CSCI1410 Fall 2021
Assignment 1: Search

Submission 1 Due Monday, September 20 at 11:59pm ET

Submission 2 Due Wednesday, September 22 at 11:59pm ET

Code Due Friday, September 24 at 11:59pm ET

Writeup Due Saturday, September 25 at 11:59pm ET

Late Day Deadline Monday, September 27 at 11:59pm ET

1 Goals

In this assignment, you will implement depth-first, breadth-first, iterative-deepening, and A* search, and you will use those search algorithms to solve instances of a Tile Game.

2 Introduction

The Sliding Tiles puzzle is a famous Tile Game. It is played on an n-by-n grid (n is usually 3 or 4, or perhaps 5), with numbers occupying all but 1 of the n^2 cells, and an empty space occupying the last. Starting from a state with the numbers arranged at random, the goal is to rearrange all the tiles into ascending order, with the space in the bottom right-hand corner (as shown). The name derives from the fact that the tiles can slide over the empty space.

The Tile Game for this assignment is a variant on Sliding Tiles, also played on a n-by-n grid, with numbers occupying the cells. There is no space, however; all n^2 tiles are occupied. The game is played by swapping any two adjacent tiles, where adjacency is defined grid-wise (diagonal tiles are not adjacent) and does not wrap around. Like in the original Sliding Tiles puzzle, the goal is to arrange the numbers in ascending order.

The goal state on a 3-by-3 board is:

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

Here is one way to arrive at this goal, from a random start state:

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

The search space (i.e., the number of states) in a Tile Game is exponential, because there are $m! \in \Omega(m^m)$ ways to order m consecutive integers. As we cannot possibly enumerate this space for all but the smallest choices of m, you will solve instances of the Tile Game using heuristic search. To do so, you will implement both blind and informed search algorithms. The informed algorithms rely on heuristics, which you will develop by encoding domain-specific knowledge. By their very nature, the informed algorithms are more sophisticated than the blind algorithms. Indeed, if your heuristics are well designed, you will find that you can solve larger Tile Game instances using informed search algorithms.

The informed search algorithms that you will implement are variants on A* search. A* is optimal when it is endowed with an admissible (i.e., optimistic) heuristic. When its heuristic is inadmissible, optimality is no longer guaranteed. Still, you will experiment with inadmissible heuristics in this assignment. Since inadmissible heuristics are less constrained than admissible ones, they are often faster, so when they are not very inadmissible (i.e., when they are only ever slightly pessimistic), they may find optimal or only slightly suboptimal solutions faster than admissible heuristics.

In Modern AI, where the goal is usually to design agents that make rational decisions relative to some objective, the name of the game is to develop heuristics that trade off between optimality and complexity (in terms of time and/or space). That is precisely the subject matter of this assignment.

3 Data Structures

In this (and perhaps all CSCI1410) assignment(s), we worked out the problem of how to represent the data—in this case, a Tile Game—for you. We chose to represent a state in the Tile Game as an n-tuple of n-tuples. The outer n-tuple represents the rows in a Tile Game, while the inner n-tuples represent the entries in the corresponding row's columns. For example, the goal state is represented in Python as:

```
((1,2,3),(4,5,6),(7,8,9))
```

4 Algorithms

Once again, the algorithms you will implement in this assignment are depth-first (DFS), breadth-first (BFS), iterative deepening (IDS), and A*. These algorithms are described in Sections 3.3 (generic search), 3.4 (blind search), and 3.5 (informed search) of Russell and Norvig.

Recall from class that search can be conceptualized as expanding a tree. Note, however, that no search algorithm builds this search tree in its totality. On the contrary, these algorithms simply move through the search space, by taking actions that lead them from one state to another.

Moreover, while it is convenient to think about searching a tree, a search space is not necessarily a tree. In general, *it is a graph*, because it is possible to revisit states. In the Sliding Tiles puzzle, for example, you can move a tile into the empty space and back again indefinitely. Hence, to the extent that memory permits, your implementation should keep track of previously visited states, and should not visit them again.

5 Your Tasks

Your primary task is to implement more and more sophisticated algorithms so as to solve larger and larger instances of the Tile Game. After implementing these algorithms, you will run a series of experiments to test them on games of varying size, and you will summarize your findings. You might find, for example, that DFS can only solve Tile Games of size 2, while A* with an admissible heuristic can optimally solve Tile Games of size 3.

5.1 Coding

In `search.py`, you will implement depth-first (`dfs`), breadth-first (`bfs`), iterative deepening (`ids`), and A* (`astar`). Each search function takes in a `SearchProblem`, called `problem`, and outputs a *path* through the search tree. This 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.

To complete your implementation of the informed search algorithms, you should use your knowledge of the Tile Game to develop your admissible heuristic. You are welcome to experiment with multiple heuristics, but you should submit only one. Section 3.6 of the textbook provides examples of heuristic functions.

With the exception of your heuristics, which exploit domain knowledge, your implementations of the search algorithms should be problem agnostic, meaning they should work on any `SearchProblem`: e.g., Sliding Tiles, as well as the Tile Game, and even a routing problem like finding a driving route from Providence to Boston or a flight from Providence to Tokyo.

5.1.1 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 (hint: consider the differences between iterative and recursive algorithms).

5.1.2 Tips

- Don't forget to write your tests *before* you write any code! Otherwise, how would you know whether your implementation was correct? Never, and we mean *never*, write your tests based on the output of your implementation.
- Don't copy code. (And we don't mean from other people; *obviously*, you should not do that!) Don't copy code from yourself. If you find some portion of your code has multiple uses, abstract it out, so that you can call it from multiple places.

5.2 Writeup

In addition to your code, you should also submit a PDF in which you answer the following questions, one at a time. While not required, we recommend that you use LaTeXto typeset your work.

1. Describe your admissible heuristic, and informally argue that it is admissible. Explain why you chose this heuristic.
2. Describe an inadmissible heuristic. Give examples where it is inadmissible. Informally speaking, how inadmissible is it?
3. Design and run experiments to test the various algorithms on problem instances of varying size (say, 10 problems of each size). Test the blind algorithms on smaller problem instances, and the informed ones on larger problem instances. Report your findings in a table, and summarize them in a few sentences.

   Your table/summary should answer questions like: How often do the blind algorithms succeed as compared to the informed algorithm? On problem instances where A* succeeds, so that you know the length of the optimal solution, how suboptimal, on average, are the paths that the not-provably-optimal algorithms find (i.e., what is the average difference between the length of the optimal path and the suboptimal ones?).

5.3 Ethics Questions

Companies such as Google, Uber, and Waze have used A* or similar algorithms for their path finding features. Please read this article and answer the following question.

1. The article gives an example of how the shortest path may not be the *best* path or the path that all parties want. When developing a product, whose needs should be prioritized? Do you believe Uber, and other tech companies, are obligated to always be "user first"?

Please submit your writeup for both Sections 5.2 and 5.3 via Gradescope (see the Gradescope guide on the course website for more details).

6 The Code Files

6.1 Files to Modify

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

6.2 Core Source Code

- `searchproblem.py` - This file contains an abstract class, `SearchProblem`, that can represent search problems. The `SearchProblem` class has three abstract methods that are shared among all search problems. Be sure to read the function headers and their docstrings before you begin.
- `tilegameproblem.py` - This file contains the `TileGame` class, which extends the `SearchProblem` class. It contains implementations of the abstract methods in `SearchProblem`. It also contains some helper functions that you may find useful when you code your heuristics.

  You will notice that the dimension of the Tile Game is adjustable. You should take advantage of this feature: while developing your code, you can test it on 2-by-2 games, which should run more quickly than 3-by-3 games or 4-by-4 games. But once you are confident in your implementations, you should of course test them on larger problem instances as well.

6.3 Testing Source Code

- `dgraph.py` - This file contains an implementation of a `DGraph`, which represents directed graphs, as a `SearchProblem`. `DGraph` allows you to create test cases (graphs) that you can use to test your search algorithms. We use `Dgraph` to test your implementations.

  `DGraph` represents directed graphs using matrices. Each state is each represented by a unique index in $\{0, 1, ..., S - 1\}$, where S is the number of states, and the cost of moving directly from state i to state j is the entry in the 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 the entry in the matrix at row i, column j is set to `None`.
- `unit_tests.py` - This file contains a testing suite with some trivial test cases. To run these tests, execute `python unit_tests.py` inside the virtual environment. We encourage you to write additional unit tests, as well as more sophisticated tests (integration, end-to-end, etc.).

6.4 Support Code

The queue module contains `Queue`, `LifoQueue`, and `PriorityQueue`, which you are free to use to implement your search algorithms. 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. Visit `https://docs.python.org/3/library/queue.html` for more information about the module.

7 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. So be sure that `get_successors` is not called unnecesarily. 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 admissible Tile Game heuristic, you will be graded based on how many states A* expands. Your score for this part will be determined by the following formula:

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

  where n_{ours} and n_{yours} are the number of nodes expanded by our heuristic and your heuristic, respectively, on a pre-selected suite of Tile Game instances.

  You can score your heuristic on your own **on a department machine** by using the following command inside the course virtual environment:

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

8 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 here.

9 Install and Handin Instructions

To install, accept the GitHub Classroom assignment at this link. This will create a private GitHub repository with the stencil code for you to work on the assignment.

To handin, first make sure to push any changes you want to test to your private repository. You can do this by running

```
git add .
git commit -m "<a commit message describing what you changed>"
git push
```

Then, on Gradescope, click on the assignment you are submitting for.

Under “Submission Method”, please select GitHub.

Under “Repository”, you can search for your repository by typing “csci-1410” and selecting the repository for this assignment.

Under “Branch”, you can select any branch that you want to be graded. So if you’re testing something on a branch, you can see how its functionality performs here, before merging it to master. Feel free to upload your assignment as many times as you like before the deadline.

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