

## Programación III

Ricardo Wehbe

UADE

24 de octubre de 2021

- 1 Repaso de la clase anterior
  - El problema de las  $n$  damas
  - Suma de subconjunto
- 2 Otros ejemplos de *backtracking*
  - Partición en partes iguales
  - Búsqueda en profundidad
- 3 Búsqueda en grafos
  - Búsqueda en amplitud (BFS)
  - DFS no recursivo
  - Uniform Cost Search (UCS)
  - El algoritmo A\*
- 4 Ejercicios propuestos

- 1 Repaso de la clase anterior
  - El problema de las  $n$  damas
  - Suma de subconjunto
- 2 Otros ejemplos de *backtracking*
  - Partición en partes iguales
  - Búsqueda en profundidad
- 3 Búsqueda en grafos
  - Búsqueda en amplitud (BFS)
  - DFS no recursivo
  - Uniform Cost Search (UCS)
  - El algoritmo  $A^*$
- 4 Ejercicios propuestos

# Introducción

# Introducción

- Hay problemas para los cuales no se conoce un algoritmo eficiente de resolución. En estos casos, la única solución posible es la exploración directa de todas las posibilidades (*brute force*).

# Introducción

- Hay problemas para los cuales no se conoce un algoritmo eficiente de resolución. En estos casos, la única solución posible es la exploración directa de todas las posibilidades (*brute force*).
- La técnica de *backtracking* es un método de búsqueda de soluciones exhaustivo sobre grafos acíclicos. Este método puede optimizarse con la “poda” de alternativas que no conduzcan a una solución.

# Introducción

- Hay problemas para los cuales no se conoce un algoritmo eficiente de resolución. En estos casos, la única solución posible es la exploración directa de todas las posibilidades (*brute force*).
- La técnica de *backtracking* es un método de búsqueda de soluciones exhaustivo sobre grafos acíclicos. Este método puede optimizarse con la “poda” de alternativas que no conduzcan a una solución.
- Los grafos representan las posibles alternativas por ser evaluadas.

# *Backtracking*. Esquema general



## *Backtracking*. Esquema general

- Se representan todas las posibilidades en un árbol (como ya sabemos, decir que un grafo conexo es acíclico equivale a decir que es un árbol).

## *Backtracking*. Esquema general

- Se representan todas las posibilidades en un árbol (como ya sabemos, decir que un grafo conexo es acíclico equivale a decir que es un árbol).
- Se busca la solución recorriendo el árbol (de una determinada forma según la estrategia).

## *Backtracking*. Esquema general

- Se representan todas las posibilidades en un árbol (como ya sabemos, decir que un grafo conexo es acíclico equivale a decir que es un árbol).
- Se busca la solución recorriendo el árbol (de una determinada forma según la estrategia).
- Hay partes del árbol (sub-árboles) que se evitan porque no pueden contener soluciones (poda).

## *Backtracking*. Esquema general

- Se representan todas las posibilidades en un árbol (como ya sabemos, decir que un grafo conexo es acíclico equivale a decir que es un árbol).
- Se busca la solución recorriendo el árbol (de una determinada forma según la estrategia).
- Hay partes del árbol (sub-árboles) que se evitan porque no pueden contener soluciones (poda).
- La solución del problema se presenta en una lista ordenada  $(x_1, x_2, \dots, x_n)$ .

## Backtracking. Esquema general

- Se representan todas las posibilidades en un árbol (como ya sabemos, decir que un grafo conexo es acíclico equivale a decir que es un árbol).
- Se busca la solución recorriendo el árbol (de una determinada forma según la estrategia).
- Hay partes del árbol (sub-árboles) que se evitan porque no pueden contener soluciones (poda).
- La solución del problema se presenta en una lista ordenada  $(x_1, x_2, \dots, x_n)$ .
- Cada elemento  $x_i$  de la lista se escoge entre un conjunto de candidatos. Cada lista representa un *estado*.

## Backtracking. Esquema general

- Se representan todas las posibilidades en un árbol (como ya sabemos, decir que un grafo conexo es acíclico equivale a decir que es un árbol).
- Se busca la solución recorriendo el árbol (de una determinada forma según la estrategia).
- Hay partes del árbol (sub-árboles) que se evitan porque no pueden contener soluciones (poda).
- La solución del problema se presenta en una lista ordenada  $(x_1, x_2, \dots, x_n)$ .
- Cada elemento  $x_i$  de la lista se escoge entre un conjunto de candidatos. Cada lista representa un *estado*.
- A veces se buscan todas las soluciones; a veces, nos conformamos con una (“estado solución”).

## Backtracking. Esquema general

- Se representan todas las posibilidades en un árbol (como ya sabemos, decir que un grafo conexo es acíclico equivale a decir que es un árbol).
- Se busca la solución recorriendo el árbol (de una determinada forma según la estrategia).
- Hay partes del árbol (sub-árboles) que se evitan porque no pueden contener soluciones (poda).
- La solución del problema se presenta en una lista ordenada  $(x_1, x_2, \dots, x_n)$ .
- Cada elemento  $x_i$  de la lista se escoge entre un conjunto de candidatos. Cada lista representa un *estado*.
- A veces se buscan todas las soluciones; a veces, nos conformamos con una (“estado solución”).
- A veces no existe ninguna solución.

# Elementos del algoritmo general de *backtracking*



## Elementos del algoritmo general de *backtracking*

- Para aplicar *backtracking* debemos proveer los datos  $S$  de cada instancia particular y cinco parámetros: *root*, *reject*, *accept*, *first* y *next*.

## Elementos del algoritmo general de *backtracking*

- Para aplicar *backtracking* debemos proveer los datos  $S$  de cada instancia particular y cinco parámetros: *root*, *reject*, *accept*, *first* y *next*.
- *root*( $T$ ) devuelve el candidato parcial en la raíz del árbol de búsqueda  $T$ .
- *reject*( $T, c$ ) devuelve *true* si no vale la pena completar el candidato  $c$ .
- *accept*( $T, c$ ) devuelve *true* si el candidato  $c$  es una solución de  $T$  y *false* si no.
- *first*( $T, c$ ) genera la primera extensión del candidato  $c$ .
- *next*( $T, c$ ) genera la siguiente extensión luego de  $c$ .

# El algoritmo general de *backtracking*

## El algoritmo general de *backtracking*

Se comienza con la llamada *backtracking*( $T$ ,  $root(T)$ ). El pseudo-código es el

siguiente:

## El algoritmo general de *backtracking*

Se comienza con la llamada *backtracking*( $T$ ,  $root(T)$ ). El pseudo-código es el

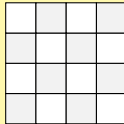
siguiente:

```
1  algoritmo backtracking( $T$ ,  $c$ )
2  if (reject( $T$ ,  $c$ )) {
3      return
4  }
5  if (accept( $T$ ,  $c$ )) {
6      return ( $T$ ,  $c$ )
7  }
8   $s = first(T, c)$ 
9  while ( $s \neq NULL$ ) {
10     backtracking( $T$ ,  $s$ )
11      $s = next(T, c)$ 
12 }
```

- 1 Repaso de la clase anterior
  - El problema de las  $n$  damas
  - Suma de subconjunto
- 2 Otros ejemplos de *backtracking*
  - Partición en partes iguales
  - Búsqueda en profundidad
- 3 Búsqueda en grafos
  - Búsqueda en amplitud (BFS)
  - DFS no recursivo
  - Uniform Cost Search (UCS)
  - El algoritmo  $A^*$
- 4 Ejercicios propuestos

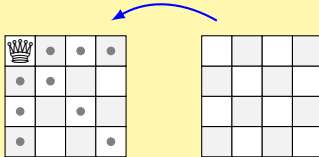
# El problema de las $n$ damas. Un ejemplo con $n = 4$

## El problema de las $n$ damas. Un ejemplo con $n = 4$

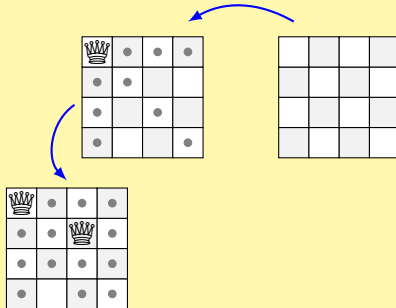




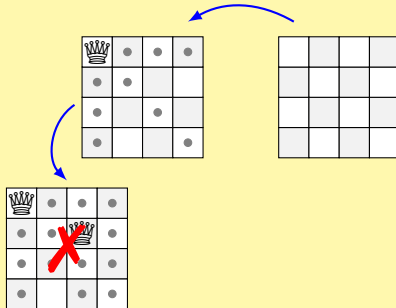
## El problema de las $n$ damas. Un ejemplo con $n = 4$



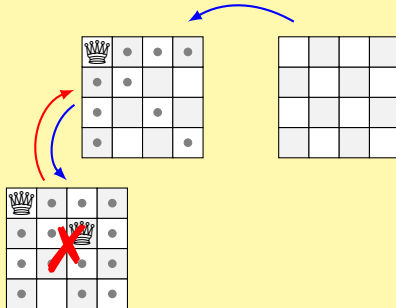
## El problema de las $n$ damas. Un ejemplo con $n = 4$



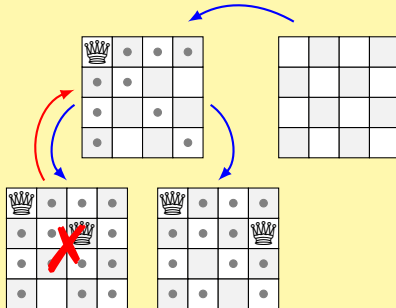
## El problema de las $n$ damas. Un ejemplo con $n = 4$



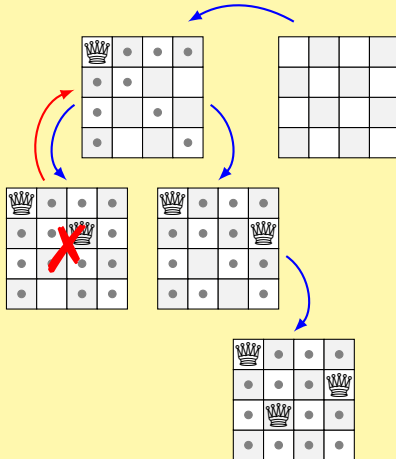
## El problema de las $n$ damas. Un ejemplo con $n = 4$



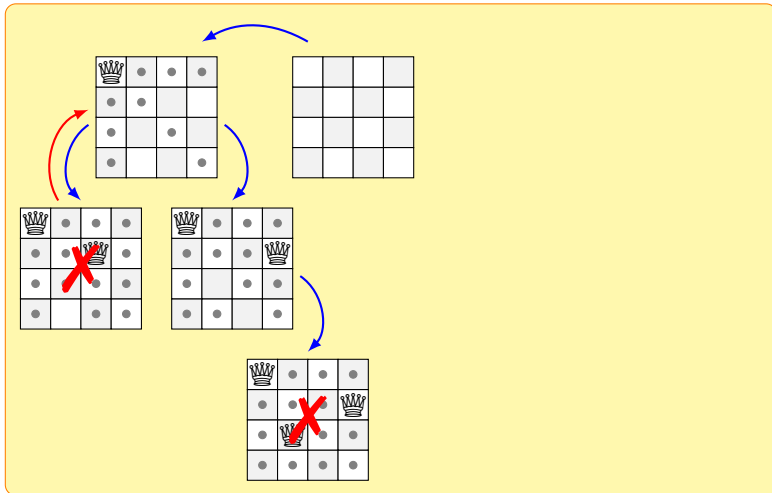
# El problema de las $n$ damas. Un ejemplo con $n = 4$



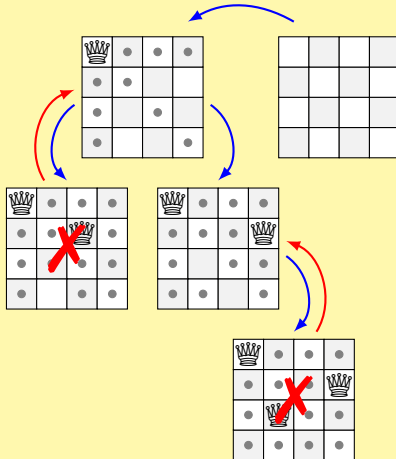
# El problema de las $n$ damas. Un ejemplo con $n = 4$



# El problema de las $n$ damas. Un ejemplo con $n = 4$

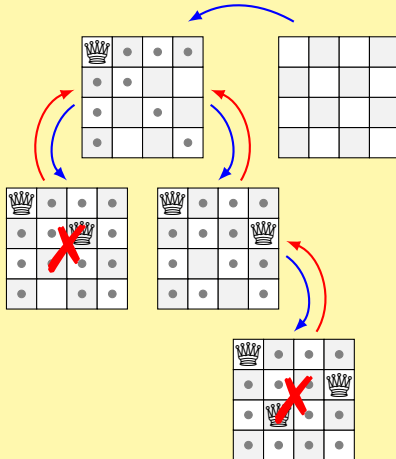


# El problema de las $n$ damas. Un ejemplo con $n = 4$

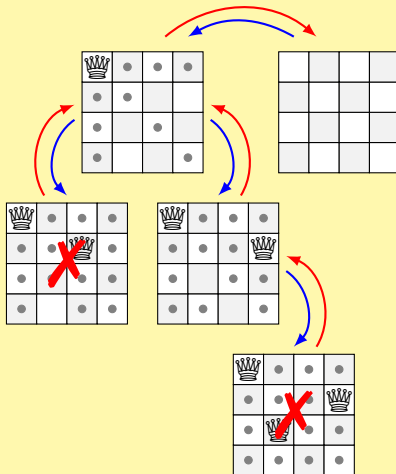




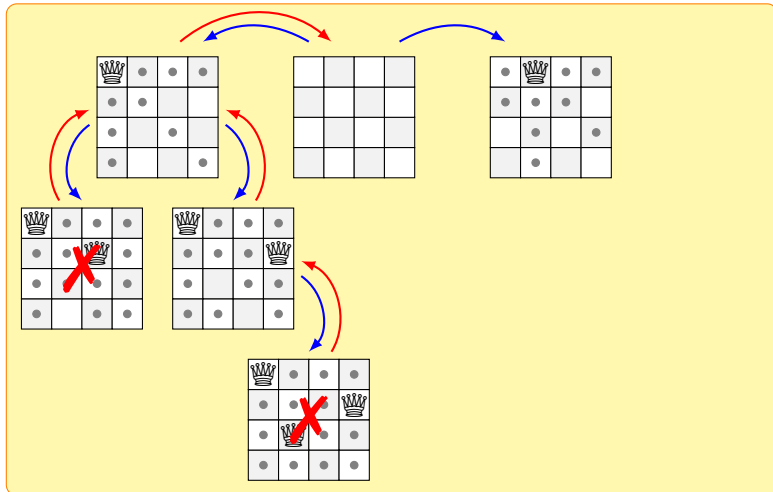
# El problema de las $n$ damas. Un ejemplo con $n = 4$



# El problema de las $n$ damas. Un ejemplo con $n = 4$

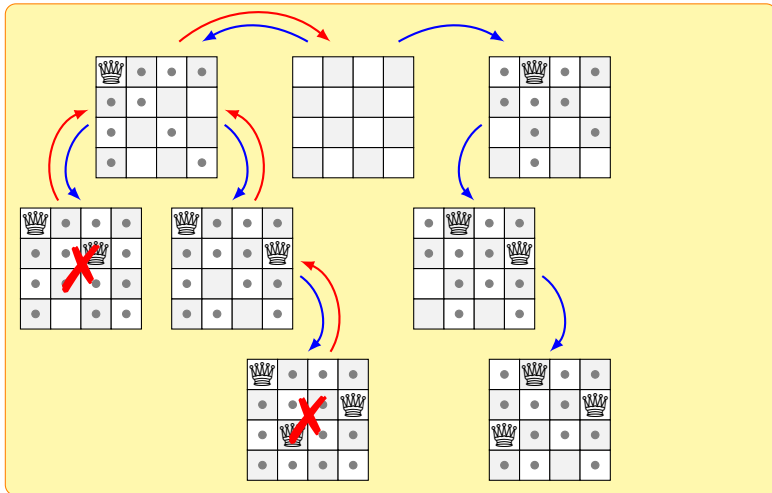


# El problema de las $n$ damas. Un ejemplo con $n = 4$

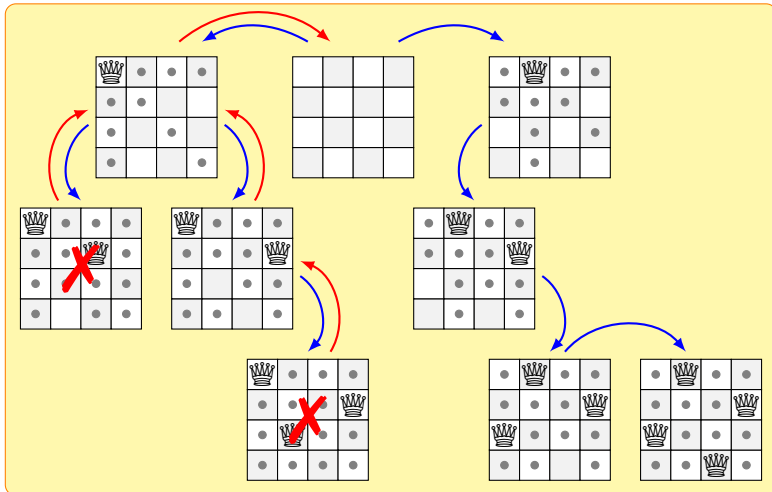




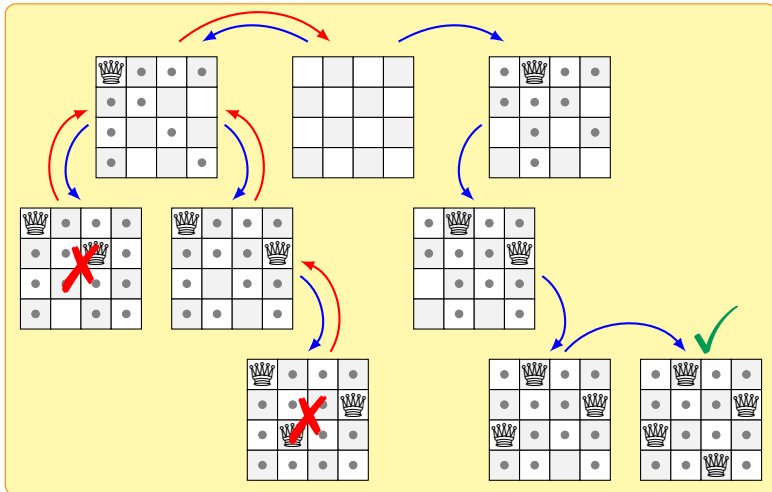
# El problema de las $n$ damas. Un ejemplo con $n = 4$



# El problema de las $n$ damas. Un ejemplo con $n = 4$



# El problema de las $n$ damas. Un ejemplo con $n = 4$



- 1 Repaso de la clase anterior
  - El problema de las  $n$  damas
  - Suma de subconjunto
- 2 Otros ejemplos de *backtracking*
  - Partición en partes iguales
  - Búsqueda en profundidad
- 3 Búsqueda en grafos
  - Búsqueda en amplitud (BFS)
  - DFS no recursivo
  - Uniform Cost Search (UCS)
  - El algoritmo A\*
- 4 Ejercicios propuestos



## El problema de la suma del subconjunto

$$V = \{5, 10, 6, 7\}$$
$$m = 11$$

5

0

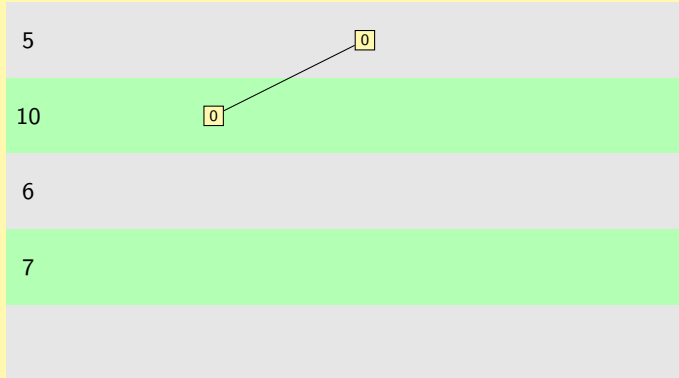
10

6

7

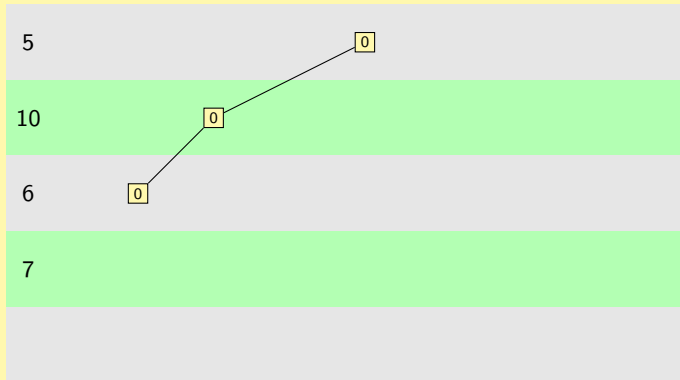
# El problema de la suma del subconjunto

$V = \{5, 10, 6, 7\}$   
 $m = 11$



# El problema de la suma del subconjunto

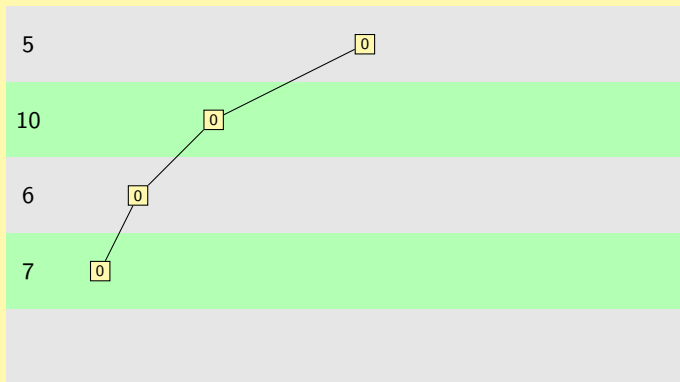
$$V = \{5, 10, 6, 7\}$$
$$m = 11$$



## El problema de la suma del subconjunto

$$V = \{5, 10, 6, 7\}$$

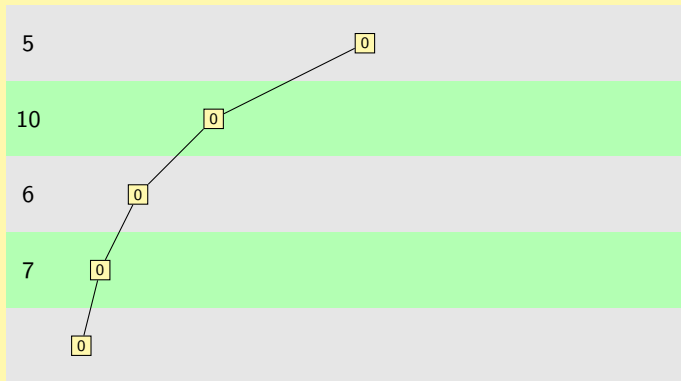
$$m = 11$$



# El problema de la suma del subconjunto

$$V = \{5, 10, 6, 7\}$$

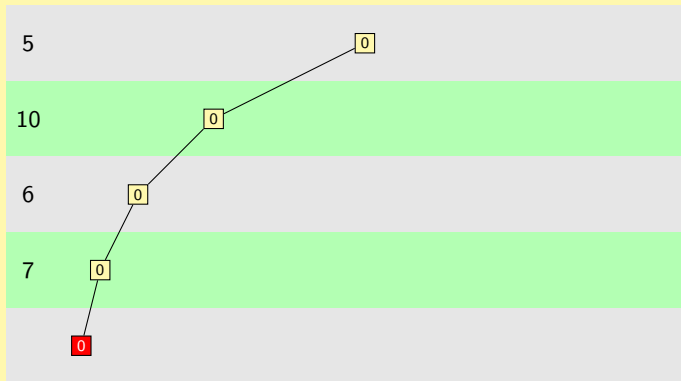
$$m = 11$$



# El problema de la suma del subconjunto

$$V = \{5, 10, 6, 7\}$$

$$m = 11$$

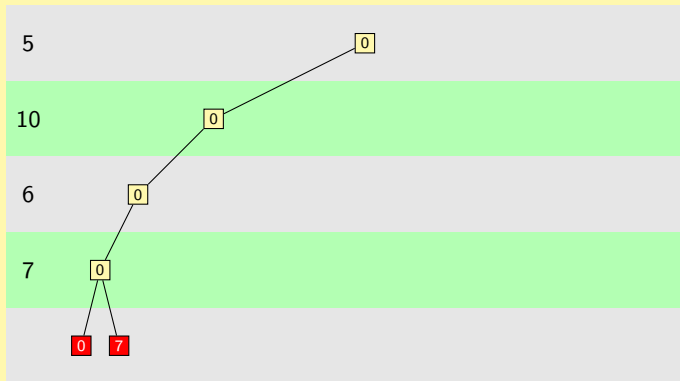




# El problema de la suma del subconjunto

$$V = \{5, 10, 6, 7\}$$

$$m = 11$$

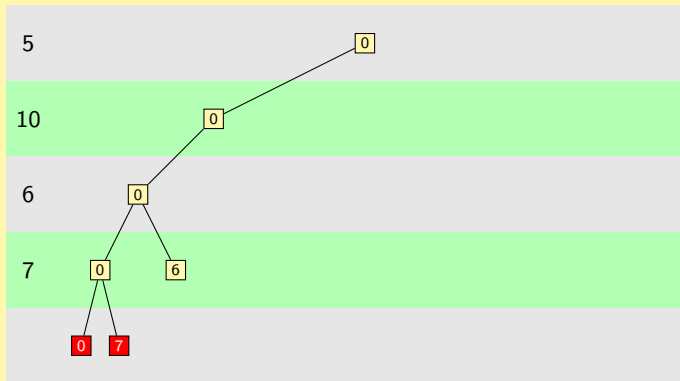




# El problema de la suma del subconjunto

$$V = \{5, 10, 6, 7\}$$

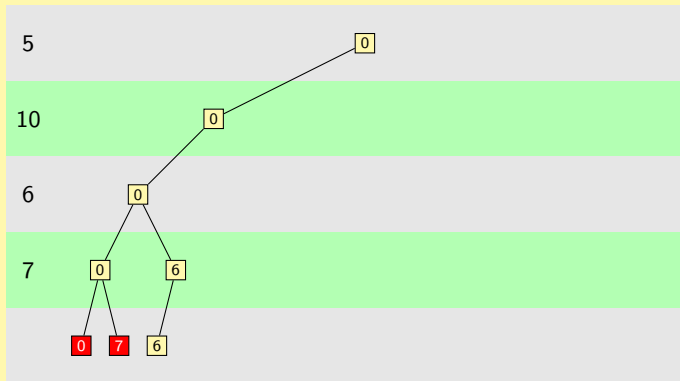
$$m = 11$$



# El problema de la suma del subconjunto

$$V = \{5, 10, 6, 7\}$$

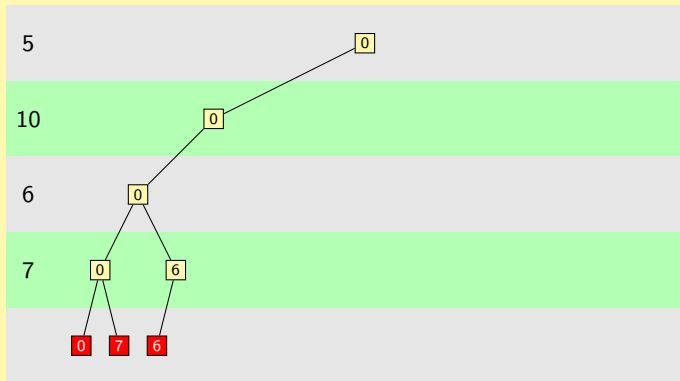
$$m = 11$$



# El problema de la suma del subconjunto

$$V = \{5, 10, 6, 7\}$$

$$m = 11$$

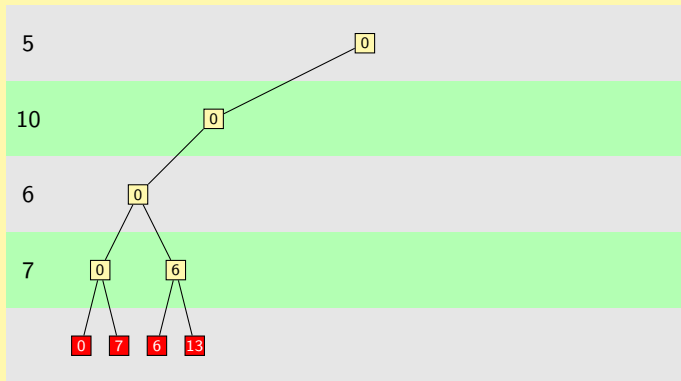




# El problema de la suma del subconjunto

$$V = \{5, 10, 6, 7\}$$

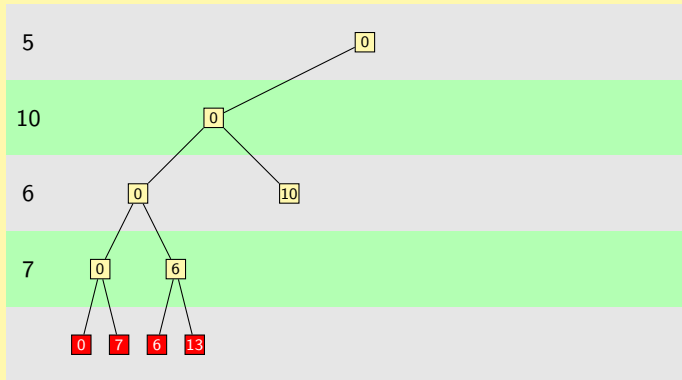
$$m = 11$$



## El problema de la suma del subconjunto

$$V = \{5, 10, 6, 7\}$$

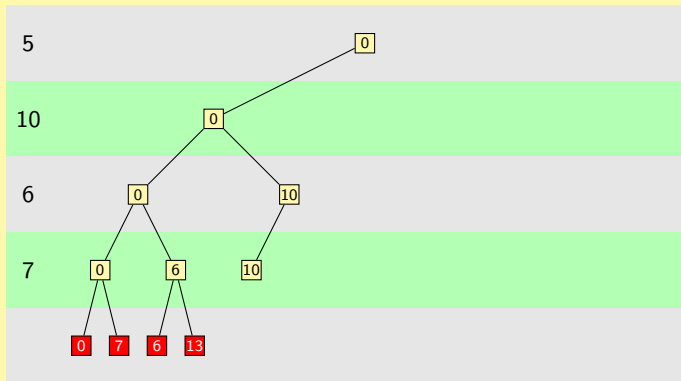
$$m = 11$$



## El problema de la suma del subconjunto

$$V = \{5, 10, 6, 7\}$$

$$m = 11$$



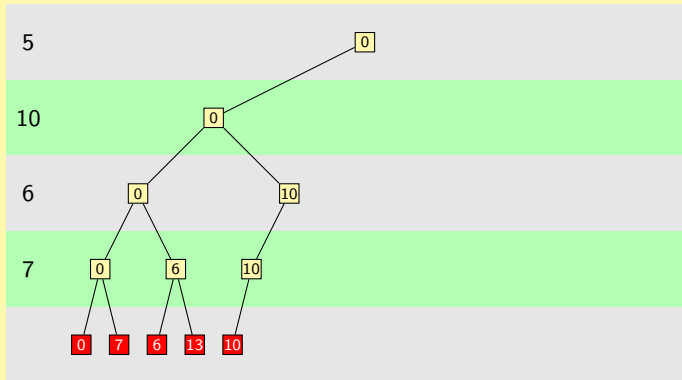




## El problema de la suma del subconjunto

$$V = \{5, 10, 6, 7\}$$

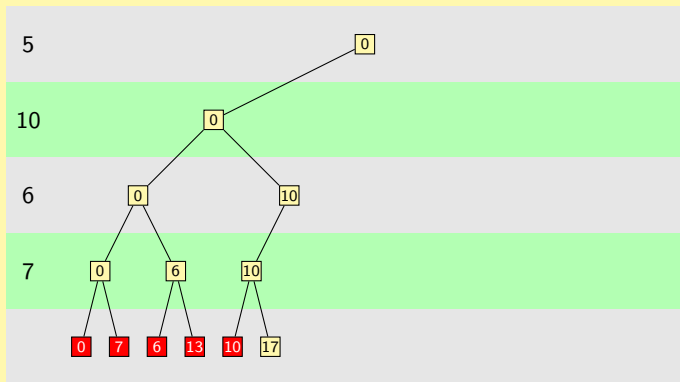
$$m = 11$$



# El problema de la suma del subconjunto

$$V = \{5, 10, 6, 7\}$$

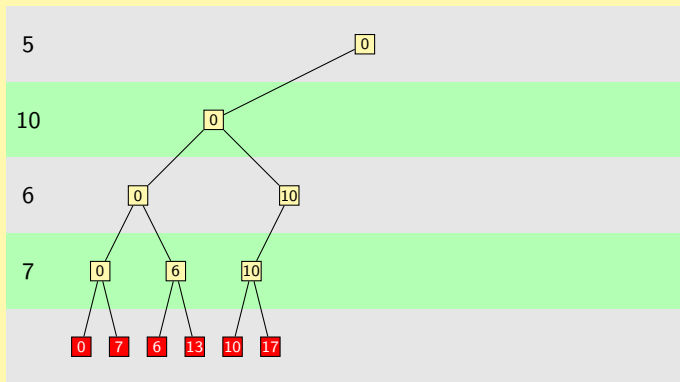
$$m = 11$$



# El problema de la suma del subconjunto

$$V = \{5, 10, 6, 7\}$$

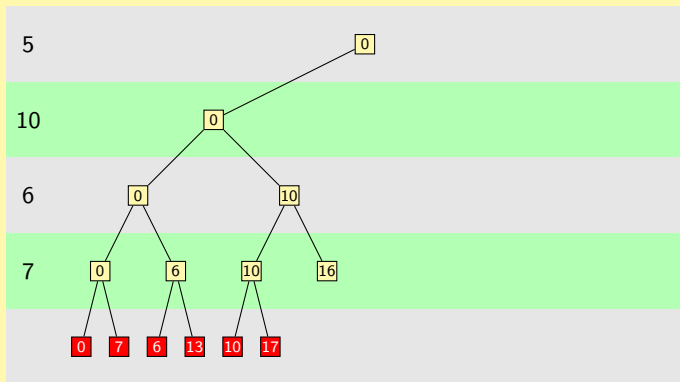
$$m = 11$$



# El problema de la suma del subconjunto

$$V = \{5, 10, 6, 7\}$$

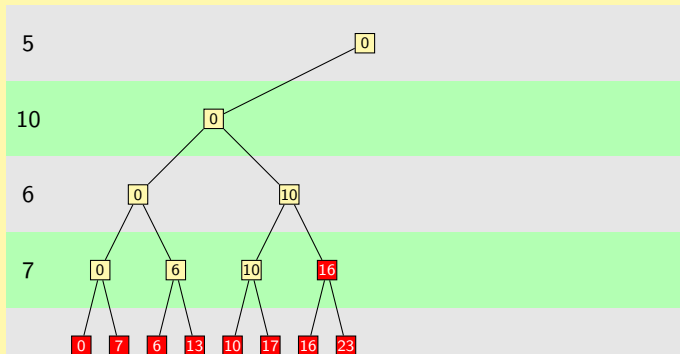
$$m = 11$$



# El problema de la suma del subconjunto

$$V = \{5, 10, 6, 7\}$$

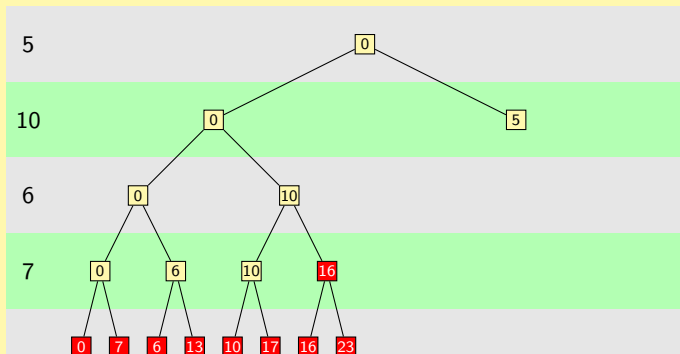
$$m = 11$$



# El problema de la suma del subconjunto

$$V = \{5, 10, 6, 7\}$$

$$m = 11$$

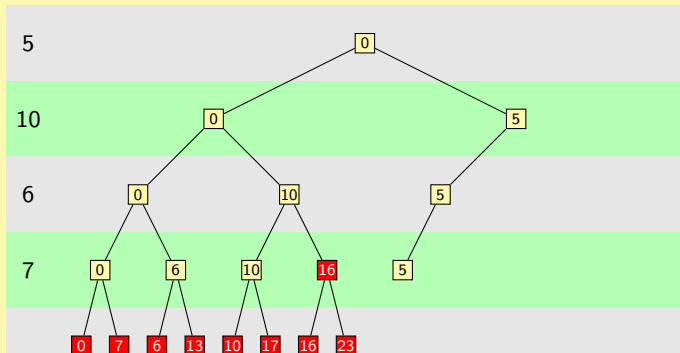




# El problema de la suma del subconjunto

$$V = \{5, 10, 6, 7\}$$

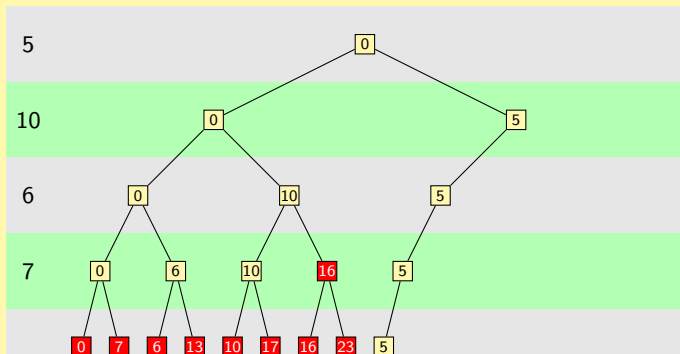
$$m = 11$$





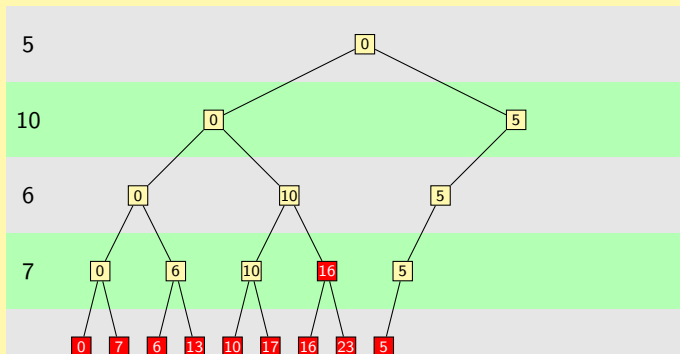
# El problema de la suma del subconjunto

$$V = \{5, 10, 6, 7\}$$
$$m = 11$$



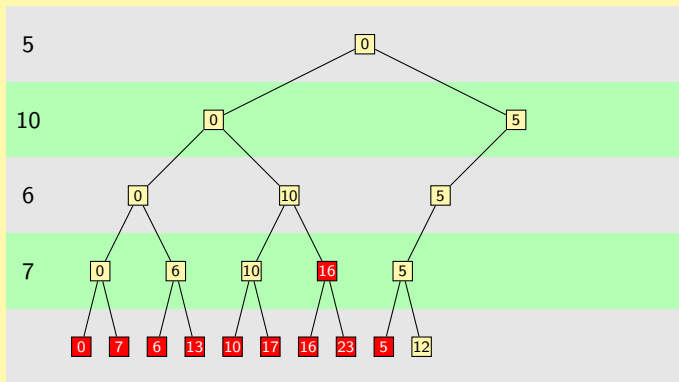
# El problema de la suma del subconjunto

$$V = \{5, 10, 6, 7\}$$
$$m = 11$$



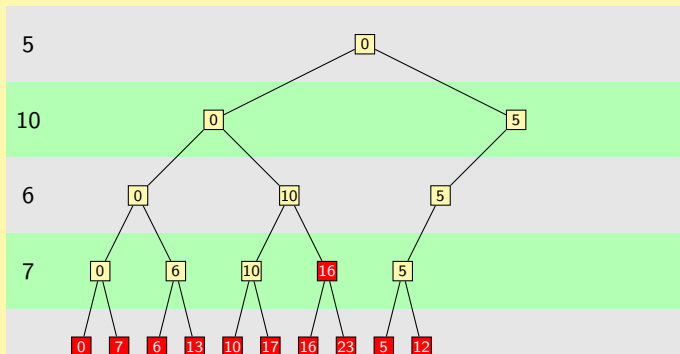
# El problema de la suma del subconjunto

$$V = \{5, 10, 6, 7\}$$
$$m = 11$$



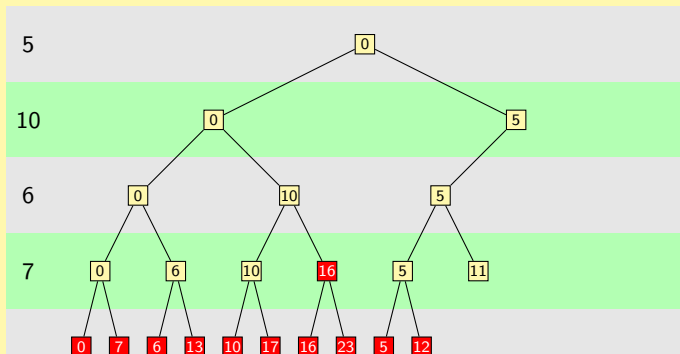
# El problema de la suma del subconjunto

$$V = \{5, 10, 6, 7\}$$
$$m = 11$$



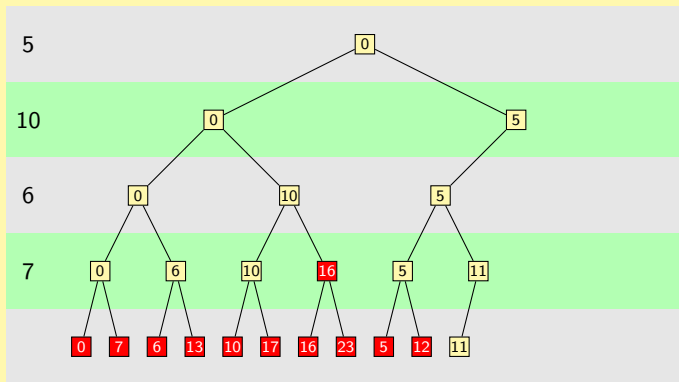
# El problema de la suma del subconjunto

$$V = \{5, 10, 6, 7\}$$
$$m = 11$$



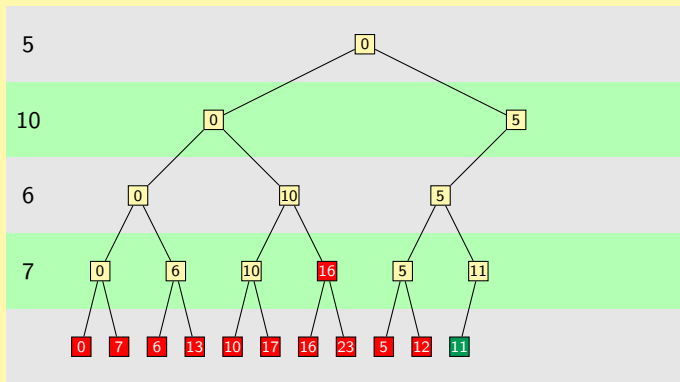
# El problema de la suma del subconjunto

$$V = \{5, 10, 6, 7\}$$
$$m = 11$$



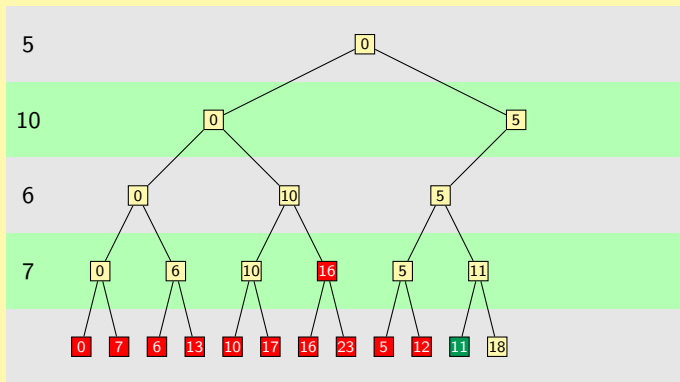
# El problema de la suma del subconjunto

$$V = \{5, 10, 6, 7\}$$
$$m = 11$$



# El problema de la suma del subconjunto

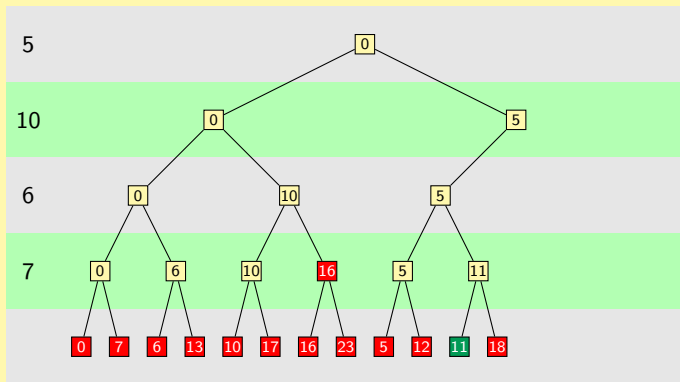
$$V = \{5, 10, 6, 7\}$$
$$m = 11$$





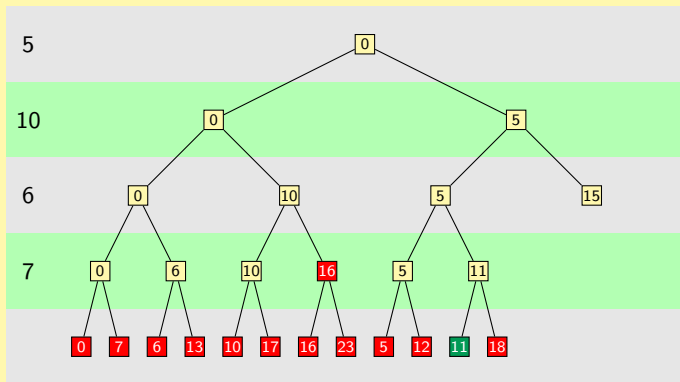
# El problema de la suma del subconjunto

$$V = \{5, 10, 6, 7\}$$
$$m = 11$$



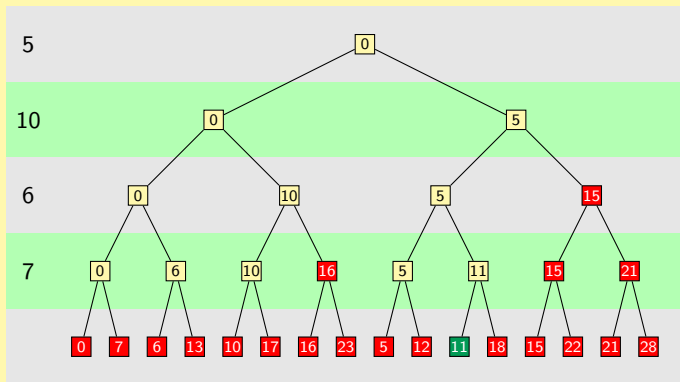
# El problema de la suma del subconjunto

$$V = \{5, 10, 6, 7\}$$
$$m = 11$$



# El problema de la suma del subconjunto

$$V = \{5, 10, 6, 7\}$$
$$m = 11$$



- 1 Repaso de la clase anterior
  - El problema de las  $n$  damas
  - Suma de subconjunto
- 2 Otros ejemplos de *backtracking*
  - Partición en partes iguales
  - Búsqueda en profundidad
- 3 Búsqueda en grafos
  - Búsqueda en amplitud (BFS)
  - DFS no recursivo
  - Uniform Cost Search (UCS)
  - El algoritmo A\*
- 4 Ejercicios propuestos

- 1 Repaso de la clase anterior
  - El problema de las  $n$  damas
  - Suma de subconjunto
- 2 Otros ejemplos de *backtracking*
  - Partición en partes iguales
  - Búsqueda en profundidad
- 3 Búsqueda en grafos
  - Búsqueda en amplitud (BFS)
  - DFS no recursivo
  - Uniform Cost Search (UCS)
  - El algoritmo A\*
- 4 Ejercicios propuestos

# Partición en mitades iguales

## Partición en mitades iguales

- Dada una colección de  $n$  números enteros, se desea encontrar, si existe, una partición de la colección en dos sub-colecciones disjuntas tales que la suma de ambas es igual y que cualquier elemento de la colección original se encuentra en una u otra de las sub-colecciones.

## Partición en mitades iguales

- Dada una colección de  $n$  números enteros, se desea encontrar, si existe, una partición de la colección en dos sub-colecciones disjuntas tales que la suma de ambas es igual y que cualquier elemento de la colección original se encuentra en una u otra de las sub-colecciones.
- La estrategia consistirá en colocar todos los elementos originalmente en una de las sub-colecciones e ir haciendo pasar selectivamente algunos elementos a la otra y verificar si las respectivas sumas de elementos son iguales.



## Antes de seguir. Un ejemplo



## Antes de seguir. Un ejemplo

(2, 5, 8, 3, 2)

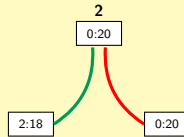
## Antes de seguir. Un ejemplo

(2, 5, 8, 3, 2)

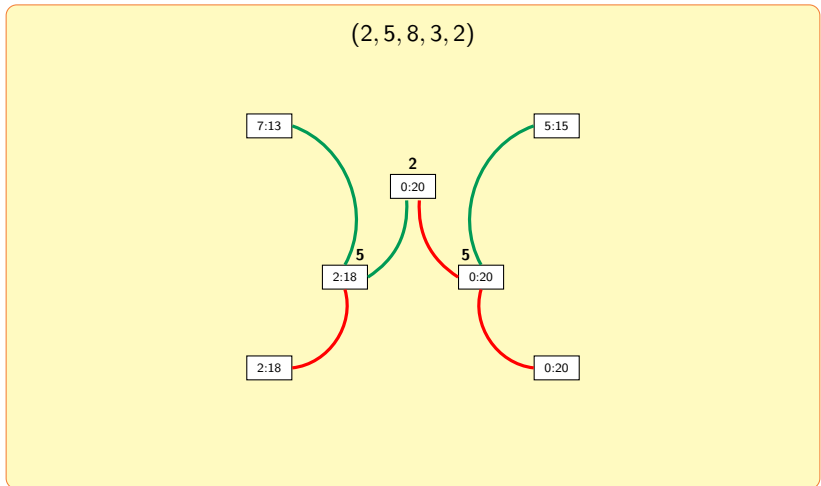
0:20

## Antes de seguir. Un ejemplo

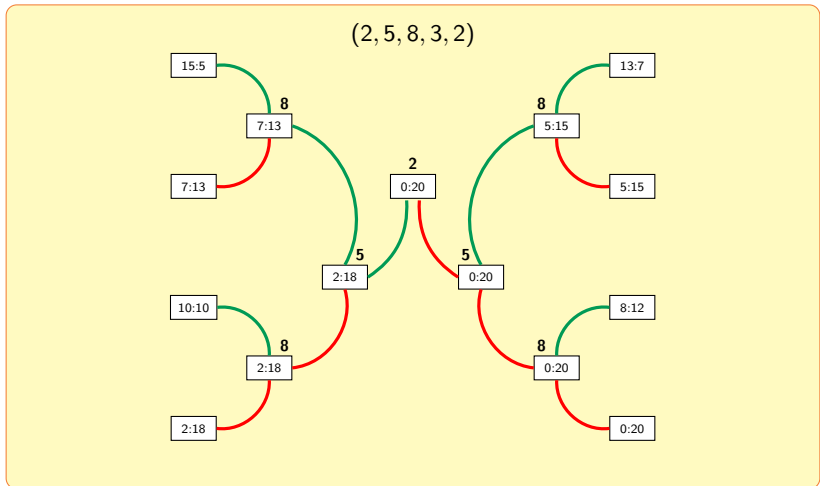
(2, 5, 8, 3, 2)



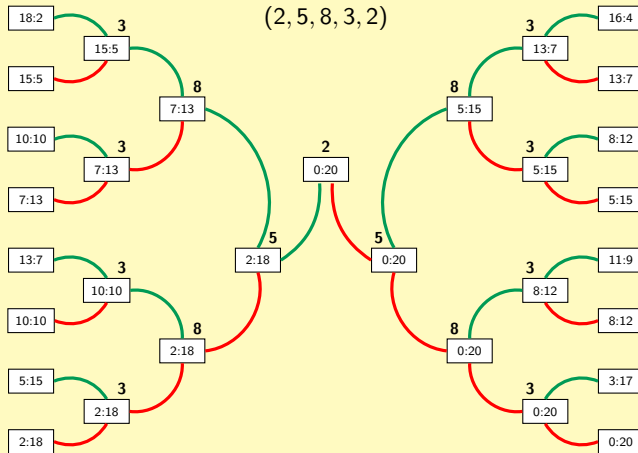
## Antes de seguir. Un ejemplo



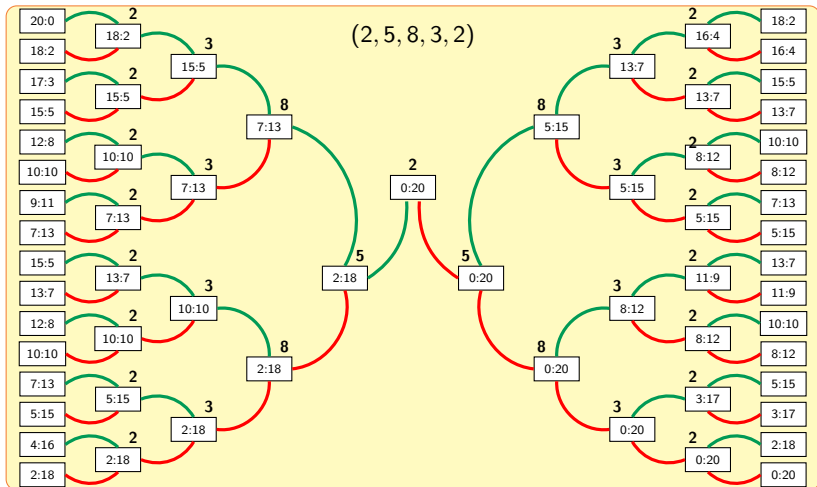
## Antes de seguir. Un ejemplo



## Antes de seguir. Un ejemplo

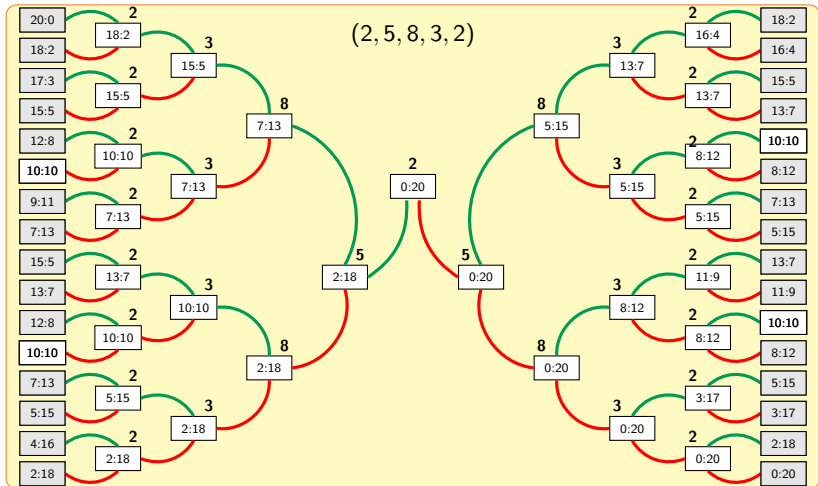


## Antes de seguir. Un ejemplo

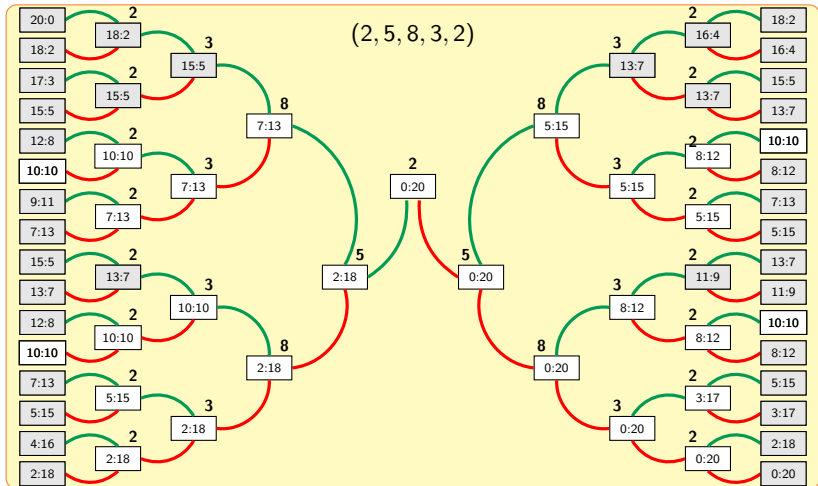




## Antes de seguir. Un ejemplo



# Antes de seguir. Un ejemplo



# El algoritmo para la partición en partes iguales

```
1.  algoritmo Partición (int [] v)
2.  int[] s = initializeArray [n]
3.  ParticiónRec(v, s, 1)

4.  algoritmo ParticiónRec (int[] v, s, int e)
5.  for (i = 0; i ≥ 1; i++) {
6.    s[e] = i
7.    if (e = n) {
8.      int s1 = 0
9.      int s2 = 0
10.     for (j = 1; j ≤ n; j++) {
11.       if (s[j] = 0) {
12.         s1 = s1 + v[j]
13.       } else {
14.         s2 = s2 + v[j]
15.       }
16.     }
17.     if s1 == s2
18.       print (s)
19.     }
20.   } else {
21.     ParticiónRec(v, s, e + 1)
22.   }
23. }
```

// colección izquierda

// colección derecha

$$a = 2, b = 1, k = 1$$

$$\mathcal{O}(n2^n)$$

- 1 Repaso de la clase anterior
  - El problema de las  $n$  damas
  - Suma de subconjunto
- 2 Otros ejemplos de *backtracking*
  - Partición en partes iguales
  - Búsqueda en profundidad
- 3 Búsqueda en grafos
  - Búsqueda en amplitud (BFS)
  - DFS no recursivo
  - Uniform Cost Search (UCS)
  - El algoritmo A\*
- 4 Ejercicios propuestos

# Búsqueda en grafos

# Búsqueda en grafos

- Para el recorrido de todos los nodos de un grafo existen dos criterios de búsqueda.

# Búsqueda en grafos

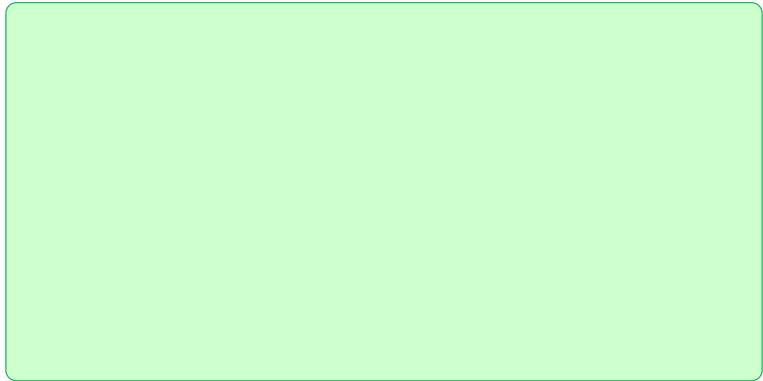
- Para el recorrido de todos los nodos de un grafo existen dos criterios de búsqueda.
- La *búsqueda en amplitud* (*breadth-first search*, o BFS): comienza la búsqueda por un nodo, toma luego sus adyacentes, visita cada uno de ellos y luego los adyacentes de los adyacentes.

## Búsqueda en grafos

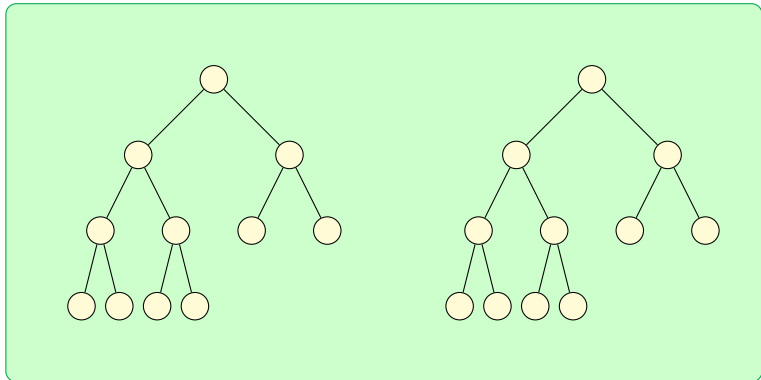
- Para el recorrido de todos los nodos de un grafo existen dos criterios de búsqueda.
- La *búsqueda en amplitud* (*breadth-first search*, o BFS): comienza la búsqueda por un nodo, toma luego sus adyacentes, visita cada uno de ellos y luego los adyacentes de los adyacentes.
- La *búsqueda en profundidad* (*depth-first search*, o DFS): comienza la búsqueda por un nodo, toma luego sus adyacentes y a partir de uno de ellos realiza a su vez la búsqueda en profundidad. Cuando termina la búsqueda para ese nodo adyacente continúa con el siguiente.



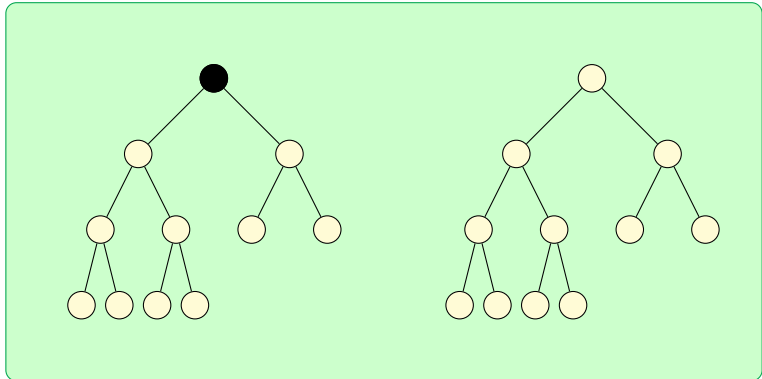
# Búsqueda en grafos: profundidad y amplitud



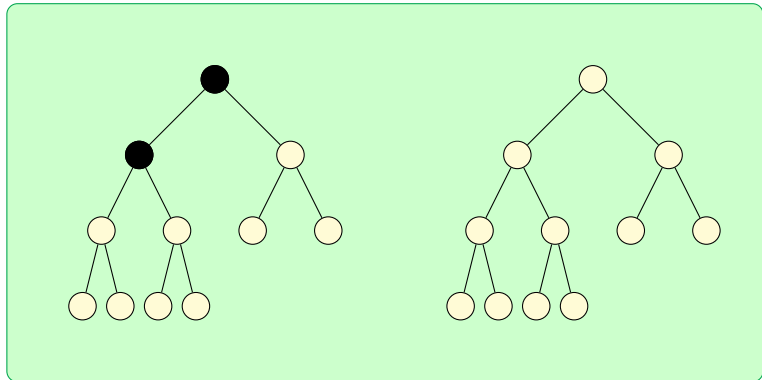
# Búsqueda en grafos: profundidad y amplitud



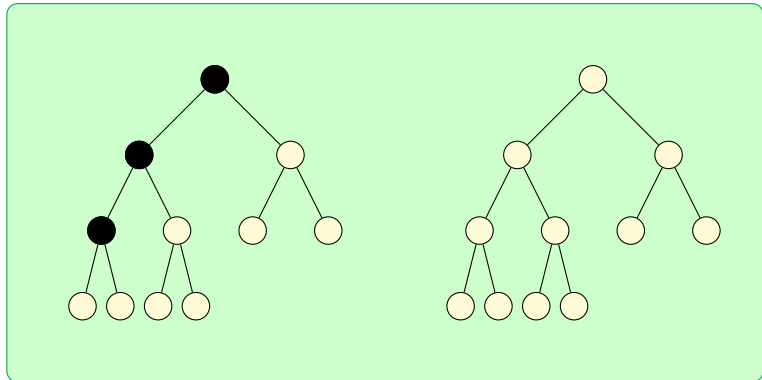
# Búsqueda en grafos: profundidad y amplitud



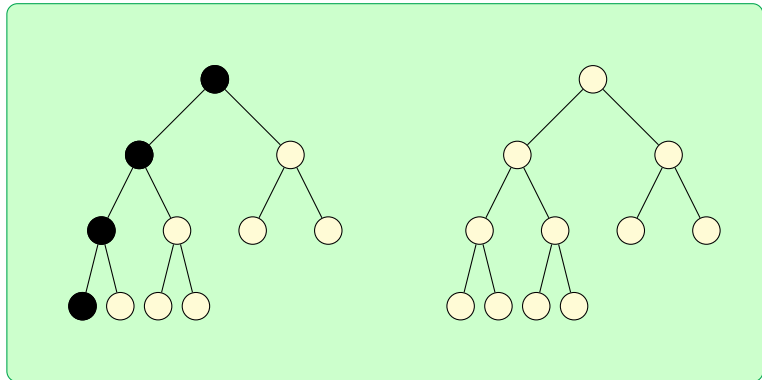
# Búsqueda en grafos: profundidad y amplitud



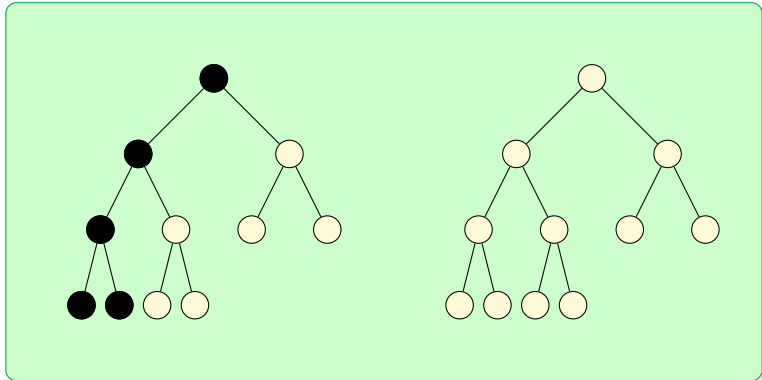
# Búsqueda en grafos: profundidad y amplitud



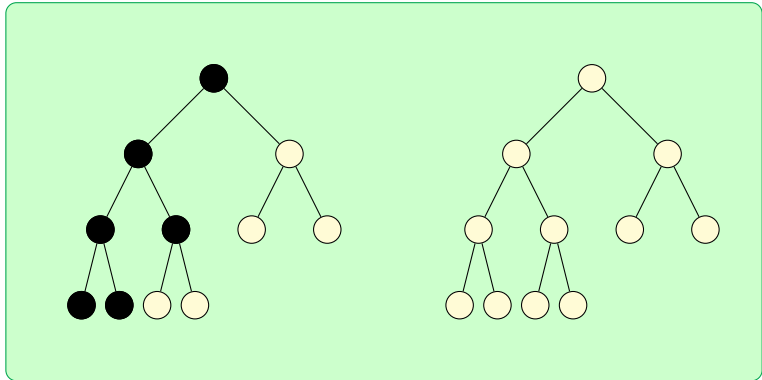
# Búsqueda en grafos: profundidad y amplitud



# Búsqueda en grafos: profundidad y amplitud

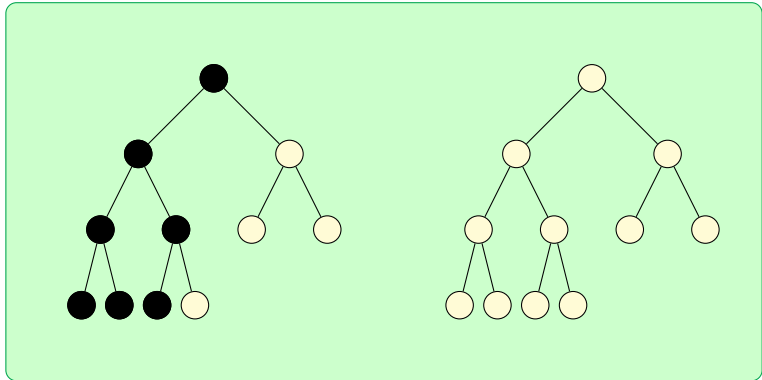


# Búsqueda en grafos: profundidad y amplitud

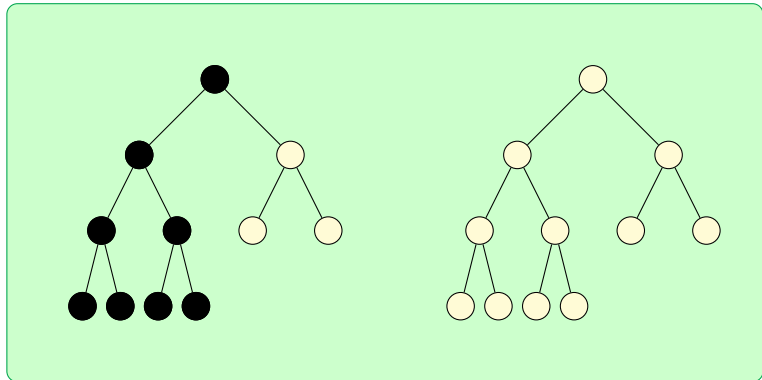




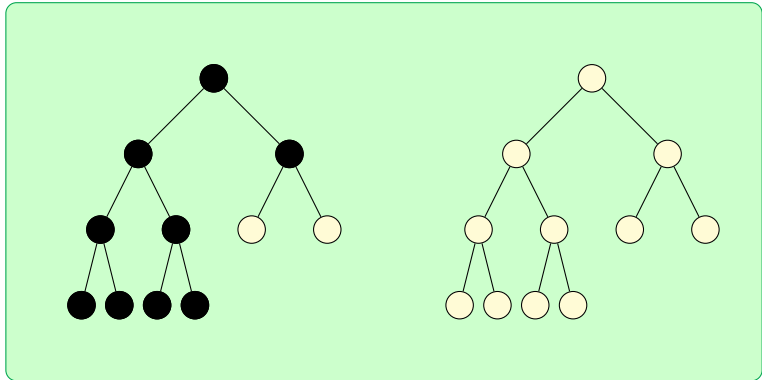
# Búsqueda en grafos: profundidad y amplitud



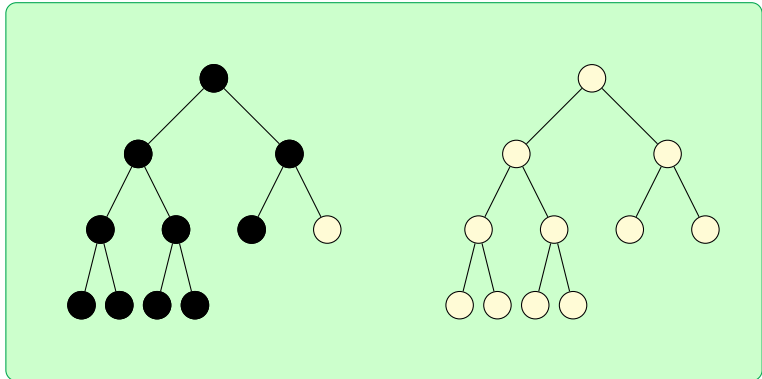
# Búsqueda en grafos: profundidad y amplitud



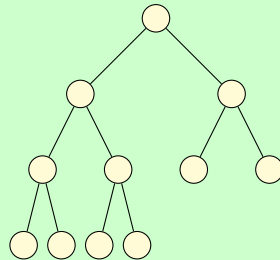
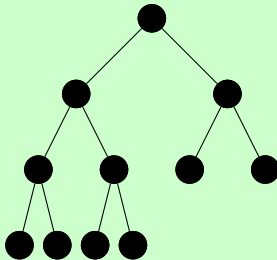
# Búsqueda en grafos: profundidad y amplitud



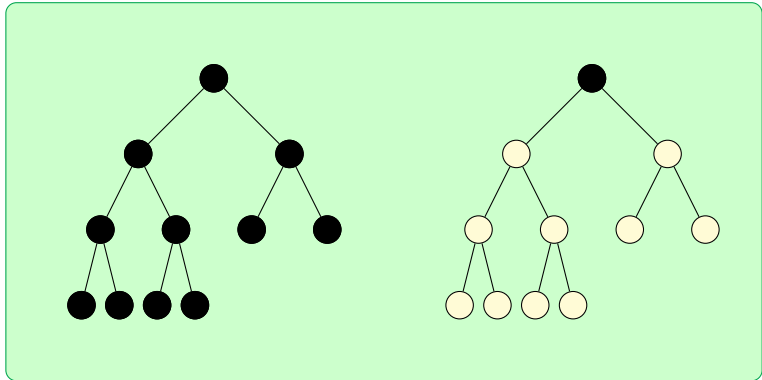
# Búsqueda en grafos: profundidad y amplitud



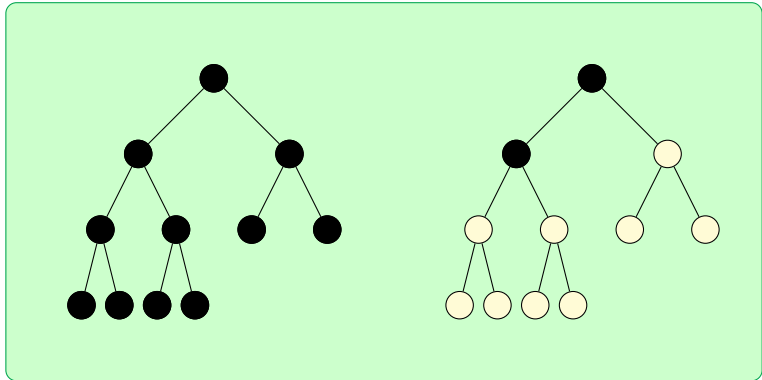
# Búsqueda en grafos: profundidad y amplitud



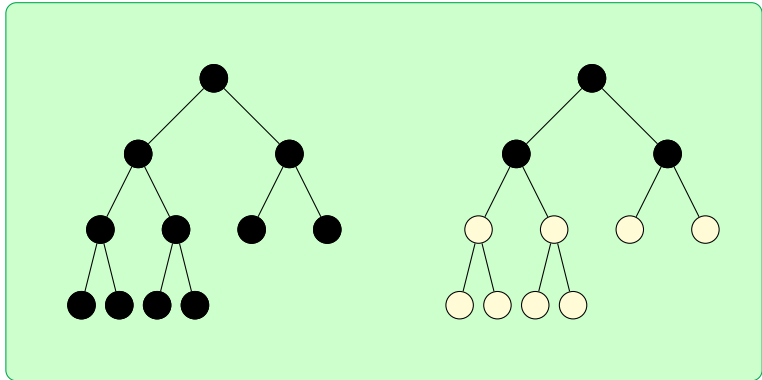
# Búsqueda en grafos: profundidad y amplitud



# Búsqueda en grafos: profundidad y amplitud

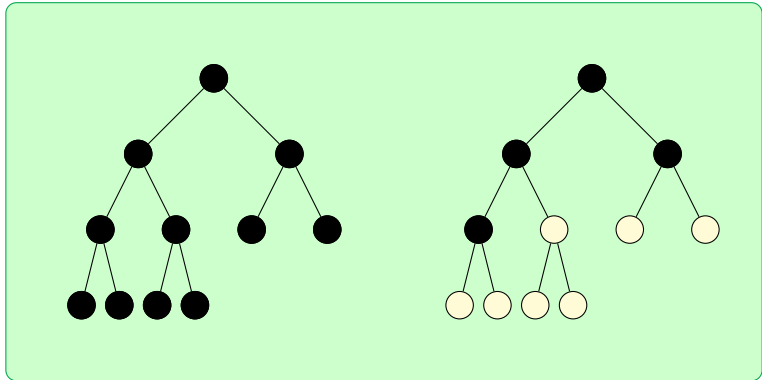


# Búsqueda en grafos: profundidad y amplitud

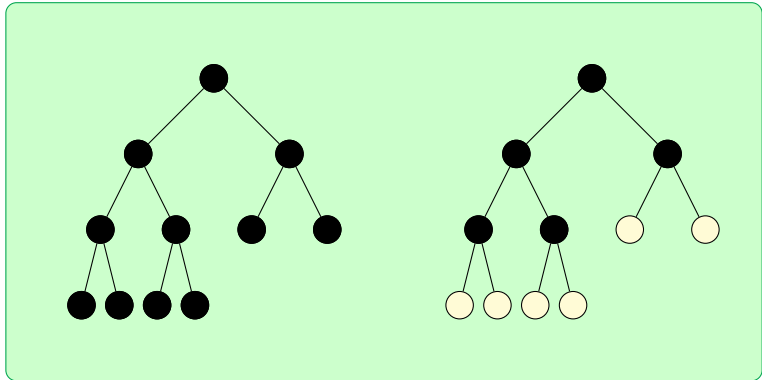




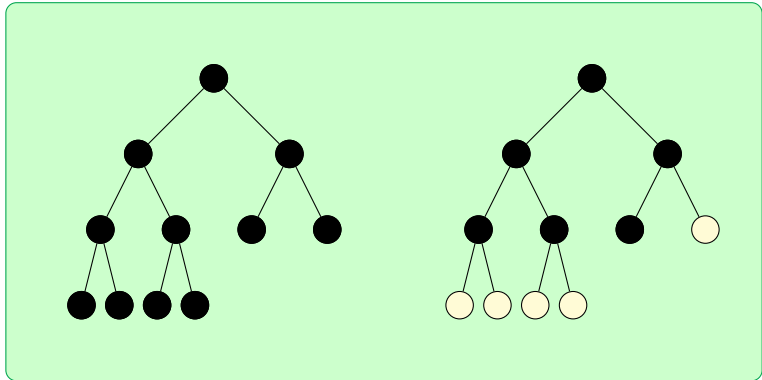
# Búsqueda en grafos: profundidad y amplitud



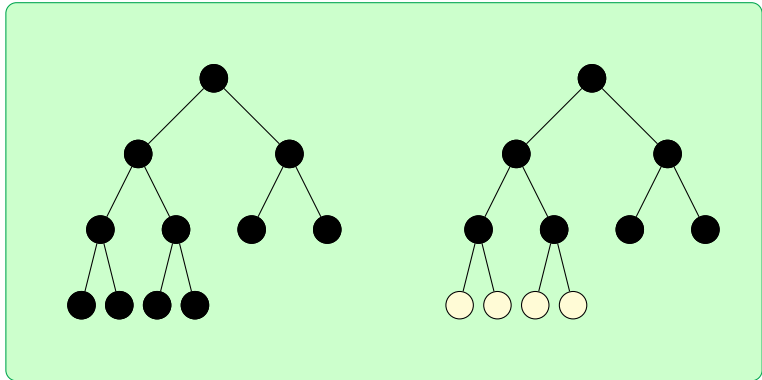
# Búsqueda en grafos: profundidad y amplitud



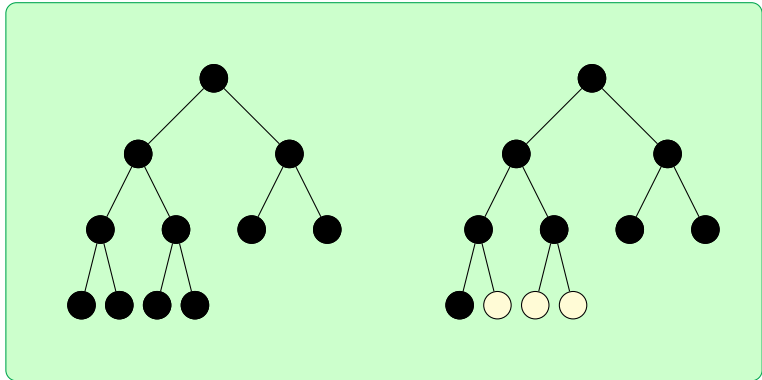
# Búsqueda en grafos: profundidad y amplitud



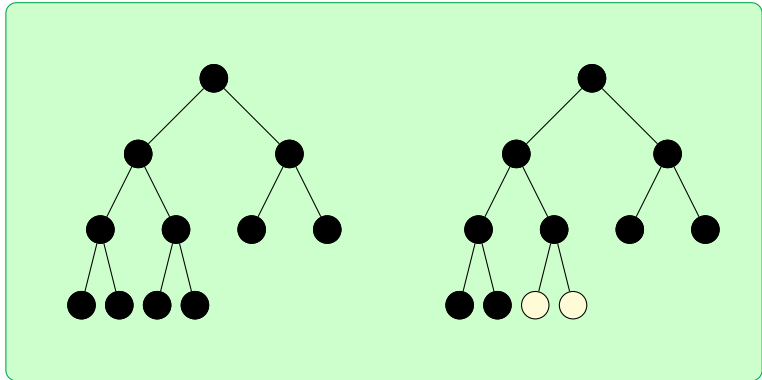
# Búsqueda en grafos: profundidad y amplitud



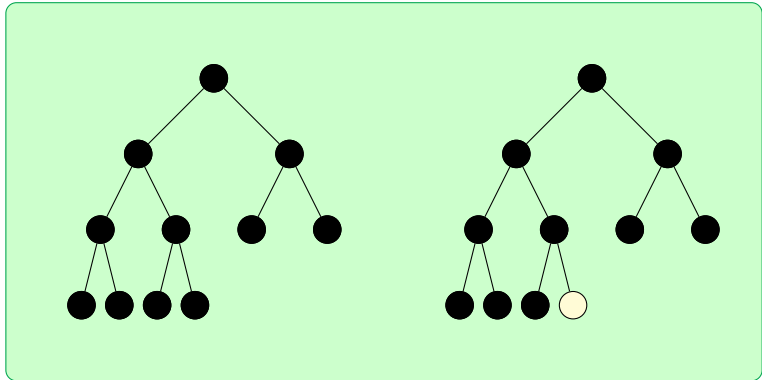
# Búsqueda en grafos: profundidad y amplitud



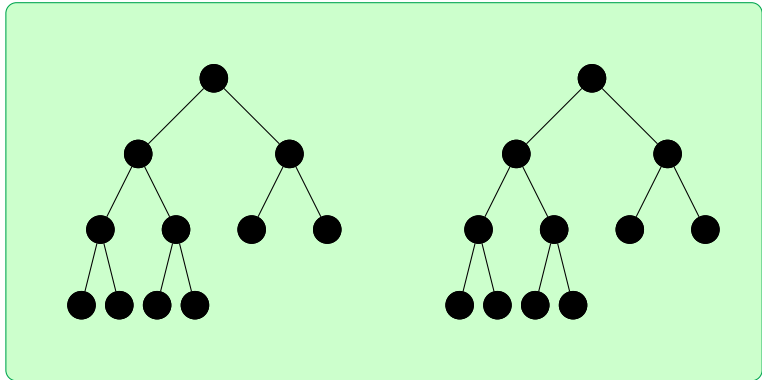
# Búsqueda en grafos: profundidad y amplitud



# Búsqueda en grafos: profundidad y amplitud

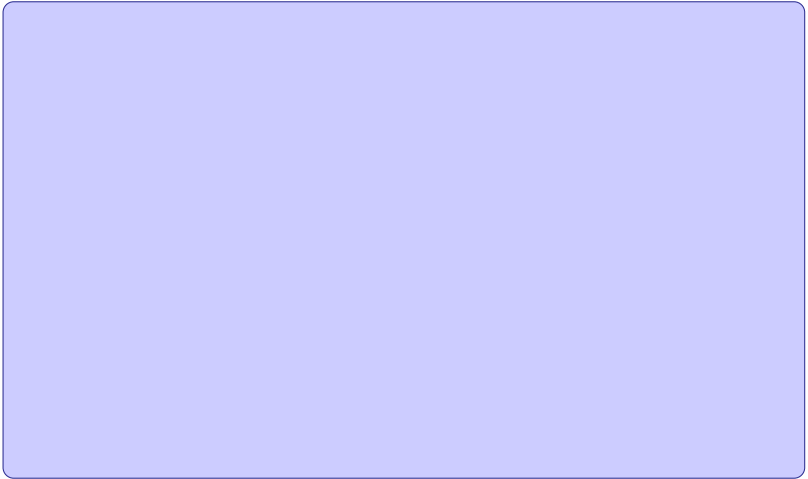


# Búsqueda en grafos: profundidad y amplitud

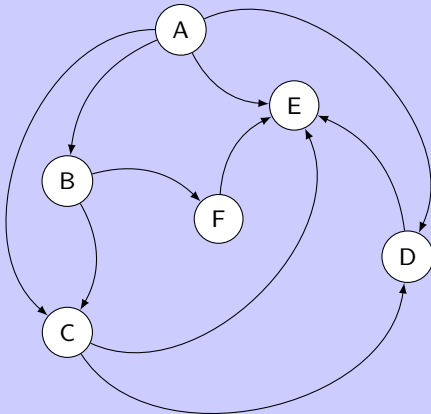




# Búsqueda en profundidad (DFS)



## Búsqueda en profundidad (DFS)



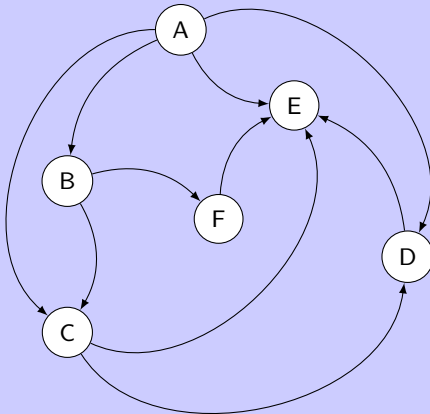
**Vértice actual:**

**Vecindario:**

**Visitados:**

**Pila:**

# Búsqueda en profundidad (DFS)



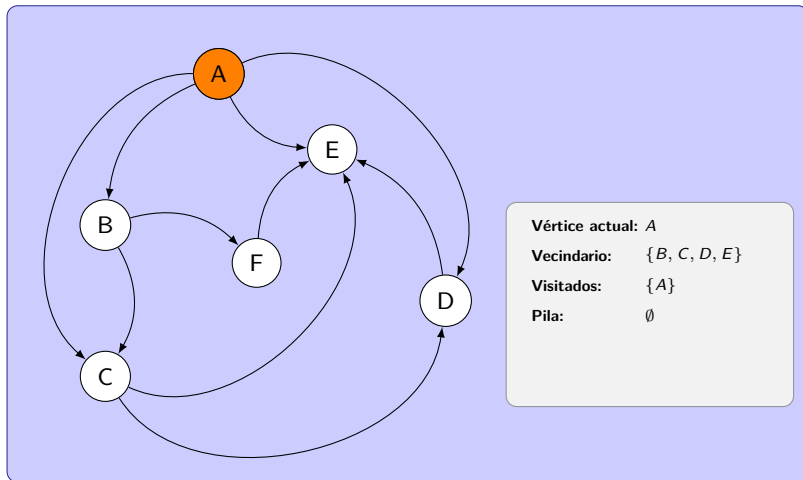
**Vértice actual:** none

**Vecindario:**  $\emptyset$

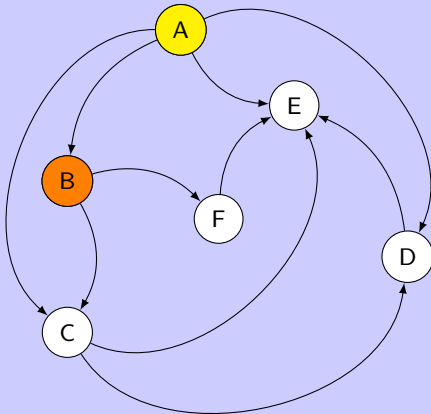
**Visitados:**  $\emptyset$

**Pila:**  $\emptyset$

# Búsqueda en profundidad (DFS)



## Búsqueda en profundidad (DFS)



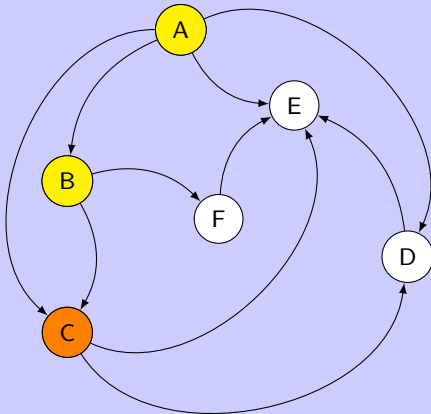
**Vértice actual:** B

**Vecindario:** {C, F}

**Visitados:** {A, B}

**Pila:** A : {C, D, E}

## Búsqueda en profundidad (DFS)



**Vértice actual:** C

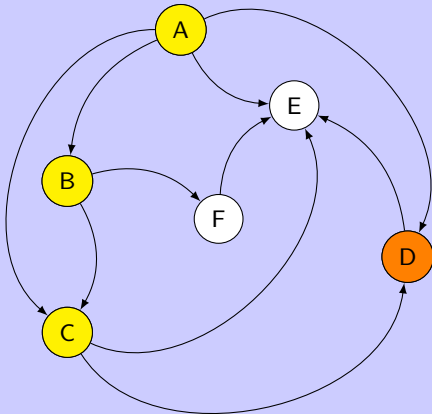
**Vecindario:** {D, E}

**Visitados:** {A, B, C}

**Pila:** B : {F}

A : {C, D, E}

## Búsqueda en profundidad (DFS)



**Vértice actual:** *D*

**Vecindario:**  $\{E\}$

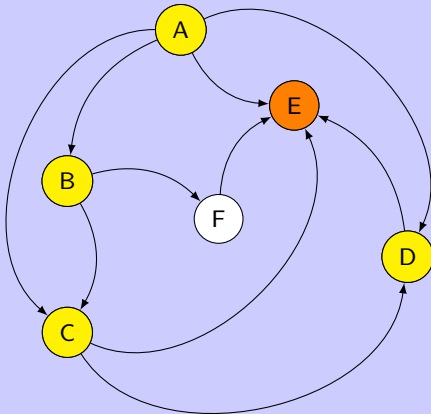
**Visitados:**  $\{A, B, C, D\}$

**Pila:**  $C : \{E\}$

$B : \{F\}$

$A : \{C, D, E\}$

## Búsqueda en profundidad (DFS)



**Vértice actual:** *E*

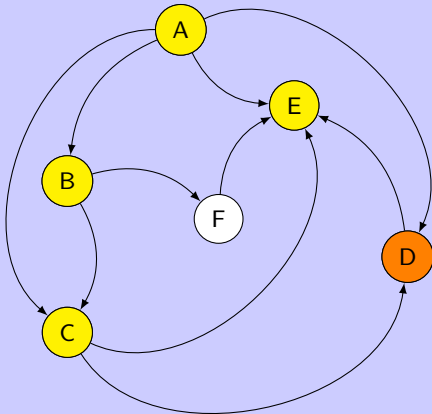
**Vecindario:**  $\emptyset$

**Visitados:**  $\{A, B, C, D, E\}$

**Pila:**  
*D* :  $\emptyset$   
*C* :  $\{E\}$   
*B* :  $\{F\}$   
*A* :  $\{C, D, E\}$



## Búsqueda en profundidad (DFS)



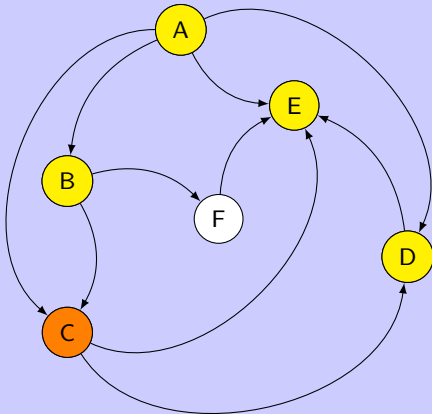
**Vértice actual:** *D*

**Vecindario:**  $\emptyset$

**Visitados:**  $\{A, B, C, D, E\}$

**Pila:**  
*C* :  $\{E\}$   
*B* :  $\{F\}$   
*A* :  $\{C, D, E\}$

## Búsqueda en profundidad (DFS)



**Vértice actual:** C

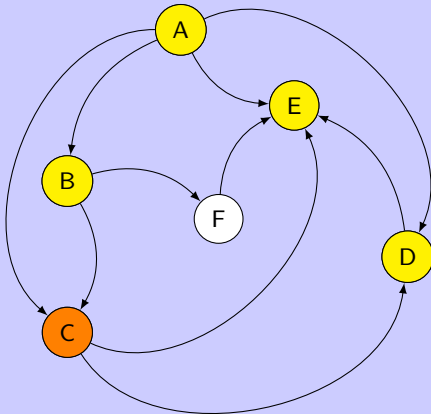
**Vecindario:** {E}

**Visitados:** {A, B, C, D, E}

**Pila:** B : {F}

A : {C, D, E}

## Búsqueda en profundidad (DFS)



**Vértice actual:** C

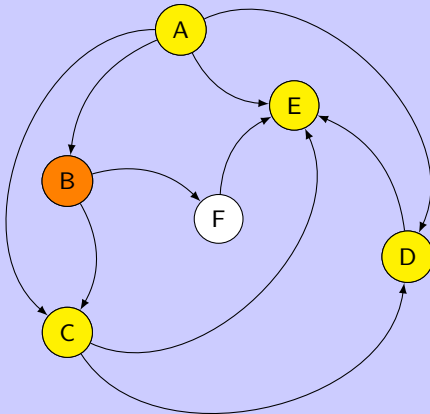
**Vecindario:**  $\emptyset$

**Visitados:** {A, B, C, D, E}

**Pila:** B : {F}

A : {C, D, E}

# Búsqueda en profundidad (DFS)



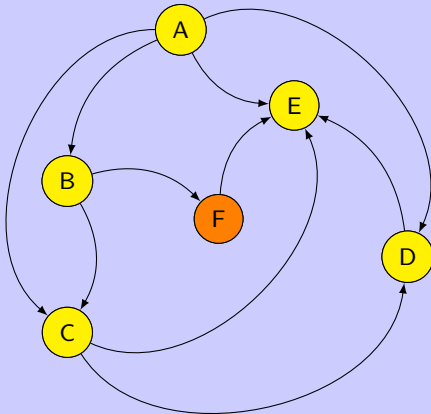
**Vértice actual:** B

**Vecindario:** {F}

**Visitados:** {A, B, C, D, E}

**Pila:** A : {C, D, E}

## Búsqueda en profundidad (DFS)



**Vértice actual:**  $F$

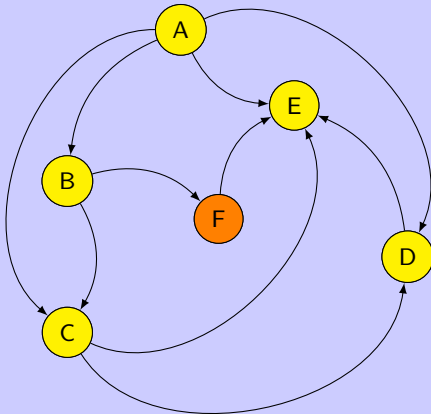
**Vecindario:**  $\{E\}$

**Visitados:**  $\{A, B, C, D, E, F\}$

**Pila:**  $B : \emptyset$

$A : \{C, D, E\}$

## Búsqueda en profundidad (DFS)



**Vértice actual:**  $F$

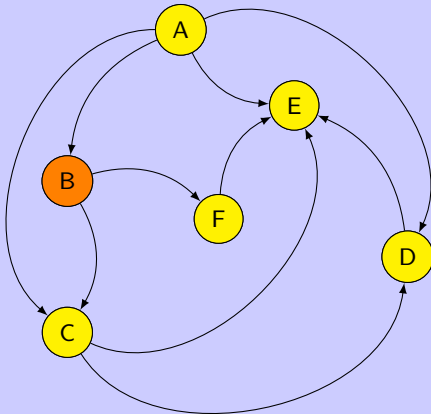
**Vecindario:**  $\emptyset$

**Visitados:**  $\{A, B, C, D, E, F\}$

**Pila:**  $B : \emptyset$

$A : \{C, D, E\}$

## Búsqueda en profundidad (DFS)



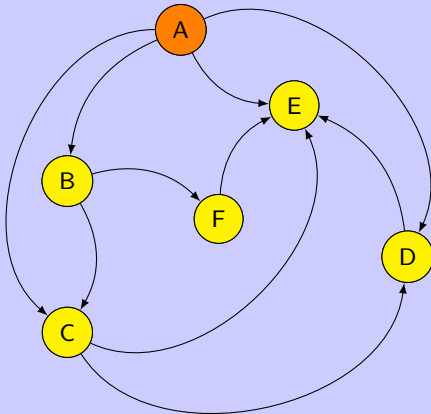
**Vértice actual:** B

**Vecindario:**  $\emptyset$

**Visitados:** {A, B, C, D, E, F}

**Pila:** A : {C, D, E}

## Búsqueda en profundidad (DFS)



**Vértice actual:** A

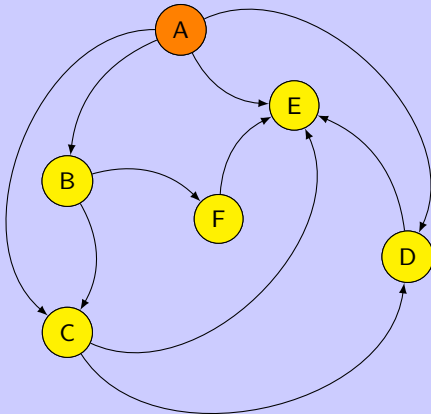
**Vecindario:** {C, D, E}

**Visitados:** {A, B, C, D, E, F}

**Pila:**  $\emptyset$



## Búsqueda en profundidad (DFS)



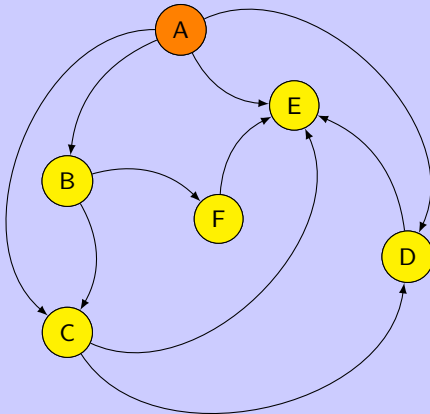
**Vértice actual:** A

**Vecindario:** {D, E}

**Visitados:** {A, B, C, D, E, F}

**Pila:**  $\emptyset$

# Búsqueda en profundidad (DFS)



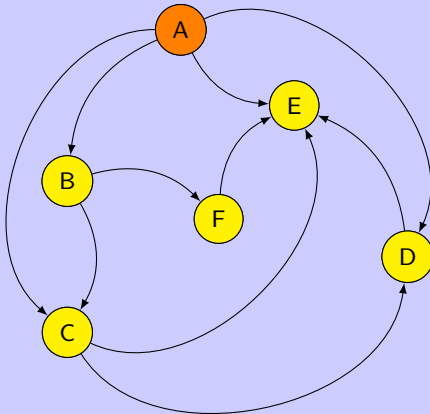
**Vértice actual:** A

**Vecindario:** {E}

**Visitados:** {A, B, C, D, E, F}

**Pila:**  $\emptyset$

## Búsqueda en profundidad (DFS)



**Vértice actual:** A

**Vecindario:**  $\emptyset$

**Visitados:** {A, B, C, D, E, F}

**Pila:**  $\emptyset$

# El algoritmo para búsqueda en profundidad

```
1.  algoritmo BusqProf (grafo G, nodo v)    // driver
2.    DFS(G, v,  $\emptyset$ )

3.  algoritmo DFS (grafo G, nodo v, set visitados)
4.    addElement(visitados, v)
5.    print (v)
6.    vecindario = adyacentes(G, v)
7.    while (vecindario  $\neq \emptyset$ ) {           // mientras haya vecinos
8.        a = choose(vecindario)             // elegir uno
9.        delete(vecindario, a)
10.       if (!Pertenece(visitados, a)) {     // si no visitado
11.           DFS(G, a, visitados)
12.       }
13.    }
```

- 1 Repaso de la clase anterior
  - El problema de las  $n$  damas
  - Suma de subconjunto
- 2 Otros ejemplos de *backtracking*
  - Partición en partes iguales
  - Búsqueda en profundidad
- 3 Búsqueda en grafos
  - Búsqueda en amplitud (BFS)
  - DFS no recursivo
  - Uniform Cost Search (UCS)
  - El algoritmo A\*
- 4 Ejercicios propuestos

- 1 Repaso de la clase anterior
  - El problema de las  $n$  damas
  - Suma de subconjunto
- 2 Otros ejemplos de *backtracking*
  - Partición en partes iguales
  - Búsqueda en profundidad
- 3 **Búsqueda en grafos**
  - **Búsqueda en amplitud (BFS)**
  - DFS no recursivo
  - Uniform Cost Search (UCS)
  - El algoritmo A\*
- 4 Ejercicios propuestos

# Búsqueda en amplitud

# Búsqueda en amplitud

- No usa backtracking.



# Búsqueda en amplitud

- No usa backtracking.
- Encuentra siempre la solución más cercana al punto de origen, si existe.

# Búsqueda en amplitud

- No usa backtracking.
- Encuentra siempre la solución más cercana al punto de origen, si existe.
- Se adapta bien a problemas de los que sabemos que la solución está cerca del punto de origen.

# Búsqueda en amplitud

- No usa backtracking.
- Encuentra siempre la solución más cercana al punto de origen, si existe.
- Se adapta bien a problemas de los que sabemos que la solución está cerca del punto de origen.
- Por supuesto, debemos conocer esto de antemano.

# Tipos abstractos de datos para BFS 1

# Tipos abstractos de datos para BFS 1

# Tipos abstractos de datos para BFS 1

- El TDA *conjunto* tiene los siguientes métodos:
  - *elegir*(*S*): devuelve un elemento arbitrario del conjunto *S*.
  - *conjuntoVacio*(*S*): devuelve *true* si el conjunto *S* está vacío o *false* en caso contrario.
  - *pertenece*(*S*, *x*): devuelve *true* si el elemento *x* pertenece al conjunto *S* y *false* en caso contrario.
  - *eliminar*(*S*, *x*): elimina el elemento *x* del conjunto *S*.
  - *agregar*(*S*, *x*): agrega el elemento *x* al conjunto *S*.

# Tipos abstractos de datos para BFS 1

- El TDA *conjunto* tiene los siguientes métodos:
  - *elegir*(*S*): devuelve un elemento arbitrario del conjunto *S*.
  - *conjuntoVacio*(*S*): devuelve *true* si el conjunto *S* está vacío o *false* en caso contrario.
  - *pertenece*(*S*, *x*): devuelve *true* si el elemento *x* pertenece al conjunto *S* y *false* en caso contrario.
  - *eliminar*(*S*, *x*): elimina el elemento *x* del conjunto *S*.
  - *agregar*(*S*, *x*): agrega el elemento *x* al conjunto *S*.
- El TDA *grafo* con el método *vecindario*(*x*) que, para un grafo dado devuelve el conjunto  $\{u \mid (x, u) \in A\}$  con los vértices adyacentes a *x*.

# Tipos abstractos de datos para BFS 2



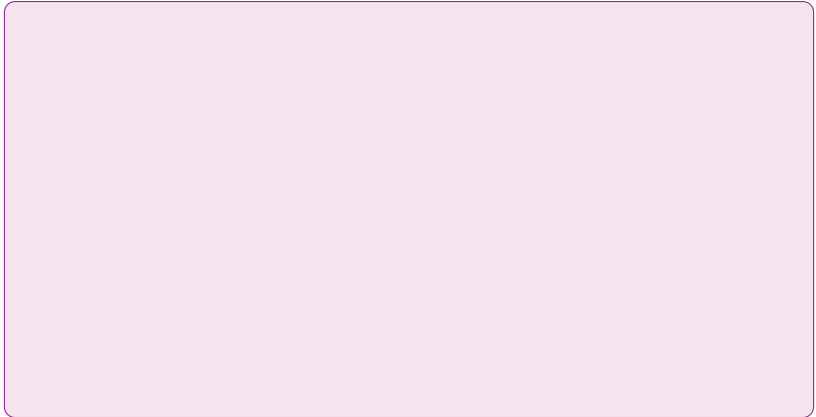
## Tipos abstractos de datos para BFS 2

- El TDA *cola* representa una estructura FIFO con los siguientes métodos:
  - *acolar*( $Q, x$ ): agrega el elemento  $x$  a la cola  $Q$ .
  - *primero*( $Q$ ): devuelve el primer elemento de la cola  $Q$ .
  - *colaVacía*( $Q$ ): devuelve *true* si la cola  $Q$  está vacía y *false* en caso contrario.
  - *desacolar*( $Q$ ): elimina el primer elemento de la cola  $Q$ .

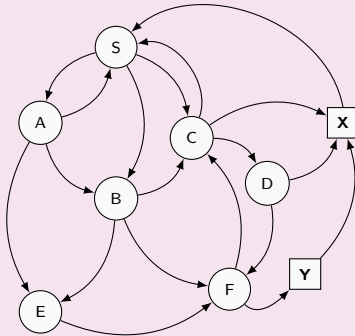
# El algoritmo BFS

```
1.  algoritmo BFS (grafo G, nodo x)
2.      Q.inicializarCola()
3.      visitados.inicializarConjunto()
4.      vecindario.inicializarConjunto()
5.      Q.acolar(n)                                // al principio Q sólo tiene x
6.      visitados.add(x)                            // al principio visitados sólo tiene x
7.      while (!Q.colaVacía()) {
8.          v = Q.primero()
9.          Q.desacolar()
10.         if (objetivo(v))                          // ¡encontrado!
11.             return v
12.         vecindario = G.vecindario(v)              // los vecinos de v
13.         while (!vecindario.conjuntoVacío()) {      // los vecinos de v van a Q
14.             u = vecindario.elegir()
15.             vecindario.eliminar(u)
16.             if (!visitados.pertenece(u)) {
17.                 u = Q.acolar(u)                  // si un elemento va a Q...
18.                 visitados.agregar(u)              // ...debe ir a visitados
19.             }
20.         }
21.     }
22.     return (nil)                                // no encontrado
```

## BFS: un ejemplo



## BFS: un ejemplo



- el nodo activo
- los vecinos del nodo activo
- un nodo objetivo
- nodos visitados

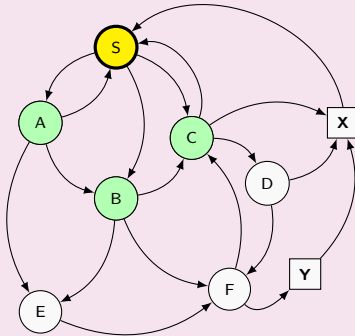
nodo:

visitados:

cola:

vecinos:

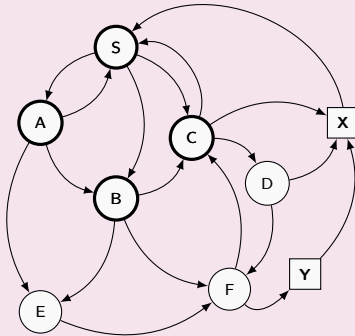
## BFS: un ejemplo



- el nodo activo
- los vecinos del nodo activo
- un nodo objetivo
- nodos visitados

nodo: S  
visitados: {S}  
cola:  $\leftarrow [] \leftarrow$   
vecinos: {A, B, C}

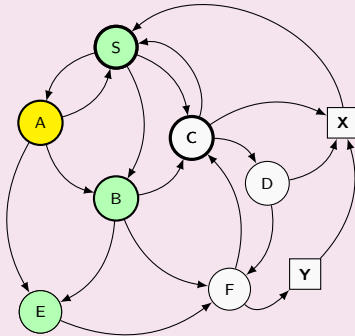
## BFS: un ejemplo



- el nodo activo
- los vecinos del nodo activo
- un nodo objetivo
- nodos visitados

nodo: `nil`  
visitados: `{A, B, C, S}`  
cola: `← [A, B, C] ←`  
vecinos: `∅`

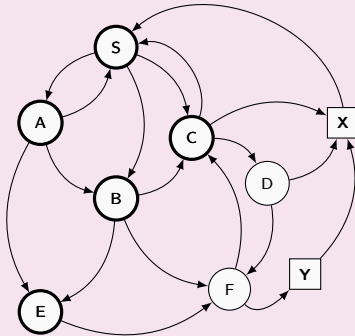
## BFS: un ejemplo



- el nodo activo
- los vecinos del nodo activo
- un nodo objetivo
- nodos visitados

nodo: A  
visitados: {A, B, C, S}  
cola:  $\leftarrow [B, C] \leftarrow$   
vecinos: {B, E, S}

## BFS: un ejemplo

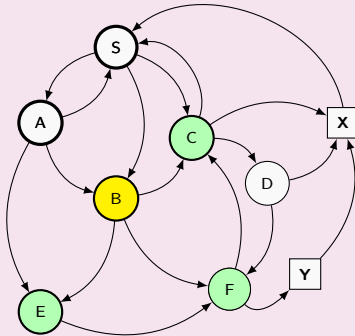


- el nodo activo
- los vecinos del nodo activo
- un nodo objetivo
- nodos visitados

nodo: `nil`  
visitados: `{ A, B, C, E, S }`  
cola: `← [ B, C, E ] ←`  
vecinos: `∅`



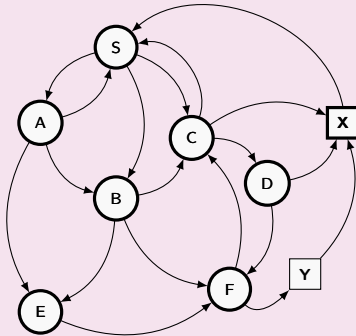
## BFS: un ejemplo



- el nodo activo
- los vecinos del nodo activo
- un nodo objetivo
- nodos visitados

nodo: *B*  
visitados: {A, B, C, E, S}  
cola:  $\leftarrow [C, E] \leftarrow$   
vecinos: {C, E, F}

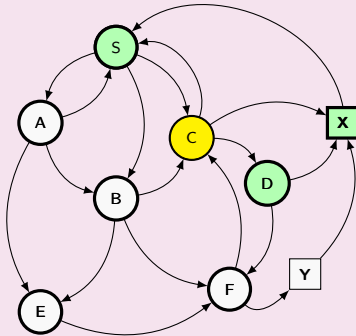
## BFS: un ejemplo



- el nodo activo
- los vecinos del nodo activo
- un nodo objetivo
- nodos visitados

nodo: `nil`  
visitados: `{A, B, C, E, F, S}`  
cola: `← [C, E, F] ←`  
vecinos: `∅`

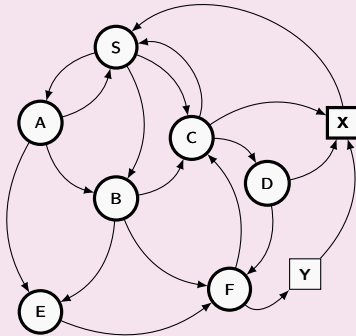
## BFS: un ejemplo



- el nodo activo
- los vecinos del nodo activo
- un nodo objetivo
- nodos visitados

nodo: C  
visitados: {A, B, C, E, F, S}  
cola:  $\leftarrow [E, F] \leftarrow$   
vecinos: {D, S, X}

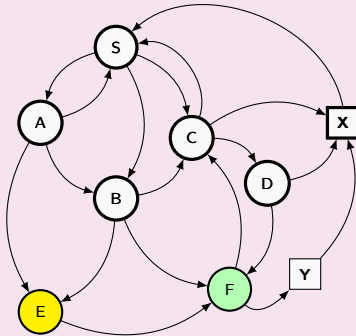
## BFS: un ejemplo



- el nodo activo
- los vecinos del nodo activo
- un nodo objetivo
- nodos visitados

nodo: `nil`  
visitados: `{ A, B, C, D, E, F, S, X }`  
cola: `← [ E, F, D, X ] ←`  
vecinos: `∅`

## BFS: un ejemplo



- el nodo activo
- los vecinos del nodo activo
- un nodo objetivo
- nodos visitados

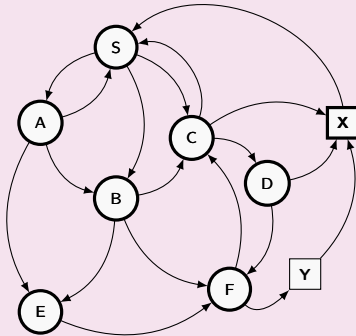
nodo:  $E$

visitados:  $\{A, B, C, D, E, F, S, X\}$

cola:  $\leftarrow [F, D, X] \leftarrow$

vecinos:  $\{F\}$

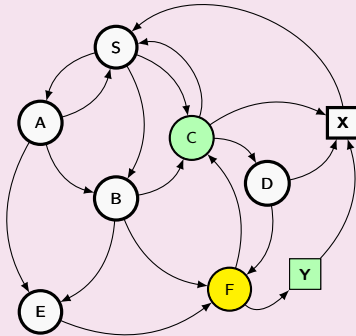
## BFS: un ejemplo



- el nodo activo
- los vecinos del nodo activo
- un nodo objetivo
- nodos visitados

nodo: `nil`  
visitados: `{A, B, C, D, E, F, S, X}`  
cola: `← [F, D, X] ←`  
vecinos: `∅`

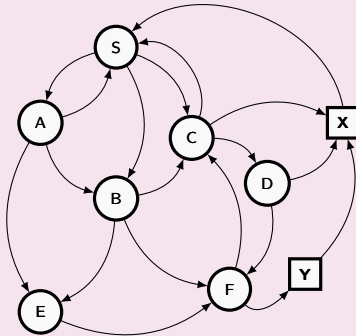
## BFS: un ejemplo



- el nodo activo
- los vecinos del nodo activo
- un nodo objetivo
- nodos visitados

nodo:  $F$   
visitados:  $\{A, B, C, D, E, F, S, X\}$   
cola:  $\leftarrow [D, X] \leftarrow$   
vecinos:  $\{C, Y\}$

## BFS: un ejemplo

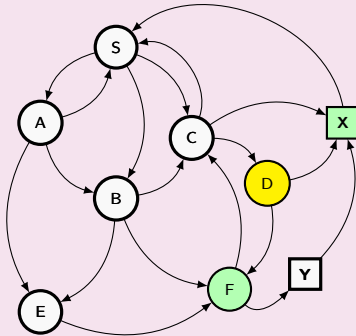


- el nodo activo
- los vecinos del nodo activo
- un nodo objetivo
- □ nodos visitados

nodo: nil  
visitados: {A, B, C, D, E, F, S, X, Y}  
cola: ← [D, X, Y] ←  
vecinos: ∅



## BFS: un ejemplo



- el nodo activo
- los vecinos del nodo activo
- un nodo objetivo
- nodos visitados

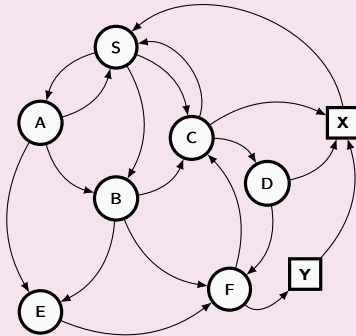
nodo: *D*

visitados: {A, B, C, D, E, F, S, X, Y}

cola:  $\leftarrow [X, Y] \leftarrow$

vecinos: {F, X}

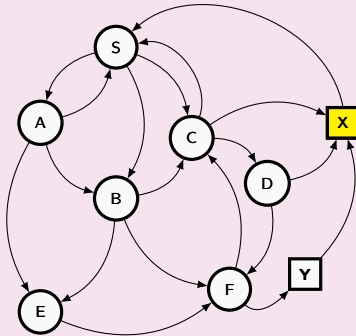
## BFS: un ejemplo



- el nodo activo
- los vecinos del nodo activo
- un nodo objetivo
- nodos visitados

nodo: `nil`  
visitados: `{A, B, C, D, E, F, S, X, Y}`  
cola: `← [X, Y] ←`  
vecinos: `∅`

## BFS: un ejemplo



- el nodo activo
- los vecinos del nodo activo
- un nodo objetivo
- nodos visitados

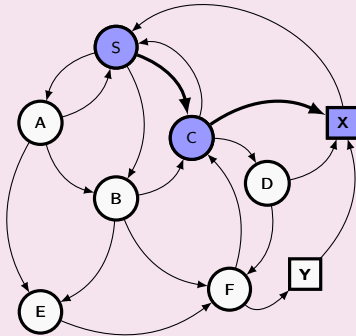
nodo: X (goal)

visitados: {A, B, C, D, E, F, S, X, Y}

cola:  $\leftarrow [Y] \leftarrow$

vecinos: {S}

## BFS: un ejemplo



- el nodo activo
- los vecinos del nodo activo
- un nodo objetivo
- nodos visitados

nodo: X (goal)

visitados: {A, B, C, D, E, F, S, X, Y}

cola:  $\leftarrow [Y] \leftarrow$

vecinos: {S}

- 1 Repaso de la clase anterior
  - El problema de las  $n$  damas
  - Suma de subconjunto
- 2 Otros ejemplos de *backtracking*
  - Partición en partes iguales
  - Búsqueda en profundidad
- 3 **Búsqueda en grafos**
  - Búsqueda en amplitud (BFS)
  - **DFS no recursivo**
  - Uniform Cost Search (UCS)
  - El algoritmo A\*
- 4 Ejercicios propuestos

# Por el mismo precio: una versión no recursiva de DFS

## Por el mismo precio: una versión no recursiva de DFS

- Puede implementarse DFS de una manera casi idéntica a BFS. Sólo es necesario reemplazar la cola con una pila.

## Por el mismo precio: una versión no recursiva de DFS

- Puede implementarse DFS de una manera casi idéntica a BFS. Sólo es necesario reemplazar la cola con una pila.
- Se define un TDA *pila* que representa una estructura LIFO con los siguientes métodos:
  - *apilar*( $S, x$ ): agrega el elemento  $x$  a la pila  $S$ .
  - *tope*( $S$ ): devuelve el elemento en el tope de la pila  $S$ .
  - *desapilar*( $S$ ): elimina el elemento en el tope de la pila  $S$ .
  - *pilaVacía*( $S$ ): devuelve `true` si la pila  $S$  está vacía y `false` en caso contrario.



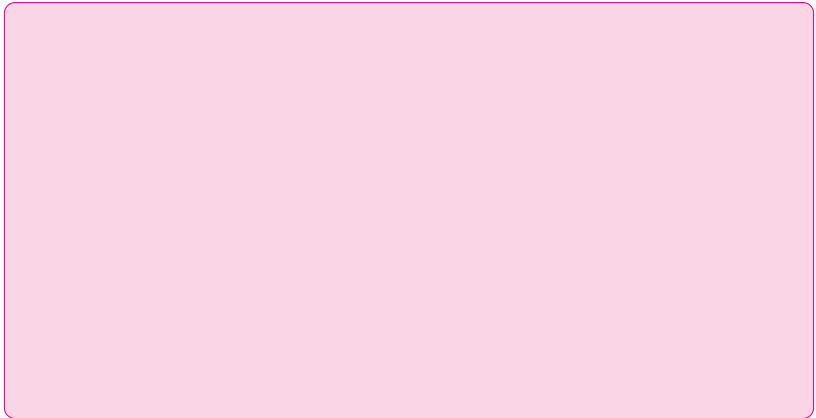
## Por el mismo precio: una versión no recursiva de DFS

- Puede implementarse DFS de una manera casi idéntica a BFS. Sólo es necesario reemplazar la cola con una pila.
- Se define un TDA *pila* que representa una estructura LIFO con los siguientes métodos:
  - *apilar*( $S, x$ ): agrega el elemento  $x$  a la pila  $S$ .
  - *tope*( $S$ ): devuelve el elemento en el tope de la pila  $S$ .
  - *desapilar*( $S$ ): elimina el elemento en el tope de la pila  $S$ .
  - *pilaVacía*( $S$ ): devuelve `true` si la pila  $S$  está vacía y `false` en caso contrario.
- La implementación no recursiva de DFS se muestra en la siguiente diapositiva.

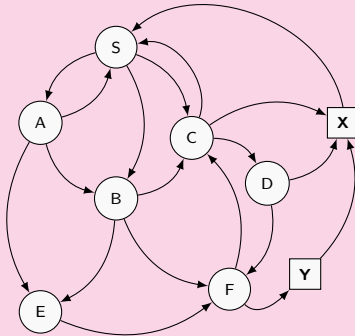
## El algoritmo DFS no recursivo

```
1.  algoritmo DFS (grafo G, nodo x)
2.      S.inicializarPila()
3.      visitados.inicializarConjunto()
4.      vecindario.inicializarConjunto()
5.      S.apilar(n)                                // al principio S sólo tiene x
6.      visitados.add(x)                            // al principio visitados sólo tiene x
7.      while (!S.pilaVacía()) {
8.          v = S.tope()
9.          S.desapilar()
10.         if (objetivo(v))                          // ¡encontrado!
11.             return v
12.         vecindario = G.vecindario(v)              // los vecinos de v
13.         while (!vecindario.conjuntoVacio()) {      // los vecinos de v van a S
14.             u = vecindario.elegir()
15.             vecindario.eliminar(u)
16.             if (!visitados.pertenece(u)) {
17.                 u = Q.apilar(u)                    // si un elemento va a Q...
18.                 visitados.agregar(u)              // ...debe ir a visitados
19.             }
20.         }
21.     }
22.     return (nil)                                // no encontrado
```

## DFS no recursivo: un ejemplo



## DFS no recursivo: un ejemplo



- el nodo activo
- los vecinos del nodo activo
- los nodos objetivos
- nodos visitados

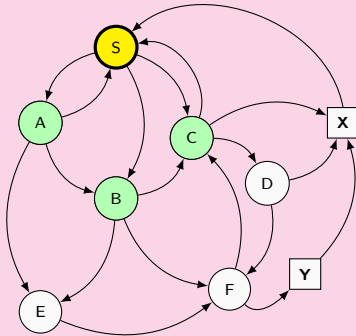
nodo:

visitados:

pila:

vecinos:

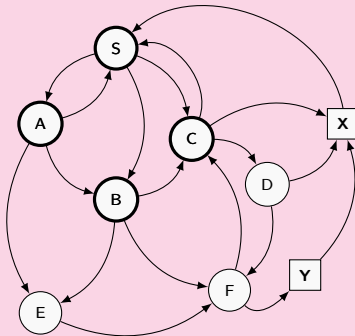
## DFS no recursivo: un ejemplo



- el nodo activo
- los vecinos del nodo activo
- los nodos objetivos
- □ nodos visitados

nodo: S  
visitados: {S}  
pila:  $\leftrightarrow$  []  
vecinos: {A, B, C}

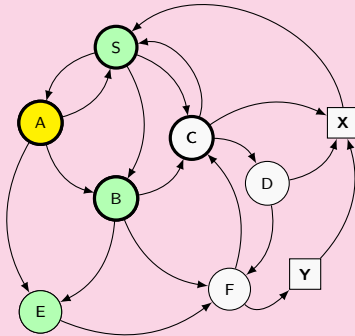
## DFS no recursivo: un ejemplo



- el nodo activo
- los vecinos del nodo activo
- los nodos objetivos
- nodos visitados

nodo: `nil`  
visitados: `{ A, B, C, S }`  
pila: `↔ [A, B, C]`  
vecinos: `∅`

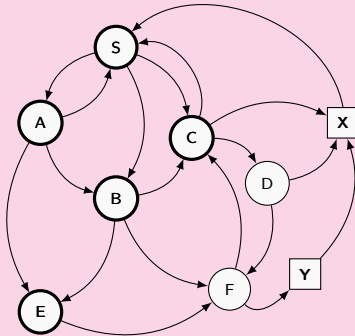
## DFS no recursivo: un ejemplo



- el nodo activo
- los vecinos del nodo activo
- los nodos objetivos
- □ nodos visitados

nodo: A  
visitados: {A, B, C, S}  
pila:  $\leftrightarrow$  [B, C]  
vecinos: {B, E, S}

## DFS no recursivo: un ejemplo

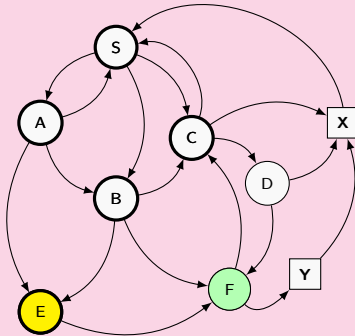


- el nodo activo
- los vecinos del nodo activo
- los nodos objetivos
- nodos visitados

nodo: `nil`  
visitados: `{A, B, C, E, S}`  
pila: `↔ [E, B, C]`  
vecinos: `∅`



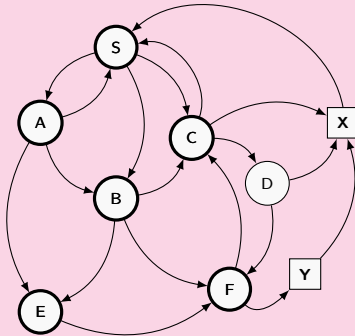
## DFS no recursivo: un ejemplo



- el nodo activo
- los vecinos del nodo activo
- los nodos objetivos
- nodos visitados

nodo: *E*  
visitados: {A, B, C, E, S}  
pila:  $\leftrightarrow$  [B, C]  
vecinos: {F}

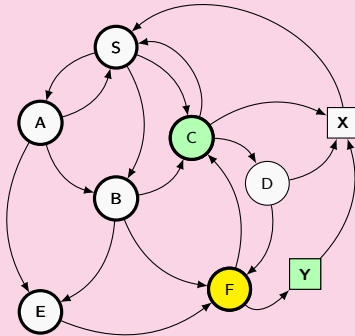
## DFS no recursivo: un ejemplo



- el nodo activo
- los vecinos del nodo activo
- los nodos objetivos
- nodos visitados

nodo: `nil`  
visitados: `{ A, B, C, E, F, S }`  
pila: `↔ [ F, B, C ]`  
vecinos: `∅`

## DFS no recursivo: un ejemplo



- el nodo activo
- los vecinos del nodo activo
- los nodos objetivos
- □ nodos visitados

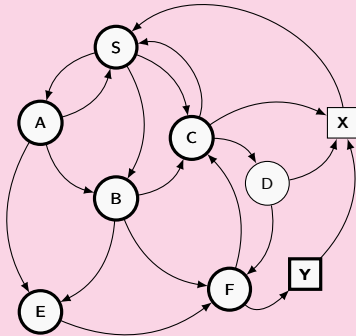
nodo:  $F$

visitados:  $\{A, B, C, E, F, S\}$

pila:  $\leftarrow [B, C] \leftarrow$

vecinos:  $\{C, Y\}$

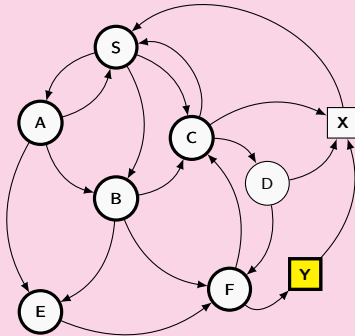
## DFS no recursivo: un ejemplo



- el nodo activo
- los vecinos del nodo activo
- los nodos objetivos
- nodos visitados

nodo: `nil`  
visitados: `{A, B, C, E, F, S, Y}`  
pila: `↔ [Y, B, C]`  
vecinos: `∅`

## DFS no recursivo: un ejemplo



- el nodo activo
- los vecinos del nodo activo
- los nodos objetivos
- nodos visitados

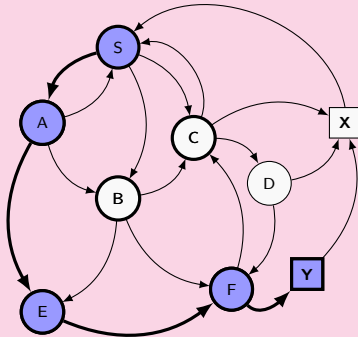
nodo: Y (goal)

visitados: {A, B, C, E, F, S, Y}

pila:  $\leftarrow [B, C] \leftarrow$

vecinos: {X}

## DFS no recursivo: un ejemplo



- el nodo activo
- los vecinos del nodo activo
- los nodos objetivos
- nodos visitados

nodo:

visitados:

pila:

vecinos:

- 1 Repaso de la clase anterior
  - El problema de las  $n$  damas
  - Suma de subconjunto
- 2 Otros ejemplos de *backtracking*
  - Partición en partes iguales
  - Búsqueda en profundidad
- 3 **Búsqueda en grafos**
  - Búsqueda en amplitud (BFS)
  - DFS no recursivo
  - **Uniform Cost Search (UCS)**
  - El algoritmo A\*
- 4 Ejercicios propuestos

# El algoritmo UCS (*Uniform Cost Search*)



## El algoritmo UCS (*Uniform Cost Search*)

- Es un algoritmo eficiente (entre los que no usan heurísticas.)

## El algoritmo UCS (*Uniform Cost Search*)

- Es un algoritmo eficiente (entre los que no usan heurísticas.)
- Los vértices se colocan en una cola con prioridad, donde la prioridad está dada por el costo acumulado para alcanzar el vértice desde la raíz.

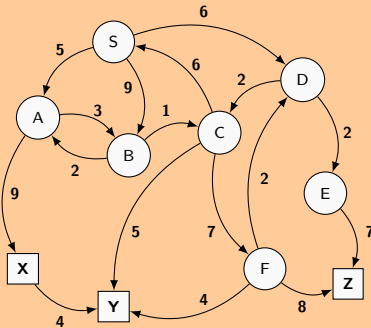
## El algoritmo UCS (*Uniform Cost Search*)

- Es un algoritmo eficiente (entre los que no usan heurísticas.)
- Los vértices se colocan en una cola con prioridad, donde la prioridad está dada por el costo acumulado para alcanzar el vértice desde la raíz.
- El vértice con el costo acumulado mínimo tiene la prioridad máxima.

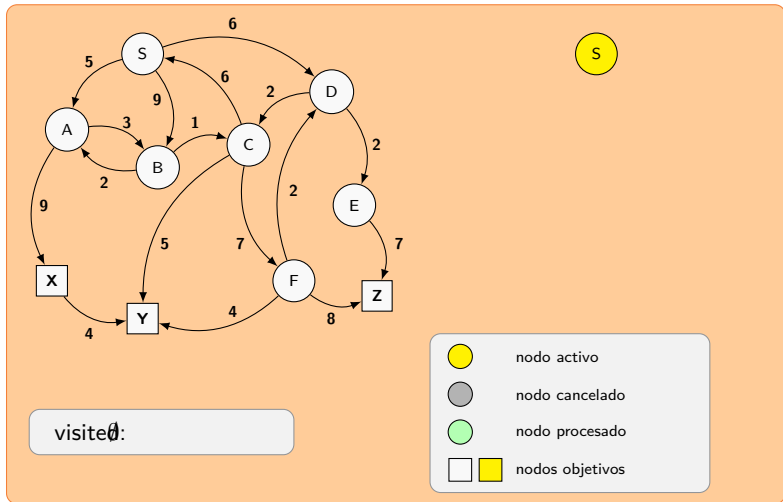
## UCS: un ejemplo



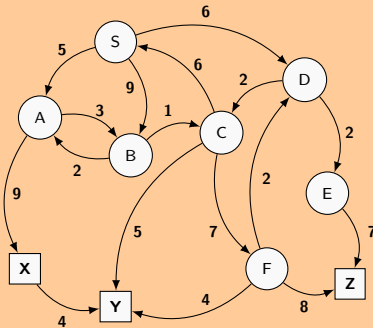
## UCS: un ejemplo



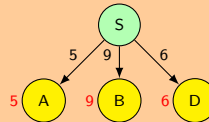
## UCS: un ejemplo








## UCS: un ejemplo

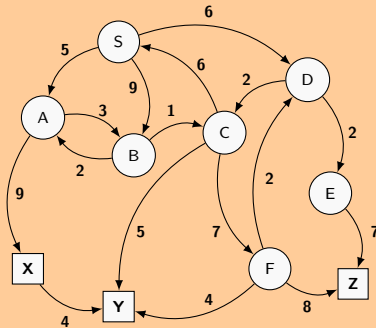


visited{S}

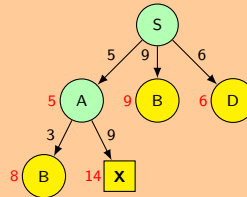







-  nodo activo
-  nodo cancelado
-  nodo procesado
-   nodos objetivos

## UCS: un ejemplo



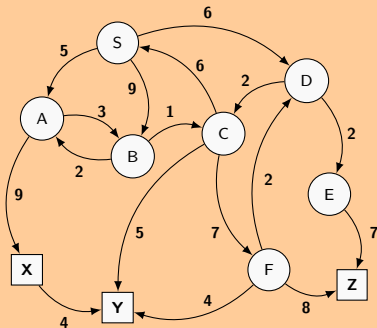
visited {A, S}



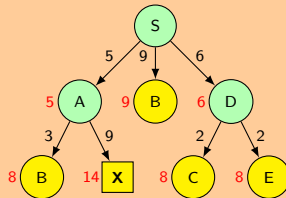
-  nodo activo
-  nodo cancelado
-  nodo procesado
-   nodos objetivos



## UCS: un ejemplo

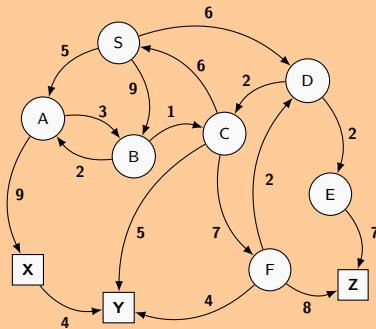


visited {A, D, S}

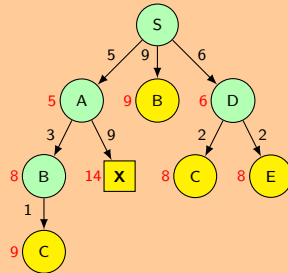


- nodo activo
- nodo cancelado
- nodo procesado
- nodos objetivos

## UCS: un ejemplo

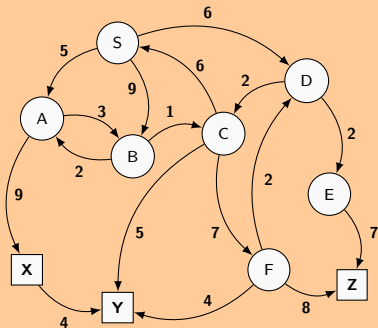


visited: {A, B, D, S}

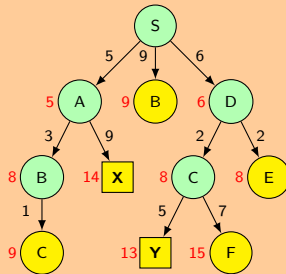


- nodo activo
- nodo cancelado
- nodo procesado
- nodos objetivos

## UCS: un ejemplo

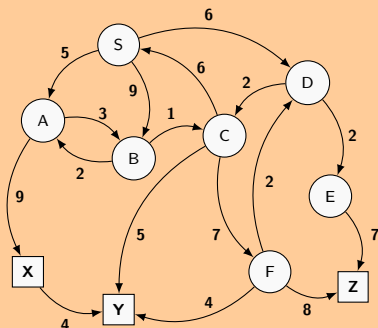


visited: {A, B, C, D, S}

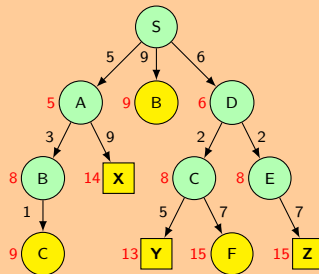


- nodo activo
- nodo cancelado
- nodo procesado
- nodos objetivos

# UCS: un ejemplo

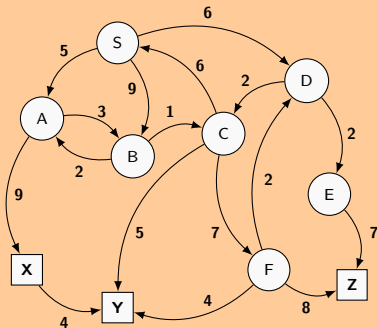


visited: {A, B, C, D, E, S}

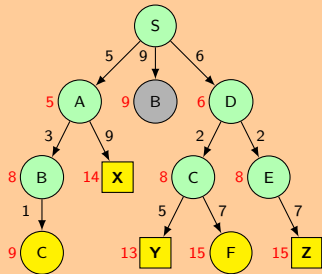


- nodo activo
- nodo cancelado
- nodo procesado
- nodos objetivos

## UCS: un ejemplo

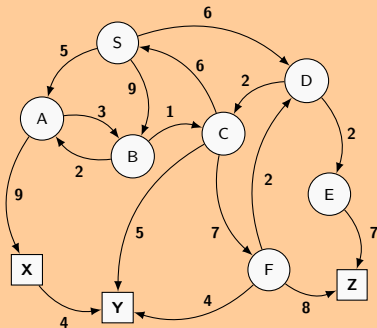


visited: {A, B, C, D, E, S}

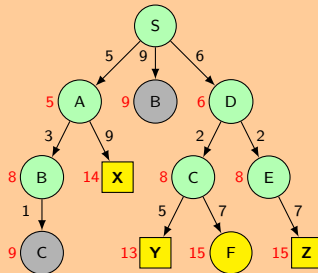







- nodo activo
- nodo cancelado
- nodo procesado
- nodos objetivos

## UCS: un ejemplo

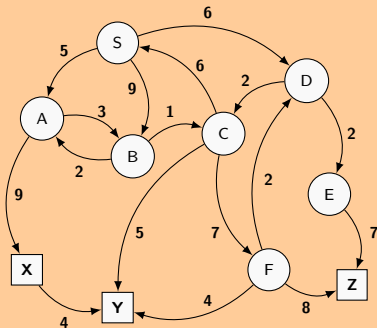


visited: {A, B, C, D, E, S}

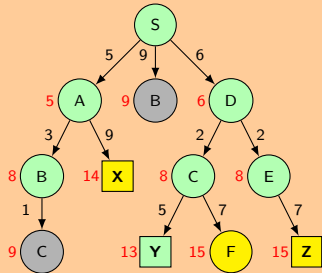


-  nodo activo
-  nodo cancelado
-  nodo procesado
-   nodos objetivos

## UCS: un ejemplo

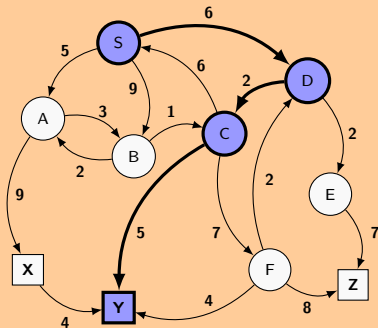


visited: {A, B, C, D, E, S}

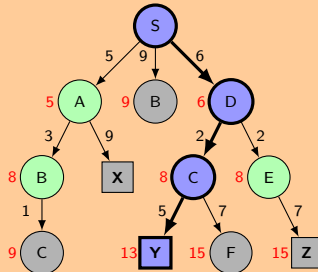


- nodo activo
- nodo cancelado
- nodo procesado
- nodos objetivos

## UCS: un ejemplo



visited {A, B, C, D, E, S}



- nodo activo
- nodo cancelado
- nodo procesado
- nodos objetivos



- 1 Repaso de la clase anterior
  - El problema de las  $n$  damas
  - Suma de subconjunto
- 2 Otros ejemplos de *backtracking*
  - Partición en partes iguales
  - Búsqueda en profundidad
- 3 **Búsqueda en grafos**
  - Búsqueda en amplitud (BFS)
  - DFS no recursivo
  - Uniform Cost Search (UCS)
  - El algoritmo A\*
- 4 Ejercicios propuestos

# Búsquedas heurísticas

# Búsquedas heurísticas

- Las búsquedas heurísticas constituyen una estrategia de búsqueda que trata de optimizar una solución basándose en una función heurística dada, generalmente una medida de costo.

# Búsquedas heurísticas

- Las búsquedas heurísticas constituyen una estrategia de búsqueda que trata de optimizar una solución basándose en una función heurística dada, generalmente una medida de costo.
- En este caso, la heurística estará dada por una estimación rápida del costo de alcanzar un nodo objetivo desde cada nodo.

# Búsquedas heurísticas

- Las búsquedas heurísticas constituyen una estrategia de búsqueda que trata de optimizar una solución basándose en una función heurística dada, generalmente una medida de costo.
- En este caso, la heurística estará dada por una estimación rápida del costo de alcanzar un nodo objetivo desde cada nodo.
- El algoritmo A\* asigna prioridades a los nodos basándose en el costo mínimo que resulta de la suma del costo incurrido para llegar al nodo desde la raíz y el costo estimado (la heurística) del nodo.

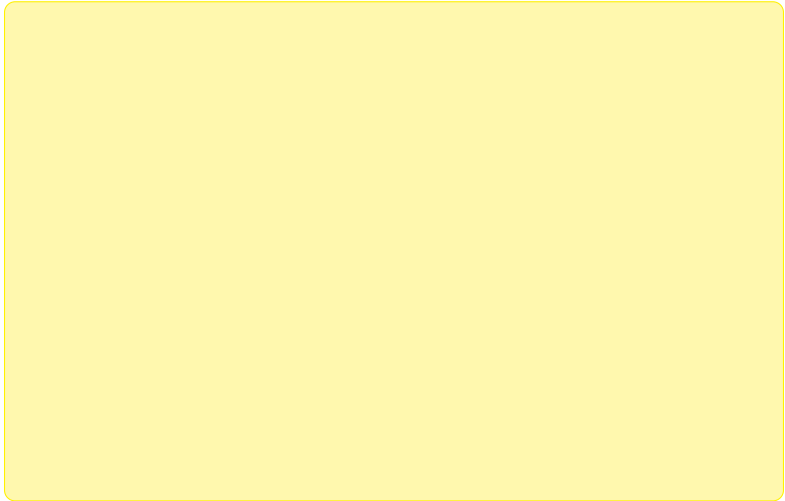
# Búsquedas heurísticas

- Las búsquedas heurísticas constituyen una estrategia de búsqueda que trata de optimizar una solución basándose en una función heurística dada, generalmente una medida de costo.
- En este caso, la heurística estará dada por una estimación rápida del costo de alcanzar un nodo objetivo desde cada nodo.
- El algoritmo A\* asigna prioridades a los nodos basándose en el costo mínimo que resulta de la suma del costo incurrido para llegar al nodo desde la raíz y el costo estimado (la heurística) del nodo.
- La heurística no debe ser ni pesimista ni demasiado optimista.

# Búsquedas heurísticas

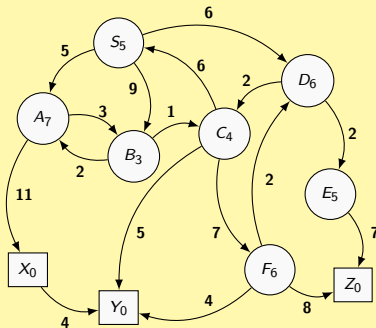
- Las búsquedas heurísticas constituyen una estrategia de búsqueda que trata de optimizar una solución basándose en una función heurística dada, generalmente una medida de costo.
- En este caso, la heurística estará dada por una estimación rápida del costo de alcanzar un nodo objetivo desde cada nodo.
- El algoritmo A\* asigna prioridades a los nodos basándose en el costo mínimo que resulta de la suma del costo incurrido para llegar al nodo desde la raíz y el costo estimado (la heurística) del nodo.
- La heurística no debe ser ni pesimista ni demasiado optimista.
- Veremos dos ejemplos, uno con una heurística mediocre y otro con una buena.

# El algoritmo de búsqueda heurística A\*

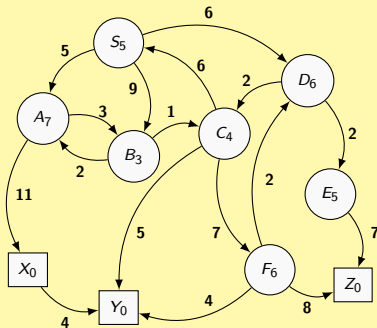




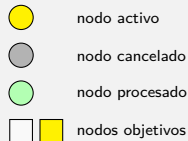
# El algoritmo de búsqueda heurística A\*



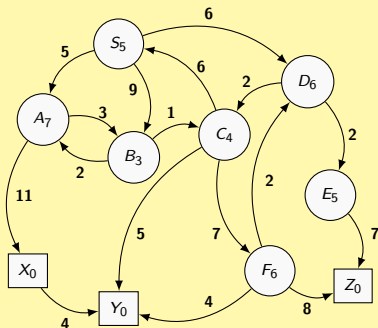
# El algoritmo de búsqueda heurística A\*



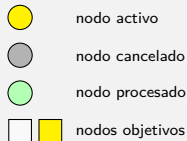
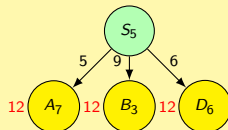
visited:  $\emptyset$



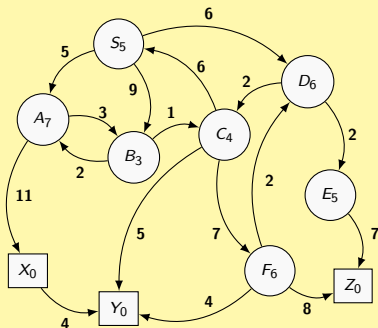
# El algoritmo de búsqueda heurística A\*



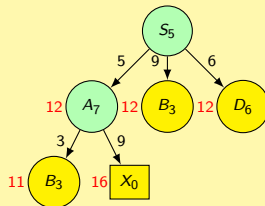
visited: {S[5]}



# El algoritmo de búsqueda heurística A\*

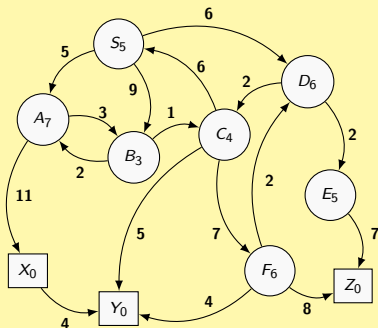


visited: {A[12], S[5]}

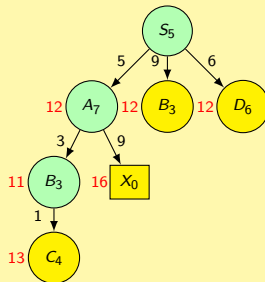


- nodo activo
- nodo cancelado
- nodo procesado
- nodos objetivos

## El algoritmo de búsqueda heurística A\*

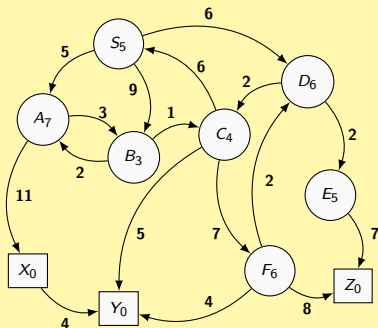


visited: {A[12], B[11], S[5]}

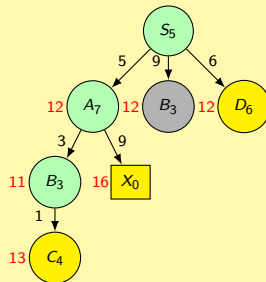


- nodo activo
- nodo cancelado
- nodo procesado
- nodos objetivos

# El algoritmo de búsqueda heurística A\*

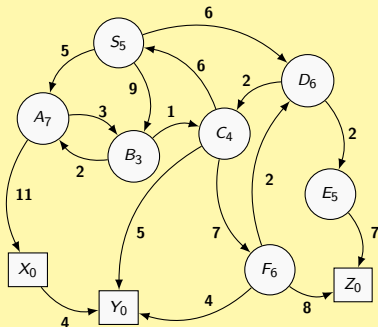


visited: {A[12], B[11], S[5]}

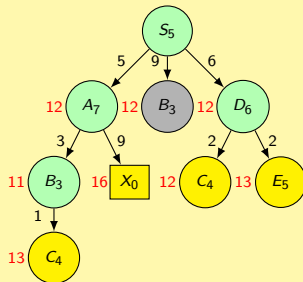


- nodo activo
- nodo cancelado
- nodo procesado
- nodos objetivos

# El algoritmo de búsqueda heurística A\*

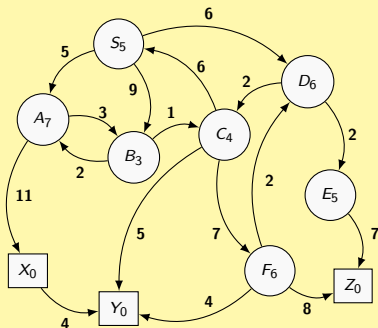


visited: {A[12], B[11], D[12], S[5]}

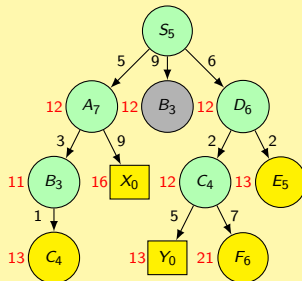


- nodo activo
- nodo cancelado
- nodo procesado
- nodos objetivos

# El algoritmo de búsqueda heurística A\*



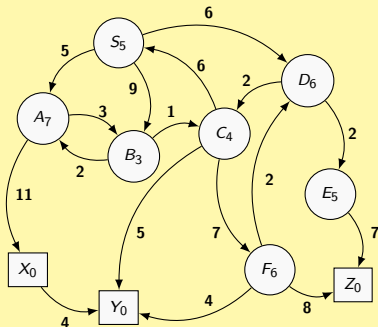
visited: { A[12], B[11], C[12], D[12], S[5] }



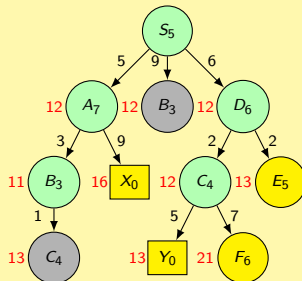
- nodo activo
- nodo cancelado
- nodo procesado
- nodos objetivos



# El algoritmo de búsqueda heurística A\*

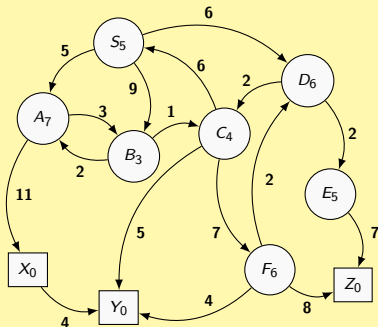


visited: { A[12], B[11], C[12], D[12], S[5] }

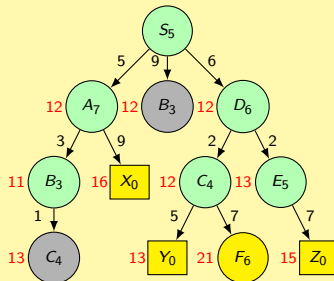


- nodo activo
- nodo cancelado
- nodo procesado
- nodos objetivos

# El algoritmo de búsqueda heurística A\*

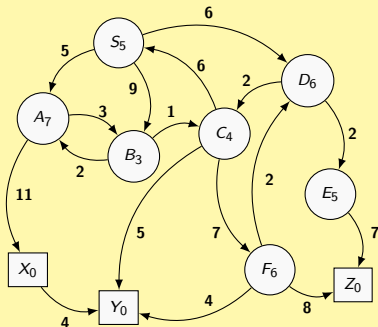


visited: {A[12], B[11], C[12], D[12], E[13], S[5]}

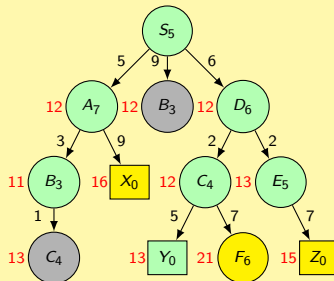


- nodo activo
- nodo cancelado
- nodo procesado
- nodos objetivos

# El algoritmo de búsqueda heurística A\*

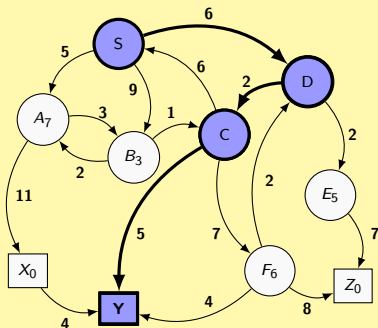


visited: { A[12], B[11], C[12], D[12], E[13], S[5] }

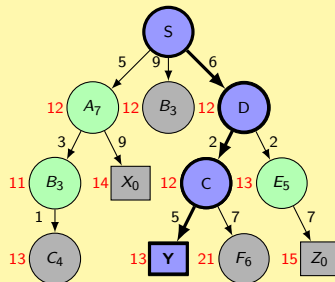


- nodo activo
- nodo cancelado
- nodo procesado
- nodos objetivos

# El algoritmo de búsqueda heurística A\*

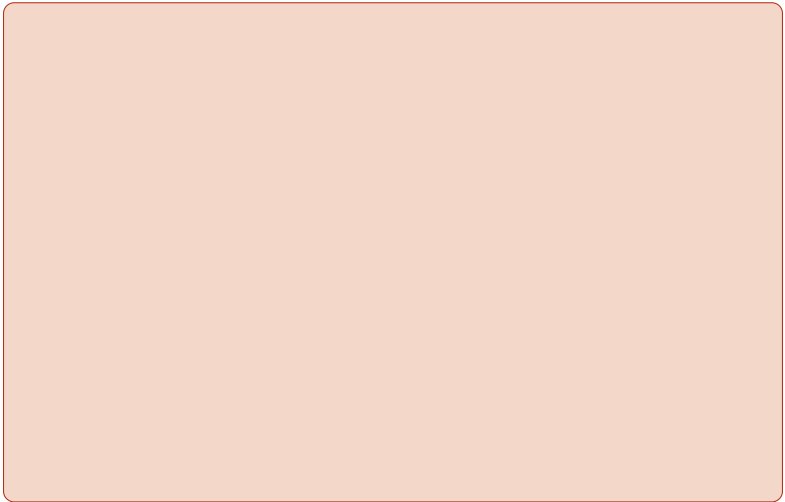


visited: {A[12], B[11], C[12], D[12], E[13], S[5]}

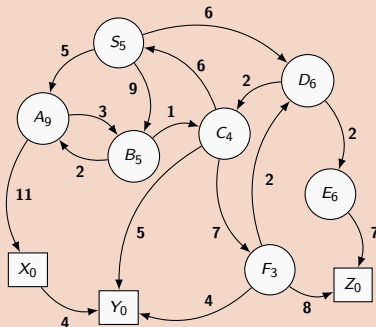


- nodo activo
- nodo cancelado
- nodo procesado
- nodos objetivos

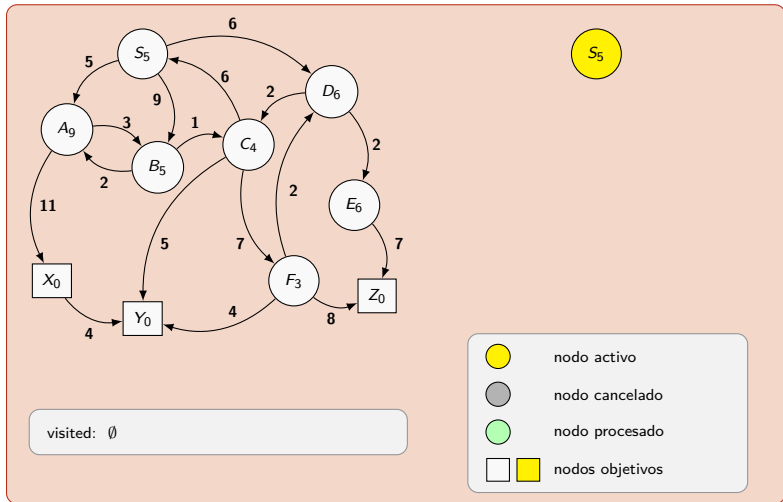
## The Same Example With a Better Heuristics



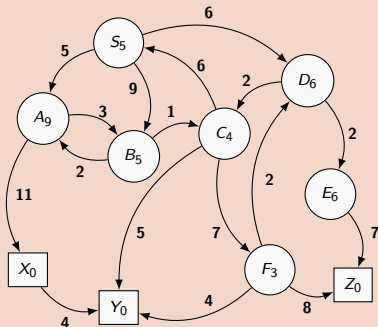
## The Same Example With a Better Heuristics



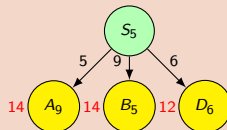
# The Same Example With a Better Heuristics








## The Same Example With a Better Heuristics



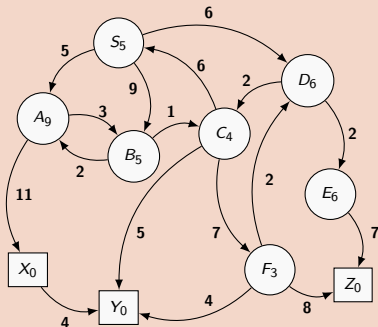
visited: {S[5]}



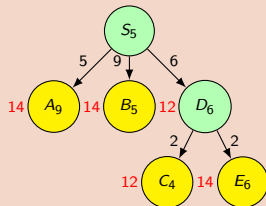
-  nodo activo
-  nodo cancelado
-  nodo procesado
-   nodos objetivos



# The Same Example With a Better Heuristics

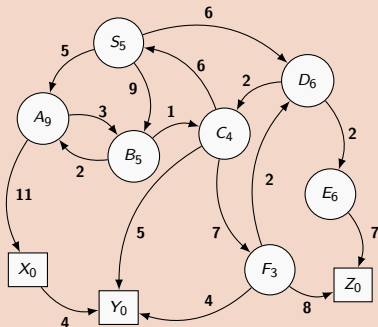


visited: {  $D[12]$ ,  $S[5]$  }

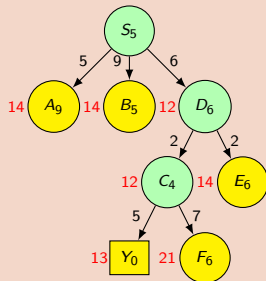


- nodo activo
- nodo cancelado
- nodo procesado
- nodos objetivos

# The Same Example With a Better Heuristics

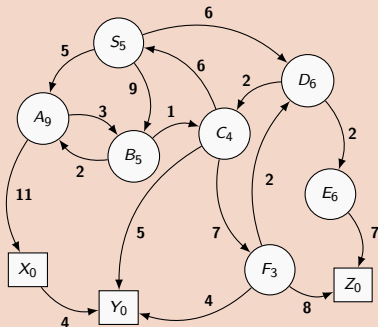


visited: { C[12], D[12], S[5] }

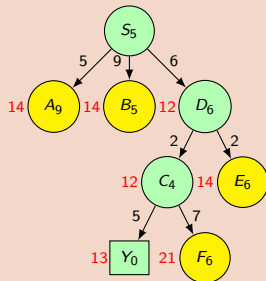


- nodo activo
- nodo cancelado
- nodo procesado
- nodos objetivos

## The Same Example With a Better Heuristics

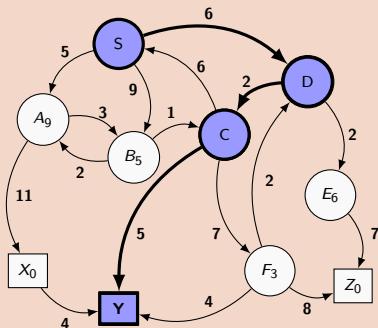


visited: { C[12], D[12], S[5], Y[13] }

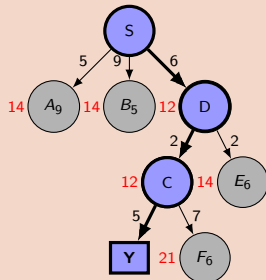







- nodo activo
- nodo cancelado
- nodo procesado
- nodos objetivos

# The Same Example With a Better Heuristics



visited: { C[12], D[12], S[5], Y[13] }



-  nodo activo
-  nodo cancelado
-  nodo procesado
-   nodos objetivos

## Ejercicios propuestos 1

- 1 Considere 5 objetos con pesos  $(w_1, w_2, \dots, w_5) = (10, 3, 5, 7, 2)$  y una mochila de tamaño  $m = 15$ . Use una estrategia de backtracking para encontrar todas las combinaciones de objetos que caben exactamente en la mochila. Calcule la complejidad de este método.
- 2 Dadas  $n = 4$  variables booleanas  $x_j$  sujetas a las restricciones de que debe ser  $x_1 \neq x_2$  y el número de variables con valor 1 debe ser impar:
  - a. Dibuje el árbol de permutaciones de este problema para un algoritmo de backtracking.
  - b. Encuentre las soluciones a este problema.
  - c. ¿Cuántos nodos se generan? ¿Cuántos sobreviven y cuántos son eliminados??
  - d. ¿Cuál es la eficiencia del método comparado con un algoritmo de fuerza bruta? O, si lo prefiere, ¿Cuántos nodos se generan con este algoritmo y cuántos se generarían con un método de pura fuerza bruta?

## Ejercicios propuestos 2

- 3 Escriba un algoritmo de backtracking con poda para encontrar todas las permutaciones de los cuatro caracteres (*A, B, C, D*) de manera que satisfagan las siguientes restricciones:
- Restricción (1): no debe haber repeticiones.
  - Restricción (2): la diferencia en el valor absoluto del código ASCII entre cada carácter y el previo debe ser siempre mayor a 2.

Muestre el árbol de permutaciones para encontrar todas las que satisfagan las restricciones. ¿Cuántas respuestas se obtienen? ¿Cuántas se obtendrían si sólo se tuviera la restricción (1)?

- 4 Implemente un algoritmo de backtracking para encontrar todas las permutaciones de los cuatro dígitos (*1, 2, 3, 4*) de manera que ningún dígito quede repetido y que el valor absoluto de la diferencia entre un dígito cualquiera y su precedente sea a menor o igual a 2. Simule el funcionamiento de su algoritmo mostrando el árbol de las permutaciones hasta encontrar la primera permutación que satisface las restricciones.

## Suggested Exercises 3

- 5 El problema de la *coloración de un grafo* consiste en asignar colores a los vértices de un grafo de manera que no haya dos vértices adyacentes con el mismo color. Considere el grafo no dirigido  $G = (V, E)$  con  $V = \{A, B, C, D, E\}$  y  $E = \{(A, B), (A, E), (C, D), (D, E)\}$ . Dibuje el árbol de permutaciones que representa un algoritmo de backtracking para encontrar una posible coloración de este grafo usando sólo dos colores (por ejemplo, rojo y negro.)
- 6 Un *ciclo Hamiltoniano* es un camino cerrado que visita todos los vértices de un grafo exactamente una vez. Aplique backtracking al problema de encontrar un ciclo Hamiltoniano al siguiente grafo.

