P5: Mini-Minecraft Planner

Goal

For this assignment, you will implement a GOAP-style planner for a Minecraft-like item crafting problem. You will pose the most complex problem your system is able to solve in 30 seconds, and turn in a description of that problem, the key elements of your solution, and your code.

Your planner will operate in the domain of Minecraft-style item crafting rules. 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 items. 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.

You may implement any planning algorithm you like (state progression, goal regression, bidirectional search, or sampling-based search) using any heuristic scheme you like (plain dijkstra’s / no heuristic, custom heuristic based on player knowledge, automatic heuristic based on a partial results, etc.), so long as the time requirements are satisfied.

Implementing a forward planner with no heuristic will get you off to an easy start.

Requirements

In addition to submitting the implementation of your planner, you will include the most difficult case that your planner can handle within the time limit. If you and your partner have worked as many hours as you can afford, stop and don’t fret about getting the last fractions of a point. Before you reach that point, ask others for heuristic ideas (on Piazza).

The validity of your planner’s plans should come from the correctness of your implementation, not a coincidence that is specific to one of the challenge problems. See the *Plutonium Challenge* described below for a quick correctness check.

Test Cases

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. These are test cases; you will not be turning solutions in.

The total cost (in recipe Time units) and length (in number of required actions) for cost-optimal plans are given below. However, your plans need not be optimal so long as they are valid *and* produced within the real-world time limit.

* 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]

Base Code

There is base code for this assignment (craft\_planner.py). However, the given functions act as a guide and are not required, should you desire to implement your own method of solving the problem. Furthermore, there are data files (Crafting.json, Plutonium.json) supplied for testing. These files lay out two different systems in the same format (json) which your code will load 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

This code examples shows how to load the crafting rules and access their details:

**import** json

**with** open('Crafting.json') **as** f:

Crafting = json.load(f)

# List of items that can be in your inventory:

**print**(Crafting['Items'])

# example: ['bench', 'cart', ..., 'wood', 'wooden\_axe', 'wooden\_pickaxe']

# List of items in your initial inventory with amounts:

**print**(Crafting['Initial'])

# {'coal': 4, 'plank': 1}

# List of items needed to be in your inventory at the end of the plan:

# (okay to have more than this; some might be satisfied by initial inventory)

**print**(Crafting['Goal'])

# {'stone\_pickaxe': 2}

# Dict of crafting recipes (each is a dict):

**print**(Crafting['Recipes']['craft stone\_pickaxe at bench'])

# example:

# { 'Produces': {'stone\_pickaxe': 1},

# 'Requires': {'bench': True},

# 'Consumes': {'cobble': 3, 'stick': 2},

# 'Time': 1

# }

Implementation Strategy

The steps in this strategy guide are strictly optional, but it may help you to follow them.

**Step 1: Load the Crafting Rules**

See the code above for how to load the rules from the JSON file.

**Step 2: Compile the Rules for Fast Application**

Here’s a use of Python’s namedtuple structure to quickly make a container class and use it to hold compiled recipes.

**from** collections **import** namedtuple

Recipe = namedtuple('Recipe',['name','check','effect','cost'])

all\_recipes = []

**for** name, rule **in** Crafting['Recipes'].items():

checker = make\_checker(rule)

effector = make\_effector(rule)

recipe = Recipe(name, checker, effector, rule['Time'])

all\_recipes.append(recipe)

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. Below is some boilerplate code to get you started on make\_checker and make\_effector.

def make\_checker(rule):

… # this code runs once

 # do something with rule['Consumes'] and rule['Requires']

def check(state):

… # this code runs millions of times

return True # or False

return check

def make\_effector(rule):

… # this code runs once

 # do something with rule['Produces'] and rule['Consumes']

def effect(state):

… # this code runs millions of times

return next\_state

return check

**Mini-lesson on wrapped/nested/decorated functions:**

The idea here is to return a *function* you have created with aspects specified by the rule argument of the outer function declaration. Below is an example of such a system:

def make\_adder(number\_to\_be\_added):

# This function is called once, constructing the function below,

# then returns that function for later use.

def adder(x):

return x + number\_to\_be\_added

return adder

Let’s walk through how this works. The general idea is to create a function that adds some number to x when called. In principle, using the above code, we can create any number of adder functions, like so:

add5 = make\_adder(5) # Returns a *function*!

x = 3

y = add5(x) # 8; using the function add5.

add20 = make\_adder(20) # Returns a *function*!

z = add20(x) # 23

So how does this apply in P5? The make\_checker and make\_effector, if implemented correctly, can be used to create functions for any number of rules, interpreting the rules as defined by the json files. In this way, you won’t have to hand code each rule independently.

As mentioned earlier, these implementation choices are optional. You could, of course, make a function that takes both the state *and* a given rule, checking to see if the state meets the rule’s requirements.

**Step 2: Implement a Generic Search Algorithm (basic A\* recommended, MCTS *not* recommended)**

Implement a generic version of search with a signature like this

def search(graph, initial, is\_goal, limit, heuristic):

...

return total\_cost, plan

Parameters:

* **graph**: a *function* that can be called on a node to get adjacent nodes
  + def graph(state):  
     for r in all\_recipes:  
       if r.check(state):  
         yield (r.name, r.effect(state), r.cost)
  + the result should be a sequence/list of **(action, next\_state, cost)** tuples
  + **action**: the name of the crafting recipe applied
  + **next\_state:** the state resulting from applying the recipe in the current state
  + **cost**: the Time associated with the crafting recipe
* **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

**Step 3: Implement the Building Blocks of your Planner**

def make\_initial\_state(inventory):

...

return state

def make\_goal\_checker(goal):

... # this code runs once

def is\_goal(state):

... # this code runs millions of times

return True # or False

return is\_goal

def make\_checker(rule):

...

def check(state):

...

return True # or False

def make\_effector(rule):

...

def effect(state):

...

return next\_state

def graph(state):  
 for r in all\_recipes:  
 if r.check(state):  
 yield (r.name, r.effect(state), r.cost)

def heuristic(state):

...

return 0 # or something more accurate

if \_\_name\_\_ == “\_\_main\_\_”:

initial\_state = make\_initial\_state(Crafting['Initial'])

is\_goal = make\_goal\_checker(Crafting['Goal'])

#etc

**Step 4: Test your Planner on the Project Requirements**

You can either edit the Initial and Goal conditions in Crafting.json or override them in your own code. Look at the list of requirements and see how far your planner can get through them (in order of difficulty) within a only few seconds of search. Don’t waste your time waiting up to 30 seconds in each edit-and-test cycle.

**Step 5: Play with Heuristics and Action Pruning**

If you planner isn’t fast enough (it won’t be at the start!), here is where you look into ways of speeding it up. **Feel free to share ideas for heuristics and action pruning mechanisms with other teams and discuss them Piazza. The primary challenge of this projects should come in formalizing the ideas on your computer, not so much as in coming up with them in the first place. However, keep these discussions to high level concepts and not specific code implementations. Do not post more than a few lines of code.**

Although you could use a profiler to figure out the slowest part of your code and speed up just that part, that strategy has limited benefits here. It is very likely that your recipe.effect(state) function is responsible for the bulk of the time in your planner. Instead of making this function run faster, you should try to find ways to simply have this function be called fewer times by having the search explore fewer states. Heuristics and action pruning are key here.

Suppose you can tell, just by looking at a state, that the planner is exploring distracting territory. In the Minecraft domain, 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.

If you apply this same logic to non-tool items (considering the 'Produces' and 'Consumes' amounts), you should be able to reach all the way to the hardests tasks. However, getting the math right can be tricky if you are trying to automatically derive these limits from the recipes. It’s okay to hard code limit values if you can estimate reasonable values by inspecting the recipes yourself.

This same inventory-limiting idea could be implemented as an extra condition to check when seeing if a recipe is applicable (instead of a heuristic that sometimes returns infinity).

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.

Helper timing code:

from timeit import default\_timer as time

start = time() # Returns 0.0 and start the clock.

time\_elapsed = time() – start # This is in seconds.

Plutonium Test Problem

Here’s a tiny test domain you can use if you want to walk through every step of your planner. You might need to disable your Minecraft-specific heuristics and action pruning mechanisms.

Try running your planner on this domain:

{  
 "Items": ["nickel", "dime", "quarter", "plutonium"],  
 "Initial": {"plutonium": 1},  
 "Goal": {"quarter": 1, "plutonium": 1},  
 "Recipes": {  
   "find nickel": {  
     "Produces": {"nickel": 1},  
     "Time": 60  
   },  
   "trade plutonium for dime": {  
     "Consumes": {"plutonium": 1},  
     "Produces": {"dime": 1},  
     "Time": 1  
   },  
   "make change for dime": {  
     "Consumes": {"nickel": 2},  
     "Produces": {"dime": 1},  
     "Time": 5  
   },  
   "make change for quarter": {  
     "Consumes": {"nickel": 1, "dime": 2},  
     "Produces": {"quarter": 1},  
     "Time": 5  
   }  
 }  
}

It should produce a resulting plan description like this:

(60, 'find nickel', {'nickel': 1, 'plutonium': 1})  
(60, 'find nickel', {'nickel': 2, 'plutonium': 1})  
(5, 'make change for dime', {'plutonium': 1, 'dime': 1})  
(60, 'find nickel', {'nickel': 1, 'plutonium': 1, 'dime': 1})  
(60, 'find nickel', {'nickel': 2, 'plutonium': 1, 'dime': 1})  
(5, 'make change for dime', {'plutonium': 1, 'dime': 2})  
(60, 'find nickel', {'nickel': 1, 'plutonium': 1, 'dime': 2})  
(5, 'make change for quarter', {'quarter': 1, 'plutonium': 1})  
  
{'total\_cost': 315, 'length': 8}

Submission Instructions

Your code, when run, should output:

* 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

You should submit:

* File: Your code.
* Form: A short description of your search approach.
* Form: A short description of any heuristic you may have implemented.
* Form: The most difficult goal you’ve found that your implementation is able to find a solution for within 30 seconds. **File: Include a text file with the solution your code finds as well as the plan’s cost.**
* Form: A reported number of states your search visited for that goal. If you are keeping track of information, say costs, in a dictionary (e.g. costs[state] = 5), this is simply len(costs) at the end of the search.

[Submission Link](http://goo.gl/doPz0p)