Laberinto búsqueda en amplitud

Universidad de Guadalajara, Centro Universitario de Ciencias Exactas e Ingenierías.

Autores:

Juan José Salazar Villegas - 215661291

Juan Emmanuel Fernández de Lara Hernández - 220286571

21/04/2023

Carrera: Ingeniería En Computación (INCO)

Asignatura: Inteligencia Artificial I

Maestro: Diego Alberto Oliva Navarro

Origen del informe: Guadalajara Jalisco México

Ciclo: 2023 A

Sección: D05 Práctica: 2

Objetivo:

Implementar los algoritmos de búsqueda no informada en problemas de prueba para poder comparar su desempeño.

Implementación:

Desarrollar un programa que encuentre la mejor solución a los problemas planteados, usando los algoritmos de búsqueda no informada vistos en clase (búsqueda en profundidad, búsqueda en amplitud, etc).

Se debe hacer el planteamiento de los problemas, en base a los conceptos: del espacio de estados las acciones (función sucesor), test objetivo y el costo del camino. Con base a esto, se definen los árboles y la estrategia de búsqueda.

Problemas:

Considere el problema de encontrar el camino más corto entre dos puntos en un plano de dos dimensiones. Dentro del plano se encuentran diversos obstáculos con formas geométricas distintas (Fig 2). El punto origen es un circulo, mientras que el destino es una estrella, ambos son de color rojo. En este caso el espacio de estados corresponde al conjunto de posiciones (x,y) presentes en el plano.

Se deben implementar ambos algoritmos de búsqueda no informada que permitan encontrar de forma automática la mejor trayectoria entre ambos puntos.

El plano puede ser distinto al de la Fig 2. sin embargo debe representar complejidad para su solución.

El estado inicial debe ser aleatorio.

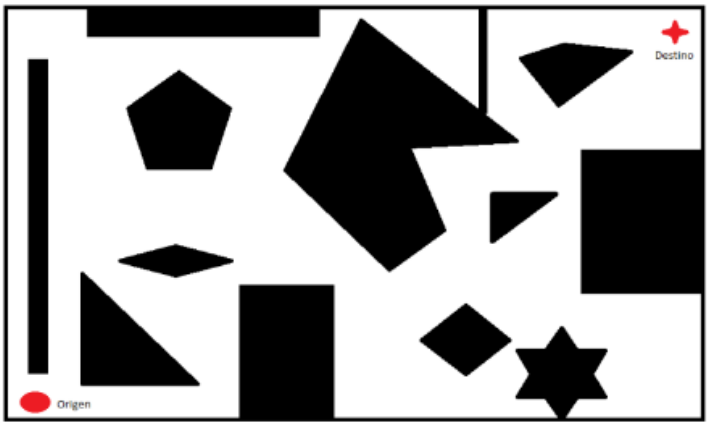


Fig.3 Plano con obstáculos poligonales.

Desarrollo

Para el desarrollo de esta práctica necesitaremos entender gráficamente cada estado es procesado, así como las acciones que se tomarán por ello partimos de finiendo los siguientes puntos:

- Estados
- Acciones
- Prueba de meta
- Costo del camino

Donde cada uno de estos le dará a nuestro agente entendiendo el ambiente que lo afecta partimos a contestar dichos puntos siendo los siguientes:

- **Estados:** Localizaciones completas de las piezas, **Cualquier estado puede ser inicial**
- **Acciones:** (Función sucesor): Mover el negro a la izquierda, derecha, arriba, abajo.
- **Test objetivo:** Comprobar si se llegó al estado objetivo.
- **Costo del camino:** 1 por movimiento.

Con esto definido podremos entender qué puntos debemos tratar a lo largo de esta práctica.

Ahora teniendo en cuenta los estados definidos y las colisiones con las coordenadas podremos entender que sigue la misma lógica que en la práctica anterior con el puzzle siendo un punto de inicio y final establecido y conforme se seleccione la búsqueda se procede a dar solución.

Codificación e implementación

Definición del estado objetivo (Generación de Terreno y Aleatoriedad):

```
def generar_terreno():
    # Definimos la matriz de 10x10 que representa el plano
    matriz = [['⬜' for i in range(10)] for j in range(10)]

    # Agregamos obstáculos aleatorios al plano
    for i in range(30):
        x = random.randint(0, 9)
        y = random.randint(0, 9)
        matriz[x][y] = '🪨'

    # Definimos las coordenadas del punto origen y destino
    origen = (random.randint(0, 9), random.randint(0, 9))
    matriz[origen[0]][origen[1]] = '🔴'
    destino = (random.randint(0, 9), random.randint(0, 9))
    matriz[destino[0]][destino[1]] = '★'

    return matriz, origen, destino
```

Comenzamos definiendo nuestro terreno donde establecemos el punto de origen y destino esto por una matriz de 10x10 teniendo un total de 100 espacios para generar desde los puntos de origen y destino así como los espacios que harán colisión para que nuestras búsquedas generen el camino.

Al igual que la práctica previa estableceremos una función para generar el terreno aleatorio para poner en práctica cada una de las búsquedas y medir su rendimiento, esta opción estará disponible durante la ejecución del mismo.

Obtención de Nodos vecinos de laberinto:

```
# Función para obtener los vecinos de un nodo dado
def obtener_vecinos(nodo):
    x, y = nodo
    vecinos = []
    if x > 0 and matriz[x-1][y] != '🌑':   # nodo de arriba
        vecinos.append((x-1, y))
    if x < 9 and matriz[x+1][y] != '🌑':  # nodo de abajo
        vecinos.append((x+1, y))
    if y > 0 and matriz[x][y-1] != '🌑':  # nodo de la izquierda
        vecinos.append((x, y-1))
    if y < 9 and matriz[x][y+1] != '🌑':  # nodo de la derecha
        vecinos.append((x, y+1))
    return vecinos
```

Ya definiendo nuestro Mundo es hora de saber que nos rodea para posterior saber por qué camino continuar sin importar la busqueda, con ello tomaremos los nodos de cada dirección ya definidos (Arriba, abajo, izquierda, derecha), tomando las coordenadas tomaremos los “nodos vecinos” que nos obstruyen en el camino para posteriormente.

Esta es una función se toma como entrada el nodo de la matriz y posterior nos devolverá una lista de sus vecinos que no son obstáculos ('🌑').

Teniendo en cuenta la posición x e y del nodo de entrada inicializamos una nueva lista vacía llamada vecinos la cual nos servirá para almacenar los vecinos válidos.

Luego, se verifica si el nodo de arriba (x-1) es válido y no es un obstáculo ('🌑'). Si cumple estas condiciones, se agrega a la lista de vecinos. Este proceso se repite para los nodos de abajo (x+1), izquierda (y-1) y derecha (y+1).

Finalmente, la función devuelve la lista de vecinos.

Es importante destacar que esta función asume que la matriz es una lista de listas (o una matriz bidimensional) donde cada elemento es una posición en el plano, y los obstáculos ('🌑') están representados por valores específicos en la matriz. Si esto no se cumple, la función puede no funcionar correctamente.

Búsqueda por amplitud

```python
# Función para realizar la búsqueda en amplitud
def busqueda_en_amplitud(origen, destino):
    visitados = set()
    cola = Queue()
    cola.put((origen, 0, []))
    while not cola.empty():
        nodo, distancia, camino = cola.get()
        if nodo == destino:
            return camino + [nodo]
        if nodo not in visitados:
            visitados.add(nodo)
            for vecino in obtener_vecinos(nodo):
                cola.put((vecino, distancia+1, camino+[nodo]))
    return None
```

Esta función implementa el algoritmo de búsqueda en amplitud (BFS) para encontrar el camino más corto desde un nodo de origen hasta un nodo de destino en una matriz.

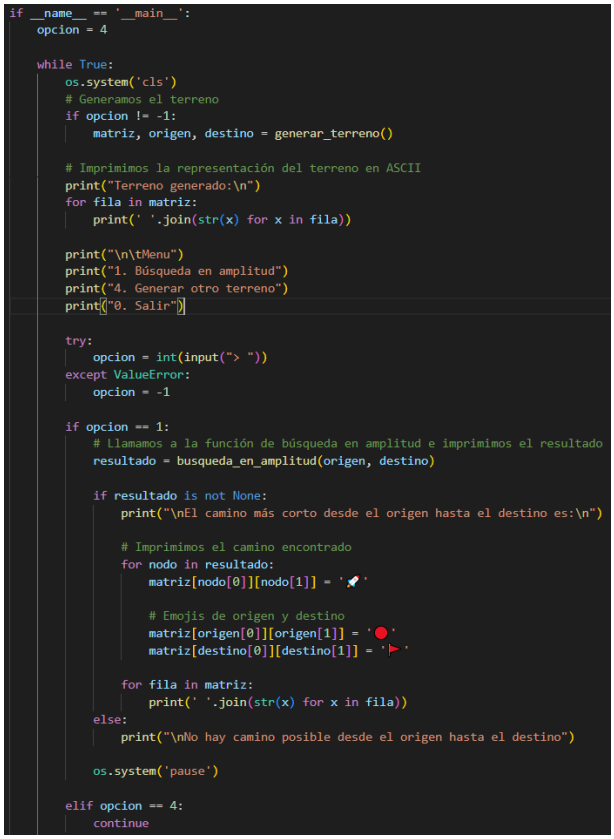
Esta función tomará como entrada el nodo de origen y destino. Al iniciar se crea un conjunto vacío llamado visitados el cual almacena los nodos que ya han sido visitados durante la búsqueda. Además de crear una cola vacía usando la clase Queue.

Luego, se agrega el nodo de origen a la cola con una distancia de 0 y un camino vacío. El ciclo while se ejecutará hasta que la cola esté vacía.

En cada iteración del ciclo, se extrae el primer elemento de la cola (nodo, distancia, camino). Si el nodo es igual al nodo de destino, se devuelve el camino encontrado (que es el camino más corto desde el nodo de origen al nodo de destino) más el nodo de destino.

Si el nodo actual no está en el conjunto de nodos visitados, se agrega a este conjunto y se obtienen los vecinos válidos del nodo actual utilizando la función obtener_vecinos(). Para cada vecino, se agrega a la cola con una distancia incrementada en 1 y se agrega el nodo actual al camino.Si no se encuentra un camino desde el nodo de origen al nodo de destino, la función devuelve None.

Main (Menu)



```python
if __name__ == '__main__':
    opcion = 4

    while True:
        os.system('cls')
        # Generamos el terreno
        if opcion != -1:
            matriz, origen, destino = generar_terreno()

        # Imprimimos la representación del terreno en ASCII
        print("Terreno generado:\n")
        for fila in matriz:
            print(' '.join(str(x) for x in fila))

        print("\n\tMenu")
        print("1. Búsqueda en amplitud")
        print("4. Generar otro terreno")
        print("0. Salir")

        try:
            opcion = int(input("> "))
        except ValueError:
            opcion = -1

        if opcion == 1:
            # Llamamos a la función de búsqueda en amplitud e imprimimos el resultado
            resultado = busqueda_en_amplitud(origen, destino)

            if resultado is not None:
                print("\nEl camino más corto desde el origen hasta el destino es:\n")

                # Imprimimos el camino encontrado
                for nodo in resultado:
                    matriz[nodo[0]][nodo[1]] = '🚀'

                    # Emojis de origen y destino
                    matriz[origen[0]][origen[1]] = '🔴'
                    matriz[destino[0]][destino[1]] = '🚩'

                for fila in matriz:
                    print(' '.join(str(x) for x in fila))
            else:
                print("\nNo hay camino posible desde el origen hasta el destino")

            os.system('pause')

        elif opcion == 4:
            continue
```

En esta sección del código solo establecemos el menú principal y se da funcionalidad a la opción de “Búsqueda en amplitud” evaluando la entrada según el resultado de la búsqueda esto será imprimiendo el camino más corto esto si es posible.

Primero, se verifica si la variable resultado (que es el camino más corto encontrado por la función de búsqueda en amplitud) no es None. Si el resultado no es None, significa que se encontró un camino desde el nodo de origen hasta el nodo de destino.

Luego, se imprime un mensaje indicando que se encontró un camino y se procede a actualizar la matriz para resaltar el camino. Para cada nodo en el camino, se cambia su valor en la matriz por el emoji de un cohete ('🚀') para indicar que forma parte del camino más corto. También se cambia el valor del nodo de origen a un emoji de un punto rojo ('🔴') y el valor del nodo de destino a un emoji de una bandera ('🚩') para indicar su ubicación en la matriz.

Finalmente, se imprime la matriz actualizada en la consola para mostrar el camino más corto resaltado en la matriz. Si no se encontró un camino desde el nodo de origen hasta el nodo de destino, se imprime un mensaje indicando que no hay un camino posible.

Resultados de Búsquedas

Amplitud

Como observamos la búsqueda en amplitud explora todos los nodos vecinos de un nodo dado antes de pasar al nodo vecino siguiente utilizando una cola que almacena los nodos que deben procesarse. Para la implementación de la búsqueda por amplitud se puede utilizar una función que toma como entrada el nodo de origen y el nodo de destino. La función inicialmente coloca el nodo de origen en la cola y crea una lista de nodos visitados vacía. A medida que se procesan los nodos de la cola, se agregan sus vecinos a la cola y se actualiza la lista de nodos visitados.

La función continúa procesando nodos de la cola hasta que se encuentra el nodo de destino o hasta que la cola esté vacía. Si se encuentra el nodo de destino, se devuelve una lista de nodos que conforman el camino más corto desde el nodo de origen hasta el nodo de destino, utilizando la información de los nodos visitados. Si no se encuentra el nodo de destino, la función devuelve None para indicar que no se encontró un camino.

La búsqueda en amplitud es un algoritmo eficiente y completo para encontrar el camino más corto en una matriz. Sin embargo, puede requerir mucho espacio de almacenamiento para almacenar la cola y la lista de nodos visitados, por lo que no es la mejor opción en todas las situaciones.

Conclusión:

Juan José Salazar Villegas:

Dentro de esta práctica pude encontrar similitudes a la realizada previamente con el puzzle esto al seguir la misma lógica al tener un espacio generado donde se tienen obstrucciones las cuales hacen que cada nodo sea analizado y de un camino para dar un orden final que para este caso aplica a como se establece el camino final y dar el camino más corto si es que este existe, esto por la posibilidad de que el objetivo final esté completamente obstruido y no se tenga ningún acceso.

En este caso la búsqueda permite resolver problemas de caminos de manera eficiente sin importar el mundo generado, Además de que tanto la búsqueda en profundidad como la búsqueda en amplitud son estrategias utilizadas para resolver problemas en Inteligencia Artificial. La búsqueda en profundidad explora el árbol de búsqueda en profundidad y busca alcanzar la solución más cercana al nodo raíz rápidamente, mientras que la búsqueda en amplitud explora el árbol de búsqueda en orden de nivel y busca encontrar la solución óptima de forma completa. En general, la elección entre ambas estrategias dependerá de las características del problema que se desea resolver.

Juan Emmanuel Fernández de Lara Hernández:

En esta práctica se pudo encontrar una similitud con la realizada previamente con el puzzle, en cuanto a la lógica que se sigue para resolver el problema. En ambos casos, se trata de buscar una solución en un espacio generado con obstrucciones, analizando cada nodo y estableciendo un camino final que permita llegar al objetivo de la manera más eficiente posible. La búsqueda permite resolver problemas de caminos de manera eficiente sin importar las características del mundo generado. En este sentido, tanto la búsqueda en profundidad como la búsqueda en amplitud son estrategias comunes utilizadas para resolver problemas en Inteligencia Artificial. La elección de la estrategia dependerá de las características del problema específico que se esté tratando de resolver. En resumen, la práctica permitió una mejor comprensión de cómo las estrategias de búsqueda pueden ser aplicadas para resolver problemas de manera eficiente en Inteligencia Artificial.