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:

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

Here is an example,

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

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

\[
\begin{bmatrix}
1 & 2 & 6 \\
7 & 8 & 3 \\
5 & 4 & 9 \\
\end{bmatrix}
\]
For testing purposes, you may choose to use the transition array (this may be useful when testing DFS solutions that may otherwise take an exorbitant amount of time to run with the tilegame problem).

The transition array is an array of elements that is the permutation from 0 to n - 1, where n is the length of the array. Similar to the tile game, you need to rearrange the elements to reach the goal state in this form: (0, 1, 2, 3, ..., n - 1). There are 2 ways you can rearrange the array.

1. Swap two elements in the array.

2. Reverse the array.

Here is an example,

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

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

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

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

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. Your heuristic function should not be a search function.

You may find classes in the `queue` module useful for implementing the search algorithms. See section 2.1.2 for more information.

To get started, take a look at textbook chapter 3.3, 3.4, and 3.5. Specifically, you can find pseudocode of BFS, DFS, IDS, and BDS in 3.3.

**Note:** Your algorithm should be problem-agnostic i.e they must work on any search problem, not just on TileGame, TArray, or DGraph. You should not call functions that specifically made for TileGame, TArray, or DGraph. Try to think in an abstract manner.

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.

BDS should return the shortest path, make sure to check if your algorithm always does so. Also, be careful to return the correct path from the start state to the goal state. That is, you need to return the path from the start state to the intersection plus the intersection to the goal state.

For full credit, please make sure to expand nodes alternatively from the start state and the goal state. The autograder will evaluate your score based on what order you expand your states.

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 \((\text{priority}, \text{item})\). By default, `PriorityQueue` retrieves the tuple with the lowest priority value. Con-
2.1.3 Additional information about DFS

With the correct implementation, you can still have a slow running time for complex cases of DFS. Try to first test the algorithm for some simple cases and then move to the complicated ones for debugging purposes. If you experience memory overflows, consider how you can further optimize your implementation to reduce its memory costs.

2.1.4 Additional Notes

Only use allowed python libraries when implementing these searches.

Pseudocode gives you a general idea of how to get started, but will not necessarily yield the most efficient solution. It is better to understand the algorithms first, and test your implementations thoroughly for edge cases.

Avoid copying or reusing your code or algorithm. Try to refactor your code as much as you can. Try to think of some data structures that may be of help.

2.2 Writeup

In addition to your code, you should also create a typed document that answers the following questions.

For grading purposes, please answer each question separately.

1. Informally describe your heuristic.

2. Prove that your heuristic is admissible.

3. Explain why you chose this heuristic over other possible heuristics. You must describe the other heuristics that you considered.

2.2.1 Additional Information

Do not use any heuristic that is identical or similar to functions listed in handout.

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.

- tarray.py - This is an implementation of a transition array as a SearchProblem. You can use tarray 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 simple array as the basic structure. All the transition rules live in get_successors(self, state). Feel free to add your own.
• 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 unnecessarily, 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*” and scored out of 10 points. Your score for this will be determined by the following formula:

\[
10 \cdot \frac{n_{ours}}{n_{yours}}
\]  

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 `source /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.

Without further notice, to handin, always run `cs1410_handin ASSIGNMENT_NAME` from `~/course/cs1410/ASSIGNMENT_NAME/`.

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 16th 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.