**CODE1:-**

def dfs(arr,s):

visted1[s]=1

print(s," ")

for i in arr[s]:

if(visted1[i]==0):

dfs(arr,i)

def bfs(arr,s):

q=[]

q.append(s)

visited2[s]=1

while len(q)!=0:

pe=q.pop(0)

print(pe)

for i in arr[pe]:

if(visited2[i]==0):

visited2[i]=1;

q.append(i)

arr={

0:[1,2],

1:[0,3],

2:[0,3],

3:[1,2]

}

visted1=[0]\*len(arr)

visited2=[0]\*len(arr)

print("the Bfs of given Grap is ")

bfs(arr,0)

print("the Dfs of given Grap is ")

dfs(arr,0)

CODE2:-

import sys

def prim\_mst(graph, start\_node):

num\_vertices = len(graph)

key = [sys.maxsize] \* num\_vertices

parent = [-1] \* num\_vertices

mst\_set = [False] \* num\_vertices

key[start\_node] = 0

for \_ in range(num\_vertices - 1):

u = min\_key(key, mst\_set)

mst\_set[u] = True

for v in range(num\_vertices):

if graph[u][v] != 0 and mst\_set[v]==False and graph[u][v] < key[v]:

parent[v] = u

key[v] = graph[u][v]

print\_mst(parent, graph)

def min\_key(key, mst\_set):

min\_val = sys.maxsize

min\_index = -1

for v in range(len(key)):

if mst\_set[v]==False and key[v] < min\_val:

min\_val = key[v]

min\_index = v

return min\_index

def print\_mst(parent, graph):

print("Edge \tWeight")

for i in range(1, len(parent)):

print(parent[i], "-", i, "\t", graph[i][parent[i]])

# Example Usage

graph = [

[0, 2, 0, 6, 0],

[2, 0, 3, 8, 5],

[0, 3, 0, 0, 7],

[6, 8, 0, 0, 9],

[0, 5, 7, 9, 0]

]

start\_node = 0

prim\_mst(graph, start\_node)

EXPLINATION

Sure, let's break down the code line by line:

1. `import sys`: This line imports the sys module which provides access to some variables used or maintained by the Python interpreter and to functions that interact strongly with the interpreter.

2. `def prim\_mst(graph, start\_node):`: This line defines a function named `prim\_mst` which takes two arguments: `graph`, representing the graph in the form of an adjacency matrix, and `start\_node`, representing the starting node for the Minimum Spanning Tree (MST) algorithm.

3. `num\_vertices = len(graph)`: This line calculates the number of vertices in the graph by taking the length of the `graph` list.

4. `key = [sys.maxsize] \* num\_vertices`: This line initializes a list named `key` with all elements set to the maximum possible integer value. This list will hold the minimum weight to reach each vertex from the start node.

5. `parent = [-1] \* num\_vertices`: This line initializes a list named `parent` with all elements set to -1. This list will hold the parent node of each vertex in the MST.

6. `mst\_set = [False] \* num\_vertices`: This line initializes a list named `mst\_set` with all elements set to `False`. This list will keep track of whether a vertex is included in the MST or not.

7. `key[start\_node] = 0`: This line sets the key value of the `start\_node` to 0, indicating that the weight to reach the start node from itself is 0.

8. `for \_ in range(num\_vertices - 1):`: This line starts a loop that will run for `num\_vertices - 1` times, as an MST contains exactly `num\_vertices - 1` edges.

9. `u = min\_key(key, mst\_set)`: This line finds the vertex `u` with the minimum key value among the vertices not yet included in the MST.

10. `mst\_set[u] = True`: This line marks the vertex `u` as included in the MST.

11. `for v in range(num\_vertices):`: This line starts a loop to iterate over all vertices `v`.

12. `if graph[u][v] != 0 and mst\_set[v]==False and graph[u][v] < key[v]:`: This line checks if there is an edge between `u` and `v`, if `v` is not yet included in the MST, and if the weight of the edge `u-v` is less than the current key value of `v`.

13. `parent[v] = u`: This line updates the parent of vertex `v` to `u`, indicating that `u` is the node through which `v` is reached in the MST.

14. `key[v] = graph[u][v]`: This line updates the key value of vertex `v` to the weight of the edge `u-v`.

15. `print\_mst(parent, graph)`: This line calls the `print\_mst` function to print the edges of the MST along with their weights.

16. `def min\_key(key, mst\_set):`: This line defines a function named `min\_key` which takes two arguments: `key`, representing the key values of vertices, and `mst\_set`, representing whether a vertex is included in the MST or not.

17. `min\_val = sys.maxsize`: This line initializes a variable `min\_val` with the maximum possible integer value.

18. `min\_index = -1`: This line initializes a variable `min\_index` with -1, which will store the index of the vertex with the minimum key value.

19. The `for` loop iterates over all vertices and finds the vertex with the minimum key value among the vertices not yet included in the MST.

20. `return min\_index`: This line returns the index of the vertex with the minimum key value.

21. `def print\_mst(parent, graph):`: This line defines a function named `print\_mst` which takes two arguments: `parent`, representing the parent array, and `graph`, representing the adjacency matrix of the graph.

22. This function prints the edges of the MST along with their weights.

Overall, this code implements Prim's algorithm to find the Minimum Spanning Tree (MST) of a given graph and prints the edges of the MST along with their weights.

CODE3->

def aStarAlgo(start\_node, stop\_node):

open\_set = set(start\_node)

closed\_set = set()

g = {} # store distance from starting node

parents = {} # parents contains an adjacency map of all nodes

# ditance of starting node from itself is zero

g[start\_node] = 0

# start\_node is root node i.e it has no parent nodes

# so start\_node is set to its own parent node

parents[start\_node] = start\_node

while len(open\_set) > 0:

n = None

# node with lowest f() is found

for v in open\_set:

if n == None or g[v] + heuristic(v) < g[n] + heuristic(n):

n = v

if n == stop\_node or Graph\_nodes[n] == None:

pass

else:

for (m, weight) in get\_neighbors(n):

# nodes 'm' not in first and last set are added to first

# n is set its parent

if m not in open\_set and m not in closed\_set:

open\_set.add(m)

parents[m] = n

g[m] = g[n] + weight

# for each node m,compare its distance from start i.e g(m) to the

# from start through n node

else:

if g[m] > g[n] + weight:

# update g(m)

g[m] = g[n] + weight

# change parent of m to n

parents[m] = n

# if m in closed set,remove and add to open

if m in closed\_set:

closed\_set.remove(m)

open\_set.add(m)

if n == None:

print('Path does not exist!')

return None

# if the current node is the stop\_node

# then we begin reconstructin the path from it to the start\_node

if n == stop\_node:

path = []

while parents[n] != n:

path.append(n)

n = parents[n]

path.append(start\_node)

path.reverse()

print('Path found: {}'.format(path))

return path

# remove n from the open\_list, and add it to closed\_list

# because all of his neighbors were inspected

open\_set.remove(n)

closed\_set.add(n)

print('Path does not exist!')

return None

# define fuction to return neighbor and its distance

# from the passed node

def get\_neighbors(v):

if v in Graph\_nodes:

return Graph\_nodes[v]

else:

return None

# for simplicity we ll consider heuristic distances given

# and this function returns heuristic distance for all nodes

def heuristic(n):

H\_dist = {

'A': 11,

'B': 6,

'C': 99,

'D': 1,

'E': 7,

'G': 0,

}

return H\_dist[n]

# Describe your graph here

Graph\_nodes = {

'A': [('B', 2), ('E', 3)],

'B': [('C', 1), ('G', 9)],

'C': None,

'E': [('D', 6)],

'D': [('G', 1)],

}

aStarAlgo('A', 'G')

Explination

Sure, let's break down the code step by step:

1. `def aStarAlgo(start\_node, stop\_node):`: This line defines a function named `aStarAlgo` that takes two arguments: `start\_node` and `stop\_node`. This function implements the A\* algorithm to find the shortest path from `start\_node` to `stop\_node` in a graph.

2. `open\_set = set(start\_node)`: Initializes an empty set `open\_set` with the `start\_node` in it. This set will store nodes that are candidates for expansion.

3. `closed\_set = set()`: Initializes an empty set `closed\_set`. This set will store nodes that have been visited and whose neighbors have been explored.

4. `g = {}`: Initializes an empty dictionary `g` to store the distance from the `start\_node` to each node.

5. `parents = {}`: Initializes an empty dictionary `parents` to store the parent node for each node.

6. `g[start\_node] = 0`: Sets the distance from `start\_node` to itself as 0.

7. `parents[start\_node] = start\_node`: Sets the parent of `start\_node` as itself.

8. `while len(open\_set) > 0:`: Begins a while loop that continues until there are nodes in the `open\_set` to be explored.

9. `n = None`: Initializes a variable `n` to `None`. This will store the node with the lowest combined cost (`g(n) + heuristic(n)`).

10. `for v in open\_set:`: Iterates through each node `v` in the `open\_set`.

11. `if n == None or g[v] + heuristic(v) < g[n] + heuristic(n):`: Checks if `n` is `None` or if the cost of reaching node `v` plus the heuristic from `v` is less than the cost of reaching `n` plus the heuristic from `n`. If true, updates `n` to `v`.

12. `if n == stop\_node or Graph\_nodes[n] == None:`: Checks if the current node `n` is the `stop\_node` or if it has no neighbors. If true, passes.

13. `for (m, weight) in get\_neighbors(n):`: Iterates through the neighbors `m` of node `n` and their corresponding edge weights.

14. Inside the loop:

- If `m` is not in `open\_set` and not in `closed\_set`, it's added to `open\_set` with its parent set to `n`, and its distance (`g`) updated.

- If `m` is already in `open\_set` or `closed\_set`, it checks if the path through `n` to `m` is shorter. If so, it updates `g`, `parents`, and moves `m` from `closed\_set` to `open\_set`.

15. `if n == None:`: Checks if `n` is still `None`, meaning no path exists.

16. If `n` is the `stop\_node`, it reconstructs the path from `start\_node` to `stop\_node` using the `parents` dictionary and prints it.

17. Finally, it removes `n` from `open\_set` and adds it to `closed\_set`.

18. If no path is found, it prints "Path does not exist!" and returns `None`.

19. The functions `get\_neighbors(v)` and `heuristic(n)` are defined later in the code. `get\_neighbors` returns the neighbors of a given node `v` along with their respective edge weights, and `heuristic` returns the heuristic distance for a given node `n`.

20. The graph `Graph\_nodes` is defined at the end of the code. It represents the adjacency list representation of the graph, where keys are nodes and values are lists of tuples representing neighbors and edge weights.

Overall, this code implements the A\* algorithm for finding the shortest path in a graph from a start node to a stop node.