

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

1 de octubre 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        |
| <b>4. Resultados</b>                      | <b>8</b> |
| 4.1. Casos de pruebas . . . . .           | 8        |
| 4.2. Autograder . . . . .                 | 8        |

# 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 nivel por nivel. Este algoritmo se implementa mediante una cola en la que se van almacenando los nodos que se deben visitar, manteniendo un orden de llegada basado en los niveles.

Su coste en tiempo es de  $O(b^d)$ , donde  $b$  es el factor de ramificación y  $d$  es la profundidad del nodo más cercano del árbol y su coste en espacio es de  $O(b^d)$ , ya que debe almacenar todos los nodos del nivel actual antes de pasar al siguiente.

### 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 informada que expande los nodos en función del costo acumulado. En contrario al BFS que prioriza la profundidad de los nodos el UCS utiliza la cola de prioridad para ordenar los nodos dependiendo su costo acumulado y expande primero el nodo con menor costo, obteniendo así una solución óptima en términos de costo.

El coste en tiempo del UCS es de  $O(b^{1+\frac{C^*}{\epsilon}})$ , donde  $b$  es el factor de ramificación,  $C^*$  es el costo de la solución óptima y  $\epsilon$  es el menor costo de transición entre nodos. Su coste de espacio es así mismo  $O(b^{1+\frac{C^*}{\epsilon}})$  ya que debe de almacenar todos los nodos en la cola de prioridad hasta encontrar la solución óptima

### 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

El A\* es un algoritmo de búsqueda informada que utiliza las ventajas del UCS y Greedy Best-First Search, utilizando una función heurística para guiar la búsqueda hacia el objetivo de una manera más eficiente. A\* expande los nodos en mediante una función de costo total. Donde  $f(n) = g(n) + h(n)$ ,  $g(n)$  siendo el costo acumulado desde el nodo inicial hasta  $n$  y  $h(n)$  es una estimación heurística del costo  $n$  hasta el objetivo. Este algoritmo se implementa mediante una cola de prioridad, donde los nodos se ordenan según su valor  $f(n)$  y se expande primero el nodo con menor valor  $f(n)$ .

Su coste en tiempo es de  $O(b^d)$ , donde  $b$  es el factor de ramificación y  $d$  es la profundidad de la solución óptima, sin embargo el tiempo puede reducirse si la heurística es eficiente. Así mismo su coste de espacio es de  $O(b^d)$ , ya que se debe almacenar todos los nodos generados en la cola de prioridad para asegurar la solución óptima.

### Implementación Final

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

Listing 3.1: Implementación final del A\*

### Comentarios



# 4. Resultados

## 4.1. Casos de pruebas

## 4.2. Autograder

```
1 Starting on 10-1 at 17:38:32
2
3 Question q1
4 =====
5
6 *** PASS: test_cases/q1/graph_backtrack.test
7 ***     solution:          ['1:A->C', '0:C->G']
8 ***     expanded_states:   ['A', 'B', 'C']
9 *** PASS: test_cases/q1/graph_bfs_vs_dfs.test
10 ***    solution:          ['0:A->B', '0:B->D', '0:D->G']
11 ***    expanded_states:   ['A', 'B', 'D']
12 *** PASS: test_cases/q1/graph_infinite.test
13 ***    solution:          ['0:A->B', '1:B->C', '1:C->G']
14 ***    expanded_states:   ['A', 'B', 'C']
15 *** PASS: test_cases/q1/graph_manypaths.test
16 ***    solution:          ['0:A->B1', '0:B1->C', '0:C->D', '0:D->E1', '0:E1->F', '0:F->G']
17 ***    expanded_states:   ['A', 'B1', 'C', 'D', 'E1', 'F']
18 *** PASS: test_cases/q1/pacman_1.test
19 ***    pacman layout:      mediumMaze
20 ***    solution length:    246
21 ***    nodes expanded:     269
22
23 ### Question q1: 3/3 ###
24
25
26 Question q2
27 =====
28
29 *** PASS: test_cases/q2/graph_backtrack.test
30 ***     solution:          ['1:A->C', '0:C->G']
31 ***     expanded_states:   ['A', 'B', 'C', 'D']
32 *** PASS: test_cases/q2/graph_bfs_vs_dfs.test
33 ***     solution:          ['1:A->G']
34 ***     expanded_states:   ['A', 'B']
35 *** PASS: test_cases/q2/graph_infinite.test
36 ***     solution:          ['0:A->B', '1:B->C', '1:C->G']
37 ***     expanded_states:   ['A', 'B', 'C']
38 *** PASS: test_cases/q2/graph_manypaths.test
39 ***     solution:          ['1:A->C', '0:C->D', '1:D->F', '0:F->G']
40 ***     expanded_states:   ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']
41 *** PASS: test_cases/q2/pacman_1.test
42 ***     pacman layout:      mediumMaze
43 ***     solution length:    68
44 ***     nodes expanded:     269
45
46 ### Question q2: 3/3 ###
47
48
49 Question q3
50 =====
51
52 *** PASS: test_cases/q3/graph_backtrack.test
53 ***     solution:          ['1:A->C', '0:C->G']
54 ***     expanded_states:   ['A', 'B', 'C', 'D']
55 *** PASS: test_cases/q3/graph_bfs_vs_dfs.test
56 ***     solution:          ['1:A->G']
57 ***     expanded_states:   ['A', 'B']
58 *** PASS: test_cases/q3/graph_infinite.test
59 ***     solution:          ['0:A->B', '1:B->C', '1:C->G']
60 ***     expanded_states:   ['A', 'B', 'C']
61 *** PASS: test_cases/q3/graph_manypaths.test
62 ***     solution:          ['1:A->C', '0:C->D', '1:D->F', '0:F->G']
```

```

63 ***     expanded_states:         ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']
64 *** PASS: test_cases/q3/ucs_0_graph.test
65 ***     solution:                 ['Right', 'Down', 'Down']
66 ***     expanded_states:         ['A', 'B', 'D', 'C', 'G']
67 *** PASS: test_cases/q3/ucs_1_problemC.test
68 ***     pacman layout:           mediumMaze
69 ***     solution length: 68
70 ***     nodes expanded:           269
71 *** PASS: test_cases/q3/ucs_2_problemE.test
72 ***     pacman layout:           mediumMaze
73 ***     solution length: 74
74 ***     nodes expanded:           260
75 *** PASS: test_cases/q3/ucs_3_problemW.test
76 ***     pacman layout:           mediumMaze
77 ***     solution length: 152
78 ***     nodes expanded:           173
79 *** PASS: test_cases/q3/ucs_4_testSearch.test
80 ***     pacman layout:           testSearch
81 ***     solution length: 7
82 ***     nodes expanded:           14
83 *** PASS: test_cases/q3/ucs_5_goalAtDequeue.test
84 ***     solution:                 ['1:A->B', '0:B->C', '0:C->G']
85 ***     expanded_states:         ['A', 'B', 'C']
86
87 ### Question q3: 3/3 ###
88
89
90 Question q4
91 =====
92
93 *** PASS: test_cases/q4/astar_0.test
94 ***     solution:                 ['Right', 'Down', 'Down']
95 ***     expanded_states:         ['A', 'B', 'D', 'C', 'G']
96 *** PASS: test_cases/q4/astar_1_graph_heuristic.test
97 ***     solution:                 ['0', '0', '2']
98 ***     expanded_states:         ['S', 'A', 'D', 'C']
99 *** PASS: test_cases/q4/astar_2_manhattan.test
100 ***     pacman layout:           mediumMaze
101 ***     solution length: 68
102 ***     nodes expanded:           221
103 *** PASS: test_cases/q4/astar_3_goalAtDequeue.test
104 ***     solution:                 ['1:A->B', '0:B->C', '0:C->G']
105 ***     expanded_states:         ['A', 'B', 'C']
106 *** PASS: test_cases/q4/graph_backtrack.test
107 ***     solution:                 ['1:A->C', '0:C->G']
108 ***     expanded_states:         ['A', 'B', 'C', 'D']
109 *** PASS: test_cases/q4/graph_manypaths.test
110 ***     solution:                 ['1:A->C', '0:C->D', '1:D->F', '0:F->G']
111 ***     expanded_states:         ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']
112
113 ### Question q4: 3/3 ###
114
115
116 Question q5
117 =====
118
119 *** Method not implemented: getStartState at line 311 of searchAgents.py
120 *** FAIL: Terminated with a string exception.
121
122 ### Question q5: 0/3 ###
123
124
125 Question q6
126 =====
127
128 *** Method not implemented: getStartState at line 311 of searchAgents.py
129 *** FAIL: Terminated with a string exception.
130
131 ### Question q6: 0/3 ###
132
133

```

```

134 Question q7
135 =====
136
137 *** PASS: test_cases/q7/food_heuristic_1.test
138 *** FAIL: test_cases/q7/food_heuristic_10.test
139 *** Heuristic failed non-triviality test
140 *** Tests failed.
141
142 ### Question q7: 0/4 ###
143
144
145 Question q8
146 =====
147
148 [SearchAgent] using function depthFirstSearch
149 [SearchAgent] using problem type PositionSearchProblem
150 Warning: this does not look like a regular search maze
151 *** Method not implemented: findPathToClosestDot at line 511 of searchAgents.py
152 *** FAIL: Terminated with a string exception.
153
154 ### Question q8: 0/3 ###
155
156
157 Finished at 17:38:32
158
159 Provisional grades
160 =====
161 Question q1: 3/3
162 Question q2: 3/3
163 Question q3: 3/3
164 Question q4: 3/3
165 Question q5: 0/3
166 Question q6: 0/3
167 Question q7: 0/4
168 Question q8: 0/3
169 -----
170 Total: 12/25
171
172 Your grades are NOT yet registered. To register your grades, make sure
173 to follow your instructor's guidelines to receive credit on your project.
174

```