Graph Formulation Ideas - 2

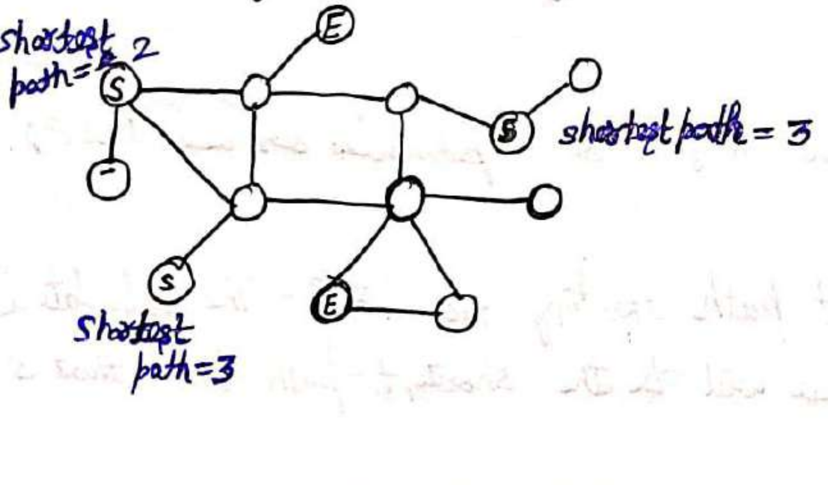
* Whatever is asked like minimise (sum), goes to the edge.
* Restriction that has to be managed goes to the nodes of the shortest path formulation.

Q1. 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

(b) For every S, shortest path to any E.

(c) For every S, shortest path to every E.



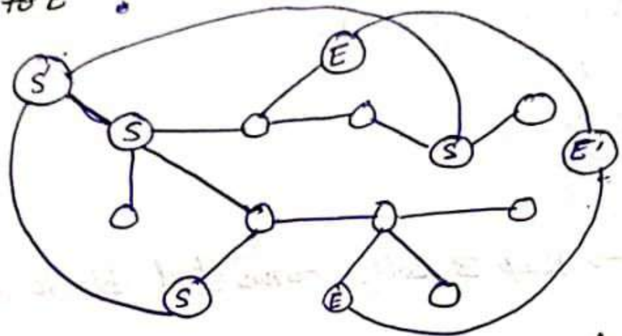
(a)

Any S → Any E

Whenever 'any S' we hear smthng like this, the concept of multisource comes into role.

Multisource → u can start anywhere or end anywhere

Ex :
suppose there is a hypothetical nodes S' and E'. Calculate shortest path from S' to E' ? 2 4



Is the min distance b/w S' and E' is also the shortest paths b/w any S and any E ?
It since tried all pbl S's and all psbl E's. The only way for the distance to be shortest if it found shortest S→E.

T.C :-
\# of nodes = (1+1+V)
\# of edges = E + # starts + # Ends
= E + V + V

BFS → O(V+E)

(b) From E', solve SSSP
Create only the destination supernode E', connecting all E nodes to it with 0-weight edges.

Key Idea : Whenever we are asked for 'Any', use supernodes or use MSPP (Multisource shortest Path Problem)

solution : Run a single-source shortest path starting from E'. The calculated distance from E' to any given S node will be the shortest path from that S to its closest E.

Perform BFS on every S. $\boxed{T.C : O(\# S * (V+E))}$

(c) Perform BFS on every S.

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

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

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

```
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);
            }
```

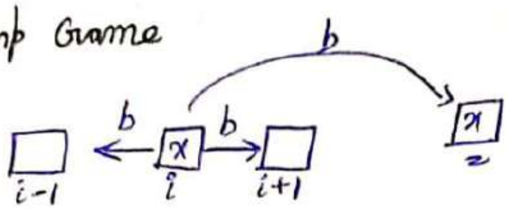
```
// SOLVE part 4
bfs(en);
for(auto v:st){
    cout<<v<<" "<<dist[v]<<" "<<origin[v]<<endl;
}
```

Four key concept of formulation :-

1. Edge
2. Node
3. Any → Supernode

~~3. Any →~~ 4. Path Info
→ Immediate parent,
→ origin
→ # of paths

Q2. Jump Game



A naive approach would create a complete graph b/w all indices with the same value. For an array with many identical elements, this would lead to a very large no. of edges ~~(O(N²)) ana~~ ($^nC_2$ edges), making the solution very slow.

Super Node solution :-
For each unique value in the ~~string~~ array, create a dedicated set supernode.
1. Connect each index i to its value's supernode. The cost to move from an index to the supernode is 0.
2. Connect the supernode back to each of its ~~corresp~~ corresponding indices.

We made supernode 1' with 0 ~~indices~~ weights to go from 1' to 1, it cost a but to come back, cost = 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.

```cpp
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++;
```