EECS 391
Write up for P1
Qiwen Luo qxl216

1. Code Design

- I design this project from bottom to top. The main idea for this project is to use A-star search or local beam search to solve a randomly initiated 8-puzzle.
- Private static final int[][] goalState contains the goal state ({0,1,2},{3,4,5},{6,7,8})
- Private static final String[][] moveDirections contains all move directions ({"up", "down", "left", "right"})
- Private staic int[][] currentState is used to record the current state. It is initialized to goal state.
- Private static PriorityQueue<Tile> queue is used to record all information of the current state as Tile object.
- Private static int maxLimit is used to record the max number of nodes allowed. It is initialized to 100000.
- Private static int node is used to record the number of nodes that are considered during the search.
- Private Random r is a random generator to randomize move directions in randomizeState.
- Private static int h1Solved is used to record the number of puzzle solved by A-star search with the first heuristic function. This is designed for (a) in Experiment.
- Private static int h2Solved is used to record the number of puzzle solved by A-star search with the second heuristic function. This is designed for (a) in Experiment.
- Private static int beamSolved is used to record the number of puzzle solved by beamsearch. This is designed for (a) in Experiment.
- There are two versions for **setState**. One inputs the puzzle tile positions as a single string. It is used when setState is called directly in the input text file. It will set the current state. The other inputs an integer double array which represents a state and the puzzle tile positions as a string. It is used to set candidate states during the search.
- There are two versions for **printState**. One inputs nothing. It prints the current state in A star search. The other inputs an integer double array which represents a state. It is used to print a specific state and is usually used when debugging.
- There are two versions for move. One inputs the direction that the blank tile is going to move as a string. It moves the current state. The other inputs an integer double array which represents a state and the direction that the blank tile is

going to move as a string. It is used when we try to move the blank tile in a candidate state during the search.

- For **randomizeState**, it inputs the number of random moves as an integer. It is used when we try to randomize the current state from the goal state.
- For **solveAStar**, we input a heuristic function as a string.
  - ⇨ There are two different heuristic functions available. One is findHFirst which uses the number of misplaced tiles as heuristic. The other is findHSecond which uses the sum of distances of the tiles from their goal positions.
  - ⇨ If the goal state is found as the head of the priority queue, the search stop and start tracing the path from the start state to the goal state.
  - ⇨ Else, we generate all child states by implementing all available moves. Then we iteratively do A star search.
- For **solveBeam**, we input the number of states that we keep tracking as an integer.
  - ⇨ I use the same evaluation function as the second heuristic in A star search.
  - ⇨ If the goal state is found as the head of the priority queue, the search stop and start tracing the path from the start state to the goal state.
  - ⇨ Else, we generate all child states by implementing all available moves. Then we recursively do local beam search with k states.
- For **printMoveSequences**, it inputs a Tile object. It is used to trace the path from the start state to the goal state.
- For **maxNodes**, it inputs an integer which indicates the max node limit. It is used to set the number of max nodes allowed.
- For the **anonymous helper method below**, a new comparator is created to compare the f = g + h between two Tile objects. With the help with the custom-designed comparator, the priority queue will be ordered by the f = g + h in an ascending order.

```
private static Comparator<Tile> cmp = new Comparator<Tile>(){
  public int compare(Tile nodeptr1, Tile nodeptr2){
    int f1 = nodeptr1.g + nodeptr1.h;
    int f2 = nodeptr2.g + nodeptr2.h;
    return f1 - f2;
  }
};
```

- For the **helper method isAvailable**, it inputs an integer double array which represents a state and an integer indicating the direction of moves in the array called moveDirections.

```
private static final String[] moveDirections = {"up", "down", "left", "right"};
```

- For the **helper method findBlank**, it inputs an integer double array which represents a state. It is used to find the position of the blank tile in the state.
- For the **helper method isFinished**, there are two versions. One inputs nothing and check whether the current state is equal to the goal state. One inputs an integer double array which represents a state and check whether the state input is equal to the goal state.
- For the **helper method myClone**, it inputs an integer double array which represents the state to be cloned. It returns the same state as the input state.
- For the **helper class Tile**, it has five private fields.
  - ⇨ Tile parent: the Tile objects which contains information of the last state.
  - ⇨ Int[][] state: the current state
  - ⇨ Int g: the actual cost to reach this state from the start state
  - ⇨ Int h: the estimate cost from the current state to the goal state
  - ⇨ Int direction: how to move from the last state to the current state

2. Code correctness

Solve A-star

Input file:

```
setState b12 345 678
printState
move right
printState
randomizeState 2
printState
solve A-star h1
randomizeState 10
solve A-star h2
maxNodes 10000
randomizeState 10
solve beam 7
```

Output:

```
setState b12 345 678
printState
0  1  2
3  4  5
6  7  8
```

move right
printState
1 0 2
3 4 5
6 7 8

randomizeState 2
printState
3 1 2
4 0 5
6 7 8

solve A-star h1
Number of tile moves 2
Start state:
3 1 2
4 0 5
6 7 8

left
3 1 2
0 4 5
6 7 8

up
0 1 2
3 4 5
6 7 8

Total nodes generated for solve A-star 8
Total Time for solve A-star 234ms

randomizeState 10
solve A-star h2
Number of tile moves 4
Start state:

3 1 0
4 5 2
6 7 8

down
3 1 2
4 5 0
6 7 8

left
3 1 2
4 0 5
6 7 8

left
3 1 2
0 4 5
6 7 8

up
0 1 2
3 4 5
6 7 8

Total nodes generated for solve A-star 13
Total Time for solve A-star 410ms

maxNodes 10000
randomizeState 10
solve beam 7
Number of tile moves 4
Start state:
3 1 0
4 5 2
6 7 8

down

3 1 2
4 5 0
6 7 8

left
3 1 2
4 0 5
6 7 8

left
3 1 2
0 4 5
6 7 8

up
0 1 2
3 4 5
6 7 8

Total nodes generated for solve beam 46
Total Time for solve beam 397ms

- Solve A-star
  - ⇨ Add the start state as a Tile object to the priority queue. Mark its parent as null, the state as the start state, g as 0 (we are now at the start state and thus no cost to reach current state from the start state), h as the result from the heuristic function chosen, direction as -1(indicates no move to reach the current state from the start state)
  - ⇨ Every time we call solve A-star search, we generate all possible moves from the state whose f = g + h is the lowest. Since the heuristic function never overestimate the cost from the current state to the goal state, we always get closer and closer to our goal state.
  - ⇨ Then we add all of them to the priority queue.
  - ⇨ Iteratively do solve A-star search until the state which has the lowest f = g + h in the queue is found.
  - ⇨ We always get the optimal solution unless the number of the nodes under consideration exceeds maxNodes.

- Solve beam
  - ⇨ Add the start state as a Tile object to the priority queue. Mark the parent as null, the state as the start state, g as 0 (we are now at the start state and thus no cost to reach current state from the start state), h as the result from the heuristic function chosen, direction as -1(indicates no move to reach the current state from the start state)
  - ⇨ Every time we call solve beam, we generate all successors of the state in priority queue and add the at most k states with the lowest f = g + h. I choose h2 in A* search as the heuristic here. Since the heuristic function never overestimate the cost from the current state to the goal state, we always get closer and closer to our goal state. Since we put all successors together, actually we get the local minimal (if not trapped by the local min, we can get the global min) steps from the random start state to the goal state.
  - ⇨ It may fail if it is trapped in a local minimal or if the number of the nodes under consideration exceeds maxNodes.

3. Experiment

(a) Always randomizeState(100) and try 5000 cases each time.

maxNodes/ solved fraction	A*(h1) (%)	A*(h2) (%)	Beam Search(k=20) (%)
100	4.06	5.38	0.74
200	5.72	9.78	1.46
300	6.74	11.66	3.74
400	7.16	13.06	5.52
500	8.2	14.22	7.38
600	8.68	15.18	10.32
700	8.94	15.9	13.02
800	9	16.48	14.8
900	9.56	17.24	15.98
1000	10.1	17.88	16.78
1100	10.38	18.34	17.1
1200	10.58	18.82	17.16
1300	10.7	19.42	17.2
1400	10.86	19.9	17.32
1500	11.14	20.34	17.34

1600	11.2	20.58	17.36
1700	11.36	21.08	17.36
1800	11.72	21.48	17.36
1900	11.94	21.86	17.36
Increase percent	**194%**	**306%**	**2246%**

For A* search, the fraction of solvable puzzles from random initial states always increases as the maxNodes limit increases.
Based on the increase percentage, initially, beam search increases faster than A* search. However, according to the last four data of beam search, it seems that it keeps constant. In conclusion, initially, the solvable fraction increases as the limit increases. Then the solvable fraction for beam search keeps constant and may continue to grow when the limit is large enough.
For the solvable fraction, initially, beam search grows the fastest, then is A*(h2) and A*(h1) is slowest.

(b) Set max nodes to 100000

d/nodes generated	A*(h1)	A*(h2)
3	11	11
4	13	13
5	14	18
7	25	66
9	148	80
11	1214	386
12	1664	111
14	26835	6534
18	Exceed max Nodes	58804

From the table above, h2 is better. If the solution lengths are relatively small, the difference is slight. However, as the solution lengths increase, for any given solution lengths, the nodes generated by h_2 is largely smaller than those generated by h_1. For any node n, h2(n) ≥ h1(n). We thus say that h2 dominates h1. Domination translates directly into efficiency: A∗ using h2 will never expand more nodes than A∗ using h1 (except d is relatively small).

(c) Set max nodes to 1000000

# of random moves/solution length	A*(h1)	A*(h2)	Beam(20)
2	2	2	2
4	4	4	4
6	2	2	2
8	4	4	4
16	6	6	6
24	12	12	12
34	10	10	12
40	10	10	12

For a give number of random moves, A*(h1) and A*(h2) always get the same solution length. The solution length of the beam search is usually greater or at least the same as that of the A* star search. Maybe it is because beam search is sometimes trapped in the local min.
For all three searches, the solution length fluctuates slightly as the # of random moves increases.

(d) The cases that do not exceed max node limit before reaching the goal state.
    1) The solution length cannot be too long or it will exceed the limit on the way to the end
    2) The path cannot go to loop during the search or it will never get out of the loop and thus exceed the limit.

4. Discussion

(a) A*(h2) is better suitable for this problem since it always guarantees an optimal solution and h2 is better than h1. However, beam search may be trapped in the local min. A* search always finds the shorter answer and beam search gets the same or the longer answer (trapped in the local min). In terms of time and space, A*(h2) seems more superior. Based on experiment (a), A*(h2) always solve most puzzles under a given max limit for nodes. It indicates that A*(h2) on average cost least space. I also count the time that the three searches spend on the same puzzle and find that generally A*(h2) run fastest, then is beam search and A*(h1) run slowest.

(b) 2 difficulties.
    ⇨ It is hard to analyze the solution length. We can only control the number of random walk and get the corresponding solution length.

- ⇨ It is always easy for beam search to exceed max nodes and thus, I cannot track the result for long solution length. Also, we cannot set the K for beam search too large because of max node limit.