P5: Mini-Minecraft Planner

Goal

Implement a state-space planner that operates in the domain of Minecraft-style item crafting rules. You may implement any planning algorithm you like (state progression, goal regression, or bidirectional search) using any heuristic scheme you like (plain dijkstra's / no heuristic, custom heuristic based on player knowledge, automatic heuristic based on a relaxation, etc.), so long as the time requirements below are satisfied.

Implementing a state progression (forward) planner with no heuristic (plain Dijkstra's algorithm) will get you off to an easy start.

Requirements / Grading Criteria

Each of the tasks 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 step length 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]
- Achieve {'wooden_pickaxe': 1} from scratch. [cost=18, len=9]
- Achieve {'stone_pickaxe': 1} from scratch. [cost=31, len=13]
- Achieve {'furnace': 1} from scratch. [cost=48, len=22]
- Achieve {'iron_pickaxe': 1} from scratch. [cost=83, len=33]
- Achieve {'cart': 1} from scratch. [cost=104, len=38]
- Achieve {'cart': 1, 'rail': 10} from scratch. [cost=172, len=58]
- Achieve {'cart': 1, 'rail': 20} from scratch. [cost=222, len=87]

During the demo in lab, be prepared to show us 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).

Base Code

There is no base code for this assignment, however, there is a data file (Crafting.json) supplied at the end of this document.

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}

# Dictionary of crafting recipes:
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)
```

Where are make_checker and make_effector defined? You have to do that as well:

```
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
```

To start off, just have the checker return True and the effector return the state it is given. You can fill these in later.

Step 2: Implement a Generic A* Search Algorithm

Implement a generic version of the A* algorithm 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 Fase
- **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: Test your Generic A*

Test your A* code on a tiny graph search problem:

```
t_initial = 'a'
t_limit = 20

edges = {'a': {'b':1,'c':10}, 'b':{'c':1}}

def t_graph(state):
     for next_state, cost in edges.items():
          yield ((state,next_state), next_state, cost)
```

```
def t_is_goal(state):
      return state == 'c'
```

```
def t_heuristic(state):
     return 0

print search(t_graph, t_initial, t_is_goal, t_limit, t_heuristic)
```

Step 4: Implement the Building Blocks of your Planner

You'll need to decide how you want to represent states. See the note in the **Caveats** sections below about *hashable* objects. We recommend a fixed-length tuple that records how many of each item are present in the current inventory.

```
def make_initial_state(inventory):
     ...
     return state

initial_state = make_initial_state(Crafting['Initial'])

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

is_goal = make_goal_checker(Crafting['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
```

Step 5: 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 6: 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.**

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?

Caveats

Hashable States

The keys of a Python dictionary need to be *hashable* objects. Dictionaries aren’t hashable, so you can’t use one as a state representation.

```
dist = {}

state_dict = {'coal': 5}
dist[state_dict] = 6
      TypeError: unhashable type: 'dict'

Items = Crafting['Items']

def inventory_to_tuple(d):
      return tuple(d.get(name,0) for i,name in enumerate(Items))

h = inventory_to_tuple(state_dict) # --> (0,0,0,0,5,0,0,0)

def inventory_to_set(d):
      return frozenset(d.items())

h = inventory_to_frozenset(state_dict) # --> frozenset({('coal':5)})

dist[h] = 6
      No error
```

Alternatively, you can implement your own State class, making sure to define the `__hash__(self)` and `__eq__(self,other)` methods in an appropriate way.

Crafting.json

```
{
  "Initial": {},
  "Goal": {
    "stone_pickaxe": 1
  },
```

```
  "Items": [
    "bench",
    "cart",
    "coal",
    "cobble",
    "furnace",
    "ingot",
    "iron_axe",
    "iron_pickaxe",
    "ore",
    "plank",
    "rail",
    "stick",
    "stone_axe",
    "stone_pickaxe",
    "wood",
    "wooden_axe",
    "wooden_pickaxe"
  ],
  "Recipes": {
    "craft wooden_pickaxe at bench": {
      "Produces": {
        "wooden_pickaxe": 1
      },
      "Requires": {
        "bench": true
      },
      "Consumes": {
        "plank": 3,
        "stick": 2
      },
      "Time": 1
    },
    "craft stone_pickaxe at bench": {
      "Produces": {
        "stone_pickaxe": 1
      },
      "Requires": {
        "bench": true
      },
      "Consumes": {
        "cobble": 3,
        "stick": 2
      },
      "Time": 1
    },
    "wooden_pickaxe for coal": {
      "Produces": {
        "coal": 1
      },
      "Requires": {
        "wooden_pickaxe": true
      },
      "Time": 4
    },
    "iron_pickaxe for ore": {
      "Produces": {
```

```
        "ore": 1
      },
      "Requires": {
        "iron_pickaxe": true
      },
      "Time": 2
    },
    "wooden_axe for wood": {
      "Produces": {
        "wood": 1
      },
      "Requires": {
        "wooden_axe": true
      },
      "Time": 2
    },
    "craft plank": {
      "Produces": {
        "plank": 4
      },
      "Consumes": {
        "wood": 1
      },
      "Time": 1
    },
    "craft stick": {
      "Produces": {
        "stick": 4
      },
      "Consumes": {
        "plank": 2
      },
      "Time": 1
    },
    "craft rail at bench": {
      "Produces": {
        "rail": 16
      },
      "Requires": {
        "bench": true
      },
      "Consumes": {
        "ingot": 6,
        "stick": 1
      },
      "Time": 1
    },
    "craft cart at bench": {
      "Produces": {
        "cart": 1
      },
      "Requires": {
        "bench": true
      },
      "Consumes": {
        "ingot": 5
      },
```

```
      "Time": 1
    },
    "iron_pickaxe for cobble": {
      "Produces": {
        "cobble": 1
      },
      "Requires": {
        "iron_pickaxe": true
      },
      "Time": 1
    },
    "stone_axe for wood": {
      "Produces": {
        "wood": 1
      },
      "Requires": {
        "stone_axe": true
      },
      "Time": 1
    },
    "craft iron_pickaxe at bench": {
      "Produces": {
        "iron_pickaxe": 1
      },
      "Requires": {
        "bench": true
      },
      "Consumes": {
        "ingot": 3,
        "stick": 2
      },
      "Time": 1
    },
    "craft furnace at bench": {
      "Produces": {
        "furnace": 1
      },
      "Requires": {
        "bench": true
      },
      "Consumes": {
        "cobble": 8
      },
      "Time": 1
    },
    "punch for wood": {
      "Produces": {
        "wood": 1
      },
      "Time": 4
    },
    "stone_pickaxe for ore": {
      "Produces": {
        "ore": 1
      },
      "Requires": {
        "stone_pickaxe": true
```

```
      },
      "Time": 4
    },
    "craft iron_axe at bench": {
      "Produces": {
        "iron_axe": 1
      },
      "Requires": {
        "bench": true
      },
      "Consumes": {
        "ingot": 3,
        "stick": 2
      },
      "Time": 1
    },
    "stone_pickaxe for coal": {
      "Produces": {
        "coal": 1
      },
      "Requires": {
        "stone_pickaxe": true
      },
      "Time": 2
    },
    "craft wooden_axe at bench": {
      "Produces": {
        "wooden_axe": 1
      },
      "Requires": {
        "bench": true
      },
      "Consumes": {
        "plank": 3,
        "stick": 2
      },
      "Time": 1
    },
    "stone_pickaxe for cobble": {
      "Produces": {
        "cobble": 1
      },
      "Requires": {
        "stone_pickaxe": true
      },
      "Time": 2
    },
    "wooden_pickaxe for cobble": {
      "Produces": {
        "cobble": 1
      },
      "Requires": {
        "wooden_pickaxe": true
      },
      "Time": 4
    },
    "iron_pickaxe for coal": {
```

```
      "Produces": {
        "coal": 1
      },
      "Requires": {
        "iron_pickaxe": true
      },
      "Time": 1
    },
    "craft bench": {
      "Produces": {
        "bench": 1
      },
      "Consumes": {
        "plank": 4
      },
      "Time": 1
    },
    "craft stone_axe at bench": {
      "Produces": {
        "stone_axe": 1
      },
      "Requires": {
        "bench": true
      },
      "Consumes": {
        "cobble": 3,
        "stick": 2
      },
      "Time": 1
    },
    "smelt ore in furnace": {
      "Produces": {
        "ingot": 1
      },
      "Requires": {
        "furnace": true
      },
      "Consumes": {
        "coal": 1,
        "ore": 1
      },
      "Time": 5
    },
    "iron_axe for wood": {
      "Produces": {
        "wood": 1
      },
      "Requires": {
        "iron_axe": true
      },
      "Time": 1
    }
  }
}
```