Thomas Mulvey

Assignment 1 (resubmitted in pdf format)

1. Answers
2. Sample output
3. Code

notes

1) BFS is super slow and takes 60-120 seconds...just wait, it will end.

2) the program is python3, not 2. use "python3 mulvey_hw1.py" to run

---Requirements---

1) Represent a state:

- My class NodeState() at the top does this. Keeps track of cannibals/missionaries on the left and right side of the river, position of the boat, and the cost so far(for A*). The constructor will take (cannibals_left, missionaries_left, "BOAT_SIDE", cannibals_right, missionaries_right)

2) Represent an action:

- Since there are no functions needed for actions, I represented it as a tuple. (#cannibals to move, # missionaries to move, direction). If the boat was on the RIGHT then the only valid options for actions were ones that moved LEFT.

3) Function to check if next state is valid.

- is_valid_move(self, action) in NodeState() class does this. I send an action to the parent and determine if its valid. If it is valid, i make a new NodeState and set the parent as the one i just verified vs. I could have created a simpler is_valid function if I created a node using the desired action, but i didn't know if that was ok.

4) Node in the search tree:

- Similiar to #1, In the class I have a parent variable and a cost so far. A PQ is made using the cost so far/depth of tree + estimation using heuristic and then the node. Thus the PQ looks something like

[(0, Node1), (1, Node2), (1, Node3), (2, Node4), (3, Node 4), ...]

The Nodes are the NodeState Class, which hold the parent, cost so far, current setup, etc. Due to the lack of a robust PQ class in python, and not necessarily wanting to waste time making one, i decided to represent the nodes in PQ like that. The A* algorithm will take in count for taking top of PQ, estimating children costs and adding them, etc. The BFS doesnt do any of this, it’s just a slow uninformed search.

A) Implement an uninformed search algorithm. Nodes visited? Optimal ?

- implement uninform search : BFS
- Nodes visited : 11878
- Optimal? : Yes, since its same cost. Just visits way too many nodes to find it...

B) Propose ADMISSIBLE HEURISTIC. Justify it. Implement it. Nodes Visited? Optimal? Compare vs BFS.

- proposition: relax/ignore rule that cannibals will eat missionaries. Also assume the boat is on the left. Count how many trips it will take by totaling the # of people on the left, and the boat can bring 2 at once.
- Admissible? :Yes. This will underestimate the # of actual trips, or stay equal to it. The Max value it will guess for everyone on the left side is 6. Shortest path is 12 total visited nodes. If 5 people are on the left, it will guess 3. Real value is about 9. It will only guess values 1-6. It can underestimate by a lot or get it equal if there are 2 or less people on the left.
- Implement it: **A_star** function does this.
- Nodes visited : **3344**
- Optimal? : **Yes** (all actions are same cost)
- Comparison: BFS and A* will both find optimal paths, due to the fact each action is costs the same amount.

The difference comes to nodes visited / time computing. The heuristic and PQ of the A* will push pointless nodes to the end, and will visit much less nodes. Searching less nodes will take less time. BFS visited around 8k more nodes, and with a better heuristic, could be even more. A* in this case will take about 20x shorter as well, (with my testing)

```
tom@tmulvey-LT0:/mnt/c/Users/tmulvey/Documents/skool/cs_ai/hw1$ python3 mulvey_hw1.py
---BFS---
cL,mL,boat,cR,mR
(3,3,LEFT,0,0)
(1,3,RIGHT,2,0)
(2,3,LEFT,1,0)
(0,3,RIGHT,3,0)
(1,3,LEFT,2,0)
(1,1,RIGHT,2,2)
(2,2,LEFT,1,1)
(2,0,RIGHT,1,3)
(3,0,LEFT,0,3)
(1,0,RIGHT,2,3)
(2,0,LEFT,1,3)
(0,0,RIGHT,3,3)
(took  73.23894357681274  seconds) and visited  11878  nodes!

---A*---
cL,mL,boat,cR,mR
(3,3,LEFT,0,0)
(1,3,RIGHT,2,0)
(2,3,LEFT,1,0)
(0,3,RIGHT,3,0)
(1,3,LEFT,2,0)
(1,1,RIGHT,2,2)
(2,2,LEFT,1,1)
(2,0,RIGHT,1,3)
(3,0,LEFT,0,3)
(1,0,RIGHT,2,3)
(1,1,LEFT,2,2)
(0,0,RIGHT,3,3)
(took  3.9591434001922607  seconds) and visited  3344  nodes!
```

```python
# purpose : solve cannibals and missionaries problem using
#           a) some uninformed search alg
#           and b) make a heuristic (admissible) and implement a*
# author : tom mulvey
# date : 2.20.19
# vers: 1.2
#-----------------#

from queue import PriorityQueue
import time
import math

class NodeState():  #representatoin of nodes/states in game
    '''
    cannibals and missionaries game will require:
      c_left : # of cannibals on left side of river
      m_left : # of missionaires on left side of river
      boat : which side of river boat is on (LEFT or RIGHT)
      c_right : # of cannibals on right side of river
      m_right : # of missionaires on right side of river
    '''
    def __init__(self, c_left=3, m_left=3, boat="LEFT", c_right=0, m_right=0):
        # default game state is 3 missionaries and cannibals on left side (With boat)
        self.c_left = c_left
        self.m_left = m_left
        self.boat = boat
        self.c_right = c_right
        self.m_right = m_right

        self.parent = None
        self.cost = 0

    '''
    comparator for PQ. compare cost/depths
    '''
    def __lt__(self, other):
        return self.cost < other.cost

    '''
    game doesnt allow for cannibals to outnumber the missoinaries
      need to cross river with either 1 or 2 persons such that all missoinaries
survive.
    '''
    def is_valid_move(self, action):
        '''
        action is a tuple like this :
          (0-2, 0-2, DIRECTION)
          first index is how many cannibals will be sent over, second are how many
miss. and third is what dir

          c_left/right and m_left/right are the calculated values after action is
applied, and used to determine if move valid
        '''
        c_left = c_right = m_left = m_right = 0 #declaring vars, i dont think this is
necessary.
```

```
        if ( len(action)==3 and ( action[2].upper()=="LEFT" or
action[2].upper()=="RIGHT" ) ):
            if(self.boat.upper()=="RIGHT"):
                #print("moving left")
                c_left = self.c_left + action[0]
                m_left = self.m_left + action[1]
                c_right = self.c_right - action[0]
                m_right = self.m_right - action[1]
            else:
                #print("moving right")
                c_left = self.c_left - action[0]
                m_left = self.m_left - action[1]
                c_right = self.c_right + action[0]
                m_right = self.m_right + action[1]
        else:
            return False

        if (m_left >=0 and m_right >=0 and c_left >=0 and c_right >=0) : #no negativ
numbers
            if ( (m_left >= c_left or m_left==0) and (m_right>=c_right or m_right==0)
):
                return True
            else:
                return False
        else:
            return False


    '''
    game end state is all missionries and cannibals on right side of river
    '''
    def check_goal_state(self):
        if ( self.c_left == 0 ) and ( self.m_left == 0 ):
            return True
        else:
            return False


'''
  given a node, generate the VALID child nodes!

  for each side of river, we have 5 possibilites:
    move 2 cannibals, 2 missionaries, one of each, only 1 missionary, only 1
cannibal.

  This function will return all possible children/successors
'''
def create_actions(node):
    child_nodes = list()

    if node.boat == "LEFT": #will move left to right
        # 1 cannibal
        action = (1,0, node.boat)
        if (node.is_valid_move(action)):
```

```
            new = NodeState(node.c_left-1, node.m_left, 'RIGHT', node.c_right+1, 
node.m_right)
            new.parent = node
            new.cost = 1+node.cost
            child_nodes.append( new )
        # 1 missionary
        action = (0,1, node.boat)
        if (node.is_valid_move(action)):
            new = NodeState(node.c_left, node.m_left-1, 'RIGHT', node.c_right, 
node.m_right+1)
            new.parent = node
            new.cost = 1+node.cost
            child_nodes.append( new )
        # 2 cannibal
        action = (2,0, node.boat)
        if (node.is_valid_move(action)):
            new = NodeState(node.c_left-2, node.m_left, 'RIGHT', node.c_right+2, 
node.m_right)
            new.parent = node
            new.cost = 1+node.cost
            child_nodes.append( new )
        # 2 missionary
        action = (0,2, node.boat)
        if (node.is_valid_move(action)):
            new = NodeState(node.c_left, node.m_left-2, 'RIGHT', node.c_right, 
node.m_right+2)
            new.parent = node
            new.cost = 1+node.cost
            child_nodes.append( new )
        # 1 each
        action = (1,1, node.boat)
        if (node.is_valid_move(action)):
            new = NodeState(node.c_left-1, node.m_left-1, 'RIGHT', node.c_right+1, 
node.m_right+1)
            new.parent = node
            new.cost = 1+node.cost
            child_nodes.append( new )
    else: # moves right to left
        # 1 cannibal
        action = (1,0, node.boat)
        if (node.is_valid_move(action)):
            new = NodeState(node.c_left+1, node.m_left, 'LEFT', node.c_right-1, 
node.m_right)
            new.parent = node
            new.cost = 1+node.cost
            child_nodes.append( new )
        # 1 missionary
        action = (0,1, node.boat)
        if (node.is_valid_move(action)):
            new = NodeState(node.c_left, node.m_left+1, 'LEFT', node.c_right, 
node.m_right-1)
            new.parent = node
            new.cost = 1+node.cost
            child_nodes.append( new )
        # 2 cannibal
```

```
        action = (2,0, node.boat)
        if (node.is_valid_move(action)):
            new = NodeState(node.c_left+2, node.m_left, 'LEFT', node.c_right-2,
node.m_right)
            new.parent = node
            new.cost = 1+node.cost
            child_nodes.append( new )
        # 2 missionary
        action = (0,2, node.boat)
        if (node.is_valid_move(action)):
            new = NodeState(node.c_left, node.m_left+2, 'LEFT', node.c_right,
node.m_right-2)
            new.parent = node
            new.cost = 1+node.cost
            child_nodes.append( new )
        # 1 each
        action = (1,1, node.boat)
        if (node.is_valid_move(action)):
            new = NodeState(node.c_left+1, node.m_left+1, 'LEFT', node.c_right-1,
node.m_right-1)
            new.parent = node
            new.cost = 1+node.cost
            child_nodes.append( new )

    return child_nodes

# it is pointless to represent an ACTION as a class, it wont have any methods
# an example ACTION will be (int, int, string)
#  where the first int is how many cannibals will be sent over,
#  the second int is how many missionaries will be sent over,
#  and the string is directoin where the boat will go
#
# So given the initial NodeState(3,3,LEFT,0,0):
#  the possible actions are :
#    (2, 0, RIGHT), (1, 1, RIGHT), (0, 2, RIGHT)

# BFS search.
bfs_nodes_visited = 0
a_star_nodes_visited = 0
def BFS():
    global bfs_nodes_visited
    start_start=NodeState(3,3,'LEFT',0,0)

    if start_start.check_goal_state():
        return start_start

    children_to_visit = list() #acts as queue
    visited = set()

    children_to_visit.append(start_start)
    bfs_nodes_visited += 1
    while len(children_to_visit) != 0 :
        cur = children_to_visit.pop(0)
        if cur.check_goal_state() :
            return cur
```

```python
        visited.add(cur)
        bfs_nodes_visited += 1
        children = create_actions(cur)
        for kids in children :
            if (kids not in children_to_visit) or (kids not in visited):
                children_to_visit.append(kids)

    return None

'''
heuristic, h(n) gives the current ADMISSABLE estimate for a given node's state
To relax the problem, we are ignoring the sde of the boat, assuming its on the left
  and we are assuming the cannibals wont be eating the missionaries. The boat will
also drive it
  self back. h(n) returns the ceil of (c_left+m_left)/2. Lowest val is 1, max is 6
'''
def h(State):
    tot_people = State.m_left + State.c_left
    return (math.ceil(tot_people/2))


'''
we need a method to determine if an obj is in the PQ, since that member func. doesnt
exist...
  contents of PriorityQueue are kept in a queue member, which is a plain list.
  so we can use 'in' to check
'''
def is_in_queue(q, goal):
    with q.mutex:
        return goal in q.queue

'''A_star() will implement the heuristic h(n)
  steps :
    1. have a PQ sorted from cost so far (depth) + h(node)
    2. pop top, add to a visited set, genereate successors, if top isnt goal state
    3. cont till top of PQ is end state
'''
def A_star():
    global a_star_nodes_visited
    start_start=NodeState(3,3,'LEFT',0,0)

    if start_start.check_goal_state():
        return start_start

    pq = PriorityQueue()
    # PQ work as (cost, object), so the queue will look like
    # [ 0=>(3,3,LEFT,0,0) , 1=>Child1, 1=>Child2, ... ]
    # cost = cost_so_far (depth) + h(node)
    visited = set()
    a_star_nodes_visited += 1
    pq.put( (0, start_start) )
    while pq.empty() == False :
        cur = pq.get()
        if cur[1].check_goal_state() :
            return cur[1]
```

```
        visited.add(cur)
        a_star_nodes_visited += 1
        children = create_actions(cur[1])
        for kids in children :
            cost = kids.cost + h(kids)
            if ( is_in_queue(pq, (cost, kids)) ) or (kids not in visited):
                pq.put( (cost, kids) )

def print_solution(solution):
        path = []
        path.append(solution)
        parent = solution.parent

        while parent is not None :
            path.append(parent)
            parent = parent.parent

        for t in range(len(path)):
            state = path[len(path) - t - 1]
            print( "(" + str(state.c_left) + "," + str(state.m_left) + "," +
state.boat + "," + str(state.c_right) + "," + str(state.m_right) + ")")


def main():
    global bfs_nodes_visited
    global a_star_nodes_visited
    print("---BFS---")
    bfs_start = time.time()
    sol = BFS()
    bfs_end = time.time()
    print("cL,mL,boat,cR,mR")
    print_solution(sol)
    print("(took ", bfs_end - bfs_start, " seconds) and visited ", bfs_nodes_visited,
" nodes!")

    print("\n---A*---")
    a_start = time.time()
    sol = A_star()
    a_end = time.time()
    print("cL,mL,boat,cR,mR")
    print_solution(sol)
    print("(took ", a_end - a_start, " seconds) and visited ", a_star_nodes_visited,
" nodes!")

#-------#
main()
```