CSDS 391: Artificial Intelligence

Programming HW1: Written Explanation

September 23, 2020

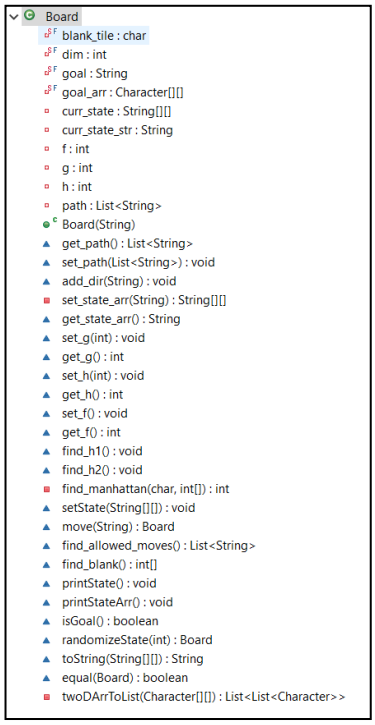
Yonghao Zhou (yxz1754)

1. Code Design

My code consists of two main classes and one supplemental classes. The "Solver" class consists of the 3 searches implemented, as well as setting the max_nodes for each search. The "Board" class consist of the representation of the puzzle instance. The two supplemental classes are the comparators created used to sort the boards given f(n).

The "Board" class contains the representation of the puzzle and encloses several important utility functions in order to run the puzzle properly. The class also contains many getter and setter methods since the instance fields are private.

The class has these instance fields:



a. **curr_state:** The current state of the board represented in a 3x3 array.
b. **curr_state_str:** The currents state of the board in string representation, in format: "b12 345 678".
c. **f(n), g(n), h(n)**
d. **path:** The path of the ancestors of this puzzle instance in a list.

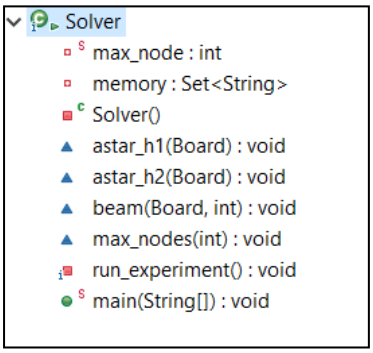
The class has these major functional methods:

a. **find_h1():** This heuristic is based on the number of misplaced tiles.
b. **find_h2():** This heuristic is the sum of the Manhattan distance of each tile.
c. **setState(String[][] state):** This method sets the current state of the board, given the input state.
d. **move(String direction):** This method makes a copy of the current state and moves the blank tile into the specified direction in the new state. This creates a child node and assigns the new state, increases g(n) and sets the parent to this.
e. **find_allowed_moves():** This method gets the possible moves in {up, down, left, right} given the blank tile location.
f. **find_blank():** This method finds the location of the blank tile, and throws exception if not found.
g. **randomizeState(int n):** This method adopts the current board and makes n random moves from the goal state.
h. **equal():** This method compares the state value of two boards.

The "Solver" class contains the implementation of the definition of A* search and beam search to search for the solution of the 8 puzzle. A* search takes in a heuristic, either h1 or h2 and uses it to increase the efficiency in solving the 8 puzzle. A* search will take the child with the lowest f(n) as the next move,

while beam search will keep the k successor states with the lowest f(n) and expands on all of those k successors and ignore the others.

The class has a two static fields:



a. **max_node**: The limit on the nodes that can be generated in a single search.
b. **memory**: This Set stores the visited nodes and stores their string representation to prevent revisiting.

In order to prevent going back to states where the puzzle has been seen before, I used a Set to keep string representation of the states and check every time where a move is permitted.

The "Solver" class also contains the main method of the program so that a text file can be read as an input for commands. There is also a private method that can be called to run the experiments.

2. Code Correctness

Below is the output for the input file I have created.

```
READING command: printState
        PRINTING STATE = [[b, 1, 2], [3, 4, 5], [6, 7, 8]]
READING command: move up
        MOVING BLANK TILE UP
java.lang.IllegalArgumentException: INVALID DIRECTION
READING command: printState
        PRINTING STATE = [[b, 1, 2], [3, 4, 5], [6, 7, 8]]
READING command: move down
        MOVING BLANK TILE DOWN
READING command: printState
        PRINTING STATE = [[3, 1, 2], [b, 4, 5], [6, 7, 8]]
READING command: move left
        MOVING BLANK TILE LEFT
java.lang.IllegalArgumentException: INVALID DIRECTION
READING command: printState
        PRINTING STATE = [[3, 1, 2], [b, 4, 5], [6, 7, 8]]
READING command: move right
        MOVING BLANK TILE RIGHT
READING command: printState
        PRINTING STATE = [[3, 1, 2], [4, b, 5], [6, 7, 8]]
READING command: randomizeState 1000
        RANDOMIZING STATE WITH N = 1000
READING command: printState
        PRINTING STATE = [[7, 2, 5], [6, b, 1], [3, 8, 4]]
READING command: maxNodes 10000
        SETTING MAX NODE = 10000
MAX NODE SET TO 10000
READING command: setState 725 6b1 384
        SETTING STATE = [[7, 2, 5], [6, b, 1], [3, 8, 4]]
READING command: solve A-star h1
STARTING A* WITH H1
DONE WITH A* H1
NUMBER OF NODES: 7195
TIME TAKEN: 298212800 NS
SIZE: 20, STEPS: [right, up, left, down, left, up, right, right, down, down, left, up, left, down, right, up, right, up, left, left]
READING command: solve A-star h2
STARTING A* WITH H2
DONE WITH A* H2
NUMBER OF NODES: 1457
TIME TAKEN: 19815900 NS
SIZE: 20, STEPS: [right, up, left, down, left, up, right, right, down, down, left, up, left, down, right, up, right, up, left, left]
READING command: solve beam 5
STARTING LOCAL BEAM WITH K=5
DONE WITH LOCAL BEAM WITH K=5
NUMBER OF NODES: 495
TIME TAKEN: 4809800 NS
SIZE: 50, STEPS: [right, down, left, left, up, up, right, down, down, left, up, right, right, up, left, left, down, down, right, up, left, up, right, right, down,
left, up, left, down, right, right, up, left, down, left, up, right, down, down, left, up, up, right, down, left, down, right, up, up, left]
```

The total number of nodes generated from A* search using h1 heuristics is larger than both the numbers of A* search with h2 heuristics and local beam search. Both A* searches with different heuristics produce the same solution size, however they differ in the number of nodes generated. This is due to h2 presents a better heuristic than h1, as h2 is at least as big as h1 for all nodes. The Manhattan distance is also a heuristic function with higher values compared to misplaced tiles.

Local beam search with k = 5 produces the least number of total nodes used in a single search. However, the size of the solution path is much larger as local beam search is not always optimal. Local beam

expands on the k best f(n) successors, considers all the generated child nodes and if no solution is found, trims and keeps only the first k best f(n) children, and continues the search. Therefore, it is much faster compared to A* search, as A* only considers the best f(n) child and traverses on that single child. However, that makes local beam search suboptimal as it doesn't always find the optimal solution. The local beam search uses heuristic h2 from the A* search and at the goal state, assuming the weight of each piece is given a value of 1 – since h2 is 0 at the goal state, the evaluation function is also 0.

3. Experiments

The fraction of solvable puzzles increases with increasing max nodes limit. I have included details of different tests I ran given different number of max nodes under the text file outputs named, "*output_statistics.txt*". The failure rate is calculated as the portion of tests ran that had the total number of nodes generated overflowed the set max node limit.

For A* search, if the start state is close from the goal state, h1 is a better heuristic option because it does less calculations compared to h2. Otherwise, h2 is better as it presents a better heuristic than h1.

For solution length, A* with h2 is identical to A* with h1 (*most of the time, h2 had identical length as h1)* and both are better than beam search. However, since beam search is an incomplete search, it is faster than A* as it has a limit on the size of the frontier.

Given the max nodes set to 15000 and running 100 different puzzles, the success rate of A* with h1 was 54%, A* with h2 was 96%, and beam search was 100%. I think by lowering the max nodes significantly, we can see a decrease in the success rate of the solution being found. A rerun of the experiment at a max node of 5000, leads to a statistics of the success rate of A* with h1 being 22%, A* with h2 being 83%, and beam search being 100%. While keeping the max nodes at 500, resulted in 3% success rates for A* with h1, and 8% with h2, and 38% for local beam search.

4. Discussions

Based on my experiments, I believe A* search with h2 heuristics is a better option for this problem. Although A* search with both h1 and h2 produced the shortest solutions since both heuristics are admissible, using h2 heuristics decreased the speed of the search and the number of total nodes generated in a single search.

Local beam search seems superior in terms of time and space as it outperforms A* in both categories. Local beam search also had the lowest failure rate among the three search algorithms, especially in limited max nodes. However due to local beam search does not always produce the optimal solution, I would still choose A* search with h2 heuristics as the size of the solution is much larger than A* using heuristics h1 and h2.

I found that actually showing the results are optimal or not seemed difficult. From using two heuristics, I can see how A* with two different admissible heuristics should produce the same solution.