ai

May 6, 2025

[1]:
```
graph = {
    'A': ['B','C'],
    'B': ['D','E'],
    'C': ['F','G'],
    'D': ['H','I'],
    'E': [],
    'F': ['J','K'],
    'G': [],
    'H': [],
    'I': [],
    'J': [],
    'K': []
}

# BFS
def bfs(graph, start):
    visited, queue = set(), [start]
    while queue:
        node = queue.pop(0)
        if node not in visited:
            print(node, end=' ')
            visited.add(node)
            queue.extend(graph[node])

# DFS - Modified to print vertically
def dfs(graph, node, visited=set()):
    if node not in visited:
        print(node)  # Print node vertically
        visited.add(node)
        for neighbor in graph[node]:
            dfs(graph, neighbor, visited)

print("BFS:")
bfs(graph, 'A')

print("\nDFS (Vertical):")
dfs(graph, 'A')
```

```
BFS:
A B C D E F G H I J K
DFS (Vertical):
A
B
D
H
I
E
C
F
J
K
G
```

[ ]:
```python
#AI - 2
import heapq

# Goal configuration for reference
goal_state = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 0]
]

# Directions to move: up, down, left, right
moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]

# Manhattan distance heuristic
def manhattan_distance(state):
    distance = 0
    for i in range(3):
        for j in range(3):
            value = state[i][j]
            if value != 0:
                goal_x = (value - 1) // 3
                goal_y = (value - 1) % 3
                distance += abs(i - goal_x) + abs(j - goal_y)
    return distance

# Check if a state is the goal
def is_goal(state):
    return state == goal_state

# Convert state to a hashable form (tuple of tuples)
def state_to_tuple(state):
    return tuple(tuple(row) for row in state)
```

```
# Find position of 0
def find_zero(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j

# Generate next valid states
def get_neighbors(state):
    neighbors = []
    x, y = find_zero(state)
    for dx, dy in moves:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            new_state = [row[:] for row in state]
            new_state[x][y], new_state[nx][ny] = new_state[nx][ny],␣
 ↪new_state[x][y]
            neighbors.append(new_state)
    return neighbors

# A* algorithm
def a_star(start_state):
    start_tuple = state_to_tuple(start_state)
    heap = [(manhattan_distance(start_state), 0, start_state, [])]  # (f, g,␣
 ↪state, path)
    visited = set()

    while heap:
        f, g, current, path = heapq.heappop(heap)

        if state_to_tuple(current) in visited:
            continue

        visited.add(state_to_tuple(current))

        if is_goal(current):
            return path + [current]

        for neighbor in get_neighbors(current):
            if state_to_tuple(neighbor) not in visited:
                new_g = g + 1
                new_f = new_g + manhattan_distance(neighbor)
                heapq.heappush(heap, (new_f, new_g, neighbor, path + [current]))

    return None
```

```
# Example Usage
start_state = [
    [1, 2, 3],
    [4, 0, 6],
    [7, 5, 8]
]

solution = a_star(start_state)

# Print the solution path
if solution:
    print("Solution found in", len(solution) - 1, "moves:")
    for step in solution:
        for row in step:
            print(row)
        print()
else:
    print("No solution found.")
```

```
Solution found in 2 moves:
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]

[1, 2, 3]
[4, 5, 6]
[7, 0, 8]

[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
```

[8]:
```
#AI - 3

#SELECTION SORT
def selection_sort(arr):
    n=len(arr)
    for i in range(n):
        min_ind = i
        for j in range(i+1,n):
            if arr[j] < arr[min_ind]:
                min_ind = j
        arr[i], arr[min_ind] = arr[min_ind], arr[i]
    return arr

arr = [10,14,5,20,6]
```

```
sorted_arr = selection_sort(arr)
print("Sorted array is : " ,sorted_arr)

#PRIMS ALGORITHM
import heapq

def prim(graph, start):
    mst = []
    visited = set([start])
    edges = [(cost, start, to) for to, cost in graph[start].items()]
    heapq.heapify(edges)

    while edges:
        cost, frm, to = heapq.heappop(edges)
        if to not in visited:
            visited.add(to)
            mst.append((frm, to, cost))

            for to_next, cost2 in graph[to].items():
                if to_next not in visited:
                    heapq.heappush(edges, (cost2, to, to_next))

    return mst
graph = {
    'A': {'B': 2, 'C': 3},
    'B': {'A': 2, 'C': 1, 'D': 15},
    'C': {'A': 3, 'B': 1, 'D': 4},
    'D': {'B': 15, 'C': 4}
}

print(prim(graph, 'A'))
```

```
Sorted array is :  [5, 6, 10, 14, 20]
[('A', 'B', 2), ('B', 'C', 1), ('C', 'D', 4)]
```

[1]:
```
#AI-4
print("Enter the number of queens: ")
N=int(input())

#Chessboard:
board=[[0]*N for _ in range(N)]
def is_attack(i,j):
    for k in range(0,N):
        if board [i][k]==1 or board[k][j]==1:
            return False
    #Checking Diagonals:
    for k in range(0,N):
```

```
        for l in range(0,N):
            if(k+l==i+j) or (k-l==i-j):
                if board[k][l]==1:
                    return False
    return True
def N_queen(n):
    if n==0:
        return True
    for i in range(0,N):
        for j in range(0,N):
            '''checking if we can place a queen here or no queen wi'''
            if (not(is_attack(i,j))) and (board[i][j]!=1):
                board[i][j]=1
                #Recursion
                if N_queen(n-1)==True:
                    return True
                board[i][j]=0
    return False
N_queen(N)
for i in board:
    print(i)
```

```
Enter the number of queens:
[0, 0, 0, 0]
[0, 0, 0, 0]
[0, 0, 0, 0]
[0, 0, 0, 0]
```

[5]:
```
#AI-5
def chat_box():
    print("Hi! I am Chatbox. How can I help you today?")
    while True:
        user_input = input("You: ").strip().lower()

        if user_input == "bye":
            print("Chatbox: Bye! Have a great day.")
            break
        elif user_input == "hello":
            print("Chatbox: Hello there! How are you doing today?")
        elif user_input == "how are you?":
            print("Chatbox: I'm doing great, thanks for asking. What would you␣
 ↪like to see?")
        elif user_input == "phone":
            print("Chatbox: Which brand are you interested in?")
        elif user_input == "apple":
            print("Chatbox: What is your budget?")
        elif user_input == "100000":
```

```
            print("Chatbox: Here are some models based on the information we␣
 ↪have.")
        elif user_input == "thank you":
            print("Chatbox: You're welcome!")
        else:
            print("Chatbox: I didn't understand that. Could you please rephrase?
 ↪")

chat_box()
```

```
Hi! I am Chatbox. How can I help you today?
Chatbox: Hello there! How are you doing today?
Chatbox: Hello there! How are you doing today?
Chatbox: I'm doing great, thanks for asking. What would you like to see?
Chatbox: Which brand are you interested in?
Chatbox: What is your budget?
Chatbox: What is your budget?
Chatbox: Here are some models based on the information we have.
Chatbox: I didn't understand that. Could you please rephrase?
Chatbox: You're welcome!
Chatbox: Bye! Have a great day.
```

[6]:
```
def restaurant_chatbot():
    print("  Hello! Welcome to ChatBite Restaurant.")
    print("I can help you book a table. Let's get started!\n")

    name = input("What's your name? ")
    date = input("On what date would you like to book a table? (e.g.␣
 ↪2025-04-15) ")
    time = input("What time? (e.g. 7:30 PM) ")
    guests = input("How many people? ")

    print("\nThanks, " + name + "!")
    print(f"  Your table for {guests} people has been booked on {date} at␣
 ↪{time}.")
    print("We look forward to serving you!  ")

# Run the chatbot
restaurant_chatbot()
```

```
  Hello! Welcome to ChatBite Restaurant.
I can help you book a table. Let's get started!


Thanks, !
  Your table for 2 people has been booked on 2025-05-05 at 7:30 PM.
We look forward to serving you!
```

[1]:
```
#AI-6
 #Knowledge Base
positive_criteria = {
    "exceeds_deadlines": 5,
    "high_quality_work": 4,
    "great_team_player": 3,
    "consistently_improves": 2,
    "excellent_communication": 1
}

negative_criteria = {
    "misses_deadlines": -5,
    "poor_quality_work": -4,
    "poor_team_player": -3,
    "resistant_to_feedback": -2,
    "poor_communication": -1
}

# Rule Engine
def evaluate_employee(positives, negatives):
    score = 0
    for pos in positives:
        score += positive_criteria.get(pos, 0)

    for neg in negatives:
        score += negative_criteria.get(neg, 0)

    return score

# User Interface
def user_interface():
    print("Employee Performance Evaluation")

    print("Please enter positive criteria from the list below, separated by␣
 ↪commas:")
    for criterion in positive_criteria:
        print(f"- {criterion}")

    print("\n")
    positives_input = input("> ")

    print("\nPlease enter negative criteria from the list below, separated by␣
 ↪commas:")
    for criterion in negative_criteria:
        print(f"- {criterion}")

    print("\n")
```

```
    negatives_input = input("> ")
    positives = [item.strip() for item in positives_input.split(",")]
    negatives = [item.strip() for item in negatives_input.split(",")]

    score = evaluate_employee(positives, negatives)

    print(f"\nEmployee Score: {score}\n")

    if score > 10:
        print("Performance: Outstanding")
    elif score > 5:
        print("Performance: Good")
    elif score > 0:
        print("Performance: Satisfactory")
    else:
        print("Performance: Needs Improvement")

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

```
Employee Performance Evaluation
Please enter positive criteria from the list below, separated by commas:
- exceeds_deadlines
- high_quality_work
- great_team_player
- consistently_improves
- excellent_communication



Please enter negative criteria from the list below, separated by commas:
- misses_deadlines
- poor_quality_work
- poor_team_player
- resistant_to_feedback
- poor_communication



Employee Score: 4

Performance: Satisfactory
```