P5: Mini-Minecraft Planner

Introduction

In this programming assignment you will need to implement, in Python, a planner for a Minecraft-like item crafting problem. Your planner must be based on the A* algorithm and it is part of your task to define a heuristic for this problem. The purpose of your planner is to "craft" a set of items described by a given goal. This is achieved by taking actions which can obtain items or which can combine items into more complex ones. In this process, some items are consumed and others may be required but not consumed (e.g. tools). A result of an action is the generation of a new item, according to the given recipes. The list of items as well as the list of actions you can use is provided by us. Your goal is to define a planning problem (initial state and goal state) using these lists and to implement the A* algorithm with a heuristic also defined by you. You will pose the most complex problem your system is able to solve in 30 seconds.

Example

In order to run and test your planner you need to execute the following command (assuming you are in the /src folder):

$ python3.5 craft_planner.py

This code will run for up to 30 seconds trying to find a path for a given planning problem. Whenever you run your code it will print the list of all items, the initial state of your inventory, the described goal and an example crafting recipe, as following:

```
All items: ['bench', 'cart', 'coal', 'cobble', 'furnace', 'ingot', 'iron_axe', 'iron_pickaxe', 'ore', 'plank', 'rail', 'stick', 'stone_axe', 'stone_pickaxe', 'wood', 'wooden_axe', 'wooden_pickaxe']
Initial inventory: {}
Goal: {'furnace': 1}
Example recipe: craft stone_pickaxe at bench -> {'Time': 1, 'Consumes': {'cobble': 3, 'stick': 2}, 'Produces': {'stone_pickaxe': 1}, 'Requires': {'bench': True}}

30.000001108000106 seconds.
Failed to find a path from {} within time limit.
```

The last two lines of the output is the time that it took to find the path and the path itself. Currently, there is no search implemented, so the code will return the time (approximately 30 seconds) and an error message saying no path was found. The list of items as well as the initial and the goal state are described in the file crafting.json. This file is parsed by the main section of the provided code.

Base Code Overview

- craft_planner.py

This is base code for this assignment. You will have to implement make_checker, make_effector, make_goal_checker, heuristic and search functions. You can also create new auxiliary functions if you need. The checker's role is to assess whether a crafting recipe is valid in a given state. The effector's function is to return the state resulting from applying the rule to a given state. The routines make_checker and make_effector are used to create functions for any number of rules, and for interpreting the rules defined in the json files. In this way, you won't have to hand code each rule independently.

The search function accepts these parameters:

- ***graph***: a *function* that can be called on a node to get adjacent nodes
- ***initial***: an initial state
- ***is_goal***: a *function* that takes a state and returns True or False
- ***limit*****:** a float or integer representing the maximum search distance**.** Without this, your algorithm has no way of terminating if the goal conditions are impossible
- ***heuristic***: a *function* that takes some next_state and returns an estimated cost

- crafting.json

This file describes in the json format an item-crafting planning problem which the craft_planner.py code loads into a dictionary. The structure is as follows:

- ***'Items'*** : list of items in the world
- ***'Initial'***: a dictionary of the initial inventory
- ***'Goal'***: a dictionary containing the minimum quantities of items for the goal state
- ***'Recipes'***: a dictionary of rules or recipes that describe actions in the world. Here, the keys are the names of the actions, e.g. "craft wooden_pickaxe at bench". Each recipe contains:
    - 'Consumes': a dictionary of resources that are used during the action
    - 'Requires': a dictionary of items that are required *but not consumed* by the action
    - 'Produces': a dictionary of items produced by the action
    - 'Time': the cost in time for the action

Heuristics and Action Pruning

Your heuristic should guide the search process in order to make it explore fewer states. This idea is called *pruning* and it makes the algorithm run faster. To create a heuristic one typically analyzes characteristics of the problem in order to find subproblems that are not worth exploring during the search. For example, in the Minecraft crafting problem there's no reason to ever craft a tool (pickaxe, furnace, etc. -- things appearing in 'Requires' conditions) if you already have one of them. If you create a heuristic that returns infinity in this case (zero otherwise), even a very plain A* state progression planner should be able to find optimal plans for crafting a furnace within the time limit.

State-space planners often waste their time considering every possible ordering of order-insensitive actions. If I need to get 8 planks from scratch, should I "punch,craft,punch,craft" or "punch,punch,craft,craft"? They have the same costs, so the default planner will try both. Can you modify your planner (either in the heuristic or in the function that computes graph edges) so that it only explores one of these interchangeable possibilities? It's okay to modify your algorithm so that the heuristic can inspect the proposed action as well. It might ignore the state, and decide what value to return just by looking at how the action relates to the goal.

Requirements

- Implement an A* based planner with a heuristic that is capable of solving all the test cases described in the next section

- Provide a description of your search approach, including any your heuristic and any modification that you might have done in the basic A* structure.
- Describe (and give a JSON file for) the most difficult goal your implementation is able to solve within 30 seconds

Grading Criteria

Each of the test cases below should be completable by your planner with less than *30* seconds of real-world time on whatever machine you chose to use for demonstration. The total cost (in recipe time units) and length (in number of required actions) for cost-optimal plans are given below.

- Given {'plank': 1}, achieve {'plank': 1}. [cost=0, len=0]
- Given {'bench': 1, 'plank': 3, 'stick': 2}, achieve {'wooden_pickaxe': 1}. [cost=1, len=1]
- Given {'plank': 3, 'stick': 2}, achieve {'wooden_pickaxe': 1}. [cost=7, len=4]
- Given {}, achieve {'wooden_pickaxe': 1} [cost=18, len=9]
- Given {}, achieve {'stone_pickaxe': 1} [cost=31, len=13]
- Given {}, achieve {'furnace': 1} [cost=48, len=22]
- Given {}, achieve {'iron_pickaxe': 1} [cost=83, len=33]
- Given {}, achieve {'cart': 1} [cost=104, len=38]
- Given {}, achieve {'cart': 1, 'rail': 10} [cost=172, len=58]
- Given {}, achieve {'cart': 1, 'rail': 20} [cost=222, len=87]

Submission Instructions

Submit a zip file named in the form of "Lastname1-Lastname2-P5.zip" containing:

- A python file name **craft_planner.py** that, when it is executed outputs the
  - The plan, if found, for satisfying the goal
  - The amount of *compute* time the search process took
  - The final cost (*game time*) of satisfying the goal
- A json file named **crafting.json** with most difficult goal you've found that your implementation is able to find a solution for within 30 seconds.
- A text file named **solution.txt** with the solution your code finds as well as the plan's cost to the difficult goal described in the **crafting.json** file. Report also the number of states your search visited for that goal.
- A text file named **README** with a short description of both your search approach and the heuristic you have implemented, along with the names of both teammates and any other useful information.
- Outstanding solutions will earn extra credit.