untitled8

May 5, 2024

[5]:
```python
def selectionSort(arr):
    for i in range(len(arr)):
        min = float('-inf')
        for j in range(i + 1, len(arr)):
            if arr[i] >arr[j]:
              arr[i],arr[j] = arr[j], arr[i]
              return arr
print(selectionSort([59,56,45,34,65,16]))
```

```
[56, 59, 45, 34, 65, 16]
```

[6]:
```python
from collections import deque
graph = {
    '5':['3','7'],
    '3':['2','4'],
    '7':['4'],
    '2':[],
    '4':['8'],
    '8':[]
}
def dfs(node, visited = set()):
    if node not in visited:
        print(node, end = "")
        visited.add(node)
        for neighbour in graph[node]:
            dfs(neighbour, visited)
def bfs(start):
    visited = set()
    queue = deque([start])
    while queue:
        node = queue.popleft()
        if node not in visited:
            print(node, end = "")
            visited.add(node)
            queue.extend(graph[node])
print("DFS")
dfs('5')
print("\nBFS")
```

```
bfs('5')
```

```
DFS
532487
BFS
537248
```

[8]:
```
def printsol(board):
    for row in board:
        for cell in row:
            print(cell, end="")
        print()

def is_safe(board, row, column):
    # Check if there is a queen in the same row
    for i in range(column):
        if board[row][i] == 1:
            return False

    # Check upper diagonal on left side
    for i, j in zip(range(row, -1, -1), range(column, -1, -1)):
        if board[i][j] == 1:
            return False

    # Check lower diagonal on left side
    for i, j in zip(range(row, N), range(column, -1, -1)):
        if board[i][j] == 1:
            return False

    return True

def solveNQUtil(board, column):
    if column >= N:
        return True

    for i in range(N):
        if is_safe(board, i, column):
            board[i][column] = 1
            if solveNQUtil(board, column + 1):
                return True
            board[i][column] = 0

    return False

def solveprob():
    board = [[0 for _ in range(N)] for _ in range(N)]
    if not solveNQUtil(board, 0):
        print("Solution does not exist")
        return False

    print("Solution for", N, "Queen problem is:")
    printsol(board)
    return True

N = int(input("Enter the number of Queens: "))
solveprob()
```

```
Enter the number of Queens:  8

Solution for 8 Queen problem is:
10000000
00000010
00001000
00000001
01000000
00010000
00000100
00100000
```

[8]: True

[ ]:
```
# Practical No - 05
import nltk
from nltk.chat.util import Chat, reflections

pairs = [
    [
        r"My name is (.*)",
        ["Hello %1, how are you?"]
    ],
    [
        r"Hi|Hello|Hey there|Hola",
        ["Hello My Name is S.I.F.R.A"]
    ],
    [
        r"What is Your Name ?",
        ["I am a Bot Created by Developers of JSPM!",]
    ],
    [
        r"How Are You ?",
        ["I'm Doing Good How About You ?",]
    ],
    [
        r"sorry (.*)",
        ["Its Alright","Its OK, Never Mind",]
    ],

]
def chat():
    print("Hey! I am S.I.F.R.A at Your Service")
    chatbot = Chat(pairs, reflections)
    chatbot.converse()

if __name__ == "__main__":
    chat()
```

[ ]:
```
import copy
final = [[1,2,3],[4,5,6],[7,8,-1]]
initial = [[1,2,3],[-1,4,6],[7,5,8]]
#function to find heuristic cost
def gn(state, finalstate):
    count = 0
    for i in range(3):
        for j in range(3):
            if(state[i][j]!=-1):
                if(state[i][j] != finalstate[i][j]):
                    count+=1
    return count
def findposofblank(state):
    for i in range(3):
        for j in range(3):
            if(state[i][j] == -1):
                return [i,j]
def move_left(state, pos):
    if(pos[1]==0):
        return None
    retarr = copy.deepcopy(state)
    retarr[pos[0]][pos[1]],retarr[pos[0]][pos[1]-1] =␣
 ↪retarr[pos[0]][pos[1]-1],retarr[pos[0]][pos[1]]
    return retarr
def move_up(state, pos):
    if(pos[0]==0):
        return None
    retarr = copy.deepcopy(state)
    #for i in state:
        #retarr.append(i)
    retarr[pos[0]][pos[1]],retarr[pos[0]-1][pos[1]] =␣
 ↪retarr[pos[0]-1][pos[1]],retarr[pos[0]][pos[1]]
    return retarr
def move_right(state, pos):
    if(pos[1]==2):
        return None
    retarr = copy.deepcopy(state)
    #for i in state:
        #retarr.append(i)
    retarr[pos[0]][pos[1]],retarr[pos[0]][pos[1]+1] =␣
 ↪retarr[pos[0]][pos[1]+1],retarr[pos[0]][pos[1]]
    return retarr
def move_down(state, pos):
    if(pos[0]==2):
        return None
    retarr = copy.deepcopy(state)
    retarr[pos[0]][pos[1]],retarr[pos[0]+1][pos[1]] =␣
 ↪retarr[pos[0]+1][pos[1]],retarr[pos[0]][pos[1]]
    return retarr
def printMatrix(matricesArray):
    print("")
    counter = 1
    for matrix in matricesArray:
        print("Step {}".format(counter))
        for row in matrix:
            print(row)
        counter+=1
        print("")
def eightPuzzle(initialstate, finalstate):
    hn=0
    explored = []
    while(True):
        explored.append(initialstate)
        if(initialstate == finalstate):
            break
        hn+=1
        left = move_left(initialstate, findposofblank(initialstate))
        right = move_right(initialstate, findposofblank(initialstate))
        up = move_up(initialstate, findposofblank(initialstate))
        down = move_down(initialstate, findposofblank(initialstate))
        fnl=1000
        fnr=1000
        fnu=1000
        fnd=1000
        if(left!=None):
            fnl = hn + gn(left,finalstate)
        if(right!=None):
            fnr = hn + gn(right,finalstate)
        if(up!=None):
            fnu = hn + gn(up,finalstate)
        if(down!=None):
            fnd = hn + gn(down,finalstate)
        minfn = min(fnl, fnr, fnu, fnd)
        if((fnl == minfn) and (left not in explored)):
            initialstate = left
        elif((fnr == minfn) and (right not in explored)):
            initialstate = right
        elif((fnu == minfn) and (up not in explored)):
            initialstate = up
        elif((fnd == minfn) and (down not in explored)):
            initialstate = down
    printMatrix(explored)
#eightPuzzle(initial, final)
def main():
    while(True):
        ch = int(input("PRESS 1 to continue and 0 to Exit : "))
        if(not ch):
            break
        start = []
        print("START STATE\n")
        for i in range(3):
            arr=[]
            for j in range(3):
                a = int(input("Enter element at  {},{}: ".format(i,j)))
                arr.append(a)
            start.append(arr)
        final = []
        print("FINAL STATE\n")
        for i in range(3):
            arr=[]
            for j in range(3):
                a = int(input("Enter element at  {},{}: ".format(i,j)))
                arr.append(a)
            final.append(arr)
        eightPuzzle(start, final)
main()
```

```
PRESS 1 to continue and 0 to Exit :  1

START STATE
```

```
[ ]:
```