xknvtqsbb

May 30, 2023

[ ]:
```
#DFS
graph = {
    'A':['B','C','D'],'B':['E'],'C':['D','E'],'D':[],'E':[]
}
visited = set()

def dfs(visited,graph,root):
    if root not in visited:
        print(root)
        visited.add(root)
        for neighbor in graph[root]:
            dfs(visited,graph,neighbor)

dfs(visited,graph,'A')
```

[ ]:
```
#DFS
graph = {
    'a':['b','c','d'],'b':['e'],'c':['d','e'],'d':[],'e':[]
}
visited = set()

def dfs(visited,graph,root):
    if root not in visited:
        print(root)
        visited.add(root)
        for neighbor in graph[root]:
            dfs(visited,graph,neighbor)

dfs(visited,graph,'a')
```

[ ]:
```
#BFS
import collections

graph = {
    0:[1,2,3],1:[0,2],2:[0,1,4],3:[0],4:[2]
}
```

```
def bfs(graph,root):
    visited = set()
    queue = collections.deque([root])
    while queue:
        vertex= queue.popleft()
        visited.add(vertex)
        for i in graph[vertex]:
            if i not in visited:
                queue.append(i)
    print(visited)

bfs(graph,0)
```

[ ]:
```
#BFS
import collections

graph = {
    0:[1,2,3],1:[0,2],2:[0,1,4],3:[0],4:[2]
}

def bfs(graph,root):
    visited = set()
    queue = collections.deque([root])

    while queue:
        vertex = queue.popleft()
        visited.add(vertex)

        for i in graph[vertex]:
            if i not in visited:
                queue.append(i)
    print(visited)

bfs(graph,0)
```

[ ]:
```
#Chatbot
from nltk.chat.util import Chat,reflections
reflections
pairs=[
    ['Hello',['Hi!, How can I help you?']],
    ['Need help',['How can I help you?']],
    ['I am Niraj',['Nice to hear that']],
    ['What is your name?',['I am Chatbot and here to help you']],
    ['What is Chatbot?',['Chatbot is python program to help you']]
]

chat=Chat(pairs,reflections)
```

```
chat.converse()
```

[ ]:
```
#chatbot
from nltk.chat.util import Chat,reflections
reflections
pairs=[
    ['Hello',['Hi!, How can I help you?']],
    ['Need help',['How can I help you?']],
    ['I am Niraj',['Nice to hear that']],
    ['What is your name?',['I am Chatbot and here to help you']],
    ['What is Chatbot?',['Chatbot is python program to help you']]
]
chat=Chat(pairs,reflections)
chat.converse()
```

[5]:
```
#n-queen
n=int(input("Enter the value of n : "))
board=[[0 for i in range(n)]for i in range(n)]
def check_column(board,row,column):
    for i in range(row,-1,-1):
        if board[i][column]==1:
            return False
    return True
def check_diagonal(board,row,column):
    for i,j in zip(range(row,-1,-1),range(column,-1,-1)):
        if board[i][j]==1:
            return False
    for i,j in zip(range(row,-1,-1),range(column,n)):
        if board[i][j]==1:
            return False
    return True
#backtracking
def nqn(board,row):
    if row==n:
        return True
    for i in range(n):
        if(check_column(board,row,i)==True and␣
 ↪check_diagonal(board,row,i)==True):
            board[row][i]=1
            if nqn(board,row+1):
                return True
            board[row][i]=0
    return False
nqn(board,0)
for row in board:
    print(row)
```

```
Enter the value of n : 8
[1, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 1, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 1]
[0, 0, 0, 0, 0, 1, 0, 0]
[0, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 0]
[0, 1, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 1, 0, 0, 0, 0]
```

[10]:
```
#n-queen
n=int(input("Enter the value of n:"))

board=[[0 for i in range(n)]for i in range(n)]

def check_column(board,row,column):
    for i in range(row,-1,-1):
        if board[i][column]==1:
            return False
    return True

def check_diagonal(board,row,column):
    for i,j in zip(range(row,-1,-1),range(column,-1,-1)):
        if board[i][j]==1:
            return False
    for i,j in zip(range(row,-1,-1),range(column,n)):
        if board[i][j]==1:
            return False
    return True
#backtrack
def nqn(board,row):
    if row==n:
        return True
    for i in range(n):
        if(check_column(board,row,i)==True and␣
 ↪check_diagonal(board,row,i)==True):
            board[row][i]=1
            if nqn(board,row+1):
                return True
            board[row][i]=0
    return False
nqn(board,0)
for row in board:
    print(row)
```

```
Enter the value of n:9
[1, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 1, 0, 0, 0, 0, 0, 0]
```

```
[0, 0, 0, 0, 0, 1, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 1, 0]
[0, 1, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 1]
[0, 0, 0, 0, 0, 0, 1, 0, 0]
[0, 0, 0, 0, 1, 0, 0, 0, 0]
```

[6]:
```
#sel-sort
arr=[]
n=int(input("Number of elements in array:"))
for i in range(0,n):
    l=int(input())
    arr.append(l)
for i in range(len(arr)):
    min_idx=i
    for j in range(i+1,len(arr)):
        if arr[min_idx]>arr[j]:
            min_idx = j
    arr[i], arr[min_idx]= arr[min_idx], arr[i]
print("Sorted array is ",arr)
```

```
Number of elements in array:4
45
2
78
9
Sorted array is  [2, 9, 45, 78]
```

[8]:
```
#sel-sort
arr=[]
n=int(input("Enter no of elments :"))
for i in range(0,n):
    l=int(input())
    arr.append(l)

for i in range(len(arr)):
    min_idx=i
    for j in range(i+1,len(arr)):
        if arr[min_idx]>arr[j]:
            min_idx=j

    arr[i],arr[min_idx]=arr[min_idx],arr[i]

print("Sorted array :",arr)
```

```
Enter no of elments :4
23
```

```
78
21
3
Sorted array : [3, 21, 23, 78]
```

[1]:
```python
#A-star
from queue import PriorityQueue

def a_star(start_state, goal_state, heuristic_fn, get_neighbors_fn):
    open_set = PriorityQueue()
    open_set.put((0, start_state))

    came_from = {}
    g_score = {start_state: 0}
    f_score = {start_state: heuristic_fn(start_state, goal_state)}

    while not open_set.empty():
        current = open_set.get()[1]

        if current == goal_state:
            return reconstruct_path(came_from, current)

        for neighbor in get_neighbors_fn(current):
            tentative_g_score = g_score[current] + 1  # Assuming uniform cost␣
 ↪for transitions

            if neighbor not in g_score or tentative_g_score < g_score[neighbor]:
                came_from[neighbor] = current
                g_score[neighbor] = tentative_g_score
                f_score[neighbor] = tentative_g_score + heuristic_fn(neighbor,␣
 ↪goal_state)
                open_set.put((f_score[neighbor], neighbor))

    return None  # No path found


def reconstruct_path(came_from, current):
    path = [current]

    while current in came_from:
        current = came_from[current]
        path.insert(0, current)

    return path


# Example implementation of a game search problem
```

```python

# Heuristic function (Manhattan distance)
def heuristic(state, goal_state):
    x1, y1 = state
    x2, y2 = goal_state
    return abs(x1 - x2) + abs(y1 - y2)

# Get neighbor function (possible moves)
def get_neighbors(state):
    x, y = state
    neighbors = []

    # Add possible moves (up, down, left, right)
    if x > 0:
        neighbors.append((x - 1, y))
    if x < 3:
        neighbors.append((x + 1, y))
    if y > 0:
        neighbors.append((x, y - 1))
    if y < 3:
        neighbors.append((x, y + 1))

    return neighbors


# Example usage
start_state = (0, 0)
goal_state = (3, 3)

path = a_star(start_state, goal_state, heuristic, get_neighbors)

if path is not None:
    print("Path found:")
    for step in path:
        print(step)
else:
    print("No path found.")
```

```
Path found:
(0, 0)
(0, 1)
(0, 2)
(0, 3)
(1, 3)
(2, 3)
(3, 3)
```

[ ]:
```python
#A-star
from queue import PriorityQueue

def a_star(start_state, goal_state, heuristic_fn, get_neighbors_fn):
    open_set = PriorityQueue()
    open_set.put((0, start_state))

    came_from = {}
    g_score = {start_state: 0}
    f_score = {start_state: heuristic_fn(start_state, goal_state)}

    while not open_set.empty():
        current = open_set.get()[1]

        if current == goal_state:
            return reconstruct_path(came_from, current)

        for neighbor in get_neighbors_fn(current):
            tentative_g_score = g_score[current] + 1  # Assuming uniform cost␣
 ↪for transitions

            if neighbor not in g_score or tentative_g_score < g_score[neighbor]:
                came_from[neighbor] = current
                g_score[neighbor] = tentative_g_score
                f_score[neighbor] = tentative_g_score + heuristic_fn(neighbor,␣
 ↪goal_state)
                open_set.put((f_score[neighbor], neighbor))

    return None  # No path found


def reconstruct_path(came_from, current):
    path = [current]

    while current in came_from:
        current = came_from[current]
        path.insert(0, current)

    return path


# Example implementation of a game search problem

# Heuristic function (Manhattan distance)
def heuristic(state, goal_state):
    x1, y1 = state
    x2, y2 = goal_state
```

```
    return abs(x1 - x2) + abs(y1 - y2)

# Get neighbor function (possible moves)
def get_neighbors(state):
    x, y = state
    neighbors = []

    # Add possible moves (up, down, left, right)
    if x > 0:
        neighbors.append((x - 1, y))
    if x < 3:
        neighbors.append((x + 1, y))
    if y > 0:
        neighbors.append((x, y - 1))
    if y < 3:
        neighbors.append((x, y + 1))

    return neighbors


# Example usage
start_state = (0, 0)
goal_state = (3, 3)

path = a_star(start_state, goal_state, heuristic, get_neighbors)

if path is not None:
    print("Path found:")
    for step in path:
        print(step)
else:
    print("No path found.")
```

[ ]:
```
#Expert-System
SYMPTOMS = {
    "fever": ["flu", "pneumonia", "COVID-19"],
    "cough": ["flu", "pneumonia", "COVID-19"],
    "shortness of breath": ["pneumonia", "COVID-19"],
    "fatigue": ["flu", "pneumonia", "COVID-19"],
    "body aches": ["flu"],
    "sore throat": ["flu", "COVID-19"],
    "headache": ["flu"],
    "loss of smell or taste": ["COVID-19"],
    "diarrhea": ["COVID-19"],
}

TREATMENTS = {
```

```
    "flu": ["get rest", "drink fluids", "take over-the-counter medication"],
    "pneumonia": ["antibiotics", "oxygen therapy", "hospitalization"],
    "COVID-19": ["quarantine", "symptomatic treatment", "seek medical attention␣
 ↪if symptoms worsen"],
}

def triage(symptoms):
    """Given a set of symptoms, returns a list of potential medical conditions␣
 ↪and recommended treatments."""
    conditions = set()
    for symptom in symptoms:
        if symptom in SYMPTOMS:
            conditions.update(SYMPTOMS[symptom])
    treatments = []
    for condition in conditions:
        if condition in TREATMENTS:
            treatments.extend(TREATMENTS[condition])
    return list(conditions), treatments

# Prompt user for symptoms
patient_symptoms = input("What are your symptoms? (separate with commas) ").
 ↪split(",")

# Remove any leading/trailing white space from symptoms
patient_symptoms = [symptom.strip() for symptom in patient_symptoms]

conditions, treatments = triage(patient_symptoms)
print("Potential conditions:", conditions)
print("Recommended treatments:", treatments)
```

[2]:
```
#Expert-System
SYMPTOMS = {
    "fever": ["flu", "pneumonia", "COVID-19"],
    "cough": ["flu", "pneumonia", "COVID-19"],
    "shortness of breath": ["pneumonia", "COVID-19"],
    "fatigue": ["flu", "pneumonia", "COVID-19"],
    "body aches": ["flu"],
    "sore throat": ["flu", "COVID-19"],
    "headache": ["flu"],
    "loss of smell or taste": ["COVID-19"],
    "diarrhea": ["COVID-19"],
}

TREATMENTS = {
    "flu": ["get rest", "drink fluids", "take over-the-counter medication"],
    "pneumonia": ["antibiotics", "oxygen therapy", "hospitalization"],
```

```
    "COVID-19": ["quarantine", "symptomatic treatment", "seek medical attention␣
 ↪if symptoms worsen"],
}

def triage(symptoms):
    """Given a set of symptoms, returns a list of potential medical conditions␣
 ↪and recommended treatments."""
    conditions = set()
    for symptom in symptoms:
        if symptom in SYMPTOMS:
            conditions.update(SYMPTOMS[symptom])
    treatments = []
    for condition in conditions:
        if condition in TREATMENTS:
            treatments.extend(TREATMENTS[condition])
    return list(conditions), treatments

# Prompt user for symptoms
patient_symptoms = input("What are your symptoms? (separate with commas) ").
 ↪split(",")

# Remove any leading/trailing white space from symptoms
patient_symptoms = [symptom.strip() for symptom in patient_symptoms]

conditions, treatments = triage(patient_symptoms)
print("Potential conditions:", conditions)
print("Recommended treatments:", treatments)
```

```
What are your symptoms? (separate with commas) fever
Potential conditions: ['COVID-19', 'flu', 'pneumonia']
Recommended treatments: ['quarantine', 'symptomatic treatment', 'seek medical
attention if symptoms worsen', 'get rest', 'drink fluids', 'take over-the-
counter medication', 'antibiotics', 'oxygen therapy', 'hospitalization']
```

```
[ ]:
```