Search Exercise 5

A Robot Worker

In this exercise we consider a true AI type of situation. We have a robot that can help us by carrying items between the rooms of a building (e.g. a home or factory) and we want to give it the "intelligence" to put the items where we want them. We also want it do deal with some constraining factors, such as it having limited strength to carry heavy objects, and perhaps soem doors may be locked and a key will be needed to open them.

In this exercise we shall see how we can implement an AI capable of solving this kind of action planning problem by working out a sequence of actions that can *potentially* acheive any possibe goal. I say *potentially* because this kind of problem can get extremely computationally intensive if we have more than a small number of possible state variable values. These variable values would correspond to information such as possible robot locations, room contents, doors that could be open or locked etc.. The search algorithm may need to try thousands or millions (or more!) action combinations to find a successful action sequence and the number of possible sequences will grow exponentially as the number variables and possible values increases.

Seup `bbSearch`

As usual we start by downloading and importing from Brandon's search module:

```
!mkdir  -p  bbmodcache
!curl  http://bb-ai.net.s3.amazonaws.com/bb-python-modules/bbSearch.py  >  bbmodcache/bbSearch.py
from  bbmodcache.bbSearch  import  SearchProblem,  search
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 18767  100 18767    0     0  71904      0 --:--:-- --:--:-- --:--:-- 71904
Loading bbSearch Version 2.1 (at 12:48, Tue 01 Mar)
Last module source code edit 9am Thursday 24th Feb 2022
```

Object Oriented Specification of State

The code below defines some objects and dictionaries that are used to specify possible states of affairs that could occur as the robot carries out actions. The robot is in a certain location, can carry items and has a given strength. Doors connect rooms (e.g. in a factory) and can have keys and can be in a state of either being locked or unlocked. The initial contents of rooms and the weights of these items are specified. The complete `State` object includes the `Robot` object, a list of `Door` objects and a `ROOM_CONTENTS` dictionary.

A complication: need unique string representations of state for the loop checker

One thing that you will notice in the following code is that I have defined a `__repr__` (representation) method for each of the classes that I define. This gives a unique string representation for these objects. The purpose of this is to provide an identifier for each state that can be efficiently stored and used for loop checking when the `loop_check` options is set. (The `search` algorithm takes care of this automatically, storing the `__repr__(state)` for every state that gets put in the queue. If the same state gets generated again later, it is discarded, not put in the queue. Note that loop checking is often very useful but does have its own computational cost, which sometime outweights its benefit.)

Class definitions for `Robot`, `Door`

Classes for represnting these elements of the state are defined as follows:

```
class Robot:
    def __init__(self, location, carried_items, strength):
        self.location      = location
        self.carried_items = carried_items
        self.strength      = strength

    def weight_carried(self):
        return sum([ITEM_WEIGHT[i] for i in self.carried_items])

    ## Define unique string representation for the state of the robot object
    def __repr__(self):
        return str( ( self.location,
                      self.carried_items,
                      self.strength ) )

class Door:
    def __init__(self, roomA, roomB, doorkey=None, locked=False):
        self.goes_between = {roomA, roomB}
        self.doorkey      = doorkey
        self.locked       = locked
        # Define handy dictionary to get room on other side of a door
        self.other_loc = {roomA:roomB, roomB:roomA}

    ## Define a unique string representation for a door object
    def __repr__(self):
        return str( ("door", self.goes_between, self.doorkey, self.locked) )
```

Definition of the `State` class

We can now define a state for the Robot Worker problem as consisting of a robot object, a list of door objects and a `room_contents` dictionary storing the locations of named items.

```
class State:
```

```
    def __init__( self, robot, doors, room_contents ):
        self.robot = robot
        self.doors = doors
        self.room_contents = room_contents

    ## Define a string representation that will be uniquely identify the state.
    ## An easy way is to form a tuple of representations of the components of
    ## the state, then form a string from that:
    def __repr__(self):
        return str( ( self.robot.__repr__(),
                      [d.__repr__() for d in self.doors],
                      self.room_contents ) )
```

Specifying the Intial State of a Particular Scenario

To specify a particular problem situation we will need to specify room contents, item weights and doors. We use dictionaries for the contents and weights and create a list of `Door` objects, as follows:

```
ROOM_CONTENTS = {
    'workshop'          : {'rusty key'},
    'store room'        : {'bucket', 'suitcase'},
    'tool cupboard'     : {'sledge hammer', 'anvil', 'saw', 'screwdriver'},
}

ITEM_WEIGHT = {
          'rusty key'   : 0,
               'bucket' : 2,
            'suitcase'  : 4,
    'screwdriver'       : 1,
  'sledge hammer'       : 5,
                'anvil' : 12,
                  'saw' : 2,
}

DOORS = [
    Door('workshop', 'store room' ),
    Door( 'store room', 'tool cupboard', doorkey='rusty key', locked=False )
]
```

Defining the `RobotWorker` problem class

We now specify an extension of the `SearchProblem` class corresponding to a `RobotWorker` problem.

```
from copy import deepcopy

class RobotWorker( SearchProblem ):
```

```
    def __init__( self, state, goal_item_locations ):
        self.initial_state = state
        self.goal_item_locations = goal_item_locations

    def possible_actions( self, state ):

        robot_location = state.robot.location
        strength = state.robot.strength
        weight_carried = state.robot.weight_carried()

        actions = []
        # Can put down any carried item
        for i in state.robot.carried_items:
            actions.append( ("put down", i) )

        # Can pick up any item in room if strong enough
        for i in state.room_contents[robot_location]:
            if strength >= weight_carried + ITEM_WEIGHT[i]:
                actions.append( ("pick up", i))

        # If there is an unlocked door between robot location and
        # another location can move to that location
        for door in state.doors:
            if door.locked==False and robot_location in door.goes_between:
                actions.append( ("move to", door.other_loc[robot_location]) )

        # Now the actions list should contain all possible actions
        return actions

    def successor( self, state, action):
        next_state = deepcopy(state)
        act, target = action
        if act== "put down":
            next_state.robot.carried_items.remove(target)
            next_state.room_contents[state.robot.location].add(target)

        if act == "pick up":
            next_state.robot.carried_items.append(target)
            next_state.room_contents[state.robot.location].remove(target)

        if act == "move to":
            next_state.robot.location = target

        return next_state

    def goal_test(self, state):
        #print(state.room_contents)
        for room, contents in self.goal_item_locations.items():
            for i in contents:
                if not i in state.room_contents[room]:
                    return False
        return True

    def display_state(self,state):
```

```
        print("Robot location:", state.robot.location)
        print("Robot carrying:", state.robot.carried_items)
        print("Room contents:", state.room_contents)
```

```
rob = Robot('store room', [], 15 )

state = State(rob, DOORS, ROOM_CONTENTS)

goal_item_locations = {"store room":{"sledge hammer", "screwdriver", "anvil"}}

RW_PROBLEM_1 = RobotWorker( state, goal_item_locations )
```

Testing the Robot

Before trying to get the robot to do something useful, we should perhaps check that it seems to be functioning as we expect and won't do anything unexpected or dangerous. (You can't be too careful with robots!)

Let us check the possible actions from the initial state. We can simply apply the `possible_actions` method to the `initial_state` for our problem instance `RW_PROBLEM_1`. The following code will enable us to check what can happen:

```
poss_acts = RW_PROBLEM_1.possible_actions( RW_PROBLEM_1.initial_state )
poss_acts
```

```
[('pick up', 'suitcase'),
 ('pick up', 'bucket'),
 ('move to', 'workshop'),
 ('move to', 'tool cupboard')]
```

Well that seems reasonable. Does that seem sensible? You should check that these are indeed the actions that one would expect to be possible for the given initial situation.

But we should also check whether the result of carrying out the actions is what we expect. We can do this by using the `successor` function and see what state we get. We can use `display_state` to display this in a nice way. So the following loop will show us the next states after each of the possible actions:

```
for act in poss_acts:
    print("Action", act, "leads to the following state:")
    next_state = RW_PROBLEM_1.successor( RW_PROBLEM_1.initial_state, act )
    RW_PROBLEM_1.display_state(next_state)
    print()
```

```
Action ('pick up', 'suitcase') leads to the following state:
Robot location: store room
Robot carrying: ['suitcase']
Room contents: {'workshop': {'rusty key'}, 'store room': {'bucket'}, 'tool cupboard': {'sledg

Action ('pick up', 'bucket') leads to the following state:
Robot location: store room
Robot carrying: ['bucket']
Room contents: {'workshop': {'rusty key'}, 'store room': {'suitcase'}, 'tool cupboard': {'sle

Action ('move to', 'workshop') leads to the following state:
Robot location: workshop
Robot carrying: []
Room contents: {'workshop': {'rusty key'}, 'store room': {'suitcase', 'bucket'}, 'tool cupboa

Action ('move to', 'tool cupboard') leads to the following state:
Robot location: tool cupboard
Robot carrying: []
Room contents: {'workshop': {'rusty key'}, 'store room': {'suitcase', 'bucket'}, 'tool cupboa
```

Does that look correct?

Shall we put the robot to work?

The tests on what the robot could do starting from the initial state appear to have gone very well. Nobody got killed and the robot is not showing any tendany to take over the world (so far).

Your boss is nagging you about putting the sledge hammer, screwdriver and anvil away in the store room. What a chore --- the anvil weighs a ton!). Maybe the robot could help? Let's enter the command and press go!

```
search(  RW_PROBLEM_1,  'BF/FIFO',  100000,  loop_check=True)
```

```
This is the general SearchProblem parent class
You must extend this class to encode a particular search problem.

** Running Brandon's Search Algorithm **
Strategy: mode=BF/FIFO, cost=None, heuristic=None
Max search nodes: 100000  (max number added to queue)
Searching (will output '.' each 1000 goal_tests)
.................
:-)) *SUCCESS* ((-:

Path length = 10
Goal state is:
Robot location: store room
Robot carrying: []
Room contents: {'workshop': {'rusty key'}, 'store room': {'bucket', 'sledge hammer', 'anvil',
The action path to the solution is:
    ('move to', 'tool cupboard')
    ('pick up', 'sledge hammer')
    ('pick up', 'screwdriver')
```

What Next?

Well looks like the robot worker can do some useful things at least in a simple situation.

But could it work in a more complex situation or achieve more complex goals?

Are there some heuristics that could enable it to effectively find solutions to acheive complex tasks?

```
Total nodes generated          =    86473  (includes start)

Nodes left in queue            =    11900

Time taken = 11.2806 seconds

'GOAL STATE FOUND'
```