CSCI1410 Fall 2018
Assignment 1: Search

Code Due Monday, September 17
Writeup Due Thursday, September 20

1 Introduction

In this assignment, you will implement several search algorithms in their general forms. Then, you will use those search algorithms to solve tile game puzzles.

The tile game is similar to the one you encountered in class. There is still a 3-by-3 grid, and each cell in the grid is still occupied by a number. Also, the win condition is the same; arrange the numbers in order from lowest in the upper left, to highest in the lower right. However, the grid now has 9 numbers on it, instead of 8 numbers and an empty space. Additionally, movement is less restricted. In this game, you can swap any pair of adjacent numbers. The win condition and your goal during search is this board configuration:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

2 Your Task

2.1 Coding

In `search.py`, you will implement the breadth-first (`bfs`), depth-first (`dfs`), iterative deepening (`ids`), bi-directional (`bds`), and A* (`astar`) search algorithms. Each search function takes in a `SearchProblem`, called `problem`, as an input, and outputs the whole *path* of the solution. The path should be represented as a list of states, where the first element is the start state and the last element is a goal state. As a rough benchmark, each search function should complete in less than 10 seconds for most TileGame problems. In addition, you will fill in `tilegame_heuristic` with a heuristic function for tile games. You may find classes in the queue module useful for implementing the search algorithms. See section 2.1.2 for more information. **Only use allowed python libraries when implementing these searches**.

2.1.1 Additional Specifications for Bi-Directional Search

- Find a path from the start state to the given goal state (if there are other goal states, ignore them).
- Run a breadth-first search from both the start state and the goal state, alternating between expanding a node from the front and expanding a node from the back.

2.1.2 Additional information about queue module

The queue module contains `Queue`, `LifoQueue`, and `PriorityQueue`. All three implement the `put`, `get`, and `empty` functions. To add an item to a `PriorityQueue` with a given priority, add a tuple in the form of *(priority, item)*. By default, `PriorityQueue` retrieves the tuple with the *lowest* priority value. Consult https://docs.python.org/3/library/queue.html for more information about the module.

2.2 Writeup

In addition to your code, you should also create a typed document that addresses the following prompts:

1. Informally describe your heuristic
2. Prove that your heuristic is admissible*
3. Explain why you chose this heuristic over other possible heuristics

* Note that if you cannot think of an admissible heuristic, you should, of course, not try to prove that your heuristic is admissible. Instead, prove that your heuristic is not admissible or, even better, that it is impossible to produce an admissible heuristic.

You must submit your document via Gradescope (see the Gradescope guide on the course website for more details). While not required, we highly recommend using Latex to typeset your work. This writeup is due at the same time as the final resubmission (Thursday at 11:59pm).

3 The Code Files

3.1 Files to Modify

- `search.py` - This is where you will implement your search algorithms.

3.2 Necessary Source Code

- `searchproblem.py` - This contains an abstract class, `SearchProblem`, for search problems. The `SearchProblem` class has three abstract methods that are shared among all search problems. Look at the function headers and their docstrings before you begin.

- `tilegameproblem.py` - The `TileGame` class contained in this file extends the `SearchProblem` class. It contains implementations of the abstract methods from `SearchProblem`, internal helper functions, and utility functions that you may use in the code for your tile game heuristic and your testing. **A state of the game is represented as a tuple of tuples, where each interior tuple is a row of the tile game.** You will notice that the dimension of the Tile Game is adjustable. This is to ease your testing: you will find it easier to test your code on 2-by-2 games than on 3-by-3 games. Your heuristic is only required to work on 3-by-3 games.

3.3 Testing Source Code

- `dgraph.py` - This is an implementation of a Directed Graph as a SearchProblem. You can use DGraph to create small test cases for your searches, so if you would not like to use it for your testing, you can safely ignore it. The implementation uses a matrix representation of a directed graph; the states are each represented by a unique index in $\{0, 1, ..., S\}$, where S is the number of states, and the cost of moving directly from state i to state j is the entry of a matrix at row i, column j. If it is impossible to move directly from i to j (i.e., j is not a successor of i), then entry of the matrix at row i, column j should be `None` instead..

- `unit_tests.py` - This contains a testing suite with some trivial test case to help ensure that the input/output of each search function works properly. To run the test functions, execute `python unit_tests.py` inside the virtual environment. We encourage you to add more unit tests here to check your code's functionality.

4 Grading

We will give you your score based on the rubric in `rubric.txt`. Here are some details about the rubric:

- We will check each of your search algorithms to ensure that states are expanded in a proper order. The autograder considers a state to be expanded whenever `get_successors` is called on it. For this reason, ensure that `get_successors` is not called unnecesarily, and is only called once for each state expansion. For most search problems, there will be many correct orders of expanding the states for each search algorithm. Our

grading scripts will give you full points if you expand each search problem in any of the proper orders.

- For your Tile Game heuristic, you are graded based on "number of expanded nodes when used with A*". Your score for this will be determined by the following formula:

$$10 \cdot \frac{n_{ours}}{n_{yours}} \tag{1}$$

where n_{yours} and n_{ours} are the number of nodes expanded by our heuristic and your heuristic, respectively, on a pre-selected suite of tile games. You can score your heuristic on your own on a department machine by using the following command inside the virtual environment:

```
cs1410_test_heuristic /path/to/your/search.py
```

5 Virtual Environment

Your code will be tested inside a virtual environment that you can activate using `/course/cs1410/venv/bin/activate`. Read more about it in the course grading policy document available on the course website.

6 Install and Handin Instructions

To install, run `cs1410_install Search` from within your cs1410 directory.

To handin, run `cs1410_handin Search` from `~/course/cs1410/Search/`, which should contain your `search.py` file.

In addition, please submit an online collaboration policy (link available on the assignments page of the course website) and the written portion of the assignment to Gradescope (instructions available on Piazza). Since we cannot grade your work until you submit a signed collaboration policy, you should submit your signed collaboration policy by September 17th at 11:59pm, even if you plan to submit your writeup later.

In accordance with the course grading policy, your written homework should not have any identifiable information on it, including banner ID, name, or cslogin.

Finally, please be sure to read the course grading policy before handing in.