CS540 Fall 2024 Homework 8
A* beyond 8-tile Puzzle
Due on Nov 18th, 9:30 AM

Assignment Goals

- Deepen understanding of state space generation.
- Practice implementation of an efficient search algorithm.

Summary

This assignment is about solving variants of the 8-tile puzzle that we have discussed in class. The 8-tile puzzle was invented and popularized by Noyes Palmer Chapman in the 1870s. The version we will consider in this homework is played on a 3x3 grid with less than 8 tiles, for example, 7 tiles labeled 1 through 7 and two empty grids. The goal is to rearrange the tiles so that they are in order and the empty places are at the bottom right.

You solve the puzzle by moving the tiles around. For each step, **you can only move one (not more!)** of the neighbor tiles (**left, right, top, bottom but not diagonally**) into an empty grid. And all tiles must stay in the 3x3 grid (so no wrap around allowed). An example is shown in the picture below. Suppose we start from the following configuration:

2	5	1
4	3	6
7		

Then, moving one tile can result in one of the following:

2	5	1
4		6
7	3	

2	5	1
4	3	
7		6

2	5	1
4	3	6
	7	

That is, in the above example, we would either move 3 or 6 down or move 7 to the right. Given these rules for the puzzle, you will generate a state space and solve this puzzle using the A* search algorithm.

Program Specification

The code for this program should be written in Python, in a file called `funny_puzzle.py`. We will provide states in a one-dimensional list of integers, with the empty spaces represented as `0`. For example, in the picture above, the initial state is represented by `[2,5,1,4,3,6,7,0,0]` and its successors are `[2,5,1,4,0,6,7,3,0]`,`[2,5,1,4,3,0,7,0,6]`,`[2,5,1,4,3,6,0,7,0]`.

In this assignment, you will need a priority queue. We highly recommend using the package `heapq` for the implementation. You should refer to heapq if you are not familiar with it.

Goal State

The goal state of the 7-tile puzzle is [1, 2, 3, 4, 5, 6, 7, 0, 0], or visually:

1	2	3
4	5	6
7		

The goal state of 6-tile will be [1, 2, 3, 4, 5, 6, 0, 0, 0], or visually:

1	2	3
4	5	6

Different numbers of tiles have different goal states. The instruction takes 7 tiles as an example.

Heuristic

Since we are using the A* search algorithm, we need a heuristic function `h(s)`. Recall the Manhattan distance mentioned in lecture (the l1-norm). We will use the sum of Manhattan distance of each tile to its goal position as our heuristic function. The Manhattan distance of two tiles in this case is the absolute difference between their x coordinates plus the absolute distance between their y coordinates.

In our first example puzzle (`[2,5,1,4,3,6,7,0,0]`), the `h()` is 6. This is computed by calculating the Manhattan distance of each title and summing them. Specifically, tiles 4/6/7 are already in place, thus they have 0 distances. Tile 1 has a Manhattan distance of 2 (manhattan([0,2], [0,0]) = abs(0-0) + abs(2-0) = 2), tiles 2/3/5 have distances of 1/2/1, respectively.

Caution: do not count the distance of tiles '0' since they are actually not tiles but are empty sections.

Functions

For this program you need to write two (2) Python functions:

1. `print_succ(state)` — given a state of the puzzle, represented as a single list of integers with a 0 in the empty spaces, print to the console all of the possible successor states.
2. `solve(state)` — given a state of the puzzle, perform the A* search algorithm and print the path from the current state to the goal state.

You may, of course, add any other functions you see fit, but these two functions must be present and work as described here.

Print Successors

This function should print out the successor states of the initial state, as well as their heuristic value according to the function described above. The number of successor states depends on the current state.

```
>>> print_succ([2,5,1,4,0,6,7,0,3])
[2, 0, 1, 4, 5, 6, 7, 0, 3] h=5
[2, 5, 1, 0, 4, 6, 7, 0, 3] h=7
[2, 5, 1, 4, 0, 6, 0, 7, 3] h=7
[2, 5, 1, 4, 0, 6, 7, 3, 0] h=7
[2, 5, 1, 4, 6, 0, 7, 0, 3] h=7
```

We do require that these be printed in a specific order: if you consider the state to be a nine-digit integer, the states should be sorted in ascending order. Conveniently, if you ask Python to sort one-dimensional arrays, it will adhere to this order by default; don't do more work than you have to:

```
>>> lists = [[2, 0, 1, 4, 5, 6, 7, 0, 3],
 [2, 5, 1, 4, 6, 0, 7, 0, 3]
 [2, 5, 1, 4, 0, 6, 7, 3, 0],
 [2, 5, 1, 4, 0, 6, 0, 7, 3],
 [2, 5, 1, 0, 4, 6, 7, 0, 3]]
>>> sorted(lists)
 [[2, 0, 1, 4, 5, 6, 7, 0, 3], [2, 5, 1, 0, 4, 6, 7, 0, 3], [2, 5, 1, 4, 0,
 6, 0, 7, 3], [2, 5, 1, 4, 0, 6, 7, 3, 0], [2, 5, 1, 4, 6, 0, 7, 0, 3]]
```

Priority Queue

Now is a good time to implement the priority queue in your code. We recommend you use the python library `heapq` to create your priority queue. Here is a quick example:

```
>>> import heapq
>>> pq = []
>>> heapq.heappush(pq,(5, [1, 2, 3, 4, 5, 0, 6, 7, 0], (0, 5, -1)))
>>> print(pq)
 [(5, [1, 2, 3, 4, 5, 0, 6, 7, 0], (0, 5, -1))]
```

The code will push an item `([1, 2, 3, 4, 5, 0, 6, 7, 0], (0, 5, -1))` with priority 5 into the queue. It would be useful to do pushes of the form

heapq.heappush(pq ,(cost, state, (g, h, parent_index)))

where pq is the priority queue, g and h are the values of the functions that we defined in L17 (g is the cost from the starting node, which in our case will be the number of moves so far, and h is the value of the heuristic function) and cost=g+h (this is what we want to use as priority). A parent index of -1 denotes the initial state, without any parent. For more details, please refer to the documentation of heapq.

To get the final path, for each element in the priority queue we need to remember its parent state. Remember to store the state when you pop it from the priority queue, so you could refer to it later when you generate the final path. Here is how you can pop from a priority queue

```
>>> b = heapq.heappop(pq)
>>> print(b)
 (5, [1, 2, 3, 4, 5, 0, 6, 7, 0], (0, 5, -1))
>>> print(pq)
 []
```

The priority queue is stored in a list and, for debugging, you can print its items (with the associated priority) by using

```
 >>> # assume that you have generated and enqueued the successors
 >>> print(*pq, sep='\n')
  (7, [1, 2, 0, 4, 5, 3, 6, 7, 0], (1, 6, 0))
  (7, [1, 2, 3, 4, 0, 5, 6, 7, 0], (1, 6, 0))
  (7, [1, 2, 3, 4, 5, 0, 6, 7, 0], (1, 6, 0))
```

Note that the `heappush` maintains the priority in ascending order, i.e., `heappop` will always pop the element with the smallest priority. We require that the states with the same cost (priority) to be popped in a specific order: if you consider the state to be a nine-digit integer, the states should be sorted in ascending order - just like we mentioned above. If you follow the format in `pq` as shown above, `heapq` will automatically take care of this and you do not need more work.

It might be unable to find the final path for some initial states, leading to an **infinity loop**. You are encouraged to do a solvability check before running into an infinity loop.

Solve the Puzzle

If the puzzle is not solvable, the function should only print(False).

If the puzzle is solvable, the function should print(True), then print the solution path from the provided initial state to the goal state, along with the heuristic values of each intermediate state according to the function described above, and total moves taken to reach the state. Recall that our cost function g(n) is the total number of moves so far, and every valid successor has an additional cost of 1.

You're encouraged to compute max queue length for debugging purposes.

```
>>> solve([4,3,0,5,1,6,7,2,0])

  True
 [4, 3, 0, 5, 1, 6, 7, 2, 0] h=7 moves: 0
 [4, 0, 3, 5, 1, 6, 7, 2, 0] h=6 moves: 1
 [4, 1, 3, 5, 0, 6, 7, 2, 0] h=5 moves: 2
 [4, 1, 3, 0, 5, 6, 7, 2, 0] h=4 moves: 3
 [0, 1, 3, 4, 5, 6, 7, 2, 0] h=3 moves: 4
 [0, 1, 3, 4, 5, 0, 7, 2, 6] h=4 moves: 5
 [0, 1, 3, 4, 0, 5, 7, 2, 6] h=5 moves: 6
 [0, 1, 3, 4, 2, 5, 7, 0, 6] h=4 moves: 7
 [1, 0, 3, 4, 2, 5, 7, 0, 6] h=3 moves: 8
 [1, 2, 3, 4, 0, 5, 7, 0, 6] h=2 moves: 9
 [1, 2, 3, 4, 5, 0, 7, 0, 6] h=1 moves: 10
 [1, 2, 3, 4, 5, 6, 7, 0, 0] h=0 moves: 11
Max queue length: 163
```

Submission Notes

Please only submit your file **funny_puzzle.py**.

Your code should contain the functions mentioned above. There is no need to have code under a if __name__ == "__main__"

Do not submit a Jupyter notebook .ipynb file. You can use numpy and anything in the standard python library.

This assignment is due on Nov 18th, 9:30AM. We highly recommend starting early. It is preferable to first submit a version well before the deadline (at least one hour before) and check the content/format of the submission to make sure it's the right version. Then, later update the submission until the deadline if needed.

Grading

The assignments will be graded in the 0-100 scale as follows:

- (40 points) `print_succ()` successfully prints the successor configurations along with their h values.
- (60 points) `solve()` successfully judge the solvability of the test case, and if it's solvable, print the solution to the puzzle along with the h values, number of moves of the intermediate configurations. You will not be graded on the max queue length.

The test cases might be 8-tile, 6-tile, 5-tile or other variants within 3*3 grids. For example, the goal state for 8-tile would be [1, 2, 3, 4, 5, 6, 7, 8, 0], and that for 6-tile would be [1, 2, 3, 4, 5, 6, 0, 0, 0]. You are encouraged to design cases beyond 7-tile, and design unsolvable cases for solvability check.

Be sure to follow the printing format specified above and remove debugging output before submission.