**Assignment No 1 DFS BFS**  
from collections import deque

# Create the graph using user input

graph = {}

num\_nodes = int(input("Enter number of nodes: "))

print("Enter node names one by one:")

nodes = []

for i in range(num\_nodes):

node = input(f"Name of node {i+1}: ")

nodes.append(node)

graph[node] = []

print("\nNow enter neighbors for each node:")

for node in nodes:

neighbors = input(f"Enter neighbors of {node} (separated by space): ").split()

graph[node] = neighbors

# DFS Function

def dfs(graph, start, visited=None):

if visited is None:

visited = set()

visited.add(start)

print(start, end=' ')

for neighbor in graph[start]:

if neighbor not in visited:

dfs(graph, neighbor, visited)

# BFS Function

def bfs(graph, start):

visited = set()

queue = deque([start])

visited.add(start)

while queue:

node = queue.popleft()

print(node, end=' ')

for neighbor in graph[node]:

if neighbor not in visited:

visited.add(neighbor)

queue.append(neighbor)

# Start traversal

start\_node = input("\nEnter starting node for traversal: ")

print("\nDFS Traversal:")

dfs(graph, start\_node)

print("\n\nBFS Traversal:")

bfs(graph, start\_node)

**Assignment No 2 : A\***

import heapq

# Goal state

goal = [[1, 2, 3],

[4, 5, 6],

[7, 8, 0]]

# Heuristic: Manhattan Distance

def heuristic(state):

dist = 0

for i in range(3):

for j in range(3):

val = state[i][j]

if val != 0:

goal\_i, goal\_j = divmod(val - 1, 3)

dist += abs(i - goal\_i) + abs(j - goal\_j)

return dist

# Generate neighboring states by moving the blank (0)

def get\_neighbors(state):

neighbors = []

for i in range(3):

for j in range(3):

if state[i][j] == 0:

x, y = i, j

break

directions = [(-1,0), (1,0), (0,-1), (0,1)] # Up, Down, Left, Right

for dx, dy in directions:

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

# Convert state to hashable format (tuple of tuples)

def serialize(state):

return tuple(tuple(row) for row in state)

# A\* Search Function

def a\_star(start):

open\_set = []

heapq.heappush(open\_set, (heuristic(start), 0, start))

came\_from = {}

g\_score = {serialize(start): 0}

while open\_set:

\_, cost, current = heapq.heappop(open\_set)

if current == goal:

path = [current]

while serialize(current) in came\_from:

current = came\_from[serialize(current)]

path.append(current)

return path[::-1]

for neighbor in get\_neighbors(current):

temp\_g = g\_score[serialize(current)] + 1

key = serialize(neighbor)

if key not in g\_score or temp\_g < g\_score[key]:

came\_from[key] = current

g\_score[key] = temp\_g

f = temp\_g + heuristic(neighbor)

heapq.heappush(open\_set, (f, temp\_g, neighbor))

return None

# Initial puzzle state

start = [[1, 2, 3],

[4, 0, 6],

[7, 5, 8]]

# Run A\* and print path

solution = a\_star(start)

for step in solution:

for row in step:

print(row)

print()

Explaination:

* **What is this code doing?**
  + It **solves the 8-puzzle problem** (sliding tile puzzle) using **A\* (A-star)** search algorithm.
* **What is the goal?**
  + Reach the state:

1 2 3

4 5 6

7 8 0

* **How does it work?**
  + heuristic(): Measures how far the current state is from the goal using **Manhattan Distance**.
  + get\_neighbors(): Moves the blank tile (0) **up/down/left/right** to generate next states.
  + a\_star(): Uses a **priority queue** to always explore the most promising state next based on:
    - f = g + h
      * g = cost from start
      * h = estimated cost to goal (heuristic)
* **How is the path found?**
  + The came\_from dictionary keeps track of where each state came from.
  + When the goal is reached, we backtrack to print the full path from start to goal.

**Assignment No 3 : Selection sort**

# Function to perform selection sort

def selection\_sort(arr):

n = len(arr)

for i in range(n):

min\_index = i

# Find the smallest element in the remaining array

for j in range(i+1, n):

if arr[j] < arr[min\_index]:

min\_index = j

# Swap the smallest element with the first unsorted element

arr[i], arr[min\_index] = arr[min\_index], arr[i]

# Take user input

size = int(input("Enter the number of elements: "))

numbers = []

print("Enter the elements one by one:")

for \_ in range(size):

num = int(input())

numbers.append(num)

print("\nOriginal array:")

print(numbers)

# Sort the array

selection\_sort(numbers)

print("\nSorted array using Selection Sort:")

print(numbers)

Explaination:

 **Selection Sort** is a **Greedy algorithm** because at each step, it **greedily selects the smallest element** from the unsorted part of the array and puts it in its correct position.

 **How it works:**

* Go through the array.
* Find the **smallest** number in the remaining part.
* **Swap** it with the current index.
* Repeat for all positions.

**Assignment No 4 : Kruskal’s Algo**

# Function to find the parent of a node (for union-find)

def find(parent, i):

if parent[i] != i:

parent[i] = find(parent, parent[i])

return parent[i]

# Function to union two sets

def union(parent, rank, x, y):

xroot = find(parent, x)

yroot = find(parent, y)

if rank[xroot] < rank[yroot]:

parent[xroot] = yroot

elif rank[xroot] > rank[yroot]:

parent[yroot] = xroot

else:

parent[yroot] = xroot

rank[xroot] += 1

# Main Kruskal's algorithm

def kruskal(nodes, edges):

result = [] # Store the MST edges

edges.sort(key=lambda x: x[2]) # Sort edges by weight

parent = []

rank = []

for node in range(nodes):

parent.append(node)

rank.append(0)

for u, v, w in edges:

x = find(parent, u)

y = find(parent, v)

if x != y:

result.append((u, v, w))

union(parent, rank, x, y)

return result

# User Input

nodes = int(input("Enter number of nodes: "))

edges = []

e = int(input("Enter number of edges: "))

print("Enter edges as: node1 node2 weight")

for \_ in range(e):

u, v, w = map(int, input().split())

edges.append((u, v, w))

# Run Kruskal's algorithm

mst = kruskal(nodes, edges)

# Display result

print("\nMinimum Spanning Tree:")

total\_cost = 0

for u, v, w in mst:

print(f"{u} -- {v} == {w}")

total\_cost += w

print(f"\nTotal cost of MST: {total\_cost}")

Explaination:

### 1. ****Disjoint Set Class (Union-Find)****

class DisjointSet:

def \_\_init\_\_(self, vertices):

self.parent = {v: v for v in vertices}

self.rank = {v: 0 for v in vertices}

* This is the constructor.
* parent: Initially, every node is its own parent (self-loop).
* rank: Used for optimization. Helps keep the tree shallow.

def find(self, item):

if self.parent[item] != item:

self.parent[item] = self.find(self.parent[item])

return self.parent[item]

* This function **finds the root parent** (representative) of a node.
* It also applies **path compression**, which makes future finds faster by directly pointing nodes to the root.

def union(self, u, v):

root\_u = self.find(u)

root\_v = self.find(v)

* It finds the **root parents** of both nodes u and v.

if root\_u == root\_v:

return False

* If both nodes have the same parent, adding the edge would create a **cycle**, so it’s skipped.

if self.rank[root\_u] < self.rank[root\_v]:

self.parent[root\_u] = root\_v

elif self.rank[root\_u] > self.rank[root\_v]:

self.parent[root\_v] = root\_u

else:

self.parent[root\_v] = root\_u

self.rank[root\_u] += 1

return True

* This does the **union** of the two sets.
* The smaller tree is attached to the larger one (based on rank).
* If both have same rank, one becomes parent, and its rank increases by 1.

### 2. ****Kruskal’s Algorithm****

def kruskal(vertices, edges):

mst = []

ds = DisjointSet(vertices)

edges.sort(key=lambda x: x[2])

* Create a **Disjoint Set** instance.
* Sort the edges by **weight (greedy step)**.

for u, v, weight in edges:

if ds.union(u, v):

mst.append((u, v, weight))

* Go through each edge.
* Try to **join** the two vertices.
* If they are in different sets, join them and add edge to MST.

return mst

* After all edges are checked, return the MST.

### 3. ****Graph Input and Output****

vertices = ['A', 'B', 'C', 'D', 'E']

edges = [

('A', 'B', 1),

('A', 'C', 3),

('B', 'C', 1),

('B', 'D', 6),

('C', 'D', 4),

('C', 'E', 2),

('D', 'E', 5)

]

* Vertices and weighted edges are manually listed.

mst = kruskal(vertices, edges)

for u, v, weight in mst:

print(f"{u} - {v}: {weight}")

* Print the edges that are part of the MST and their weights.

**Assignment No 5 : Prims Algo**

import heapq

def prim(graph, start):

visited = set()

min\_heap = [(0, start)]

mst\_weight = 0

mst\_edges = []

while min\_heap:

weight, current = heapq.heappop(min\_heap)

if current in visited:

continue

visited.add(current)

mst\_weight += weight

if weight != 0:

mst\_edges.append((current, weight))

for neighbor, edge\_weight in graph[current]:

if neighbor not in visited:

heapq.heappush(min\_heap, (edge\_weight, neighbor))

return mst\_edges, mst\_weight

graph = {

'A': [('B', 1), ('C', 4)],

'B': [('A', 1), ('C', 2), ('D', 6)],

'C': [('A', 4), ('B', 2), ('D', 3)],

'D': [('B', 6), ('C', 3)]

}

edges, total\_weight = prim(graph, 'A')

print("Edges in MST:")

for node, weight in edges:

print(f"{node} with edge weight {weight}")

print(f"Total weight of MST: {total\_weight}")

Explaination :

### 🔸 Import the heap module (priority queue)

import heapq

* heapq is a module in Python to implement a **Min Heap**, which is useful to always pick the **minimum weight edge**.

### 🔸 Prim’s Algorithm Function

def prim(graph, start):

* Function prim() takes a **graph** (dictionary) and the **starting node**.

#### Initialization

visited = set() # Keeps track of visited nodes

min\_heap = [(0, start)] # Heap starts with the starting node with cost 0

mst\_weight = 0 # Total weight of MST

mst\_edges = [] # Stores edges in MST

* visited: to prevent revisiting the same node.
* min\_heap: priority queue to always choose the **minimum weight edge**.
* mst\_weight: total weight of the MST.
* mst\_edges: keeps track of the MST edges (optional).

#### Main loop

while min\_heap:

weight, current = heapq.heappop(min\_heap)

* Get the node with the **smallest edge weight** from the heap.

if current in visited:

continue

* If the node is already visited, skip it (to avoid cycles).

visited.add(current)

mst\_weight += weight

* Mark the node as visited.
* Add the edge's weight to the total MST cost.

if weight != 0:

mst\_edges.append((current, weight))

* Add the edge to the MST (we skip weight 0 because the first node doesn't have a parent edge).

#### Add neighbors to heap

for neighbor, edge\_weight in graph[current]:

if neighbor not in visited:

heapq.heappush(min\_heap, (edge\_weight, neighbor))

* Go through each neighbor of the current node.
* If the neighbor is unvisited, push the edge (with its weight) to the heap.

#### Return result

return mst\_edges, mst\_weight

* Return the list of edges in the MST and the total weight.

## 🔸 Graph Definition

graph = {

'A': [('B', 1), ('C', 4)],

'B': [('A', 1), ('C', 2), ('D', 6)],

'C': [('A', 4), ('B', 2), ('D', 3)],

'D': [('B', 6), ('C', 3)]

}

* It's an **undirected graph** defined as an adjacency list.
* For example, 'A': [('B', 1), ('C', 4)] means:
  + There is an edge between A and B with weight 1.
  + There is an edge between A and C with weight 4.

## 🔸 Running Prim’s Algorithm

edges, total\_weight = prim(graph, 'A')

* Start building the MST from node 'A'.

## 🔸 Printing the MST

print("Edges in MST:")

for node, weight in edges:

print(f"{node} with edge weight {weight}")

print(f"Total weight of MST: {total\_weight}")

* Displays each edge added to MST and the total weight.

**Assignment No 6 : Chatbot**

class Chatbot:

def \_\_init\_\_(self):

self.responses = {

"greet": "Hello! How can I assist you today?",

"order\_status": "Please provide your order ID to check the status.",

"order\_tracking": "You can track your order using the tracking number provided in the confirmation email.",

"return\_policy": "You can return items within 30 days of purchase, as long as they are in original condition.",

"thank\_you": "You're welcome! If you have any other questions, feel free to ask.",

"bye": "Goodbye! Have a great day!",

}

def greet(self):

return self.responses["greet"]

def order\_status(self):

return self.responses["order\_status"]

def order\_tracking(self):

return self.responses["order\_tracking"]

def return\_policy(self):

return self.responses["return\_policy"]

def thank\_you(self):

return self.responses["thank\_you"]

def bye(self):

return self.responses["bye"]

def get\_response(self, user\_input):

if "hi" in user\_input.lower() or "hello" in user\_input.lower():

return self.greet()

elif "order status" in user\_input.lower():

return self.order\_status()

elif "track order" in user\_input.lower():

return self.order\_tracking()

elif "return policy" in user\_input.lower():

return self.return\_policy()

elif "thank you" in user\_input.lower():

return self.thank\_you()

elif "bye" in user\_input.lower():

return self.bye()

else:

return "I'm sorry, I didn't understand that. Can you please rephrase?"

def chat():

bot = Chatbot()

print("Customer Support Chatbot: Hello! Type 'bye' to end the conversation.")

while True:

user\_input = input("You: ")

if "bye" in user\_input.lower():

print("Customer Support Chatbot:", bot.bye())

break

response = bot.get\_response(user\_input)

print("Customer Support Chatbot:", response)

chat()

Explaination:

### 1. ****Class Definition (****Chatbot****)****:

The class Chatbot defines the behavior of the chatbot, including its responses and methods to interact with the user.

* **\_\_init\_\_(self)**:  
  This is the constructor of the Chatbot class. It initializes the responses dictionary, which contains predefined responses for various customer queries like greeting, order status, tracking orders, return policy, etc.
* **Methods (e.g., greet, order\_status, etc.)**:  
  Each method corresponds to a specific query that the chatbot can respond to. For example, the greet() method returns a greeting message when called, and order\_status() returns a message asking the user for an order ID to check the status.
* **get\_response(self, user\_input)**:  
  This method takes the user's input as a parameter and determines which predefined response to return based on the user's message. It uses simple if and elif conditions to check the presence of certain keywords (like "hi", "order status", etc.) in the user’s input. If no matching condition is found, it returns a generic message saying it didn't understand the input.

### 2. ****Function**** chat():

* This function is responsible for interacting with the user in a loop. It creates an instance of the Chatbot class and prompts the user for input.
* It prints a greeting message and enters an infinite loop (while True) where it continuously asks for user input (input("You: ")).
* The chatbot uses the get\_response() method to process the user's input and print a corresponding response.
* If the user types "bye", the loop breaks, and the chatbot says goodbye before ending the conversation.

### Example Flow:

1. The chatbot starts the conversation with a greeting: "Hello! Type 'bye' to end the conversation."
2. The user types something like "Hi, what is the order status?" The chatbot checks if the word "order status" is in the input and responds with:  
   "Please provide your order ID to check the status."
3. If the user asks something the chatbot doesn't understand, like "What's your name?", the chatbot will reply with:  
   "I'm sorry, I didn't understand that. Can you please rephrase?"
4. The user can type "bye" to end the conversation, and the chatbot will respond with a goodbye message: "Goodbye! Have a great day!".