AI_HW

Homework 1 for Artificial Intelligence W4701

To run hw1_rx2119.py

```
usage: python hw1_rx2119.py [-random N-size R-num | -test -method LIST]
```

Eg. to generate 5 random tests for 8-puzzle(n=3) game using all four methods(BFS, DFS, A*, IDA*) to solve

```
python hw1_rx2119.py -r 3 5
```

Eg. to test solver for a specific configuration like the one show below with all methods(`-a`)

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

```
python hw1_rx2119.py -t -a 5,8,1,6,3,2,0,4,7
```

or just test it with one method like A* (`-bfs`, `-dfs`, `-astar`, `-idastar`)

```
python hw1_rx2119.py -t -astar 5,8,1,6,3,2,0,4,7
```

I also attached an output.txt file which a set of randomly generated solvable 8-puzzle game for BFS, DFS, and A*. random puzzles are generated by calling shuffle method, which shuffles the goal state board for 200 randomly moves. It's not truly a uniformly distributed random method. But it does produce non-preconfigured solved puzzles for testing. Because the DSF too long from time to time produces solution that is too long. In the output.txt file I only printed solutions that are less than 50 moves. (for this file, a 60,000 long path solution is included in dfs.txt file) However, if I did implement a `validate()` method and an `assert` that validate the correctness of all solution, which make testing more convenient.

1. **BFS is implemented according to the specification**: I implemented BFS using a `deque` as the queue for nodes, I also have a visited set, which is a set of tuple(immutable and hashable to increase efficiency) to record all configuration that has been expanded. I check the goal state before add nodes to the queue which is a farther optimization.
2. **DFS is implemented according to the specification**: The implementation for DFS is similar to BFS, the only difference is, it is implemented with a list as stack, to release FIFO.
3. **A* is implemented according to the specification**: A* like the previous two methods had similar optimization with visited set and goal state check. It uses `list` and `heapq` as it's priority queue and it's value is the f function (= cost from the origin state + heuristic cost which is the manhattan distance for all tiles except Zero, the empty cell).
4. **Summarize and compare**

```
initializing for BFS
```

```
 4 2 3
 7 6 1
 5   8
--- 246.860027313 milliseconds --
n = 3
path to solution is ['UP', 'UP', 'RIGHT', 'DOWN', 'LEFT', 'LEFT', 'DOWN', 'RIGHT',
'UP', 'LEFT', 'UP', 'RIGHT', 'RIGHT', 'DOWN', 'LEFT', 'LEFT', 'UP']
Cost of path = 17
Memory Usage: 37 KB
nodes expanded = 10674
max stack/queue size = 5883
max depth of the stack/ queue 18

initializing for DFS
 4 2 3
 7 6 1
 5   8
--- 2745.47219276 milliseconds --
n = 3
Cost of path = 66491
Look at the file dfs.txt inside the zip file
Memory Usage: 227 KB
nodes expanded = 91946
max stack/queue size = 42440
max depth of the stack/ queue 66492

initializing for ASTAR
 4 2 3
 7 6 1
 5   8
--- 11.3000869751 milliseconds --
n = 3
path to solution is ['LEFT', 'UP', 'RIGHT', 'UP', 'RIGHT', 'DOWN', 'LEFT', 'DOWN',
'LEFT', 'UP', 'UP', 'RIGHT', 'RIGHT', 'DOWN', 'LEFT', 'LEFT', 'UP']
Cost of path = 17
Memory Usage: 10 KB
nodes expanded = 126
max stack/queue size = 75
max depth of the stack/ queue 18
```

Cost of the path * **BFS**: It always returns the shortest which is also the optimal solution, in test case it's no different with path of 17. * **DFS**: Unusually returns some deep and long solution in this case 66491, which it not printed out for obvious reason. * **A***: Return the shortest and optimal solution like BFS in this case also it's 17.

Number of nodes expanded * **BFS**: It expands nodes layer by layer, in this case because the goal state is not very deep, it only traversed 10,674 nodes, theoretical speaking it has to traverse B**M but since I implemented duplicate check, it just have to find check all unique configurations (for 8-puzzle is 9!) up until level 17.** * **DFS**: It goes straight down which expanded 91,946 nodes before reached goal state. It's not the most efficient way to solve the problem, obviously. * **A***: Using rule of thumb it is able to find the optimal solution only expanding 126 nodes, because it expands the most promising direction first based on it's distance from origin and manhattan distance to goal state.

Max depth of the stack/queue

```
As the TA explained it on Piazza, it is the depth of the tree when a solution is
found. It's related to the tree path. I will talk more about it related to memory
later.
```

* **BFS**: the depth of the tree 18. * **DFS**: the depth of the tree 66,492. * **A***: the depth of the tree 18.

Memory requirements of each approach

The methods are run separately to monitor their memory usage. I imported `resource` for memory usage monitoring. The memory usage is related to the number of nodes expanded and the max stack/queue size, the exact number are hard to calculate because because python's automatic garbage collector, and type inference, but the general view is that the more node expand and large max stack/queue size is the more memory it will consume. On that note, we can assume that in a DFS for an average configuration board will consume most memory and expand most node and have a significant size stack, and A* being the most efficient in note expanded and and heap size, while BFS being the middle, however we should be aware that DFS is actually quite efficient in memory usage with Space of O(B*M) much better that the other two, however because it usually extends quite deep into the tree, so in practice space performance is not as it looks on paper. * **BFS**: 37KB * **DFS**: 227KB * **A***: 10KB

Running time * **BFS**: Since the solution is not very deep in the tree and branch factor is controlled through a set for duplicated states, the runtime is as below. --- `246.860027313 milliseconds` -- * **DFS**: Because it charges straight down, regardless of the direction that will lead to optimal results. It is slower than the other two methods. --- `2745.47219276 milliseconds` -- * **A***: Because it's using rule of thumb and duplicated check it will run in shortest time. --- `11.3000869751 milliseconds` --

Justify your choice of a heuristic I chose to calculate manhattan distance for each element in the table from its current position to the its goal state position except the empty cell as the heuristic class. This heuristic is admissible because, The to reach the goal state this is the least number of move we have to make, and in most cases we have to make more move that this, because we can only swap empty cell and it's adjacent cell to change configuration. This mean it underestimate the cost to reach goal state and thus lead to optimal solution by A* search.

Discuss your knowledge representation board configuration are kept as a `list` in side State class. It has n as in n-puzzle, `full_size` for the length of the list, which is asserted to be the n**2, `move` which is the last move make default the be `None`, `child` is a list of adjacent list used for simulation, while `makeMove(direct)` actually moves the node for validation(`validate`) and testing purpose. `parent` specifies the parent of each node and default to be `None` for root node. Users can also specify a configuration with argument `original`. A State class instance in this case represent a node and are simulated for variable of methods that solved n-puzzles.

IDA* is implemented according to the specification Essentially it's and a DFS that iterate on the f function of the node that it traverse. I didn't use a visited set for duplication, because it may lead to suboptimal solution. However I did optimize it by using prevent node to go the direct that it comes from, and start iteration from F value of the node. These two method speed up IDA* quite a lot. It's at least on par if not faster that the A*, the `output.txt` file also included it's test results as well.