

eman ta zabal zazu



Universidad  
del País Vasco

Euskal Herriko  
Unibertsitatea

Técnicas de Inteligencia Artificial

Ingeniería Informática de Gestión y Sistemas de Información

# Practica 1

## Problemas de Búsqueda

Autor(es):

Xabier Gabiña

29 de septiembre de 2024

# Índice general

<b>1. Introducción</b>	<b>2</b>
<b>2. Algoritmos no informados</b>	<b>3</b>
2.1. DFS - Depth First Search . . . . .	3
2.2. BFS - Breadth First Search . . . . .	5
2.3. UCS - Uniform Cost Search . . . . .	6
<b>3. Algoritmos informados</b>	<b>7</b>
3.1. A* . . . . .	7

# 1. Introducción

En el marco de la asignatura de Técnicas de Inteligencia Artificial, se nos ha propuesto implementar y analizar diversos algoritmos de búsqueda aplicados al contexto de un proyecto académico desarrollado por la Universidad de Berkeley, basado en el clásico juego Pacman. El objetivo principal de esta práctica es profundizar en el funcionamiento de diferentes estrategias de búsqueda, estudiando su eficiencia y comportamiento en diferentes escenarios.

Los algoritmos de búsqueda son fundamentales en el campo de la inteligencia artificial, ya que permiten encontrar soluciones óptimas o satisfactorias en problemas complejos. En esta práctica, nos enfocaremos en tres tipos de algoritmos de búsqueda no informados: Depth First Search (DFS), Breadth First Search (BFS) y Uniform Cost Search (UCS). Además, exploraremos un algoritmo de búsqueda informado: A\*. Cada uno de estos algoritmos tiene sus propias características y aplicaciones, y su estudio nos permitirá comprender mejor sus ventajas y limitaciones.

A lo largo de este documento, se presentarán las implementaciones de cada uno de estos algoritmos, junto con una descripción detallada de su funcionamiento y análisis de su rendimiento. Se incluirán ejemplos prácticos y se discutirán los resultados obtenidos en diferentes escenarios de búsqueda. El objetivo es proporcionar una visión completa y comprensiva de cómo estos algoritmos pueden ser aplicados en la resolución de problemas de búsqueda en inteligencia artificial.

## 2. Algoritmos no informados

### 2.1. DFS - Depth First Search

#### Descripción

DFS o Depth First Search es un algoritmo de búsqueda no informado que se basa en la exploración de todos los nodos de un grafo siguiendo una rama hasta llegar a un nodo hoja, para después retroceder y explorar otra rama. Este algoritmo se implementa mediante una pila, en la que se van almacenando los nodos a visitar. Su coste en tiempo es de  $O(b^m)$ , donde  $b$  es el factor de ramificación y  $m$  es la profundidad máxima del árbol. Su coste en espacio es de  $O(bm)$ , donde  $b$  es el factor de ramificación y  $m$  es la profundidad máxima del árbol.

#### Primera implementación

```
1 def depthFirstSearch(problem):
2     """
3     Implementación del algoritmo de búsqueda en profundidad.
4
5     Args:
6         problem (SearchProblem): Problema de búsqueda
7     Returns:
8         list: Lista de acciones para llegar al objetivo
9     """
10    stack = [problem] # Pila para almacenar los nodos a visitar
11    visited = set()    # Conjunto para almacenar los nodos visitados
12    path = []          # Lista para almacenar el camino al nodo objetivo
13
14    while stack:       # Mientras haya elementos en el stack
15        nodo_actual = stack.pop() # Sacar el último elemento de la pila
16        if nodo_actual in visited: # Si el nodo actual ya ha sido visitado
17            continue
18        visited.add(nodo_actual) # Marcar el nodo actual como visitado
19        path.append(nodo_actual.contenido) # Añadir el nodo actual al camino
20        if nodo_actual.isGoalState(): # Si el nodo actual es el objetivo
21            return path
22        for hijo in reversed(nodo_actual.getSucesor()): # Añadir los hijos del nodo actual a la
23            # pila
24            stack.append(hijo)
```

Listing 2.1: Implementación final del DFS

## Implementación Final

```
1 def depthFirstSearch(problem):
2     """
3     Implementación del algoritmo de búsqueda en profundidad.
4
5     Args:
6         problem (SearchProblem): Problema de búsqueda
7     Returns:
8         list: Lista de acciones para llegar al objetivo
9     """
10    stack = util.Stack() # Añadir el nodo inicial a la pila
11    stack.push([problem.getStartState(), []])
12    visited = set() # Conjunto para almacenar los nodos visitados
13
14    while not stack.isEmpty(): # Mientras haya elementos en el stack
15        nodo_actual = stack.pop() # Sacar el último elemento de la pila
16        if problem.isGoalState(nodo_actual[0]): # Si el nodo actual es el objetivo
17            return nodo_actual[1] # Devolver el camino
18        if nodo_actual[0] not in visited:
19            visited.add(nodo_actual[0])
20            for estado, accion, costo in reversed(problem.getSuccessors(nodo_actual[0])):
21                camino = nodo_actual[1] + [accion]
22                stack.push([estado, camino])
23
```

Listing 2.2: Implementación final del DFS

## Comentarios

## 2.2. BFS - Breadth First Search

### Descripción

BFS o Breadth First Search es un algoritmo de búsqueda no informado que se basa en la exploración de todos los nodos de un grafo siguiendo un orden de anchura, es decir, se visitan todos los nodos de un nivel antes de pasar al siguiente.

Este algoritmo se implementa mediante una cola, en la que se van almacenando los nodos a visitar.

Su coste en tiempo es de  $O(b^m)$ , donde  $b$  es el factor de ramificación y  $m$  es la profundidad máxima del árbol.

Su coste en espacio es de  $O(bm)$ , donde  $b$  es el factor de ramificación y  $m$  es la profundidad máxima del árbol.

### Implementación Final

```
1 def breadthFirstSearch(problem):
2     """
3     Implementacion del algoritmo de busqueda en anchura.
4
5     Args:
6         problem (SearchProblem): Problema de busqueda
7     Returns:
8         list: Lista de acciones para llegar al objetivo
9     """
10    queue = util.Queue() # Añadir el nodo inicial a la cola
11    queue.push([problem.getStartState(), []])
12    visited = set()      # Conjunto para almacenar los nodos visitados
13
14    while not queue.isEmpty(): # Mientras haya elementos en la cola
15        nodo_actual = queue.pop() # Sacar el primer elemento de la cola
16        if problem.isGoalState(nodo_actual[0]): # Si el nodo actual es el objetivo
17            return nodo_actual[1] # Devolver el camino
18        if nodo_actual[0] not in visited:
19            visited.add(nodo_actual[0])
20            for estado, accion, costo in problem.getSuccessors(nodo_actual[0]): # Añadir los hijos del
21                nodo actual a la cola
22                camino = nodo_actual[1] + [accion]
23                queue.push([estado, camino])
```

Listing 2.3: Implementación final del BFS

### Comentarios

## 2.3. UCS - Uniform Cost Search

### Descripción

UCS o Uniform Cost Search es un algoritmo de búsqueda no informado que se basa en la exploración de todos los nodos de un grafo siguiendo un orden de coste, es decir, se visitan los nodos con menor coste antes que los nodos con mayor coste.

Este algoritmo se implementa mediante una cola de prioridad, en la que se van almacenando los nodos a visitar.

Su coste en tiempo es de  $O(b^m)$ , donde  $b$  es el factor de ramificación y  $m$  es la profundidad máxima del árbol.

Su coste en espacio es de  $O(bm)$ , donde  $b$  es el factor de ramificación y  $m$  es la profundidad máxima del árbol.

### Primera implementación

### Implementación Final

```
1 def uniformCostSearch(problem):
2     """
3     Implementacion del algoritmo de busqueda de coste uniforme.
4
5     Args:
6         problem (SearchProblem): Problema de busqueda
7     Returns:
8         list: Lista de acciones para llegar al objetivo
9     """
10    queue = util.PriorityQueue() # Añadir el nodo inicial a el heap
11    queue.push([problem.getStartState(), [], 0], 0)
12    visited = set() # Conjunto para almacenar los nodos visitados
13
14    while not queue.isEmpty(): # Mientras haya elementos en el stack
15        nodo_actual = queue.pop() # Sacar el último elemento de la pila
16        if problem.isGoalState(nodo_actual[0]): # Si el nodo actual es el objetivo
17            return nodo_actual[1] # Devolver el camino
18        if nodo_actual[0] not in visited:
19            visited.add(nodo_actual[0])
20            for estado, accion, costo in problem.getSuccessors(nodo_actual[0]): # Añadir los hijos del
21                # nodo actual a la pila
22                camino = nodo_actual[1] + [accion]
23                queue.push([estado, camino, nodo_actual[2] + costo], nodo_actual[2] + costo)
```

Listing 2.4: Implementación final del UCS

### Comentarios

# 3. Algoritmos informados

## 3.1. A\*

Descripción

Primera implementación

Implementación Final

Comentarios