# Tema 1. Búsqueda en Espacios de Estados

Algoritmos de Búsqueda no Informada en árboles

# Objetivos

1. Conocer los fundamentos de los algoritmos de búsqueda y el papel que juegan en la Inteligencia Artificial
2. **Conocer el paradigma de Búsqueda en Espacios de Estados y los algoritmos básicos de búsqueda a ciegas** y sobre todo de búsqueda inteligente o heurística
3. Saber cómo modelar problemas para resolverlos con Búsqueda en Espacios de Estados, en particular cómo introducir conocimiento específico del dominio del problema

# Contenidos

1. Introducción
2. Espacios de búsqueda
3. **Algoritmos de búsqueda no informada**
    1. **Generalidades sobre los algoritmos de búsqueda**
    2. **Algoritmos de búsqueda en árboles**
    3. Algoritmos de búsqueda en grafos
4. Algoritmos de búsqueda informada o heurística
5. Técnicas de diseño de funciones heurísticas

# 1.3. Algoritmos de Búsqueda no Informada
## *Generalidades de los algoritmos de búsqueda*

- Dado un Espacio de Búsqueda, un Algoritmo de Búsqueda calcula un camino desde el estado inicial a uno de los estados objetivo.
- Para ello, el algoritmo desarrolla un "árbol de búsqueda" que representa distintos "caminos abiertos" desde el estado inicial a varios estados intermedios
- En cada paso (iteración si se trata de un algoritmo iterativo), el algoritmo expande una hoja del árbol de búsqueda elegida de acuerdo con la "estrategia de control"

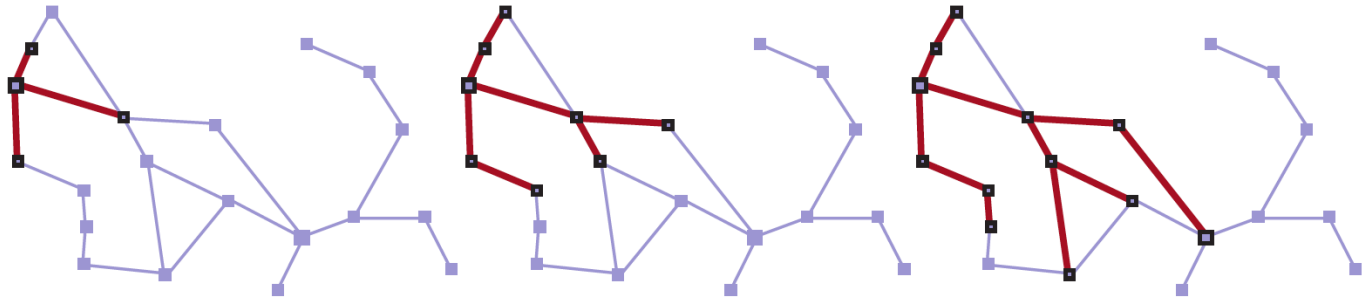
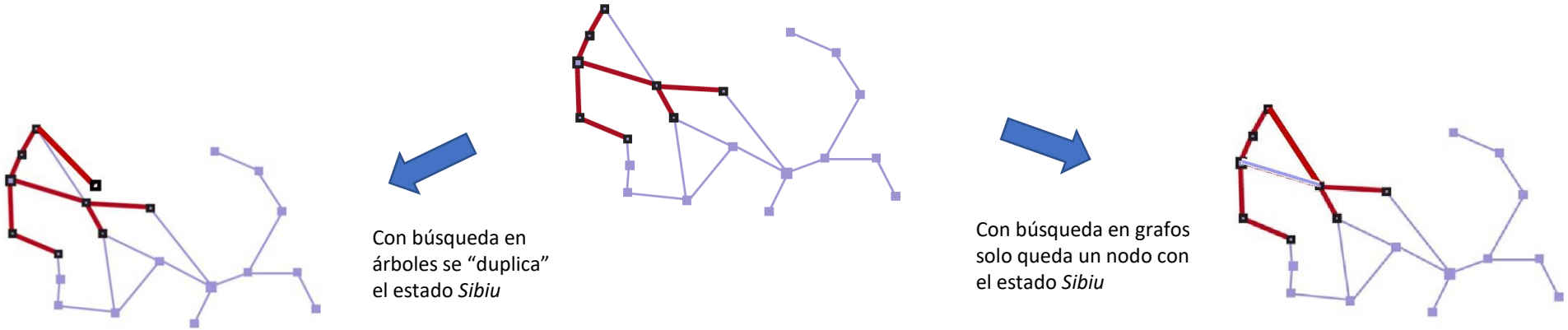


**Figure 3.5** A sequence of search trees generated by a graph search on the Romania problem

# *Generalidades de los algoritmos de búsqueda (I)*

- Un nodo del árbol de búsqueda es un objeto que contiene
  - Un estado del espacio de búsqueda
  - Un enlace al nodo padre en el árbol de búsqueda
  - Otras informaciones que puedan ser de utilidad para el algoritmo de búsqueda como la profundidad del nodo o el coste del inicial al nodo
- Distinguimos dos tipos de búsqueda
  - Búsqueda en grafos: el árbol de búsqueda no tiene nodos con estados repetidos
  - Búsqueda en árboles: el árbol de búsqueda puede tener estados repetidos en los nodos
  - Ejemplo: al expandir el nodo *Oradea* tendríamos lo siguiente



Con búsqueda en árboles se "duplica" el estado *Sibiu*

Con búsqueda en grafos solo queda un nodo con el estado *Sibiu*

# *Generalidades de los algoritmos de búsqueda (II)*

- Los algoritmos pueden ser iterativos o recursivos
  - *Iterativos*: búsqueda en grafos o en árboles
  - *Recursivos*: búsqueda en árboles (backtracking)

- Propiedades de los algoritmos de búsqueda
  - *Corrección*: las soluciones que encuentran son correctas
  - *Completitud*: si el problema tiene solución, el algoritmo la encuentra (aunque no sea óptima)
  - *Admisibilidad*: siempre encuentra una solución óptima (siempre y cuando disponga de tiempo y espacio suficiente)
  - *Coste Computacional*: viene dado por el tamaño del árbol de búsqueda y el tiempo necesario para llegar a una solución

# 2.3.2. Algoritmos de Búsqueda en Árboles

- Algoritmo General: *Best First Search* (*BF*)
  - Se puede adaptar para búsqueda en árboles y en grafos. Y para búsqueda a ciegas y búsqueda inteligente
  - Es iterativo: en cada iteración desarrolla un nodo hoja del árbol de búsqueda.
  - Casos particulares de búsqueda a ciegas:
    - *Búsqueda en Anchura,*
    - *Búsqueda en Profundidad,*
    - *Coste Uniforme.*
- Otros algoritmos de búsqueda en árboles
  - *Búsqueda en profundidad limitada.*
  - *Búsqueda en profundidad iterativa.*
  - *Backtracking* (recursivo).

# *Algoritmo Best First Search (BFS) para búsqueda a ciegas en árboles*
*[Russell&Norvig, 2022]*

```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element
  [redacted]
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      [redacted]
        add child to frontier
  return failure
```

- f:Nodos $\rightarrow R^+$, es la función de evaluación (en búsqueda a ciegas no utiliza información específica del problema, es decir de los estados)
- **frontier** es una lista ordenada por los valores de f() de menor a mayor
- Un objeto NODE incluye un enlace al nodo padre, el PATH-COST (coste desde el INICIAL al NODE ) y un STATE (estado del espacio de búsqueda). No incluye el valor de f() ni enlaces a los sucesores
- El test **Is-Goal()** se hace al expandir un nodo y no al crearlo

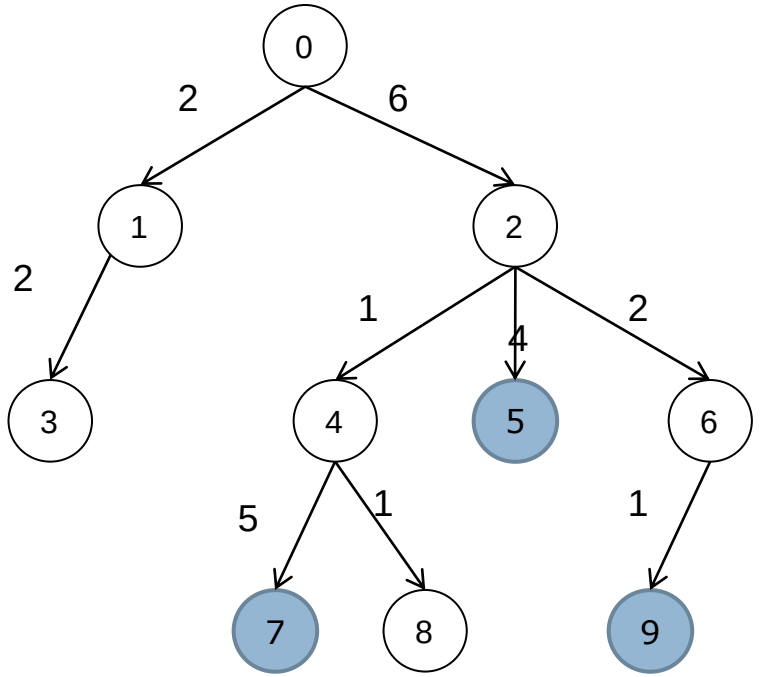
# *Algoritmo Best First Search (BFS)*
## *Expansión de nodos*

```
function EXPAND(problem, node) yields nodes
  s ← node.STATE
  for each action in problem.ACTIONS(s) do
    s' ← problem.RESULT(s, action)
    cost ← node.PATH-COST + problem.ACTION-COST(s, action, s')
    yield NODE(STATE=s', PARENT=node, ACTION=action, PATH-COST=cost)
```

- Para cada estado sucesor en el espacio de búsqueda al estado del nodo (*node.STATE*), se genera un nodo sucesor candidato a ser incluido en el árbol de búsqueda
  - *El padre del nodo sucesor es el nodo que se expande*
  - *El coste del sucesor al INICIAL es el coste del INICIAL al nodo que se expande más el coste de la acción que lleva del nodo que se expande al sucesor*
- En el proceso de expansión, se puede prescindir de un sucesor si el estado correspondiente es el estado del padre del nodo que se está expandiendo. Así, se evitan muchas ramas infinitas

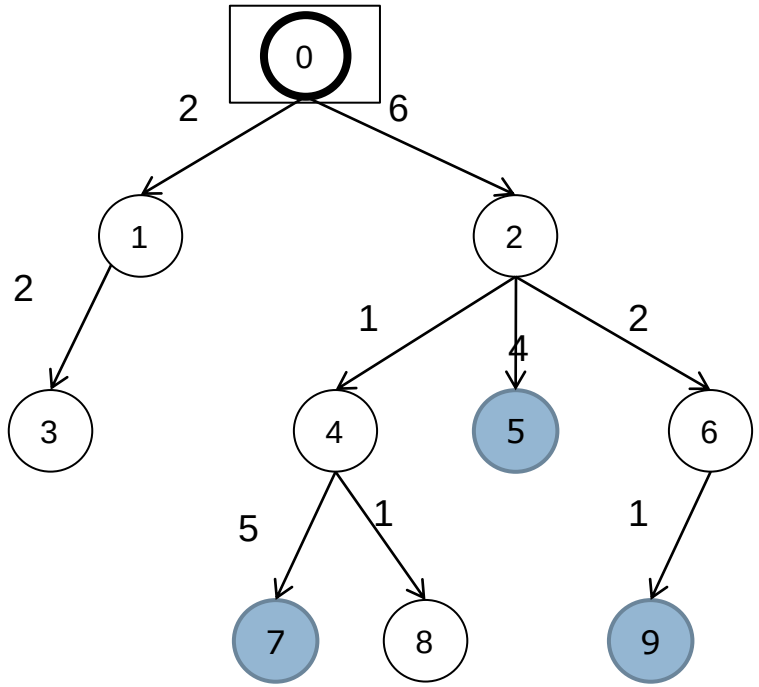
# *Algoritmo Best First Search (BFS)*

*Ejemplo abstracto: Espacio de búsqueda en forma de árbol*



# *Algoritmo Best First Search (BFS)*

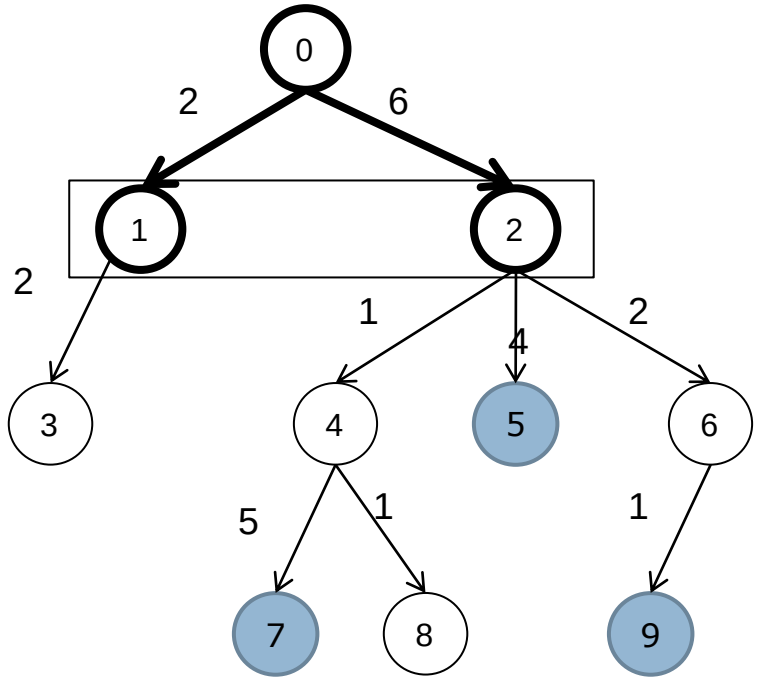
*Ejemplo abstracto*




**frontier = (0)**

# *Algoritmo Best First Search (BFS)*

*Ejemplo abstracto*

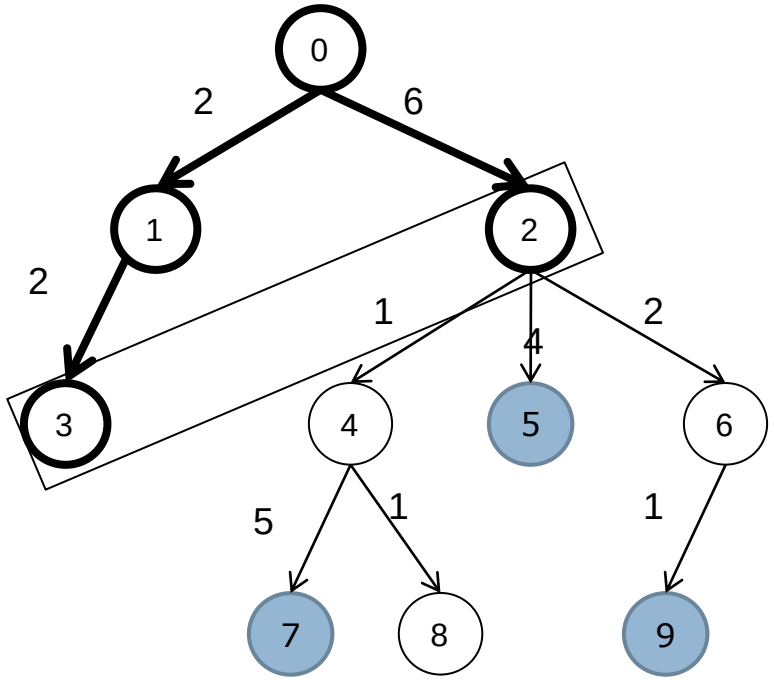



frontier = (0)

**frontier = (1,2)**

# *Algoritmo Best First Search (BFS)*

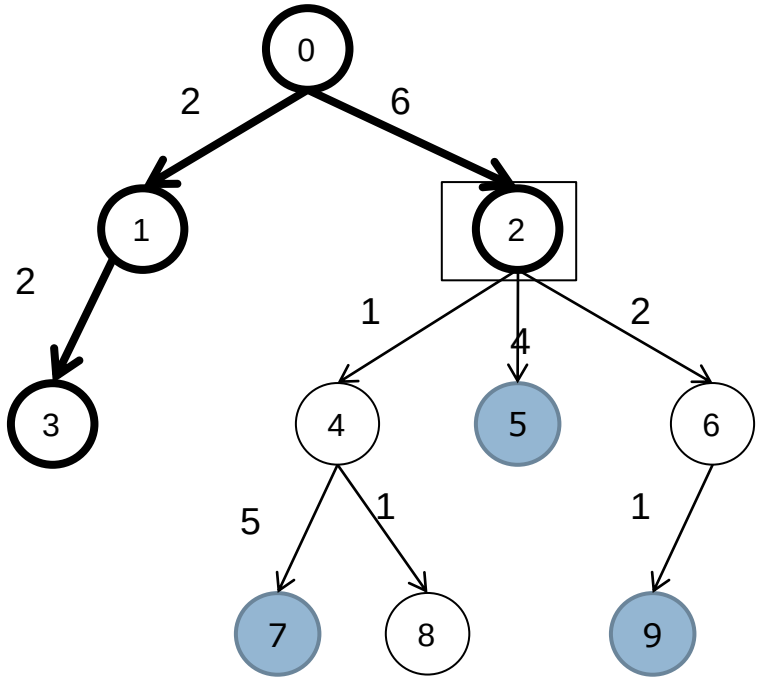
*Ejemplo abstracto*




frontier = (0)
frontier = (1,2)
**frontier = (3,2)**

# *Algoritmo Best First Search (BFS)*

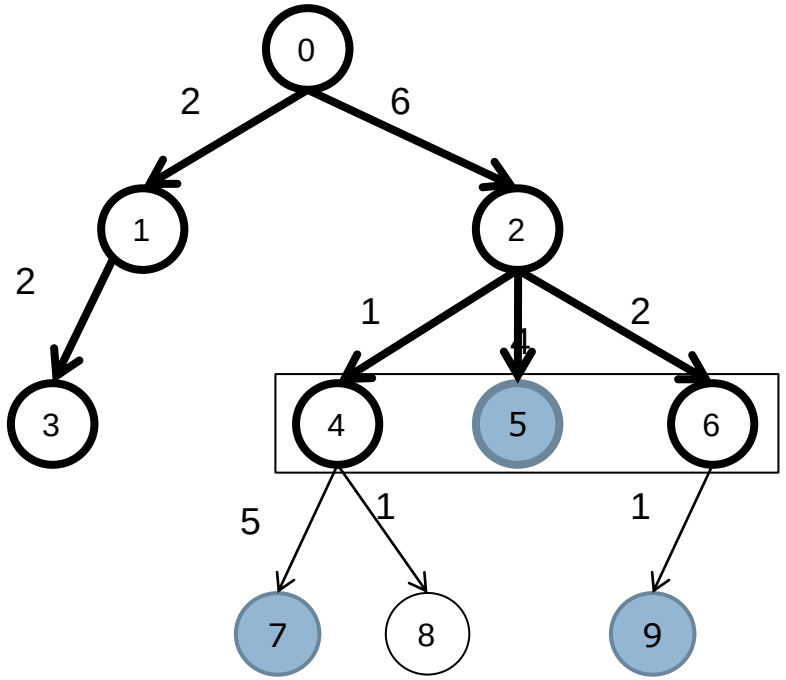
*Ejemplo abstracto*




frontier = (0)
frontier = (1,2)
frontier = (3,2)
**frontier = (2)**

# *Algoritmo Best First Search (BFS)*

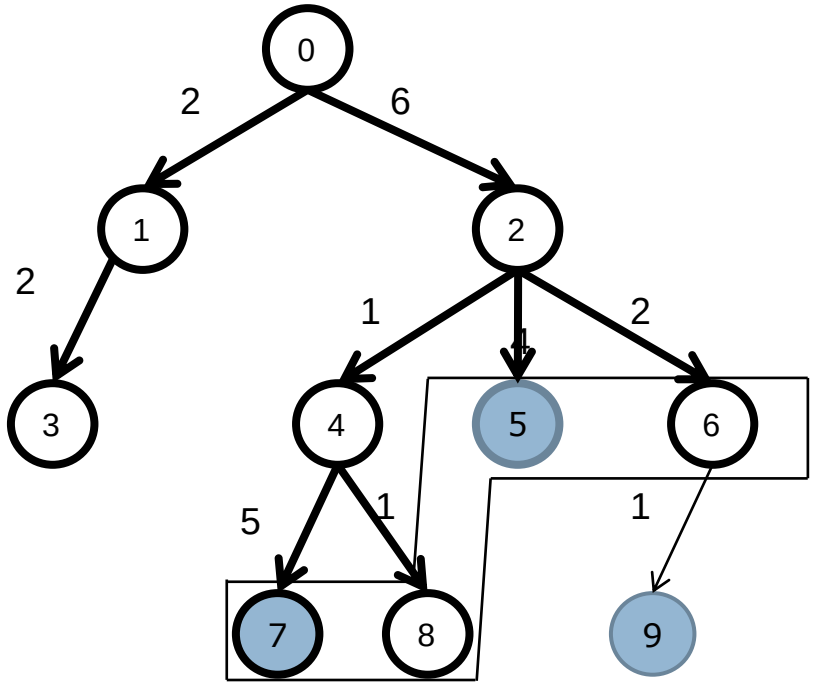
*Ejemplo abstracto*




frontier = (0)
frontier = (1,2)
frontier = (3,2)
frontier = (2)
**frontier = (4,5,6)** **5 es solución pero no está el primero**

# *Algoritmo Best First Search (BFS)*

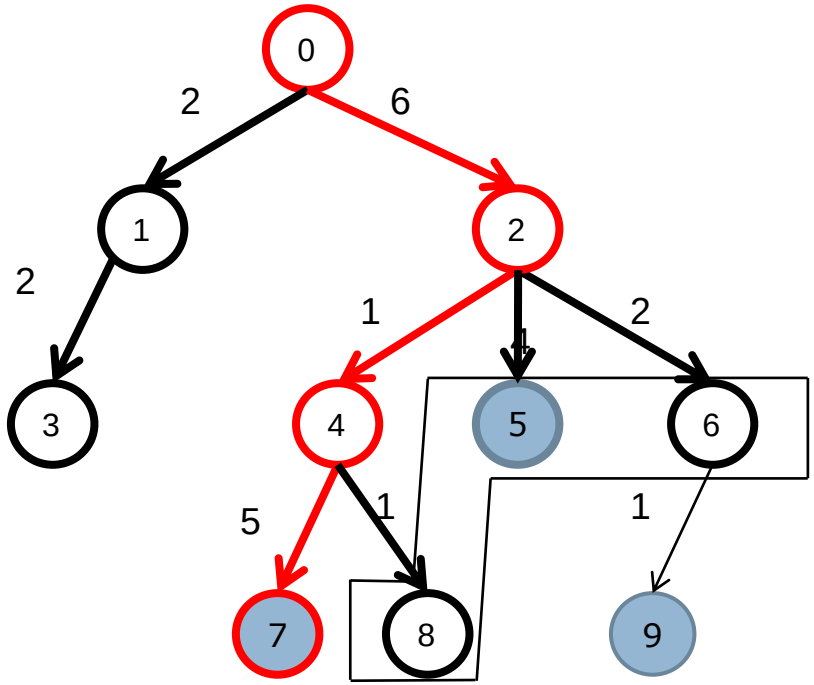
*Ejemplo abstracto*




frontier = (0)
frontier = (1,2)
frontier = (3,2)
frontier = (2)
frontier = (4,5,6) 5 es solución pero no está el primero
**frontier = (7,8,5,6)**

# *Algoritmo Best First Search (BFS)*

*Ejemplo abstracto*




frontier = (0)
frontier = (1,2)
frontier = (3,2)
frontier = (2)
frontier = (4,5,6) 5 es solución pero no está el primero
frontier = (7,8,5,6)
**frontier = (8,5,6)**

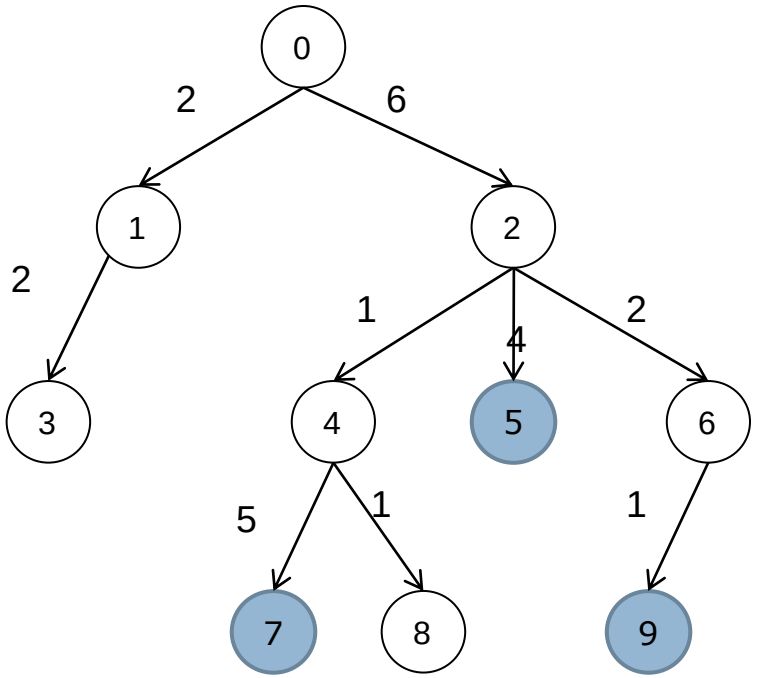
**Solución: 0-2-4-7**
**Coste: 6+1+5 = 11 (no óptima)**

# Algoritmos de búsqueda a ciegas

*Casos particulares del Algoritmo General BF*

- Búsqueda en Profundidad
  - Insertar *child* al principio de *frontier* (f(n) = 1 / profundidad(n))
  - Inconvenientes
    - Ramas infinitas
    - ¿Cómo eliminar nodos sin sucesores en frontier?
- Búsqueda en Anchura
  - Insertar *child* al final de *frontier* (f(n) = profundidad(n))
  - Es una estrategia Completa
- Coste Uniforme
  - Insertar *child* de forma ordenada en *frontier* según el coste al inicial (f(n) = n.PATH-COST)
  - Es una estrategia Admisible

# *Resumen de resultados de las tres estrategias de búsqueda a ciegas en el ejemplo anterior*




- Búsqueda en Profundidad
  - f(n) = 1 / profundidad(n)
  - Nodos expandidos: 0, 1, 3, 2, 4, 7
- Búsqueda en Anchura
  - f(n) = profundidad(n)
  - Nodos expandidos: 0, 1, 2, 3, 4, 5
- Coste Uniforme
  - f(n) = n.PATH-COST (coste inicial a n)
  - Nodos expandidos: 0, 1, 3,  2, 4, 6, 8, 9

# *Implementación específica de Búsqueda en Anchura en árboles*
## *[Russel&Norvig, 2022]*

```
function BREADTH-FIRST-SEARCH(problem) returns a solution node or failure
  node ← NODE(problem.INITIAL)
  if problem.IS-GOAL(node.STATE) then return node
  frontier ← a FIFO queue, with node as an element
  [redacted]
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if problem.IS-GOAL(s) then return child
      [redacted]
        add child to frontier
  return failure
```

- No es necesaria la función de evaluación f()
- **frontier** se gestiona como una cola
- El test **Is-Goal(s)** se hace al generar los estados nuevos, no al expandir los estados, así se puede detectar antes la solución menos profunda
- El primer camino encontrado a una solución es el de menor profundidad

# *Implementación específica de Búsqueda en Profundidad en árboles [Russel&Norvig, 2022]*

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution node or failure
  for depth = 0 to ∞ do
    result ← DEPTH-LIMITED-SEARCH(problem, depth)
    if result ≠ cutoff then return result

function DEPTH-LIMITED-SEARCH(problem, ℓ) returns a node or failure or cutoff
  frontier ← a LIFO queue (stack) with NODE(problem.INITIAL) as an element
  result ← failure
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    if DEPTH(node) > ℓ then
      result ← cutoff
    else if not IS-CYCLE(node) do
      for each child in EXPAND(problem, node) do
        add child to frontier
  return result
```

- No es necesaria la función de evaluación f()
- **frontier** se gestiona como una pila
- El test **Is-Goal(s)** se hace al expandir el estado s, solo así se puede encontrar la solución “más a la izquierda”
- Se pueden controlar la profundidad y la existencia de ciclos para evitar ramas infinitas
- Iterando en la profundidad, se consigue la solución menos profunda

# *Algoritmo de Backtracking (búsqueda a ciegas en profundidad recursiva)*

```
function BACKTRACKING-SEARCH(problem, l) returns solution-node or failure
        node = NODE(problem.INITIAL)
        return BACKTRACKING(node, problem, l)

function BACKTRACKING(node, problem, l) returns solution-node or failure
        if (IS-GOAL(node)) return node
        actions ← ACTION-LIST(node, problem)
        if (IS-EMPTY(actions) or l = 0) return failure
        node' ← failure
        while ( !IS-EMPTY(actions) and node' = failure) do
                action ← POP(actions)
                node-suc ←EXPAND(problem, node, action)
                node' ← BACKTRACKING(node-suc, problem, l-1)
        return node'

function EXPAND(problem ,node, action) returns node
        s ← node.STATE
        s'← problem.RESULT(s,action)
        cost ← node.PATH-COST + problem.ACTION-COST(s, action, s')
        return NODE(STATE=s', PARENT=node, ACTION=action, PATH-COST=cost)
```

- El árbol de búsqueda y **frontier** se sustituyen por la pila de llamadas recursivas (solo hay una rama activa)
- Con el límite de la profundidad se controlan las ramas infinitas
- Se puede utilizar con profundidad iterativa, como el algoritmo de búsqueda en profundidad iterativo
- También se pueden controlar los ciclos, con el coste correspondiente
- Los sucesores se generan de 1 en 1 en lugar de todos a la vez como en los algoritmos iterativos

# *Coste computacional de los algoritmos de búsqueda a ciegas en árboles*

| | Tiempo | Espacio |
|---|---|---|
| Primero en profundidad | $O(b^d)$ | $O(bd)$ |
| Primero en anchura | $O(b^d)$ | $O(b^d)$ |

*b* factor de ramificación
*d* profundidad del árbol

- Estos valores se refieren a una situación particular en la que cada estado del espacio de búsqueda tiene un número de *b* sucesores y una profundidad *d*. Además las soluciones se encuentran a esta profundidad *d*.
- En el caso del algoritmo primero en profundidad, se puede conseguir un coste polinomial en espacio si se utilizan mecanismos para ir descartando la parte del árbol de búsqueda que no tiene sucesores en **frontier**, y que por lo tanto representa caminos cerrados que no pueden llegar a una solución.
- El algoritmo de Backtracking resuelve el problema del coste espacial mediante el uso de la recursión.