

BFS AND DFS

Model a given problem in terms of state space search problem and solve the same using BFS/ DFS

Ankur Singh
MTECH CSE (2020-22)
IIIT Vadodara
Gandhinagar, India
202061001@iiitvadodara.ac.in

Anand Mundhe
MTECH CSE (2020-22)
IIIT Vadodara
Gandhinagar, India
202061006@iiitvadodara.ac.in

I. PROBLEM STATEMENT

A. The missionaries and cannibals problem is usually stated as follows. Three missionaries and three cannibals are on one side of a river, along with a boat that can hold one or two people. Find a way to get everyone to the other side without ever leaving a group of missionaries in one place outnumbered by the cannibals in that place. This problem is famous in AI because it was the subject of the first paper that approached problem-formulation from an analytical viewpoint.

B. In the rabbit leap problem, three east-bound rabbits stand in a line blocked by three west-bound rabbits. They are crossing a stream with stones placed in the east west direction in a line. There is one empty stone between them. The rabbits can only move forward one step or two steps. They can jump over one rabbit if the need arises, but not more than that. Are they smart enough to cross each other without having to step into the water?

II. SOLUTION OF MISSIONARIES AND CANNIBALS PROBLEM

- 1) Model the problem as a state space search problem. How large is the search space?

The state space tree can be viewed in figure 1. Each state has value of form (M, C, B) where M represents number of missionaries at left end, C represents number of cannibals at left end and B represents boat position, i.e. 1 if at left side and 0, if at right side.

- 2) Solve the problem using BFS. The optimal solution is the one with the fewest number of steps. Is the solution that you have acquired an optimal one? The program should print out the solution by listing a sequence of steps needed to reach the goal state from the initial state.

```
1 class Node:
2     def __init__(self, puzzle):
3         self.children = list()
4         self.parent = None
5         self.puzzle = list()
6         self.setPuzzle(puzzle)
7
8     def setPuzzle(self, puzzle):
9         for i in range(len(puzzle)):
10            self.puzzle.append(puzzle[i])
```

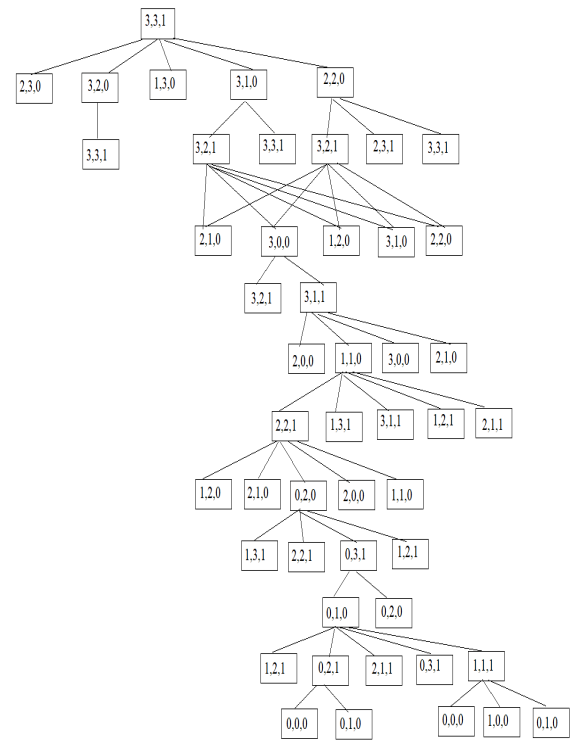


Fig. 1. State space search tree of missionaries cannibals problem

```
11
12 def goalTest(self):
13     return self.puzzle == goal
14
15 def printPuzzle(self):
16     print(self.puzzle)
17
18 def isSamePuzzle(self, p):
19     samePuzzle = True
20     for i in range(len(p)):
21         if self.puzzle[i] != p[i]:
22             samePuzzle = False
23
24     return samePuzzle
25
26 def isValid(self, state):
27     if (state[0] > 3 or state[1] > 3 or state
28         [2] > 1 or state[0] < 0 or state[1] < 0 or state
29         [2] < 0 or (0 < state[0] < state[1]) or (0 < (3 -
```

```

state[0]) < (3 - state[1]))):
    return False
else:
    return True

def expandNode(self):
    moves = [[1, 0, 1], [0, 1, 1], [2, 0, 1], [0, 2, 1], [1, 1, 1]]
    state = self.puzzle
    for each in moves:
        if (state[2] == 1): next_state = [x1 -
x2 for (x1, x2) in zip([state[0], state[1],
state[2]], each)]
        else: next_state = [x1 + x2 for (x1,
x2) in zip([state[0], state[1], state[2]],
each)]
        if (self.isValid(next_state)):
            # print(next_state)
            child = Node(next_state)
            self.children.append(child)
            child.parent = self

def copyPuzzle(self, a, b):
    for i in range(len(b)):
        a.append(b[i])

def pathTrace(path, n):
    print("\nTracing path...")
    current = n
    path.append(current)
    while (current.parent != None):
        current = current.parent
        path.append(current)

class uninformedSearch:
    def bfs(self, root):
        pathToSolution = []
        openList = []
        closedList = []

        openList.append(root)
        goalFound = False
        #print(openList[0].puzzle)
        while len(openList) != 0 and goalFound ==
False:
            currentNode = openList[0]
            closedList.append(currentNode)
            openList.pop(0)
            currentNode.expandNode()

            #currentNode.printPuzzle()
            for i in range(len(currentNode.
children)):
                currentChild = currentNode.
children[i]
                #print(currentNode.puzzle,
currentChild.puzzle)
                #print("curr", currentChild.
goalTest())
                if currentChild.goalTest():
                    print("\nGoal Found")
                    goalFound = True
                    pathTrace(pathToSolution,
currentChild)
                    break
                if (self.contains(openList,
currentChild, "gg") == False and self.contains(
closedList, currentChild, "ff") == False):
                    openList.append(
currentChild)

```

```

return pathToSolution

def contains(self, lis, c, s):
    contains = False
    #print(lis, c, s)
    for i in range(len(lis)):
        if lis[i].isSamePuzzle(c.puzzle) ==
True:
            contains = True
    return contains

puzzle = [3, 3, 1]
goal = [0, 0, 0]
root = Node(puzzle)
ui = uninformedSearch()
solution = ui.bfs(root)
solution.reverse()
if len(solution) > 0:
    for i in range(len(solution)):
        solution[i].printPuzzle()
        if i < len(solution) - 1:
            #print("\n\n*****Next move
*****", i)
            if (solution[i].puzzle[2] == 1):
                print("Move", solution[i].puzzle
[0] - solution[i+1].puzzle[0], "missionary
and ",
                    solution[i].puzzle[1] - solution[
i+1].puzzle[1], "cannibal from left to right
")
                elif (solution[i].puzzle[2] == 0):
                    print("Move", solution[i+1].
puzzle[0] - solution[i].puzzle[0], "
missionary and ",
                        solution[i+1].puzzle[1] -
solution[i].puzzle[1], "cannibal from right
to left")
else:
    print("\nNo path to solution is found")

```

Listing 1. Missionaries and cannibals solution using BFS

Output steps using BFS

```

1
2 Goal Found
3
4 Tracing path...
5 [3, 3, 1]
6 Move 0 missionary and 2 cannibal from left to
right
7 [3, 1, 0]
8 Move 0 missionary and 1 cannibal from right
to left
9 [3, 2, 1]
10 Move 0 missionary and 2 cannibal from left to
right
11 [3, 0, 0]
12 Move 0 missionary and 1 cannibal from right
to left
13 [3, 1, 1]
14 Move 2 missionary and 0 cannibal from left to
right
15 [1, 1, 0]
16 Move 1 missionary and 1 cannibal from right
to left
17 [2, 2, 1]
18 Move 2 missionary and 0 cannibal from left to
right
19 [0, 2, 0]
20 Move 0 missionary and 1 cannibal from right
to left
21 [0, 3, 1]

```

```
22 Move 0 missionary and 2 cannibal from left to
    right
23 [0, 1, 0]
24 Move 1 missionary and 0 cannibal from right
    to left
25 [1, 1, 1]
26 Move 1 missionary and 1 cannibal from left to
    right
27 [0, 0, 0]
```

- 3) Solve the problem using DFS. The program should print out the solution by listing a sequence of steps needed to reach the goal state from the initial state

```

1 class Node:
2     def __init__(self, puzzle):
3         self.children = list()
4         self.parent = None
5         self.puzzle = list()
6         self.setPuzzle(puzzle)
7
8     def setPuzzle(self, puzzle):
9         for i in range(len(puzzle)):
10             self.puzzle.append(puzzle[i])
11
12     def goalTest(self):
13         return self.puzzle == goal
14
15     def printPuzzle(self):
16         print(self.puzzle)
17
18     def isSamePuzzle(self, p):
19         samePuzzle = True
20         for i in range(len(p)):
21             if self.puzzle[i] != p[i]:
22                 samePuzzle = False
23
24         return samePuzzle
25
26     def isValid(self, state):
27         if (state[0] > 3 or state[1] > 3 or state
28             [2] > 1 or state[0] < 0 or state[1] < 0 or state
29             [2] < 0 or (0 < state[0] < state[1]) or (0 < (3 -
30             state[0]) < (3 - state[1]))):
31             return False
32         else:
33             return True
34
35     def expandNode(self):
36         moves = [[1, 0, 1], [0, 1, 1], [2, 0,
37             1], [0, 2, 1], [1, 1, 1]]
38         state = self.puzzle
39         for each in moves:
40             if (state[2] == 1): next_state = [x1 -
41                 x2 for (x1, x2) in zip([state[0], state[1],
42                 state[2]], each)]
43             else: next_state = [x1 + x2 for (x1,
44                 x2) in zip([state[0], state[1], state[2]],
45                 each)]
46             if (self.isValid(next_state)):
47                 # print(next_state)
48                 child = Node(next_state)
49                 self.children.append(child)
50                 child.parent = self
51
52     def copyPuzzle(self, a, b):
53         for i in range(len(b)):
54             a.append(b[i])
55
56 def pathTrace(path, n):
57     print("\nTracing path...")
58     current = n
59     path.append(current)
60     while (current.parent != None):
61         current = current.parent
62         path.append(current)
63
64     def contains(self, lis, c, s):
65         contains = False
66         # print(lis, c, s)
67         for i in range(len(lis)):
68             if lis[i].isSamePuzzle(c.puzzle) ==
69                 True:

```

```

70                 contains = True
71                 return contains
72
73     class uninformedSearch:
74     def dfs(self, root):
75         pathToSolution = []
76         openList = []
77         closedList = []
78
79         openList.append(root)
80         goalFound = False
81         # print(openList[0].puzzle)
82         while len(openList) != 0 and goalFound ==
83             False:
84             currentNode = openList[-1]
85             closedList.append(currentNode)
86             openList.pop(-1)
87             currentNode.expandNode()
88
89             # currentNode.printPuzzle()
90             for i in range(len(currentNode.
91                 children)):
92                 currentChild = currentNode.
93                 children[i]
94                 # print(currentNode.puzzle,
95                 currentChild.puzzle)
96                 # print("curr", currentChild.
97                 goalTest())
98                 if currentChild.goalTest():
99                     print("\nGoal Found")
100                     goalFound = True
101                     pathTrace(pathToSolution,
102                         currentChild)
103                     break
104                 if (self.contains(openList,
105                     currentChild, "gg") == False and self.contains
106                     (closedList, currentChild, "ff") == False):
107                     openList.append(
108                         currentChild)
109
110             return pathToSolution
111     def contains(self, lis, c, s):
112         contains = False
113         # print(lis, c, s)
114         for i in range(len(lis)):
115             if lis[i].isSamePuzzle(c.puzzle) ==
116                 True:
117                 contains = True
118                 return contains
119
120 puzzle = [3, 3, 1]
121 goal = [0, 0, 0]
122 root = Node(puzzle)
123 ui = uninformedSearch()
124 solution = ui.dfs(root)
125 solution.reverse()
126 if len(solution) > 0:
127     for i in range(len(solution)):
128         solution[i].printPuzzle()
129         if i < len(solution) - 1:
130             # print("\n\n*****Next move
131             *****", i)
132             if (solution[i].puzzle[2] == 1):
133                 print("Move", solution[i].puzzle
134                     [0] - solution[i + 1].puzzle[0], " missionary
135                     and ",
136                     solution[i].puzzle[1] - solution[
137                     i + 1].puzzle[1], "cannibal from left to right
138                     ")
139                 elif (solution[i].puzzle[2] == 0):
140                     print("Move", solution[i + 1].
141                         puzzle[0] - solution[i].puzzle[0], "
142                         missionary and ",

```

```

119         solution[i+1].puzzle[1]-
            solution[i].puzzle[1],"cannibal from right
            to left")
120     else:
121         print("\nNo path to solution is found")

```

Listing 2. Missionaries and cannibals solution using DFS

Output steps using DFS

```

1 Goal Found
2
3 Tracing path...
4 [3, 3, 1]
5 Move 1 missionary and 1 cannibal from left to
  right
6 [2, 2, 0]
7 Move 1 missionary and 0 cannibal from right
  to left
8 [3, 2, 1]
9 Move 0 missionary and 2 cannibal from left to
  right
10 [3, 0, 0]
11 Move 0 missionary and 1 cannibal from right
  to left
12 [3, 1, 1]
13 Move 2 missionary and 0 cannibal from left to
  right
14 [1, 1, 0]
15 Move 1 missionary and 1 cannibal from right
  to left
16 [2, 2, 1]
17 Move 2 missionary and 0 cannibal from left to
  right
18 [0, 2, 0]
19 Move 0 missionary and 1 cannibal from right
  to left
20 [0, 3, 1]
21 Move 0 missionary and 2 cannibal from left to
  right
22 [0, 1, 0]
23 Move 0 missionary and 1 cannibal from right
  to left
24 [0, 2, 1]
25 Move 0 missionary and 2 cannibal from left to
  right
26 [0, 0, 0]

```

- 4) Compare solutions found from BFS and DFS. Comment on solutions. Also compare the time and space complexities of both.

Both BFS and DFS took 11 steps to reach the solution. However the steps given by these 2 approaches are different. For branching factor b and depth d Time complexity of BFS is $O(b^d)$ and space complexity is $O(b^d)$ whereas for DFS time complexity is $O(b^d)$ and space complexity is $O(bd)$. Further BFS gives optimality while DFS does not. Completeness is given by BFS when b is finite and is given by DFS when both b and d are finite

III. SOLUTION OF RABBIT LEAP PROBLEM

- 1) Model the problem as a state space search problem. How large is the search space?
- 2) Solve the problem using BFS. The optimal solution is the one with the fewest number of steps. Is the solution that you have acquired an optimal one? The program

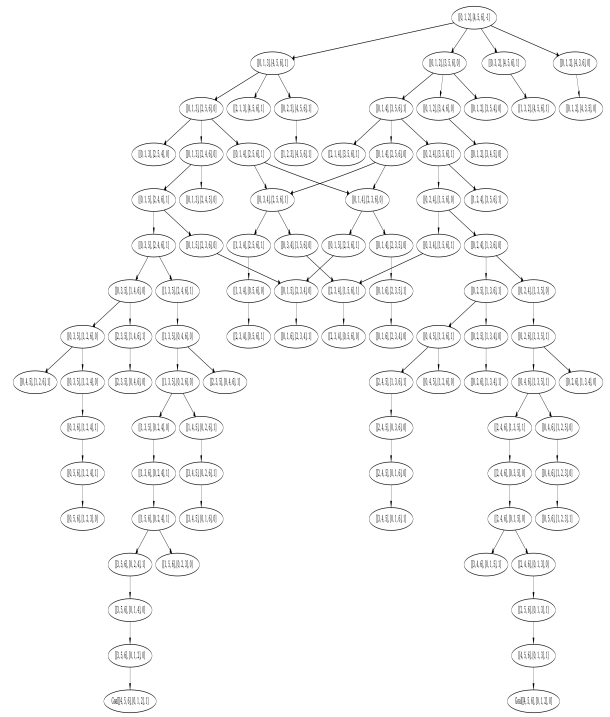


Fig. 2. State space search tree of rabbit leap problem

should print out the solution by listing a sequence of steps needed to reach the goal state from the initial state.

```

1 import numpy as np
2 class Node:
3     def __init__(self,puzzle):
4         self.children=list()
5         self.parent=None
6         self.puzzle=list()
7         self.setPuzzle(puzzle)
8
9     def setPuzzle(self,puzzle):
10         for i in range(len(puzzle)):
11             self.puzzle.append(puzzle[i])
12
13     def goalTest(self):
14         return self.puzzle==goal
15     def isValid(self,m,c):
16         for i in range(len(m)):
17             if c.count(m[i])!=0 or m.count(m[i])
18             !=1 or c.count(c[i])!=1:return False
19             if(m[i]>6 or c[i]<0):return False
20             else:return True
21     def moveLeftToRight(self,p,k):
22         pc=[]
23         tmp=np.array([0,0,0])
24         for i in range(3):
25             tmp[i]+=k
26             if self.isValid((np.array(p[0])+tmp
27             ).tolist(),p[1]):#(p[0]-k>=p[1] or p[0]-k
28             ==0) and not p[0]-k<0:
29                 #print("kk")
30                 pc=[(np.array(p[0])+tmp).tolist
31                 (),p[1],1]
32                 child=Node(pc)
33                 self.children.append(child)
34                 child.parent=self
35                 tmp[i]-=k

```

```

32
33 def moveRightToLeft(self,p,k):
34     pc=[]
35     tmp=np.array([0,0,0])
36     for i in range(3):
37         tmp[i]+=k
38         if self.isValid(p[0],(np.array(p
39 [1])-tmp).tolist()):#(p[0]-k>=p[1] or p
40 [0]-k==0) and not p[0]-k<0:
41             #print("kk")
42             pc=[p[0],(np.array(p[1])-tmp).
43 tolist(),0]
44             child=Node(pc)
45             self.children.append(child)
46             child.parent=self
47             tmp[i]-=k
48
49 def printPuzzle(self):
50     print(self.puzzle)
51
52 def isSamePuzzle(self,p):
53     samePuzzle=True
54     for i in range(len(p)):
55         if self.puzzle[i]!=p[i]:
56             samePuzzle=False
57
58     return samePuzzle
59
60 def expandNode(self):
61     self.moveLeftToRight(self.puzzle,1)
62     self.moveLeftToRight(self.puzzle,2)
63     self.moveRightToLeft(self.puzzle,1)
64     self.moveRightToLeft(self.puzzle,2)
65
66 def copyPuzzle(self,a,b):
67     for i in range(len(b)):
68         a.append(b[i])
69
70 def pathTrace(path,n):
71     print("\nTracing path...")
72     current=n
73     path.append(current)
74     while (current.parent!=None):
75         current=current.parent
76         path.append(current)
77
78 class uninformedSearch:
79     def bfs(self,root):
80         pathToSolution=[]
81         openList=[]
82         closedList=[]
83
84         openList.append(root)
85         goalFound=False
86         explored=0
87         #print(openList[0].puzzle)
88         while len(openList)!=0 and goalFound==
89 False:
90             currentNode=openList[0]
91             closedList.append(currentNode)
92             openList.pop(0)
93             currentNode.expandNode()
94             explored+=1
95             #currentNode.printPuzzle()
96             for i in range(len(currentNode.
97 children)):
98                 currentChild=currentNode.
99 children[i]
100                 #print(currentNode.puzzle,
101 currentChild.puzzle)
102                 #print("curr",currentChild.
103 goalTest())

```

```

98         if currentChild.goalTest():
99             print("\nGoal Found")
100             goalFound=True
101             pathTrace(pathToSolution,
102 currentChild)
103             break
104             if(self.contains(openList,
105 currentChild,"gg")==False and self.contains
106 (closedList,currentChild,"ff")==False):
107                 openList.append(
108 currentChild)
109
110         print("Nodes explored are :",explored)
111         return pathToSolution
112
113 def contains(self,lis,c,s):
114     contains=False
115     #print(lis,c,s)
116     for i in range(len(lis)):
117         if lis[i].isSamePuzzle(c.puzzle)==
118 True:
119             contains=True
120             return contains
121
122 puzzle=[[0,1,2],[4,5,6],-1]
123 goal=[[4,5,6],[0,1,2],0]
124 root=Node(puzzle)
125 ui=uninformedSearch()
126 solution=ui.bfs(root)
127 solution.reverse()
128 k=0
129 if len(solution)>0:
130     for i in range(len(solution)):
131         solution[i].printPuzzle()
132         k+=1
133         if i<len(solution)-1:
134             #print("\n\n*****Next move
135 *****",i)
136             if(solution[i+1].puzzle[2]==1):
137                 for j in range(len(solution[i].
138 puzzle[0])):
139                     if solution[i].puzzle[0][j
140 ]!=solution[i+1].puzzle[0][j]:
141                         f=j
142                         break
143                         print(k,"Move",f,"rabbit from
144 left to right")
145                     elif(solution[i+1].puzzle[2]==0):
146                         for j in range(len(solution[i].
147 puzzle[0])):
148                             if solution[i].puzzle[0][j
149 ]!=solution[i+1].puzzle[0][j]:
150                                 f=j
151                                 break
152                                 print(k,"Move",f,"rabbit from
153 right to left")
154 else:
155     print("\nNo path to solution is found")

```

Listing 3. Rabbit Leap problem using BFS

Output steps using BFS

```

1 Goal Found
2
3 Tracing path...
4 Nodes explored are : 89
5 [[0, 1, 2], [4, 5, 6], -1]
6 1 Move 0 rabbit from right to left
7 [[0, 1, 2], [3, 5, 6], 0]
8 2 Move 2 rabbit from left to right
9 [[0, 1, 4], [3, 5, 6], 1]
10 3 Move 1 rabbit from left to right

```

```
11 [[0, 2, 4], [3, 5, 6], 1]
12 4 Move 1 rabbit from right to left
13 [[0, 2, 4], [1, 5, 6], 0]
14 5 Move 1 rabbit from right to left
15 [[0, 2, 4], [1, 3, 6], 0]
16 6 Move 1 rabbit from right to left
17 [[0, 2, 4], [1, 3, 5], 0]
18 7 Move 2 rabbit from left to right
19 [[0, 2, 6], [1, 3, 5], 1]
20 8 Move 1 rabbit from left to right
21 [[0, 4, 6], [1, 3, 5], 1]
22 9 Move 0 rabbit from left to right
23 [[2, 4, 6], [1, 3, 5], 1]
24 10 Move 0 rabbit from right to left
25 [[2, 4, 6], [0, 3, 5], 0]
26 11 Move 0 rabbit from right to left
27 [[2, 4, 6], [0, 1, 5], 0]
28 12 Move 0 rabbit from right to left
29 [[2, 4, 6], [0, 1, 3], 0]
30 13 Move 1 rabbit from left to right
31 [[2, 5, 6], [0, 1, 3], 1]
32 14 Move 0 rabbit from left to right
33 [[4, 5, 6], [0, 1, 3], 1]
34 15 Move 0 rabbit from right to left
35 [[4, 5, 6], [0, 1, 2], 0]
```

- 3) Solve the problem using DFS. The program should print out the solution by listing a sequence of steps needed to reach the goal state from the initial state

```

1 import numpy as np
2 class Node:
3     def __init__(self, puzzle):
4         self.children = list()
5         self.parent = None
6         self.puzzle = list()
7         self.setPuzzle(puzzle)
8
9     def setPuzzle(self, puzzle):
10        for i in range(len(puzzle)):
11            self.puzzle.append(puzzle[i])
12
13    def goalTest(self):
14        return self.puzzle == goal
15    def isValid(self, m, c):
16        for i in range(len(m)):
17            if c.count(m[i]) != 0 or m.count(m[i]) != 1 or c.count(c[i]) != 1: return False
18            if m[i] > 6 or c[i] < 0: return False
19            else: return True
20    def moveLeftToRight(self, p, k):
21        pc = []
22        tmp = np.array([0, 0, 0])
23        for i in range(3):
24            tmp[i] += k
25            if self.isValid((np.array(p[0]) + tmp).tolist(), p[1]): # (p[0]-k >= p[1] or p[0]-k == 0) and not p[0]-k < 0:
26                #print("kk")
27                pc = [(np.array(p[0]) + tmp).tolist(), p[1], 1]
28                child = Node(pc)
29                self.children.append(child)
30                child.parent = self
31                tmp[i] -= k
32
33    def moveRightToLeft(self, p, k):
34        pc = []
35        tmp = np.array([0, 0, 0])
36        for i in range(3):
37            tmp[i] += k
38            if self.isValid(p[0], (np.array(p[1]) - tmp).tolist()): # (p[0]-k >= p[1] or p[0]-k == 0) and not p[0]-k < 0:
39                #print("kk")
40                pc = [p[0], (np.array(p[1]) - tmp).tolist(), 0]
41                child = Node(pc)
42                self.children.append(child)
43                child.parent = self
44                tmp[i] -= k
45
46    def printPuzzle(self):
47        print(self.puzzle)
48
49    def isSamePuzzle(self, p):
50        samePuzzle = True
51        for i in range(len(p)):
52            if self.puzzle[i] != p[i]:
53                samePuzzle = False
54
55        return samePuzzle
56
57    def expandNode(self):
58        self.moveLeftToRight(self.puzzle, 1)
59        self.moveLeftToRight(self.puzzle, 2)
60        self.moveRightToLeft(self.puzzle, 1)
61        self.moveRightToLeft(self.puzzle, 2)
62
63

```

```

64 def copyPuzzle(self, a, b):
65     for i in range(len(b)):
66         a.append(b[i])
67
68 def pathTrace(path, n):
69     print("\nTracing path...")
70     current = n
71     path.append(current)
72     while (current.parent != None):
73         current = current.parent
74         path.append(current)
75
76 class uninformedSearch:
77     def dfs(self, root):
78         pathToSolution = []
79         openList = []
80         closedList = []
81
82         openList.append(root)
83         goalFound = False
84         explored = 0
85         #print(openList[0].puzzle)
86         while len(openList) != 0 and goalFound == False:
87             currentNode = openList[-1]
88             closedList.append(currentNode)
89             openList.pop(-1)
90             currentNode.expandNode()
91             explored += 1
92             #currentNode.printPuzzle()
93             for i in range(len(currentNode.children)):
94                 currentChild = currentNode.children[i]
95                 #print(currentNode.puzzle, currentChild.puzzle)
96                 #print("curr", currentChild.goalTest())
97                 if currentChild.goalTest():
98                     print("\nGoal Found")
99                     goalFound = True
100                    pathTrace(pathToSolution, currentChild)
101                    break
102                    if (self.contains(openList, currentChild, "gg") == False and self.contains(closedList, currentChild, "ff") == False):
103                        openList.append(currentChild)
104
105        print("Nodes explored are :", explored)
106        return pathToSolution
107
108    def contains(self, lis, c, s):
109        contains = False
110        #print(lis, c, s)
111        for i in range(len(lis)):
112            if lis[i].isSamePuzzle(c.puzzle) == True:
113                contains = True
114                return contains
115
116 puzzle = [[0, 1, 2], [4, 5, 6], -1]
117 goal = [[4, 5, 6], [0, 1, 2], 0]
118 root = Node(puzzle)
119 ui = uninformedSearch()
120 solution = ui.dfs(root)
121 solution.reverse()
122 k = 0
123 if len(solution) > 0:
124     for i in range(len(solution)):
125         solution[i].printPuzzle()
126         k += 1
127         if i < len(solution) - 1:
128             #print("\n\n*****Next move *****", i)

```



```

127         if(solution[i+1].puzzle[2]==1):
128             for j in range(len(solution[i].
puzzle[0])):
129                 if solution[i].puzzle[0][j
] != solution[i+1].puzzle[0][j]:
130                     f=j
131                     break
132             print(k, "Move", f, "rabbit from
left to right")
133             elif(solution[i+1].puzzle[2]==0):
134                 for j in range(len(solution[i].
puzzle[0])):
135                     if solution[i].puzzle[0][j
] != solution[i+1].puzzle[0][j]:
136                         f=j
137                         break
138             print(k, "Move", f, "rabbit from
right to left")
139 else:
140     print("\nNo path to solution is found")

```

Listing 4. Rabbit leap problem using DFS

Output steps using DFS

```

1 Goal Found
2 Goal Found
3
4 Tracing path...
5 Nodes explored are : 40
6 [[0, 1, 2], [4, 5, 6], -1]
7 1 Move 0 rabbit from right to left
8 [[0, 1, 2], [3, 5, 6], 0]
9 2 Move 2 rabbit from left to right
10 [[0, 1, 4], [3, 5, 6], 1]
11 3 Move 1 rabbit from left to right
12 [[0, 2, 4], [3, 5, 6], 1]
13 4 Move 1 rabbit from right to left
14 [[0, 2, 4], [1, 5, 6], 0]
15 5 Move 1 rabbit from right to left
16 [[0, 2, 4], [1, 3, 6], 0]
17 6 Move 1 rabbit from right to left
18 [[0, 2, 4], [1, 3, 5], 0]
19 7 Move 2 rabbit from left to right
20 [[0, 2, 6], [1, 3, 5], 1]
21 8 Move 1 rabbit from left to right
22 [[0, 4, 6], [1, 3, 5], 1]
23 9 Move 0 rabbit from left to right
24 [[2, 4, 6], [1, 3, 5], 1]
25 10 Move 0 rabbit from right to left
26 [[2, 4, 6], [0, 3, 5], 0]
27 11 Move 0 rabbit from right to left
28 [[2, 4, 6], [0, 1, 5], 0]
29 12 Move 0 rabbit from right to left
30 [[2, 4, 6], [0, 1, 3], 0]
31 13 Move 1 rabbit from left to right
32 [[2, 5, 6], [0, 1, 3], 1]
33 14 Move 0 rabbit from left to right
34 [[4, 5, 6], [0, 1, 3], 1]
35 15 Move 0 rabbit from right to left
36 [[4, 5, 6], [0, 1, 2], 0]

```

- 4) Compare solutions found from BFS and DFS. Comment on solutions. Also compare the time and space complexities of both.

Both BFS and DFS took 15 steps to reach the solution. However the steps given by these 2 approaches are different. For branching factor b and depth d Time complexity of BFS is $O(b^d)$ and space complexity is $O(b^d)$ whereas for DFS time complexity is $O(b^d)$ and space complexity is $O(bd)$. Further BFS gives optimality

while DFS does not. Completeness is given by BFS when b is finite and is given by DFS when both b and d are finite

REFERENCES

- [1] Artificial Intelligence: a Modern Approach, Russell and Norvig (Fourth edition)