Proyecto 1

Informe de proyecto 1 de Inteligencia Artificial.

None

None

Table of contents

1. Integrantes

Nombre	Codigo
Vanessa Duran Mona	2359394
Jesus Estenllos Loaiza	2313021
Juan Damian Cuervo	2359413
Joan Esteban Villamil	2380466

2. Informe de Proyecto 1 - Inteligencia Artificial

2.1 Introducción

En este informe se detalla el desarrollo del proyecto 1 de la asignatura Inteligencia Artificial, el cual consiste en implementar un agente capaz de buscar la mejor ruta en un mapa utilizando algoritmos de Beam Search y Dynamic Weighting.

2.2 Desarrollo

El repositorio del proyecto es:

Enlace al repositorio

El proyecto está estructurado en varios archivos Python, cada uno encargado de una parte específica del algoritmo.

Requiere de la libreria `PySide6` como se puede ver en el archivo `requirements.txt`.

2.2.1 Beam Search

En el archivo `beam_search.py`, se implementan funciones como operadores de movimiento y de búsqueda.

Operadores de Movimiento

```
moverArriba(tupla_x_y)
moverAbajo(tupla_x_y)
moverIzquierda(posicion)
moverDerecha(posicion)
```

Operadores de Búsqueda

```
# Determina distancia manhattan
manhattan(pos1, pos2)
# Determina posibilidades de movimiento
isPosibleArriba(posicion)
isPosibleAbajo(posicion, n)
isPosibleIzquierda(posicion)
isPosibleDerecha(posicion, n)
isNodoMeta(meta, posicion)
# Determina el ancho del beam
calcular_beam_width(n, num_obstaculos)
# Expande un nodo generando sus hijos
expandir_nodo(nodo_actual, meta, obstaculos, n, indice_padre)
```

```
# Reconstruye el camino de inicio a meta
reconstruir_camino(closedList, indice_meta)
```

Funcion `calcular_beam_width`

```
def calcular_beam_width(n, num_obstaculos):
    """
    n: tamaño del tablero (nxn)
    num_obstaculos: cantidad de obstáculos en el tablero
    """
    densidad = num_obstaculos / (n * n)

    if n <= 10:
        base = 5
    elif n <= 30:
        base = 4
    elif n <= 50:
        base = 3
    else:
        base = 3

    # Ajustar por densidad de obstáculos
    if densidad > 0.4:  # Muchos obstáculos
        multiplicador = 1.5
    elif densidad > 0.2:  # Obstáculos moderados
        multiplicador = 1.2
    else:  # Pocos obstáculos
        multiplicador = 1.0

    beam_width = int(base * multiplicador)

    # Limitar entre valores razonables
    return max(3, min(beam_width, 10))
```

La funcion `calcular_beam_width` ajusta el ancho del beam basado en el tamaño del tablero y la densidad de obstáculos, buscando un balance entre exploración y eficiencia.

Funcion `expandir_nodo`

```
def expandir_nodo(nodo_actual, meta, obstaculos, n, indice_padre):
    """
    Expande un nodo generando todos sus sucesores válidos
    Retorna una lista de tuplas: (indice_padre, posicion, g_n, h_n, f_n)
    """
    posicion_actual = nodo_actual[0]
    g_actual = nodo_actual[2]

    sucesores = []

    # Definir movimientos posibles
    movimientos = [
        (isPosibleArriba(posicion_actual), moverArriba, "arriba"),
        (isPosibleAbajo(posicion_actual, n), moverAbajo, "abajo"),
        (isPosibleIzquierda(posicion_actual), moverIzquierda, "izquierda"),
        (isPosibleDerecha(posicion_actual, n), moverDerecha, "derecha")
    ]

    for es_posible, mover, direccion in movimientos:
```

```
        if es_posible:
            nueva_posicion = mover(posicion_actual)

            # Calcular costos
            costo_movimiento = 3 if nueva_posicion in obstaculos else 1
            g_n = g_actual + costo_movimiento
            h_n = manhattan(nueva_posicion, meta)
            f_n = g_n + h_n

            sucesores.append((indice_padre, nueva_posicion, g_n, h_n, f_n))

    return sucesores
```

La función `expandir_nodo` genera todos los sucesores válidos de un nodo, calculando sus costos asociados y retornándolos en una lista estructurada.

Funcion `reconstruir_camino`

```
def reconstruir_camino(closedList, indice_meta):
    """
    Reconstruye el camino desde el inicio hasta la meta
    siguiendo los índices de padres
    """
    camino = []
    indice_actual = indice_meta

    while indice_actual is not None:
        nodo = closedList[indice_actual]
        camino.append(nodo[0])  # Agregar la posición
        indice_actual = nodo[1]  # Moverse al padre

    camino.reverse()  # Invertir para tener el camino de inicio a meta
    return camino
```

La función `reconstruir_camino` sigue los índices de los nodos padres para reconstruir el camino desde el nodo meta hasta el nodo inicial.

Función Principal de Beam Search

```
def beam_search(n, inicio, meta, obstaculos):

    # Convertir obstáculos a set para búsquedas O(1)
    obstaculos_set = set(obstaculos) if not isinstance(obstaculos, set) else obstaculos

    beamWidth = calcular_beam_width(n, len(obstaculos))

    # closedList: [posicion, indice_padre, g_n, h_n]
    closedList = []

    visitados = set()
    visitados.add(inicio)

    h_inicial = manhattan(inicio, meta)
    closedList.append([inicio, None, 0, h_inicial])

    if inicio == meta:
        return [inicio]
```

```
    openList = [0]
    iteracion = 0
    max_iteraciones = n * n * 2

    # Detección de estancamiento
    mejor_h_previo = h_inicial
    iteraciones_sin_mejora = 0

    while openList and iteracion < max_iteraciones:
        iteracion += 1
        todos_sucesores = []

        # Expandir beam actual
        for indice_nodo in openList:
            nodo = closedList[indice_nodo]
            sucesores = expandir_nodo(nodo, meta, obstaculos_set, n, indice_nodo)

            for sucesor in sucesores:
                indice_padre, posicion, g_n, h_n, f_n = sucesor

                if posicion == meta:
                    closedList.append([posicion, indice_padre, g_n, h_n])
                    return reconstruir_camino(closedList, len(closedList) - 1)

                if posicion not in visitados:
                    todos_sucesores.append((posicion, indice_padre, g_n, h_n, f_n))
                    visitados.add(posicion)

        if not todos_sucesores:
            return None

        todos_sucesores.sort(key=lambda x: x[4])

        # Seleccionar los w mejores
        mejores_sucesores = todos_sucesores[:beamWidth]

        mejor_h_actual = min(s[3] for s in mejores_sucesores)
        if mejor_h_actual >= mejor_h_previo:
            iteraciones_sin_mejora += 1
            if iteraciones_sin_mejora > beamWidth * 2:
                return None  # Probablemente no hay camino
        else:
            iteraciones_sin_mejora = 0
            mejor_h_previo = mejor_h_actual

        # Actualizar openList
        openList = []
        for posicion, indice_padre, g_n, h_n, f_n in mejores_sucesores:
            closedList.append([posicion, indice_padre, g_n, h_n])
            openList.append(len(closedList) - 1)

    return None
```

La función `beam_search` implementa el algoritmo de búsqueda Beam Search, utilizando las funciones auxiliares definidas anteriormente para expandir nodos, calcular costos y reconstruir el camino.

Este algoritmo de Beam Search esta modificado, evita estancamientos evitando ciclos, y tiene mayor gasto en memoria.

En las variables `closedList` y `openList`, se gestionan los nodos expandidos y los candidatos para la siguiente expansión, respectivamente. El algoritmo continúa hasta encontrar la meta o agotar las posibilidades.

La funcion heuristica utilizada es la distancia Manhattan, que es adecuada para este tipo de problemas en una cuadrícula. Esta heurística es admisible debido a que el costo tomado para los `venenos` es 3, asegurando que manhattan siempre subestima el costo real.

Las variables `obstaculos_set`, `visitados`, y las condiciones de estancamiento ayudan a optimizar la búsqueda y evitar ciclos innecesarios.

Completitud y Optimalidad

Originalmente el algoritmo Beam Search no es completo ni óptimo debido a su naturaleza de limitar el número de nodos expandidos.

Esto no cambia en nuestra implementación, ya que seguimos limitando el número de nodos en cada nivel a un ancho fijo (beam width). Por lo tanto, el algoritmo podria no encontrar una solución incluso si existe una, y la solución encontrada puede no ser la óptima.

Complejidad Temporal y Espacial

Complejidad Temporal

La complejidad temporal del algoritmo Beam Search depende del ancho del beam (w) y la profundidad del árbol de búsqueda (d). En el peor de los casos, la complejidad temporal es $O(d \cdot w \log w)$, donde w es el ancho del beam y d es la profundidad máxima del árbol.

Debido a que tambien es posible usar la variable `max_iteraciones`, la complejidad temporal puede verse limitada por esta variable, resultando en una complejidad temporal de $O(n^2 \cdot w \log w)$ en el peor de los casos.

Ya hablando directamente en nuestra implementacion, el uso del conjunto `visitados` y `closedList` evita que el algoritmo revisite nodos reduciendo las iteraciones maximas a $O(\frac{n^2}{w})$.

Por lo tanto al realizar el despeje, la complejidad temporal final de nuestra implementación de Beam Search es $O(n^2 \log w)$.

Complejidad Espacial

La complejidad espacial original del algoritmo puro es $O(b \cdot w)$ debido a que se descartan nodos fuera del beam.

La implementacion actual utiliza `closedList` y `visitados`, lo que incrementa la complejidad espacial a $O(n^2)$ en el peor de los casos, ya que en el peor escenario se podrían almacenar todos los nodos del tablero.

2.2.2 Dynamic Weighting

En el archivo `dynamic.py`, se implementan solamente las funciones necesarias para el algoritmo de Dynamic Weighting.

Operadores de Búsqueda

```python
# Determina distancia manhattan
manhattan(pos1, pos2)
```

`# Determina posibilidades de movimiento`
`generar_sucesores(posicion, n, obstaculos)`

Función Principal de Dynamic Weighting

```python def dynamic_weighting_search(n, inicio, meta, obstaculos, epsilon=3): N = n * n open_list = []
heapq.heappush(open_list, (0, inicio, 0))
came_from = {inicio: None} g_score = {inicio: 0}

```
obstaculos_set = set(obstaculos) if not isinstance(obstaculos, set) else obstaculos

while open_list:
    f_actual, actual, depth = heapq.heappop(open_list)

    if actual == meta:
        camino = []
        while actual is not None:
            camino.append(actual)
            actual = came_from[actual]
        return camino[::-1]

    for sucesor in generar_sucesores(actual, n, obstaculos):
        costo = 3 if sucesor in obstaculos_set else 1
        tentative_g = g_score[actual] + costo
        if sucesor not in g_score or tentative_g < g_score[sucesor]:
            g_score[sucesor] = tentative_g
            h = manhattan(sucesor, meta)

            f = tentative_g + h + epsilon * (1 - (depth / N)) * h

            heapq.heappush(open_list, (f, sucesor, depth + 1))
            came_from[sucesor] = actual
return None
```

```

La función `dynamic_weighting_search` implementa el algoritmo de Dynamic Weighting, usando las funciones auxiliares definidas anteriormente para generar sucesores y calcular la distancia Manhattan.

En las variables `open_list`, `came_from`, y `g_score`, se gestionan los nodos a explorar, el seguimiento de los nodos padres, y los costos acumulados respectivamente. El algoritmo continúa hasta encontrar la meta o agotar las posibilidades.

La funcion heuristica utilizada es la distancia Manhattan, que es adecuada para este tipo de problemas en una cuadrícula. Esta heurística es admisible debido a que el costo tomado para los `venenos` es 3, asegurando que manhattan siempre subestima el costo real.

Se utiliza la librería `heapq` para manejar la `open_list` como una cola de prioridad, para tener siempre el nodo con el menor costo f en la parte superior.

## Completitud y Optimalidad

El algoritmo de Dynamic Weighting es completo, ya que explora todos los nodos posibles hasta encontrar la solución, siempre y cuando exista una solución. Sin embargo, no es óptimo debido a la naturaleza del peso dinámico que puede llevar a seleccionar caminos subóptimos.

Esto se debe a que $\epsilon$ ejerce un factor multiplicativo causando lo que en una busqueda $A*$ seria una heurisitica no admisible.

# Complejidad Temporal y Espacial

## Complejidad Temporal

Al ser una variante del algoritmo A*, la cantidad de nodos expandidos por ser una cuadricula de tamaño $n \times n$ es $O(n^2)$.

Debido a que cada operación de inserción y extracción en la cola de prioridad (heap) tiene un costo de $O(\log m)$, donde $m$ es el número de nodos en la cola, la complejidad temporal total del algoritmo Dynamic Weighting es $O(n^2 \log n)$.

## Complejidad Espacial

La complejidad espacial del algoritmo Dynamic Weighting es $O(n^2)$ en el peor de los casos, ya que en el peor escenario se podrían almacenar todos los nodos del tablero en las estructuras de datos `open_list`, `came_from`, y `g_score`.

---

La implementacion no modifica el funcionamiento de Dynamic Weighting.