

# 2. ESTRATEGIAS DE BÚSQUEDA NO INFORMADAS

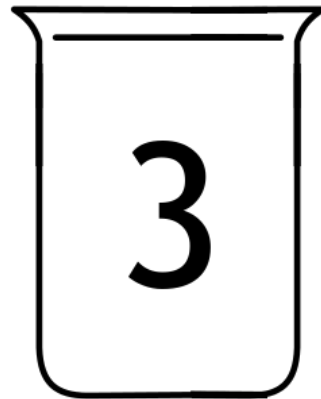
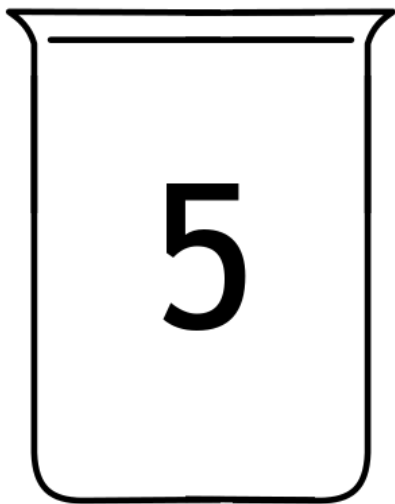
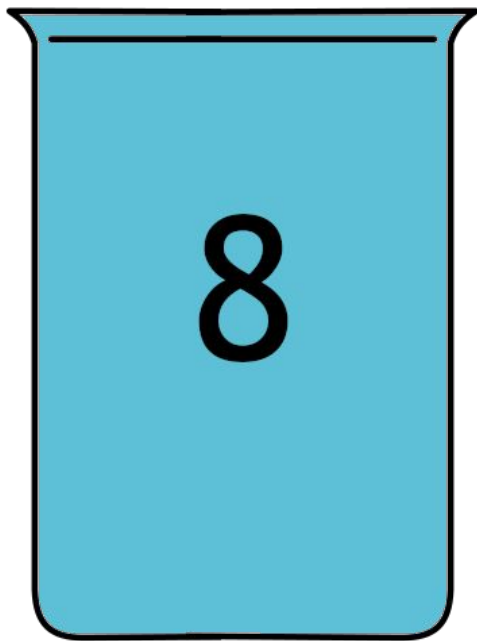
IA 3.2 - Programación III

1º C - 2023

Lic. Mauro Lucci

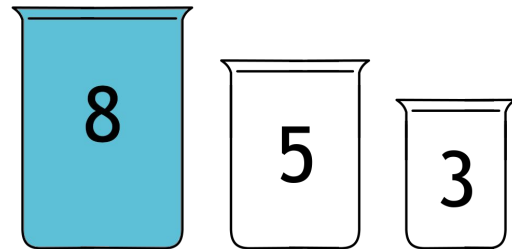
# Problema de vertido de agua

---



# Descripción

---



**Objetivo.** Lograr que uno de los recipientes contenga exactamente 4 litros de agua, en el menor número de vertidos.

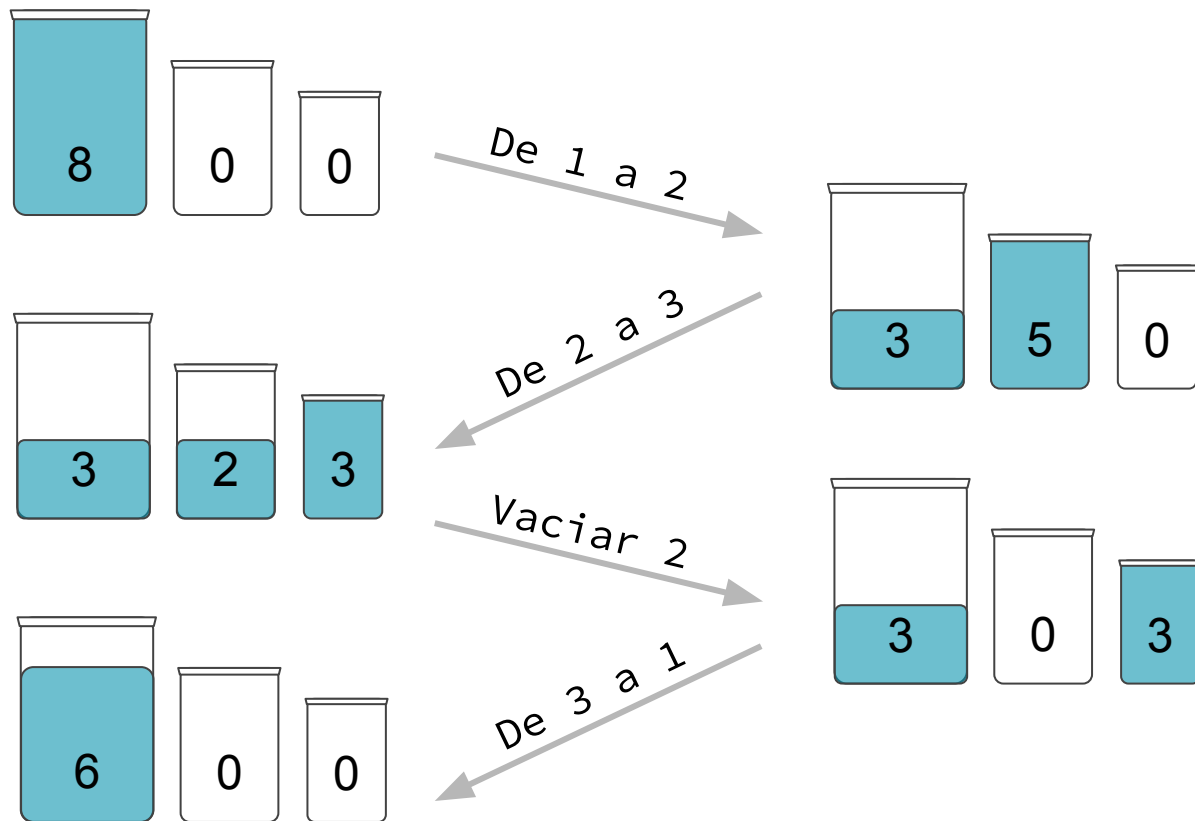
## Reglas.

1. Los recipientes no tienen marcas.
2. Se puede verter agua de un recipiente a otro, hasta que uno se vacíe o el otro se llene, lo que ocurra primero.
3. Se puede verter todo el contenido de un recipiente al piso.



# Ejemplo

---





## Ejercicio de repaso

— — —

1. Dar una formulación para este problema.
2. Encontrar una solución.



# Respuesta – 1. Propuesta de formulación

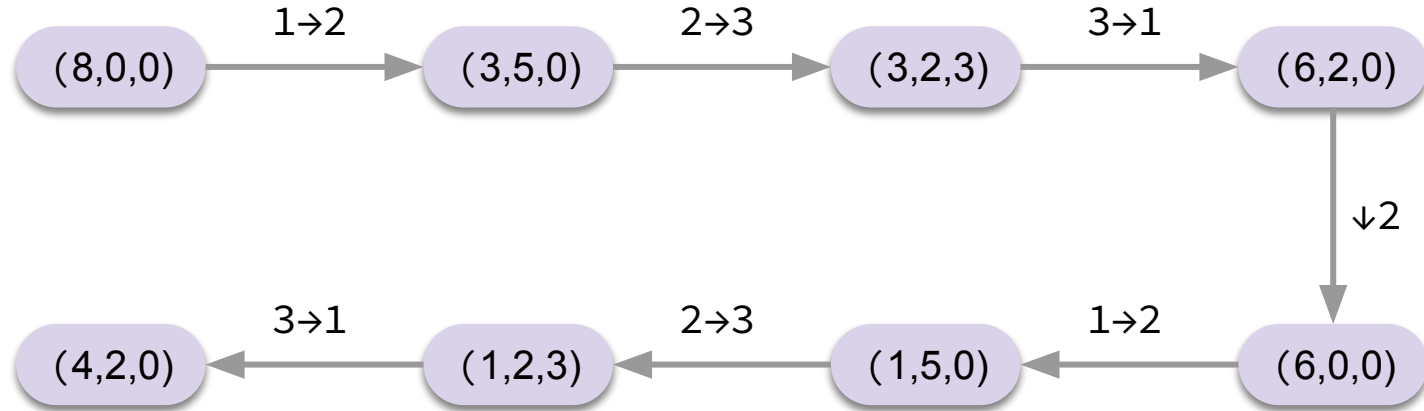
— — —

- **Estados.** Tuplas  $(x_1, x_2, x_3)$  con  $x_1 \in \{0, \dots, 8\}$ ,  $x_2 \in \{0, \dots, 5\}$ ,  $x_3 \in \{0, \dots, 3\}$ . Hay  $9 \cdot 6 \cdot 4 = 216$  estados.
- **Estado inicial.**  $(8, 0, 0)$
- **Acciones.**
  - $i \rightarrow j$ : verter del recipiente  $i$  al  $j$  ( $1 \rightarrow 2$ ,  $1 \rightarrow 3$ ,  $2 \rightarrow 1$ ,  $2 \rightarrow 3$ ,  $3 \rightarrow 1$ ,  $3 \rightarrow 2$ ).
  - $\downarrow i$ : vaciar el recipiente  $i$  ( $\downarrow 1$ ,  $\downarrow 2$ ,  $\downarrow 3$ ).
- **Modelo transicional.**
  - $i \rightarrow j$ : el recipiente  $i$  tendrá  $x_i - d$  y el  $j$  tendrá  $x_j + d$ , donde  $d$  es el mínimo entre  $x_i$  y lo que le falta a  $j$  para llenarse.
  - $\downarrow i$ : el recipiente  $i$  tendrá 0.
- **Test objetivo.**  $x_1 = 4$  ó  $x_2 = 4$
- **Costo de camino.** El costo individual es 1, luego el costo de camino es el número de vertidos realizados.



## Respuesta – 2. Una solución

— — —





# Repaso

— — —

El procedimiento básico para buscar una solución de un problema ya formulado es:

1. Crear la raíz con el estado inicial y agregarlo a la frontera.
2. Mientras la frontera sea no-vacía, **elegir** un nodo y removerlo.
3. Si el nodo contiene un estado objetivo, retornar la solución.
4. Sino, expandir el nodo y agregar los hijos generados a la frontera.

Vimos dos algoritmos generales de búsqueda.

- **TREE-SEARCH**. Considera todos los posibles caminos a una solución.
- **GRAPH-SEARCH**. Evita caminos redundantes, pero necesita más memoria.

Un nodo es una estructura de datos con: un estado, un nodo padre, un costo de camino, una acción y opcionalmente una profundidad.





# Repaso

— — —

**¿Cómo elegimos de la frontera el próximo nodo a expandir?**

Veremos diferentes **estrategias de búsqueda no informadas**:  
sólo tienen acceso a la definición del problema.

Compararemos su completitud, optimalidad, tiempo y memoria.

# Búsqueda primero en anchura

Breadth-First Search (BFS)

Los nodos del árbol de búsqueda se expanden por niveles.

Primero se expande la raíz, luego los hijos de la raíz, luego los hijos de los hijos de la raíz, y así hasta encontrar un estado objetivo.

— — —



# Ejemplo

— — —

Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1			
2			
3			
4			
5			

A: (8,0,0)



# Ejemplo

— — —

Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	
2			
3			
4			
5			

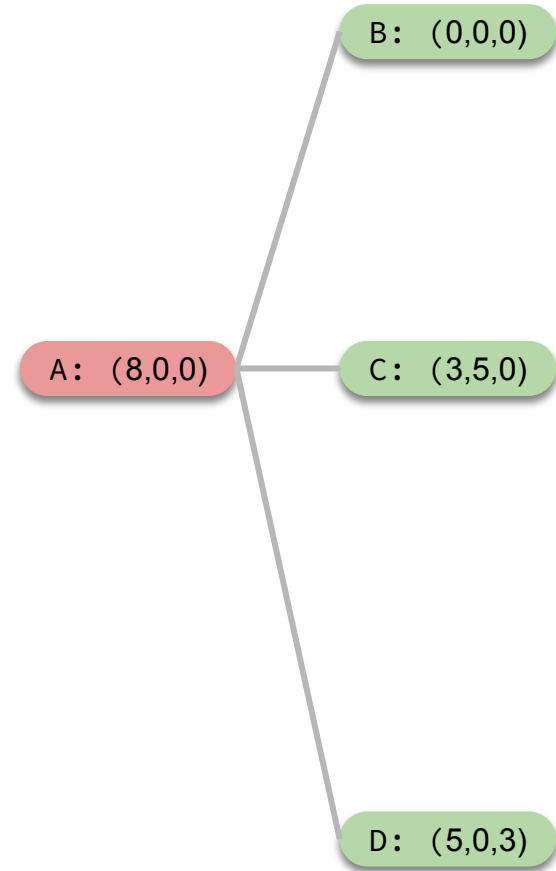
A: (8,0,0)



# Ejemplo

— — —

Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	{B,C,D}
2			
3			
4			
5			

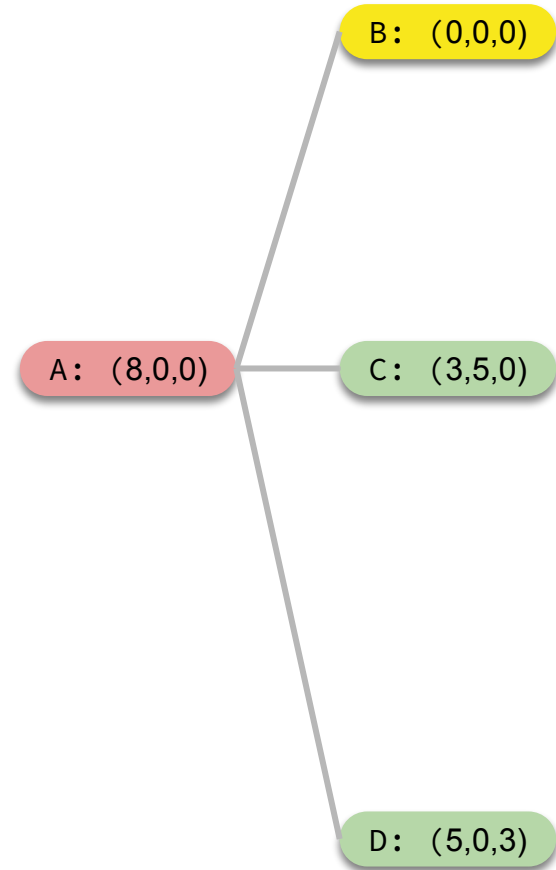




# Ejemplo

— — —

Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	{B,C,D}
2	B	{C,D}	
3			
4			
5			

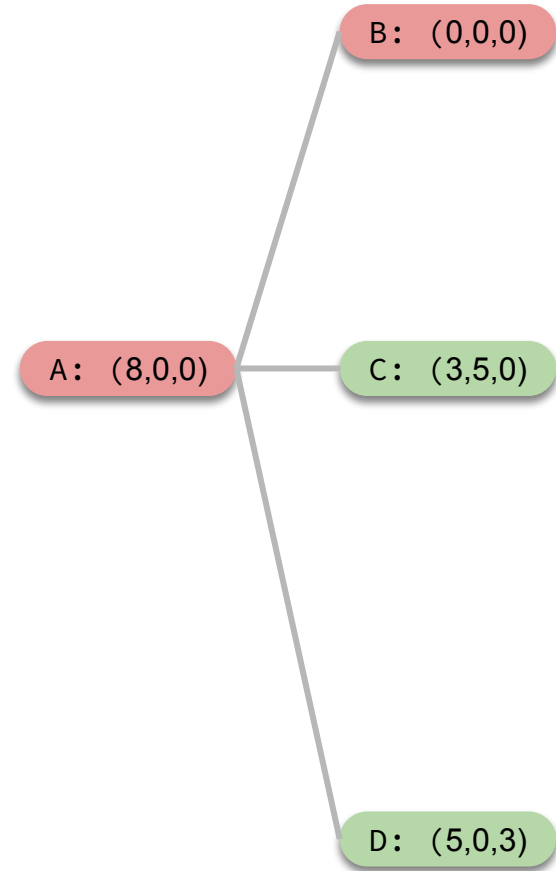




# Ejemplo

— — —

Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	{B,C,D}
2	B	{C,D}	{C,D}
3			
4			
5			

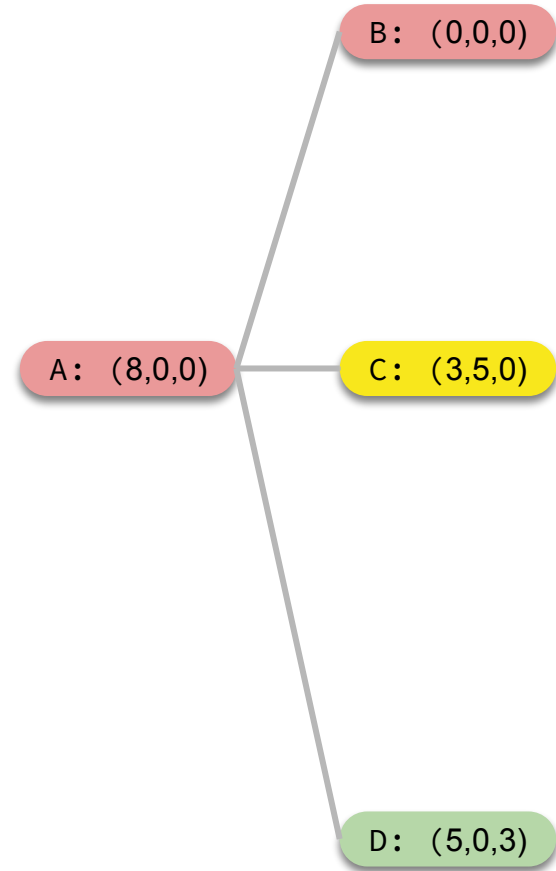




# Ejemplo

— — —

Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	{B,C,D}
2	B	{C,D}	{C,D}
3	C	{D}	
4			
5			



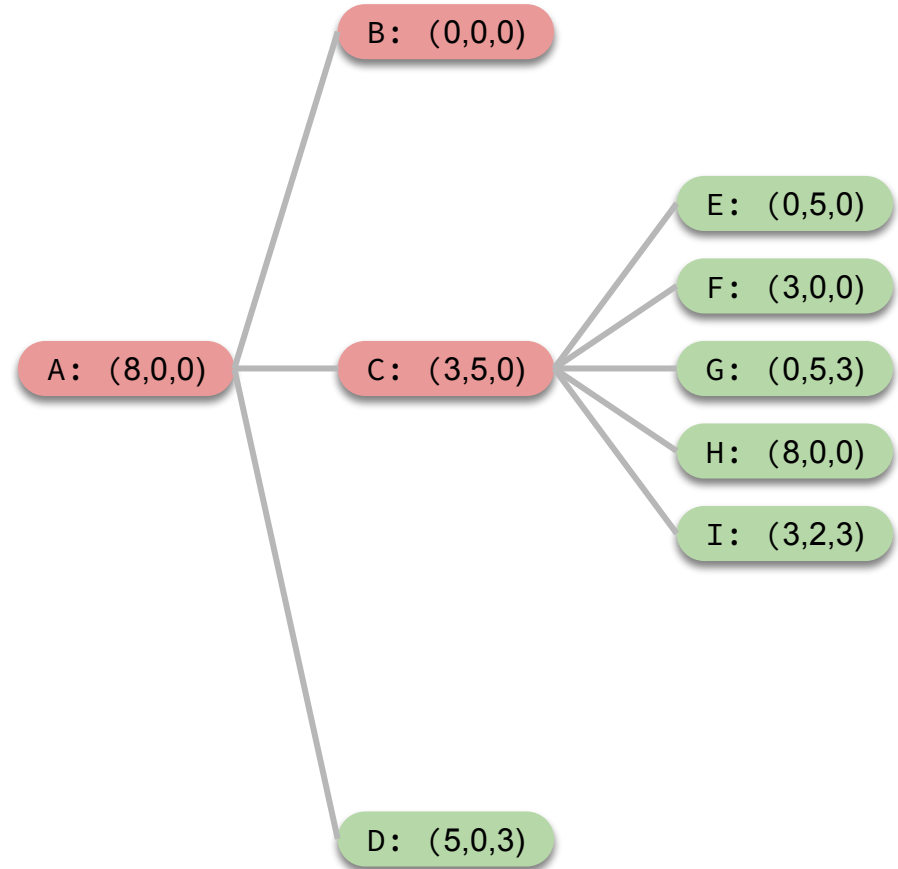




# Ejemplo

— — —

Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	{B,C,D}
2	B	{C,D}	{C,D}
3	C	{D}	{D,E,F,G,H,I}
4			
5			

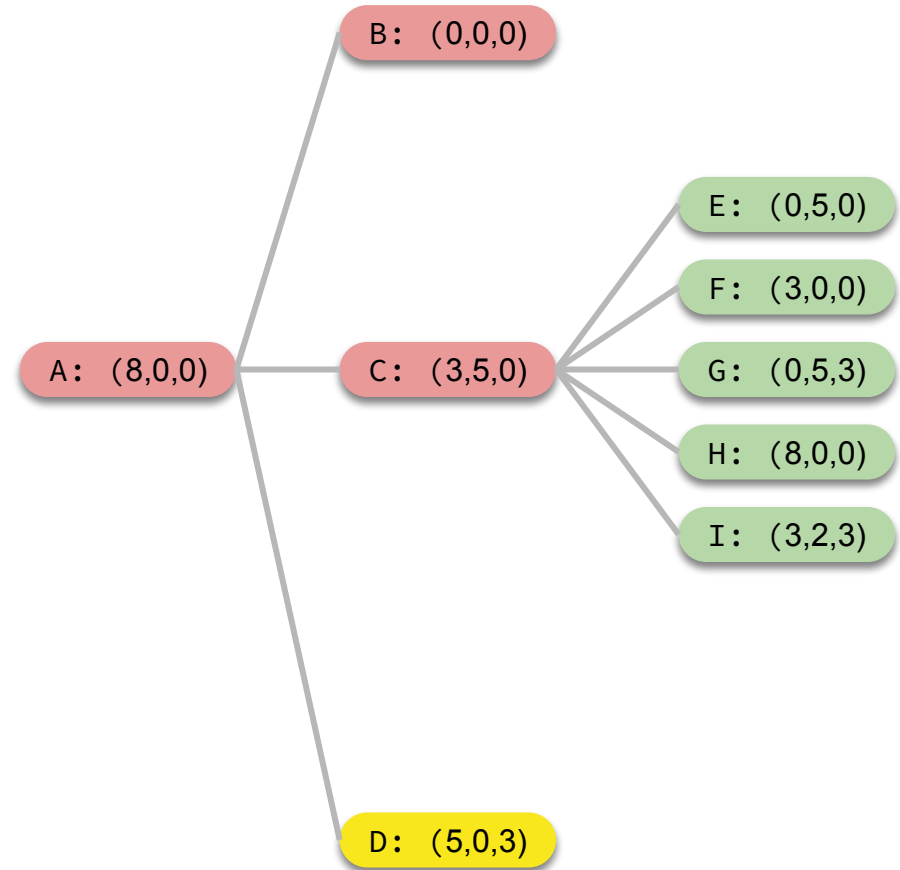




# Ejemplo

— — —

Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	{B,C,D}
2	B	{C,D}	{C,D}
3	C	{D}	{D,E,F,G,H,I}
4	D	{E,F,G,H,I}	
5			

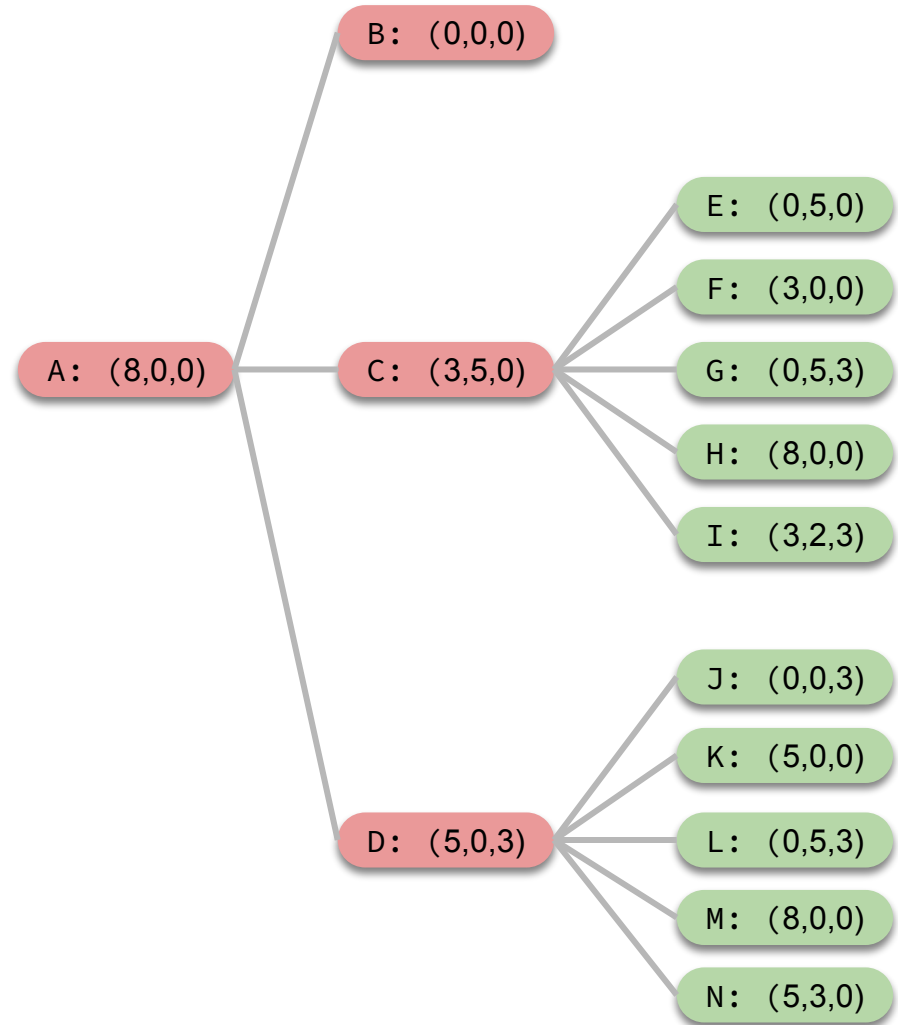




# Ejemplo

— — —

Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	{B,C,D}
2	B	{C,D}	{C,D}
3	C	{D}	{D,E,F,G,H,I}
4	D	{E,F,G,H,I}	{E,F,G,H,I,J,K,L,M,N}
5			

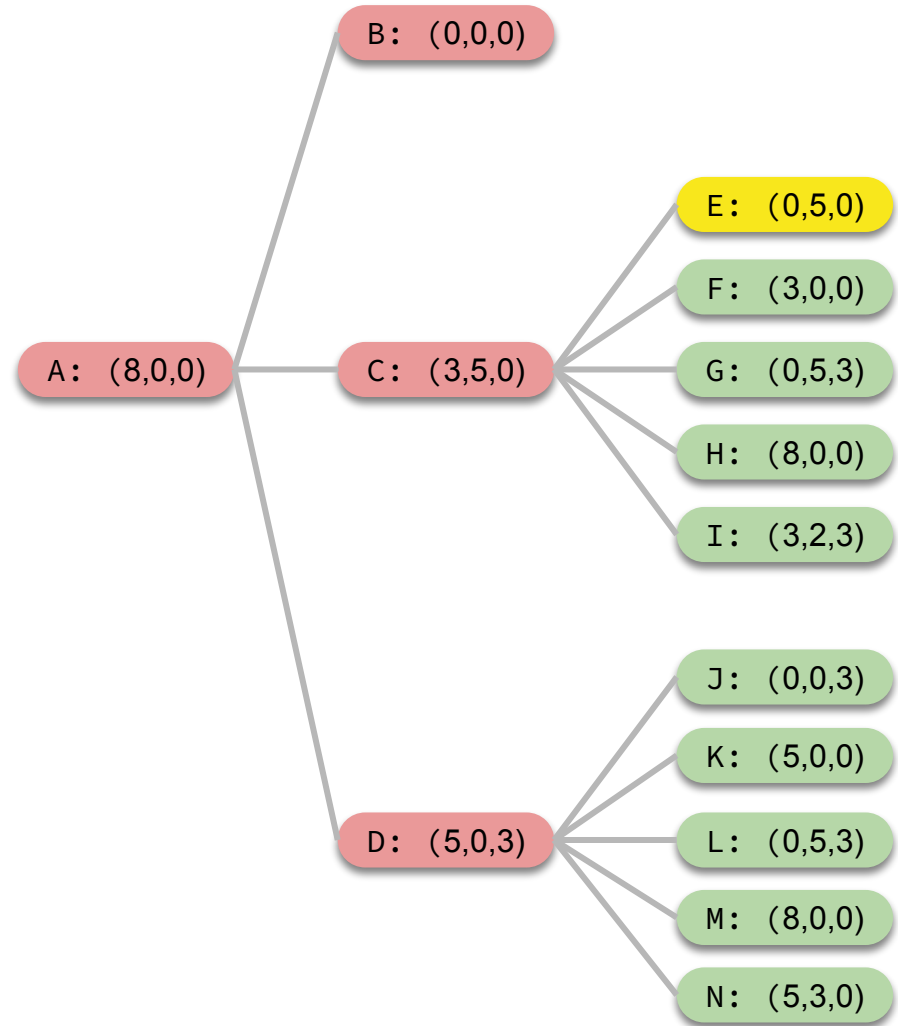




# Ejemplo

— — —

Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	{B,C,D}
2	B	{C,D}	{C,D}
3	C	{D}	{D,E,F,G,H,I}
4	D	{E,F,G,H,I}	{E,F,G,H,I,J,K,L,M,N}
5	E	{F,G,H,I,J,K,L,M,N}	...CONTINÚA...

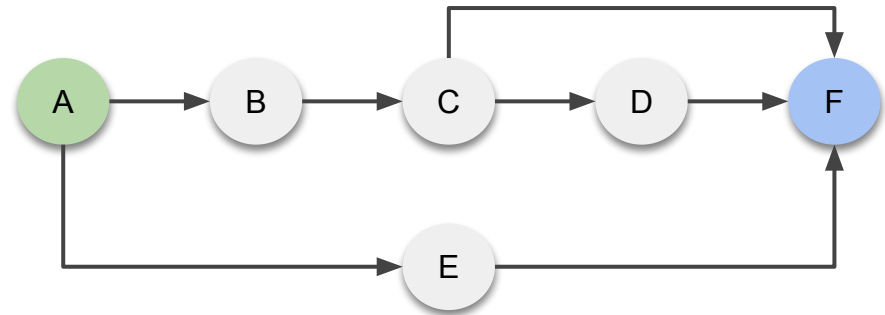




## Ejercicio

Sea un problema descrito por el siguiente espacio de estados, donde A es el estado inicial y F es el estado objetivo.

Resolverlo con el algoritmo general de búsqueda en árboles con la estrategia BFS.

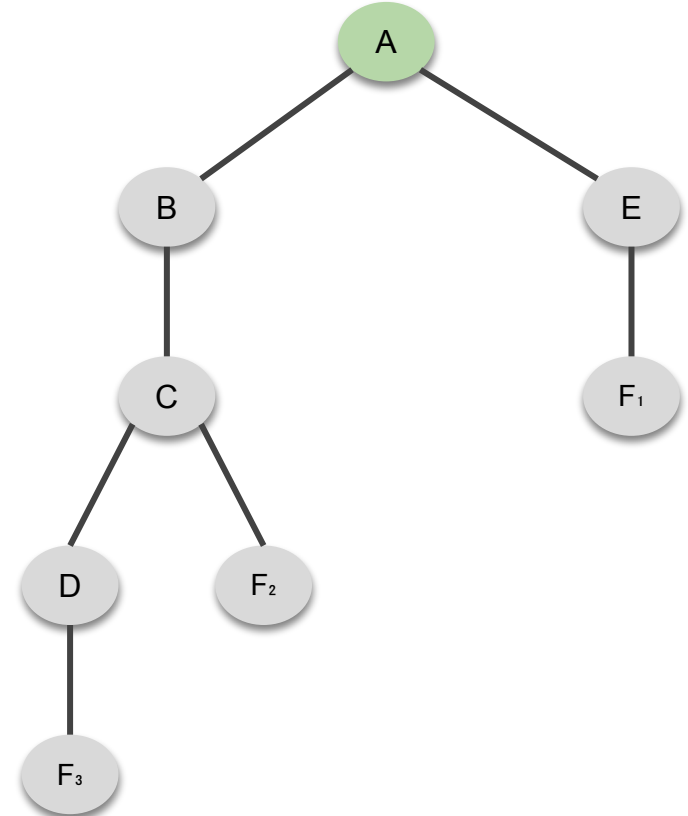




# Respuesta

— — —

Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1			
2			
3			
4			
5			

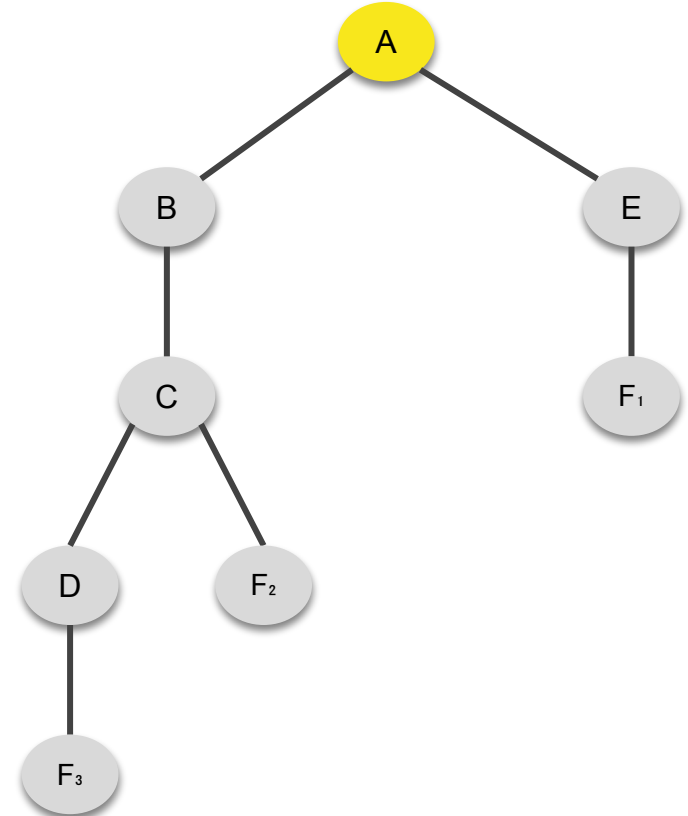




# Respuesta

— — —

Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	
2			
3			
4			
5			

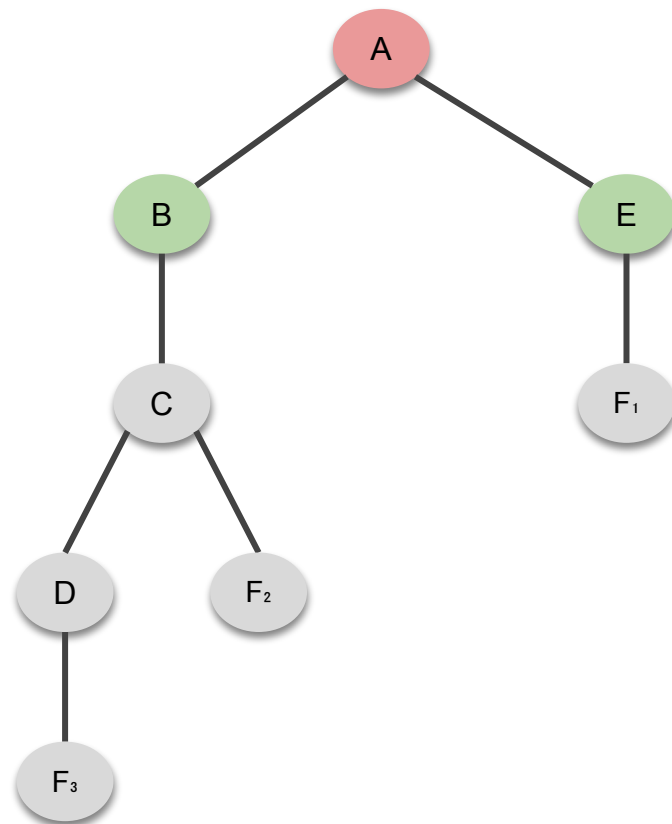




# Respuesta

— — —

Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	{B,E}
2			
3			
4			
5			



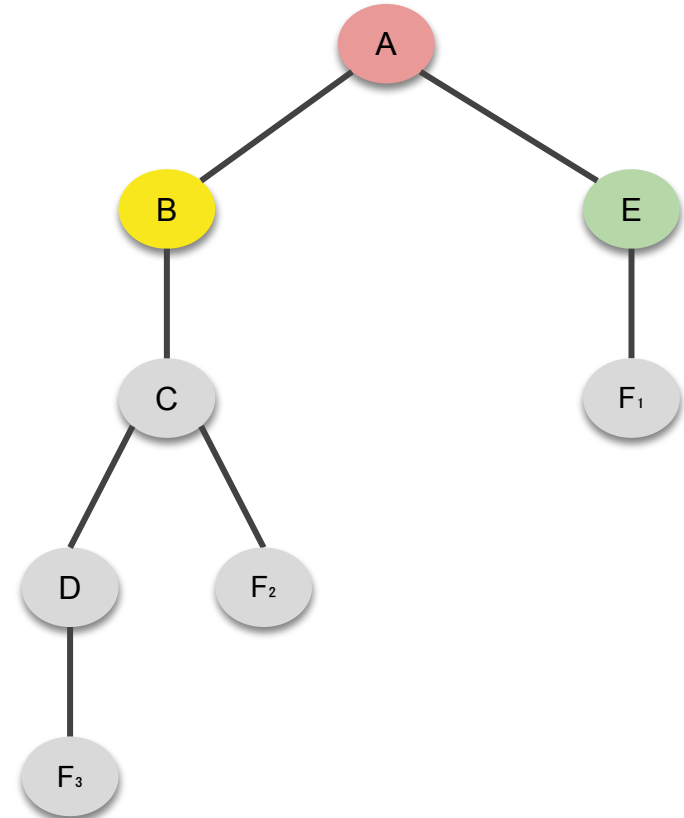




# Respuesta

— — —

Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	{B,E}
2	B	{E}	
3			
4			
5			

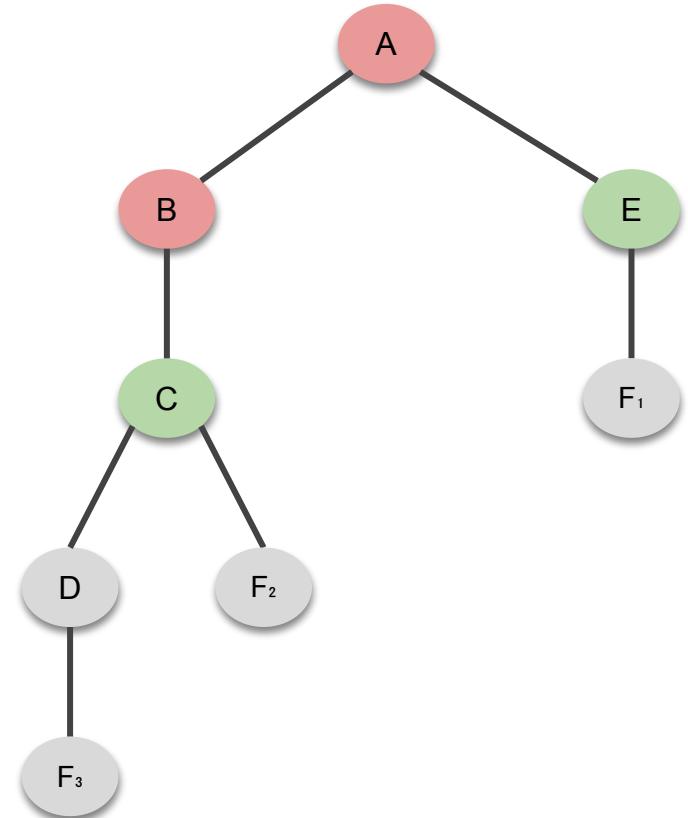




# Respuesta

— — —

Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	{B,E}
2	B	{E}	{E,C}
3			
4			
5			

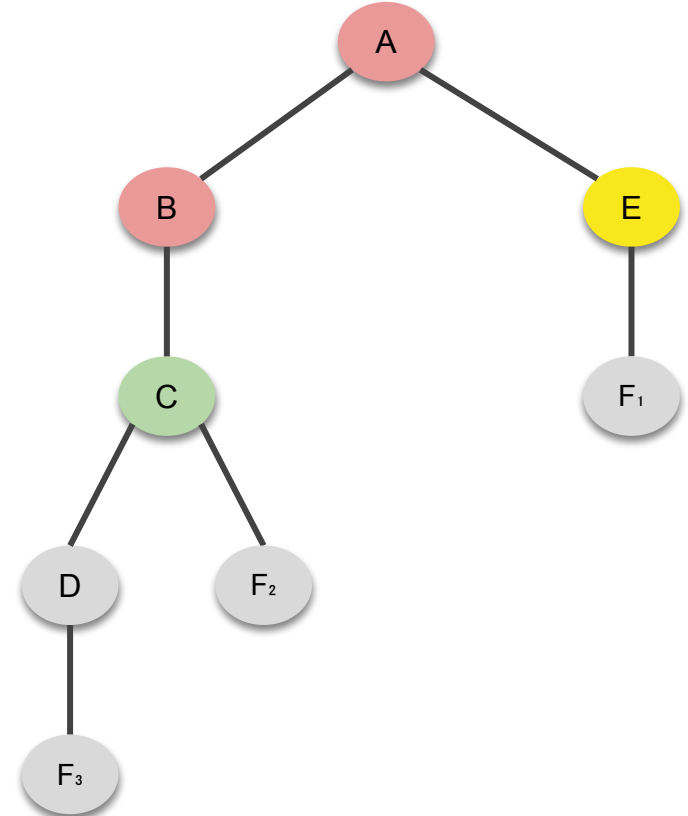




# Respuesta

— — —

Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	{B,E}
2	B	{E}	{E,C}
3	E	{C}	
4			
5			

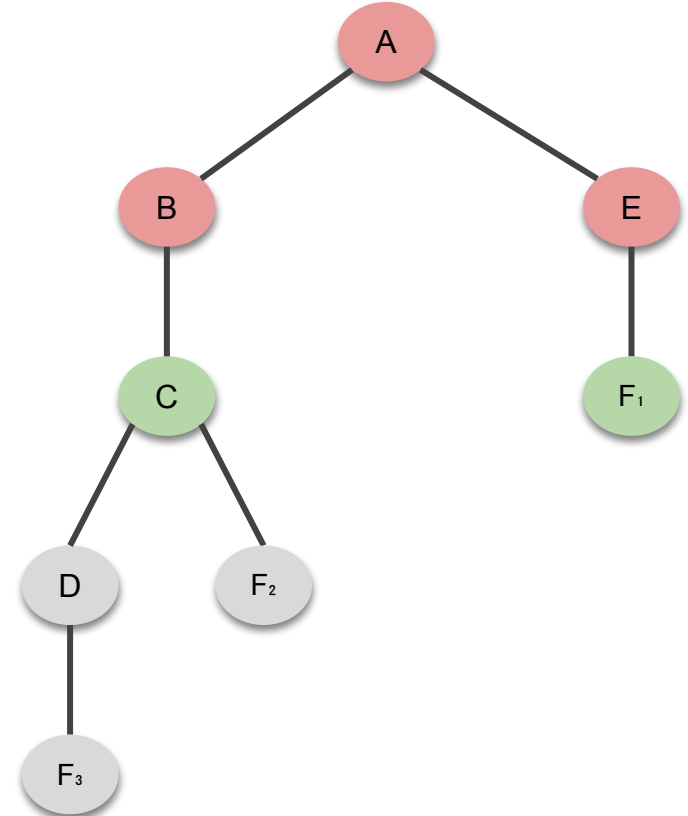




# Respuesta

— — —

Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	{B,E}
2	B	{E}	{E,C}
3	E	{C}	{C,F <sub>1</sub> }
4			
5			

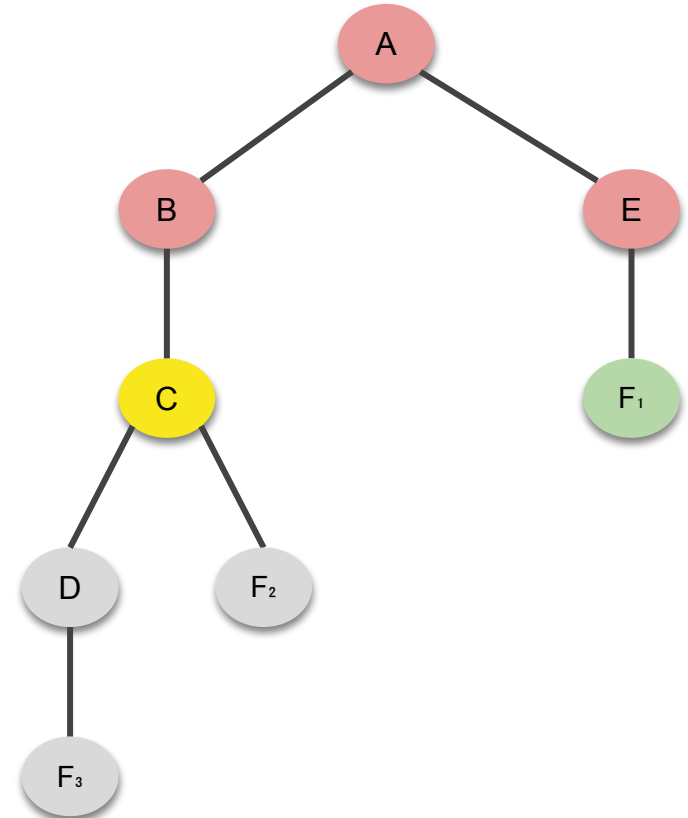




# Respuesta

— — —

Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	{B,E}
2	B	{E}	{E,C}
3	E	{C}	{C,F <sub>1</sub> }
4	C	{F <sub>1</sub> }	
5			

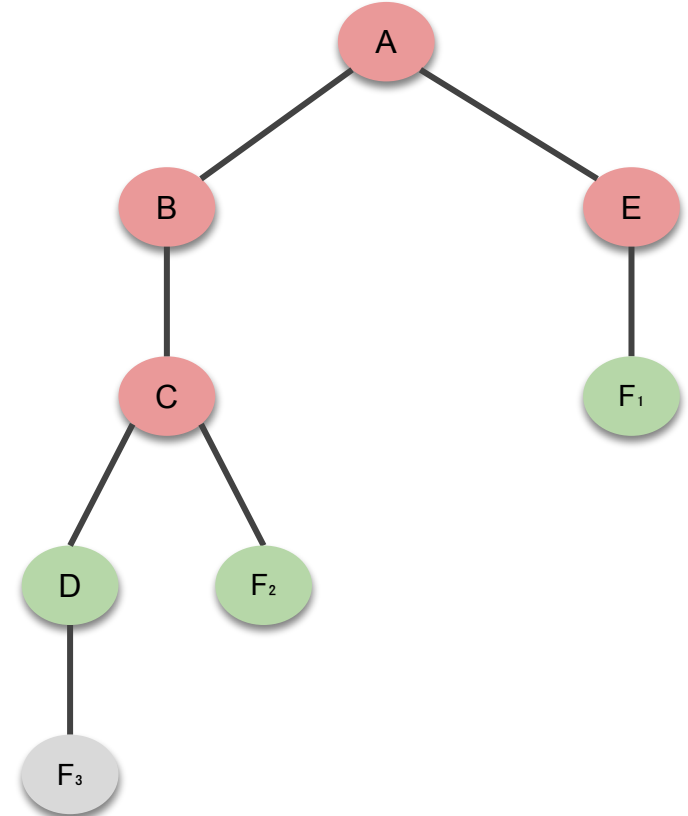




# Respuesta

— — —

Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	{B,E}
2	B	{E}	{E,C}
3	E	{C}	{C,F <sub>1</sub> }
4	C	{F <sub>1</sub> }	{F <sub>1</sub> ,D,F <sub>2</sub> }
5			

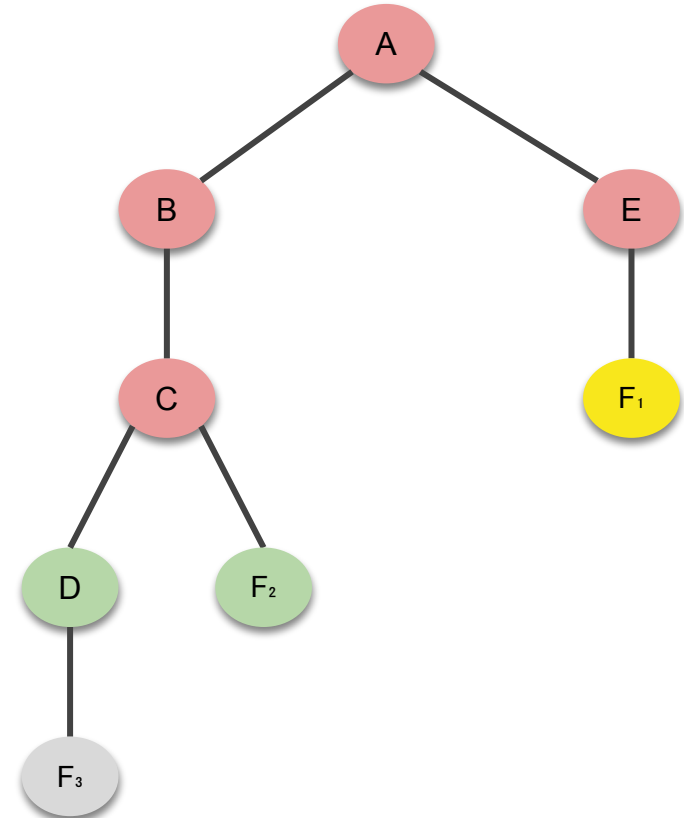




# Respuesta

— — —

Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	{B,E}
2	B	{E}	{E,C}
3	E	{C}	{C,F <sub>1</sub> }
4	C	{F <sub>1</sub> }	{F <sub>1</sub> ,D,F <sub>2</sub> }
5	{F <sub>1</sub> }	{D,F <sub>2</sub> }	<b>FIN</b>

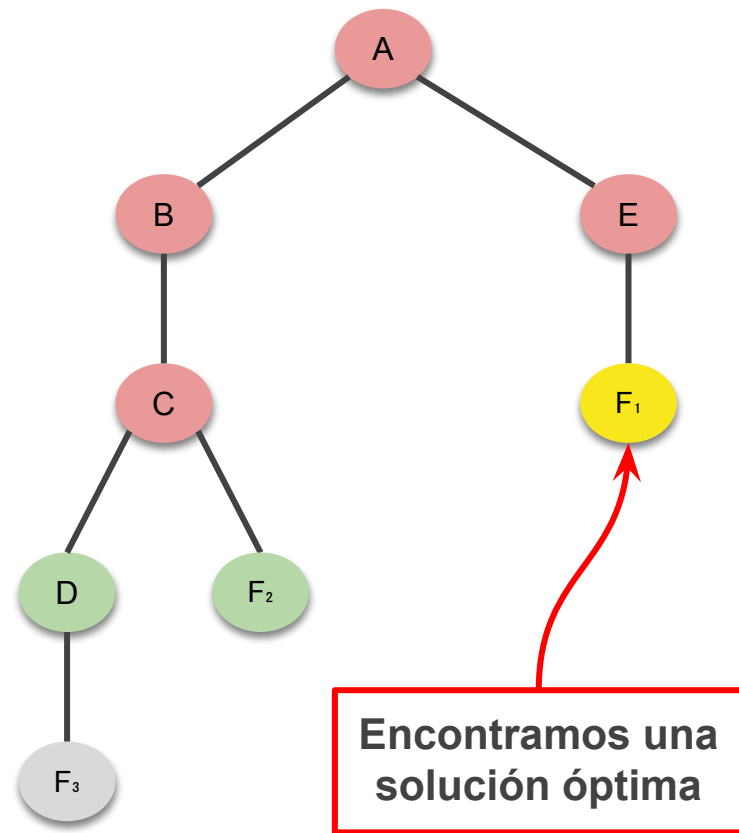




# Respuesta

— — —

Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	{B,E}
2	B	{E}	{E,C}
3	E	{C}	{C,F <sub>1</sub> }
4	C	{F <sub>1</sub> }	{F <sub>1</sub> ,D,F <sub>2</sub> }
5	{F <sub>1</sub> }	{D,F <sub>2</sub> }	<b>FIN</b>





# Implementación de BFS

— — —

- ¿Cómo elegimos de la frontera el próximo nodo a expandir?

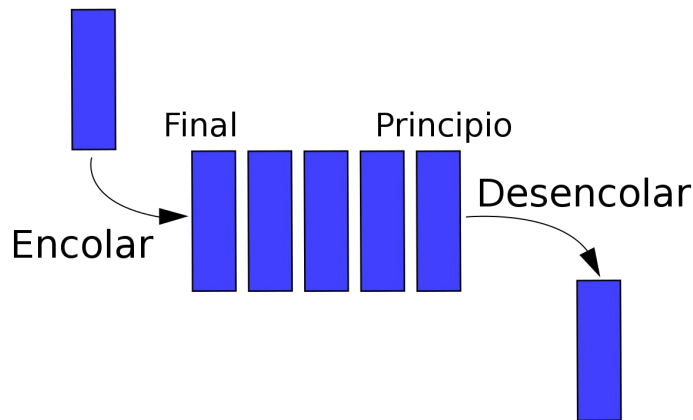
El de menor profundidad.

- ¿Cómo lo logramos?

Usando el TAD **cola** para la frontera.

# Cola — El primero en entrar es el primero en salir

— — —



Los nuevos nodos (que están a mayor profundidad que sus padres) van al final de la cola, y los nodos más viejos (que están a menor profundidad que los nuevos nodos) quedan adelante y se expanden primero.

# Algoritmo BFS en árboles

```
1 function TREE-BFS(problema) return solución o fallo
2   raíz ← Nodo(estados = problema.estado-inicial, costo = 0)
3   if (problema.test-objetivo(raíz.estado)) then return solución(raíz)
4   frontera ← Cola()
5   frontera.encolar(raíz)
6   do
7     if (frontera.vacía()) then return fallo
8     nodo ← frontera.desencolar()
9     forall acción in problema.acciones(nodo.estado) do
10       hijo ← Nodo(estados = problema.resultado(nodo.estado, acción),
11                    costo = nodo.costo + problema.c(nodo.estado, acción),
12                    padre = nodo, acción = acción)
11       if (problema.test-objetivo(hijo.estado)) then return solución(hijo)
12       frontera.encolar(hijo)
```



# Algoritmo BFS en árboles

El test objetivo se ejecuta antes de encolar el nodo para evitar expandir nodos de más.

```
1 function TREE-BFS(problema) return solución o fallo
2   raíz ← Nodo(estados = problema.estado-inicial, costo = 0)
3   if (problema.test-objetivo(raíz.estado)) then return solución(raíz)
4   frontera ← Cola()
5   frontera.encolar(raíz)
6   do
7     if (frontera.vacía()) then return fallo
8     nodo ← frontera.desencolar()
9     forall acción in problema.acciones(nodo.estado) do
10       hijo ← Nodo(estados = problema.resultado(nodo.estado, acción),
11                  costo = nodo.costo + problema.c(nodo.estado, acción),
12                  padre = nodo, acción = acción)
13       if (problema.test-objetivo(hijo.estado)) then return solución(hijo)
14       frontera.encolar(hijo)
```

# Performance de TREE-BFS

— — —

- **Compleitud.** 
- **Optimalidad.**  (si todas las acciones tienen el mismo costo).
- **Tiempo.** Se generan  $1 + b + b^2 + \dots + b^d \approx b^d$  nodos.
- **Memoria.** Se mantienen a lo sumo  $b^d$  nodos en la frontera.

Donde ***b*** es el factor de ramificación (máximo número de hijos de cualquier nodo) y ***d*** es la menor profundidad de un nodo objetivo.

# Tiempos vs. memoria

— — —

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	$10^6$	1.1 seconds	1 gigabyte
8	$10^8$	2 minutes	103 gigabytes
10	$10^{10}$	3 hours	10 terabytes
12	$10^{12}$	13 days	1 petabyte
14	$10^{14}$	3.5 years	99 petabytes
16	$10^{16}$	350 years	10 exabytes

Se asume  $b = 10$ , 1M de nodos generados por segundo y 1 kb de memoria por nodo.

# Tiempos vs. memoria

— — —

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	$10^6$	1.1 seconds	1 gigabyte
8	$10^8$	2 minutes	103 gigabytes
10	$10^{10}$	3 hours	10 terabytes
12	$10^{12}$	13 days	1 petabyte
14	$10^{14}$	3.5 years	99 petabytes
16	$10^{16}$	350 years	10 exabytes

Se asume  $b = 10$ , 1M de nodos generados por segundo y 1 kb de memoria por nodo.

**El requerimiento de memoria es una complicación más seria que el de tiempo.**

# Algoritmo BFS en grafos

— — —

Muy parecido a BFS en árboles, salvo que:

1. Se mantiene un **conjunto de estados alcanzados**.

Antes de agregar un nodo a la frontera, se marca su estado como alcanzado.

2. Se **descarta** cualquier nodo generado con un estado ya alcanzado (ya que su costo de camino es mayor o igual al del primer nodo encontrado).

Mantiene la completitud y optimalidad... pero los requerimientos de tiempo y memoria ahora son proporcionales al tamaño del espacio de estados.



# Algoritmo BFS en grafos

```
1 function GRAPH-BFS(problema) return solución o fallo
2   raíz ← Nodo(estado = problema.estado-inicial, costo = 0)
3   if (problema.test-objetivo(raíz.estado)) then return solución(raíz)
4   frontera ← Cola()
5   frontera.encolar(raíz)
6   alcanzados ← {raíz.estado}
7   do
8     if (frontera.vacía()) then return fallo
9     nodo ← frontera.desencolar()
10    forall acción in problema.acciones(nodo.estado) do
11      hijo ← Nodo(estado = problema.resultado(nodo.estado, acción),
12                  costo = nodo.costo + problema.c(nodo.estado, acción),
13                  padre = nodo, acción = acción)
14      if (problema.test-objetivo(hijo.estado)) then return solución(hijo)
15      if hijo.estado is not in alcanzados then
16        alcanzados.insertar(hijo.estado)
17        frontera.encolar(hijo)
```

# Búsqueda primero en profundidad

Depth-First Search (DFS)

Los nodos del árbol de búsqueda se expanden por profundidad.

Comenzando por la raíz, se expande siempre el nodo más profundo de la frontera.

La búsqueda desciende rápidamente a una hoja. Para continuar, se retrocede al nodo más profundo con hijos aún no expandidos y se elige uno de ellos.

— — —



# Ejemplo

— — —

A: (8,0,0)

Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1			
2			
3			
4			
5			
6			



# Ejemplo

— — —

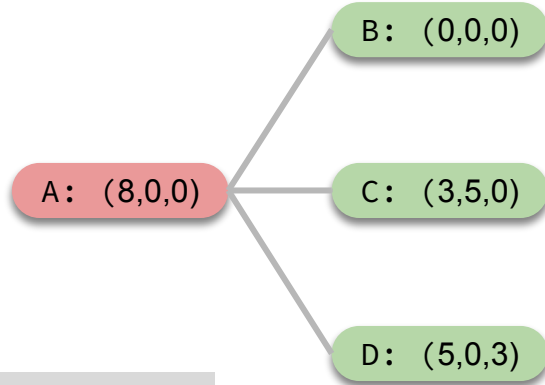
A: (8,0,0)

Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	
2			
3			
4			
5			
6			



# Ejemplo

— — —

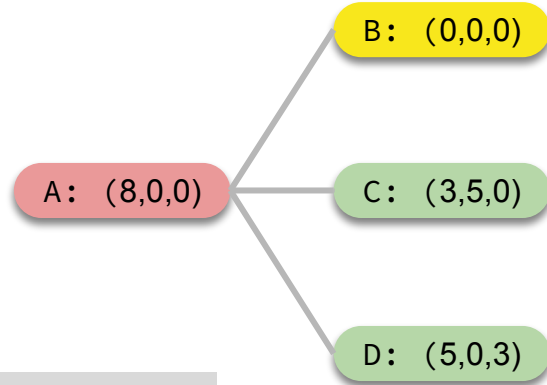


Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	{B,C,D}
2			
3			
4			
5			
6			



# Ejemplo

— — —

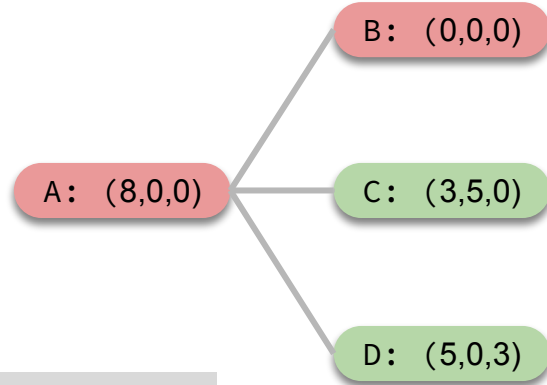


Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	{B,C,D}
2	B	{C,D}	
3			
4			
5			
6			



# Ejemplo

— — —

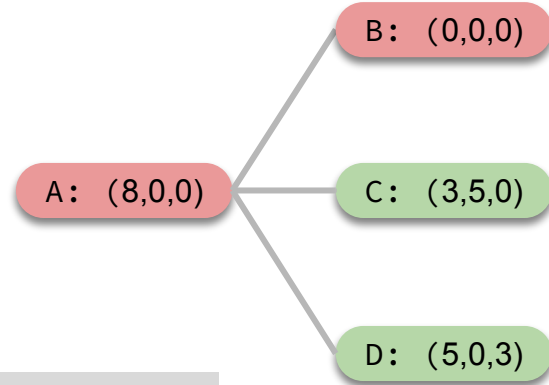


Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	{B,C,D}
2	B	{C,D}	{C,D}
3			
4			
5			
6			



# Ejemplo

— — —



**Llegamos a una hoja,  
la búsqueda retrocede  
al nodo A y se elige  
un hijo de A no  
expandido.**

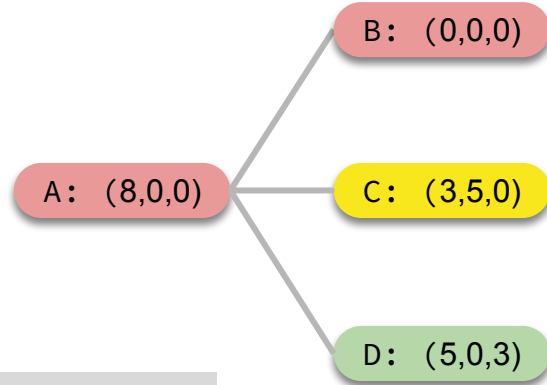
Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	{B,C,D}
2	B	{C,D}	{C,D}
3			
4			
5			
6			





# Ejemplo

— — —

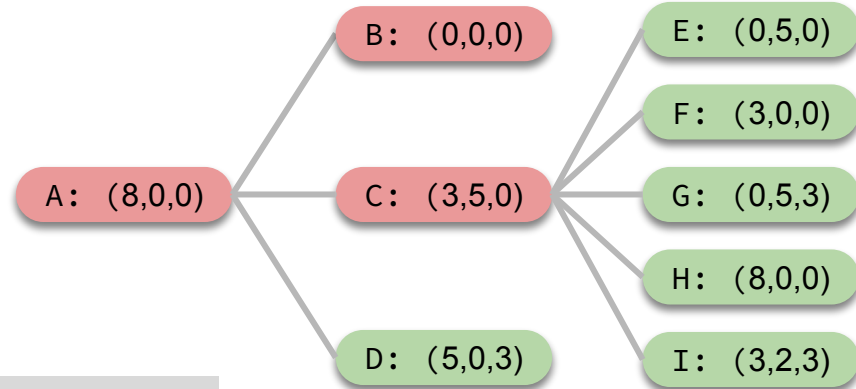


Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	{B,C,D}
2	B	{C,D}	{C,D}
3	C	{D}	
4			
5			
6			



# Ejemplo

— — —

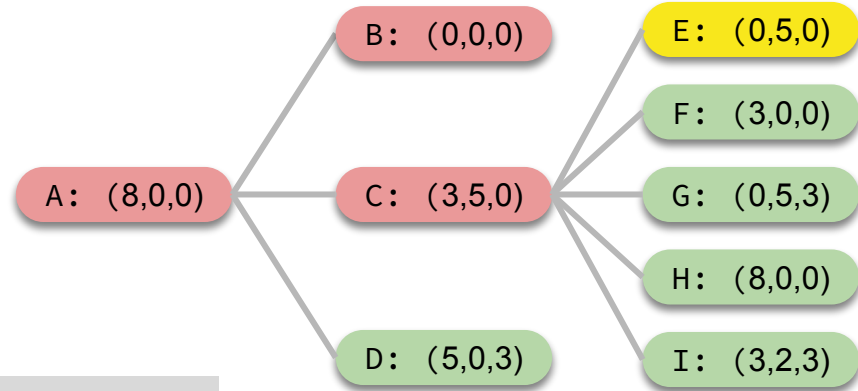


Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	{B,C,D}
2	B	{C,D}	{C,D}
3	C	{D}	{E,F,G,H,I,D}
4			
5			
6			



# Ejemplo

— — —

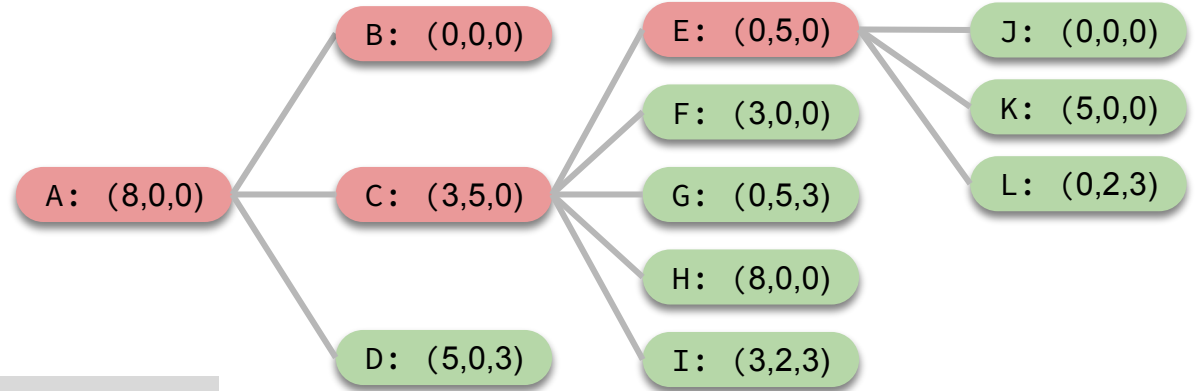


Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	{B,C,D}
2	B	{C,D}	{C,D}
3	C	{D}	{E,F,G,H,I,D}
4	E	{F,G,H,I,D}	
5			
6			



# Ejemplo

— — —

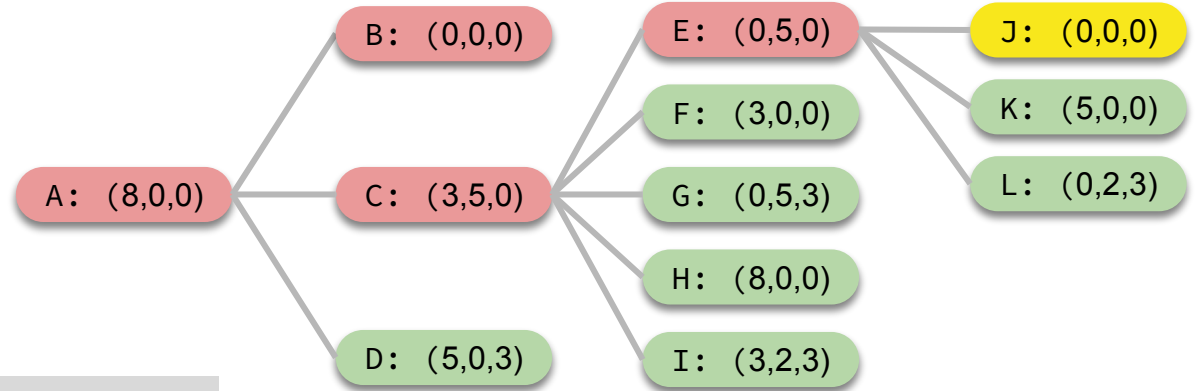


Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	{B,C,D}
2	B	{C,D}	{C,D}
3	C	{D}	{E,F,G,H,I,D}
4	E	{F,G,H,I,D}	{J,K,L,F,G,H,I,D}
5			
6			



# Ejemplo

— — —

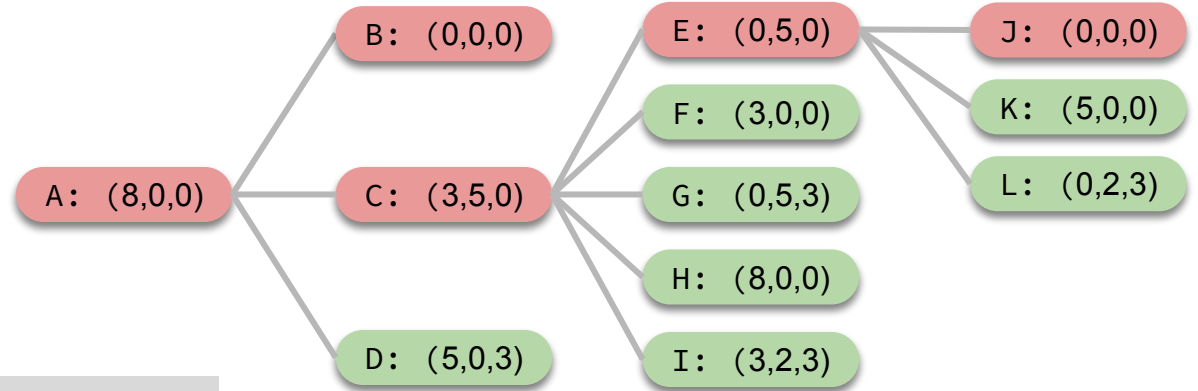


Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	{B,C,D}
2	B	{C,D}	{C,D}
3	C	{D}	{E,F,G,H,I,D}
4	E	{F,G,H,I,D}	{J,K,L,F,G,H,I,D}
5	J	{K,L,F,G,H,I,D}	
6			



# Ejemplo

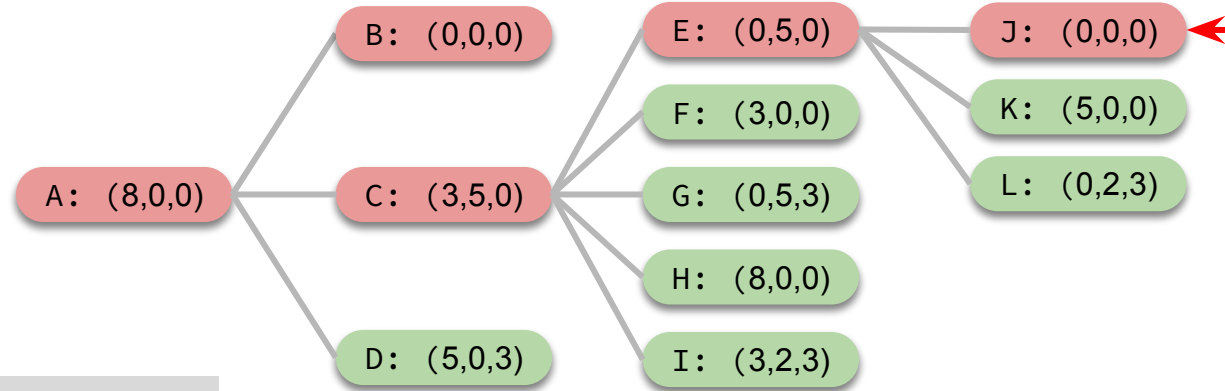
— — —



Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	{B,C,D}
2	B	{C,D}	{C,D}
3	C	{D}	{E,F,G,H,I,D}
4	E	{F,G,H,I,D}	{J,K,L,F,G,H,I,D}
5	J	{K,L,F,G,H,I,D}	{K,L,F,G,H,I,D}
6			



# Ejemplo



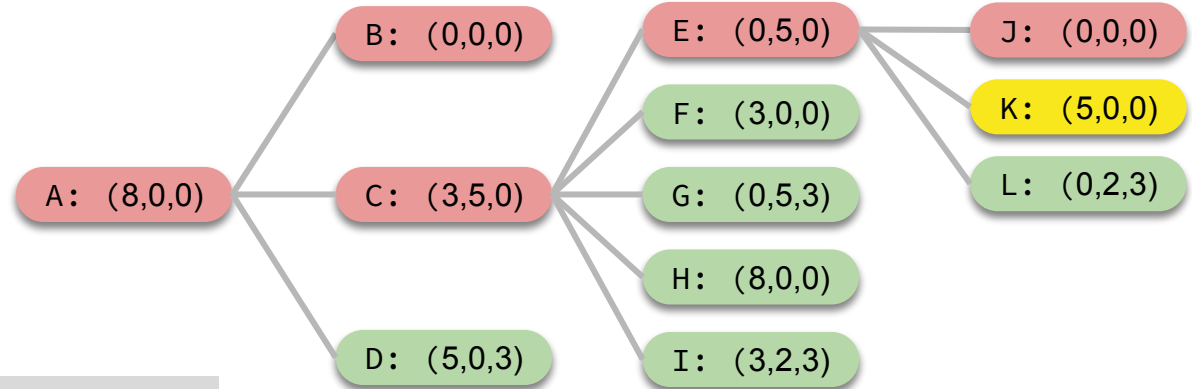
**Llegamos a una hoja,  
la búsqueda retrocede  
al nodo E y se elige  
un hijo de E no  
expandido.**

Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	{B,C,D}
2	B	{C,D}	{C,D}
3	C	{D}	{E,F,G,H,I,D}
4	E	{F,G,H,I,D}	{J,K,L,F,G,H,I,D}
5	J	{K,L,F,G,H,I,D}	{K,L,F,G,H,I,D}
6			



# Ejemplo

— — —



Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	{B,C,D}
2	B	{C,D}	{C,D}
3	C	{D}	{E,F,G,H,I,D}
4	E	{F,G,H,I,D}	{J,K,L,F,G,H,I,D}
5	J	{K,L,F,G,H,I,D}	{K,L,F,G,H,I,D}
6	K	{L,F,G,H,I,D}	...CONTINÚA...

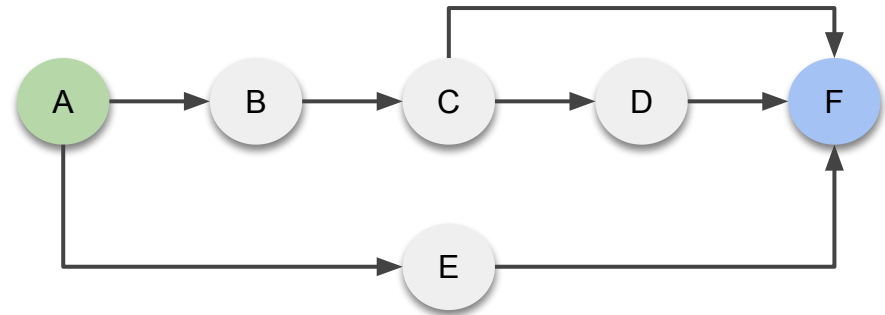




## Ejercicio

Sea un problema descrito por el siguiente espacio de estados, donde A es el estado inicial y F es el estado objetivo.

Resolverlo con el algoritmo general de búsqueda en árboles con la estrategia DFS.

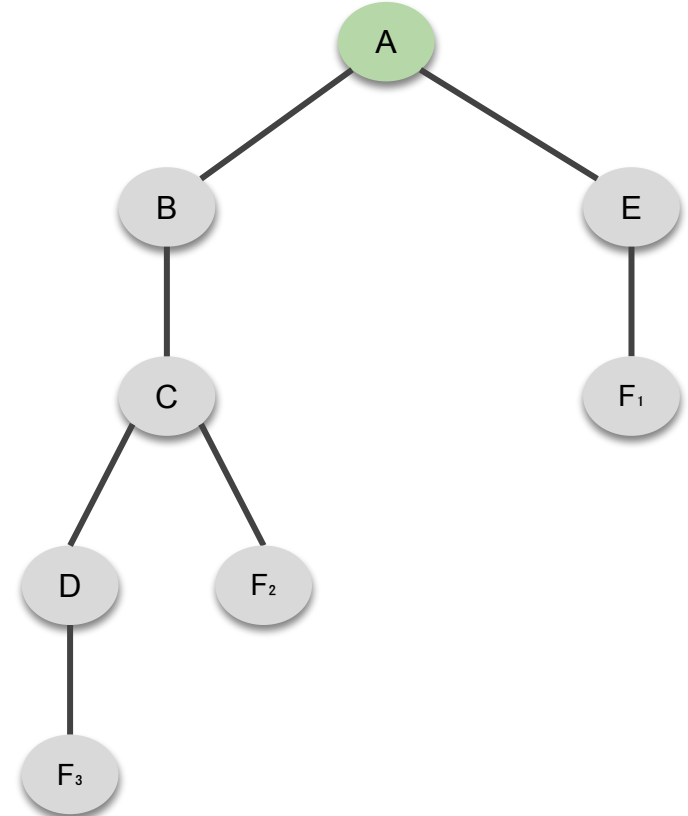




# Respuesta

— — —

Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1			
2			
3			
4			
5			

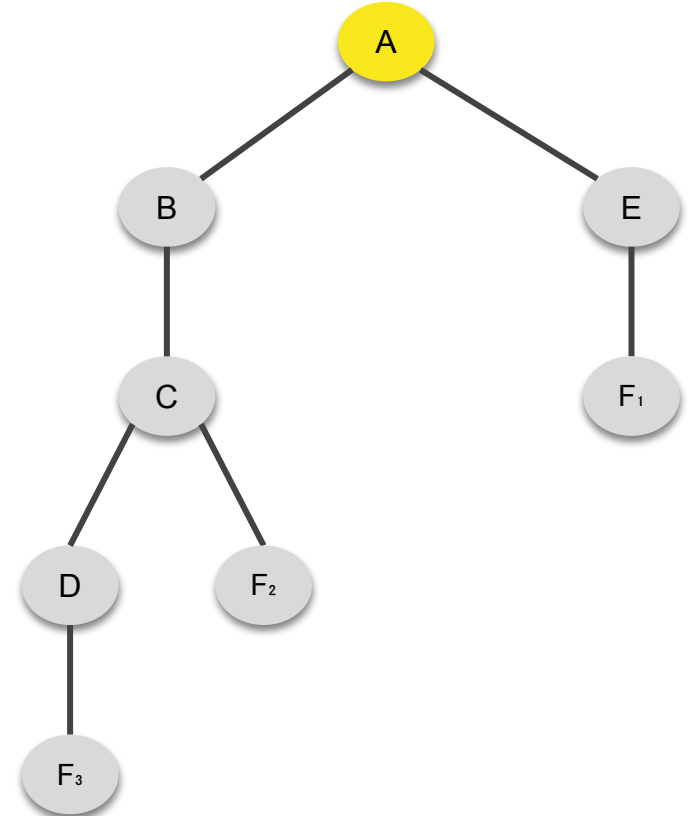




# Respuesta

— — —

Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	
2			
3			
4			
5			

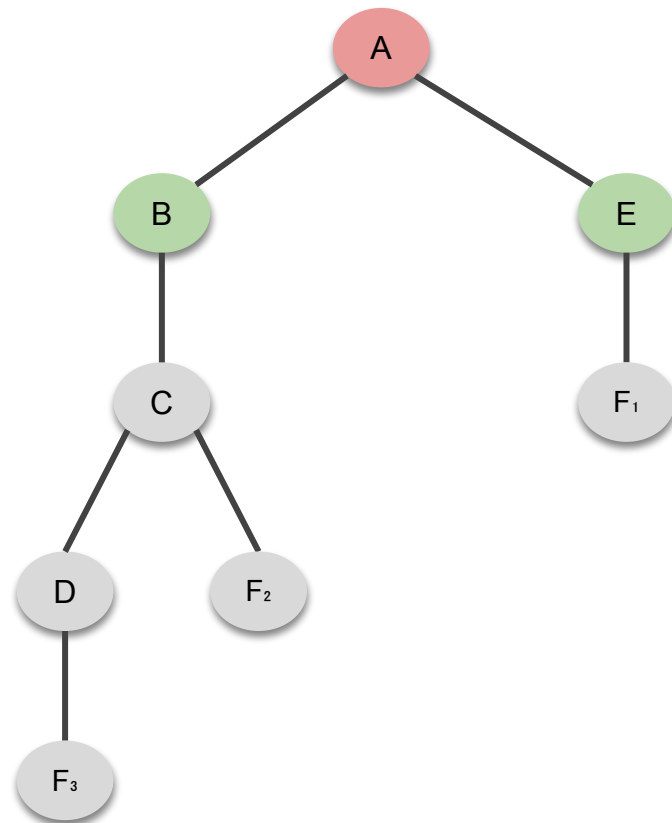




# Respuesta

— — —

Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	{B,E}
2			
3			
4			
5			

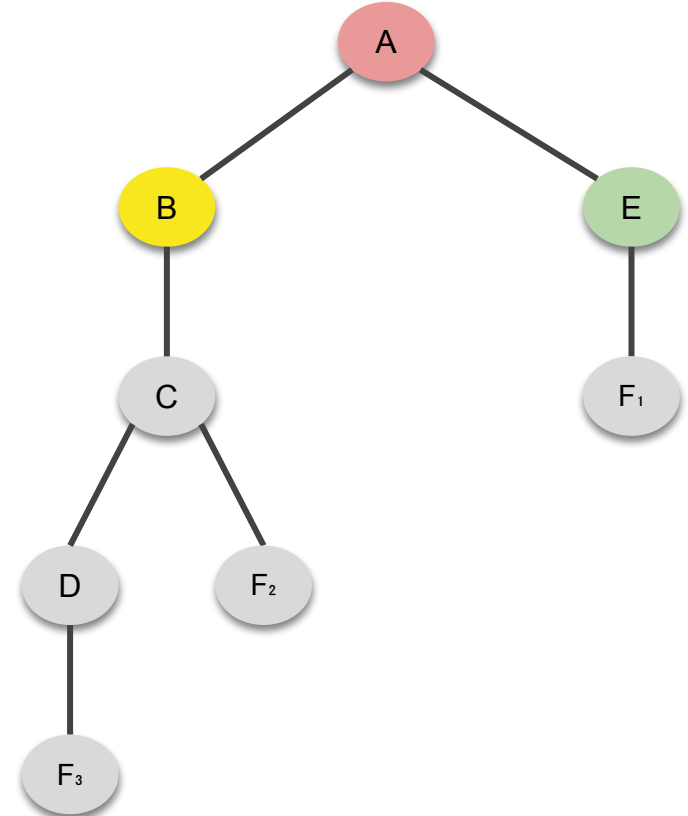




# Respuesta

— — —

Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	{B,E}
2	B	{E}	
3			
4			
5			

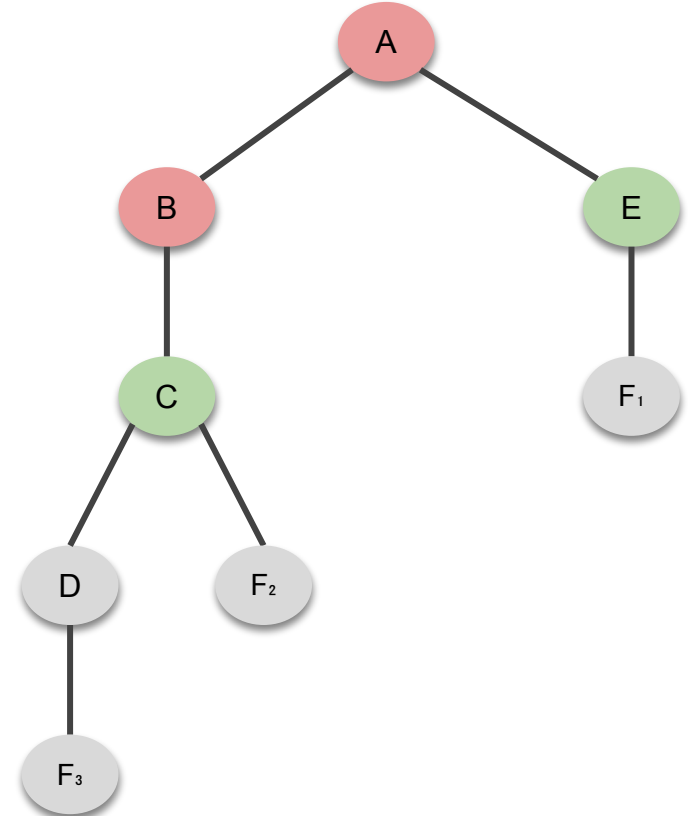




# Respuesta

— — —

Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	{B,E}
2	B	{E}	{C,E}
3			
4			
5			

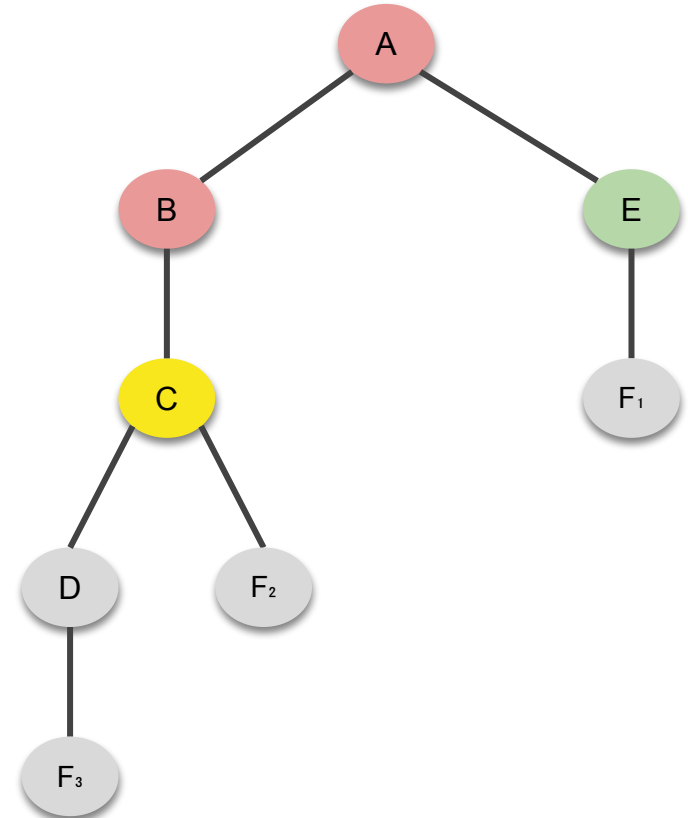




# Respuesta

— — —

Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	{B,E}
2	B	{E}	{C,E}
3	C	{E}	
4			
5			

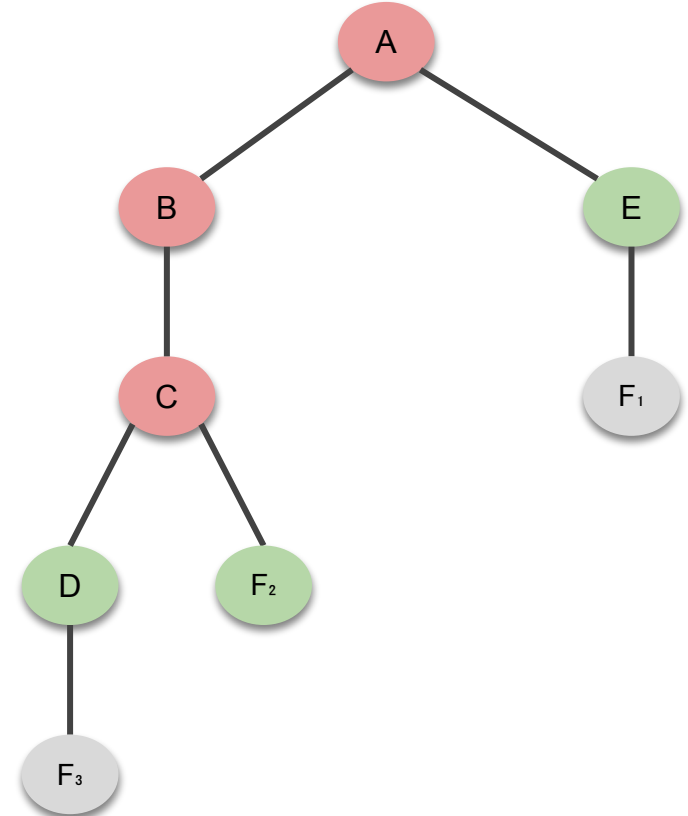




# Respuesta

— — —

Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	{B,E}
2	B	{E}	{C,E}
3	C	{E}	{D,F <sub>2</sub> ,E}
4			
5			



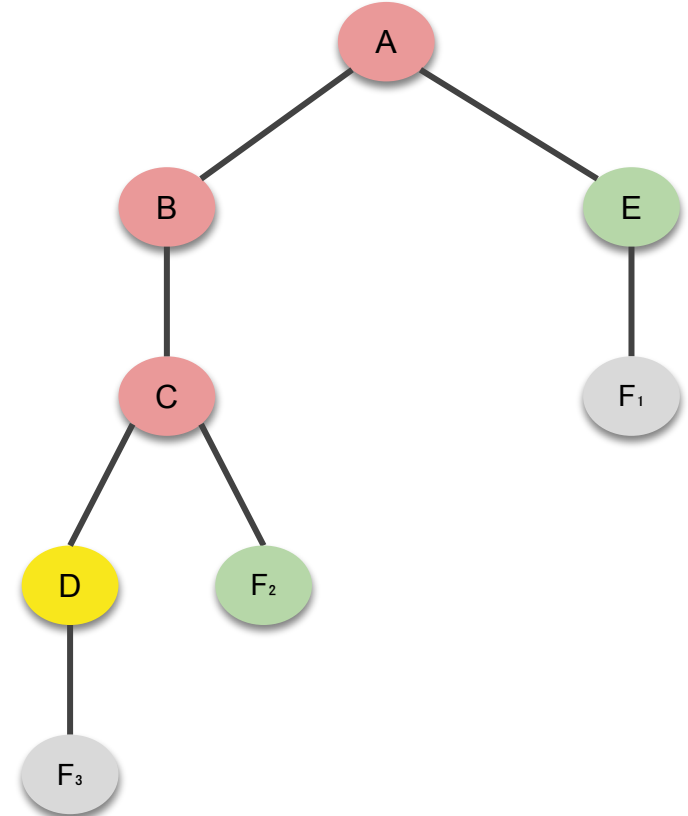




# Respuesta

— — —

Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	{B,E}
2	B	{E}	{C,E}
3	C	{E}	{D,F <sub>2</sub> ,E}
4	D	{F <sub>2</sub> ,E}	
5			

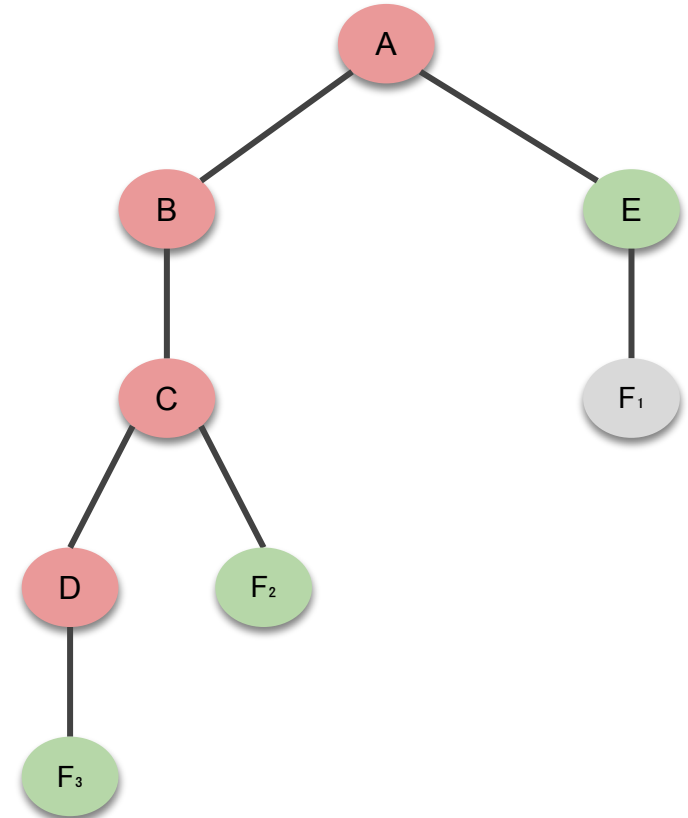




# Respuesta

— — —

Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	{B,E}
2	B	{E}	{C,E}
3	C	{E}	{D,F <sub>2</sub> ,E}
4	D	{F <sub>2</sub> ,E}	{F <sub>3</sub> ,F <sub>2</sub> ,E}
5			

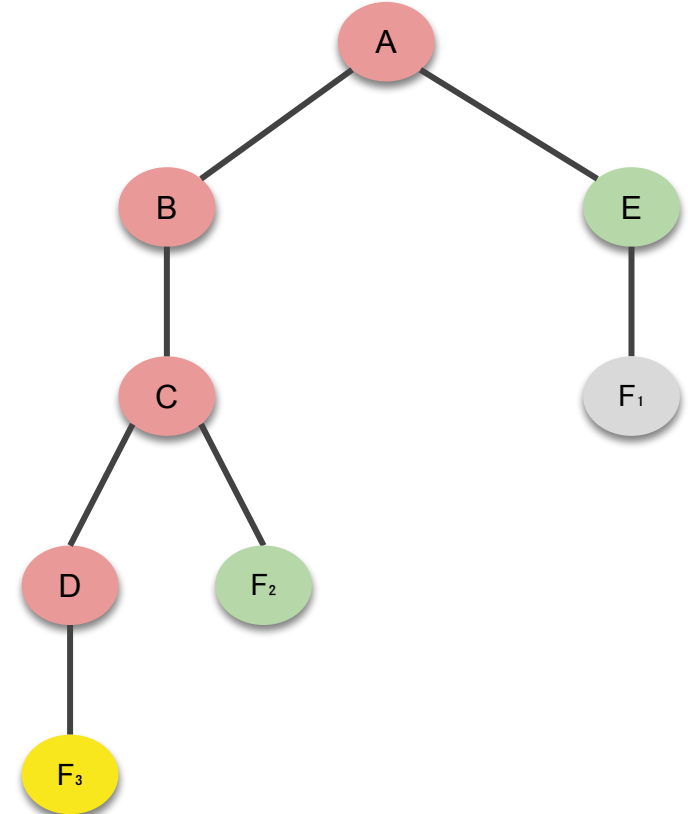




# Respuesta

— — —

Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	{B,E}
2	B	{E}	{C,E}
3	C	{E}	{D,F <sub>2</sub> ,E}
4	D	{F <sub>2</sub> ,E}	{F <sub>3</sub> ,F <sub>2</sub> ,E}
5	F <sub>3</sub>	{F <sub>2</sub> ,E}	<b>FIN</b>

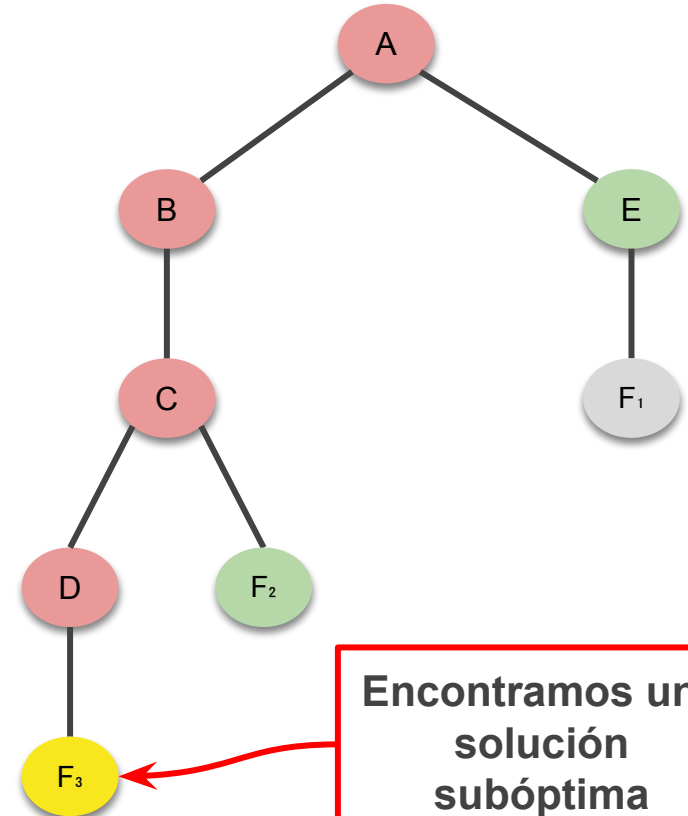




# Respuesta

— — —

Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	{B,E}
2	B	{E}	{C,E}
3	C	{E}	{D,F <sub>2</sub> ,E}
4	D	{F <sub>2</sub> ,E}	{F <sub>3</sub> ,F <sub>2</sub> ,E}
5	F <sub>3</sub>	{F <sub>2</sub> ,E}	<b>FIN</b>



# Implementación de DFS

---

- ¿Cómo elegimos de la frontera el próximo nodo a expandir?

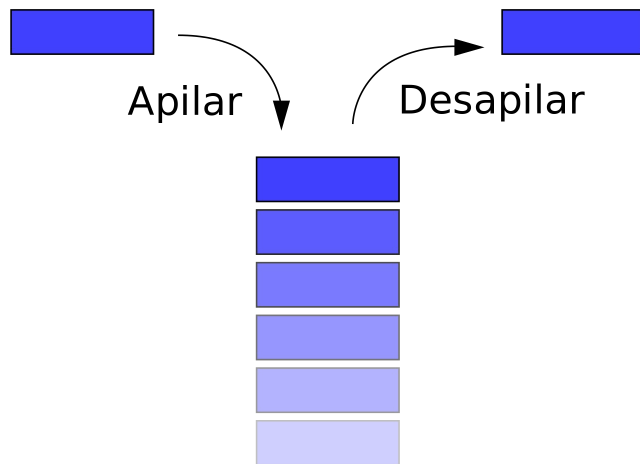
El de mayor profundidad.

- ¿Cómo lo logramos?

Usando el TAD **pila** para la frontera.

# Pila — El último en entrar es el primero en salir

— — —



Los nuevos nodos (que están a mayor profundidad que sus padres) van en el tope de la pila y se expanden primero.



# Algoritmo DFS en árboles

— — —

```
1 function TREE-DFS(problema) return solución o fallo
2   raíz ← Nodo(estado = problema.estado-inicial, costo = 0)
3   frontera ← Pila()
4   frontera.apilar(raíz)
5   do
6     if (frontera.vacía()) then return fallo
7     nodo ← frontera.desapilar()
8     if (problema.test-objetivo(nodo.estado)) then return solución(nodo)
9     forall acción in problema.acciones(nodo.estado) do
10       hijo ← Nodo(estado = problema.resultado(nodo.estado, acción),
11                  costo = nodo.costo + problema.c(nodo.estado, acción),
12                  padre = nodo, acción = acción)
13       frontera.apilar(hijo)
```

# Performance de TREE-DFS

— — —

- **Completitud.**  (si se detectan caminos cíclicos).
- **Optimalidad.** 
- **Tiempo.** Se generan  $1 + b + b^2 + \dots + b^m \approx b^m$  nodos.
- **Memoria.** Se mantienen a lo sumo  $b.m$  nodos en la frontera (el camino de la raíz a la hoja y los hijos de cada nodo del camino).

Asumimos espacios de estados finitos. De lo contrario, es **incompleto**.

Donde ***b*** es el factor de ramificación (máximo número de hijos de cualquier nodo), ***d*** es la menor profundidad de un nodo objetivo y ***m*** es el último nivel del árbol.



# BFS vs. DFS

---

Suponiendo  $b = 10$  y 1kb por nodo, en el nivel  $d = m = 16$

BFS en árboles usaba:

10 exabytes

y DFS usa:

160 kilobytes.

**El requerimiento de memoria de DFS puede llegar a ser varios órdenes de magnitud menor al de BFS, pero se pierde la garantía de optimalidad.**

# Algoritmo DFS en grafos

— — —

Muy parecido a DFS en árboles, salvo que:

1. Se mantiene un **conjunto de estados expandidos** (a diferencia de BFS en grafos que mantiene los estados alcanzados).

Al sacar un nodo de la frontera, si su estado ya fue expandido previamente se lo desecha. De lo contrario, se expande y se marca su estado como expandido.

2. No se generan nuevos nodos con estados ya expandidos.

Pero la frontera podría tener nodos con estados repetidos, aunque a lo sumo uno de ellos es expandido (el de mayor profundidad) mientras que los otros se desapilan sin expandir.

Empeora el requerimiento de memoria.

- En árboles: proporcional al largo del camino más profundo.
- En grafos: proporcional al tamaño del espacio de estados (al igual que BFS).

# Algoritmo DFS en grafos

```
1 function GRAPH-DFS(problema) return solución o fallo
2   raíz ← Nodo(estado = problema.estado-inicial, costo = 0)
3   frontera ← Pila()
4   frontera.apilar(raíz)
5   expandidos ← {}
6   do
7     if (frontera.vacía()) then return fallo
8     nodo ← frontera.desapilar()
9     if (problema.test-objetivo(nodo.estado)) then return solución(nodo)
10    if nodo.estado is not in expandidos then
11      expandidos.insertar(nodo.estado)
12      forall acción in problema.acciones(nodo.estado) do
13        hijo ← Nodo(estado = problema.resultado(nodo.estado, acción),
14                     costo = nodo.costo + problema.c(nodo.estado, acción),
15                     padre = nodo, acción = acción)
16        if hijo.estado is not in expandidos then
17          frontera.apilar(hijo)
```



# Resumen

- ❑ Vimos dos estrategias de búsqueda: BFS y DFS.
- ❑ BFS expande siempre el nodo no expandido con la menor profundidad. Es óptima para costos individuales unitarios pero su consumo de memoria puede ser excesivo.
- ❑ DFS expande siempre el nodo no expandido con la mayor profundidad. Bajo consumo de memoria (en árboles) pero no tiene garantía de optimalidad.
- ❑ En grafos, BFS y DFS tienen el mismo consumo de memoria, con lo cual se prefiere a BFS como algoritmo de búsqueda por su optimalidad.



# Próximamente

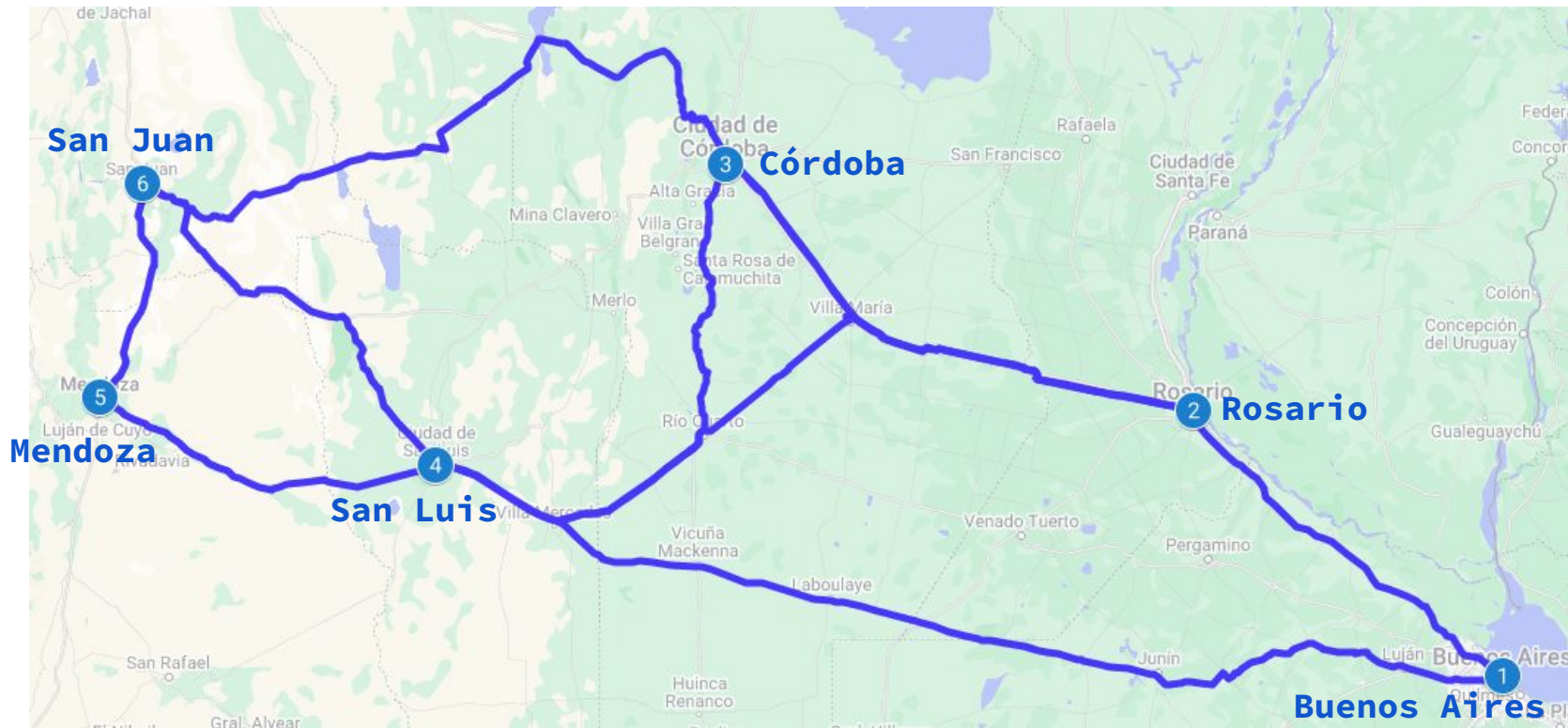
— — —

¿Es posible garantizar la optimalidad cuando los costos individuales no son unitarios?

¿Es posible que DFS garantice completitud y optimalidad incluso en espacios de estados infinitos?

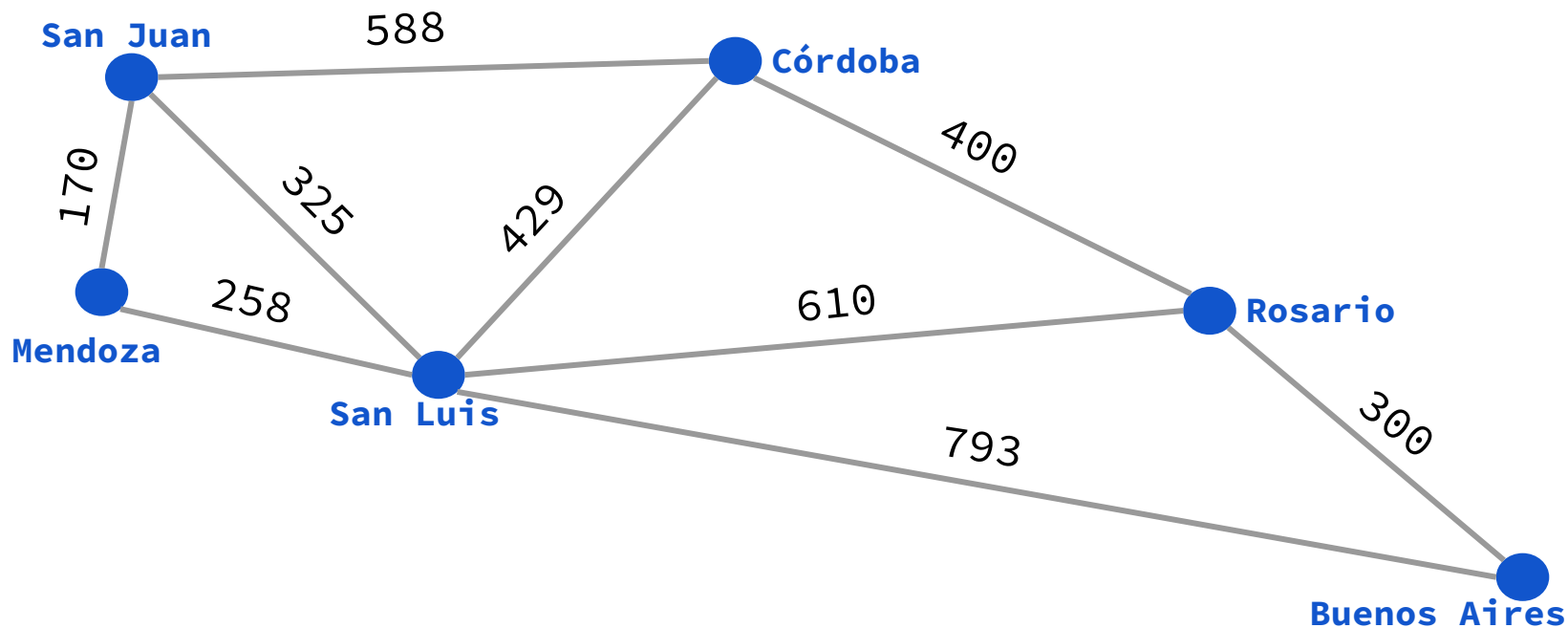
# Problema de camino más corto

— — —



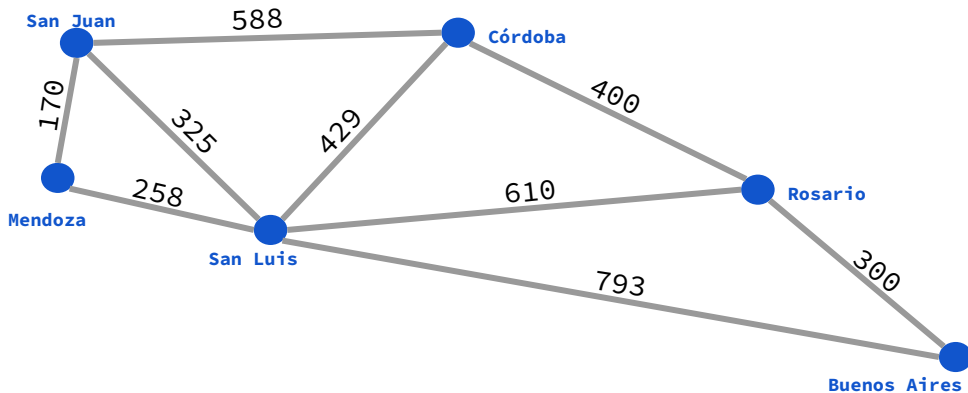
# Problema de camino más corto

---



# Descripción

— — —



**Objetivo.** Encontrar el camino más corto de Buenos Aires a San Juan.

**Reglas.** Sólo se puede mover de una ciudad a otra adyacente, es decir, conectadas con una arista.





## Ejercicio de repaso

— — —

1. Dar una formulación para este problema.
2. Encontrar una solución óptima.



# Respuesta – 1. Propuesta de formulación

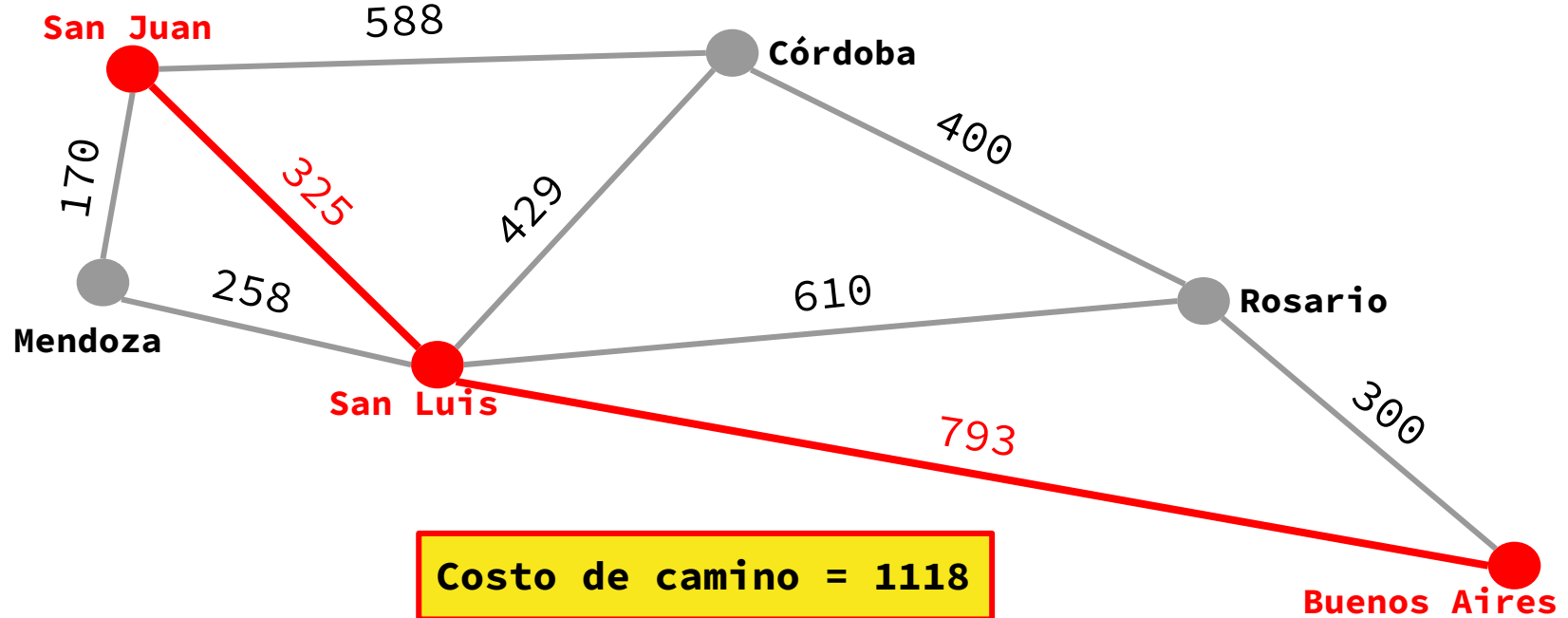
— — —

- **Estado.** Cada ciudad. Hay 6 estados.
- **Estado inicial.** Buenos Aires.
- **Acciones.**  $\rightarrow j$  viajar a la ciudad adyacente  $j$ .
- **Modelo transicional.** El resultado de  $\rightarrow j$  es  $j$ .
- **Test objetivo.** ¿ $j$  = San Juan?
- **Costo de camino.** El costo individual de  $\rightarrow j$  desde  $i$  es la distancia entre  $i$  y  $j$ . El costo de camino es la suma de los costos individuales.



## Respuesta – 2. Solución óptima

---



# Búsqueda de costo uniforme

Uniform-Cost Search (UCS)

Los nodos del árbol de búsqueda se expanden por su costo de camino.

Se expande siempre el nodo con el menor costo de camino de la frontera.

---



# Ejemplo

— — —

Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1			
2			
3			
4			
5			
6			

A:  
Bs. As.  
0



# Ejemplo

— — —

Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{ }	
2			
3			
4			
5			
6			

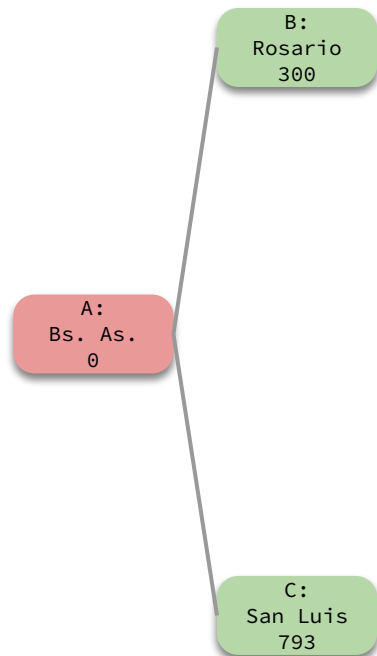
A:  
Bs. As.  
0



# Ejemplo

— — —

Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	{B,C}
2			
3			
4			
5			
6			

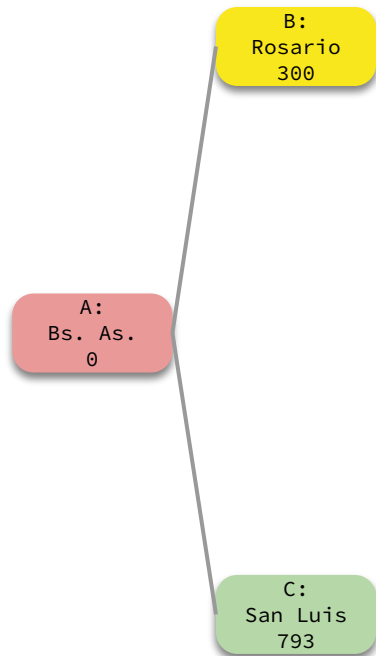




# Ejemplo

— — —

Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	{B,C}
2	B	{C}	
3			
4			
5			
6			



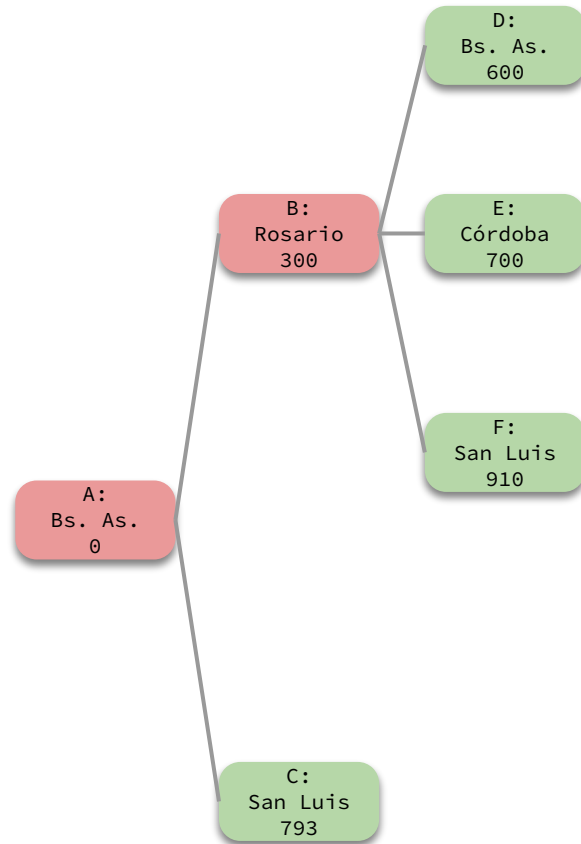




# Ejemplo

— — —

Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	{B,C}
2	B	{C}	{C,D,E,F}
3			
4			
5			
6			

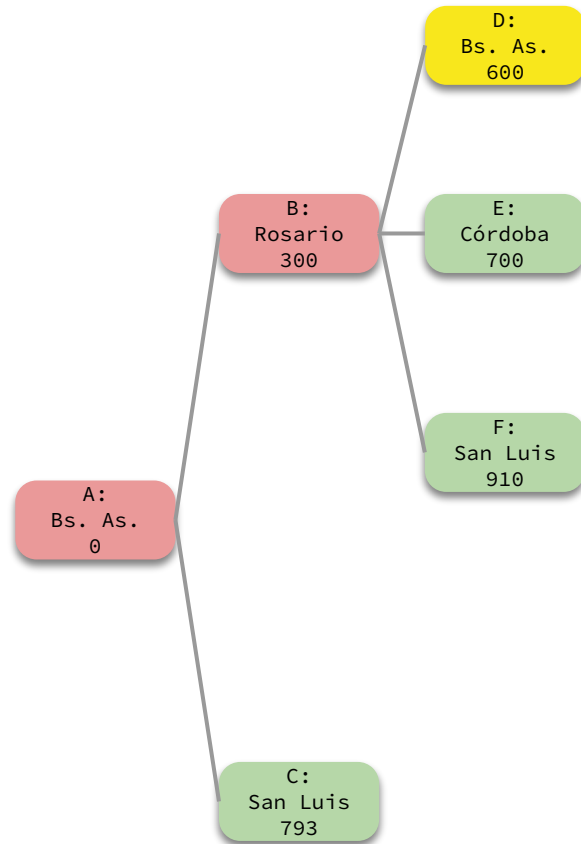




# Ejemplo

— — —

Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	{B,C}
2	B	{C}	{C,D,E,F}
3	D	{C,E,F}	
4			
5			
6			

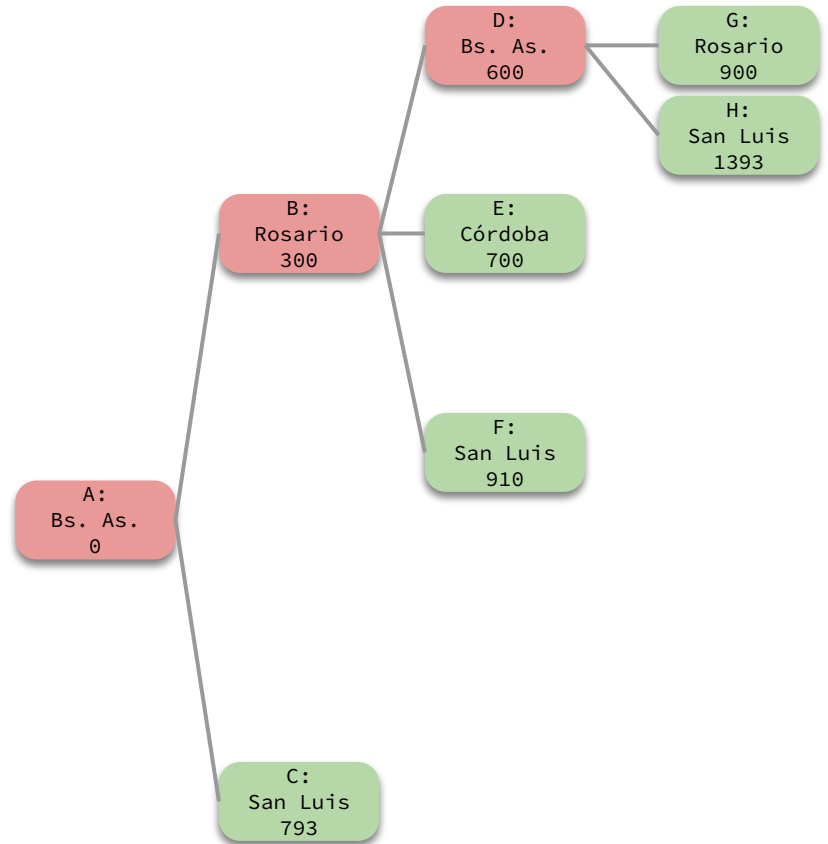




# Ejemplo

— — —

Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	{B,C}
2	B	{C}	{C,D,E,F}
3	D	{C,E,F}	{C,E,F,G,H}
4			
5			
6			

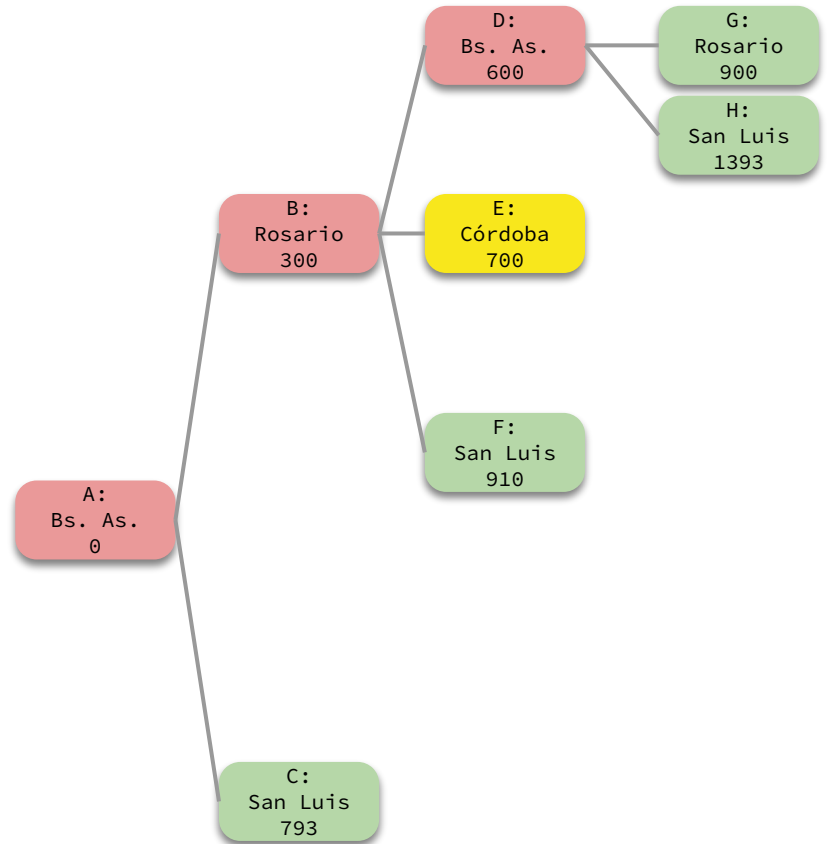




# Ejemplo

— — —

Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	{B,C}
2	B	{C}	{C,D,E,F}
3	D	{C,E,F}	{C,E,F,G,H}
4	E	{C,F,G,H}	
5			
6			

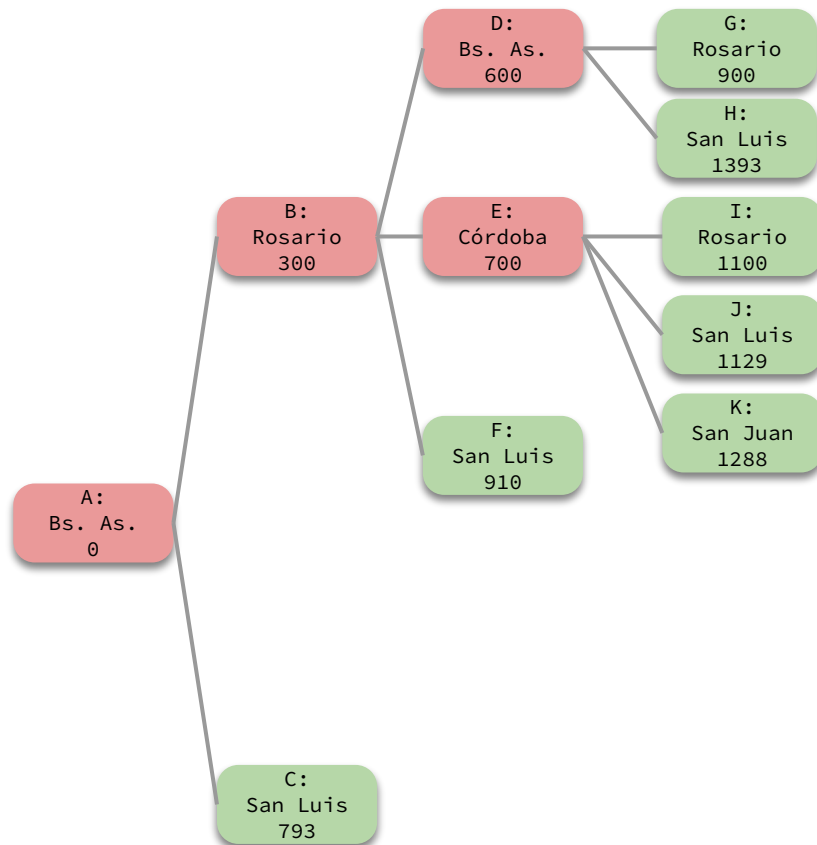




# Ejemplo

— — —

Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	{B,C}
2	B	{C}	{C,D,E,F}
3	D	{C,E,F}	{C,E,F,G,H}
4	E	{C,F,G,H}	{C,F,G,H,I,J,K}
5			
6			

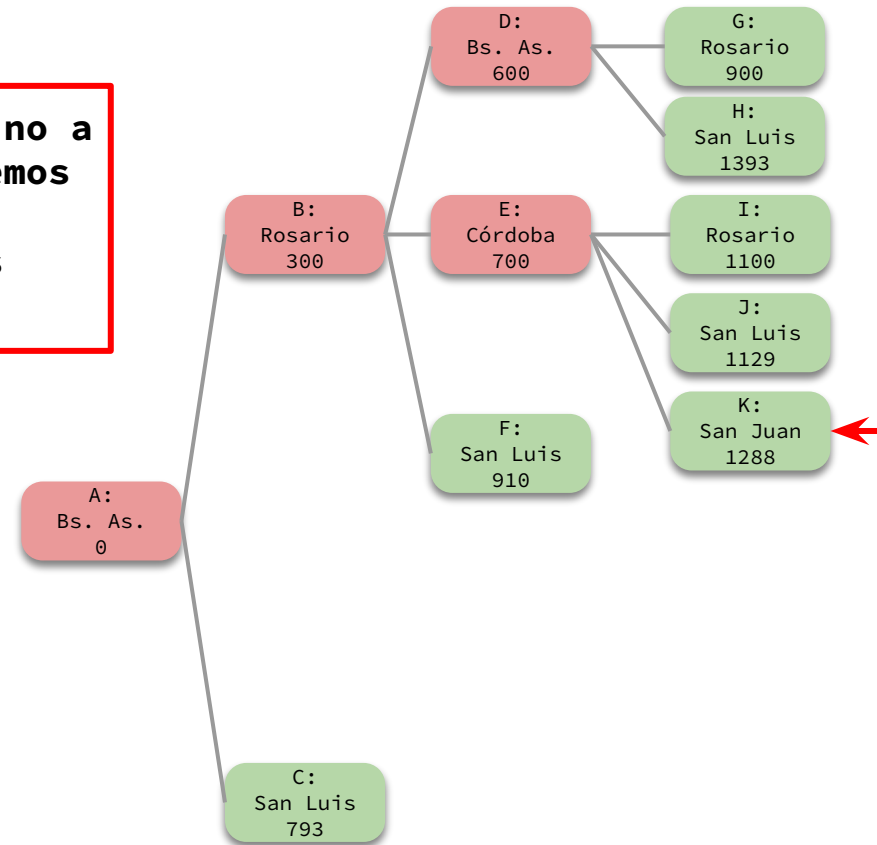




# Ejemplo

— — —

Encontramos un camino a San Juan, pero tenemos que seguir por si encontramos uno más corto.



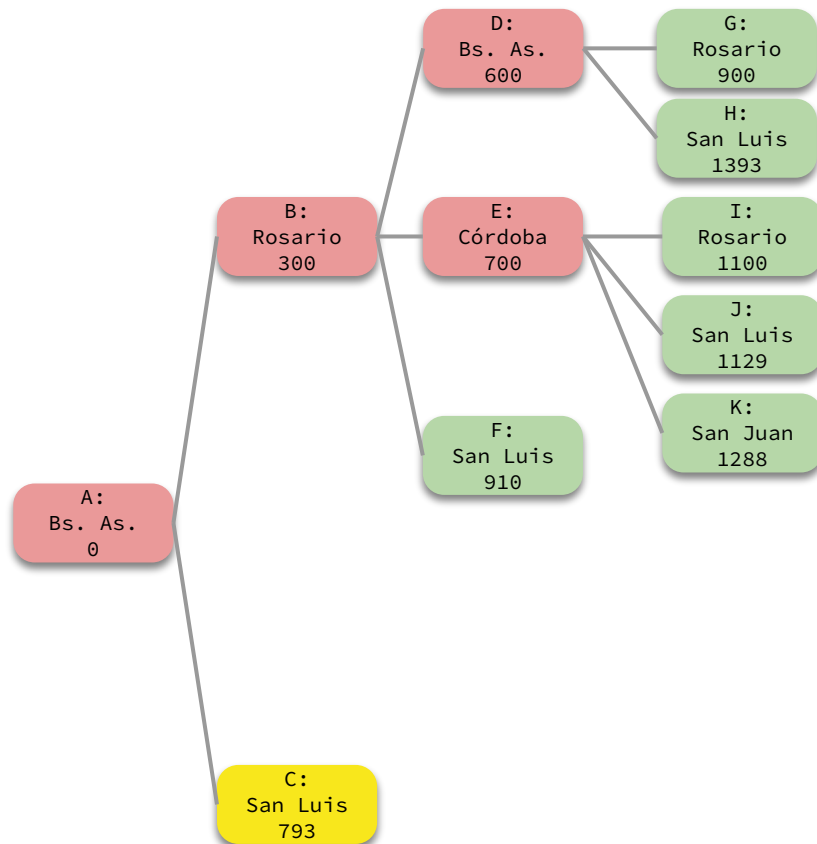
Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	{B,C}
2	B	{C}	{C,D,E,F}
3	D	{C,E,F}	{C,E,F,G,H}
4	E	{C,F,G,H}	{C,F,G,H,I,J,K}
5			
6			



# Ejemplo

— — —

Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	{B,C}
2	B	{C}	{C,D,E,F}
3	D	{C,E,F}	{C,E,F,G,H}
4	E	{C,F,G,H}	{C,F,G,H,I,J,K}
5	C	{F,G,H,I,J,K}	
6			

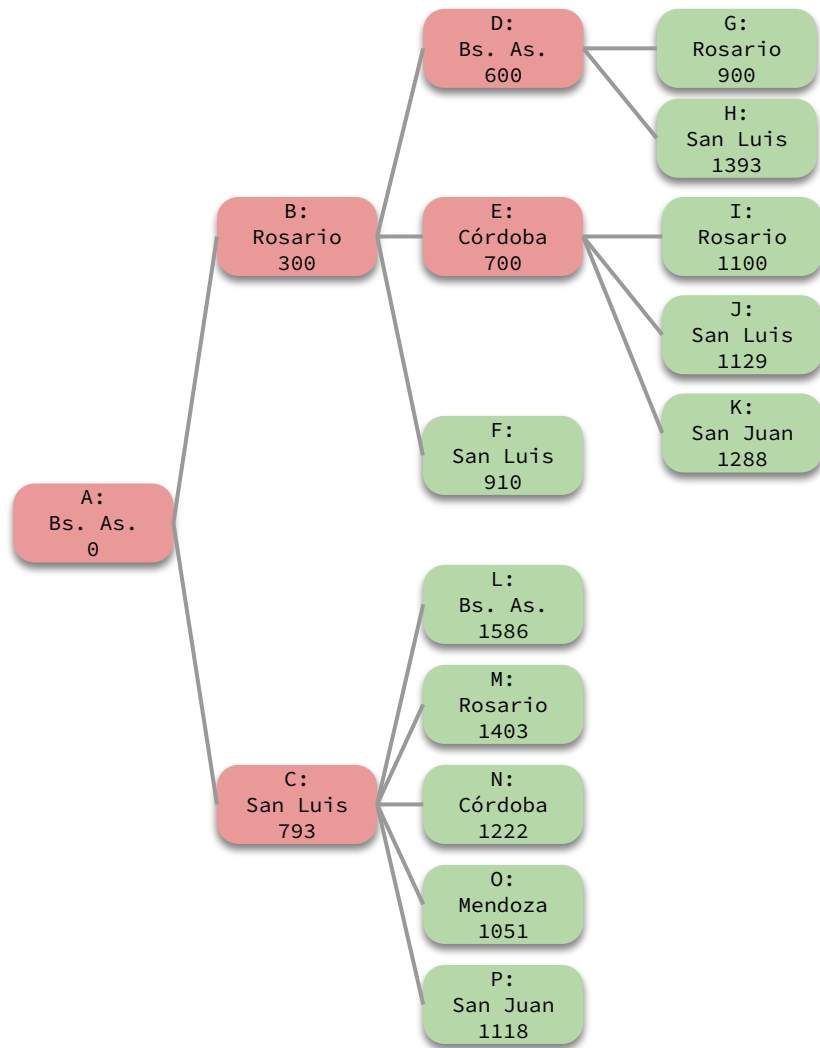




# Ejemplo

— — —

Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	{B,C}
2	B	{C}	{C,D,E,F}
3	D	{C,E,F}	{C,E,F,G,H}
4	E	{C,F,G,H}	{C,F,G,H,I,J,K}
5	C	{F,G,H,I,J,K}	{F,G,H,I,J,K,L,M,N,O,P}
6			





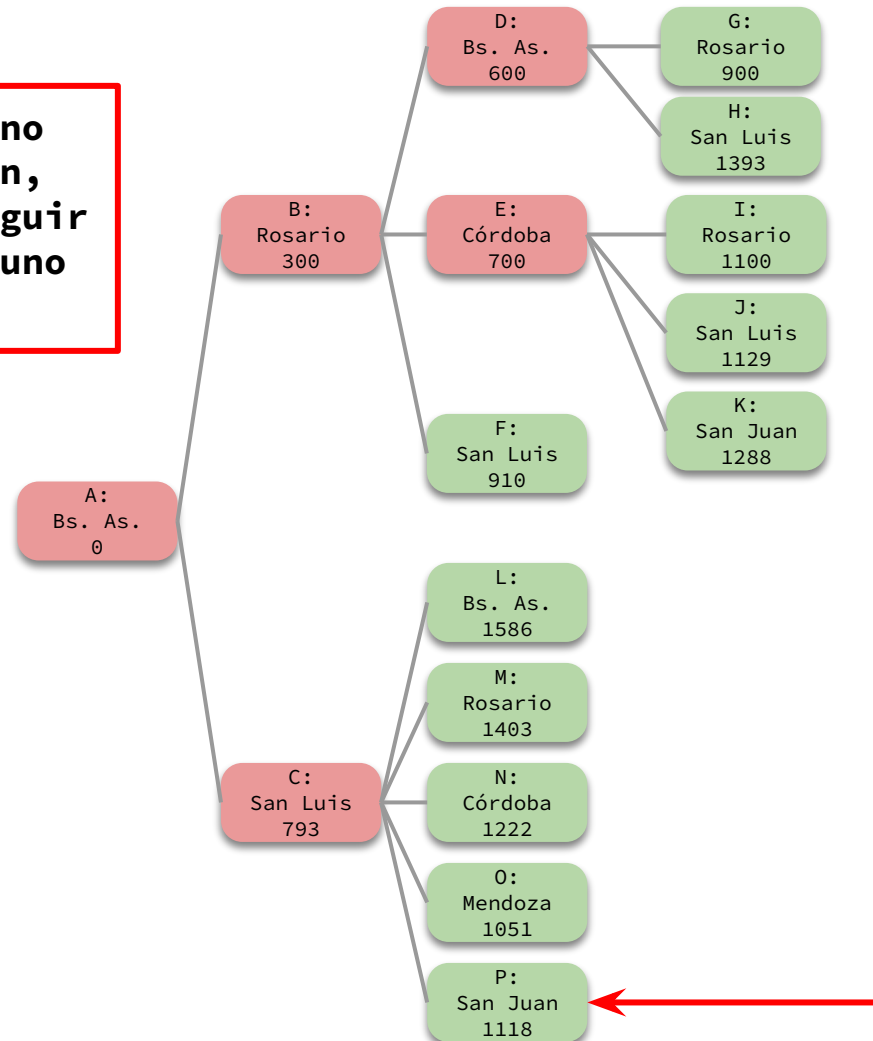


# Ejemplo

— — —

Encontramos un camino más corto a San Juan, pero tenemos que seguir por si encontramos uno aún más corto.

Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	{B,C}
2	B	{C}	{C,D,E,F}
3	D	{C,E,F}	{C,E,F,G,H}
4	E	{C,F,G,H}	{C,F,G,H,I,J,K}
5	C	{F,G,H,I,J,K}	{F,G,H,I,J,K,L,M,N,O,P}
6			

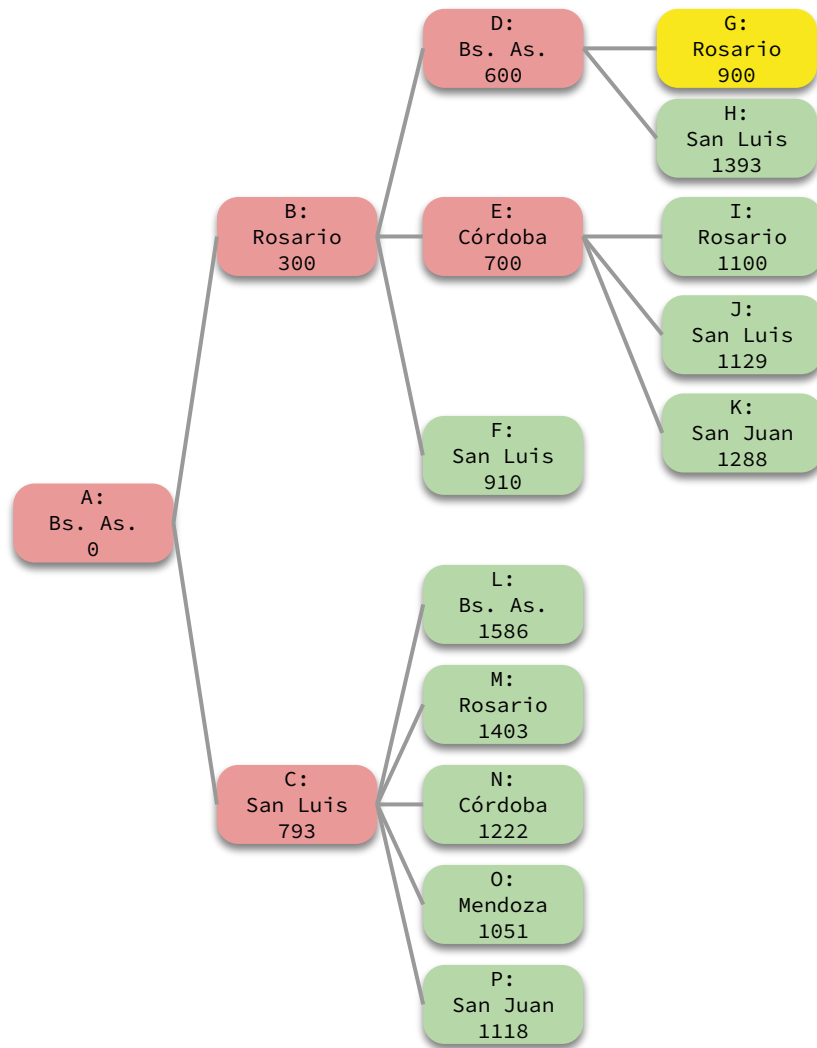




# Ejemplo

— — —

Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	{B,C}
2	B	{C}	{C,D,E,F}
3	D	{C,E,F}	{C,E,F,G,H}
4	E	{C,F,G,H}	{C,F,G,H,I,J,K}
5	C	{F,G,H,I,J,K}	{F,G,H,I,J,K,L,M,N,O,P}
6	G	{F,H,I,J,K,L,M,N,O,P}	...CONTINÚA...

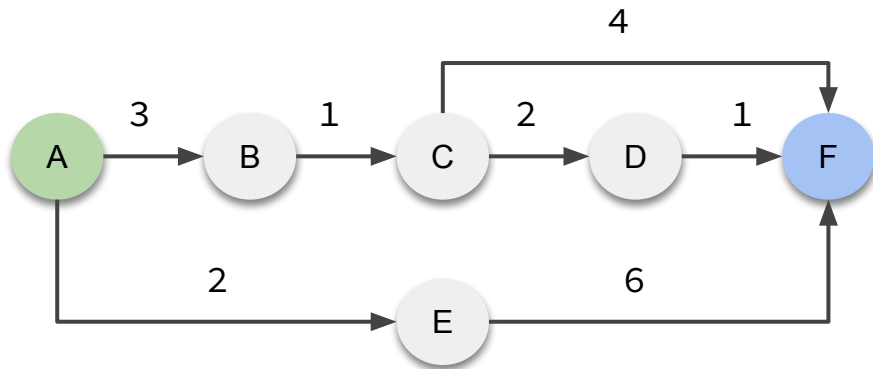




## Ejercicio

Sea un problema descrito por el siguiente espacio de estados, donde A es el estado inicial y F es el estado objetivo. El costo de cada acción se encuentra sobre el arco correspondiente.

Resolverlo con el algoritmo general de búsqueda en árboles con la estrategia UCS.





# Respuesta

— — —

A  
0

Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1			
2			
3			
4			
5			
6			



# Respuesta

— — —

A  
0

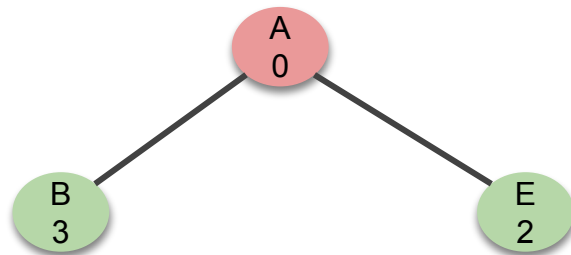
Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{ }	
2			
3			
4			
5			
6			



# Respuesta

— — —

Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	{B,E}
2			
3			
4			
5			
6			

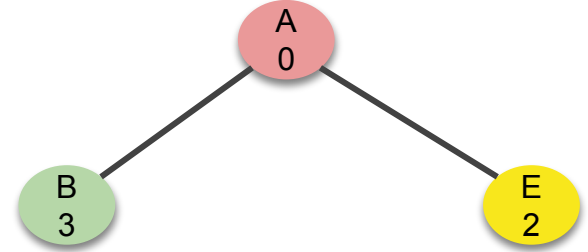




# Respuesta

— — —

Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	{B,E}
2	E	{B}	
3			
4			
5			
6			

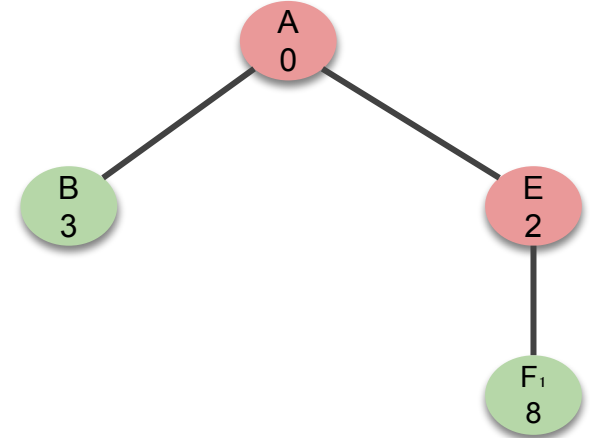




# Respuesta

— — —

Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	{B,E}
2	E	{B}	{B,F <sub>1</sub> }
3			
4			
5			
6			



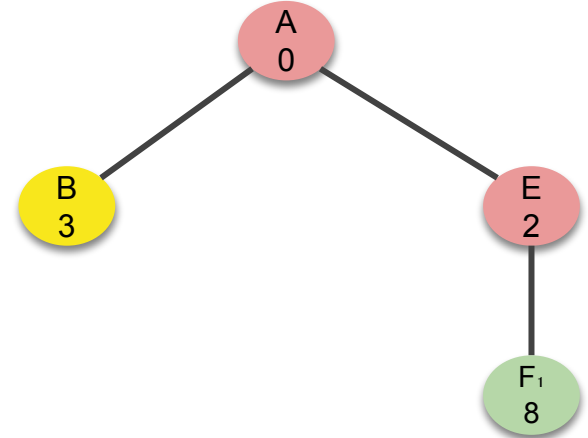




# Respuesta

— — —

Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	{B,E}
2	E	{B}	{B,F <sub>1</sub> }
3	B	{F <sub>1</sub> }	
4			
5			
6			

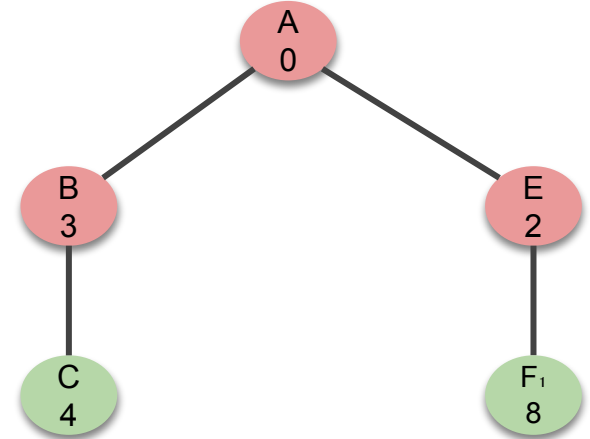




# Respuesta

— — —

Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	{B,E}
2	E	{B}	{B,F <sub>1</sub> }
3	B	{F <sub>1</sub> }	{F <sub>1</sub> ,C}
4			
5			
6			

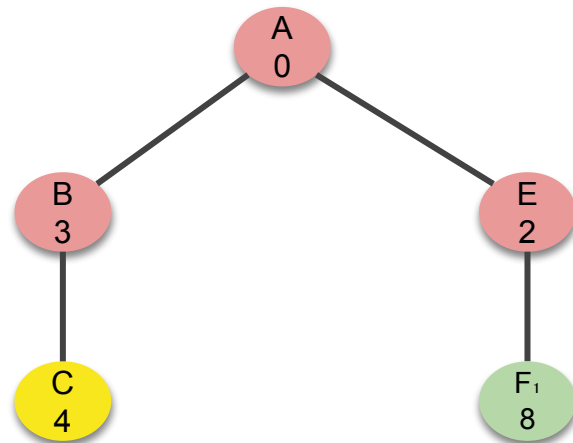




# Respuesta

— — —

Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	{B,E}
2	E	{B}	{B,F <sub>1</sub> }
3	B	{F <sub>1</sub> }	{F <sub>1</sub> ,C}
4	C	{F <sub>1</sub> }	
5			
6			

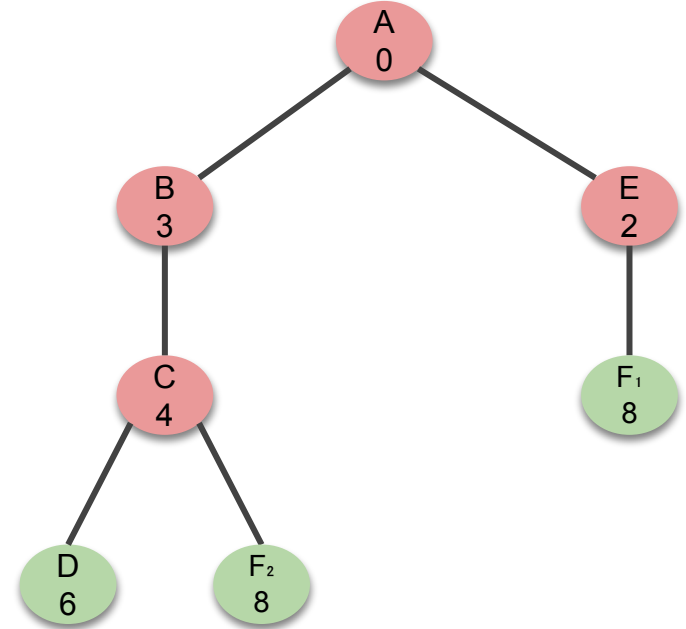




# Respuesta

— — —

Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	{B,E}
2	E	{B}	{B,F <sub>1</sub> }
3	B	{F <sub>1</sub> }	{F <sub>1</sub> ,C}
4	C	{F <sub>1</sub> }	{F <sub>1</sub> ,D,F <sub>2</sub> }
5			
6			

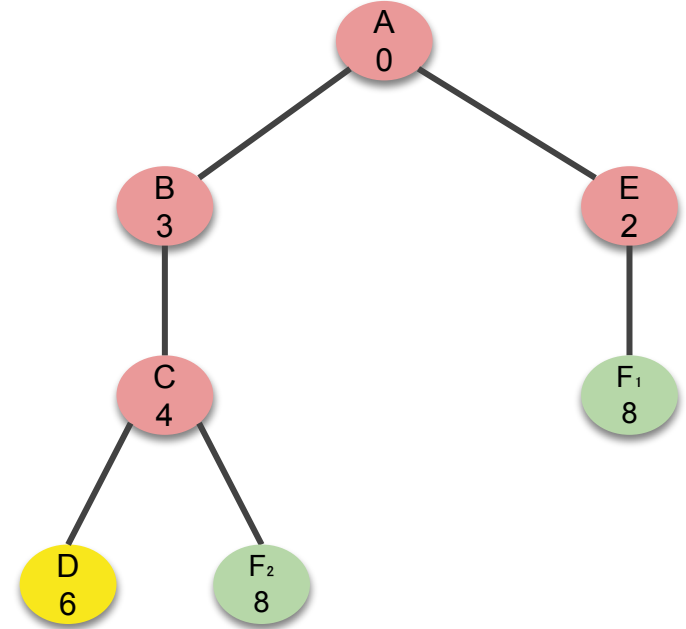




# Respuesta

— — —

Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	{B,E}
2	E	{B}	{B,F <sub>1</sub> }
3	B	{F <sub>1</sub> }	{F <sub>1</sub> ,C}
4	C	{F <sub>1</sub> }	{F <sub>1</sub> ,D,F <sub>2</sub> }
5	D	{F <sub>1</sub> ,F <sub>2</sub> }	
6			

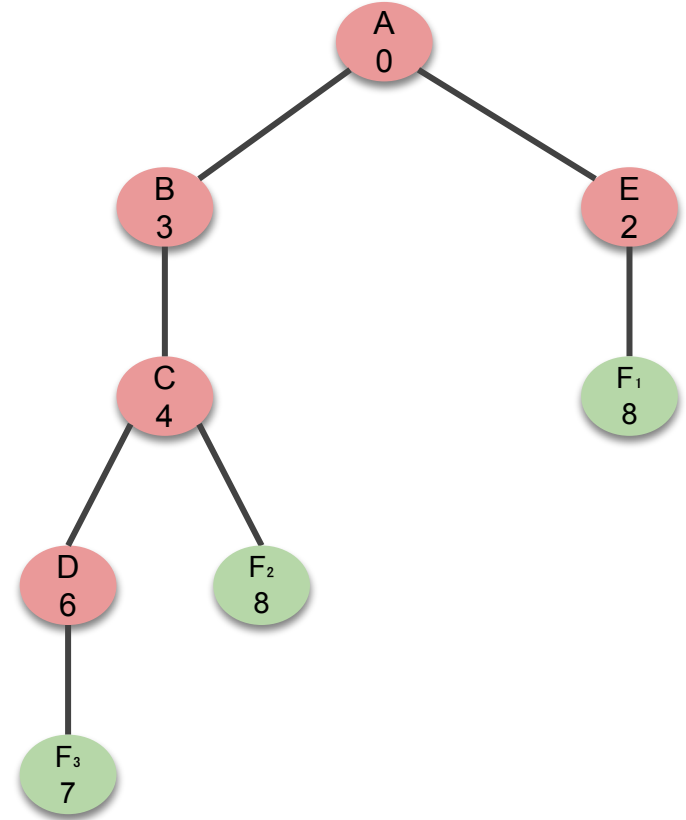




# Respuesta

— — —

Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	{B,E}
2	E	{B}	{B,F <sub>1</sub> }
3	B	{F <sub>1</sub> }	{F <sub>1</sub> ,C}
4	C	{F <sub>1</sub> }	{F <sub>1</sub> ,D,F <sub>2</sub> }
5	D	{F <sub>1</sub> ,F <sub>2</sub> }	{F <sub>1</sub> ,F <sub>2</sub> ,F <sub>3</sub> }
6			

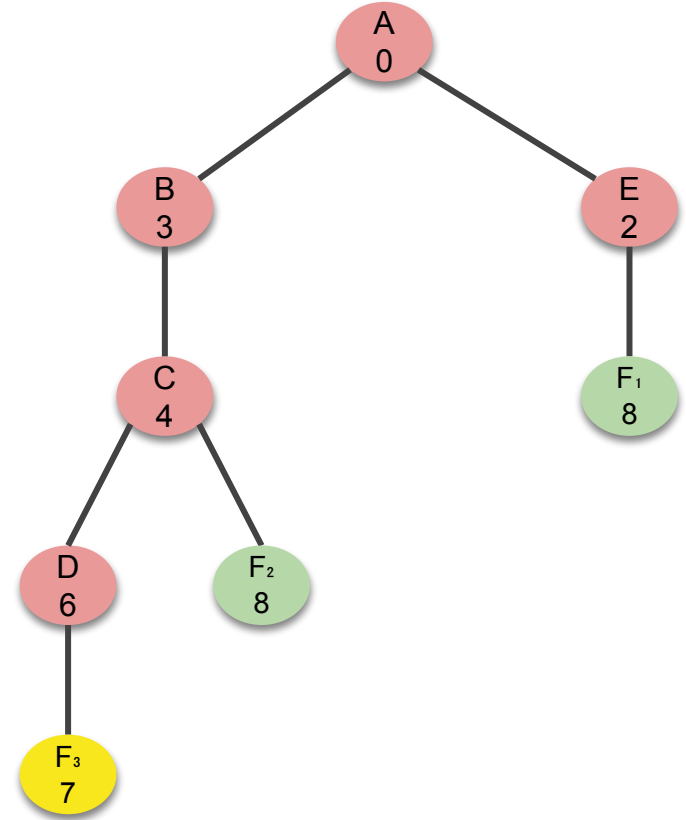




# Respuesta

— — —

Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	{B,E}
2	E	{B}	{B,F <sub>1</sub> }
3	B	{F <sub>1</sub> }	{F <sub>1</sub> ,C}
4	C	{F <sub>1</sub> }	{F <sub>1</sub> ,D,F <sub>2</sub> }
5	D	{F <sub>1</sub> ,F <sub>2</sub> }	{F <sub>1</sub> ,F <sub>2</sub> ,F <sub>3</sub> }
6	F <sub>3</sub>	{F <sub>1</sub> ,F <sub>2</sub> }	<b>FIN</b>

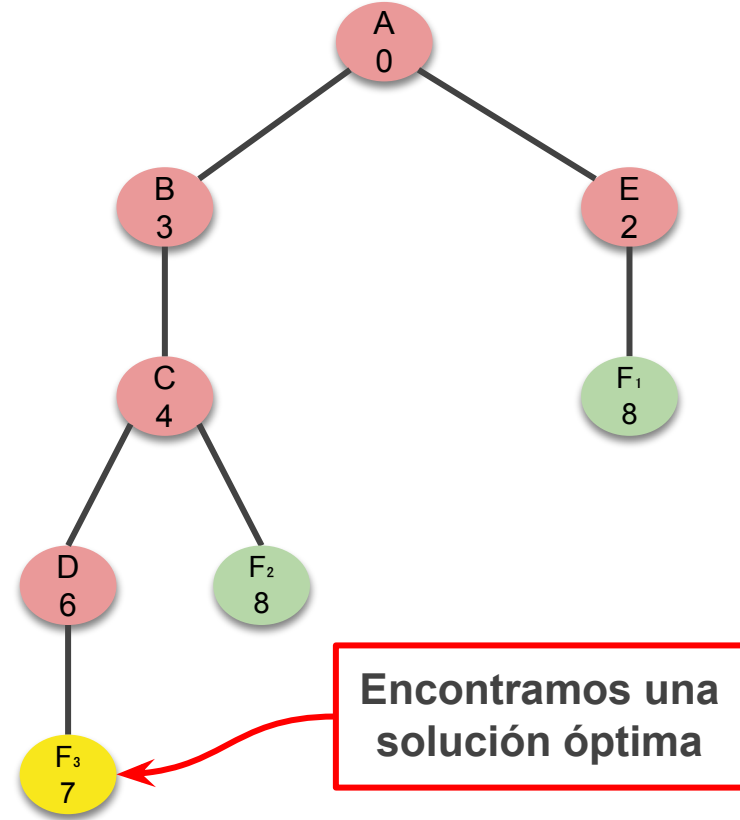




# Respuesta

— — —

Nº	Nodo actual	Frontera antes de expandir	Frontera después de expandir
0	-	{A}	-
1	A	{}	{B,E}
2	E	{B}	{B,F <sub>1</sub> }
3	B	{F <sub>1</sub> }	{F <sub>1</sub> ,C}
4	C	{F <sub>1</sub> }	{F <sub>1</sub> ,D,F <sub>2</sub> }
5	D	{F <sub>1</sub> ,F <sub>2</sub> }	{F <sub>1</sub> ,F <sub>2</sub> ,F <sub>3</sub> }
6	F <sub>3</sub>	{F <sub>1</sub> ,F <sub>2</sub> }	<b>FIN</b>





# Implementación de UCS

— — —

- ¿Cómo elegimos de la frontera el próximo nodo a expandir?

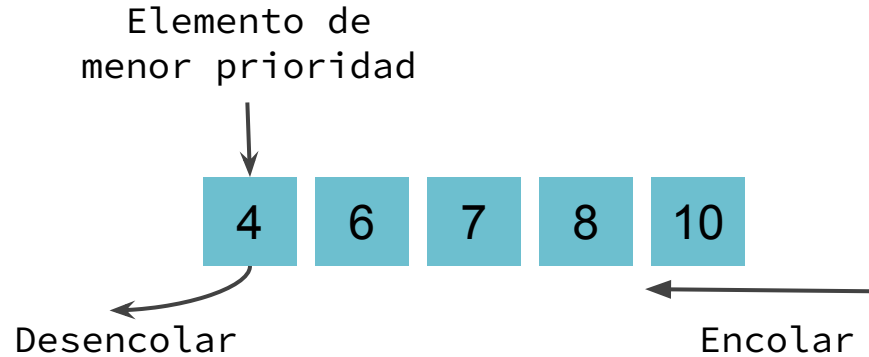
El de menor costo de camino.

- ¿Cómo lo logramos?

Usando el TAD **cola de prioridades** para la frontera.

# Cola de prioridades – El de menor prioridad es el próximo en salir

— — —



Los nuevos nodos se encolan en una cola de prioridades ordenada por el costo de camino. El nodo que se desencola es siempre el de menor costo de camino.

# Algoritmo UCS en grafos

— — —

Similar a BFS en grafos, con cuatro modificaciones principales.

1. La frontera es una **cola de prioridad**.
2. El test objetivo se aplica antes de expandir el nodo y no cuando se lo genera por primera vez, para no devolver soluciones **subóptimas**.
3. Los estados alcanzados se mantienen en un **diccionario**:

**dict[estado, nodo]**

donde cada **estado** (clave) se asocia al **nodo** (valor) del árbol de búsqueda con el menor costo de camino entre todos los nodos generados hasta el momento que repiten ese estado.

3. Se descarta cualquier nuevo camino a un estado ya alcanzado **sólo si empeora el costo de camino**. De lo contrario, se actualiza el diccionario con el nuevo nodo y se lo encola con su nuevo costo.

# Algoritmo UCS en grafos

— — —

```
1 function GRAPH-UCS(problema) return solución o fallo
2   raíz ← Nodo(estado = problema.estado-inicial, costo = 0)
3   frontera ← ColaPrioridad()
4   frontera.encolar(raíz, raíz.costo)
5   alcanzados ← {raíz.estado: raíz}
6   do
7     if (frontera.vacia()) then return fallo
8     nodo ← frontera.desencolar()
9     if (problema.test-objetivo(nodo.estado)) then return solución(nodo)
10    forall acción in problema.acciones(nodo.estado) do
11      hijo ← Nodo(estado = problema.resultado(nodo.estado, acción),
12                  costo = nodo.costo + problema.c(nodo.estado, acción),
13                  padre = nodo, acción = acción)
14      if hijo.estado is not in alcanzados or hijo.costo < alcanzados[hijo.estado].costo then
15        alcanzados[hijo.estado] = hijo
16        frontera.encolar(hijo, hijo.costo)
```

# Algoritmo UCS en grafos

El algoritmo en árboles se obtiene borrando las partes relacionadas con el diccionario de alcanzados.

```
1 function GRAPH-UCS(problema) return solución o fallo
2   raíz ← Nodo(estado = problema.estado-inicial, costo = 0)
3   frontera ← ColaPrioridad()
4   frontera.encolar(raíz, raíz.costo)
5   alcanzados ← {raíz.estado: raíz}
6   do
7     if (frontera.vacía()) then return fallo
8     nodo ← frontera.desencolar()
9     if (problema.test-objetivo(nodo.estado)) then return solución(nodo)
10    forall acción in problema.acciones(nodo.estado) do
11      hijo ← Nodo(estado = problema.resultado(nodo.estado, acción),
12                  costo = nodo.costo + problema.c(nodo.estado, acción),
13                  padre = nodo, acción = acción)
14      if hijo.estado is not in alcanzados or hijo.costo < alcanzados[hijo.estado].costo then
15        alcanzados[hijo.estado] = hijo
16        frontera.encolar(hijo, hijo.costo)
```

# Performance de TREE-UCS y GRAPH-UCS

— — —

- **Compleitud.** ✓
- **Optimalidad.** ✓ para cualquier función de costo individual.
- **Tiempo y memoria.**
  - En árboles, no se puede caracterizar fácilmente en términos de  $b$  y  $d$ . La cantidad de nodos generados depende del menor costo individual de una acción.

# Búsqueda de profundización iterativa

Iterative-Deepening Search  
(IDS)

Se introduce un **límite de profundidad**  $l$ .

En cada iteración, se aplica DFS sobre el árbol de búsqueda limitado a la profundidad  $l$ .

Es decir, los nodos del árbol de búsqueda en el nivel  $l$  de profundidad se tratan como hojas.

Comenzando con  $l = 0$ , el límite se incrementa gradualmente en 1 hasta encontrar un nodo objetivo.

— — —



# Ejemplo



Nodo de la frontera.



Nodo expandido con descendientes por expandir.

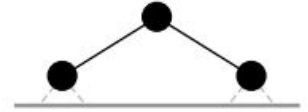
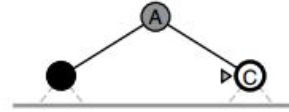
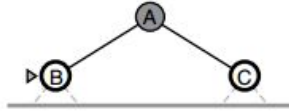
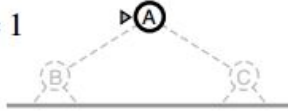


Nodo expandido sin descendientes por expandir.

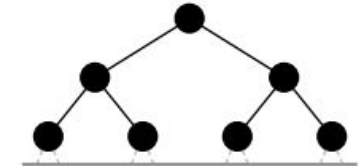
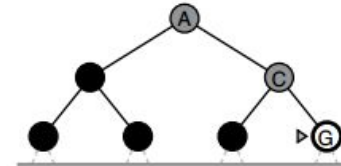
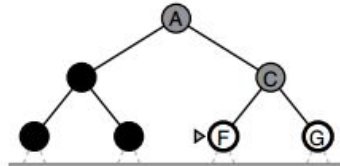
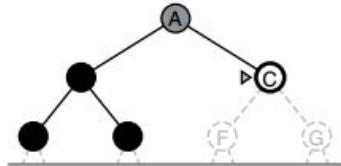
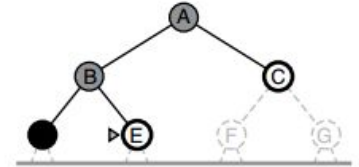
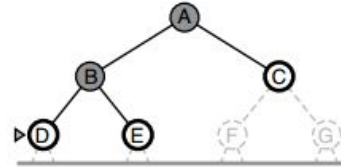
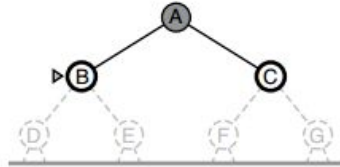
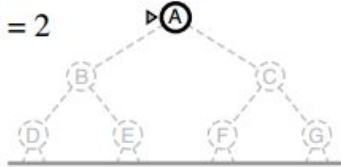
Limit = 0



Limit = 1



Limit = 2





# Algoritmo IDS en árboles

— — —

```
1 function TREE-IDS(problema) return solución o fallo
2   for  $l = 0$  to  $\infty$  do
3     resultado  $\leftarrow$  TREE-LIMITED-DFS(problema, $l$ )
4     if resultado  $\neq$  límite then resultado
```

Donde **TREE-LIMITED-DFS** es similar al algoritmo DFS en árboles, pero se agrega el control de profundidad.

# Algoritmo IDS en árboles

— — —

¿Qué límite de profundidad usar?

```
1 function TREE-IDS(problema) return solución o fallo
2   for  $l = 0$  to  $\infty$  do
3     resultado  $\leftarrow$  TREE-LIMITED-DFS(problema,  $l$ )
4     if resultado  $\neq$  límite then resultado
```

Donde **TREE-LIMITED-DFS** es similar al algoritmo DFS en árboles, pero se agrega el control de profundidad.

# Algoritmo IDS en árboles

```
1 function TREE-LIMITED-DFS(problema, l) return solución o fallo o límite
2   raíz ← Nodo(estados = problema.estado-inicial, costo = 0, profundidad = 0)
3   frontera ← Pila()
4   frontera.apilar(raíz)
5   resultado ← fallo
6   do
7     if (frontera.vacia()) then return resultado
8     nodo ← frontera.desapilar()
9     if (nodo.profundidad > l) then resultado ← límite
10    else if (problema.test-objetivo(nodo.estado)) then return solución(nodo)
11    else
12      forall acción in problema.acciones(nodo.estado) do
13        hijo ← Nodo(estados = problema.resultado(nodo.estado, acción),
14                     costo = nodo.costo + problema.c(nodo.estado, acción),
15                     padre = nodo, acción = acción, profundidad = nodo.profundidad + 1)
16        frontera.apilar(hijo)
```

# Algoritmo IDS en árboles



No expandimos los nodos que superen el límite de profundidad

```
1 function TREE-LIMITED-DFS(problema, l) return solución o fallo o límite
2   raíz ← Nodo(estados = problema.estado-inicial, costo = 0, profundidad = 0)
3   frontera ← Pila()
4   frontera.apilar(raíz)
5   resultado ← fallo
6   do
7     if (frontera.vacia()) then return resultado
8     nodo ← frontera.desapilar()
9     if (nodo.profundidad > l) then resultado ← límite
10    else if (problema.test-objetivo(nodo.estado)) then return solución(nodo)
11    else
12      forall acción in problema.acciones(nodo.estado) do
13        hijo ← Nodo(estados = problema.resultado(nodo.estado, acción),
14                     costo = nodo.costo + problema.c(nodo.estado, acción),
15                     padre = nodo, acción = acción, profundidad = nodo.profundidad + 1)
16        frontera.apilar(hijo)
```

# Performance de TREE-IDS

— — —

Combina los beneficios de BFS y DFS.

- **Compleitud.** 
- **Optimalidad.**  (si todas las acciones tienen el mismo costo).
- **Tiempo.** Se generan  $\approx b^d$  nodos. Ya veremos por qué.
- **Memoria.** Se mantienen a lo sumo  $b.d$  nodos en la frontera.

Donde ***b*** es el factor de ramificación (máximo número de hijos de cualquier nodo) y ***d*** es la menor profundidad de un nodo objetivo.

# Performance de TREE-IDS

— — —

Los nodos de menor profundidad son generados múltiples veces. Por ejemplo, la raíz es generada  $d$  veces, sus hijos  $(d-1)$  veces, y así. El número de nodos generados por IDS es:

$$d \cdot 1 + (d-1) \cdot b^1 + (d-2) \cdot b^2 + \dots + 1 \cdot b^d$$

y por DFS es:

$$1 + b^1 + b^2 + \dots + b^d$$

Para  $b = 10$  y  $d = 5$ :

$$5 + 40 + 300 + 2000 + 10000 = 12345 \text{ (IDS)}$$

$$1 + 10 + 100 + 1000 + 10000 = 11111 \text{ (DFS)}$$

**La diferencia es cada vez menos significativa con el aumento de la profundidad, pues la mayoría de los nodos del árbol de búsqueda se encuentran en el último nivel.**

# Algoritmo IDS en grafos

---

- Mantiene en memoria el conjunto de estados ya expandidos, de la misma forma que DFS en grafos.
- Aumenta el consumo de memoria, ahora es proporcional al tamaño del espacio de estados.
- No tiene ninguna ventaja respecto a BFS en grafos.



# Resumen

- ❑ Vimos dos estrategias de búsqueda: UCS e IDS.
- ❑ UCS expande siempre el nodo no expandido con el menor costo de camino. Es óptima para cualquier función de costo individual.
- ❑ IDS llama a DFS incrementando el límite de profundidad hasta alcanzar un nodo objetivo. Es completa, óptima para costos individuales unitarios, su consumo de tiempo es comparable a BFS y su consumo de memoria es comparable con DFS.
- ❑ IDS es la estrategia de búsqueda no informada de preferencia cuando el espacio de estados es grande, hay pocos caminos redundantes y la profundidad de la solución es conocida.
- ❑ En grafos, IDS no ofrece ninguna ventaja respecto a BFS.





# Próximamente

— — —

Las estrategias de búsqueda que vimos hasta ahora toman sus decisiones basadas únicamente en la formulación del problema.

Veremos algunas **estrategias de búsqueda informadas**, cuyas decisiones están guiadas por una función **heurística** que estima el costo de una solución desde un estado en particular.