Algoritmo de Dijkstra

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

Autores:

Juan José Salazar Villegas - 215661291

Juan Emmanuel Fernández de Lara Hernández - 220286571

21/05/2023

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: 4 - Bonus

Objetivo:

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

Implementación:

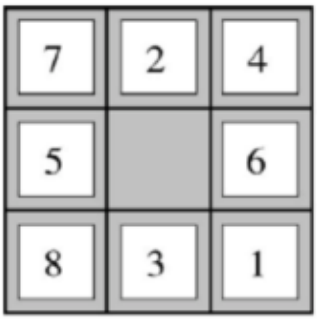
Desarrollar un programa que encuentre la mejor solución a los problemas planteados, usando los algoritmos de búsqueda no informada vistos en clase (Algoritmo de Dijkstra).

Se debe hacer el planteamiento de los problemas, en base a los conceptos de primero mejor de los programas desarrollados en la práctica 2.

Problemas:

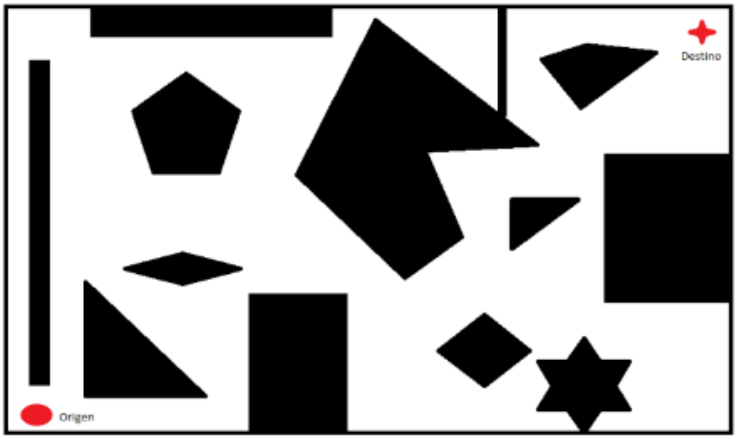
Siguiendo la lógica de cada problema realizado previamente se establecerá el estrategia de primero Mejor para observar el desempeño de los diferentes problemas previamente establecidos

Recordando que el estado inicial debe ser aleatorio.



Estado Inicial

Estado Objetivo



Desarrollo

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

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

Como lo observamos previamente estos tendrán diferentes ambientes donde nuestros agentes serán afectados desde los estados iniciales así como sus acciones de movimiento así como el test objetivo y de ser necesario el costo por del camino para no detallar de manera directa obviamos estos puntos tratados en los reportes previos para pasar directamente a la codificación.

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 en ambos problemas:

Puzzle

```python
# Función para comprobar si se ha llegado al estado objetivo
def es_objetivo(estado):
  return estado == [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
```

En este apartado definiremos el estado objetivo final de nuestro puzzle esto derivado a que cada uno de ellos al correrlo dará un acomodo aleatorio así al implementar la búsqueda en amplitud y en profundidad de la solución automáticamente.

Laberinto

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

```python
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
```

Recapitulando la definición de 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 generan el camino.

Con estas bases planteadas pasaremos a la comprensión detallada del código primero es mejor en ambos código así como sus resultados.

Primero el mejor:

Puzzle

```python
# Función para calcular el costo de moverse desde el estado actual al siguiente estado
def costo_movimiento(estado_actual, siguiente_estado):
    return 1

# Función para buscar la solución por el algoritmo de Dijkstra
def busqueda_dijkstra(estado_inicial):
    objetivo = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
    distancia = {}  # Distancia acumulada desde el estado inicial hasta el estado actual
    camino = {}  # Camino desde el estado inicial hasta el estado actual

    estado_inicial_str = str([[str(digito) for digito in fila] for fila in estado_inicial])
    objetivo_str = str([[str(digito) for digito in fila] for fila in objetivo])

    distancia[estado_inicial_str] = 0
    camino[estado_inicial_str] = []

    while estado_inicial_str != objetivo_str:
        sucesores_estado = sucesores(estado_inicial)

        for sucesor in sucesores_estado:
            sucesor_str = str([[str(digito) for digito in fila] for fila in sucesor])
            costo = costo_movimiento(estado_inicial, sucesor)

            if sucesor_str not in distancia or distancia[estado_inicial_str] + costo < distancia[sucesor_str]:
                distancia[sucesor_str] = distancia[estado_inicial_str] + costo
                camino[sucesor_str] = camino[estado_inicial_str] + [estado_inicial]

        del distancia[estado_inicial_str]

        min_distancia = float('inf')

        for estado, dist in distancia.items():
            if dist < min_distancia:
                min_distancia = dist
                estado_inicial_str = estado

    return camino[objetivo_str] + [objetivo]
```

El algoritmo de Dijkstra se utilizará para encontrar el camino más corto desde el estado inicial hasta el estado objetivo, considerando cada estado como un nodo en un grafo y las transiciones entre estados como aristas ponderadas.

La funcionalidad del código es el siguiente:

Comenzamos estableciendo nuestros estados y llegar a nuestro estado objetivo con la siguiente función:

sucesores(estado) nos ayudará a generar nuestros sucesores válidos de un estado dado, Itera sobre la matriz del estado y busca el espacio en blanco (representado por el número 0). Luego, para cada posición del espacio en blanco, verifica si se puede mover hacia arriba, abajo, izquierda o derecha.

Si es posible, realiza el intercambio y agrega el nuevo estado a la lista de sucesores. Al final, retorna la lista de sucesores generada.

```python
# Función para generar los sucesores de un estado dado
def sucesores(estado):
    sucesores = []
    # Buscar el espacio en blanco
    for i in range(3):
        for j in range(3):
            if estado[i][j] == 0:
                # Mover hacia arriba
                if i > 0:
                    sucesor = [fila[:] for fila in estado]
                    sucesor[i][j], sucesor[i-1][j] = sucesor[i-1][j], sucesor[i][j]
                    sucesores.append(sucesor)
                # Mover hacia abajo
                if i < 2:
                    sucesor = [fila[:] for fila in estado]
                    sucesor[i][j], sucesor[i+1][j] = sucesor[i+1][j], sucesor[i][j]
                    sucesores.append(sucesor)
                # Mover hacia la izquierda
                if j > 0:
                    sucesor = [fila[:] for fila in estado]
                    sucesor[i][j], sucesor[i][j-1] = sucesor[i][j-1], sucesor[i][j]
                    sucesores.append(sucesor)
                # Mover hacia la derecha
                if j < 2:
                    sucesor = [fila[:] for fila in estado]
                    sucesor[i][j], sucesor[i][j+1] = sucesor[i][j+1], sucesor[i][j]
                    sucesores.append(sucesor)
    return sucesores
```

Ahora daremos el costo de moverse para ello entra nuestra función:

costo_movimiento(estado_actual, siguiente_estado) Está calculará el costo de moverse desde el estado actual al siguiente estado. En este caso, el costo es constante y se establece en 1 para todos los movimientos.

Entramos al algoritmo esperado:

busqueda_dijkstra(estado_inicial)

Esta función realiza la búsqueda de la solución utilizando el algoritmo de Dijkstra. Comienza iniciando un diccionario a distancia para almacenar la distancia acumulada desde el estado inicial hasta cada estado visitado, y un diccionario camino para almacenar el camino desde el estado inicial hasta cada estado visitado.

Se convierten los estados inicial y objetivo en cadenas de texto para utilizarlos como claves en los diccionarios distancia y camino.

```python
# Función para buscar la solución por el algoritmo de Dijkstra
def busqueda_dijkstra(estado_inicial):
    objetivo = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
    distancia = {}  # Distancia acumulada desde el estado inicial hasta el estado actual
    camino = {}  # Camino desde el estado inicial hasta el estado actual

    estado_inicial_str = str([[str(digito) for digito in fila] for fila in estado_inicial])
    objetivo_str = str([[str(digito) for digito in fila] for fila in objetivo])

    distancia[estado_inicial_str] = 0
    camino[estado_inicial_str] = []

    while estado_inicial_str != objetivo_str:
        sucesores_estado = sucesores(estado_inicial)

        for sucesor in sucesores_estado:
            sucesor_str = str([[str(digito) for digito in fila] for fila in sucesor])
            costo = costo_movimiento(estado_inicial, sucesor)

            if sucesor_str not in distancia or distancia[estado_inicial_str] + costo < distancia[sucesor_str]:
                distancia[sucesor_str] = distancia[estado_inicial_str] + costo
                camino[sucesor_str] = camino[estado_inicial_str] + [estado_inicial]

        del distancia[estado_inicial_str]

        min_distancia = float('inf')

        for estado, dist in distancia.items():
            if dist < min_distancia:
                min_distancia = dist
                estado_inicial_str = estado

    return camino[objetivo_str] + [objetivo]
```

Ahora entramos al bucle, donde se generan los sucesores del estado actual y se actualizan las distancias y los caminos si se encuentra un camino más corto hacia un sucesor o si el sucesor no ha sido visitado anteriormente. Una vez que se han procesado todos los sucesores, se elimina el estado actual de los diccionarios distancia y camino y se busca el estado no visitado con la menor distancia acumulada para convertirlo en el nuevo estado actual.

Nuestro bucle continuará hasta que se alcance el estado objetivo. En ese momento, se devuelve el camino desde el estado inicial hasta el estado objetivo.

Una vez que se han procesado todos los sucesores, se elimina el estado actual de los diccionarios distancia y camino. Luego se busca el estado no visitado con la menor distancia acumulada para convertirlo en el nuevo estado actual. Finalmente, cuando se alcanza el estado objetivo, se retorna el camino desde el estado inicial hasta el objetivo concatenado con el estado objetivo mismo.

Laberinto

```python
# 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

# Función para realizar la búsqueda utilizando el algoritmo de Dijkstra
def busqueda_dijkstra(origen, destino):
    distancia = {nodo: sys.maxsize for nodo in [(x, y) for x in range(10) for y in range(10)]}
    distancia[origen] = 0
    visitados = set()
    camino = {nodo: [] for nodo in [(x, y) for x in range(10) for y in range(10)]}
    while True:
        nodo_actual = None
        min_distancia = sys.maxsize
        for nodo in [(x, y) for x in range(10) for y in range(10)]:
            if nodo in visitados:
                continue
            if distancia[nodo] < min_distancia:
                nodo_actual = nodo
                min_distancia = distancia[nodo]
        if nodo_actual is None:
            break
        visitados.add(nodo_actual)
        if nodo_actual == destino:
            break
        for vecino in obtener_vecinos(nodo_actual):
            distancia_vecino = distancia[nodo_actual] + 1
            if distancia_vecino < distancia[vecino]:
                distancia[vecino] = distancia_vecino
                camino[vecino] = camino[nodo_actual] + [nodo_actual]
    return camino[destino] if camino[destino] else None
```

Partimos a analizar ahora el siguiente problema presentado observando que la lógica se repite adaptándose a la problemática con el mismo análisis por camino. Con un enfoque de búsqueda exhaustiva con esta exploración por nodos en un orden de distancia estimada desde el nodo de origen. Tomando al igual los nodos no visitados y asignados a cada nodo una distancia provisional que se actualiza a medida que se exploran los nodos adyacentes.

La funcionalidad del código es el siguiente:

Partimos de obtener nuestro entorno gracias a la función de **obtener_vecinos(nodo)** recibe las coordenadas de un nodo y devuelve una lista de sus vecinos válidos. Se verifica si el nodo de arriba, abajo, izquierda y derecha del nodo actual no está fuera de los límites de la matriz y no es un obstáculo ('🌑'). Si el vecino cumple con estas condiciones, se agrega a la lista de vecinos.

Ahora adaptamos nuestra búsqueda dijkstra con nuestra función:

busqueda_dijkstra(origen, destino) Con ella encontramos nuestro camino más corto desde el nodo de origen hasta el nodo de destino en el terreno de juego.

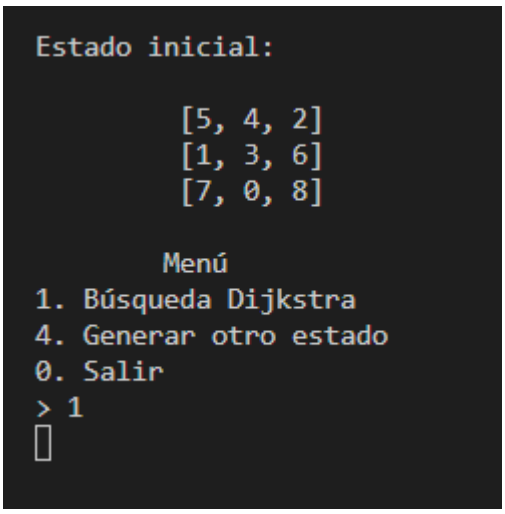
Comenzando iniciando un diccionario a distancia con todas las distancias establecidas en un valor máximo. Luego, establece la distancia del nodo de origen en 0. El conjunto visitado se utiliza para almacenar los nodos visitados durante la búsqueda. El diccionario camino se utiliza para almacenar el camino desde el nodo de origen hasta cada nodo.

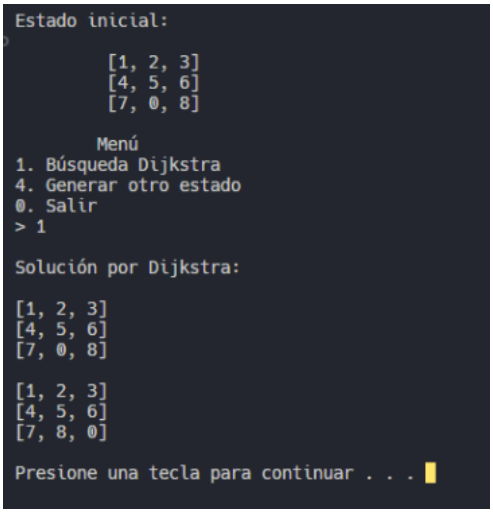
En el bucle principal, se selecciona el nodo con la distancia mínima no visitada y se actualizan las distancias y caminos de sus vecinos si se encuentra un camino más corto. El algoritmo continúa hasta que se visiten todos los nodos o hasta que se alcance el nodo de destino. Al finalizar, la función retorna el camino hasta el nodo de destino si se encuentra, de lo contrario, retorna None.

Como observamos el algoritmo sigue la misma secuencia para todos los problemas presentados solo adaptándolo a las necesidades de la misma, teniendo el común denominador el cómo se buscan en cada uno de los nodos con el diccionario referenciado a la distancia.

Resultados de implementación

Puzzle





para este resultado se esperaba una respuesta como las previas realizadas, pero está al dar la busqueda con nuestro algoritmo este nunca dio una respuesta temprana ni final ya que este solo consumía recursos del conmutador mostrando que funcionaba pero al dejarlo finalmente no dio respuesta alguna o solución a nuestro problema. Posteriormente se buscó generar otro estado donde no tuviera tanta complejidad y al encontrar uno este dio una solución rápida, dándonos a entender que entre más complejidad del problema más recursos y tiempo será para dar solución que no es garantizada.

Laberinto

A diferencia de la problemática anterior este sin importar la complejidad puede dar solución rápida sin importar el terreno generado, donde podremos entender que dentro del terrenos y nodos visitados se va dando el camino hasta encontrar el camino más corto.

Conclusión:

Juan José Salazar Villegas:

La implementación del algoritmo de Dijkstra en cada problemática nos muestra una manera eficiente de encontrar el camino más corto en un grafo ponderado. Utilizando una cola de prioridad para seleccionar el siguiente nodo a explorar, actualizando las distancias a medida que se encuentran caminos más cortos.

Al analizar este código y comprender el funcionamiento del algoritmo, puedo concluir que el algoritmo de Dijkstra es una herramienta poderosa para resolver problemas de búsqueda de caminos más cortos. Su enfoque basado en pesos y el uso de una estructura de datos eficiente como la cola de prioridad permite encontrar soluciones óptimas en diferentes contextos, como la planificación de rutas en mapas, la optimización de redes de transporte, entre otros.

Además, la representación del terreno de juego mediante una matriz bidimensional muestra cómo el algoritmo puede aplicarse a escenarios específicos, adaptándose a diferentes situaciones y considerando obstáculos en el camino.

El algoritmo de Dijkstra ofrece una solución elegante y eficiente para encontrar el camino más corto en un grafo ponderado, y su comprensión y aplicación pueden ser de gran utilidad en diversas áreas, desde la informática hasta la logística y la planificación.

Juan Emmanuel Fernández de Lara Hernández:

Al haber utilizado el algoritmo de Dijkstra para resolver los problemas del laberinto y el 8-puzzle, se puede concluir que, si bien es un enfoque válido, no es el más eficiente en términos de tiempo de ejecución para el problema del 8-puzzle en particular.

El algoritmo de Dijkstra se basa en encontrar el camino más corto en un grafo ponderado, explorando todos los nodos y actualizando las distancias a medida que se encuentran caminos más cortos. Sin embargo, en el caso del 8-puzzle, donde el espacio de búsqueda es bastante grande, este enfoque puede resultar lento y poco

eficiente en comparación con otros algoritmos más especializados, como A* o búsqueda informada.

Los resultados obtenidos al aplicar el algoritmo de Dijkstra al problema del 8-puzzle demuestran que el tiempo requerido para encontrar una solución puede ser considerablemente mayor en comparación con otros algoritmos más eficientes. Esto se debe a la naturaleza exhaustiva de Dijkstra, que explora todos los caminos posibles sin aprovechar las características específicas del problema. Aunque el algoritmo de Dijkstra sigue siendo una herramienta valiosa para encontrar caminos más cortos en problemas de búsqueda de rutas en general, su eficiencia en el contexto del 8-puzzle es limitada.