



Ayudantía 4

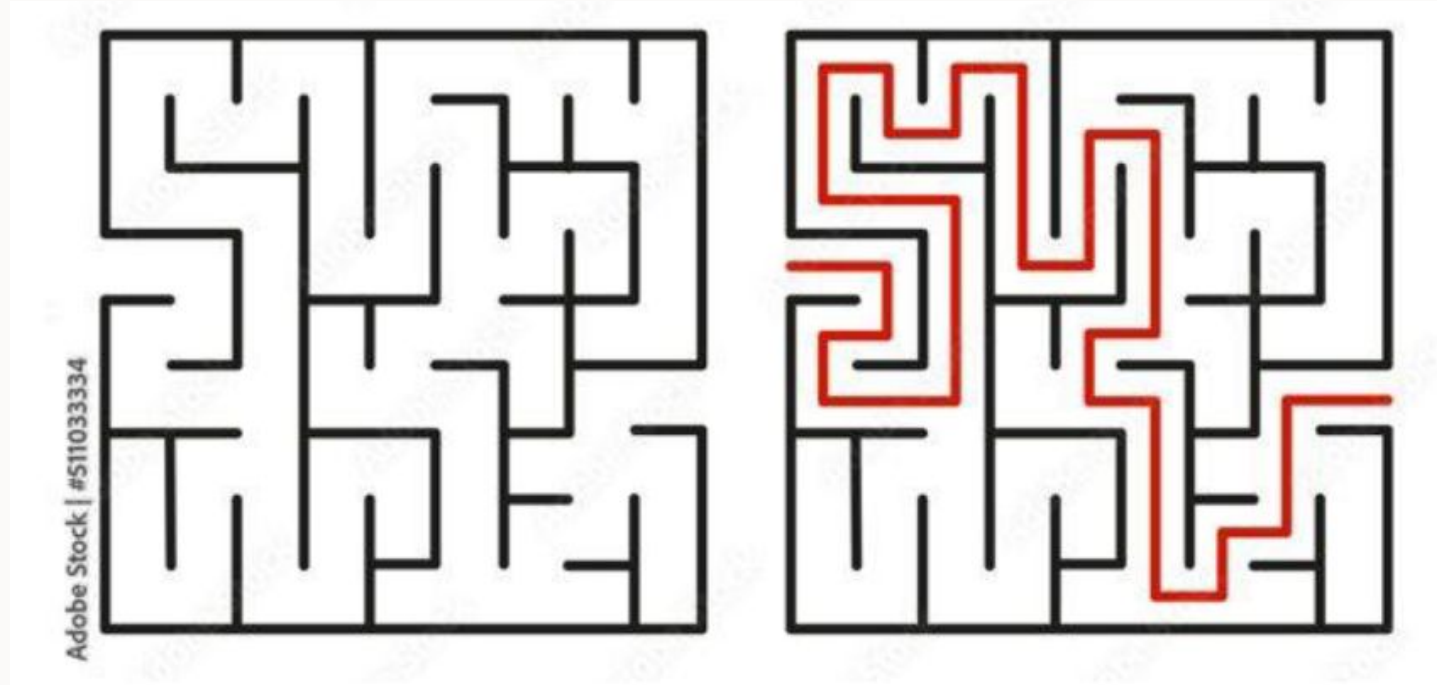
Introducción a la búsqueda

Por Blanca Romero y Felipe Vidal

8 de septiembre 2023



¿Qué es búsqueda?





Formalmente...

- **Estado** (s) : Configuración específica de un sistema.
- **Acción** (a) : Función que hace pasar al sistema de un estado (s) a otro (s').
- **Conjunto de acciones** (A) : Conjunto de todas las acciones posibles.
- **Espacio de Búsqueda** (S) : Conjunto de todos los estados posibles.
- **Grafo de búsqueda** : Todos los estados posibles conectados por las acciones que los unen



¿Cómo se define un problema de búsqueda?

- G es un subconjunto de S con los estados objetivo
- s_{init} el estado inicial
- La notación de un problema de búsqueda es...

$$(S, A, s_{init}, G)$$



1	2	3
4		6
7	8	5

Caso típico: Puzzle de 8



Formalizamos el problema

Problema de búsqueda (S, A, s_{init}, G)

S = conjunto de estados

A = conjunto de acciones

s_{init} = estado inicial

G = conjunto de estados finales

1	2	3
4		6
7	8	5

Estado inicial

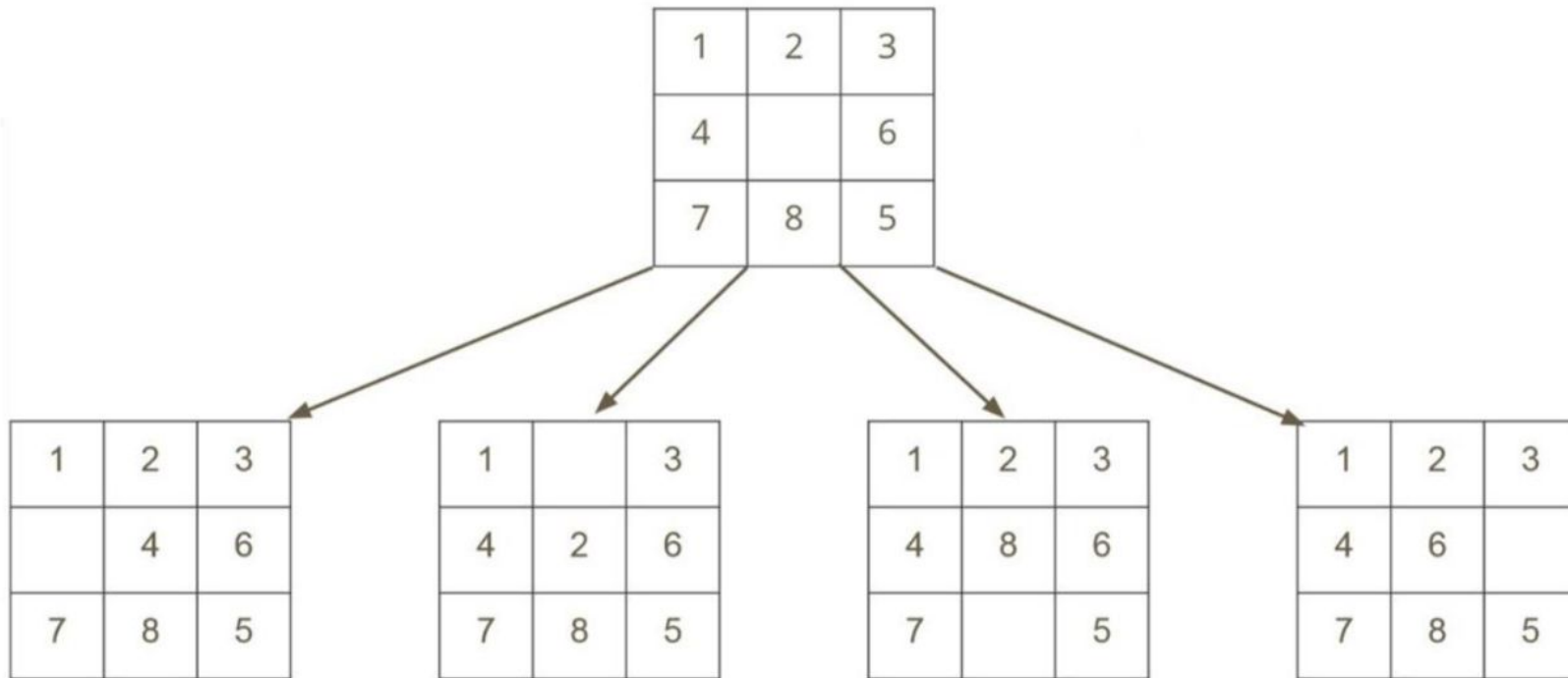


	1	2
3	4	5
6	7	8

Estado final $\in G$

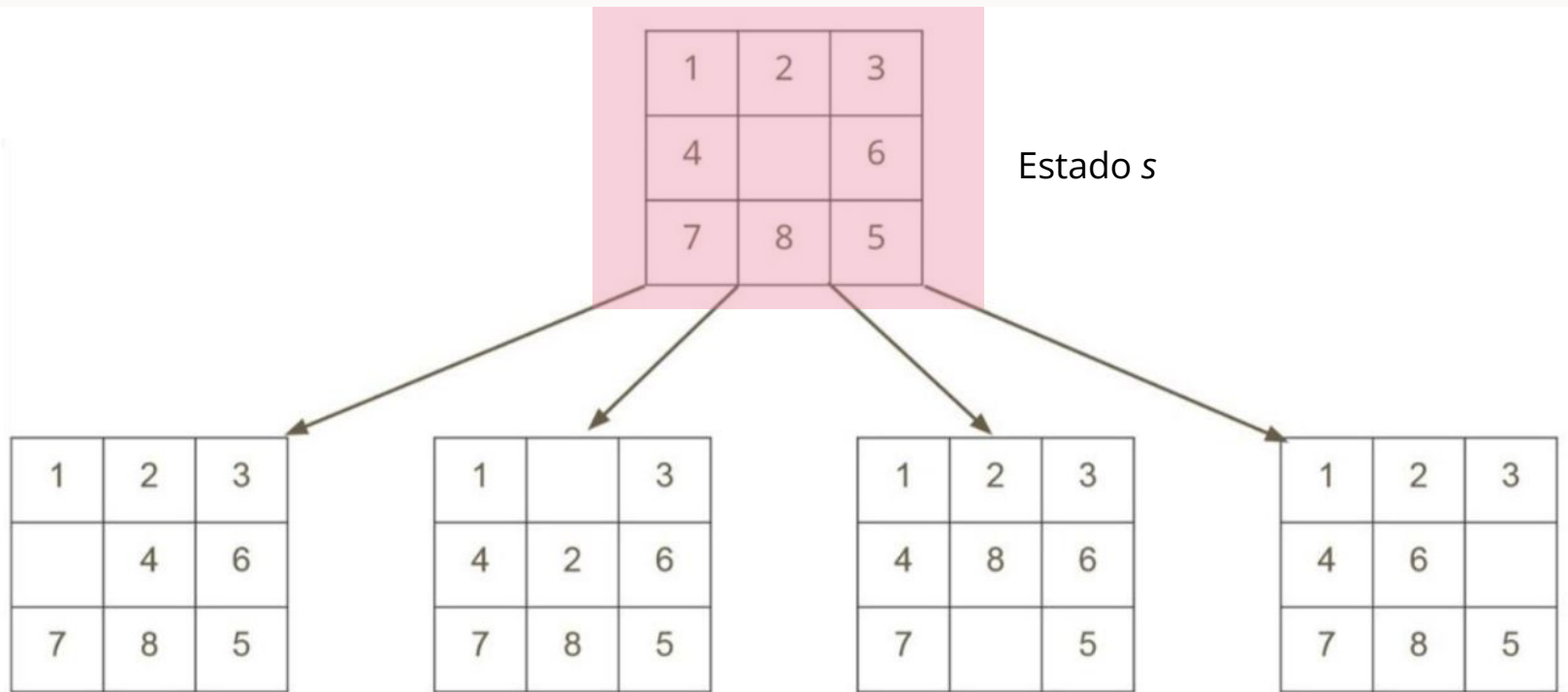


¿Cuáles son estados? ¿Cuáles son acciones?



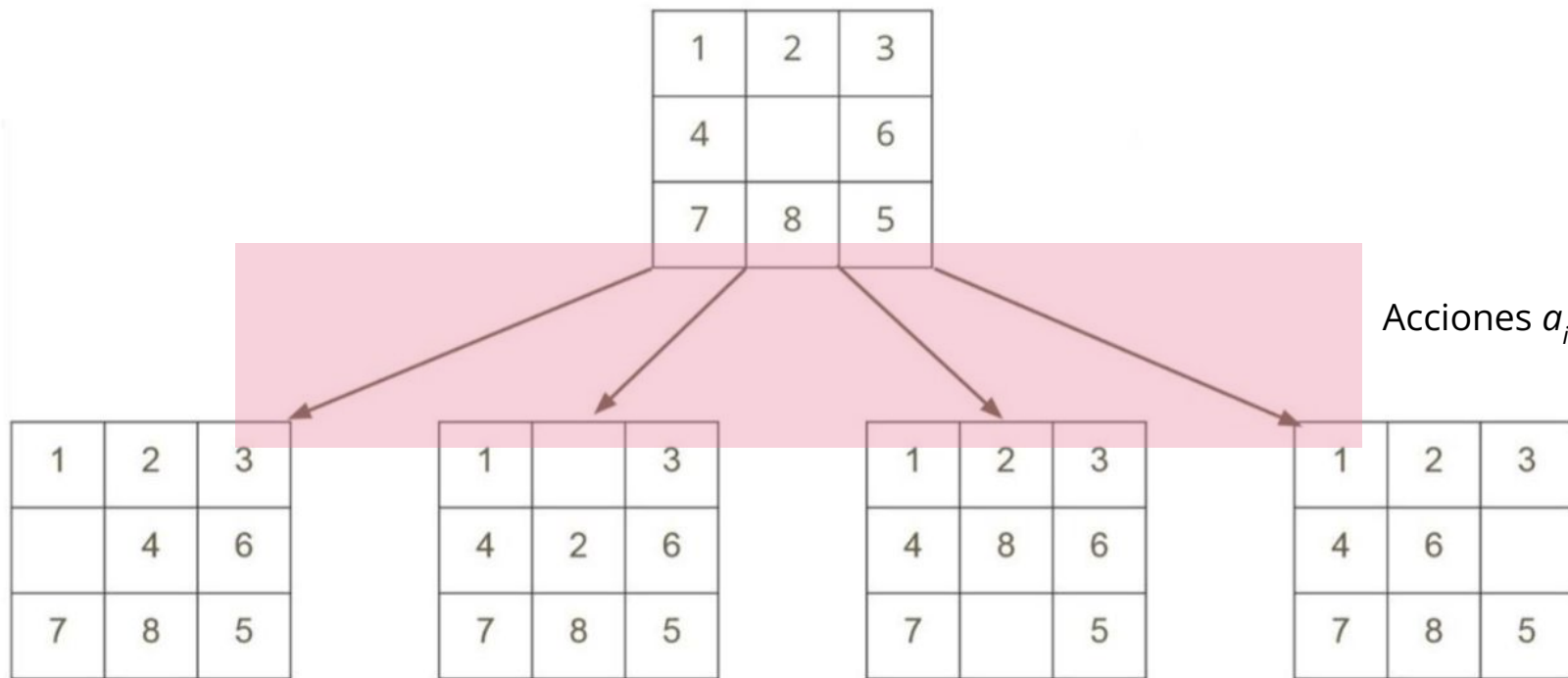


¿Cuáles son estados? ¿Cuáles son acciones?



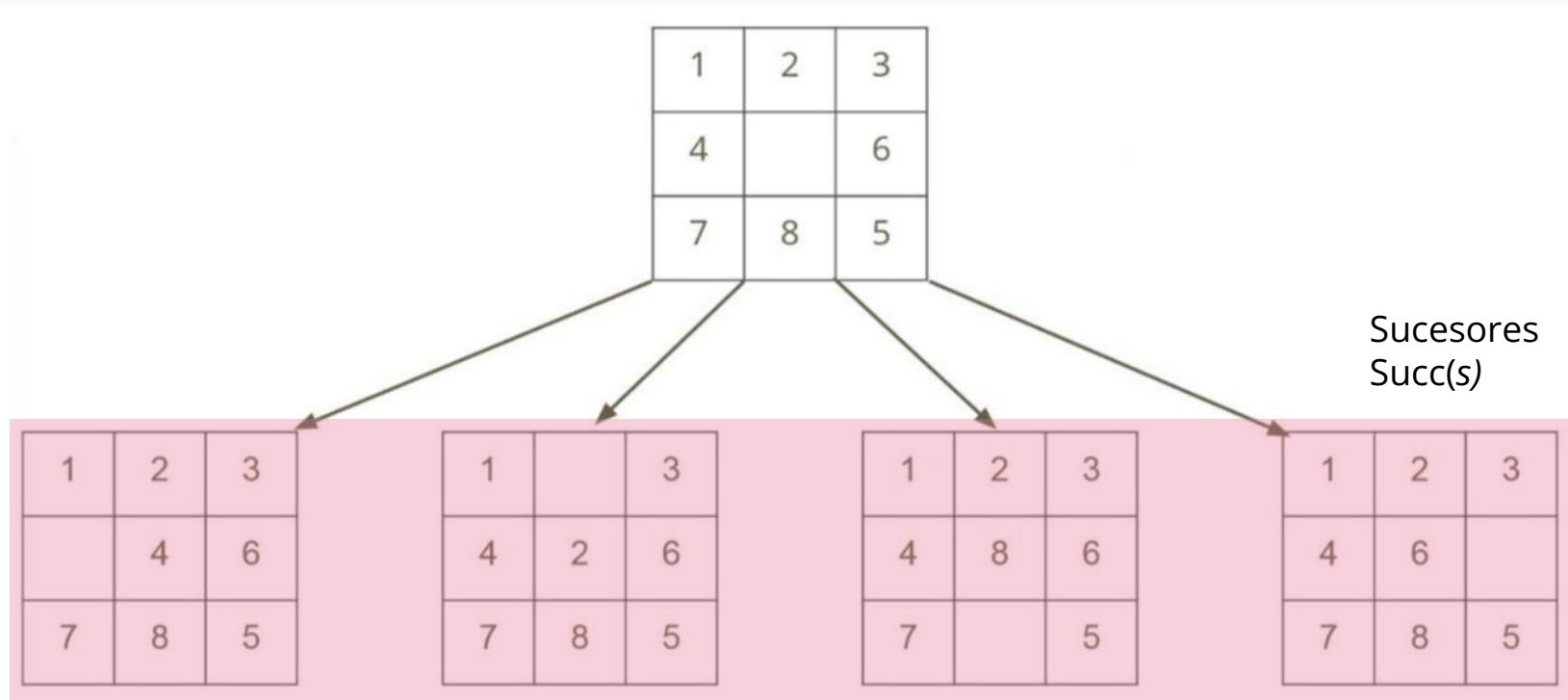


¿Cuáles son estados? ¿Cuáles son acciones?



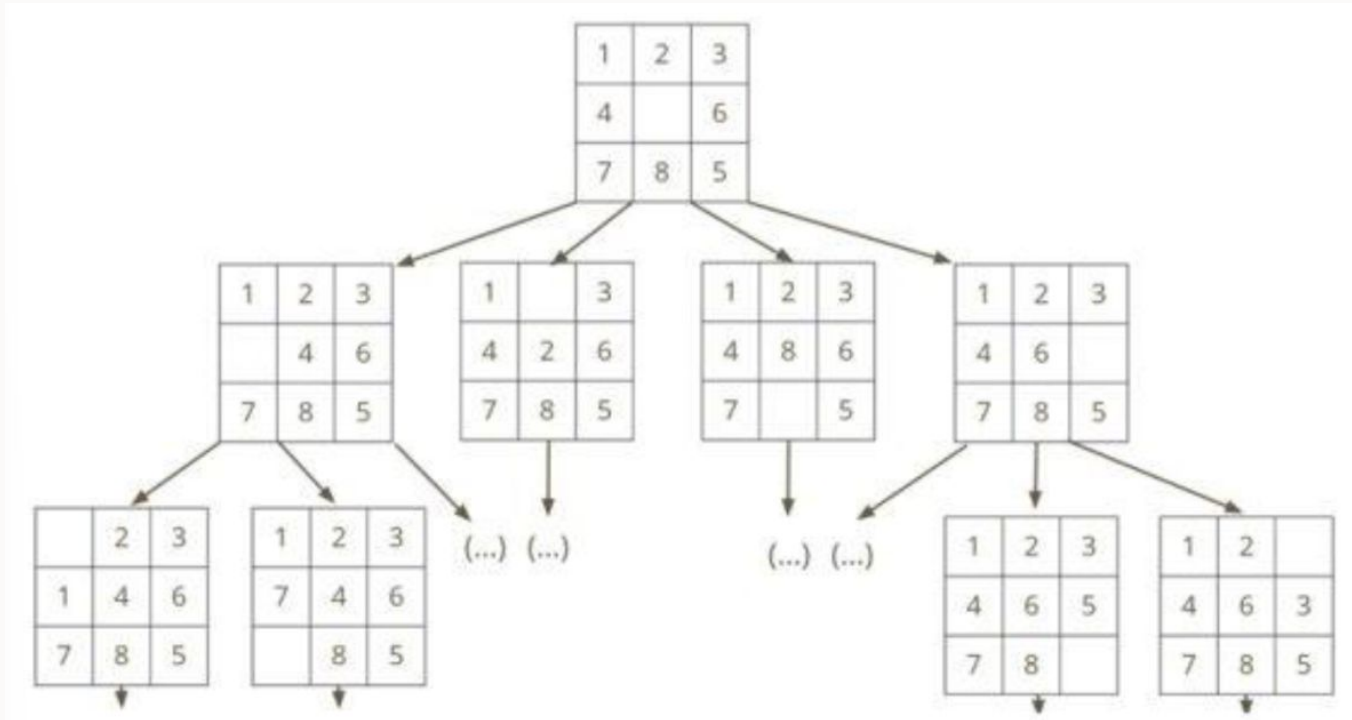


¿Cuáles son estados? ¿Cuáles son acciones?





Grafo de búsqueda





Espacio de búsqueda

1		3
5	2	4
6	7	8

1	3	
5	2	4
6	7	8

1	2	3
5		4
6	7	8

1	2	3
5	4	8
6	7	

...

1	2	3
	5	4
6	7	8

1	2	3
5	4	
6	7	8

1	2	3
5	7	4
6	8	

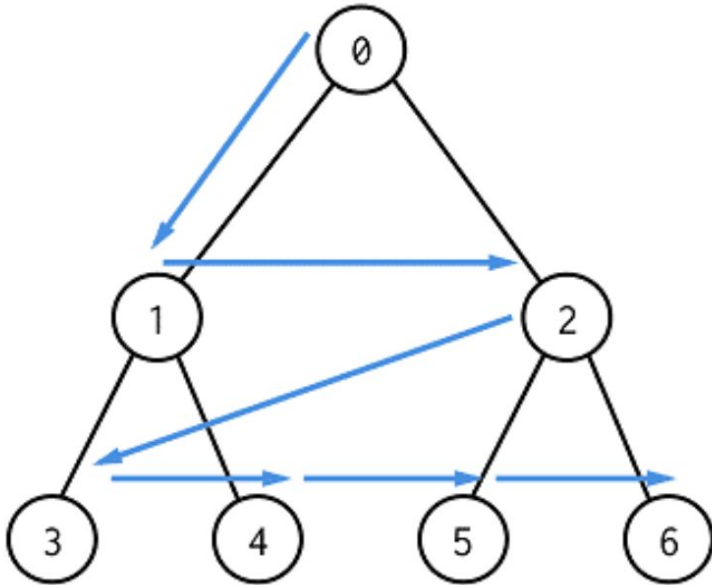
1	2	3
6	5	4
7		8



Algoritmos de búsqueda

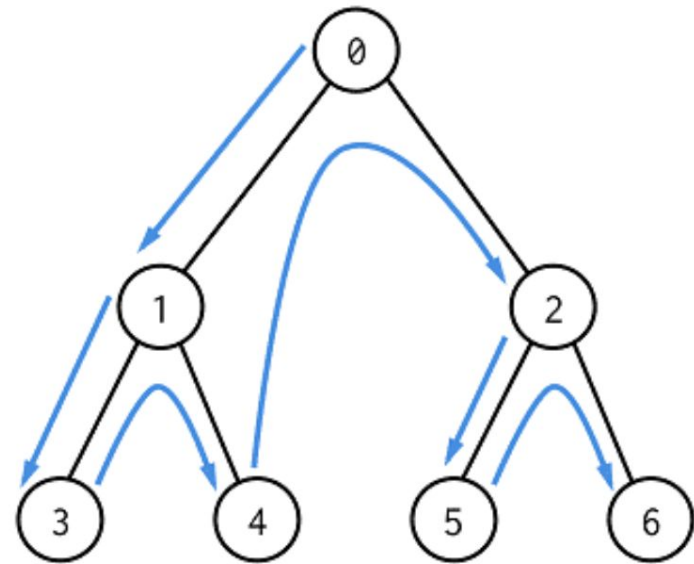
BFS - Breadth First Search

Búsqueda en anchura



DFS - Depth First Search

Búsqueda en profundidad





Algoritmos de búsqueda

BFS - Breadth First Search

Búsqueda en anchura

- Memoria usada: $O(b^p)$
- Tiempo requerido: $O(b^p)$

DFS - Depth First Search

Búsqueda en profundidad

- Memoria usada: $O(bm)$
- Tiempo requerido: $O(b^m)$

b : factor de ramificación (cuántas ramas salen de un nodo)

p : profundidad a la que se encuentra la solución

m : largo de la rama más larga del árbol de búsqueda

Algoritmo de búsqueda



El siguiente es un algoritmo de búsqueda genérico.

Input: Un problema de búsqueda (S, A, s_{init}, G)

Output: Un nodo objetivo

Open es un conjunto vacío

Closed es un conjunto vacío

Insert s_{init} a *Open*

$\text{parent}(s_{init}) = \text{null}$

While *Open* $\neq \emptyset$:

$u \leftarrow \text{Extraer}(\text{Open})$

 Inserta u a *Closed*

 for each $v \in \text{Succ}(u) \setminus (\text{Open} \cup \text{Closed})$

$\text{parent}(v) = u$

 if $v \in G$ **return** v

 Inserta v a *Open*

La diferencia entre DFS y BFS



El siguiente es un algoritmo de búsqueda genérico.

Input: Un problema de búsqueda (S, A, s_{init}, G)

Output: Un nodo objetivo

Open es un conjunto vacío

Closed es un conjunto vacío

Insertar s_{init} a *Open*

$\text{parent}(s_{init}) = \text{null}$

While *Open* $\neq \emptyset$:

$u \leftarrow \text{Extraer}(\text{Open})$

 Inserta u a *Closed*

 for each $v \in \text{Succ}(u) \setminus (\text{Open} \cup \text{Closed})$

$\text{parent}(v) = u$

 if $v \in G$ **return** v

 Insertar v a *Open*

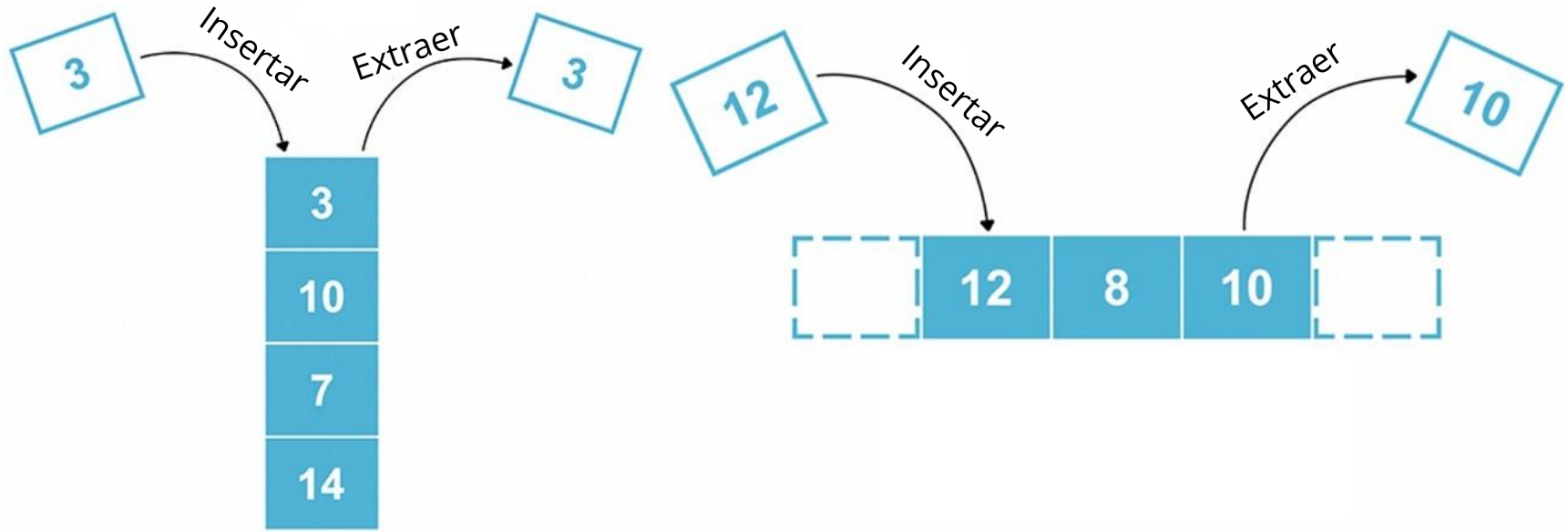
La diferencia en la implementación de DFS y BFS es la estructura de datos que se utiliza para el *Open*



Quiz en Menti! Pregunta 1

¿Cuál algoritmo crees que usa un Stack y cuál una Queue?

La diferencia entre DFS y BFS



Stack

Estructura del
Open en **DFS**

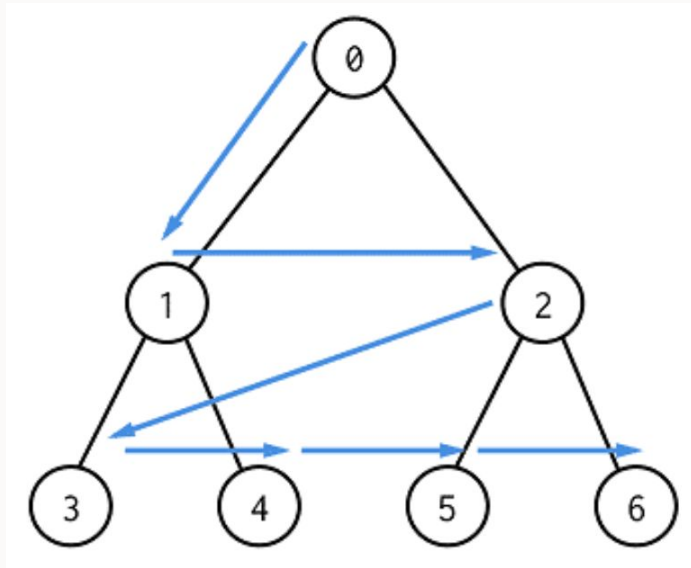
Queue

Estructura del
Open en **BFS**



Breadth First Search - Búsqueda en anchura/amplitud

BFS





BFS

* Recuerda: el *Open* se comporta como Queue

Open es un conjunto vacío

Closed es un conjunto vacío

Insert s_{init} a *Open*

$parent(s_{init}) = \text{null}$

While *Open* $\neq \emptyset$:

$u \leftarrow \text{Extraer}(\text{Open})$

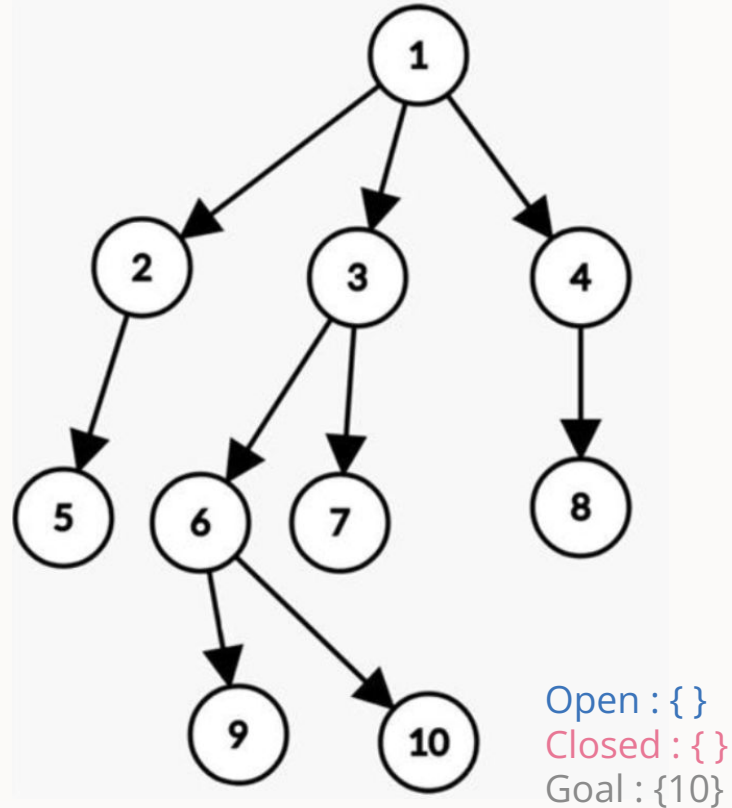
Inserta u a *Closed*

for each $v \in \text{Succ}(u) \setminus (\text{Open} \cup \text{Closed})$

$parent(v) = u$

if $v \in G$ **return** v

Inserta v a *Open*





Quiz en Menti! Pregunta 2

Al usar BFS, ¿Cómo lucen Open y Closed después de llegar al nodo Goal (10)?



BFS

* Recuerda: el *Open* se comporta como Queue

Open es un conjunto vacío

Closed es un conjunto vacío

Insert s_{init} a *Open* ←

$parent(s_{init}) = null$

While *Open* $\neq \emptyset$:

$u \leftarrow \text{Extraer}(\textit{Open})$

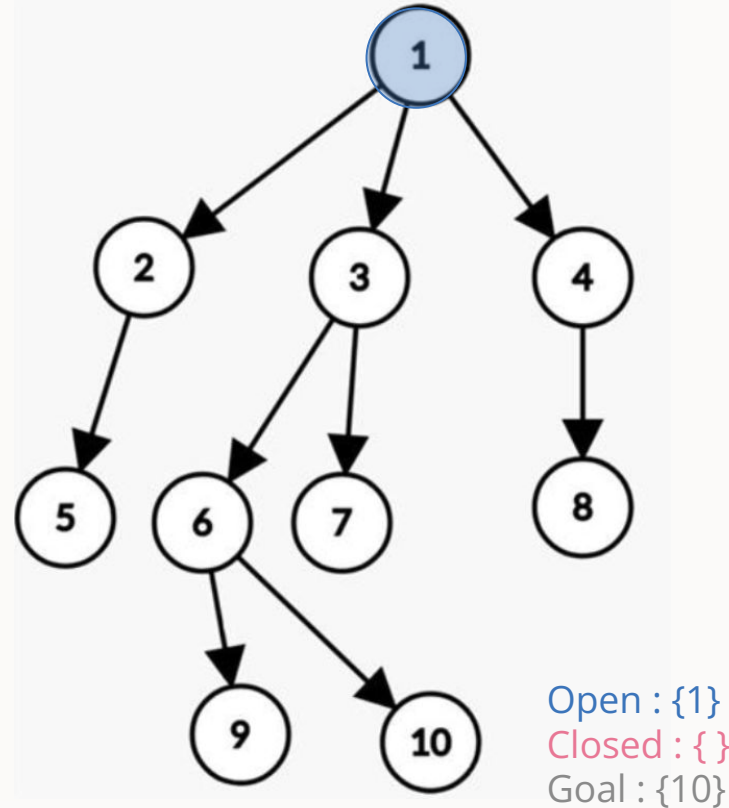
Inserta u a *Closed*

for each $v \in \text{Succ}(u) \setminus (\textit{Open} \cup \textit{Closed})$

$parent(v) = u$

if $v \in G$ **return** v

Inserta v a *Open*





BFS

* Recuerda: el *Open* se comporta como Queue

Open es un conjunto vacío

Closed es un conjunto vacío

Insert s_{init} a *Open*

$parent(s_{init}) = null$

While *Open* $\neq \emptyset$: ←

$u \leftarrow \text{Extraer}(\text{Open})$

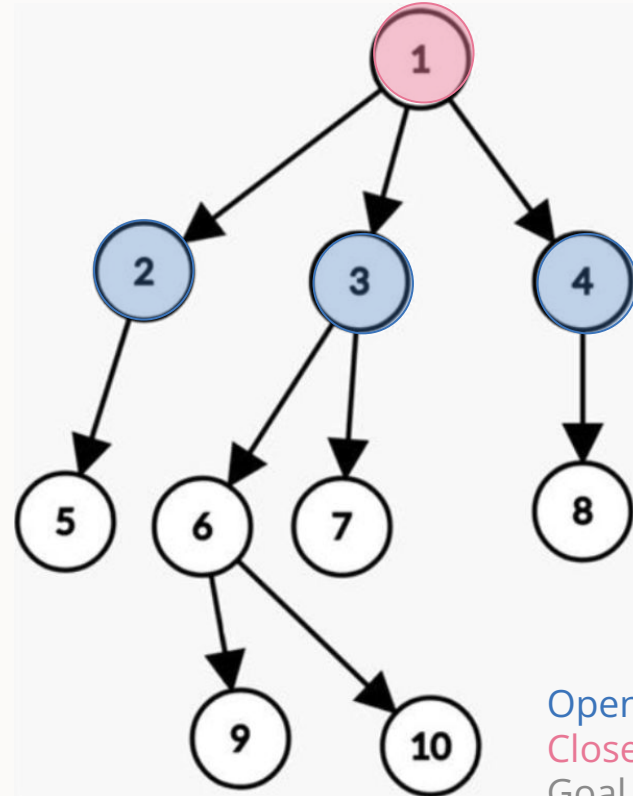
Inserta u a *Closed*

for each $v \in \text{Succ}(u) \setminus (\text{Open} \cup \text{Closed})$

$parent(v) = u$

if $v \in G$ **return** v

Inserta v a *Open*



Open : {2, 3, 4}

Closed : {1}

Goal : {10}



BFS

* Recuerda: el *Open* se comporta como Queue

Open es un conjunto vacío

Closed es un conjunto vacío

Insert s_{init} a *Open*

$parent(s_{init}) = null$

While *Open* $\neq \emptyset$: ←

$u \leftarrow \text{Extraer}(\text{Open})$

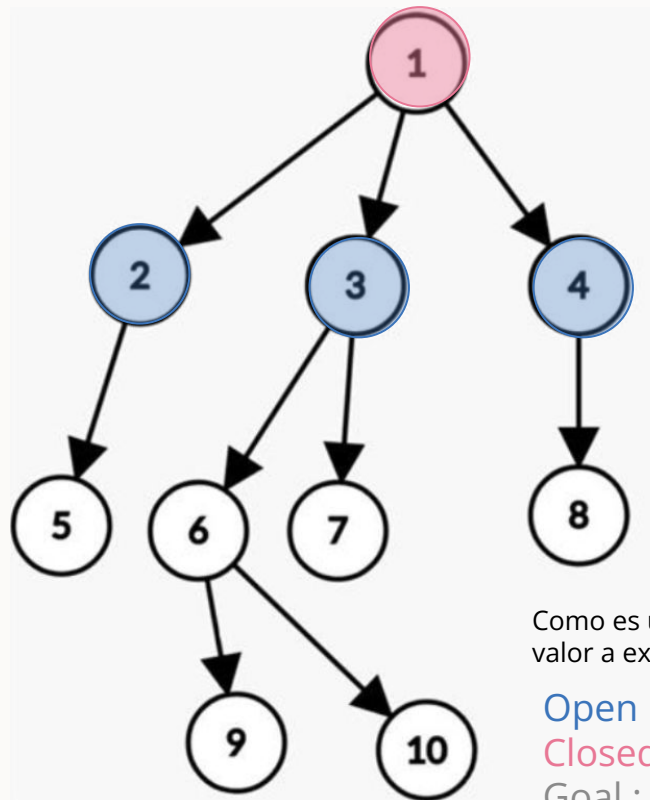
Inserta u a *Closed*

for each $v \in Succ(u) \setminus (\text{Open} \cup \text{Closed})$

$parent(v) = u$

if $v \in G$ **return** v

Inserta v a *Open*



Como es un Queue, el siguiente valor a extraer del *Open* es el 2

Open : {2, 3, 4}

Closed : {1}

Goal : {10}



BFS

* Recuerda: el *Open* se comporta como Queue

Open es un conjunto vacío

Closed es un conjunto vacío

Insert s_{init} a *Open*

$parent(s_{init}) = null$

While *Open* $\neq \emptyset$: ←

$u \leftarrow \text{Extraer}(\text{Open})$

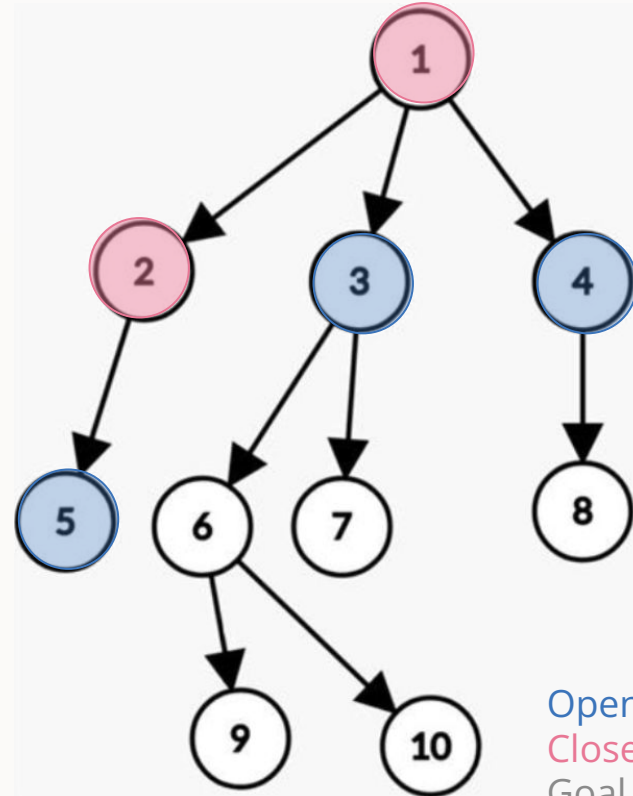
Inserta u a *Closed*

for each $v \in \text{Succ}(u) \setminus (\text{Open} \cup \text{Closed})$

$parent(v) = u$

if $v \in G$ **return** v

Inserta v a *Open*



Open : {3, 4, 5}

Closed : {1, 2}

Goal : {10}



BFS

* Recuerda: el *Open* se comporta como Queue

Open es un conjunto vacío

Closed es un conjunto vacío

Insert s_{init} a *Open*

$parent(s_{init}) = null$

While *Open* $\neq \emptyset$: ←

$u \leftarrow \text{Extraer}(\text{Open})$

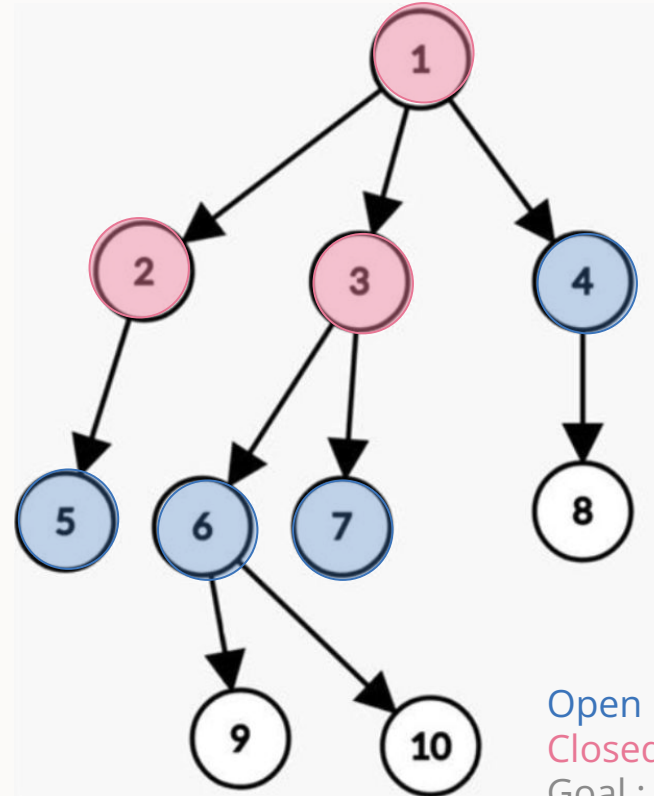
Inserta u a *Closed*

for each $v \in \text{Succ}(u) \setminus (\text{Open} \cup \text{Closed})$

$parent(v) = u$

if $v \in G$ **return** v

Inserta v a *Open*



Open : {4, 5, 6, 7}

Closed : {1, 2, 3}

Goal : {10}



BFS

* Recuerda: el *Open* se comporta como Queue

Open es un conjunto vacío

Closed es un conjunto vacío

Insert s_{init} a *Open*

$parent(s_{init}) = null$

While *Open* $\neq \emptyset$: 

$u \leftarrow \text{Extraer}(\textit{Open})$

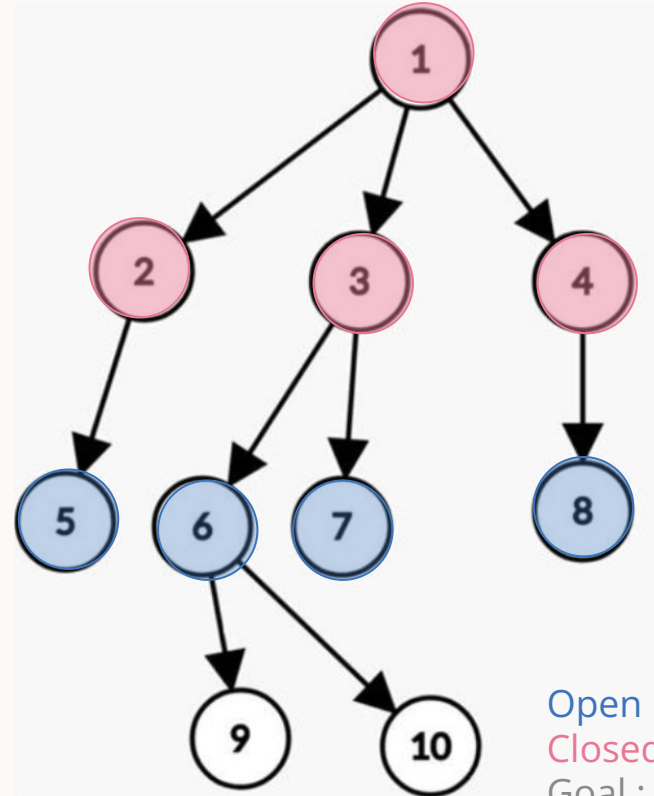
Inserta u a *Closed*

for each $v \in \textit{Succ}(u) \setminus (\textit{Open} \cup \textit{Closed})$

$parent(v) = u$

if $v \in G$ **return** v

Inserta v a *Open*



Open : {5, 6, 7, 8}

Closed : {1, 2, 3, 4}

Goal : {10}



BFS

* Recuerda: el *Open* se comporta como Queue

Open es un conjunto vacío

Closed es un conjunto vacío

Insert s_{init} a *Open*

$parent(s_{init}) = null$

While *Open* $\neq \emptyset$: 

$u \leftarrow \text{Extraer}(\textit{Open})$

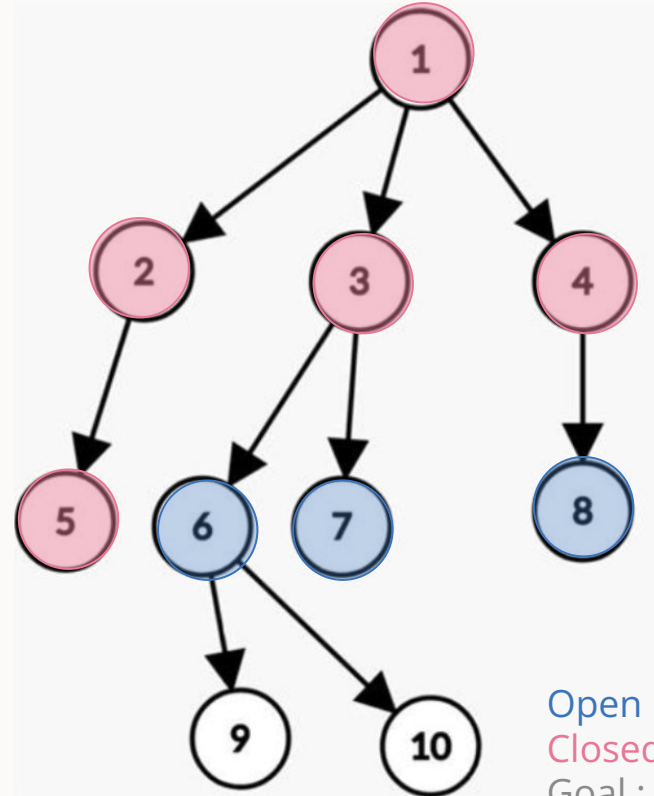
Inserta u a *Closed*

for each $v \in \textit{Succ}(u) \setminus (\textit{Open} \cup \textit{Closed})$

$parent(v) = u$

if $v \in G$ **return** v

Inserta v a *Open*



Open : {6, 7, 8}

Closed : {1, 2, 3, 4, 5}

Goal : {10}



BFS

* Recuerda: el *Open* se comporta como Queue

Open es un conjunto vacío

Closed es un conjunto vacío

Insert s_{init} a *Open*

$parent(s_{init}) = null$

While *Open* $\neq \emptyset$:

$u \leftarrow \text{Extraer}(\text{Open})$

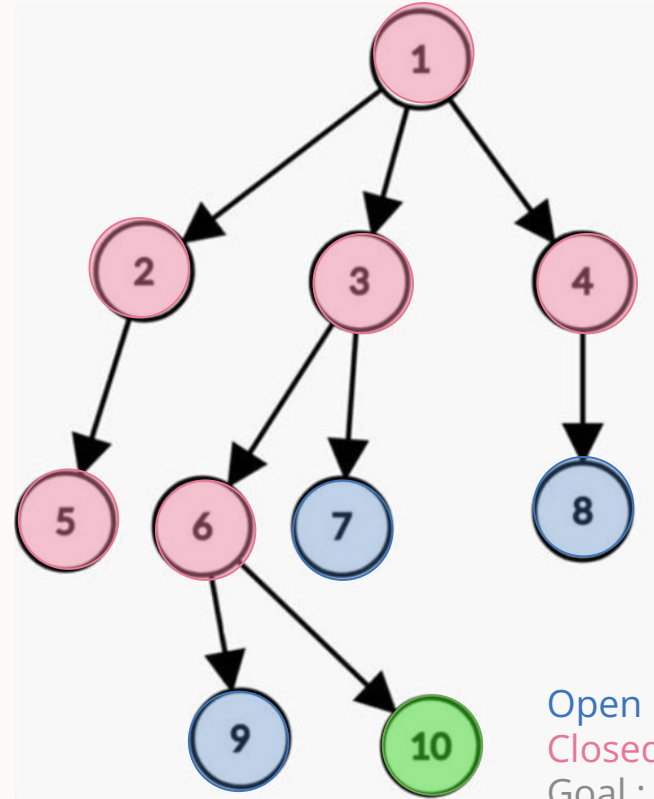
Inserta u a *Closed*

for each $v \in \text{Succ}(u) \setminus (\text{Open} \cup \text{Closed})$

$parent(v) = u$

if $v \in G$ **return** v ←

Inserta v a *Open*



Open : {7, 8, 9}

Closed : {1, 2, 3, 4, 5, 6}

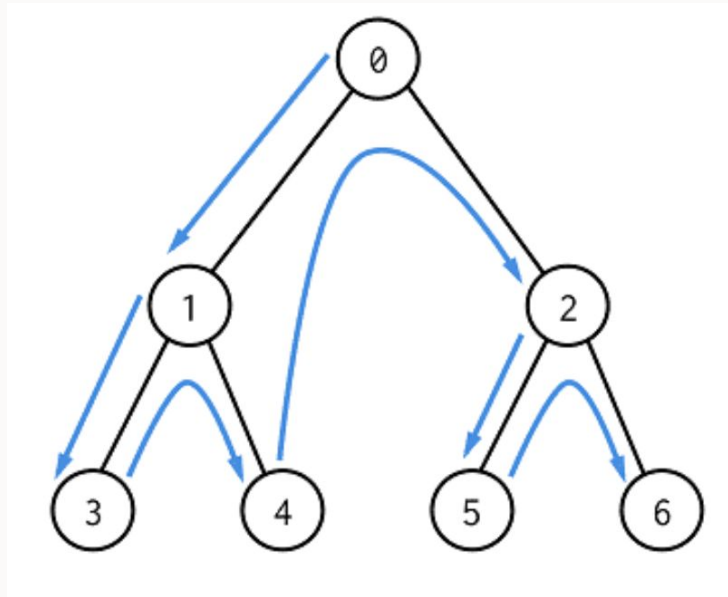
Goal : {10}

Se retorna el nodo 10!



Depth First Search - Búsqueda en profundidad

DFS





DFS

* Recuerda: el *Open* se comporta como Stack

Open es un conjunto vacío

Closed es un conjunto vacío

Insert s_{init} a *Open*

$parent(s_{init}) = \text{null}$

While *Open* $\neq \emptyset$:

$u \leftarrow \text{Extraer}(\text{Open})$

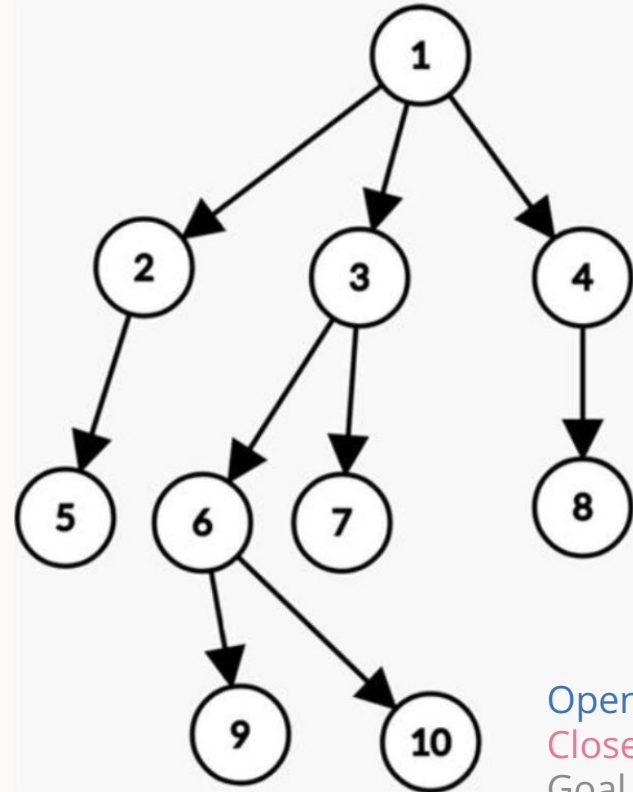
Inserta u a *Closed*

for each $v \in \text{Succ}(u) \setminus (\text{Open} \cup \text{Closed})$

$parent(v) = u$

if $v \in G$ **return** v

Inserta v a *Open*



Open : { }

Closed : { }

Goal : {10}



Quiz en Menti! **Pregunta 3**

Al usar DFS, ¿Cómo lucen Open y Closed después de llegar al nodo Goal (10)?



DFS

* Recuerda: el *Open* se comporta como Stack

Open es un conjunto vacío

Closed es un conjunto vacío

Insert s_{init} a *Open* ←

$\text{parent}(s_{init}) = \text{null}$

While *Open* $\neq \emptyset$:

$u \leftarrow \text{Extraer}(\text{Open})$

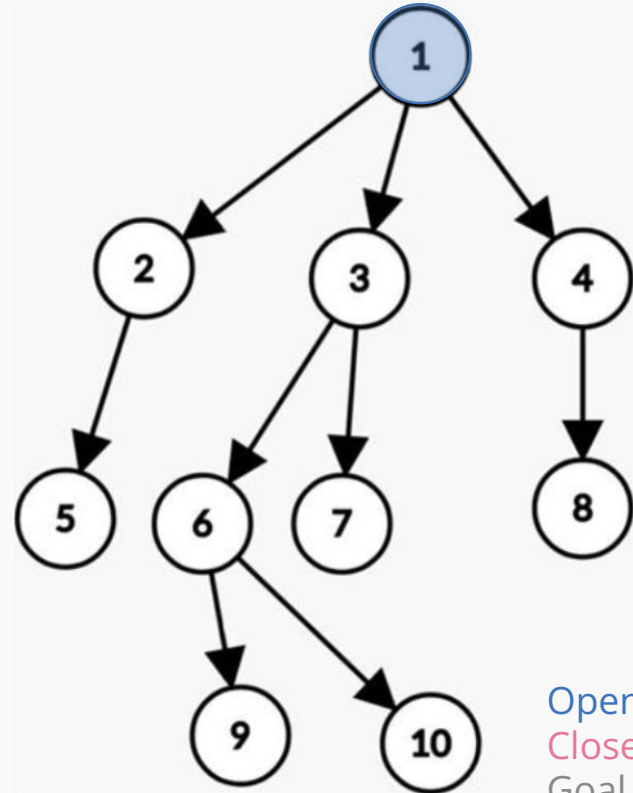
Inserta u a *Closed*

for each $v \in \text{Succ}(u) \setminus (\text{Open} \cup \text{Closed})$

$\text{parent}(v) = u$

if $v \in G$ **return** v

Inserta v a *Open*



Open : {1}

Closed : { }

Goal : {10}



DFS


* Recuerda: el *Open* se comporta como Stack

Open es un conjunto vacío

Closed es un conjunto vacío

Insert s_{init} a *Open*

$parent(s_{init}) = \text{null}$

While *Open* $\neq \emptyset$: 

$u \leftarrow \text{Extraer}(\textit{Open})$

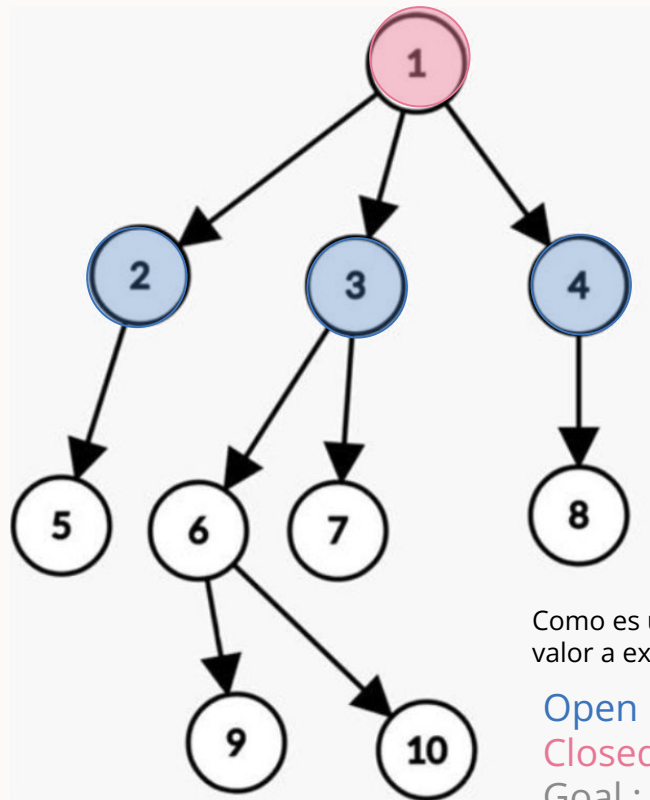
Inserta u a *Closed*

for each $v \in \textit{Succ}(u) \setminus (\textit{Open} \cup \textit{Closed})$

$parent(v) = u$

if $v \in G$ **return** v

Inserta v a *Open*



Como es un Stack, el siguiente valor a extraer del *Open* es el 4

Open : {2, 3, 4}

Closed : {1}

Goal : {10}



DFS

* Recuerda: el *Open* se comporta como Stack

Open es un conjunto vacío

Closed es un conjunto vacío

Insert s_{init} a *Open*

$parent(s_{init}) = null$

While *Open* $\neq \emptyset$: 

$u \leftarrow \text{Extraer}(\textit{Open})$

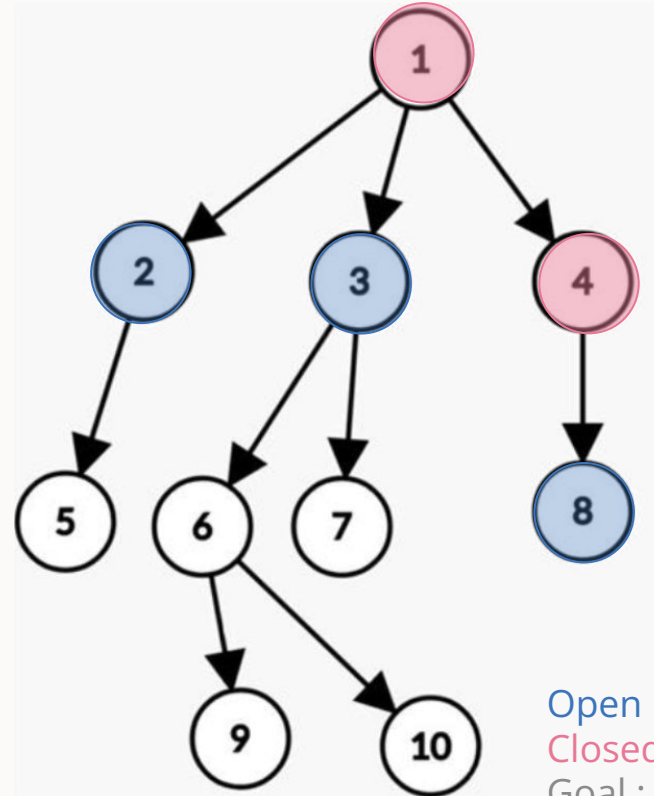
Inserta u a *Closed*

for each $v \in \textit{Succ}(u) \setminus (\textit{Open} \cup \textit{Closed})$

$parent(v) = u$

if $v \in G$ **return** v

Inserta v a *Open*



Open : {2, 3, 8}

Closed : {1, 4}

Goal : {10}



DFS

* Recuerda: el *Open* se comporta como Stack

Open es un conjunto vacío

Closed es un conjunto vacío

Insert s_{init} a *Open*

$parent(s_{init}) = null$

While *Open* $\neq \emptyset$: 

$u \leftarrow \text{Extraer}(\text{Open})$

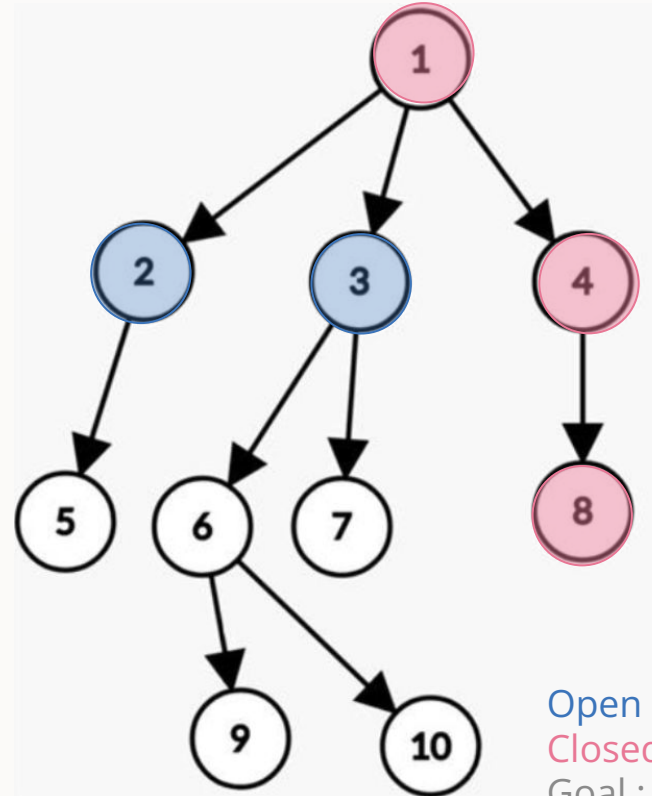
Inserta u a *Closed*

for each $v \in \text{Succ}(u) \setminus (\text{Open} \cup \text{Closed})$

$parent(v) = u$

if $v \in G$ **return** v

Inserta v a *Open*



Open : {2, 3}

Closed : {1, 4, 8}

Goal : {10}



DFS

* Recuerda: el *Open* se comporta como Stack

Open es un conjunto vacío

Closed es un conjunto vacío

Insert s_{init} a *Open*

$parent(s_{init}) = null$

While *Open* $\neq \emptyset$: 

$u \leftarrow \text{Extraer}(\textit{Open})$

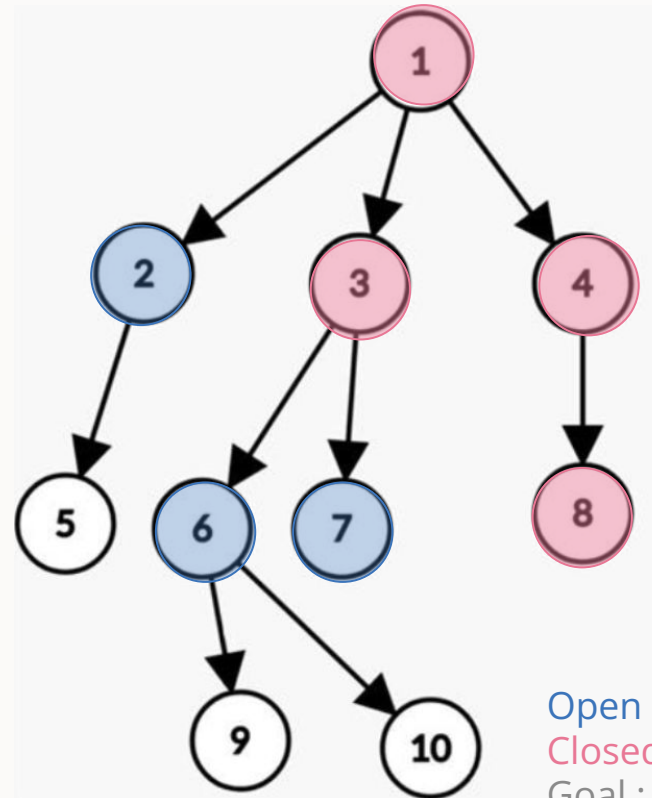
Inserta u a *Closed*

for each $v \in \textit{Succ}(u) \setminus (\textit{Open} \cup \textit{Closed})$

$parent(v) = u$

if $v \in G$ **return** v

Inserta v a *Open*



Open : {2, 6, 7}

Closed : {1, 4, 8, 3}

Goal : {10}



DFS

* Recuerda: el *Open* se comporta como Stack

Open es un conjunto vacío

Closed es un conjunto vacío

Insert s_{init} a *Open*

$parent(s_{init}) = null$

While *Open* $\neq \emptyset$: 

$u \leftarrow \text{Extraer}(\textit{Open})$

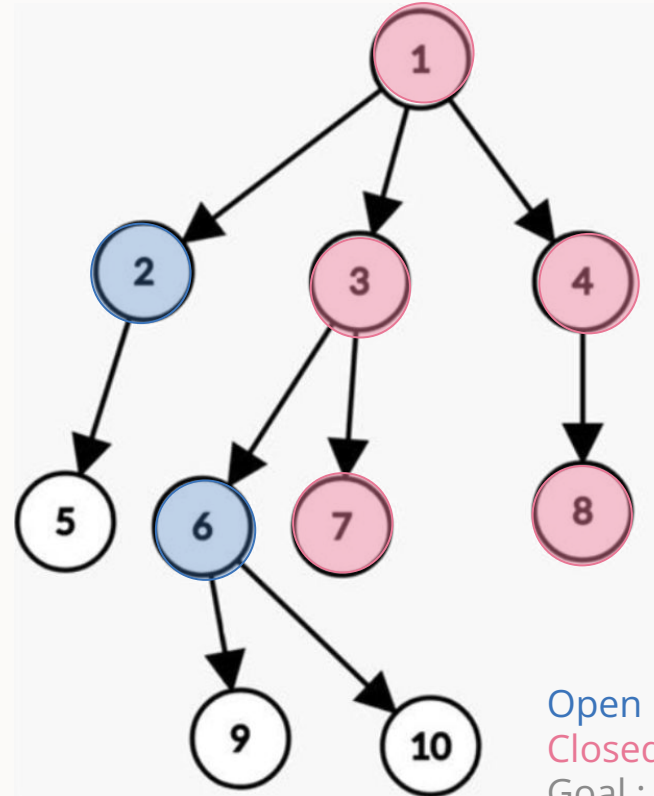
Inserta u a *Closed*

for each $v \in \textit{Succ}(u) \setminus (\textit{Open} \cup \textit{Closed})$

$parent(v) = u$

if $v \in G$ **return** v

Inserta v a *Open*



Open : {2, 6}

Closed : {1, 4, 8, 3, 7}

Goal : {10}



DFS

* Recuerda: el *Open* se comporta como Stack

Open es un conjunto vacío

Closed es un conjunto vacío

Insert s_{init} a *Open*

$parent(s_{init}) = null$

While *Open* $\neq \emptyset$:

$u \leftarrow \text{Extraer}(\text{Open})$

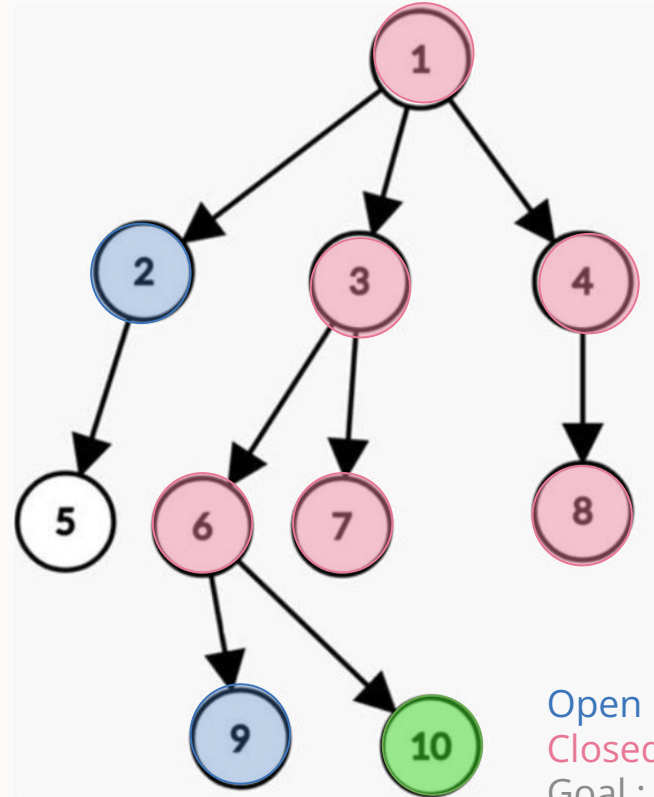
Inserta u a *Closed*

for each $v \in \text{Succ}(u) \setminus (\text{Open} \cup \text{Closed})$

$parent(v) = u$

if $v \in G$ **return** v ←

Inserta v a *Open*



Open : {2, 9}

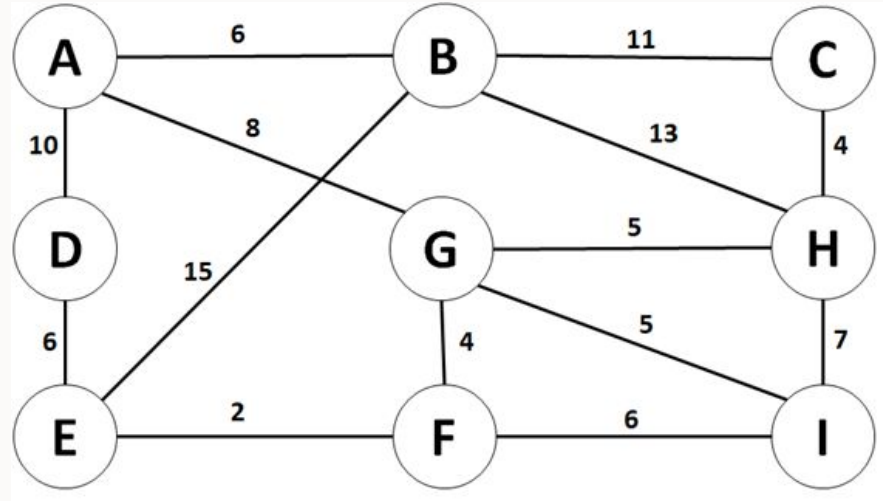
Closed : {1, 4, 8, 3, 7, 6}

Goal : {10}

Se retorna el nodo 10!



Algoritmo de Dijkstra





Algoritmo de Dijkstra

- Es un algoritmo que encuentra **caminos más cortos en un grafo ponderado** (grafos con pesos en las aristas) → encuentra soluciones **óptimas**.
- Comienza por el nodo fuente y explora gradualmente todos los nodos adyacentes, **actualizando las distancias mínimas** a medida que se encuentran caminos más cortos.
- Mantiene una **lista de nodos por visitar** y **selecciona el nodo no visitado con la distancia mínima** para explorar en cada iteración.
- **Usos:** Se puede utilizar en sistemas de transporte para determinar la **ruta más corta entre dos puntos**.



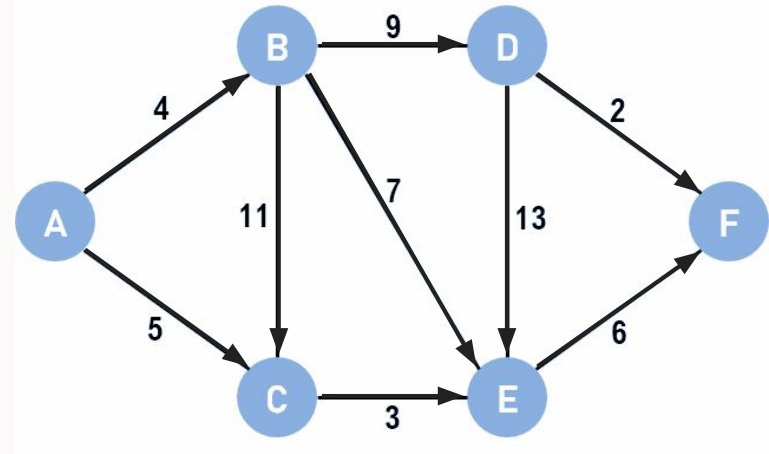
Algoritmo de Dijkstra

```
// Pseudocódigo
for each nodo n ∈ grafo G:
    dist[n] ← inf
    prev[n] ← None
    Insert n a cola Q

dist[origen] ← 0
while Q ≠ ∅:
    u ← nodo en Q con menor dist[u]
    Extraer u de Q

    for each v ∈ Succ(u) ∩ Q:
        new_dist ← dist[u] + arista(u, v)
        If new_dist < dist[v]:
            dist[v] ← new_dist
            prev[v] ← u

return dist[], prev[]
```





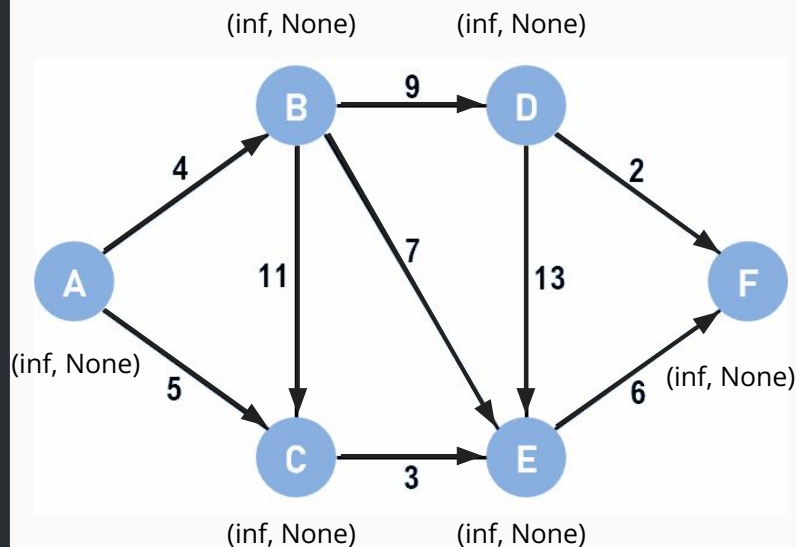
Algoritmo de Dijkstra

```
// Pseudocódigo
for each nodo n ∈ grafo G: ←
    dist[n] ← inf
    prev[n] ← None
    Insert n a cola Q

dist[origen] ← 0
while Q ≠ ∅:
    u ← nodo en Q con menor dist[u]
    Extraer u de Q

    for each v ∈ Succ(u) ∩ Q:
        new_dist ← dist[u] + arista(u, v)
        If new_dist < dist[v]:
            dist[v] ← new_dist
            prev[v] ← u

return dist[], prev[]
```



Q: {A, B, C, D, E, F}



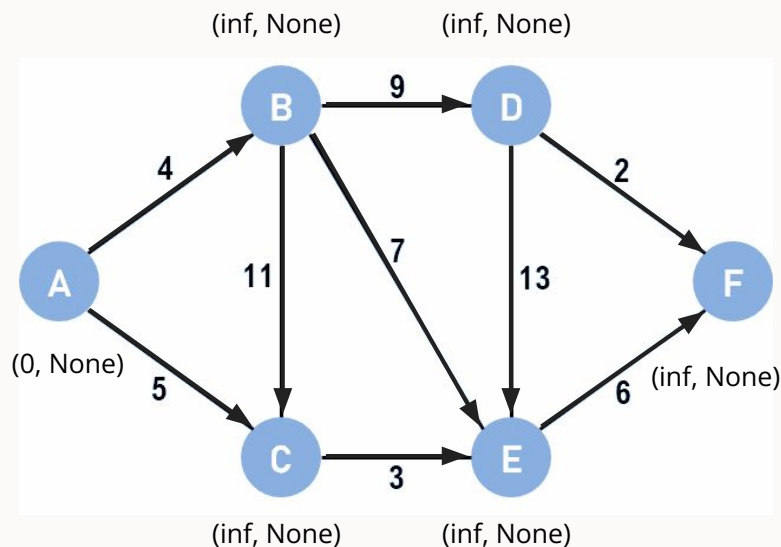
Algoritmo de Dijkstra

```
// Pseudocódigo
for each nodo n ∈ grafo G:
    dist[n] ← inf
    prev[n] ← None
    Insert n a cola Q

dist[origen] ← 0 ←
while Q ≠ ∅:
    u ← nodo en Q con menor dist[u]
    Extraer u de Q

    for each v ∈ Succ(u) ∩ Q:
        new_dist ← dist[u] + arista(u, v)
        If new_dist < dist[v]:
            dist[v] ← new_dist
            prev[v] ← u

return dist[], prev[]
```



Q: {A, B, C, D, E, F}



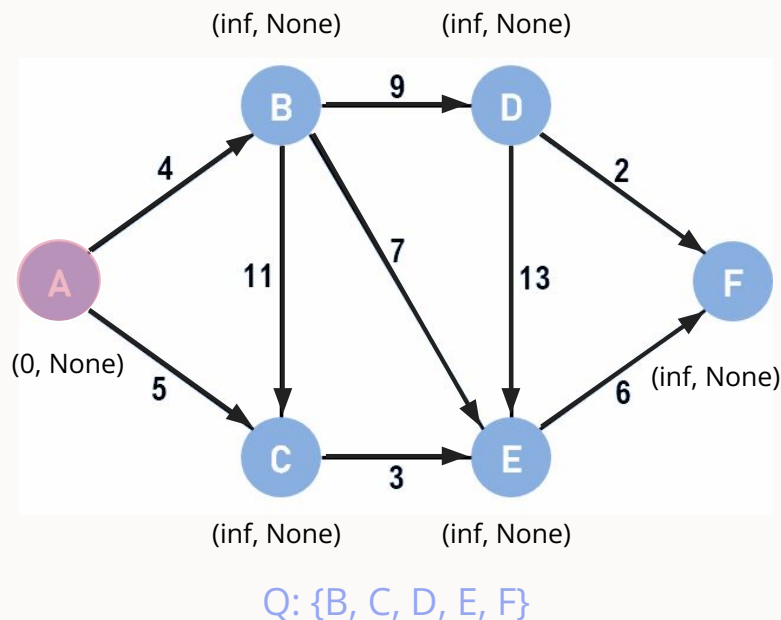
Algoritmo de Dijkstra

```
// Pseudocódigo
for each nodo n ∈ grafo G:
    dist[n] ← inf
    prev[n] ← None
    Insert n a cola Q

dist[origen] ← 0
while Q ≠ ∅:
    u ← nodo en Q con menor dist[u]
    Extraer u de Q

    for each v ∈ Succ(u) ∩ Q:
        new_dist ← dist[u] + arista(u, v)
        If new_dist < dist[v]:
            dist[v] ← new_dist
            prev[v] ← u

return dist[], prev[]
```





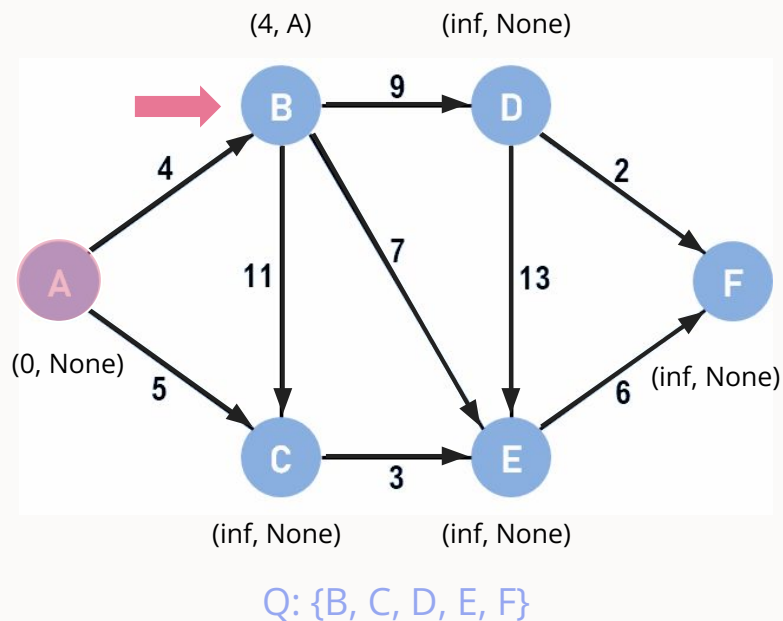
Algoritmo de Dijkstra

```
// Pseudocódigo
for each nodo n ∈ grafo G:
    dist[n] ← inf
    prev[n] ← None
    Insert n a cola Q

dist[origen] ← 0
while Q ≠ ∅:
    u ← nodo en Q con menor dist[u]
    Extraer u de Q

    for each v ∈ Succ(u) ∩ Q: ←
        new_dist ← dist[u] + arista(u, v)
        if new_dist < dist[v]:
            dist[v] ← new_dist
            prev[v] ← u

return dist[], prev[]
```





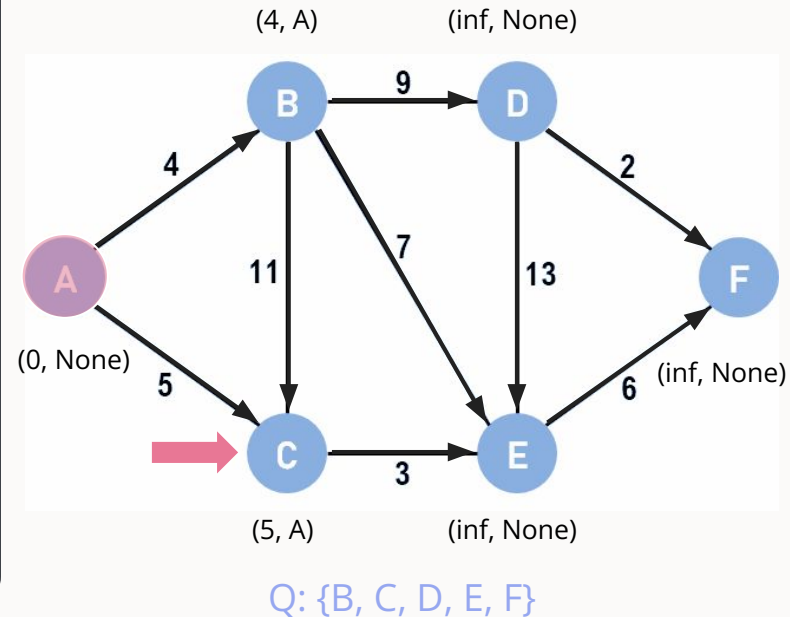
Algoritmo de Dijkstra

```
// Pseudocódigo
for each nodo n ∈ grafo G:
    dist[n] ← inf
    prev[n] ← None
    Insert n a cola Q

dist[origen] ← 0
while Q ≠ ∅:
    u ← nodo en Q con menor dist[u]
    Extraer u de Q

    for each v ∈ Succ(u) ∩ Q: ←
        new_dist ← dist[u] + arista(u, v)
        if new_dist < dist[v]:
            dist[v] ← new_dist
            prev[v] ← u

return dist[], prev[]
```





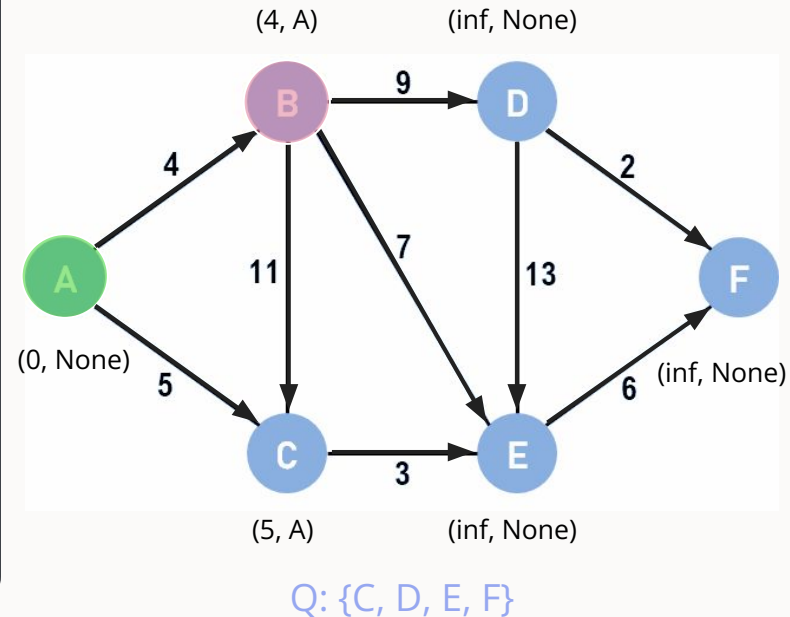
Algoritmo de Dijkstra

```
// Pseudocódigo
for each nodo n ∈ grafo G:
    dist[n] ← inf
    prev[n] ← None
    Insert n a cola Q

dist[origen] ← 0
while Q ≠ ∅:
    u ← nodo en Q con menor dist[u]
    Extraer u de Q

    for each v ∈ Succ(u) ∩ Q:
        new_dist ← dist[u] + arista(u, v)
        if new_dist < dist[v]:
            dist[v] ← new_dist
            prev[v] ← u

return dist[], prev[]
```





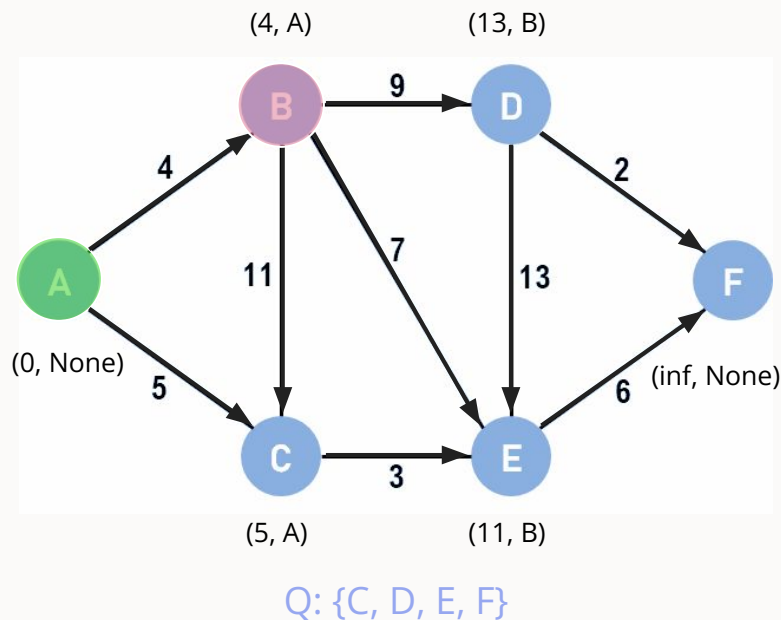
Algoritmo de Dijkstra

```
// Pseudocódigo
for each nodo n ∈ grafo G:
    dist[n] ← inf
    prev[n] ← None
    Insert n a cola Q

dist[origen] ← 0
while Q ≠ ∅:
    u ← nodo en Q con menor dist[u]
    Extraer u de Q

    for each v ∈ Succ(u) ∩ Q: ←
        new_dist ← dist[u] + arista(u, v)
        if new_dist < dist[v]:
            dist[v] ← new_dist
            prev[v] ← u

return dist[], prev[]
```





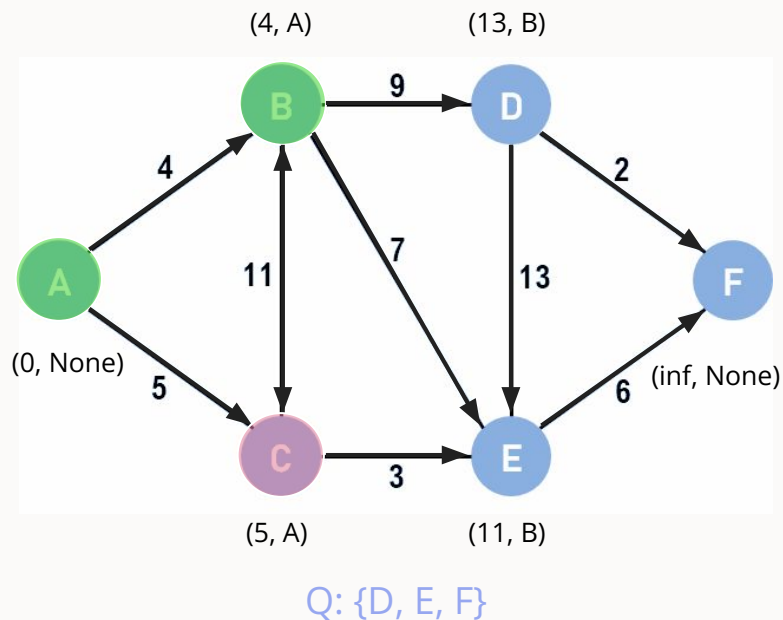
Algoritmo de Dijkstra

```
// Pseudocódigo
for each nodo n ∈ grafo G:
    dist[n] ← inf
    prev[n] ← None
    Insert n a cola Q

dist[origen] ← 0
while Q ≠ ∅:
    u ← nodo en Q con menor dist[u]
    Extraer u de Q

    for each v ∈ Succ(u) ∩ Q:
        new_dist ← dist[u] + arista(u, v)
        if new_dist < dist[v]:
            dist[v] ← new_dist
            prev[v] ← u

return dist[], prev[]
```





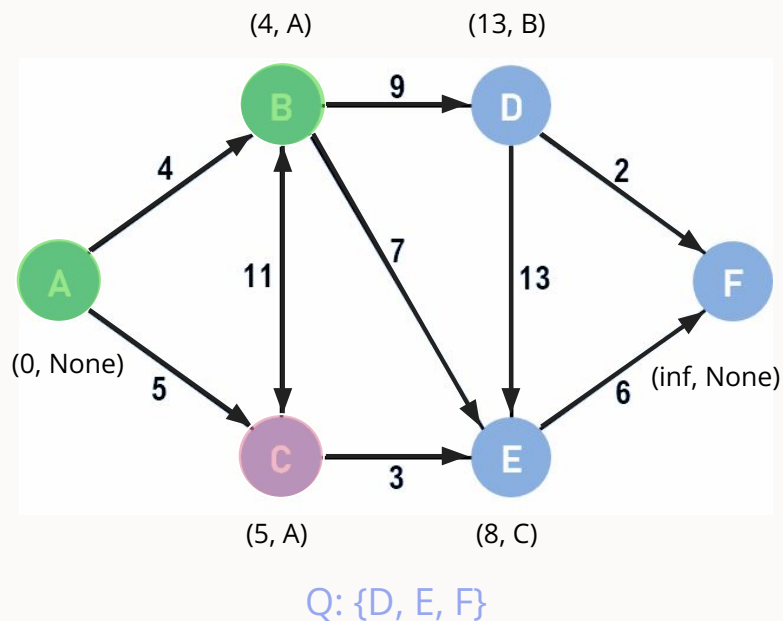
Algoritmo de Dijkstra

```
// Pseudocódigo
for each nodo n ∈ grafo G:
    dist[n] ← inf
    prev[n] ← None
    Insert n a cola Q

dist[origen] ← 0
while Q ≠ ∅:
    u ← nodo en Q con menor dist[u]
    Extraer u de Q

    for each v ∈ Succ(u) ∩ Q: ←
        new_dist ← dist[u] + arista(u, v)
        if new_dist < dist[v]:
            dist[v] ← new_dist
            prev[v] ← u

return dist[], prev[]
```





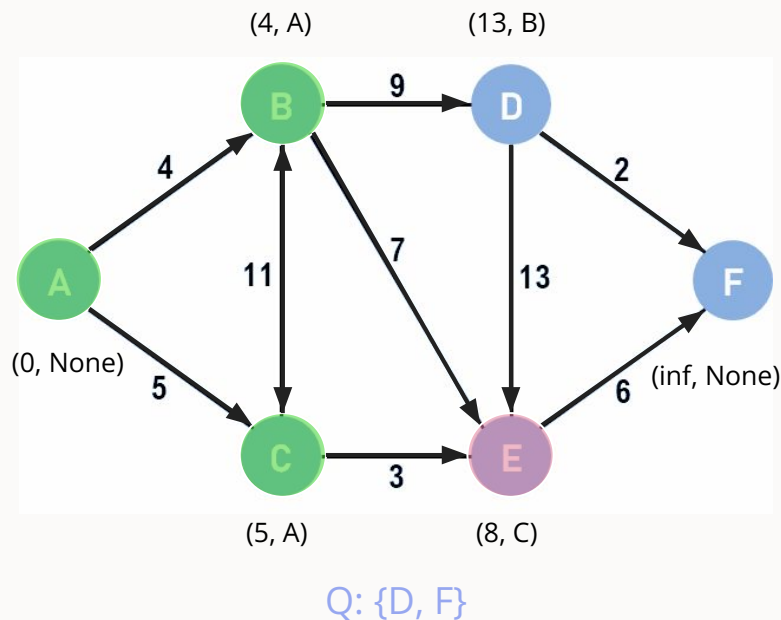
Algoritmo de Dijkstra

```
// Pseudocódigo
for each nodo n ∈ grafo G:
    dist[n] ← inf
    prev[n] ← None
    Insert n a cola Q

dist[origen] ← 0
while Q ≠ ∅:
    u ← nodo en Q con menor dist[u]
    Extraer u de Q

    for each v ∈ Succ(u) ∩ Q:
        new_dist ← dist[u] + arista(u, v)
        if new_dist < dist[v]:
            dist[v] ← new_dist
            prev[v] ← u

return dist[], prev[]
```





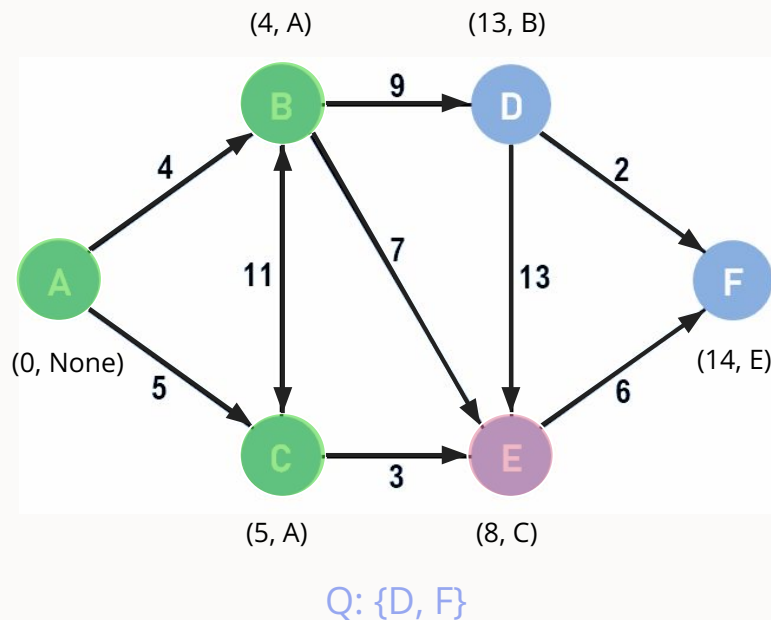
Algoritmo de Dijkstra

```
// Pseudocódigo
for each nodo n ∈ grafo G:
    dist[n] ← inf
    prev[n] ← None
    Insert n a cola Q

dist[origen] ← 0
while Q ≠ ∅:
    u ← nodo en Q con menor dist[u]
    Extraer u de Q

    for each v ∈ Succ(u) ∩ Q: ←
        new_dist ← dist[u] + arista(u, v)
        if new_dist < dist[v]:
            dist[v] ← new_dist
            prev[v] ← u

return dist[], prev[]
```





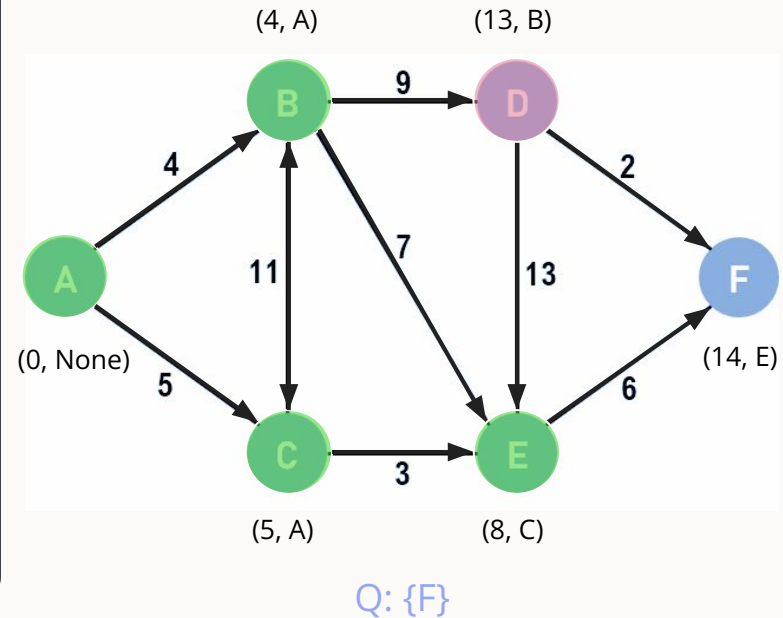
Algoritmo de Dijkstra

```
// Pseudocódigo
for each nodo n ∈ grafo G:
    dist[n] ← inf
    prev[n] ← None
    Insert n a cola Q

dist[origen] ← 0
while Q ≠ ∅:
    u ← nodo en Q con menor dist[u]
    Extraer u de Q

    for each v ∈ Succ(u) ∩ Q:
        new_dist ← dist[u] + arista(u, v)
        if new_dist < dist[v]:
            dist[v] ← new_dist
            prev[v] ← u

return dist[], prev[]
```





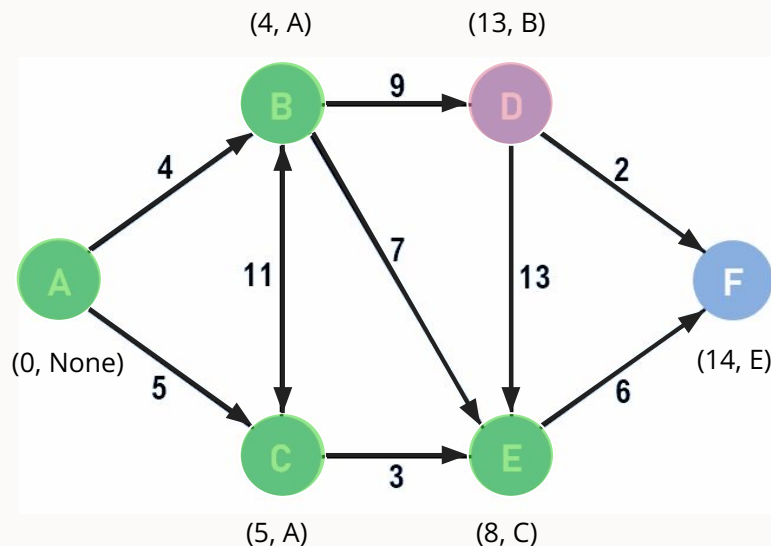
Algoritmo de Dijkstra

```
// Pseudocódigo
for each nodo n ∈ grafo G:
    dist[n] ← inf
    prev[n] ← None
    Insert n a cola Q

dist[origen] ← 0
while Q ≠ ∅:
    u ← nodo en Q con menor dist[u]
    Extraer u de Q

    for each v ∈ Succ(u) ∩ Q: ←
        new_dist ← dist[u] + arista(u, v)
        if new_dist < dist[v]:
            dist[v] ← new_dist
            prev[v] ← u

return dist[], prev[]
```



Q: {F}



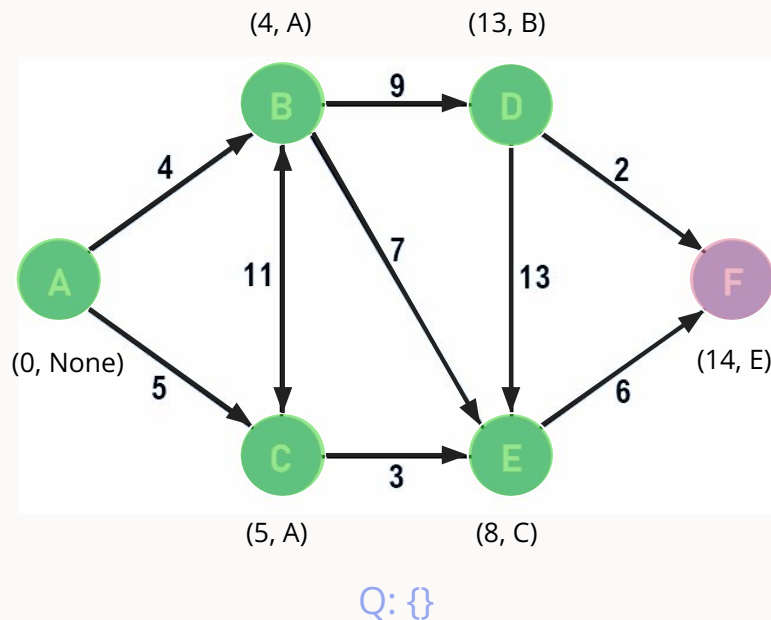
Algoritmo de Dijkstra

```
// Pseudocódigo
for each nodo n ∈ grafo G:
    dist[n] ← inf
    prev[n] ← None
    Insert n a cola Q

dist[origen] ← 0
while Q ≠ ∅:
    u ← nodo en Q con menor dist[u]
    Extraer u de Q

    for each v ∈ Succ(u) ∩ Q:
        new_dist ← dist[u] + arista(u, v)
        if new_dist < dist[v]:
            dist[v] ← new_dist
            prev[v] ← u

return dist[], prev[]
```





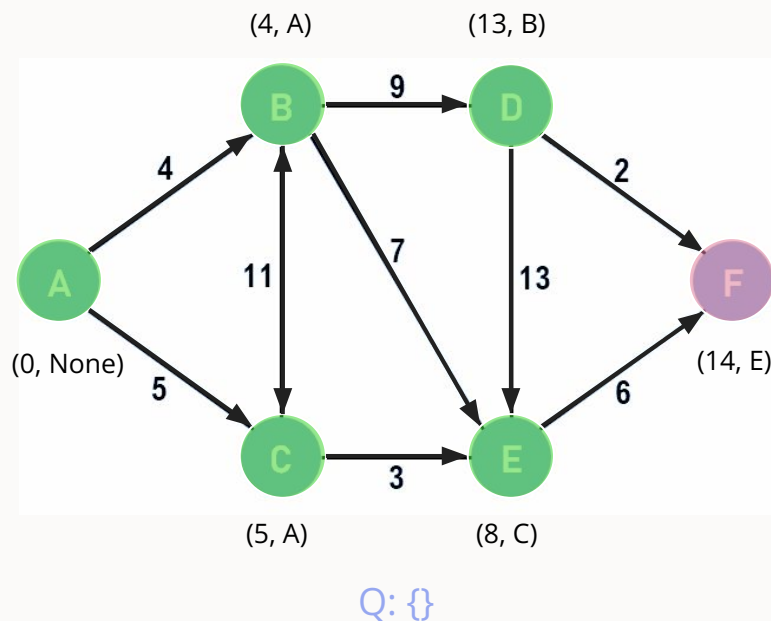
Algoritmo de Dijkstra

```
// Pseudocódigo
for each nodo n ∈ grafo G:
    dist[n] ← inf
    prev[n] ← None
    Insert n a cola Q

dist[origen] ← 0
while Q ≠ ∅:
    u ← nodo en Q con menor dist[u]
    Extraer u de Q

    for each v ∈ Succ(u) ∩ Q:
        new_dist ← dist[u] + arista(u, v)
        if new_dist < dist[v]:
            dist[v] ← new_dist
            prev[v] ← u

return dist[], prev[]
```





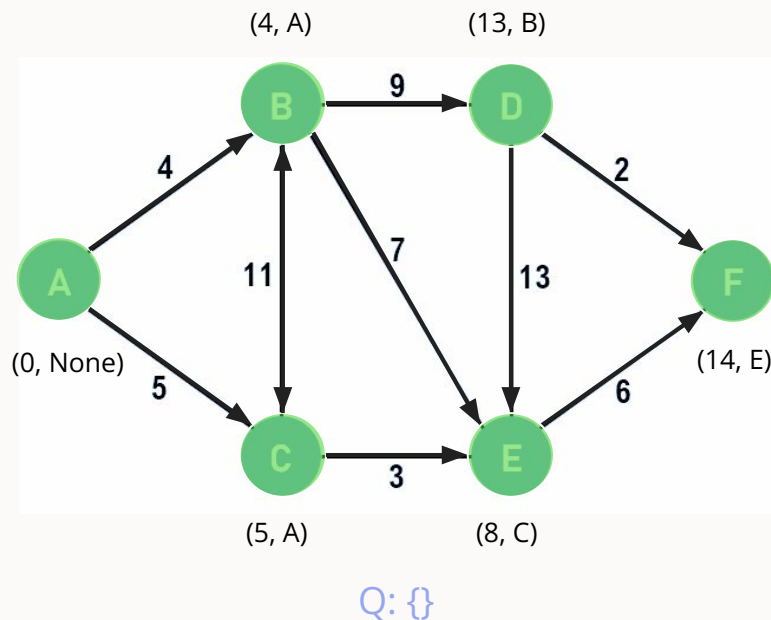
Algoritmo de Dijkstra

```
// Pseudocódigo
for each nodo n ∈ grafo G:
    dist[n] ← inf
    prev[n] ← None
    Insert n a cola Q

dist[origen] ← 0
while Q ≠ ∅:
    u ← nodo en Q con menor dist[u]
    Extraer u de Q

    for each v ∈ Succ(u) ∩ Q:
        new_dist ← dist[u] + arista(u, v)
        if new_dist < dist[v]:
            dist[v] ← new_dist
            prev[v] ← u

return dist[], prev[]
```





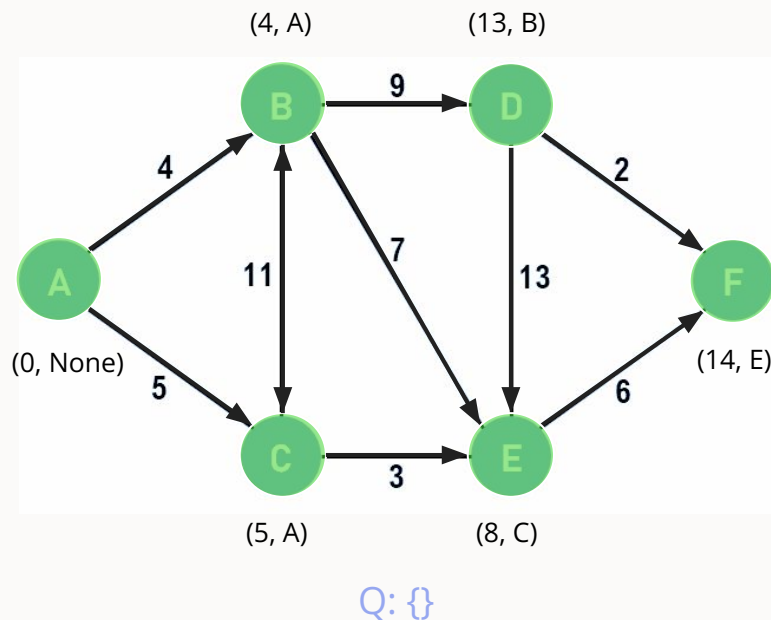
Algoritmo de Dijkstra

```
// Pseudocódigo
for each nodo n ∈ grafo G:
    dist[n] ← inf
    prev[n] ← None
    Insert n a cola Q

dist[origen] ← 0
while Q ≠ ∅:
    u ← nodo en Q con menor dist[u]
    Extraer u de Q

    for each v ∈ Succ(u) ∩ Q:
        new_dist ← dist[u] + arista(u, v)
        if new_dist < dist[v]:
            dist[v] ← new_dist
            prev[v] ← u

return dist[], prev[] ← ¡FIN!
```





Quiz en Menti! Pregunta 4

¿Qué es lo que retorna esta versión del algoritmo de Dijkstra?



Algoritmo de Dijkstra

- **Retorna las distancias al origen** de cada nodo, además del antecesor en la secuencia de cada uno de ellos.
- Se encuentra la **distancia mínima hacia cualquier nodo, desde el origen**.
- Para reconstruir la **ruta óptima**, basta con tomar un nodo destino, y **preguntar iterativamente** por el nodo anterior, hasta llegar al origen.



Ayudantía 4

Introducción a la búsqueda

Por Blanca Romero y Felipe Vidal

8 de septiembre 2023