Assignment 1: Search

Niranjan Balsubramanian, Tianyi Zhao, and Tao Sun

CSE 352 Spring 2021

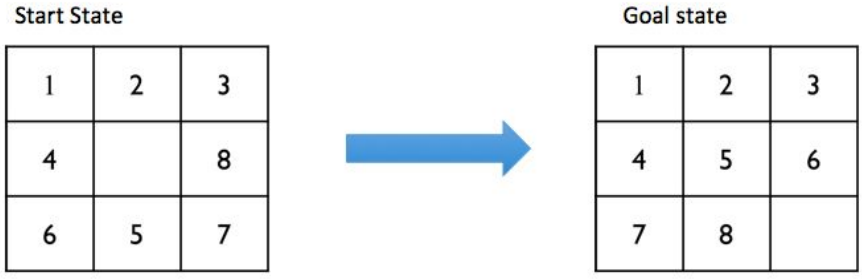
1 Due Date and Collaboration

- The assignment is due on **Mar 5 at 11:59 pm**. You have a total of four extra days for late submission across all the assignments. You can use all four days for a single assignment, one day for each assignment – whatever works best for you. Submissions between fourth and fifth day will be docked 20%. Submissions between fifth and sixth day will be docked 40%. Submissions after sixth day won't be evaluated.
- You can collaborate to discuss ideas to understand the concepts and math.
- You should NOT collaborate at the code or pseudo-code level. This includes all implementation activities: design, coding, and debugging.
- You should NOT not use any code that you did not write to complete the assignment.
- The homework will be cross-checked. **Do not cheat at all! It's worth doing the homework partially instead of cheating and copying your code and get 0 for the whole homework.**

2 Goals

The main goal of this assignment is to 1) go through the steps 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.

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 target final configuration, all numbers in the increasing order with the blank tile in the last available space.



3 Problem Formulation

First define a problem, in terms of state, actions, transition function, and goal test. Define a python class named `TileProblem` that with clearly labeled components (e.g. variables, and methods) that point to these elements.

Given an input file that contains the state, your program should first construct an instance of this problem as a `TileProblem` object.

4 Implementation

4.1 Input and Output

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 puzzle files. The files will be of the following format:

Exemplary 8-puzzle file

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

Exemplary 15-puzzle file

```
1,2,3,4
5,6,7,8
10,11,,12
9,13,14,15
```

Pay attention to the location of blank tile in the exemplary files. To generate your own test puzzle files, run

python puzzleGenerator.py <N> <D> <OUTPUT_FILE_PATH>

where N is the size of puzzle (N=3 for 8-puzzle and N=4 for 15-puzzle), D is the random moves to generate this test puzzle and OUTPUT_FILE_PATH is the output filename.

Your solver should be a python script named **puzzleSolver.py** and run from the command line as follows:

python puzzleSolver.py <A> <N> <H> <INPUT_FILE_PATH> <OUTPUT_FILE_PATH>

where A is the algorithm (A=1 for A* and A=2 for RBFS), N is the size of puzzle (N=3 for 8-puzzle and N=4 for 15-puzzle), H is for heuristics (H=1 for h_1 and H=2 for h_2), INPUT_FILE_PATH and OUTPUT_FILE_PATH are the input and output filenames.

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.

4.2 Admissible Heuristics

Define a `Heuristics` class that defines the methods that compute the heuristics for calculating distance to the goal.

You will implement two different admissible (or consistent) heuristic functions (h_1 and h_2). One should dominate the other: a heuristic h_1 dominates h_2 if $h_1(n) > h_2(n)$ for all n.

4.3 A* Search

You will implement the graph search version of the A* algorithm. That is the version where you track which nodes have been explored.

4.4 Memory-bounded Informed Search

The memory-bounded algorithms make choices on how much memory to use and what to keep in memory for graph-search. You should implement **Recursive Best First Search (RBFS)** as the memory bounded algorithm.

You could find some online resources for RBFS algorithm.

- https://pages.mtu.edu/~nilufer/classes/cs5811/2012-fall/lecture-slides/cs5811-ch03-search-b-informed-v2.pdf
- http://mas.cs.umass.edu/classes/cs683/lectures-2010/Lec5_Search4-F2010-4up.pdf

4.5 Requirements

1. You must implement the algorithms from scratch in **Python 3.7+**.
2. You CANNOT use external libraries that provide search related capabilities directly.
3. You CAN use native python libraries that provide support for data structures (e.g., priority queues) but your program should run on its own.
4. You CANNOT use existing implementations of any kind for development or reference.
5. You CAN use pseudo-code or descriptions of the algorithms for reference but not actual code.
6. You CAN discuss with your friends about which algorithms to use and to understand the algorithms but not share or discuss code.
7. All code will be tested by running through plagiarism detection software.
8. 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.
9. Your algorithm should be documented at a method level briefly so they can be read and understood by the TAs.

5 What should you turn in?

- README - Code Implementation details. Please include your Python version or any other information that is necessary for TAs to run your code.
- Code - You should turn in your source code. Your source code can be structured however you like but we will run a single file named **puzzleSolver.py** to evaluate your implementation. Your source code should includes at least two files: **puzzleSolver.py** and **TileProblem.py**

    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.

2. **TileProblem.py** – This program should be able to define the search problem with clearly labeled components that point to the state, actions, transition function and goal test.

Before submitting your code, please use the **formatCheck.py** to check the format of your output file. You can run the python script from the command line as follows:

python formatCheck.py <OUTPUT_FILE_PATH>.

where OUTPUT_FILE_PATH is the output filename from puzzleSolver.py.

- Report (PDF) – This should be a 3-4 page report, which includes five sections:

1. Problem Formulation - Describe how you define the problem in the `TileProblem` class.
2. Heuristics – Describe the two heuristics you used for A*. Show why they are consistent and why h_1 dominates h_2.
3. 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?
4. Memory-bounded algorithm – Describe your memory bounded search algorithm. How does this address the memory issue with A* graph search. Is this algorithm complete? Is it optimal? Give a brief complexity analysis. This analysis doesn't have to be rigorous but clear enough and correct.
5. A table describing the performance of your A* and memory-bounded implementations on the **five test input files** shipped as part of the assignment. You should include the output of your solver on the five test input files. You should tabulate the number of states explored, time (in milliseconds, you can use datetime to measure the time) to solve the problem, and the depth at which the solution was found for both heuristics.

6 Grading

Here is a coarse grained grading rubric.

- A* implementation (50 points)

1. Points will be deducted for failures on solvable puzzles.
2. Points will also be deducted if you implementation doesn't solve the problems within a reasonable amount of time, reasonable being 30 seconds or less.

3. If your output includes a sequence that leads to an illegal state (i.e. space is outside the tile) then it is considered a failure.
4. If it doesn't solve any of the tested puzzles then you will get a zero.

- Memory bounded search (30 points) – Same as above.
- Report (20 points) – Four points for each section.