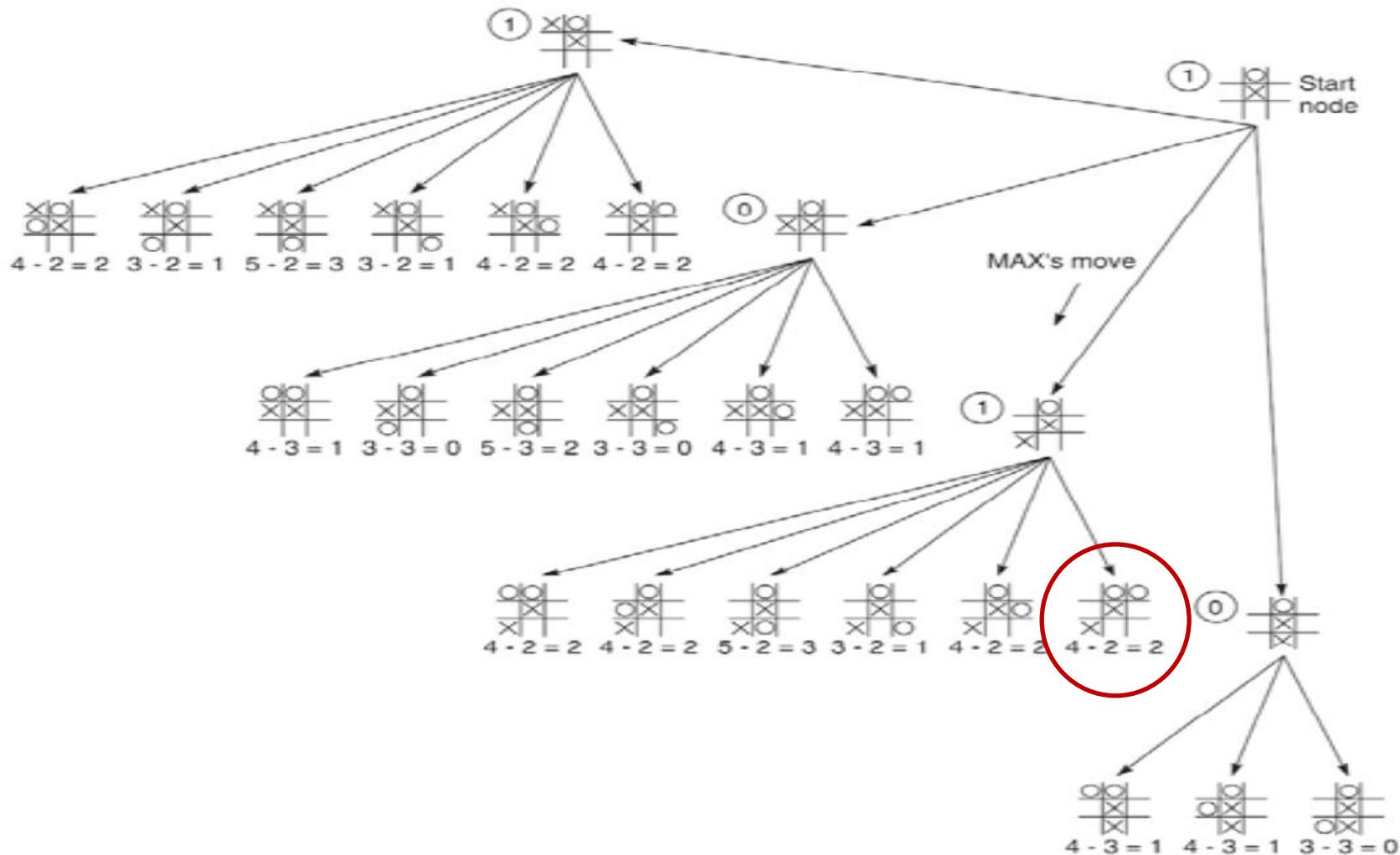
- Movimiento inicial (con profundidad $k=1$ para MAX)





2.1 Minimax con decisiones imperfectas

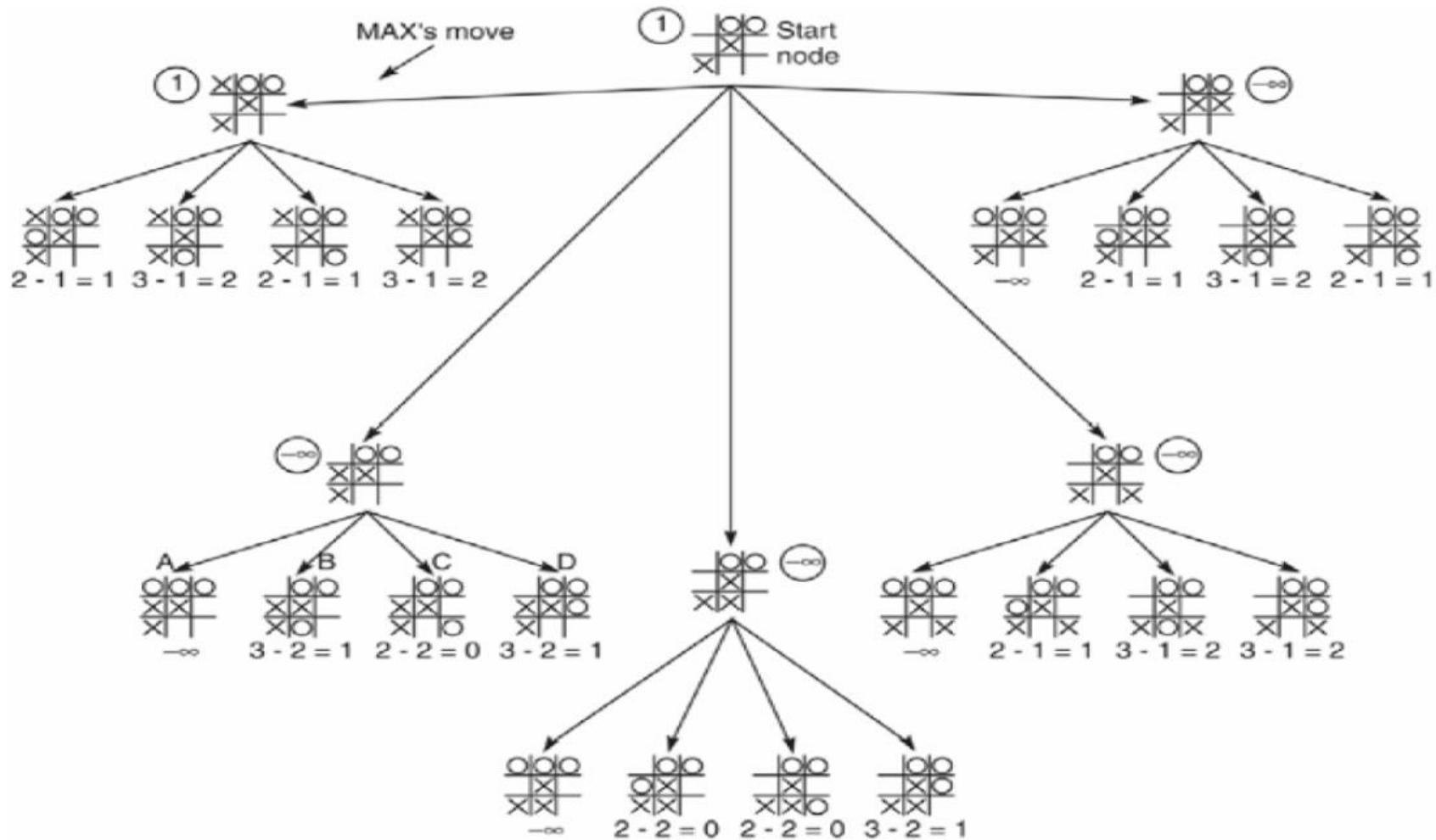
- Segundo movimiento de MAX ($k=1$)





2.1 Minimax con decisiones imperfectas

- Tercer movimiento de MAX ($k=1$)

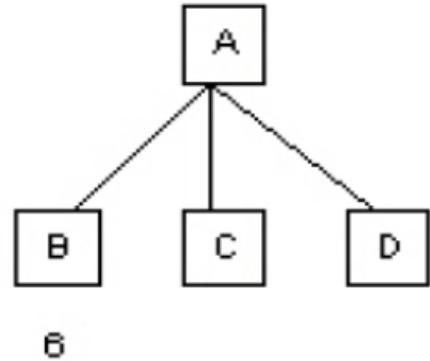


2.1 Minimax con decisiones imperfectas

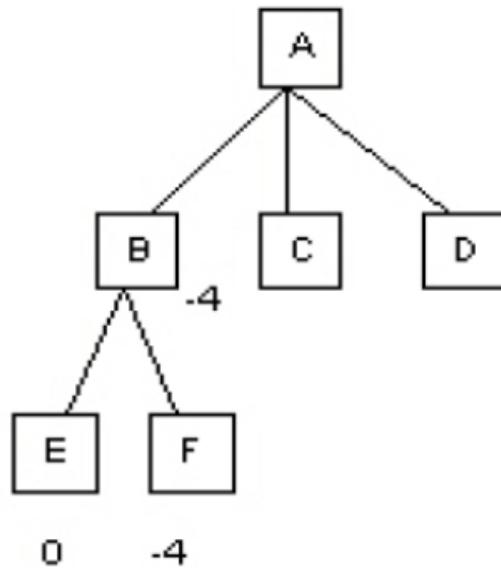


- Existen distintas opciones para interrumpir la búsqueda, pero todas tienen problemas derivados del efecto horizonte
 - Simple: llegar al límite máximo de profundidad del árbol de acuerdo en el tiempo disponible
 - Astuta:
 - Distintas búsquedas por profundización iterativa. Se devuelve la búsqueda más profunda que se haya conseguido en el tiempo disponible
 - Búsquedas secundarias: búsquedas parciales en los nodos en los que se detecten problemas
 - Posiciones en reposo: No se puede cortar si la posición no es estable. $h(n)$ debe ser aplicada solo a posiciones que no cambien bruscamente de valor después
 - Ajedrez: que un jugador capture una pieza importante puede resultar en ventaja para el otro (*sacrificios*)
 - Extensiones singulares: Si un nodo hoja es muy diferente a sus hermanos, el nodo se expande una capa más
 - Ajedrez: por situación de captura inminente (*jugada forzada*)

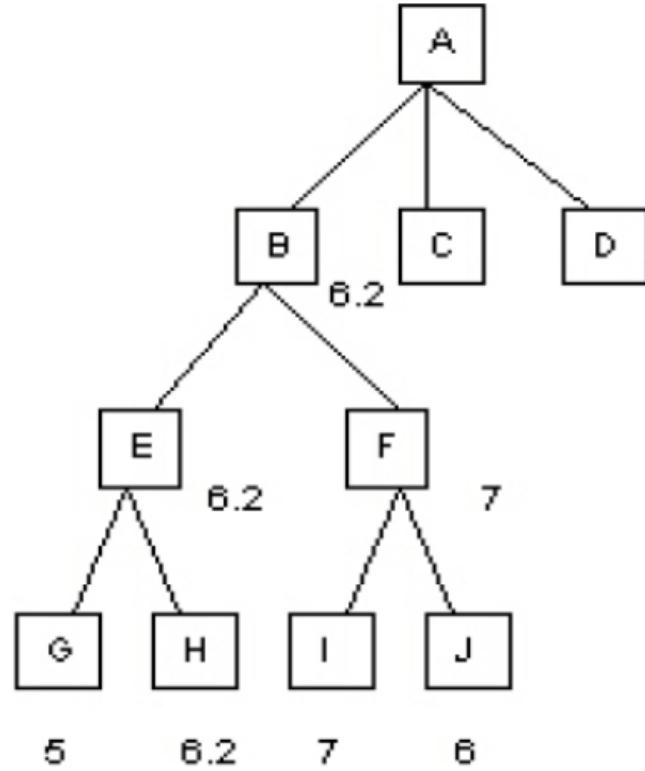
2.1 Minimax con decisiones imperfectas



Si exploramos un nivel, B es la mejor opción

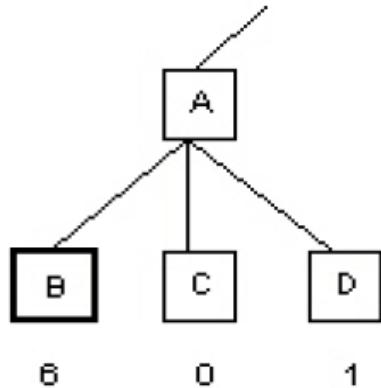


La cosa cambia si exploramos un nivel más

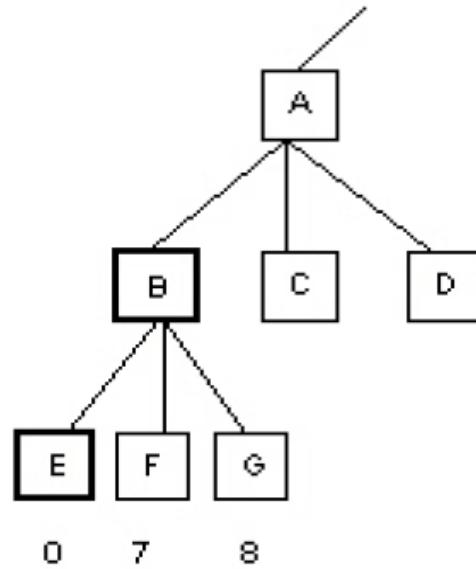


Posición de reposo: la situación se parece a la inicial

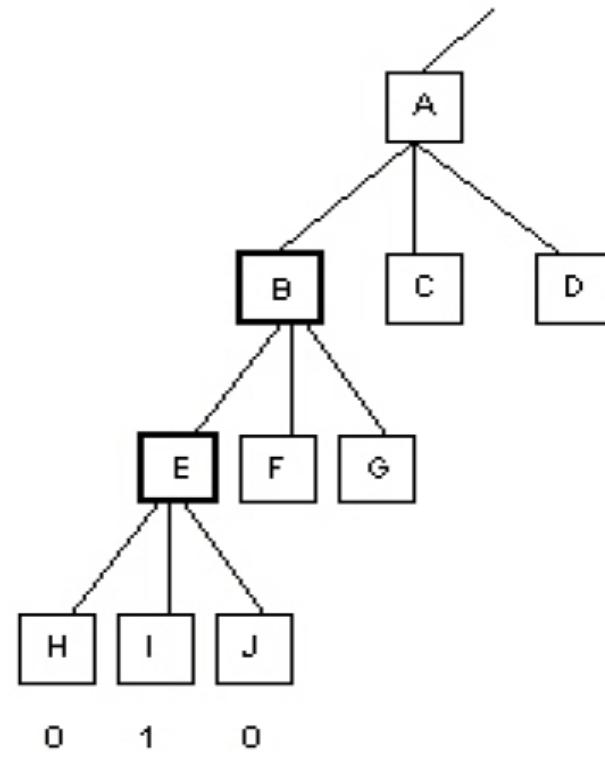
2.1 Minimax con decisiones imperfectas



B tiene un valor superior a sus hermanos.
¿Captura inminente?



¿Captura por parte del oponente?



No hay captura: valores en un intervalo estrecho. Se interrumpe la búsqueda secundaria

2.2 Poda Alfa-Beta



- Es posible calcular la decisión minimax correcta sin explorar todos los nodos del árbol de búsqueda
- El proceso por el cual determinadas ramas no se consideran en la búsqueda se denomina **poda** del árbol de búsqueda
- Optimización del minimax: *podar las ramas que no proporcionen mejoras sobre el mejor camino hasta el momento*
 - Poda α - β (α - β pruning)
 - Elimina los nodos que ofrezcan un valor menor (para MAX) o mayor (para MIN) que los que hemos encontrado ya
 - Sin comprobar cuan malo es el sub-árbol
 - Convierte un factor de ramificación de b en \sqrt{b}
 - Búsqueda Primero en Profundidad



2.2 Poda Alfa-Beta

- α = valor de la mejor jugada hasta el momento para *MAX*
 - guarda el valor máximo de los sucesores de *MAX* encontrados hasta el momento
 - cota inferior para el valor final que pueda alcanzar el nodo *MAX* (lo peor que le podría ir a *MAX*)
 - Valor monótono no-decreciente
 - $\alpha_0 = -\infty$

- β = valor de la mejor jugada hasta el momento para *MIN*
 - guarda el valor mínimo de los sucesores de *MIN* encontrados hasta el momento
 - cota superior para el valor final que pueda alcanzar el nodo *MIN* (lo mejor que le podría ir a *MIN*)
 - Valor monótono no-creciente
 - $\beta_0 = +\infty$

2.2 Poda Alfa-Beta



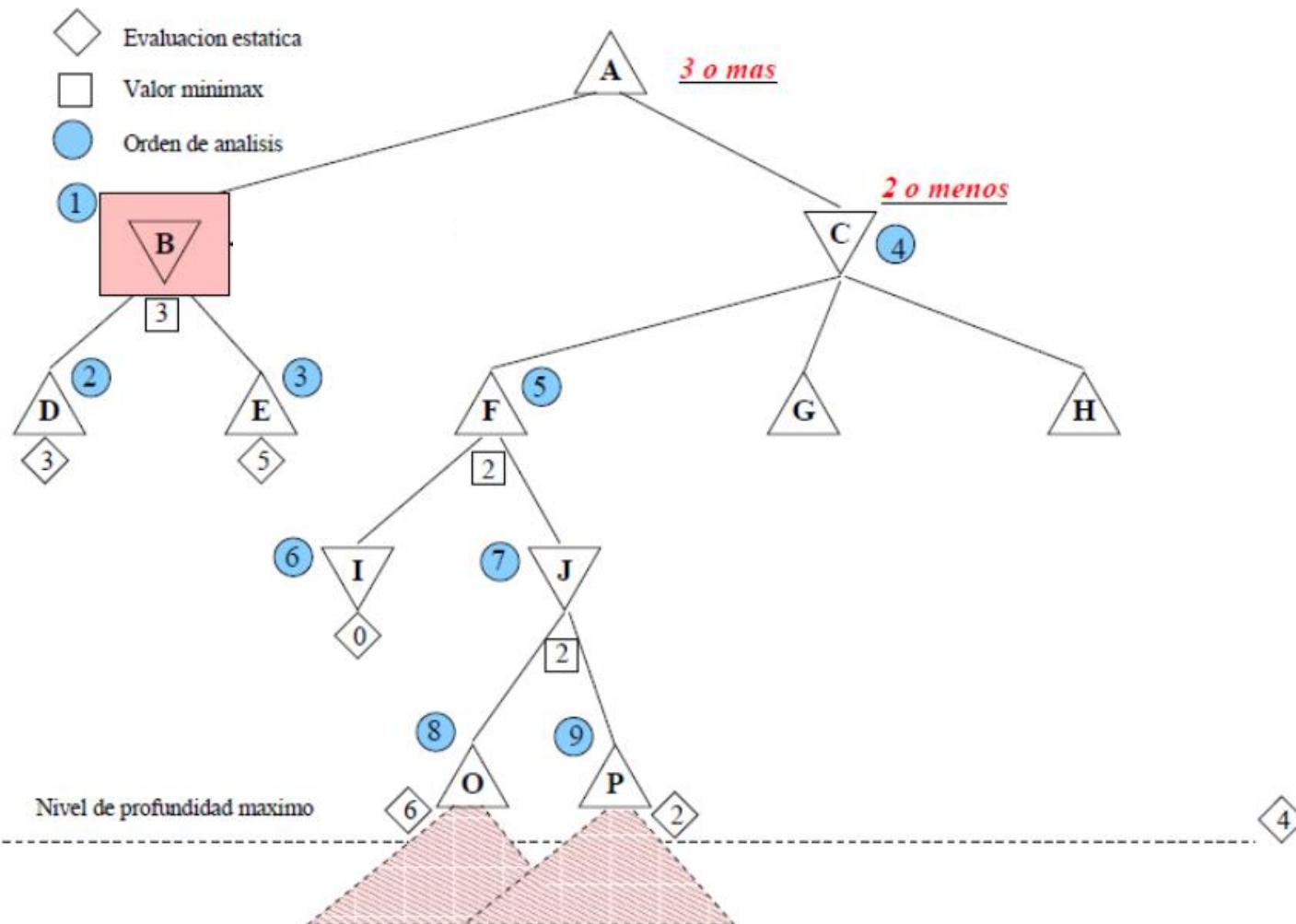
- Nodos *MAX*: Poda α
 - α es el valor actual del nodo (que tendrá eso o más) y β es el valor actual del padre (que tendrá eso o menos)
 - Poda en (n) cuando $\alpha_n \geq \beta_{n-1}$ porque ya hay jugadas mejores para *MIN*
 - No hace falta analizar los restantes sucesores del nodo

- Nodos *MIN*: Poda β
 - β es el valor actual del nodo (que tendrá eso o menos) y α es el valor actual del padre (que tendrá eso o más)
 - Poda en (n): cuando $\beta_n \leq \alpha_{n-1}$ porque ya hay jugadas mejores para *MAX*
 - No hace falta analizar los restantes sucesores del nodo

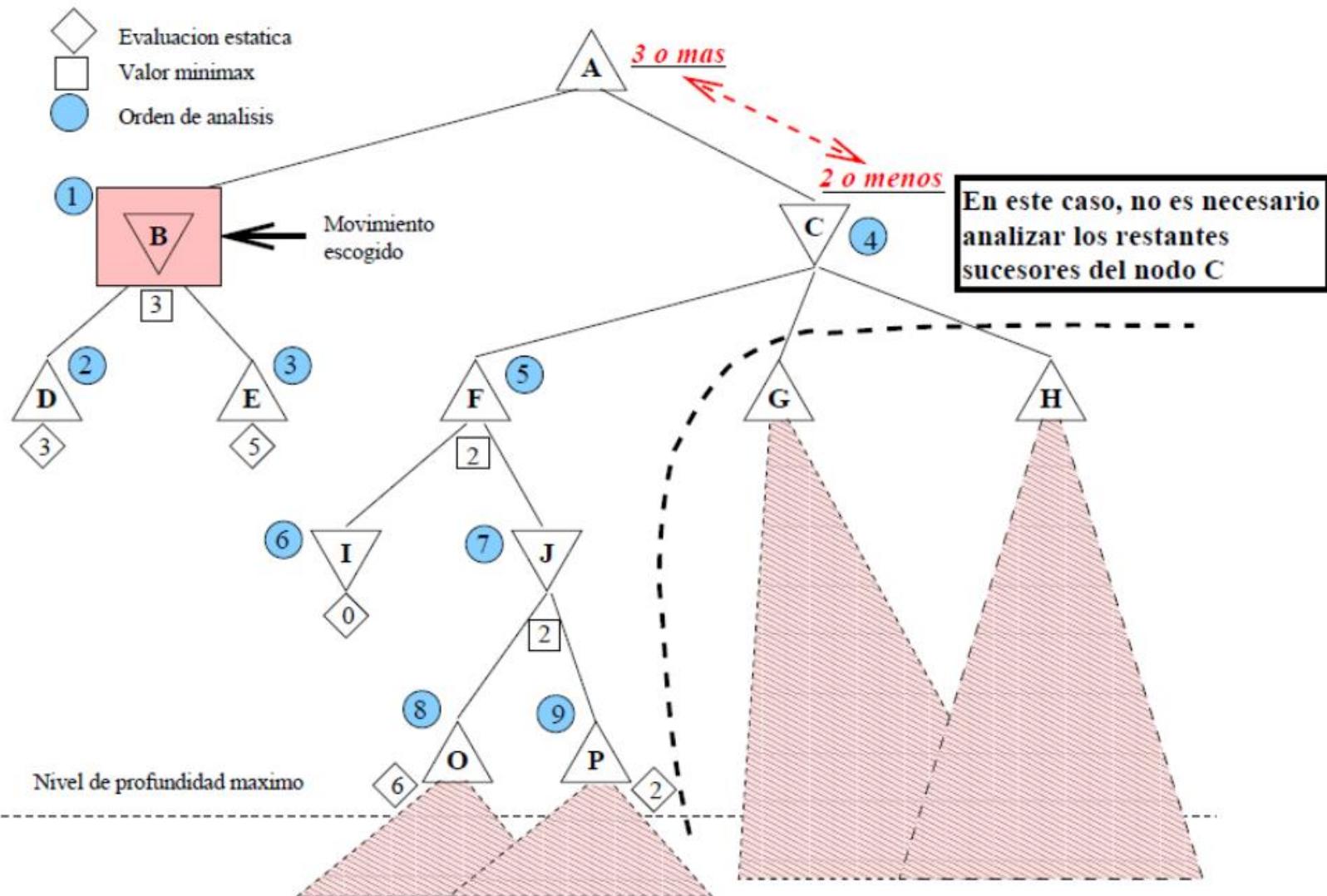


2.2 Poda Alfa-Beta

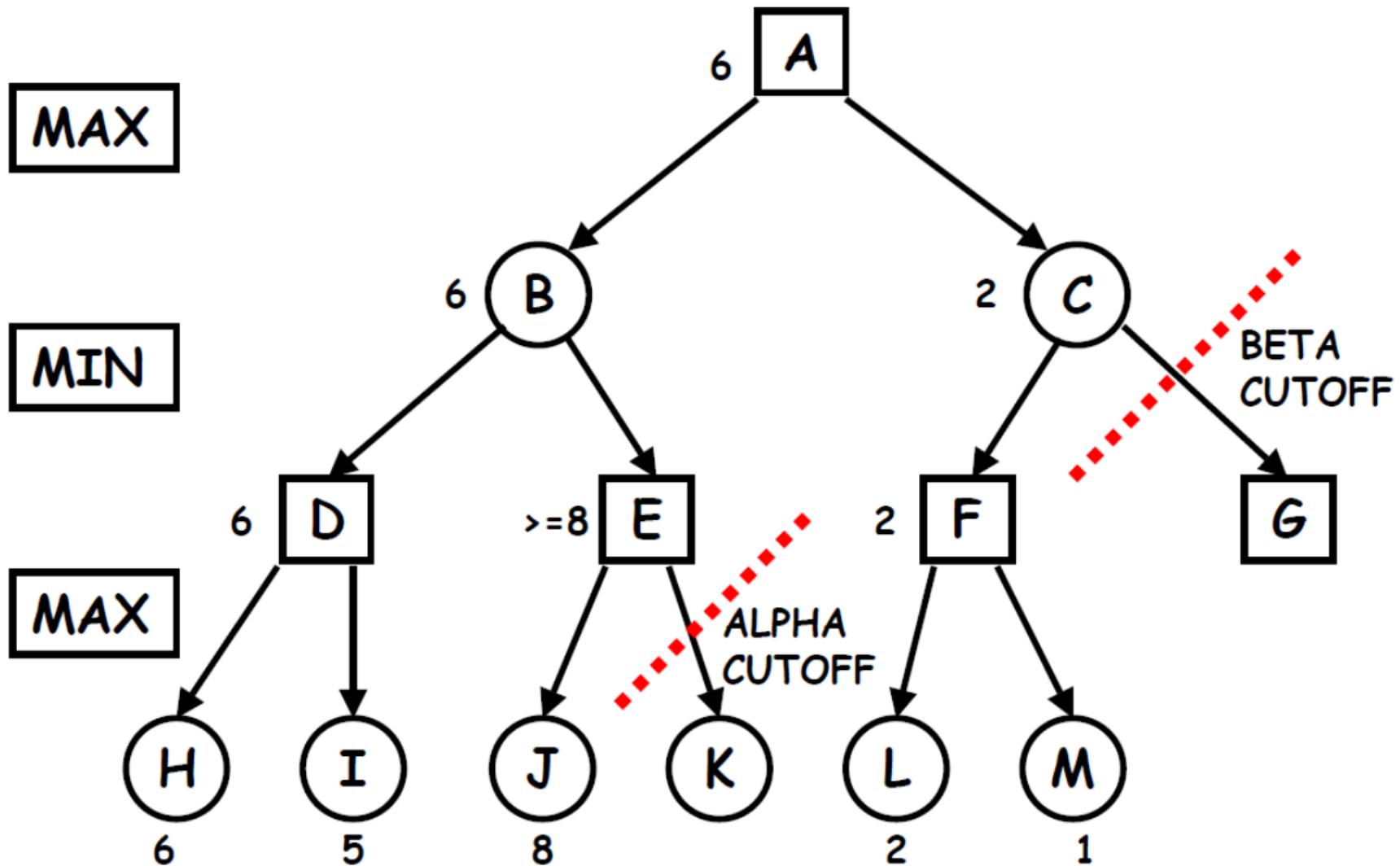
- Situación justo después de analizar el subárbol F:



2.2 Poda Alfa-Beta

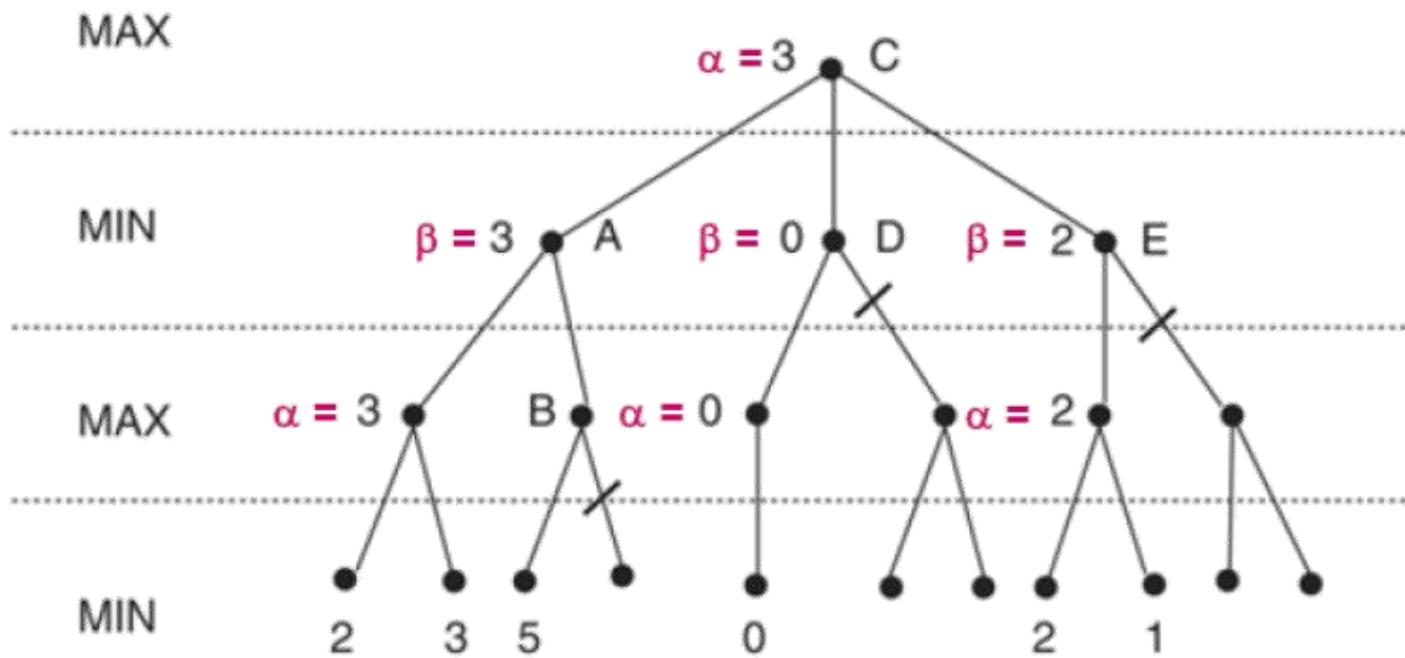


2.2 Poda Alfa-Beta





2.2 Poda Alfa-Beta



$$A \leq 3 \rightarrow \beta_A = 3$$

$B \geq 5; 5 > 3 \rightarrow B$ es β -podado

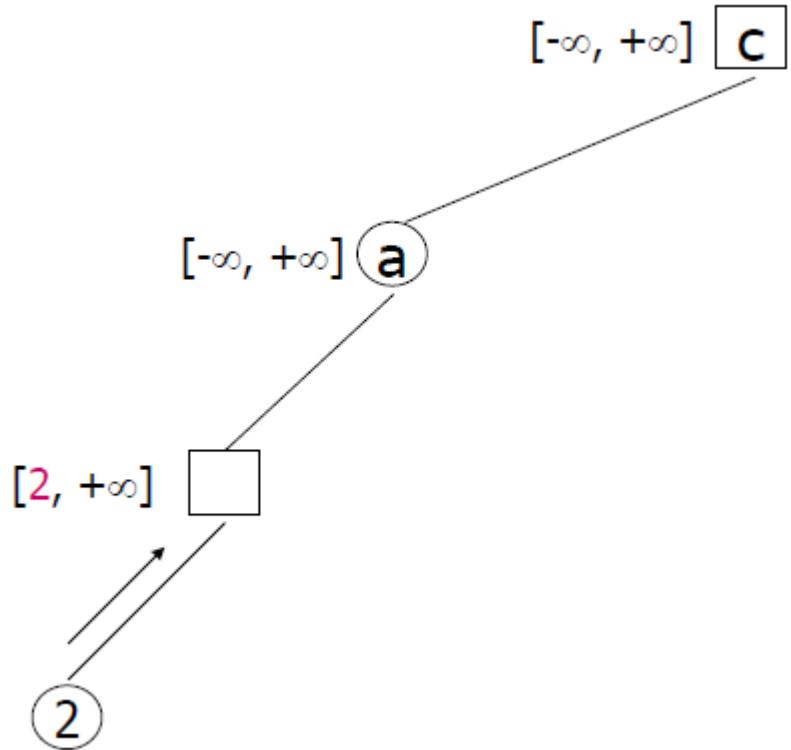
$$C \geq 3 \rightarrow \alpha_C = 3$$

$D \leq 0; 0 < 3 \rightarrow D$ es α -podado

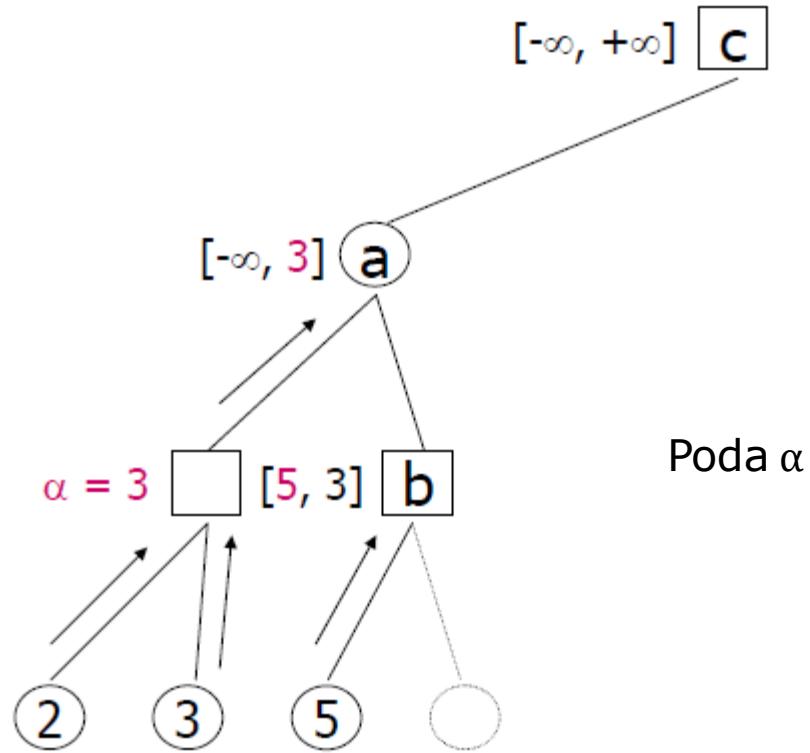
$E \leq 2; 2 < 3 \rightarrow E$ es α -podado

Valor minimax de C = 3

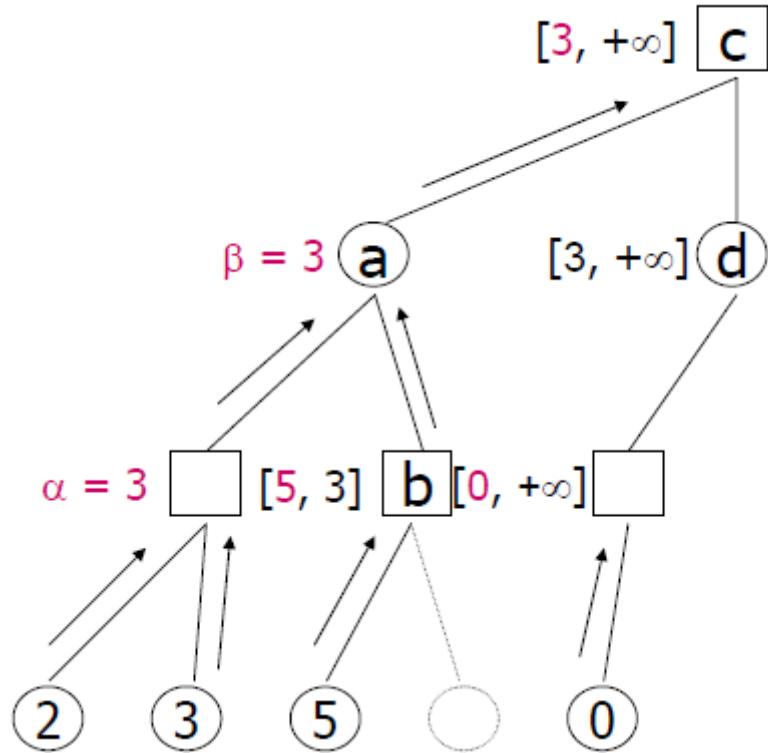
2.2 Poda Alfa-Beta



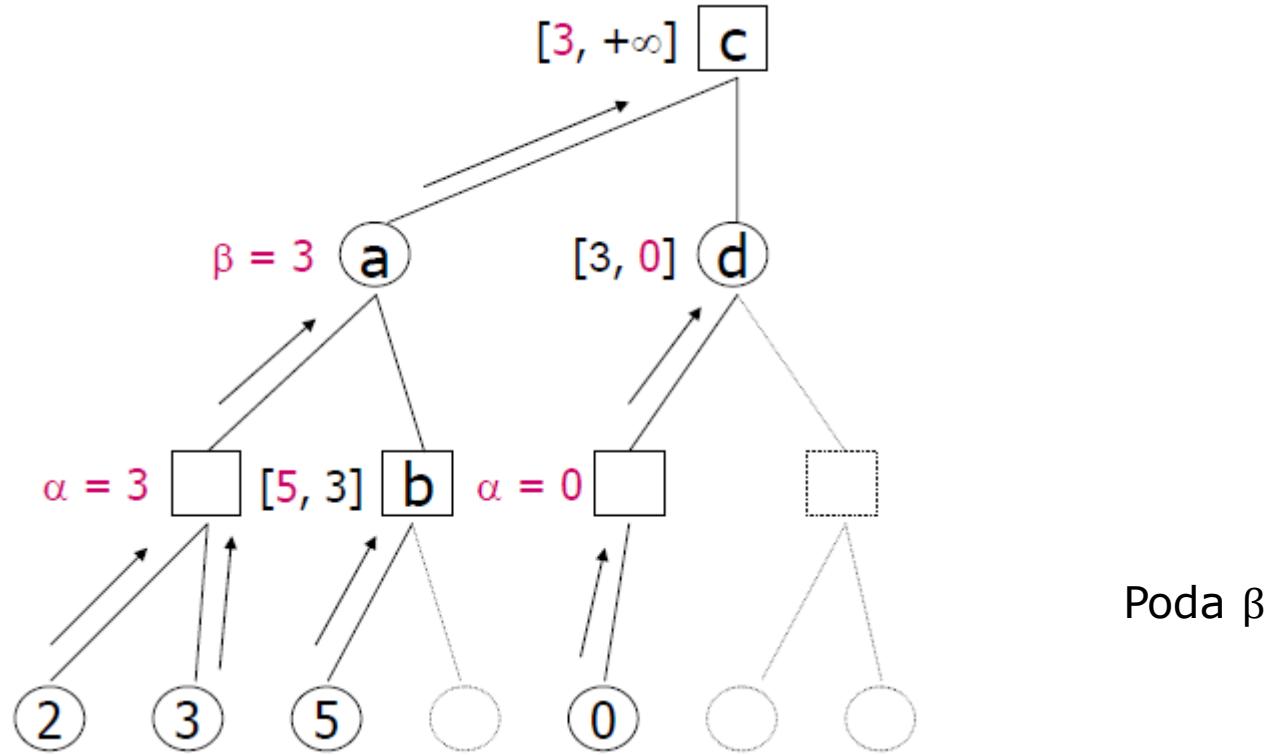
2.2 Poda Alfa-Beta



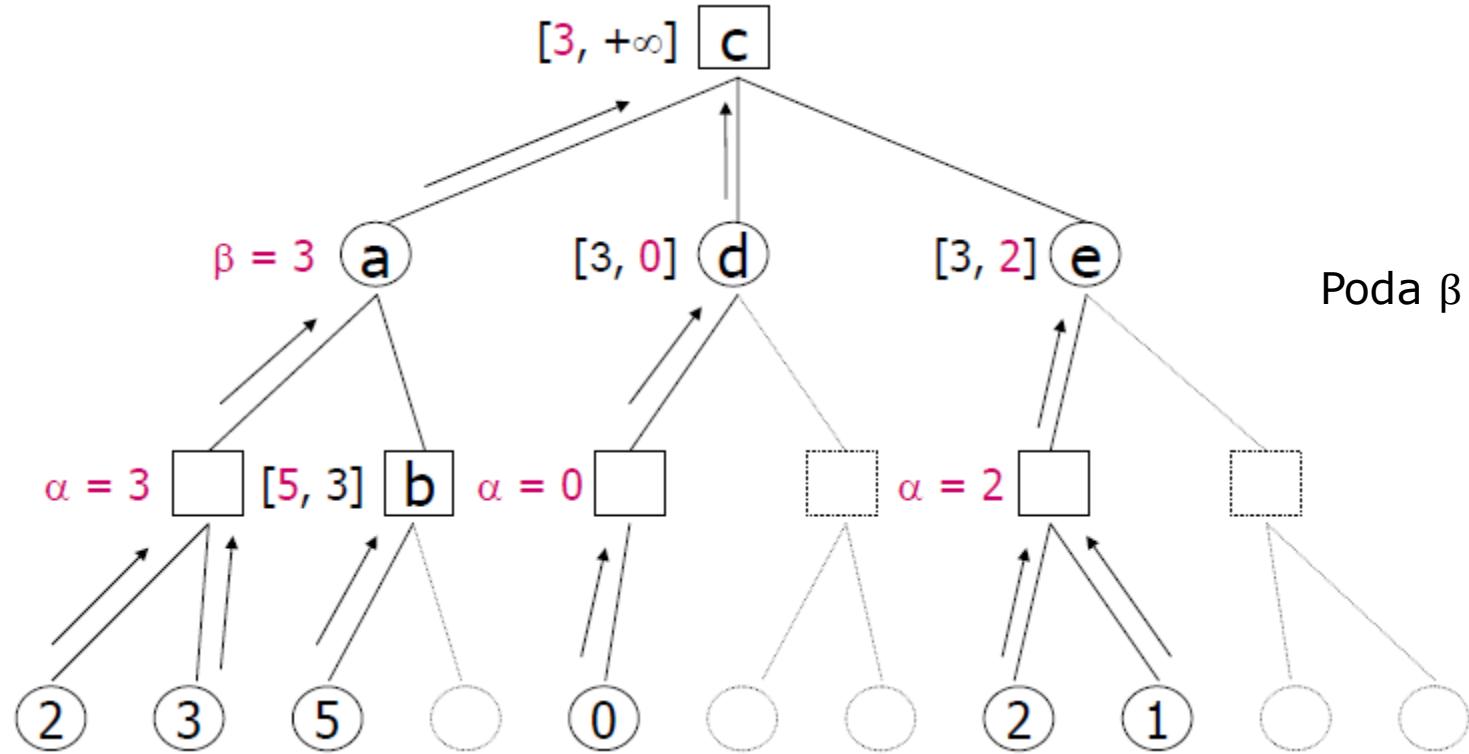
2.2 Poda Alfa-Beta



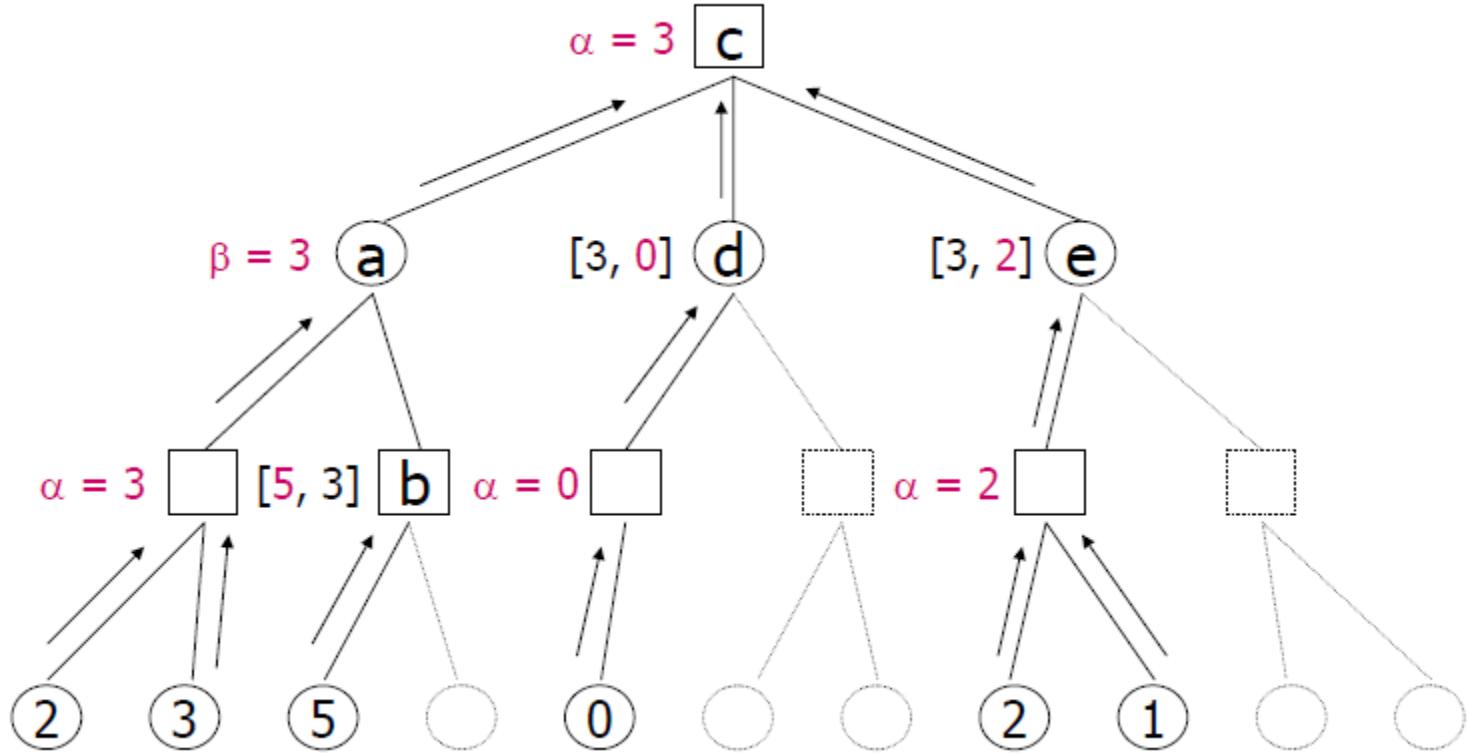
2.2 Poda Alfa-Beta



2.2 Poda Alfa-Beta



2.2 Poda Alfa-Beta



2.2 Poda Alfa-Beta



- Complejidad:
 - Complejidad en tiempo: $O(b^{3d/4})$
 - Complejidad en espacio: $O(bd)$.
- La poda $\alpha-\beta$, aplicada a árboles minimax, devuelve el mismo resultado que minimax, pero más eficientemente
 - Si los sucesores se exploran en orden de valor minimax (creciente o decreciente dependiendo de si es un nodo MIN o MAX) se produce la máxima poda
 - La complejidad temporal se reduce a $O(b^{d/2})$
 - ¡¡En el mismo tiempo se puedan considerar el doble de jugadas que en minimax!!
 - La máquina es más competitiva

- Sitio web para probar Minimax y Poda $\alpha-\beta$:

https://raphsilva.github.io/utilities/minimax_simulator/#



1. Búsqueda con adversarios
2. Búsqueda Minimax
 1. Minimax con decisiones imperfectas
 2. Poda Alfa-Beta
3. Otros tipos de juegos
4. Estado del arte en problemas de juegos



3. Otros tipos de juegos

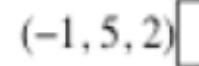
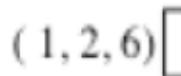
Juegos con varios jugadores

- La función de utilidad se convierte en un vector de valores

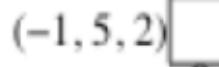
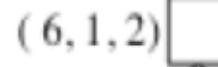
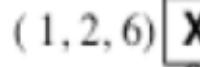
A



B



C



A

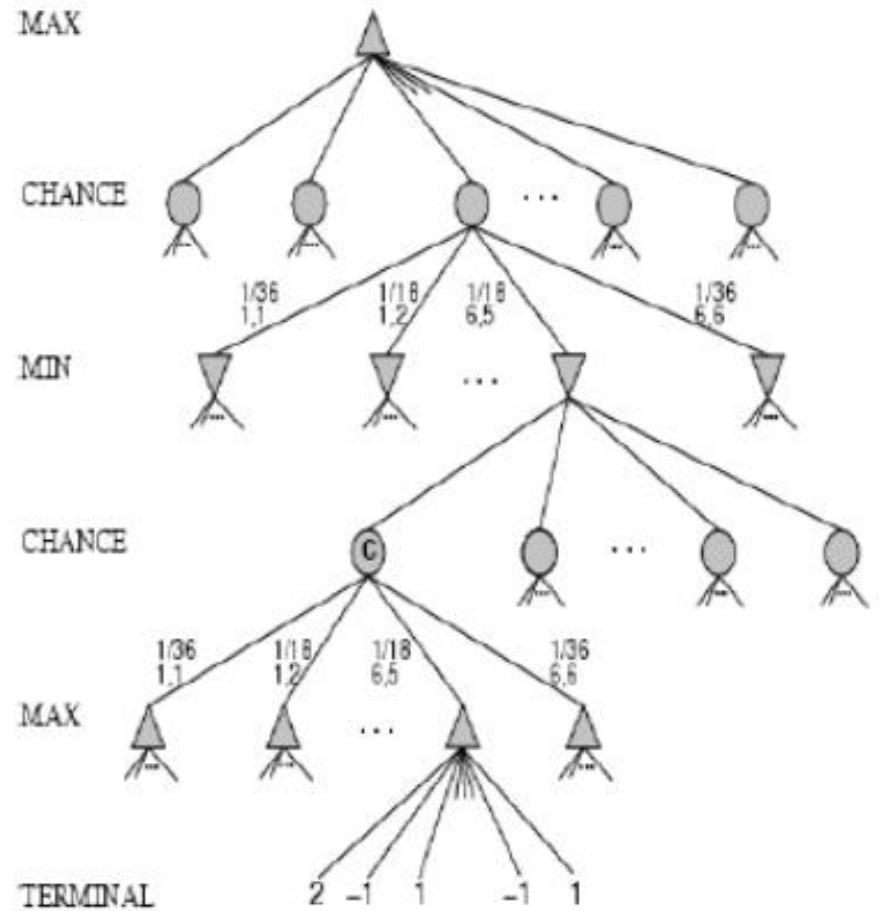




3. Otros tipos de juegos

Juegos con azar

- Se modelan probabilidades de jugada mediante un jugador ficticio (*CHANCE*)
- Se alternan nodos *MAX* y *MIN* con *CHANCE*
- El valor de la función de utilidad es aproximado
- Muy complejo





3. Otros tipos de juegos

Juegos con suma no nula

- Minimax se puede aplicar a juegos con suma cero y en los que el oponente juega a ganar
 - La ganancia o pérdida de MAX se equilibra exactamente con las pérdidas o ganancias de MIN
- Sin embargo, muchas de las situaciones del mundo real habitualmente tienen suma no nula: los participantes pueden beneficiarse o perder al mismo tiempo
 - Ejemplos: ciclismo, actividades económicas, la guerra
 - En estas situaciones, la conclusión es que resulta mejor maximizar el beneficio conjunto ([Teoría de Juegos](#))



3. Otros tipos de juegos

- Ejemplo: El dilema del prisionero

La policía arresta a dos sospechosos. No hay pruebas suficientes para condenarles, y tras haberles separado, les visita a cada uno y les ofrece el mismo trato:

	Tú niegas	Tú confiesas
Él niega	Ambos condena a 5 meses	Él 10 años, tú libre
Él confiesa	Él libre, tú 10 años	Ambos condena a 6 años

- Estrategia dominante: confesión
 - Independientemente de lo que decida el otro, puedes reducir tu condena confesando
 - Sin embargo, el resultado no es óptimo
- Estrategia óptima: colaboración* (equilibrio de Nash)
- La decisión en realidad depende de la matriz de costes



1. Búsqueda con adversarios
2. Búsqueda Minimax
 1. Minimax con decisiones imperfectas
 2. Poda Alfa-Beta
3. Otros tipos de juegos
4. Estado del arte en problemas de juegos

4. Estado del Arte en problemas de juegos



- Ajedrez:
 - Software para enseñar ajedrez o máquinas dedicadas
 - Distribución de tiempos (en % de tiempo dedicado)
 - Generación de movimientos (ordenados por h) y poda $\alpha - \beta$: 50%
 - Evaluación estática: 40%
 - Búsqueda ¡10%!
 - Aperturas y finales almacenados en bases de datos
 - DEEP BLUE (1997)
 - Derrota a Kasparov 3,5 a 2,5
 - Explora hasta 30 capas por delante
 - Mejoras algorítmicas: bajan ramificación efectiva a 3 (de 35)
 - AlphaZero (Diciembre, 2018)
- Otello
 - Los ordenadores superiores al campeón del mundo
 - LOGISTELLO (1997)

4. Estado del Arte en problemas de juegos



- Damas (oficialmente resuelto desde 2007),
 - Chinook (1994)
 - Poda alfa-beta con 444 posiciones de finales
 - BD de finales de partida: juego perfecto para configuraciones de tablero con 8 o menos piezas
 - Campeón del mundo en 1994
- Backgammon
 - Complicación: aleatoriedad (tiradas de dados)
 - Los programas de búsqueda simple son malos
 - TD-GAMMON (1995) entre los 3 mejores del mundo
 - Sistema de aprendizaje por refuerzo (mucha búsqueda y uso de los resultados para construir una muy buena función heurística)
- Bridge
 - Información oculta (las cartas de los otros jugadores)
 - Comunicación con el compañero mediante un lenguaje restringido
 - Nivel bueno



4. Estado del Arte en problemas de juegos

- Go
 - Como el ajedrez: información perfecta, no aleatoriedad ni comunicación
 - Problema: el enorme factor de ramificación ($b \sim 361$)
 - Los métodos de búsqueda que funcionan bien en ajedrez no son susceptibles de aplicarse en el go
 - Los jugadores humanos parecen basarse en algo mucho más complejo: comprensión de patrones espaciales
 - Planteamiento de métodos basados en mejores heurísticas y menos en búsqueda por fuerza bruta, con bases de conocimiento de patrones
 - Nivel de campeón (AlphaGo de Google, Marzo, 2016, Seúl)
 - AlphaGo Zero (Octubre, 2017) autoentrenado



Universidad
Francisco de Vitoria
UFV Madrid

Ingeniería del Conocimiento

Tema 6:
Búsqueda con Restricciones



Objetivos del tema

- Ubicación
 - Unidad 2: **BUSQUEDA EN ESPACIO DE ESTADOS**
 - *Tema 6: Búsqueda con Restricciones*
- Objetivos generales
 - Entender los **Problemas de Satisfacción de Restricciones** (PSR o CSP en inglés) cuyos estados y test objetivo forman una **representación simple, estándar y estructurada**.
 - Definir la **estructura de los estados** y utilizar **heurísticas de propósito general** permitiendo extrapolarse a problemas grandes.
 - Conocer la estructura del problema mediante la **representación estándar del test objetivo**.
 - Utilizar **métodos de descomposición** de problemas y entender la **conexión entre la estructura** de un problema **y la dificultad de resolverlo**.



1. Problemas de Satisfacción de Restricciones.
2. Búsqueda con vuelta atrás
 1. Variables y orden
 2. Implicaciones variables actuales
 3. Evitar repetir fracasos
3. Métodos de Búsqueda local
4. La estructura de los problemas

1. Visión General de los PSR



- Hasta ahora, el algoritmo de búsqueda ve cada estado como una caja negra (estructura de datos arbitraria accesible solo por función sucesor, función heurística y test objetivo).
 - Vamos a estudiar los **Problemas de Satisfacción de Restricciones** (PSR o CSP en inglés) cuyos estados y test objetivo forman una **representación simple, estándar y estructurada**.
 - Se definen aprovechándose de la **estructura de los estados** y utilizan **heurísticas de propósito general** (no específicas) permitiendo extrapolarse a problemas grandes.
 - La **representación estándar del test objetivo** revela la estructura del problema.
 - Se utilizan **métodos de descomposición** de problemas y una **conexión entre la estructura** de un problema **y la dificultad de resolverlo**.



- 1. Problemas de Satisfacción de Restricciones.**
- 2. Búsqueda con vuelta atrás**
 1. Variables y orden
 2. Implicaciones variables actuales
 3. Evitar repetir fracasos
- 3. Métodos de Búsqueda local**
- 4. La estructura de los problemas**



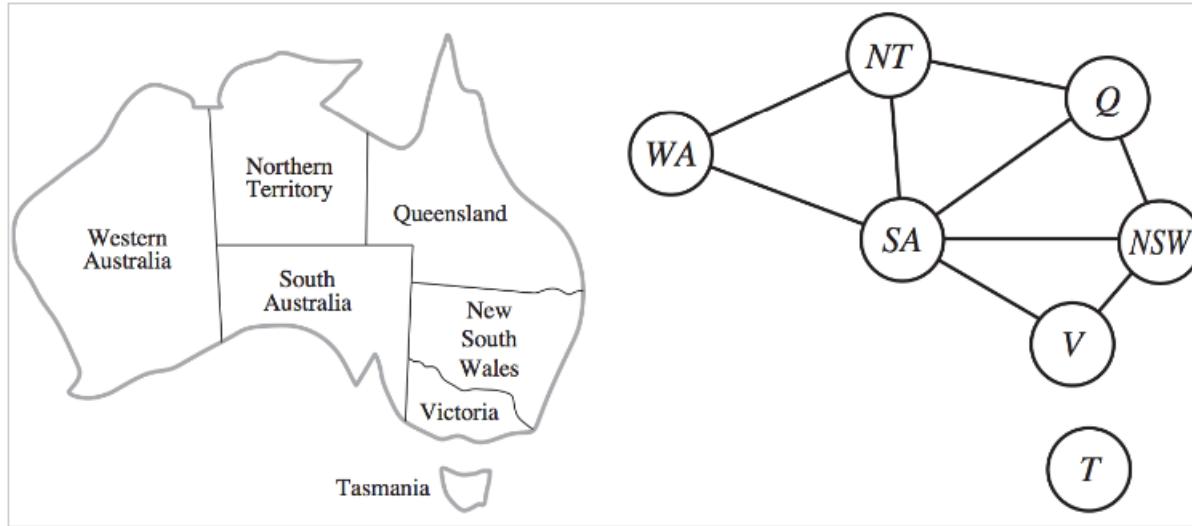
1. Problemas de satisfacción de restricciones

- La idea es **representar problemas mediante la declaración de restricciones** sobre el área del problema (el espacio de posibles soluciones) y **encontrar soluciones que satisfagan todas las restricciones**. A veces, se buscan soluciones que, además, optimicen algunos criterios determinados.
- La resolución de problemas con restricciones puede dividirse en dos ramas diferenciadas, donde se aplican metodologías distintas: aquella que trata con problemas con **dominios finitos**, y la que trata problemas sobre **dominios infinitos** o dominios más complejos.
- Nos centraremos en **problemas con dominios finitos**, que básicamente consisten en un **conjunto finito de variables**, un **dominio de valores finito para cada variable** y un **conjunto de restricciones que acotan** las posibles combinaciones de valores que estas variables pueden tomar en su dominio.



1. Problemas de satisfacción de restricciones

- Antes de seguir, un ejemplo: **problema de coloración de mapas.**



Los estados y territorios principales de Australia. Colorear este mapa puede verse como un problema de satisfacción de restricciones. El objetivo es asignar colores a cada región de modo que ninguna de las regiones vecinas tengan el mismo color.

El problema del coloreo del mapa representado como un grafo de restricciones.



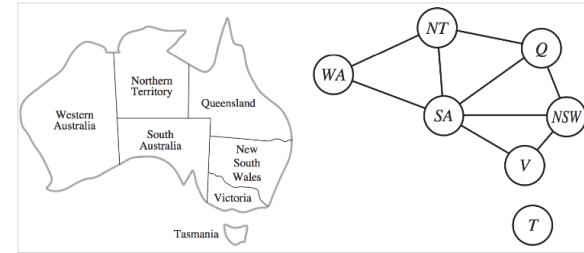
1. Problemas de satisfacción de restricciones

Suponga que, cansados de Rumanía, miramos un **mapa de Australia** que muestra cada uno de sus estados y territorios y que nos encargan la tarea de **colorear cada región de rojo, verde o azul** de modo que **ninguna de las regiones vecinas tenga el mismo color**.

Para formularlo como un PSR, **definimos las variables de las regiones**: AO , TN , Q , NGS , V , AS y T . El **dominio de cada variable** es el conjunto $\{\text{rojo, verde, azul}\}$. Las **restricciones** requieren que las regiones vecinas tengan colores distintos; por ejemplo, las combinaciones aceptables para AO y TN son los pares: $\{(rojo,verde), (rojo,azul), (verde,rojo), (verde,azul), (azul,rojo), (azul,verde)\}$

Hay **muchas soluciones posibles**, como $\{AO = \text{rojo}, TN = \text{verde}, Q = \text{rojo}, NGS = \text{verde}, V = \text{rojo}, AS = \text{azul}, T = \text{rojo}\}$

Es bueno visualizar un PSR como un **grafo de restricciones**, Los nodos del grafo corresponden a variables del problema y los arcos corresponden a restricciones.





1. Problemas de satisfacción de restricciones

Tratar un problema como un PSR confiere varias ventajas importantes:

1. Como la **representación del estado** en un PSR se ajusta a un **modelo estándar** (es decir, un conjunto de variables con valores asignados) la **función sucesor** y el **test objetivo** pueden escribirse de un **modo genérico** para que se aplique a todo PSR.
2. Además, podemos desarrollar **heurísticas eficaces y genéricas** que no requieran ninguna información adicional ni experta del dominio específico.
3. Finalmente, la **estructura del grafo de las restricciones** puede usarse para **simplificar el proceso de solución**, en algunos casos produciendo una reducción exponencial de la complejidad.



1. Problemas de satisfacción de restricciones

A un PSR se le puede dar una **formulación incremental** como en un problema de búsqueda estándar:

- **Estado inicial:** la asignación vacía $\{\}$, en la que todas las variables no están asignadas.
- **Función de sucesor:** un valor se puede asignar a cualquier variable no asignada, a condición de que no suponga ningún conflicto con variables antes asignadas.
- **Test objetivo:** la asignación actual es completa.
- **Costo del camino:** un coste constante (por ejemplo, 1) para cada paso.

Cada solución debe ser una asignación completa y por lo tanto aparecen a profundidad n si hay n variables (el árbol de búsqueda se extiende sólo a profundidad n). Por estos motivos, los algoritmos de búsqueda **primero en profundidad** son los más utilizados para solucionar PSRs.

También **el camino que alcanza una solución es irrelevante**. De ahí, que podemos usar también una **formulación completa de estados**, en la cual cada estado es una asignación completa que podría o no satisfacer las restricciones; por ello, los métodos de búsqueda local trabajan bien para esta formulación.

Hay $n!d^n$ hojas!



1. Problemas de satisfacción de restricciones

La clase más simple de PSR implica variables **discretas** y **dominios finitos** (colorear mapas). Otros ejemplos son las 8 reinas.

Los PSR con dominio finito incluyen a los **PSRs booleanos**, cuyas variables pueden ser verdaderas o falsas. Los PSRs booleanos incluyen como casos especiales algunos problemas NP-completos, como 3SAT.

Las **variables discretas** pueden tener también **dominios infinitos** (por ejemplo, el conjunto de números enteros o de cadenas). Por ejemplo, cuando programamos trabajos de la construcción en un calendario, la fecha de comienzo de cada trabajo es una variable y los valores posibles son números enteros de días desde la fecha actual. Con dominios infinitos, no es posible describir restricciones enumerando todas las combinaciones permitidas de valores. En cambio, se debe utilizar un lenguaje de restricción.

Los problemas de **satisfacción de restricciones con dominios continuos** son muy comunes en el mundo real y son ampliamente estudiados en el campo de la investigación operativa. (la programación de experimentos sobre el Telescopio Hubble requiere el cronometraje muy preciso de las observaciones; al comienzo y al final de cada observación y la maniobra son variables continuas que deben obedecer a una variedad de restricciones)



1. Problemas de satisfacción de restricciones

Ejemplo 1:

Sudoku as a Constraint Satisfaction Problem (CSP)

- Variables: 81 variables
 - A1, A2, A3, ..., I7, I8, I9
 - Letters index rows, top to bottom
 - Digits index columns, left to right
- Domains: The nine positive digits
 - $A1 \in \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$
 - Etc.; all domains of all variables are $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- Constraints: 27 *Alldiff* constraints
 - *Alldiff(A1, A2, A3, A4, A5, A6, A7, A8, A9)*
 - Etc.; all rows, columns, and blocks contain all different digits

	1	2	3	4	5	6	7	8	9
A	6		1		4		5		
B		8	3		5	6			
C	2								1
D	8		4		7				6
E		6						3	
F	7		9		1				4
G	5								2
H		7	2		6	9			
I	4	5		8				7	



1. Problemas de satisfacción de restricciones

Ejemplo 2:

Consideremos una restricción entre 4 variables x_1, x_2, x_3, x_4 , todas ellas con dominios el conjunto $\{1,2\}$, donde la suma entre las variables x_1 y x_2 es menor o igual que la suma entre x_3 y x_4 .

Esta restricción puede representarse intencionalmente mediante la expresión

$$x_1 + x_2 \leq x_3 + x_4.$$

Además, esta restricción también puede representarse extensionalmente mediante el conjunto de tuplas permitidas:

$$\{(1,1,1,1), (1,1,1,2), (1,1,2,1), (1,1,2,2), (2,1,2,2), (1,2,2,2), \\(1,2,1,2), (1,2,2,1), (2,1,1,2), (2,1,2,1), (2,2,2,2)\}$$

o mediante el conjunto de tuplas no permitidas:

$$\{(1,2,1,1), (2,1,1,1), (2,2,1,1), (2,2,1,2), (2,2,2,1)\}$$



1. Problemas de satisfacción de restricciones

Ejemplo 3:

Vamos a ver como ejemplo diversas representaciones de un mismo problema usando la representación PSR: el **problema de las N reinas**, que consiste en disponer en un tablero de ajedrez de tamaño $N \times N$, N reinas de forma que no haya amenazas entre ningún par de ellas.

Observa las **diferencias que se producen en la complejidad de las representaciones**, tanto por el **espacio de combinaciones** en las soluciones como por **las restricciones**.



1. Problemas de satisfacción de restricciones

Ejemplo 3: Primera Representación

1. $X = \{R_1, \dots, R_N\} = \{R_i : 1 \leq i \leq N\}$
2. $D = \{(x, y) : 1 \leq x, y \leq N\}$, donde $R_i = (x_i, y_i)$
3. Las restricciones:
 - No hay amenaza horizontal: $x_i \neq x_j$, para todo $i \neq j$.
 - No hay amenaza vertical: $y_i \neq y_j$, para todo $i \neq j$.
 - No hay amenaza en la diagonal principal: $x_i - x_j \neq y_i - y_j$, para todo $i \neq j$.
 - No hay amenaza en la diagonal secundaria: $x_i - x_j \neq y_j - y_i$, para todo $i \neq j$.

(Las dos diagonales se pueden tratar conjuntamente con: $|x_i - x_j| \neq |y_i - y_j|$, para todo $i \neq j$)

En esta representación el dominio es finito y hace uso de restricciones binarias de obligación. Para $N=8$ hay 64^8 posibles asignaciones y 154 restricciones ($28 + 28 + 49 + 49$)



1. Problemas de satisfacción de restricciones

Ejemplo 3: Segunda Representación

1. $X = \{R_{i,j} : 1 \leq i, j \leq N\}$ (una por cada casilla).
2. $D = \{0, 1\}$
3. Las restricciones:
 - $\sum_i R_{i,j} \leq 1$, para todo $1 \leq j \leq N$.
 - $\sum_j R_{i,j} \leq 1$, para todo $1 \leq i \leq N$.
 - $\sum_{(i,j) \in D} R_{i,j} \leq 1$, para toda D diagonal.

En esta representación el dominio es finito y hace uso de restricciones de varias aridades. Para $N=8$ hay 64^2 posibles asignaciones y unas 300 restricciones.



1. Problemas de satisfacción de restricciones

Ejemplo 3: Tercera Representación (igual a la anterior pero con predicados lógicos).

1. $X = \{R_{i,j} : 1 \leq i, j \leq N\}$ (una por cada casilla).
2. $D = \{0, 1\}$
3. Las restricciones:
 - $(R_{i,j} \rightarrow \neg R_{i,j'})$, para todo $j' \neq j$.
 - $(R_{i,j} \rightarrow \neg R_{i',j})$, para todo $i' \neq i$.
 - $(R_{i,j} \rightarrow \neg R_{i+k,j+k})$, para todo $1 \leq i+k, j+k \leq N$.
 - $(R_{i,j} \rightarrow \neg R_{i+k,j-k})$, para todo $1 \leq i+k, j-k \leq N$.

En esta representación el dominio es finito y hace uso de restricciones de varias aridades. Para $N=8$ hay 64^2 posibles asignaciones y unas 300 restricciones.

1. Problemas de satisfacción de restricciones



Ejemplo 3: Cuarta Representación.

Haciendo uso de conocimiento implícito del problema (que va a haber una, y solo una, reina en cada columna):

1. $X = \{R_i : 1 \leq i \leq N\}$
2. $D = \{1, \dots, N\}$
3. Las restricciones:
 - $R_i \neq R_j$, para todo $i \neq j$.
 - $|R_i - R_j| \neq |i - j|$, para todo $i \neq j$.

En esta representación el dominio es finito y hace uso de restricciones binarias. Para $N=8$ hay 8^8 posibles asignaciones y unas 80 restricciones.

LA REPRESENTACIÓN IMPORTA EN UN PSR!!!!



1. Problemas de Satisfacción de Restricciones.
- 2. Búsqueda con vuelta atrás**
 1. Variables y orden
 2. Implicaciones variables actuales
 3. Evitar repetir fracasos
3. Métodos de Búsqueda local
4. La estructura de los problemas

2. Búsqueda con vuelta atrás



Hemos formulado un PSR como un problema de búsqueda...

Supongamos que aplicamos la **búsqueda primero en anchura**... el factor de ramificación en la raíz es nd (cualquiera de los d valores se puede asignar a cualquiera de las n variables); en el siguiente nivel el factor de ramificación es $(n-1)d$ y así para los siguientes niveles. Es decir, generamos un árbol con **$n!d^n$ hojas** aunque haya solo d^n asignaciones posibles...

Propiedad (crucial) a todos los PSRs: **Commutatividad**. El orden de aplicación de cualquier conjunto de acciones no tiene ningún efecto sobre el resultado.

AS = rojo y AO = verde es lo mismo que AS = verde y AS = rojo.

Con esta restricción el número de hojas es d^n

2. Búsqueda con vuelta atrás



El algoritmo que utilizaremos será una **búsqueda con vuelta atrás**; este término se utiliza para una búsqueda primero en profundidad que elige valores para una variable a la vez y vuelve atrás cuando una variable no tiene ningún valor legal para asignarle.

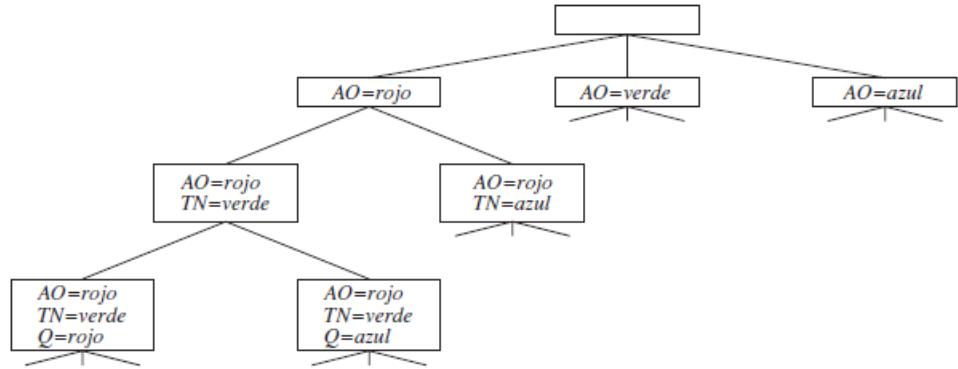
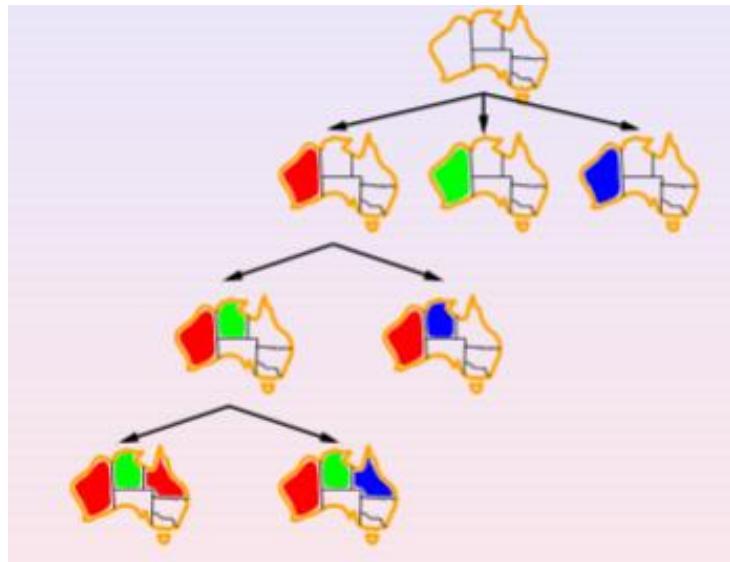
Se llama a la función recursiva VUELTAATRAS({},PSR)

Función VUELTAATRAS(Asignacion,PSR)

```
1: if Asignacion es completa then return Asignacion
2: Seleccionar una variable no asignada X //Heurística
3: for Valor posible de X do //Orden de valores. Heurística
4:   if X=Valor es consistente con las restricciones then
5:     Añadir X=Valor a Asignacion
6:     Sol ← VUELTAATRAS(Asignacion,PSR)
7:     if Sol no es No-Solucion then
8:       return Resultado
9: return No-solucion
```



2. Búsqueda con vuelta atrás



La vuelta atrás es un algoritmo sin información (según la terminología que hemos utilizado) por lo que no esperemos que sea muy eficaz para problemas grandes.



2. Búsqueda con vuelta atrás

Problema	Vuelta atrás	VA + MVR	Comprobación hacia delante	CD + MVR	Mínimo conflicto
EE.UU.	(> 1.000K)	(> 1.000K)	2K	60	64
n -reinas	(> 40.000K)	13.500K	(> 40.000K)	817K	4K
Zebra	3.859K	1K	35K	0,5K	2K
Aleatorio 1	415K	3K	26K	2K	
Aleatorio 2	942K	27K	77K	15K	

En cada celda está el número medio de comprobaciones consistentes (sobre cinco ejecuciones) requerido para resolver el problema; notemos que todas las entradas excepto las dos de la parte superior derecha están en miles (K). Los números en paréntesis significan que no se ha encontrado ninguna respuesta en el número asignado de comprobaciones.

El primer problema es colorear, con cuatro colores, los 50 estados de los Estados Unidos de América. El segundo problema cuenta el número total de comprobaciones requeridas para resolver todos los problemas de n -reinas para n de dos a 50. El tercero es el «puzle Zebra». Los dos últimos son problemas artificiales aleatorios.

2. Búsqueda con vuelta atrás



Empezamos con búsquedas sin información; las mejoramos incluyéndoles una heurística específica del dominio obtenida de nuestro conocimiento del problema (búsqueda con información).

Pero los PSRs los podemos resolver sin un conocimiento específico del dominio (métodos de propósito general) si nos respondemos a las siguientes preguntas:

1. Qué **variable** debe asignarse después, y en qué **orden** deberían intentarse sus valores?
2. ¿Cuáles son las implicaciones de las **variables actuales para el resto de variables** no asignadas?
3. Cuando un camino falla, ¿puede la búsqueda **evitar repetir este fracaso** en caminos siguientes?



1. Problemas de Satisfacción de Restricciones.
2. **Búsqueda con vuelta atrás**
 1. **Variables y orden**
 2. Implicaciones variables actuales
 3. Evitar repetir fracasos
3. Métodos de Búsqueda local
4. La estructura de los problemas

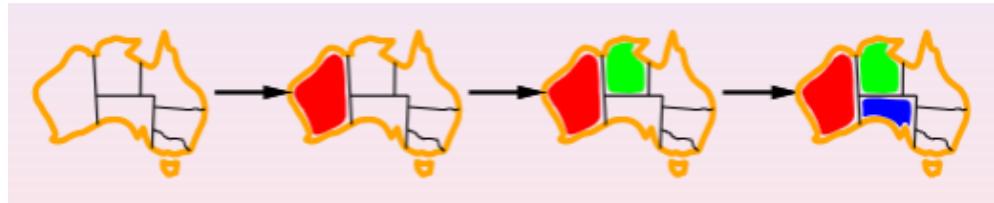


2. Búsqueda con vuelta atrás

1. Qué **variable** debe asignarse después, y en qué **orden** deberían intentarse sus valores?

Tenemos varias respuestas:

- Heurística Mínimos Valores Restantes (MVR)
- Grado heurístico
- Valor menos restringido



Heurística MVR (o variable más restringida): Elegir la variable a la que le quedan menos valores posibles (está más restringida).

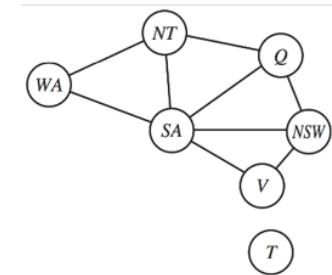
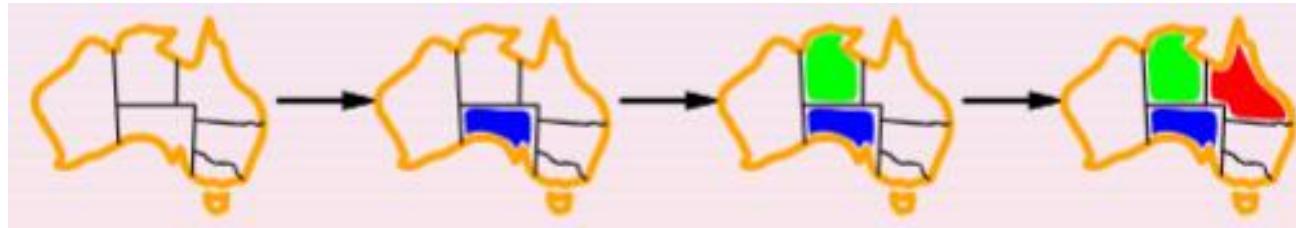
Problema	Vuelta atrás	VA + MVR	Comprobación hacia delante	CD + MVR	Mínimo conflicto
EE.UU.	(> 1.000K)	(> 1.000K)	2K	60	64
<i>n</i> -reinas	(> 40.000K)	13.500K	(> 40.000K)	817K	4K
Zebra	3.859K	1K	35K	0,5K	2K
Aleatorio 1	415K	3K	26K	2K	
Aleatorio 2	942K	27K	77K	15K	



2. Búsqueda con vuelta atrás

Grado Heurístico: la heurística MVR no ayuda en la elección de la primera región a colorear...

El Grado Heurístico intenta reducir el factor de ramificación sobre futuras opciones seleccionando la variable, entre las no asignadas, que esté implicada en el mayor número de restricciones. La heurística MVR es, en general, una guía más poderosa, pero el Grado Heurístico puede ser útil en casos de empate.

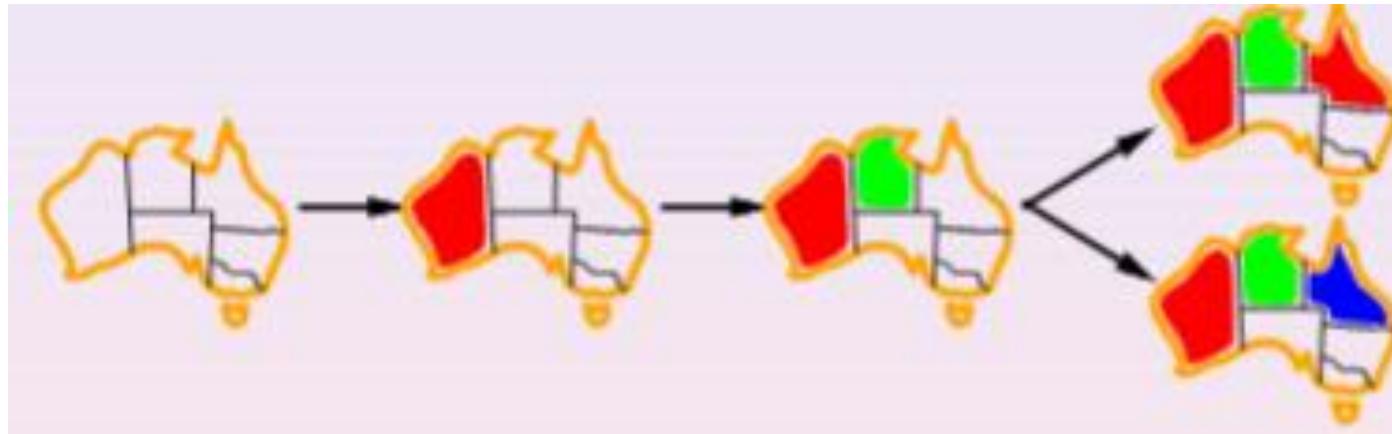


Una vez elegida *SA*, aplicando el grado heurístico que resuelve el problema sin pasos en falso puede elegir cualquier color consistente en cada punto seleccionado y todavía llegar a una solución sin vuelta atrás.



2. Búsqueda con vuelta atrás

Valor menos restringido: Dada una variable, elegir el valor que restringe menos los valores de las variables no asignadas.



Se elige $Q = \text{Roja}$.

Trata de dejar la flexibilidad máxima de las asignaciones de las variables siguientes (no válido si queremos encontrar todas las soluciones o el problema no tiene solución).



1. Problemas de Satisfacción de Restricciones.
2. **Búsqueda con vuelta atrás**
 1. Variables y orden
 2. **Implicaciones variables actuales**
 3. Evitar repetir fracasos
3. Métodos de Búsqueda local
4. La estructura de los problemas



2. Búsqueda con vuelta atrás

2. ¿Cuáles son las implicaciones de las **variables actuales para el resto de variables** no asignadas?

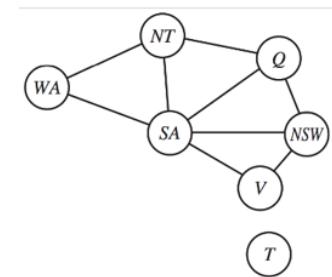
Tenemos varias respuestas:

- Comprobación hacia delante
- Propagación de restricciones
- Manejo de restricciones especiales



2. Búsqueda con vuelta atrás

- **Comprobación hacia delante:** Siempre que se asigne una variable X, el proceso de búsqueda hacia delante mira cada variable no asignada Y que esté relacionada con X por una restricción y suprime del dominio de Y cualquier valor que sea inconsistente con el valor elegido para X. Si hay una variable sin valores posible vuelve atrás.



WA	NT	Q	NSW	V	SA	T
Red	Green	Blue	Red	Green	Blue	Red



WA	NT	Q	NSW	V	SA	T
Red	Green	Blue	Red	Green	Blue	Red
Red		Green	Blue	Green	Blue	Red

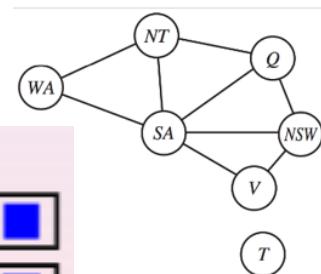


WA	NT	Q	NSW	V	SA	T
Red	Green	Blue	Red	Green	Blue	Red
Red	Green	Blue	Red	Green	Blue	Red



2. Búsqueda con vuelta atrás

■ Comprobación hacia delante:

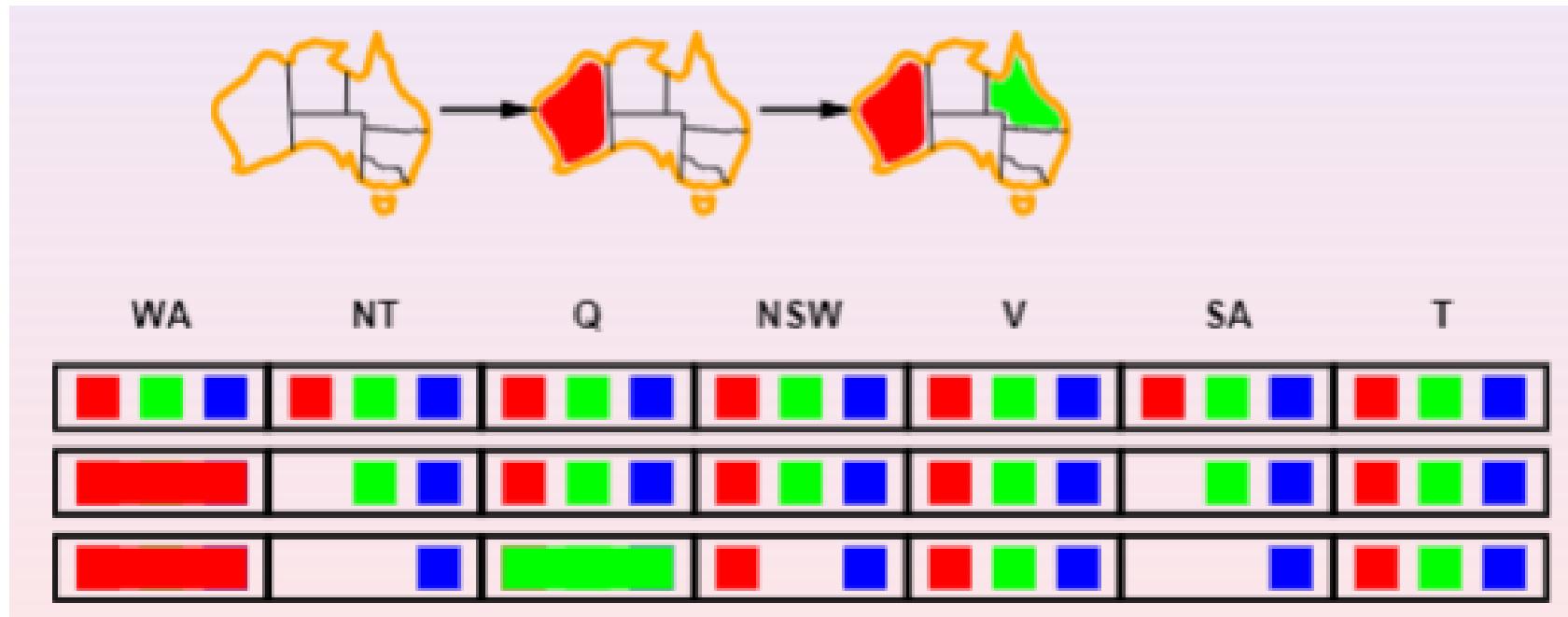


Problema	Vuelta atrás	VA + MVR	Comprobación hacia delante	CD + MVR	Mínimo conflicto
EE.UU. <i>n</i> -reinas	(> 1.000K) (> 40.000K)	(> 1.000K) 13.500K	2K (> 40.000K)	60 817K	64 4K
Zebra	3.859K	1K	35K	0,5K	2K
Aleatorio 1	415K	3K	26K	2K	
Aleatorio 2	942K	27K	77K	15K	



2. Búsqueda con vuelta atrás

- **Comprobación hacia delante:** La comprobación hacia delante no descubre todas las inconsistencias, por ejemplo, después de asignar WA=R y Q=V, tanto NT como SA están obligadas a tomar el valor A, pero como son adyacentes no pueden tomar el mismo valor.





2. Búsqueda con vuelta atrás

Propagación de restricciones: Una forma más sofisticada de comprobar que los valores son los **arcos consistentes**.

Se basa en comprobar que toda arista ordenada (X,Y) del grafo de restricciones es consistente.

Si las restricciones no son binarias, se puede aplicar pero es más complejo. Habría que considerar las aristas consistentes para toda pareja ordenada (X,Y) tales que haya una restricción que involucre a ambas variables.

La consistencia de los arcos se puede hacer al principio y, de forma recursiva, habría que comprobar la consistencia de todo arco (X,Z) después de que Z haya perdido alguno de sus valores posibles

Arco consistente: Un arco (X,Y) es consistente si para todo valor posible x de X existe un valor posible y de la variable Y .

Si una arco no es consistente, se eliminan todos los valores posibles de X para los que no exista un valor consistente de Y .



2. Búsqueda con vuelta atrás

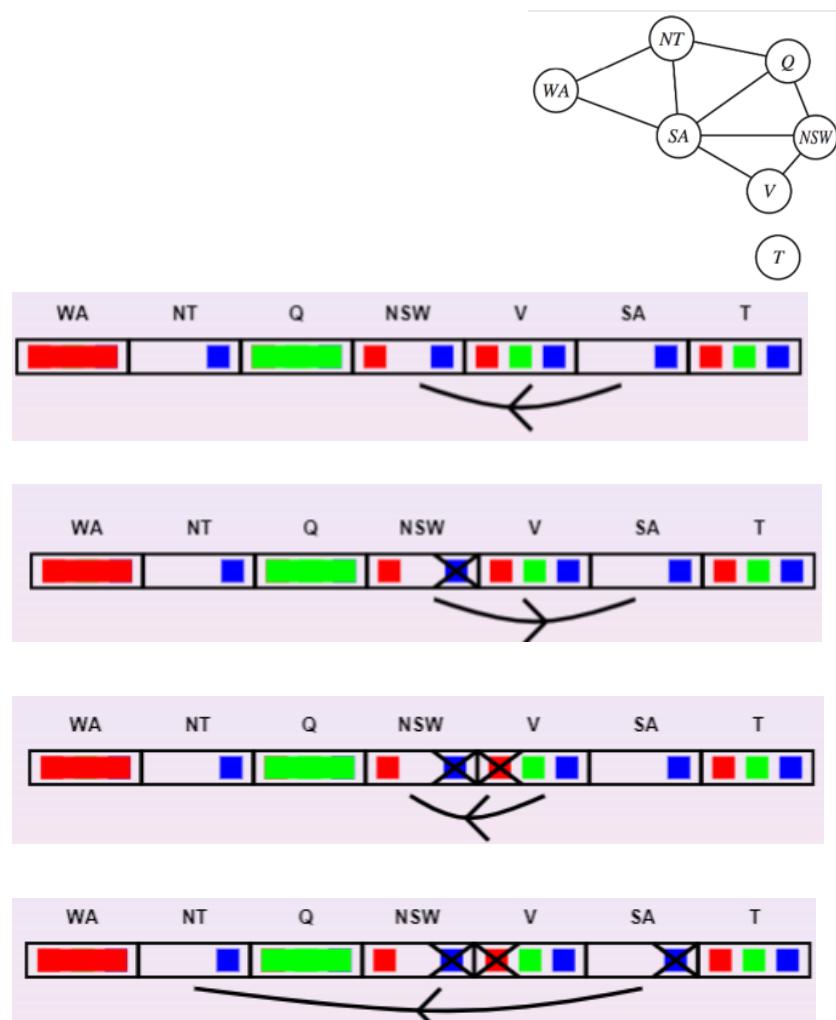
Propagación de restricciones :

- Ejemplo de arco consistente



WA	NT	Q	NSW	V	SA	T
Red	Blue	Blue	Blue	Blue	Green	Blue
Red	White	Blue	Blue	Blue	Green	Blue
Red	White	Green	Red	Red	Blue	Blue

Se comprueba que no existe ningún valor posible para SA con lo cual podemos realizar una vuelta atrás.





2. Búsqueda con vuelta atrás

Propagación de restricciones :

- **Arco consistente – Algoritmo AC-3**

El siguiente algoritmo se aplica al principio a todos los arcos (X,Y) del grafo de restricciones y, se aplica a todo arco (Z,X) cada vez que X restringe el conjunto de valores posibles.

```
1: for Cada valor posible x de X do
2:   if No existe un valor y de Y que sea compatible con x then
3:     Eliminar x del conjunto de valores posibles de X
```

La propagación de arcos consistentes (AC-3) realiza una inferencia para eliminar posibles valores de las variables y determinar si hay inconsistencia en una asignación parcial. Sin embargo, **este método no es completo**: es posible que haya inconsistencia y no se detecte. Por ejemplo, si queremos determinar al principio si se puede colorear a Australia con 2 colores. No es posible, pero AC-3 no lo detecta. Existen otros métodos más potentes. Siempre hay que buscar el equilibrio entre el coste de la inferencia y lo que nos ahorraremos en la búsqueda.



2. Búsqueda con vuelta atrás

- **Manejo de Restricciones Especiales:**

1. **Restricciones todas distintas:** Todas Distintas (X_1, \dots, X_m)

Si hay m variables implicadas en la restricción, y si tienen n valores posibles distintos, y $m > n$, entonces la restricción no puede satisfacerse. Mejora el criterio de arcos consistentes.

2. **Restricciones de recursos:**

Dos vuelos, 271 y 272 con capacidades 156, 385 respectivamente: Vuelo 271 pertenece [0,165] y Vuelo 272 pertenece [0,385]

Restricción adicional: los dos vuelos juntos deben llevar a 420 personas.

Entonces se deduce: Vuelo 271 pertenece [35, 165] y

Vuelo 272 pertenece [255, 385]

Esto se conoce como **propagación de límites**.

.



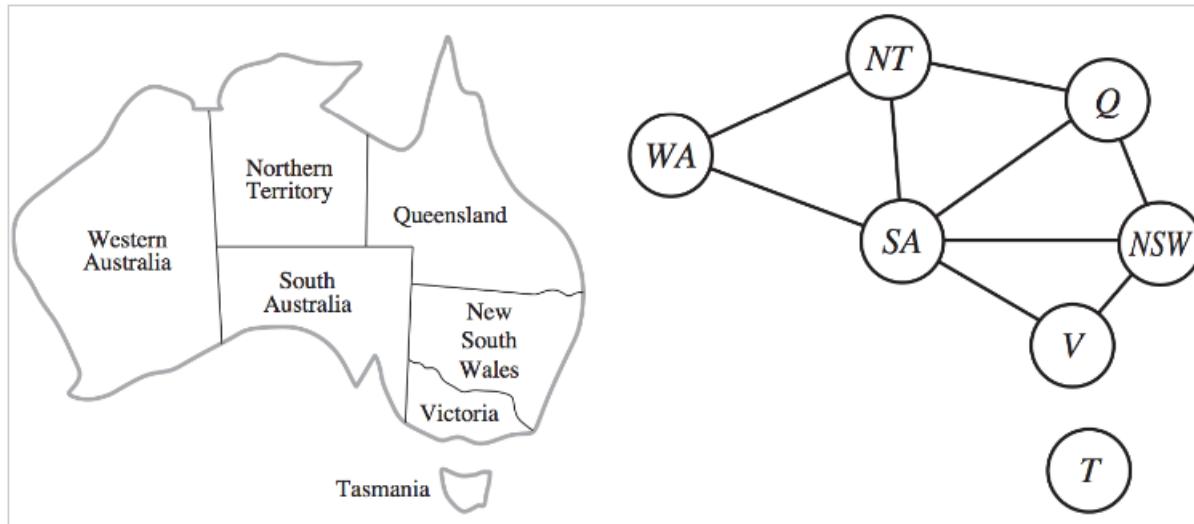
1. Problemas de Satisfacción de Restricciones.
2. **Búsqueda con vuelta atrás**
 1. Variables y orden
 2. Implicaciones variables actuales
 3. **Evitar repetir fracasos**
3. Métodos de Búsqueda local
4. La estructura de los problemas



2. Búsqueda con vuelta atrás

3. Cuando un camino falla, ¿puede la búsqueda **evitar repetir este fracaso** en caminos siguientes?

Vuelta Atrás Inteligente: Se visita el punto de decisión más reciente.



Orden de actuación: Q, NSW, V, T, SA, WA, NT

{Q = rojo, NSW = verde, V = azul, T = rojo}

SA ?

Vuelta atrás a T inútil



2. Búsqueda con vuelta atrás

■ **Vuelta Atrás Inteligente:**

Conjunto conflicto para la variable X es el conjunto de variables previamente asignadas Y, tales que el valor asignado a Y evita uno de los valores de X, debido a la restricción entre ambas.

El método de salto-atrás retrocede a la variable más reciente en el conjunto conflicto.

Ejemplo:

Conjunto conflicto para AS:{Q,NSW,V}

El método salto-atrás retrocede a V sin considerar T

La propagación hacia delante puede proporcionar el conjunto de conflicto inicial.

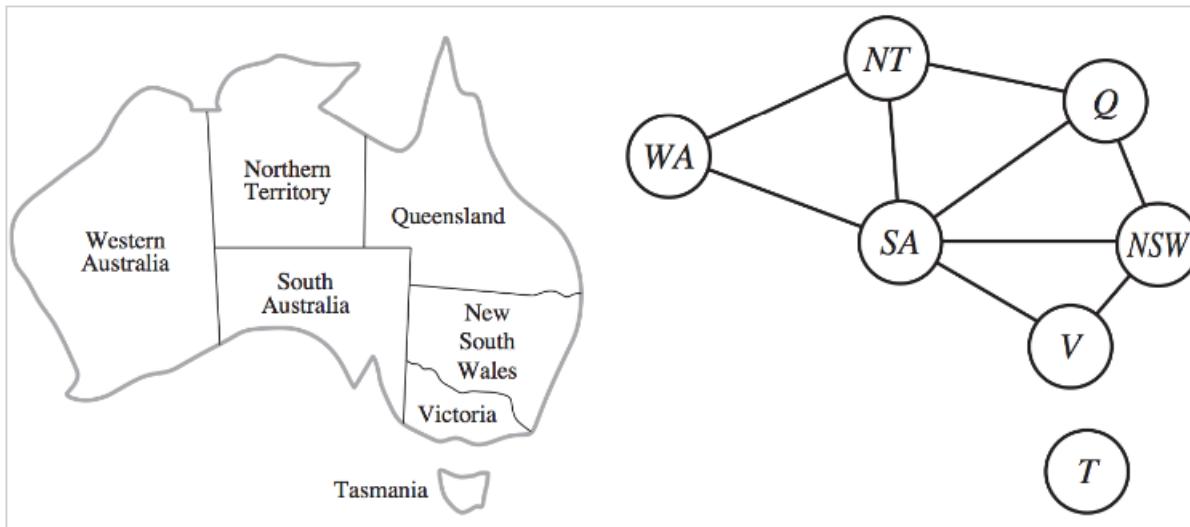
Si desde X_i volvemos hasta X_j , porque X_i se ha quedado sin valores, entonces debemos de actualizar el conflicto de X_j con:

$$\text{Conf}(X_j) \leftarrow \text{Conf}(X_j) \cup \text{Conf}(X_i) - \{X_j\}.$$



2. Búsqueda con vuelta atrás

■ Vuelta Atrás Inteligente:



1. Orden de actuación: WA, NSW, T, NT, Q, V, SA
2. Asignación parcial: WA=Rojo, NSW=Rojo, T=Rojo
3. Cuando NT se queda sin valores, entonces su conjunto de conflicto es:
{WA,NSW}
 - Cuando SA falla, su conjunto de conflicto es {WA,NT,Q}.
 - Volvemos a Q, con conjunto de conflicto {NT,NSW}, que queda actualizado a {WA,NT,NSW}.
 - Volvemos a NT con conjunto de conflicto {WA} que se actualiza a {WA,NSW}



1. Problemas de Satisfacción de Restricciones.
2. Búsqueda con vuelta atrás
 1. Variables y orden
 2. Implicaciones variables actuales
 3. Evitar repetir fracasos
- 3. Métodos de Búsqueda local**
4. La estructura de los problemas



3. Métodos de Búsqueda Local

Volvemos a la formulación completa de estados.

Aplicamos **algoritmos de búsqueda local**: ascensión de colinas, enfriamiento simulado, algoritmos genéticos, etc.

Heurística: Mínimos conflictos. Seleccionar el **valor que cause el número mínimo de conflictos** con otras variables.

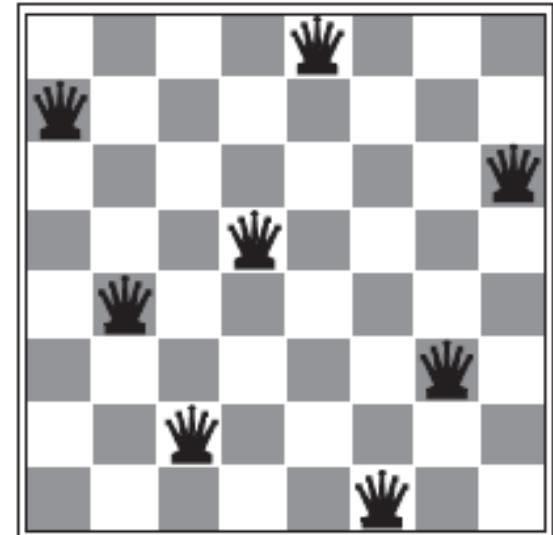
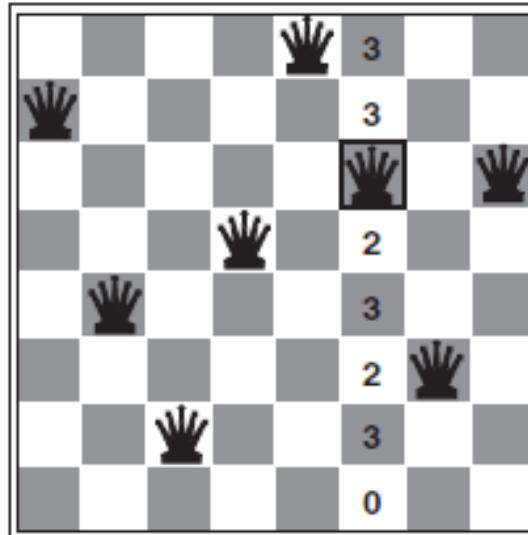
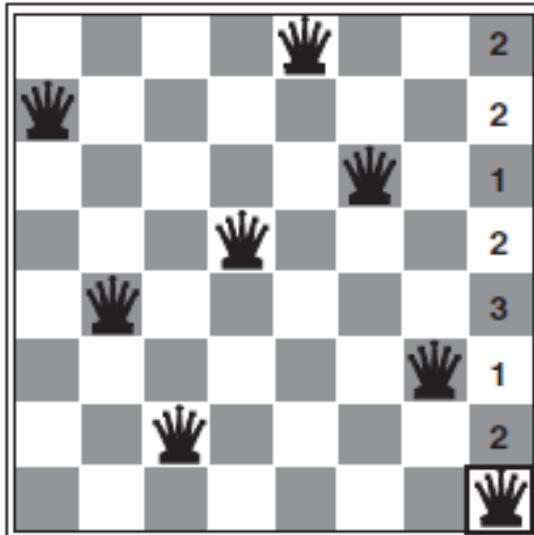
Problema	Vuelta atrás	VA + MVR	Comprobación hacia delante	CD + MVR	Mínimo conflicto
EE.UU. <i>n</i> -reinas	(> 1.000K) (> 40.000K)	(> 1.000K) 13.500K	2K (> 40.000K)	60 817K	64 4K
Zebra	3.859K	1K	35K	0,5K	2K
Aleatorio 1	415K	3K	26K	2K	
Aleatorio 2	942K	27K	77K	15K	

3. Métodos de Búsqueda Local



Ejemplo: 8 reinas.

En cada etapa, se elige una reina para la reasignación en su columna. El número de conflictos (en este caso, el número de reinas atacadas) se muestra en cada cuadrado. El algoritmo mueve a la reina a un cuadrado de mínimo conflicto, deshaciendo los empates de manera aleatoria.



3. Métodos de Búsqueda Local



Con un estado inicial aleatorio, con esta heurística se puede resolver el **problema de las n-reinas en tiempo constante** y con una alta probabilidad (por ejemplo con $n = 10,000,000$).

La técnica de los mínimos conflictos es sorprendentemente eficaz para muchos PSRs, en particular, cuando se parte de un estado inicial razonable.

Por ejemplo, se ha utilizado para programar las observaciones del telescopio Hubble, reduciendo el tiempo utilizado para programar **1 semana** de observaciones, de **3 semanas (!!??)** a alrededor de **10 minutos**.

Otra ventaja es que se puede utilizar en ajuste online cuando las condiciones del problema cambian.



1. Problemas de Satisfacción de Restricciones.
2. Búsqueda con vuelta atrás
 1. Variables y orden
 2. Implicaciones variables actuales
 3. Evitar repetir fracasos
3. Métodos de Búsqueda local
4. **La estructura de los problemas**

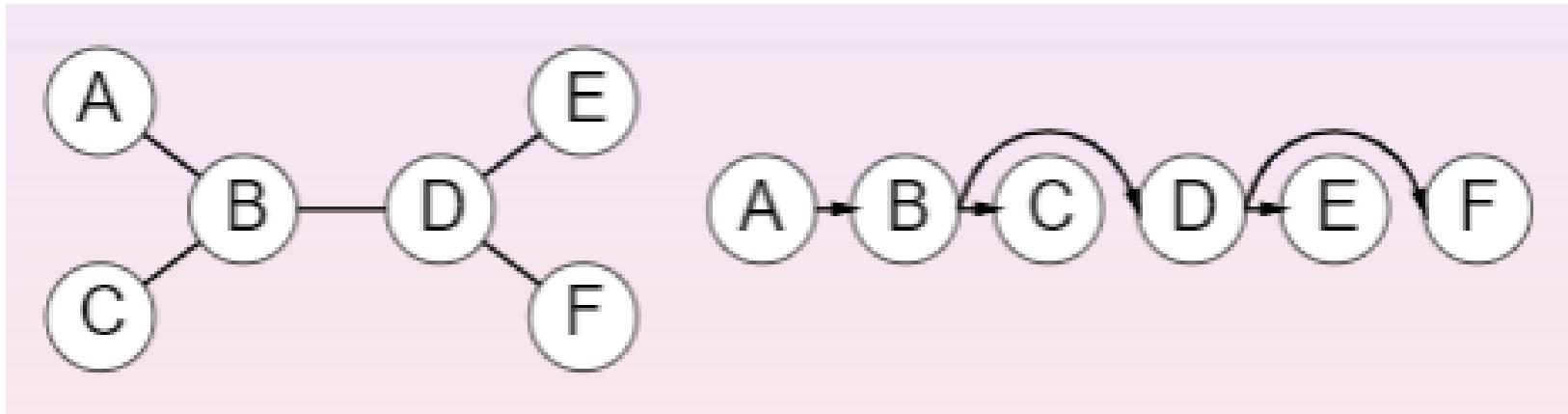
4. La estructura de los problemas



Idea: aprovecharse de la existencia de problemas independientes.

Técnicas:

1. Algoritmos en árboles.
2. Método del condicionamiento.
3. Construcción de árbol agrupando variables.



4. La estructura de los problemas



Algoritmos en árboles.

Algoritmo:

1. Elegir un nodo como raíz
2. Ordenar los nodos desde esta raíz a las hojas, de forma que el padre de cada nodo lo preceda en este orden.
3. Elegir todos los nodos X_j en orden inverso y aplicar la consistencia de arco entre X_i y X_j , donde X_i es el padre de X_j . Eliminar los valores de X_i que sea necesario.
4. Recorrer los nodos en orden directo empezando por la raíz. Para cada nodo X_j elegir un valor que sea consistente con el elegido para su padre X_i

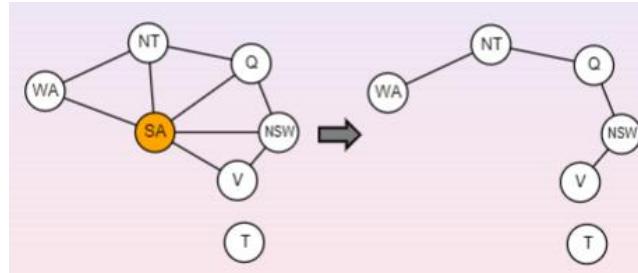
El algoritmo tiene complejidad $O(nd^2)$ donde n es el número de variables y d el número de casos por variable.

4. La estructura de los problemas



Método del condicionamiento

Si el grafo no es un árbol, se puede tratar de conseguir un conjunto pequeño de variables S , tal que si las quitamos del grafo de restricciones, nos queda un árbol.



1. Elegir un subconjunto S de variables que al quitarlas nos quede una estructura de árbol
2. **for** Cada posible asignación consistente de valores s de S **do**
 1. Quitar del resto de las variables, los valores inconsistentes con estos valores
 2. Resolver el problema de satisfacción de restricciones para el resto de las variables (árbol)
 3. Si se encuentra la solución, adjuntarla a $S=s$ para obtener una solución al problema original y parar

4. La estructura de los problemas



Construcción de un árbol agrupando variables.

¿Los grafos de restricción pueden reducirse a árboles?

Si. Mediante un proceso de agrupación de variables:

- Cada variable tiene que estar, al menos, en un grupo, pero puede estar en varios.
- Si dos variables están unidas en el grafo, debe de existir un grupo que las contenga.
- Si una variable está en dos grupos, debe de estar en todo grupo que se encuentre en el camino entre ambas.

Si un grafo tiene la anchura de árbol w ,
y **nos dan la descomposición en árboles correspondiente**, entonces el problema
puede resolverse en $O(nd^{w+1})$ veces.

Los PSRs con grafos de restricciones de
anchura de árbol acotada son resolubles
en tiempo polinomial.

