P4: HTN Planning for Minecraft

While graph search algorithms find solutions by sequentially trying operators, problem decomposition planners work by divide and conquer. Hierarchical Task Networks (HTNs) are an example. They represent planning problems as tasks to be performed vs states to achieve, and solve problems by decomposing tasks into subtasks, and subtasks into primitive operators that can be applied to change problem state. The HTN planner finds a solution by searching across what is called an AND/OR tree, consisting of the subtasks required to accomplish a task (the ANDs), and the alternate methods for accomplishing a task (the ORs).

For this assignment, you will use HTNs to construct a variety of artifacts in a minecraft-style planning domain. We provide a python implementation of HTNs, recipes for assembling elements from their component parts (stated as json scripts), and a list of tasks to solve specified via initial and desired final states. Your job will be to translate the json scripts into HTN operators and methods, to write heuristics that guide the HTN system's search through the AND/OR tree, and to write additional methods (ways to decompose a task) as required.

Note that an HTN planner typically finds a *satisficing* vs an *optimal* plan - it accepts the 1st solution found for accomplishing a task. We will ask you to find solutions for construction tasks that meet minecraft time bounds, and to look for incrementally better solutions, but we are not asking you to find solutions that minimize the quantity of resources required.

Operators

HTN operators represent primitive actions in the problem domain. For example, these are primitive operators for creating wood, planks, and a bench in a minecraft domain:

```
def op_punch_for_wood (state, ID):
    state.wood[ID] += 1
    return state

def op_craft_plank (state, ID):
    if state.wood[ID] >= 1:
        state.plank[ID] += 4
        state.wood[ID] -= 1
        return state
    return False
```

```
def op_craft_bench (state, ID):
    if and state.plank[ID] >= 4:
        state.bench[ID] += 1
        state.plank[ID] -= 4
        return state
    return False

pyhop.declare_operators (op_punch_for_wood, op_craft_plank, op_craft_bench)
```

Here, *state* is an object that contains the quantities of available resources, indexed by the ID of the agent doing the planning.

pyhop is an implementation of HTN planning in Python, which we will use in this assignment. You must declare operators to pyhop. In pyhop, <u>an operator returns the state or False</u>. False lets the planner know that this particular path down the AND/OR tree failed so that it will try the next thing.

Methods

A method defines how to decompose a task into a sequence of subtasks (the AND part of the search tree mentioned above).

Here is an example of a method for making a bench from component parts. It works by first establishing the operator's preconditions, then performing the actual action. Here, the only precondition of crafting a bench is to have 4 planks. *Have_enough* is a supplied method (see below).

```
def craft_bench (state, ID):
    return [('have_enough', ID, 'plank', 4), ('op_craft_bench', ID)]
```

An HTN typically includes many methods for performing a task (the OR part of the search tree mentioned above). The following example defines two methods for producing wood (using a wooden axe, and punching trees to make wood). Each individual method is defined by a python function, and the *pyhop.declare_methods* call associates alternative methods with the task name *produce_wood*.

```
def punch_for_wood (state, ID):
    return [('op_punch_for_wood', ID)]

def wooden_axe_for_wood (state, ID):
    return [('have_enough', ID, 'wooden_axe', 1),                    \
            ('op_wooden_axe_for_wood', ID)]
```

```
pyhop.declare_methods ('produce_wood', wooden_axe_for_wood, \
                                          punch_for_wood)
```

<u>A method returns a list of subtasks (possibly the empty list) or False</u>. False tells the planner that this particular path down the AND/OR tree failed, indicating that it should try the next thing.

Note: each call to declare_methods *overwrites* the previous definition of a given task - it does not add alternate methods.

Supplied Methods and Tasks:

We supply two methods for accomplishing a common subtask encountered in minecraft construction problems; having enough of a given item (see the file, manualHTN.py). The first method checks to see if enough of the item is already on hand, while the second produces some and then recursively calls have_enough.

```
def check_enough (state, ID, item, num):
   #  methods either fail or return a list of subtasks,
   # so returning an empty list indicates success

   if getattr(state,item)[ID] >= num: return []
   return False

def produce_enough (item, num, state, ID):
   return [ (‘produce’, item, state, ID), (‘have_enough’, num, state, ID)]

pyhop.declare_methods (‘have_enough’, check_enough, produce_enough)
```

This code can drive the planner to create an arbitrary amount of an item (if feasible), and it works for any item type. However, the subtask to ‘produce’ an item is generic, so we need a method that branches to tasks for producing specific item types:

```
def produce_m (item, state, ID):
  # case on the type of item
  if item = ‘wood’ return [(‘produce_wood’, state, ID)]
  if item = ‘plank’ return [(‘produce_plank’, state, ID)]
  # add other item types here
  …
  return False
```

```
pyhop.declare_methods('produce', produce_m)
```

You will need to complete this function in order to utilize the *have_enough* subtask for the purpose of producing other types of items.

Heuristics

In principle, you can employ heuristics to simplify any search task. In this assignment, you can use heuristics (domain knowledge) to manually order the search task by declaring operators and methods to pyhop in some order. We've also modified pyhop to let you filter out potentially unproductive tasks at run time.

The supplied version of pyhop calls the function *heuristic* with a list of methods for a task, and considers the list of methods that *heuristic* returns in order. Your implementation of *heuristic* must satisfy this interface:

```
def add_heuristic (data, ID):
    # do not change parameters to heuristic()
    def heuristic (state, curr_task, tasks, plan, depth, calling_stack):
        # your code here
        return False                    # if True, pyhop prunes this branch

    pyhop.add_check(heuristic)
```

Note that your implementation of *heuristic* will be called with a variety of context information (you are free to use or ignore any of his information):

state	the current problem state
curr_task	the task addressed by every method in *methods*
tasks	the list of tasks that follow curr_task in its sequential task list
plan	the currently accrued plan of operations
depth	the depth of curr_task in pyhop's search tree
calling-stack	the list of subtasks connecting the overall task to curr_task

Note that you may also store additional data in state to record information that is useful in subsequent calls to heuristic.

Hint: you can use the pruning capability of heuristic to keep the planner from engaging in an infinite regress, given that there are cycles in our recipes.

Supplied Files

crafting.json:

file containing construction recipes

manualHTN.py:

starter code for writing hard-coded HTN operators and methods

autoHTN.py:

starter code for writing programmatically generated HTN operators and methods

pyhop.py:

a modified version of the pyhop distribution available on the web

travel.py:

official examples of how to use pyhop; works with our modified pyhop.py

Requirements for this assignment

(1) Solve and submit a solution for this task

- Given {}, achieve {'wood': 12} [time <= 46]

Use the supplied subtask and methods for *have_enough* in the file **manualHTN.py** to complete this task. Edit manualHTN.py to include new methods and operators that implement recipes seen in the json scripts as necessary.

(2) Create HTN operators from the supplied json scripts

Edit declare_operator (in the file autoHTN.py) to go through the data from crafting.json and call make_operator on each recipe.

Hint: call make_operator, then declare the operator to pyhop using pyhop.declare_operators.

If you name your operators (you should), use op_ as a prefix for the name.

'make_operator'

```
def make_operator (rule):
    def operator (state, ID):
        # your code here
        pass
    return operator

def declare_operators (data):
    # your code here
    # hint: call make_operator, then declare the operator to pyhop using
    # pyhop.declare_operators(o1, o2, ..., ok)
    pass
```

(3) Create HTN methods from the supplied json scripts

Replace the stub for declare_methods (in the file autoHTN.py) with code that goes through the data from *crafting.json* and calls make_method on each recipe, and then declares the resulting methods to pyhop.

Be careful to order the subtasks in a recipe appropriately when you declare them to pyhop, as the planner will have trouble finding solutions for a given set of subtasks if it considers them in certain orders.

If you name your methods (you should), use *produce_* as a prefix for the name.

'make_method'

```
def make_method (name, rule):
    def method (state, ID):
        # your code here
        pass

    return method

def declare_methods (data):
    # your code here
    pass
```

(4) Create a mechanism to turn json problem descriptions into HTN problems by initializing initial resource state and goals (top level task).   We provide two functions (in autoHTN.py) that may help:

'set_up_state' initializes pyhop state from a json script
'set_up_goals' creates a top-level task from a json script

You can choose to use these functions or not, and you may edit them as necessary.

(5) Solve and submit solutions for these test cases. Your code must programmatically generate operators and methods using make_operator and make_method, as described in (2) and (3) above.

- Solve these test cases:
    a. Given {'plank': 1}, achieve {'plank': 1} [time <= 0]
    b. Given {}, achieve {'plank': 1} [time <= 300]
    c. Given {'plank': 3, 'stick': 2}, achieve {'wooden_pickaxe': 1} [time <= 10]
    d. Given {}, achieve {'iron_pickaxe': 1} [time <= 100]
    e. Given {}, achieve {'cart': 1, 'rail': 10} [time <= 175]
    f. Given {}, achieve {'cart': 1, 'rail': 20} [time <= 250]

You will need to write a heuristic that prevents pyhop from engaging in an infinite regression as it searches the problem space, as the minecraft domain contains circular dependencies (e.g., you need wood to make a pickaxe to make wood).

Also, your planner can fail when there is no feasible plan within the resource/time constraints, and/or when the search space is too large for your computational resources. Good ordering heuristics address the second, not the first. Every task in this list is solvable within 30 seconds of real time, so don't wait longer than that.

(6) (Extra Credit) Define the most complicated case your HTN planner can solve in 30 seconds of real-world time.

Submission

- manualHTN.py, which should solve case (1) from the section above.
- autoHTN.py, which should solve cases in (5) from the section above.
    - autoHTN.py must programmatically create methods and operators
- A README file that describes the heuristics you chose/programmed.
- (Optional Extra Credit) A file *custom_case.txt* that states your chosen problem for (6) in the format "Given x, achieve y" and the solution found by your HTN planner. You should identify the task, the solution, the time cost in recipe time and the time cost in real-world time.
    - You can build on provided test cases, but do not reuse one as is.