Técnicas de Inteligencia Artificial

Tema 3.1

Búsqueda en espacio de estados

Resolución de problemas como búsqueda

- La búsqueda en espacio de estados es un método usado en IA para encontrar la solución a un problema mediante la búsqueda a partir de un conjunto de posibles estados del mismo.
- Se usa el espacio de estados para navegar desde un estadio inicial al estado objetivo.
- Existen algunos problemas de IA que pueden resolverse como una búsqueda dentro del espacio de estados, cuando el entorno cumple ciertos factores:
  - **Estático**. El mundo no cambia por sí mismo, y nuestras acciones no lo modifican.
  - **Discreto**. Existe un número finito de estados individuales, en lugar de un espacio continuo de opciones.
  - **Observable**. Los estados se pueden determinar mediante la observación.
  - **Determinístico**. Cada acción produce unos resultados determinados.

Resolución de problemas como búsqueda

- El **entorno** es toda la información acerca del mundo que permanece constante mientras resolvemos el problema.
- Un **estado** es un conjunto de propiedades que definen las condiciones actuales del mundo donde se encuentra el agente.
  - Se puede considerar como una captura instantánea del mundo en un momento determinado.
  - El conjunto de todos los estados posibles se llama espacio de estados.
- El **estado inicial** es el estado donde el agente comienza el problema.
- El **estado objetivo** es el estado donde el agente pretende llegar.
- Un agente puede tomar diferentes **acciones** que le pueden llegar a nuevos estados diferentes.

Formulación de problemas como búsqueda

- Para formular un problema como búsqueda se deben definir ciertas cuestiones:
  - ¿Cómo se pueden representar mis estados?
  - ¿Cuál es el estado inicial?
  - ¿Cuál es el estado objetivo?
  - ¿Qué forma tiene el espacio de estados?
    - ¿Se puede formalizar como grafo o como árbol?
  - ¿Cuál es la función de coste?
    - ¿Cómo se puede saber cómo de bueno es un estado o una acción?
    - Habitualmente se desea minimizar en lugar de maximizar.
    - Se suele formular como una función del camino desde el estado inicial al estado final.

Formulación de problemas como búsqueda

- Las soluciones obtenidas por estos algoritmos tienen la siguiente forma:
  - Un camino entre el estado inicial y el estado objetivo.
  - La calidad es medida mediante el coste del camino.
  - Las soluciones óptimas son aquellas que tienen el menor coste de todos los caminos posibles.

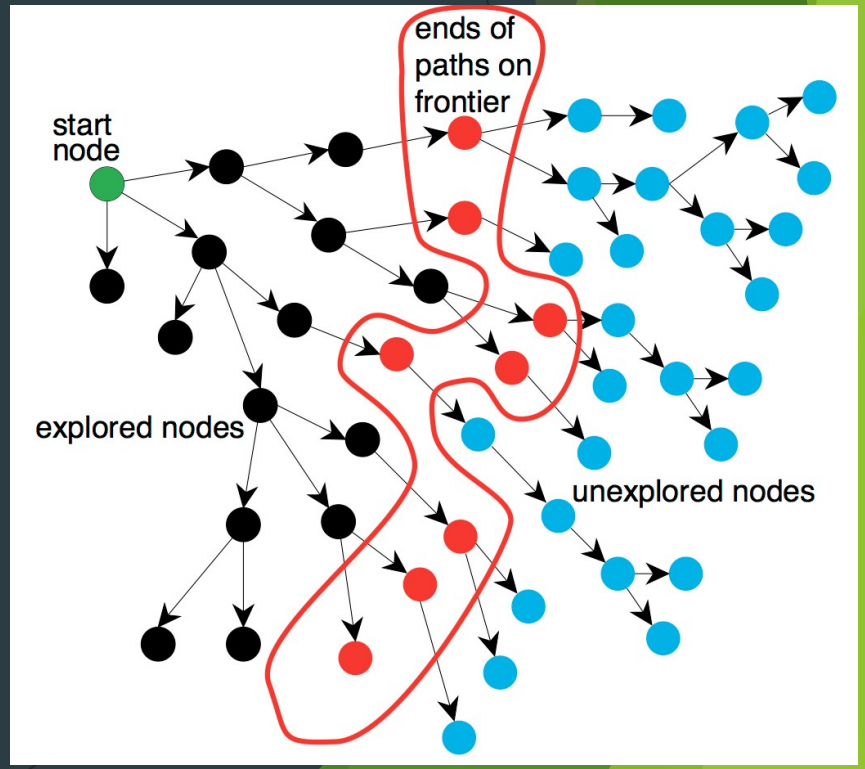
- La búsqueda en espacios de estados implica utilizar algoritmos de búsqueda en grafos.
- Existen dos estructuras simultáneas usadas en la búsqueda:
  - Árbol o grafo del espacio de estados subyacente al problema.
    - No depende de la búsqueda actual que se está realizando.
    - En muchos casos no se almacena en memoria porque es demasiado grande.
  - Árbol manteniendo el registro de la búsqueda actual en progreso (árbol de búsqueda).
    - Siempre depende de la búsqueda actual y siempre se almacena en memoria.

Árbol de búsqueda

- Un nodo del árbol de búsqueda almacena:
  - Un estado (del espacio de estados).
  - Una referencia al nodo anterior (padre) del camino (habitualmente).
  - La acción que provocó la transición desde el nodo padre al actual (en ocasiones).
  - El coste del camino desde el estado inicial al actual.
- **Frontera**. Estructura de datos que almacena el conjunto de nodos que se pueden examinar a continuación en el algoritmo.
  - Se representan habitualmente como una pila, cola o cola de prioridad.
- **Conjunto explorado**. Almacena la colección de estados que ya han sido examinados (y que no deben volverse a visitar).
  - Se utilizan estructuras de datos que permitan determinar de manera rápida si un elemento pertenece al conjunto.

Algoritmo de búsqueda genérico

- Todos los algoritmos de búsqueda funcionan esencialmente de la misma forma:
  - Comienzo con un estado inicial.
  - Expandir un nodo. Generar todos los posibles estados siguientes.
  - Seleccionar un nuevo nodo para expandir.
  - Continuar hasta encontrar un estado objetivo.



Algoritmo de búsqueda genérico

```
function TREE-SEARCH(problem) returns a solution, or failure
    initialize the frontier using the initial state of problem
    loop do
        if the frontier is empty then return failure
        choose a leaf node and remove it from the frontier
        if the node contains a goal state then return the corresponding solution
        expand the chosen node, adding the resulting nodes to the frontier
```

Frontier = stack, queue, or priority queue.

Algoritmo de búsqueda genérico

function GRAPH-SEARCH(*problem*) **returns** a solution, or failure

initialize the frontier using the initial state of *problem*

initialize the explored set to be empty

loop do

if the frontier is empty **then return** failure

choose a leaf node and remove it from the frontier

if the node contains a goal state **then return** the corresponding solution

add the node to the explored set

expand the chosen node, adding the resulting nodes to the frontier

only if not in the frontier or explored set

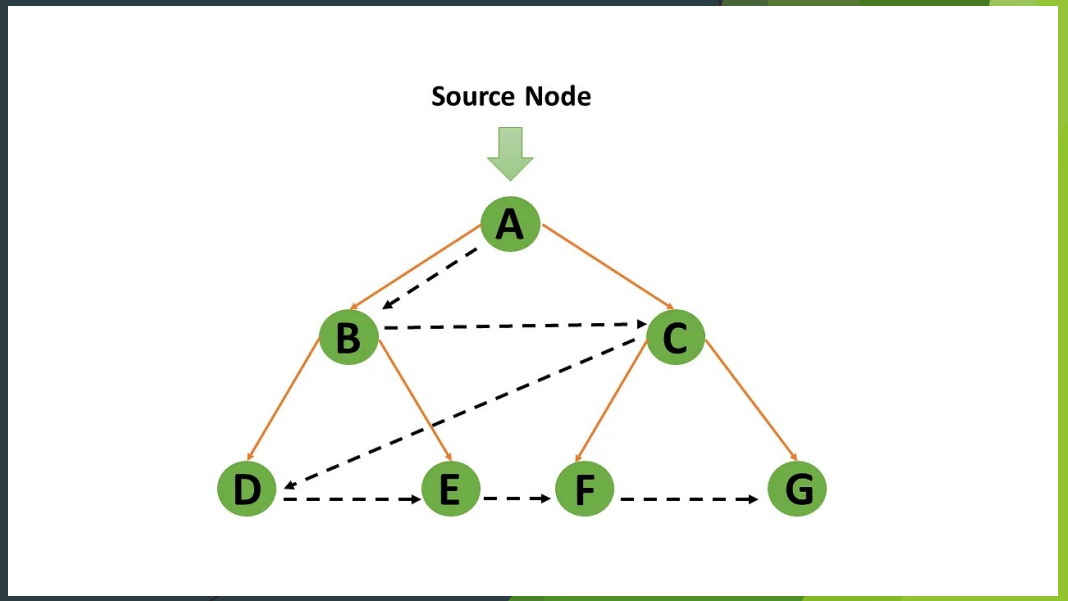
Explored set = hash table.

Estrategias de búsqueda

- Evaluación de una estrategia de búsqueda:
  - Completitud. ¿Siempre encuentra una solución si existe?
  - Optimalidad. ¿Encuentra la mejor solución?
  - Complejidad temporal.
  - Complejidad espacial.
- Variantes:
  - *Desinformados*. No tienen información adicional sobre el estado objetivo.
    - Búsqueda en anchura (Breadth-First Search).
    - Búsqueda en profundidad (Depth-First Search).
  - *Informados*. Tienen información adicional sobre el estado objetivo.
    - Búsqueda de mejor primero (Greedy Best-First Search).
    - Algoritmo A*.

Búsqueda en anchura (BFS)

- Selección del nodo más superficial para la exploración.
- Estructura para la frontera: cola FIFO.
- No permite examinar el mismo estado dos veces, incluso con diferentes caminos.
- Completo.
- Óptimo si el coste es una función no decreciente de la profundidad del nodo.
- Variantes.
  - **Búsqueda por coste uniforme**.

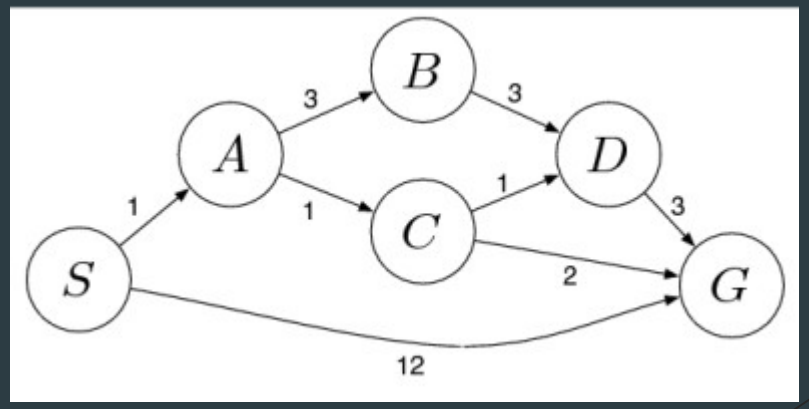


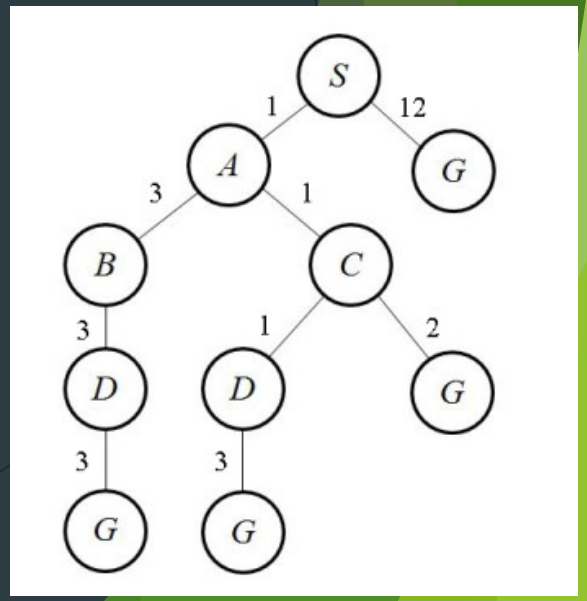
Búsqueda en anchura (BFS)

```
 1  procedure BFS(G, root) is
 2      let Q be a queue
 3      label root as explored
 4      Q.enqueue(root)
 5      while Q is not empty do
 6          v := Q.dequeue()
 7          if v is the goal then
 8              return v
 9          for all edges from v to w in G.adjacentEdges(v) do
10              if w is not labeled as explored then
11                  label w as explored
12                  w.parent := v
13                  Q.enqueue(w)
```

Búsqueda por coste uniforme (UCS)

- Selección del nodo con el menor coste de camino g(n) para la expansión.
- Estructura para la frontera: cola de prioridad.
- Suponiendo que se alcanza el mismo estado dos veces, ¿se puede reañadir a la frontera?
  - Sí, se reemplaza si el coste del camino es mejor.
- Se convierte en búsqueda en anchura si todas las aristas pesan lo mismo.
- Completo si el grafo es finito o tiene un coste creciente.
- Óptimo.





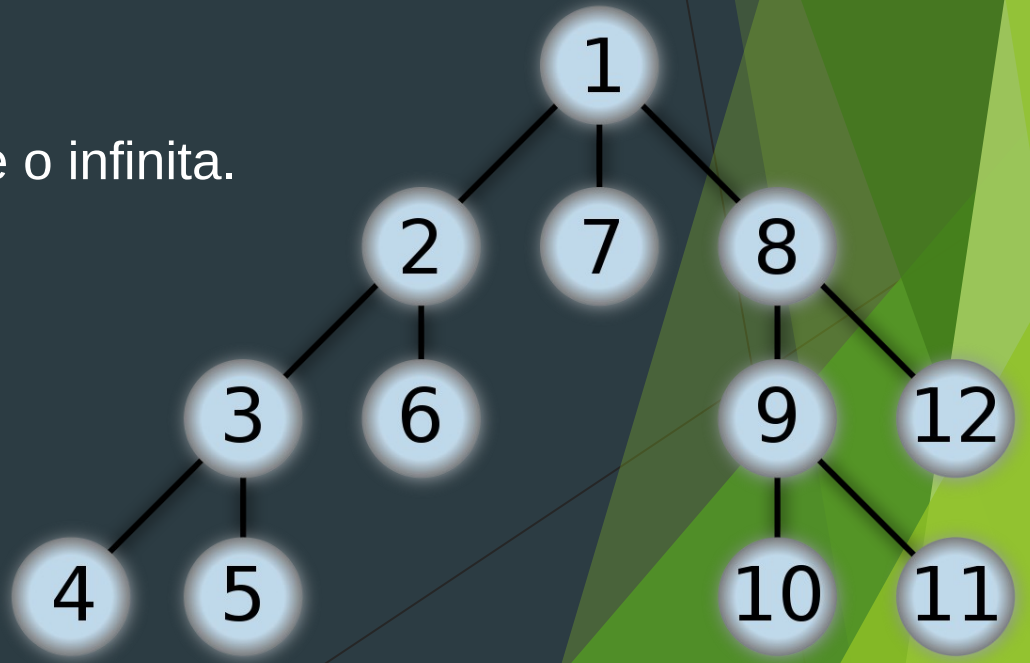
Búsqueda por coste uniforme (UCS)

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure

    node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
    frontier ← a priority queue ordered by PATH-COST, with node as the only element
    explored ← an empty set
    loop do
        if EMPTY?(frontier) then return failure
        node ← POP(frontier)  /* chooses the lowest-cost node in frontier */
        if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
        add node.STATE to explored
        for each action in problem.ACTIONS(node.STATE) do
            child ← CHILD-NODE(problem, node, action)
            if child.STATE is not in explored or frontier then
                frontier ← INSERT(child, frontier)
            else if child.STATE is in frontier with higher PATH-COST then
                replace that frontier node with child
```

Búsqueda en profundidad (DFS)

- Selección del nodo más profundo para expandir.
- Estructura para la frontera: pila o recursión.
- Completo si el árbol de búsqueda es finito y se controlan los bucles.
- No se asegura la optimalidad.
- Requiera mucha menos memoria que BFS.
- Problemas cuando la altura del árbol es muy grande o infinita.
- Variantes:
  - Búsqueda en profundidad limitada.
  - Búsqueda en profundidad iterativa.



Búsqueda en profundidad (DFS)

Algorithm 1: Recursive DFS

Data: G: The graph stored in an adjacency list
root: The starting node
Result: Prints all nodes inside the graph in the DFS order
$visited \leftarrow \{false\}$;
$DFS(root)$;

Function $DFS(u)$**:**
    if $visited[u] = true$ **then**
        return;
    end
    $print(u)$;
    $visited[u] \leftarrow true$;
    for $v \in G[u].neighbors()$ **do**
        $DFS(v)$;
    end
end

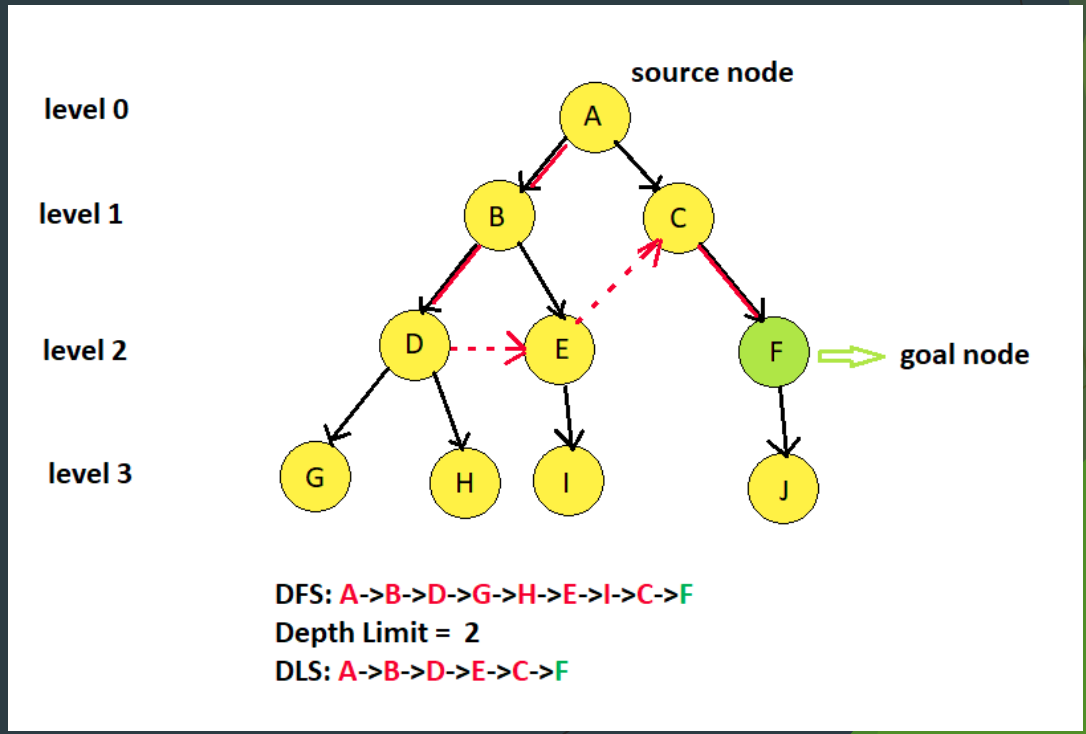
Búsqueda en profundidad (DFS)

Algorithm 2: Iterative DFS

```
Data: G: The graph stored in an adjacency list
      root: The starting node
Result: Prints all nodes inside the graph in the DFS order
visited ← {false};
stack ← {};
stack.push(root);
while ¬stack.empty() do
    u ← stack.top();
    stack.pop();
    if visited[u] = true then
        continue;
    end
    print(u);
    visited[u] ← true;
    for v ∈ G[u] do
        if visited[v] = false then
            DFS(v);
        end
    end
end
```

Búsqueda en profundidad limitada (DLDFS)

- Evita problemas de DFS poniendo un límite de profundidad (altura) a la búsqueda.
- No encuentra la solución cuando está por debajo del límite de profundidad.
- Completo si la solución está por encima del límite de profundidad.
- No se asegura optimalidad.



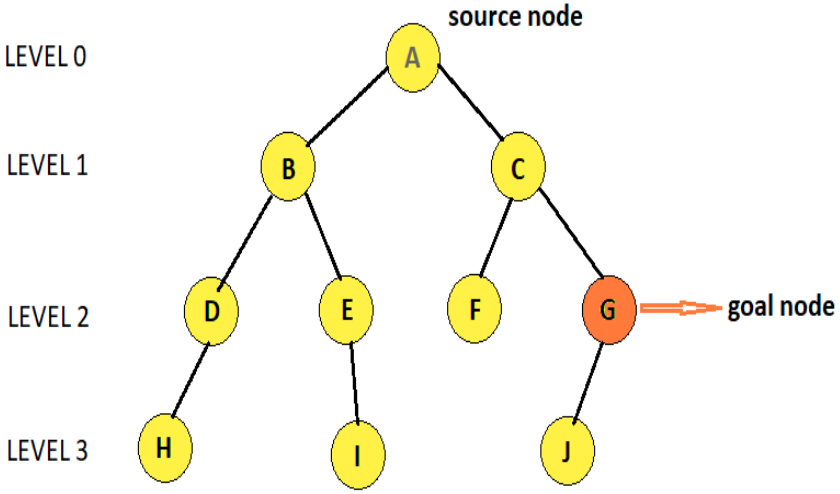
Búsqueda en profundidad limitada (DLDFS)

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
      cutoff_occurred? ← false
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          result ← RECURSIVE-DLS(child, problem, limit − 1)
          if result = cutoff then cutoff_occurred? ← true
          else if result ≠ failure then return result
      if cutoff_occurred? then return cutoff else return failure
```

Búsqueda en profundidad iterativa (IDS)

- Combinación de los algoritmos DFS y BFS.
- Encuentra el mejor límite de profundidad incrementándolo gradualmente hasta que el objetivo es encontrado.
- Eficiente en términos de búsqueda rápida y eficiencia de memoria.
- Visita los estados múltiples veces (una por cada aumento de la profundidad), aunque no suele ser un problema ya que en los árboles la mayoría de los nodos suelen estar en niveles inferiores.
- Estrategia utilizada en juegos con límite de tiempo.



IDDFS with max depth-limit = 3
Note that iteration terminates at depth-limit=2
Iteration 0: A
Iteration 1: A->B->C
Iteration 2: A->B->D->E->C->F->G

Búsqueda en profundidad iterativa (IDS)

Algorithm 3: Iterative Deepening

Data: s: the start node, *target*: the function that identifies a target node, *children*: the function that returns the children of a node.

Result: The shortest path between u and a target node

for $\ell = 0, 1, \ldots, \infty$ **do**
    $result \leftarrow$ apply DLDFS with limit depth ℓ
    if $result \neq CUTOFF$ **then**
        return *result*
    end
end

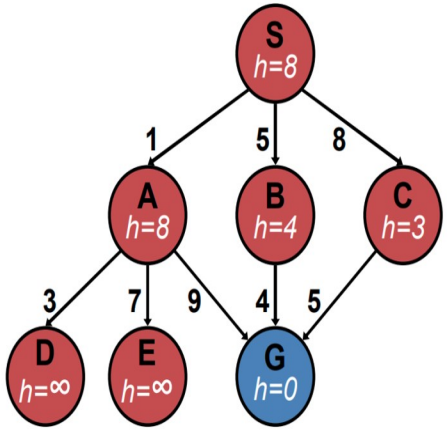
Búsqueda de mejor primero (GBFS)

- Mismo algoritmo que la búsqueda por coste uniforme.
- Utiliza una función de evaluación diferente para ordenar la cola de prioridad.
- Necesita una función heurística h(n).
  - h(n). Estimación del menor coste del camino desde el nodo actual a un nodo objetivo.
- No asegura completitud.
- No asegura optimalidad.
- Más eficiente que BFS y DFS con una buena heurística.
- No tiene en cuenta el coste del camino desde el inicio al nodo actual.

$f(n) = h(n)$

# of nodes tested: 3, expanded: 2

expnd. node	Frontier
	{S:8}
S	{C:3,B:4,A:8}
C	{G:0,B:4, A:8}
G goal	{B:4, A:8} not expanded



Búsqueda de mejor primero (GBFS)

Algorithm 3 Best first search

```
1:  procedure BEST_FIRST(START, GOAL)
2:      closed = set({})
3:      open = set({start})
4:      score_of = {}
5:      score_of [start] = calculate_heuristics(start)
6:      while not open.is_empty do
7:          current = min(open)                          ▷ Find node with minimum f
8:          if current = goal then
9:              return RECOVER_PATH(current)  ▷ Reconstruct the path to goal
10:         closed.add(current)
11:         open.remove(current)
12:         for each neighbor in current:neighbors() do
13:             if not closed.contains(neighbor) then
14:                 score_of [node] = calculate_heuristics(neighbor)
15:                 open.add(neighbor)
```

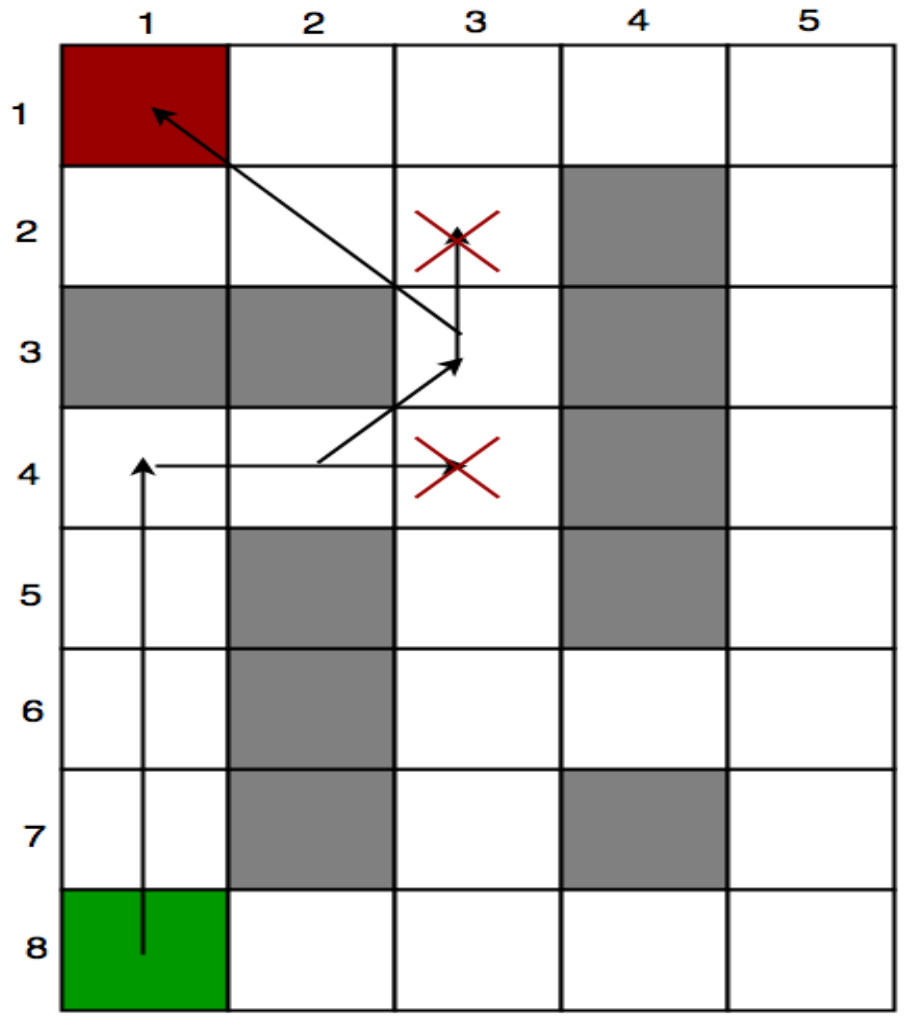
Algoritmo A*

- Ordenación de la cola de prioridad utilizando una función f(n) que debe ser el menor coste estimado del camino que se genera a partir del nodo actual.
  - f(n) = g(n) + h(n)
    - g(n) → coste actual del camino
    - h(n) → estimación del coste desde el nodo actual al objetivo.
- Una función heurística h(n) es admisible si nunca sobrestima el verdadero coste desde el nodo actual al objetivo.
  - h(n) siempre debe ser menor o igual que el verdadero coste desde el nodo actual al objetivo.
- ¿Qué ocurre si se establece h(n) = 0 para todo n?
  - Búsqueda de coste uniforme.

Algoritmo A*

- Una función heurística h(n) es consistente si los valores de h(n) a través de cualquier camino del árbol de búsqueda no decrece.
  - Dado un nodo n y una acción que nos lleva desde n a n':
    - h(n) <= coste(n, a, n') + h(n').
    - h(n) – h(n') <= coste(n, a, n')
- Consistencia implica admisibilidad (pero no al revés).
- Resulta complicado crear heurísticas que sean admisibles pero no consistentes.
- El algoritmo A* es óptimo si h(n) es consistente y admisible.

Algoritmo A*



Unblocked

Blocked

Target

Source

A* Search Algorithm makes the most intelligent choice at each step. Hence you can see that algorithm goes from (4,2) to (3,3) and not (4,3) (shown by cross).

Similarly the algorithm goes from (3,3) to (2,2) and not (2,3) (shown by cross).

Algoritmo A*

```
A* search {

 closed list = [ ]
 open list = [start node]

    do {
            if open list is empty then {
                    return no solution
            }
            n = heuristic best node
            if n == final node then {
                    return path from start to goal node
            }
            foreach direct available node do{
                    if current node not in open and not in closed list do {
                            add current node to open list and calculate heuristic
                            set n as his parent node
                    }
                    else{
                            check if path from star node to current node is
                            better;
                            if it is better calculate heuristics and transfer
                            current node from closed list to open list
                            set n as his parrent node
                    }
            delete n from open list
            add n to closed list
    } while (open list is not empty)

}
```