Práctica 1: Búsqueda Heurística con A*

Universidad de Alicante
Departamento de Ciencia de la Computación e Inteligencia Artificial
Asignatura: Sistemas Inteligentes
Curso 2024/2025
Nombre del Estudiante: [Tu Nombre]
Fecha de Entrega: 3 de noviembre de 2024

Contents

1 Introducción

Esta práctica se centra en la implementación y análisis del algoritmo A* para la búsqueda de caminos óptimos en un entorno de rejilla 2D. El objetivo principal es desarrollar un sistema que permita a un conejo encontrar la ruta más eficiente hacia una zanahoria, considerando diferentes tipos de terreno y obstáculos.

El algoritmo A* es una técnica de búsqueda informada que combina las ventajas de la búsqueda en amplitud y la búsqueda guiada por heurística. Es ampliamente utilizado en videojuegos, sistemas de navegación y problemas de planificación de rutas debido a su eficiencia y garantía de encontrar el camino óptimo si existe.

2 Descripción del Problema

El problema consiste en un mapa representado por una rejilla de celdas cuadradas. Cada celda puede ser:

- Hierba (transitable, coste bajo)
- Agua (transitable, coste medio)
- Roca (transitable, coste alto)
- Obstáculo (no transitable)

El conejo (punto de inicio) debe encontrar el camino más corto hasta la zanahoria (punto objetivo), considerando los costes de movimiento y las calorías consumidas en cada tipo de terreno.

Los movimientos permitidos son en 8 direcciones: horizontal, vertical y diagonal. Los costes asociados son:

- Movimiento horizontal o vertical: 1
- Movimiento diagonal: 1.5

Además, cada tipo de terreno tiene un coste en calorías:

- Hierba: 2 calorías
- Agua: 4 calorías
- Roca: 6 calorías

El objetivo es implementar el algoritmo A* para encontrar el camino óptimo, considerando tanto el coste del movimiento como el gasto calórico.

3 Implementación del Algoritmo A*

3.1 Estructura de Datos

Para implementar el algoritmo A*, se han utilizado las siguientes estructuras de datos principales:

1. **PriorityQueue**: Para mantener la lista de nodos a explorar, ordenados por su valor f(n).
2. Lista 2D de booleanos: Para llevar un registro de los nodos visitados.
3. Lista 2D de flotantes: Para almacenar los costes g(n) de cada nodo.
4. Lista 2D de cadenas: Para representar el camino final.

Estas estructuras se inicializan en el método `buscar`:

```
pq = PriorityQueue()
visited = [[False for _ in range(self.m)] for _ in range(self.n)]
g_costs = [[float('inf') for _ in range(self.m)] for _ in range(self.n)]
camino = [['.'] * self.m for _ in range(self.n)]
```

3.2 Funciones Principales

3.2.1 init

```
def __init__(self, mapa: Mapa, origen: Casilla, destino: Casilla):
self.mapa = mapa
self.origen = origen
self.destino = destino
self.n = mapa.getAlto()
self.m = mapa.getAncho()
self.dx = [-1, -1, 0, 1, 1, 1, 0, -1]
self.dy = [0, 1, 1, 1, 0, -1, -1, -1]
self.moves_map = [
[8, 1, 2],
[7, 0, 3],
[6, 5, 4]
]
self.best_cost = float('inf')
self.best_moves = []
```

Esta función inicializa los atributos necesarios para el algoritmo A*. Se almacenan el mapa, las posiciones de origen y destino, y se definen las direcciones de movimiento posibles.

Complejidad temporal: O(1), ya que todas las operaciones son de tiempo constante.

3.2.2 calcular_coste_movimiento

```
def calcular_coste_movimiento(self, x1: int, y1: int, x2: int, y2: int) -> flo
if x1 == x2 or y1 == y2:  # Movimiento horizontal o vertical
return 1.0
else:  # Movimiento diagonal
return 1.5
```

Esta función calcula el coste de movimiento entre dos celdas adyacentes.

Complejidad temporal: O(1), ya que solo realiza una comparación simple.

3.2.3 generar_hijos

```
def generar_hijos(self, node: Tuple[int, int, float, float, List[int]], pq: Pr
x, y, g, _, path = node

for i in range(8):
nx, ny = x + self.dx[i], y + self.dy[i]
if 0 <= nx < self.n and 0 <= ny < self.m and self.mapa.getCelda(nx, ny) != 1:
new_g = g + self.calcular_coste_movimiento(x, y, nx, ny)
if not visited[nx][ny] or new_g < g_costs[nx][ny]:
h = heuristica(nx, ny)
f = new_g + h
new_path = path + [self.moves_map[self.dx[i] + 1][self.dy[i] + 1]]
pq.put((f, (nx, ny, new_g, h, new_path)))
g_costs[nx][ny] = new_g
```

Esta función genera los nodos hijos del nodo actual, calculando sus costes y añadiéndolos a la cola de prioridad si son válidos y mejores que los existentes.

Complejidad temporal: O(1) para cada hijo, ya que se generan 8 hijos como máximo.

3.2.4 buscar

```
def buscar(self, heuristica=None) -> Tuple[float, List[List[str]]]:
if heuristica is None:
heuristica = self.heuristica_cero

pq = PriorityQueue()
visited = [[False for _ in range(self.m)] for _ in range(self.n)]
g_costs = [[float('inf') for _ in range(self.m)] for _ in range(self.n)]
camino = [['.'] * self.m for _ in range(self.n)]

start_h = heuristica(self.origen.getFila(), self.origen.getCol())
pq.put((start_h, (self.origen.getFila(), self.origen.getCol(), 0, start_h, [])
g_costs[self.origen.getFila()][self.origen.getCol()] = 0

while not pq.empty():
_, (x, y, g, h, path) = pq.get()

if x == self.destino.getFila() and y == self.destino.getCol():
self.best_cost = g
self.best_moves = path
break

if visited[x][y]:
continue

visited[x][y] = True
self.generar_hijos((x, y, g, h, path), pq, visited, g_costs, heuristica)

if self.best_cost != float('inf'):
x, y = self.origen.getFila(), self.origen.getCol()
camino[x][y] = 'x'
for move in self.best_moves:
# ... (código para actualizar el camino)

return self.best_cost, camino
```

Esta es la función principal que implementa el algoritmo A*. Inicia la búsqueda desde el origen y continúa hasta encontrar el destino o agotar todos los nodos.

Complejidad temporal: O(N log N), donde N es el número de nodos en el grafo. La operación más costosa es la inserción y extracción de la cola de prioridad, que tiene un coste de O(log N) por operación.

3.3 Heurísticas Implementadas

Se han implementado tres heurísticas diferentes:

1. **Heurística de Manhattan**:

```
def heuristica_manhattan(self, x: int, y: int) -> float:
return abs(x - self.destino.getFila()) + abs(y - self.destino.getCol())
```

2. **Heurística Euclidea**:

```
def heuristica_euclidea(self, x: int, y: int) -> float:
return math.sqrt((x - self.destino.getFila())**2 + (y - self.destino.getCol())
```

3. **Heurística Cero** (para búsqueda de coste uniforme):

```
def heuristica_cero(self, x: int, y: int) -> float:
return 0
```

Todas estas heurísticas tienen una complejidad temporal de O(1).

4 Análisis del Algoritmo

4.1 Complejidad Temporal

La complejidad temporal del algoritmo A* depende principalmente de la heurística utilizada. En el peor caso, cuando la heurística no es informativa (como la heurística cero), el algoritmo puede explorar todos los nodos del grafo.

- Peor caso: $O(b^d)$, donde b es el factor de ramificación y d es la profundidad de la solución.
- Mejor caso: $O(d)$, cuando la heurística guía perfectamente al algoritmo hacia el objetivo.

En la práctica, con una buena heurística, el rendimiento suele estar entre estos dos extremos.

4.2 Complejidad Espacial

La complejidad espacial del A* es uno de sus principales inconvenientes, ya que necesita mantener todos los nodos generados en memoria.

- En el peor caso: $O(b^d)$, donde b es el factor de ramificación y d es la profundidad de la solución.

Este es el motivo por el que se sugieren alternativas como A* con profundización iterativa o A* con memoria limitada para problemas con espacios de estados muy grandes.

5 Pruebas y Resultados

[En esta sección, deberías incluir los resultados de tus pruebas, comparando las diferentes heurísticas implementadas. Puedes usar tablas o gráficos para mostrar el número de nodos explorados y el tiempo de ejecución para diferentes mapas y heurísticas.]

6 Implementación de A*ϵ

Aquí deberías explicar cómo has implementado el algoritmo A*ϵ, sus diferencias con el A* estándar, y mostrar algunos resultados comparativos.

7 Conclusiones

Resume los principales hallazgos de tu implementación y análisis. Discute las ventajas y desventajas de las diferentes heurísticas y del algoritmo A*ϵ en comparación con el A* estándar.

8 Referencias

1. Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A Formal Basis for the Heuristic Determination of Minimum Cost Paths. IEEE Transactions on Systems Science and Cybernetics, 4(2), 100-107.

2. Russell, S. J., & Norvig, P. (2020). Artificial Intelligence: A Modern Approach (4th ed.). Pearson.

3. Algorithmic Approaches to Playing Minesweeper. (n.d.). Retrieved [insert date] from https://dash.harvard.edu/bitstream/handle/1/14398552/BECERRA-SENIORTHESIS-2015.pdf