



Puzzle búsqueda en profundidad y profundidad iterativa

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:

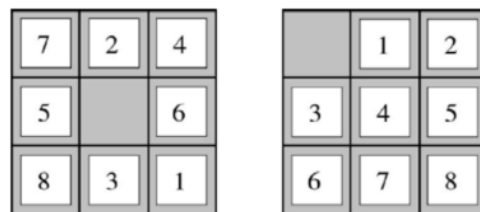
Desarrollar un programa que encuentre la mejor solución a los problemas planteados, usando los algoritmos de búsqueda no informada vistos en clase (búsqueda en profundidad, búsqueda en amplitud, etc).

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:

Implementar la búsqueda en amplitud y en profundidad para dar solución de forma automática al problema 8-puzzle.

El estado inicial debe ser aleatorio.



Estado Inicial

Estado Objetivo

Fig.1 Ejemplo del problema 8 Puzzle

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 definiendo 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.

[Nota: solución óptima de la familia del n-puzzle es NP-Completo]

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

Ahora teniendo los estados de localizaciones siendo izquierda o derecha donde se Moverán las piezas deberemos tener presente los estado iniciales y finales para saber de donde partir siendo los siguientes:

7	2	4
5		6
8	3	1

Estado Inicial

	1	2
3	4	5
6	7	8

Estado Objetivo

Codificación e implementación

Definición del estado objetivo:

```
# 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.

Aleatoriedad de Puzzle:

```
# Función para generar un estado aleatorio válido
def estado_aleatorio():
    estado = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
    movimientos = ['arriba', 'abajo', 'izquierda', 'derecha']
    n_movimientos = random.randint(20, 50)
    for i in range(n_movimientos):
        movimiento = random.choice(movimientos)
        espacio = 0
        for j in range(3):
            if espacio != 0:
                break
            for k in range(3):
                if estado[j][k] == 0:
                    espacio = (j, k)
                    break
        if movimiento == 'arriba':
            if espacio[0] > 0:
                estado[espacio[0]][espacio[1]] = estado[espacio[0]-1][espacio[1]]
                estado[espacio[0]-1][espacio[1]] = 0
        elif movimiento == 'abajo':
            if espacio[0] < 2:
                estado[espacio[0]][espacio[1]] = estado[espacio[0]+1][espacio[1]]
                estado[espacio[0]+1][espacio[1]] = 0
        elif movimiento == 'izquierda':
            if espacio[1] > 0:
                estado[espacio[0]][espacio[1]] = estado[espacio[0]][espacio[1]-1]
                estado[espacio[0]][espacio[1]-1] = 0
        elif movimiento == 'derecha':
            if espacio[1] < 2:
                estado[espacio[0]][espacio[1]] = estado[espacio[0]][espacio[1]+1]
                estado[espacio[0]][espacio[1]+1] = 0
    return estado
```

Dentro de este apartado tenemos los espacios así como los movimientos para poder hacer movimientos o instancias diferentes del Puzzle al iniciar el programa así las búsquedas pueden ser implementadas sin definir solamente un estado inicial fijo con ello podremos entender cómo cada distinto orden puede ser operado sin problema alguno.

Además de esto tenemos en el menú una opción para recrear una nueva instancia del puzzle.

Solución del puzzle:

```
# 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]]

# 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)
    # Devolver los sucesores
    return sucesores
```

Este apartado tiene como objetivo mover el espacio en blanco esto basado al estado inicial del Puzzle al ser aleatorio estableciendo los movimientos y acciones posibles las cuales dependen del entorno o alrededor del espacio en blanco así se evalúa junto al tipo de búsqueda seleccionado y dará un movimiento posible con estos movimientos se va teniendo un registro de los sucesores para seguir con la solución.

Búsqueda por amplitud y profundidad:

```
# Función para buscar la solución por amplitud
def busqueda_amplitud(estado_inicial):
    cola = deque([(estado_inicial, [])])
    explorado = set()
    while cola:
        estado_actual, camino = cola.popleft()
        explorado.add(str(estado_actual))
        for suceso in sucesores(estado_actual):
            if str(suceso) not in explorado:
                if es_objetivo(suceso):
                    return camino + [suceso]
                cola.append((suceso, camino + [suceso]))
    return None

def busqueda_profundidad(estado_inicial):
    # Inicializar la pila de estados a visitar
    pila = [(estado_inicial, [])]
    # Inicializar el conjunto de estados visitados
    visitados = set()
    visitados.add(tuple(map(tuple, estado_inicial)))
    # Buscar el estado objetivo
    while len(pila) > 0:
        estado, camino = pila.pop()
        if es_objetivo(estado):
            # Construir el camino desde el estado inicial hasta el estado objetivo
            return camino + [estado]
        for suceso in sucesores(estado):
            if tuple(map(tuple, suceso)) not in visitados:
                pila.append((suceso, camino + [estado]))
                visitados.add(tuple(map(tuple, suceso)))
    return None
```

Ahora es donde entra el objetivo principal esto evaluando con los métodos de búsqueda esté respaldado por la función previa donde se evalúa el puzzle actual para mover el espacio y dar forma, seleccionando una u otra opción haciendo que el alcanzar la solución más cercana al nodo raíz lo antes posible (Profundidad) o explorando los nodos de un árbol de búsqueda en orden de su nivel o profundidad, comenzando por la raíz.

Cada uno de estos dará una eficiencia distinta y un tiempo de resolución muy distinto.

Búsqueda por profundidad iterativa:

```
# Función para buscar la solución por profundidad limitada
def busqueda_profundidad_limitada(estado, limite):
    frontera = [(estado, [])]
    explorado = set()
    while frontera:
        estado_actual, camino = frontera.pop() # Extraer el último estado de la frontera
        explorado.add(str(estado_actual))
        if es_objetivo(estado_actual): # Si es el estado objetivo, devolver el camino
            return camino + [estado_actual]
        if len(camino) < limite:
            for sucesor in sucesores(estado_actual): # Generar los sucesores del estado actual
                if str(sucesor) not in explorado: # Si el sucesor no ha sido explorado
                    frontera.append((sucesor, camino + [sucesor])) # Agregarlo a la frontera
    return 'corte' # Devolver corte si no se encontró la solución

# Función para buscar la solución por profundidad iterativa
def busqueda_profundidad_iterativa(estado):
    limite = 0
    while True:
        solucion = busqueda_profundidad_limitada(estado, limite) # Buscar la solución
        if solucion != 'corte': # Si no se cortó la búsqueda
            return solucion
        limite += 1 # Incrementar el límite
```

Para esta sección del código es necesario la implementación de búsqueda en profundidad limitada para controlar la profundidad máxima de exploración y evitar problemas de falta de memoria durante la búsqueda. Con esto la Búsqueda por profundidad interactiva realiza las múltiples búsquedas en profundidad máxima creciente en cada iteración limitada esto para que se adapte a la complejidad del problema y evita la necesidad de un profundidad máxima de manera fija.

Ya definida esta sección de código establecemos la búsqueda iterativa donde explora el árbol esto de manera incremental y controlada. basado en el código previo de profundidad máxima establece el rango de iteraciones hasta encontrar la solución.

Main (Menu)

```
if __name__ == '__main__':  
    while True:  
        os.system('cls')  
  
        # Generar un estado aleatorio  
        estado_inicial = estado_aleatorio()  
        # Imprimir el estado inicial  
        print('Estado inicial: \n')  
        for fila in estado_inicial:  
            print('\t', fila)  
  
        print("\nElige una opción para resolver el puzzle:")  
        print("1. Amplitud")  
        print("2. Profundidad")  
        print("3. Generar otro estado")  
        print("0. Salir")  
        try:  
            opcion = int(input("> "))  
        except ValueError:  
            opcion = -1
```

Aquí se establece la pantalla principal del programa mostrando las diferentes opciones a realizar entre estas seleccionar la manera de solucionar el puzzle y de caso ser necesario poder generar otro estado inicial de nuestro puzzle esto gracias a la aleatoriedad que definimos previamente.

Ya con una opción se pasa a un "Try" la cual hará entrada a cada una de las mismas y accionan a las funciones seleccionadas.

Main (Opciones)

```
if opcion == 1:
    tiempo_inicial = time.time()

    # Buscar la solución por amplitud
    solucion = busqueda_amplitud(estado_inicial)

    tiempo_final = time.time()
    tiempo_ejecucion = (tiempo_final - tiempo_inicial)
    minutos, segundos = divmod(tiempo_ejecucion, 60)

    # Imprimir la solución
    print('\nSolución en amplitud: \n')
    for estado in solucion:
        for fila in estado:
            print(fila)
        print()
    print(f'Tiempo de ejecución: [{minutos:01d}:{segundos:02d}']')
    os.system('pause')

elif opcion == 2:
    tiempo_inicial = time.time()

    # Buscar la solución por profundidad
    solucion = busqueda_profundidad(estado_inicial)

    tiempo_final = time.time()
    tiempo_ejecucion = int(tiempo_final - tiempo_inicial)
    minutos, segundos = divmod(tiempo_ejecucion, 60)

    # Imprimir la solución
    print('\nSolución en profundidad: \n')
    for estado in solucion:
        for fila in estado:
            print(fila)
        print()
    print(f'Tiempo de ejecución: [{minutos:01d}:{segundos:02d}']')
    os.system('pause')

elif opcion == 3:
    pass

elif opcion == 0:
    break

else:
    print("\nOpción no válida, intente de nuevo.")
    os.system('pause')
```

Aquí damos dirección a cada una de las posibles opciones de nuestro menú donde abre la solución por profundidad o amplitud, algo que podemos rescatar es que al seleccionar cualquiera de las dos se evaluará o tomara en cuenta el tiempo de solución esto basado un contador tomando el tiempo de ejecución y posterior el final para imprimir el total del mismo.

Resultados de Búsquedas

Amplitud

```
Elige una opción para resolver el puzzle:
1. Amplitud
2. Profundidad
3. Generar otro estado
0. Salir
> 1

Solución en amplitud:

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

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

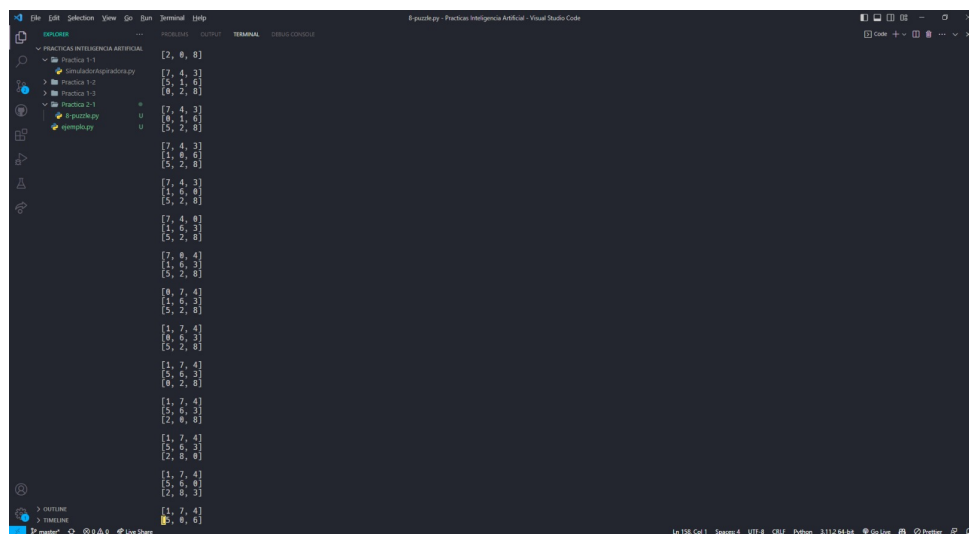
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]

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

Tiempo de ejecución: 0.0009980201721191406 segundos
Presione una tecla para continuar . . .
```

Siendo la búsqueda más rápida y menos demandante de recursos ésta sin importar la aleatoriedad de nuestro estado inicial del “Puzzle” esto por explorar hacia los nodos del árbol en orden de nivel o profundidad, comenzando por la raíz.

Profundidad iterativa



Este al inicio pareciera que no funciona pero es el tiempo que tarda por el tipo de búsqueda lo que hace que no de la solución al ir tan profundo del árbol ya que este haya encontrado solución o no despliega un sin fin de los pasos que tuvo que recorrer para llegar hasta esa solución este algoritmo cabe recalcar que consume demasiados recursos para operar haciendo que la computadora se estrese y esto sin dar fe de que se encuentre llegue al estado objetivo.

Profundidad iterativa:

```
Estado inicial:

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

Elije una opción para resolver el puzzle:
1. Amplitud
2. Profundidad
3. Profundidad iterativa
4. Generar otro estado
0. Salir
> 3

Solución en profundidad iterativa:

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

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

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

[1, 2, 0]
[4, 5, 3]
[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: 0.0009925365447998047 segundos
Presione una tecla para continuar . . .
```

Este método de búsqueda me sorprende que sea tan eficiente y esto puede deberse a la limitación de los nodos en la memoria para establecer un rango del mismo árbol así se garantiza que la exploración de todos los nodos de búsqueda siendo completa y oprima además de adaptarse a la complejidad del problema esto por la profundidad máxima fija evitando que la falta memoria en la búsqueda en profundidad sin limitación.

Conclusión:

Juan José Salazar Villegas:

Dentro de esta práctica pude comprender mejor los diferentes tipos de búsqueda y lo que implica su uso ya que al trabajar de manera distinta pueden exigir más o nula cantidad de recursos por ello es necesario comprenderlos y sobre todo saber que tipo se adapta más a nuestra necesidad así no se sacrifican recursos y se obtienen mejores resultados.

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:

Es fundamental tener en cuenta que la elección de una estrategia de búsqueda en particular, ya sea la búsqueda en profundidad o en amplitud, debe estar basada en las características del problema que se está tratando de resolver. Al entender las diferencias entre estas dos estrategias de búsqueda, es posible elegir la más adecuada para nuestra necesidad y así evitar la pérdida de recursos innecesaria y obtener resultados más eficaces.

A través de esta práctica se pudo adquirir una comprensión más profunda de los diferentes tipos de búsqueda en Inteligencia Artificial y su implicación en la utilización de recursos. Por lo tanto, es esencial tener una sólida comprensión de estas estrategias de búsqueda para aplicarlas de manera efectiva en la solución de problemas en Inteligencia Artificial.