Solving Tile Puzzles Using Search

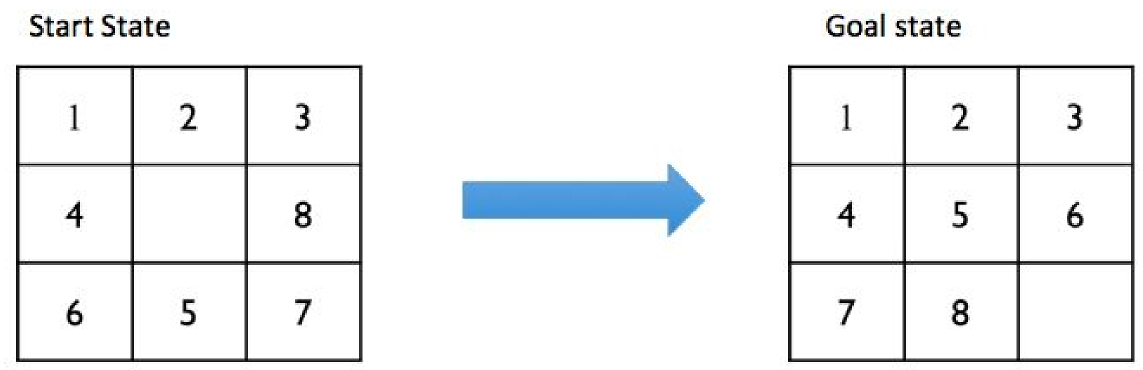
Assignment 1

Due: 11:59 pm Oct 9th, 2016

Goals: The main goal of this assignment is to 1) go through the paces of the A* algorithm on a toy problem, and 2) learn the details of a memory bounded variant of the algorithm, which we did not see in class.

Please do not cheat. The purpose of this assignment is to learn a bit more about search algorithms. You are allowed to discuss with friends about the algorithms but the implementations, and write-ups must be your own.

In this programming assignment you will build a program that can solve 8 and 15 tile puzzles. Just as a reminder, the tile problem is where you are given a matrix of numbers in a *n x n* board with one blank tile. Given a scrambled initial order, your goal is to move the tiles over the blank space to achieve the final configuration, all numbers in the increasing order with the blank tile in space 9.



Implementation

Your program will solve the 8-puzzle (as above) and the 15-puzzle. You will be given a puzzle generator that you can use for generating test puzzles. You will implement 1) A* algorithm and 2) another memory bounded informed search algorithm of your choice with two heuristics. RBFS and IDA* are two memory bounded algorithms that should work. You are welcome to try any other algorithms as well. If you use either one of these algorithms the TA will be able to provide high level help or clarifications if necessary.

Implementation Requirements

- You must implement the algorithms from scratch in **Python 3x**.
- You cannot use external libraries.
- It is ok to use native libraries for data structures (e.g., priority queues) but your program should run on its own.
- You cannot use existing implementations of any kind for development or reference. You can use pseudo-code or descriptions of the algorithms for reference but not actual code.
- You can discuss with your friends about which algorithms to use and to understand the algorithms but not share or discuss code.
- All code will be tested by running through plagiarism detection software.
- You should include cite any web resources or friends you used/discussed with for pseudo-code or the algorithm itself. **Failure to do so will result in an F.**
- Your algorithm should be documented at a *method level* briefly so they can be read and understood by the TAs.

What to turn in?

You should turn in your source code and the output of your program on the two test input files shipped as part of the assignment.

Your source code can be structured however you like but we will run a single file named puzzleSolver.py to evaluate your implementation.

1) **puzzleSolver.py** – This program should be able to take as input an 8 or a 15 puzzle and output the set of moves required to solve the problem.

The program should run from the command line as follows:

```
python puzzleSolver.py <#Algorithm> <N> <INPUT_FILE_PATH> <OUTPUT_FILE_PATH>
where,
    #Algorithm: 1 = A* and 2 = Memory bounded variant.
    N: 3 = 8-puzzle 4 = 15-puzzle format.
    INPUT_FILE_PATH = The path to the input file.
    OUTPUT_FILE_PATH = The path to the output file.
```

e.g. The command `python puzzleSolver.py 1 3 test-input.txt test-output.txt` should run the A* algorithm on the 8 puzzle provided in test-input.txt and write the moves out in test-output.txt.

The input and output file formats should be as follows. The format is a strict requirement as we will run automated tests on the program.

Input	Output

`1,3,`
`4,2,5`
`7,8,6`
	`L,D,R,D`

L → Move blank tile left, R → Move blank tile right
D → Move blank tile down, U → Move blank tile up

Note: If any of your moves is illegal (e.g., moves the blank space out-of-bounds) then the output is considered a failure.

Sample puzzles:

You can use the `puzzleGenerator.py` provided with the assignment on blackboard.

There will be a randomly generated configuration of a puzzle with size N (N=3 for 8-puzzle and N=4 for 15-puzzle) written to OUTPUT_FILE_PATH.

```
python puzzleGenerator.py N D OUTPUT_FILE_PATH
```

Starting from the solved configuration of a puzzle with size N, we do D random moves and write the output to OUTPUT_FILE_PATH.

Note: This version, by design, always generates solvable puzzles. If you are curious to test what happens on a completely random puzzle, which may or may not be solvable, you can use:

```
python puzzleGenerator.py N OUTPUT_FILE_PATH
```

We will evaluate your program only on solvable puzzles.

2) Output files for the two input files provided as part of the blackboard assignment.

Report

The report should include the following sections.

1. Heuristics – Describe the two heuristics you used for A*. Show why they are admissible (or consistent).
2. Memory issue with A* -- Describe the memory issue you ran into when running A*. Why does this happen? How much memory do you need to solve the 15-puzzle?
3. Memory-bounded algorithm – Describe your memory bounded search algorithm. Is this algorithm complete? Is it optimal? Give a brief complexity analysis. This doesn't have to be rigorous but clear enough and correct.
4. A table describing the performance of your A* implementation on a randomly drawn set of 20 solvable puzzles. You should tabulate the number of states explored, time (in milliseconds) to solve the problem and the depth at which the solution was found for both heuristics.

Grading

1. A* implementation (50 points) -- Points will be deducted for failures on solvable puzzles. as well as failing to solve within a reasonable amount of time i.e., 30 seconds or less. If your output includes a sequence that leads to an illegal state then it is considered a failure. If it doesn’t solve any of the tested puzzles then you will get a zero.

2. Memory bounded search (30 points) -- Same as above.

3. Report (20 points) -- Five points for each section.