Q

M

N

S				
	#		#	#
	#			E
		#		
	S	#		E

start at any S

end at any E

① Find shortest path from any S to any E.

② Find SP from every S to any E

③ Find SP from every S to every E

1 S — 1 E

direct BFS

1 S → multiple E

BFS → SSSP SP to all nodes

if we have multiple sources

for each source S

BFS(S) → O(N·M)

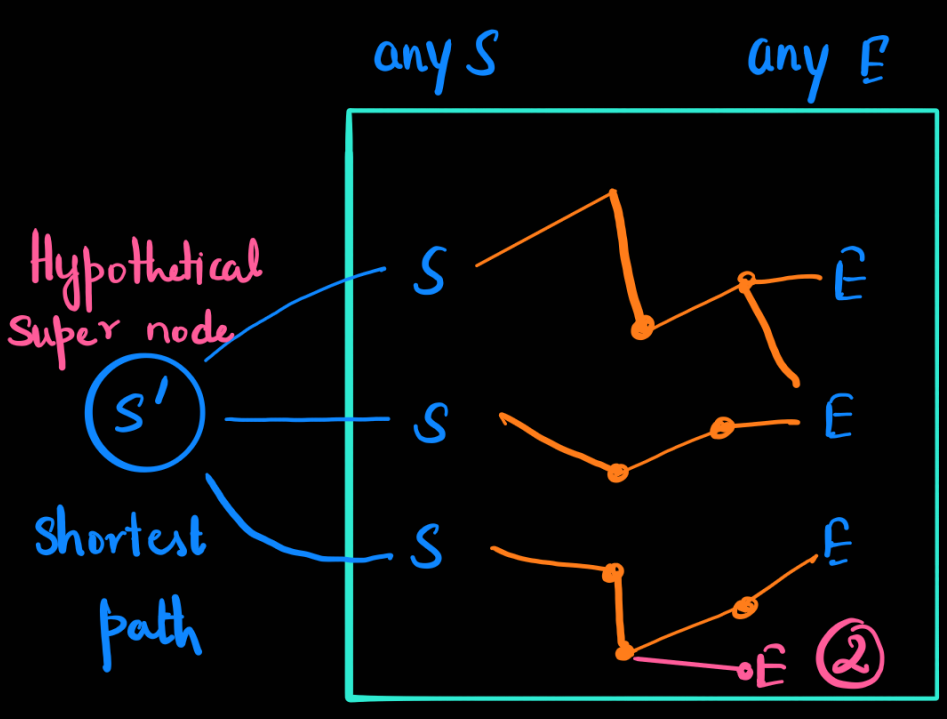
Find best E → O(N·M)

$O(S \cdot N \cdot M)$

Solves ①, ②, ③ since we have perfect data

$O(N^2 \cdot M^2)$ worst case

Multisource BFS



S' to all other nodes

S' - single source

So SSSP by BFS

$S' \rightsquigarrow E = x$ then ans $= x - 1$

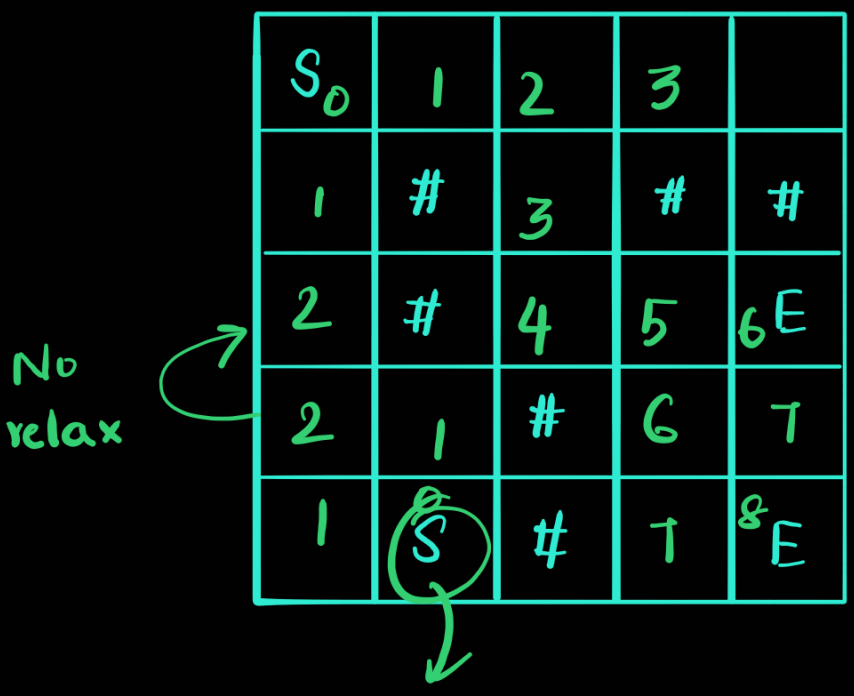
0	1	1	1		x
~~S'~~	S_1	S_2	S_3	....	E

instead of creating S', put S into queue

0	0	0		x-1
S_1	S_2	S_3	....	E

It will solve (1)

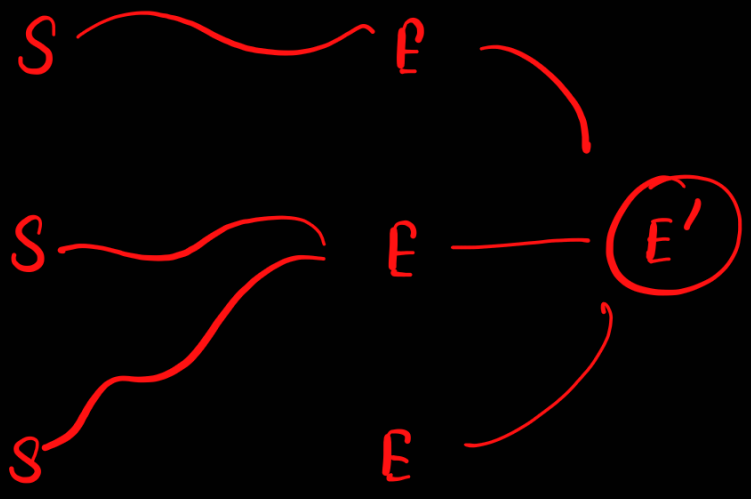
For every S we might not get an E



Can use MBFS to solve ②

How ?

From endpoints ?



if we do MBFS from end points

AlgoZenith

We can get SP from any E to any S
All S get matched.

$$S \xrightarrow{SP=x} E$$

then

$$E \xrightarrow[SP=x]{} S$$

TRICK :- (ANY) → Do MSSP on that

(2) any E, do multisrc on E.

(3) Have to do brute force

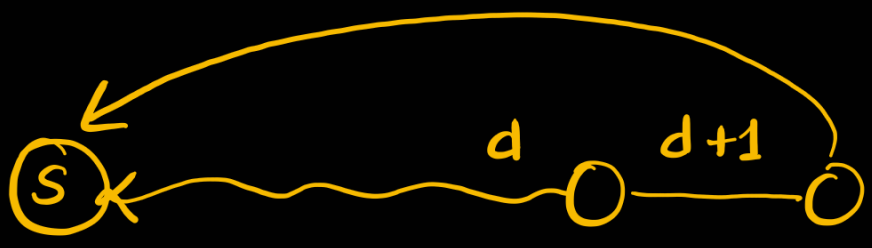
```cpp
vector<state> st,en;
for(int i=0;i<n;i++){
    cin>>arr[i];
    for(int j=0;j<m;j++){
        if(arr[i][j]=='S'){
            st.push_back({i,j});
        }else if(arr[i][j]=='E'){
            en.push_back({i,j});
        }
    }
}
```

```cpp
void bfs(vector<state> allst){
    vis = vector<vector<int>>(n,vector<int>(m,0));
    dist = vector<vector<int>>(n,vector<int>(m,
    INF));

    queue<state> q;

    for(auto st:allst){
        dist[st.F][st.S] = 0;
        q.push(st);
    }
```

from every S to any E

```
bfs(en);
for(int i=0;i<n;i++){
    for(int j=0;j<m;j++){
        cout<<dist[i][j]<<"\t";
    }
    cout<<endl;
}
```

① Minimize → Edge
② Restrictions → Node
③ Any → MSSP
④ property of path → parents, dist, no. of paths, first parent
↓
flipping switch some → auxiliary table like dist



AlgoZenith

origin auxiliary table

```
vector<vector<int>> vis, dist;
// auxialiary tables.
vector<vector<state>> par,origin;
```

```
// process the node
for(auto v:neighbour(cur)){
    // relaxing edge.
    if(dist[v.F][v.S] > dist[cur.F][cur.S]+1){
        dist[v.F][v.S] = dist[cur.F][cur.S]+1;
        par[v.F][v.S] = cur;
        origin[v.F][v.S] = origin[cur.F][cur.S];
        q.push(v);
    }
}
```

passing aux data

```cpp
void bfs(vector<state> allst){
    vis = vector<vector<int>>(n,vector<int>(m,0));
    dist = vector<vector<int>>(n,vector<int>(m,INF));

    par = origin = vector<vector<state>>(n,vector<int>(m,INF));

    queue<state> q;

    for(auto st:allst){
        dist[st.F][st.S] = 0;
        par[st.F][st.S] = {-1,-1};
        origin[st.F][st.S] = st;
        q.push(st);
    }
```

{-1,-1}

} Aux data

```cpp
for(auto [x,y]:st){
    cout<<x<<","<<y<<":"<<dist[x][y]<<" "<<origin[x][y].F<<",
    "<<origin[x][y].S<<endl;
}
```

printing

Number of SP other node

```
vector<vector<int>> num_path;
```

```
num_path = vector<vector<int>>(n,vector<int>
(m,0));
```

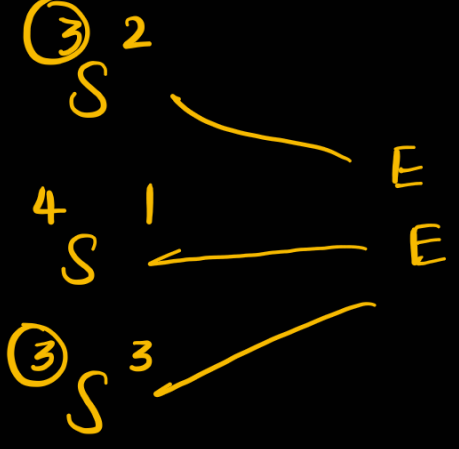
```
for(auto st:allst){
    dist[st.F][st.S] = 0;
    q.push(st);
    // auxiliary for the origins.
    par[st.F][st.S] = {-1,-1};
    origin[st.F][st.S] = st;
    num_path[st.F][st.S] = 1;
}
```

```
for(auto v:neighbour(cur)){
    // relaxing edge.
    if(dist[v.F][v.S] > dist[cur.F][cur.S]+1){
        dist[v.F][v.S] = dist[cur.F][cur.S]+1;
        q.push(v);
        // propagate auxiliary data.
        num_path[v.F][v.S] += num_path[cur.F][cur.S];
        par[v.F][v.S] = cur;
        origin[v.F][v.S] = origin[cur.F][cur.S];
    }else if(dist[v.F][v.S] == dist[cur.F][cur.S]+1){
        num_path[v.F][v.S] += num_path[cur.F][cur.S];
    }
}
```

6 4 S	5 4 .	#	1 1 .	0 1 E
#	4 4 .	3 4 .	2 4 .	1 2 .
6 4 S	5 4 .	#	1 1 .	0 1 E

any S to E → Add S 4+4=8

every S to E → 4



any S any E

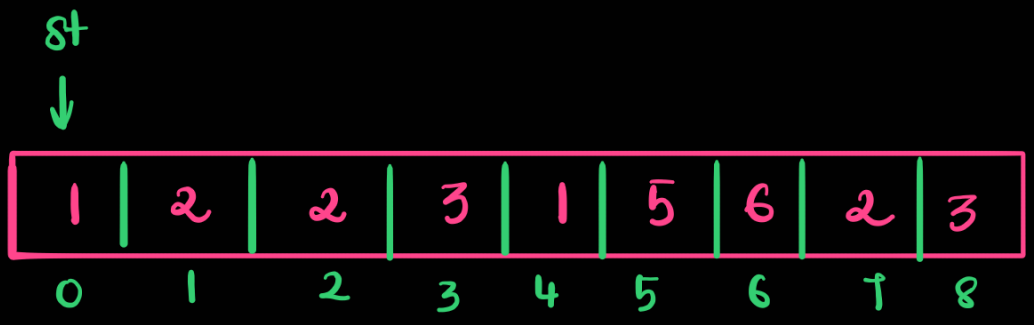
2+3=5

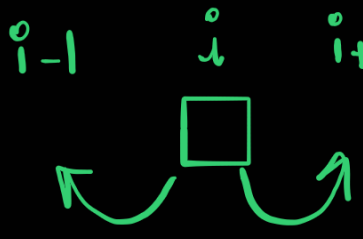
AlgoZenith

```
int min_dist = INF;
for(auto [x,y]:st){
    cout<<x<<","<<y<<":"<<dist[x][y]<<" "<<origin[x][y].
    F<<","<<origin[x][y].S<<endl;

    min_dist = min(dist[x][y],min_dist);

}
```

```
int tot_paths = 0;
// find the total closest S , E pair paths.
for(auto [x,y]:st){
    if(dist[x][y]==min_dist){
        tot_paths+= num_path[x][y];
    }
}
cout<<tot_paths<<endl;
```

Q





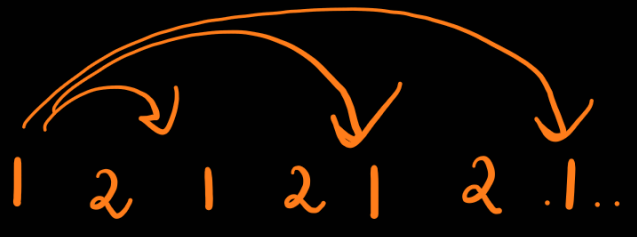
a units cost

[OR]

b units cost

teleport to same value cell

SSSP find



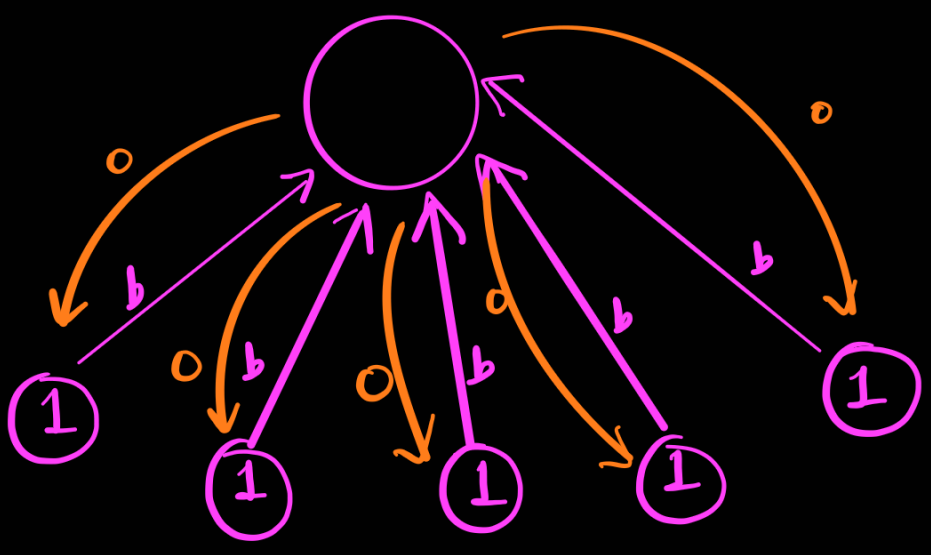
No. of edges is very high

O(N) per 1

$E \rightarrow O(N^2)$

$V \rightarrow O(N)$

$O((V+E)\log N) \quad - \quad O(N^2 \log N)$



$O(N^2) \rightarrow O(2 \cdot N)$

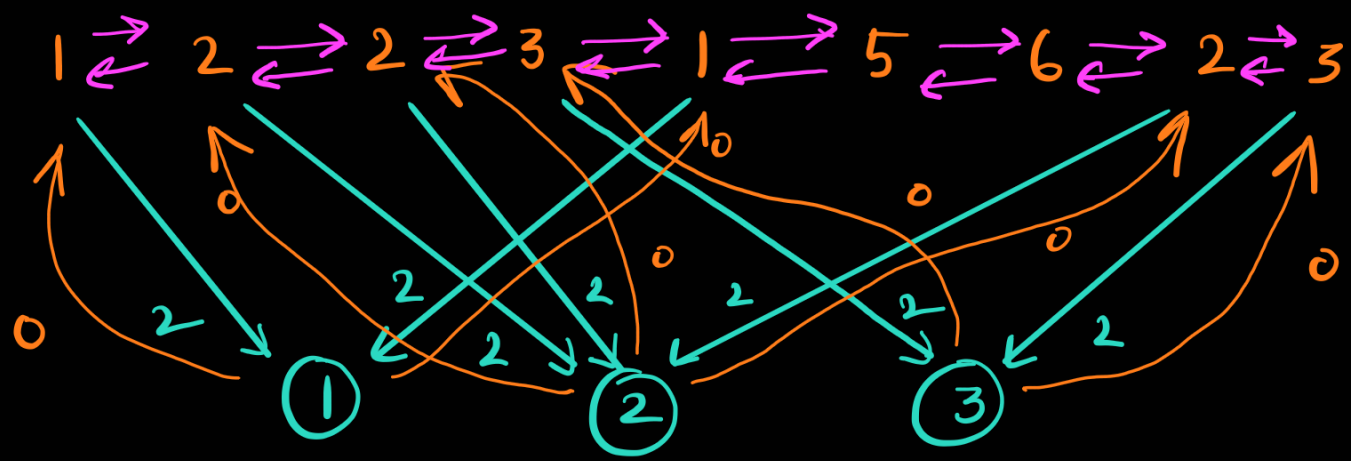
for every distinct color maintain supernode

$E \rightarrow 4N$

$V \rightarrow 2N$

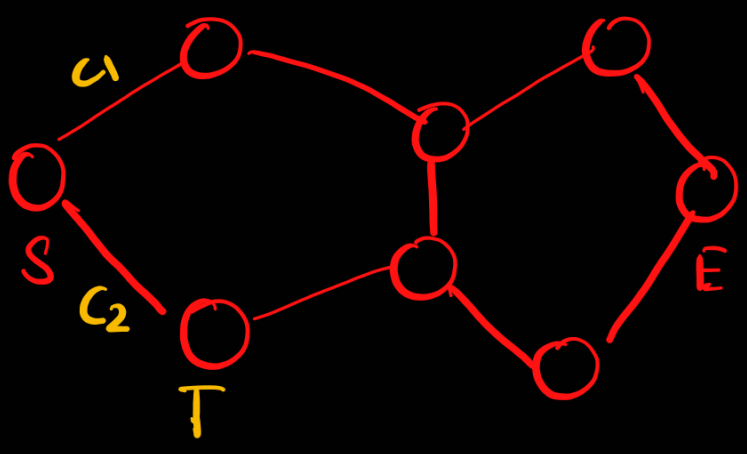
AlgoZenith

a = 1 b = 2



4 edges per node maximum

Q

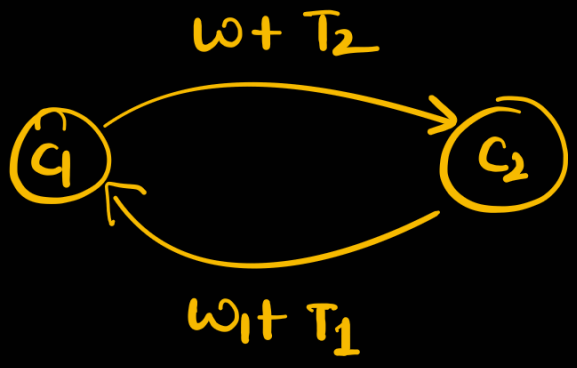


Toll tax

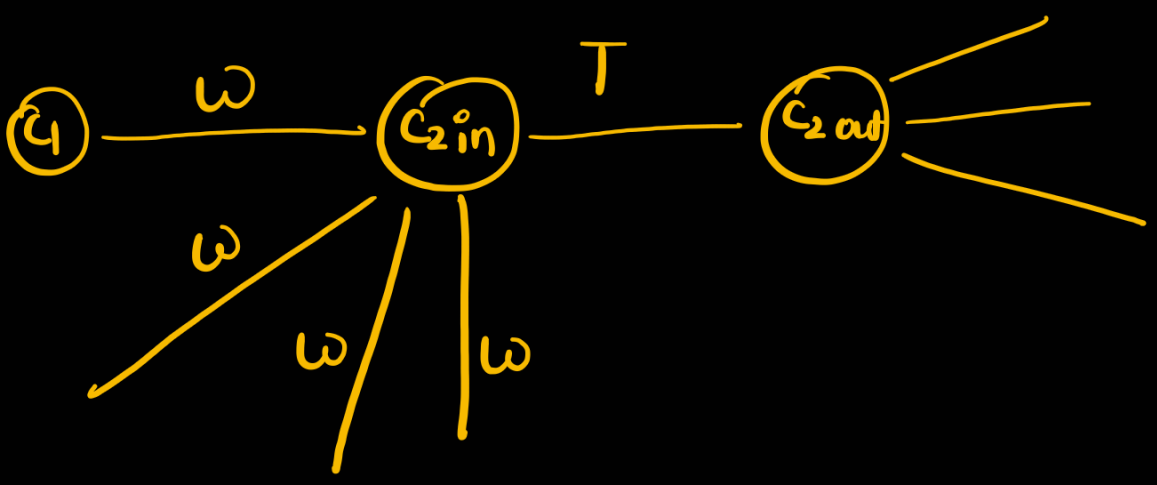
2 ideas

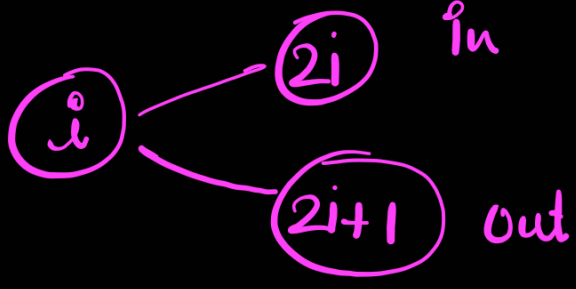
AlgoZenith

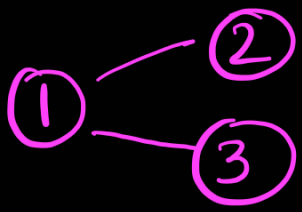
1

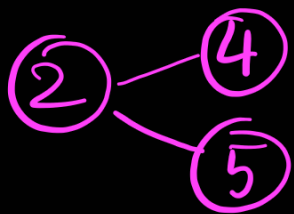

w + T2
c1
c2
w1 + T1

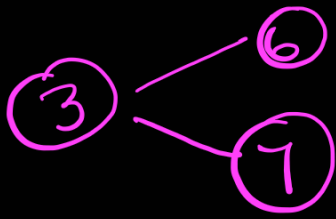
2


c1
w
c2 in
T
c2 out
w
w
w


i
2i
in
2i+1
out


1
2
3


2
4
5


3
6
7


AlgoZenith

```
int n,a,b;
vector<int> arr;
```

```
cin>>n>>a>>b;
arr.resize(n);
for(int i=0;i<m;i++){
    cin>>arr[i];
}
```

```cpp
map<int,int> mp;
for(int i=0;i<n;i++){
    cin>>arr[i];
    mp[arr[i]];
}
int node_num = n;
for(auto v:mp){
    mp[v.first]=node_num++;
}
```

```cpp
vector<vector<pair<int,int>>> g;
```

```cpp
for(int i=0;i<n;i++){
    if(i-1>=0)g[i].push_back({i-1,a});
    if(i+1<n)g[i].push_back({i+1,a});
    g[i].push_back({mp[arr[i]],b});
    g[mp[arr[i]]].push_back({i,0});
}
```

```
dijkstra(0);
```

```
for(int i=0;i<n;i++){
    cout<<dist[i]<<" ";
}
```

```
vector<vector<pair<int,int>>> g;

vector<int> vis, dist;

void dijkstra(state st){
    vis = vector<int>(g.size());
    dist = vector<int>(g.size(),INF);
```

```
priority_queue<pair<int,state>> q;
```

```
dist[st]= 0;
q.push({-0,st});
```

```
while(!q.empty()){
    auto [dis,cur] = q.top();
    dis = -dis;
    q.pop();

    if(vis[cur])continue;
    vis[cur]=1;
    // process the node
    for(auto v:g[cur]){
        // relaxing edge.
        if(dist[v.F] > dis+v.S){
            dist[v.F] = dis+v.S;
            q.push({-dist[v.F],v.F});
        }
    }
}
```

AlgoZenith