Minh Pham - CaseID: mqp4
CSDS 391
Prof. Mike Lewicki

PROGRAMMING ASSIGNMENT 1

1. Overview:

The goal of this project is to design an artificially intelligent agent that can solve the 8-puzzle problem using A-star search with 2 different heuristics and local beam search. The project was coded entirely in Java. It reads commands written in a specific manner to solve the puzzle with either randomized or pre-determined states.

2. Code design:

To store each state of the board with relevant information such as previous state, path cost, heuristic score, coordinates of the blank tile, a new data structure was designed and implemented, as demonstrated below:

```
class Node {
    int[][] board;
    Node parent;
    int cost, level;
    int row, col;

    Node(int[][] board, Node parent, int cost, int level, int row, int col) {
      this.board = new int[3][3];
      for (int i = 0; i < 3; i++)
        for (int j = 0; j < 3; j++)
          this.board[i][j] = board[i][j];
      this.parent = parent;
      this.cost = cost;
      this.level = level;
      this.row = row;
      this.col = col;
    }
  }
```

Each node keeps track of the board and its parent, the cost (heuristic score), level (path cost), and position of the blank tile to make it easier for keeping track. Note that the matrix must be duplicated since the 2-D array is a reference variable in Java.

In addition, 2 comparators are also designed for A-star search and beam search:

```
import java.util.*;

class NodeComparator implements Comparator<Node> {
  public int compare(Node n1, Node n2) {
    if (n1.cost + n1.level < n2.cost + n2.level)
      return -1;
    else if (n1.cost + n1.level > n2.cost + n2.level)
      return 1;
    else
      return 0;
  }
}
```

```
import java.util.*;

class NodeComparator2 implements Comparator<Node> {
  public int compare(Node n1, Node n2) {
    if (n1.cost < n2.cost)
      return -1;
    else if (n1.cost > n2.cost)
      return 1;
    else
      return 0;
  }
}
```

NodeComparator is designed for A-star search, where nodes are ranked in ascending order of f(n) = cost + level, with cost depending on whether the commands includes “h1”, the number of misplaced tiles, or “h2”, the Manhattan distance. On the other hand, NodeComparator2 is designed for local beam search, where nodes are ranked based on the evaluation function. This function is chosen to be f(n) = h1_cost + h2_cost, the reason for which is that both h1 and h2 heuristics are admissible, consistent, and are proven to work well with A-star search. Since f(n) is the total of 2 costs, the lower f(n) is the better, and so the best k states will be the k states with the lowest values of f(n).

The code implements all methods in the prompt along with several helper methods, which will be listed below:

a. **int[][] setState**(String state)
   This method takes a string as the input and outputs a 2-D array representing the desired input state. The blank tile is represented by the character ‘b’. For example, a string input of “724-5b6-183” is represented by a matrix with 7, 2, and 4 on the first row, 5, blank (denoted by ‘0’) and 6 on the second row, and 1, 8 and 3 on the last row.
   A choice of design I have made is changing the <space>, which signifies a new row in the prompt, to a hyphen. The reason for this is because Java doesn’t handle <space> well when it comes to reading commands from a text file. It automatically separates string into smaller strings when the scanner reads a <space>, hence the change.

b. **int[] findBlank**(int[][] board)
   This method inputs a matrix and and outputs an 2-element array representing the coordinates of the blank tile in the form of {row, col}.

c. **void printState**(int[][] board)
   This method prints the state of the puzzle.

d. **boolean isValid**(int x, int y, int moveX, int moveY)
   Here, x and y are the row and column positions of the blank tile, and moveX and moveY indicate the number of moves across the rows and columns. This method checks if the move is valid, i.e. checks whether the move causes an out-of-bound exception. Although we are moving the numbered tiles, the moves are thought of as moving the blank tile. Hence, there are only at most 4 possible moves at each state, and they are stored as shown:

```
static int[] dirX = {1,0,-1,0};
static int[] dirY = {0,1,0,-1};
```

   2 elements at index i of each array makes a pair that signifies the moving direction of the blank tile across the rows or columns. From left to right, they are down, right, up, left.

e. **int[][] move**(Node node, int moveX, int moveY)
This method executes the move. The new row and column positions are calculated, and the swap is made to reflect the move of the blank tile. Note that in this method, the input node is duplicated since the matrix is a reference variable in Java, and we want to move the tile without changing the original board.

f. **int[][] randomizedState**(int n)
This method chooses a random number between 0 and 3 to use as an index for the arrays dirX and dirY, then checks if the move in that direction is valid. If it is, the method makes the random move, and after k random moves, the board is returned.

g. **int calculateCost**(String heuristic, int[][] board)
Based on the input for heuristic, the method calculates the cost, i.e. the heuristic score, for the board. The method can only calculate using 2 heuristics, h1 and h2, as mentioned above. Although the method reads if (heuristic is h1)... else…, meaning it can take multiple inputs for heuristic, it is later handled in the readCommands method that only h1 and h2 can be taken, or else a message will be printed.

h. **boolean contains**(List<int[][]> list, int[][] board)
Since Java has no way to check if a matrix is in a list of matrices (even if the 2 matrices have the same values, false will still be returned since they have different address), this helper method is used to determined if a state of the board has been explored, i.e. included in the list.

i. **solveAstar**(String heuristic, int[][] board, int capacity)
This method solves the puzzle by implementing A-star search with the heuristic input. First, a priority queue is used to pick the state with the best f-score to explore. A node is created to store the initial state with calculated cost and position of the blank tile, and it is added to the queue. A list is also created to keep track of explored states and avoid infinitely going back and forth between states.

At every iteration, the node with the smallest f-score is picked. If it is the goal state, the number of moves and path to that state is printed, and the method returns. If it is not the goal state, then every possible successor from that state is generated and checked if that successor has been explored. If it is not, the successor will be added to the queue and the list. The list is checked every time a new state is added to see if the size has reached the capacity. If it has, a message is printed and the method returns. The loop continues until the goal state is found or the capacity is exceeded.

j. **solveBeam**(int k, int[][] board, int capacity)
This method solves the puzzle by implementing local beam search with a beam width of k. The setting up is similar to solveAstar, with the queue, the list and the creation of the initial node.

The main difference between the 2 search methods is that another loop wraps our main while loop from solveAstar. At the beginning of each iteration of the outer loop, a new priority queue is created to store all neighbors of all nodes from the main queue. The generation of successors is similar to solveAstar. However, after all nodes from the main queue have been removed and all successors have been added to the neighbor queue, k best states from the neighbor queue are chosen to be added to the main queue. The neighbor queue is re-created at every iteration, and the procedure keeps going until the goal state is found or the capacity is reached.

k. **printPath**(Node node)
This recursive method prints the path to the goal state.

l. **execute**(String name)
This method inputs the name of the text file as a string, scans the command word-by-word, and runs the methods accordingly. If the command is invalid, a message is printed. Examples of valid and invalid commands will be shown in the next section.

3. **Code correctness:**
Here is an example of a valid command:

```
randomizeState 20 printState solveAstar h2 200
```

This command reads: Randomize a state 20 moves from the goal state, print that state, then solve the state using A-star search and h2 heuristic, with a capacity of 200 nodes. Here is the result, printed to the console:

```
Current state being randomized...
Current state:
4 7 2
1 0 5
3 6 8

Number of moves: 8
4 7 2
1 0 5
3 6 8

4 0 2
1 7 5
3 6 8

0 4 2
1 7 5
3 6 8

1 4 2
0 7 5
3 6 8

1 4 2
3 7 5
0 6 8

1 4 2
3 7 5
6 0 8

1 4 2
3 0 5
6 7 8

1 0 2
3 4 5
6 7 8

0 1 2
3 4 5
6 7 8
```

We can see that the state is randomized 20 moves from the goal state, but it only takes 8 moves to solve the puzzle. This is because the moves were randomized, and so some pairs of moves could be (left, right) or (up, down), which brings us to the state explored before.

Instead of randomizing the state, we can also set the state as follow:

```
setState 724-5b6-831 printState solveAstar h1 200
```

```
Input state: 724-5b6-831
Current state:
7 2 4
5 0 6
8 3 1

The puzzle either is unsolvable or takes too long to be solved.
```

We can see that using h1 and a capacity of 200 nodes, the method can't solve the puzzle. We can either increase the capacity, change the heuristic or change the method.

Increasing the capacity, we have:

```
setState 724-5b6-831 printState solveAstar h1 100000
```

```
Input state: 724-5b6-831
Current state:
7 2 4
5 0 6
8 3 1

Number of moves: 26
7 2 4
5 0 6
8 3 1

7 2 4
0 5 6
8 3 1

0 2 4
7 5 6
8 3 1

2 0 4
7 5 6
8 3 1

2 5 4
7 0 6
8 3 1

2 5 4
7 6 0
8 3 1

2 5 4
7 6 1
8 3 0

2 5 4
7 6 1
8 0 3

2 5 4
7 6 1
0 8 3

2 5 4
0 6 1
7 8 3

2 5 4
6 0 1
7 8 3

2 5 4
6 1 0
7 8 3

2 5 4
6 1 3
7 8 0

2 5 4
6 1 3
7 0 8

2 5 4
6 1 3
0 7 8

2 5 4
0 1 3
6 7 8

2 5 4
1 0 3
6 7 8

2 5 4
1 3 0
6 7 8

2 5 0
1 3 4
6 7 8

2 0 5
1 3 4
6 7 8

0 2 5
1 3 4
6 7 8

1 2 5
0 3 4
6 7 8

1 2 5
3 0 4
6 7 8

1 2 5
3 4 0
6 7 8

1 2 0
3 4 5
6 7 8

1 0 2
3 4 5
6 7 8

0 1 2
3 4 5
6 7 8
```

Changing the heuristic, we have:

```
setState 724-5b6-831 printState solveAstar h2 100000
```

```
Input state: 724-5b6-831
Current state:
7 2 4
5 0 6
8 3 1

Number of moves: 26
7 2 4
5 0 6
8 3 1

7 2 4
0 5 6
8 3 1

0 2 4
7 5 6
8 3 1
```

```
2 0 4
7 5 6
8 3 1

2 5 4
7 0 6
8 3 1

2 5 4
7 6 0
8 3 1

2 5 4
7 6 1
8 3 0

2 5 4
7 6 1
8 0 3
```

```
2 5 4
7 6 1
0 8 3

2 5 4
0 6 1
7 8 3

2 5 4
6 0 1
7 8 3

2 5 4
6 1 0
7 8 3

2 5 4
6 1 3
7 8 0
```

```
2 5 4
6 1 3
7 0 8

2 5 4
6 1 3
0 7 8

2 5 4
0 1 3
6 7 8

2 5 4
1 0 3
6 7 8

2 5 4
1 3 0
6 7 8
```

```
2 5 0
1 3 4
6 7 8

2 0 5
1 3 4
6 7 8

0 2 5
1 3 4
6 7 8

1 2 5
0 3 4
6 7 8

1 2 5
3 0 4
6 7 8
```

```
1 2 5
3 4 0
6 7 8

1 2 0
3 4 5
6 7 8

1 0 2
3 4 5
6 7 8

0 1 2
3 4 5
6 7 8
```

Changing the method to solveBeam, we have:

```
setState 724-5b6-831 printState solveBeam 3 10000
```

```
Input state: 724-5b6-831
Current state:
7 2 4
5 0 6
8 3 1

Solved! Number of moves: 300
7 2 4
5 0 6
8 3 1

7 2 4
5 3 6
8 0 1

7 2 4
5 3 6
8 1 0
```

(it takes 300 moves but it still reaches the goal state)

If we adjust the bandwidth to 4, we have:

```
setState 724-5b6-831 printState solveBeam 4 10000
```

```
Input state: 724-5b6-831
Current state:
7 2 4
5 0 6
8 3 1

Solved! Number of moves: 162
7 2 4
5 0 6
8 3 1

7 2 4
5 3 6
8 0 1

7 2 4
5 3 6
0 8 1
```

(it takes 162 moves but it still reaches the goal state)

Testing the move command:

```
setState 724-5b6-831 move up move right printState
```

```
Input state: 724-5b6-831
Move performed: up
Performed!

Move performed: right
Performed!

Current state:
7 4 0
5 2 6
8 3 1
```

Note that if we use both setState and randomizeState together and solve the problem, then whatever that comes after will be set as the initial state.

```
setState 724-5b6-831 printState randomizeState 20 printState solveAstar h2 5000
```

```
Input state: 724-5b6-831
Current state:
7 2 4
5 0 6
8 3 1

Current state being randomized...
Current state:
3 1 2
4 8 7
6 5 0

Number of moves: 8
3 1 2
4 8 7
6 5 0

3 1 2
4 8 0
6 5 7

3 1 2
4 0 8
6 5 7

3 1 2
4 5 8
6 0 7

3 1 2
4 5 8
6 7 0

3 1 2
4 5 0
6 7 8

3 1 2
0 4 5
6 7 8

3 1 2
4 0 5
6 7 8

0 1 2
3 4 5
6 7 8
```

If we try an invalid command, we get an error message:

```
randomizeState 20 printState solveAstar 5000
```

```
Current state being randomized...
Current state:
1 2 5
3 0 8
6 4 7

Sorry, command is invalid.
```

4. **Experiment:**
    a. Here is a table of generated nodes for different search methods with varying number of randomized moves from the goal state

	A-star with h1	A-star with h2	Local beam, k = 10	Local beam, k = 100
2	6	6	14	14
4	12	12	48	40
6	24	20	162	69
8	45	28	431	115
10	103	47	678	156
12	240	82	1002	245
14	590	152	1257	431
16	1309	295	1474	631
18	2854	453	1772	701
20	5196	692	2103	854

Hence, if maxNodes is lower than the number indicated in the table, then the puzzle can't be solved for that method. For example, if we use A-star search with h2 to solve a puzzle that is 14 moves from the goal state, then if the maxNodes is lower than 152, the puzzle won't be solved.

Based on the table, for A-star search with h1, if the number of random moves made from the goal state increases by 2, then the number of puzzles that can't be solved due to maxNodes constraints doubles. For A-star with h2, it increases by 40%, and for local beam, the result varies.

However, note that even if the number of moves made from the goal state is 14, the state might be less than 14 moves from the goal state since the moves are randomized.

b. For A-star search, I personally think h2 is a better heuristic since it takes into consideration the distance between the current position of a tile and where it should be, whereas h1 only considers the number of different tiles between the current state and the goal state.

c. In terms of solution length, usually, A-star search finds a much shorter path compared to local beam search since A-star prioritizes optimality. Within A-star search, h1 takes longer than h2 since it is not as good of a heuristic to evaluate a good state, but both heuristics are still able to find a short path. Local beam

search is way faster since it doesn't look through all the nodes, but because of that, the path to the goal state is longer since some nodes that are better than the later ones are already eliminated at the beginning. The bigger the beam width, the shorter the solution, but the longer the runtime.

d. I would say 60% of the puzzles are solvable using A-star with h1 since it is not as good at looking for good nodes, and so it has a higher probability of exceeding the capacity. For A-star with h2, 80% of the puzzles are solvable, and for local beam search, the rate is almost 100% (I actually never came across a puzzle that can't be solved using local beam).

5. Discussion:

I personally think A-star with h2 is the best-suited method to solve the 8-puzzle problem, since it doesn't take too long to find a relatively short solution. However, it depends on the priority of the solution. If the solution prioritizes time, then local beam search is a better choice. In terms of space, if "space" here means the memory space for the solution, then A-star is better since it finds a shorter path, but if "space" refers to the total memory cost, then local beam search is better since it doesn't evaluate all members.