Universidad
Francisco de Vitoria
UFV Madrid

Ingeniería del Conocimiento

Tema 3:
Búsqueda NO Informada

Departamento de Ingeniería Informática
Universidad Francisco de Vitoria

Roberto Rodriguez Galán
Curso 2021/22

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

Contenido

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*)

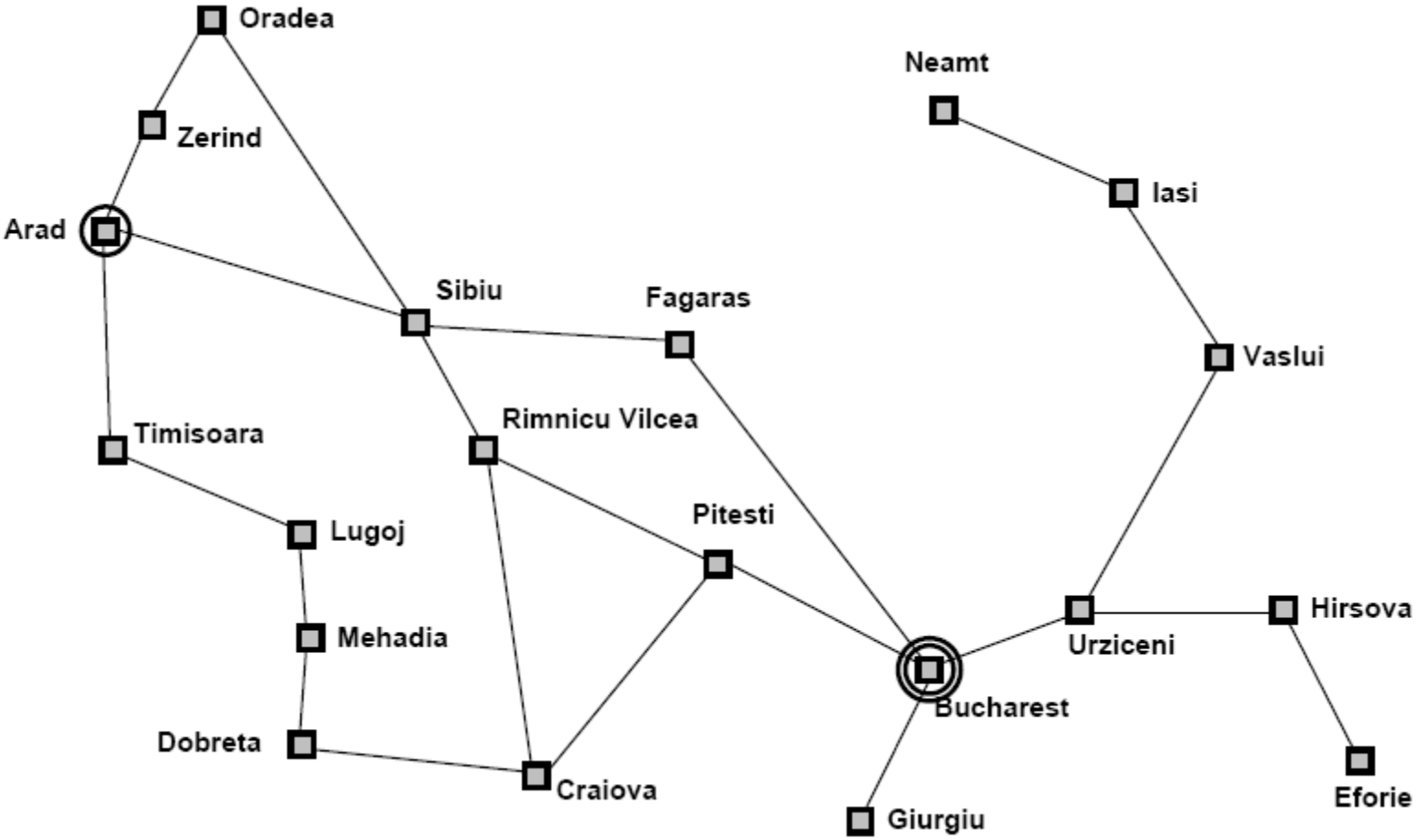
1. Introducción

- 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*"

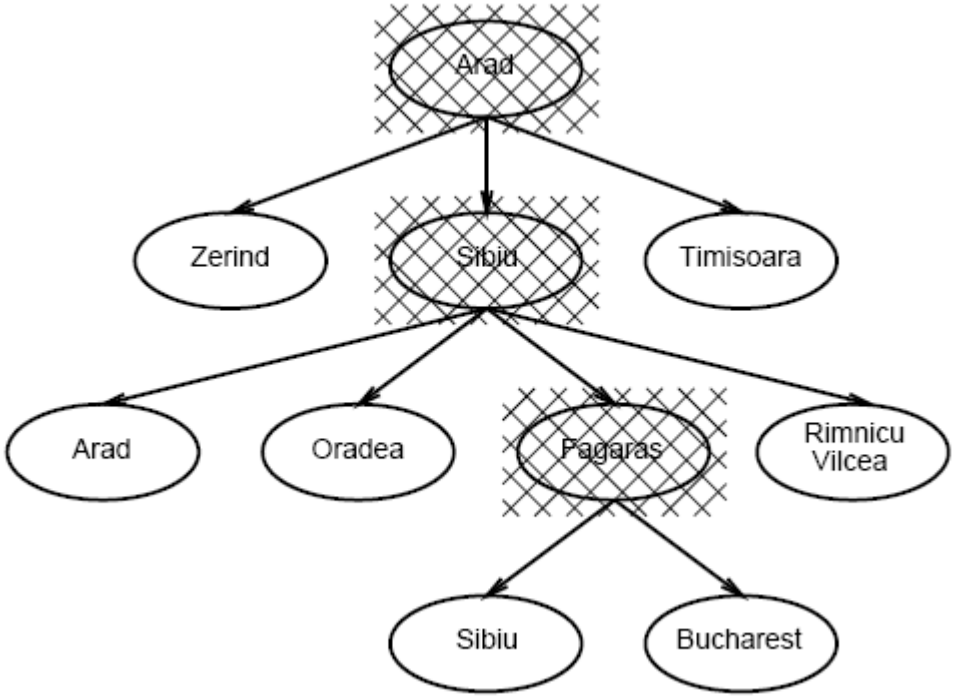
- Definición de problema (*problema bien definido*):
  - Estado inicial
  - Descripción de acciones posibles
  - Test Objetivo, que determina si el estado es objetivo
  - Función de Coste del camino

- 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 <u>búsqueda no informada en espacio de estados</u>

  - 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 *SUCESORES* vacío
    2. Para cada OPERADOR en *OPERADORES*,
            si SUCESOR(NODO,OPERADOR) ≠ NO-APLICABLE,
                    incluir SUCESOR(NODO,OPERADOR) en *SUCESORES*
    3. Devolver *SUCESORES*
```

```
GESTIONA-COLA(ABIERTO,SUCESORES)
            Insertar *SUCESORES* 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)          (generar sucesores)
         GESTIONA-COLA(ABIERTO,SUCESORES)
FIN DE BUCLE

Devuelve FALLO
```

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 que 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

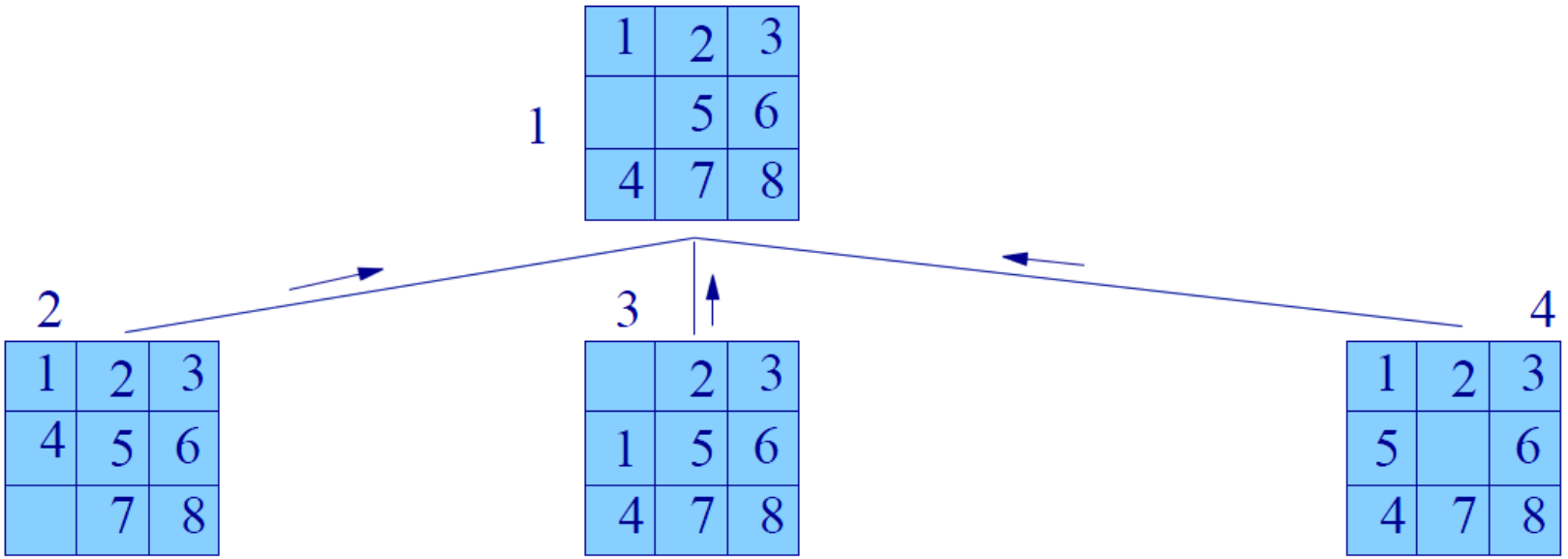
1. Introducción
2. Implementación
3. Métodos no informados
    1. Búsqueda en anchura
    2. Búsqueda de coste uniforme
    3. Búsqueda en profundidad
    4. Búsqueda en profundidad limitada
    5. Búsqueda en profundidad iterativa
    6. Búsqueda bidireccional
4. Complejidad

3. Métodos no informados

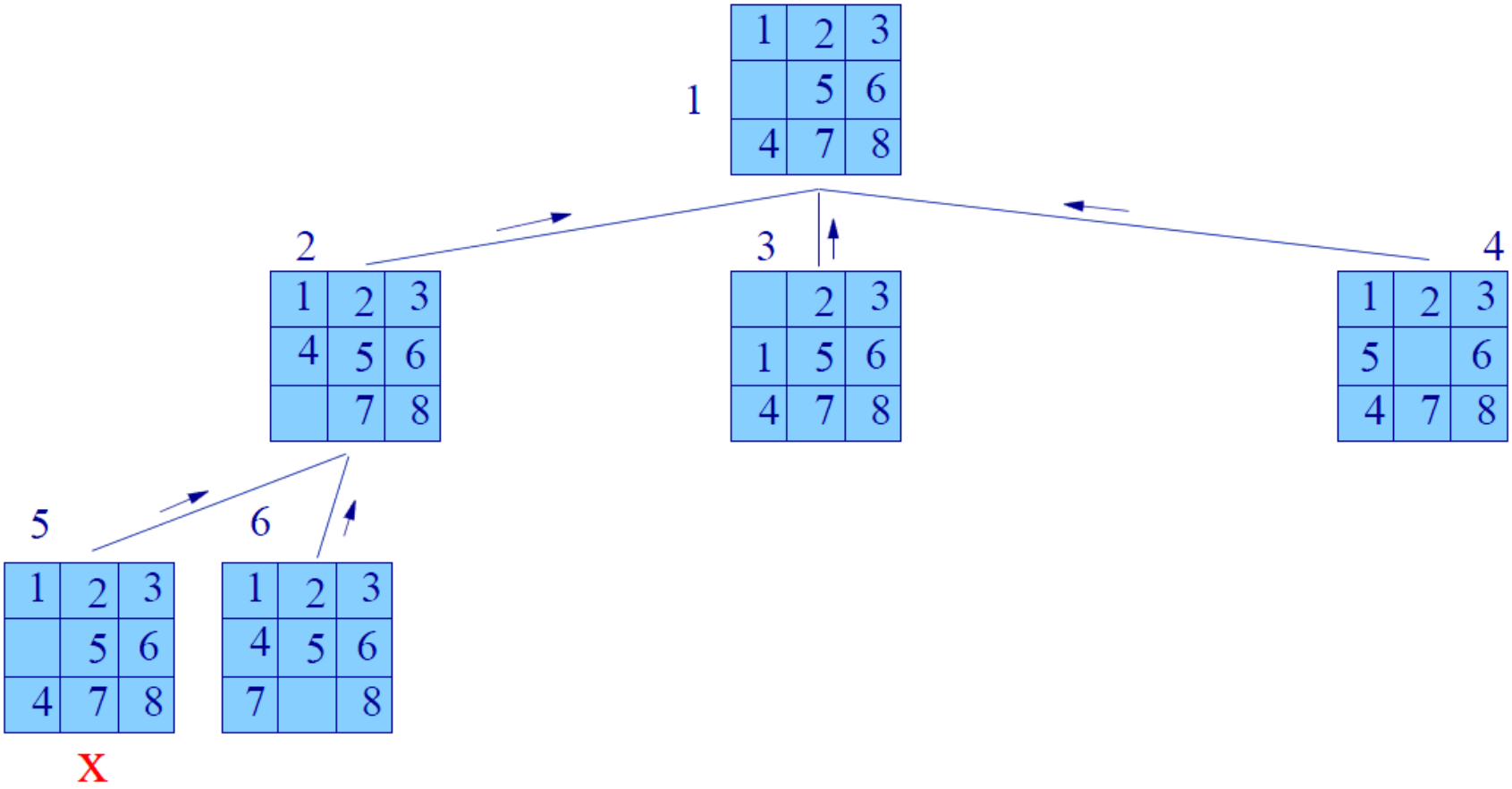
- Para problemas ciegos, se pueden usar varias estrategias según se genere la frontera y en qué orden se exploran los nodos en ella

- 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***)

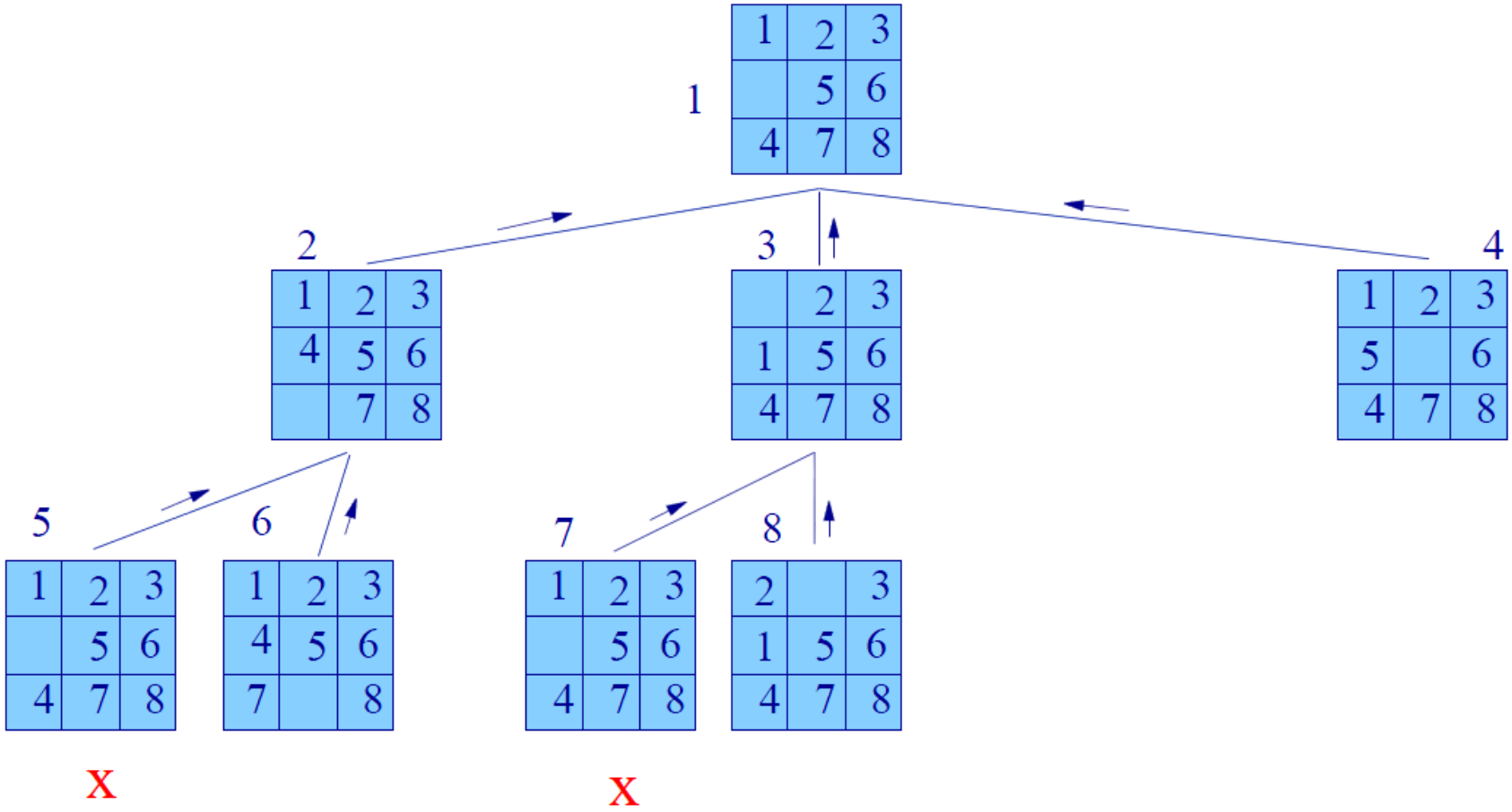
3.1 Búsqueda en anchura



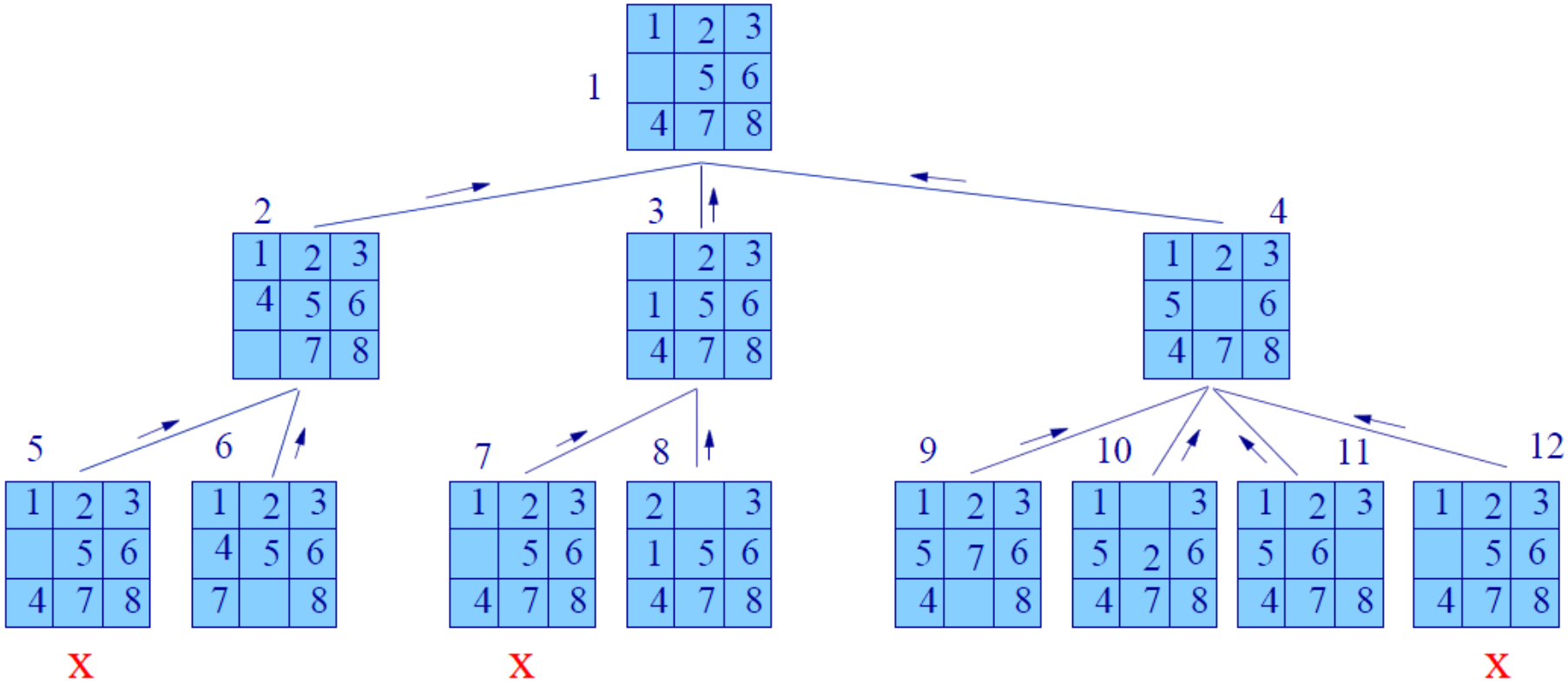
3.1 Búsqueda en anchura



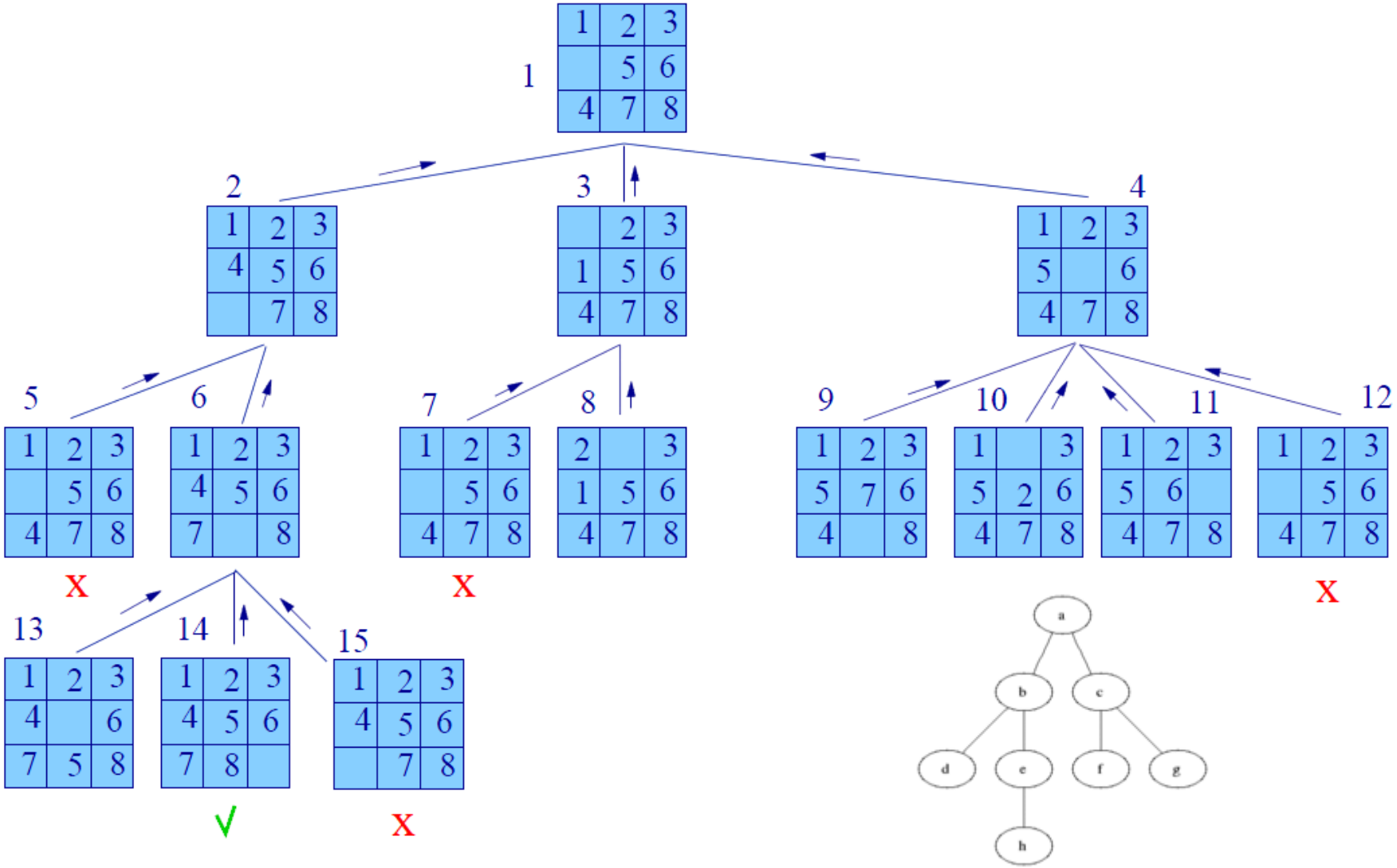
3.1 Búsqueda en anchura



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: $1+ b + b^2 + b^3 + \ldots.+ b^d = O(b^{d+1})$
  - 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:

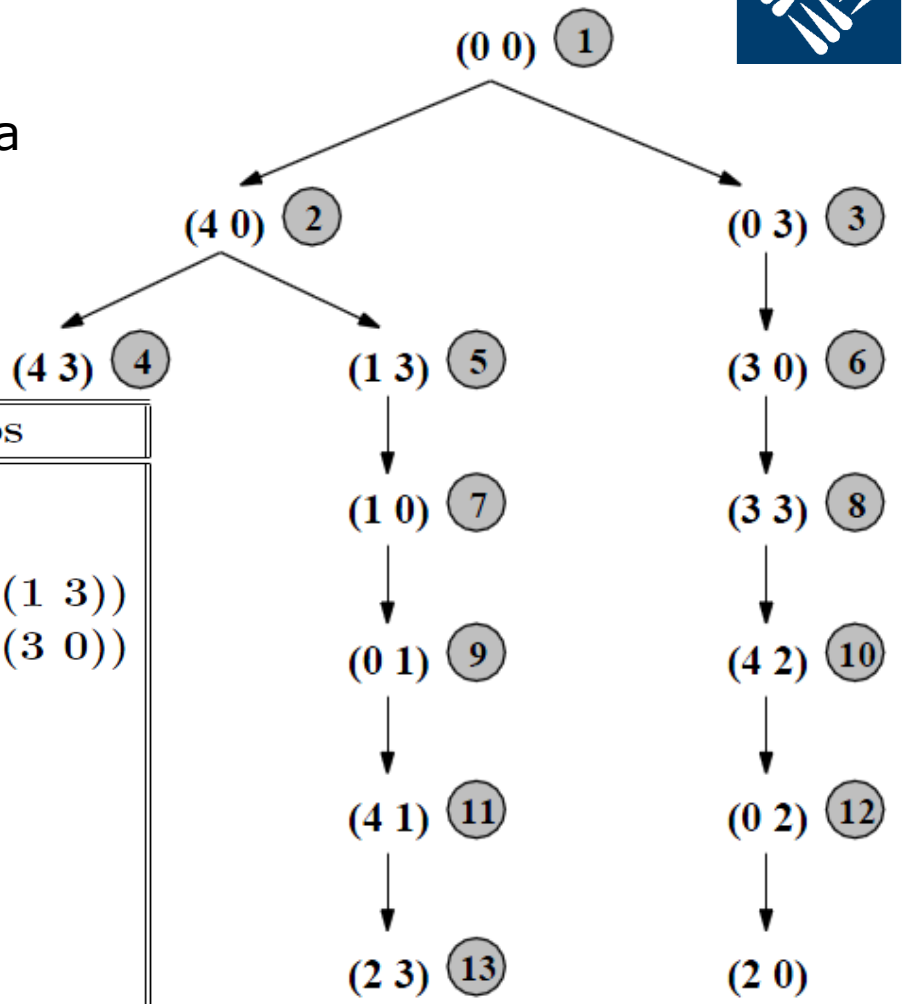
Profundidad	Nodos	Tiempo	Espacio
0	1	1 ms.	100 b
2	111	0.1 seg.	11 Kb
4	11111	11 seg.	1 Mb
6	10^6	18 min.	11 Mb
8	10^8	31 horas	11 Gb
10	10^{10}	128 días	1 Tb
12	10^{12}	33 años	11 Tb
14	10^{14}	3500 años	11.111 Tb

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:



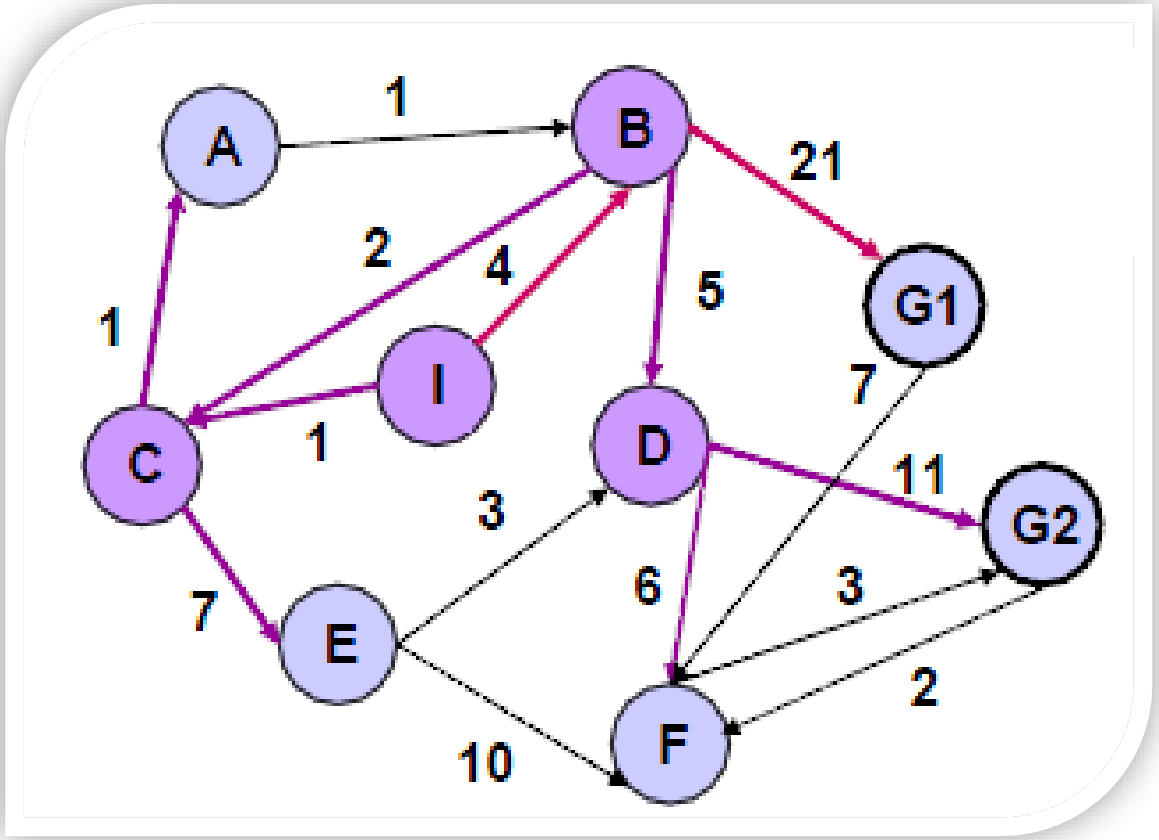
Nodo	Actual	Sucesores	Abiertos
			((0 0))
1	(0 0)	((4 0) (0 3))	((4 0) (0 3))
2	(4 0)	((4 3) (1 3))	((0 3) (4 3) (1 3))
3	(0 3)	((3 0))	((4 3) (1 3) (3 0))
4	(4 3)	()	((1 3) (3 0))
5	(1 3)	((1 0))	((3 0) (1 0))
6	(3 0)	((3 3))	((1 0) (3 3))
7	(1 0)	((0 1))	((3 3) (0 1))
8	(3 3)	((4 2))	((0 1) (4 2))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

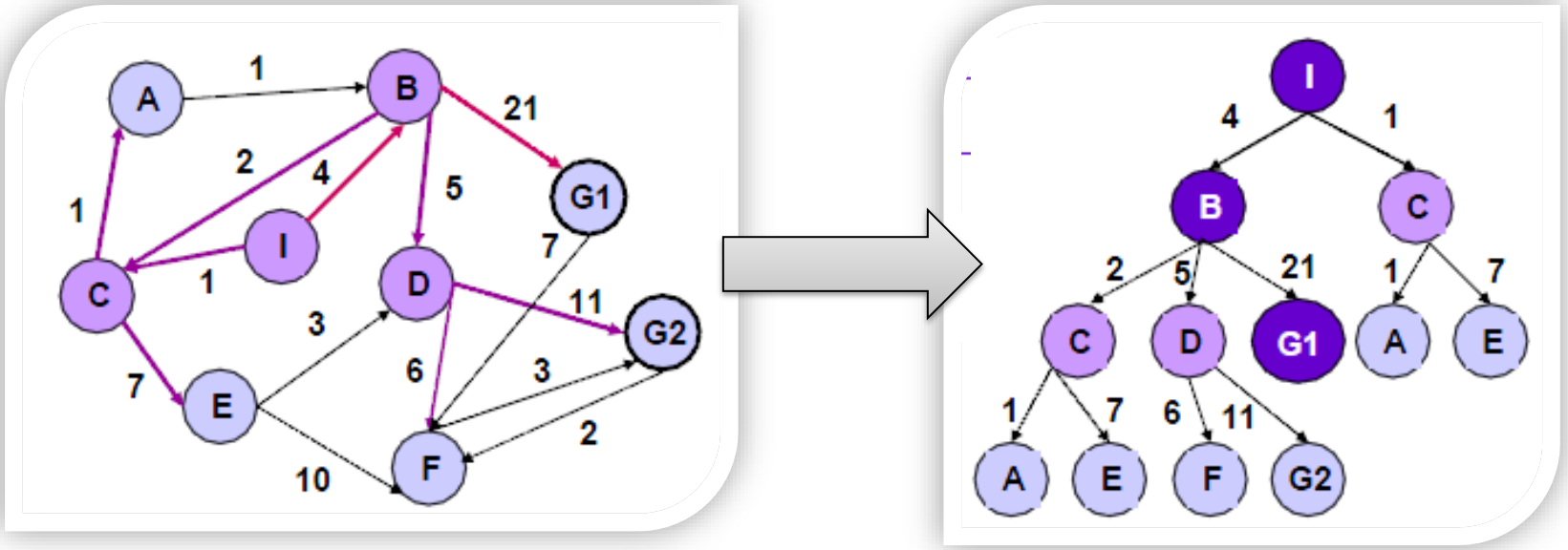
- Estadísticas de casos conocidos

	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



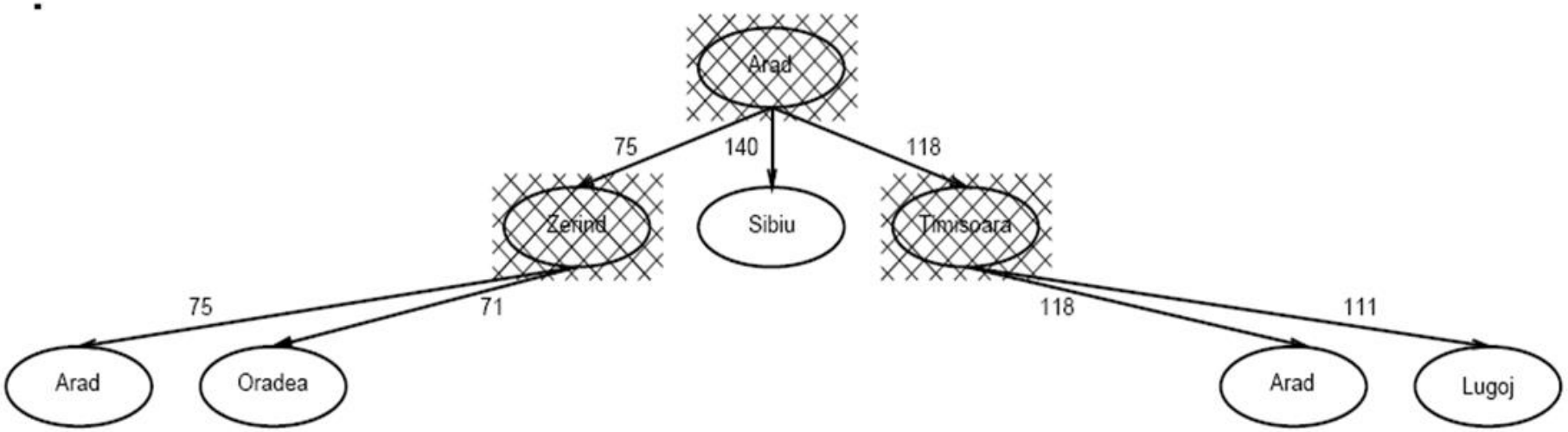
3.1 Búsqueda en anchura



- 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 mas 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

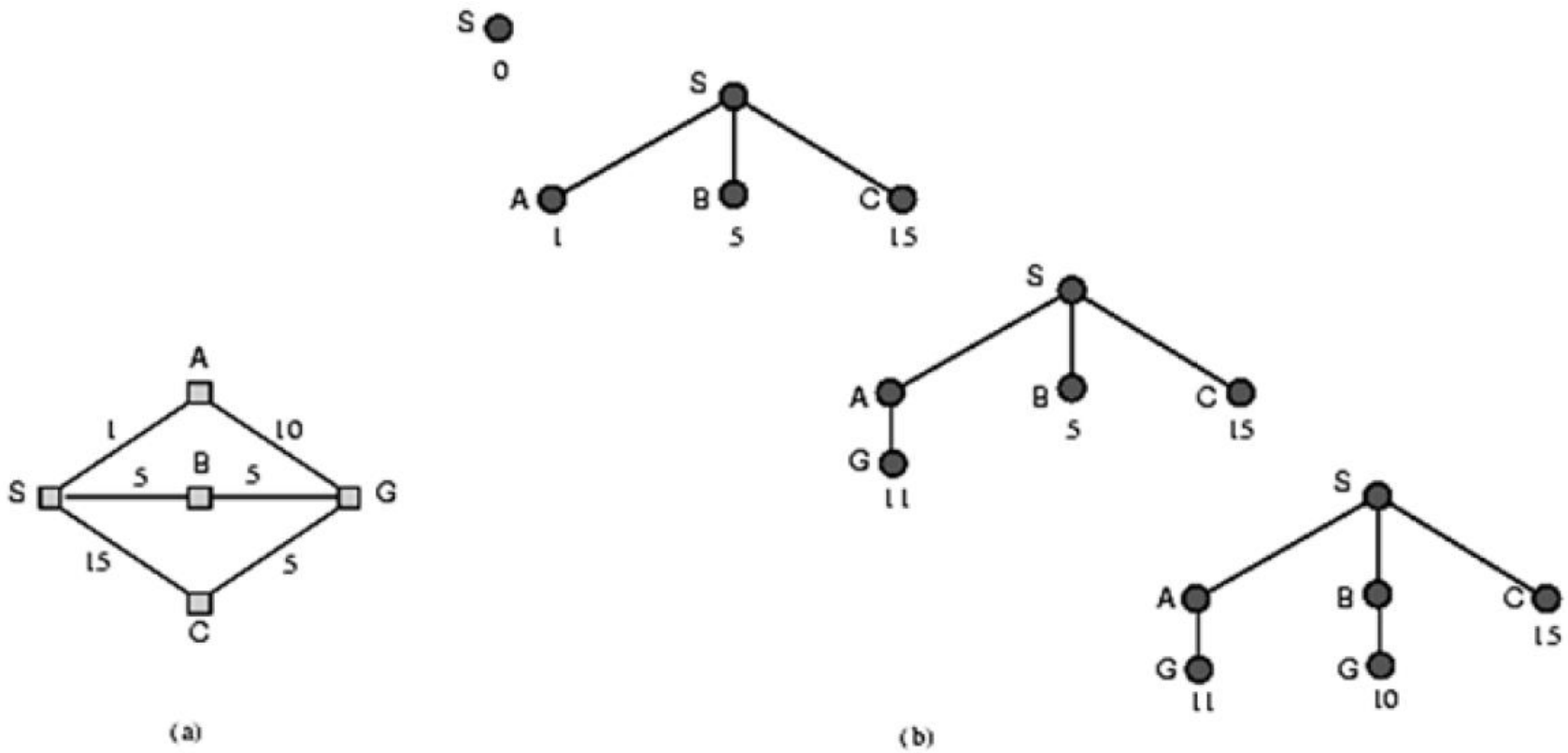
- Propiedades:
  - Completa: **Si**, encuentra la solución de menor coste
  - Complejidad tiempo: $O(b^{C^*/\varepsilon})$ del peor caso
  - Complejidad espacio: $O(b^{C^*/\varepsilon})$ del peor caso
  - Óptima: **Si**, si se cumple que g(SUCESOR(N)) ≥ g(N)
    - C* es el coste de la solución óptima
    - ε es el mínimo coste de acción
    - Problema si ε=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

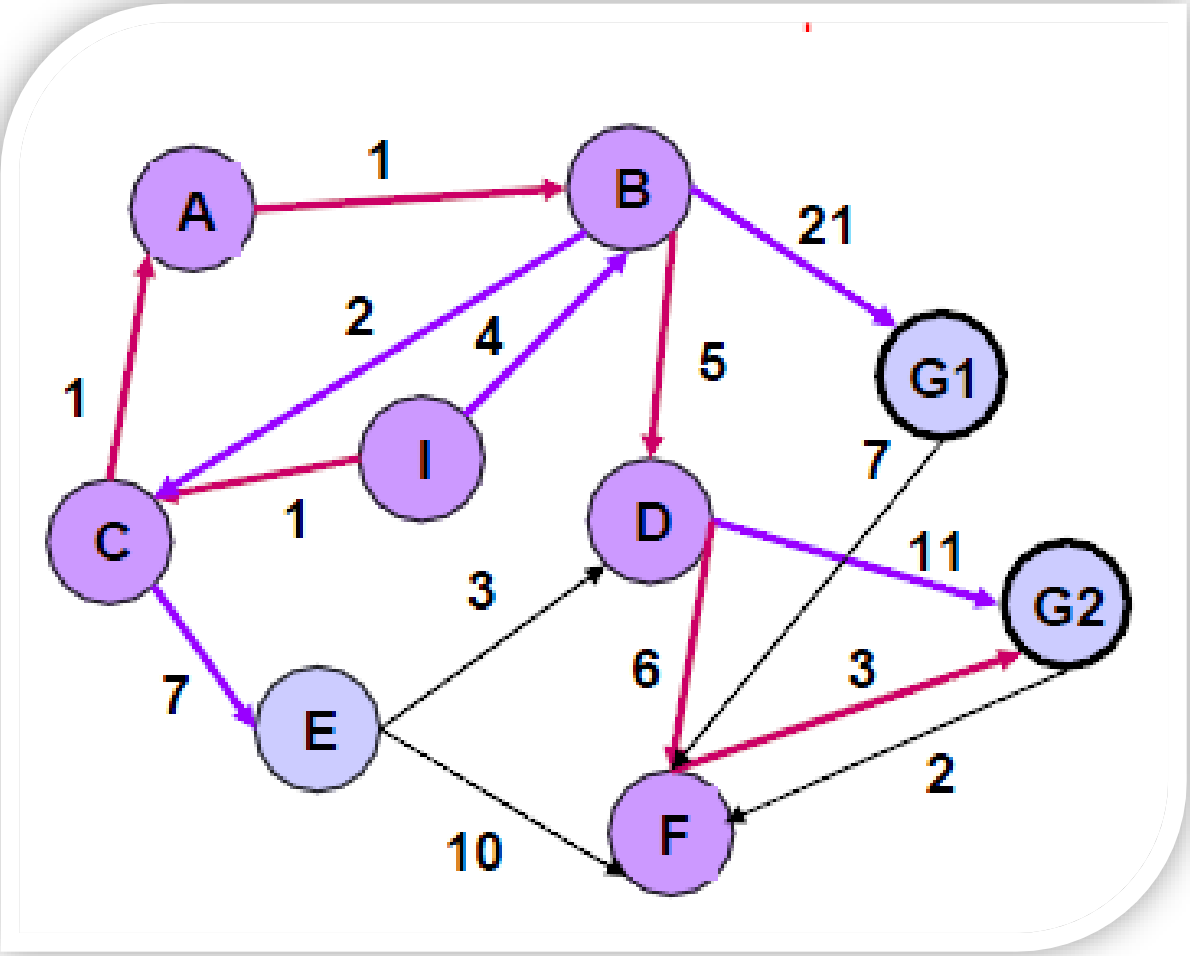
3.2 Búsqueda de coste uniforme

```
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

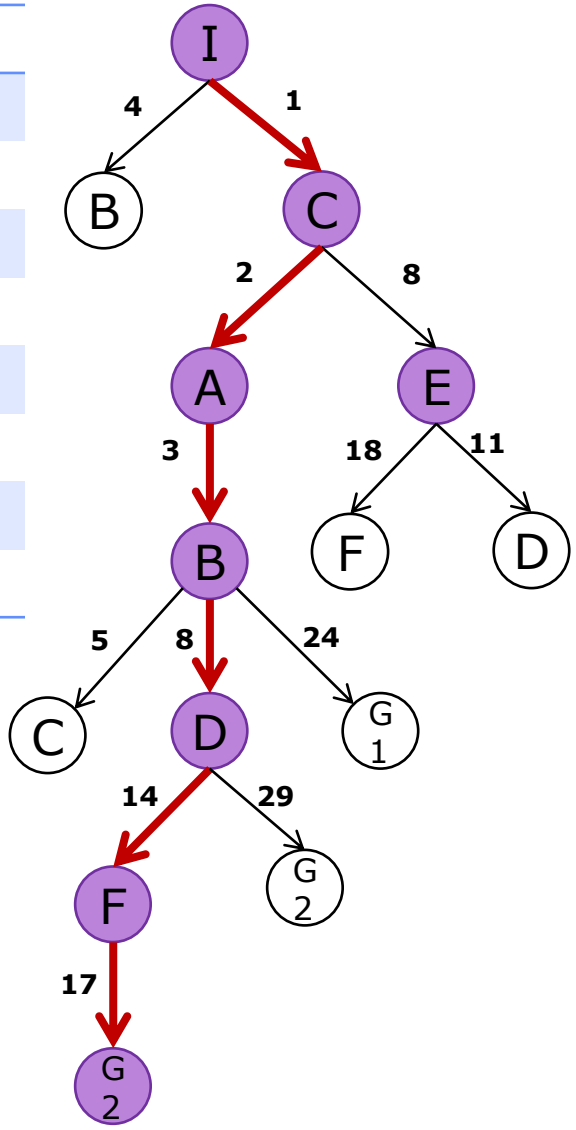


3.2 Búsqueda de coste uniforme



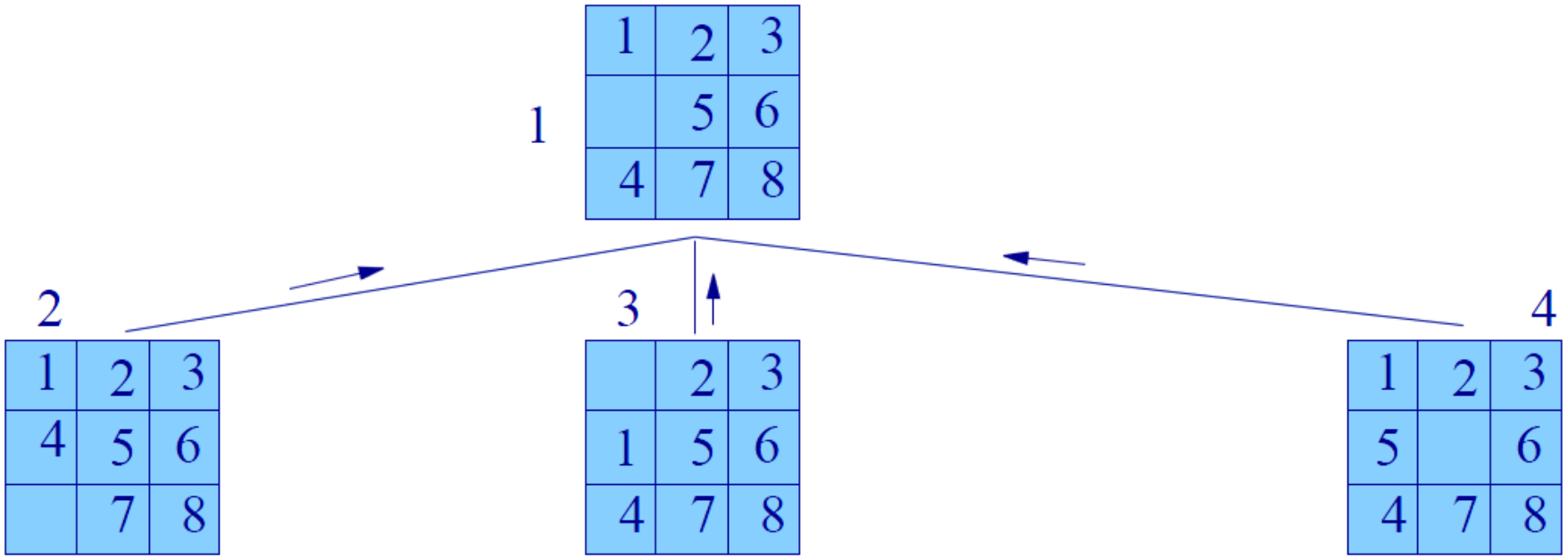
3.2 Búsqueda de coste uniforme

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)	

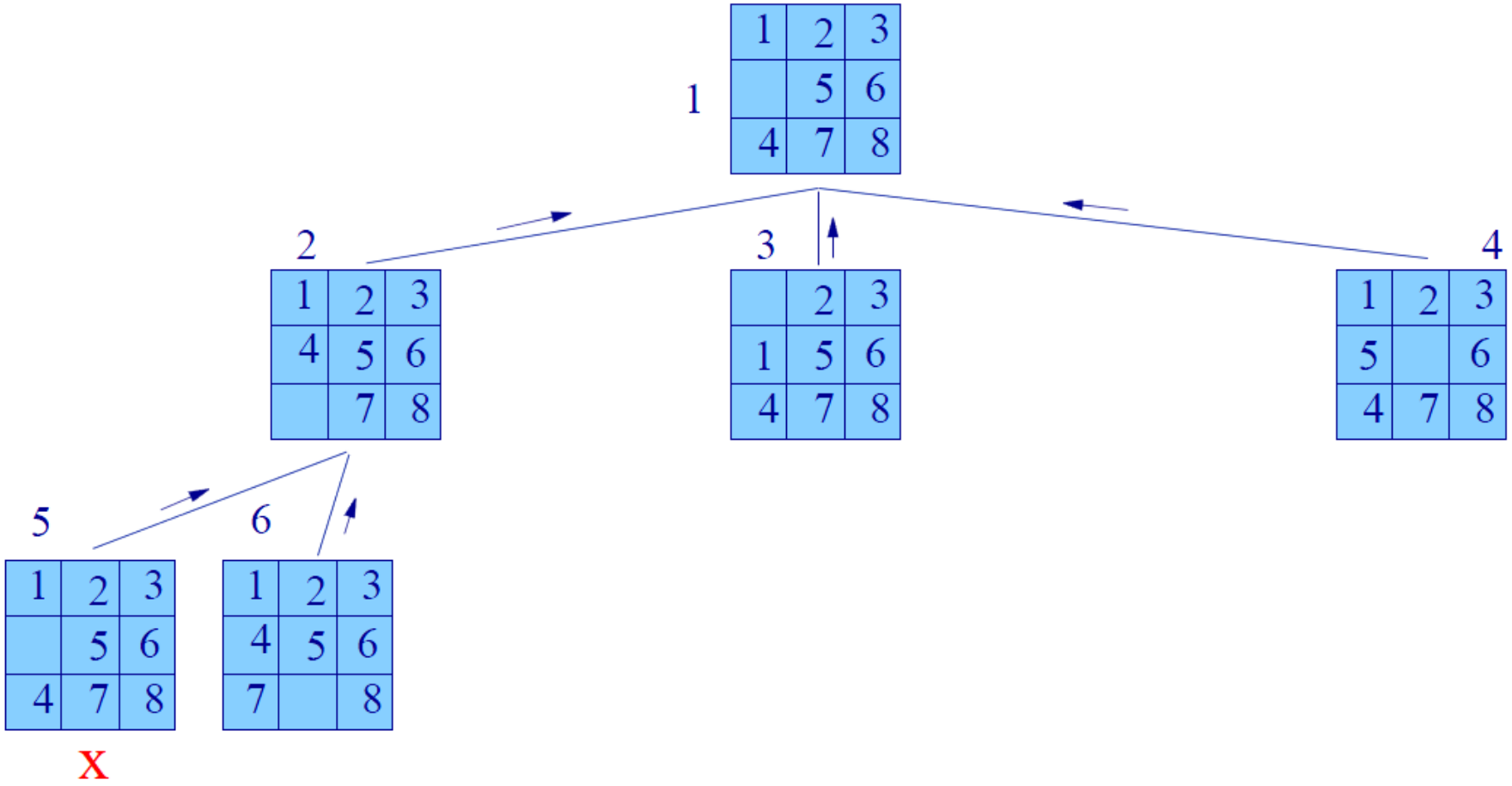


- Camino a la solución: `I C A B D F G2`
- Coste: 1+1+1+5+6+3 = 17

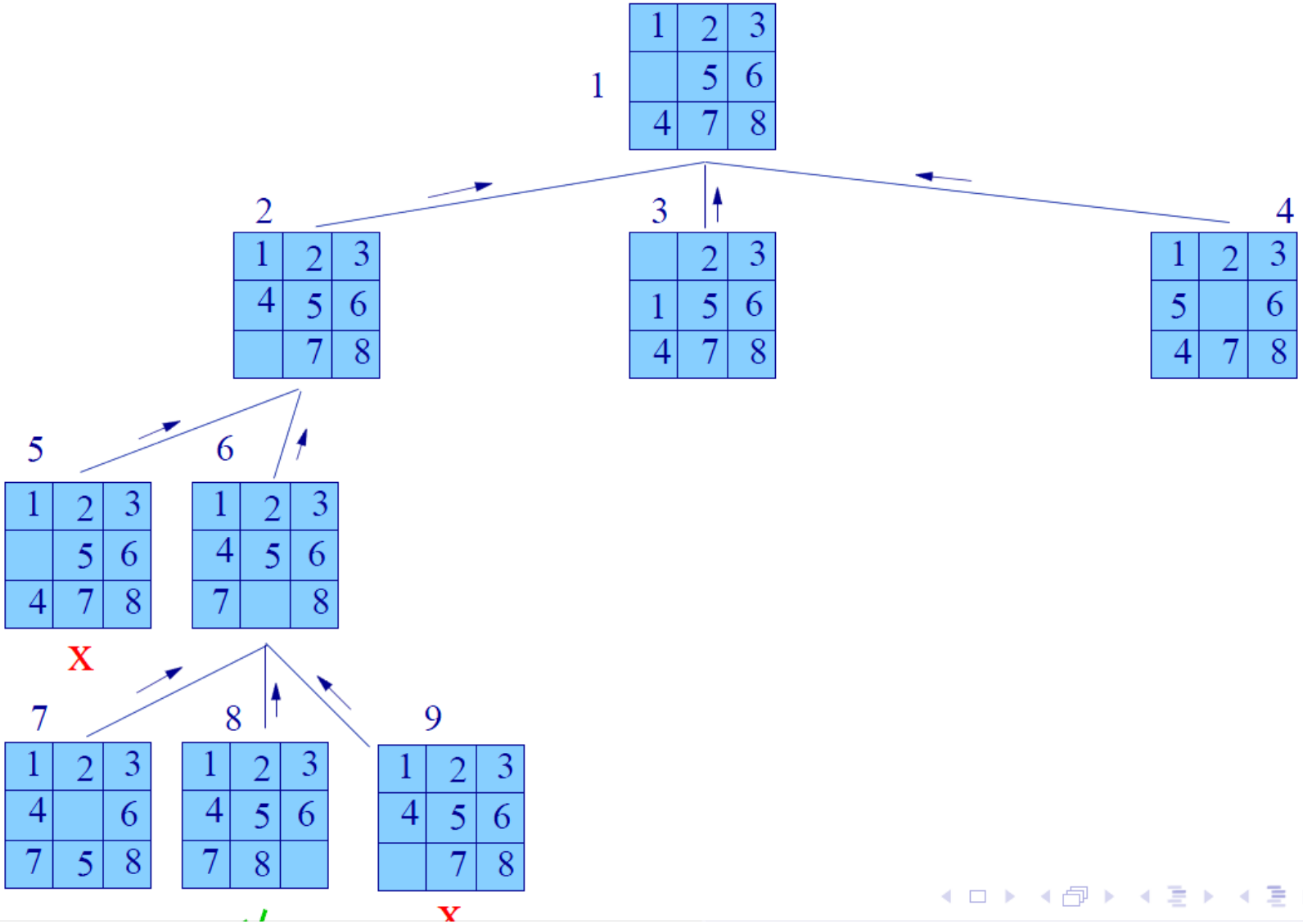
3.3 Búsqueda en profundidad



3.3 Búsqueda en profundidad



3.3 Búsqueda en profundidad

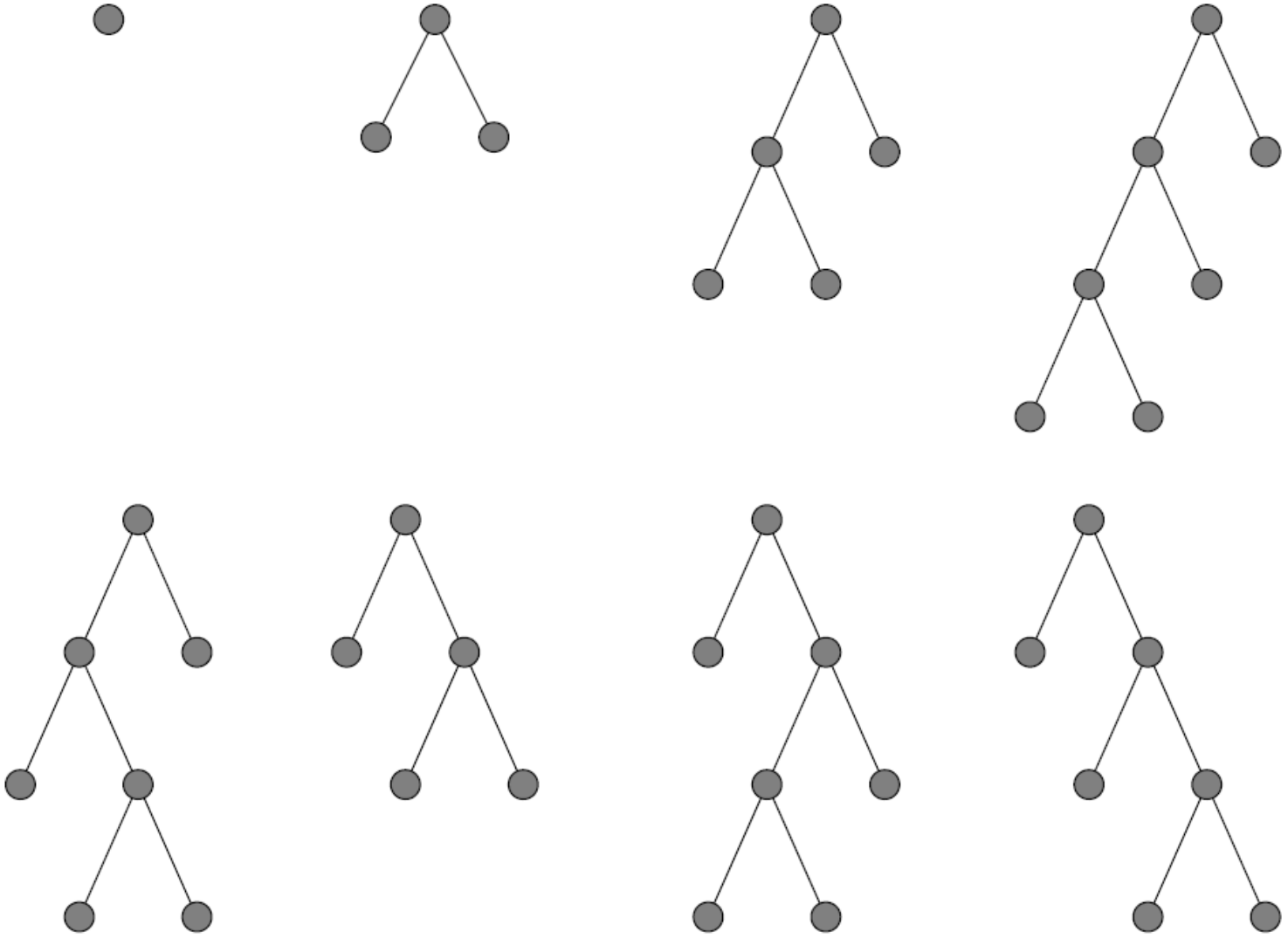


3.3 Búsqueda en profundidad

- 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

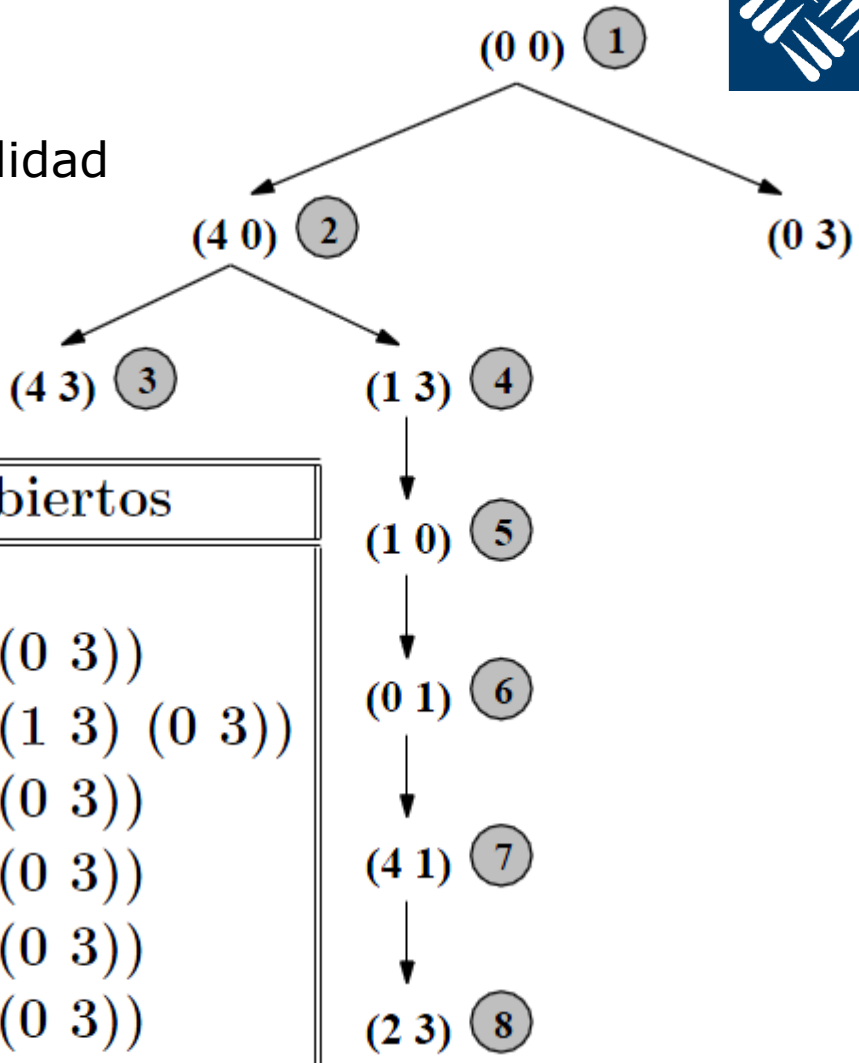
- Resumen
  - Eficiencia: bueno con metas alejadas de estado inicial o problemas de memoria
  - Problema: No es bueno cuando hay ciclos
- Características
  - Requiere técnica de retroceso ("*backtracking*")
  - Razones para retroceso:
    - Se ha llegado al límite de profundidad
    - Se han estudiado 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 ☹
```

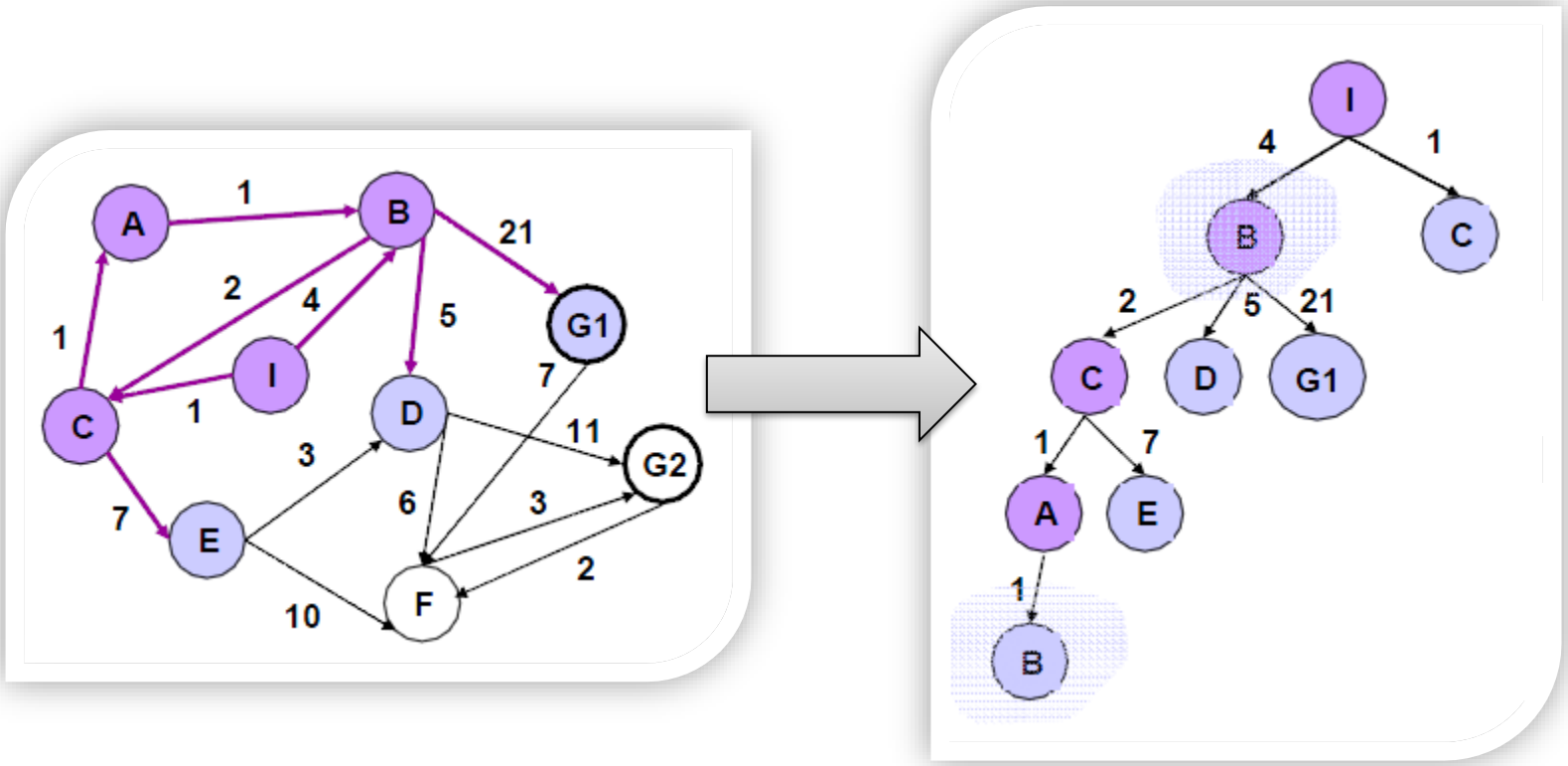
3.3 Búsqueda en profundidad

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



Nodo	Actual	Sucesores	Abiertos
			((0 0))
1	(0 0)	((4 0) (0 3))	((4 0) (0 3))
2	(4 0)	((4 3) (1 3))	((4 3) (1 3) (0 3))
3	(4 3)	()	((1 3) (0 3))
4	(1 3)	((1 0))	((1 0) (0 3))
5	(1 0)	((0 1))	((0 1) (0 3))
6	(0 1)	((4 1))	((4 1) (0 3))
7	(4 1)	((2 3))	((2 3) (0 3))
8	(2 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 cíclicos y no se necesita `CERRADO` para detectar ciclos
  - Para detectar ciclos solo es necesario almacenar los nodos de la rama que se esta explorando en cada momento

3.3 Búsqueda en profundidad

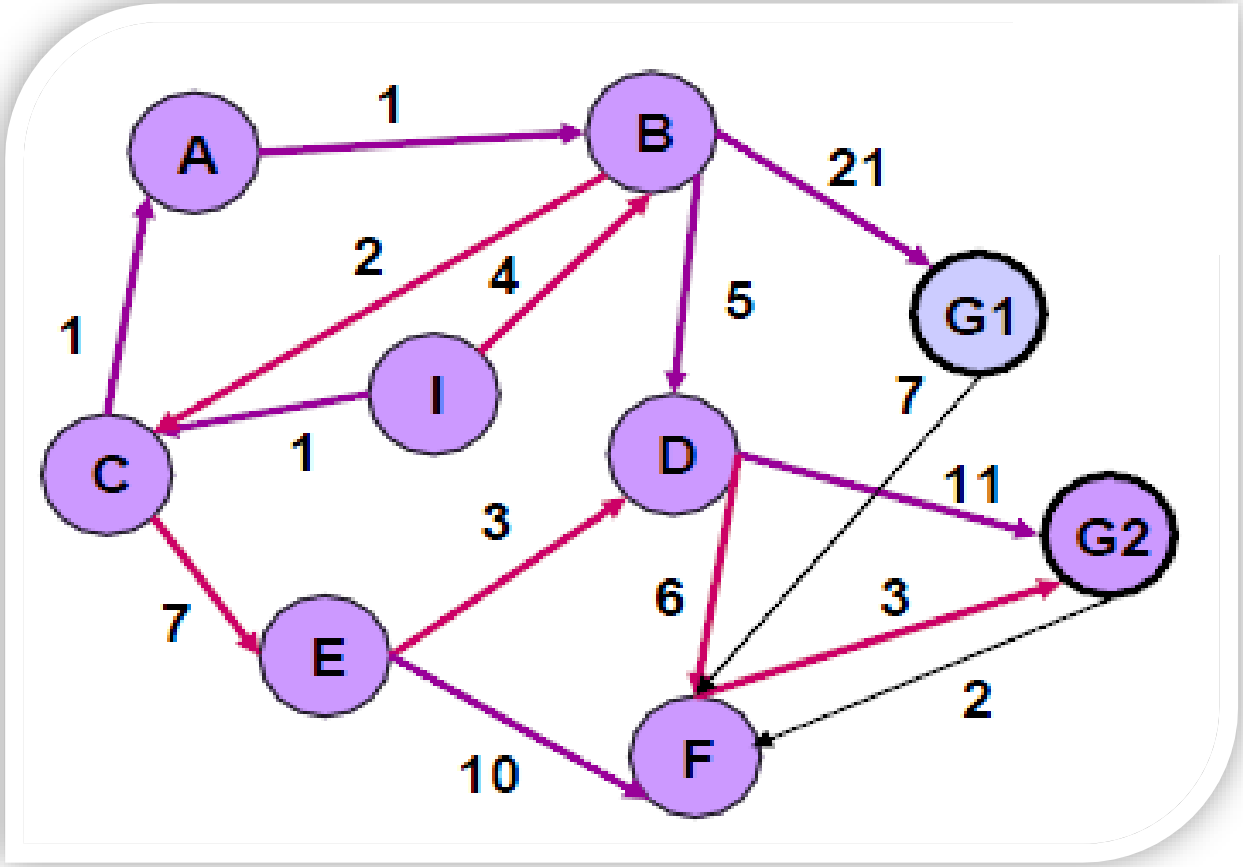
- Mecanismos de control de ciclos
  - Mirar los nodos del camino actual es lo más sencillo y lo menos costoso (cada nodo tiene acceso a su padre)
  - Mirar los nodos de `ABIERTO` (generados sin expandir)
  - Mirar los nodos de `CERRADO` (estados ya expandidos)

- 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

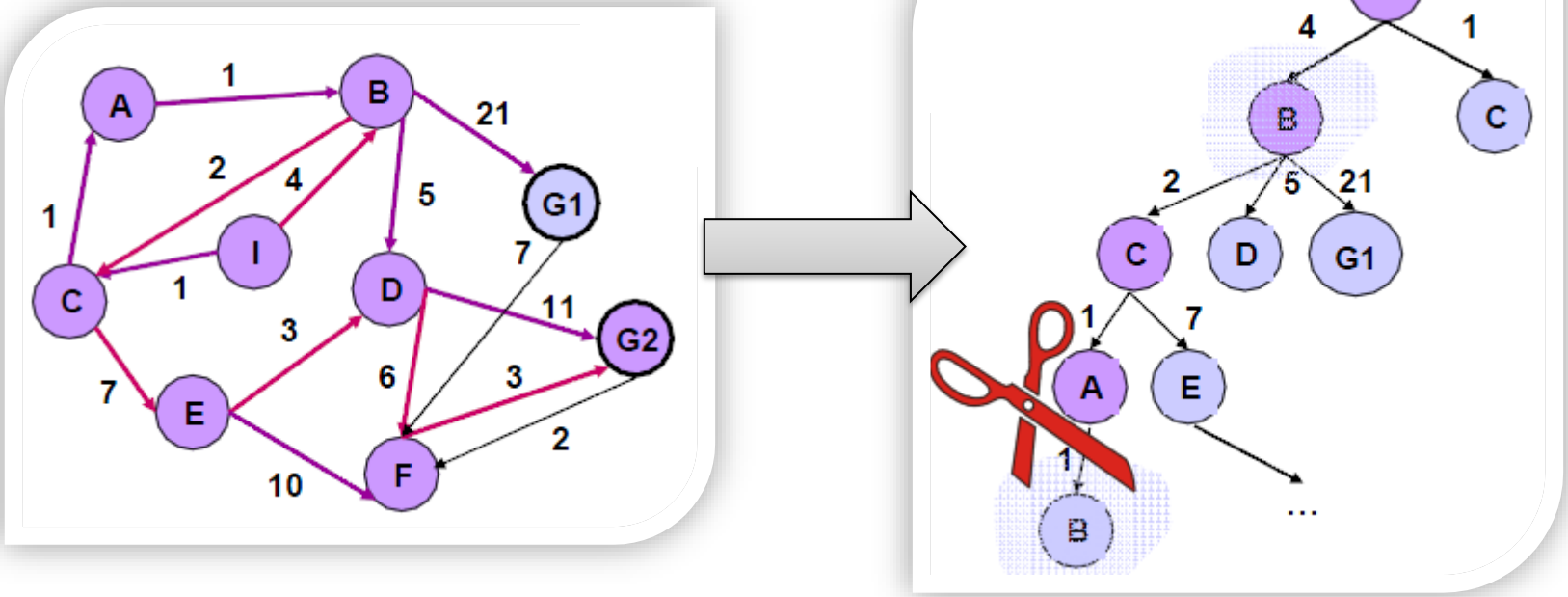
Búsqueda en profundidad + control de ciclos:

Mirar en la rama actual



3.3 Búsqueda en profundidad

Búsqueda en profundidad +
control de ciclos



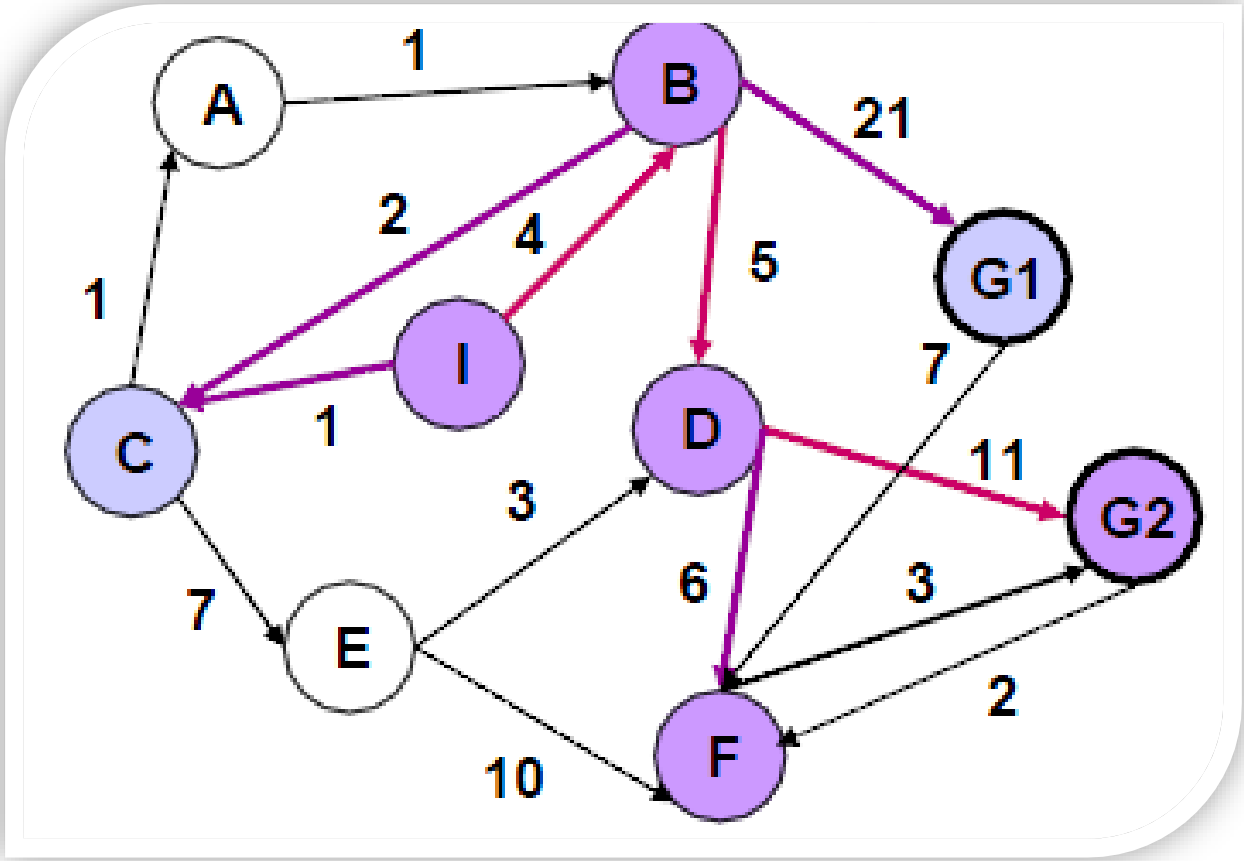
Mirar en la rama actual

- 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

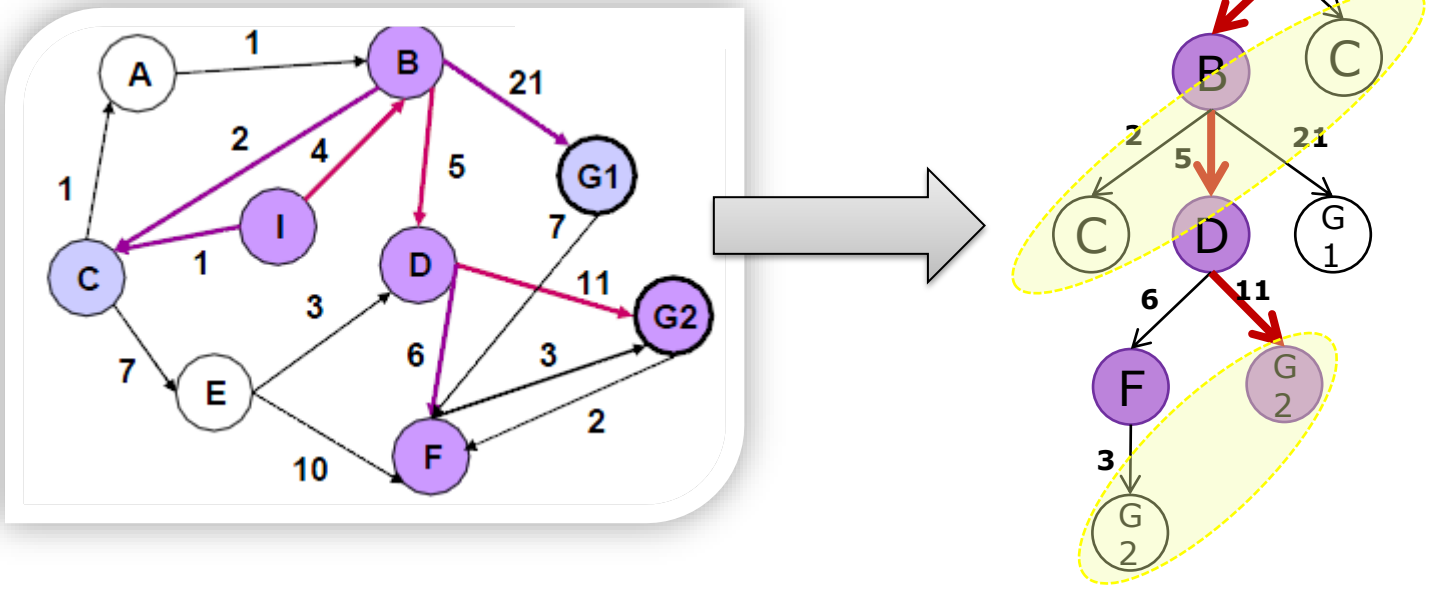
Búsqueda en profundidad + control de ciclos:

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 ℓ=∞ es idéntica a la búsqueda en profundidad
  - A veces se conoce el "diámetro" del espacio de estados a priori

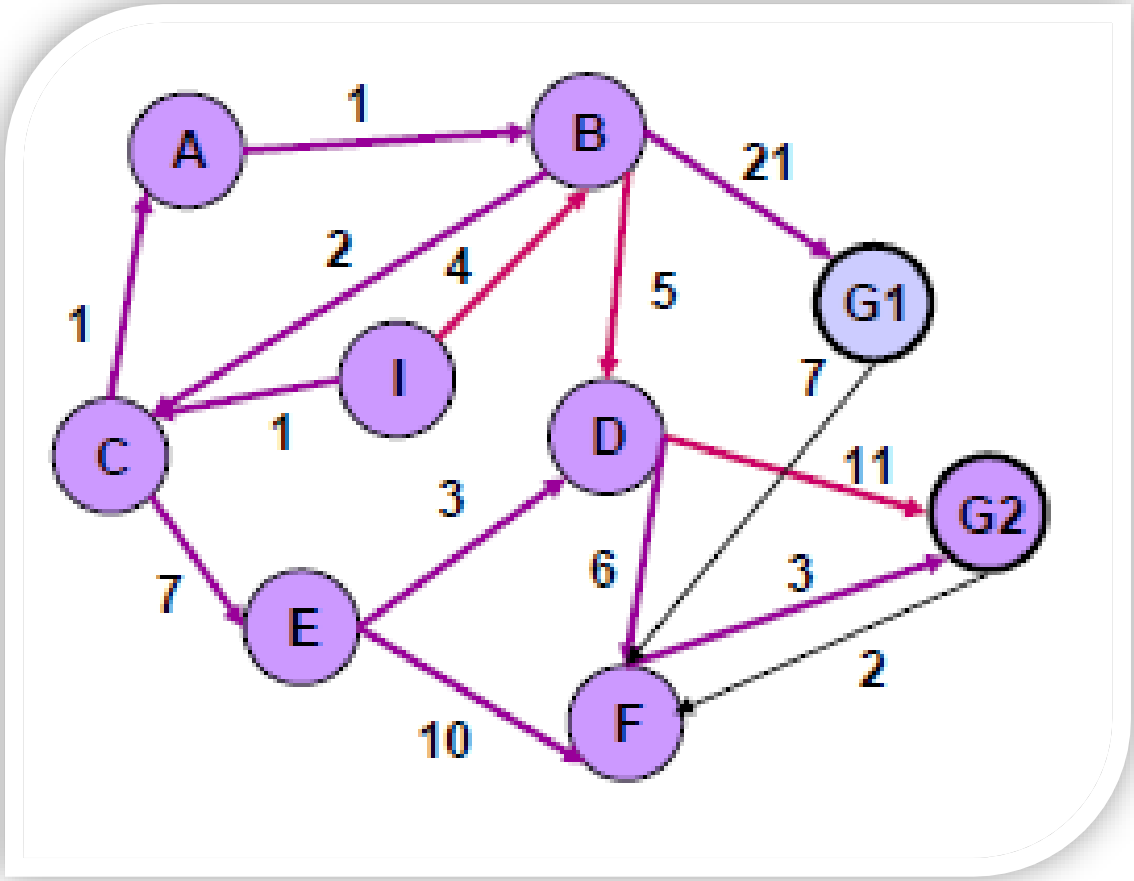
3.4 Búsqueda en profundidad limitada

- Propiedades:
  - Completa: **Si,** *si* $\ell >= d$ (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

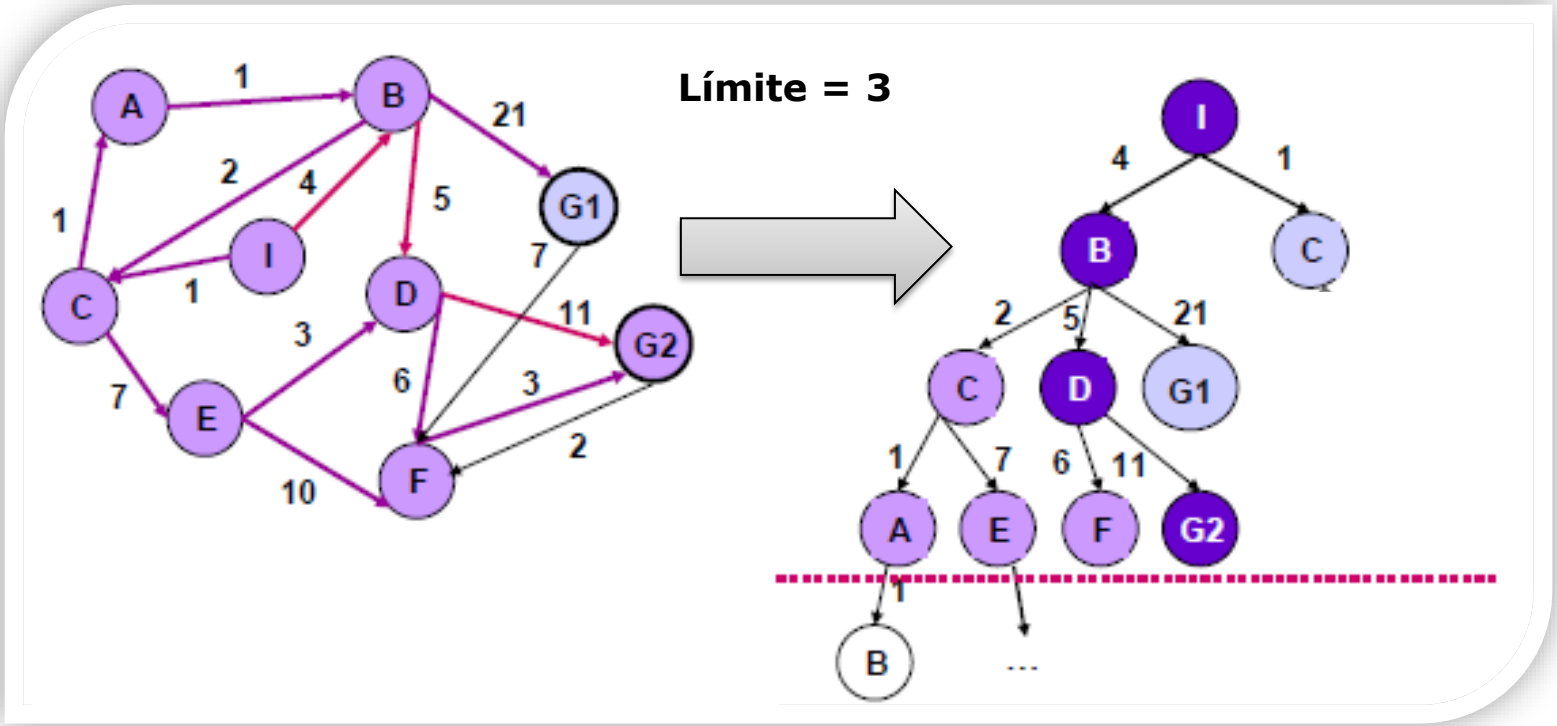
```
PROCEDIMIENTO PROFUNDIDAD_LIMITADA(Estado-Inicial, Estado-Final, LIMITE)

    *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

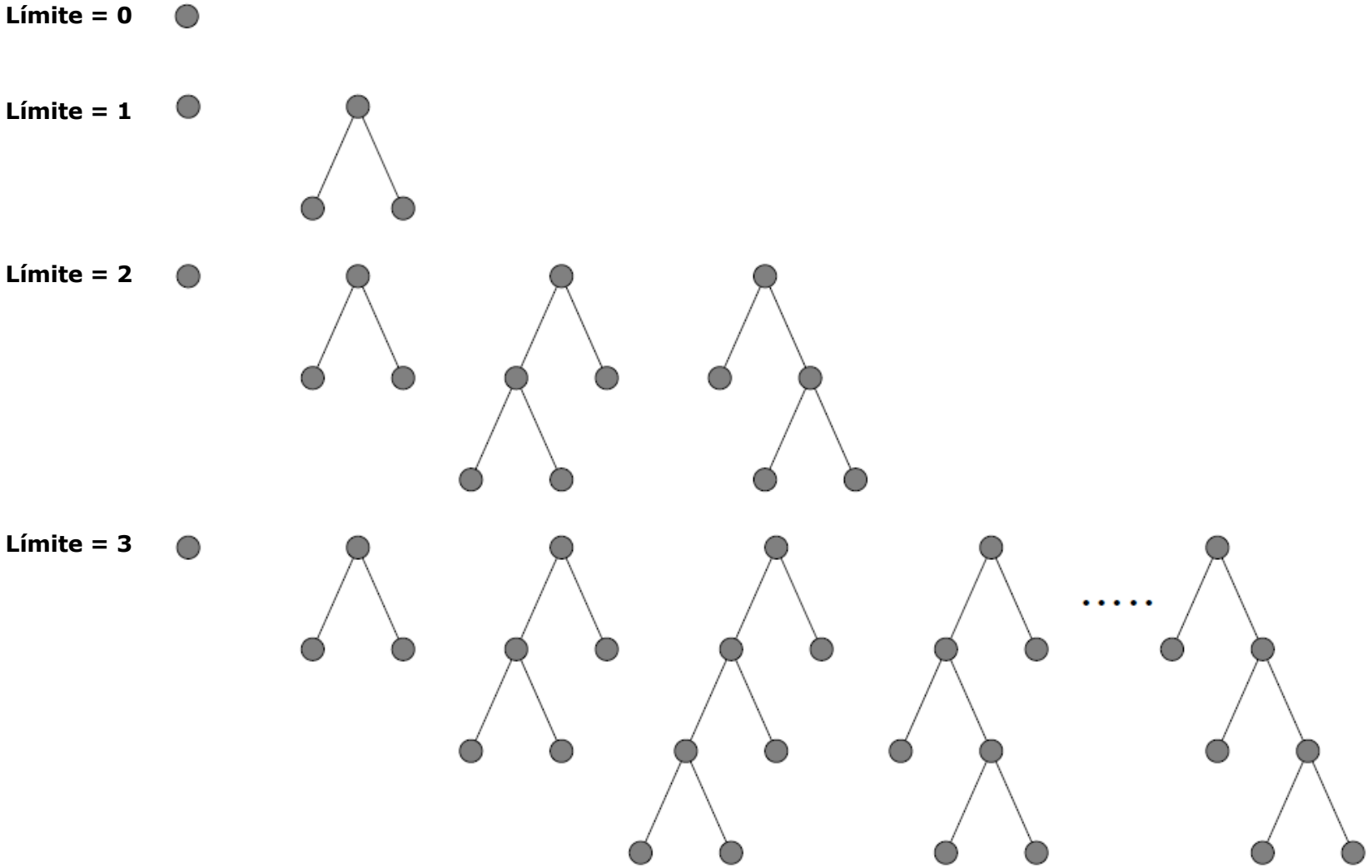
3.5 Búsqueda en profundidad iterativa

- 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:

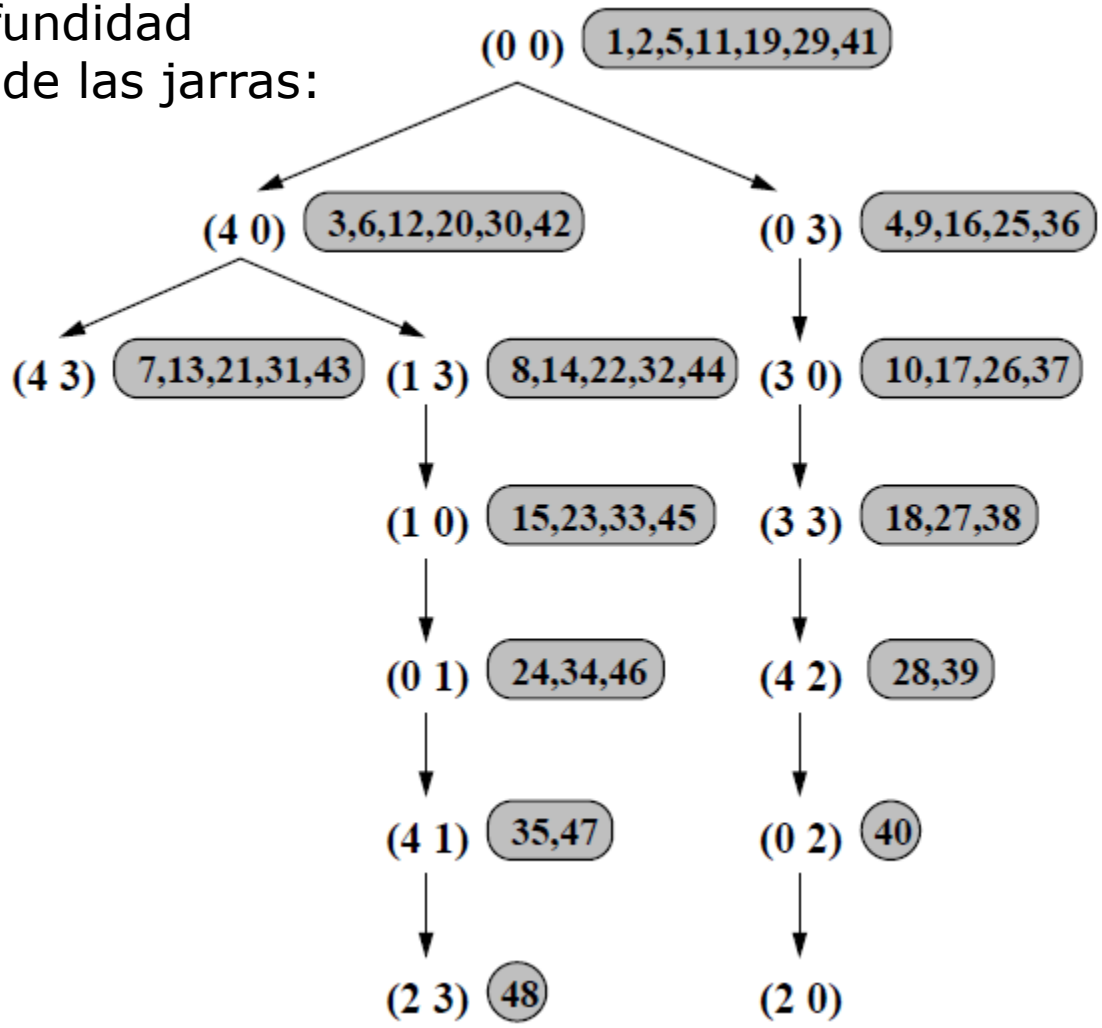
La mayoría de los nodos están en el último nivel del árbol

3.5 Búsqueda en profundidad iterativa

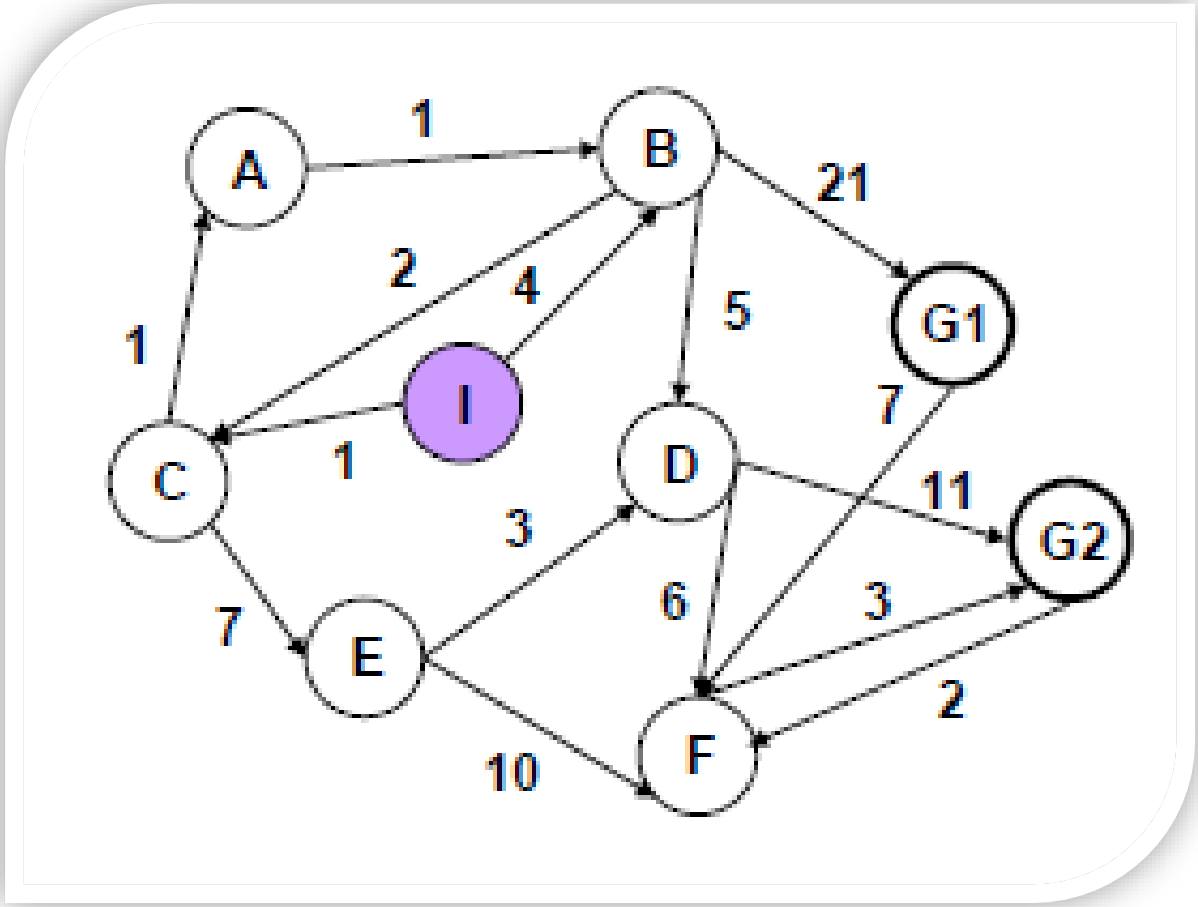
```
PROCEDIMIENTO PROFUNDIDAD_ITERATIVA(Estado-inicial, Estado-Final, COTA-INICIAL)

    Hacer N = COTA-INICIAL

    BUCLE (Repetir el proceso mientras N =< 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

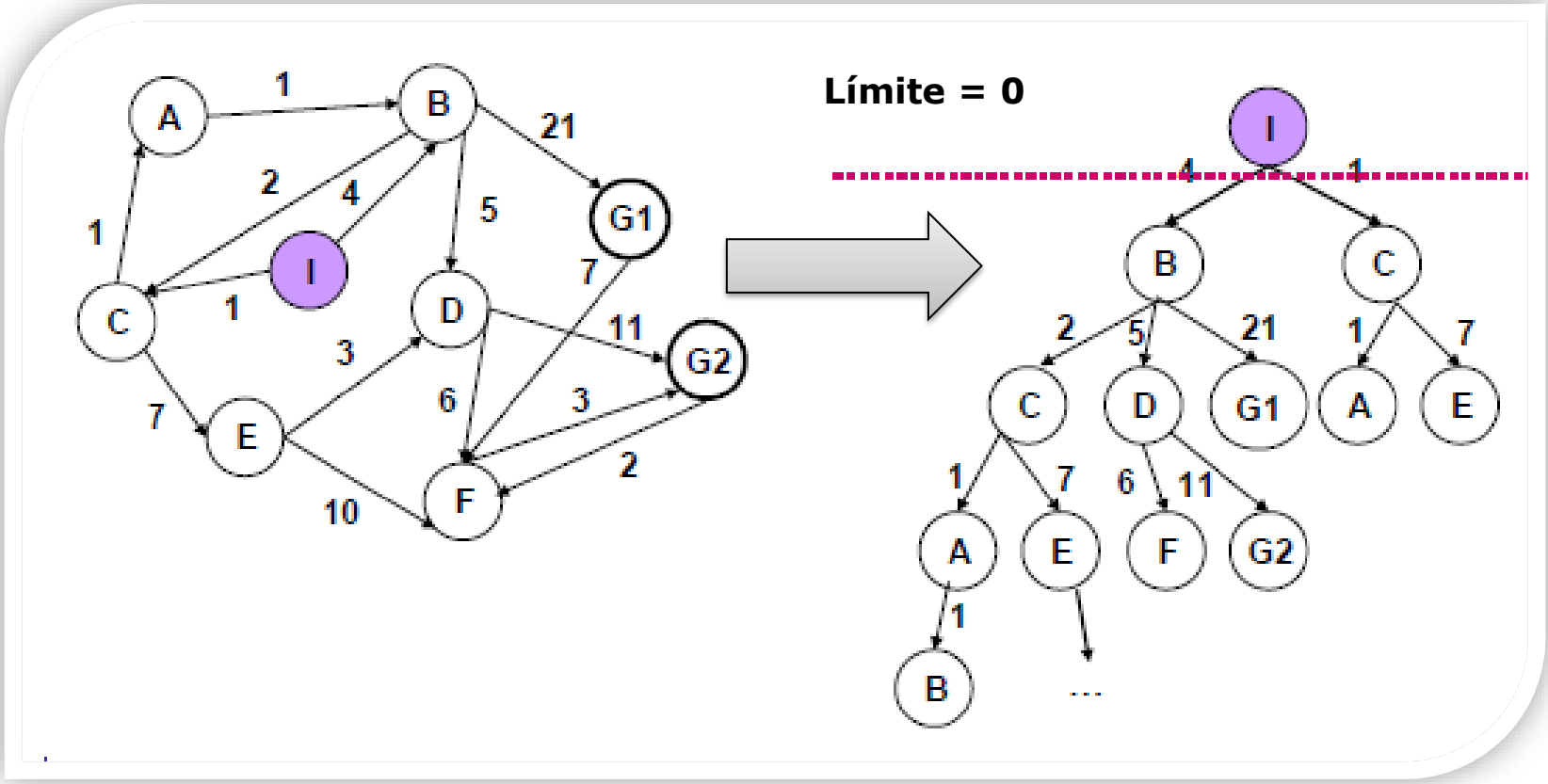
Árbol de búsqueda en profundidad iterativa para el problema de las jarras:



3.5 Búsqueda en profundidad iterativa

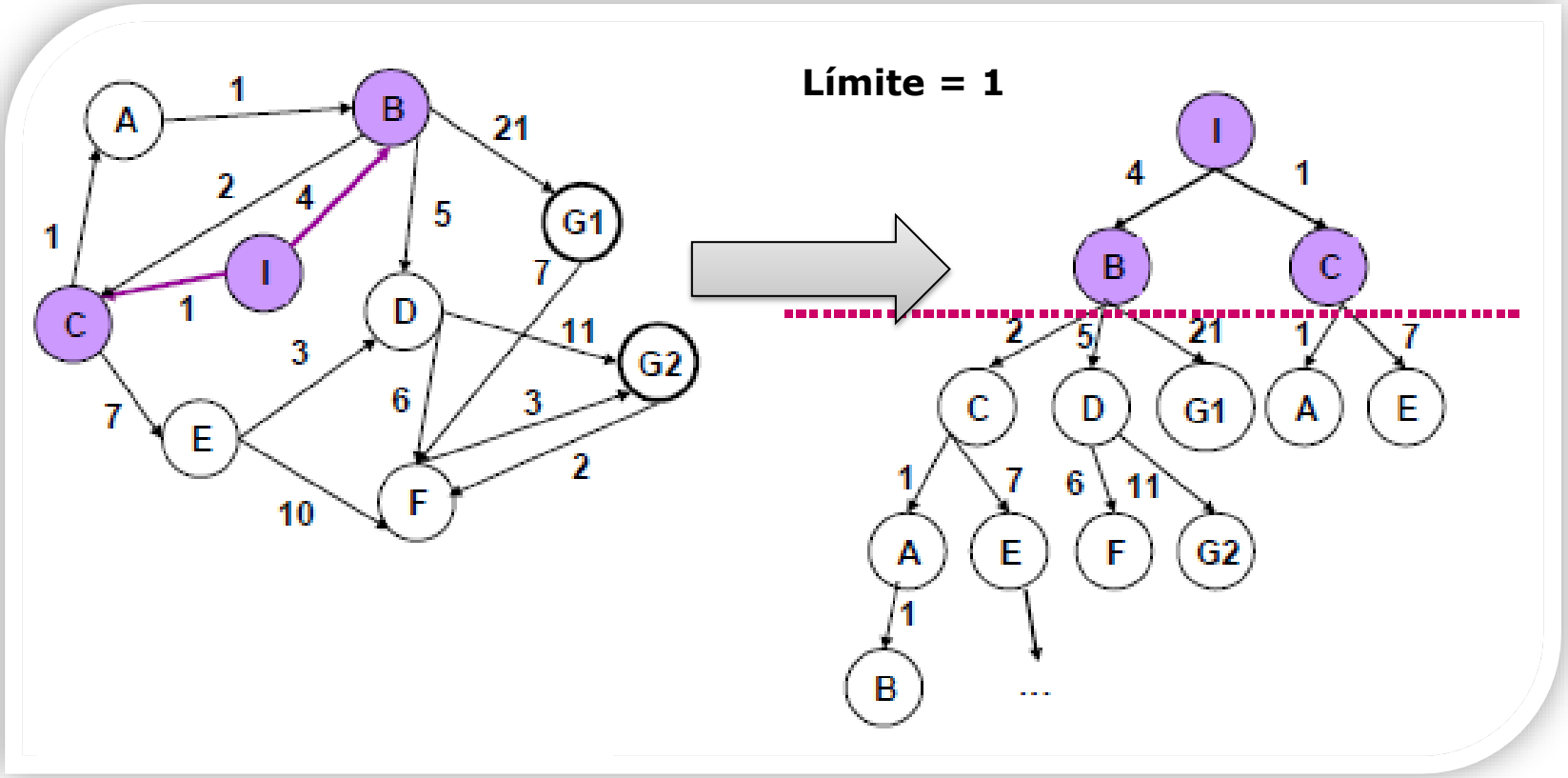


3.5 Búsqueda en profundidad iterativa



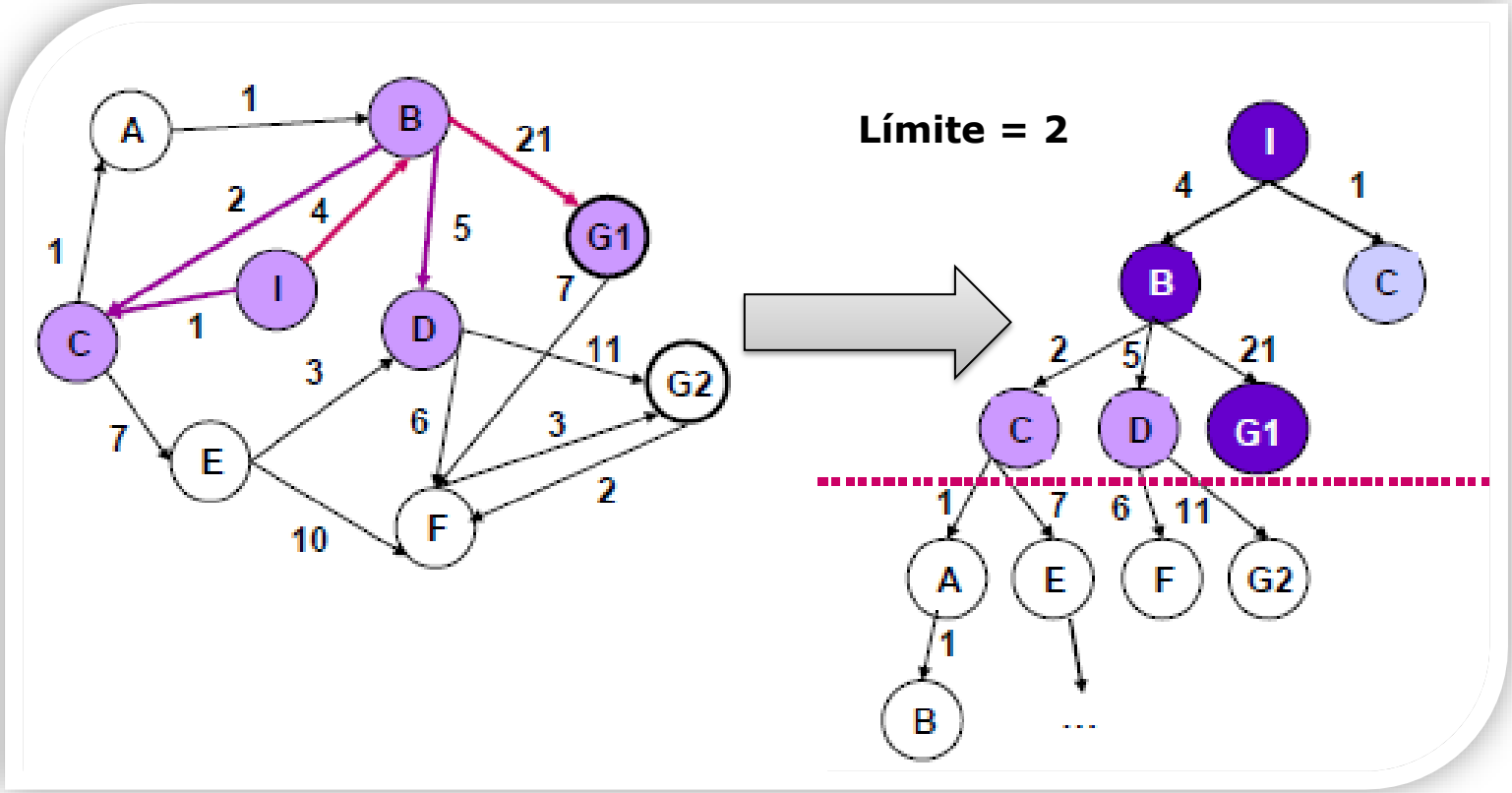
- Nodos cerrados: `I`

3.5 Búsqueda en profundidad iterativa



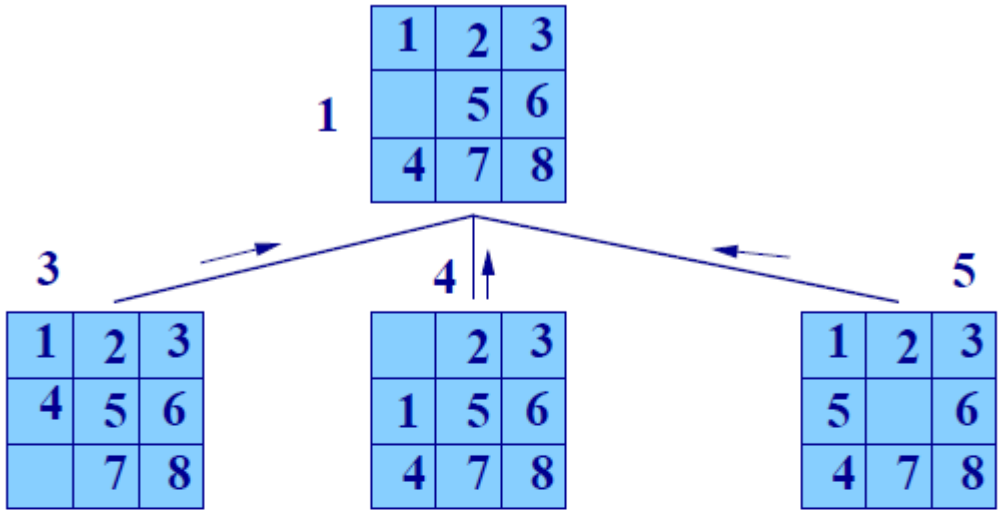
- Nodos cerrados: `I I B C`

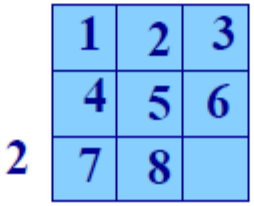
3.5 Búsqueda en profundidad iterativa



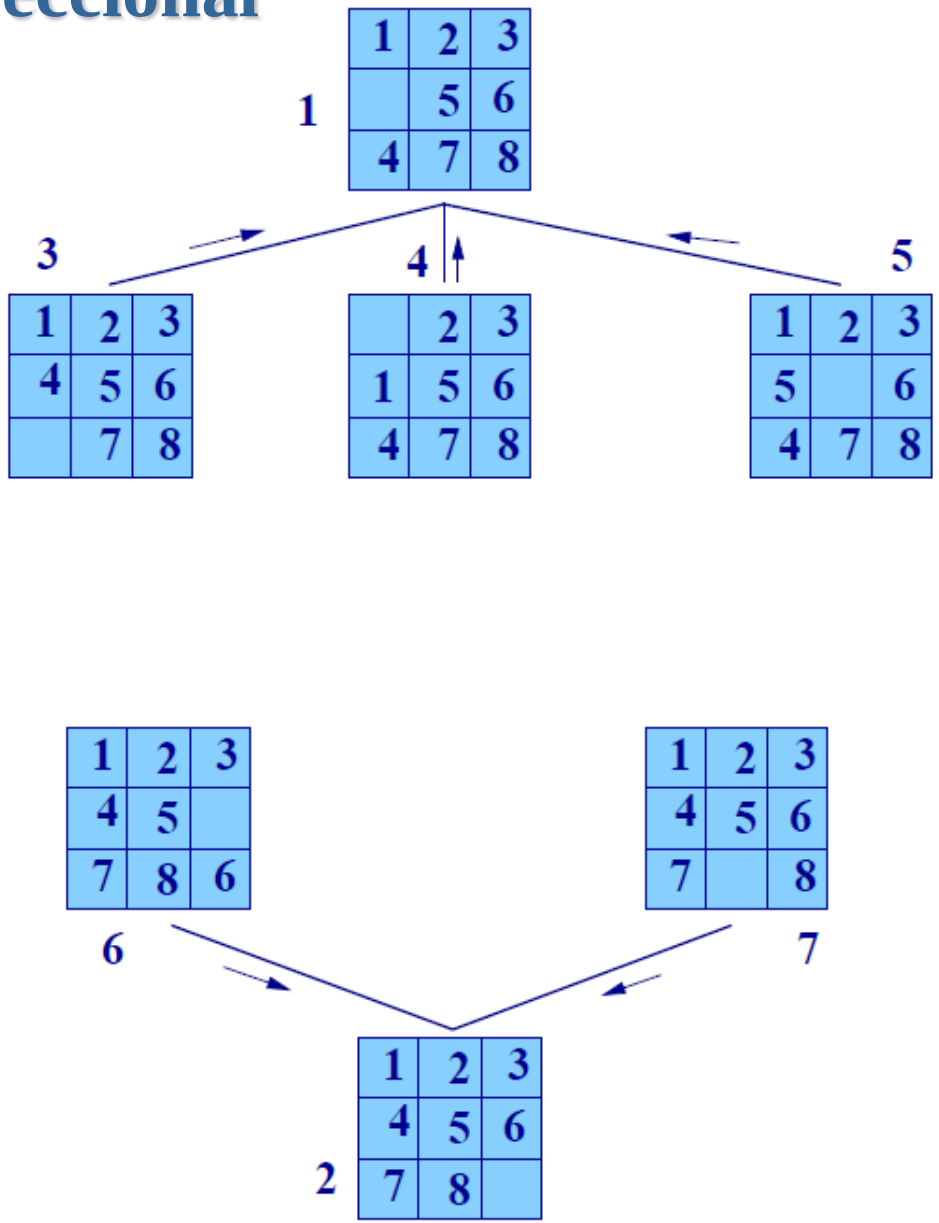
- Nodos cerrados: `I I B C I B C D G1`
- Camino a la solución: `I B G1`
- Coste: 4+21 = 25

3.6 Búsqueda Bidireccional

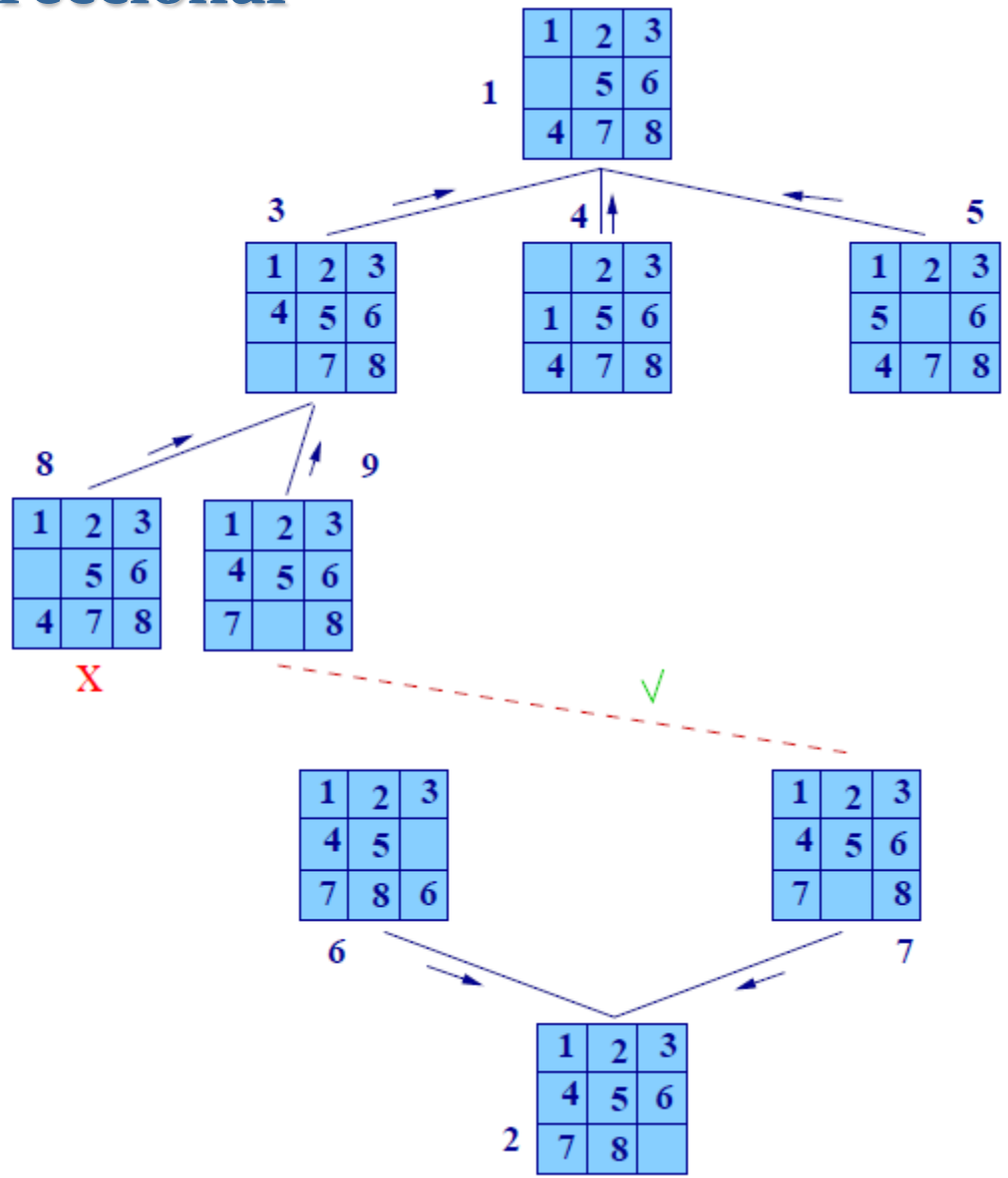




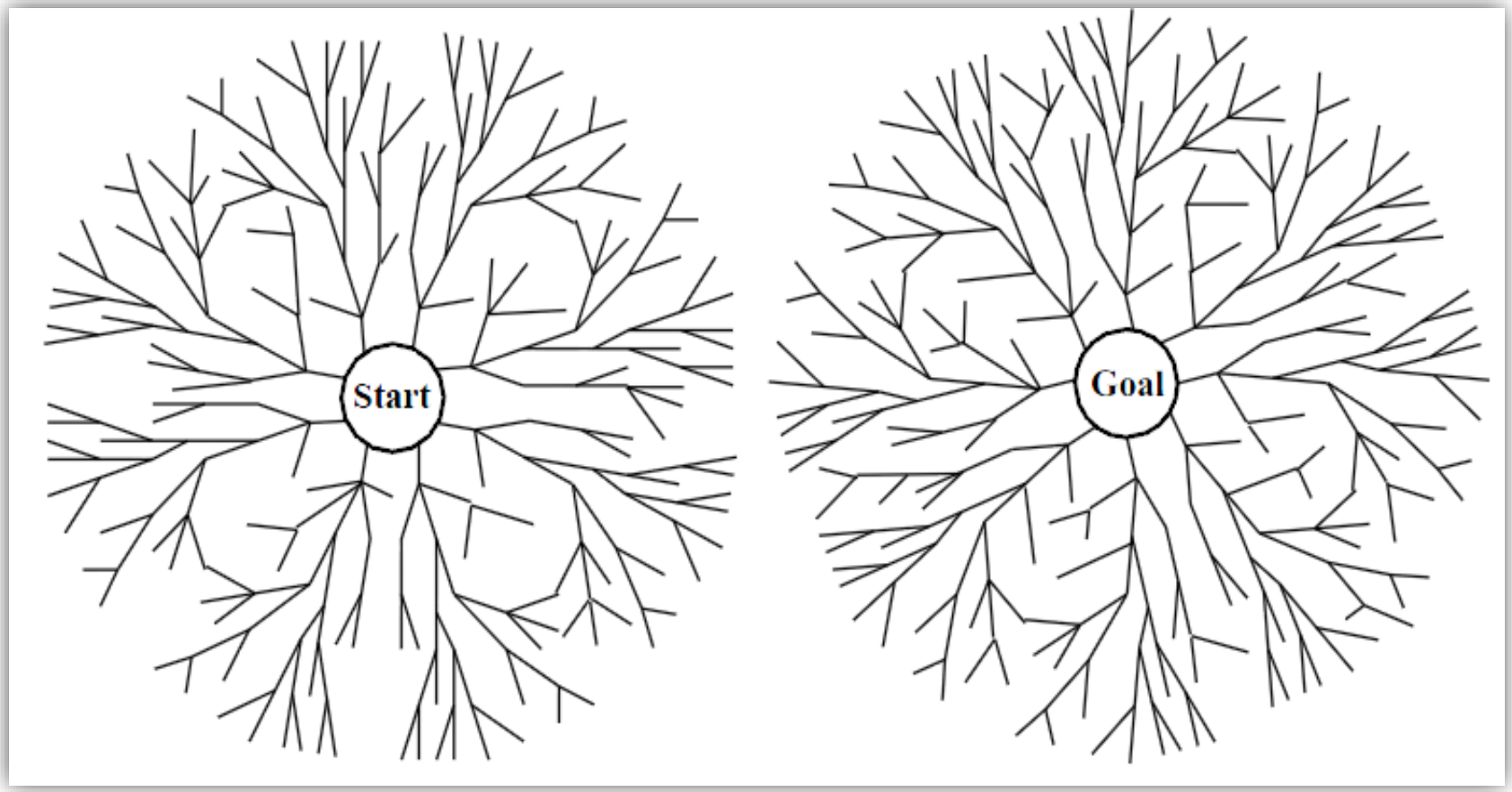
3.6 Búsqueda Bidireccional



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

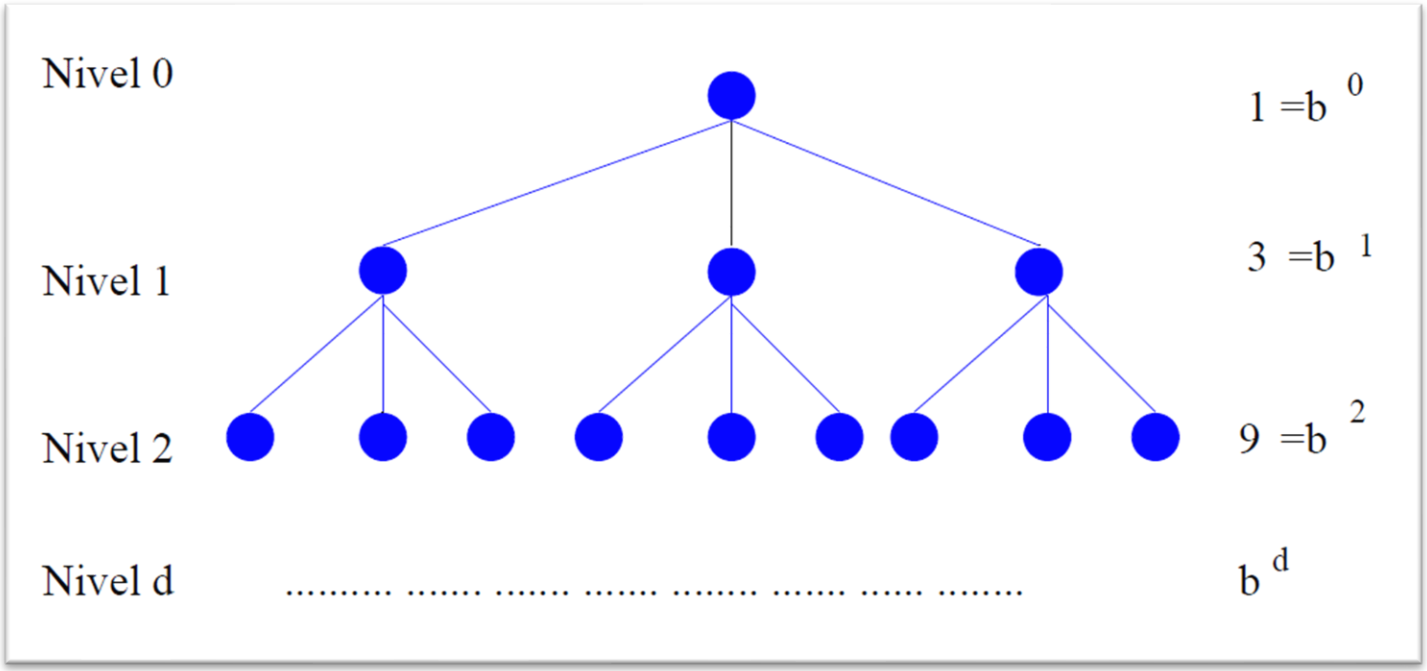
4. Complejidad

- 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)

- ¿Cual seria, en el peor de los casos, el numero de nodos que examinaría cada técnica?

- Ejemplo:
  - Supongamos b = 3, y vamos incrementando d
  - Calcular de forma inductiva el numero 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 ≈ 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 ≈ profundidad	$O(b^d)$	$O(b*d)$
Bidireccional	Si (anchura)	Si	$O(b^{d/2})$	$O(b^{d/2})$