Primero el mejor

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: 3

Objetivo:

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

Implementación:

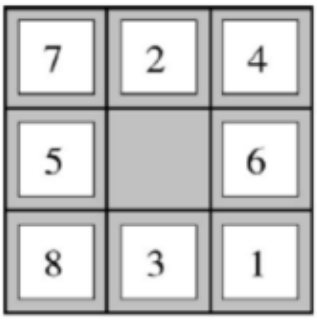
Desarrollar un programa que encuentre la mejor solución a los problemas planteados, usando los algoritmos de búsqueda no informada vistos en clase (Primero Mejor).

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:

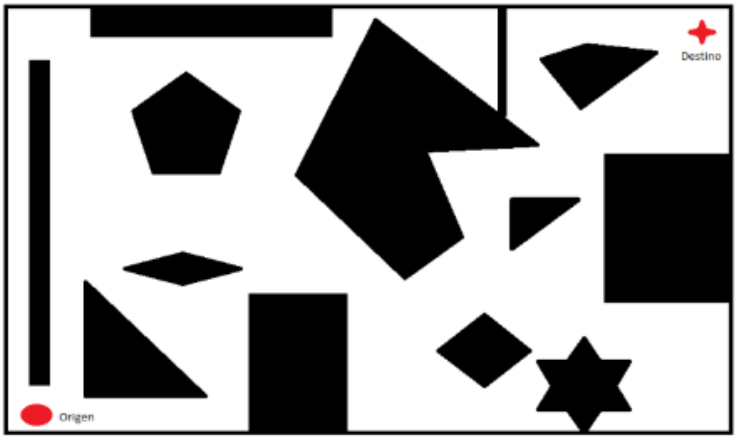
Siguiendo la lógica de cada problema realizado previamente se establecerá el estrategia de primero Mejor para observar el desempeño de los diferentes problemas previamente establecidos

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 generen 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 la distancia Manhattan entre dos puntos en una matriz 3x3
def distancia_manhattan(punto1, punto2):
    x1, y1 = punto1
    x2, y2 = punto2
    return abs(x1 - x2) + abs(y1 - y2)


# Función para buscar la posición de un elemento en el estado
def buscar_posicion(estado, elemento):
    for i in range(3):
        for j in range(3):
            if estado[i][j] == elemento:
                return (i, j)


# Función para buscar la solución por el algoritmo "Primero el mejor"
def busqueda_primero_el_mejor(estado_inicial):
    objetivo = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
    cola_prioridad = []
    explorado = set()

    # Calcular la heurística para el estado inicial
    heuristica_inicial = distancia_manhattan(buscar_posicion(estado_inicial, 0), (2, 2))
    cola_prioridad.append((heuristica_inicial, estado_inicial, []))

    while cola_prioridad:
        _, estado_actual, camino = cola_prioridad.pop(0)  # Extraer el estado con menor heurística
        explorado.add(str(estado_actual))

        if estado_actual == objetivo:
            return camino + [estado_actual]

        for sucesor in sucesores(estado_actual):
            if str(sucesor) not in explorado:
                heuristica_sucesor = distancia_manhattan(buscar_posicion(sucesor, 0), (2, 2))
                cola_prioridad.append((heuristica_sucesor, sucesor, camino + [sucesor]))

        # Ordenar la cola de prioridad según la heurística
        cola_prioridad.sort(key=lambda x: x[0])

    return None
```

Para la implementación del algoritmo “Primero el mejor” utilizaremos la heurística de distancia Manhattan la cual nos ayudará a la estimación de proximidad de un estado al estado objetivo así se expanden los sucesores del estado actual en función de esta heurística.

La funcionalidad del código es el siguiente:

Comenzaremos explicando nuestro algoritmo de búsqueda “Primero es mejor” siendo la función: “**busqueda_primero_el_mejor”** la cual toma como argumento el estado inicial del problema y devuelve una secuencia de movimientos que lleva al estado objetivo.

Aquí es donde nuestro estado objetivo siendo nuestra matriz de 3x3 resuelto entra como punto final u objetivo esperado y se inicializa una cola de prioridad que denominamos **“cola prioridad”** así como un conjunto llamado **“explorado”** para realizar el seguimiento de aquellos estados donde ya hemos explorado.

Ahora pasamos a lo antes mencionado utilizando la heurística para el estado inicial usando nuestra función **distancia_manhattan** el cual calcula la distancia de Manhattan entre la posición actual de cada número en el estado y junto su posición objetivo en el estado resuelto así nuestra heurística se utiliza para estimar cuán cerca está nuestro estado actual con el estado objet

```python
# Función para calcular la distancia Manhattan entre dos puntos en una matriz 3x3
def distancia_manhattan(punto1, punto2):
    x1, y1 = punto1
    x2, y2 = punto2
    return abs(x1 - x2) + abs(y1 - y2)
```

Se calcula la heurística para el estado inicial utilizando la función **distancia_manhattan**, que calcula la distancia de Manhattan entre la posición actual de cada número en el estado y su posición objetivo en el estado resuelto. Esta heurística se utiliza para estimar cuán cerca está el estado actual del estado objetivo.

Posterior a esto se agrega una tupla a la cola de prioridad que contiene la heurística del estado inicial, el estado inicial y una lista vacía que representa el camino hasta el estado actual.

Ahora comenzamos con nuestra iteración ejecutándose mientras la prioridad no esté vacía.

```
# Calcular la heurística para el estado inicial
heuristica_inicial = distancia_manhattan(buscar_posicion(estado_inicial, 0), (2, 2))
cola_prioridad.append((heuristica_inicial, estado_inicial, []))

while cola_prioridad:
    _, estado_actual, camino = cola_prioridad.pop(0)  # Extraer el estado con menor heurística
    explorado.add(str(estado_actual))

    if estado_actual == objetivo:
        return camino + [estado_actual]

    for sucesor in sucesores(estado_actual):
        if str(sucesor) not in explorado:
            heuristica_sucesor = distancia_manhattan(buscar_posicion(sucesor, 0), (2, 2))
            cola_prioridad.append((heuristica_sucesor, sucesor, camino + [sucesor]))

    # Ordenar la cola de prioridad según la heurística
    cola_prioridad.sort(key=lambda x: x[0])
```

En cada iteración del bucle, se extrae de la cola de prioridad el estado con la menor heurística **(el estado más prometedor)** utilizando el método pop(0). Conforme se avance,se agrega el estado actual al conjunto de estados explorados.

Y se verifica si el estado actual es igual al estado objetivo. Si es así, se devuelve el camino hasta el estado actual concatenado con el estado actual como la solución encontrada.

Si el estado actual no es el estado objetivo, se generan los sucesores del estado actual utilizando la función sucesores. Los sucesores son los posibles movimientos desde el estado actual, es decir, los estados que se pueden obtener moviendo el espacio en blanco hacia arriba, abajo, izquierda o derecha.

Para cada sucesor, se verifica si el estado ya ha sido explorado previamente. Si no ha sido explorado, se calcula la heurística del sucesor utilizando la función **distancia_manhattan** y se agrega a la cola de prioridad con la heurística, el sucesor y el camino actualizado.

Después de generar los sucesores, se ordena la cola de prioridad en función de la heurística de cada elemento. Esto asegura que en cada iteración del bucle se seleccione el estado con la menor heurística, lo que representa el "primero el mejor" en la estrategia de búsqueda.

Si se agota la cola de prioridad sin encontrar el estado objetivo, se devuelve None para indicar que no se encontró solución.

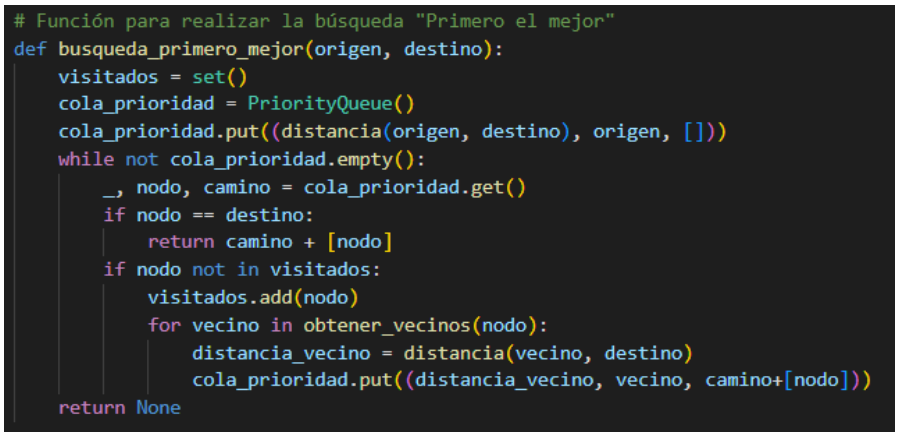
Laberinto

```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 heurística para calcular la distancia entre dos puntos
def distancia(punto1, punto2):
    x1, y1 = punto1
    x2, y2 = punto2
    return abs(x2 - x1) + abs(y2 - y1)

# Función para realizar la búsqueda "Primero el mejor"
def busqueda_primero_mejor(origen, destino):
    visitados = set()
    cola_prioridad = PriorityQueue()
    cola_prioridad.put((distancia(origen, destino), origen, []))
    while not cola_prioridad.empty():
        _, nodo, camino = cola_prioridad.get()
        if nodo == destino:
            return camino + [nodo]
        if nodo not in visitados:
            visitados.add(nodo)
            for vecino in obtener_vecinos(nodo):
                distancia_vecino = distancia(vecino, destino)
                cola_prioridad.put((distancia_vecino, vecino, camino+[nodo]))
    return None
```

Partimos observando la misma lógica antes mencionada la cual es necesaria para este algoritmo donde es basada en la heurística la cual determina que nodo explorara a continuación teniendo un cálculo de distancia.

La funcionalidad del código es el siguiente:

La función **distancia(punto1, punto2)** calcula la distancia de Manhattan entre dos puntos en un plano. Recibe dos puntos como argumentos, cada uno representado por una tupla (x, y) que indica sus coordenadas. La función calcula la diferencia en las coordenadas x y y de los dos puntos, toma el valor absoluto de cada diferencia y las suma para obtener la distancia de Manhattan.



```python
# Función para realizar la búsqueda "Primero el mejor"
def busqueda_primero_mejor(origen, destino):
    visitados = set()
    cola_prioridad = PriorityQueue()
    cola_prioridad.put((distancia(origen, destino), origen, []))
    while not cola_prioridad.empty():
        _, nodo, camino = cola_prioridad.get()
        if nodo == destino:
            return camino + [nodo]
        if nodo not in visitados:
            visitados.add(nodo)
            for vecino in obtener_vecinos(nodo):
                distancia_vecino = distancia(vecino, destino)
                cola_prioridad.put((distancia_vecino, vecino, camino+[nodo]))
    return None
```

La función **busqueda_primero_mejor(origen, destino)** implementa el algoritmo **"Primero el mejor"** para buscar el camino más corto desde el nodo de origen hasta el nodo de destino.

Este recibe como argumentos el nodo de origen y el nodo de destino. La función utiliza una cola de prioridad cola_prioridad para almacenar los nodos a explorar, donde cada elemento de la cola tiene la forma **(valor_heuristico, nodo, camino)**.

Utilizando el valor heurístico calculando la distancia entre el nodo actual y el nodo de destino. y realizado la misma iteración mientras la cola de prioridad no esté vacía.

Así en cada iteración se extrae nuevamente el nodo con el menor valor heurístico de la cola y se verifica si es el nodo destino. Si es el nodo de destino, se devuelve el camino encontrado y de ser contrario se marcaría el nodo como visitado generando los vecinos del nodo actual utilizando la función **obtener_vecinos,** y se agregan a la cola de prioridad con el valor heurístico correspondiente, actualizando y agregando el nodo actual al final del camino anterior. Si no se encuentra un camino válido, volviéndose None.

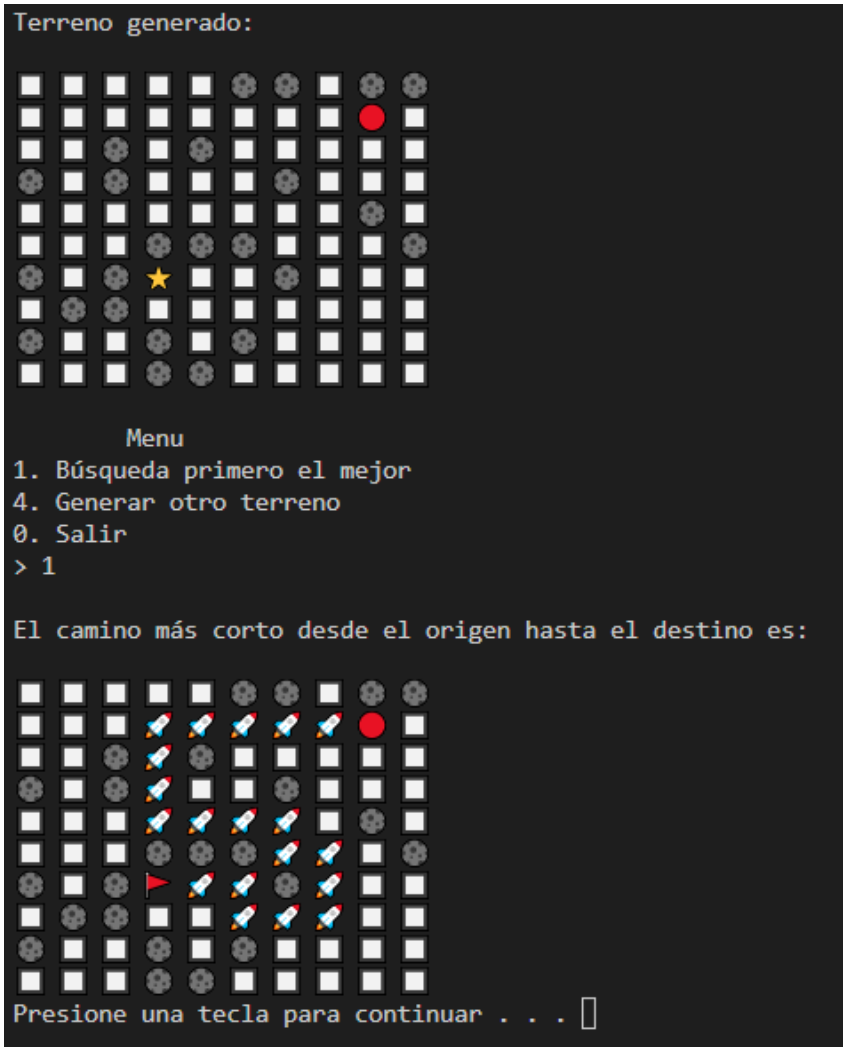
Resultados de implementación

Puzzle

```
Estado inicial:

        [1, 3, 0]
        [4, 2, 5]
        [7, 8, 6]

       Menú
1. Búsqueda primero el mejor
4. Generar otro estado
0. Salir
> 1

Solución en amplitud:

[1, 0, 3]
[4, 2, 5]
[7, 8, 6]

[1, 2, 3]
[4, 0, 5]
[7, 8, 6]

[1, 2, 3]
[4, 5, 0]
[7, 8, 6]

[1, 2, 3]
[4, 5, 6]
[7, 8, 0]

[1, 2, 3]
[4, 5, 6]
[7, 8, 0]

Tiempo de ejecución: 0.0009999275207519531 segundos
Presione una tecla para continuar . . .
```

Como podemos observar nuestra búsqueda y comparación de los desarrollos previos esta búsqueda es rápida y no tan demandante aunque al momento de ponerlo a una complejidad más amplia del problema puede tardar pero no tanto como lo fue en las implementaciones previas.

Laberinto



Observamos que al igual que el previo problema el algoritmo de primero el mejor dando un desempeño excelente en tiempos dando una rápida respuesta sin importar la dificultad para este problema planteado teniendo una gran diferencia de lo que fue la búsqueda en amplitud.

Conclusión:

Juan José Salazar Villegas:

La implementación del algoritmo **"Primero el mejor"** con la heurística de distancia de Manhattan proporciona una estrategia eficiente para encontrar caminos óptimos en un plano. El uso de la heurística de distancia de Manhattan permite estimar la proximidad de un estado al estado objetivo, lo que guía la búsqueda hacia las áreas más prometedoras y reduce el número de nodos explorados. Dentro del desarrollo es una lógica que casi sin importar la problemática se adapta para funcionar como lo fue el calcular la distancia Manhattan entre dos puntos en el plano así nuestra función de busqueda_primero_mejor utiliza una cola de prioridad donde como observamos se almacenan los nodos a explorar, priorizando aquellos con menor valor heurístico. Si nuestro algoritmo continúa la exploración de los sucesores del nodo actual hasta encontrar el nodo objetivo, construiremos un camino óptimo en función de la heurística utilizada.

Juan Emmanuel Fernández de Lara Hernández:

En el caso del laberinto, el algoritmo es capaz de encontrar un camino óptimo desde el punto de inicio hasta la meta, evitando obstáculos y minimizando la distancia recorrida. La heurística de distancia de Manhattan proporciona una estimación precisa de la proximidad de un estado al estado objetivo, lo que permite guiar la búsqueda hacia las áreas más prometedoras y reducir el número de nodos explorados. Esto resulta en un tiempo de ejecución más rápido y una solución eficiente.

En el caso del 8-puzzle, el algoritmo también es capaz de encontrar la solución óptima, es decir, la secuencia de movimientos más corta para llegar al estado objetivo. La heurística de distancia de Manhattan se utiliza para estimar la distancia total desde el estado actual hasta el estado objetivo, teniendo en cuenta la posición de cada ficha en relación con su posición correcta. Esta estimación guía la búsqueda hacia los movimientos que tienen un impacto más significativo en la reducción de la distancia al estado objetivo.