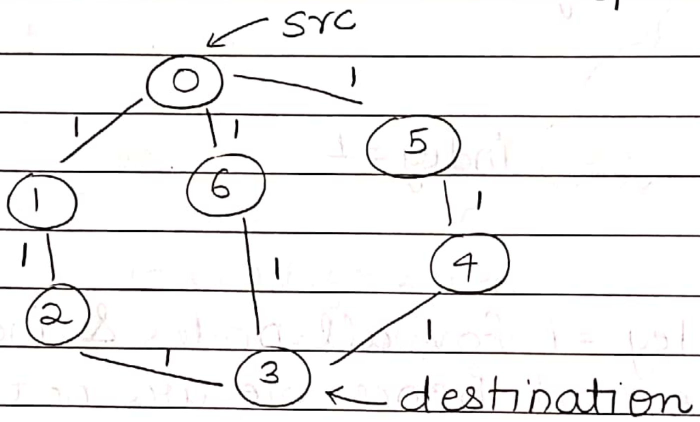
23/06/2023

Shortest distance concept



0 → 1 → 2 → 3

0 → 6 → 3 } shortest

0 → 5 → 4 → 3

(i) Using BFS

A node is having shortest path when it is visited first time.

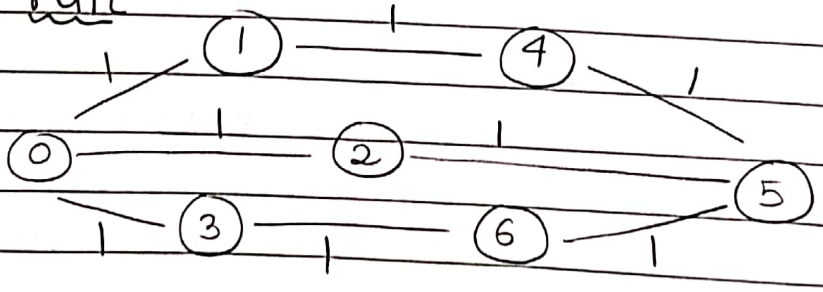
6 → 3

1 → 2 → 3

5 → 4 → 3

3 is visited first from node 6 and hence this will be the shortest path.

Dry run



Parent	Visited
0 → -1	0 → ~~F~~ T
1 → 0	1 → ~~F~~ T
2 → 0	2 → ~~F~~ T
3 → 0	3 → ~~F~~ T
4 → 1	4 → ~~F~~ T
5 → 2	5 → ~~F~~ T
6 → 3	6 → ~~F~~ T

queue → {0, 1, 2, 3, 4, 5, 6}

parent → {-1, 0, 0, 0, 1, 2, 3}
0 1 2 3 4 5 6

dest = 5
parent of 5 = 2

5 → 2
Now parent of 2 is 0
5 → 2 → 0 } simply reverse the answer

Hence shortest path ⇒ 0 → 2 → 5

Note → This concept will be valid for same weights on all edges.

Code

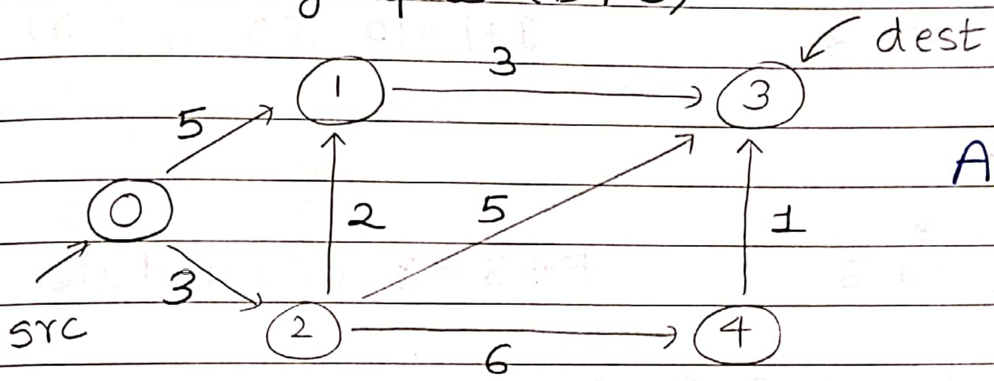
```
void shortestPath (int src, int dest){
     queue <int> q;
     unordered_map <int, bool> visited;
     unordered_map <int, int> parent;
//Initial steps for source node
q.push (src);
visited [src] = true;
parent [src] = -1 ;
while (!q.empty ()) {
     int frontNode = q.front ();
     q.pop();
     for (auto nbr : adjList [frontNode]){
         if (!visited [nbr.first]) {
               q.push (nbr.first);
               visited [nbr.first] = true;
               parent [nbr.first] = frontNode;
         }
     }
}
 // Parent array is ready
 vector <int>ans;
 int node = dest;
 while (node != -1) {
       ans.push_back (node);
       node = parent [node];
 }

 reverse (ans.begin(), ans.end ());
 for (auto i : ans)
       cout << i << "   ";
}
```

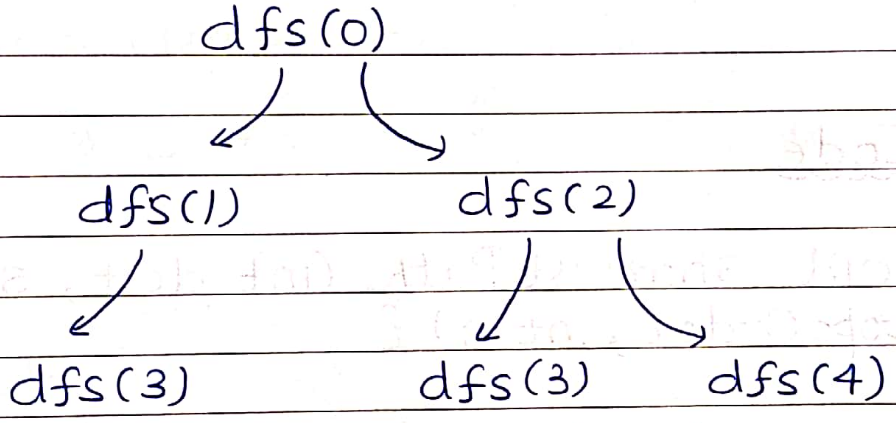
Time complexity → linear

Note→ Instead of using vector, we can use stack also.

(ii) Directed graphs (DFS)



Ans = ⑧
↓
shortest path distance



0
2
4
1
3

distance array

0	5	3	8	9
~~∞~~	~~∞~~	~~∞~~	~~∞~~	~~∞~~
0	1	2	3	4

d[src] = 0

* Pick 0.

$0 \xrightarrow{5} 1$

$0 \xrightarrow{3} 2$

* Pick 2.

③ 2 $\xrightarrow{2}$ 1   3 + 2 = 5 (no update)

2 $\xrightarrow{5}$ 3   3 + 5 = 8 (update)

2 $\xrightarrow{6}$ 4   3 + 6 = 9 (update)

* Pick 4.

⑨ 4 $\xrightarrow{1}$ 3 $9+1=10$ (no update)

* Pick 1.

⑤ 1 $\xrightarrow{3}$ 3 $5+3=8$ (no update)

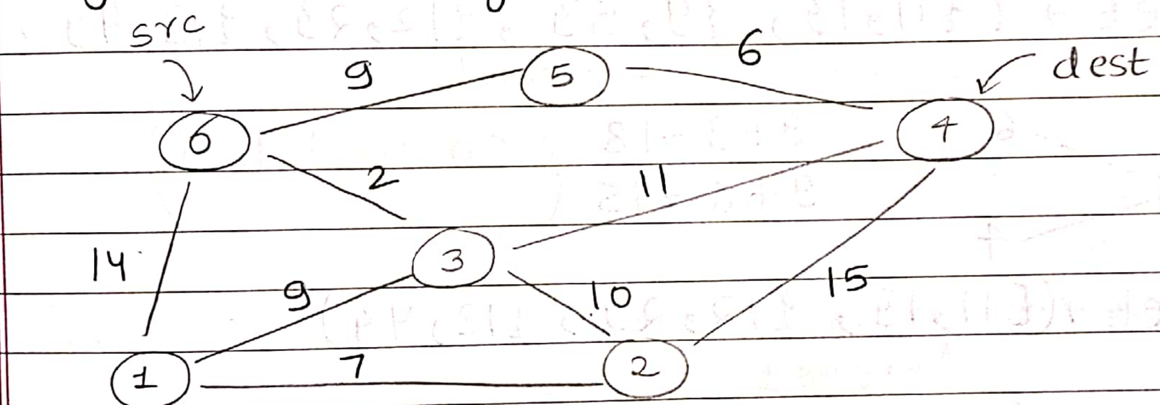
dist → {0, 5, 3, 8, 9}

Code

```
void shortestPath (int dest, stack <int>&
topoOrder, int n) {

    vector <int> dist (n, INT_MAX);
    int src = topoOrder.top();
    topoOrder.pop();
    dist [src] = 0;
    for (auto nbr : adjList [0]){
        if (dist [0] + nbr.second <dist[nbr.first])
            dist [nbr.first] = dist [0]+nbr.second;
   }
   while (! topo Order . empty ()) {
        int topElement = topo Order .top();
        topoOrder.pop();
        if (dist [topElement]! = INT_MAX){
            for (auto nbr : adjList [topElement])
            {
             if (dist [topElement] + nbr.second <
                dist[nbr.first]) {
               dist[nbr . first] = dist [topElement]
                    + nbr. second;
            }
        }
      }
    }
    // Distance array is ready
    for (auto i : dist)
          cout << i << "  ";
}
```

Dijkstra's algorithm



Think in terms of greedy. This algorithm can be implemented via min heap or set.
Here we will be implementing via set.

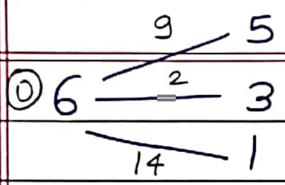
set → pair <int, int>
↓ dist ↓ node

Why dist first → set will sort according to the 1 st value.

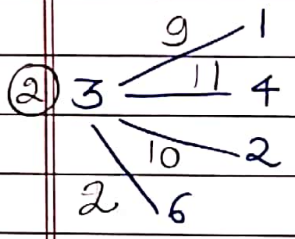
(14,1)
(9,5)
~~(2,3)~~
~~(0,6)~~

dist →

0	1	2	3	4	5	6
∞	~~∞~~	~~∞~~	~~∞~~	~~∞~~	~~∞~~	~~∞~~
	~~14~~ 11	12	2	13	9	0

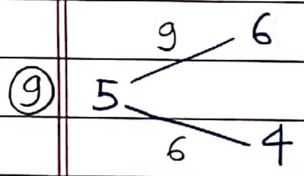


Now we will pick 2 distance, node = 3



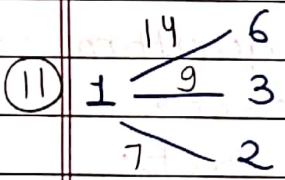
2+9=11 (update)
2+11=13 (update)
2+10=12 (update)
2+2=4 (don't update)

Set → ({11,1}, {9,5} , {12,2}, {13,4})
↳ picked



9+9=18
9+6=15
(no updation)

Set →({11,1} , {12,2} , {13,4})
↑ picked



11+14=25
11+9=20
11+7=18
(no updation)

Similarly we need to perform the same steps until set is not empty.

Code

```
void dijkstraAlgo (int src, int n){
    vector <int> dist (n, INT_MAX);
    set <pair <int, int>> st;
    dist [src] =0 ;
    st.insert ({0, src}) ;
    While (!st.empty()) {
    // Fetch smallest distance element
    auto top = *(st.begin());
    int nodeDistance = top.first;
    int node = top.second;
    //pop from set
    st.erase(st.begin());
    //Traverse neighbour
   for (auto nbr : adjList[node]){
        if (nodeDistance + nbr.second <dist[nbr.
        first]) {
         //updating distance
         // finding entry in set
         auto result = st.find({dist[nbr.first],
                                   nbr.first});
          // if found → delete
          if(result != st.end())
                  st.erase(result);
         // Update in dist array & set
         dist[nbr.first]=nodeDistance+nbr.second;
         st.insert({dist[nbr.first],nbr.first});
        }
      }
  }
  cout << "Printing array << endl;
  for (int i=0 ; i<n ; i++)
            cout << dist[i]<< " ";
  }
```