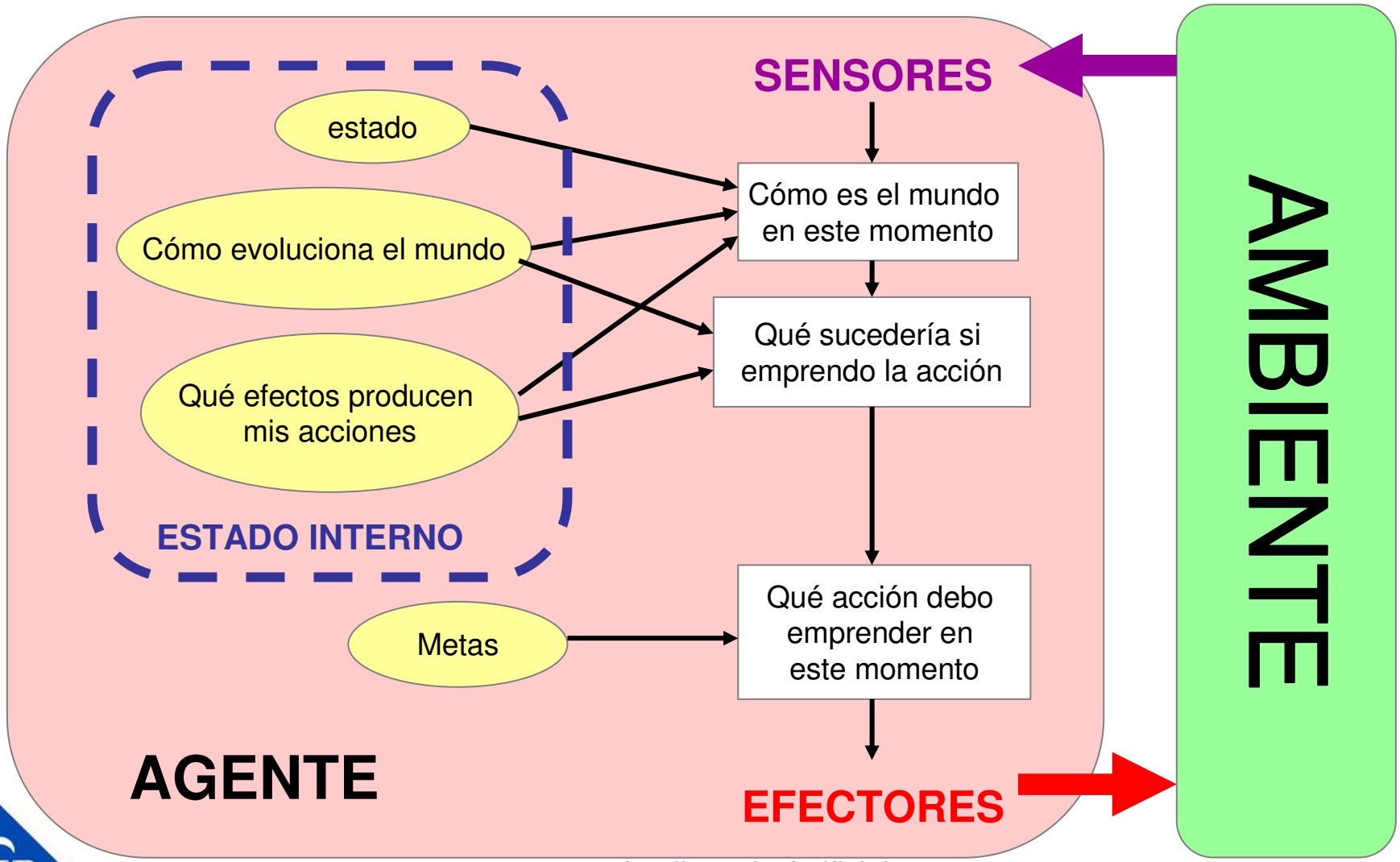
Agentes basado en metas: búsqueda

Agente basado en metas



Resolución de problemas

- La resolución de problemas es una capacidad que consideramos inteligente
- Somos capaces de resolver problemas muy diferentes
  - Encontrar el camino en un laberinto
  - Resolver un crucigrama
  - Jugar a un juego
  - Diagnosticar una enfermedad
  - Decidir si invertir en bolsa
  - ...
- El objetivo es que un programa también sea capaz de resolverlos

Resolución de problemas

- Queremos definir cualquier tipo de problema de manera que se pueda resolver automáticamente
- Necesitamos:
  - Una representación común para todos los problemas
  - Algoritmos que usen alguna estrategia para resolver problemas definidos en esa representación común

Definición de un problema

Si abstraemos los elementos de un problema podemos identificar:

- Un punto de partida
- Un objetivo a alcanzar (meta)
- Acciones a nuestra disposición para resolver el problema
- Restricciones sobre el objetivo
- Elementos que son relevantes en el problema definidos por el tipo de dominio

Representación de problemas

- Existen diferentes formas de representar problemas para resolverlos de manera automática
- Representaciones generales
  - **Espacio de estados:** un problema se divide en un conjunto de pasos de resolución desde el inicio hasta el objetivo
  - **Reducción a subproblemas:** un problema se puede descomponer en una jerarquía de subproblemas
- Representaciones para problemas específicos
  - **Resolución de juegos**
  - **Satisfacción de restricciones**

Representación de problemas: estados

- Podemos definir un problema por los elementos que intervienen y sus relaciones
- En cada instante de la resolución de un problema esos elementos tendrán unas características y relaciones específicas
- Denominaremos **Estado** a la representación de los elementos que describen el problema en un momento
- Distinguiremos dos estado especiales el **Estado Inicial** (punto de partida) y el **Estado Final** (objetivo del problema)
- ¿Que incluir en el estado?

Ejemplo: misioneros y caníbales

- Estados:
  - número misioneros lado izquierdo, número caníbales lado izquierdo, posición bote (izquierda o derecha)
- Operadores: condiciones
  - Transportar 1 misionero y 1 caníbal
  - Transportar 1 misionero
  - Transportar 1 caníbal
  - Transportar 2 misioneros
  - Transportar 2 caníbales
  - Se debe verificar la condición: No puede ocurrir nunca que haya un número mayor de caníbales que misioneros (se los comerían)
- Coste operador: 1

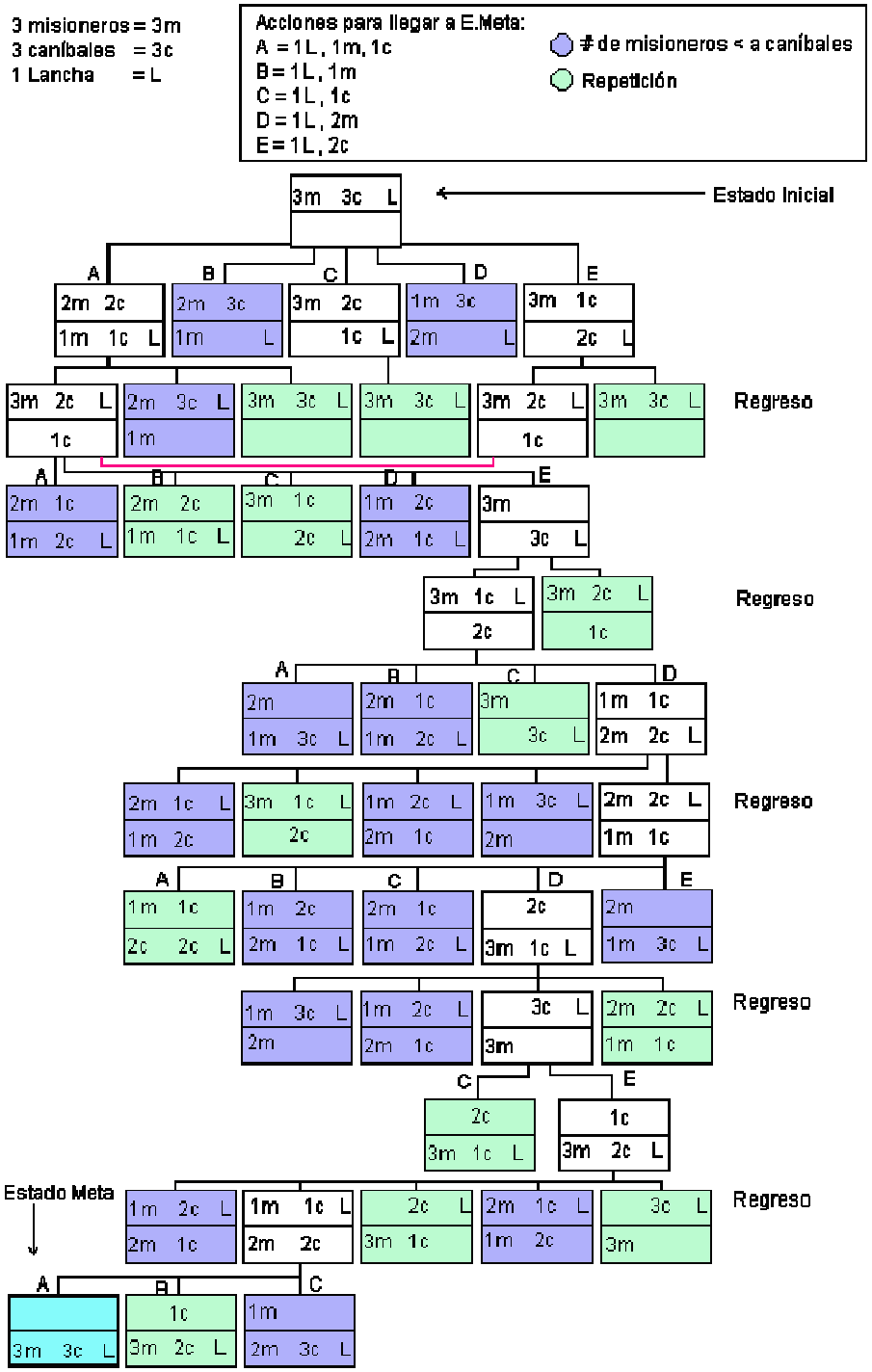
3m	3c	L

Modificación del estado: operadores

- Para poder movernos entre los diferentes estados necesitamos operadores de transformación
- **Operador:** Función de transformación sobre la representación de un estado que lo convierte en otro estado
- Los operadores definen una relación de accesibilidad entre estados
- Representación de un operador:
  - Condiciones de aplicabilidad
  - Función de transformación
- ¿Que operadores? ¿Cuantos? ¿Que granularidad? (¿a qué nivel definimos el dominio?)

Operadores (acciones):

- A = L, 1m, 1c
- B = L, 1m
- C = L, 1c
- D = L, 2m
- E = L, 2c


3 misioneros = 3m
3 caníbales = 3c
1 Lancha = L
Acciones para llegar a E.Meta:
A = 1L, 1m, 1c
B = 1L, 1m
C = 1L, 1c
D = 1L, 2m
E = 1L, 2c
# de misioneros < a caníbales
Repetición
Estado Inicial
Regreso
Regreso
Regreso
Regreso
Regreso
Estado Meta

Espacio de estados

- Los estados y su relación de accesibilidad conforman lo que se denomina **espacio de estados**
- Representa todos los caminos que hay entre todos los estados posibles de un problema
- Podría asimilarse con un mapa de carreteras de un problema
- La solución de nuestro problema esta dentro de ese mapa

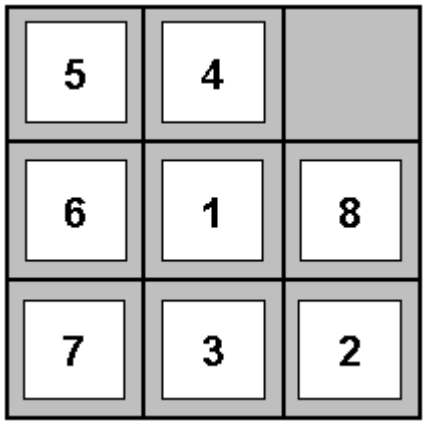
Solución de un problema en el espacio de estados

- **Solución:** Secuencia de pasos que llevan del estado inicial al final (secuencia de operadores) o también el estado final
- **Tipos de solución:** una cualquiera, la mejor, todas
- **Coste de una solución:** Gasto en recursos de la aplicación de los operadores a los estados. Puede ser importante o no según el problema y que tipo de solución que busquemos

Descripción de un problema en el espacio de estados

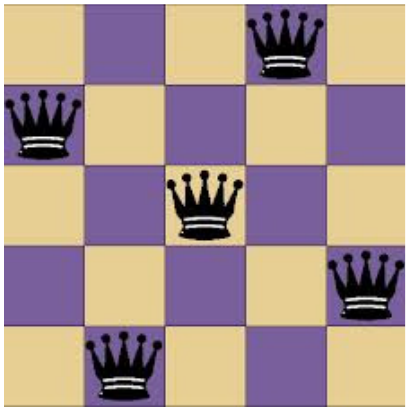
- Definir el conjunto de estados del problema (explícita o implícitamente)
- Especificar el estado inicial
- Especificar el estado final o las condiciones que cumple
- Especificar los operadores de cambio de estado (condiciones de aplicabilidad y función de transformación)
- Especificar el tipo de solución:
  - La secuencia de operadores o el estado final
  - Una solución cualquiera, la mejor (definición de coste), *. . .*

Ejemplo: 8-puzzle



- Espacio de estados: Configuraciones de 8 fichas y un hueco en el tablero
- Estado inicial: Cualquier configuración
- Estado final: Fichas en orden específico
- Operadores: Mover hueco
  - Condiciones: El movimiento está dentro del tablero
  - Transformación: Intercambio entre el hueco y la ficha en la posición del movimiento
- Solución: qué pasos (secuencia) + el menor número

Ejemplo: N Reinas



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

Ejemplo: jarras de agua

- Tenemos dos jarras con una capacidad de 4 y 3 litros. Ninguna tiene marca alguna de medida. El problema consiste en obtener exactamente 2 litros de agua en la jarra de 4 litros. Podemos obtener toda el agua que sea necesaria de un depósito en el que también podemos verter agua si esto fuese necesario

Ejemplo: jarras de agua

- El espacio de estados: El conjunto de pares ordenados (x,y), tal que x = 0, 1, 2, 3, 4 e y = 0, 1, 2, 3, donde x e y representan respectivamente el volumen de agua contenido en la jarra de 4 y de 3 li
- Estado inicial: (0,0)
- Estado final: (2, n), para cualquier valor posible de n

Ejemplo: jarras de agua

- **Operadores:**
  - (x,y), si $x < 4 \rightarrow (4,y)$; Llena la jarra de 4 litros
  - (x,y), si $y < 3 \rightarrow (x,3)$; Llena la jarra de 3 litros
  - (x,y), si $x > 0 \rightarrow (0,y)$; Vacía la jarra de 4 litros
  - (x,y), si $y > 0 \rightarrow (x,0)$; Vacía la jarra de 3 litros
  - (x,y), si $(x+y) \geq 4$ e $y > 0 \rightarrow (4,y-(4-x))$; Llena la jarra de 4 litros con el contenido de la jarra de 3 litros
  - (x,y), si $(x+y) \geq 3$ e $y > 0 \rightarrow (x-(3-y),3)$; Llena la jarra de 3 litros con el contenido de la jarra de 4 litros
  - (x,y), si $(x+y) \leq 4$ e $y > 0 \rightarrow (x+y,0)$; Vacía el contenido de la jarra de 3 litros en la de 4 litros
  - (x,y), si $(x+y) \leq 3$ e $x > 0 \rightarrow (0,x+y)$; Vacía el contenido de la jarra de 4 litros en la de 3 litros

Jarras de agua: una solución

	x	y	Operador
Estado inicial	0	0	Llena la de 3
	0	3	Vacía la de 3 en la de 4
	3	0	Llena la de 3
	3	3	Llena la de 4 con parte la de 3
	4	2	Vacía la de 4
	0	2	Vacía la de 3 en la de 4
	2	0	Estado final

¿Cómo escoger estados y acciones?

- El verdadero arte de la resolución de problemas consiste en decidir que es lo que servirá para representar los estados, y que no
- Hay que realizar un proceso de eliminación de los detalles que sean innecesarios en la representación de estados y acciones
- Hay que realizar una abstracción

Búsqueda en el espacio de estados

- La resolución de un problema con esta representación pasa por explorar el espacio de estados
- Partimos del estado inicial evaluando cada paso hasta encontrar un estado final
- En el caso peor exploraremos todos los posibles caminos entre el estado inicial del problema hasta llegar al estado final

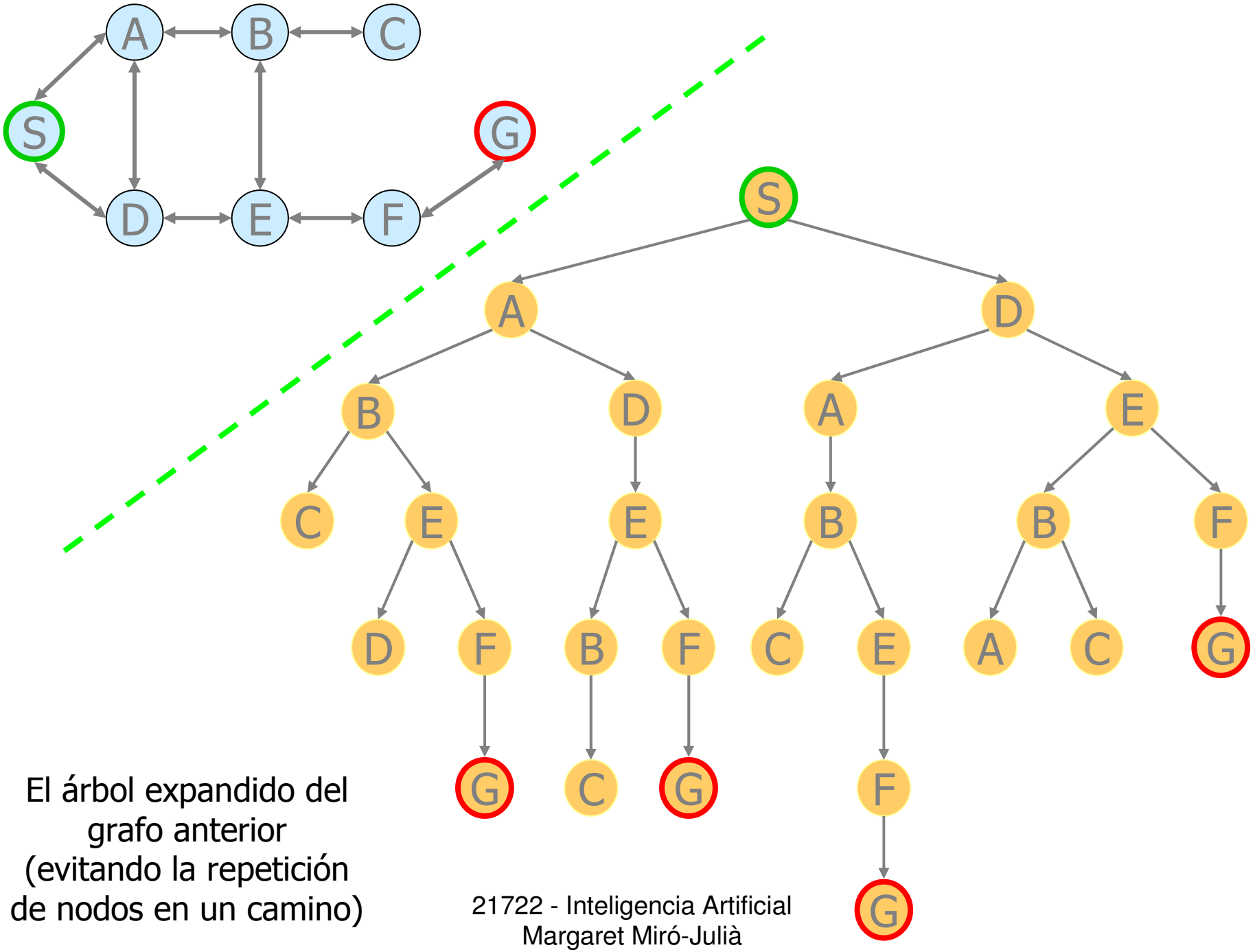
Estructura del espacio de estados

- Primero definiremos una representación del espacio de estados para poder implementar algoritmos que busquen soluciones
- Estructuras de datos: Árboles y Grafos
- Estados = Nodos
- Operadores = Arcos entre nodos (dirigidos)
- Árboles: Solo un camino lleva a un nodo
- Grafos: Varios caminos pueden llevar a un nodo

La diferencia entre nodos y estados

Es importante destacar la diferencia entre nodos y estados

- Un nodo es simplemente una estructura de datos auxiliar que resulta adecuada para representar el proceso de búsqueda en un determinado problema y en un instante dado, de acuerdo con un cierto procedimiento de búsqueda
- Un estado representa por contra una cierta configuración del mundo o entorno donde se desarrolla el problema
- Los nodos tienen antecesores (nodos padre), profundidad, …, mientras los estados no



El árbol expandido del grafo anterior (evitando la repetición de nodos en un camino)

Estructura de datos para la exploración en árboles

Un nodo se definirá como una estructura de datos con las siguientes componentes:

- El estado en el espacio de estados al que este nodo representa
- El nodo que lo generó o nodo padre
- El operador que se aplicó para generar este nodo
- El número de nodos desde este nodo al nodo raíz (la profundidad del nodo)
- El coste acumulado en el trayecto desde el nodo raíz a este nodo

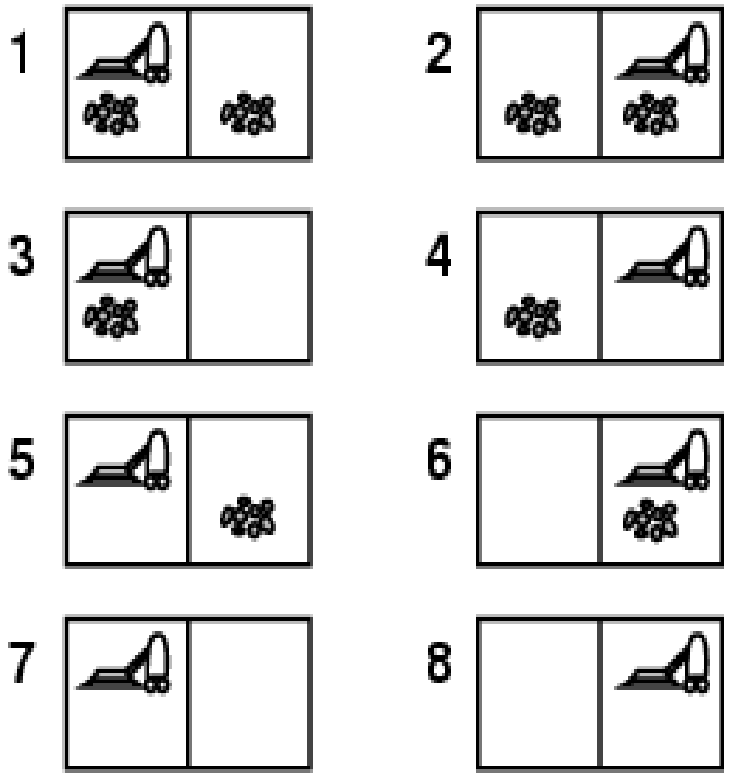
Enfoque de agente para la solución de problemas

- Agente formula una *META*
- Agente tiene una *META* y debe esforzarse para alcanzarla
- Formulará el problema dependiendo de:
  - ¿conoce su estado actual?
  - ¿conoce el resultado de sus acciones?

  (¿qué relación tiene el agente con su ambiente a través de sus percepciones y acciones?)

El mundo de la aspiradora

En el mundo de la aspiradora hay dos ubicaciones, en las que puede haber suciedad o no, encontrándose el agente en una de ellas.

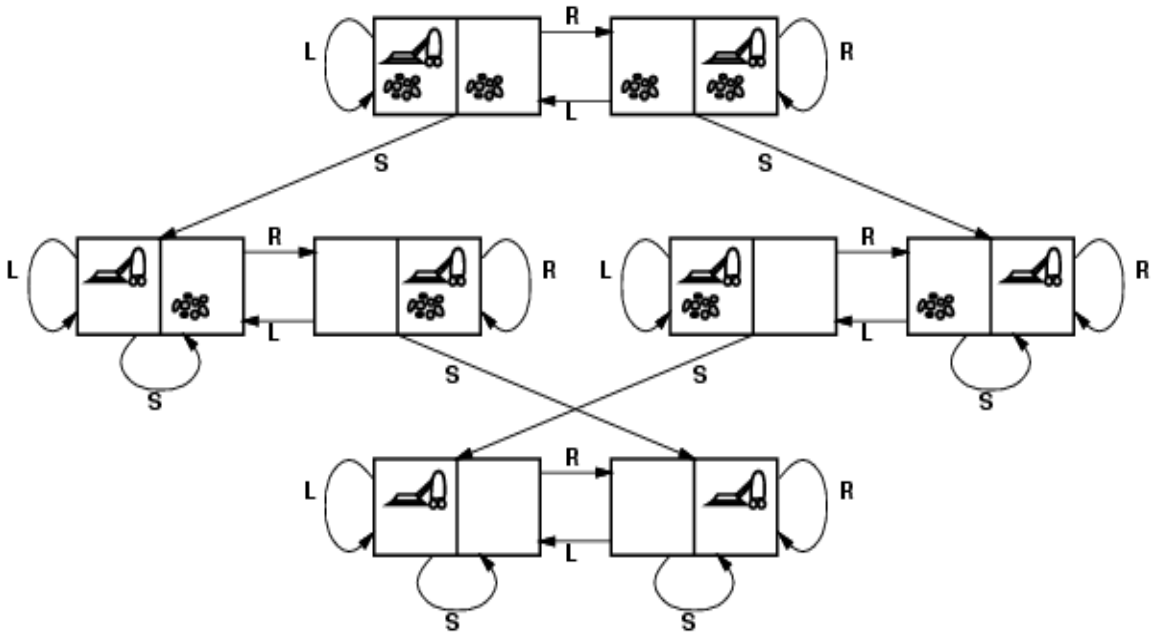


El mundo de la aspiradora

- Formulación de la meta:
  - Ambas habitaciones limpias
- Formulación del problema:
  - estados: 8 posibilidades
    - estado inicial: cualquiera (bien definida),
    - estado final: todo limpio
  - acciones (operadores):
    - ir a la izquierda
    - ir a la derecha
    - aspirar
- Encontrar solución:
  - Llegar a los estados 7 o 8
- Costo de la ruta
  - Una unidad por acción

Propiedades: ubicaciones discretas, suciedad discreta (binario), determinístico

El mundo de la aspiradora



- estados? suciedad, ubicación
- acciones? *izda (L)*, *dcha (R)*, *aspirar (S)*
- meta? ambas ubicaciones no sucias

- Si el ambiente es accesible, el agente sabe el estado en que se encuentra
- Si el ambiente es determinista conoce el resultado producido por cada acción
- El agente puede calcular el estado en que se encuentra después de una serie de acciones
- Si el estado actual es 5, puede calcular que la secuencia de acciones dcha (R), aspirar (S) le lleva a la meta
- Ambientes accesibles y deterministas: en cada paso el agente razona en términos de un estado

PROBLEMAS DE UN SOLO ESTADO

- Si el ambiente no es accesible, el agente no sabe el estado en que se encuentra (caso extremo, es uno de los 8 estados posibles)
- Si el ambiente es determinista conoce el resultado producido por cada acción
- El agente puede calcular el estado en que se encuentra después de una serie de acciones
- La secuencia de acciones dcha (R), aspirar (S), izda (L), aspirar (S) le lleva siempre a la meta
- Ambientes no accesibles y deterministas: en cada paso el agente razona en términos de un conjunto de estados

PROBLEMAS DE ESTADOS MÚLTIPLES

- Si el ambiente no es accesible, el agente no sabe el estado en que se encuentra
- Si el ambiente no es determinista el agente no conoce el resultado producido por cada acción (queda suciedad después de aspirar)
- El agente debe usar sensores después de cada acción
- Ninguna acción fija garantiza una buena solución
- A menudo hay que entremezclar búsqueda y ejecución
- Ambientes no accesibles y no deterministas: es necesario construir el árbol, cada rama es una posible contingencia

PROBLEMAS DE CONTINGENCIA

- Si el espacio de estados es desconocido
  - El agente no conoce los efectos de sus acciones
  - No conoce las acciones

PROBLEMAS DE EXPLORACIÓN

Definición del problema

ESPACIO DE ESTADOS

- Estado inicial del problema
- Acciones posibles (operador o regla)

Se agrega a la definición del problema

- Prueba de meta: ¿Es el estado actual un estado META?
- COSTO DE RUTA (g): Suma de costos de las acciones individuales que componen la ruta

Medición de la eficiencia de la estrategia

- ¿Garantiza encontrar una solución?
- ¿Es una buena solución? (de acuerdo a un criterio)
- ¿Cuál es el costo de búsqueda en tiempo y memoria para encontrar una solución?

Algoritmo básico

- El espacio de estados puede ser infinito
- Es necesaria una aproximación diferente par buscar y recorrer árboles y grafos (no podemos tener la estructura en memoria)
- La estructura la construimos a medida que hacemos la búsqueda

Idea general

Seleccionar el primer estado como el estado actual,

mientras *estado actual ≠ estado final* **hacer**

Generar y guardar sucesores del estado actual (expansión)

Escoger el siguiente estado entre los pendientes (selección)

fin

- La selección del siguiente nodo determinará el tipo de búsqueda (orden de selección o expansión)
- Es necesario definir un orden entre los sucesores de un nodo (orden de generación)

Algoritmo general de búsqueda

```
función BUSQUEDA-GENERAL(problema) retorna nodo o fallo

inicio
   INSERTA-LISTA(lista, HACER-NODO(ESTADO-INICIAL[problema])
   bucle hacer
      si VACIA(lista) entonces retorna fallo
      sino
         nodo ← ELIMINA-ELEMENTO(lista),
         si PRUEBA-META[problema, estado(nodo)] es éxito
         entonces retorna nodo
         sino
            INSERTA-LISTA(lista, EXPANDIR(nodo,
                     OPERADORES[problema]))
         fin si
      fin si
   fin bucle hacer
fin
```

Árbol de búsqueda

- Generada mientras atravesamos el espacio de búsqueda
  - El árbol especifica posibles rutas a través del espacio de búsqueda
- Expansión de nodos
  - Conforme exploramos (visitamos) los estados, los nodos correspondientes son expandidos (generados) por los operadores
    - Esto genera un nuevo conjunto de nodos (hijos)
  - La lista está formada por el conjunto de nodos no visitados (o no elegidos)
    - Los nodos generados se añaden a la lista
- Estrategia de búsqueda
  - Determina cual es el próximo nodo que se expande
  - Depende del orden de nodos en la lista
    - cola (FIFO),
    - pila (LIFO),
    - “mejor” nodo con respeto a alguna medida (costo)

Estrategias de búsqueda

Una estrategia de búsqueda queda definida al escoger el orden en que se expanden los nodos

- Búsqueda no informada o a ciegas

  únicamente se utiliza la información disponible en la definición del problema (no se tiene en cuenta el costo)

- Búsqueda informada o heurística

  el agente tiene información adicional del problema (estimación del costo hasta la meta)

Tipos de algoritmos

- Algoritmos de búsqueda ciega
  - No tienen en cuenta el coste de la solución en la búsqueda
  - Su funcionamiento es sistemático, siguen un orden de visitas y generación de nodos establecido por la estructura del espacio de búsqueda
  - Primero en anchura, primero en profundidad, profundidad iterativa, …
- Algoritmos de búsqueda heurística
  - Utilizan una estimación del coste de la solución para guiar la búsqueda
  - No siempre garantizan el óptimo, ni una solución
  - Hill-climbing, Branch and Bound, A*, IDA, …

Primero en anchura

- Se construye el espacio de estados mediante la aplicación de los operadores disponibles al nodo inicial
- Después se aplica los operadores disponibles a los nodos sucesores directos del estado inicial, y así sucesivamente (de izda a dcha)
- Los nodos que se encuentran a profundidad d se expanden antes que los nodos de profundidad d+1
- Este tipo de búsqueda se implementa con una estructura de datos *cola (FIFO)*

Algoritmo: primero en anchura

- Crear una lista de nodos, LISTA, y asignarle el estado inicial
- Hasta que LISTA esté vacía o se encuentre una meta, hacer:
  - Eliminar el primer nodo de LISTA, llamarlo Actual
  - Si es meta, terminar y devolver como éxito
  - Para cada operador aplicable,
    - Aplicar el operador a Actual y generar un nuevo estado
    - Incluir el nuevo estado al final de LISTA

Primero en profundidad

- Se construye el espacio de estados mediante la aplicación de los operadores disponibles al nodo inicial
- Después se aplica los operadores disponibles a los nodos sucesores directos del estado inicial, y así sucesivamente
- Los nodos que se encuentran a mayor profundidad se expanden antes que los nodos de profundidad menor
- Este tipo de búsqueda se implementa con una estructura de datos pila (LIFO)
- La búsqueda en una rama termina cuando se llega a un callejón sin salida, si esto ocurre se produce un backtracking, revisitándose el estado alcanzado más recientemente desde el que sea posible la aplicación de otro operador

Algoritmo: primero en profundidad

- Crear una lista de nodos, LISTA, y asignarle el estado inicial
- Hasta que se encuentre una meta o se devuelva fallo, hacer:
  - Si LISTA está vacía, terminar con fallo
  - Extraer el primer nodo de LISTA y llamarlo Actual
  - Generar un sucesor de Actual, si no existen más sucesores terminar con fallo y volver al nodo padre
  - Expandir creando punteros para que pueda saberse cuál es el predecesor. Añadir los sucesores a LISTA.
    - Si algún sucesor es meta terminar
    - Si algún sucesor se encuentra en un callejón sin salida eliminarlo de LISTA.

- Primero en anchura
- Primero en profundidad

Evaluación de las Estrategias

Las estrategias se evalúan de acuerdo a su:

- Completitud: ¿La estrategia garantiza encontrar una solución, si ésta existe?
- Complejidad temporal: ¿Cuánto tiempo se necesitará para encontrar una solución? (proporcional a los nodos generados)
- Complejidad espacial: ¿Cuánta memoria se necesita para efectuar la búsqueda? (proporcional a los nodos almacenados)
- Optimalidad: ¿Con esta estrategia se encontrará una solución de la más alta calidad, si hay varias soluciones?

Las complejidades temporal y espacial se miden en términos de:

r → máximo factor de ramificación del árbol de búsqueda

p → profundidad de la solución de menor coste

m → profundidad máxima del espacio de estados (puede ser ∞)

Búsqueda primero en anchura

- La estrategia:
  - Los nodos se visitan y generan por niveles
  - La estructura para los nodos generados es una cola (FIFO), los últimos nodos expandidos se insertan en el final de la lista
  - Un nodo es visitado cuando todos los nodos de los niveles superiores y sus hermanos precedentes han sido visitados
- Características:
  - Completitud: El algoritmo garantiza el encontrar una solución si existe; y si existen varias devuelve la de menor profundidad y más a la izquierda (si *r* es finito)

Búsqueda primero en anchura

- Características:
  - Complejidad: Supongamos un árbol cuya solución se encuentra a profundidad p y cuyo factor de ramificación es r. Entonces el nº de nodos expandidos antes de alcanzar la solución será:

$$1+r+r^2+r^3+ \ldots +r^{p+1}$$

Dado que es necesario expandir cada uno de los nodos (en el peor de los casos) y, además, conservarlos en memoria, los requerimientos en tiempo y en espacio (memoria) son ambos $O(r^{p+1})$

Búsqueda primero en anchura

- Características:
  - Complejidad temporal: Exponencial respecto al factor de ramificación y la profundidad de la solución $O(r^{p+1})$
  - Complejidad espacial: Exponencial respecto al factor de ramificación y la profundidad de la solución $O(r^{p+1})$
  - Optimalidad: Si hay varias soluciones, encuentra la más superficial, la solución es óptima si el costo es proporcional a la profundidad (coste = 1 por paso)

Espacio es el problema (más que tiempo)

Búsqueda primero en profundidad

- La estrategia
  - Los nodos se visitan y generan buscando los nodos a mayor profundidad y retrocediendo cuando no se encuentran nodos sucesores
  - La estructura para los nodos generados es una pila (LIFO), los últimos nodos expandidos se insertan en el inicio de la lista
- Características
  - Completitud:
    - falla con profundidades infinitas y espacios con “loops” (hay que evitar estados repetidos en una misma ruta)

    El algoritmo encuentra una solución en espacios finitos o si se impone un límite de profundidad y existe una solución dentro de ese límite

Búsqueda primero en profundidad

- Características
  - Complejidad temporal: Exponencial respecto al factor de ramificación y la profundidad del límite de exploración $O(r^p)$, terrible si $p > r$

    ($O(r^m)$ si hay límite máximo de profundidad)
  - Complejidad espacial: Los requerimientos en memoria son muy modestos, sólo es necesario guardar una única trayectoria desde el nodo raíz. La complejidad es lineal respecto al factor de ramificación y el límite de profundidad $O(r*p)$. Si consideramos una profundidad máxima m, sólo es necesario almacenar 1+r*m nodos
  - Optimalidad: El método de exploración no es óptimo, no garantiza la solución de menor coste

No es aconsejable la búsqueda primero en profundidad si p es grande

- La búsqueda primero en profundidad suele funcionar mejor que la búsqueda en amplitud en problemas con muchas soluciones, siempre que no se requiera la optimalidad de la solución, pues encuentra la solución tras explorar sólo una pequeña porción del árbol
- Debe evitarse en árboles infinitos o de gran profundidad

Coste Uniforme

- Los nodos con el coste mínimo son explorados primeros
  - similar PRIMERO-EN-ANCHURA (izquierda a derecha), pero con una evaluación del costo para cada nodo
  - *g(n)* = costedelaruta(n) = suma de costos individuales para alcanzar el nodo actual

NOTA: los nodos de igual coste se ordenan de izquierda a derecha

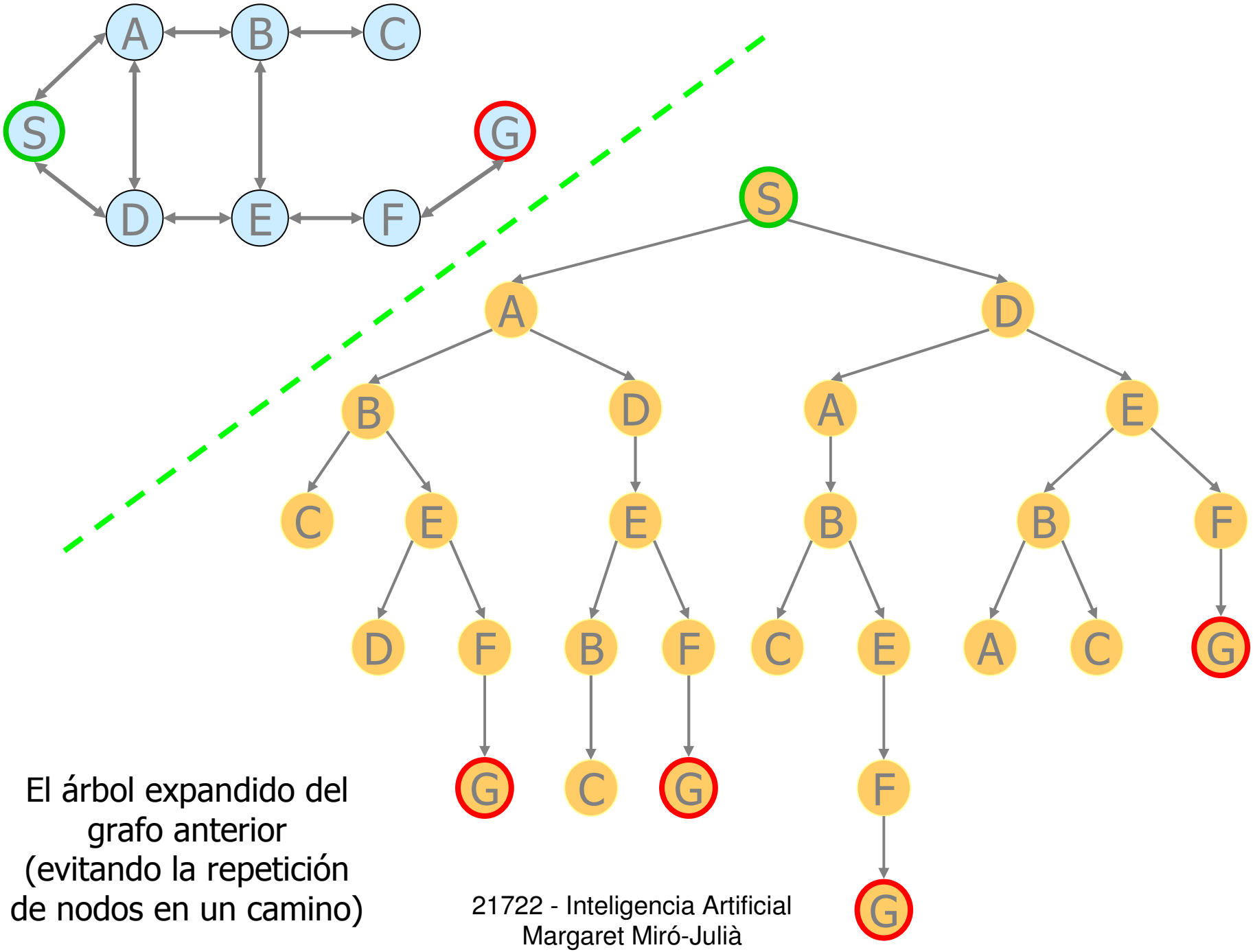
Búsqueda coste uniforme

- La estrategia
  - Los nodos con el coste mínimo son explorados primeros
- Características
  - Completitud: Si (en espacios finitos)

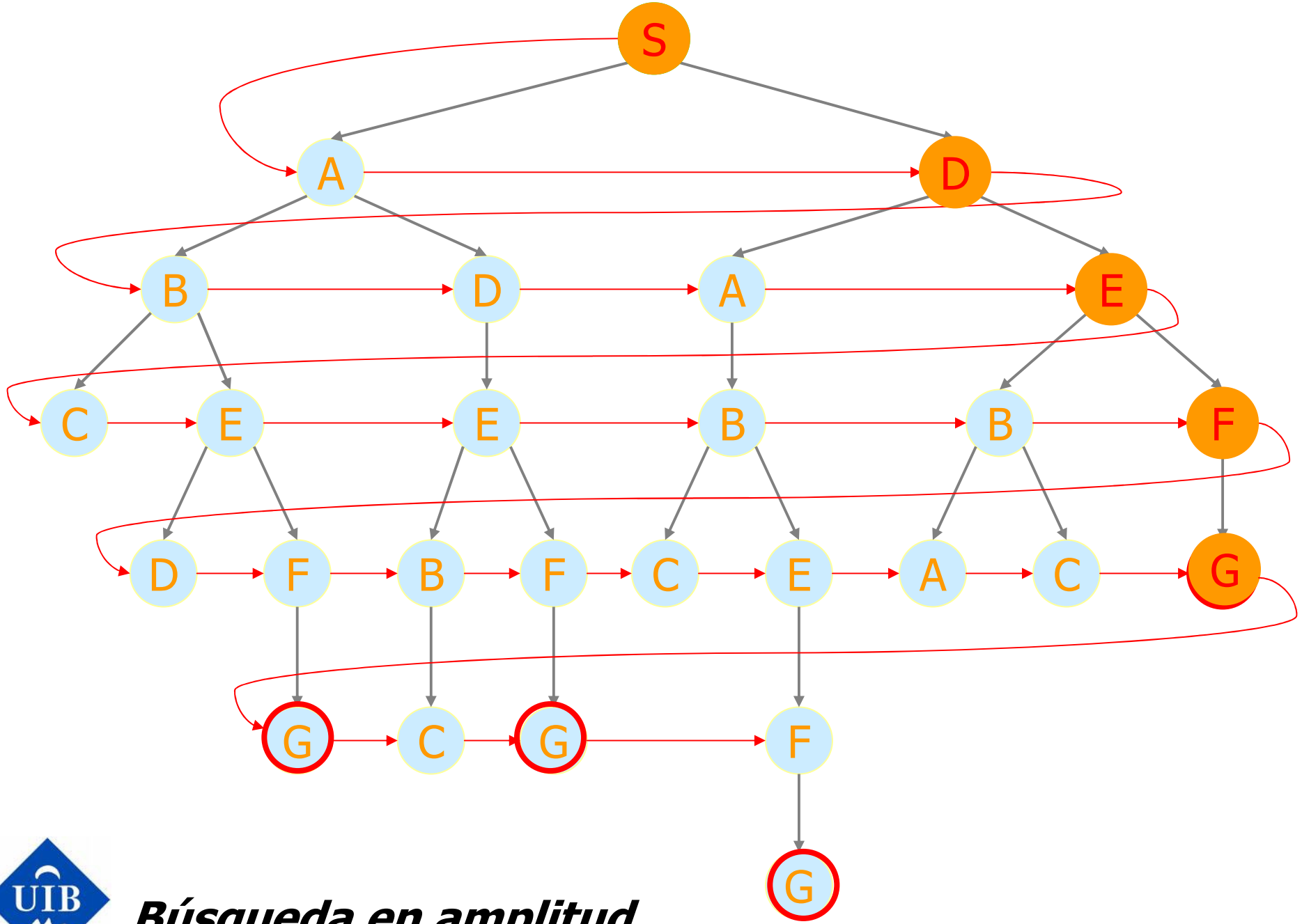
    Garantiza obtener la ***solución más barata*** si el costo de ruta nunca disminuye al avanzar g(sucesor(n)) >= g(n)
  - Complejidad temporal: $O(r^p)$
  - Complejidad espacial: $O(r^p)$
  - Optimalidad: Si

Comparación de las estrategias de búsqueda

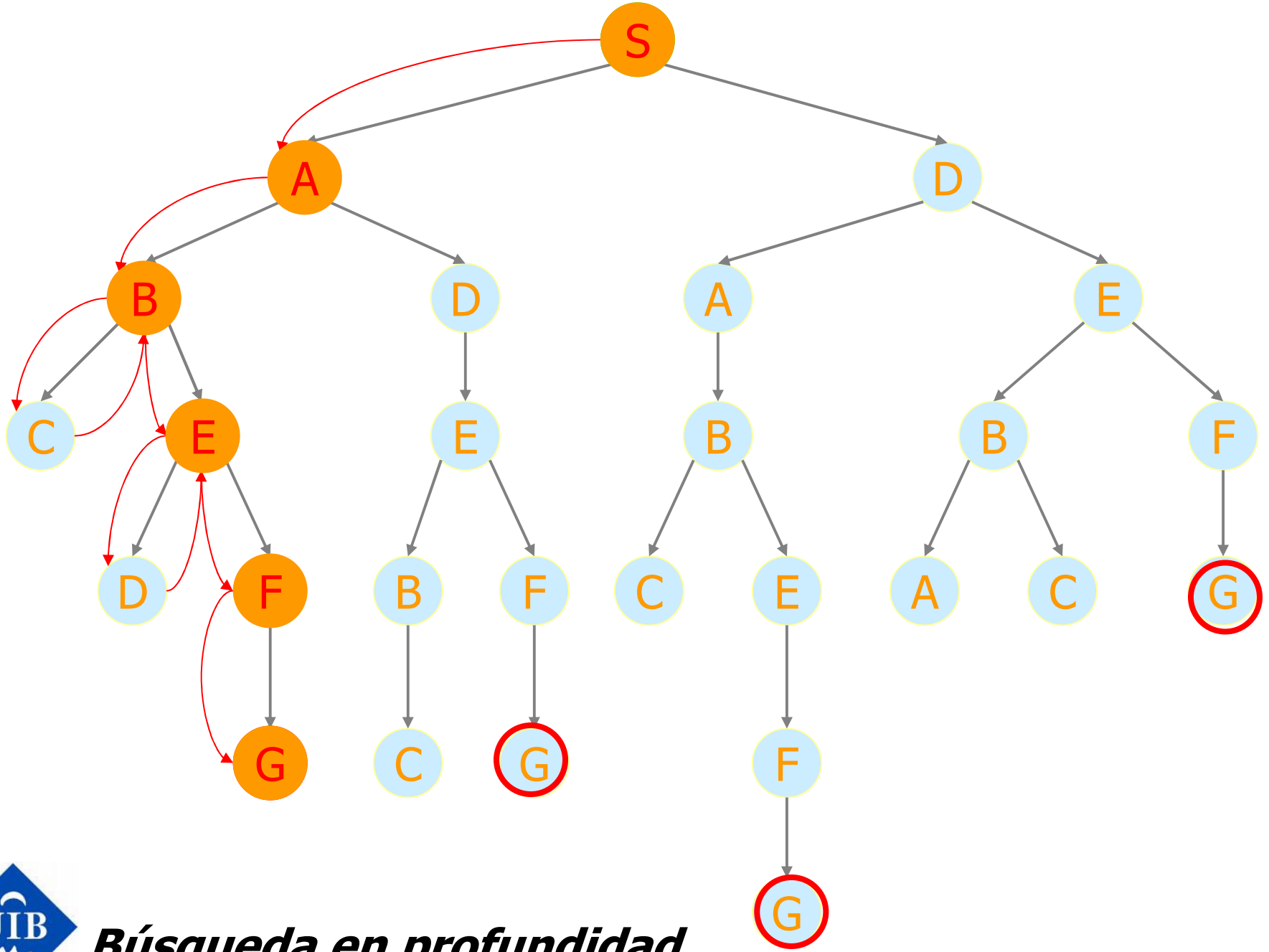
Criterio	Tiempo	Espacio	Óptima?	Completa?
Primero en amplitud	$O(r^p)$	$O(r^p)$	SI	SI
Costo Uniforme	$O(r^p)$	$O(r^p)$	SI	SI
Primero en profundidad	$O(r^m)$	$O(r*m)$	NO	NO



El árbol expandido del grafo anterior (evitando la repetición de nodos en un camino)



Búsqueda en amplitud



Búsqueda en profundidad

Búsqueda en profundidad limitada

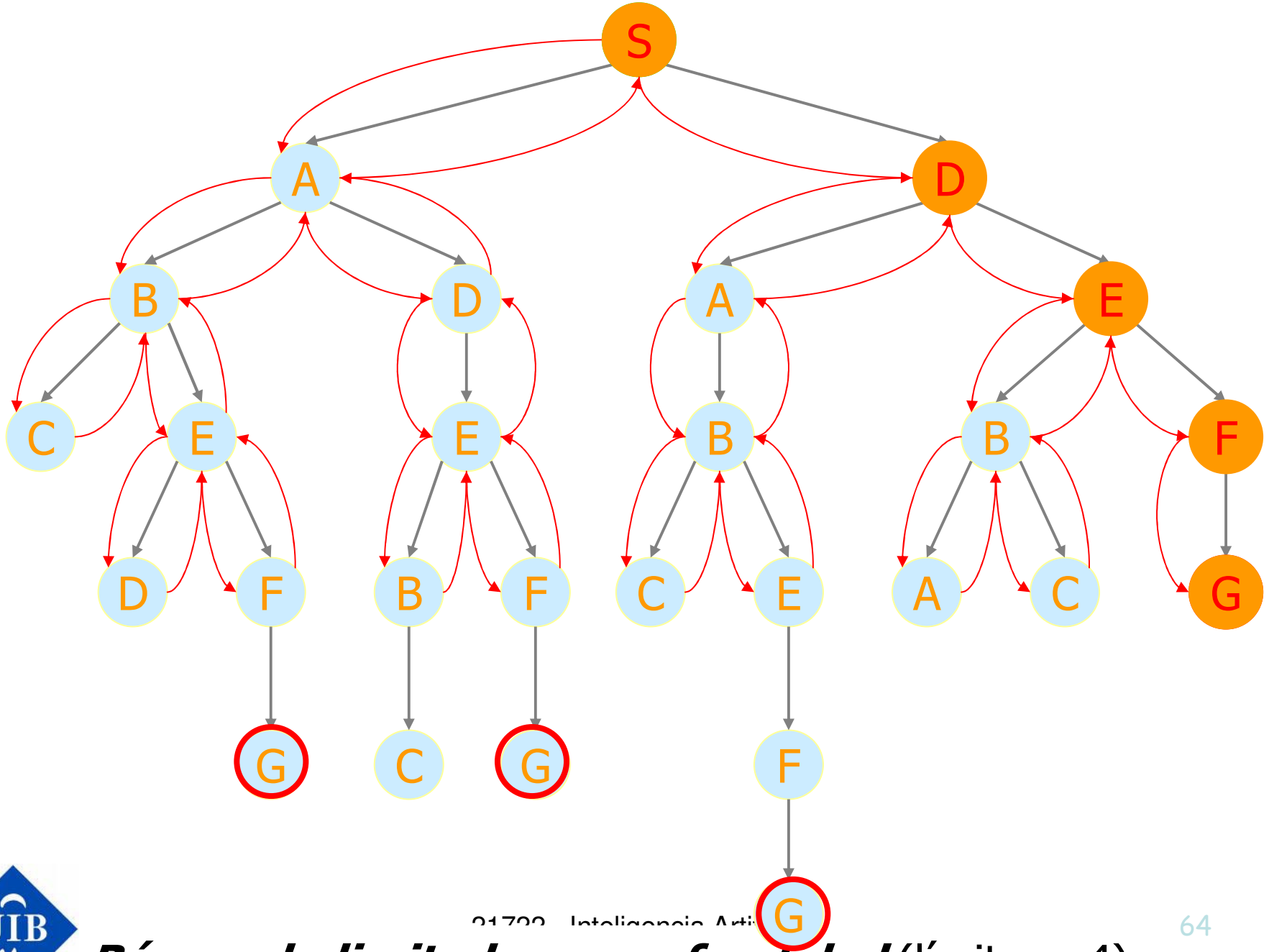
- Se impone un límite máximo a la profundidad de la ruta
- Se elimina la posibilidad de atascamientos de la búsqueda primero en profundidad
- Se utiliza el algoritmo de BÚSQUEDA GENERAL colocando los NODOS generados al expandir, al comienzo de la LISTA
- Se incluyen operadores que garanticen la vuelta atrás cuando se ha alcanzado el límite

Búsqueda en profundidad limitada

Procedimiento: Busqueda en profundidad limitada (limite: entero)
Est_abiertos.insertar(Estado inicial)
Actual Est_abiertos.primero()
mientras no *es_final?(Actual)* **y no** *Est_abiertos.vacia?()* **hacer**
    Est_abiertos.borrar_primero()
    Est_cerrados.insertar(Actual)
    si *profundidad(Actual) limite* **entonces**
        Hijos generar_sucesores (Actual)
        Hijos tratar_repetidos (Hijos, Est_cerrados, Est_abiertos)
        Est_abiertos.insertar(Hijos)
    fin
    Actual Est_abiertos.primero()
fin

- La estructura de lista es ahora una pila
- Se dejan de generar sucesores cuando se llega al límite de profundidad
- Esta modificación garantiza que el algoritmo acaba
- Si tratamos repetidos el ahorro en espacio es nulo

Est_abiertos = lista ordenada de estados generados
Est_cerrados = estados visitados



Búsqueda limitada en profundidad (límite = 4)

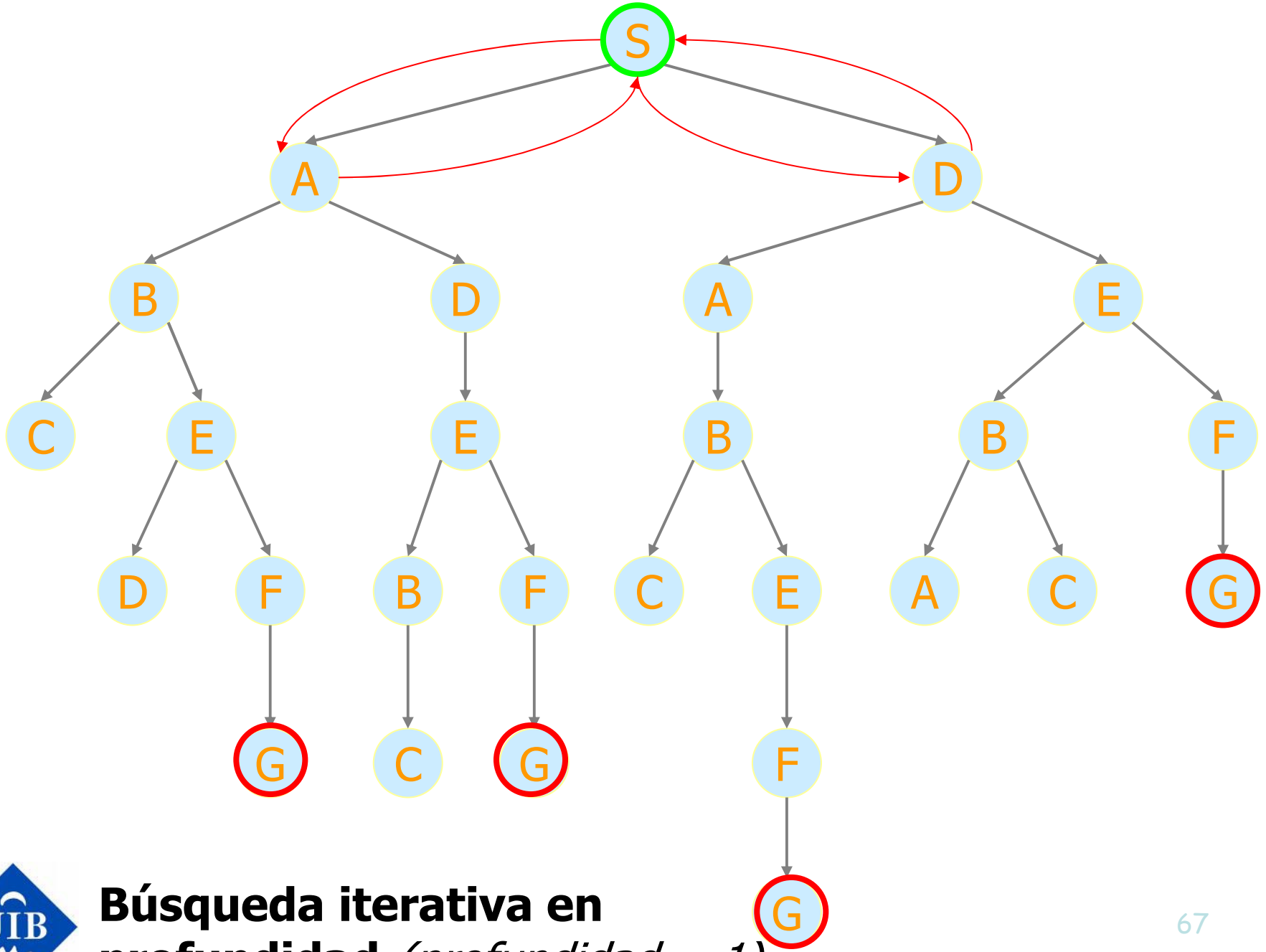
Búsqueda en profundidad limitada

- Características
  - Completitud: Si (debe elegirse el límite l adecuado para el problema)
  - Complejidad temporal: O(r^l)
  - Complejidad espacial: O($r*l$)
  - Optimalidad: No garantiza encontrar la mejor solución

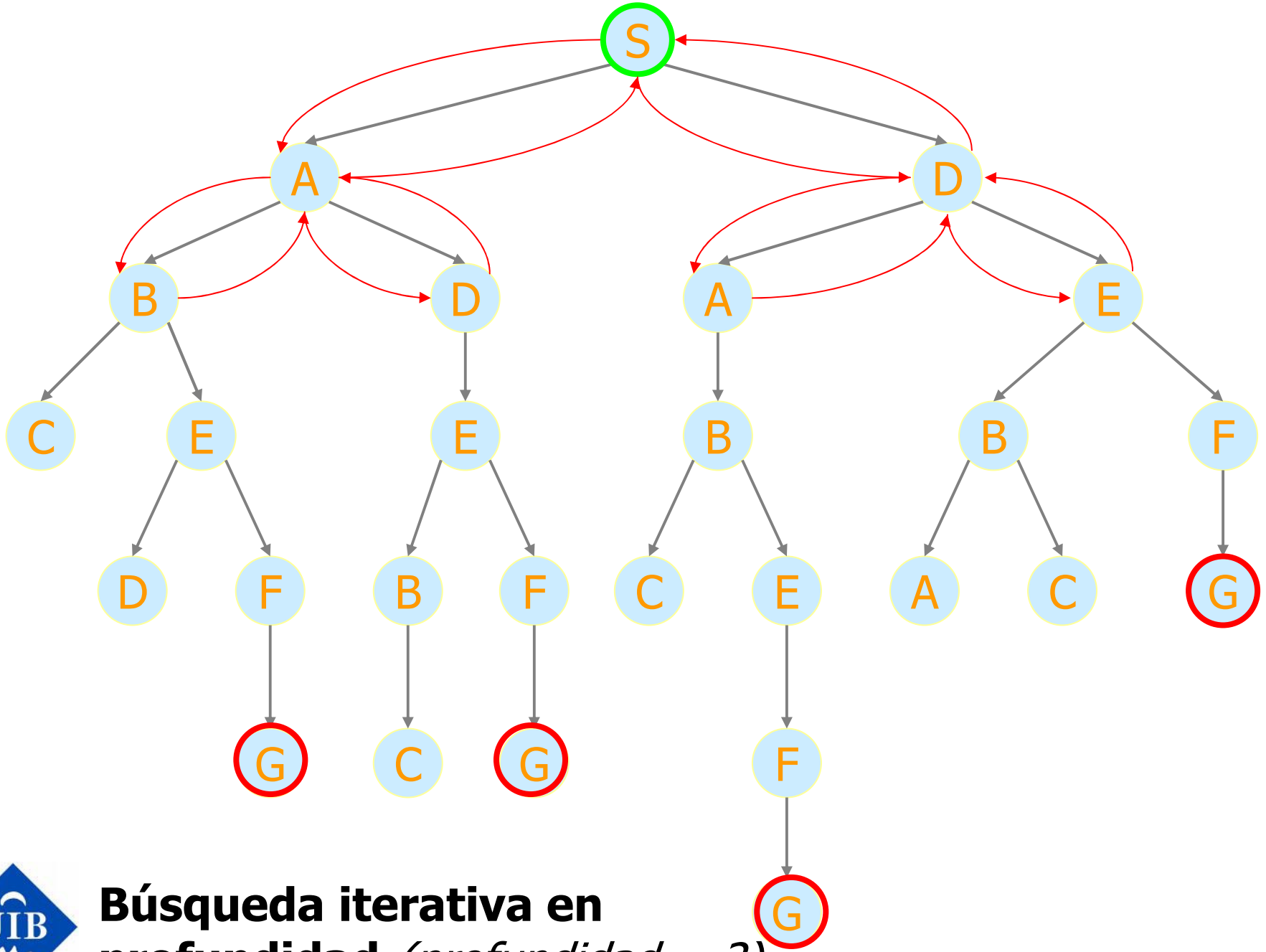
Si el límite es demasiado pequeño no puede garantizarse completitud. Es aconsejable en problemas donde se tenga idea de un límite razonable

Profundidad iterativa

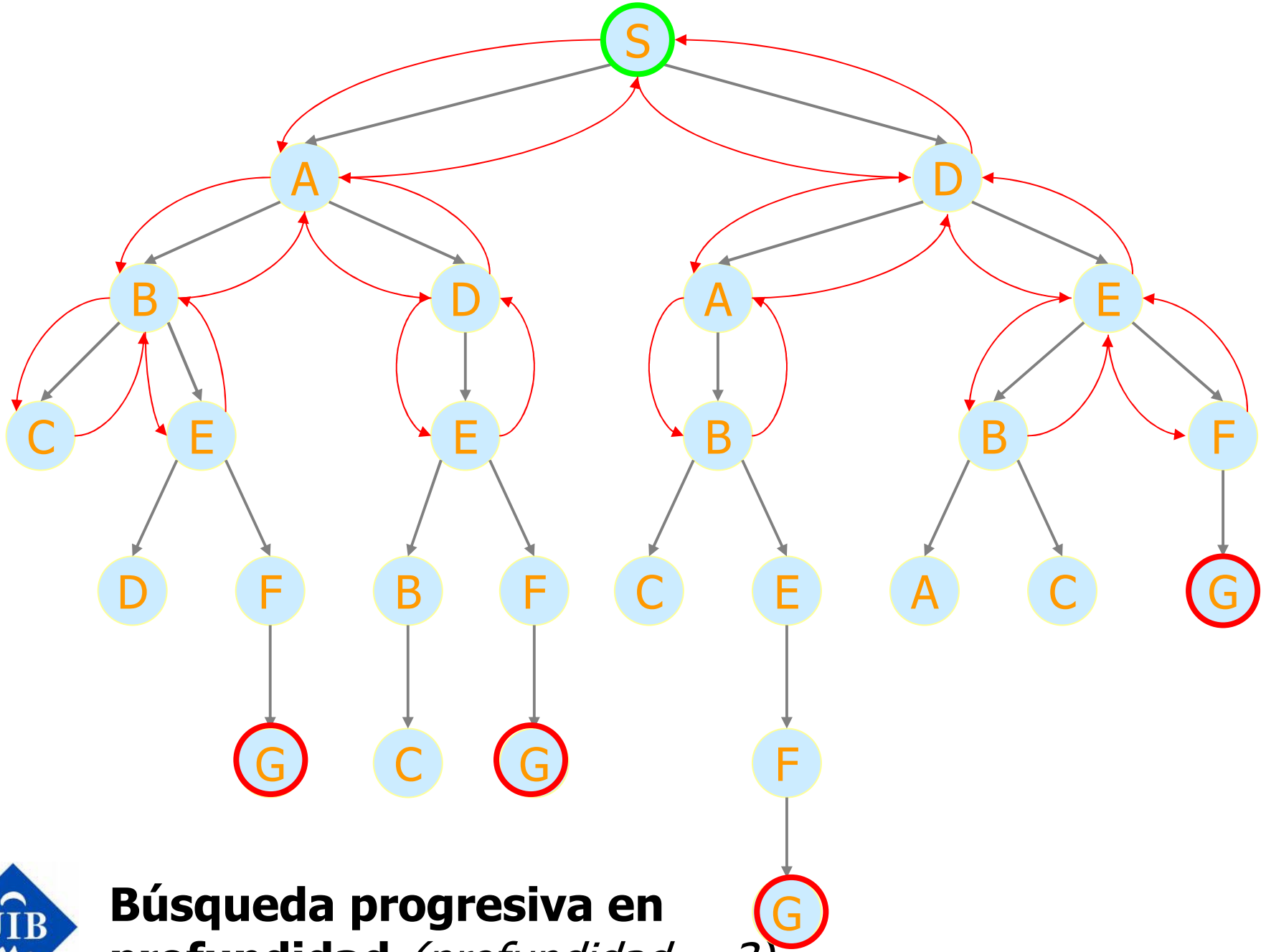
- Intenta combinar el comportamiento espacial de la búsqueda en profundidad con la optimalidad de la búsqueda en amplitud
- El algoritmo consiste en realizar búsquedas en profundidad sucesivas con un nivel de profundidad máximo acotado y creciente en cada iteración
- Así se consigue el comportamiento de la búsqueda en amplitud pero sin su coste espacial, ya que la exploración es en profundidad, y además los nodos se regeneran a cada iteración
- Además esto permite evitar los casos en que la búsqueda en profundidad no acaba (existen ramas infinitas)
- En la primera iteración la profundidad máxima será 1 y este valor irá aumentando en sucesivas iteraciones hasta llegar a la solución
- Para garantizar que el algoritmo acaba si no hay solución, se puede definir una cota máxima de profundidad en la exploración



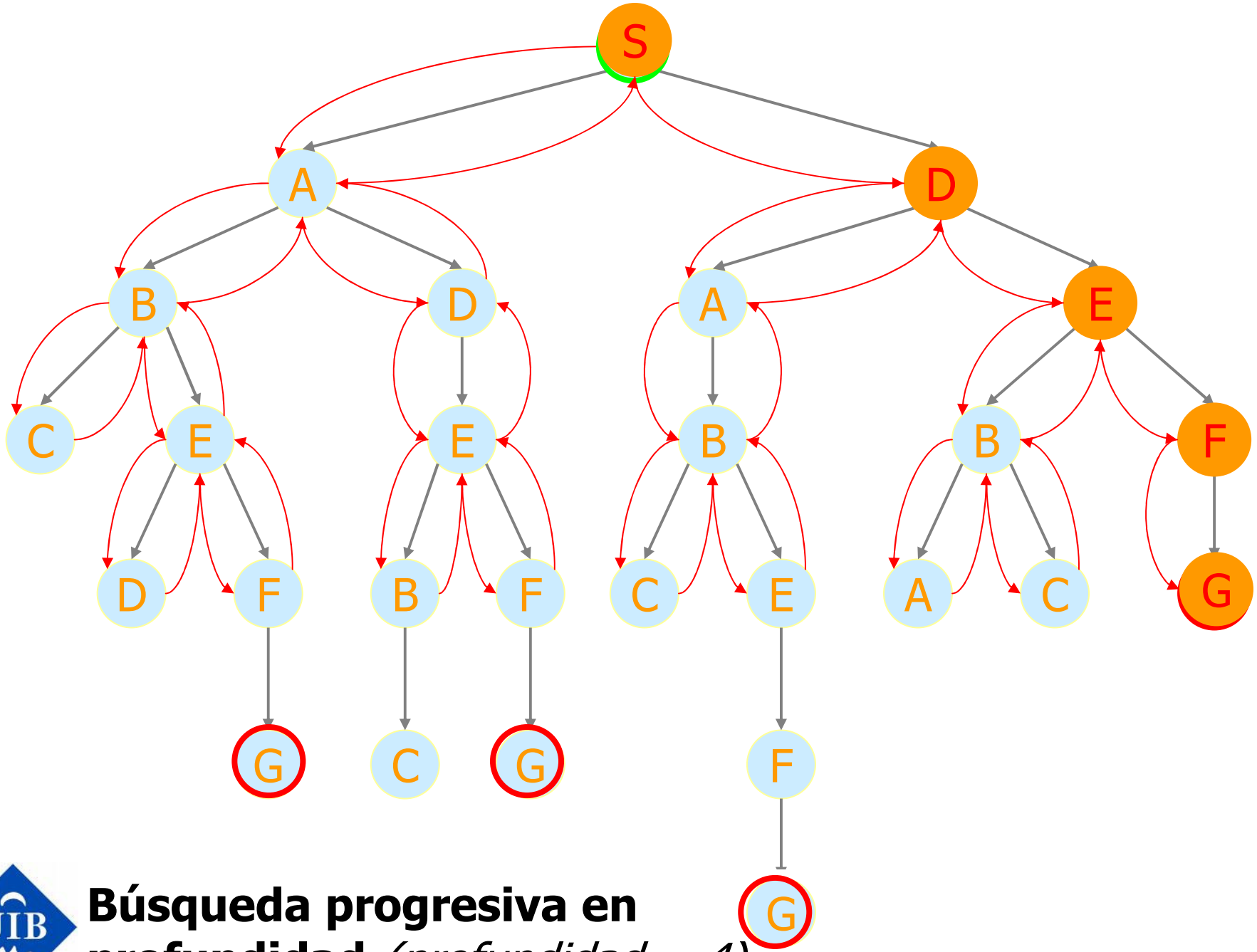
Búsqueda iterativa en profundidad *(profundidad = 1)*



Búsqueda iterativa en profundidad *(profundidad = 2)*



Búsqueda progresiva en profundidad *(profundidad = 3)*



Búsqueda progresiva en profundidad *(profundidad = 4)*

Búsqueda en profundad iterativa

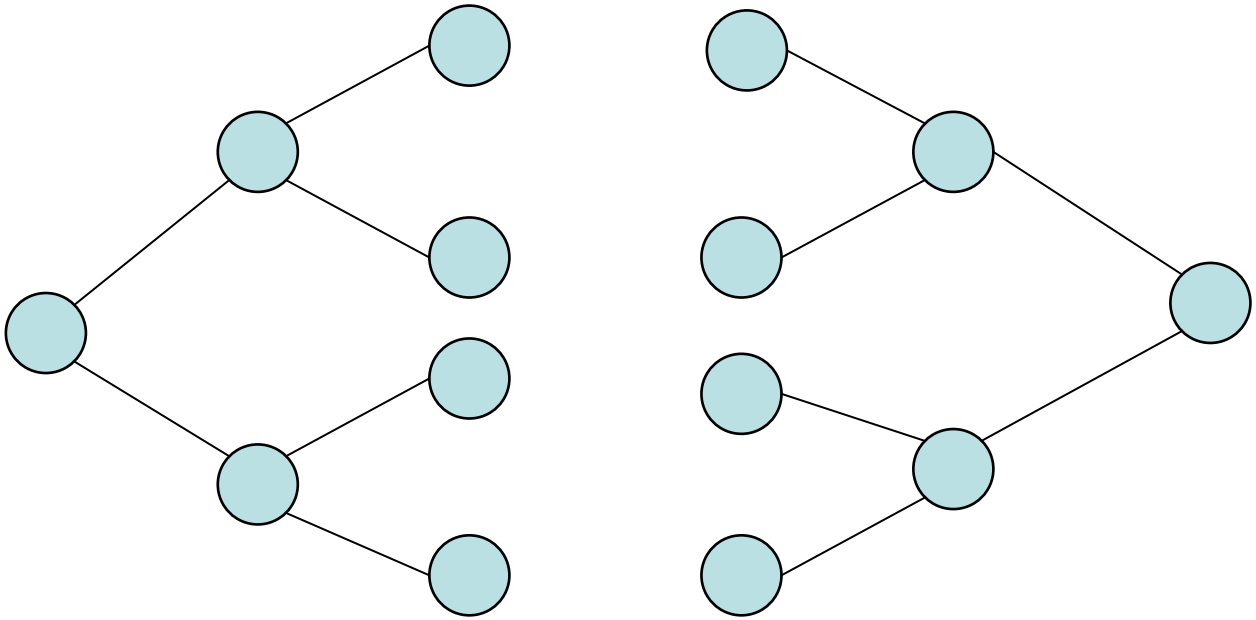
- Características
  - Completitud: Si
  - Complejidad temporal: O(r^p)
  - Complejidad espacial: O($r*p$)
  - Optimalidad: Si

  Es aconsejable en espacios grandes dónde se ignora la profundidad de la solución

Búsqueda bidireccional

- Es una búsqueda simultánea que avanza desde el estado inicial y retrocede desde la meta, y se detiene cuando se encuentran
- Como la solución estará a $O(r^{p/2})$ pasos (**p** profundidad de la solución), puede ser muy buena, pero hay problemas a resolver
  - Se debe conocer explícitamente cuáles son los estados meta
  - Se debe contar con operadores que permitan retroceder desde la meta (operadores reversibles)
  - Se debe tener una forma eficiente de verificar si los nodos nuevos están en la otra mitad de la búsqueda

Búsqueda bidireccional



Estado inicial Estado final

Búsqueda bidireccional

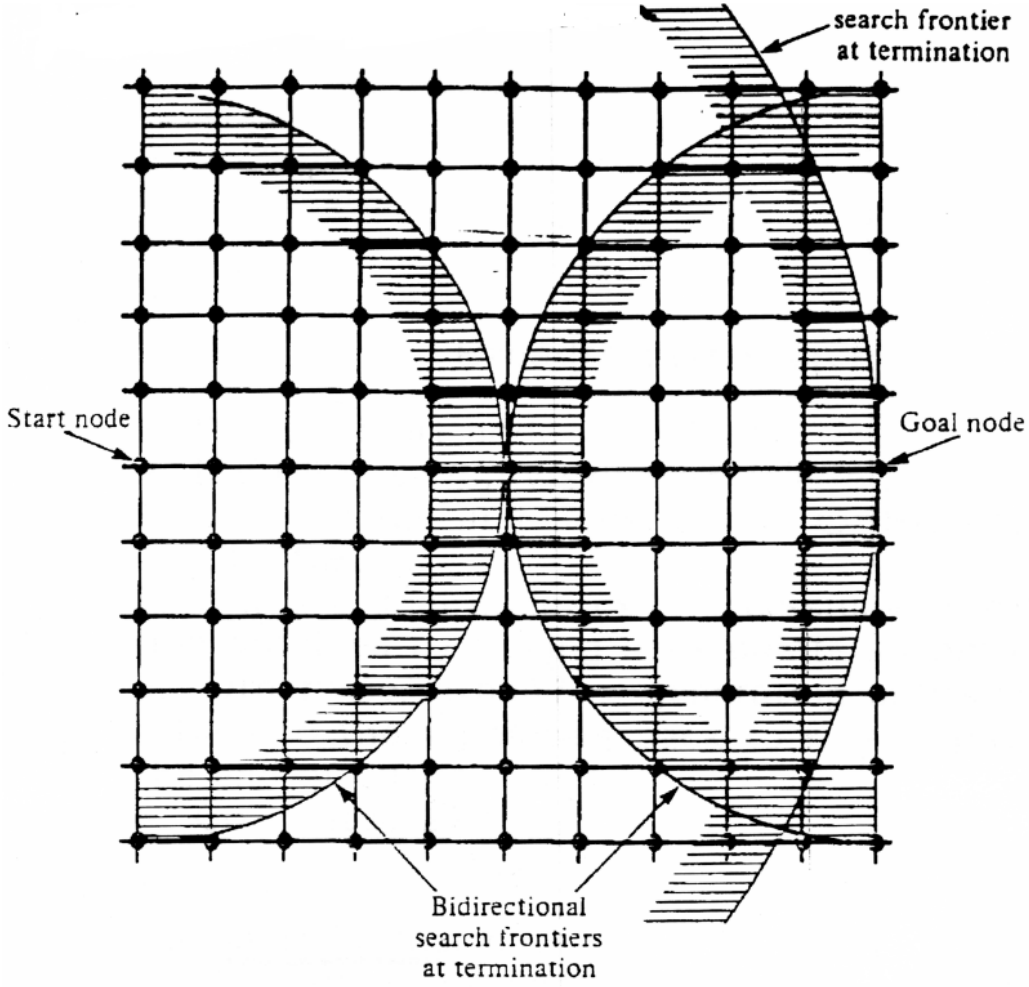
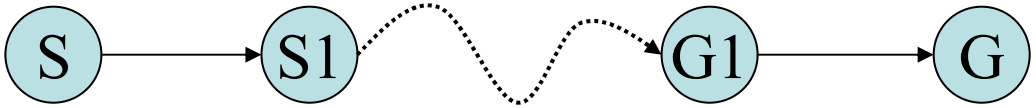


Fig. 2.10 Bidirectional and unidirectional breadth-first searches.

Búsqueda bidireccional

(S, S1) paso adelante
(G1, G) paso atrás

Espacio original:



- Usamos estrategias conocidas (búsqueda en anchura) en un nuevo espacio donde los nodos son pares del tipo S – E
- Los operadores (S - G, S1 - G1)
- Meta: (S – S) coinciden

- Numero total de nodos generados $O(2r^{p/2}) = O(r^{p/2})$
- Pero hay que comparar cada nodo con los de la frontera opuesta $O(r^{p/2})$
- La complejidad temporal total es $O(r^{p})$
- Para ser eficiente conviene usar tablas de hashing
- Caso standard
  - El tiempo para realizar el hashing y comparar es constante
  - La complejidad temporal es $O(r^{p/2})$

Comparación de las estrategias de búsqueda

Criterio	Tiempo	Espacio	Óptima?	Completa?
Primero en amplitud	$O(r^p)$	$O(r^p)$	SI	SI
Costo Uniforme	$O(r^p)$	$O(r^p)$	SI	SI
Primero en profundidad	$O(r^m)$	$O(r*m)$	NO	NO
Limitada en profundidad	$O(r^l)$	$O(r*l)$	NO	SI, cuando l >= p
Profundidad iterativa	$O(r^p)$	$O(r*p)$	SI	SI
Bidireccional	$O(r^{p/2})$	$O(r^{p/2})$	SI	SI

Búsqueda heurística

- La eficiencia en la búsqueda puede mejorar en gran medida si existe una forma de ordenar los nodos de tal manera que los nodos más prometedores se exploren primero
- Permite reducir la extensión de la búsqueda a ciegas incorporando conocimiento adicional
- Una “heurística” son criterios, métodos o principios que sirven para decidir cuál de las muchas alternativas puede ser la más efectiva para cumplir con un objetivo

Búsqueda heurística

- Usamos una “heurística” para decidir qué nodo expandir
  - la heurística es una función de evaluación basada en la información específica del dominio o contexto del problema
  - el problema de búsqueda informada se puede considerar como la maximización o minimización de una función
  - la función de evaluación proporciona una evaluación “local” del nodo basado en una estimación del costo para llegar desde el nodo al nodo meta
- Problemas con la heurística
  - la heurística suele ser poco certera (problema abierto)
  - da el valor de la actividad a un meta-nivel (problema abierto)
  - puede no encontrar la mejor respuesta

Búsqueda heurística

- Existencia de una función de evaluación que debe medir la distancia estimada a un estado meta: h(n)
- Esta función de evaluación se utiliza para guiar el proceso haciendo que en cada momento se seleccione el estado o las operaciones más prometedores
- No siempre se garantiza encontrar una solución (de existir ésta)
- No siempre se garantiza encontrar la solución “mejor” (la que se encuentra a una distancia, número de operaciones, … menor)
- Existen múltiples algoritmos:
  - Primero “el mejor”
  - A*
  - Búsqueda local (hill climbing, simulated annealing, algoritmos genéticos,…)

Búsqueda primero “el mejor”

- Es una mezcla de búsqueda en profundidad y, eventualmente, en amplitud debido a un cambio de rama, también llamada de “ramificación y salto” (branch-and-bound)
- Se guarda para cada estado el coste (hasta el momento) de llegar desde el estado inicial a dicho estado; y el coste mínimo global hasta el momento
- Se selecciona para expansión el nodo más prometedor de todos los generados hasta ese momento
- Si tomamos como nodo más prometedor aquel de menor coste acumulado en la ruta, se le denomina de “coste uniforme”

Búsqueda primero “el mejor”

función BUSQUEDA-PRIMERO-EL-MEJOR (*problema*, FUNCION-EVALUACION) **responde con** una secuencia de solución

entradas: *problema*, un problema

Función-Eval, una función de evaluación

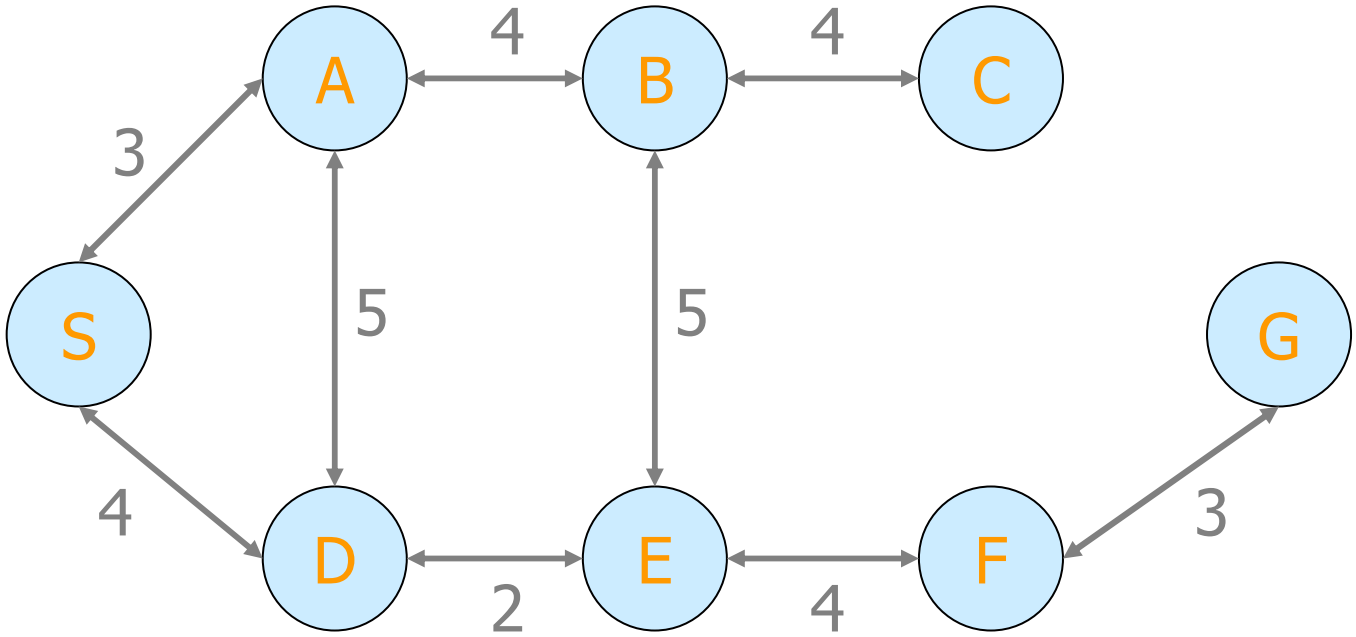
Función-lista-de-espera ← una función que ordena los nodos mediante FUNCIÓN-EVAL

responde con BUSQUEDA-GENERAL (*Problema*, *Función-lista-de-espera*)

Búsqueda primero “el mejor”

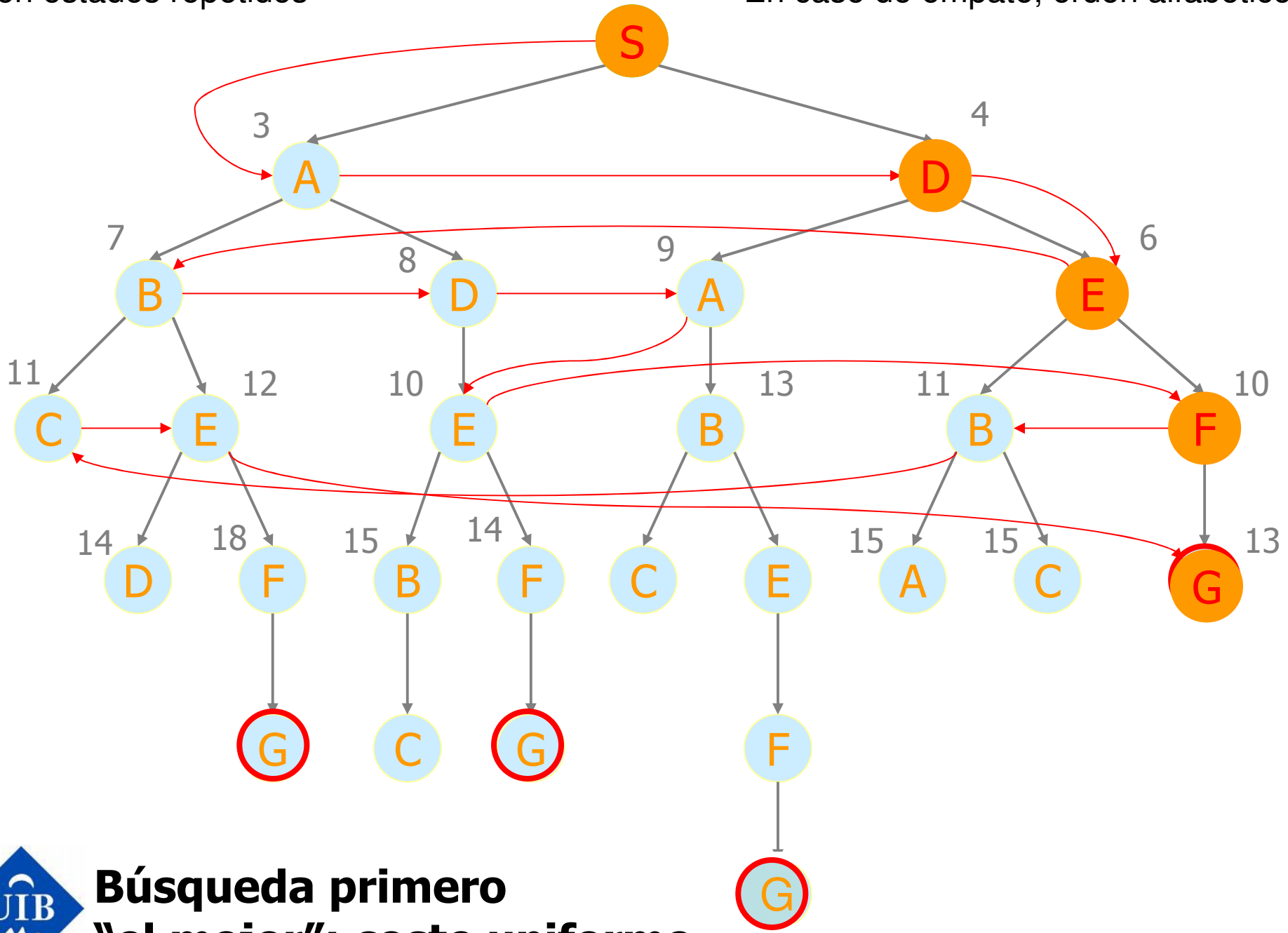
- Así como existe una familia de algoritmos BUSQUEDA-GENERAL, con distintas funciones de ordenamiento, también existe una familia de algoritmos BUSQUEDA-PRIMERO-EL-MEJOR, que varían la función de evaluación

Ejemplo que se empleará para ilustrar los diferentes métodos de búsqueda informada



Con estados repetidos

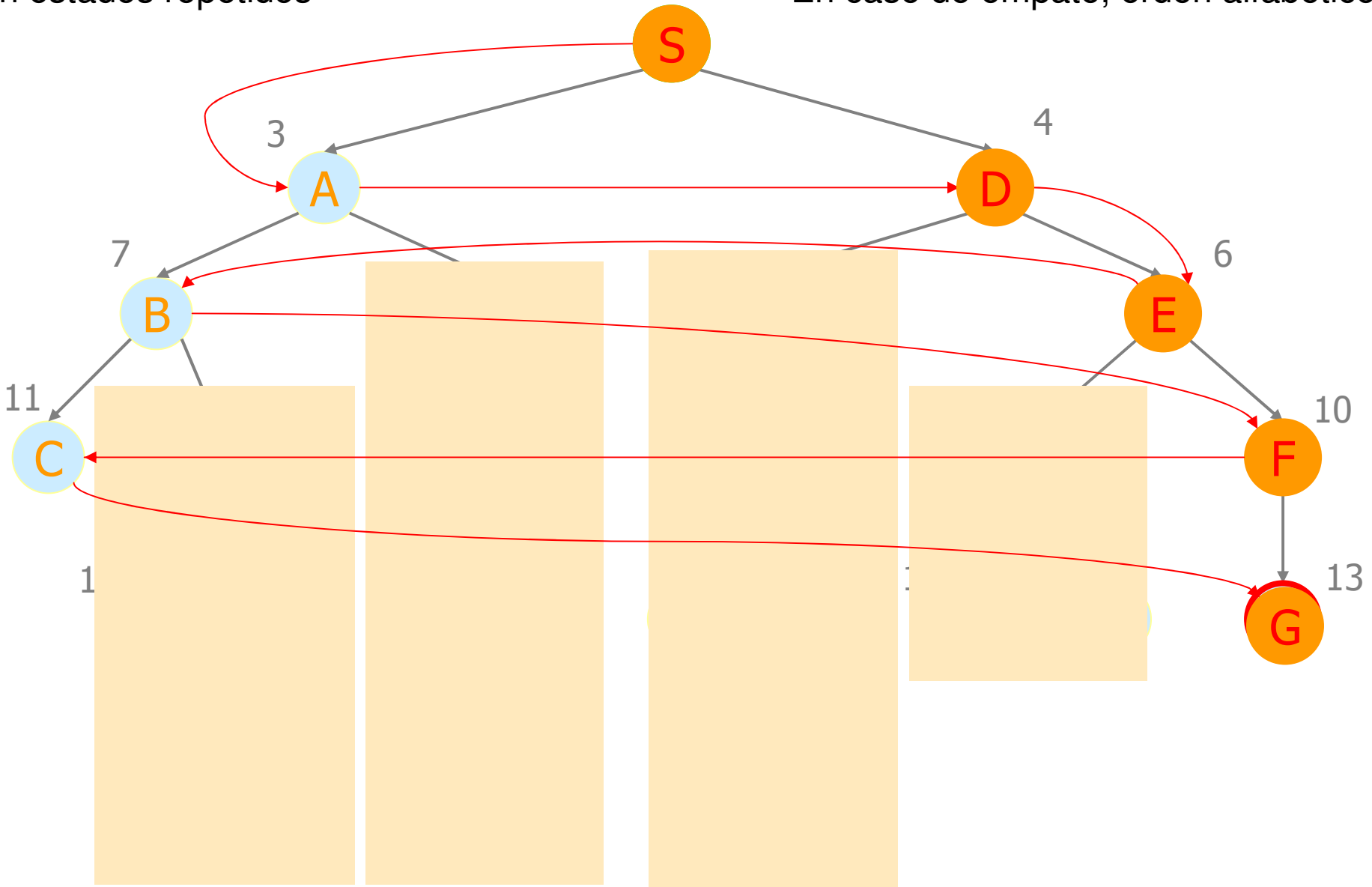
En caso de empate, orden alfabético



Búsqueda primero "el mejor": coste uniforme

Sin estados repetidos

En caso de empate, orden alfabético

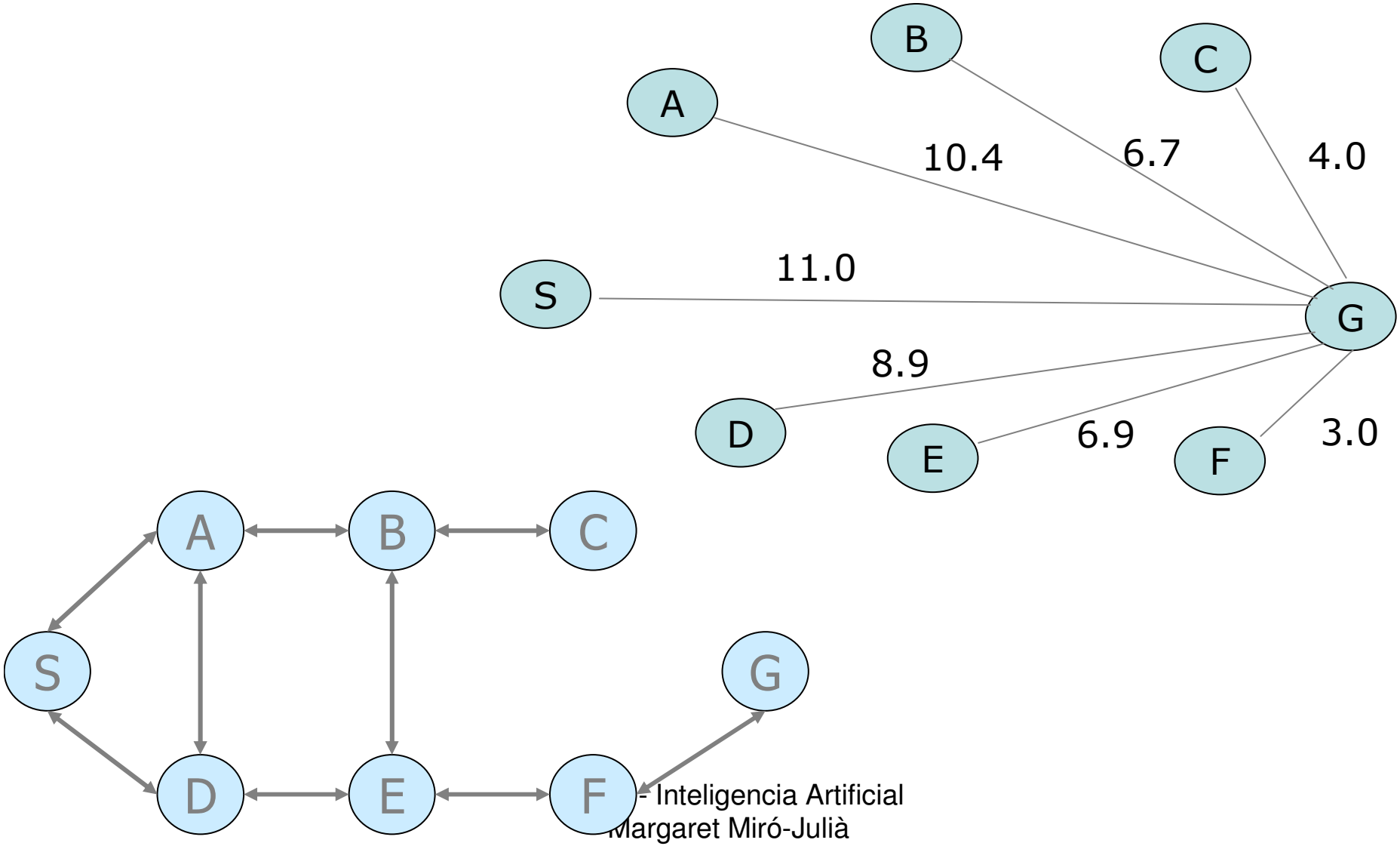


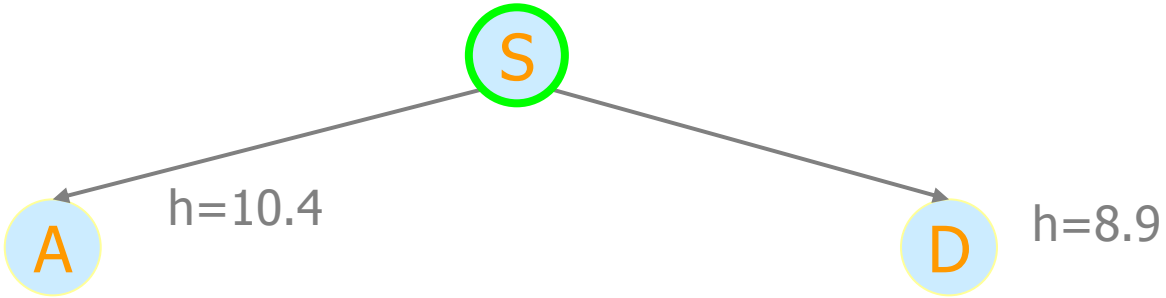
Búsqueda primero "el mejor": coste uniforme

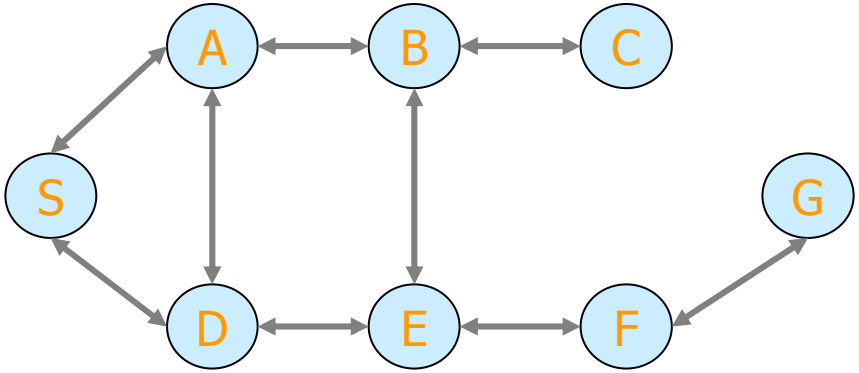
Búsqueda primero “el mejor”: búsqueda ávida (greedy)

- En el método de costo uniforme, se emplea el costo de ruta para decidir que nodo expandir, g(n) es el costo acumulado desde el nodo inicial hasta el nodo n
- Esta medida no proporciona una búsqueda dirigida a la meta
- Si se quiere enfocar la búsqueda hacia la meta, en la medida usada debe figurar algún tipo de estimación del costo de ruta del estado actual n a la meta h(n)

Se tiene una **estimación** de las distancias de los nodos a la meta

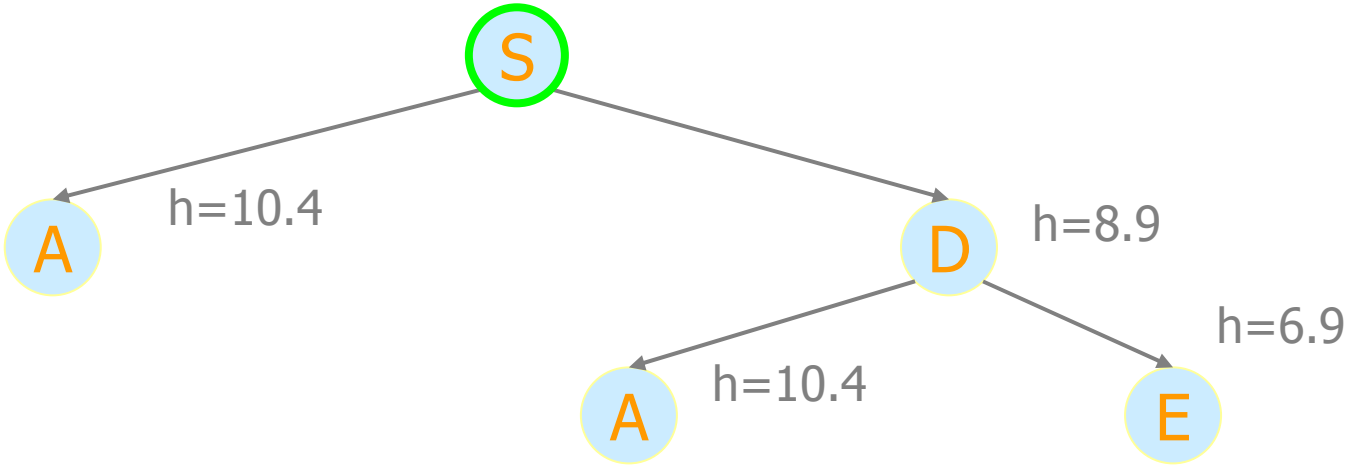


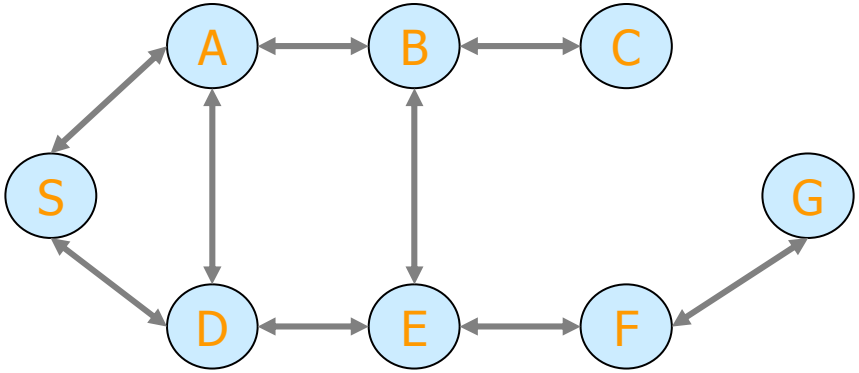




Búsqueda primero "el mejor": búsqueda ávida (greedy)

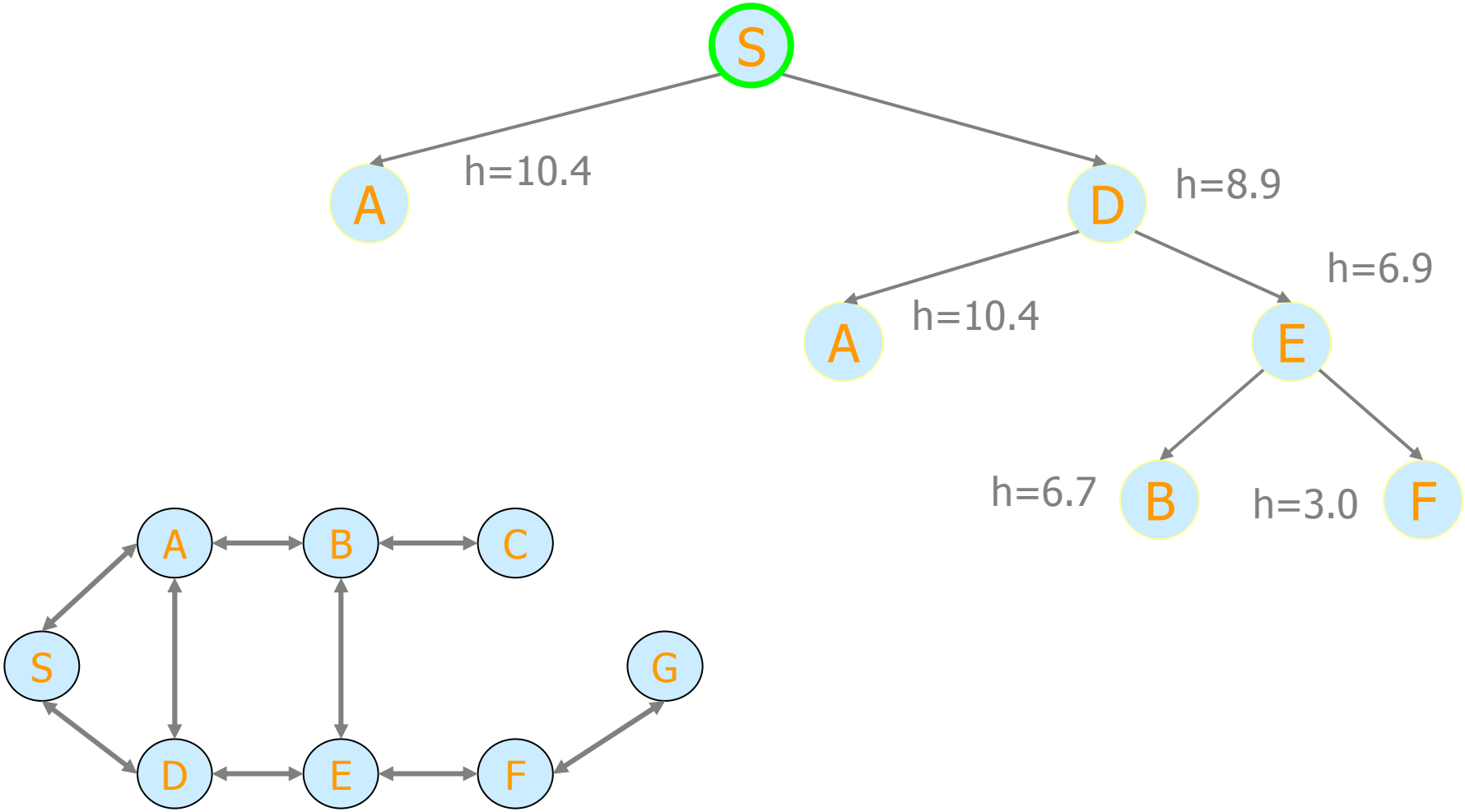
h(A,G) = 10.4 h(D,G) = 8.9
h(E,G) = 6.9 h(B,G) = 6.7
h(C,G) = 4.0 h(F,G) = 3.0





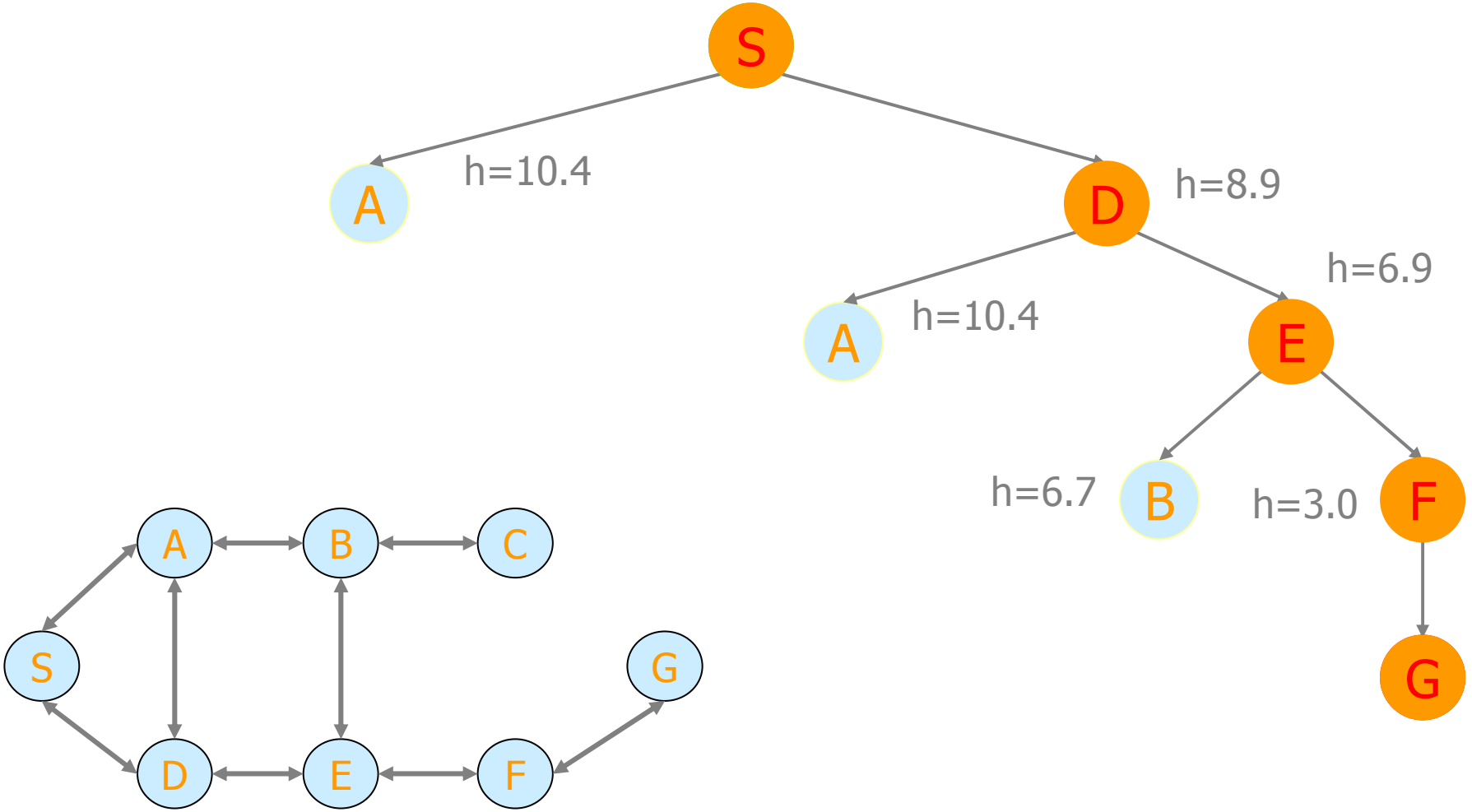
Búsqueda primero "el mejor": búsqueda ávida (greedy)

h(A,G) = 10.4 h(D,G) = 8.9
h(E,G) = 6.9 h(B,G) = 6.7
h(C,G) = 4.0 h(F,G) = 3.0



Búsqueda primero "el mejor": búsqueda ávida (greedy)

h(A,G) = 10.4 h(D,G) = 8.9
h(E,G) = 6.9 h(B,G) = 6.7
h(C,G) = 4.0 h(F,G) = 3.0



Búsqueda primero "el mejor": búsqueda ávida (greedy)

h(A,G) = 10.4 h(D,G) = 8.9
h(E,G) = 6.9 h(B,G) = 6.7
h(C,G) = 4.0 h(F,G) = 3.0

Búsqueda ávida (greedy)

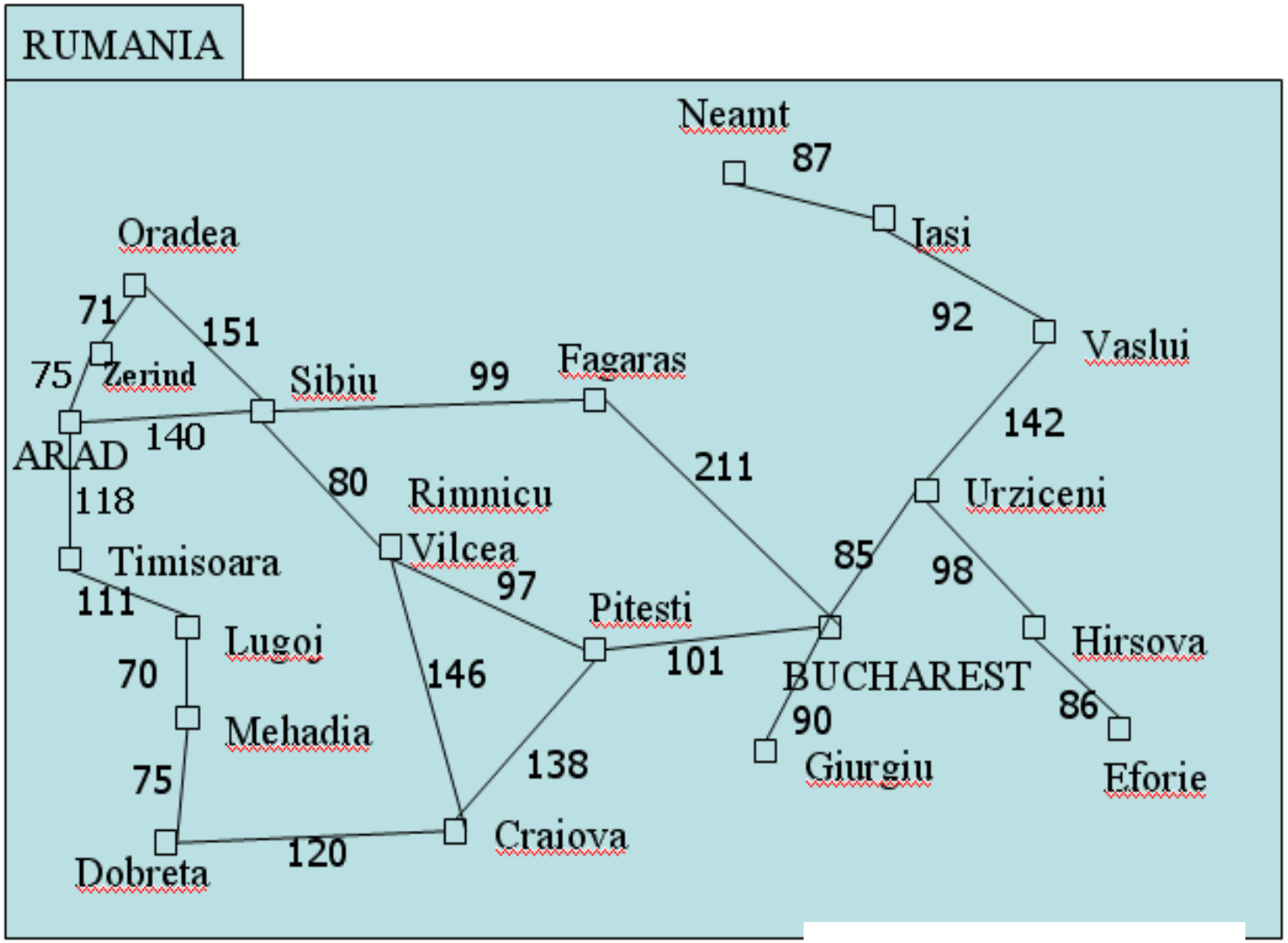
Algoritmo: Greedy

```
Est_abiertos.insertar(Estado inicial)
Actual  Est_abiertos.primero()
mientras no es_final?(Actual) y no Est_abiertos.vacía?() hacer
        Est_abiertos.borrar_primero()
        Est_cerrados.insertar(Actual)
        hijos   generar_sucesores (Actual)
        hijos   tratar_repetidos (Hijos, Est_cerrados, Est_abiertos)
        Est_abiertos.insertar(Hijos)
        Actual   Est_abiertos.primero()
fin
```

- La estructura de abiertos es una cola con prioridad
- La prioridad la marca la función de estimación (coste del camino que falta hasta la solución)
- En cada iteración se escoge el nodo más cercano a la solución (el primero de la cola), esto provoca que no se garantice la solución óptima

Est_abiertos = lista ordenada de estados generados
Est_cerrados = estados visitados

Búsqueda primero “lo mejor”



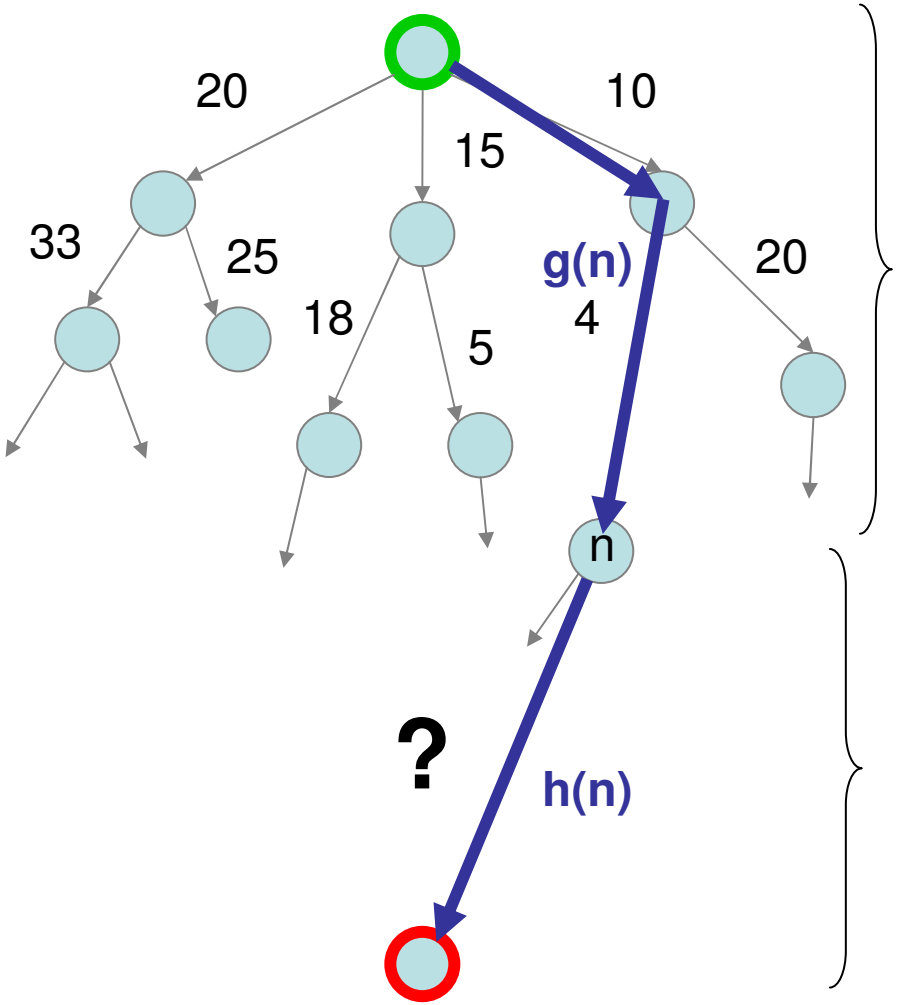
Dist. Línea Recta DLR de Bucarest a:	
Arad:	366
Bucarest:	0
Craiova:	160
Dobreta:	242
Eforie:	161
Fagaras:	178
Giurgiu:	77
Hirsova:	151
Iasi:	226
Lugoj:	244
Mehadia:	241
Neamt:	234
Oradea:	380
Pitesti:	98
Rimnicu Vilcea:	193
Sibiu:	253
Timisoara:	329
Urziceni:	80
Vaslui:	199
Zerind:	374

ACTIVIDAD 9

Búsqueda ávida (greedy)

- Estrategia: se escoge el nodo más cercano a la solución de acuerdo a la estimación realizada (heurística)
- Características:
  - Completitud: No es completa, encuentra soluciones en espacios finitos
  - Complejidad temporal: en el peor de los casos es $O(b^m)$, siendo m la profundidad máxima del espacio
  - Complejidad espacial: igual a la temporal (guarda todos los nodos en memoria)

    Si h es buena la complejidad disminuye
  - Optimalidad: No



La búsqueda de costo uniforme minimiza el costo hasta ese momento g(n)

- óptima y completa
- puede ser muy ineficiente

La búsqueda ávida minimiza el costo estimado hasta la meta h(n)

- poda el costo de búsqueda
- ni óptima, ni completa

f(n) = g(n) + h(n) = costo estimado de la solución más barata pasando por (n)

Búsqueda primero “el mejor”: búsqueda A*

- Estrategia: se escoge el nodo de acuerdo a la estimación realizada mediante f(n) = g(n) + h(n)
  - f es un valor estimado del coste total del camino que pasa por n
  - h (heurística) es un valor **estimado** de lo que falta para llegar desde n a la (a una) meta
  - g es un coste **real**, lo gastado por el camino más corto conocido desde el estado inicial hasta n
- La preferencia es siempre del nodo con menor f, en caso de empate, la preferencia es del nodo con menor h

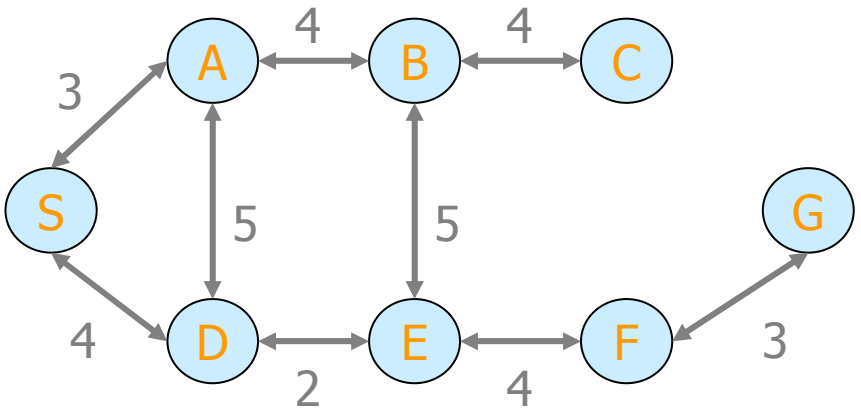
Algoritmo A*

Algoritmo: A*

```
Est_abiertos.insertar(Estado inicial)
Actual   Est_abiertos.primero()
mientras no es_final?(Actual) y no Est_abiertos.vacía?() hacer
        Est_abiertos.borrar_primer
        Est_cerrados.insertar(Actual)
        hijos   generar_sucesores (Actual)
        hijos   tratar_repetidos (Hijos, Est_cerrados, Est_abiertos)
        Est_abiertos.insertar(Hijos)
        Actual   Est_abiertos.primero()
fin
```

Est_abiertos = lista ordenada de estados generados
Est_cerrados = estados visitados

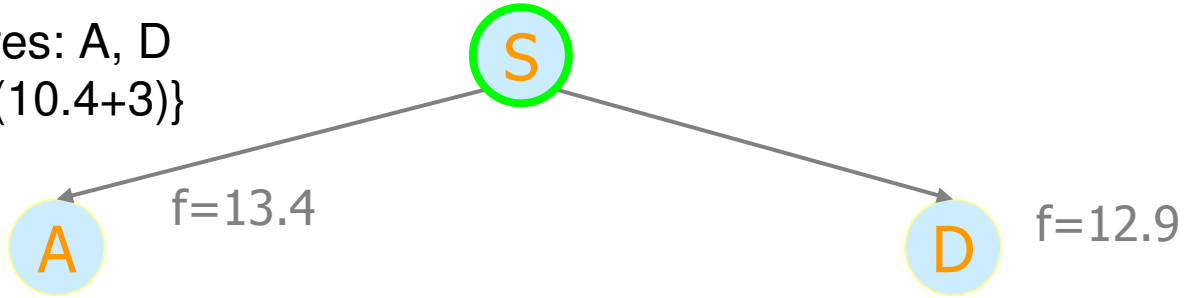
S

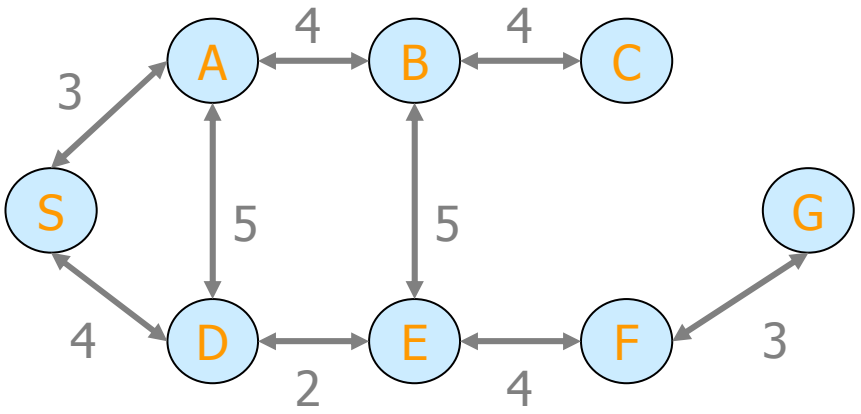


Búsqueda primero "el mejor": búsqueda A*

h(A,G) = 10.4 h(D,G) = 8.9
h(E,G) = 6.9 h(B,G) = 6.7
h(C,G) = 4.0 h(F,G) = 3.0

S no es solución,
generamos sucesores: A, D
LISTA={D(8.9+4),A(10.4+3)}



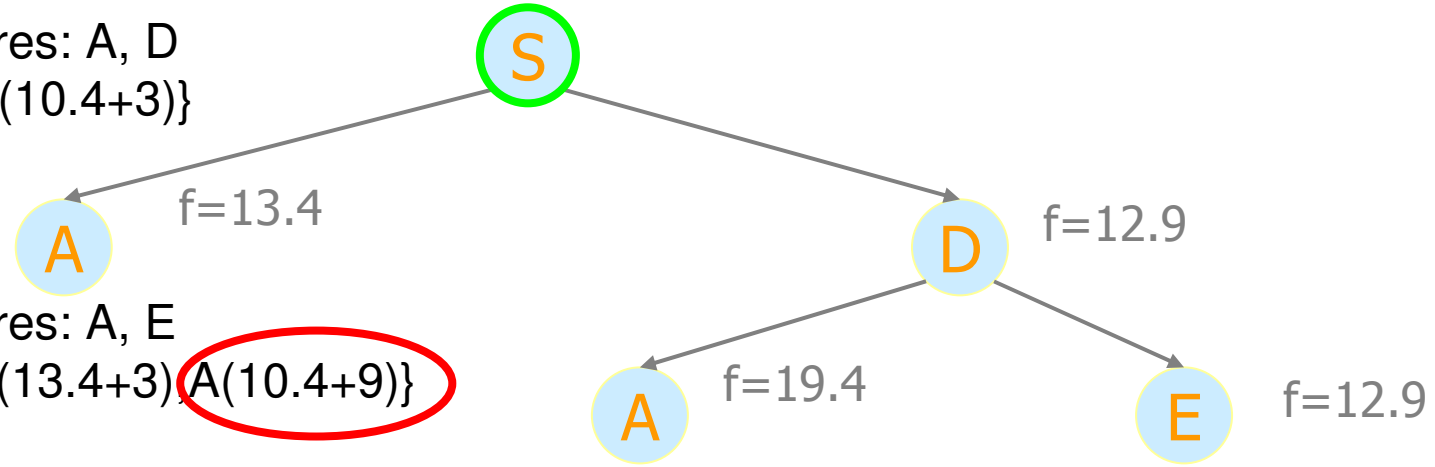


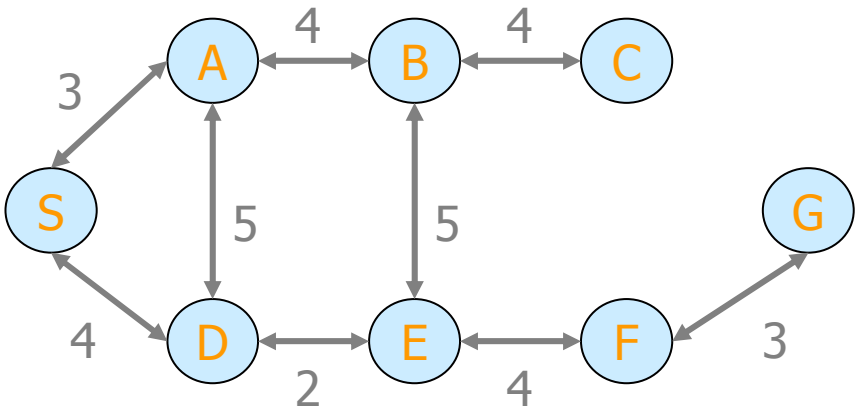
Búsqueda primero "el mejor": búsqueda A*

h(A,G) = 10.4 h(D,G) = 8.9
h(E,G) = 6.9 h(B,G) = 6.7
h(C,G) = 4.0 h(F,G) = 3.0

S no es solución,
generamos sucesores: A, D
LISTA={D(8.9+4),A(10.4+3)}

D no es solución,
generamos sucesores: A, E
LISTA={E(6.9+6),A(13.4+3),A(10.4+9)}





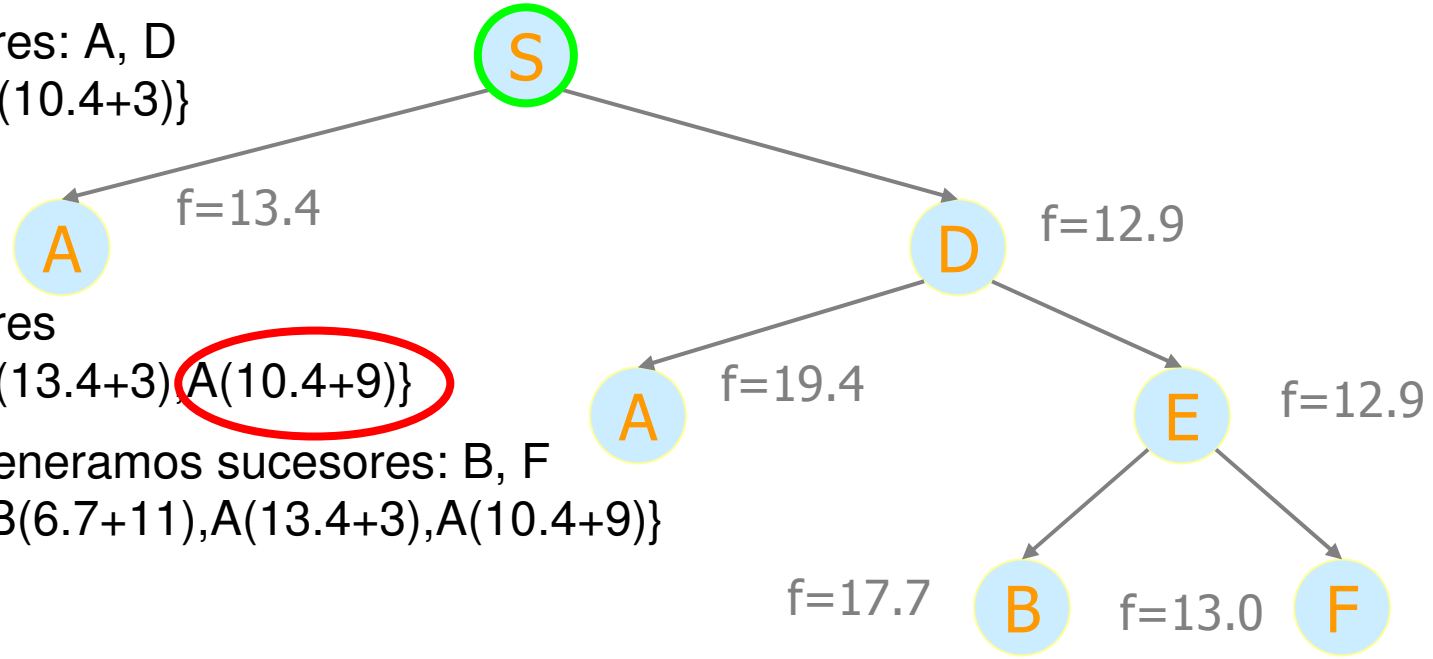
Búsqueda primero "el mejor": búsqueda A*

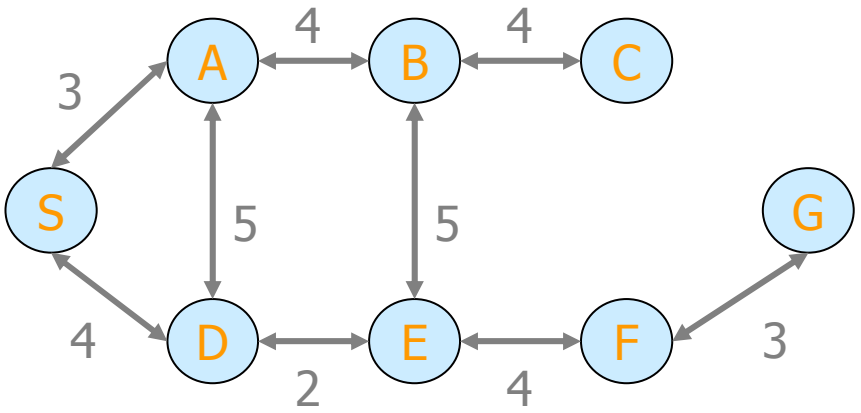
h(A,G) = 10.4 h(D,G) = 8.9
h(E,G) = 6.9 h(B,G) = 6.7
h(C,G) = 4.0 h(F,G) = 3.0

S no es solución,
generamos sucesores: A, D
LISTA={D(8.9+4),A(10.4+3)}

D no es solución,
generamos sucesores
LISTA={E(6.9+6),A(13.4+3),A(10.4+9)}

E no es solución, generamos sucesores: B, F
LISTA={F(3.0+10),B(6.7+11),A(13.4+3),A(10.4+9)}





Búsqueda primero "el mejor": búsqueda A*

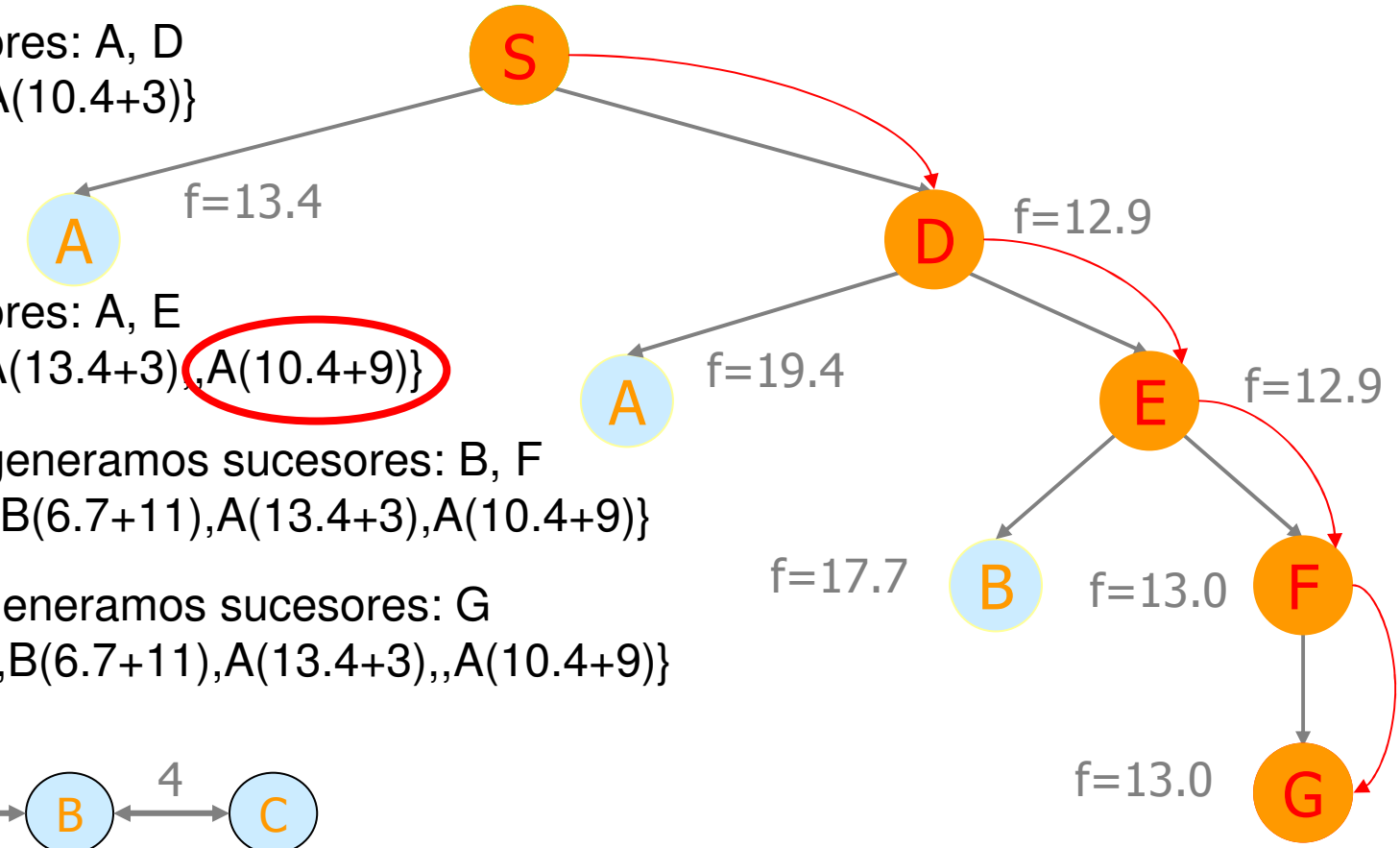
h(A,G) = 10.4 h(D,G) = 8.9
h(E,G) = 6.9 h(B,G) = 6.7
h(C,G) = 4.0 h(F,G) = 3.0

S no es solución,
generamos sucesores: A, D
LISTA={D(8.9+4),A(10.4+3)}

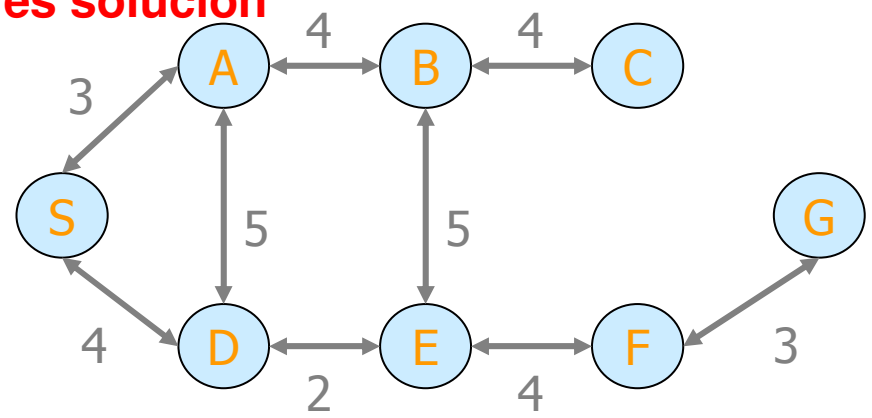
D no es solución,
generamos sucesores: A, E
LISTA={E(6.9+6),A(13.4+3),A(10.4+9)}

E no es solución, generamos sucesores: B, F
LISTA={F(3.0+10),B(6.7+11),A(13.4+3),A(10.4+9)}

F no es solución, generamos sucesores: G
LISTA={G(0.0+13),B(6.7+11),A(13.4+3),,A(10.4+9)}



G es solución

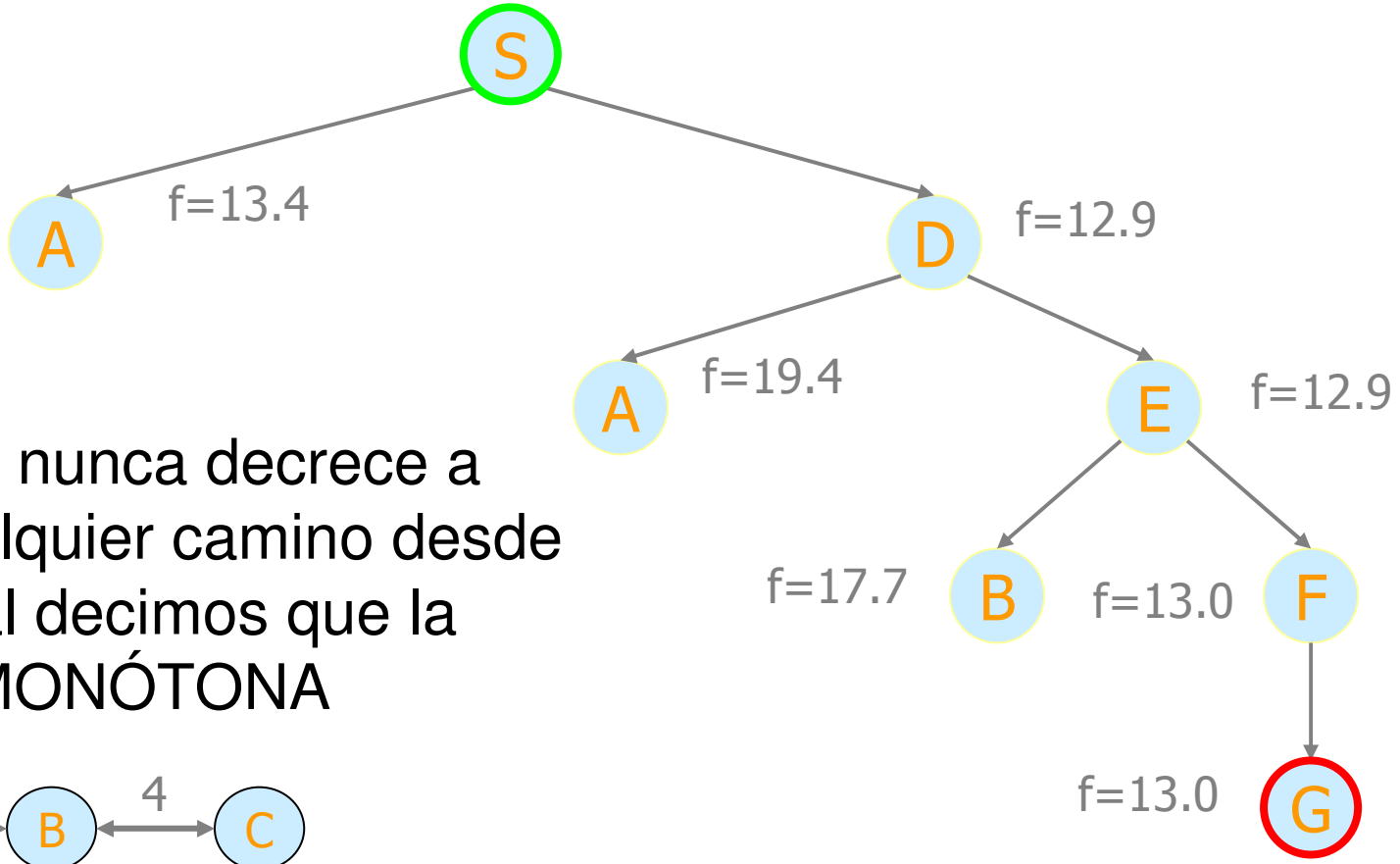


h(A,G) = 10.4 h(D,G) = 8.9
h(E,G) = 6.9 h(B,G) = 6.7
h(C,G) = 4.0 h(F,G) = 3.0

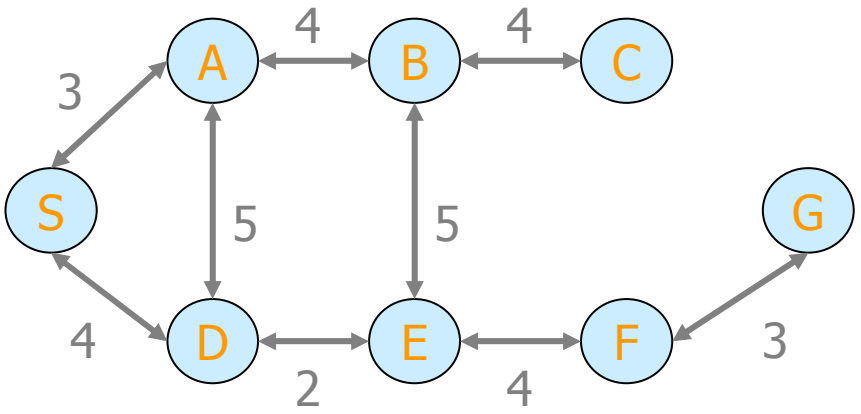
Búsqueda primero "el mejor": búsqueda A*

Algoritmo A*

- La estructura de abiertos es una cola con prioridad
- La prioridad la marca la función de estimación $f(n) = g(n) + h(n)$
- En cada iteración se escoge el mejor camino (el primero de la cola)
- ¡Es el mismo algoritmo que el primero “el mejor”!



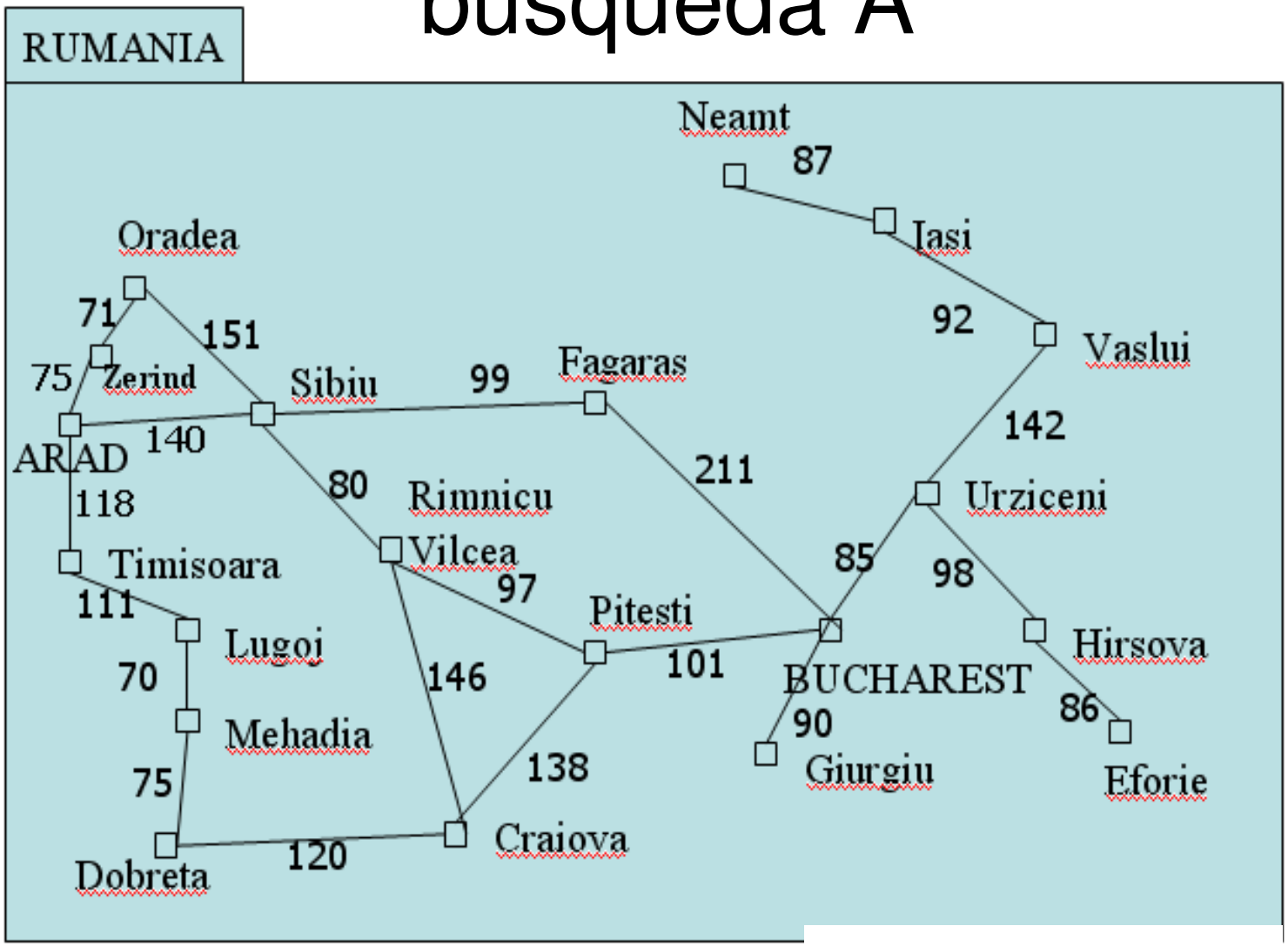
Si el costo de f nunca decrece a lo largo de cualquier camino desde el estado inicial decimos que la heurística es MONÓTONA



Búsqueda primero "el mejor": búsqueda A*

h(A,G) = 10.4 h(D,G) = 8.9
h(E,G) = 6.9 h(B,G) = 6.7
h(C,G) = 4.0 h(F,G) = 3.0

Búsqueda primero “lo mejor”: búsqueda A*



Dist. Línea Recta DLR de Bucarest a:	
Arad:	366
Bucarest:	0
Craiova:	160
Dobreta:	242
Eforie:	161
Fagaras:	178
Giurgiu:	77
Hirsova:	151
Iasi:	226
Lugoj:	244
Mehadia:	241
Neamt:	234
Oradea:	380
Pitesti:	98
Rimnicu Vilcea:	193
Sibiu:	253
Timisoara:	329
Urziceni:	80
Vaslui:	199
Zerind:	374

ACTIVIDAD 10

Búsqueda A*

- Estrategia: se escoge el nodo de acuerdo a la estimación realizada mediante f(n) = g(n) + h(n)
  - f es un valor estimado del coste total del camino que pasa por n
  - h (heurística) es un valor estimado de lo que falta para llegar desde n al (a un) objetivo
  - g es un coste real, lo gastado por el camino más corto conocido hasta n

Búsqueda A*

Características:

- Complejidad temporal: $O(r^p)$
- Complejidad espacial: $O(r^p)$

  el espacio de búsqueda de A* crece exponencialmente a no ser que

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

  donde h^* es el costo real para ir de n a la meta
- Usualmente A* se queda sin espacio antes de quedarse sin tiempo, puesto que mantiene a todos los nodos en memoria

  problema de memoria > problema del tiempo

El error en la función heurística no crezca más rápidamente que el logaritmo del coste del camino actual

Búsqueda A*

Características

- Completitud: Es un método completo de búsqueda, encuentra la solución, si existe, para cualquier tipo de grafos
- Optimalidad: encuentra la solución optima si para todo n, la estimación h(n) es menor que la distancia real a la meta h*(n) (**h es admisible**)
  - h no sobreestime el costo a la meta
  - f no sobrestima el costo real de la solución
  - Ejemplo: h = distancia en línea recta

Búsqueda A*

Casos límites de A*

- Si h=0 y g=0 $\Rightarrow$ Búsqueda aleatoria
- Si h=0 y g=1 $\Rightarrow$ Búsqueda Primero en Anchura
- Si h=h y g=0 $\Rightarrow$ Búsqueda ávida
- Si h=0 y g=g $\Rightarrow$ Búsqueda de costo uniforme
- Si h(n) > h*(n) $\Rightarrow$ se puede perder la ruta óptima
- Si h(n) << h*(n) $\Rightarrow$ ruta bien, puedo expandir nodos de más

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

Importancia de la heurística

- Desde un punto de vista formal, las funciones heurísticas deben cumplir:
  - $h(n) \geq 0$, para todo nodo n
  - $h(meta) = 0$
- Pero, si se le asigna a un nodo un coste estimado, h(n), superior al valor real, puede ocurrir que la búsqueda no se realice en el orden correcto y la trayectoria resultante no sea óptima
- Hay que encontrar buenas heurísticas

Funciones heurísticas

- Las heurísticas son criterios, métodos o principios para decidir cuál de entre varias acciones promete ser la mejor para alcanzar una determinada meta
- El uso de heurísticas guía la búsqueda de una solución, se obtiene una solución más rápidamente que con estrategias de búsqueda a ciegas

Funciones heurísticas

En problemas de búsqueda:

- Una heurística es una función que estima cómo de cerca estamos de la solución
- Cada heurística estará diseñada para un problema de búsqueda particular
- El valor de la heurística depende exclusivamente del estado; el valor es función de la información disponible hasta ese momento sobre la búsqueda

Funciones heurísticas

Heurísticas admisibles:

- Las heurísticas pesimistas (inadmisibles) impiden la optimalidad del algoritmo A* al descartar buenos planes
- Las heurísticas optimistas (admisibles) no subestiman la calidad de un buen plan
- Una heurística es admisible si $h(n) \leq h^*(n)$, donde $h^*(n)$ es el coste verdadero de la solución desde n

Funciones heurísticas

Diseño de heurísticas admisibles

- Para resolver problemas difíciles de búsqueda hay que encontrar heurísticas admisibles
- Para obtener heurísticas admisibles hay que resolver problemas relajados (problemas similares con menos restricciones sobre posibles acciones)
- Una heurística que devuelve un factor de ramificación más bajo es computacionalmente más eficiente para el problema en particular

- La construcción de heurísticas es un proceso de descubrimiento, no existe un mecanismo
- Sin embargo: *las heurísticas se descubren consultando modelos simplificados del dominio del problema*

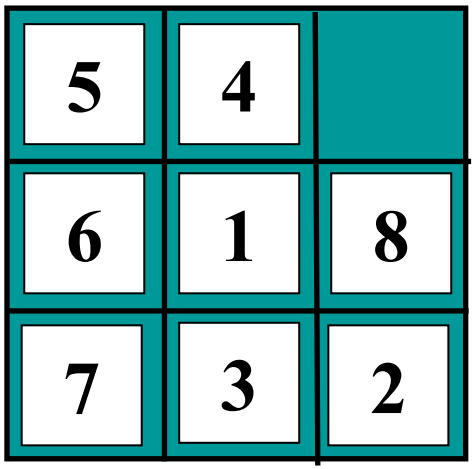
Funciones heurísticas

PROBLEMA: 8-puzzle

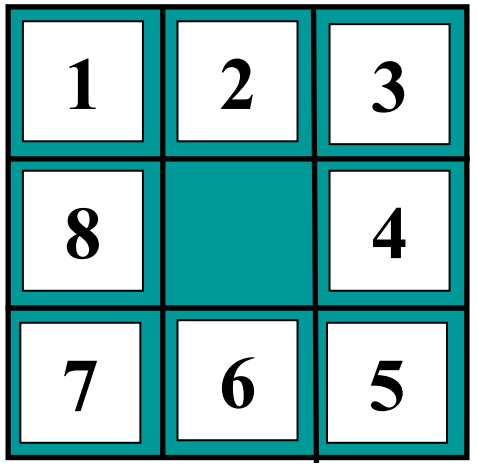
- Solución típica: 20 pasos
- r ≈ 3 (4 en centro, 2 en esquina, 3 en borde)
- Una búsqueda de profundidad 20 corresponde a una búsqueda a través de 3^{20}=3.5 10^{9} estados. Si se eliminan los estados repetidos quedan 9!=362880 estados diferentes
- Es imprescindible usar una buena función heurística

Funciones heurísticas

- Encontrar una heurística para saber o estimar cuánto me falta para llegar a la meta

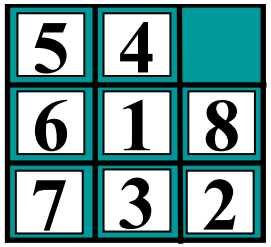


estado inicial

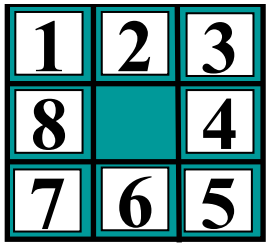


estado final

Funciones heurísticas



estado inicial

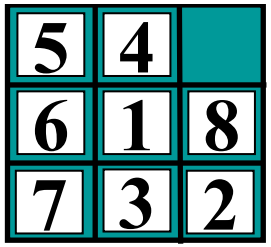


estado final

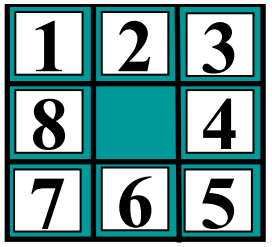
Existen diferentes heurísticas

- $h_0(n) = 0$
  - Búsqueda primero en anchura
  - A_0^* es admisible
  - muchas generaciones y expansiones
- $h_1(n) =$ nº piezas mal colocadas (respeto a la meta)
  - $h_1 = 7$
  - A_1^* admisible
  - A_1^* proporciona más información que A_0^*

Funciones heurísticas



estado inicial



estado final

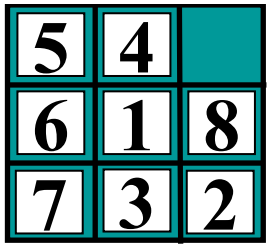
Existen diferentes heurísticas

- $h_2 = \Sigma_i d_i$

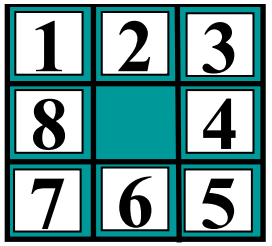
  d_i = distancia entre posición de la pieza i y su posición final = número mínimo de movimientos para llevar la pieza de una posición i a la final (distancia de Manhattan)

  – $h_2 = 18$
  – A2* es admisible.
  – Estadísticamente se comprueba que A_2^* es mejor que A_1^*, pero no se puede decir que sea más informado

Funciones heurísticas



estado inicial



estado final

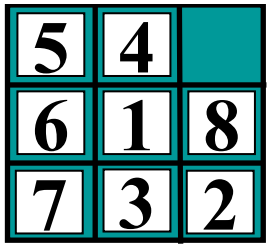
Existen diferentes heurísticas

- $h_3 = \Sigma_i p_i$

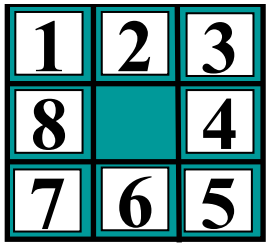
  p_i = pasos entre posición de la pieza i y su posición final incluyendo diagonal
  - $h_3 = 11$
  - A_3^* es admisible
- $h_4 = \Sigma_i D_i$

  d_i = distancia euclídea entre posición de la pieza i y su posición final = distancia en línea recta
  - $h_4 = 13.54$
  - A_4^* es admisible

Funciones heurísticas



estado inicial



estado final

Existen diferentes heurísticas

- $h_5 = 3 * \Sigma_i s_i$

$$s_i = \begin{cases} 0 \text{ si pieza i no en el centro y sucesora correcta} \\ 1 \text{ si pieza i en el centro} \\ 2 \text{ si pieza i no en el centro y sucesora incorrecta} \end{cases}$$

1		3
8	2	4
7	6	5

$h_5 = 3(2+1) = 9$
$h = 1$

A_5*no admisible

A_5* no se puede comparar con A_1* o A_2* pero va más rápido (aunque la *h* a calcular requiera más tiempo)

Funciones heurísticas

- Al realizar los cálculos he violado restricciones del caso real y he descubierto situaciones "relajadas"
- Los valores obtenidos son estimaciones de la distancia real a recorrer (en pasos)
- La heurística óptima es la máxima entre $h_1,...,h_4$, he sido optimista
- Todas son estrategias admisibles $\Rightarrow$ optimistas (calculo menos pasos) $\Rightarrow$ nunca sobreestiman la distancia (definen la "admisibilidad")

Funciones heurísticas

Para encontrar una buena heurística

- Hay que identificar todas las restricciones del problema
- Simplificar el modelo, relajar estas restricciones (quitar una o varias), de manera que se convierte en el mismo problema, pero menos estricto (subproblema más sencillo)
- Obtenemos una función heurística más simple, pero lo suficientemente útil para que estime lo bueno que es el estado para llegar a la meta en función de las restantes restricciones
- Para poder simplificar el modelo, el problema ha de poderse descomponer en subproblemas más sencillos

Funciones heurísticas

Para encontrar una buena heurística: 8-puzzle

- Relajar las restricciones
  - Una placa se puede mover del cuadrado A al B si A es adyacente a B y B está vacío.
  - Una placa se puede mover desde el cuadrado A al B si A está adyacente a B
  - Una placa se puede mover del cuadrado A al B si B está vacío
  - Una placa se puede mover del cuadrado A al B
- En general, el costo de una solución exacta obtenida al relajar un problema sirve como una buena heurística para el problema original
- Siempre será mejor usar una función heurística mayor, sin sobreestimar: $h(n) = \max(h_1(n), ..., h_m(n))$
- La evaluación heurística debiera ser eficiente
- Hay que considerar también el costo de usar h en un nodo

Calidad de la heurística

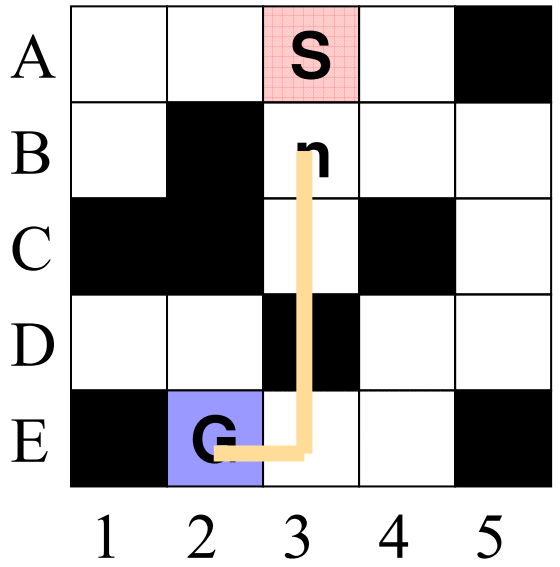
- h_2 está más informada que h_1 si para cualquier nodo n, $h_2(n) \geq h_1(n)$ y ambos son admisibles
- Si h_2 domina a h_1, entonces A* al usar h_2 va a expandir menos nodos que usando h_1
- Siempre será mejor usar una función heurística con números mayores sin pasarse a la sobreestimación

El laberinto

Problema: Ir de **A3** a **E2**, paso a paso, evitando obstáculos (cuadrados negros)
Estado: n=(i, j) i = A, B, .., E; j = 1, 2, …, 5
Operadores: (ordenados)

- **IZ**(n)
- **AB**(n)
- **DE**(n)
- **AR**(n)

Cada operador tiene costo 1
Heurística: ?



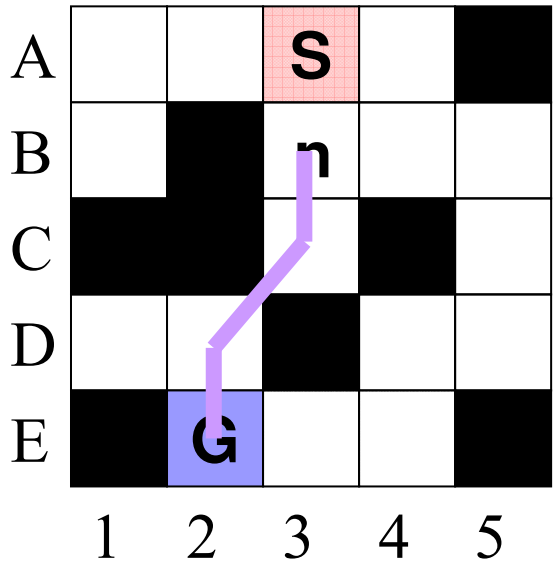
Relajamos obstáculos
h=número mínimo de movimientos permitidos desde nodo n hasta G sin considerar obstáculos

El laberinto

Problema: Ir de **A3** a **E2**, paso a paso, evitando obstáculos (cuadrados negros)
Estado: n=(i, j) i = A, B, .., E; j = 1, 2, …, 5
Operadores: (ordenados)

- **IZ**(n)
- **AB**(n)
- **DE**(n)
- **AR**(n)

Cada operador tiene costo 1
Heurística: ?



Relajamos obstáculos
h=número mínimo de movimientos permitidos desde nodo n hasta G sin considerar obstáculos

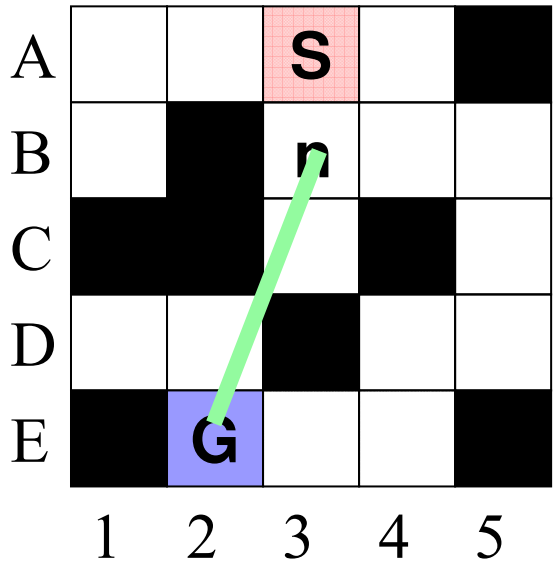
Relajamos movimientos
h=número mínimo de movimientos desde nodo n hasta G considerando obstáculos y permitiendo movimiento en diagonal

El laberinto

Problema: Ir de **A3** a **E2**, paso a paso, evitando obstáculos (cuadrados negros)
Estado: n=(i, j) i = A, B, .., E; j = 1, 2, …, 5
Operadores: (ordenados)

- **IZ**(n)
- **AB**(n)
- **DE**(n)
- **AR**(n)

Cada operador tiene costo 1
Heurística: ?



Relajamos obstáculos
h=número mínimo de movimientos permitidos desde nodo n hasta G sin considerar obstáculos

Relajamos movimientos
h=número mínimo de movimientos desde nodo n hasta G considerando obstáculos y permitiendo movimiento en diagonal

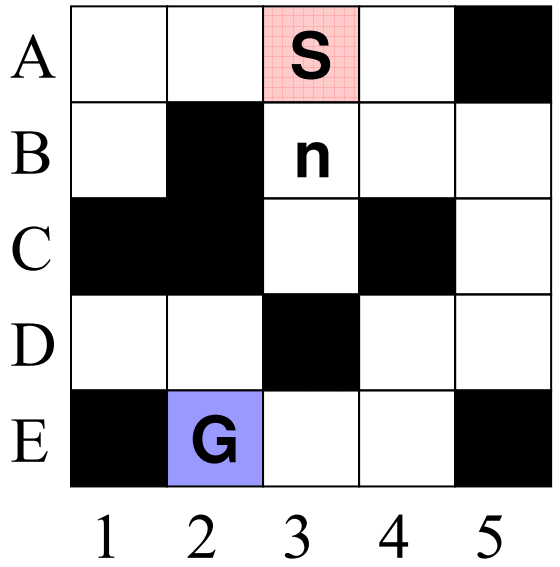
Relajamos obstáculos y movimientos
h=distancia en línea recta entre nodo n y G sin considerar obstáculos y permitiendo cualquier movimiento (Pitágoras)

El laberinto

Problema: Ir de **A3** a **E2**, paso a paso, evitando obstáculos (cuadrados negros)
Estado: n=(i, j) i = A, B, .., E; j = 1, 2, …, 5
Operadores: (ordenados)

- **IZ**(n)
- **AB**(n)
- **DE**(n)
- **AR**(n)

Cada operador tiene costo 1
Heurística:



Relajamos obstáculos
h=número mínimo de movimientos permitidos desde nodo n hasta G sin considerar obstáculos

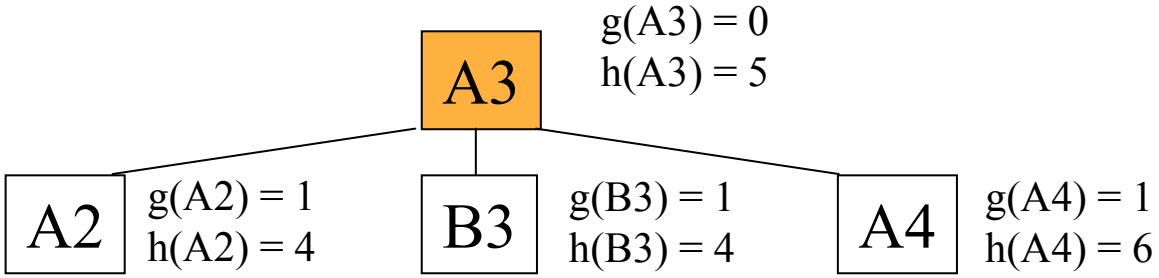
$$h = |i_n - i_G| + |j_n - j_G|$$

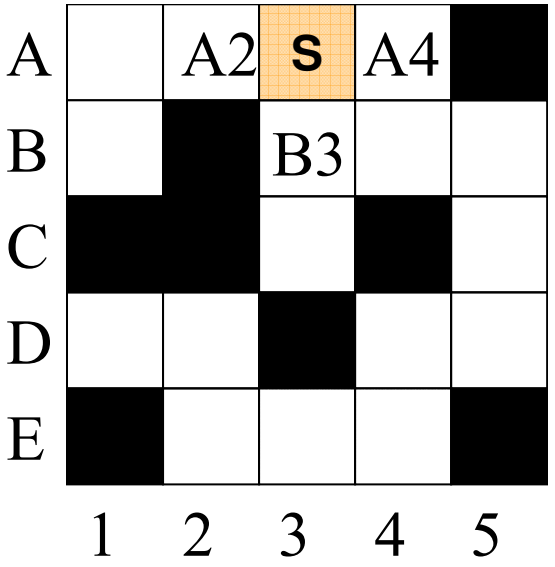
Tomar A=1; B=2; C=3; D=4; E=5

¿Estados repetidos?

Sin estados repetidos
h es igual para todos los estados repetidos, el valor mínimo de f corresponde al mínimo g

Aplicamos A*, solución óptima





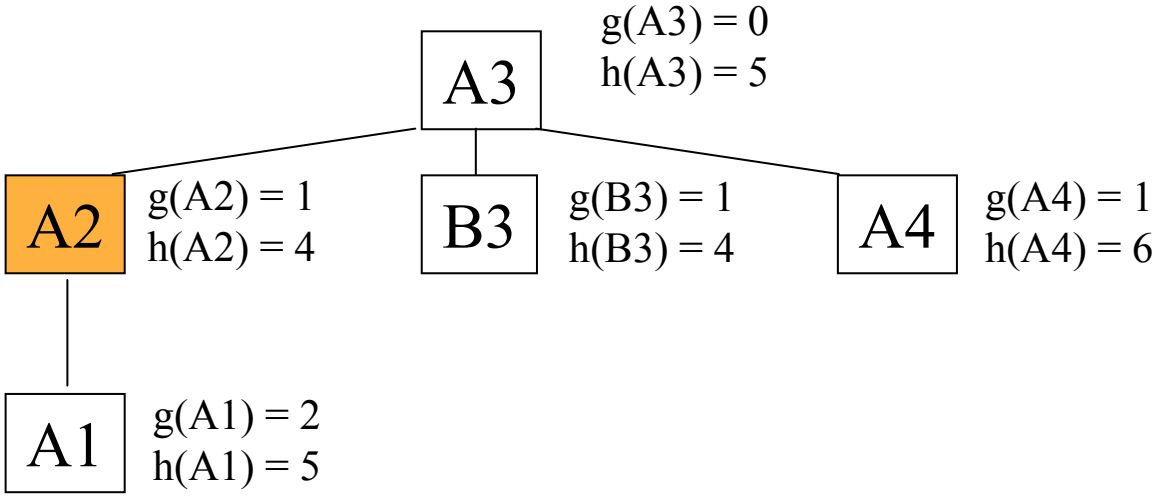
LISTA = {S=A3(5)}
S es estado actual, no es estado meta

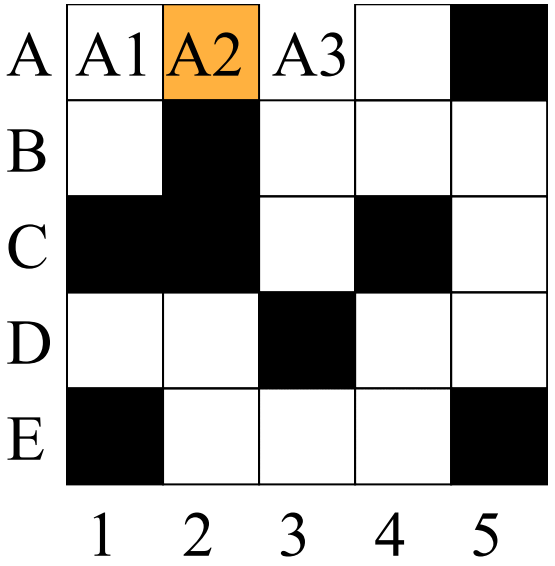
Generamos sucesores: A2; B3; A4

Para ordenar los sucesores, hay que calcular f(n)

LISTA = {A2(5); B3(5); A4(7)}
VISITADOS = {S=A3(5)}

A2 es estado actual





LISTA = {A2(5); B3(5); A4(7)}
VISITADOS = {S=A3(5)}

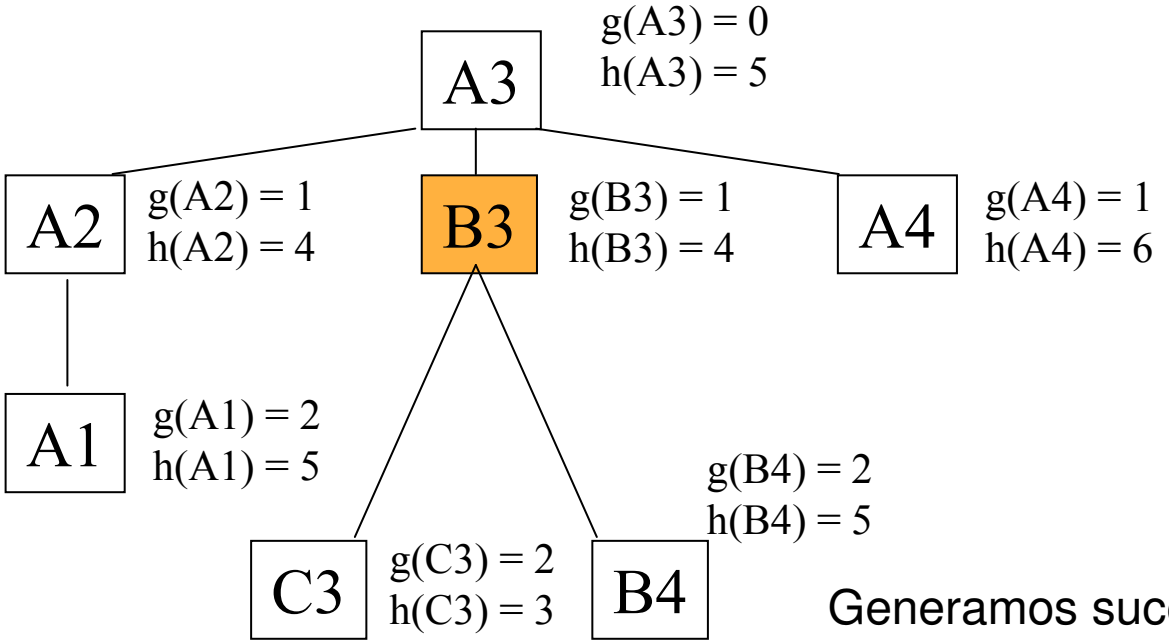
A2 es estado actual, no es estado meta

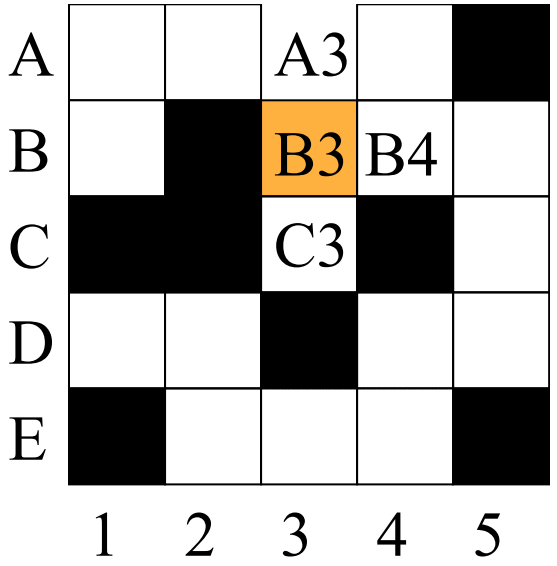
Generamos sucesores: A1; A3

Para ordenar los nodos, hay que calcular f(n)

LISTA ={B3(5); A1(7); A4(7)}
VISITADOS = {S=A3(5); A2(5)}

B3 es estado actual





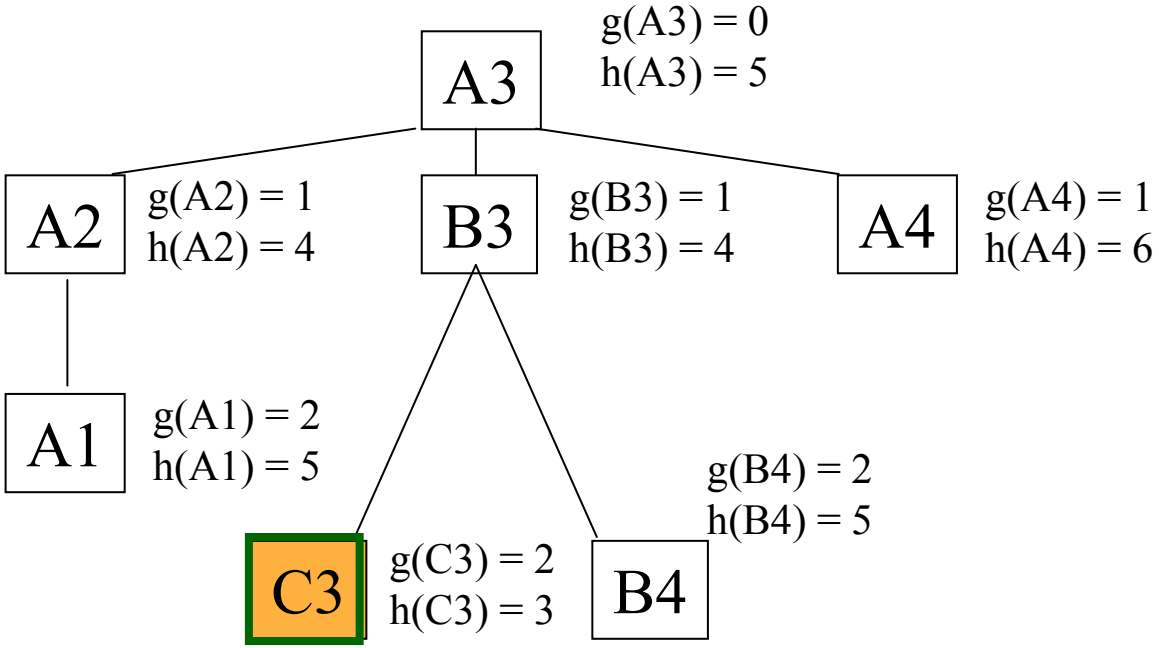
Generamos sucesores: B4; C3; A3

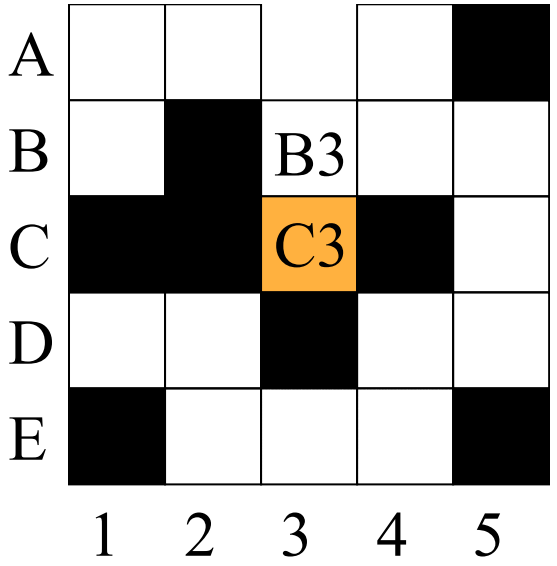
Para ordenar los nodos, hay que calcular f(n)
LISTA ={C3(5); A1(7);B4(7); A4(7)}
VISITADOS = {S(5); A2(5); B3(5)}

C3 es estado actual

LISTA ={B3(5); A1(7); A4(7)}
VISITADOS = {S(5); A2(5)}

B3 es estado actual, no es estado meta





LISTA ={C3(5); A1(7);B4(7); A4(7)}
VISITADOS = {S(5); A2(5); B3(5)}

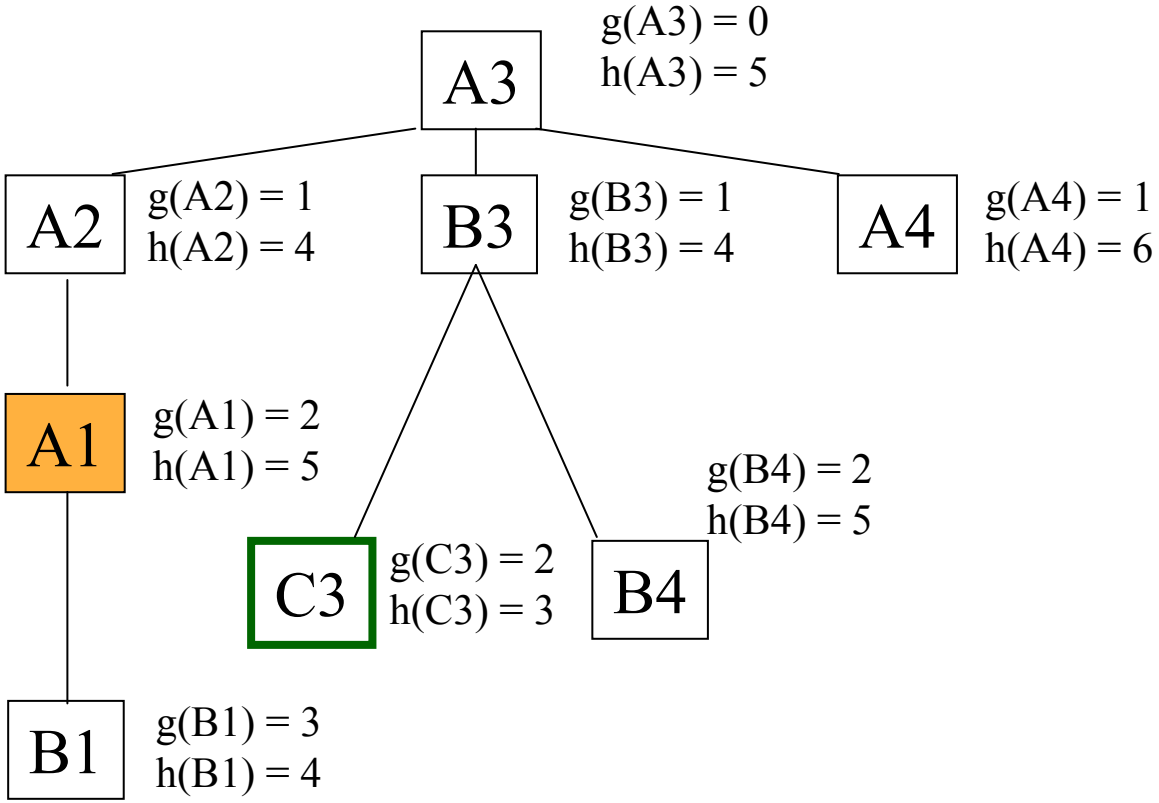
C3 es estado actual, no es estado meta

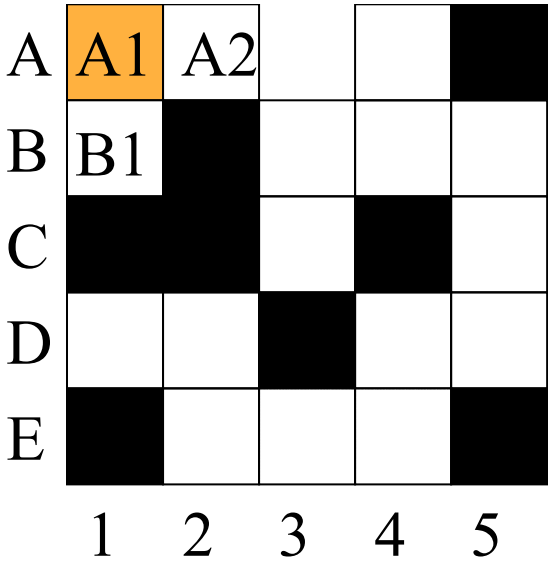
Generamos sucesores: B3

C3 es nodo hoja

LISTA ={A1(7);B4(7); A4(7)}
VISITADOS = {S(5); A2(5); B3(5); C3(5)}

A1 es estado actual





Generamos sucesores: B1; A2

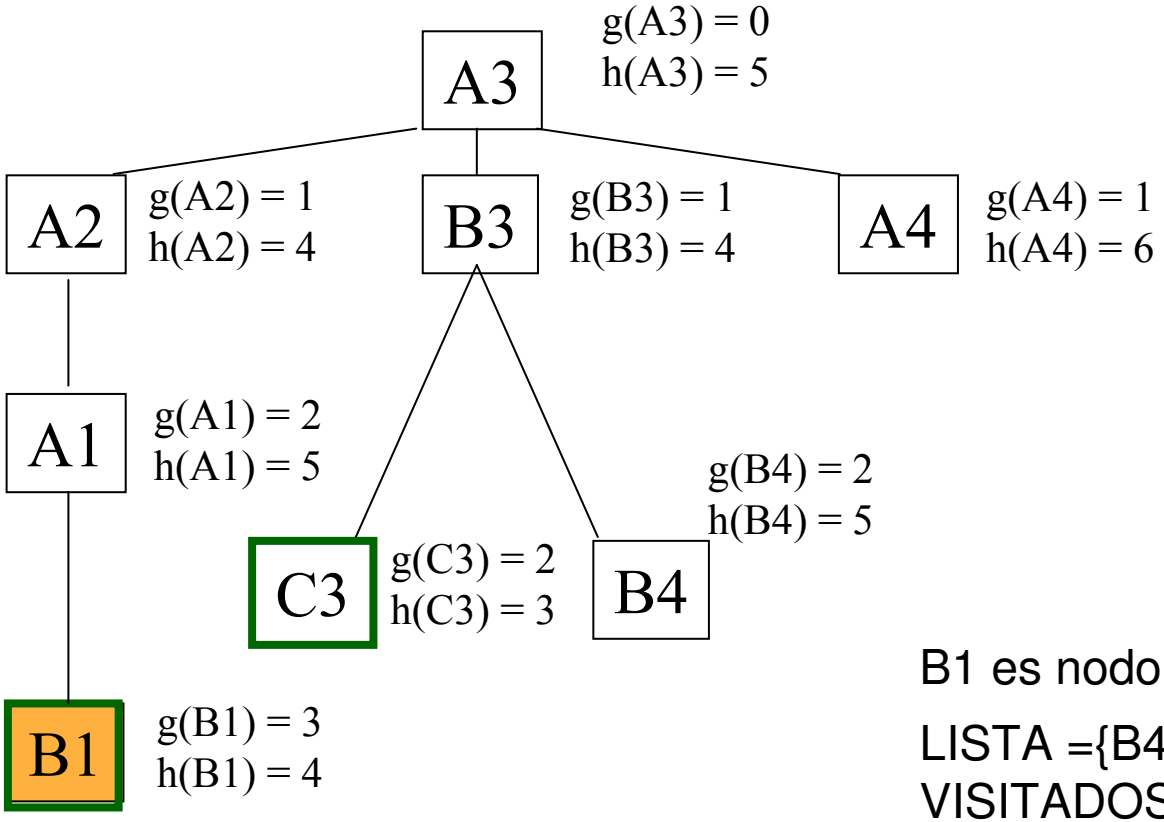
Para ordenar los nodos, hay que calcular f(n)

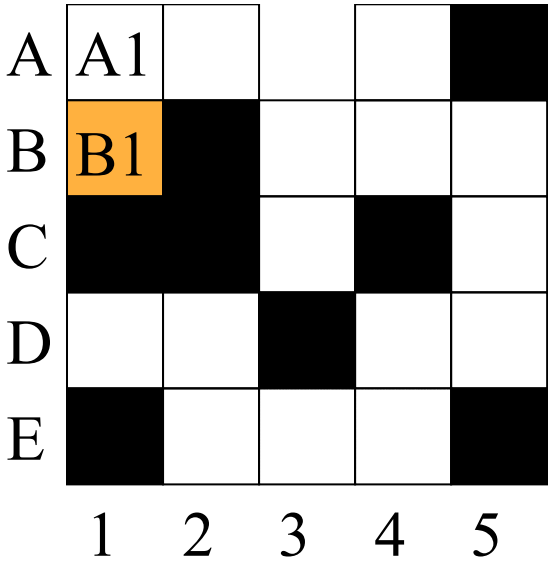
LISTA ={B1(7); B4(7); A4(7)}
VISITADOS = {S(5); A2(5); B3(5);
C3(5); A1(7)}

B1 es estado actual

LISTA ={A1(7); B4(7); A4(7)}
VISITADOS = {S(5); A2(5); B3(5); C3(5)}

A1 es estado actual, no es estado meta





B1 es nodo hoja

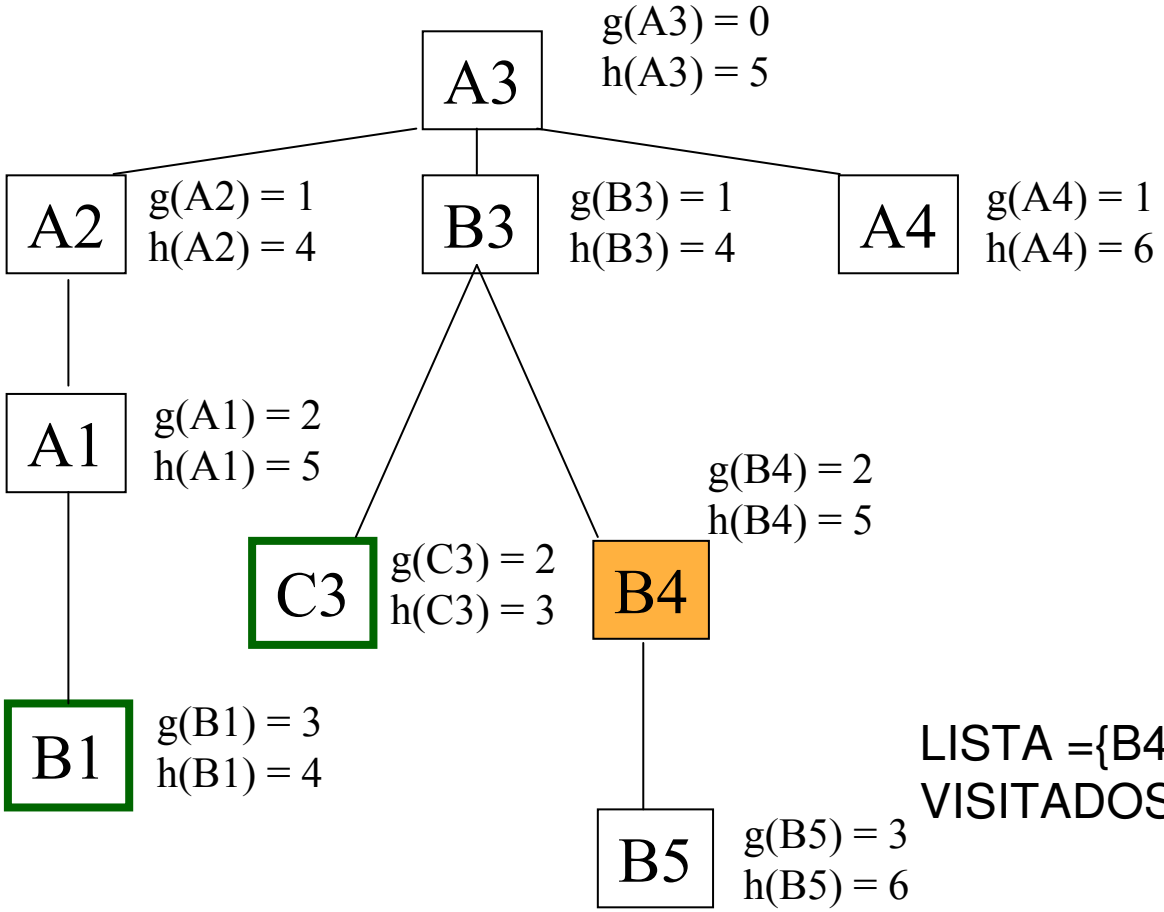
LISTA ={B4(7); A4(7)}
VISITADOS = {S(5); A2(5); B3(5);
C3(5); A1(7); B1(7)}

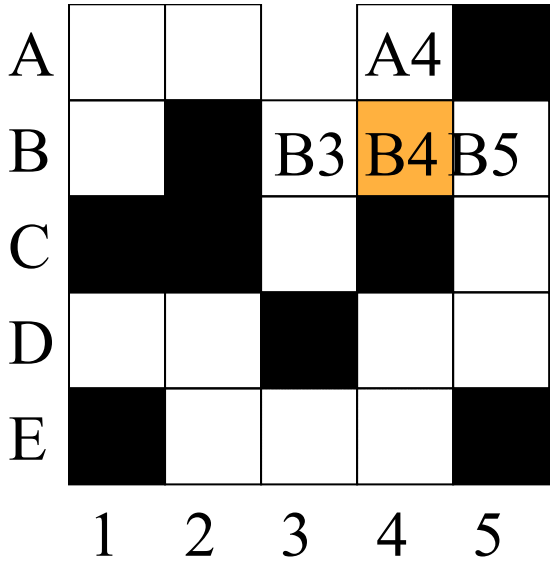
B4 es estado actual

LISTA ={B1(7); B4(7); A4(7)}
VISITADOS = {S(5); A2(5); B3(5);
C3(5); A1(7)}

B1 es estado actual, no es estado meta

Generamos sucesores: A1





LISTA ={B4(7); A4(7)}
VISITADOS = {S(5); A2(5); B3(5);
C3(5); A1(7); B1(7)}

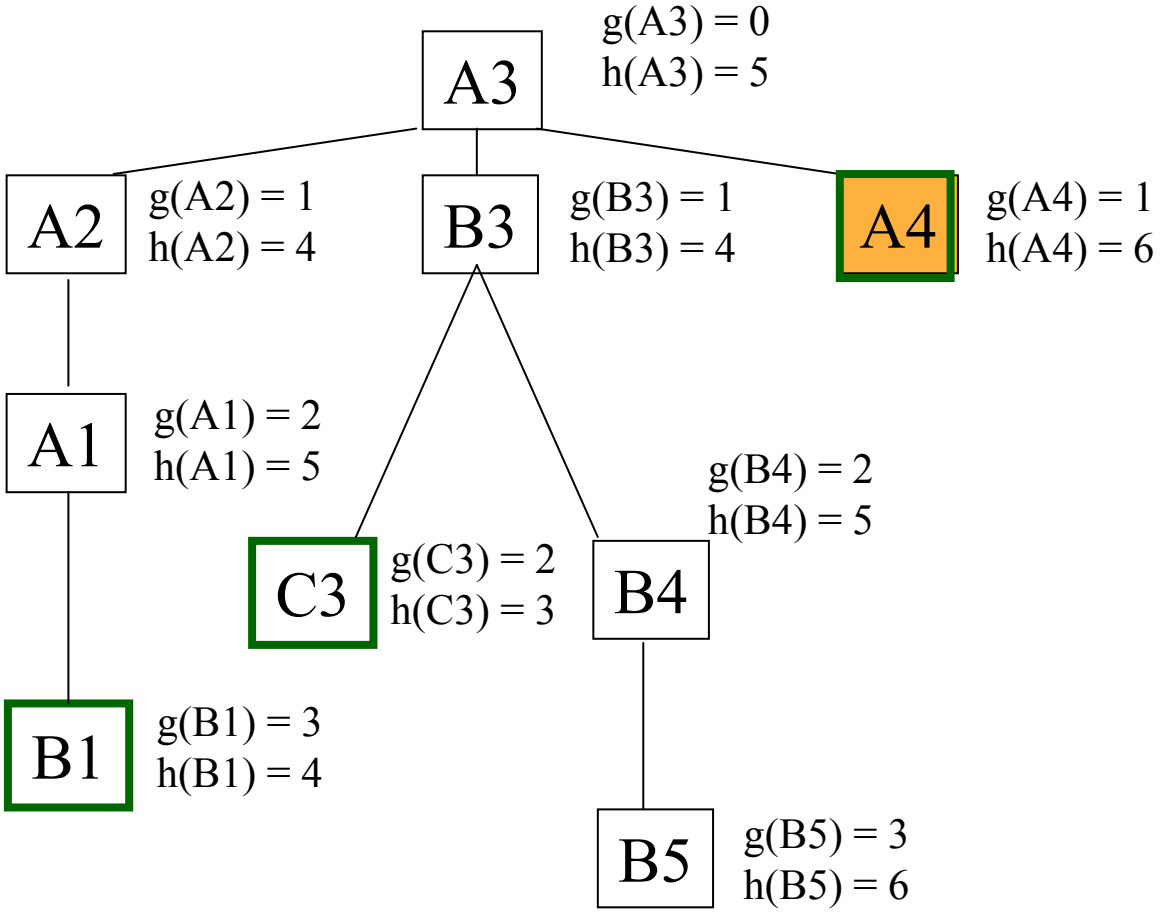
B4 es estado actual, no es estado meta

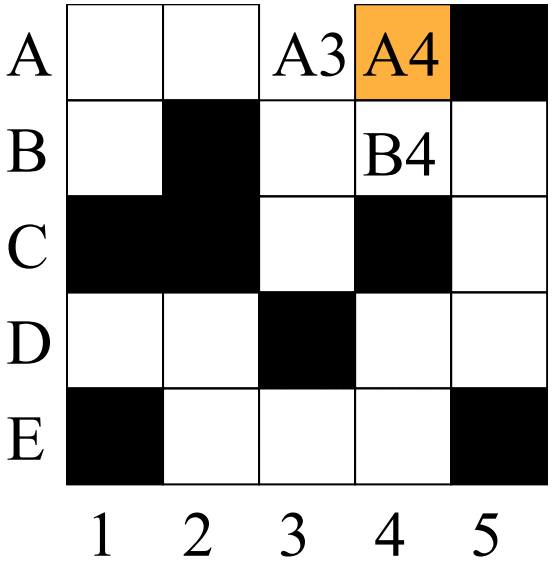
Generamos sucesores: B3; B5; A4

Para ordenar los nodos, hay que calcular f(n)

LISTA ={A4(7); B5(9)}
VISITADOS = {S(5); A2(5); B3(5); C3(5);
A1(7); B1(7); B4(7)}

A4 es estado actual





Generamos sucesores: A3, B4

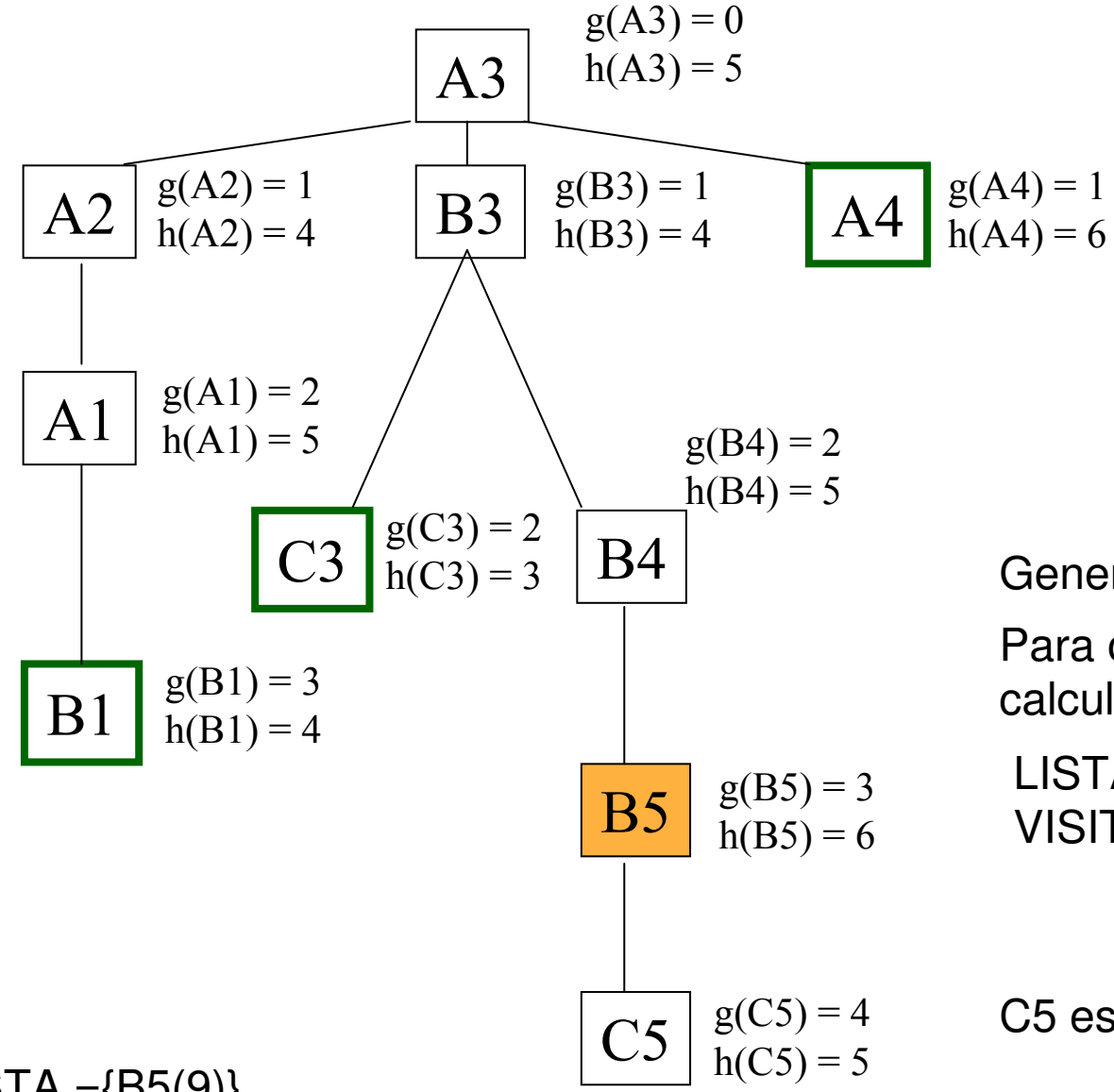
A4 es nodo hoja

LISTA ={B5(9)}
VISITADOS = {S(5); A2(5); B3(5); C3(5);
A1(7); B1(7); B4(7); A4(7)}

B5 es estado actual

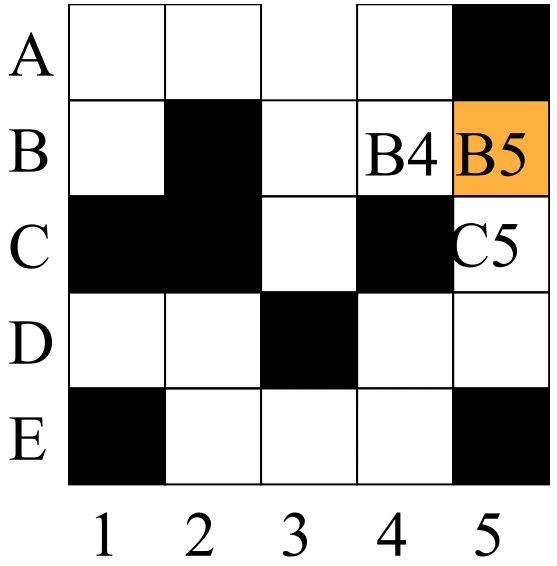
LISTA ={A4(7); B5(9)}
VISITADOS = {S(5); A2(5); B3(5); C3(5);
A1(7); B1(7); B4(7)}

A4 es estado actual, no es estado meta



LISTA ={B5(9)}
VISITADOS = {S(5); A2(5); B3(5); C3(5);
A1(7); B1(7); B4(7); A4(7)}

B5 es estado actual, no es estado meta

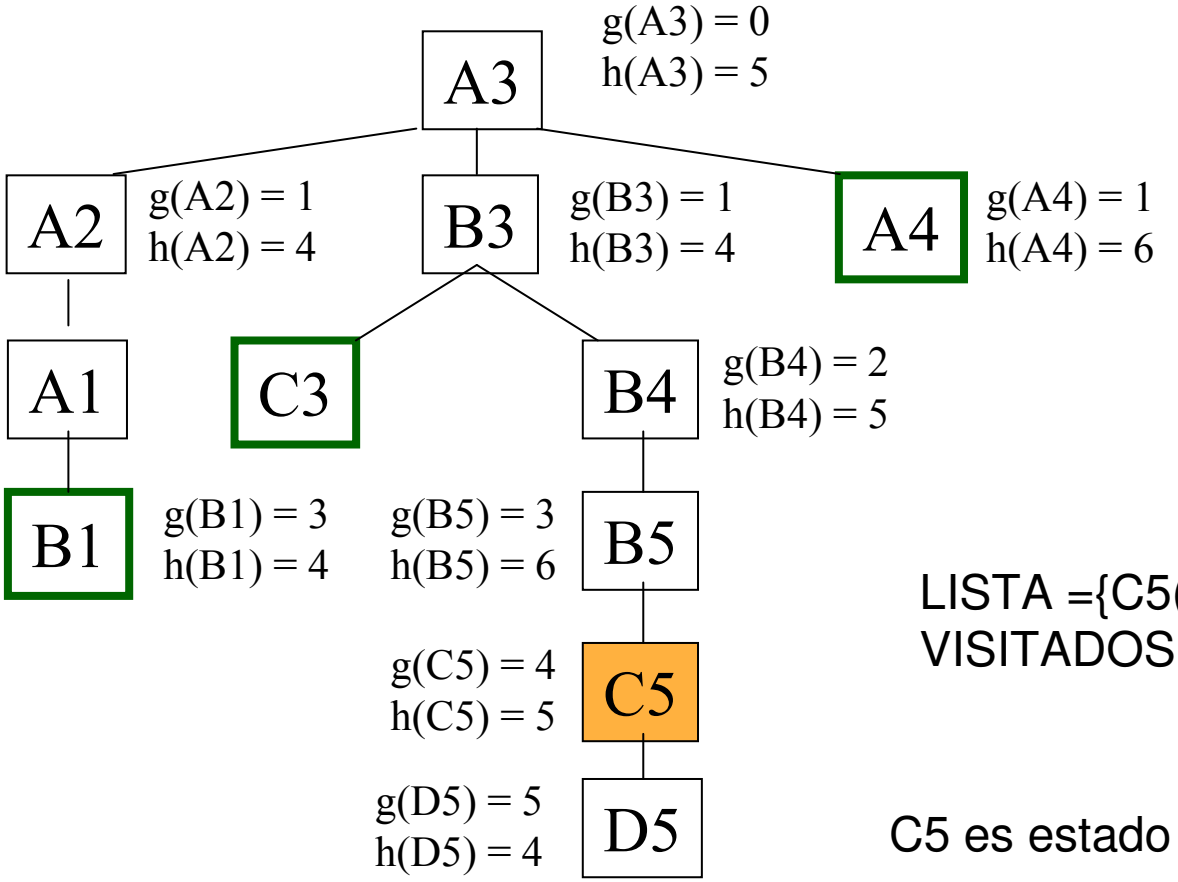


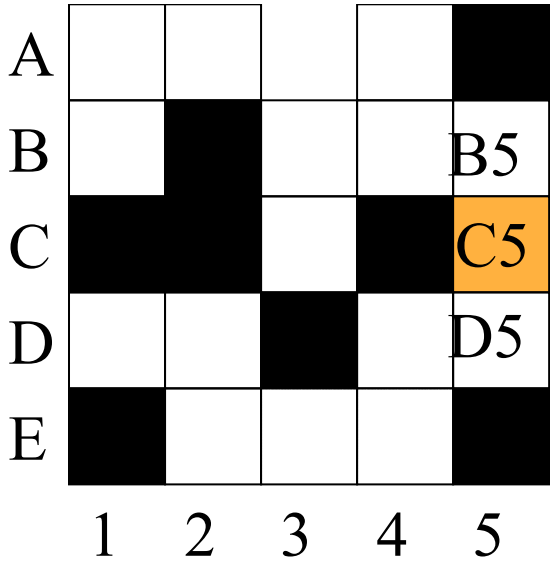
Generamos sucesores: B4; C5

Para ordenar los nodos, hay que calcular f(n)

LISTA ={C5(9)}
VISITADOS = {S(5); A2(5); B3(5);
C3(5); A1(7); B1(7);
B4(7); A4(7); B5(9)}

C5 es estado actual





LISTA ={C5(9)}
VISITADOS = {S(5); A2(5); B3(5); C3(5);
A1(7); B1(7); B4(7); A4(7);
B5(9)}

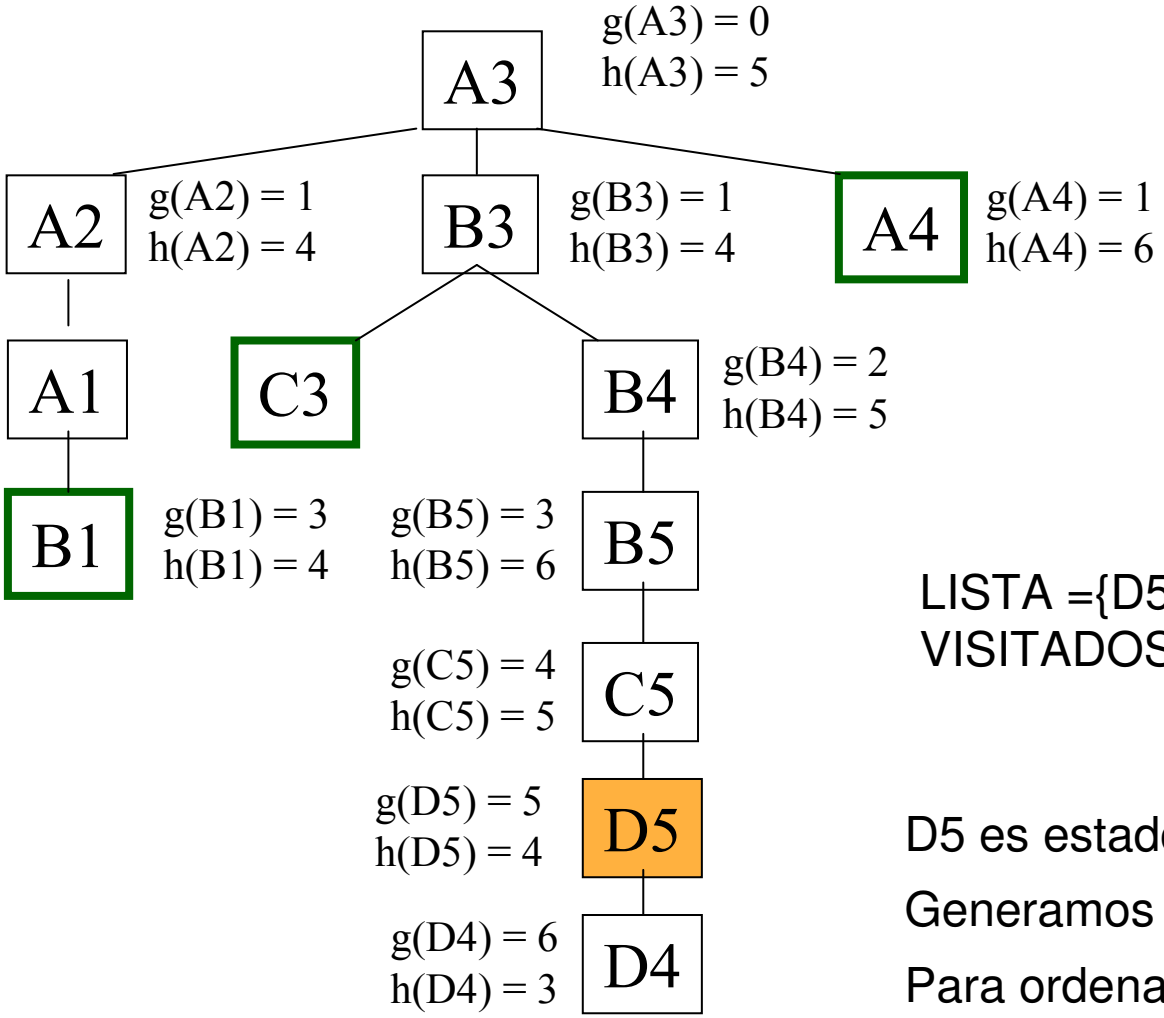
C5 es estado actual, no es estado meta

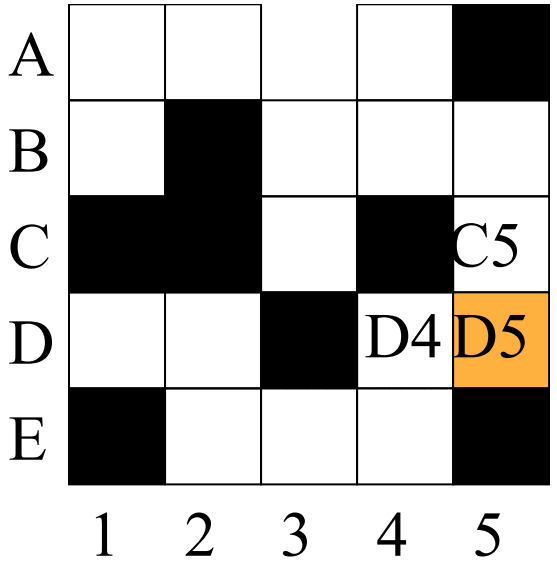
Generamos sucesores: B5; D5

Para ordenar los nodos, hay que calcular f(n)

LISTA ={D5(9)}
VISITADOS = {S(5); A2(5); B3(5); C3(5);
A1(7); B1(7); B4(7); A4(7);
B5(9); C5(9)}

D5 es estado actual





LISTA ={D5(9)}
VISITADOS = {S(5); A2(5); B3(5); C3(5);
A1(7); B1(7); B4(7); A4(7);
B5(9); C5(9)}

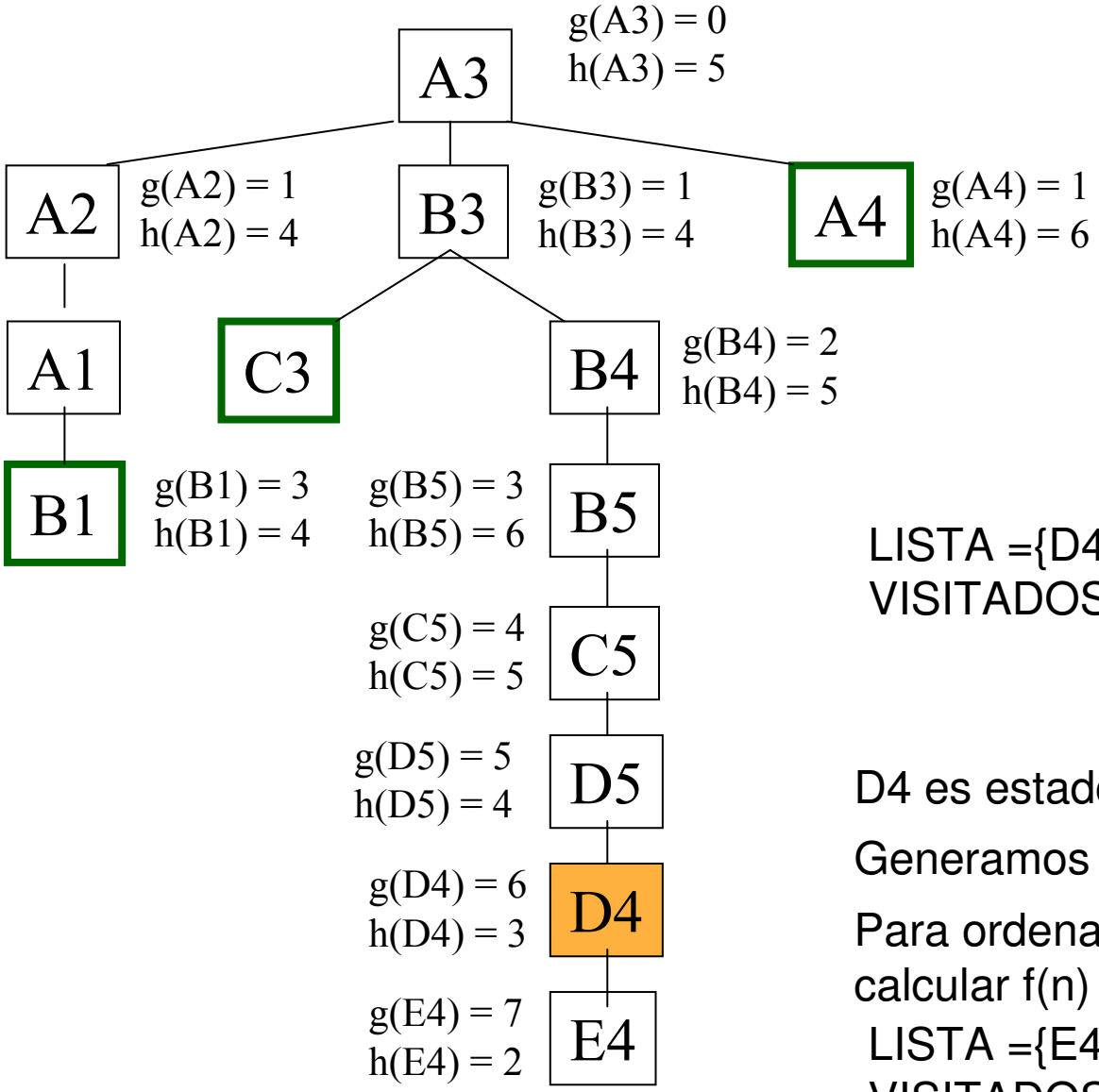
D5 es estado actual, no es estado meta

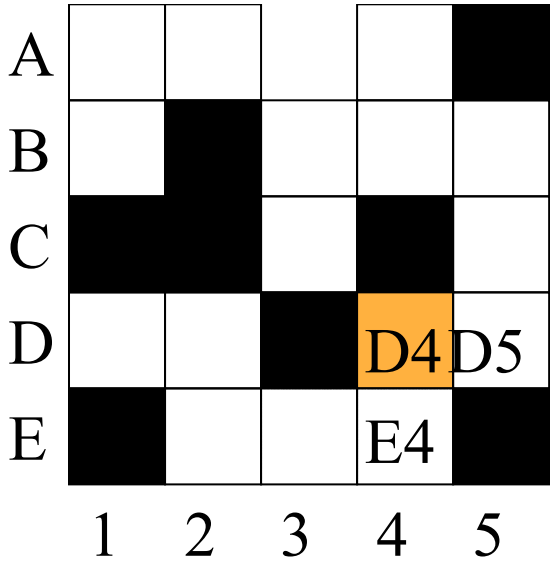
Generamos sucesores: C5; D4

Para ordenar los nodos, hay que calcular f(n)

LISTA ={D4(9)}
VISITADOS = {S(5); A2(5); B3(5); C3(5);
A1(7); B1(7); B4(7); A4(7);
B5(9); C5(9); D5(9)}

D4 es estado actual





LISTA ={D4(9)}
VISITADOS = {S(5); A2(5); B3(5); C3(5);
A1(7); B1(7); B4(7); A4(7);
B5(9); C5(9); D5(9)}

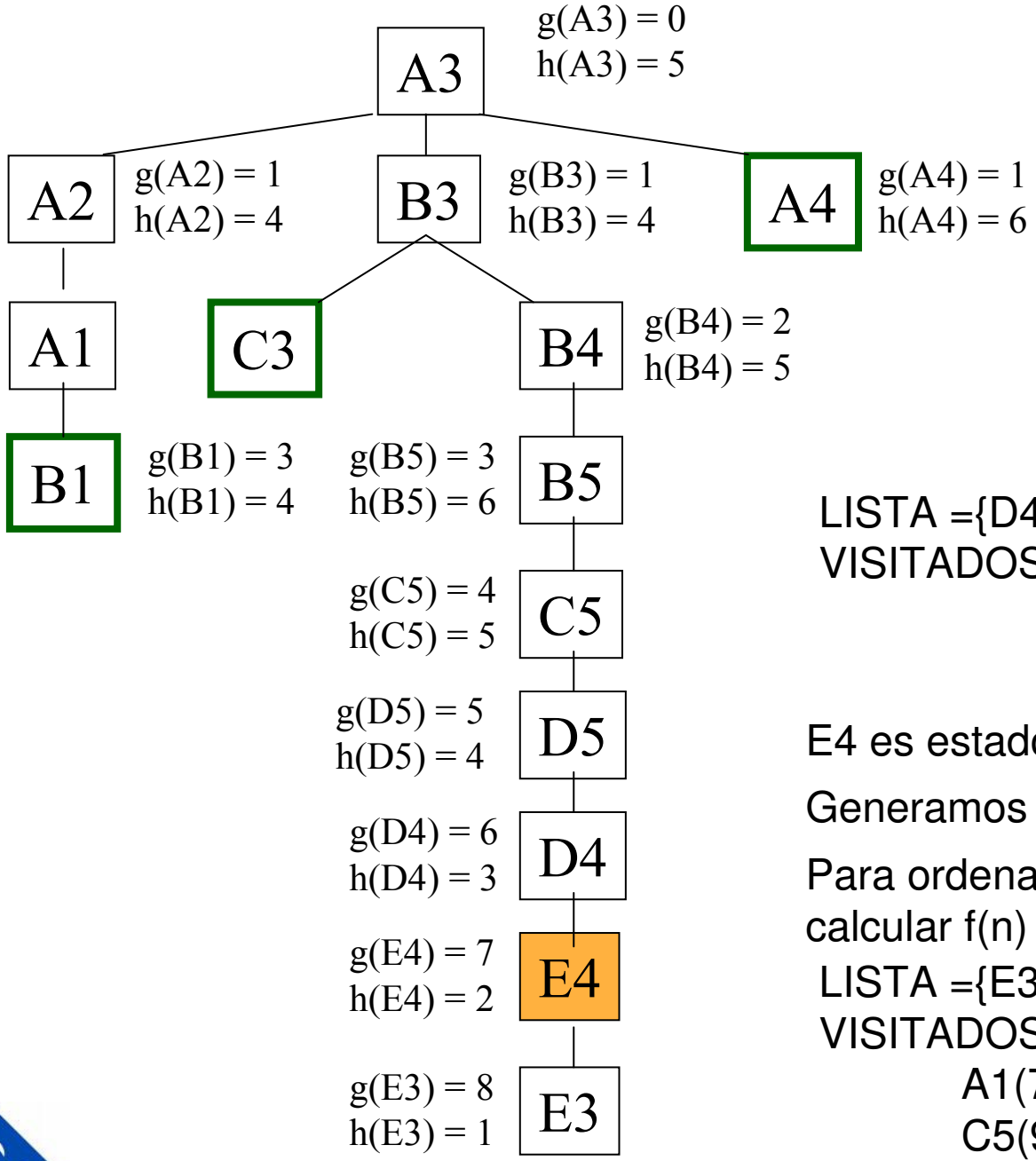
D4 es estado actual, no es estado meta

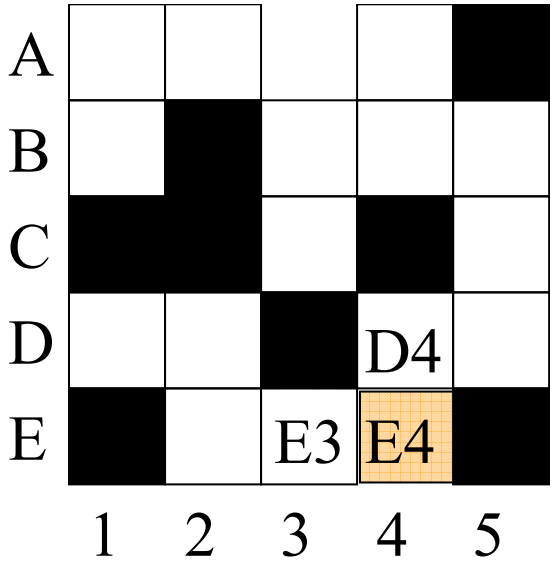
Generamos sucesores: D5; E4

Para ordenar los nodos, hay que calcular f(n)

LISTA ={E4(9)}
VISITADOS = {S(5); A2(5); B3(5); C3(5);
A1(7); B1(7); B4(7); A4(7);
B5(9); C5(9); D5(9); D4(9)}

E4 es estado actual





LISTA ={D4(9)}
VISITADOS = {S(5); A2(5); B3(5); C3(5);
A1(7); B1(7); B4(7); A4(7);
B5(9); C5(9); D5(9)}

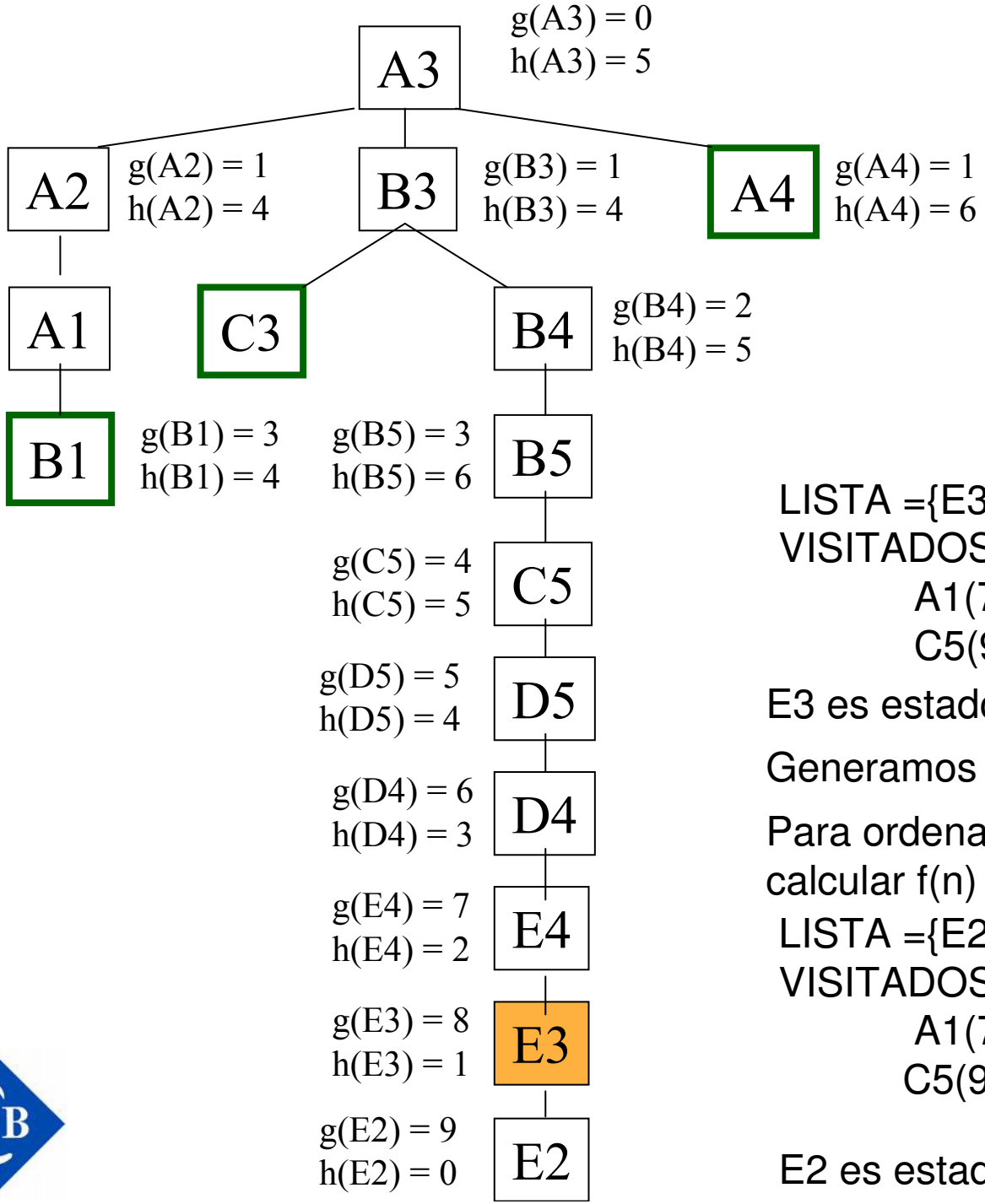
E4 es estado actual, no es estado meta

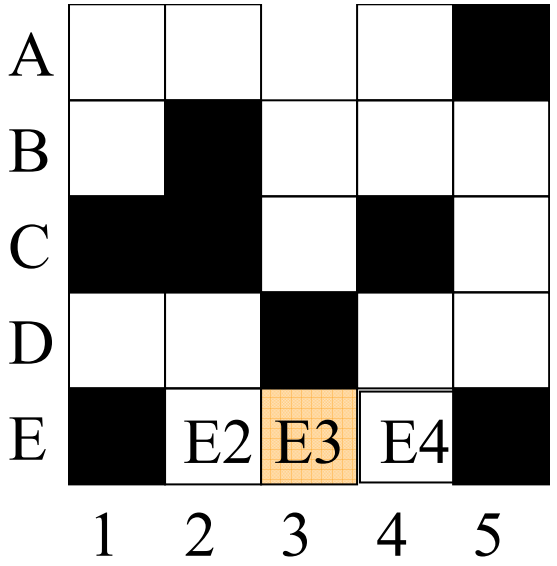
Generamos sucesores: D4; E3

Para ordenar los nodos, hay que calcular f(n)

LISTA ={E3(9)}
VISITADOS = {S(5); A2(5); B3(5); C3(5);
A1(7); B1(7); B4(7); A4(7); B5(9);
C5(9); D5(9); D4(9); E4(9)}

E3 es estado actual





LISTA ={E3(9)}
VISITADOS = {S(5); A2(5); B3(5); C3(5);
A1(7); B1(7); B4(7); A4(7); B5(9);
C5(9); D5(9); D4(9); E4(9)}

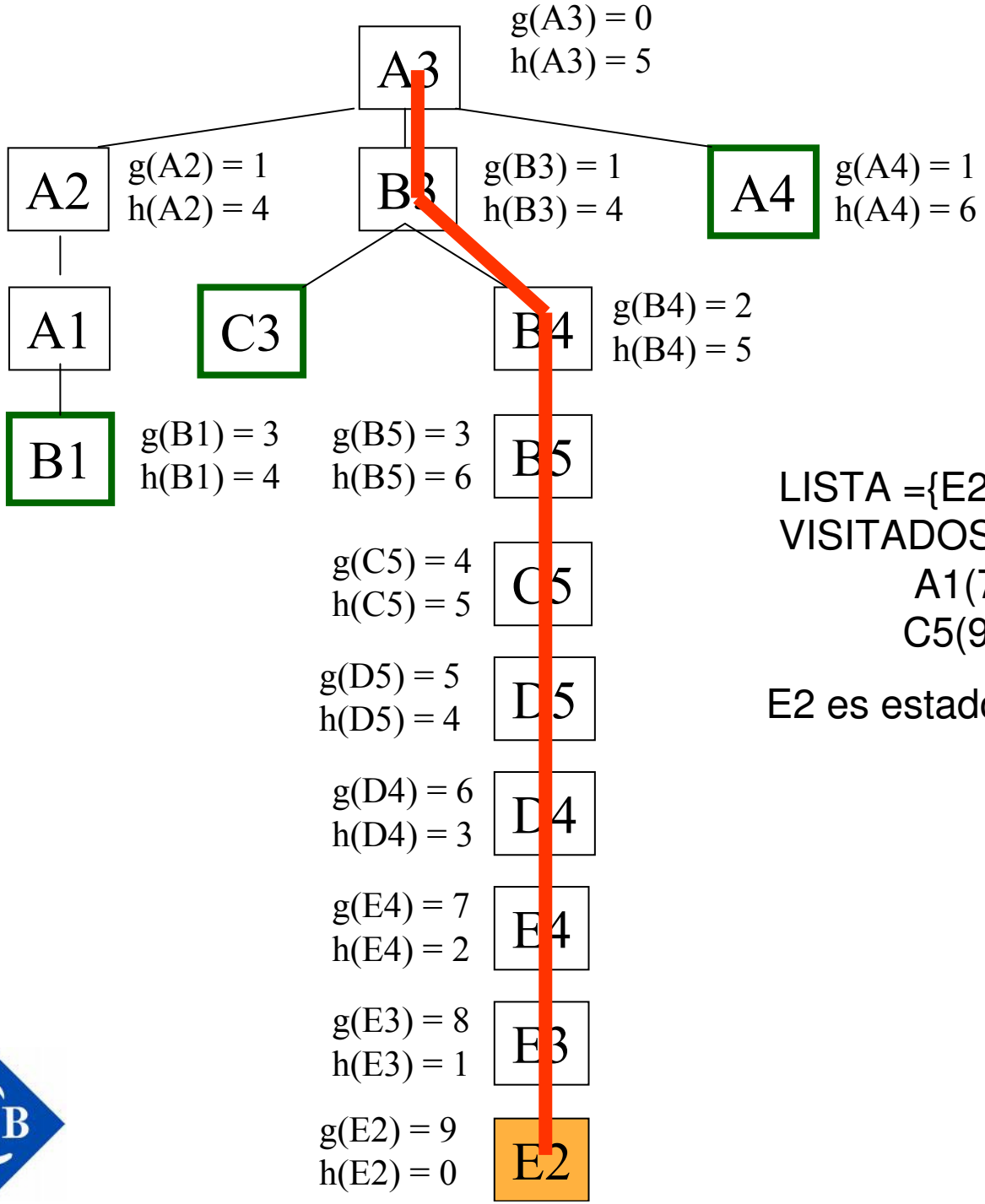
E3 es estado actual, no es estado meta

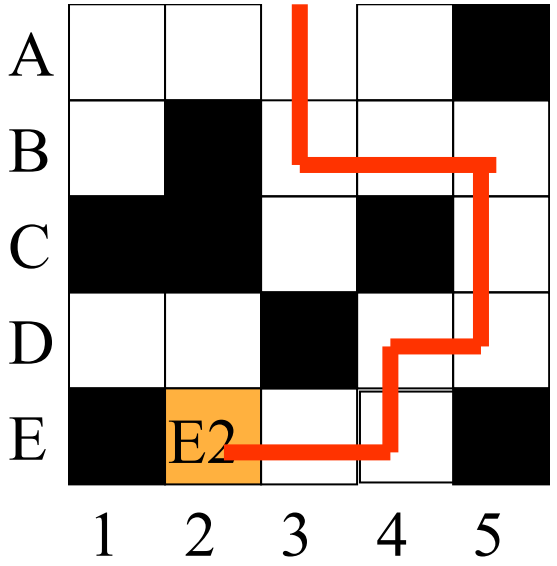
Generamos sucesores: E4; E2

Para ordenar los nodos, hay que calcular f(n)

LISTA ={E2(9)}
VISITADOS = {S(5); A2(5); B3(5); C3(5);
A1(7); B1(7); B4(7); A4(7); B5(9);
C5(9); D5(9); D4(9); E4(9); E3(9)}

E2 es estado actual



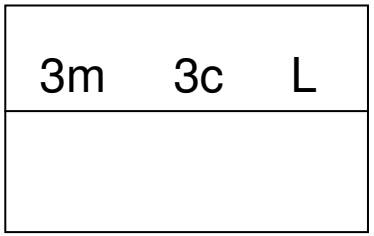


LISTA ={E2(9)}
VISITADOS = {S(5); A2(5); B3(5); C3(5);
A1(7); B1(7); B4(7); A4(7); B5(9);
C5(9); D5(9); D4(9); E4(9); E3(9)}

E2 es estado actual, ES ESTADO META

Misioneros y caníbales

- Estados:
  - número misioneros y caníbales en cada orilla, posición bote
  - $[M_i, C_i, L, M_d, C_d]$
    $M_i, C_i, M_d, C_d \in \{0, 1, 2, 3\}; L \in \{i, d\}$
- Operadores:
  - Transportar 1 misionero y 1 caníbal
  - Transportar 1 misionero
  - Transportar 1 caníbal
  - Transportar 2 misioneros
  - Transportar 2 caníbales
- Coste operador: 1



Condición 1
Debe de haber personas para transportarlas

Condición 2
Posición de la lancha

Condición 3
Evitar que los caníbales se coman a los misioneros

Mover de la orilla x a la orilla y

Operador	Condición 1	Condición 2	Condición 3
Mover1C(x,y)	Cx ≥ 1	L = x	My ≥ Cy + 1 o My = 0
Mover2C(x,y)	Cx ≥ 2	L = x	My ≥ Cy + 2 o My = 0
Mover1M(x,y)	Mx ≥ 1	L = x	(Mx ≥ Cx + 1 o Mx = 1) y My ≥ Cy – 1
Mover2M(x,y)	Mx ≥ 2	L = x	(Mx ≥ Cx + 2 o Mx = 2) y My ≥ Cy – 2
Mover1C1M(x,y)	Cx ≥ 1, Mx ≥ 1	L = x	Mx ≥ Cx