CSE 150 - Project 1

Due 18 October 2012

For this project you will write a program to solve a more general form of the 8-puzzle. The puzzle can be described as an m x n array of digits 1 through (m*n - 1) with one empty square, marked with a 0. The goal is to have the digits arranged in order (left to right, top to bottom), with the empty square located in the **upper left corner**.

An action consists of moving a tile that is adjacent to the blank space into the blank space. However, we will think of this as moving the blank space and swapping it with one of its neighbors. So, at any given time, there are at most 4 valid actions: moving the blank space North, South, East, or West.

Please see pages 70-71 in the Russell and Norvig textbook for an alternate description of the 8 puzzle.

1 Logistics

You can work alone or in groups of at most 3. You may use Java or Python for this assignment. We will not accept late submissions.

2 Requirements

You must write a program called PuzzleSolver that is run as follows:

```
java PuzzleSolver [arguments (described below)]
```

or

```
python PuzzleSolver.py  [arguments (described below)]
```

The program will take as the first argument a text file representing a puzzle problem to solve. Columns are separated by commas and rows are separated by newlines, as shown in the example below, which is the solution state for a 3 x 3 puzzle.

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

We have posted several simple puzzle files in Piazza. Note that any random permutation of numbers is not guaranteed to be solvable, so be aware of this when creating your own test puzzles. To guarantee they can be solved, make sure your test puzzles are the result of a series of valid moves performed on the goal state.

Your program must implement the following search algorithms:

- Breadth-first search
- Depth-limited depth-first search
- Iterative deepening search
- A* search
- Greedy best-first search

Your program must follow the command line interface and output formatting described below. Also, you must submit a README file. Please download the skeleton README file from Piazza and follow the directions inside.

2.1 Breadth-first search

Your breadth-first search should be run from the command line as follows:

```
java PuzzleSolver puzzleFile.txt BFS
```

2.2 Depth-first search

Your depth-first search should be run from the command line as follows:

```
java PuzzleSolver puzzleFile.txt DFS 5
```

where the third parameter is the search tree depth limit.

2.3 Iterative deepening search

Your iterative deepening search should be run from the command line as follows:

```
java PuzzleSolver puzzleFile.txt ID 5
```

where the third parameter is the depth limit of the deepest search tree created by ID.

2.4 A* search

Your A* search should be run from the command line as follows:

```
java PuzzleSolver puzzleFile.txt A_Star Manhattan
```

where the third parameter is the heuristic to use. You must implement two heuristics: manhattan heuristic, and another of your choosing. These will be invoked with the command line arguments "Manhattan", and "Other", respectively.

The manhattan heuristic is computed by looking at each tile (not including the blank tile) and determining the manhattan distance to move that tile to its correct location, then summing each of these distances. For example, the following puzzle has a manhattan heuristic cost of 6:

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

The "Other" heuristic can be as simple as counting the number of misplaced tiles, but we encourage you to come up with other heuristics. Feel free to experiment with inadmissible heuristics (which would technically make the algorithm the A algorithm, instead of A*).

2.5 Greedy best-first search

Your greedy search should be run from the command line as follows:

```
java PuzzleSolver puzzleFile.txt Greedy Manhattan
```

where the third parameter represents the heuristic to use, as in A*.

2.6 Output format

Print the length of the solution path, number of nodes expanded, and solution path to std out. eg.

```
$java PuzzleSolver puzzleFile.txt Greedy Manhattan
solution length: 6
nodes expanded: 7
NESWNW
```

"nodes expanded" represents the number of nodes actually removed from the queue and expanded. Do not count nodes that are simply inserted in the queue.

The solution path is shown in the bottom line of the above output. It represents the solution returned by the chosen search algorithm. It is a string of the characters 'N', 'S', 'E', and 'W', which represent the actions of moving the BLANK tile up, down, right, and left, respectively. Performing these moves in order (left to right), should transform the original input puzzle into the goal state.

Please note that we are testing your programs using automated testing, so be sure to match the command line interface and output format described here.

3 Turn-in procedure

Please turn in all source code files, your README, test puzzle used in your README analysis, and optionally any other interesting test puzzles you created. We will post the turn-in procedure on Piazza soon.

4 Evaluation

Your project will be graded based on whether you implemented all the requirements, how well your program performs on test puzzles, and your answers to questions in the README.

Note that we will be using an automated tester, so please be sure to follow the command line interface and output format exactly.

Object-oriented code that is easy to understand may help us give you more partial credit if you do not pass some of the tests. Comments help also! If you tested your program thoroughly (as evidenced by your explanation in the README) we may also give more partial credit.

We may give extra credit if we feel you came up with a clever implementation or heuristic function, or if your testing procedure was especially thorough. Document any of these details in the README.