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.

```
class Node:
    def __init__(self,puzzle):
        self.children=list()
        self.parent=None
        self.puzzle=list()
        self.setPuzzle(puzzle)

    def setPuzzle(self,puzzle):
        for i in range(len(puzzle)):
            self.puzzle.append(puzzle[i])

    def goalTest(self):
        return self.puzzle==goal

    def printPuzzle(self):
        print(self.puzzle)

    def isSamePuzzle(self,p):
        samePuzzle=True
        for i in range(len(p)):
            if self.puzzle[i]!=p[i]:
                samePuzzle=False

        return samePuzzle

    def isValid(self,state):
        if(state[0]>3 or state[1]>3 or state
[2]>1 or state[0]<0 or state[1]<0 or state
[2]<0 or (0<state[0]<state[1]) or (0<(3-
```

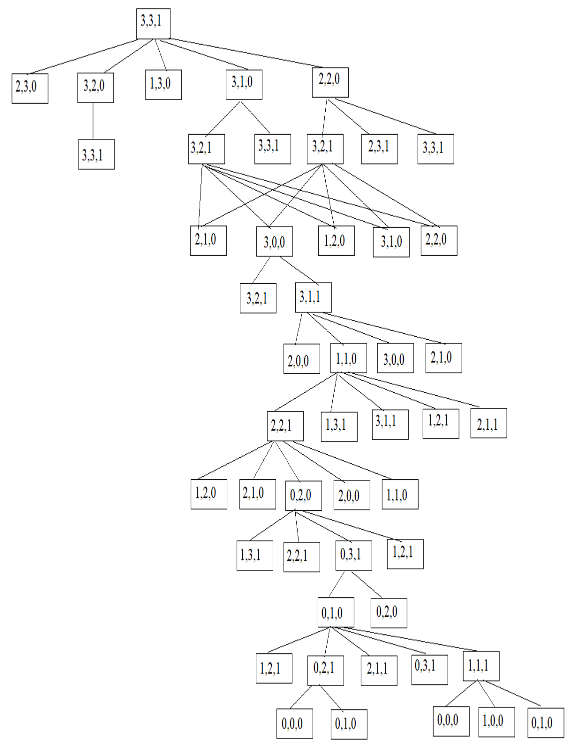

Fig. 1. State space search tree of missionaries cannibals problem

```
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

```

Goal Found

Tracing path...
[3, 3, 1]
Move 0  missionary and  2 cannibal from left to right
[3, 1, 0]
Move 0  missionary and  1 cannibal from right to left
[3, 2, 1]
Move 0  missionary and  2 cannibal from left to right
[3, 0, 0]
Move 0  missionary and  1 cannibal from right to left
[3, 1, 1]
Move 2  missionary and  0 cannibal from left to right
[1, 1, 0]
Move 1  missionary and  1 cannibal from right to left
[2, 2, 1]
Move 2  missionary and  0 cannibal from left to right
[0, 2, 0]
Move 0  missionary and  1 cannibal from right to left
[0, 3, 1]
```

```
Move 0  missionary and  2 cannibal from left to right
[0, 1, 0]
Move 1  missionary and  0 cannibal from right to left
[1, 1, 1]
Move 1  missionary and  1 cannibal from left to right
[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

```
class Node:
    def __init__(self,puzzle):
        self.children=list()
        self.parent=None
        self.puzzle=list()
        self.setPuzzle(puzzle)

    def setPuzzle(self,puzzle):
        for i in range(len(puzzle)):
            self.puzzle.append(puzzle[i])

    def goalTest(self):
        return self.puzzle==goal

    def printPuzzle(self):
        print(self.puzzle)

    def isSamePuzzle(self,p):
        samePuzzle=True
        for i in range(len(p)):
            if self.puzzle[i]!=p[i]:
                samePuzzle=False

        return samePuzzle

    def isValid(self,state):
        if(state[0]>3 or state[1]>3 or state[2]>1 or state[0]<0 or state[1]<0 or state[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)

    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

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

        openList.append(root)
        goalFound=False
        #print(openList[0].puzzle)
        while len(openList)!=0 and goalFound==False:
            currentNode=openList[-1]
            closedList.append(currentNode)
            openList.pop(-1)
            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.dfs(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 2. Missionaries and cannibals solution using DFS

Output steps using DFS

```
Goal Found

Tracing path...
[3, 3, 1]
Move 1  missionary and  1 cannibal from left to right
[2, 2, 0]
Move 1  missionary and  0 cannibal from right to left
[3, 2, 1]
Move 0  missionary and  2 cannibal from left to right
[3, 0, 0]
Move 0  missionary and  1 cannibal from right to left
[3, 1, 1]
Move 2  missionary and  0 cannibal from left to right
[1, 1, 0]
Move 1  missionary and  1 cannibal from right to left
[2, 2, 1]
Move 2  missionary and  0 cannibal from left to right
[0, 2, 0]
Move 0  missionary and  1 cannibal from right to left
[0, 3, 1]
Move 0  missionary and  2 cannibal from left to right
[0, 1, 0]
Move 0  missionary and  1 cannibal from right to left
[0, 2, 1]
Move 0  missionary and  2 cannibal from left to right
[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 compexity 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 should print out the solution by listing a sequence of steps needed to reach the goal state from the initial state.

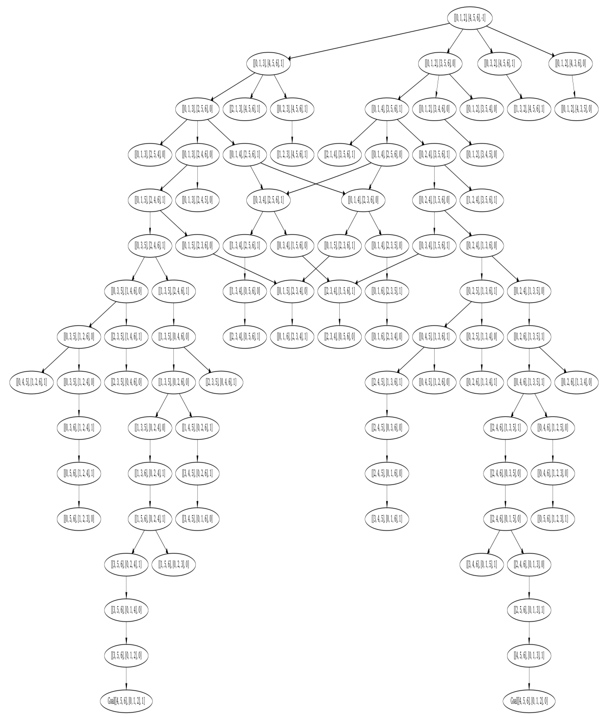

Fig. 2. State space search tree of rabbit leap problem

```
import numpy as np
class Node:
    def __init__(self,puzzle):
        self.children=list()
        self.parent=None
        self.puzzle=list()
        self.setPuzzle(puzzle)

    def setPuzzle(self,puzzle):
        for i in range(len(puzzle)):
            self.puzzle.append(puzzle[i])

    def goalTest(self):
        return self.puzzle==goal
    def isValid(self,m,c):
        for i in range(len(m)):
            if c.count(m[i])!=0 or m.count(m[i]) !=1 or c.count(c[i])!=1:return False
            if(m[i]>6 or c[i]<0):return False
        else:return True
    def moveLeftToRight(self,p,k):
        pc=[]
        tmp=np.array([0,0,0])
        for i in range(3):
            tmp[i]+=k
            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:
                #print("kk")
                pc=[(np.array(p[0])+tmp).tolist(),p[1],1]
                child=Node(pc)
                self.children.append(child)
                child.parent=self
            tmp[i]-=k
```

```python

    def moveRightToLeft(self,p,k):
        pc=[]
        tmp=np.array([0,0,0])
        for i in range(3):
            tmp[i]+=k
            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:
                #print("kk")
                pc=[p[0],(np.array(p[1])-tmp).tolist(),0]
                child=Node(pc)
                self.children.append(child)
                child.parent=self
            tmp[i]-=k

    def printPuzzle(self):
        print(self.puzzle)

    def isSamePuzzle(self,p):
        samePuzzle=True
        for i in range(len(p)):
            if self.puzzle[i]!=p[i]:
                samePuzzle=False

        return samePuzzle

    def expandNode(self):
        self.moveLeftToRight(self.puzzle,1)
        self.moveLeftToRight(self.puzzle,2)
        self.moveRightToLeft(self.puzzle,1)
        self.moveRightToLeft(self.puzzle,2)


    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
        explored=0
        #print(openList[0].puzzle)
        while len(openList)!=0 and goalFound==False:
            currentNode=openList[0]
            closedList.append(currentNode)
            openList.pop(0)
            currentNode.expandNode()
            explored+=1
            #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)

        print("Nodes explored are :",explored)
        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=[[0,1,2],[4,5,6],-1]
goal=[[4,5,6],[0,1,2],0]
root=Node(puzzle)
ui=uninformedSearch()
solution=ui.bfs(root)
solution.reverse()
k=0
if len(solution)>0:
    for i in range(len(solution)):
        solution[i].printPuzzle()
        k+=1
        if i<len(solution)-1:
            #print("\n\n******Next move **********",i)
            if(solution[i+1].puzzle[2]==1):
                for j in range(len(solution[i].puzzle[0])):
                    if solution[i].puzzle[0][j]!=solution[i+1].puzzle[0][j]:
                        f=j
                        break
                print(k,"Move",f,"rabbit from left to right")
            elif(solution[i+1].puzzle[2]==0):
                for j in range(len(solution[i].puzzle[0])):
                    if solution[i].puzzle[0][j]!=solution[i+1].puzzle[0][j]:
                        f=j
                        break
                print(k,"Move",f,"rabbit from right to left")
else:
    print("\nNo path to solution is found")
```

Listing 3. Rabbit Leap problem using BFS

Output steps using BFS

```
Goal Found

Tracing path...
Nodes explored are : 89
[[0, 1, 2], [4, 5, 6], -1]
1 Move 0 rabbit from right to left
[[0, 1, 2], [3, 5, 6], 0]
2 Move 2 rabbit from left to right
[[0, 1, 4], [3, 5, 6], 1]
3 Move 1 rabbit from left to right
```

```
[[0, 2, 4], [3, 5, 6], 1]
4 Move 1 rabbit from right to left
[[0, 2, 4], [1, 5, 6], 0]
5 Move 1 rabbit from right to left
[[0, 2, 4], [1, 3, 6], 0]
6 Move 1 rabbit from right to left
[[0, 2, 4], [1, 3, 5], 0]
7 Move 2 rabbit from left to right
[[0, 2, 6], [1, 3, 5], 1]
8 Move 1 rabbit from left to right
[[0, 4, 6], [1, 3, 5], 1]
9 Move 0 rabbit from left to right
[[2, 4, 6], [1, 3, 5], 1]
10 Move 0 rabbit from right to left
[[2, 4, 6], [0, 3, 5], 0]
11 Move 0 rabbit from right to left
[[2, 4, 6], [0, 1, 5], 0]
12 Move 0 rabbit from right to left
[[2, 4, 6], [0, 1, 3], 0]
13 Move 1 rabbit from left to right
[[2, 5, 6], [0, 1, 3], 1]
14 Move 0 rabbit from left to right
[[4, 5, 6], [0, 1, 3], 1]
15 Move 0 rabbit from right to left
[[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

```python
 import numpy as np
class Node:
    def __init__(self,puzzle):
        self.children=list()
        self.parent=None
        self.puzzle=list()
        self.setPuzzle(puzzle)

    def setPuzzle(self,puzzle):
        for i in range(len(puzzle)):
            self.puzzle.append(puzzle[i])

    def goalTest(self):
        return self.puzzle==goal
    def isValid(self,m,c):
        for i in range(len(m)):
            if c.count(m[i])!=0 or m.count(m[i]) !=1 or c.count(c[i])!=1:return False
            if(m[i]>6 or c[i]<0):return False
        else:return True
    def moveLeftToRight(self,p,k):
        pc=[]
        tmp=np.array([0,0,0])
        for i in range(3):
            tmp[i]+=k
            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:
                #print("kk")
                pc=[(np.array(p[0])+tmp).tolist(),p[1],1]
                child=Node(pc)
                self.children.append(child)
                child.parent=self
            tmp[i]-=k

    def moveRightToLeft(self,p,k):
        pc=[]
        tmp=np.array([0,0,0])
        for i in range(3):
            tmp[i]+=k
            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:
                #print("kk")
                pc=[p[0],(np.array(p[1])-tmp).tolist(),0]
                child=Node(pc)
                self.children.append(child)
                child.parent=self
            tmp[i]-=k

    def printPuzzle(self):
        print(self.puzzle)

    def isSamePuzzle(self,p):
        samePuzzle=True
        for i in range(len(p)):
            if self.puzzle[i]!=p[i]:
                samePuzzle=False

        return samePuzzle

    def expandNode(self):
        self.moveLeftToRight(self.puzzle,1)
        self.moveLeftToRight(self.puzzle,2)
        self.moveRightToLeft(self.puzzle,1)
        self.moveRightToLeft(self.puzzle,2)


    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 dfs(self,root):
        pathToSolution=[]
        openList=[]
        closedList=[]

        openList.append(root)
        goalFound=False
        explored=0
        #print(openList[0].puzzle)
        while len(openList)!=0 and goalFound==False:
            currentNode=openList[-1]
            closedList.append(currentNode)
            openList.pop(-1)
            currentNode.expandNode()
            explored+=1
            #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)

        print("Nodes explored are :",explored)
        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=[[0,1,2],[4,5,6],-1]
goal=[[4,5,6],[0,1,2],0]
root=Node(puzzle)
ui=uninformedSearch()
solution=ui.dfs(root)
solution.reverse()
k=0
if len(solution)>0:
    for i in range(len(solution)):
        solution[i].printPuzzle()
        k+=1
        if i<len(solution)-1:
            #print("\n\n******Next move **********",i)
```

```
            if(solution[i+1].puzzle[2]==1):
                for j in range(len(solution[i].puzzle[0])):
                    if solution[i].puzzle[0][j]!=solution[i+1].puzzle[0][j]:
                        f=j
                        break
                print(k,"Move",f,"rabbit from left to right")
            elif(solution[i+1].puzzle[2]==0):
                for j in range(len(solution[i].puzzle[0])):
                    if solution[i].puzzle[0][j]!=solution[i+1].puzzle[0][j]:
                        f=j
                        break
                print(k,"Move",f,"rabbit from right to left")
else:
    print("\nNo path to solution is found")
```

Listing 4. Rabbit leap problem using DFS

Output steps using DFS

```
Goal Found
Goal Found

Tracing path...
Nodes explored are : 40
[[0, 1, 2], [4, 5, 6], -1]
1 Move 0 rabbit from right to left
[[0, 1, 2], [3, 5, 6], 0]
2 Move 2 rabbit from left to right
[[0, 1, 4], [3, 5, 6], 1]
3 Move 1 rabbit from left to right
[[0, 2, 4], [3, 5, 6], 1]
4 Move 1 rabbit from right to left
[[0, 2, 4], [1, 5, 6], 0]
5 Move 1 rabbit from right to left
[[0, 2, 4], [1, 3, 6], 0]
6 Move 1 rabbit from right to left
[[0, 2, 4], [1, 3, 5], 0]
7 Move 2 rabbit from left to right
[[0, 2, 6], [1, 3, 5], 1]
8 Move 1 rabbit from left to right
[[0, 4, 6], [1, 3, 5], 1]
9 Move 0 rabbit from left to right
[[2, 4, 6], [1, 3, 5], 1]
10 Move 0 rabbit from right to left
[[2, 4, 6], [0, 3, 5], 0]
11 Move 0 rabbit from right to left
[[2, 4, 6], [0, 1, 5], 0]
12 Move 0 rabbit from right to left
[[2, 4, 6], [0, 1, 3], 0]
13 Move 1 rabbit from left to right
[[2, 5, 6], [0, 1, 3], 1]
14 Move 0 rabbit from left to right
[[4, 5, 6], [0, 1, 3], 1]
15 Move 0 rabbit from right to left
[[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 compexity 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)