A estrella (A*)

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:

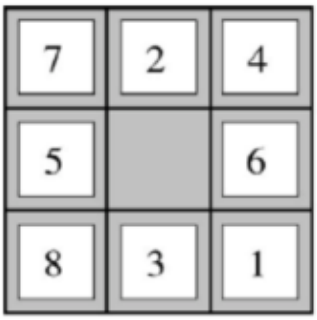
Desarrollar un programa que encuentre la mejor solución a los problemas planteados, usando los algoritmos de búsqueda no informada vistos en clase (A estrella (A*)).

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:

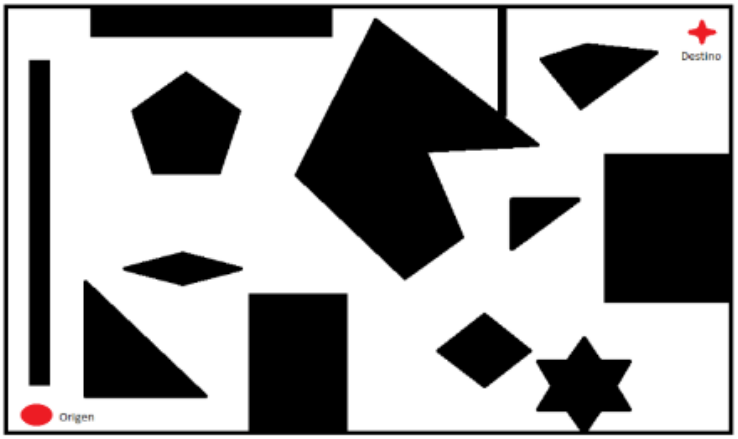
Siguiendo la lógica de cada problema realizado previamente se establecerá el estrategia de primero Mejor para observar el desempeño de los diferentes problemas previamente establecidos

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 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 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 A*
def busqueda_a_estrella(estado_inicial):
    objetivo = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
    cola_prioridad = []
    explorado = set()
    g = {}  # Costo acumulado desde el estado inicial hasta el estado actual
    f = {}  # Estimación del costo total desde el estado inicial hasta el objetivo

    # Calcular los valores iniciales de g y f para el estado inicial
    g[str(estado_inicial)] = 0
    f[str(estado_inicial)] = distancia_manhattan(buscar_posicion(estado_inicial, 0), (2, 2))
    cola_prioridad.append((f[str(estado_inicial)], estado_inicial, []))

    while cola_prioridad:
        _, estado_actual, camino = cola_prioridad.pop(0)  # Extraer el estado con menor f

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

        explorado.add(str(estado_actual))

        for sucesor in sucesores(estado_actual):
            costo = costo_movimiento(estado_actual, sucesor)
            nuevo_costo = g[str(estado_actual)] + costo

            if str(sucesor) not in g or nuevo_costo < g[str(sucesor)]:
                g[str(sucesor)] = nuevo_costo
                f[str(sucesor)] = nuevo_costo + distancia_manhattan(buscar_posicion(sucesor, 0), (2, 2))
                cola_prioridad.append((f[str(sucesor)], sucesor, camino + [sucesor]))

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

    return None
```

Para la implementación del algoritmo “A estrella (A*)” volveremos a utilizar la lógica del desarrollo previo con la heurística y la 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:

Comenzamos con la Función base que previamente desarrollamos y detallamos siendo la **distancia Manhattan:**

distancia_manhattan(punto1, punto2) calculara la distancia Manhattan entre dos puntos en una matriz 3x3. Esta distancia se utiliza como heurística para estimar el costo desde un estado dado hasta el estado objetivo. Ahora entra la Función de **búsqueda de posición:** La función **buscar_posicion(estado, elemento)** busca la posición de un elemento en el estado actual. Se utiliza para encontrar la posición del espacio en blanco (representado por el número 0) en el estado.

```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)
```

Ahora planteamos el costo del movimiento con nuestra función **costo_movimiento(estado_actual, siguiente_estado).**

Esta función devolverá el costo de moverse desde el estado actual hasta el siguiente estado. En tu implementación, el costo siempre es 1, lo que indica que todos los movimientos tienen el mismo costo.

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

Pasamos a nuestro algoritmo principal siendo nuestra función:

busqueda_a_estrella(estado_inicial) dicha función implementa el algoritmo A* Estrella. Utiliza una cola de prioridad **(cola_prioridad)** para explorar los estados en orden de costo total (f). Se utiliza un diccionario g para almacenar el costo acumulado desde el estado inicial hasta el estado actual, y otro diccionario f para almacenar la estimación del costo total desde el estado inicial hasta el objetivo.

Dentro del bucle principal, se extrae el estado con el menor costo total de la cola de prioridad.

```python
# Función para buscar la solución por el algoritmo A*
def busqueda_a_estrella(estado_inicial):
    objetivo = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
    cola_prioridad = []
    explorado = set()
    g = {}  # Costo acumulado desde el estado inicial hasta el estado actual
    f = {}  # Estimación del costo total desde el estado inicial hasta el objetivo

    # Calcular los valores iniciales de g y f para el estado inicial
    g[str(estado_inicial)] = 0
    f[str(estado_inicial)] = distancia_manhattan(buscar_posicion(estado_inicial, 0), (2, 2))
    cola_prioridad.append((f[str(estado_inicial)], estado_inicial, []))

    while cola_prioridad:
        _, estado_actual, camino = cola_prioridad.pop(0)  # Extraer el estado con menor f

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

        explorado.add(str(estado_actual))

        for sucesor in sucesores(estado_actual):
            costo = costo_movimiento(estado_actual, sucesor)
            nuevo_costo = g[str(estado_actual)] + costo

            if str(sucesor) not in g or nuevo_costo < g[str(sucesor)]:
                g[str(sucesor)] = nuevo_costo
                f[str(sucesor)] = nuevo_costo + distancia_manhattan(buscar_posicion(sucesor, 0), (2, 2))
                cola_prioridad.append((f[str(sucesor)], sucesor, camino + [sucesor]))

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

    return None
```

Si este estado es el estado objetivo, se devuelve el camino hasta ese estado. De lo contrario, se generan los sucesores del estado actual y se actualizan los costos acumulados y estimados. Los sucesores se agregan a la cola de prioridad para su explotación posterior.

El bucle continúa hasta que no quedan estados en la cola de prioridad o se encuentra el estado objetivo. Si la cola de prioridad se vacía sin encontrar la solución, se retorna None.

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 A*
def busqueda_a_estrella(origen, destino):
    visitados = set()
    cola_prioridad = PriorityQueue()
    g_score = {origen: 0}
    f_score = {origen: distancia(origen, destino)}
    cola_prioridad.put((f_score[origen], 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):
                g_score_vecino = g_score[nodo] + 1
                if vecino not in g_score or g_score_vecino < g_score[vecino]:
                    g_score[vecino] = g_score_vecino
                    f_score[vecino] = g_score_vecino + distancia(vecino, destino)
                    cola_prioridad.put((f_score[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:

Recataremos la misma logica asi como puntos planteados previamente analizando cada una de las funciones para ello necesitamos obtener los nodos vecinos esto gracias a la siguiente función:

obtener vecinos(nodo) Está devolverá una lista de los vecinos válidos de un nodo en una matriz. En tu implementación, se verifican las posiciones arriba, abajo, izquierda y derecha del nodo y se agregan a la lista de vecinos si no son obstáculos ('🌑').

```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
```

Ahora pondremos nuestra Función heurística para calcular la distancia entre dos puntos esto con la función **distancia(punto1, punto2)** la cual calcula como lo hemos visto repite la misma idea con la distancia Manhattan entre dos puntos en la matriz.

Esta función se utiliza como heurística para estimar el costo desde un nodo dado hasta el nodo destino.

Función para realizar la búsqueda A*:

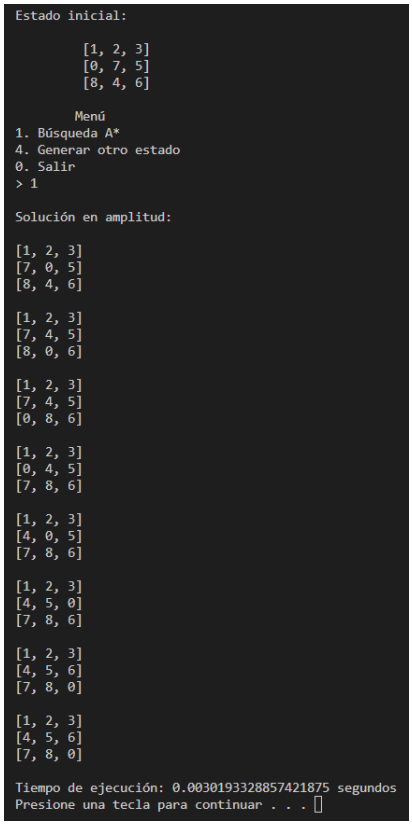
```python
# Función para realizar la búsqueda A*
def busqueda_a_estrella(origen, destino):
    visitados = set()
    cola_prioridad = PriorityQueue()
    g_score = {origen: 0}
    f_score = {origen: distancia(origen, destino)}
    cola_prioridad.put((f_score[origen], origen, []))
    while not cola_prioridad.empty():
        _, 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):
                g_score_vecino = g_score[nodo] + 1
                if vecino not in g_score or g_score_vecino < g_score[vecino]:
                    g_score[vecino] = g_score_vecino
                    f_score[vecino] = g_score_vecino + distancia(vecino, destino)
                    cola_prioridad.put((f_score[vecino], vecino, camino+[nodo]))
    return None
```

La función **busqueda_a_estrella(origen, destino)** implementa el algoritmo A* Estrella.

Dentro de este algoritmo reincorporamos los puntos previamente definidos, siendo la cola de prioridad que para resumir explora los nodos en orden con el costo total (f) y aquellos previamente visitados y almacenados el diccionario g_score (costo acumulado) desde el origen y el otro diccionario f_score para almacenar la estimacion del costo total desde el origen hasta el destino.

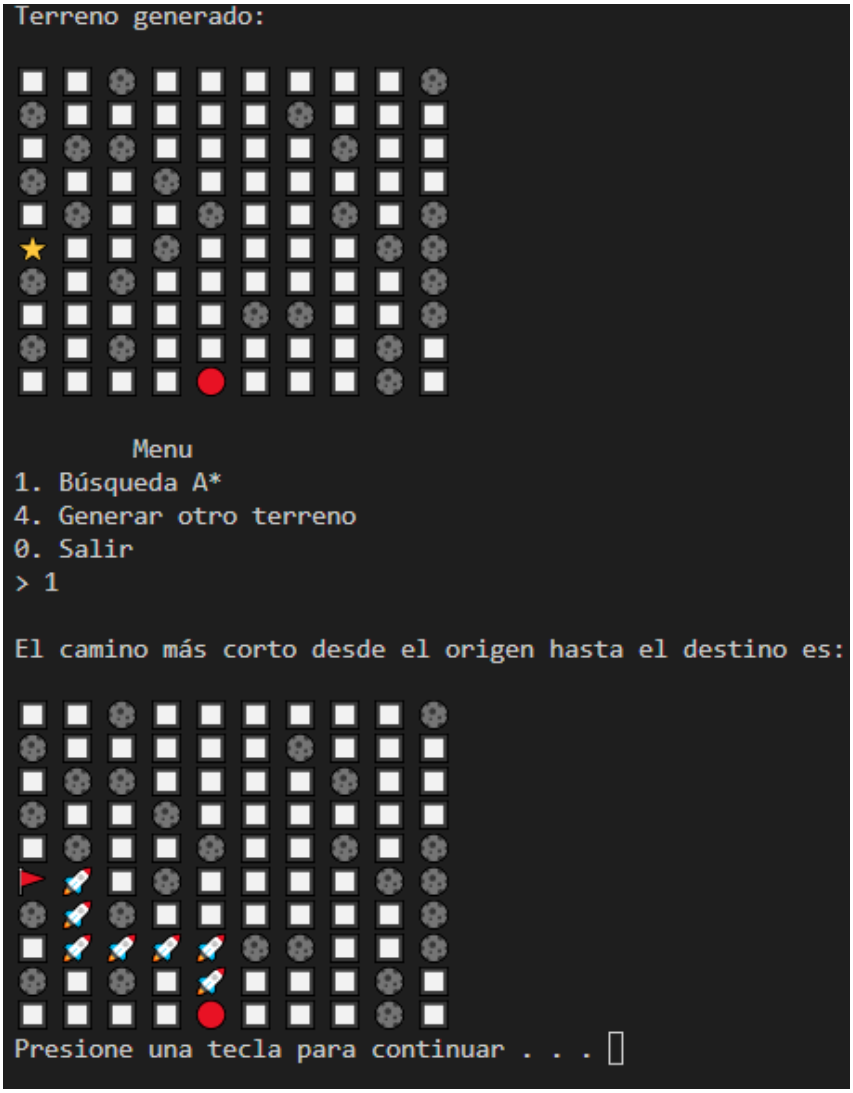
Resultados de implementación

Puzzle



Comparación de los desarrollos previos este algoritmo da una eficiencia muy precisa gracias a las heurísticas y con la cola de prioridad esto nos ayuda demasiado al tener un orden menor costo estimado expandiendo los nodos más prometedores y reducir la cantidad total de nodos explorados.

Laberinto



Vemos y comprendemos cómo el algoritmo de A* da una versatilidad al ser utilizado en problemas con restricciones adicionales esto por las limitantes que como lo vimos de pasar a tener un terreno chico a algo más extenso y aun sin tanto recursos siendo adaptable y aplicable con diferentes tipos de problemas.

Conclusión:

Juan José Salazar Villegas:

El algoritmo A* Estrella es una herramienta poderosa y versátil para la resolución de problemas de búsqueda de caminos. Su aplicación en problemas previos demuestra cómo puede adaptarse a diferentes contextos y problemas específicos. Como lo fue el desafío del 8-puzzle, encontrando la secuencia de movimientos óptima para llegar a un estado objetivo. Por otro lado, el segundo código se aplica a la búsqueda de caminos en un laberinto, evitando obstáculos para encontrar el camino más corto desde un punto de inicio hasta un destino.

Ambas implementaciones hacen uso de una heurística como punto de partida, como la distancia Manhattan, para estimar los costos restantes y guiar la exploración. Además, ambas utilizan estructuras de datos como colas de prioridad para organizar los nodos según el costo estimado.

Juan Emmanuel Fernández de Lara Hernández:

La implementación del algoritmo A* ha demostrado ser una herramienta poderosa y versátil en la resolución de problemas de búsqueda de caminos.

En el caso del 8-puzzle, el algoritmo A* se encarga de encontrar la secuencia de movimientos óptima para llegar al estado objetivo. Utiliza la heurística de distancia de Manhattan para estimar los costos restantes y guiar la exploración. Esta heurística tiene en cuenta la posición de cada ficha en relación con su posición correcta, lo que permite tomar decisiones informadas sobre los movimientos a realizar.

En el caso del laberinto, el algoritmo A* se utiliza para encontrar el camino más corto desde un punto de inicio hasta un destino, evitando obstáculos en el proceso. La heurística de distancia de Manhattan nuevamente desempeña un papel clave al estimar los costos restantes y guiar la exploración. Al igual que en el caso del 8-puzzle, el uso de una cola de prioridad permite organizar los nodos según el costo estimado, lo que resulta en una búsqueda eficiente y una solución óptima.