September 13, 2025

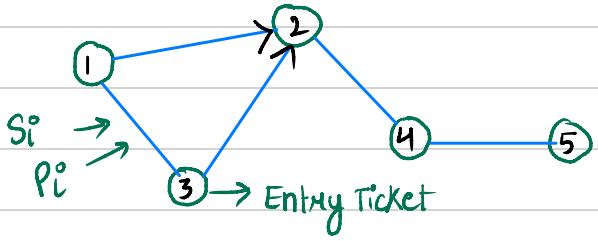
GRAPH SHORTEST PATH ALGORITHMS

Single Source Shortest Path (SSSP)	
Edge Weights	Algorithms used
1	BFS
0/1	0-1 BFS
1 ... ∞	Dijkstra
-∞ ... -∞	Bellman Ford
All Pair Shortest Path (APSP)	
-∞ ... -∞	Floyd Warshall

* STORING GRAPH

Suppose you are given a graph of cities [City 1, City 2 etc.]. Each city has Entry Ticket which you have to pay if you visit the city. Each city (nodes of the graph) is connected by roads (edges) and they have speed limits, S_i and P_i as Petrol needed to cover it. Some roads are one-directional.

HOW TO STORE THIS DATA?



Information related to edge:
- → Si
- → Pi

Info related to nodes: Entry Ticket

1.

So, Nodes information can be stored in an array like:

Entry [ ] = {-, -, -, -}
1 2 3 4

2.

Previously our graph structure was of the type: vector<vector<int>>

Now, we need to store not just the neighbours but also its edge info. So, with the neighbour, we will also store edge info. (like a pair)

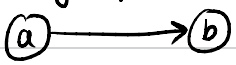
vector<vector<pair<int, pair<int,int>>>>
- ↑ city
- ↑ neighbour city
- ↑ Si
- ↑ Pi

3.

For directions, when the graph is bidirectional


```
g[a].pushback(b)
g[b].pushback(a)
```

When the graph is one directional

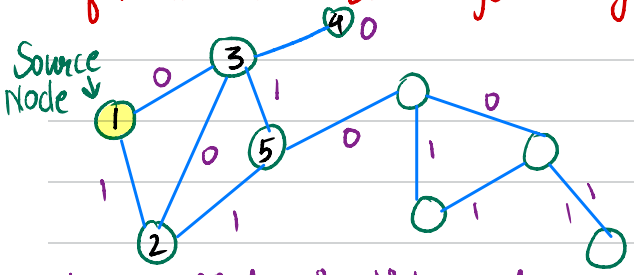



```
g[a].pushback(b)
```

So, we can use the above three approach to store this additional information.

Based on 0-1 BFS Algo

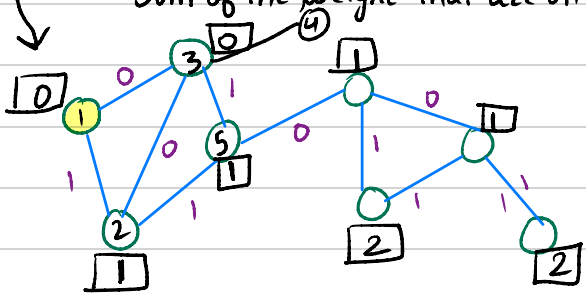
Ques 1. Run SSSP Algorithm from the given source node to every other node in the graph to find distance. Each edge is weighted.



Have to find min distance from source.

Distance = $(\Sigma W)_{path}$

Sum of the weight that are on path



Shortest Path is minimum Distance.

Main things:

1. Shortest Node not visited comes out
2. Process exactly once.

distance: 0 — 1 is taken out, Now there are 2 edges to explore. 3rd is inserted first.

Node: ~~1~~

Distance: 0 0 1

Node: ~~1~~ ~~3~~ 2

Now, 3 is taken out and we have 3 edges to explore. Edge of 2 at distance 0. Edge of 5 at distance 1. Edge of 4 at distance 0.

Node 2 5 4

Dist. 0 1 0

So, edges to explore will be in the order

Nodes: 2 4 5

Dist.: 0 0 1

But if (0,2) is inserted from the back then (1,2) which already exist will come before it (which is wrong)

0 0 1 0 0 1

~~1~~ ~~3~~ 2 2 4 5 (Wrong)

Even if the same things are coming twice. The shorter one should come first. So, instead of a queue, we will use a deque so that insertion from front is also possible which will easily solve this problem.

Iteration 1: 1 is inserted with its distance.

0

1

Iteration 2:

0

~~1~~

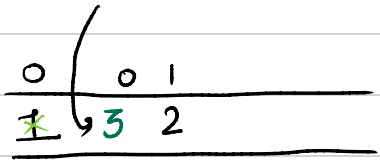
When 1 comes out, 2 edges to explore

1 → 0 → 3 → Nodes with distance=0 will be inserted from front

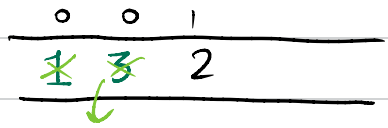
1 → 2

→ Nodes with distance = 1 will be inserted from Back

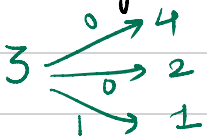
Green indicate element came from front



Iteration 3: 3 will come out

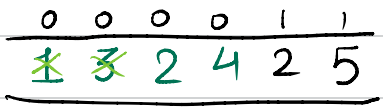


It has 3 edges to explore.



4 & 2 will be inserted from front.

1 will be inserted from the back.



We can see that the partition of 0 & 1 is well maintained here. The main idea of 0-1 BFS Algo is the distance should remain sorted and the smaller distance node should get explored first.

Note that this is the 0-1 BFS algorithm. If you have distance like 0-x, in this case 0-1 BFS can be used by converting all x to 1 and then applying algorithm. Then simply multiply the answer by x.

T.C = O(V+E)

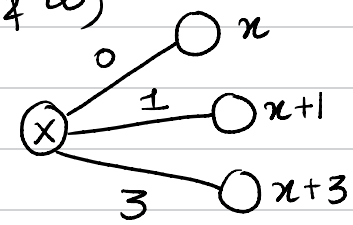
Code for the problem using 0-1 BFS.

```cpp
#include<bits/stdc++.h>
using namespace std;
#define F first
#define S second
const int INF = 1e9;

int n,m;
vector<vector<pair<int,int>>> g;

vector<int> dist,vis;
void bfs01(int sc){
    dist.assign(n+1,INF);
    vis.assign(n+1,0);

    // DS
    deque<int> dq;
    // setup sourse
    dist[sc]=0;
    dq.push_back(sc);

    // keep exploring until empty
    while(!dq.empty()){
        int cur = dq.front(); dq.pop_front();
        if(vis[cur])continue;

        // explore
        vis[cur] = 1;
        for(auto v:g[cur]){
            if(!vis[v.F] && dist[v.F]>dist[cur]+v.S){
                dist[v.F] = dirt[cur]+v.S;
                if(v.S==0){
                    dq.push_front(v.F);
                }else{
                    dq.push_back(v.F);
                }
            }
        }
    }
}
void solve(){

    cin>>n>>m;
    g.resize(n+1);
    for(int i=0;i<m;i++){
        int a,b,c;
        cin>>a>>b>>c;
        g[a].push_back({b,c});
        g[b].push_back({a,c});
    }
    bfs01(1);
    for(int i=1;i<=n;i++){
        cout<<dist[i]<<" ";
    }
}
```

* Dijkstra Algorithm

$O((V+E)\log V)$

Here, the edges can be at a distance of $0 --- \infty$ (any number b/w 0 & ∞)



If we release x, then the above 3 are our possibilities. So, in our queue, the distance will still remain sorted.

Idea is to make a pair (dist, node) and keep some sort of Priority Queue which will keep our queue sorted by default.

Another thing to note that we require minimum distance, but PQ output max on pop. Hence, we will insert with a negative distance to make our PQ reverse sorted. We will negate it back once it comes out.

```
void dijkstra(int sc){
    dist.assign(n+1,INF);
    vis.assign(n+1,0);

    // DS
    priority_queue<pair<int,int>> pq;
```

```
    // setup sourse
    dist[sc]=0;
    pq.push({-0,sc});

    // keep exploring until empty
    while(!pq.empty()){
        auto temp = pq.top(); pq.pop();
        int cur = temp.S;
        if(vis[cur])continue;

        // explore
        vis[cur] = 1;
        for(auto v:g[cur]){
            if(!vis[v.F] && dist[v.F]>dist[cur]+v.S){
                dist[v.F] = dist[cur]+v.S;
                pq.push(make_pair(-dist[v.F],v.F));
            }
        }
    }
}
```

Test Case 1

Input

```
4 3
1 2 1
2 3 2
3 4 3
```

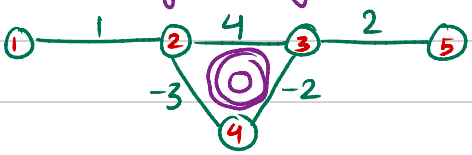
Output

```
0 1 3 6
```

* Bellman Ford $O(V \cdot E)$

helps working with negative edges, negative cycle.

What is a negative cycle?



The distance between node 1 & 5 is $-\infty$

Because you can go from N1 to N2

But traversing from N2 to N5, you will encounter a cycle

that will keep you looping in the cycle N2 → N4 → N3 → N2 with sum = -1 and if you loop ∞ times with -1 so when you reach N5, the distance will be ∞(-1) = $-\infty$ or it will give some wrong answer.

We use Bellman Ford Algorithm to address problems like these.

It says :

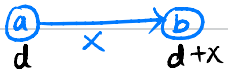
→ For (v-1) times — O(V)

for (e ∈ edges) — O(E)

O(1) → relax(e) relax all edges

⇒ O(V·E)

What does relaxing edge mean?



If you have an edge from a→b & if the distance is d. If the val at b is bigger than x then set the distance = d+x.

Why does this work?

From one node to any node, the longest shortest path will have v nodes because a shortest path cannot repeat a node unless there is a negative loop. which makes the number of edges in the longest path will always be at max (v-1)

```
void bellman(){
    cin>>n>>m;
    // g.resize(n+1);
    vector<vector<int>> edges;
    for(int i=0;i<m;i++){
        int a,b,c;
        cin>>a>>b>>c;
        // g[a].push_back({b,c});
        // g[b].push_back({a,c});
        edges.push_back({a,b,c});
    }
    // dijkstra(1);
    vector<int> dist(n+1,INF);
    dist[1]=0;
    for(int i=0;i<n-1;i++){
        for(auto edge:edges){
            if(dist[edge[1]] > dist[edge[0]]+edge[2]){
                dist[edge[1]] = dist[edge[0]]+edge[2];
            }
        }
    }
    int changed = 0;
    for(auto edge:edges){
        if(dist[edge[1]] > dist[edge[0]]+edge[2]){
            dist[edge[1]] = dist[edge[0]]+edge[2];
```

```
int changed = 0;
for(auto edge:edges){
    if(dist[edge[1]] > dist[edge[0]]+edge[2]){
        dist[edge[1]] = dist[edge[0]]+edge[2];
        changed = 1;
    }
}

if(changed){
    cout<<"Negative loop reached from 1"<<endl;
}else{
    cout<<dist[n]<<endl;
}
```

* APSP Algo $O(V^3)$

Previously what we were trying to do is find distance from a source node. In this algo, we find shortest path between any nodes. It is called Floyd Warshall.

The code is as follows:

```cpp
void floyd_warshall(){
    int n,m;
    int adj[n+1][n+1];
    // adj[i][j] -> cost to go from i to j.
    for(int i=1;i<=n;i++){
        for(int j=1;j<=n;j++){
            if(i==j)adj[i][i]=0;
            else adj[i][j]=INF;
        }
    }
    for(int i=0;i<m;i++){
        int a,b,c;
        cin>>a>>b>>c;
        adj[a][b]=min(adj[a][b],c);
    }
    // 4 line magic
    for(int k=1;k<=n;k++){
        for(int i=1;i<=n;i++){
            for(int j=1;j<=n;j++){
                adj[i][j] = min(adj[i][j],adj[i][k]+adj[k][j]);
            }
        }
    }
    // adj[i][j] -> i to j shortest path value.
}
```

Good Algorithm Summary :

https://cses.fi/book/book.pdf