Assignment 1: Search Algorithms

CS 383 Fall 2019

You are encouraged to discuss the assignment in general with your classmates, and may collaborate with 1-2 other students on the design and logic of your solution. If you choose to do so, you must indicate with whom you worked. In addition, the code you submit must be entirely your own; two students submitting the same code will be considered plagiarism.

Code must be written in a reasonably current version of Python (>3.6), and be executable from a Unix command line using the provided `run.py` script. You are free to use Python's standard libraries for data structures, but you must implement the search method you use yourself. Do not use a library for search.

Please post questions or problems to Piazza. If you know the answer to a posted question, you can help your classmates by posting a response. This will constribute to your participation score for class. However, be smart about your responses: Don't post completed code or give away answers!

Preliminary: Gridworlds

Gridworlds are a type of environment that are frequently used in AI research. They are designed to model at an abstract level the problem of navigating in a physical environment. Generally, the task is to find the path from *initial state* to some other *goal state* with the lowest cost. There may be obstacles or hazards in the world that the agent needs to avoid on its way to the goal. Your job in this assignment will be to solve this task using four different algorithms: breadth-first search (BFS), uniform cost search (UCS), greedy best-first search (GBFS), and A*.

In this assignment, gridworlds are specified in text files. Each gridworld is a discrete, two-dimensional environment. Each state is referenced by a tuple, `(x, y)`, where `x` represents the column and `y` represents the row. The top-left state is always `(0, 0)`. In each state, the agent can move up, down, left, or right (but not diagonally). However, the agent may not go "off the grid" (even if it wishes to avoid paying taxes). Consider the gridworld specified in the file `gw/1.txt`:

```
 s11
#21
g11
```

Each character in the file corresponds to a single state. The character `s` specifies the initial state (in this case, `(0, 0)`). The character `g` specifies the goal state (in this case, `(0, 2)`). The character `#` specifies a wall. For instance, in this example, the agent cannot enter `(0, 1)`. Another way of thinking about this is that the state *doesn't exist*, that is, there is no state corresponding to `(0, 1)`. For every other state, the character specifies the cost of entering that state, which can be any integer from 0 to 9. For example, the cost of entering `(1, 0)` is 1, whereas the cost of entering `(2, 0)` is 2. The cost of entering the initial or goal state is always 1. However, because the agent *starts* in the initial state, its cost should not be counted when considering the total path, unless it was reentered. On the other hand, the cost of entering the goal state

should *always* be counted.

Step 0: Hello Gridworld!

We have provided starter code which you are required to use in order to complete the assignment, contained in the `assignment1` module. In order to make this module available to your python path, run the following command from the root directory of the assignment:

```
pip install -e .
```

Don't worry: This will not really "install" anything on your machine. All this command does is add an entry in `pip` that allows the run script to access the code. Depending on the specifics of your Python installation, you may need to call `pip3` instead of `pip`. Additionally, you may need to run Python scripts using `python3` instead of `python`. In order to test that the module is accessible, run the command:

```
python hello_gridworld.py
```

You should see output that looks like:

```
Hello Gridworld!
[['s', 1, 1], ['#', 2, 1], ['g', 1, 1]]
```

If you do not see this output, there is likely a problem. While we encourage you to write your own test cases/scripts in separate files (e.g., using the standard `unittest` module included with Python), feel free to modify this script as you see fit. Do not, however, modify the `run.py` script as this may impact the grading of your assignment!

Step 1: Finish the Gridworld Implementation (20%)

We provided a partial implementation of a `Gridworld` class in `assignment1/gridworld.py`. The starter code reads in a gridworld specified by `filename` and stores it in a 2D array referenced by `self.state`. Your first task is to complete the implementation by specifying the `successors` and `cost` functions. The `successors` function should take a state and return a list of all possible successors of that state. The `cost` function should take a state and return the cost of moving into that state. We encourage you to write your own tests to ensure the correctness of these functions. Remember to consider edge cases! Literally!

Step 2: Implement Four Search Algorithms (60%)

You must implement the following four agents:

1. Breadth-first search (`BFS`)
2. Uniform cost search (`UCS`)
3. Greedy best-first search (`GBFS`)
4. A* (`AStar`)

Each of them should be implemented in the appropriate file in `assignment1/agents`. Your job is to implement the `search` function for each class. The `search` function should return three things:

1. An array of tuples representing the solution path,
2. The cost of the solution,
3. The number of nodes expanded during the search.

We provided a run script which may be used to test your agents. For an example, we also included a `RandomSearch` agent which searches randomly for a solution, and returns the first path found. The run script may be used as follows:

```
python run.py random gw/0.txt
```

This will run the random agent on example Gridworld 0 stored in `gw/0.txt`. If you implemented the gridworld correctly, you should see output that looks like:

```
solution [(0, 0), (1, 0), (0, 0), (1, 0), (1, 1)]
cost 4
nodes_expanded 4
```

Your output may be different depending on the path found by the random agent, but the format should be the same regardless of the agent!

In general, to invoke the `run.py` script, you should run

```
python run.py [agent] [gw]
```

where `[agent]` is one of the strings `bfs`, `ucs`, `gbfs`, `astar`, or `random`, and `[gw]` is the file path to a gridworld text file (all parts of this command are case sensitive).

Heuristics

Some of the algorithms will require you to write heuristic functions. Come up with the best admissible heuristic you can think of. You may lose points for using a sufficiently bad heuristic.

To make sure the code works, run:

The output of the agents you implement will be the same information: A python list of tuples containing the state visited by the solution found, the path cost of the solution, and the number of nodes expanded while searching for the solution.

Step 3: Compare the Algorithms by Example

Finally, we would like you to demonstrate your understanding of the behavior of each algorithm by constructing examples where their outputs differ. Please provide:

1. An example where BFS returns the optimal solution

2. An example where GBFS returns a suboptimal solution
3. An example where GBFS returns the optimal solution while expanding fewer nodes than A*
4. An example where A* expands fewer nodes than UCS
5. (Extra Credit) An example where each algorithm produces a different output. (The output being the solution, the cost, and the number of nodes expanded.) Don't worry about random search for this test, just the four you've written.

Each solution should be stored in a text file in the `example` folder. For example, the solution to question 1 should be put in `example/1.txt`.

Submitting Your Assignment

Your code should be submitted on Gradescope. The autograder will run a subset of tests on your code to ensure that you have implemented the API correctly. If the autograder fails to run, there is a problem in your code. However, do not rely on the autograder to test the correctness of your code! It will only test that your algorithm returns correctly formatted results.