

Graph shortest Path Algorithm

Shortest Path

Path

Algo] limited
# idea	

Graph Modelling / Formulation

Algo :-

single-source shortest
Path
(SSSP)

Edge Weights

Alyo

Algo
BFS — O(V+E)

0-1 BFS — $O(V+E)$

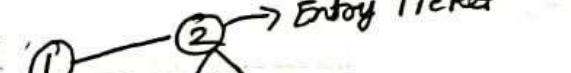
Dijkstra - $O((V+E) \log V)$

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

All Pair shortest Path (APSP) $\rightarrow -\infty \sim \infty$: Floyd Warshall Algo
 $\hookrightarrow O(V^3)$

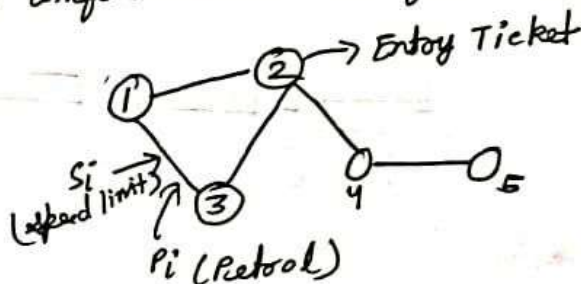
Storage Graph :-

Suppose you are given a graph of cities. In there there are city 1, city 2, ..., city 5. You are given edges. Each city has Entry Ticket which you have to pay when you visit the city. The ~~edges~~ roads (edges) have speed limit (S_i) and petrol (P_i). How will you store this information in your code? (some roads are one-directional).



```

graph LR
    1((1)) --> 2((2))
    2 --- ET[Entry Ticket]
  
```



1. For every i , we need to store some extra information,

Ex by $[] = \begin{Bmatrix} 0 & 0 & 0 & 0 \\ 1 & 2 & 3 & 4 \end{Bmatrix}$

2. Previously in edge information, we are storing its neighbour as `vector<vector<int>>g`

Now, with the neighbors will be store edge info,

1: $\left[\begin{pmatrix} 2 \\ 1 \end{pmatrix} 3, 5 \right]$
Edge Info.

⇒ vector < vector < pair < int, int >> pair < int, int >> >>

↑
neighbour
city

↑ ↑
s_i p_i

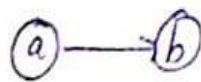
└──────────┘
Edge info.

3. For bidirectional graph,



```
g[a].push-back(b);  
g[b].push-back(a);
```

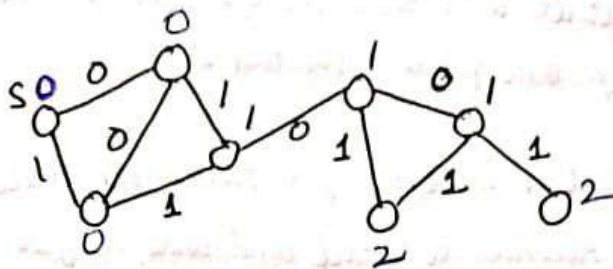
For unidirectional graph,
 $g[a].push_back(b);$



* these are the things to store information in graph.

0-1 BFS Algo :-

Q. Run SSSP algo from the given node to every other node to find the distance. Each edge is weighted

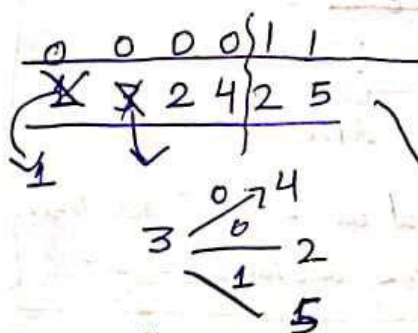


Distance = $(\sum w)$ path \downarrow
 \rightarrow sum of edge weight that are on path

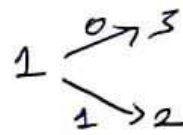
shortest path \rightarrow Min distance

- ① Shortest node not visited comes out
- ② Process exactly once.

Even if the same things are coming out twice, The shorter one should come first. \therefore instead of a queue we will use a deque and this will solve gives us 0-1 Algorithm.



When 1 comes out,



Nodes with '0' distance will come from front & node with '1' distance will come from back.

2 & 4 will be inserted from front & 5 from back

Now, we can see that partition of 0 & 1 is maintained here. The distance n is processed before distance $n+1$.

Main Trick of 0-1 BFS :-

- 1) ^{change} Queue to Deque
- 2) If 0 edge put it at front and if 1 edge put it at back ^{maintained}
- 3) As long as sorted property of distances, it will always work.

T.C : $O(V+E)$

```
#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 source
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] = dist[cur]+v.S;
            if(v.S==0){
                dq.push_front(v.F);
            }else{
                dq.push_back(v.F);
            }
        }
    }
}
```

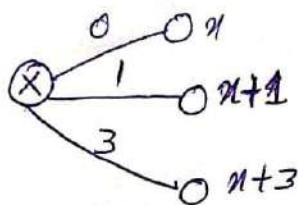
```

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 :-

If the edges can be anything from $0 \dots \infty$, then we use Dijkstra Algo.



If we release x , then any of these x , $x+1$ and $x+3$ can be possible in a queue, you are putting nodes with some distance but it will have to remain sorted. We will use priority queue which will keep our queue sorted.

* We require min distance but PQ output max on pop. Hence we will insert negative distance to make our PQ reverse sorted and 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 source
    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
    }
}

```

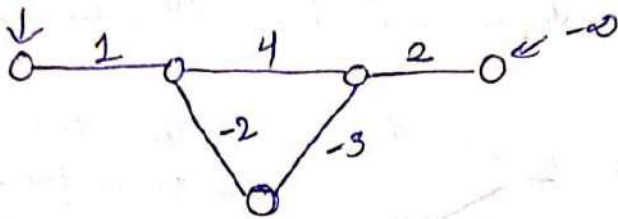
```
// keep exploring until empty
while(!dq.empty()){
    auto temp = dq.top(); dq.pop();
    int cur = temp.S;
    if(vis[cur])continue;

    // 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;
            dq.push({-dist[v.F],v.F})
        }
    }
}
```

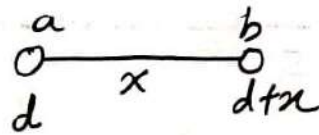
```
void solve(){  
  
    cin>>n>>m;  
    g.resize(n+1);  
    for(int i=0;i<m;i++){  
        int a,b,c;  
        cin>>a>>b>>c;  
        g[a].push_back({b,c});  
        g[b].push_back({a,c});  
    }  
    dijkstra(1);  
    for(int i=1;i<=n;i++){  
        cout<<dist[i]<<" ";  
    }  
}
```

Bellman Ford :-

↳ Used in negative cycles



Relaxing edges



If the val at b is bigger than x , set the distance $= d+x$

This algorithm says that for $(v-1)$:

[for $((v-1)$ times)
relax all Edges]

⇓

for $(v-1)$ times // $O(V)$
for $(e \in \text{Edges})$ // $O(E)$
relax (e) // $O(1)$

⇒ So, Total complexity = $O(V \cdot E)$

Why will this always 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 $-ve$ loop which makes the no. of edges in the longest path will 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];
        changed = 1;
    }
}

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

* APSP : Floyd Warshall $O(V^3)$

↳ Used to find shortest path between any two nodes.

```
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.
```