**Optimizing real road networks by applying time efficient shortest path algorithms**

A Mid Semester Report

in Partial Fulfillment of the Requirements

for the Course of

# Minor Project - I

In

Third year – Fifth Semester of

**Bachelor of Technology**

specialization

In

# Business Analytics and Optimization

Under

**Mr.Amit Singh**

By

**500067103 R103218043 Divyam Verma**

**500069069 R103218081 Mayank Rastogi**

**500067983 R103218037 Devansh Messon**



DEPARTMENT OF INFORMATICS

SCHOOL OF COMPUTER SCIENCE

UNIVERSITY OF PETROLEUM AND ENERGY STUDIES, BIDHOLI, DEHRADUN, UTTRAKHAND, INDIA

**Problem Statement**

In this pandemic, many people encountered various lethal health problems, therefore it becomes an emergency case and such situation expects an ambulance to reach the patient’s place as soon as possible. Obviously, ambulance must take the shortest possible path from the hospital to patient’s place in the real road network given in the form of a graph to reach the destination faster to

save a life.

**Objective**

Our objective is to suggest the ambulance a shortest possible path as soon as possible in the real road network by applying time efficient shortest path algorithms.

For single destination: determine the shortest possible path from source to destination.

For multiple destinations: determine a route such that the vehicle visits all the destinations and comes back to the source location in the least possible time.

**Methodology**

**For Dijkstra algorithm:**

4 solving optimizations are being proposed:

* Brute force implementation
* Binary heap implementation
* Fibonacci heap implementation
* Bi-directional Dijkstra algorithm

**For solving Travelling Salesman Problem:**

1 solving optimization is being proposed:

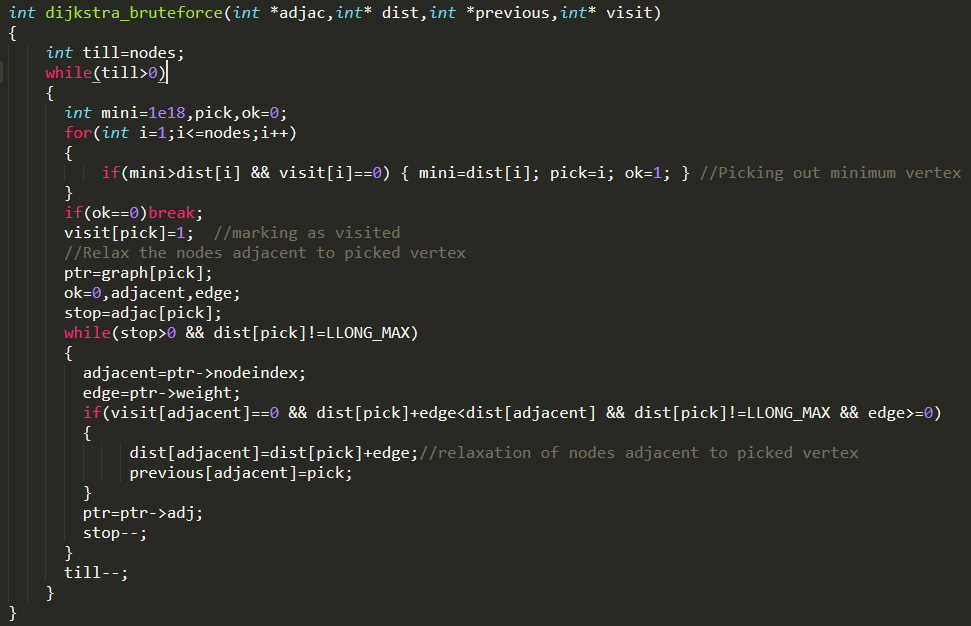
* Backtracking/brute force implementation
* Dynamic programming method

**Dijkstra algorithm :**

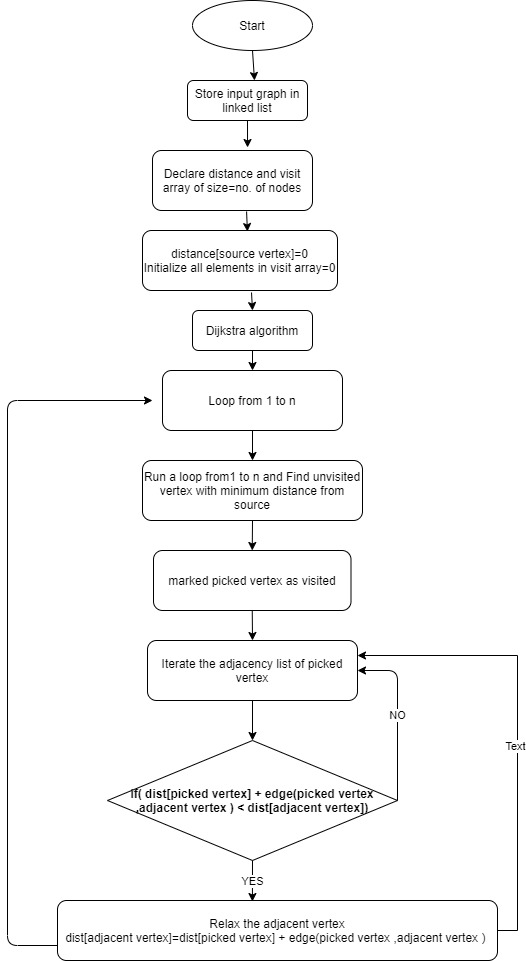
**1.Bruteforce Algorithm**

1. Distance of nodes(except source node) from source node is infinity.
2. Find the unvisited node with minimum distance by running a O(n) loop.
3. Mark the picked node as visited.
4. Relax the nodes adjacent to picked node.
5. Adjacent nodes which are marked as visited cannot be relaxed further.
6. Relaxing condition:
7. if( dist[picked vertex] + edge(picked vertex , adjacent vertex ) < dist[adjacent vertex])
8. Repeat the above steps n times or break if there is no vertex found with minimum distance.

**Implementation of Bruteforce Algorithm**



**Flowchart of Bruteforce Algorithm**

****

**2.Binary Heap Implementation**

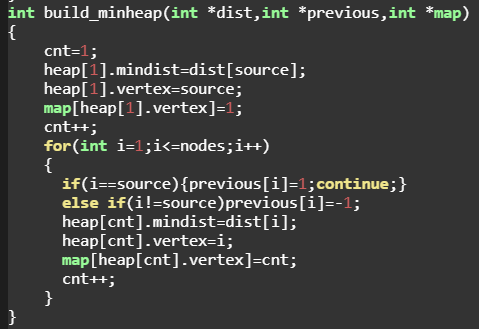
**(a).Building Minimum Heap:**

1.Distance of nodes from source node is infinity except source node.

2.Put distance[source node]=0 as root of the heap.

3.Traverse the heap array from 2nd index and put distance of all other nodes.

**Implementation of Building Minimum Heap**

****

**(b)Deleting Minimum Node from binary heap:**

1.Move the last element of heap to 1st index of heap array.

2. Rearrange the heap such that min. element Is on top.

3.Start from top element:

4.Child1=2\*(parent’s index)+1

5.Child2=2\*(parent’s index)+2

6.If distances[children] < distance[parent]:

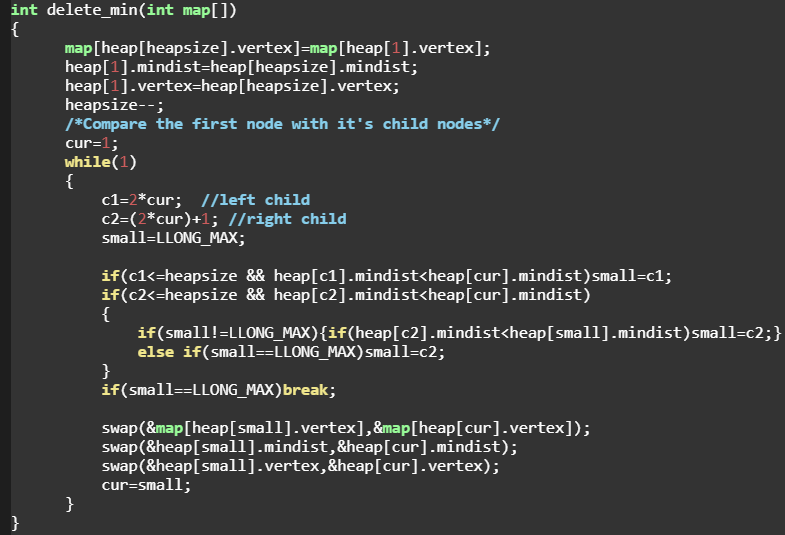
7. Take min. of distances of Child1 , Child2 and swap the minimum one with the parent node.

8. Make the minimum element as parent and

9. Repeat the above steps.

10.Else: Do nothing

**Implementation of Deleting Minimum Node from binary heap:**

******

**(c)Decrease a value of ith element in heap:**

1.Map array is used to get the position of a particular element in heap.

2.Assign value of ith element=Decreased value

3.Heap might get disturbed so rearrange it.

4.Start from the ith element and go upwards In heap.Compare it’s value with parent.

5.Parent’s index in heap=i / 2;

6.while Parent’s value>ith element’s value:

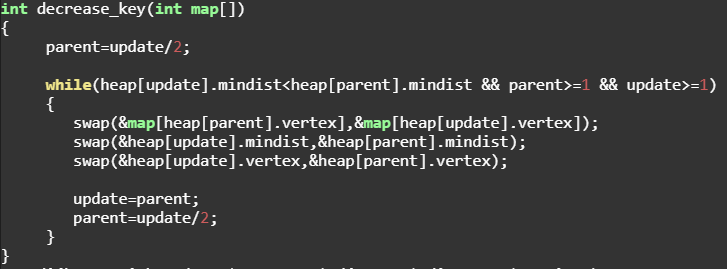
7. -swap(parent,ith element)

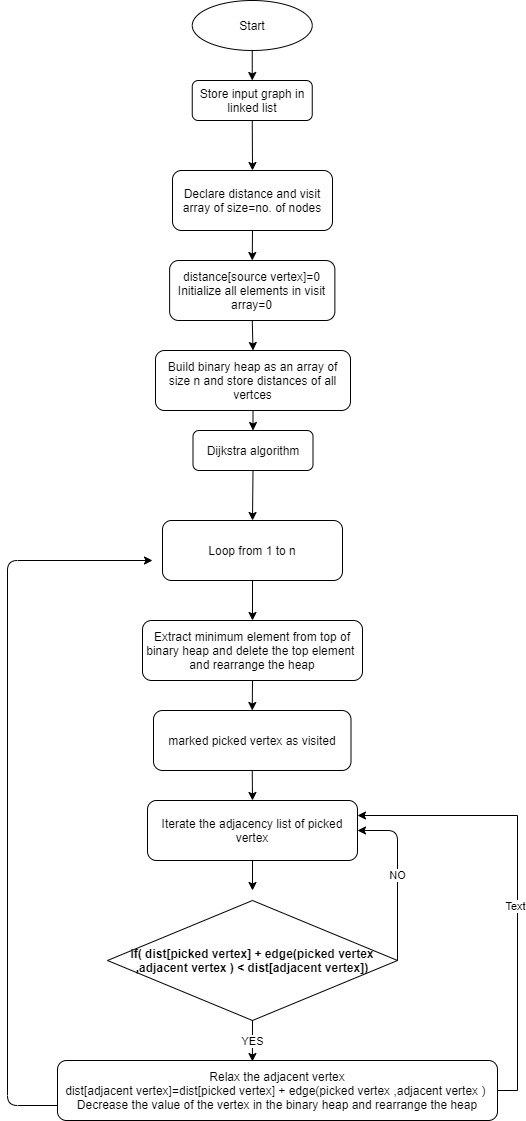
8 -update the map array

9 -ith element=parent(going upwards).

Time-Complexity-O(LogN)

**Implementation of Decrease a value of ith element in heap:**

****

****

**Flowchart for binary heap implementation**

1. **Fibonacci Heap Implementation-**

**(a)Building fibonacci heap:**

1.Allocate a new node.

2.Make its parent,child,left,right as NULL.

3.degree,mark=0

4.Distance of nodes from source node is infinity except source node.

5.If heapsize==0:

6.Minimum element=input

7.if heapsize>0:

8. Minimum element=min(Minimum element,input)

Time-Complexity-O(n)

**Implementation of Building fibonacci heap:**

******

**(B)Deleting Minimum element from Fibonacci heap:**

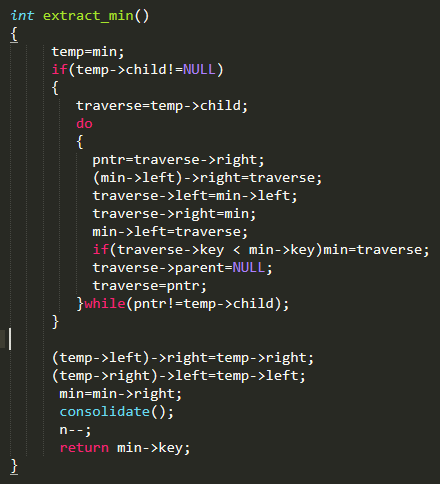
1. Unlink the minimum node from the root list and add all its children to the root list.
2. Degree of node=number of children.
3. All Subtrees should have unique degree.
4. Max. Degree in the Root list= Log2 (N)
5. Declare array pointer which is pointing to node which has degree=array’s index
6. Traverse the root list of Fibonacci heap.
7. If array[current node’s degree]==NULL:
8. then array[current node’s degree] will point to current node.
9. If array[current node’s degree]!=NULL:
10. If array[current node’s degree] value > current node value
11. then make current node as parent of array[current node’s degree]

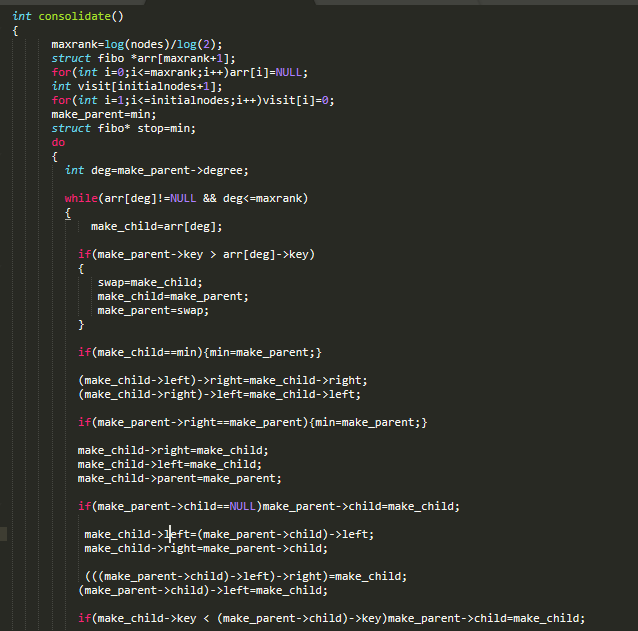
array[current node’s degree] will point to current node

1. else vice versa
2. After every subtree has unique degree in the Fibonacci heap.

14.Traverse the array pointer and find the minimum element of the Fibonacci heap

**Implementation of Minimum element from Fibonacci heap:**

******

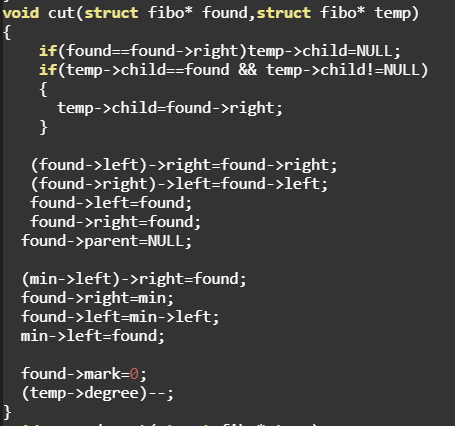
******

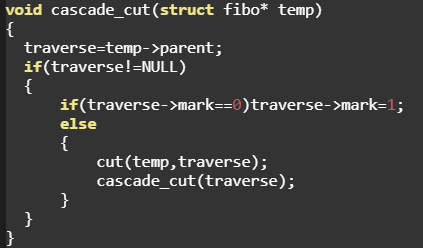
**(C)Decreasing key of Fibonacci heap:**

1. Map array is used to get the position of a particular element in heap.
2. Assign:
3. value of ith element=Decreased value.
4. **CUT Function():**
5. If value of parent of ith node > value of ith node:
6. then : Move ith node to the root list and mark it 0.
   1. Degree of parent is reduced by 1
7. **CASCADE Function():**
8. If parent of ith node is marked as 0:
9. then parent of ith node is marked as 1
10. else if parent of ith node is already marked as 1.
11. (Recursion)
12. then: CUT()

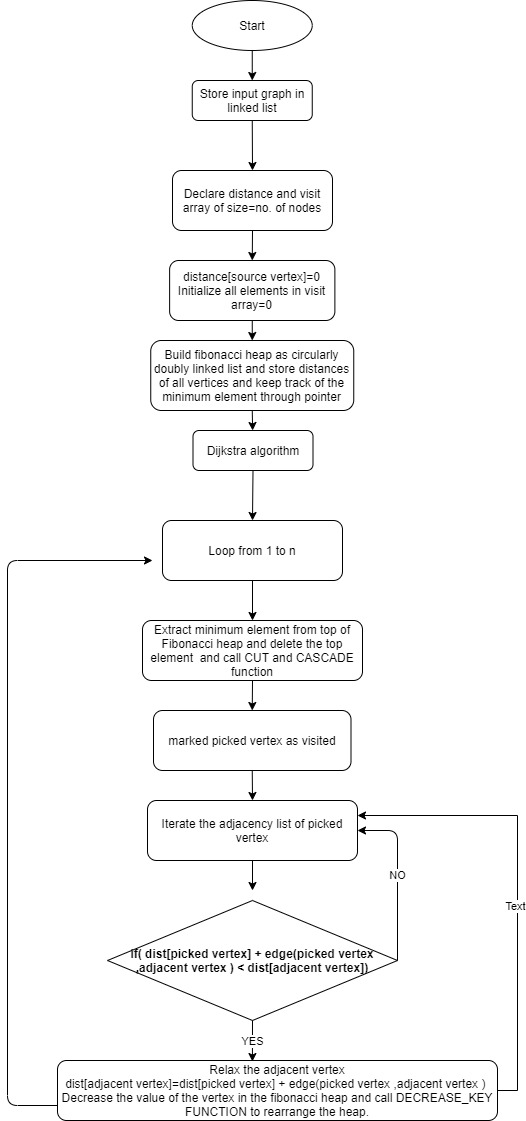
CASCADE\_CUT()

**Implementation of decreasing key of fibonacci heap:**

******



**Flowchart of Fibonacci heap**



**Bidirectional Dijikstra Implementation-**

In Bi-directional dijkstra algorithm, Dijkstra algorithm runs from both source(forward) and destination(backward) simultaneously.When a node is being processed by both sides then this algorithm is terminated.

A common point is found to link both forward and backward Dijkstra algorithm which gives the shortest path.

This signifies that if path goes through this common point,then distance is minimum.

Bi-directional algorithm runs 2 times faster than ordinary dijkstra algorithm.

Two priority queues are required in this algorithm.

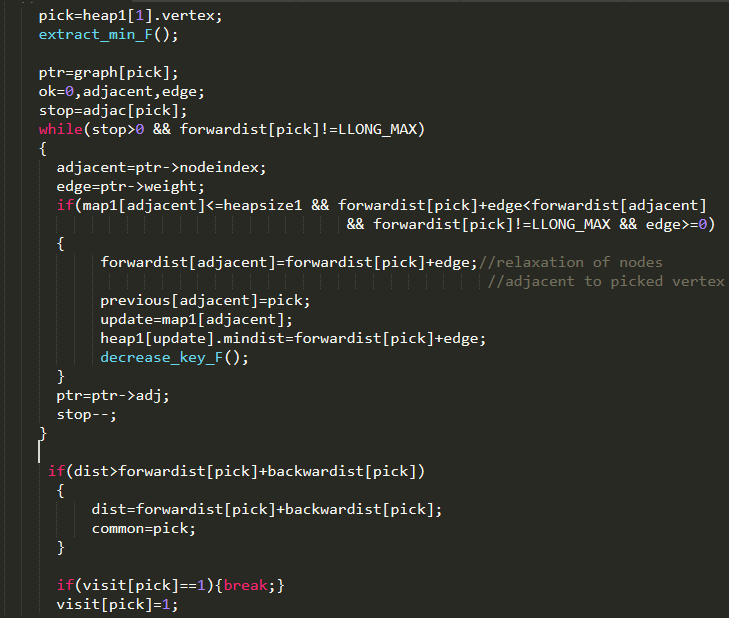
1.Forward priority queue

2.Backward priority queue

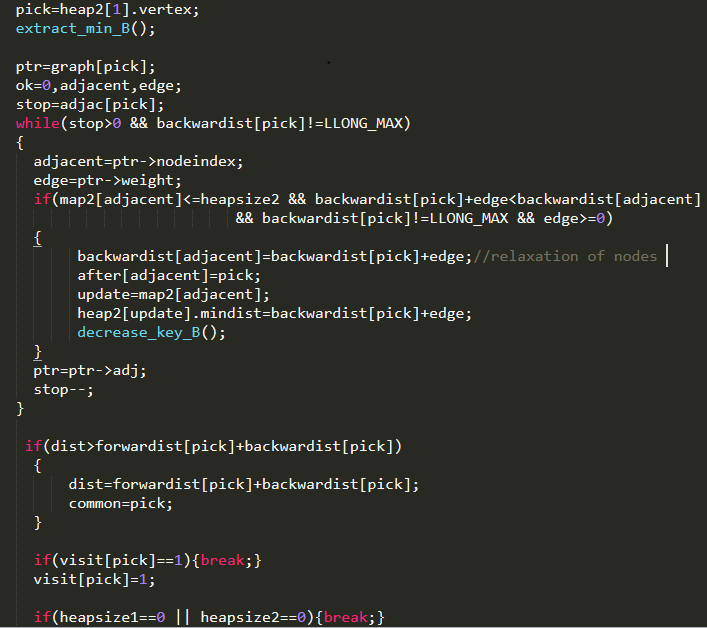
Binary heaps are used to implement priority queue in this algorithm.

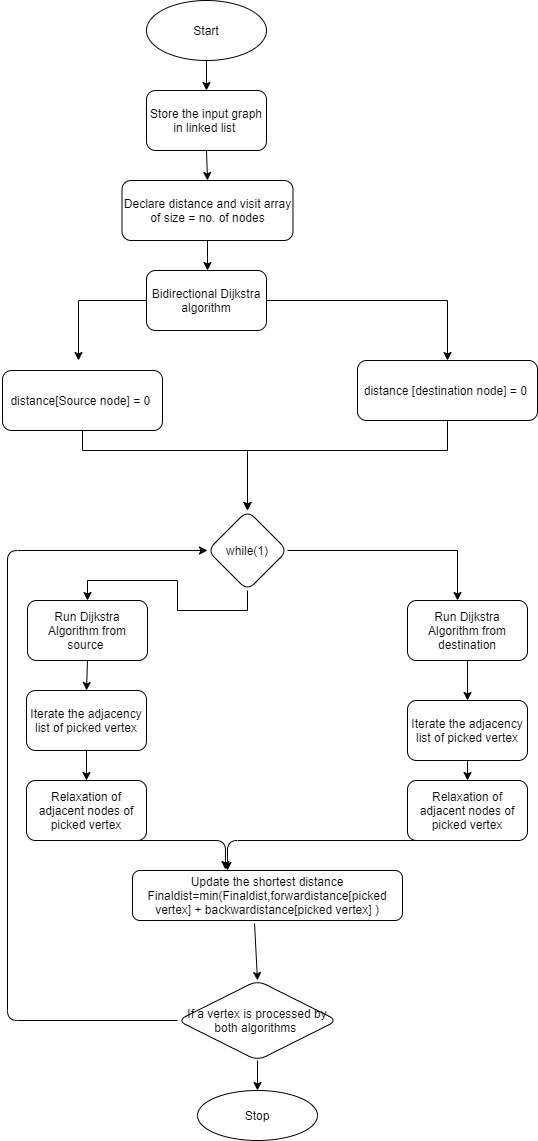
1. Build forward binary heap [source=original source of the input graph]
2. Build backward binary heap [source=destination of input graph.]
3. while(1)
4. {
5. Run Dijkstra algorithm from source**(forward dijkstra)**
6. find the linking point i=top node
7. final\_distance=min(final\_distance , forwardist[i] + backwardist[i])
8. if(visit[top node]==1)**break; //Termination Condition**
9. else visit[i]=1;
10. Run Dijkstra algorithm from destination**(backward dijkstra)**
11. find the linking point i=top node
12. final\_distance=min(final\_distance , forwardist[i] + backwardist[i])
13. if(visit[top node]==1)**break; //Termination Condition**
14. else visit[i]=1;
15. if(forward heap is empty || backward heap is empty)break;
16. }

**Forward Dijkstra**



**Backward dijkstra**

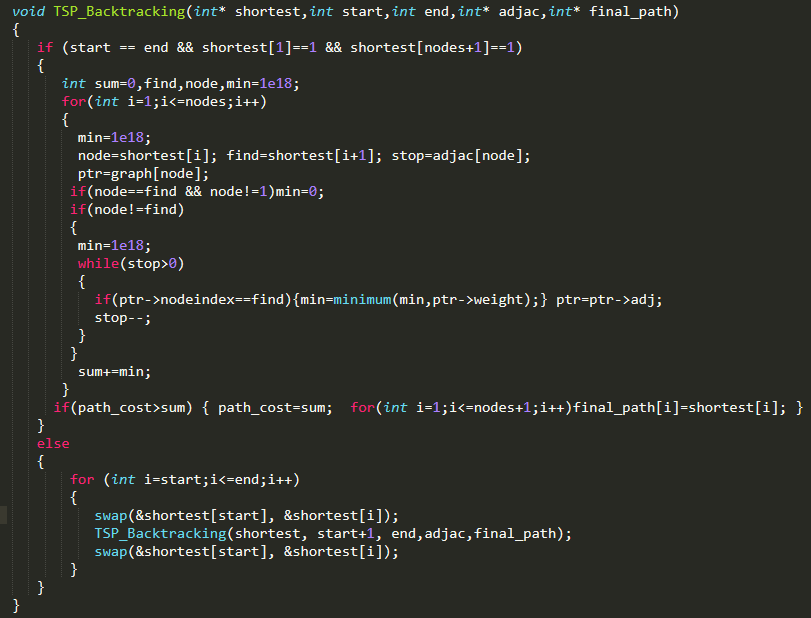
******

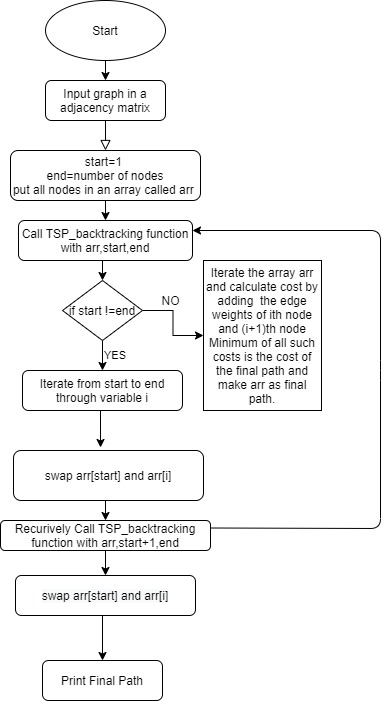
******

**Travelling Salesman Problem**

**Backtracking/Brute Force Implementation-**

1. Root node or source node(1) is the starting point and ending point of the final path.
2. Generate all (n-1)! permutations of the nodes as:
3. -Traverse the array of nodes by variable i.
4. -Swap starting node and ith node.
5. -Recursively call the function with starting node=starting node+1.
6. -Swap starting node and ith node.
7. when starting node==ending node,Calculate cost of the path.
8. Return the permutation having the minimum cost.

****

****

**Flowchart of TSP Backtracking**

**Dynamic programming method**

1.Considering Starting vertex and ending vertex as 1.

2.Find ith vertex which must be second last vertex in the path.

3.For point 2,Generate all 2^n subsets of the nodes through bit manipulation.

4.Bit manipulation works as visited arr,bits of the bitmask will represent nodes.

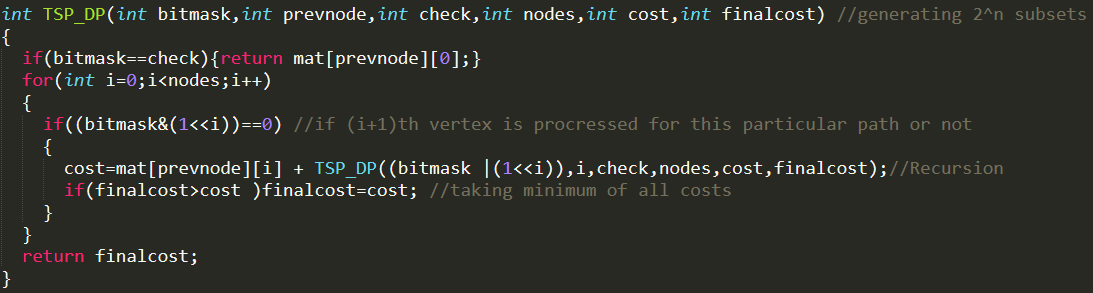
5.1 means visited and 0 means not visited.

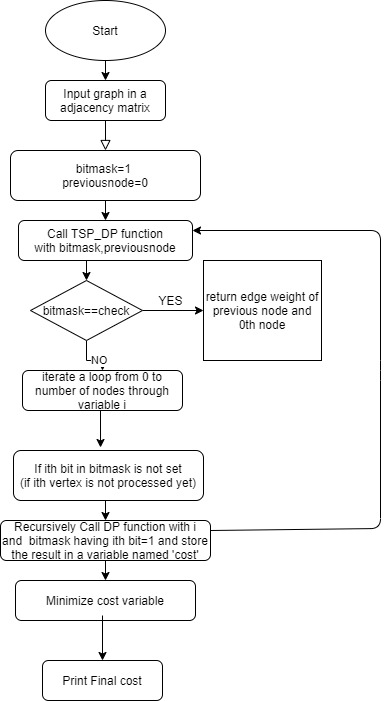
6.Recurrence relation for point 2:

cost(S,i)=dist[i][j]= + cost( {S-i-1} , j ) , where S represents set of all vertices and

j belongs to S.

Time complexity for computing minimum cost is: O(n\*(2^n)).

****

****

**Flowchart for TSP dynamic programming**