CS 461 - Homework 1

Group Members

1. Ayhan Okuyan
2. Barış Akçin (Uploader)
3. Berkan Özdamar
4. Deniz Doğanay
5. Mustafa Bay

Introduction

The aim of this homework is to work on the Missionaries and Cannibals Problem for various cases in order to observe if there are 4 missionaries and 4 cannibals, moving them to other side of the river is not possible. We used python programming language with .IPYNB format.

Work Done

We started with manually solving Missionaries and Cannibals Problem for 3 cannibals and 3 missionaries. Since it is challenging, we spend some time on how to solve this problem. After we solved it manually, we decided to use Breadth First Search method rather using Depth First Search. The reason behind choosing Breadth First Search is it gives us an opportunity to check the possibilities in each depth in order to understand if there is a missing term in our code.

To conduct a breadth-first search,

- Form a one-element queue consisting of a zero-length path that contains only the root node.
- Until the first path in the queue terminates at the goal node or the queue is empty,
  - Remove the first path from the queue; create new paths by extending the first path to all the neighbors of the terminal node.
  - Reject all new paths with loops.
  - Add the new paths, if any, to the *back* of the queue.
- If the goal node is found, announce success; otherwise, announce failure.

Image 1: Pseudocode for Bread-First Search

Since we have solved the problem for 3 cannibals and 3 missionaries manually and understand the pseudocode of Bread First Search, we need to implement a program which is the combination of understanding the problem and the Breadth First Search method.

We first implementing the code for 3 cannibals and 3 missionaries in order to see the problems that challenged us during implementation of the program. The results we obtained for 3 cannibals and 3 missionaries are given below;

```
Depth: 0
(3, 3, 1) , None
Depth: 1
(3, 2, -1) , (3, 3, 1) , None
(3, 1, -1) , (3, 3, 1) , None
(2, 2, -1) , (3, 3, 1) , None
Depth: 2
(3, 2, 1) , (3, 1, -1) , (3, 3, 1) , None
(3, 2, 1) , (2, 2, -1) , (3, 3, 1) , None
Depth: 3
(3, 0, -1) , (3, 2, 1) , (3, 1, -1) , (3, 3, 1) , None
(2, 2, -1) , (3, 2, 1) , (3, 1, -1) , (3, 3, 1) , None
(3, 1, -1) , (3, 2, 1) , (2, 2, -1) , (3, 3, 1) , None
(3, 0, -1) , (3, 2, 1) , (2, 2, -1) , (3, 3, 1) , None
Depth: 4
(3, 1, 1) , (3, 0, -1) , (3, 2, 1) , (3, 1, -1) , (3, 3, 1) , None
(3, 1, 1) , (3, 0, -1) , (3, 2, 1) , (2, 2, -1) , (3, 3, 1) , None
Depth: 5
(1, 1, -1) , (3, 1, 1) , (3, 0, -1) , (3, 2, 1) , (3, 1, -1) , (3, 3, 1) , None
(1, 1, -1) , (3, 1, 1) , (3, 0, -1) , (3, 2, 1) , (2, 2, -1) , (3, 3, 1) , None
Depth: 6
(2, 2, 1) , (1, 1, -1) , (3, 1, 1) , (3, 0, -1) , (3, 2, 1) , (3, 1, -1) , (3, 3, 1) , None
(2, 2, 1) , (1, 1, -1) , (3, 1, 1) , (3, 0, -1) , (3, 2, 1) , (2, 2, -1) , (3, 3, 1) , None
Depth: 7
(0, 2, -1) , (2, 2, 1) , (1, 1, -1) , (3, 1, 1) , (3, 0, -1) , (3, 2, 1) , (3, 1, -1) , (3, 3, 1) , None
(0, 2, -1) , (2, 2, 1) , (1, 1, -1) , (3, 1, 1) , (3, 0, -1) , (3, 2, 1) , (2, 2, -1) , (3, 3, 1) , None
Depth: 8
(0, 3, 1) , (0, 2, -1) , (2, 2, 1) , (1, 1, -1) , (3, 1, 1) , (3, 0, -1) , (3, 2, 1) , (3, 1, -1) , (3, 3, 1) , None
(0, 3, 1) , (0, 2, -1) , (2, 2, 1) , (1, 1, -1) , (3, 1, 1) , (3, 0, -1) , (3, 2, 1) , (2, 2, -1) , (3, 3, 1) , None
Depth: 9
(0, 1, -1) , (0, 3, 1) , (0, 2, -1) , (2, 2, 1) , (1, 1, -1) , (3, 1, 1) , (3, 0, -1) , (3, 2, 1) , (3, 1, -1) , (3, 3, 1) , No
ne
(0, 1, -1) , (0, 3, 1) , (0, 2, -1) , (2, 2, 1) , (1, 1, -1) , (3, 1, 1) , (3, 0, -1) , (3, 2, 1) , (2, 2, -1) , (3, 3, 1) , No
ne
Depth: 10
(0, 2, 1) , (0, 1, -1) , (0, 3, 1) , (0, 2, -1) , (2, 2, 1) , (1, 1, -1) , (3, 1, 1) , (3, 0, -1) , (3, 2, 1) , (3, 1, -1) ,
(3, 3, 1) , None
(1, 1, 1) , (0, 1, -1) , (0, 3, 1) , (0, 2, -1) , (2, 2, 1) , (1, 1, -1) , (3, 1, 1) , (3, 0, -1) , (3, 2, 1) , (3, 1, -1) ,
(3, 3, 1) , None
(0, 2, 1) , (0, 1, -1) , (0, 3, 1) , (0, 2, -1) , (2, 2, 1) , (1, 1, -1) , (3, 1, 1) , (3, 0, -1) , (3, 2, 1) , (2, 2, -1) ,
(3, 3, 1) , None
(1, 1, 1) , (0, 1, -1) , (0, 3, 1) , (0, 2, -1) , (2, 2, 1) , (1, 1, -1) , (3, 1, 1) , (3, 0, -1) , (3, 2, 1) , (2, 2, -1) ,
(3, 3, 1) , None
Depth: 11
(0, 0, -1) , (0, 2, 1) , (0, 1, -1) , (0, 3, 1) , (0, 2, -1) , (2, 2, 1) , (1, 1, -1) , (3, 1, 1) , (3, 0, -1) , (3, 2, 1) ,
(3, 1, -1) , (3, 3, 1) , None
```

Image 2: Breadth First Search for 3 Cannibals and 3 Missionaries

```
solution: 11 steps
0 missionaries and 2 cannibals go to the east side. State: [3, 1, -1]
0 missionaries and 1 cannibals come back to the west side. State: [3, 2, 1]
0 missionaries and 2 cannibals go to the east side. State: [3, 0, -1]
0 missionaries and 1 cannibals come back to the west side. State: [3, 1, 1]
2 missionaries and 0 cannibals go to the east side. State: [1, 1, -1]
1 missionaries and 1 cannibals come back to the west side. State: [2, 2, 1]
2 missionaries and 0 cannibals go to the east side. State: [0, 2, -1]
0 missionaries and 1 cannibals come back to the west side. State: [0, 3, 1]
0 missionaries and 2 cannibals go to the east side. State: [0, 1, -1]
0 missionaries and 1 cannibals come back to the west side. State: [0, 2, 1]
0 missionaries and 2 cannibals go to the east side. State: [0, 0, -1]
```

Image 3: Solution Converted in Easy to Read Version

As it can be seen from Image 2, we applied the breadth first search algorithm, we remove the first path from the queue, find the new paths according to the path we removed, add all the possible paths to the beginning of the queue and then remove the loops and moves that we cannot do (e.g. more cannibals then missionaries in one side). Then, repeat the process again.

The representation in Image 2, (3, 3, 1), shows our understanding from the question. The first term represents the number of missionaries in the original position (west side in our case). Second term represents the number of cannibals in the original position and third term represents the position of the boat (1 represents boat is in the west side, -1 represents boat is in the east side). While we are moving missionarie(s) and/or cannibal(s), we check if there is a loop made by new paths and if there is a prohibited move that we are making. We removed these paths from the queue.

The initial state is (3, 3, 1) in our case and our goal state is (0, 0, ±1). Our solution according to the Breadth First Search is given in the Image 3.

We implemented the code by adding a variable PEOPLE_COEFF which is 3 or 4. If it is 3, it means that the program is working for 3 missionaries and 3 cannibals. In order to convert the the program for 4 cannibals and 4 missionaries, we need to change PEOPLE_COEFF to 4. When we run the program for 4 missionaries and 4 cannibals, we obtain the result below;

```
Depth: 0
(4, 4, 1) , None
Depth: 1
(4, 3, -1) , (4, 4, 1) , None
(4, 2, -1) , (4, 4, 1) , None
(3, 3, -1) , (4, 4, 1) , None
Depth: 2
(4, 3, 1) , (4, 2, -1) , (4, 4, 1) , None
(4, 3, 1) , (3, 3, -1) , (4, 4, 1) , None
Depth: 3
(4, 1, -1) , (4, 3, 1) , (4, 2, -1) , (4, 4, 1) , None
(3, 3, -1) , (4, 3, 1) , (4, 2, -1) , (4, 4, 1) , None
(4, 2, -1) , (4, 3, 1) , (3, 3, -1) , (4, 4, 1) , None
(4, 1, -1) , (4, 3, 1) , (3, 3, -1) , (4, 4, 1) , None
Depth: 4
(4, 2, 1) , (4, 1, -1) , (4, 3, 1) , (4, 2, -1) , (4, 4, 1) , None
(4, 2, 1) , (4, 1, -1) , (4, 3, 1) , (3, 3, -1) , (4, 4, 1) , None
Depth: 5
(4, 0, -1) , (4, 2, 1) , (4, 1, -1) , (4, 3, 1) , (4, 2, -1) , (4, 4, 1) , None
(2, 2, -1) , (4, 2, 1) , (4, 1, -1) , (4, 3, 1) , (4, 2, -1) , (4, 4, 1) , None
(4, 0, -1) , (4, 2, 1) , (4, 1, -1) , (4, 3, 1) , (3, 3, -1) , (4, 4, 1) , None
(2, 2, -1) , (4, 2, 1) , (4, 1, -1) , (4, 3, 1) , (3, 3, -1) , (4, 4, 1) , None
Depth: 6
(4, 1, 1) , (4, 0, -1) , (4, 2, 1) , (4, 1, -1) , (4, 3, 1) , (4, 2, -1) , (4, 4, 1) , None
(3, 3, 1) , (2, 2, -1) , (4, 2, 1) , (4, 1, -1) , (4, 3, 1) , (4, 2, -1) , (4, 4, 1) , None
(4, 1, 1) , (4, 0, -1) , (4, 2, 1) , (4, 1, -1) , (4, 3, 1) , (3, 3, -1) , (4, 4, 1) , None
(3, 3, 1) , (2, 2, -1) , (4, 2, 1) , (4, 1, -1) , (4, 3, 1) , (3, 3, -1) , (4, 4, 1) , None
No solution for this start node is present in the graph.
```

Image 4: No Solution for 4 cannibals and 4 missionaries

We expect to have no solution as output and we obtained it as it can be seen from Image 4. All the possible paths are either loops or prohibited moves mentioned in the description of the Missionaries and Cannibals Problem. Our initial state in this case is (4, 4, 1) and our goal state is (0, 0, ±1), but as it can be seen from Image 4, we cannot obtain the goal state for the conditions in the problem. We checked all possible paths by using Breadth First Search method, but we cannot reach to the goal state. Thus, we proved that no solution exists for 4 cannibals and 4 missionaries problem.

Conclusion

We first started to solve Missionaries and Cannibals Problem manually in order to understand the logic behind the problem. Then, we analyzed the Breadth First Search method to implement a program which will solve the problem for 3 missionaries and 3 cannibals in order to understand and apply the solution of the problem by implementing a program. Then, we modified the program in order to check if a solution exists for 4 missionaries and 4 cannibals. We observed that for 4 missionaries and 4 cannibals, all possible paths are not approcahing to goal state whether they are in a loop or prohibited move according to the rules of the game. We find all the possible paths and proved that there is no solution by trying all possibilities.

The Code in Python Language

```
from queue import Queue

PEOPLE_COEFF = 4 #put 3 to see the valid solution and put 4 to see the no solution

#1 for this side, -1 for across side
#This class is used to hold the states that are configured through
#tree traversal
class State(object):
    def __init__(self,  miss, cann, boat, parent_node):
        self.miss = miss
        self.cann = cann
        self.boat = boat

        self.parent_node = parent_node

    def children(self):
        if self.boat == 1:
            for m in range(PEOPLE_COEFF):
                for c in range(PEOPLE_COEFF):
                    newState = State(self.miss - m, self.cann - c, -1, self);
                    if m + c >= 1 and m + c <= 2 and newState.isValid() and
newState.isUnique():
                        yield newState

        else:
            for m in range(PEOPLE_COEFF):
                for c in range(PEOPLE_COEFF):
                    newState = State(self.miss + m, self.cann + c, 1, self);
                    if m + c >= 1 and m + c <= 2 and newState.isValid() and
newState.isUnique():
                        yield newState

    def getStateVector(self):
        return [self.miss,self.cann,self.boat]

    def getDepth(self):
        node = self
        depth = 0
        while node.parent_node is not None:
            depth += 1
            node = node.parent_node
        return depth

    def isGoal(self):
        if(self.miss == 0 and self.cann == 0 and self.boat == -1):
            return True
```

```
        return False

    def isUnique(self):
        node = self
        prNode = node.parent_node
        while(prNode is not None):
            if(node.getStateVector() == prNode.getStateVector()):
                return False
            prNode = prNode.parent_node
        return True

    def isValid(self):
        if self.miss < 0 or self.cann < 0 or self.miss > PEOPLE_COEFF or self.cann
> PEOPLE_COEFF  or (self.boat != -1 and self.boat != 1):
            return False
        if (self.cann > self.miss) and (self.miss > 0):
            return False
        if (self.cann < self.miss) and (self.miss < PEOPLE_COEFF):
            return False
        return True

    def path(self):
        solution = []
        node = self
        while node.parent_node is not None:
            node = node.parent_node
            solution.append(node.getStateVector)
        solution.reverse()
        return solution

    #Here we modify the repr function of object class to recursively print the
path from the beginning to the end
    def __repr__(self):
        return "(%d, %d, %d) , %r" % (self.miss, self.cann, self.boat,
self.parent_node)

def printPath(finalNode):
    pathList = []
    node = finalNode
    while node.parent_node != None:
        pathList.append(node.getStateVector())
        node = node.parent_node
    pathList.append(node.getStateVector())
    pathList.reverse()
    print("solution: " + str(len(pathList)-1) + " steps")
    for i in range(len(pathList)-1):
        state1 = pathList[i]
        state2 = pathList[i+1]
        if(state1[2] == 1):
            #karşıya
            stateStr = '%d missionaries and %d cannibals go to the east side.' %
(state1[0] - state2[0], state1[1] - state2[1])
        else:
            #karşıdan
            stateStr = '%d missionaries and %d cannibals come back to the west
side.' % (state2[0] - state1[0], state2[1] - state1[1])
        print(stateStr +  ' State: ' + str(state2))

def bfs(root):
```

```
    queue = Queue()
    queue.put(root)
    cur_depth = -1
    while True:
        #not solvable
        if(queue.empty()):
           return None

        curNode = queue.get()

        if(curNode.getDepth() > cur_depth):
            cur_depth = curNode.getDepth()
            print('Depth:', cur_depth)

        print(curNode)
        if(curNode.isGoal()):
            printPath(curNode)
            return curNode.path()

        children = []
        for child in curNode.children():
            queue.put(child)

initial_state = State(PEOPLE_COEFF, PEOPLE_COEFF, 1, None)
solution = bfs(initial_state)
if solution is None:
    print("No solution for this start node is present in the graph.")
```