September 20, 2025

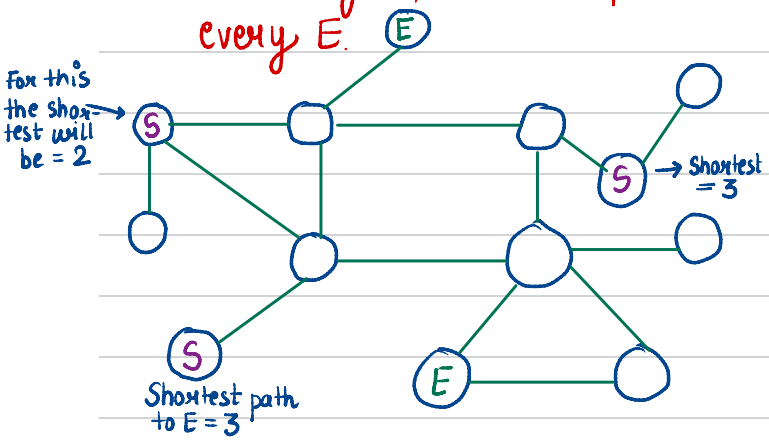
GRAPH FORMULATION IDEAS - 2

* Summary of what to keep in nodes, what to keep in edge

Whatever is asked like minimise(sum), goes to the edge.
Restriction that has to be managed goes to the nodes of the shortest path formulation.

Ques 1. Given a city plan, there are certain start cities (S) and end cities (E). Find the following:

a. Shortest path from any S to any E. Print which S→E (d)
b. For every S, shortest path to any E. Print which S→E (d)
c. For every S, shortest path to every E.

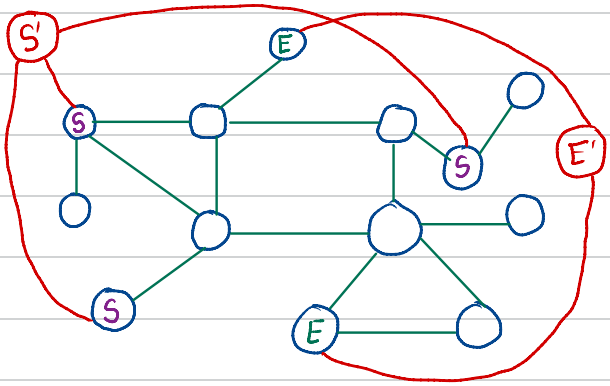


For solution b, we can just return the distance to nearest E. (done above)
For solution a, we can take the minimum of above calculated distances.

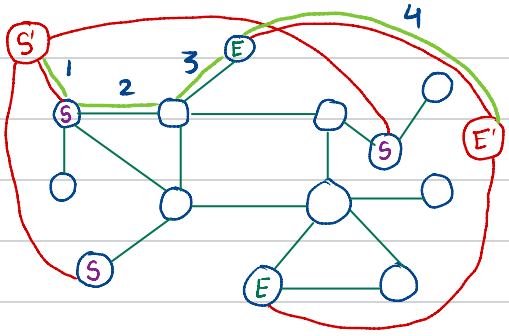
Solution a. Any S → Any E (will pick the closest E)

The concept of Mutisource, can start or end anywhere.

Example: ← supernodes
New Hypothetical node S' and E'.



Calculate the distance b/w S' and E'.
⇒ 4



Is the minimum distance between the S' and E' is also the shortest path between any S and any E?
It is since it tried all possible S's and all possible E's. The only way for the distance to be shortest if it found the shortest S→E as well.

Introduced 2 nodes, we can assign them 0 weight (like S'→S or E'→E) and perform 0-1 BFS to find the shortest path.

Time Complexity :

# of nodes = (1+1+V)

# of edges = E + #starts + #End's
= E + V + V

BFS → O(V+E)

Solution b.

From E', solve SSSP.

So, for every S, it will find out the shortest path to E'. Then for every S, we will know which E to go to because for every S, we know the shortest path to E' and it will only go through the shortest path to the closest E.

Key idea : Wherever we are asked for 'ANY', use supernodes or use MSSP(Multisource Shortest Path Problem)

So here we have ANY E, hence we will use a supernode E'.

In the earlier a part, we had ANY S→ ANY E. Hence used 2 supernodes.

Solution c.

There is no better way but to perform DFS on every S.

T.C = O(#S *(V+E))

```
int n,m;
vector<vector<int>> g
string s;


vector<int> dis;
vector<int> vis;
void bfs(vector<int> src){
    queue<int> q;
    dis.assign(n+1,INF);
    vis.assign(n+1,0);

    for(auto sc:src){
        q.push(sc);
        dis[sc]=0;
    }

    while(!q.empty()){
        int cur = q.front();q.pop();
        if(vis[cur])continue;
        vis[cur]=1;
        for(auto v:g[cur]){
            if(!vis[v] && dist[v]>dist[cur]+1){
                dist[v] = dist[cur]+1;
                q.push(v);
            }
        }
    }
}
void solve(){
    cin>>n>>m;
    cin>>s;
    g.resize(n+1);
    for(int i=0;i<m;i++){
        int a,b;cin>>a>>b;
        g[a].push_back(b);
        g[b].push_back(a);
    }
    vector<int> st,en;
    for(int i=0;i<n;i++){
        if(s[i]=='S')st.push_back(i+1);
        if(s[i]=='E')en.push_back(i+1);
    }
}
```

Addition to the BFS function above for part 1, 2 & 3 :

```
// SOLVE part 1
bfs(st);
int ans = INF;
for(auto v:en){
    ans = min(ans,dist[v]);
}
```

```
// SOLVE part 2
bfs(en);
for(auto v:st){
    cout<<v<<" "<<dist[v]<<endl;
}
```

```
// SOLVE part 3
for(auto v:st){
    bfs({v});
    for(auto x:en){
        cout<<v<<" "<<x<<" "<<dist[x]<<endl;
    }
}
```

or,
after bfs call, we can save each distance array (will help in querying later)

```
vector<int> distfrom[n+1];

for(auto v:st){
    bfs({v});
    distfrom[v] = dist;
}
for(auto v:st){
    for(auto x:en){
        cout<<v<<" "<<x<<" "<<distfrom[v][x]<<endl;
    }
}
```

* Print S to E (Part 4)

use vector <int> origin;
(original parent)

```
void bfs(vector<int> src){
    queue<int> q;
    dist.assign(n+1,INF);
    vis.assign(n+1,0);
    origin.assign(n+1,-1);

    for(auto sc:src){
        q.push(sc);
        dist[sc]=0;
        origin[sc]=sc;
    }
    while(!q.empty()){
        int cur = q.front();q.pop();
        if(vis[cur])continue;
        vis[cur]=1;
        for(auto v:g[cur]){
            if(!vis[v] && dist[v]>dist[cur]+1){
                dist[v] = dist[cur]+1;
                origin[v] = origin[cur];
                q.push(v);
            }
        }
    }
```

```
// SOLVE part 4
bfs(en);
for(auto v:st){
    cout<<v<<" "<<dist[v]<<" "<<origin[v]<<endl;
}
```

* Number of shortest path (ways)

```
vector<int> par;
vector<int> ways;

void bfs(vector<int> src){
    queue<int> q;
    dist.assign(n+1,INF);
    vis.assign(n+1,0);
    origin.assign(n+1,-1);
    par.assign(n+1,-1);
    ways.assign(n+1,0);

    for(auto sc:src){
        q.push(sc);
        dist[sc]=0;
        origin[sc]=sc;
        par[sc]=-1;
        ways[sc]=1;
    }
```

```
while(!q.empty()){
    int cur = q.front();q.pop();
    if(vis[cur])continue;
    vis[cur]=1;
    for(auto v:g[cur]){
        if(!vis[v] && dist[v]>dist[cur]+1){
            dist[v] = dist[cur]+1;
            origin[v] = origin[cur];
            par[v] = cur;
            ways[v] = ways[cur];
            q.push(v);
        }
        else if(dist[v]==dist[cur]+1){
            ways[v] += ways[cur];
        }
    }
}
```

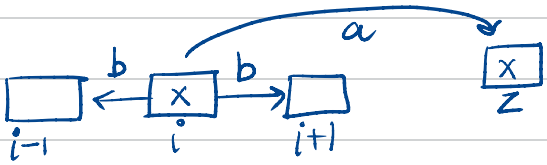
AlgoZenith

* Four Key concept of formulation

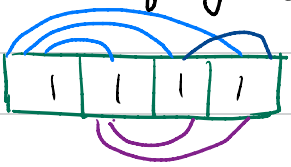
1. Edge
2. Node
3. Any → Supernode
4. Path Info
   - → Immediate parent, par
   - → Origin
   - → # of paths

Ques 2. Jump Game

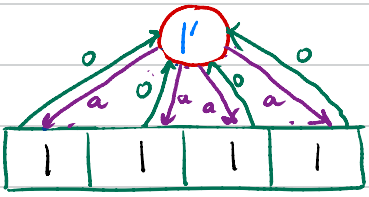
Given an array having n elements, the cost to move from i^{th} element to its adjacent element (*if exist*) at $i+1$ and $i-1$ is b, and the cost to move to other same valued index is a. Find min cost to reach every index from a given source index of the array.



could have easily done it using graph but thinking of worst case, we can have a situation where an array of all 1's could come which will lead to a very high number of edges ($^nC_2$ edges).



Use of supernode concept can be done in such a case.



We made the supernode 1' with 0 weights To go from 1' to 1, it costs a but to come back, there is cost of 0.

Now, instead of $^nC_2$ edges, we have 2N edges and 1 extra node.

Now, we can go to any 1 at the cost of a.

(Add supernode for every color)

```
void solve() {
    cin >> n;
    cin >> a >> b;
    adj.assign(2 * n + 10, vector<array<int, 2>>());
    for (int i = 1; i <= n; i++) {
        cin >> arr[i];
        sadj[arr[i]].push_back(i);
        if (i != n)adj[i].push_back({i + 1, b});
        if (i != 1)adj[i].push_back({i - 1, b});
    }
    int super_node = n + 1;
    for (auto x : sadj) {
        for (auto i : x.second) {
            adj[super_node].push_back({i, a});
            adj[i].push_back({super_node, 0});
        }
        super_node++;
}// now we have at max 2*n nodes in the graph and 4*n-2 edges.
int src;
cin >> src;
dijktra(src);
for (int i = 1; i <= n; i++)cout << dist[i] << " ";
```