Homework 1 - Deterministic Search

Introduction

In this exercise, you assume the role of a head of a drone delivery agency. Your main goal is to deliver the packages to your clients 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 delivery drones. Moreover, there are packages lying in different locations around the grid. The packages can be picked up by drones and delivered to clients.
Clients can move on a pre-determined and known path, and each client has a list of required packages. In general, the client can request non-existing packages (then the problem is unsolvable). Moreover, there could be packages that no one needs. One drone can carry up to two packages at once.

The task is done in timestamps (or turns), so the environment changes only after you apply an action.

Actions

You assume control of the drones, that can do various things:

1. Move on the grid up to one tile vertically or horizontally (drones cannot move diagonally). The drones cannot move to an impassable tile. The action’s syntax is (“move”, “drone_name”, (x, y)), so if you want to move a drone that is named “drone 1” to a tile (0,2), the correct syntax for this action is (“move”, “drone 1”, (0, 2)).
2. Pick up packages if they are on the same tile as the package. The syntax is (“pick up”, “drone_name”, “package_name”).
3. Deliver a package to a client if the drone has the package and is on the same tile as the client. The syntax is (“deliver”, “drone_name”, “client_name”, “package_name”). Note that if the delivery occurs at a time stamp t, the client may move out of the tile, and be at a neighboring tile already in the time stamp t+1.
4. Wait. Does not change anything about the drone. The syntax is (“wait”, “drone_name”)

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

Examples of a valid action:
If you have one drone: ((“pick up”, “drone 1”, “package 2”),)
If you have 2 drones: ((“deliver”, “drone 1”, “Yossi”, “package 3”), (“wait”, “drone 2”))

Additional rules

Every turn, first the drones execute their commands, and only after that clients move. I.e. if you have a client on tile X that is about to go to a tile Y, you have to do the delivery on tile X.

The delivered packages disappear and cannot be picked up again. You cannot drop off packages for purposes other than delivery. No two clients will require the same package. Clients will not pick up the packages by themselves, every package must be delivered by a drone.

Clients move along a pre-determined path, which is given by a list of coordinates. For example, if a path is [(0, 0), (0, 1), (1, 1), (1, 0)], a client starts on a tile (0, 0), is on tile (0, 1) after 1 action, on (1, 1) after 2 actions, on (1, 0) after 3 actions, on (0, 0) after 4 actions, on (0, 1) after 5 actions and so on.

Clients’ path can go through impassable areas (they are impassable only for the drones), and does not have to include only adjacent tiles (for example, [(0, 0), (2, 2), (5, 5)] is a valid path too).

Goal

Your goal is to deliver to each customer every package on his list in the shortest amount of steps (or turns) possible.

Input and the task

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

```
{
    "map": [['P', 'P', 'P', 'P'],
            ['P', 'P', 'P', 'P'],
            ['P', 'I', 'P', 'P'],
            ['P', 'P', 'P', 'P'], ],
    "drones": {'drone 1': (3, 3)},
    "packages": {'package 1': (0, 2),
                 'package 2': (2, 0)},
    "clients": {'Yossi': {"path": [(0, 1), (1, 1), (1, 0), (0, 0)],
                          "packages": ('package 1', 'package 2')}}
}
```

Where “map” is a list of lists of strings. ‘P’ means terrain passable for drones, while ‘I’ means impassable. In the example, tile (2, 1) is impassable for drones.
“drones” contains the names of the drones in the problem along with their initial position.
“packages” contains the names of packages with their starting locations.
“clients” gives us the path and the required packages for every client. A client starts on the first tile on its path.

This input is given to the constructor of the class DroneProblem 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 DroneProblem 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.

Note: you are free to choose your own representation of the state. Only one restriction applies - the state should be *Hashable*. We **strongly suggest** not to create a new class for storing the state and stick 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 vise versa.

Evaluating your solution

Having implemented all the functions above, you may launch the GBFS 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, amount of turns)

Code handout

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 that has 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 the solution your code provided 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 GBFS 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:

- 80% - if your code finishes solving the test inputs within the timeout, with no regard to the cost of the solution
- 20% - Grading on the relative performance of the algorithm, which is judged by the length of the solution only (as long it finishes within the timeout).
- 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
- 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 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.