CSC 425 525 – 8 Puzzle Problem Solving with BFS and Heuristic Search

Instructor: Dr. Junxiu Zhou
Semester: Fall 202
Team Members: Selva Subramani Damodaran

1. Introduction

An eight-puzzle game is a board game where one can move around eight tiles in 9 spaces. There are eight numerical tiles and one blank tile in the eight-puzzle game. We specify a beginning state and a goal state for the 8-puzzle as shown below:

2	3	6
1	4	8
7	5	0

Figure 1: Example of initial State

1	2	3
4	5	6
7	8	0

Figure 2: Example of Goal State

A state is a panel contains the 9 spaces with 9 tile numbers. The blank tile is moved legally in the 3×3 panel until the state reaches the final goal state. Figure below exemplifies partial of a state generation graph with state space and arcs. A naive implementation of brute force exhaustive search may successfully find the goal state. But the problem is that time to goal increases as the puzzle size increases.

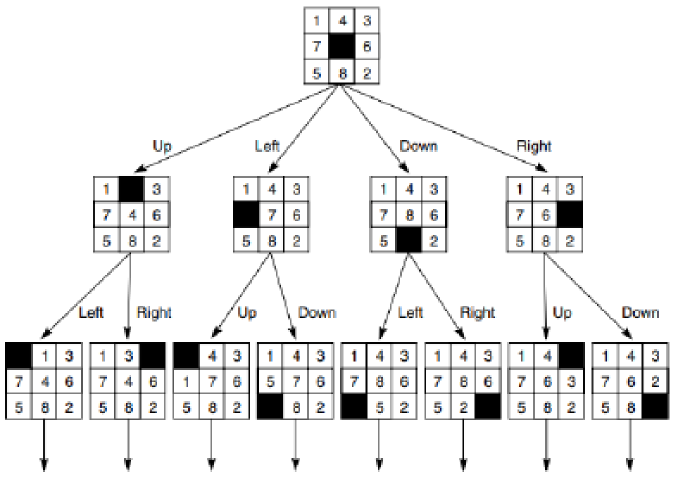


Figure 3: State Generation Graph

Formally, we define a state space as [N, A, S, GD]

N = set of nodes or states of a graph → all the different configurations of tiles that the game can have

A = set of arcs (links) between nodes that correspond to the steps in the problem (the legal actions or operators) → legal moves of the game (move tile with value 0 to left ←, right →, up ↑, down ↓)
S = a nonempty subset of N that represents start state(s) of the problem → initial state

GD = a nonempty subset of N that represents goal states of the problem

- A measurable property of the states
- A measurable property of the path developed in the search
- ➔ find a series of moves of tiles into the blank space that places the board in a goal configuration

Path = A solution path is a path through this graph from a node in S to a node in GD → from the start state to a goal state gives the series of moves.

The total number of paths possible on an 8- Puzzle game is **8!**

2. Implementation Process and Results

The code for solving this 8-puzzle game is written in Python. The solver works by sliding the tile labelled "0" and arranging the digits in ascending order, while putting the 0 in the end. There are 4 class files in this 8-puzzle solver solution. All of them implements NumPy library, since the 8-puzzle solver works with 2D array objects

1. EightPuzzleGame.py: This file contains the necessary coding for initializing and running the puzzle solver. Here the values for initial state and goal state are set. And are then passed to the Uninformed and Informed Search solver class objects.

2. EightPuzzleGame_State.py: This class contains the properties and definitions of a State. A state contains the 2D array of the current node, the depth at which the node exists currently, and the weight.

3. EightPuzzleGame_UinformedSearch.py: This class implements a Breadth first Search algorithm to solve the 8-puzzle problem. The class accepts initial and goal states as input, and then performs a set of functions to solve the initial state into goal.

   At first, the current state is added to the Open List (a list of type State class). Then the function *chk_inputstate_correct* checks if the input initial state is correct (i.e., if any number is repeated twice then it will throw error).

In the next step, the function *chk_state_solvable* checks if the input State is solvable or not. Once both the validations are satisfied by the initial state, it is then tested for goal. When initial (current) state is not equal to goal, then the function *state_walk* is performed, and the path variable is incremented on each step until the current becomes goal state.

Upon function call to *state_walk* method, the following steps are performed:
- The initial (current) state is moved from open list into the closed list.
- Then the index of tile "0" is retrieved from the current state using the function *get_zeroindex*
- Based on the index of tile "0" (row, column), four types of walks are performed: left ←, right →, up ↑, down ↓
- On each move, the child state obtained is tested through the function *check_inclusive* whether it already exists in open / closed list and then added to the open list (if not previously present) via the method *add_child_to_open*

Finally, when the goal state is achieved, the solver prints the number of iterations and the depth at which the goal is achieved.

4. <u>EightPuzzleGame_InformedSearch.py</u>: This class implements a Heuristic function in addition to the BFS algorithm, so that the path to goal state is short. Similar to Uninformed Search class, this class also accepts initial and goal states as input, and then performs a set of functions to solve the initial state into goal.

   The first step tests the current state for goal. When initial (current) state is not equal to goal, then the function *state_walk* is performed, and the path variable is incremented on each step until the current becomes goal state.

   Upon the function call to *state_walk()*, the initial (current) state is moved from open list into the closed list. The, using *get_zeroindex()* function the row and column index of tile "0" is retrieved. Based on this row & column index value, a BFS algorithm as explained in Uninformed Search class is performed (4 types of state walks are performed – left, right, up and down).

   The child state obtained at the end of each move is then assigned a **Heuristic value** based on an evaluation function. The heuristics are used as sorting criteria. In this informed search, reducing the state space search complexity is the main criterion. We define heuristic evaluations to reduce the states that need to be checked every iteration. Evaluation function is used to express the quality of informedness of a heuristic algorithm.

   Evaluation function $f(n) = g(n) + h(n)$

   $g(n)$ measures the actual length of the path from any state n to the start state
   $h(n)$ is a heuristic estimate of the distance from state n to a goal.

In this 8-puzzle solver, we have
$g(n)$ = actual distance from n to the start state
$h(n)$ = Number of tiles out of place + Sum of distance out of place + Number of direct tile of reversals

As can be seen, three heuristics are adopted in this implementation:

A. *Number of tiles out of place*: In this heuristic function, the heuristic value is incremented for each time the value of a tile between current and goal mismatches.

B. *Sum of distances out of place*: This heuristic evaluates the sum of number of steps needed to move a tile to its correct location (as in goal). Here, the 8-puzzle is treated as a 2d array and the absolute value of offsets between the correct location in goal state and deviant location in start state of the same tile is measured.

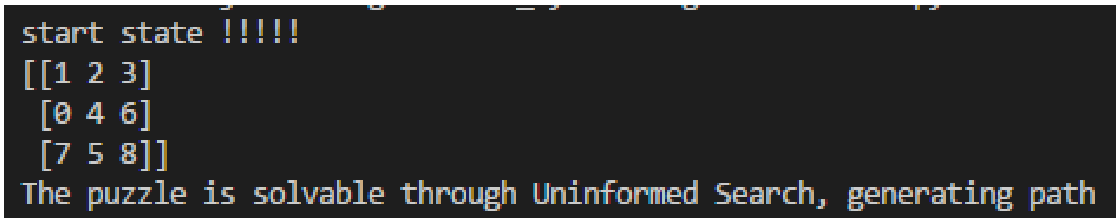
C. *Number of direct tile of reversals*: This heuristic calculates the number of adjacent tile pair whose locations are exactly reversed, and for each there is a tile reversal the heuristic value is incremented by one. Finally, this third heuristic is multiplied by a factor of 2.

Once all 3 heuristic is calculated, the values are summed up to get the final heuristic value of that tile. This is then added to the depth of current tile to get the final value of evaluation function.

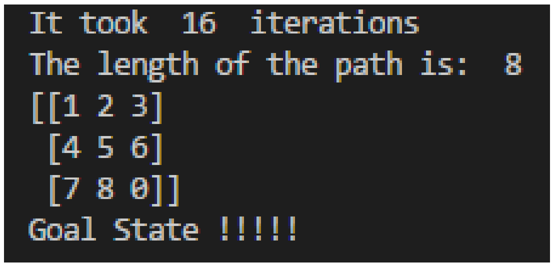
3. Comparison Analysis

Uninformed Search Algorithm

Sample Run 1:



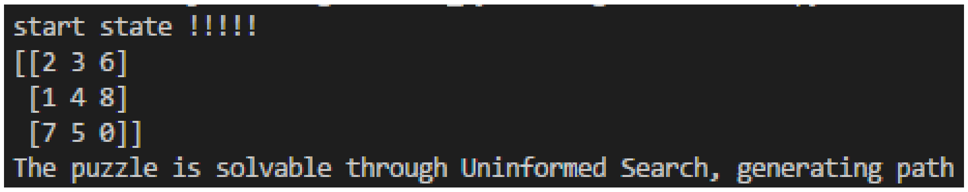
```
start state !!!!!
[[1 2 3]
 [0 4 6]
 [7 5 8]]
The puzzle is solvable through Uninformed Search, generating path
```



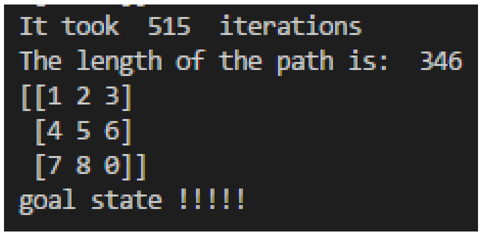
```
It took  16  iterations
The length of the path is:  8
[[1 2 3]
 [4 5 6]
 [7 8 0]]
Goal State !!!!!
```

Time Taken: 0.1230001449584961 s

Sample Run 2:



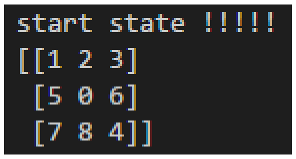
```
start state !!!!!
[[2 3 6]
 [1 4 8]
 [7 5 0]]
The puzzle is solvable through Uninformed Search, generating path
```



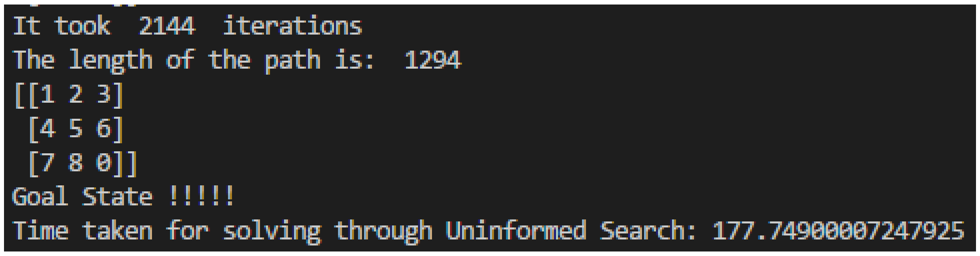
```
It took  515  iterations
The length of the path is:  346
[[1 2 3]
 [4 5 6]
 [7 8 0]]
goal state !!!!!
```

Time Taken: 28.54700016975403 s

Sample Run 3:



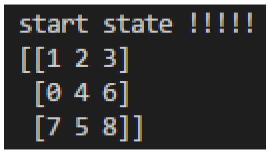
```
start state !!!!!
[[1 2 3]
 [5 0 6]
 [7 8 4]]
```



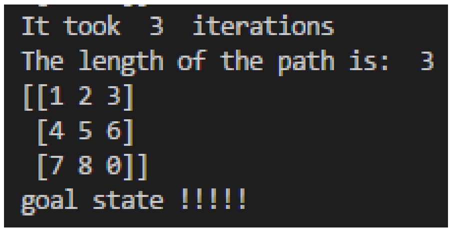
```
It took  2144  iterations
The length of the path is:  1294
[[1 2 3]
 [4 5 6]
 [7 8 0]]
Goal State !!!!!
Time taken for solving through Uninformed Search: 177.74900007247925
```

Informed Search Algorithm

Sample Run 1:



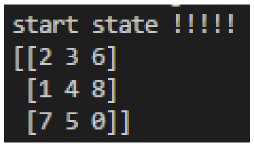
```
start state !!!!!
[[1 2 3]
 [0 4 6]
 [7 5 8]]
```



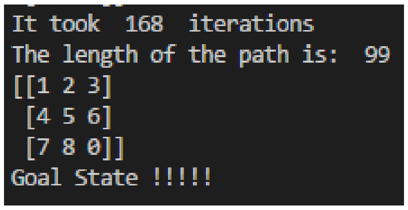
```
It took  3  iterations
The length of the path is:  3
[[1 2 3]
 [4 5 6]
 [7 8 0]]
goal state !!!!!
```

Time Taken: 0.05199980735778808 6 s

Sample Run 2:



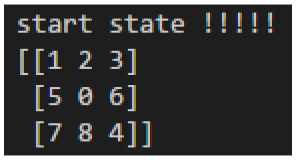
```
start state !!!!!
[[2 3 6]
 [1 4 8]
 [7 5 0]]
```



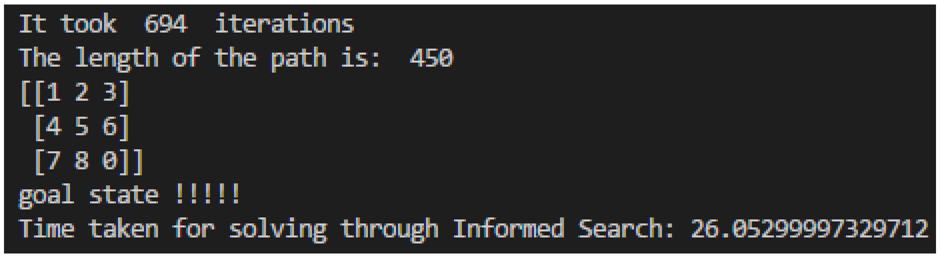
```
It took  168  iterations
The length of the path is:  99
[[1 2 3]
 [4 5 6]
 [7 8 0]]
Goal State !!!!!
```

Time Taken: 3.993000030517578 s

Sample Run 3:



```
start state !!!!!
[[1 2 3]
 [5 0 6]
 [7 8 4]]
```



```
It took  694  iterations
The length of the path is:  450
[[1 2 3]
 [4 5 6]
 [7 8 0]]
goal state !!!!!
Time taken for solving through Informed Search: 26.05299997329712
```

	Initial State	**Uninformed Search**			**Informed Search**		
		Time to Goal	**Iteration**	**Length**	**Time to Goal**	**Iteration**	**Length**
Sample Run 1	[[1 2 3] [0 4 6] [7 5 8]]	0.123 s	16	8	0.052 s	3	3
Sample Run 2	[[2 3 6] [1 4 8] [7 5 0]]	28.54 s	515	346	3.993 s	168	99
Sample Run 3	[[1, 2, 3] [5, 0, 6] [7, 8, 4]]	177.75 s	2144	1294	26.05 s	694	450

5. Conclusion

Clearly, the Informed Search using A* heuristic performs a better search than the Uninformed BFS. On comparing the number of nodes, it generated, the depth at which solution is found as well as the time it takes to get the solution path is all lesser in A* search when compared to BFS.

As can be seen from the results above, space and time are big problems for BFS. This is because the search implementing a BFS algorithm is known to take O (bd) time and space complexity, where b being the number of nodes and d being the depth of solution.

Whereas the time complexity of A* search is Exponential. Since this is a puzzle of 8 state spaces, the time complexity is lesser when compared to a n-puzzle game.