Homework 1 - Deterministic Search

Introduction

In this exercise, you take the role of the owner of a taxi business. Your goal is to deliver the passengers to their destinations in the shortest time possible. To achieve this most efficiently, you must make use of the search algorithms shown in class, with the first task being modeling the problem precisely.

Environment

The environment is a rectangular grid - given as a list of lists (exact representation can be found in the “Input” section). Each point on a grid represents an area. An area can be either passable or impassable for the taxis. On the grid, there are passengers waiting to be delivered to their destinations. Moreover, there are gas stations, where the taxis can get additional fuel to continue serving the customers.

Actions

You assume control of the taxi fleet, with the intention of bringing the passengers to their destinations in the shortest time possible.

1. Move on the grid up to one tile vertically or horizontally (taxis cannot move diagonally). The taxis cannot move to an impassable tile. Every movement takes 1 unit of fuel, and cannot be performed if the fuel level of a certain taxi is equal to 0. The action’s syntax is (“move”, “taxi_name”, (x, y)), so if you want to move a taxi that is named “taxi 1” to a tile (0,2), the correct syntax for this action is (“move”, “taxi 1”, (0, 2)). This is the only action that decreases the amount of fuel, and the only action that cannot be performed when the taxi has no fuel.

2. Pick up passengers if they are on the same tile as the taxi. The number of passengers in the taxi at any given turn cannot exceed this taxi’s capacity. The syntax is (“pick up”, “taxi_name”, “passenger_name”).

3. Drop off passengers on the same tile as the taxi. The passenger can only be dropped off on his destination tile and will refuse to leave the vehicle otherwise. The syntax is (“drop off”, “taxi_name”, “passenger_name”).

4. Refuel the taxi. Refueling can be performed only at gas stations and brings the amount of fuel back to the maximum capacity. The syntax is ("refuel", "taxi_name")

5. Wait. Does not change anything about the taxi. The syntax is (“wait”, “taxi_name”)

These five are known as atomic actions, as they relate to a single taxi. Since you control a fleet of taxis, you can command them to do things simultaneously. The syntax of a full action is a tuple of atomic actions. Each taxi has to do something during any given turn, even if it is waiting.

Example of a valid action:

Assume an input has 3 taxis, and all of the atomic actions are valid:

((“move”, “taxi 1”, (1, 2)), (“wait”, “taxi 2”), (“pick up”, “very_fancy_taxi”, “Yossi”))

Additional rules

- Assume taxis start empty, i.e. don't have passengers inside them.
- Assume taxis start with full tanks, i.e. maximal amount of fuel.
- Assume taxis can only start at a passable tile.
- Only one taxi can be at any tile at the same time.
- Some problems are unsolvable! This may stem from a passenger located in the impassable terrain, fuel issues, lack of connectivity of the map and several other reasons. The expected output in this case is (-2, -2, None) - more on the outputs in the “Output” section

Goal

Your goal is to deliver every passenger to their destination in the shortest number of steps (or turns) possible.

Input and the task

As input, you will get a dictionary that describes the initial environment. For example:

```
{
    "map": [['P', 'P', 'P', 'P'],
            ['P', 'P', 'P', 'P'],
            ['P', 'I', 'G', 'P'],
            ['P', 'P', 'P', 'P'], ],
    "taxis": {'taxi 1': {"location": (3, 3),
                         "fuel": 15,
                         "capacity": 2}},
    "passengers": {'Yossi': {"location": (0, 0),
                             "destination": (2, 3)},
                   'Moshe': {"location": (3, 1),
                             "destination": (0, 0)}
                   }
},
```

Where:

- “map” is a two-dimensional list of strings. ‘P’ means terrain passable for taxis, while ‘I’ means impassable. In the example, tile (2, 1) is impassable for taxis. ‘G’ is a gas station, and is passable for taxis.
- “taxis” is a dictionary of dictionaries that contains the names of the taxis in the problem along with their initial position, fuel tank size, and passenger capacity. The fuel value is both the maximal fuel capacity and the initial fuel capacity.
- “passengers” is also a dictionary of dictionaries that contain each passenger's name, starting locations, and destinations.

This input is given to the constructor of the class TaxiProblem as the variable “initial”. The variable “initial” in the constructor is then used to create the root node of the search, so you will have to transform it into your representation of the state somewhere before the `search.Problem.__init__(self, initial)` line.

Moreover, you have to implement the following functions in the TaxiProblem class:
`def actions(self, state)` - The function that returns all available actions from a given state.
`def result(self, state, action)` - The function that returns the next state, given a previous state and an action.
`def goal_test(self, state)` - Returns “True” if a given state is a goal, “False” otherwise.
`def h(self, node)` - returns a heuristic estimate of a given node. Your heuristic needs to be admissible to guarantee an optimal solution with A*.

Moreover, you are to implement two more heuristic functions:
`def h_1(self, node)` - A simple heuristic function that returns the following: (number of unpicked passengers * 2 + the number of picked but yet undelivered passengers)/(number of taxis in the problem).

`def h_2(self, node)` is defined as follows:
Let $D(i)$ be a Manhattan distance between the initial location of an unpicked passenger i and her destination.
Let $T(i)$ be a Manhattan distance between the taxi where a picked but undelivered passenger i is, and her destination.

Then the heuristic must compute the following:

$$\frac{\sum_{i \in unpicked-passengers}(D(i)) + \sum_{j \in picked-but-undelivired-passengers}(T(j))}{number-of-taxis}$$

`h` may call `h_1` and `h_2`, or be completely different if you so wish. We run your code with the function `h` you define, but the correct implementation of `h_1` and `h_2` is mandatory.

Important: you are free to choose your own representation of the state. Only one restriction applies - the state should be *Hashable*. We **strongly suggest** not creating a new class for storing the state and sticking to the built-in data types. You may, however, find unhashable data structures useful, so you can run functions that transform your data structure to a hashable and vice versa.

We require the plan to be optimal, which means you have to come up with an **admissible** heuristic.

Evaluating your solution

Having implemented all the functions above, you may launch the A* search (already implemented in the code) by running check.py. Your code is expected to finish the task in 60 seconds.

Output

You may encounter one of the following outputs:

- A bug - self-explanatory.
- (-2, -2, None) - No solution was found. This output can stem either from the unsolvability of the problem or from a timeout (it took more than 60 seconds of real-time for the algorithm to finish).
- A solution of the form (list of actions taken, run time, number of turns).

Code handout

The code that you receive has 4 files:

1. ex1.py - The only file that you should modify, implements the specific problem.
2. check.py - The file that includes some wrappers and inputs, the file that you should run.
3. search.py - A file with implementations of different search algorithms (including GBFS, A*, and many more).
4. utils.py - The file that contains some utility functions. You may use the contents of this file as you see fit.

Note: We do not provide any means to check whether your code's solution is correct, so it is your responsibility to validate your solutions.

Submission and grading

You are to submit **only** the file named ex1.py as a python file (no zip, rar, etc.). We will run check.py with our own inputs, and your ex1.py, using A* as the search algorithm of choice. The check is fully automated, so it is important to be careful with the names of functions and classes. The grades will be assigned as follows:

- 60% - For valid outputs (optimal plans) on inputs with one taxi.
- 25% - For valid outputs (optimal plans) on inputs with more than one taxi.
- For a correct implementation of h_1 and h_2 - 5% each.
- 5% - For submission (only the fact of submission, no grading) of exercise 0
- The submission is due on the 24.11 at 23:59
- Submission in pairs/singles only.
- Write your ID numbers in the appropriate field ('ids' in ex1.py) as strings. If you submit alone, leave only one string.
- The name of the submitted file should be "ex1.py". **Do not change it.**

Important notes

- You are free to add your own functions and classes as needed, keep them in the ex1.py file. Do not submit other files.
- Note that you can access the state of the node by using node.state (for calculating h for example).
- We encourage you to start by implementing a working system without a heuristic (by making the h function return 0 for example) and add the heuristic later.
- We encourage you to double-check the syntax of the actions as the check is automated.
- More inputs will be released a week after the release of the exercise.
- You may use any package that appears in the standard library and the Anaconda package list, however, the exercise is built in a way that most packages will be useless. You may also use the aima3 package which accompanies the coursebook.
- We encourage you not to optimize your code prematurely. Simpler solutions tend to work best.