Tarea 2 IA

Roberto Vásquez Martínez
Profesor: Arturo Hernández Aguirre

09/Noviembre/2021

1 Búsqueda Ciega

1.1 Configuración sugerida

Iniciamos con las configuraciones proporcionadas. La configuración inicial sugerida es

$$\begin{pmatrix} 3 & 2 & 1 \\ 6 & 5 & 4 \\ 8 & 7 & 0 \end{pmatrix}$$

mientras que la configuración final es

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{pmatrix}$$

Cabe notar que el 0 representa el espacio vacío. Aplicamos el algoritmo de búsqueda para estos estados.

[9]:
```
import sys
import numpy as np
sys.path.append('..')
from blind_search import node
from blind_search import BFS
# Case 1
init_value_1=np.array([[3,2,1],[6,5,4],[8,7,0]])
final_value_1=np.array([[1,2,3],[4,5,6],[7,8,0]])
init_state_1=node(init_value_1)
# Search Algorithm
BFS_1=BFS(init_state_1,final_value_1)
BFS_1.main()
```

```
No se puede alcanzar el nodo final
```

A partir de lo anterior vemos que a partir de la configuración incial proporcionada no es alcanzable la configuración inicial. La heurística empleada fue considerar como base la configuración

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{pmatrix}$$

Se puede puede poner en forma de vector

$$(1,2,3,4,5,6,7,8,0)$$

Cualquier configuración la podemos poner en forma de vector como la anterior. En cualquier configuración del tablero, una inversión entre las entradas i, j , con $i < j$, del vector asociado a la configuración sucede cuando el valor de la celda en la posición i es mayor que el valor en la posición j.

Usamos el hecho de para poder alcanzar el estado final a partir de un estado inicial la paridad del número de inversiones debe ser la misma entre el estado final e inicial.

Para profundizar aún más en este ejemplo calculamos el número de inversiones del estado inicial y final. El número de inversiones del estado inicial es:

[2]:
```
BFS_1.getInvCount([j for sub in init_value_1 for j in sub])
```

[2]: 7

Mientras que el número de inversiones en el estado final es:

[3]:
```
BFS_1.getInvCount([j for sub in final_value_1 for j in sub])
```

[3]: 0

De lo anterior podemos ver que en efecto las paridades son distintas y en consecuencia el estado final no es alcanzable en este caso.

Para obtener un configuración inicial que alcance la final propuesta basta hacer un intercambio de fichas en la configuración inicial para modificar su paridad, por ejemplo, usemos la configuración inicial

$$\begin{pmatrix} 3 & 2 & 1 \\ 6 & 5 & 4 \\ 7 & 8 & 0 \end{pmatrix}$$

,

donde intercambiamos de lugar el 7 y 8 en la configuración inicial que da la tarea. La paridad de esta configuración inicial es la siguiente.

[4]:
```
init_value_1=np.array([[3,2,1],[6,5,4],[7,8,0]])
BFS_1.getInvCount([j for sub in init_value_1 for j in sub])
```

[4]: 6

Sin embargo al correr al hacer la búsqueda. Ya sea con el BFS o DFS el tiempo de ejecución no es razonable y no se termino el proceso.

En el algoritmo el i -ésimo nodo explorado lo comparamos con los $1, 2, \ldots, i-1$ anteriores para no crear hijos repetidos. Por lo que hacemos al menos el siguiente número de operaciones para

un árbol de n estados.

$$\sum_{i=1}^{n}(i-1) = \frac{n(n-1)}{2}$$

Como hay a lo más 9!/2 nodos entonces en ese caso hacemos aproximadamente el siguiente número de operaciones

$$\left(\frac{9!}{2}\right)^2 \approx 3.29 \times 10^{10}$$

Lo anterior tardará varias horas por lo que desistimos de terminar el proceso.

1.2 Otras configuraciones

A continuación vamos a analizar el desempeño del BFS y DFS con otro par de configuraciones inicial y final.

1.2.1 Primera configuración

La configuración inicial sugerida es

$$\begin{pmatrix} 1 & 2 & 3 \\ 0 & 4 & 6 \\ 7 & 5 & 8 \end{pmatrix}$$

y la final es

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 6 & 0 \\ 7 & 5 & 8 \end{pmatrix}$$

BFS Procedemos a aplicar el algoritmo de búsqueda BFS con estas configuraciones.

[5]:
```
init_value_2=np.array([[1,2,3],[0,4,6],[7,5,8]])
final_value_2=np.array([[1,2,3],[4,6,0],[7,5,8]])
init_state_2=node(init_value_2)
BFS_2=BFS(init_state_2,final_value_2)
BFS_2.main()
```

```
1 : 2 : 3
0 : 4 : 6
7 : 5 : 8


1 : 2 : 3
4 : 0 : 6
7 : 5 : 8


1 : 2 : 3
```

```
4 : 6 : 0
7 : 5 : 8


Numero de movimientos de la solución: 2
Numero de nodos visitados: 9
Numero de nodos por visitar: 8
Numero de nodos expandidos: 17
```

Desarrollando el árbol a mano se puede ver que el nodo final está en tercer nivel y es el último nodo que el BFS valida en ese nivel. Cabe mencionar que la búsqueda a profundidad que se ha implementado en el módulo *blind_search.py* hace la búsqueda en anchura de izquierda a derecha.

El número de nodos expandidos es aquellos que se han construido en el árbol, ya sea que se han visitados o se han creado a partir de ser hijos de nodos visitados.

Es claro que el tipo de búsqueda BFS da el camino más corto a la solución, sin embargo, como se ha notado en este ejemplo, puede que se tengan que recorrer todos los nodos de cada nivel hasta encontrar la solución en el último nodo del último nivel.

DFS La versión implementada en *blind_search.py* del DFS hace la búsqueda en profundidad considerando las ramas de derecha a izquierda. Realizamos la búsqueda DFS con los estados inicial y final sugeridos en esta sección.

[6]:
```
from blind_search import DFS
DFS_2=DFS(init_state_2,final_value_2)
DFS_2.main()
```

```
1 : 2 : 3
0 : 4 : 6
7 : 5 : 8


1 : 2 : 3
4 : 0 : 6
7 : 5 : 8


1 : 2 : 3
4 : 6 : 0
7 : 5 : 8


Numero de movimientos de la solución: 2
Numero de nodos visitados: 3
Numero de nodos por visitar: 4
Numero de nodos expandidos: 7
```

Este ejemplo fue construido para ilustrar la ventaja que tiene en memoria el DFS, pues como el nodo solución estaba en el nivel 3 y era el último nodo a validar para el BFS, entonces el BFS se

vió obligado a expandir el número de nodos correspondiente a 3 niveles del árbol, mientras que el DFS sólo expandió uno del último nivel, pues la primer rama que eligió fue la correcta para hallar la solución.

1.2.2 Segunda configuración

A continuación elegimos otro par de configuraciones. La configuración inicial sugerida en esta ocasión es

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 0 & 5 \\ 6 & 7 & 8 \end{pmatrix}$$

y la configuración final es

$$\begin{pmatrix} 0 & 1 & 3 \\ 4 & 2 & 5 \\ 6 & 7 & 8 \end{pmatrix}$$

BFS Realizamos con la configuración anterior la búsqueda BFS. Obtenemos los siguientes resultados.

[7]:
```
init_value_3=np.array([[1,2,3],[4,0,5],[6,7,8]])
final_value_3=np.array([[0,1,3],[4,2,5],[6,7,8]])
init_state_3=node(init_value_3)
BFS_3=BFS(init_state_3,final_value_3)
BFS_3.main()
```

```
1 : 2 : 3
4 : 0 : 5
6 : 7 : 8


1 : 0 : 3
4 : 2 : 5
6 : 7 : 8


0 : 1 : 3
4 : 2 : 5
6 : 7 : 8


Numero de movimientos de la solución: 2
Numero de nodos visitados: 6
Numero de nodos por visitar: 7
Numero de nodos expandidos: 13
```

La solución se encuentra en el tercer nivel del árbol, en la primer rama de izquierda a derecha. El nodo inicial tiene 4 hijos y cada uno sus hijos tiene 2 hijos, por lo que tenemos 13 nodos en total en el árbol.

DFS Por último, realizamos la búsqueda en profundidad a la segunda configuración propuesta. El resultado es el siguiente.

[8]:
```
DFS_3=DFS(init_state_3,final_value_3)
DFS_3.main()
```

```
1 : 2 : 3
4 : 0 : 5
6 : 7 : 8


1 : 0 : 3
4 : 2 : 5
6 : 7 : 8


0 : 1 : 3
4 : 2 : 5
6 : 7 : 8


Numero de movimientos de la solución: 2
Numero de nodos visitados: 311
Numero de nodos por visitar: 0
Numero de nodos expandidos: 311
```

Como nuestra implementación del DFS explora las ramas de derecha a izquierda entonces de las ramas que construyo el BFS hasta el nivel 3 el algoritmo DFS tuvo que explorar todas hasta encontrar la solución en la última rama de derecha a izquierda en el nivel 3. Al explorar todas estas ramas hasta una profunidad de 10 (es el límite de profundidad del árbol) se visitaron cada uno de los nodos de estas ramas antes de visitar el nodo solución, por eso no hay nodos en la lista de open o de por visitar. En consecuencia, por no elegir el *orden correcto* de exploración de las ramas se exploró exhaustivamente todas las ramas hasta encontrar la solución en la última. Incluso, sin no ponemos el límite de profunidad y sin tener algún criterio para elegir el orden de exploración en profundidad en las ramas podemos tener la mala suerte de en nuestras primeras elecciones tomar una rama que se vaya a infinito, aunque este no es el caso porque el espacio de estados es finito, pero si aumenta enormemente la complejidad por una mala elección.

Vemos que si se hubiera elegido el orden de exploración de las ramas de izquierda a derecha tendríamos una solución con mucha menos exploración del árbol en consecuencia mayor ahorro en memoria como en la primer configuración propuesta. Precisamente esto es la virtud de pensar en una heurística, un criterio para elegir la mejor rama que explorar a profunidad y aprovechar la virtud del DFS de tener una complejidad en memoria lineal a diferencia del BFS que tiene una complejidad exponencial en memoria.