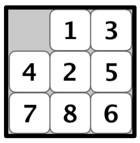
Artificial Intelligence - FALL 2021
Project 1 Sliding Block Puzzle

Xinxuan Lu

September 29, 2021

1 Program Design

To build up the solver, I firstly build a class Block which refers to each status of the sliding block during the game. The class includes several member variables that will shows the status of the block including the locations of each number which is represented in an array, the empty block will be represented by 0.(Example shows in Figure 1) . In this case, The blank is at right bottom, and in the project, the blank will be With the block object we can build a class named Solver which



```
array([[0, 1, 3],
       [4, 2, 5],
       [7, 8, 6]])
```

Figure 1: Example of Block to array

refers to the solver of the sliding block puzzle. The Solver class include two function that can be used to solve puzzle: BFS and A*.
The BFS function will explore all nodes at the present depth, in this case a node refers to a neighbor of our current sliding block state (neighbor means a state which can be reached in one step). For those state that has appeared before in the tree. The solver function will prune the sub-tree that root from the node.
The code of A* function is based on BFS function. Moreover, A* algorithm needs a score function which is used to judge which node the program should explore in the next loop. In the block class every block will have a score when I run the initialize function. Here n is the next node on the path, $g(n)$ is the cost of the path from the start node to n, and $h(n)$ is a heuristic function that estimates the cost of the cheapest path from n to the goal. So a neighbor state with lowest $f(n)$ refers to the state which may be the closest way to the goal in our estimation.

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

Here I use the Manhattan distance as the heuristic function.

$$h(\mathbf{c}, \mathbf{g}) = \|\mathbf{c} - \mathbf{g}\|_1 = \sum_{i=1}^{n} |c_i - g_i|$$

crefers to current state array and **g** refers to the goal state, both $\mathbf{c}, \mathbf{g}$ are vectors

$$\mathbf{p} = (c_1, c_2, \ldots, c_n), \mathbf{g} = (g_1, g_2, \ldots, g_n)$$

The A* function will sort those current existed state in the list with score function and choose the state with lowest score in the next loop.

2 Results

Firstly, we use the example in Figure 1 to test the two functions. The example is an extremely simple puzzle which can be done in 4 steps. Seems that both BFS and A* function can complete in a moment, so there is no essential bug in the code. Next step I will use the problem set to test the performance of both functions. (The example has blank in the bottom, I have change it to the top in the project)

To test the function I used the problem set that has already been provided to us. Here we have puzzles from easy to extreme level. For each level, I will use both function to solve every puzzles and calculate out the average time the functions cost to solve the puzzles. To make sure the effectiveness, I will also limit the function to complete the puzzle in 100000 loops (scan 10000 nodes) and time should be in 1 minute. If the function runs out of the limitation, I will label that

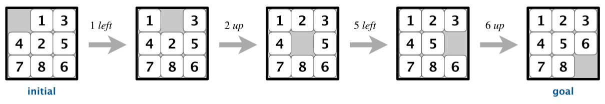
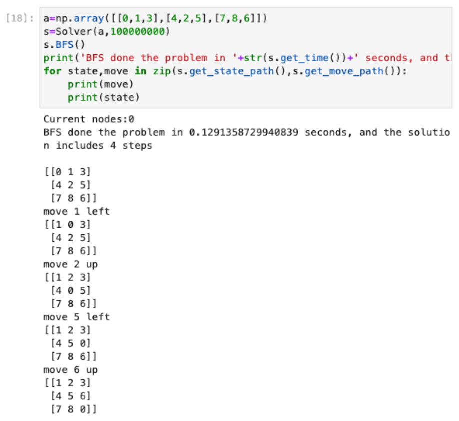


Figure 2: 4 moves sliding block puzzle

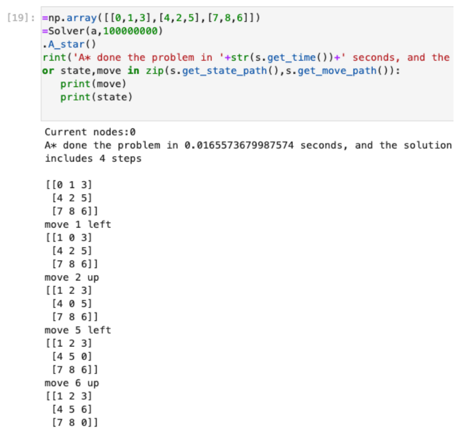


```
[18]: a=np.array([[0,1,3],[4,2,5],[7,8,6]])
      s=Solver(a,100000000)
      s.BFS()
      print('BFS done the problem in '+str(s.get_time())+' seconds, and t
      for state,move in zip(s.get_state_path(),s.get_move_path()):
          print(move)
          print(state)
```

```
Current nodes:0
BFS done the problem in 0.1291358729940839 seconds, and the solutio
n includes 4 steps

[[0 1 3]
 [4 2 5]
 [7 8 6]]
move 1 left
[[1 0 3]
 [4 2 5]
 [7 8 6]]
move 2 up
[[1 2 3]
 [4 0 5]
 [7 8 6]]
move 5 left
[[1 2 3]
 [4 5 0]
 [7 8 6]]
move 6 up
[[1 2 3]
 [4 5 6]
 [7 8 0]]
```

Figure 3: Result of BFS



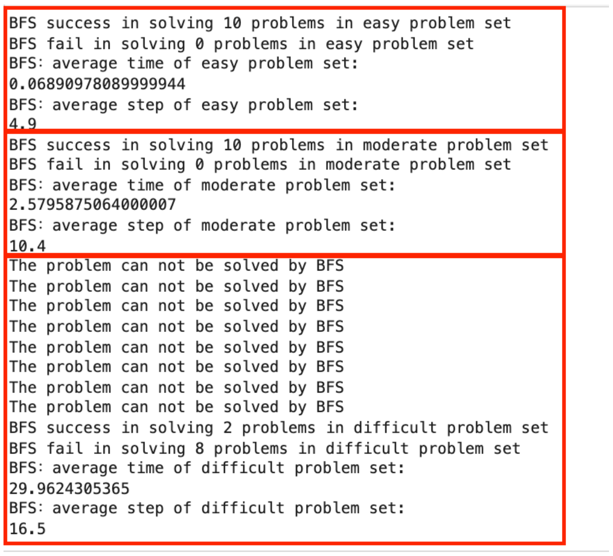
```
[19]: =np.array([[0,1,3],[4,2,5],[7,8,6]])
      =Solver(a,100000000)
      .A_star()
      rint('A* done the problem in '+str(s.get_time())+' seconds, and the
      or state,move in zip(s.get_state_path(),s.get_move_path()):
         print(move)
         print(state)
```

```
Current nodes:0
A* done the problem in 0.0165573679987574 seconds, and the solution
includes 4 steps

[[0 1 3]
 [4 2 5]
 [7 8 6]]
move 1 left
[[1 0 3]
 [4 2 5]
 [7 8 6]]
move 2 up
[[1 2 3]
 [4 0 5]
 [7 8 6]]
move 5 left
[[1 2 3]
 [4 5 0]
 [7 8 6]]
move 6 up
[[1 2 3]
 [4 5 6]
 [7 8 0]]
```

Figure 4: Result of A*

the function fail to solve the puzzle.
First I check the performance of BFS function and result shows in Figure 5. BFS failed in most of the difficult puzzles and takes up to 29 seconds in average to solve difficult puzzles in average. Apparently, difficult level puzzle will be too hard for it to solve. I use difficult $3x3_1$ problem to test whether the BFS function can solve the puzzle without limitation. The result shows:
BFS done the problem in 8.59333631200002 seconds, and the solution includes 15 steps.
The BFS can solve the puzzle and pick out the optimal solution, but it takes a long time. Next

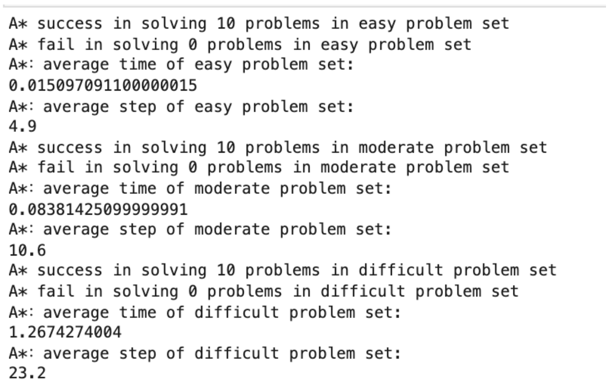


```
BFS success in solving 10 problems in easy problem set
BFS fail in solving 0 problems in easy problem set
BFS: average time of easy problem set:
0.06890978089999944
BFS: average step of easy problem set:
4.9
BFS success in solving 10 problems in moderate problem set
BFS fail in solving 0 problems in moderate problem set
BFS: average time of moderate problem set:
2.5795875064000007
BFS: average step of moderate problem set:
10.4
The problem can not be solved by BFS
The problem can not be solved by BFS
The problem can not be solved by BFS
The problem can not be solved by BFS
The problem can not be solved by BFS
The problem can not be solved by BFS
The problem can not be solved by BFS
The problem can not be solved by BFS
BFS success in solving 2 problems in difficult problem set
BFS fail in solving 8 problems in difficult problem set
BFS: average time of difficult problem set:
29.9624305365
BFS: average step of difficult problem set:
16.5
```

Figure 5: Performance of BFS function

I check the performance of A* function, result shows in Figure 6. Compared to BFS function A* function is much more faster. It complete difficult level function in 1.26 seconds in average. By

contract, A* can not always find the optimal solutions. Take difficult 3x3_1 as an example:
BFS done the problem in 8.59 seconds, and the solution includes 15 steps
A* done the problem in 2.18 seconds, and the solution includes 25 steps
Unfortunately neither of the functions can solve extreme problem in limited time. It shows that



```
A* success in solving 10 problems in easy problem set
A* fail in solving 0 problems in easy problem set
A*: average time of easy problem set:
0.015097091100000015
A*: average step of easy problem set:
4.9
A* success in solving 10 problems in moderate problem set
A* fail in solving 0 problems in moderate problem set
A*: average time of moderate problem set:
0.08381425099999991
A*: average step of moderate problem set:
10.6
A* success in solving 10 problems in difficult problem set
A* fail in solving 0 problems in difficult problem set
A*: average time of difficult problem set:
1.2674274004
A*: average step of difficult problem set:
23.2
```

Figure 6: Performance of A* function

even the A* function is not effective enough to solve complicated 4x4 blocks puzzle. Maybe this issue can be solved with the coming knowledge of our class.
p.s.: I am still running the extreme puzzle with A* function and the program has been continuing more than 5 hours. I am curious to see whether it can finally run out of the solution.

3 Discussion

- Challenge:

  How to find the heuristic function is really a challenge. As I use matrix to represent the sliding box. I can just use distance between matrix as the heuristic. I have tried different distances and choose Manhattan distance. Actually there are more accurate distance functions that can represent the similarity between the state and the goal state. But, heuristic is just an estimation function and has no need to be totally accord with the true value. Above all, it can represent the similarity and is easy to calculate. Here I listed some common used matrix distance formula:

$$\begin{aligned}
d_1(\mathbf{A},\mathbf{B}) &= \sum_{i=1}^{n}\sum_{j=1}^{n}|a_{ij}-b_{ij}| \\
d_2(\mathbf{A},\mathbf{B}) &= \sqrt{\sum_{i=1}^{n}\sum_{j=1}^{n}(a_{ij}-b_{ij})^2} \\
d_\infty(\mathbf{A},\mathbf{B}) &= \max_{1\le i\le n}\max_{1\le j\le n}|a_{ij}-b_{ij}| \\
d_m(\mathbf{A},\mathbf{B}) &= \max\{\|(\mathbf{A}-\mathbf{B})\mathbf{x}\| : \mathbf{x}\in\mathbb{R}^n, \|\mathbf{x}\|=1\}
\end{aligned}$$

- Further interesting:

  This project truly helped me to throughly review the concepts of BFS and A* algorithms. It also trained me about how to represent a puzzle with programming language. I am also curious about how to solve 4x4 block puzzle effectively. Maybe I can discover the answer in the upcoming courses.