

2. ESTRATEGIAS DE BÚSQUEDA NO INFORMADAS (PARTE 1)

IA 3.2 - Programación III

2° C - 2025

Dr. Mauro Lucci



Repaso

— — —

En **GRAPH-SEARCH** y **TREE-SEARCH**,

¿En qué orden se expanden los nodos de la frontera?

Estrategia de búsqueda

Estrategias de búsqueda

— — —

1. **No-informadas**. Toman decisiones basadas únicamente en la definición del problema de búsqueda.
2. **Informadas**. Cuentan con información adicional para saber cuándo un estado no-objetivo es más prometedor que otro.


Introducción árboles

Definiciones

- Vamos a introducir algunas **definiciones** y **fórmulas** para árboles.
- Serán de utilidad para estudiar la **performance** de los algoritmos de búsqueda.
- La terminología no es universal y suele variar de un libro a otro.

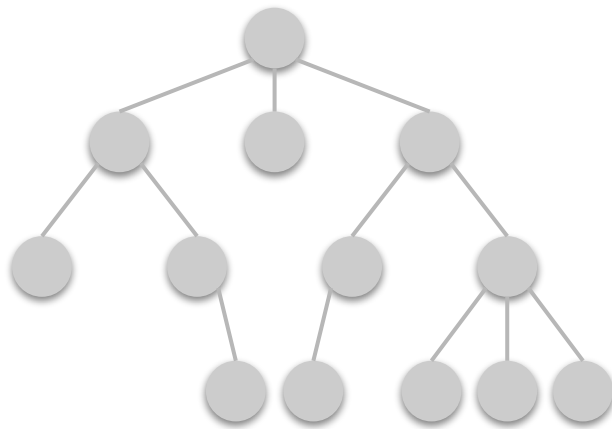
Árbol k-ario

Definición 1. Dado $k \in \mathbb{N}$, un árbol es **k-ario** si cada nodo tiene a lo sumo k hijos.

 **Ejemplo 1.** Un **árbol binario** es un caso particular de árbol k-ario para $k = 2$.



Ejemplo 2.



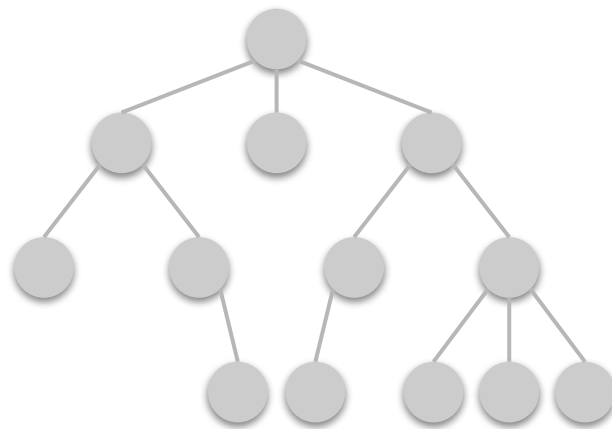
Árbol 3-ario.

Factor de ramificación

Definición 2. El **factor de ramificación** es el máximo número de hijos de cualquier nodo, es decir, el mínimo $k \in \mathbb{N}$ tal que el árbol es k -ario.



Ejemplo.



El factor de ramificación es 3.

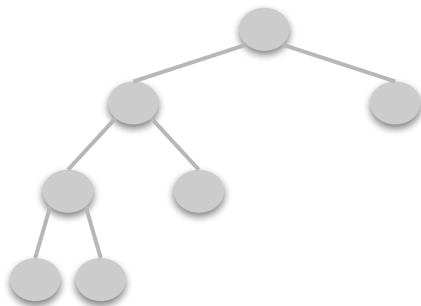
Lleno y completo

Definición 3. Un árbol k -ario está **lleno** si todos los nodos tienen 0 o k hijos.

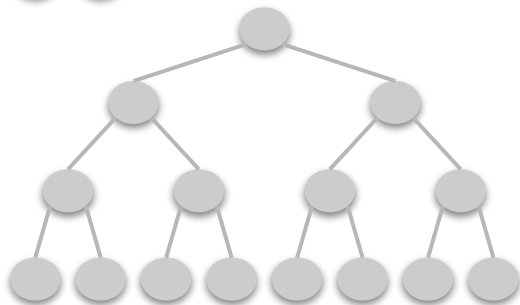
Definición 4. Un árbol k -ario está **completo** si está **lleno** y todas las hojas están en el **mismo nivel**.



Ejemplos.



Está lleno
pero no
completo.



Está
completo.

Número de nodos

Dado un árbol completo con factor de ramificación b , el **número de nodos**...

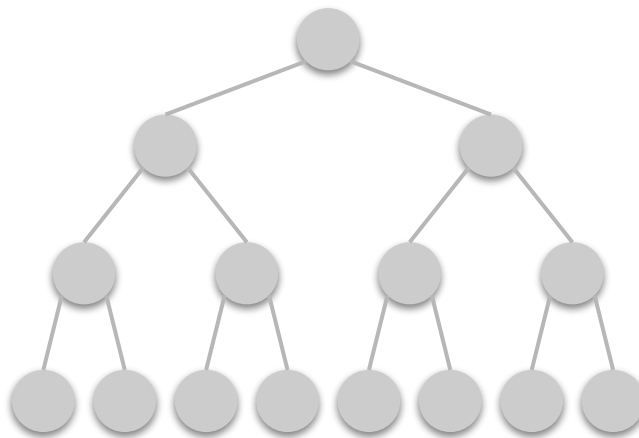
- ...en el nivel d es:

$$b^d$$

- ...hasta el nivel d es:

$$(b^{d+1}-1)/(b-1)$$

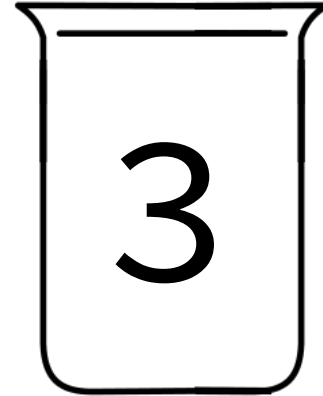
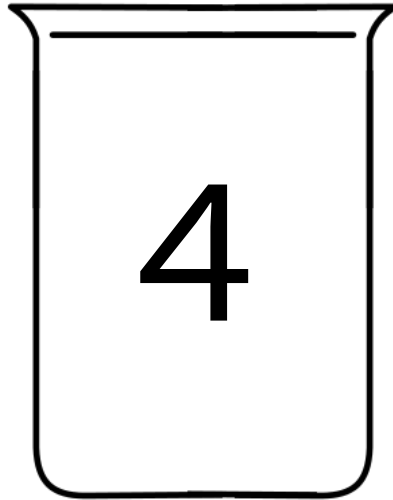
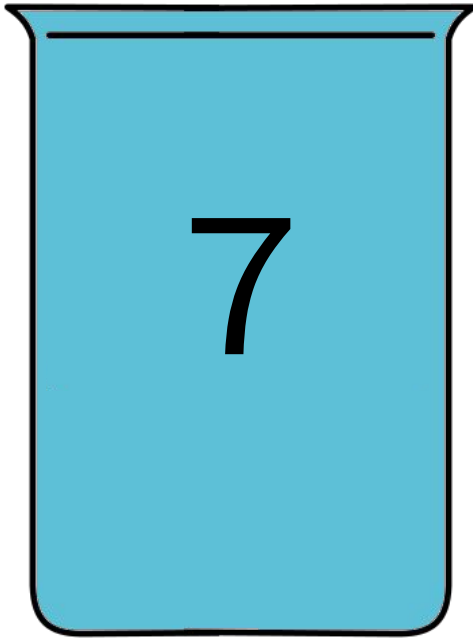
📖 **Ejemplo.** Árbol con $b = 2$.



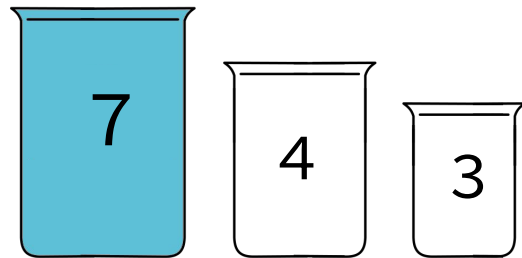
Nivel d	Nodos en d	Nodos en d o menos
0	$2^0=1$	$(2^1-1)/1=1$
1	$2^1=2$	$(2^2-1)/1=3$
2	$2^2=4$	$(2^3-1)/1=7$
3	$2^3=8$	$(2^4-1)/1=15$

Estrategias de búsqueda no informadas (clásicas)

Problema de vertido de agua



Descripción



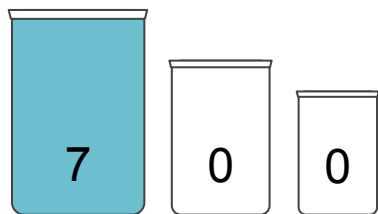
Objetivo. Lograr que uno de los recipientes contenga exactamente 2 litros de agua, en el menor número de vertidos.

Reglas.

1. Los recipientes no tienen marcas.
2. Se puede verter agua de un recipiente a otro, hasta que el primero se vacíe o el segundo se llene, lo que ocurra primero.
3. No se puede verter agua fuera de los recipientes y tampoco se pueden llenar con otra fuente de agua.



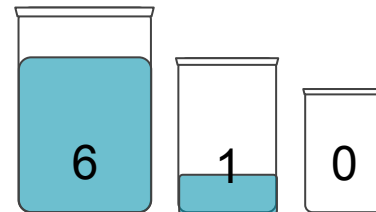
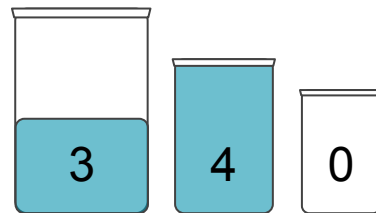
Ejemplo



De 1 a 2

De 2 a 3

De 3 a 1



¿Cómo podemos
representar a los
estados y las
acciones? 🤔

Etc...



Propuesta: Estados y Acciones

En esta formulación, un **estado** es una 3-tupla

$$(x_1, x_2, x_3) \in \{0, \dots, 7\} \times \{0, \dots, 4\} \times \{0, \dots, 3\}$$

que representa el volumen de agua contenida en cada uno de los recipientes. El número total de estados es

$$8 \cdot 5 \cdot 4 = 160$$

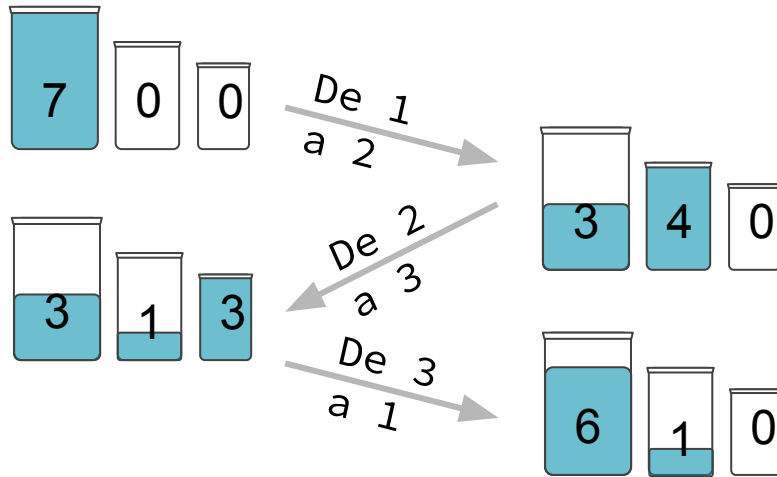
Cada **acción** es una tupla

$$(i, j) \in \{1, 2, 3\} \times \{1, 2, 3\}, \text{ tal que } i \neq j$$

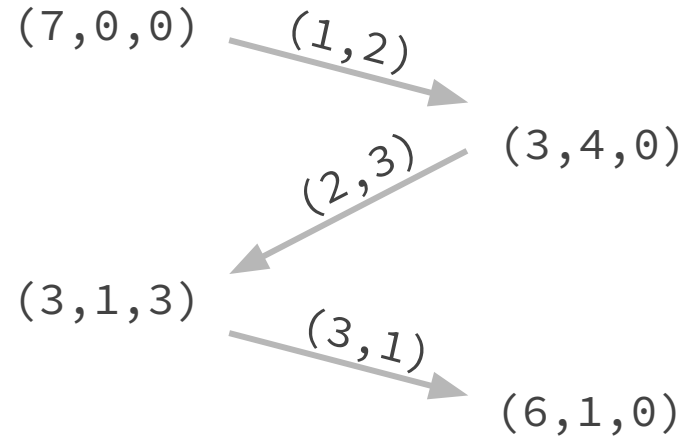
que representa que se vierte el contenido del recipiente i en el recipiente j .



Ejemplo



Ejemplo de vertidos.



Su representación.



Propuesta: Formulación

Notación. Para cada $i \in \{1,2,3\}$, notamos con v_i a la capacidad del recipiente i . Es decir, $v_1 = 7$, $v_2 = 4$, $v_3 = 3$.

- **Estado inicial.** $(7,0,0)$
- **Acciones.** Dado un estado (x_1, x_2, x_3) ,

$$\text{ACCIONES}(x_1, x_2, x_3) = \{(i, j) : i, j \in \{1, 2, 3\}, i \neq j,$$

$$\underbrace{x_i > 0, x_j < v_j\}$$

El recipiente i no está vacío y el j no está lleno.



Propuesta: Formulación

— — —

- **Modelo de transiciones.** Dado un estado (x_1, x_2, x_3) y una acción (i, j) ,

$$\text{RESULTADO}((x_1, x_2, x_3), (i, j)) = (x_1', x_2', x_3')$$

Donde, para $k \in \{1, 2, 3\}$,

$$x_k' = \begin{cases} x_k & \text{si } k \notin \{i, j\} \\ x_i - \min(x_i, v_j - x_j) & \text{si } k = i \\ x_j + \underbrace{\min(x_i, v_j - x_j)} & \text{si } k = j \end{cases}$$

Mínimo entre lo que i puede dar y lo que j puede recibir.



Propuesta: Formulación

— — —

- **Test objetivo.** Dado un estado (x_1, x_2, x_3) ,
$$\text{TEST-OBJETIVO}(x_1, x_2, x_3) = (x_1 == 2) \vee (x_2 == 2) \vee (x_3 == 2)$$
- **Costo de camino.** Es la suma de los costos individuales de cada acción del camino.

Dado un estado (x_1, x_2, x_3) y una acción (i, j) ,
$$\text{COSTO-INDIVIDUAL}((x_1, x_2, x_3), (i, j)) = 1.$$

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 - Vertido de agua

— — —

Vamos a resolver el problema de vertido de agua con el **algoritmo de búsqueda de árbol** usando la **estrategia de búsqueda primero en anchura** (BFS).

Usaremos detección de ciclos para podar nodos y escribir menos (aunque como veremos más adelante, no es estrictamente necesario).



Ejemplo - Vertido de agua

— — —

$$\begin{matrix} n_0 \\ (7, 0, 0) \end{matrix}$$

$$\text{FRONTERA} = \{n_0\}$$



Ejemplo - Vertido de agua

— — —

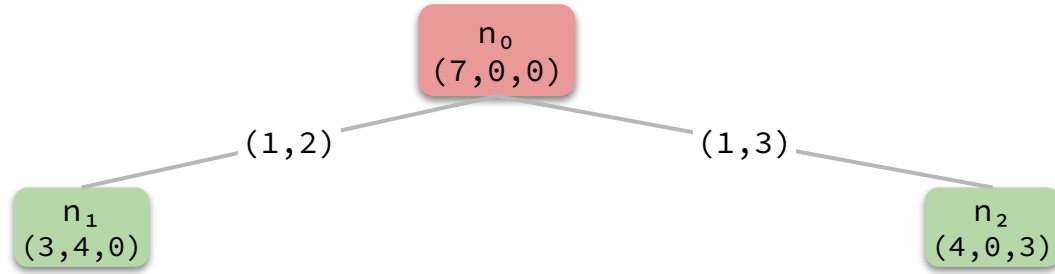
$$\begin{matrix} n_0 \\ (7, 0, 0) \end{matrix}$$

FRONTERA = {}



Ejemplo - Vertido de agua

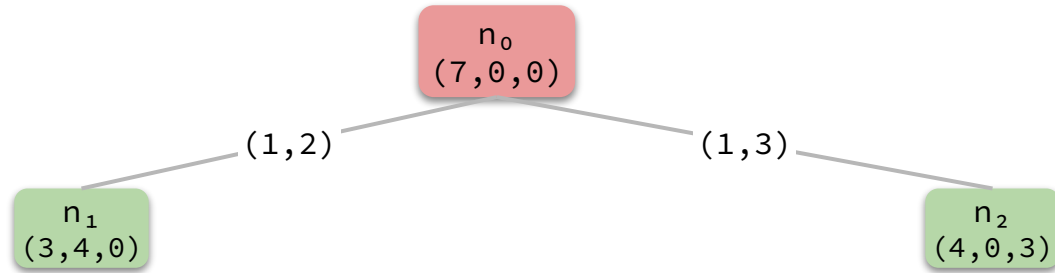
— — —



FRONTERA = $\{n_1, n_2\}$



Ejemplo - Vertido de agua



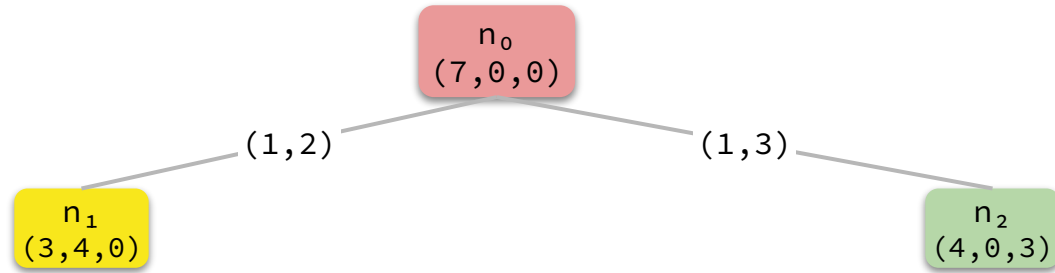
Podemos expandir indistintamente a n_1 o n_2 , pues ambos son los de menor nivel en la frontera. En este ejemplo, vamos a tomar como criterio de **desempate** su posición de izquierda a derecha, pero podríamos desempatar de cualquier otra forma (de derecha a izquierda, aleatoria, etc.). En este caso, se expande n_1 .

FRONTERA = $\{n_1, n_2\}$



Ejemplo - Vertido de agua

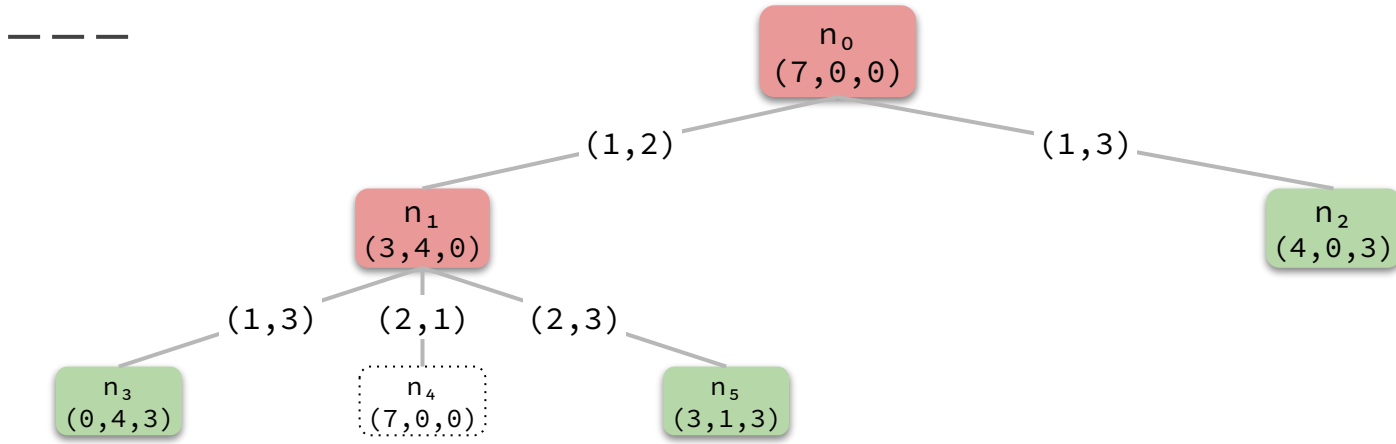
— — —



FRONTERA = $\{n_2\}$



Ejemplo - Vertido de agua



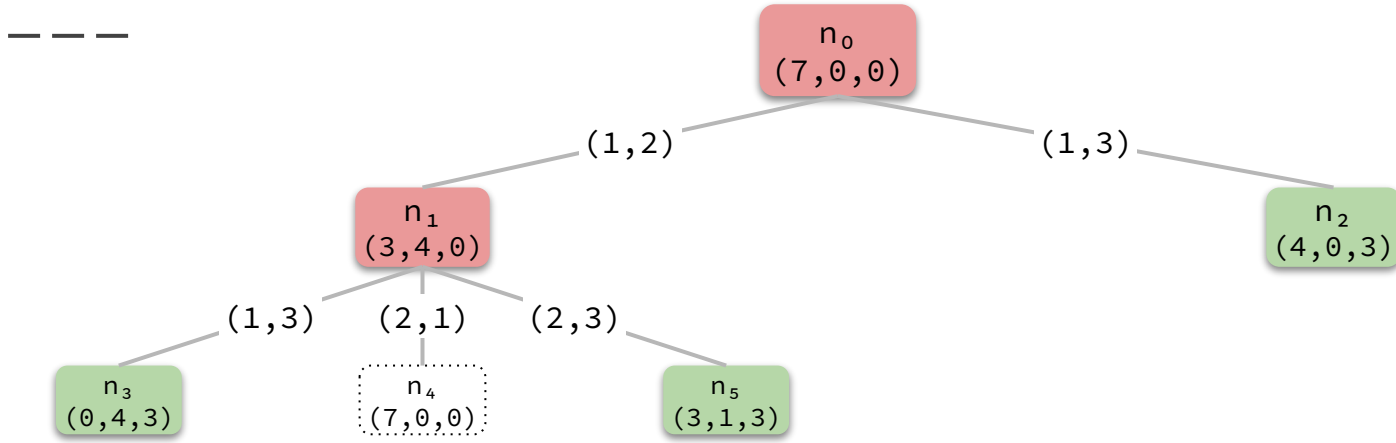
El nodo n_4 no se agrega a la frontera porque repite el estado del padre de su padre.

Estos nodos se representan con línea de punto.

FRONTERA = $\{n_2, n_3, n_5\}$



Ejemplo - Vertido de agua

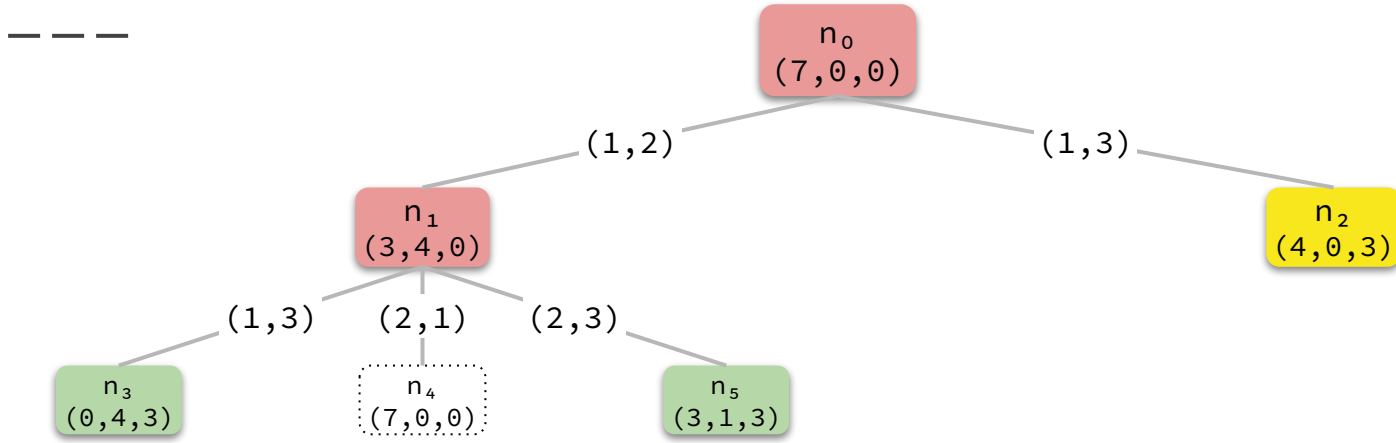


Podemos expandir **únicamente** a n_2 , pues es el de menor nivel en la frontera.

FRONTERA = $\{n_2, n_3, n_5\}$



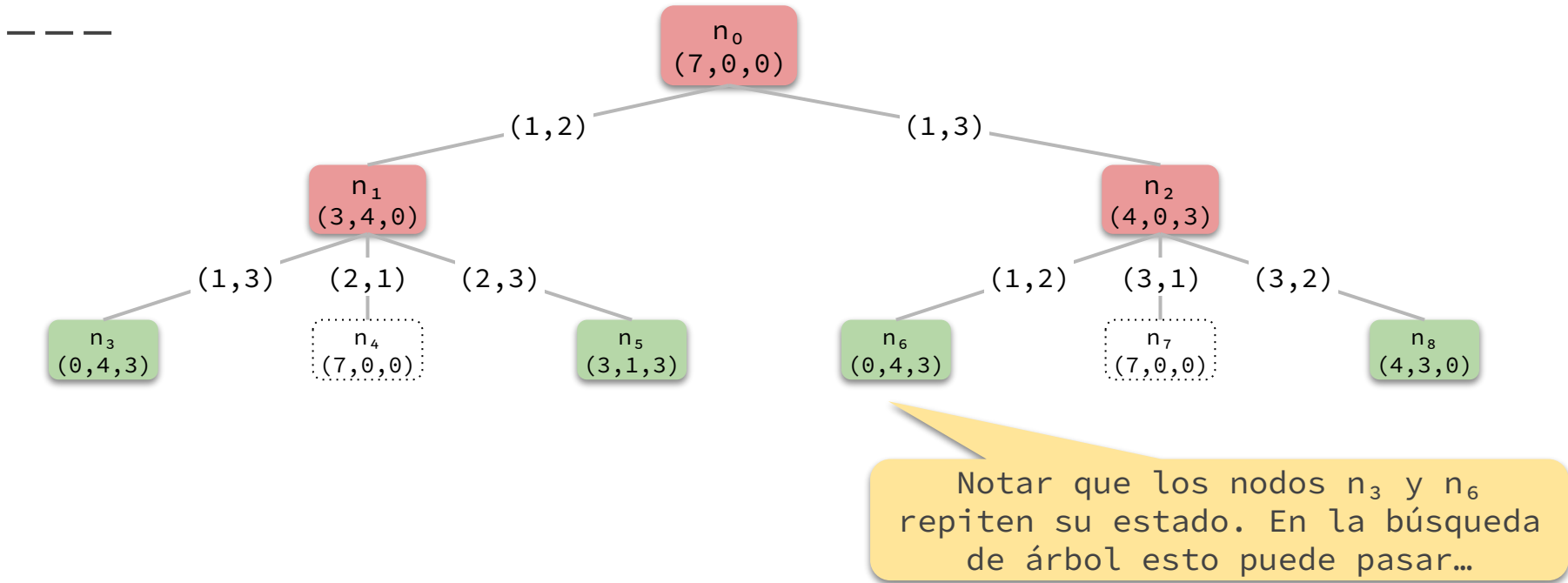
Ejemplo - Vertido de agua



FRONTERA = $\{n_3, n_5\}$



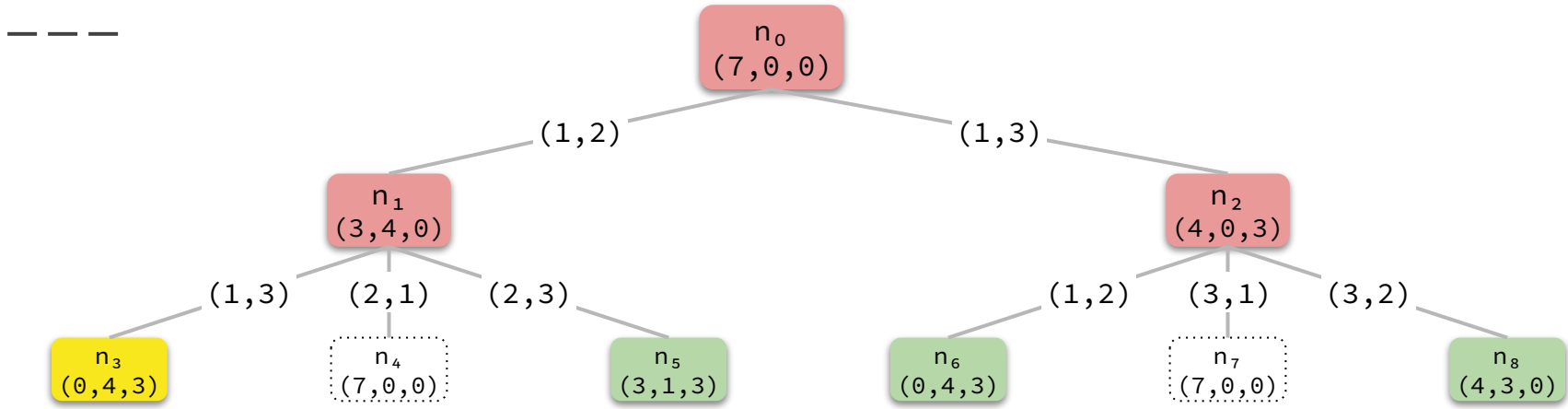
Ejemplo - Vertido de agua



FRONTERA = $\{n_3, n_5, n_6, n_8\}$



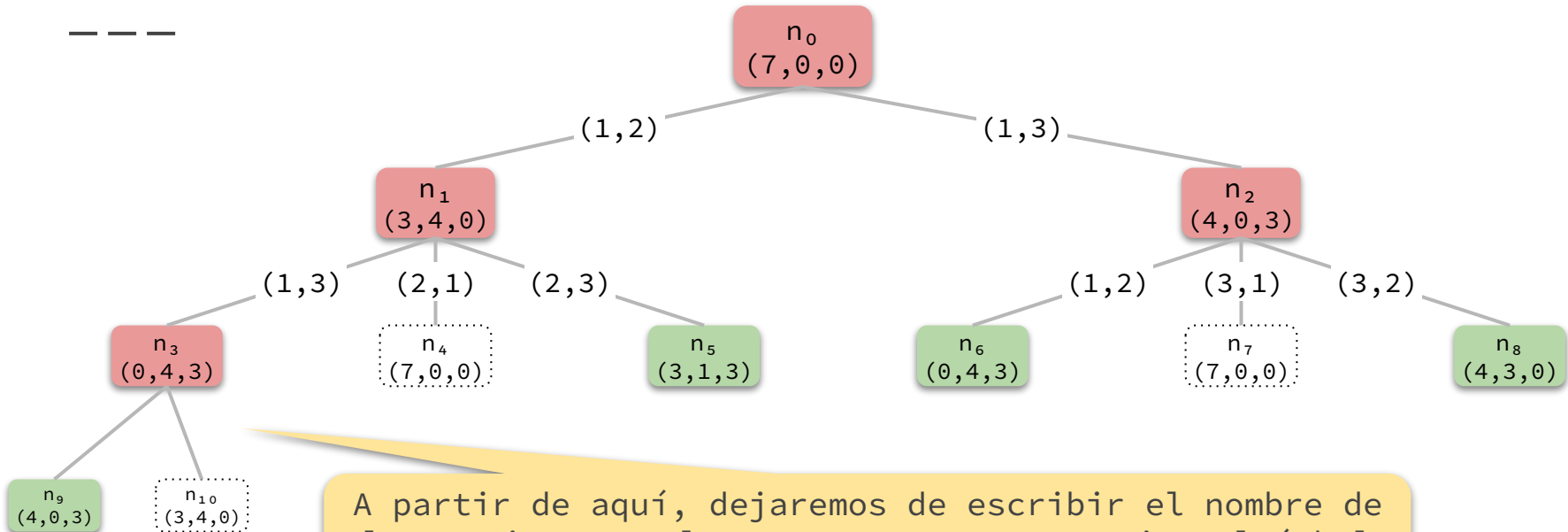
Ejemplo - Vertido de agua



FRONTERA = $\{n_5, n_6, n_8\}$



Ejemplo - Vertido de agua

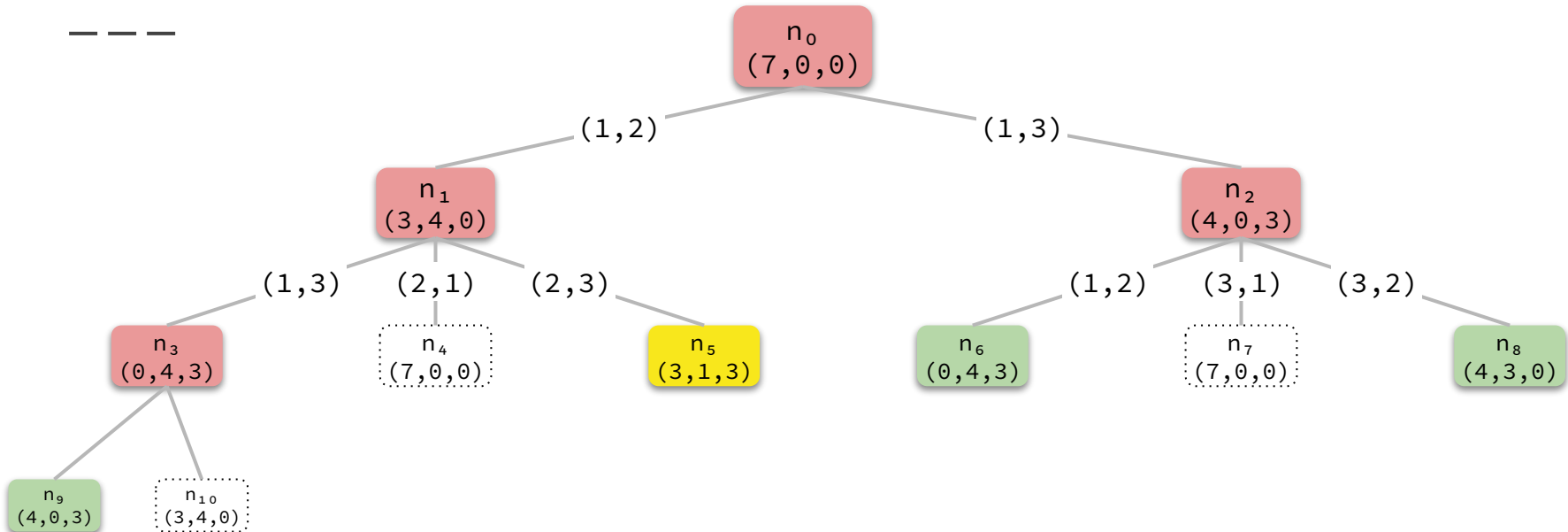


A partir de aquí, dejaremos de escribir el nombre de las acciones en las ramas para no ensuciar el árbol de búsqueda, pero lo ideal es escribirlo.

FRONTERA = $\{n_5, n_6, n_8, n_9\}$



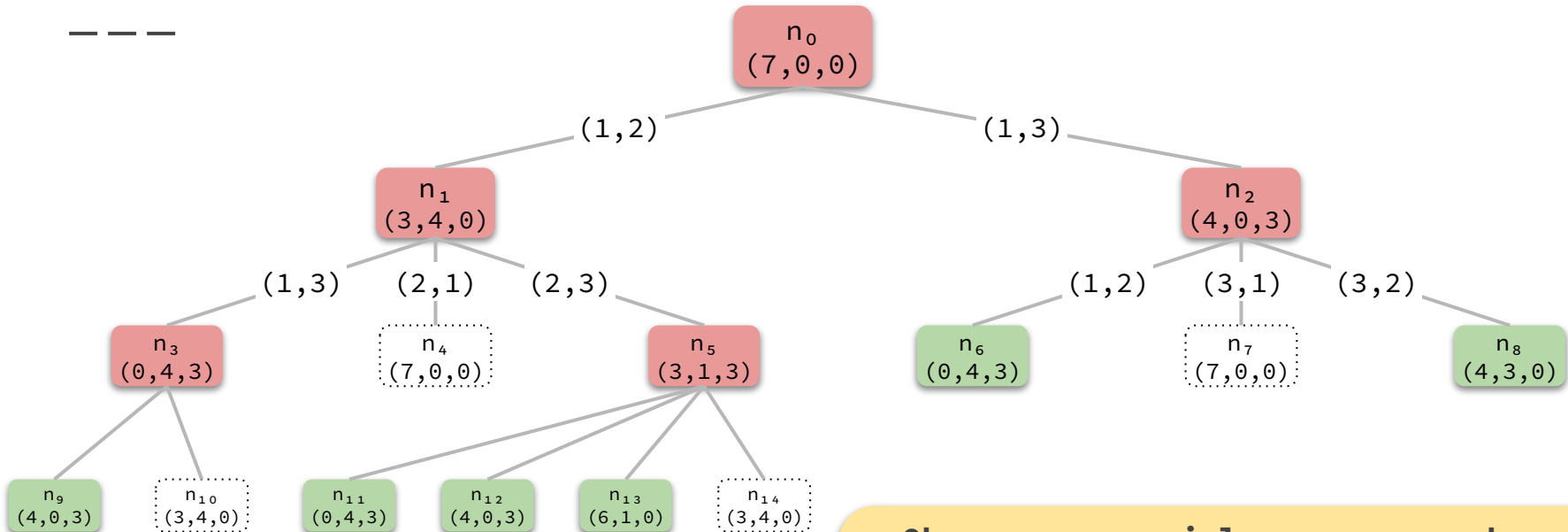
Ejemplo - Vertido de agua



FRONTERA = $\{n_6, n_8, n_9\}$



Ejemplo - Vertido de agua

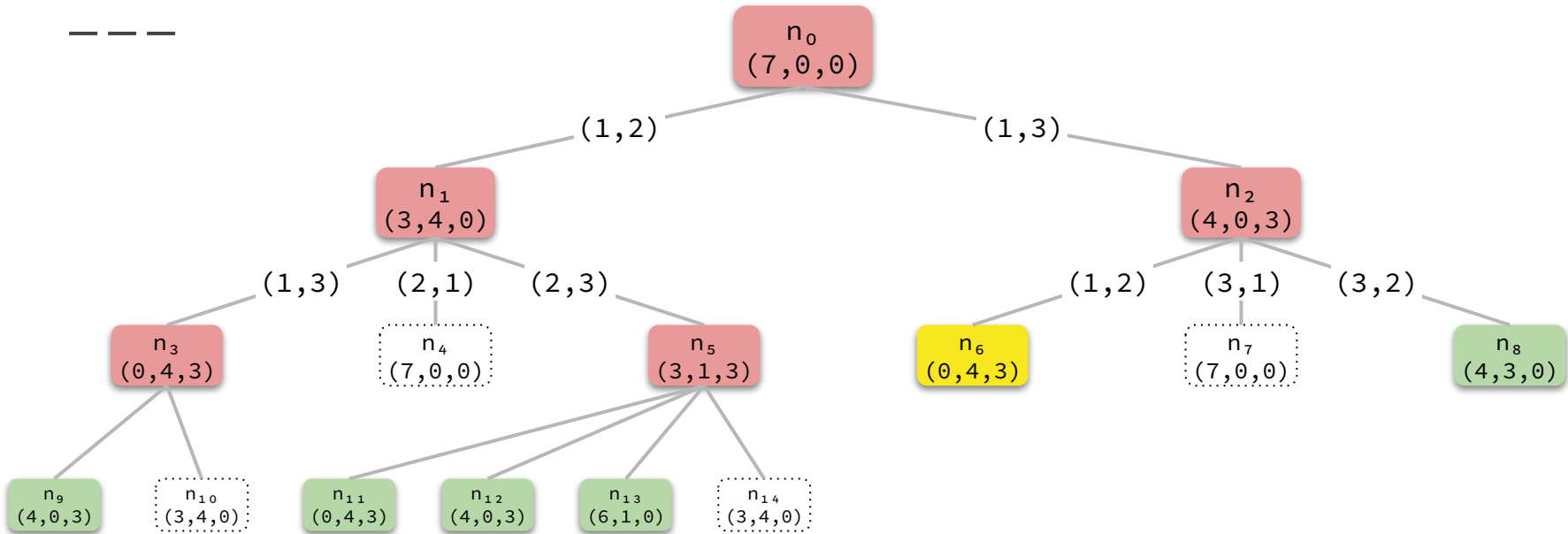


FRONTERA = $\{n_6, n_8, n_9, n_{11}, n_{12}, n_{13}\}$

Observar que si los nuevos nodos se agregan al final de la frontera, entonces los nodos de menor nivel se encuentran siempre al principio. Más adelante volveremos a hablar de esto.



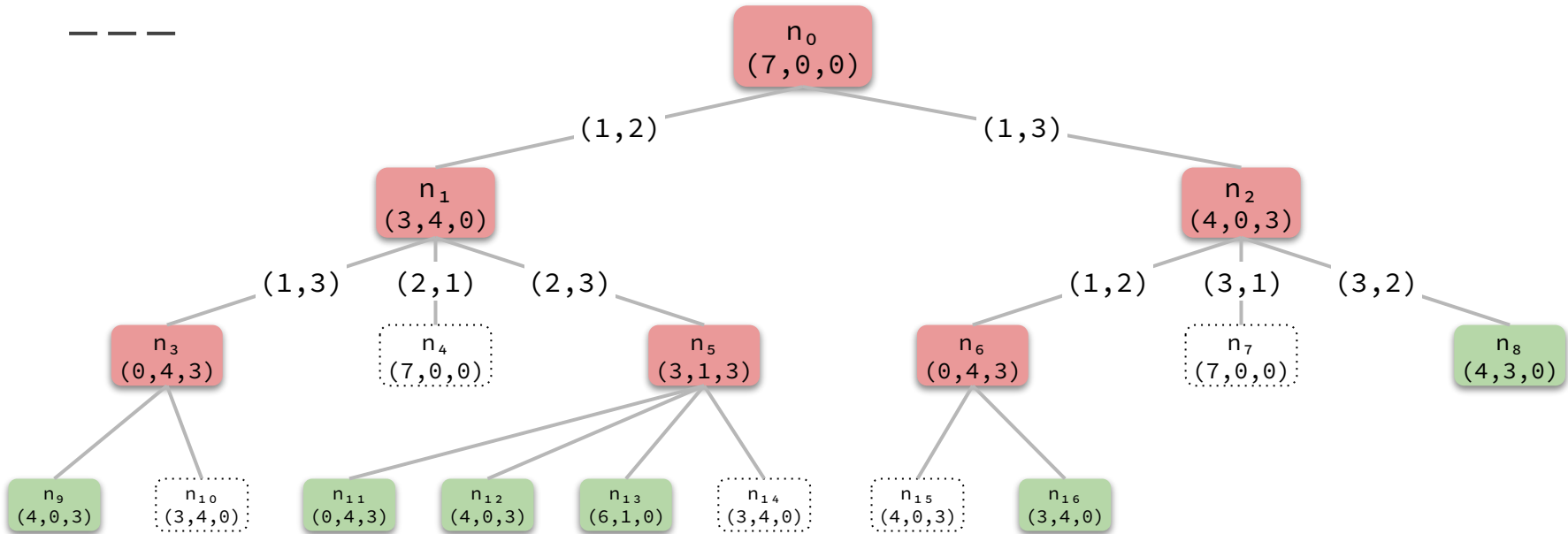
Ejemplo - Vertido de agua



FRONTERA = $\{n_8, n_9, n_{11}, n_{12}, n_{13}\}$



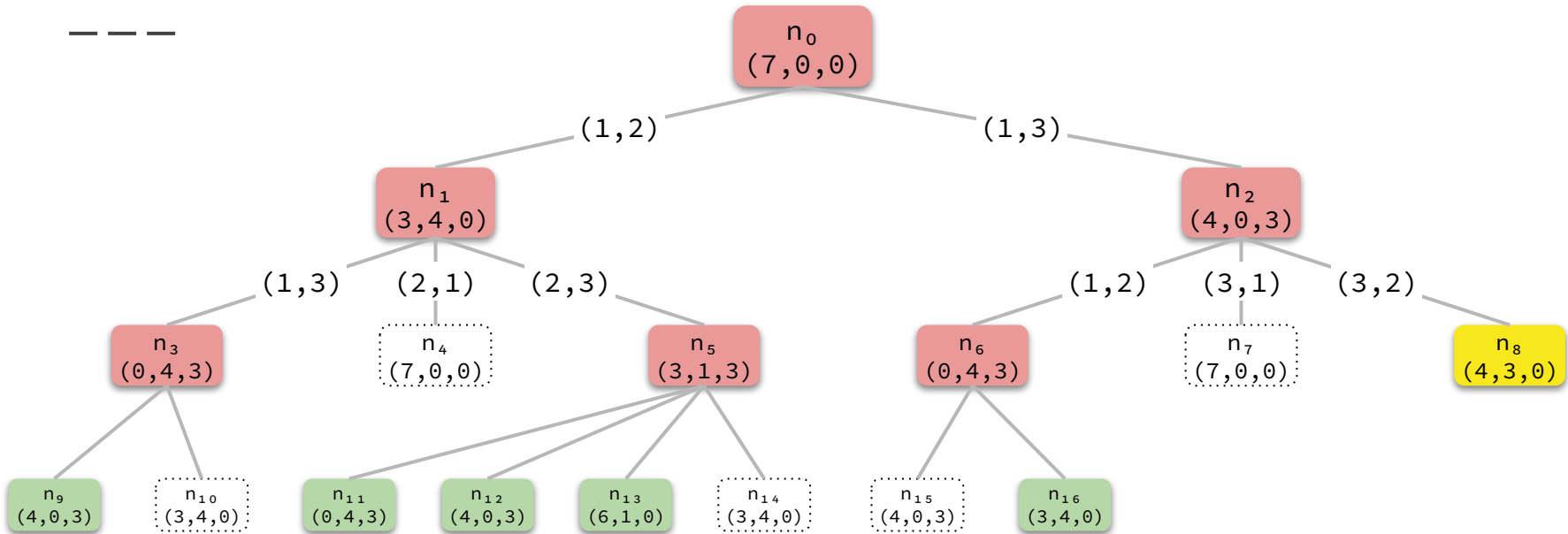
Ejemplo - Vertido de agua



FRONTERA = $\{n_8, n_9, n_{11}, n_{12}, n_{13}, n_{16}\}$



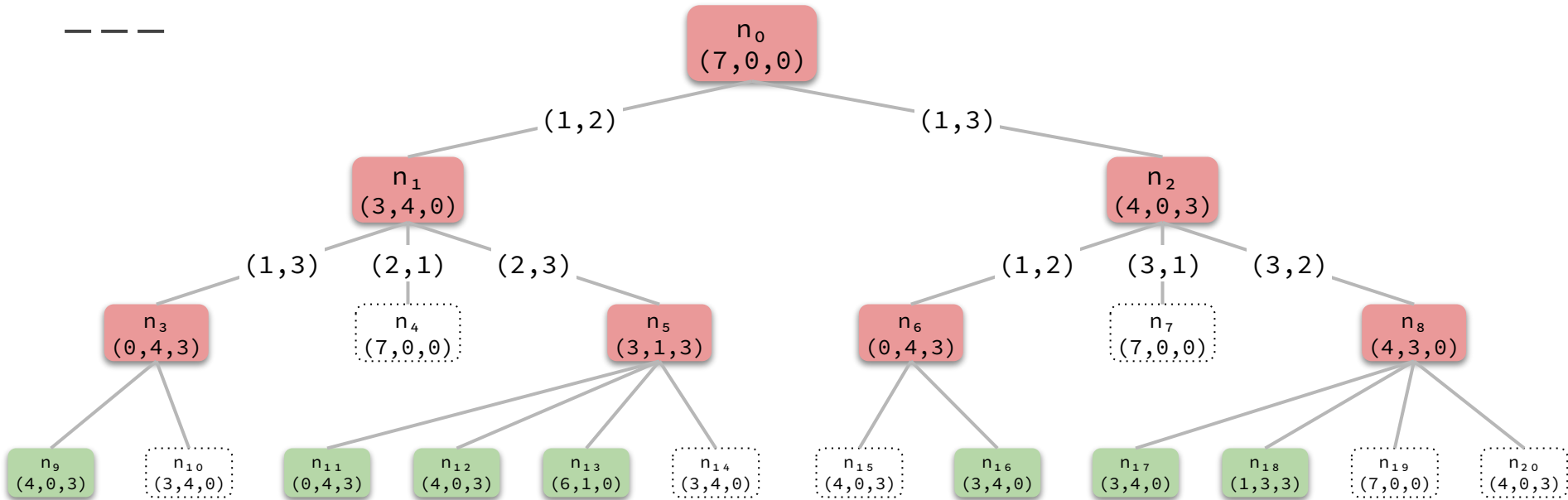
Ejemplo - Vertido de agua



FRONTERA = $\{n_9, n_{11}, n_{12}, n_{13}, n_{16}\}$



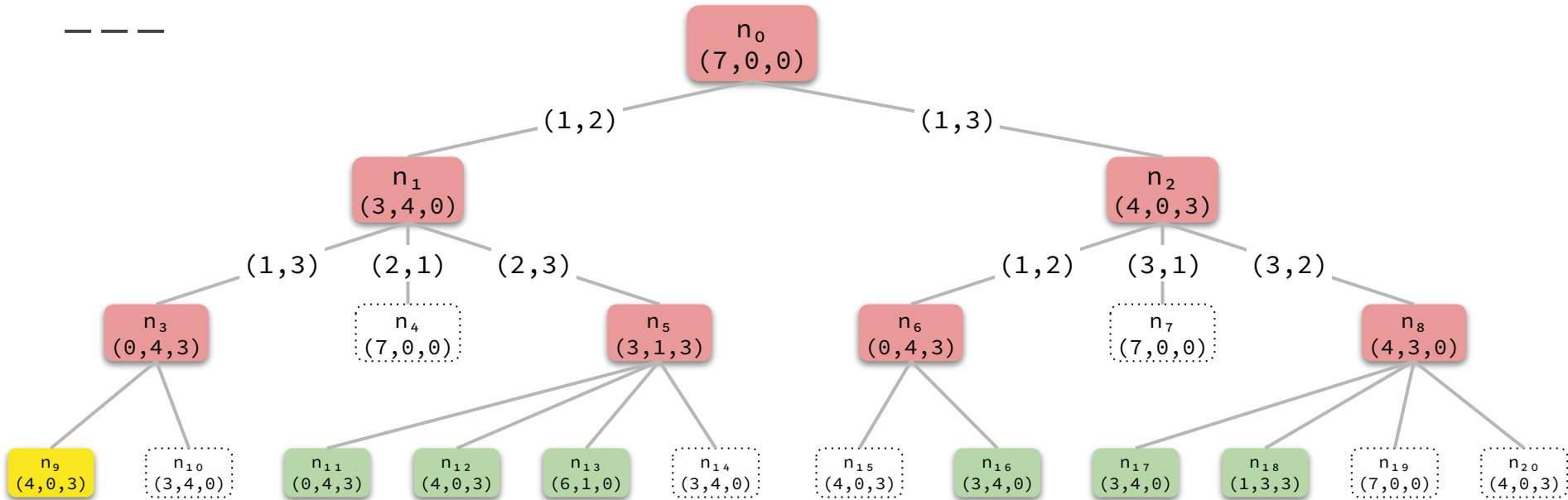
Ejemplo - Vertido de agua



FRONTERA = $\{n_9, n_{11}, n_{12}, n_{13}, n_{16}, n_{17}, n_{18}\}$



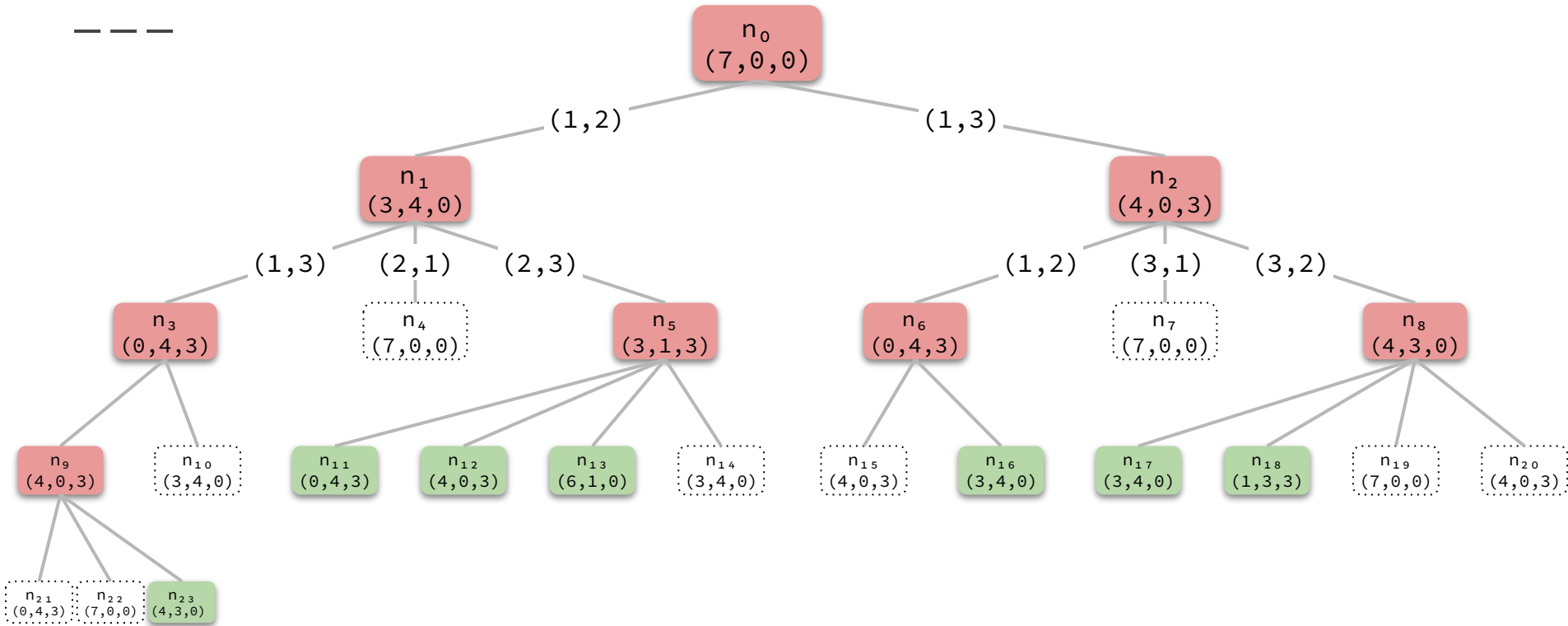
Ejemplo - Vertido de agua



FRONTERA = $\{n_{11}, n_{12}, n_{13}, n_{16}, n_{17}, n_{18}\}$



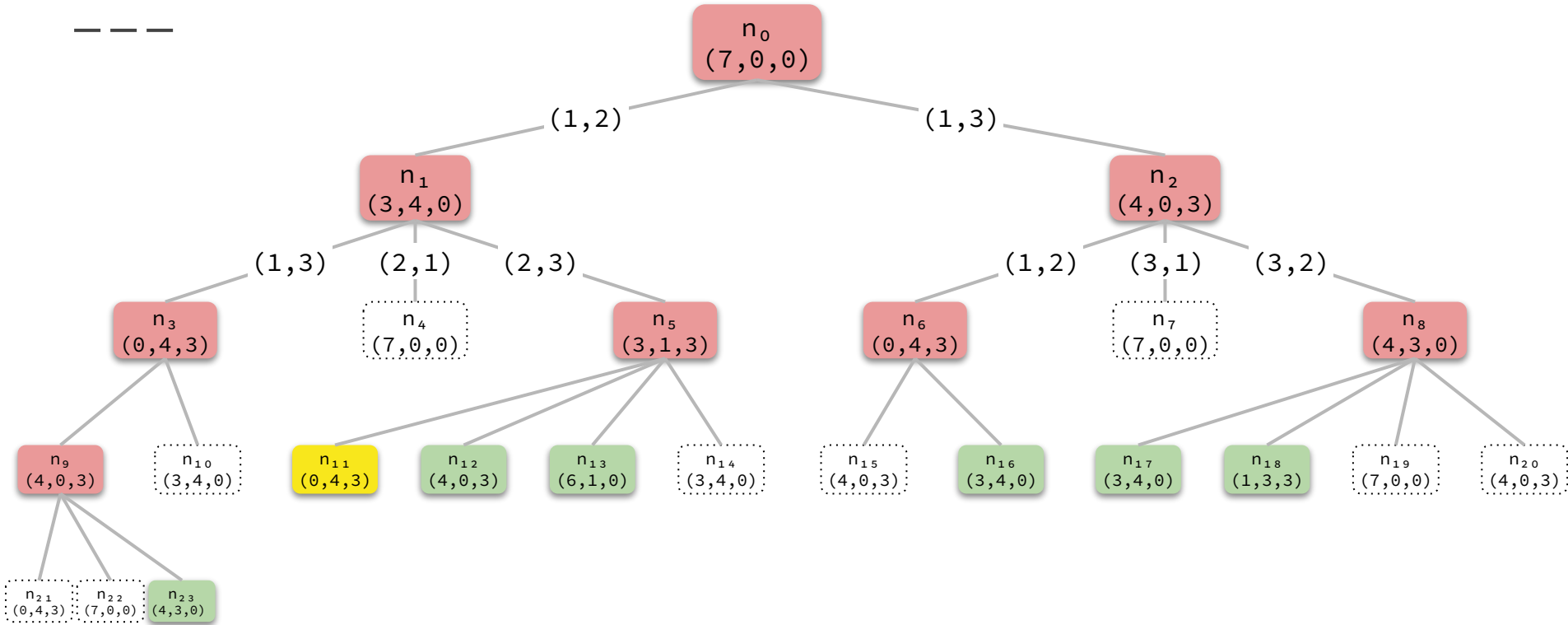
Ejemplo - Vertido de agua



FRONTERA = { $n_{11}, n_{12}, n_{13}, n_{16}, n_{17}, n_{18}, n_{23}$ }



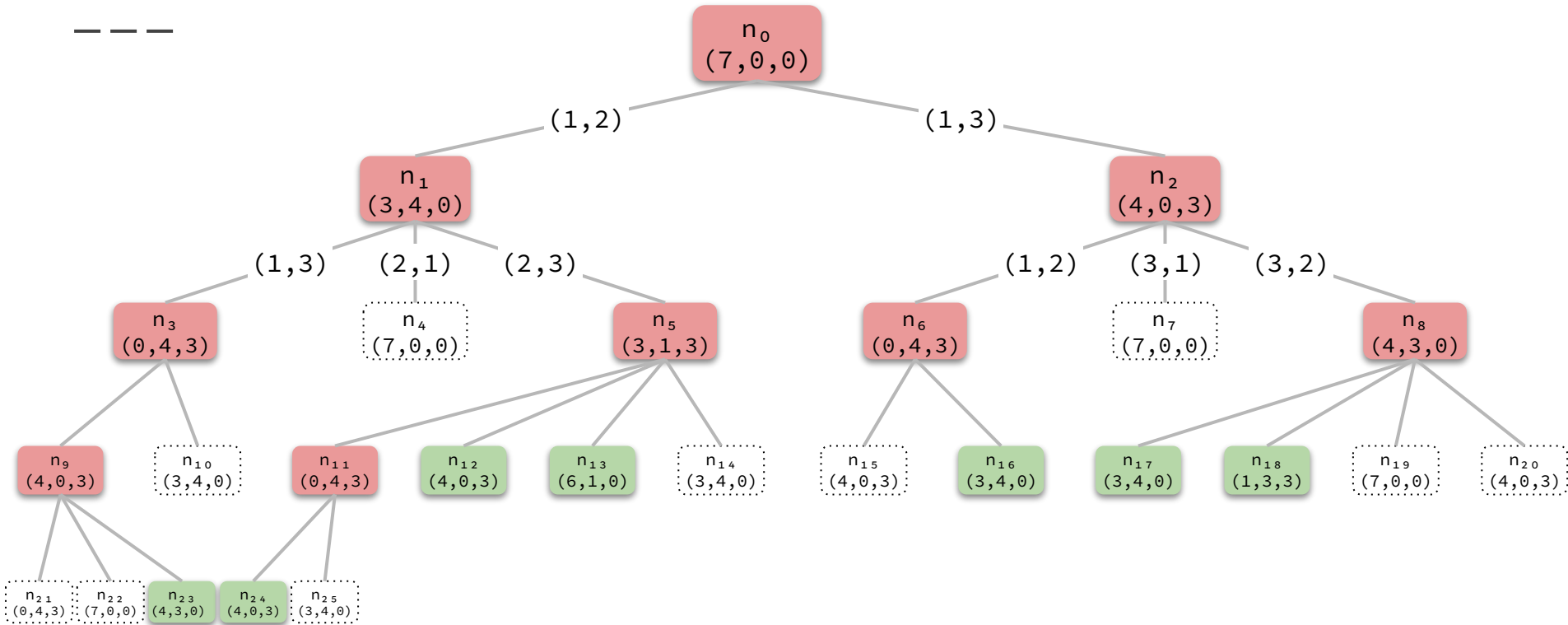
Ejemplo - Vertido de agua



FRONTERA = $\{n_{12}, n_{13}, n_{16}, n_{17}, n_{18}, n_{23}\}$



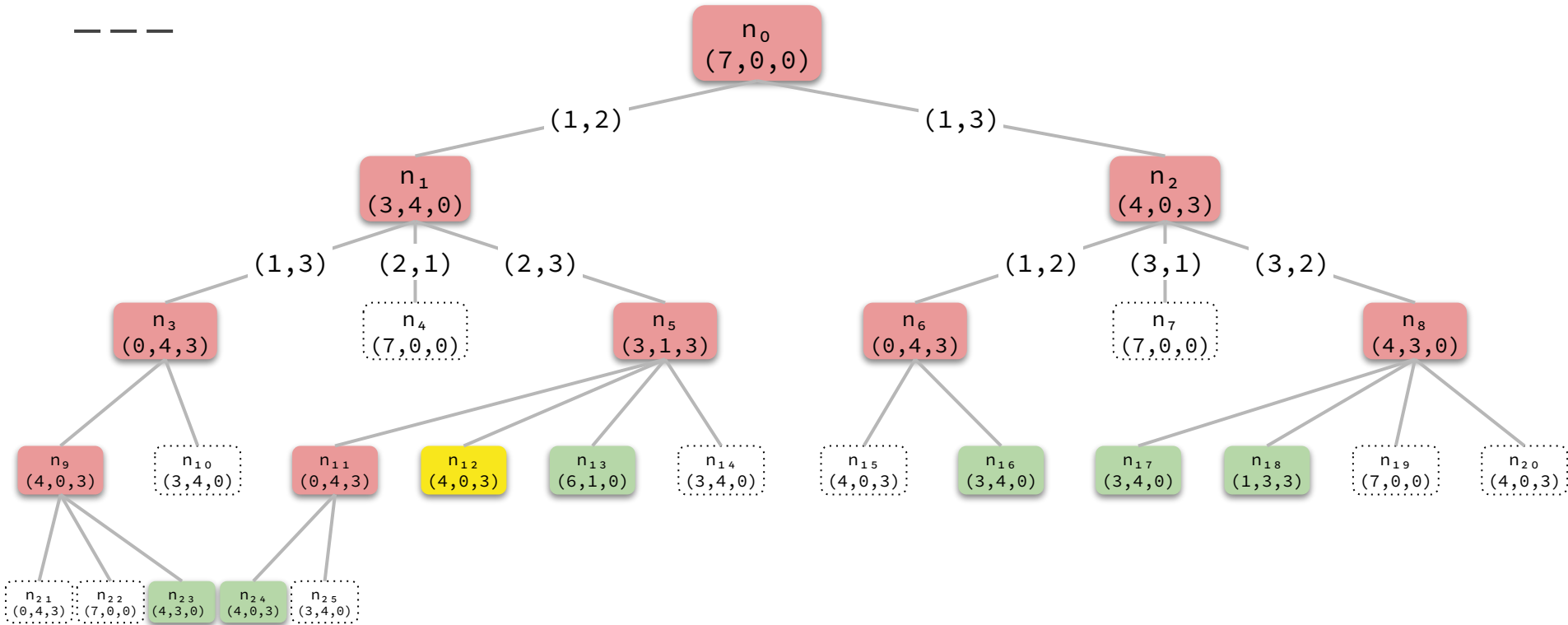
Ejemplo - Vertido de agua



FRONTERA = { $n_{12}, n_{13}, n_{16}, n_{17}, n_{18}, n_{23}, n_{24}$ }



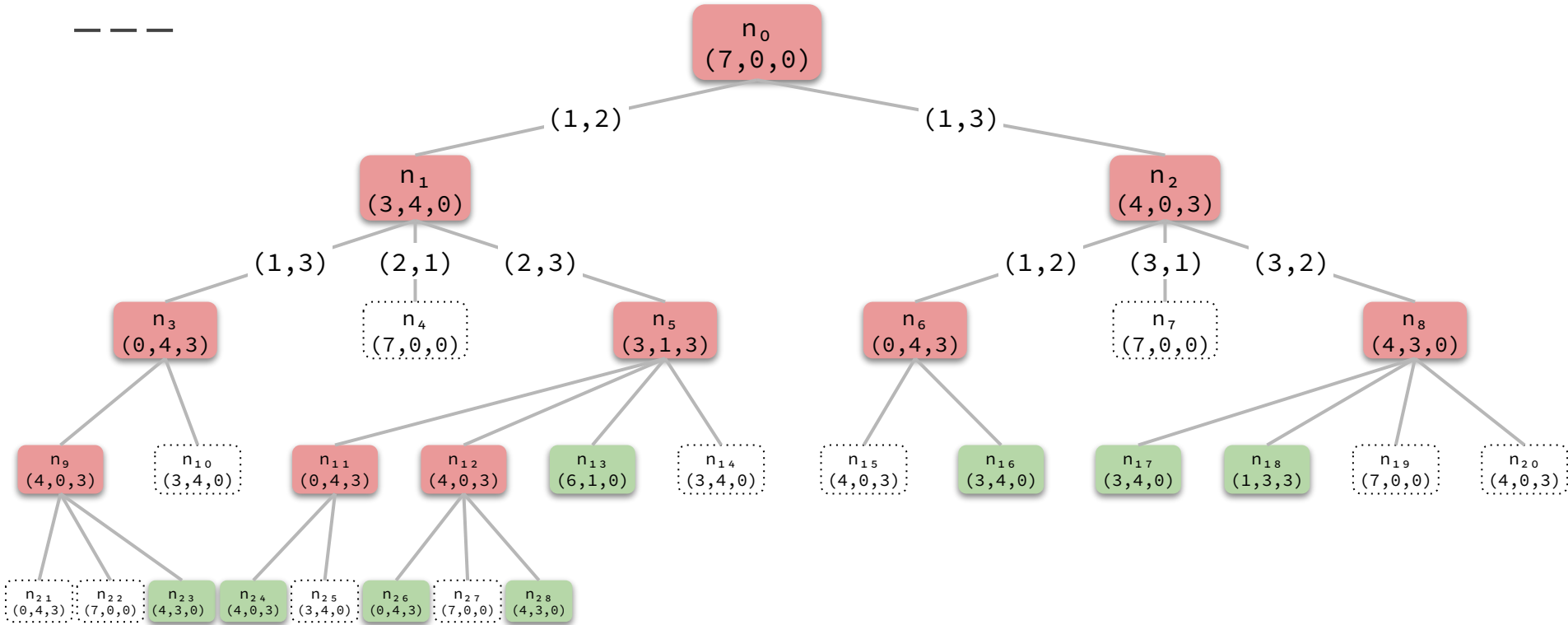
Ejemplo - Vertido de agua



FRONTERA = { n_{13} , n_{16} , n_{17} , n_{18} , n_{23} , n_{24} }



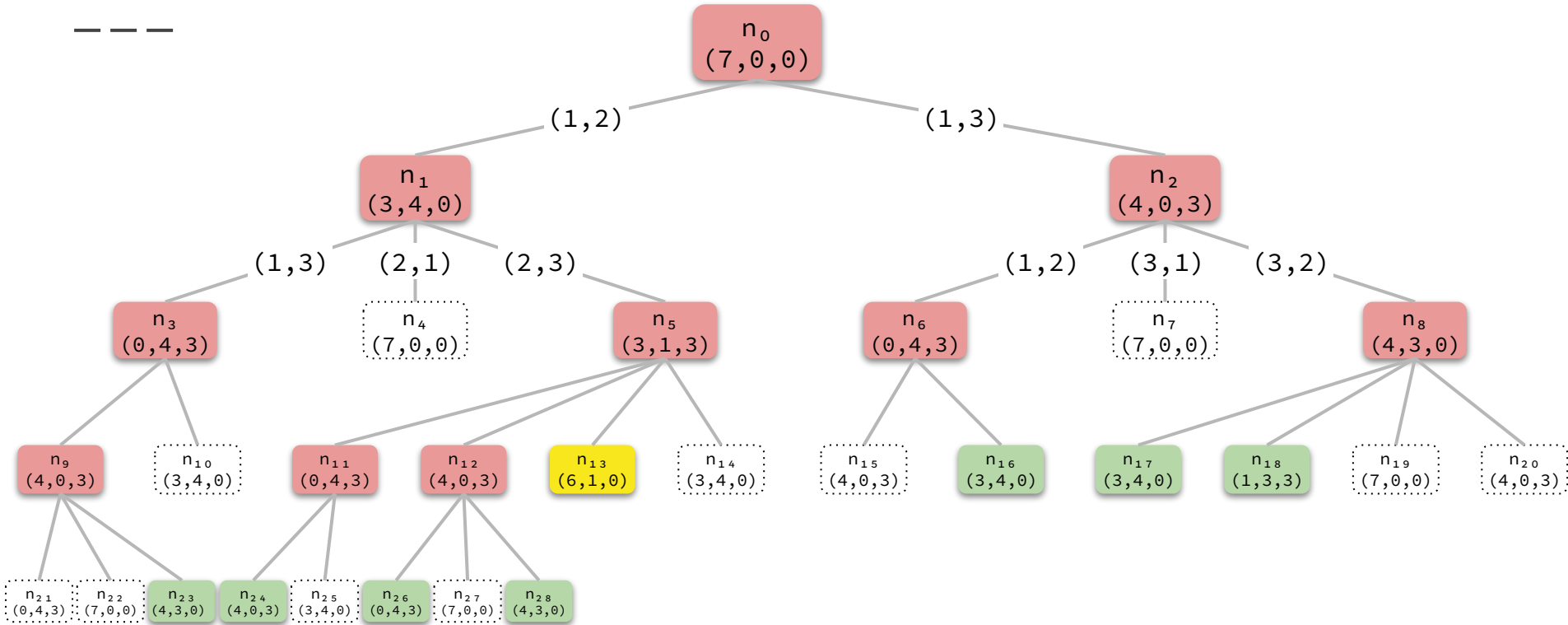
Ejemplo - Vertido de agua



FRONTERA = { $n_{13}, n_{16}, n_{17}, n_{18}, n_{23}, n_{24}, n_{26}, n_{28}$ }



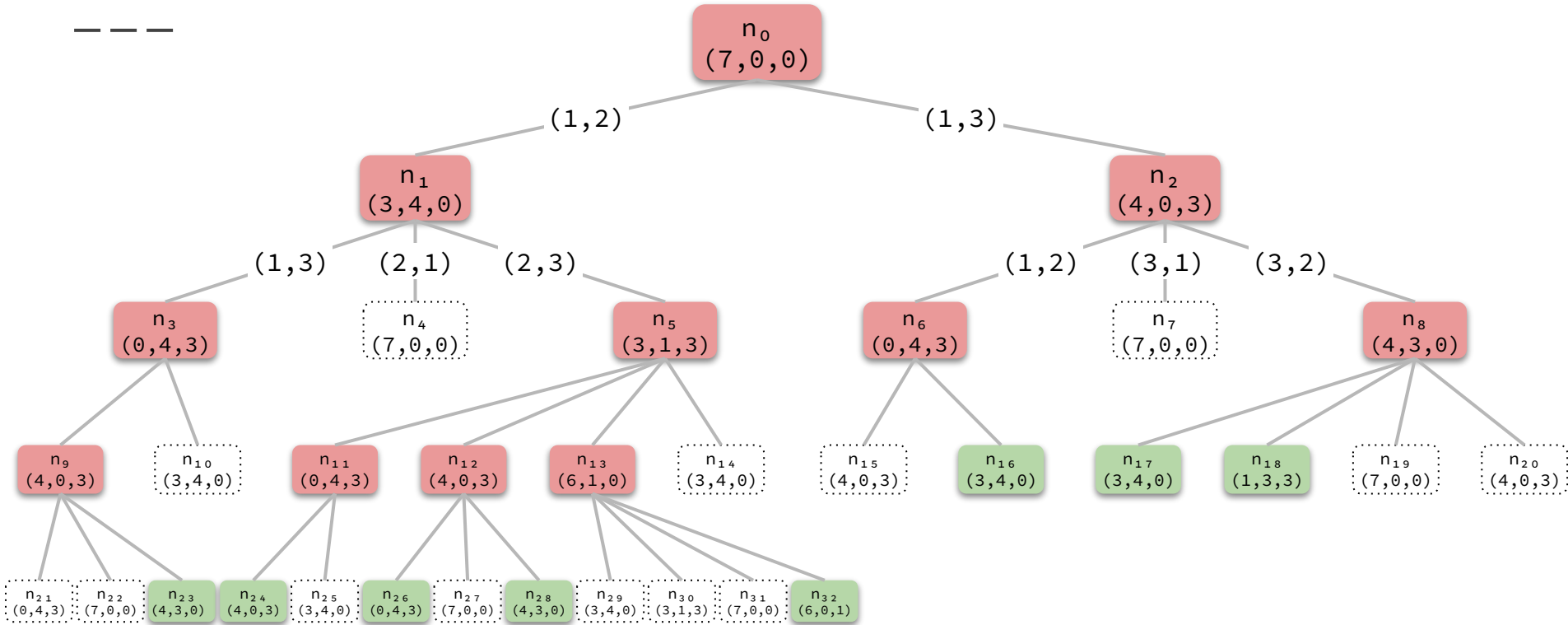
Ejemplo - Vertido de agua



FRONTERA = { n_{16} , n_{17} , n_{18} , n_{23} , n_{24} , n_{26} , n_{28} }



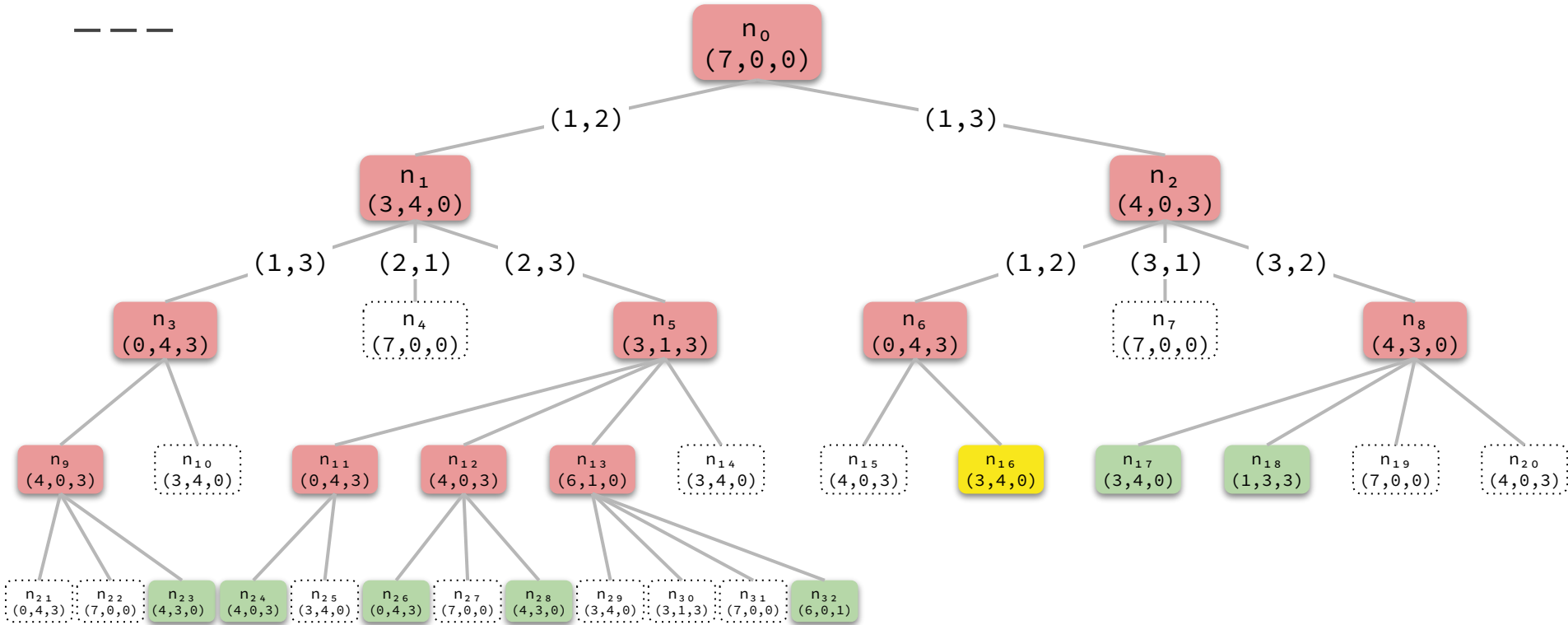
Ejemplo - Vertido de agua



FRONTERA = { $n_{16}, n_{17}, n_{18}, n_{23}, n_{24}, n_{26}, n_{28}, n_{32}$ }



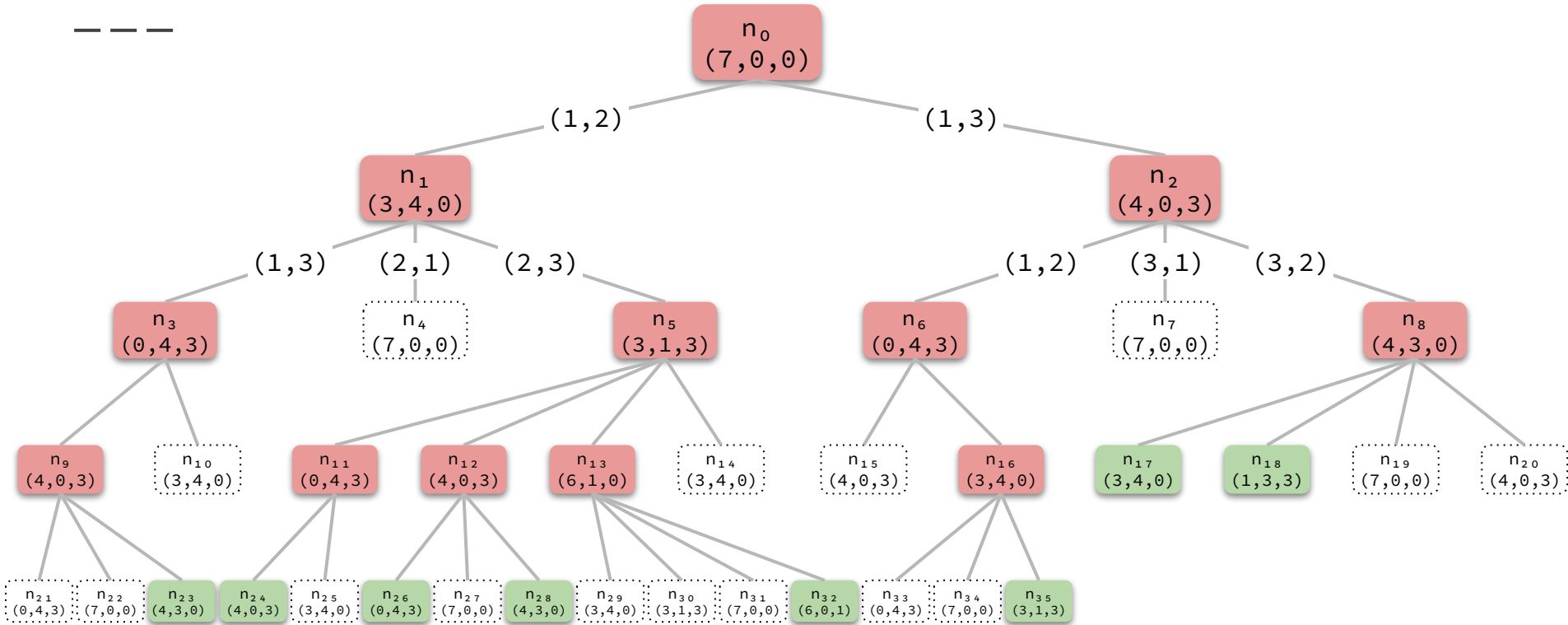
Ejemplo - Vertido de agua



FRONTERA = { $n_{17}, n_{18}, n_{23}, n_{24}, n_{26}, n_{28}, n_{32}$ }



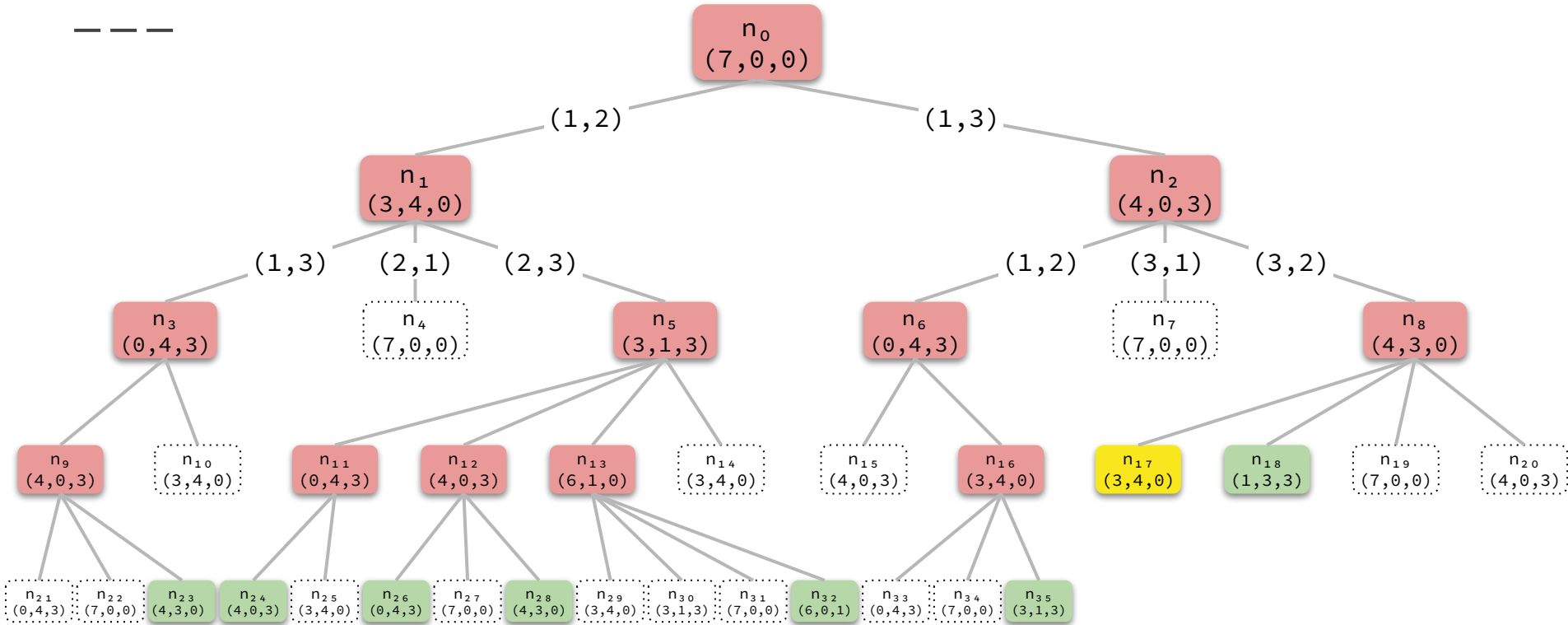
Ejemplo - Vertido de agua



FRONTERA = {n₁₇, n₁₈, n₂₃, n₂₄, n₂₆, n₂₈, n₃₂, n₃₅}



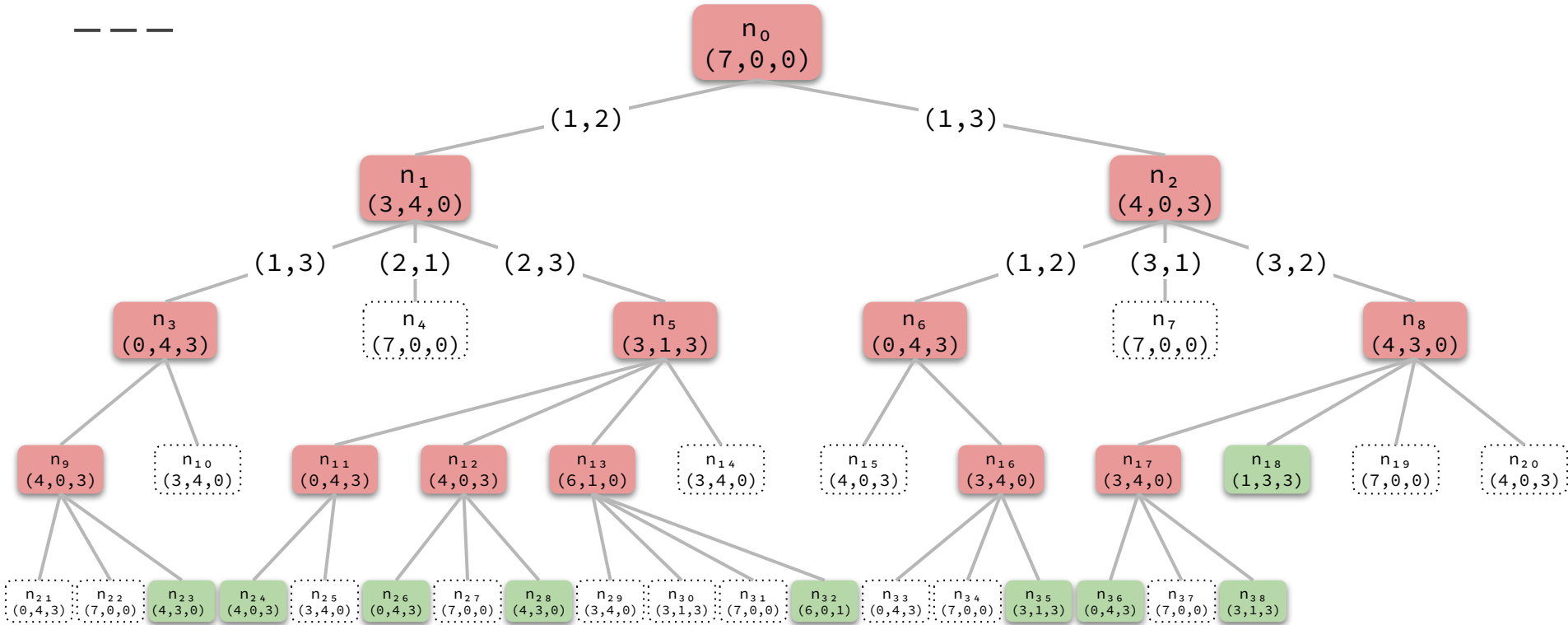
Ejemplo - Vertido de agua



FRONTERA = { n_{18} , n_{23} , n_{24} , n_{26} , n_{28} , n_{32} , n_{35} }



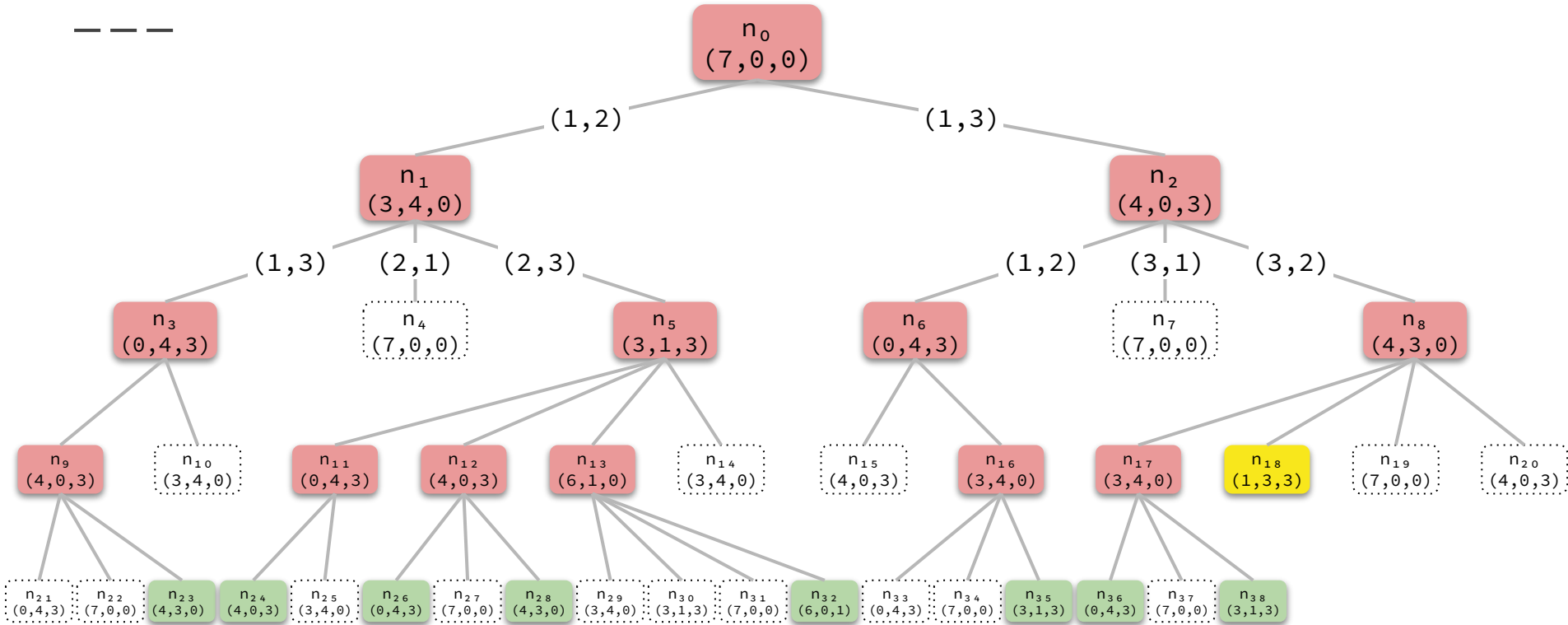
Ejemplo - Vertido de agua



FRONTERA = { n_{18} , n_{23} , n_{24} , n_{26} , n_{28} , n_{32} , n_{35} , n_{36} , n_{38} }



Ejemplo - Vertido de agua



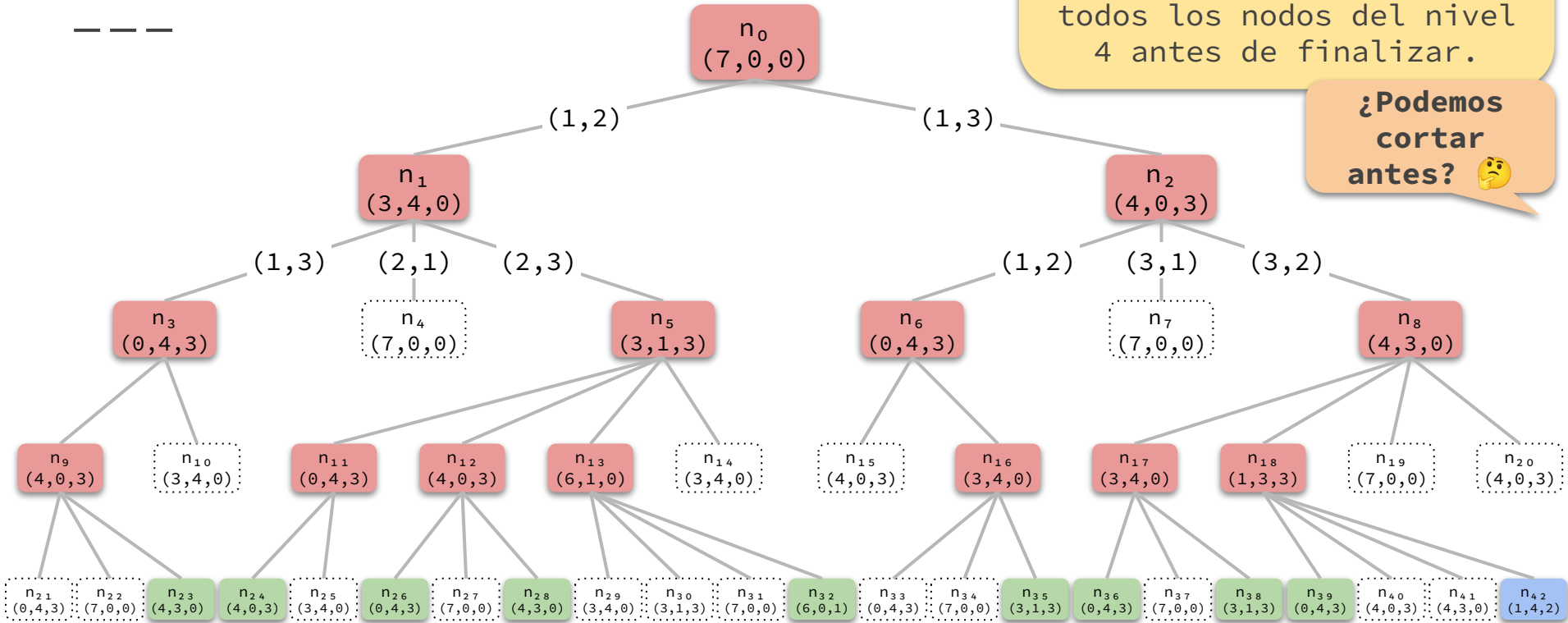
FRONTERA = {n₂₃, n₂₄, n₂₆, n₂₈, n₃₂, n₃₅, n₃₆, n₃₈}



Ejemplo - Vertido de agua

Si **TEST-OBJETIVO** se llama al sacar un nodo de la frontera, el algoritmo todavía debe expandir todos los nodos del nivel 4 antes de finalizar.

¿Podemos cortar antes? 🤔



FRONTERA = { $n_{23}, n_{24}, n_{26}, n_{28}, n_{32}, n_{35}, n_{36}, n_{38}, n_{42}$ }

Tiene un estado objetivo.

Criterio de parada

En la estrategia BFS, es **conveniente** ejecutar el test objetivo...

~~❑ después de sacar un nodo de la frontera.~~

❑ **antes de agregar un nodo a la frontera.**

⊖ Más adelante veremos estrategias donde esto puede ser contraproducente... ⊖

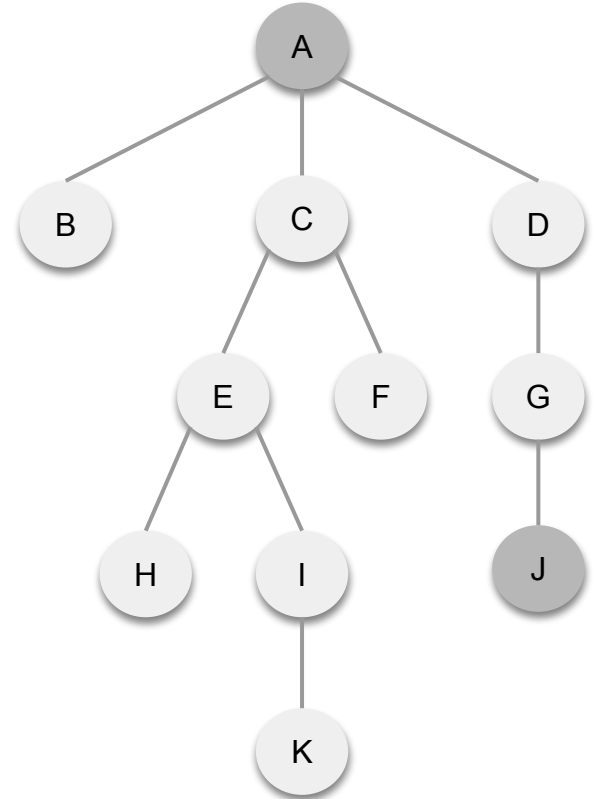
En el mejor de los casos, **evitamos generar un nivel completo del árbol.**



Ejercicio

Dado el siguiente árbol de búsqueda, donde A es la raíz y J es el único nodo objetivo, decidir cuáles secuencias se corresponden con posibles órdenes de expansión de nodos según la estrategia BFS.

1. A, B, C, D, E, F, G.
2. A, C, B, D, G.
3. A, D, B, C, E, F, I, G.





Respuestas

1. A, B, C, D, E, F, G.

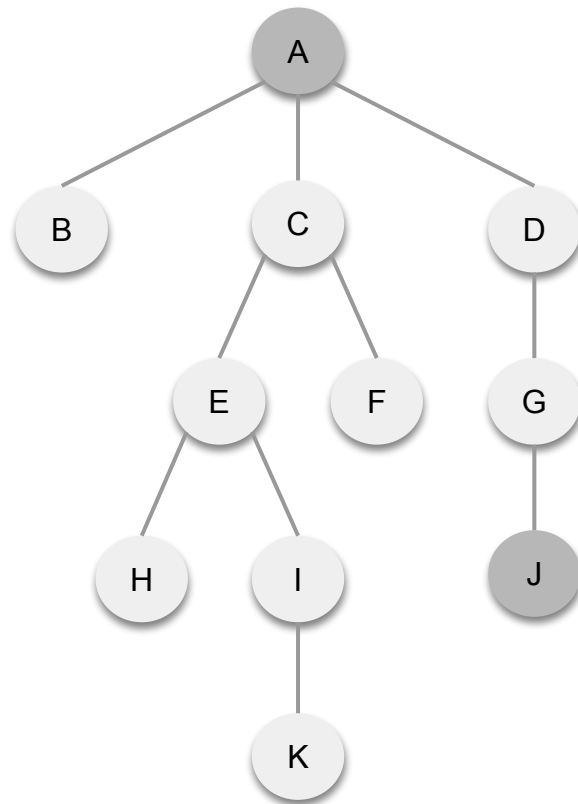
✓ Los nodos se expanden por niveles. Primero A con nivel 0. Luego B, C y D con nivel 1. Luego E, F y G con nivel 2. Tras expandir a G, se genera a J y el algoritmo se detiene.

2. A, D, B, C, G.

✓ Los nodos también se expanden por niveles. Primero A con nivel 0. Luego D, B y C con nivel 1 (no se expanden de izquierda a derecha pero esto no importa). Luego G en el nivel 2. Tras expandir a G, se genera a J y el algoritmo se detiene (el nivel 2 no fue expandido por completo, pero esto tampoco importa).

3. A, D, B, C, E, F, I, G.

✗ El nodo I con nivel 3 se expande antes que el nodo G con nivel 2.



Implementación de BFS

- ¿Qué nodo se elige de la frontera?

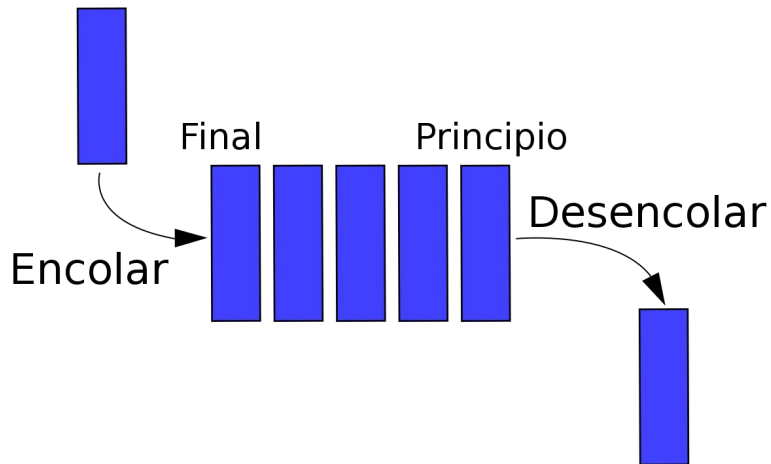
El de **menor** profundidad.

- ¿Cómo lo logramos?

Si los nuevos nodos generados (que están un nivel más abajo que sus padres) se agregan al **final** de la frontera, entonces en la parte de **adelante** de la frontera siempre se encuentran los de menor nivel.

¿Qué estructura de datos es ideal para esto? 🤔

TAD Cola



FIFO: el primero que entra es el primero que sale.

Operaciones disponibles:

- **Encolar**: agrega un nodo al final de la cola.
- **Desencolar**: elimina el nodo del principio de la cola y lo devuelve.
- **Vacía**: determina si la cola es vacía.



Algoritmo BFS de árbol

```
1 function TREE-BFS(problema) return solución o fallo
2    $n_0 \leftarrow \text{NODO}(\text{problema.estado-inicial}, \text{None}, \text{None}, 0)$ 
3   if (problema.test-objetivo( $n_0$ .estado)) then return solución( $n_0$ )
4   frontera  $\leftarrow$  Cola()
5   frontera.encolar( $n_0$ )
6   do
7     if frontera.vacia() then return fallo
8      $n \leftarrow$  frontera.desencolar()
9     forall  $a$  in problema.acciones( $n$ .estado) do
10        $s' \leftarrow$  problema.resultado( $n$ .estado,  $a$ )
11        $n' \leftarrow$  Nodo( $s'$ ,  $n$ ,  $a$ ,  $n$ .costo + problema.costo-individual( $n$ .estado, $a$ ))
12       if problema.test-objetivo( $s'$ ) then return solución( $n'$ )
13     frontera.encolar( $n'$ )
```

solución(n): retorna la solución escalando por los padres desde el nodo n hasta la raíz.

El test objetivo (líneas 3 y 12) se aplica **antes** de agregar un nodo a frontera.

Performance de TREE-BFS

— — —

Suponer que el árbol de búsqueda tiene un factor de ramificación b y que la menor profundidad de una solución es d .

Del inglés, b : branching factor y d : depth.

Complejidad de TREE-BFS

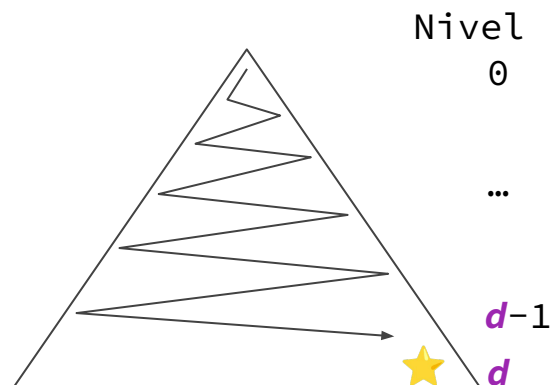
Si existen soluciones, ¿encuentra al menos una? 🤔

Completitud de TREE-BFS

Si existen soluciones, ¿encuentra al menos una? 🤔

✓ **Siempre*** encuentra una solución, aunque no se detecten caminos cíclicos. De hecho encuentra una de profundidad mínima (d).

*Salvo que el factor de ramificación sea **infinito**.



Optimalidad de TREE-BFS

Si existen soluciones, ¿encuentra una óptima? 🤔

Optimalidad de TREE-BFS

Si existen soluciones, ¿encuentra una óptima? 🤔

Depende de los costos...

✓ **Para costos individuales unitarios ($= 1$), siempre encuentra una solución óptima.**

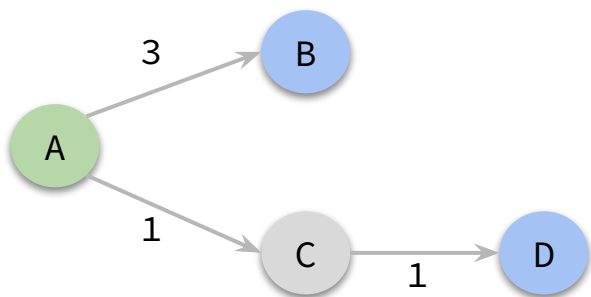
En estos casos, la **profundidad** de un nodo es exactamente su **costo de camino**. Y ya dijimos que BFS encuentra una solución de profundidad mínima (costo de camino mínimo).

Optimalidad de TREE-BFS

Para otros costos individuales, puede no ser óptimo.

Ejemplo.

Sea un problema con el siguiente grafo de espacio de estados. El estado A es inicial y los estados B y D son objetivo. El número sobre cada arco representa el costo individual de la acción correspondiente.



En este caso, TREE-BFS encuentra la solución $A \rightarrow B$ de nivel 1, la cual no es óptima porque su costo es 3 y existe la solución $A \rightarrow C \rightarrow D$ en el nivel 2 con costo 2.

Consumo de tiempo de TREE-BFS

— — —

¿Cuál es el consumo de tiempo? 🤔

Consumo de tiempo de TREE-BFS

¿Cuál es el consumo de tiempo? 🤔

En el peor caso, se generan todos los nodos hasta el nivel d :

$$(b^{d+1} - 1) / (b - 1) \leq b^{d+1}$$

Consumo de memoria de TREE-BFS

— — —

¿Cuál es el consumo
de memoria? 🤔

Consumo de memoria de TREE-BFS

El máximo número de nodos en la frontera es a lo sumo el número de nodos en el nivel d :

$$b^d$$




¿Cuál es el consumo de memoria? 🤔

Sin embargo, **sus ancestros no pueden liberarse de la memoria, ya que son necesarios para reconstruir la solución**. En el peor caso, se deben almacenar en memoria todos los nodos hasta el nivel d :

$$(b^{d+1} - 1) / (b - 1) \leq b^{d+1}$$

Performance de TREE-BFS

— — —

- **Compleitud.** 
- **Optimalidad.**
 -  Cuando las acciones tienen costo individual unitario.
 -  En otros casos.
- **Tiempo.** Se generan a lo sumo b^{d+1} nodos.
- **Memoria.** Se mantienen a lo sumo b^{d+1} nodos en memoria.

El consumo de tiempo y memoria es exponencial 🤯

Tiempos vs. memoria

— — —

Nivel <i>d</i>	Nodos $(10^{d+1} - 1)/9$	Tiempo	Memoria
2	111	0,11 milisegundos	111 kilobytes
4	11111	11 milisegundos	11 megabytes
6	$1,1 \times 10^6$	1,1 segundo	1,1 gigabyte
8	$1,1 \times 10^8$	2 minutos	110 gigabyte
10	$1,1 \times 10^{10}$	3 horas	11 terabyte
12	$1,1 \times 10^{12}$	13 días	1,1 petabyte
14	$1,1 \times 10^{14}$	3,5 años	110 petabyte
16	$1,1 \times 10^{16}$	349 años	11 exabyte

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

Tiempos vs. memoria

— — —

Nivel <i>d</i>	Nodos $(10^{d+1} - 1)/9$	Tiempo	Memoria
2	111	0,11 milisegundos	111 kilobytes
4	11111	11 milisegundos	11 megabytes
6	$1,1 \times 10^6$	1,1 segundo	1,1 gigabyte
8	$1,1 \times 10^8$	2 minutos	110 gigabyte
10	$1,1 \times 10^{10}$	3 horas	11 terabyte
12	$1,1 \times 10^{12}$	13 días	1,1 petabyte
14	$1,1 \times 10^{14}$	3,5 años	110 petabyte
16	$1,1 \times 10^{16}$	349 años	11 exabyte

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

Para BFS, el requerimiento de memoria es una complicación más seria que el de tiempo.

Hay estrategias con mejor consumo de memoria que BFS. Pero el consumo de tiempo sigue siendo un problema...

Por eso, problemas de búsqueda complejos no pueden resolverse con estrategias no-informadas, salvo casos pequeños.

Búsqueda primero en profundidad

Depth-First Search (DFS)

Los nodos del árbol de búsqueda se expanden en profundidad.

Comenzando por la raíz, se expande uno de los hijos del último nodo procesado.

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

— — —



Ejemplo - Vertido de agua

— — —

Vamos a resolver el problema de vertido de agua con el **algoritmo de búsqueda de árbol** usando la **estrategia de búsqueda primero en profundidad** (DFS).

Usaremos detección de ciclos para podar nodos (como veremos más adelante, es necesario para garantizar completitud).



Ejemplo - Vertido de agua

— — —

n_0
 $(7, 0, 0)$

FRONTERA = $\{n_0\}$



Ejemplo - Vertido de agua

— — —

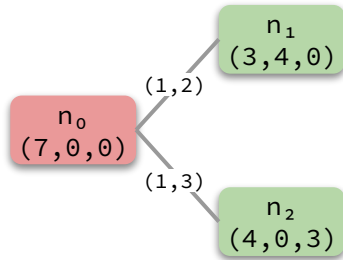
$$\begin{matrix} n_0 \\ (7, 0, 0) \end{matrix}$$

FRONTERA = {}



Ejemplo - Vertido de agua

— — —

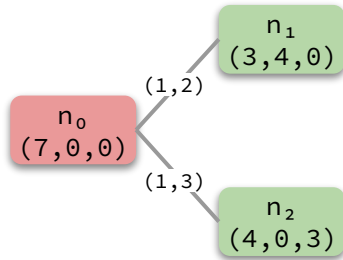


FRONTERA = $\{n_1, n_2\}$



Ejemplo - Vertido de agua

— — —



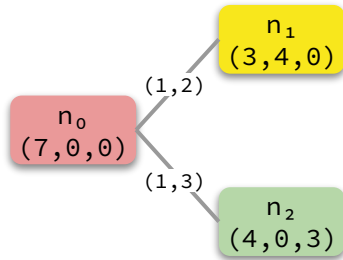
FRONTERA = $\{n_1, n_2\}$

Podemos expandir indistintamente a n_1 o n_2 , pues ambos son los de mayor nivel en la frontera. Elegimos por ejemplo a n_1 .



Ejemplo - Vertido de agua

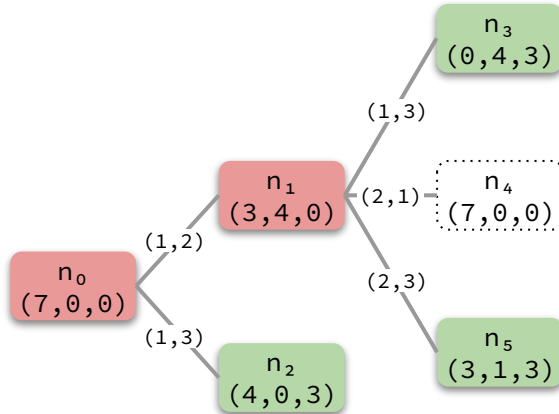
— — —



FRONTERA = $\{n_2\}$



Ejemplo - Vertido de agua



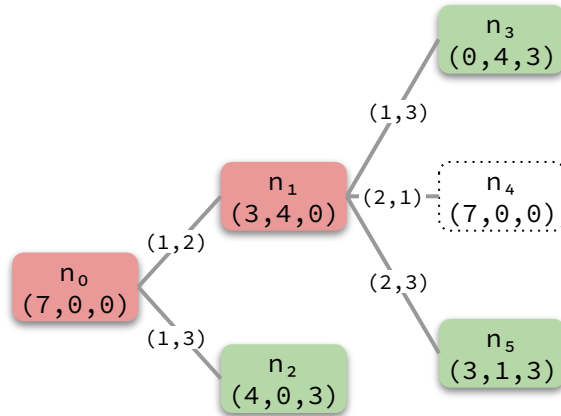
El nodo n_4 no se agrega a la frontera porque repite el estado del padre de su padre.

Estos nodos se representan con línea de punto.

FRONTERA = $\{n_2, n_3, n_5\}$



Ejemplo - Vertido de agua



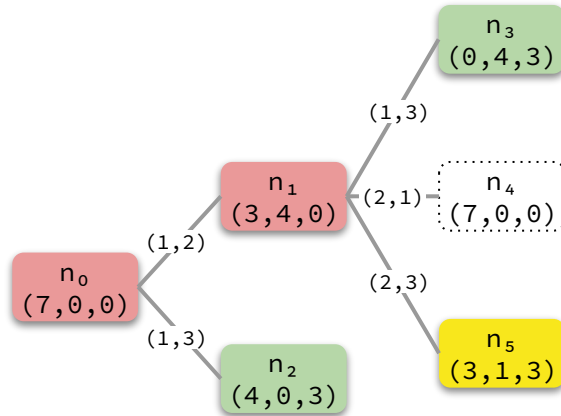
FRONTERA = $\{n_2, n_3, n_5\}$

Podemos expandir indistintamente a n_3 o n_5 , pues ambos son los de mayor nivel en la frontera. Elegimos por ejemplo a n_5 .



Ejemplo - Vertido de agua

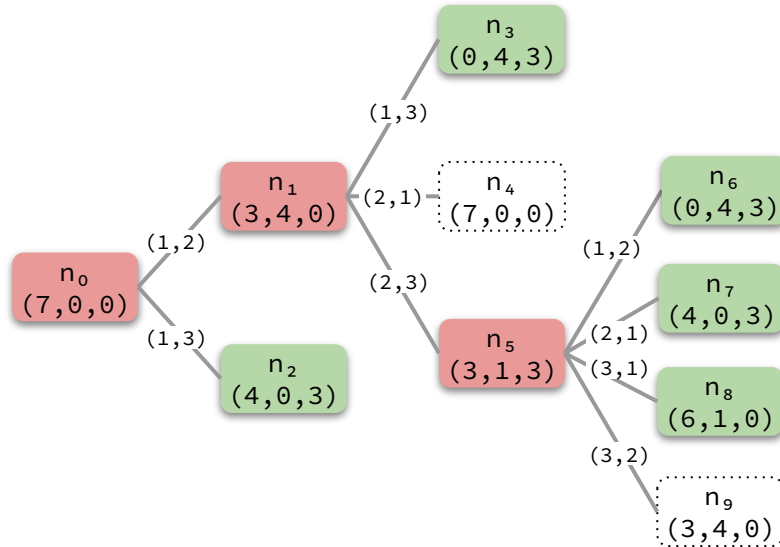
— — —



FRONTERA = $\{n_2, n_3\}$



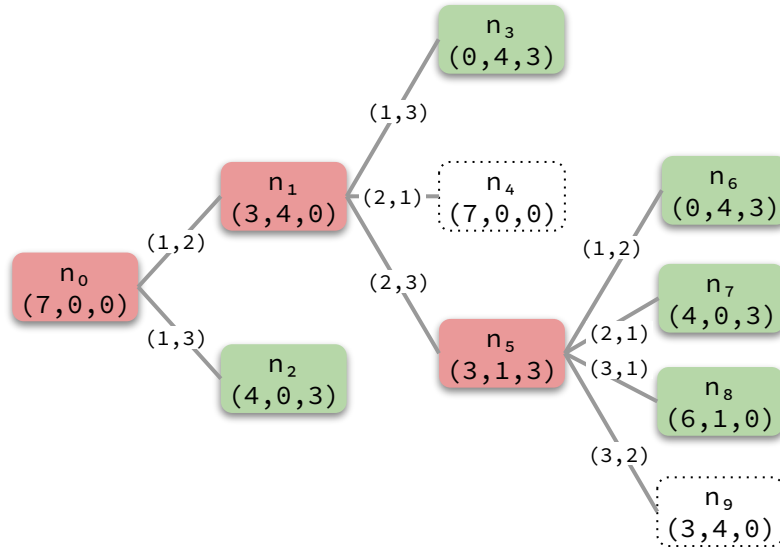
Ejemplo - Vertido de agua



FRONTERA = $\{n_2, n_3, n_6, n_7, n_8\}$



Ejemplo - Vertido de agua

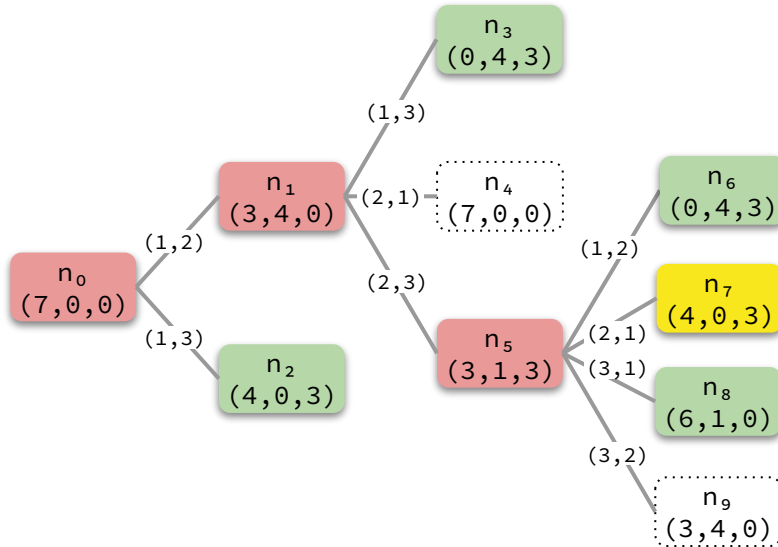


Podemos expandir indistintamente a n_6 , n_7 o n_8 , pues son los de mayor nivel en la frontera. Elegimos por ejemplo a n_7 .

FRONTERA = $\{n_2, n_3, n_6, n_7, n_8\}$



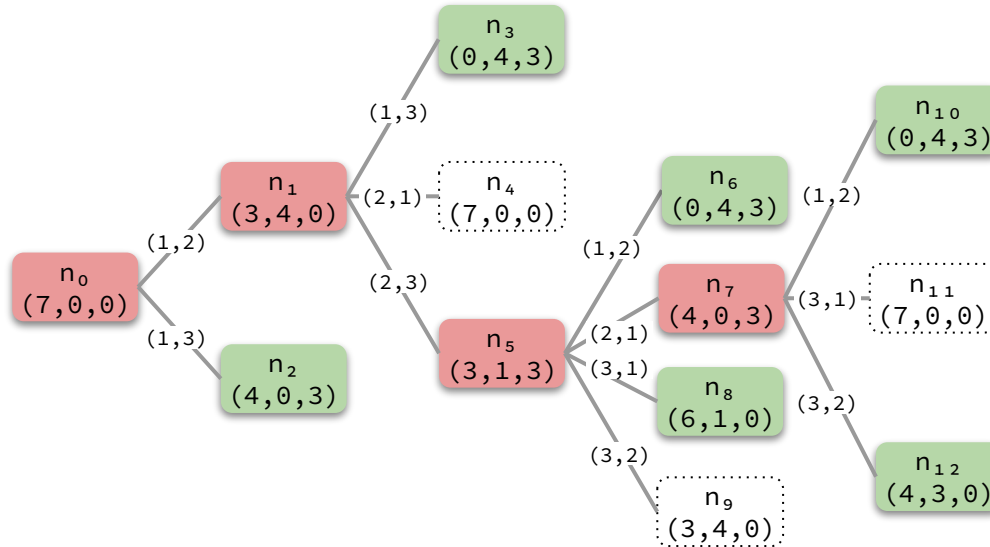
Ejemplo - Vertido de agua



FRONTERA = $\{n_2, n_3, n_6, n_8\}$



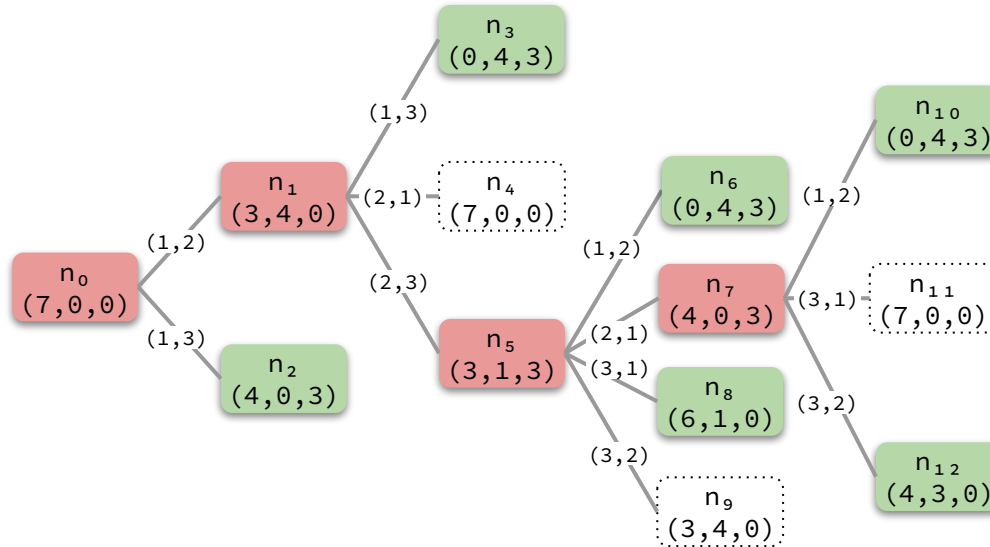
Ejemplo - Vertido de agua



FRONTERA = $\{n_2, n_3, n_6, n_8, n_{10}, n_{12}\}$



Ejemplo - Vertido de agua



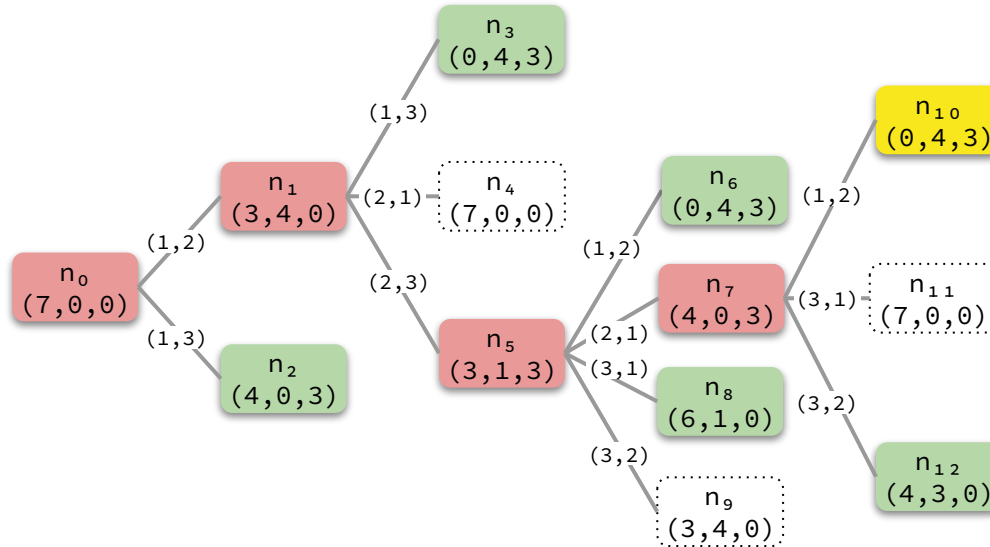
Observar que si los nuevos nodos se agregan al final de la frontera, entonces los nodos de mayor nivel se encuentran siempre al final. Más adelante volveremos a hablar de esto.

Podemos expandir indistintamente a n_{10} o n_{12} , pues ambos son los de mayor nivel en la frontera. Elegimos por ejemplo a n_{10} .

FRONTERA = $\{n_2, n_3, n_6, n_8, n_{10}, n_{12}\}$



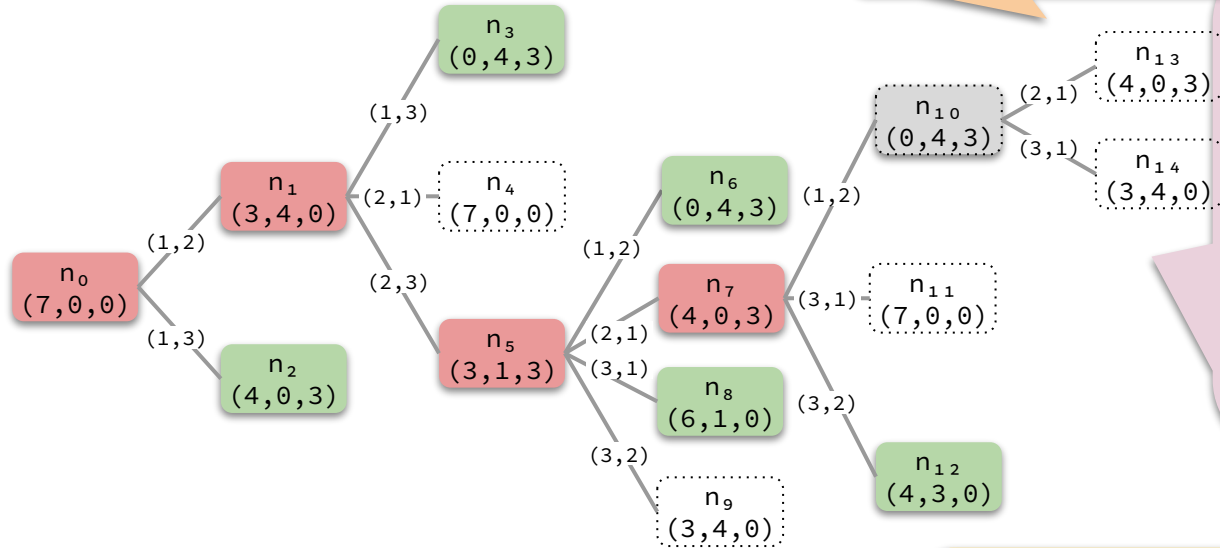
Ejemplo - Vertido de agua



FRONTERA = $\{n_2, n_3, n_6, n_8, n_{12}\}$



Ejemplo - Vertido de agua



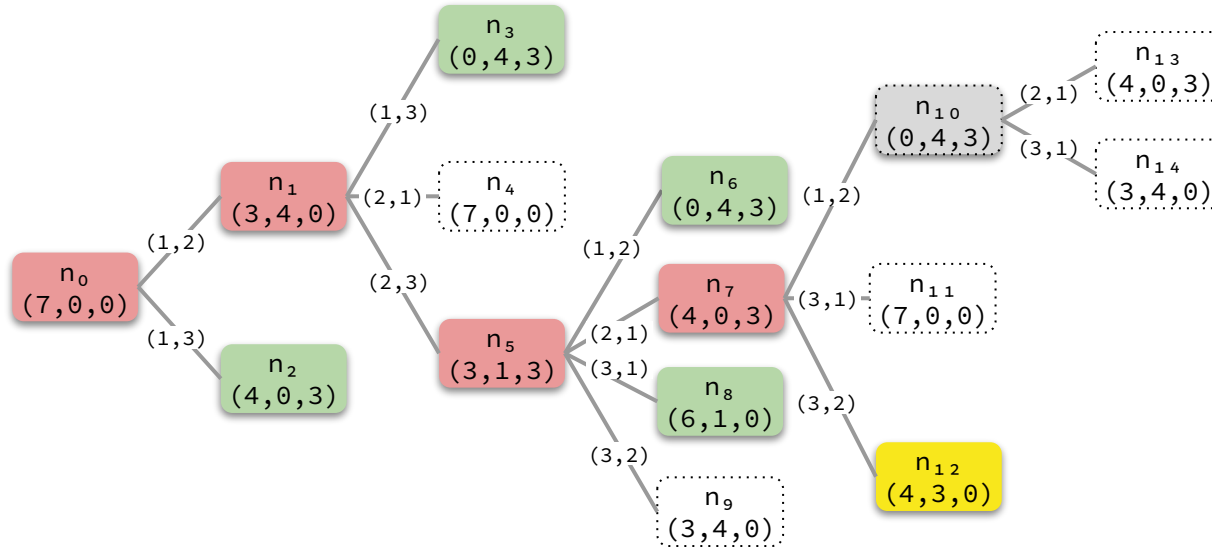
La búsqueda retrocede a n_7 , pues es el ancestro de n_{10} más profundo con hijos aún no expandidos, y expande a n_{12} . O lo que es equivalente, expande a n_{12} , pues es el de mayor nivel en la frontera.

Observar que n_{10} se puede liberar de la memoria, pues no está en la frontera y no es ancestro de ningún nodo de la frontera 🎉

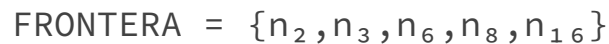
FRONTERA = $\{n_2, n_3, n_6, n_8, n_{12}\}$



Ejemplo - Vertido de agua

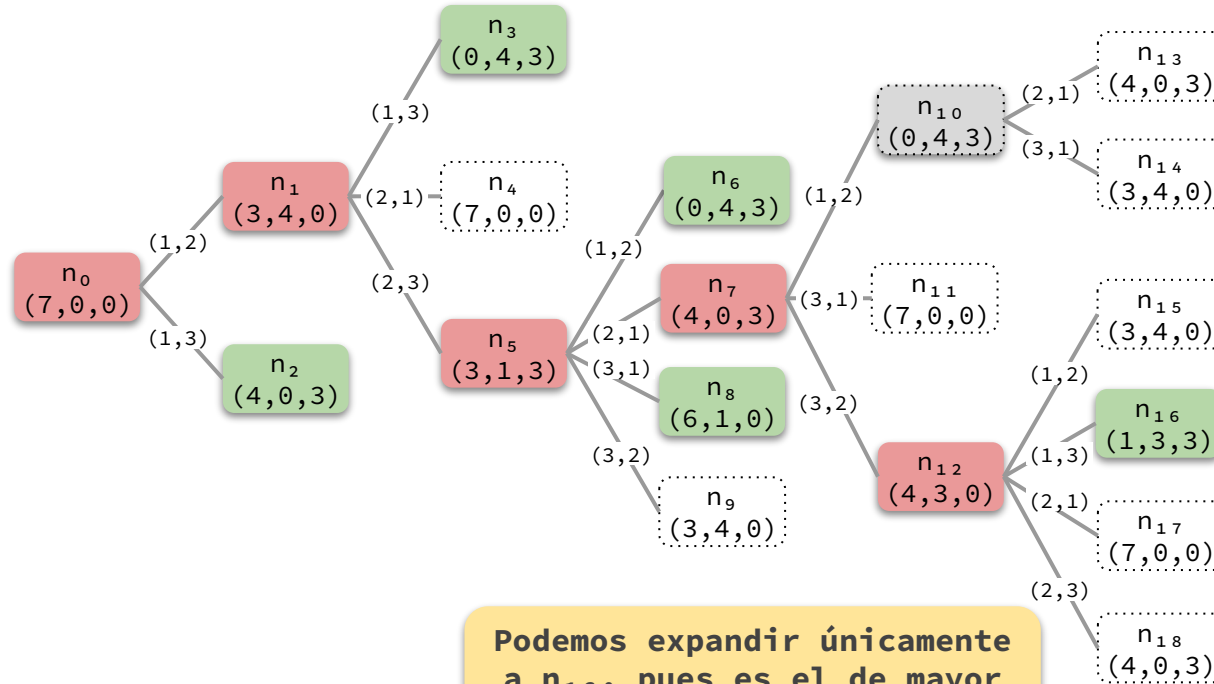


FRONTERA = $\{n_2, n_3, n_6, n_8\}$

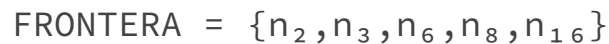




Ejemplo - Vertido de agua

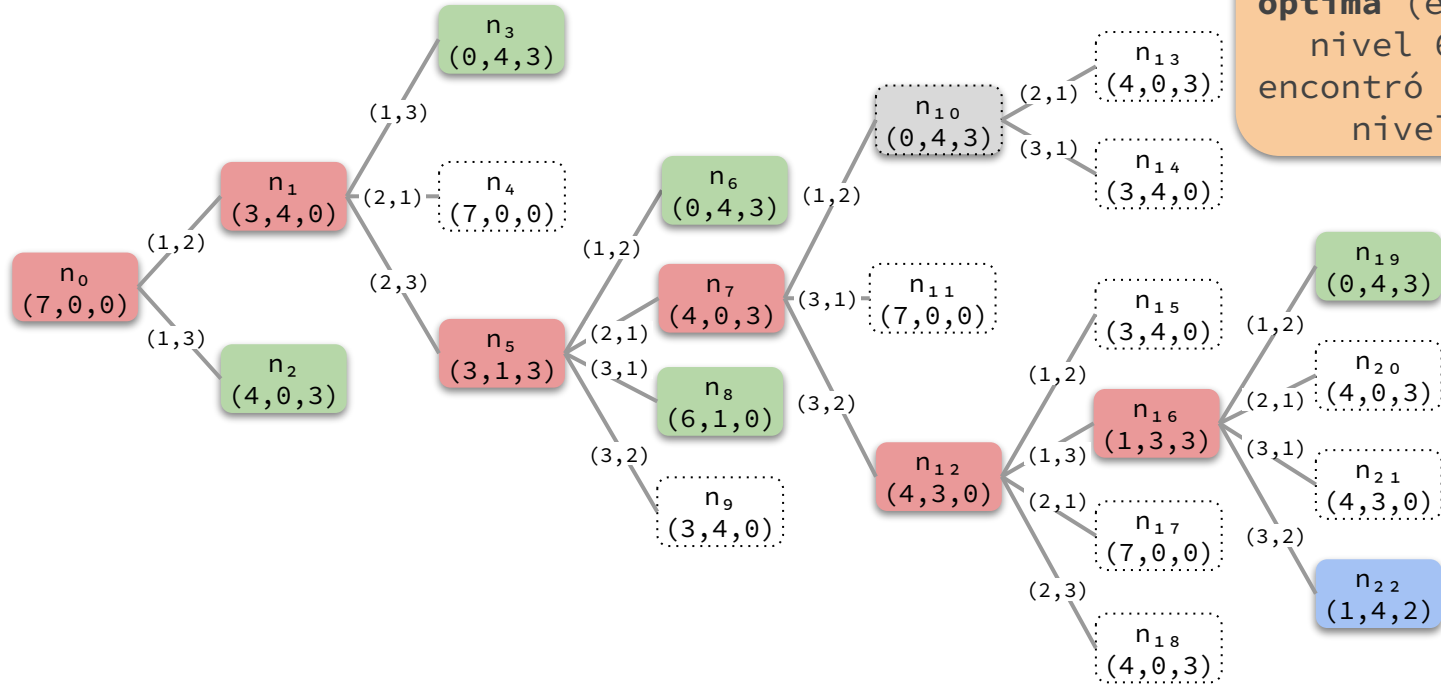


FRONTERA = $\{n_2, n_3, n_6, n_8, n_{16}\}$





Ejemplo - Vertido de agua



FRONTERA = $\{n_2, n_3, n_6, n_8, n_{16}, n_{19}\}$

Criterio de parada

En general, en la estrategia DFS **el test objetivo se llama antes de agregar un nuevo nodo a la frontera** (al igual que en BFS). Con respecto a llamarlo después de sacar de la frontera, se logra:

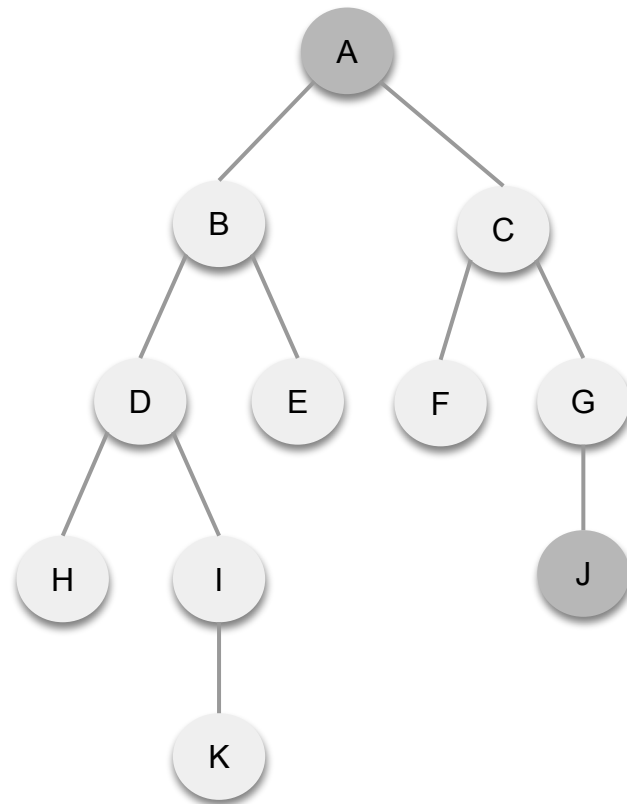
- ✅ **Menor consumo de tiempo** (aunque el ahorro es difícil de medir).
- 🚨 Para costos individuales no unitarios, nada garantiza que la solución encontrada antes sea mejor/igual/peor.



Ejercicio

Dado el siguiente árbol de búsqueda, donde A es la raíz y J es el único nodo objetivo, decidir cuáles secuencias se corresponden con posibles órdenes de expansión de nodos según la estrategia DFS.

1. A, C, G.
2. A, C, F, G.
3. A, B, E, C, G.





Respuestas

1. A, C, G.

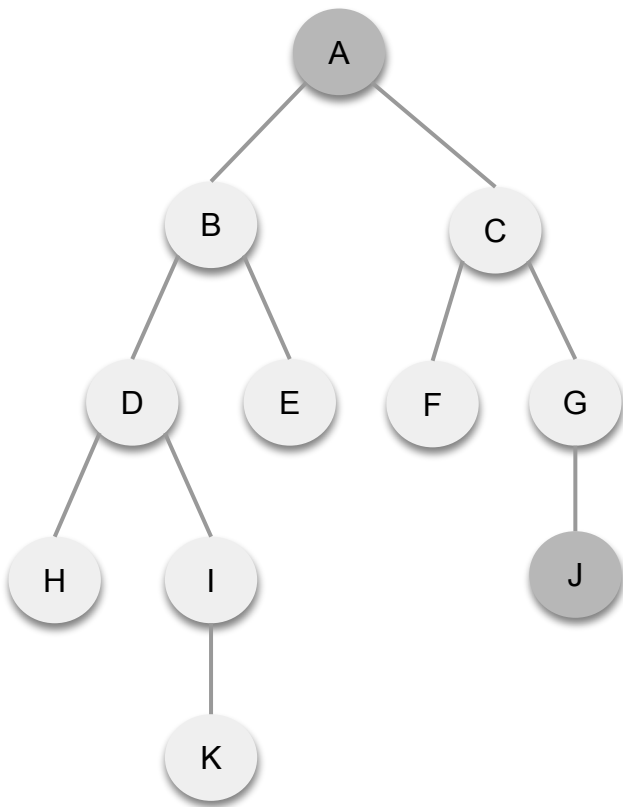
✓ El orden de expansión sigue un camino descendente en el árbol. El algoritmo se detiene tras expandir a G y generar a J.

2. A, C, F, G.

✓ El orden de expansión sigue un camino descendente en el árbol hasta F. Dado que F es una hoja, la búsqueda retrocede a su padre C y continúa expandiendo su otro hijo G. El algoritmo se detiene tras expandir a G y generar a J.

3. A, B, E, C, G.

✗ El orden de expansión sigue un camino descendente en el árbol hasta E. Pero dado que E es una hoja, la búsqueda tendría que haber retrocedido hasta su padre B y continuado expandiendo su otro hijo D.



Implementación de DFS

- ¿Qué nodo se elige de la frontera?

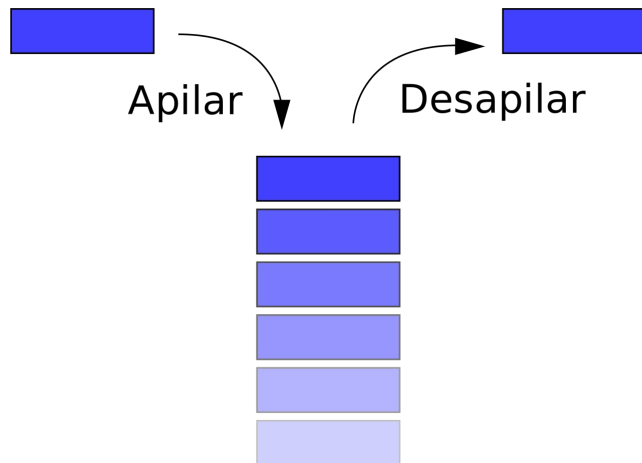
El de **mayor** profundidad.

- ¿Cómo lo logramos?

Si los nuevos nodos generados (que están un nivel más abajo que sus padres) se agregan al **final** de la frontera, entonces en la parte de **atrás** de la frontera siempre se encuentran los de mayor nivel.

¿Qué estructura de datos es ideal para esto? 🤔

TAD Pila



LIFO: el último que entra es el primero que sale.

Operaciones disponibles:

- **Apilar**: agrega un nodo en el tope de la pila.
- **Desapilar**: elimina el nodo del tope de la pila y lo devuelve.
- **Vacía**: determina si la pila es vacía.



Algoritmo DFS de árbol

```
1 function TREE-DFS(problema) return solución o fallo
2    $n_0 \leftarrow \text{NODO}(\text{problema.estado-inicial}, \text{None}, \text{None}, 0)$ 
3   if (problema.test-objetivo( $n_0$ .estado)) then return solución( $n_0$ )
4   frontera  $\leftarrow$  Pila()
5   frontera.apilar( $n_0$ )
6   do
7     if frontera.vacia() then return fallo
8      $n \leftarrow$  frontera.desapilar()
9     forall  $a$  in problema.acciones( $n$ .estado) do
10        $s' \leftarrow$  problema.resultado( $n$ .estado,  $a$ )
11        $n' \leftarrow$  Nodo( $s'$ ,  $n$ ,  $a$ ,  $n$ .costo + problema.costo-individual( $n$ .estado, $a$ ))
12       if not hay_ciclo( $s'$ ,  $n$ ) then
13         if problema.test-objetivo( $s'$ ) then return solución( $n'$ )
14         frontera.apilar( $n'$ )
```

Performance de TREE-BFS

Suponer que el árbol de búsqueda tiene un factor de ramificación b y que tiene m niveles (**altura** del árbol).

Es decir, suponemos que el árbol de búsqueda (y en consecuencia el espacio de estados) es **finito**.

Notar que $m \geq d$, siendo d la menor profundidad con un nodo objetivo.

Del inglés, b : branching factor, d : depth y m : maximum depth.

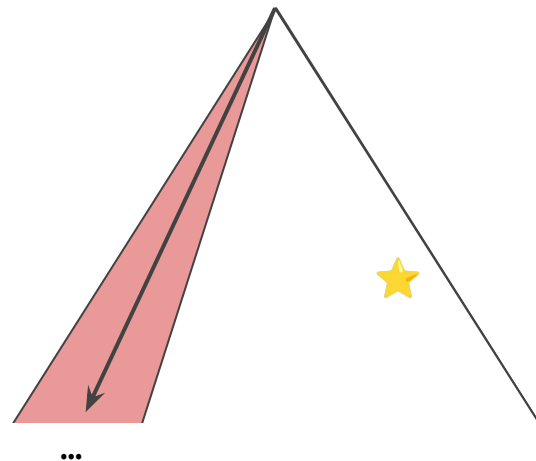
Complejidad de TREE-DFS

Si existen soluciones, ¿encuentra al menos una? 🤔

Completitud de TREE-DFS

Si existen soluciones, ¿encuentra al menos una? 🤔

❌ Si **no se detectan caminos cíclicos**, entonces el árbol de búsqueda podría ser infinito y DFS podría seguir un camino descendente infinito, sin nunca pasar por un nodo objetivo.

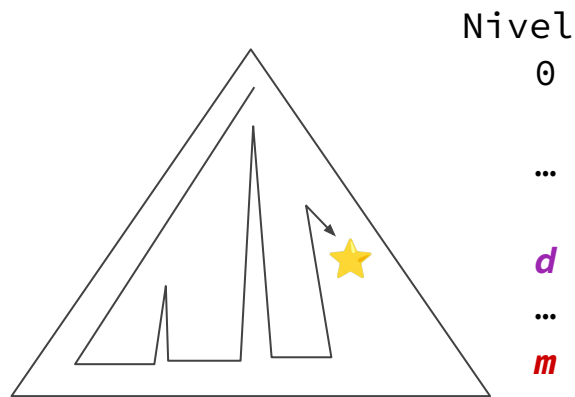


Completitud de TREE-DFS

Si existen soluciones, ¿encuentra al menos una? 🤔

✓ Si **se detectan caminos cíclicos**, entonces el árbol de búsqueda es **finito*** y DFS siempre encuentra una solución.

* Siempre que el espacio de estados sea finito.



Optimalidad y consumo de tiempo de TREE-DFS

— — —

Si existen soluciones
¿encuentra una
óptima? 🤔

¿Cuál es el consumo
de tiempo? 🤔

Optimalidad y consumo de tiempo de TREE-DFS

Si existen soluciones
¿encuentra una
óptima? 🤔

✗ No, ya lo sabemos del ejemplo anterior.

¿Cuál es el consumo
de tiempo? 🤔

En el peor caso, se generan todos los nodos del árbol de búsqueda:

$$(b^{m+1}-1)/(b-1) \leq b^{m+1}$$

Notar que m puede ser mucho mayor que d .

Consumo de memoria de TREE-DFS

— — —

¿Cuál es el consumo de memoria? 🤔

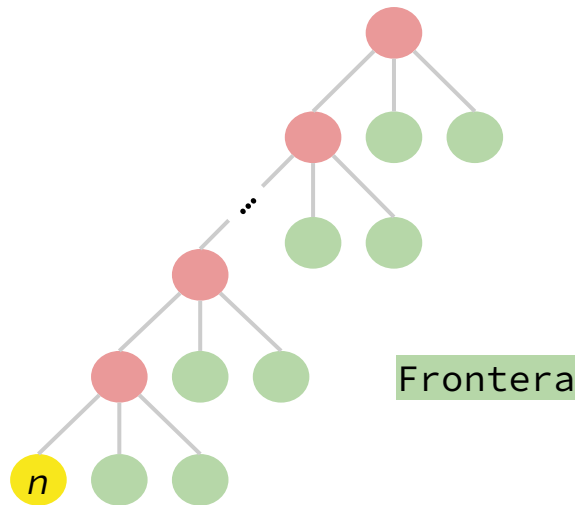
Consumo de memoria de TREE-DFS

¿Cuál es el consumo de memoria? 🤔

Es necesario mantener en memoria los **nodos del camino desde n a la raíz** (a lo sumo m nodos) y a la frontera, que contiene a **sus hermanos no expandidos** (a lo sumo b nodos por cada nodo del camino).

El máximo número de nodos en memoria está acotado por:

$$b \cdot m$$



Performance de TREE-DFS

— — —

- **Compleitud.**

✗ Si el espacio de estados es infinito o es finito pero no se detectan ciclos.

✓ Si el espacio de estados es finito y se detectan ciclos.

- **Optimalidad.** ✗

- **Tiempo.** Se generan a lo sumo b^{m+1} nodos. Si se detectan ciclos, el consumo de tiempo es $m \cdot b^{m+1}$.

- **Memoria.** Se mantienen a lo sumo $b \cdot m$ nodos en memoria.

El consumo de memoria es lineal en la altura 🎉

... pero no se garantiza optimalidad 🙄

TREE-BFS vs. TREE-DFS

— — —

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

BFS consume:

11 exabytes

y DFS consume:

160 kilobytes

es decir, $6,9 \times 10^{12}$ veces menos memoria.

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.

Adaptación a búsqueda de grafo



Algoritmo BFS de grafo

— — —

La adaptación de BFS de **árbol** a **grafo** ya la vimos en la Unidad 1.

- **ALCANZADOS**: conjunto de estados alcanzados.
- Al expandir un nodo, se descartan (**podan**) los hijos con estados ya alcanzados. Para los restantes, se marcan sus estados como alcanzados y se encolan a la frontera.



Algoritmo BFS de grafo

```
1 function GRAPH-BFS(problema) return solución o fallo
2    $n_0 \leftarrow \text{NODO}(\text{problema.estado-inicial}, \text{None}, \text{None}, 0)$ 
3   if (problema.test-objetivo( $n_0$ .estado)) then return solución( $n_0$ )
4   frontera  $\leftarrow \text{Cola}()$ 
5   frontera.encolar( $n_0$ )
6   alcanzados  $\leftarrow \{n_0.\text{estado}\}$ 
7   do
8     if frontera.vacía() then return fallo
9      $n \leftarrow \text{frontera.desencolar}()$ 
10    forall  $a$  in problema.acciones( $n$ .estado) do
11       $s' \leftarrow \text{problema.resultado}(n.\text{estado}, a)$ 
12      if  $s'$  is not in alcanzados then
13         $n' \leftarrow \text{Nodo}(s', n, a, n.\text{costo} + \text{problema.cost-individual}(n.\text{estado}, a))$ 
14        if problema.test-objetivo( $s'$ ) then return solución( $n'$ )
15        alcanzados.insertar( $s'$ )
16        frontera.encolar( $n'$ )
```

Performance de GRAPH-BFS

— — —

¿GRAPH-BFS preserva la **completitud** y **optimalidad** de TREE-BFS? 🤔

Performance de GRAPH-BFS

— — —

¿GRAPH-BFS preserva la **completitud** y **optimalidad** de TREE-BFS? 🤔

- ¿Cómo justificamos que la poda es **segura**?

Si la expansión genera un nodo con un estado alcanzado previamente, entonces ese estado se alcanzó por primera vez en un nivel menor o igual. Luego, **el nodo podado conduce a un camino redundante de largo mayor o igual.**

- Por lo tanto, **es completo y óptimo para costos individuales unitarios.**

Performance de GRAPH-BFS

— — —

¿Cuál es el consumo de **tiempo** y **memoria** de GRAPH-BFS comparado con TREE-BFS? 🤔

Performance de GRAPH-BFS

— — —

¿Cuál es el consumo de **tiempo** y **memoria** de GRAPH-BFS comparado con TREE-BFS? 🤔

La respuesta es difícil porque depende del problema...

- Si hay muchos caminos redundantes, el ahorro de GRAPH-BFS puede ser significativo. Aunque en el peor caso, se generan **tantos nodos como estados alcanzables**.
- Si no hay caminos redundantes, consumen mismo tiempo y GRAPH-BFS más memoria por los estados alcanzados, aunque no es significativo (recordar que TREE-BFS también recuerda la frontera y sus ancestros).
- Existen análisis más sofisticados (factor de ramificación promedio), nosotros no los vamos a ver.



Algoritmo DFS de grafo

— — —

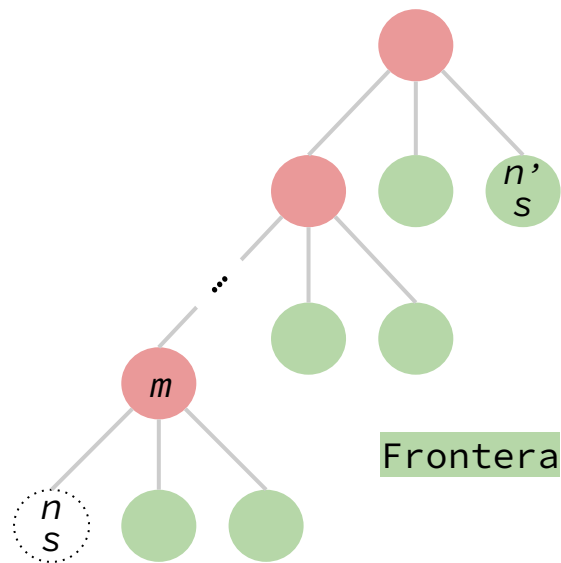
En la adaptación de DFS de **árbol** a **grafo** existe un problema...



Algoritmo DFS de grafo

Suponer que al expandir un nodo m , se genera un hijo n con un estado s tal que en la frontera hay un nodo n' con el mismo estado s en algún nivel anterior (generado previamente).

Manteniendo un conjunto de estados **alcanzados**, el nodo n sería descartado (pues s ya fue alcanzado al generar a n'), cuando lo correcto sería descartar a n' pues tiene **menor profundidad**.

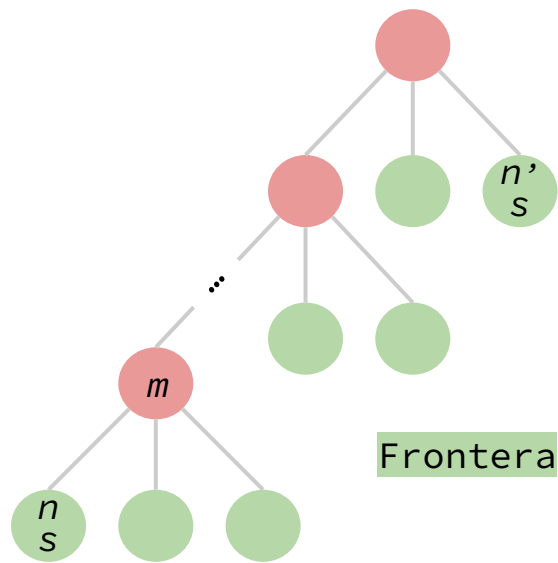




Algoritmo DFS de grafo

En lugar de mantener un conjunto de estados alcanzados, se mantiene un conjunto de estados **expandidos** (provenientes de nodos ya sacados de la frontera).

Ahora, en la frontera pueden aparecer **nodos con estado repetidos**, pero solo se permite expandir el de mayor profundidad
⇒ **Se agrega un control que evita expandir un nodo con un estado ya expandido.**






Algoritmo DFS de grafo

```
1 function GRAPH-DFS(problema) return solución o fallo
2    $n_0 \leftarrow \text{NODO}(\text{problema.estado-inicial}, \text{None}, \text{None}, 0)$ 
3   if problema.test-objetivo( $n_0$ .estado) then return solución( $n_0$ )
4   frontera  $\leftarrow$  Pila()
5   frontera.apilar( $n_0$ )
6   expandidos  $\leftarrow$  {}
7   do
8     if frontera.vacía() then return fallo
9      $n \leftarrow$  frontera.desapilar()
10    if  $n$ .estado is in expandidos then continue
11    expandidos.insertar( $n$ .estado)
12    forall  $a$  in problema.acciones( $n$ .estado) do
13       $s' \leftarrow$  problema.resultado( $n$ .estado,  $a$ )
14      if  $s'$  is not in expandidos then
15         $n' \leftarrow$  Nodo( $s'$ ,  $n$ ,  $a$ ,  $n$ .costo + problema.costo-individual( $n$ .estado, $a$ ))
16        if problema.test-objetivo( $s'$ ) then return solución( $n'$ )
17      frontera.apilar( $n'$ )
```

Control que evita expandir un estado ya expandido.

Performance de GRAPH-DFS

— — —

- Al igual que TREE-BFS, es **completo** en espacios de estados finitos y puede ser **subóptimo** (devolver solución no óptima).
 - El **consumo de tiempo** depende de la cantidad de caminos redundantes del problema (similar a GRAPH-BFS).
-  **El consumo de memoria aumenta.** Deja de ser lineal en la altura, pues ahora se almacenan todos los estados **expandidos**, que pueden ser tantos como el número de estados alcanzables (similar a GRAPH-BFS).

GRAPH-BFS vs. GRAPH-DFS

— — —

En el peor caso, tienen el mismo requerimiento de tiempo y memoria, pero GRAPH-BFS es óptimo (para costos individuales unitarios).

Luego, **se prefiere a GRAPH-BFS como algoritmo de búsqueda de grafo.**



Próximamente

— — —

¿Es posible que **BFS** garantice **optimalidad** cuando los costos individuales son **arbitrarios**?

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