Shortest Path

Nafis Ul Haque Shifat and Jarif Rahman

11th February 2022

সূচীপত্র

1 Shortest Path Theory

Jarif Rahman

Problem

তোমাকে একটা গ্রাফ দেওয়া হয়েছে। তোমাকে কোনো এক node a থেকে অন্য কোনো node b এর সবচেয়ে ছোট path বের করা লাগবে।

Competitive Programming এ এই ধরনের প্রব্লেম অহরহ আসে। আমরা দুইটা কেইস এর কথা চিন্তা করব। Unweighted গ্রাফ আর Weighted graph। Unweighted এর ক্ষেত্রে আমরা BFS দিয়ে কাজে চালিয়ে নিতে পারব। Weighted এর ক্ষেত্রে Dijkstra বা Bellman-Ford বা Floyd-Warshal এর মত অ্যালগরিদম লাগবে। Floyd-Warshal বাদে প্রায় সব shortest path অ্যালগরিদমে একটা source node থাকে আর আমরা ওই node থেকে বাকি সব node এর দূরত্ব একবারে বের করি।

1.1 Unweighted Graph

Unweighted গ্রাফে shortest path বের করার সময় আমরা সব edge এর weight 1 ধরে নিতে পারি। অর্থাৎ আমাদের এমন এক path বের করা লাগবে যেখানে edge এর সংখ্যা সবচেয়ে কম। Unweighted গ্রাফে আমরা BFS ব্যবহার করে shortest path ব্যবহার করতে পারি।

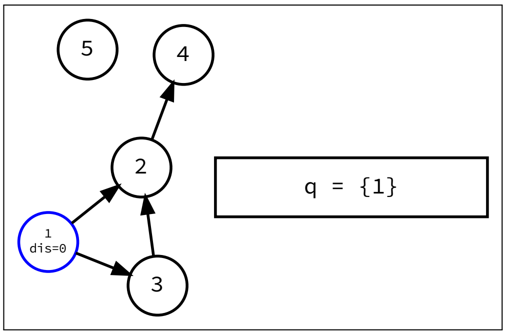
1.1.1 Breadth First Search (BFS)

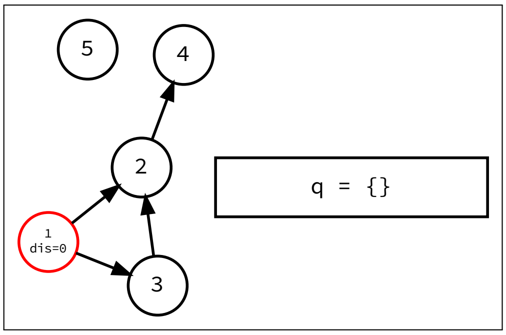
ধর আমাদের source node nd। অর্থাৎ আমরা nd থেকে বাকি সব node এর দূরত্ব (দূরত্ব বলতে shortest path এর দূরত্ব) বের করতে চাই।

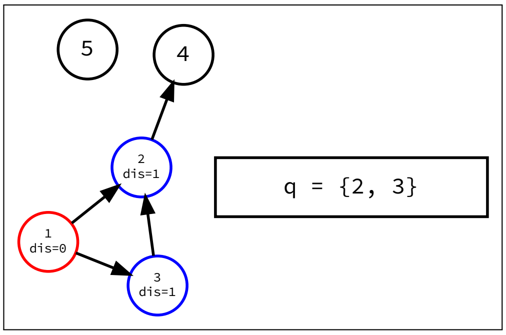
আমরা DFS এর মতো nd থেকে শুরু করে অন্য সব node গুলাকে visit করব। কিন্তু এ ক্ষেত্রে যাদের দূরত্ব কম তাদেরকে আগে visit করব। অর্থাৎ যাদের দূরত্ব 1 তাকে আগে visit করব, তারপর যাদের দূরত্ব 2, তারপর যাদের দূরত্ব 3, $\cdots$ এভাবে চলতে থাকবে যতক্ষণ না সব node visit করা শেষ হয়। এভাবে visit করতে পারলে আমরা খুব সহজেই সবার দূরত্ব বের করে ফেলতে পারব। কিন্তু আমরা এভাবে visit করব কিভাবে?

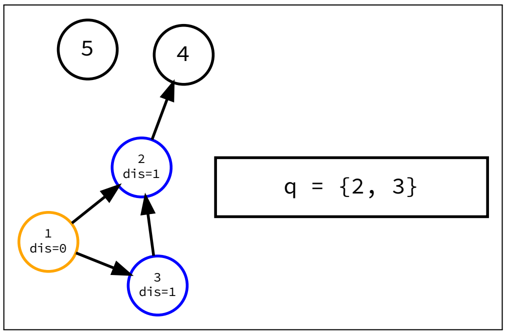
Observation: nd থেকে যদি কোনো node এর দূরত্ব $x(x > 0)$ হয় তাহলে এটা এমন একটা node এর সাথে যুক্ত আছে (edge দ্বারা) যার দূরত্ব $x - 1$।

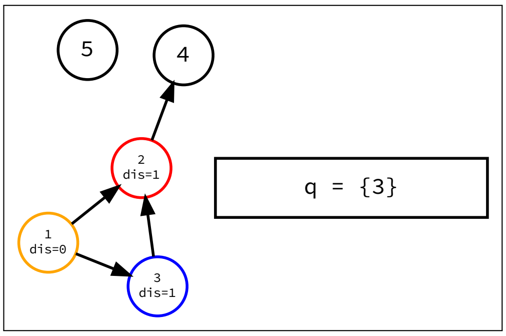
এইবার আমরা খুব সহজেই BFS করতে পারব। আমরা একটা অ্যারে dis এ দূরত্বগুলা রাখব। dis_x হবে nd থেকে x এর দূরত্ব। আমরা nd থেকে BFS শুরু করব। আর একটা queue q রাখব যেখানে আমরা কোন ক্রমে node গুলাকে visit করব তা স্টোর করে রাখব। শুরুতে q তে শুধু nd থাকবে। আমরা যখন কোনো node কে visit করব, তার দূরত্ব যদি x হয় তাহলে এর সাথে যতগুলা node যুক্ত আছে যাদেরকে visit করা হয় নি তাদের দূরত্ব $x + 1$ হবে। আমরা এই ধরনের সব node এর দূরত্ব $x + 1$ করে দিব আর এদেরকে q এর শেষে ঢুকায়ে দিব। আমরা এটা করতে থাকব যতক্ষণ না q এর সাইজ 0 হয়।

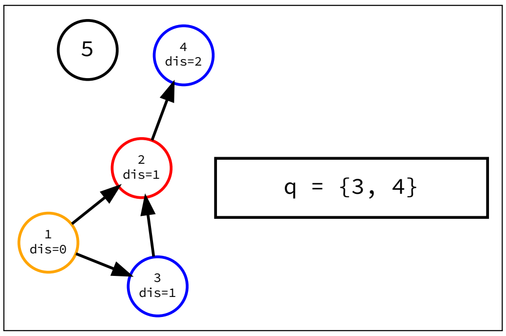

5
4
2
1
dis=0
3
q = {1}

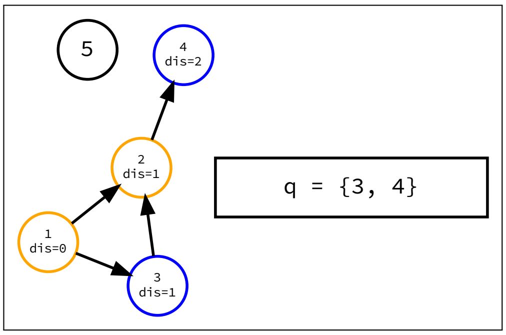

5
4
2
1
dis=0
3
q = {}

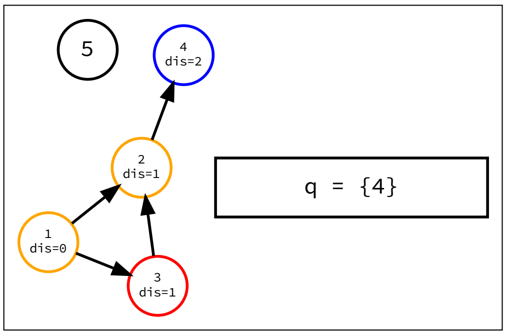

5
4
2
dis=1
1
dis=0
3
dis=1
q = {2, 3}

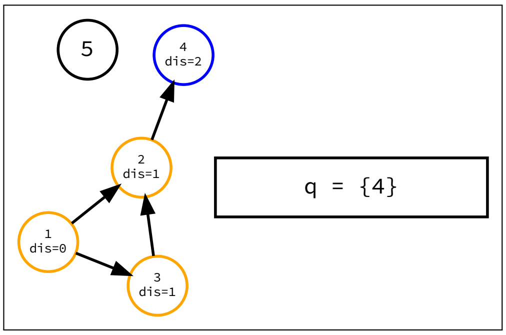

5
4
2
dis=1
1
dis=0
3
dis=1
q = {2, 3}

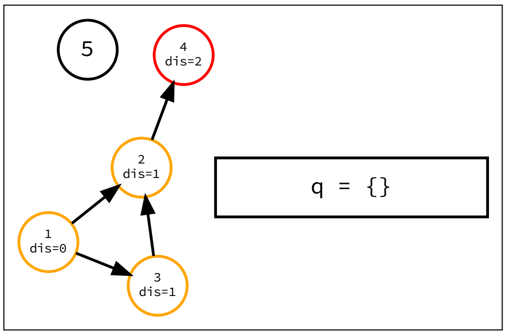

5
4
2
dis=1
1
dis=0
3
dis=1
q = {3}

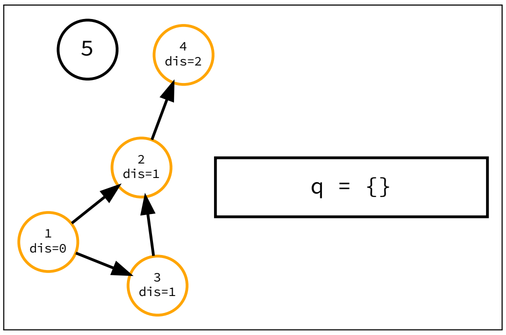

5
4
dis=2
2
dis=1
1
dis=0
3
dis=1
q = {3, 4}

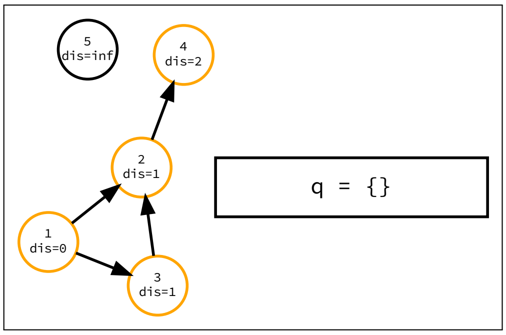

5
4
dis=2
2
dis=1
1
dis=0
3
dis=1
q = {3, 4}


5
4
dis=2
2
dis=1
1
dis=0
3
dis=1
q = {4}


5
4
dis=2
2
dis=1
1
dis=0
3
dis=1
q = {4}


5
4
dis=2
2
dis=1
1
dis=0
3
dis=1
q = {}





Implementation

```
// n is the number of nodes
// v is the adjacency list
// st is the starting node

queue<int> q;
vector<int> dis(n);
vector<bool> bl(n, false);

q.push(0);
dis[st] = 0;
bl[st] = true;

while(!q.empty()){
    int nd = q.front();
    q.pop();
    for(int x: v[nd]){
        if(bl[x]) continue;
        bl[x] = 1;
        dis[x] = dis[nd]+1;
        q.push(x);
    }
}
```

BFS এর time complexity $O(n+m)$ যেখানে n ও m যথাক্রমে node ও edge এর সংখ্যা।

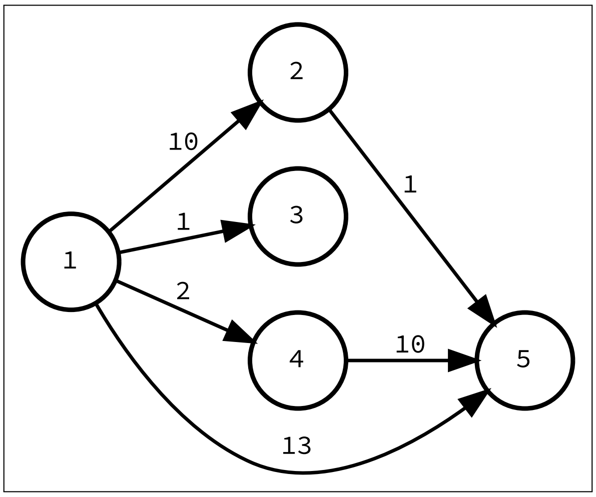
1.1.2 Multi-source BFS

ধর আমাদের একটা node এর সেট S দেয়া হয়েছে এবং আমাদেরকে এর কোনো একটা থেকে বাকি সব node এর দূরত্ব বের করা লাগবে। অর্থাৎ এক্ষেত্রে source node একাধিক এবং আমাদেরকে এর সব source node এর দূরত্ব এর সবচেয়ে কম দূরত্ব বের করা লাগবে। আমরা সবগুলা থেকে BFS করে সর্বনিম্ন দূরত্বটা নিতে পারি। কিন্তু এক্ষেত্রে time complexity $O(n(n+m))$ হয়ে যাবে।

Single-source BFS এ আমরা কোনো এক node nd থেকে শুরু করেছিলাম এবং $dis_{nd} = 0$ ধরে নিয়েছিলাম। কারণ কোনো এক node থেকে সেই node এরই দূরত্ব 0। Mulit-source BFS এর ক্ষেত্রে আমার সব source node এর দূরত্ব 0 করে দিতে পারি। অর্থাৎ প্রত্যেক $x(x \in S)$ এর জন্য $dis_x = 0$ ধরে নিতে পারি। তারপর প্রত্যেক x কেই আমরা q তে ঢুকায় দিব। যদি BFS ঠিক মতো বুঝে থাকো তাহলে সহজেই বুঝতে পারবে এটা কেন কাজ করে। এটারও time complexity $O(n+m)$।

1.2 Weighted Graph

Weighted গ্রাফের ক্ষেত্রে আমাদের এমন এক path বের করা লাগবে যার edge গুলার weight এর সমষ্টি সর্বনিম্ন।



যেমন উপরের গ্রাফে 1 থেকে 5 পর্যন্ত তিনটা path আছে, যেগুলা হলো: $1 \to 5$ যার দৈর্ঘ্য 13, $1 \to 4 \to 5$ যার দৈর্ঘ্য 12 আর $1 \to 2 \to 5$ যার দৈর্ঘ্য 11। এর মধ্যে সবচেয়ে ছোট হলো 11। তাই 1 থেকে 5 এর shortest path $1 \to 2 \to 5$।

আমরা weighted গ্রাফে shortest path বের করার জন্য Dijkstra বা Bellman-Ford ব্যবহার করে থাকি।

1.2.1 Dijkstra's Algorithm

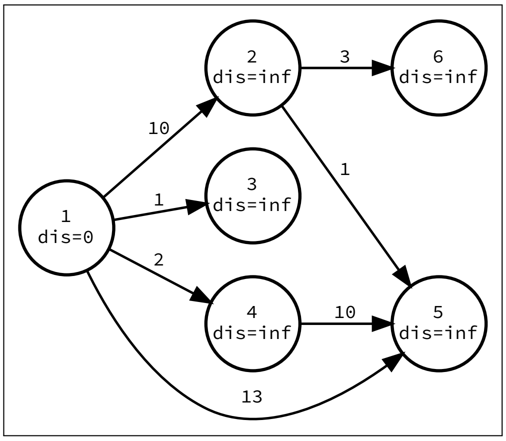
Dijkstra's Algorithm শুধু তখনই কাজ করবে যখন গ্রাফের সব edge এর weight ধনাত্মক হবে। এটা কিছুটা BFS এর মতো কাজে করে। BFS এর মতো আমরা ধরি আমাদের source node nd এবং আমরা dis অ্যারেতে দূরত্ব স্টোর করে রাখব। Dijkstra's Algorithm এর ক্ষেত্রে শুরুতে $dis_{nd} = 0$ হবে এবং বাকি সব x এর জন্য $dis_x = \infty$ হবে। আমরা একটা একটা করে সব node এর দূরত্ব **reduce** করব। কিভাবে?

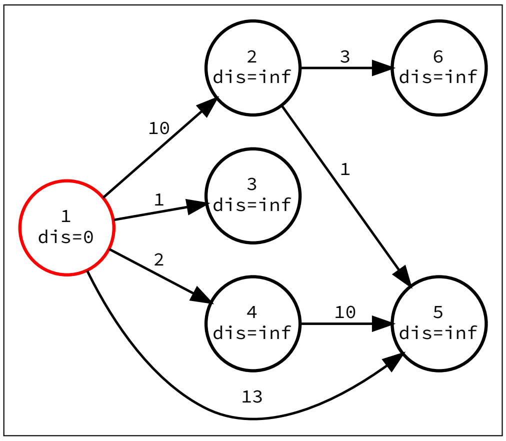
আমরা একটা সেট S রাখব। এতে শুরুতে শুধু nd থাকবে। তারপর যতক্ষণ না S ফাঁকা হয়ে যায় আমরা এটা থেকে এমন একটা node x কে select করব যার দূরত্ব (অর্থাৎ dis_x) এই মুহূর্তে সবচেয়ে কম। আমরা যখন x কে select করব এর প্রত্যেক neighbour y এর জন্য $dis_y = \min(dis_y, dis_x + W_{x,y})$ করে দিব যেখানে $W_{x,y}$ হলো x আর y এর মাঝের edge এর weight। এভাবে আমরা dis_y কে **reduce** করব। যদি $dis_x + W_{x,y} < dis_y$ হয় তাহলে dis_y **reduce** হবে। আমরা তখন y কে S এ ঢুকায় দিব। এভাবে করতে করতে যখন S ফাঁকা হয়ে যাবে তখন dis_x হবে nd থেকে x এর shortest path এর দূরত্ব। কেন?

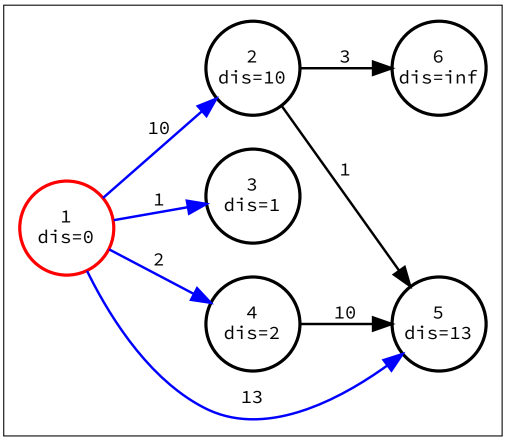
Dijkstra's Algorithm এর মজার ব্যাপার হলো যখন কোন node x কে select করা হবে তখনই এর দূরত্ব final হয়ে যাবে। কেন?

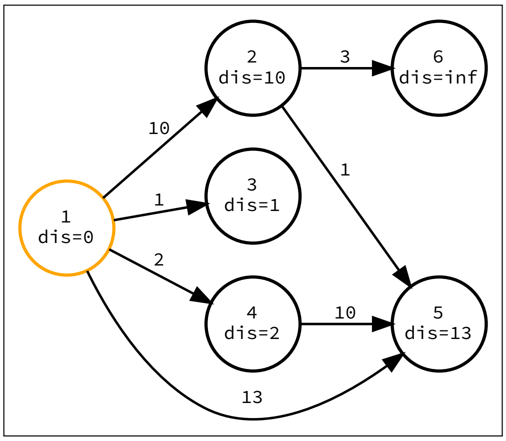
চলো এটাকে contradict করার চেষ্টা করি। ধর কোন node x এর shortest path এর দূরত্ব w কিন্তু আমরা সেটাকে consider করি নি। BFS এর মতো চিন্তা করি। যদি কোন node x এর shortest path

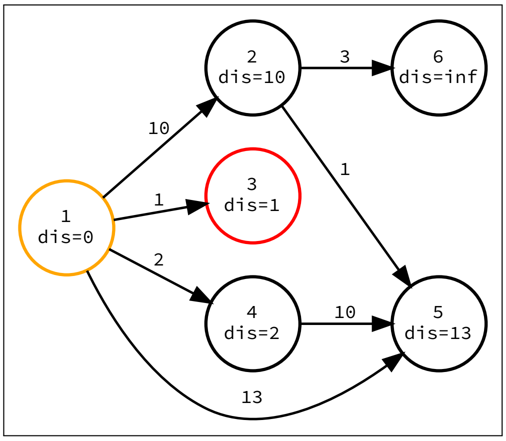
এর দূরত্ব w হয় তাহলে এটা এমন এক node এর সাথে যুক্ত আছে যার shortest path এর দূরত্ব w এর চেয়ে কম। (এটা সত্য কারণ আমরা ধরে নিয়েছি আমাদের edge গুলার weight ধনাত্মক। আমরা যদি w এর সমান বা বেশি দূরত্বের node দিয়ে কোনো node এ আসি তাহলে দূরত্ব কমবে না, উল্টা বাড়বে।) যেহেতু Dijkstra তে যার দূরত্ব কম তাকে আগে select করছি তাই আমরা সেই node কে আগেই select করে ফেলেছি এবং x এর দূরত্ব আগেই কমে w হয়ে গেছে। অর্থাৎ আমরা এটাকেও consider করেছি।

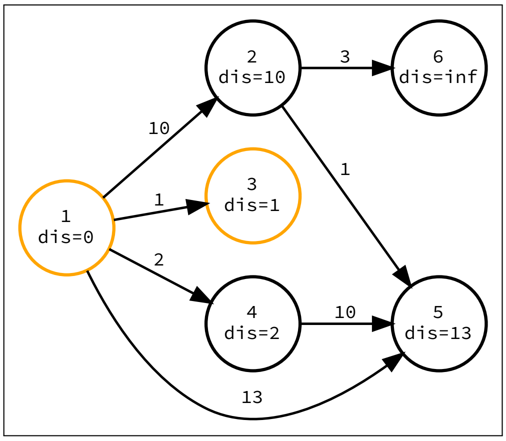


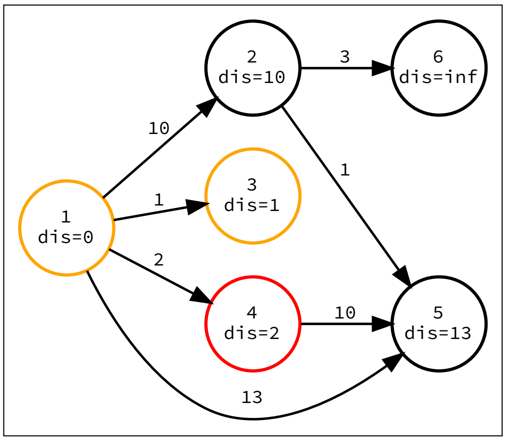


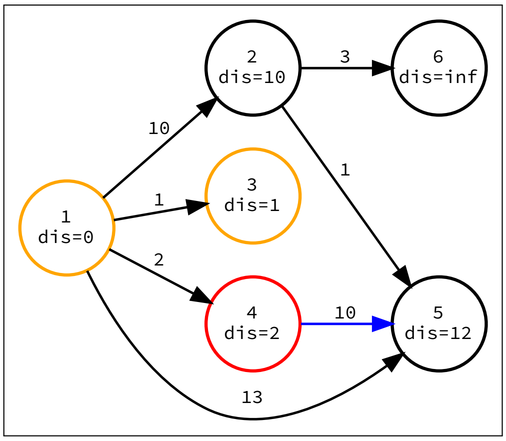


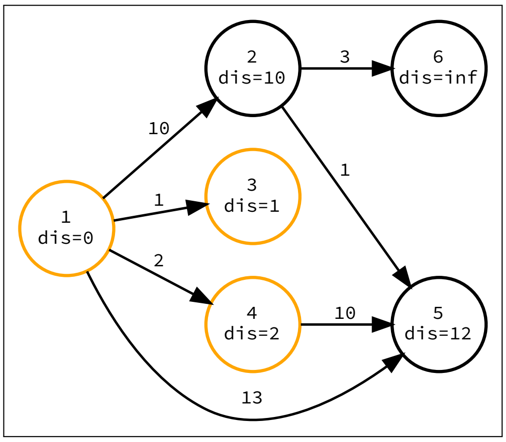


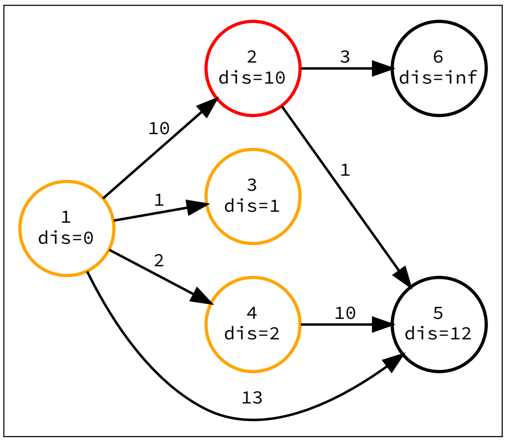


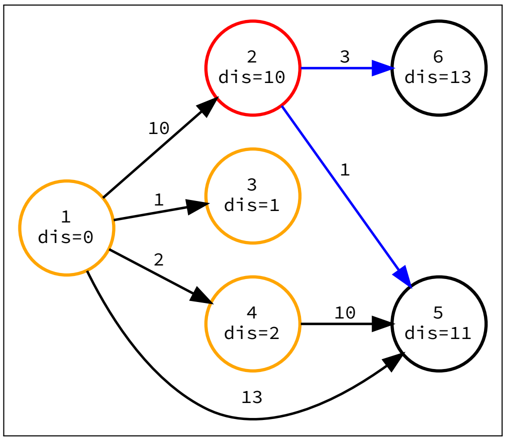


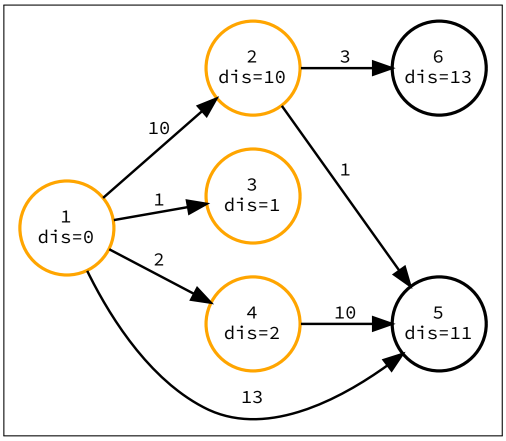

1
dis=0
2
dis=10
3
dis=1
4
dis=2
5
dis=13
6
dis=inf
10
1
2
3
1
10
13

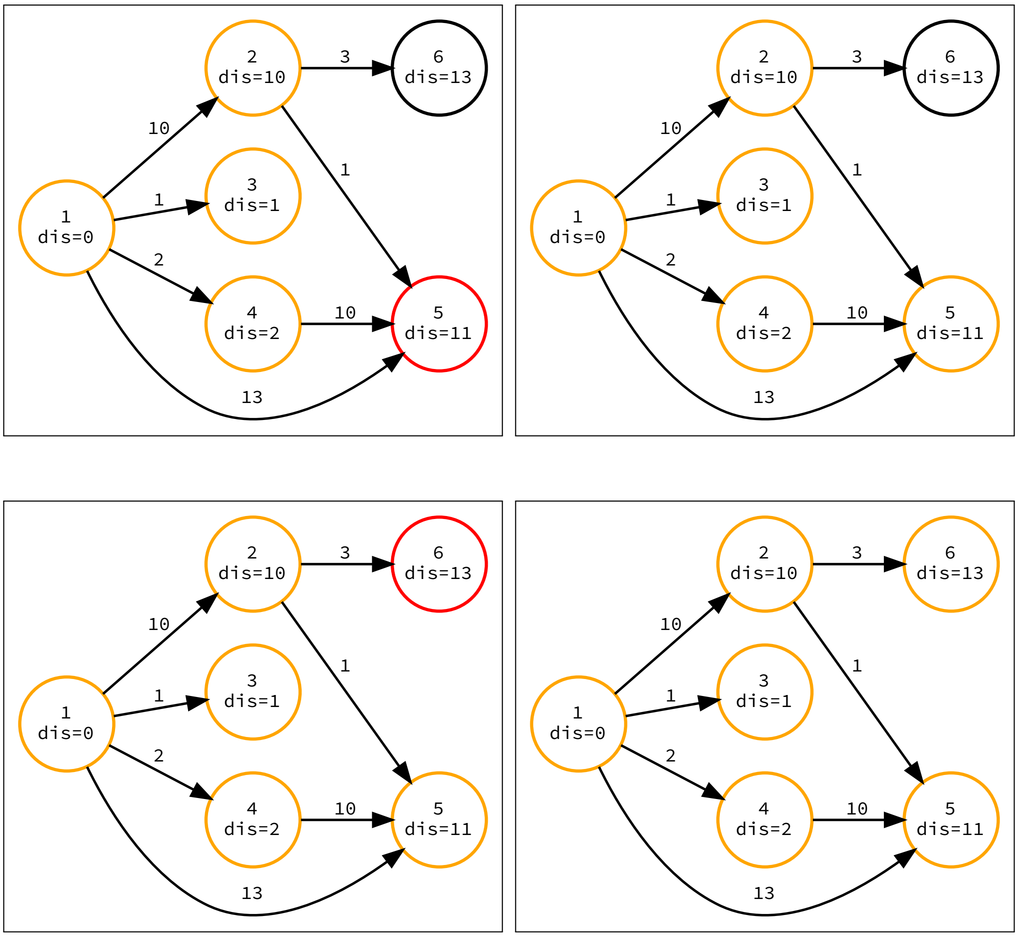

1
dis=0
2
dis=10
3
dis=1
4
dis=2
5
dis=12
6
dis=inf
10
1
2
3
1
10
13


1
dis=0
2
dis=10
3
dis=1
4
dis=2
5
dis=12
6
dis=inf
10
1
2
3
1
10
13


1
dis=0
2
dis=10
3
dis=1
4
dis=2
5
dis=12
6
dis=inf
10
1
2
3
1
10
13


1
dis=0
2
dis=10
3
dis=1
4
dis=2
5
dis=11
6
dis=13
10
1
2
3
1
10
13


1
dis=0
2
dis=10
3
dis=1
4
dis=2
5
dis=11
6
dis=13
10
1
2
3
1
10
13



Impelementation আমরা impelementation এর জন্য `std::priority_queue` ব্যবহার করতে পারি সেট S এর জন্য। এটা `std::set` এর চেয়ে একটু ফাস্ট কাজ করে। `std::priority_queue` সবচেয়ে বড় এলেমেন্টকে আগে রাখে। কিন্তু আমাদেরকে সবচেয়ে ছোটকে আগে রাখা লাগবে। এটার জন্য আমরা custom comparator ব্যবহার করতে পারি। আমরা `priority_queue<pair<long long, int>>` এর জায়গায়

```
priority_queue<pair<long long, int>, vector<pair<long long, int>>,
    less<pair<long long, int>>>
```

ব্যবহার করতে পারি।

```
// n is the number of nodes
// v is the adjacency list
// st is the starting node

priority_queue<pair<long long, int>, vector<pair<long long, int>>,
    less<pair<long long, int>>>
vector<long long> dis(n, 1e18);
vector<bool> bl(n, 0);

dis[st] = 0;
pq.push({0, st});

while(!pq.empty()){
```

```
    int nd = pq.top().second;
    pq.pop();
    if(bl[nd]) continue;
    bl[nd] = 1;

    for(auto [x, w]: v[nd]) if(dis[nd]+w < dis[x]){
        dis[x] = dis[nd]+w;
        pq.push({-dis[x], x});
    }
}
```

Dijkstra's Algorithm $O(n + m \log m)$ এ কাজ করে যেখানে n ও m হলো যথাক্রমে node আর edge এর সংখ্যা।

আমরা Multi-source BFS এর মতো Mulit-source Dijkstra ও করতে পারব। এ ক্ষেত্রে BFS এর মতোই সব source node এর dis 0 ধরে নিব আর সব source node কেই pq তে ঢুকায় দিব।

2 Shortest Path Problems

Nafis Ul Haque Shifat

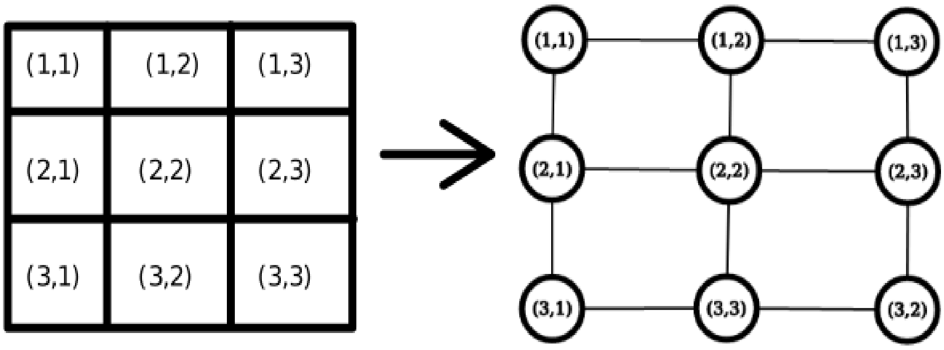
2.1 BFS Problems

2.1.1 Black and White

Statement

তোমাকে একটি $n \times m$ সাইজের গ্রিড দেয়া হবে, যার কিছু cell কালো রঙ করা, বাকিগুলো সব সাদা। গ্রিডের প্রতি cell এর জন্য তোমাকে বের করতে হবে ওই cell এর সবচেয়ে কাছের সাদা cell এর দূরত্ব। দুটি cell (i_1, j_1) ও (i_2, j_2) এর দূরত্ব হচ্ছে $|\, i_1 - i_2 \,| + |\, j_1 - j_2 \,|$।

এখানে প্রতি cell এর জন্য সবচেয়ে কাছের সাদা cell এর দূরত্ব বের করতে হবে, যেহেতু বুঝাই যাচ্ছে Shortest Path নিয়ে কিছু একটা করতে হবে, তাই প্রথমেই আমাদের গ্রিডটাকে গ্রাফ হিসেবে চিন্তা যাক। প্রতি cell (R, C) এর adjacent cell গুলো, অর্থাৎ $(R-1, C), (R+1, C), (R, C-1), (R, C+1)$ মধ্যে edge দিয়ে আমরা গ্রিড টাকে একটি গ্রাফ এ convert করতে পারি!



চিত্র 1: Grid to Graph

এবার এই গ্রাফ এ সাদা node গুলো থেকে multi source bfs চালালেই প্রতি cell থেকে সবচেয়ে কাছের সাদা cell এর দূরত্ব পেয়ে যাব!

2.1.2 Shortest path of even length!

Statement

তোমার জোড় সংখ্যা খুব ই পছন্দ! তোমার কাছে একটি গ্রাফ আছে, তুমি চাচ্ছ নোড s ও t এর মধ্যে এমন একটি পথ খুঁজে বের করতে যেন তাতে জোড় সংখ্যক এজ থাকে। এমন সব পথের মধ্যে যেই পথের দূরত্ব সবচেয়ে কম, সেই পথের দূরত্ব কত তা বের করাই তোমার কাজ!

s থেকে কোনো নোড x এ আমরা জোড় সংখ্যক edge ব্যাবহার করে পৌছাতে পারি, আবার বিজোড় সংখ্যক edge ব্যাবহার করেও পৌছাতে পারি! আমরা দুটোই মনে রাখব! ধরা যাক, $dist[0][x]$ হচ্ছে s থেকে x এ জোড় সংখ্যক edge ব্যাবহার করে আসলে মিনিমাম দূরত্ব কত, আর $dist[1][x]$ হচ্ছে বিজোড় সংখ্যক edge ব্যাবহার করে। তবে x এ আসার পর নতুন একটি edge ব্যাবহার করে y এ গেলে মোট edge এর প্যারিটি পালটে যাবে, অর্থাৎ x এ জোড় সংখ্যক edge ব্যাবহার করে এলে y তে যেতে লাগছে বিজোড় সংখ্যক, আর x এ আসতে বিজোড় সংখ্যক edge লাগলে y তে লাগবে জোড় সংখ্যক।

```cpp
while(!q.empty()) {
    ....

    for(int y : adj[x]) {
        int new_parity = 1 - parity;
        // if parity is 0, new_parity is 1 and new_parity is 0 otherwise

        if(dist[new_parity][y] > dist[parity][x] + 1) {
            dist[new_parity][y] = dist[parity][x] + 1;
            q.push({new_parity, y});
        }
    }
}

cout << dist[0][t] << endl;
```

2.1.3 Banned Triplets

Statement

একটি গ্রাফ দেওয়া আছে, যার node সংখ্যা n এবং edge সংখ্যা m $(n \leq 1000, m \leq 100000)$। সাথে k টি Triplet (a_i, b_i, c_i) দেওয়া আছে $(k \leq 100000)$। শর্ত হচ্ছে কখনো তুমি a_i, b_i, c_i পর পর visit করতে পারবে না, অর্থাৎ কখনো a_i থেকে b_i তে আসলে b_i থেকে c_i তে যাওয়া বৈধ নয়। তবে a_i, c_i, b_i সহ অন্য কোনো order এ visit করা যাবে। তোমাকে এই শর্ত গুলো মেনে 1 থেকে n এর Shortest Distance বের করতে হবে।

k টি Triplet না দেয়া থাকলে চোখ বন্ধ করে bfs চালিয়ে দেয়া যেত, তবে এবার কাজটি এত সোজা না। ধরা যাক কোনো নোড x এ এসে পৌঁছেছি, এবার নোড y তে যাব। কিন্তু কথা হচ্ছে কিভাবে বুঝব যে y

তে গেলে আমরা কোনো শর্ত ভঙ্গ করছি না? নোড w থেকে যদি আমরা x এ এসে থাকি, তবে (w, x, y) যদি Triplet হয়ে থাকে তবে y তে যাওয়া এখন বৈধ নয়। কাজেই কোনো নোড y তে যাওয়া বৈধ কিনা বুঝার জন্য আমাদের x এর আগে কোন নোড এ ছিলাম তাও মনে রাখতে হবে।

আগে আমরা $dist[x]$ এ 1 থেকে x এ আসার মিনিমাম দূরত্ব রাখতাম, এবার সাথে কোন নোড থেকে x এ এসেছি তাও রাখতে হবে। ধরা যাক $dist[w][x]$ হচ্ছে 1 থেকে x এ আসার মিনিমাম দূরত্ব, যেন x এ আসার আগে নোড w তে ছিলাম (অর্থাৎ w হয়ে সরাসরি x এ এসেছি)। এবার x থেকে y তে যাওয়ার সময় দেখব (w, x, y) বৈধ কিনা, বৈধ হলে আগে যেভাবে দূরত্ব আপডেট করতাম এখানেও সেভাবেই করব!

```
map<tuple<int,int,int>, int> banned;
.....
while(!q.empty()) {
    ....
    for(int y : adj[x]) {
        if(!banned[{w, x, y}]) {
            if(dist[x][y] > dist[w][x] + 1) {
                dist[x][y] = dist[w][x] + 1;
                q.push({x, y});
            }
        }
    }
}
```

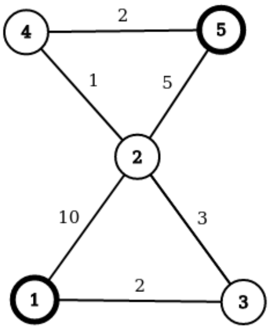
2.2 Dijkstra Problems

2.2.1 Second shortest path!

Statement

n নোড এবং m edge এর একটি weighted undirected গ্রাফ দেয়া আছে ($n \leq 10^5, m \leq 10^5$), অর্থাৎ প্রতি edge এর আলাদা Cost আছে। আমাদের 1 থেকে n এর 2nd Shortest Path বের করতে হবে।

নিচের গ্রাফটি দেখা যাক।



চিত্র 2: A weighted graph

এখানে 1 থেকে n এর বেশ কয়েকটি পথ রয়েছে। এর মধ্যে $(1 \to 3 \to 2 \to 4 \to 5)$ Path টির Cost সবচেয়ে কম (8), তাই এটি Shortest Path। আর $(1 \to 3 \to 5)$ হচ্ছে দ্বিতীয় Shortest Path, যার Cost 10।

দ্বিতীয় Shortest Path বের করব কিভাবে? ধরা যাক $dist[0][u]$ হচ্ছে 1 থেকে u এ আসার Shortest Path, আর $dist[1][u]$ হচ্ছে 1 থেকে u এ আসার দ্বিতীয় Shortest Path। ধরা যাক আমরা priority queue থেকে u pop করেছি, এর দূরত্ব হচ্ছে w। আমরা যখন Path Relaxation করব তখন আমাদের কাছে ৩টি distance থাকবে। একটি হচ্ছে $dist[0][v], dist[1][v]$ আর $w + C$, যেখানে C হচ্ছে edge (u, v) এর cost। এই ৩টির মধ্যে সর্বনিম্নটিকে $dist[0][v]$ বানিয়ে দিব, আর তার থেকে বড়টিকে $dist[1][v]$, আর সবচেয়ে বড়টির কোনো প্রয়োজন নেই আমাদের! এর আগে আমরা একটি নোড একবার ভিজিট করলেই কাজ হয়ে যেত, কারণ আমাদের Shortest Path এর দরকার ছিল, তবে এবার যেহেতু দ্বিতীয় Shortest Path দরকার, তাই একটি নোডকে আমরা দুবার ভিজিট করতে দিব।

```
while(!pq.empty()) {
    auto [w, u] = pq.top();
    pq.pop();
    if(vis[u] == 2) continue;
    vis[u]++;

    for(auto [v, C] : adj[u]) {
        if(w + C <= dist[0][v]) {
            dist[1][v] = dist[0][v];
            dist[0][v] = w + C;
            pq.push({dist[0][v], v});
        } else if(w + C < dist[1][v]) {
            dist[1][v] = w + C;
            pq.push({dist[1][v], v});
        }
    }
}
```

2.2.2 Minimum Path

Statement

n নোড এবং m edge এর একটি weighted undirected গ্রাফ দেয়া আছে $(n \leq 10^5, m \leq 10^5)$। কোনো একটি Path এর edge গুলো $e_1, e_2, ..., e_k$ হলে সেই Path এর Cost হচ্ছে $\sum_{i=1}^{k} w_{e_i} - \max_{i=1}^{k} w_{e_i} + \min_{i=1}^{k} w_{e_i}$, যেখানে w_i হচ্ছে i তম edge এর weight। আমাদের 1 থেকে n এ যাওয়ার মিনিমাম Cost বের করতে হবে।

ধরা যাক, S হচ্ছে কোনো Path এর সবগুলো edge এর weight এর যোগফল, অর্থাৎ $S = \sum_{i=1}^{k} w_{e_i}$। Cost ফাংশনটির দিকে লক্ষ্য কর, এটি আসলে বলছে, Path এর যেই edge এর weight সর্বোচ্চ সেটি S থেকে বিয়োগ যাবে,আর যেটির weight সর্বনিম্ন সেটি যোগ হবে। অর্থাৎ সর্বোচ্চ weight যার সেটি কাটাকাটি হয়ে যাবে (S এর মধ্যে সেটই একবার গুনা হয়েছে, তাই S থেকে বিয়োগ করায় কাটাকাটি হয়ে যাচ্ছে)। কাজেই Path এর Cost এ তার কোনো ভূমিকা থাকবে না। আর যার weight সর্বনিম্ন সেটি আবার যোগ হবে, কাজেই Cost এ সেটি মোট দুবার গুনা হবে (S এর মধ্যে সেটি একবার ছিল, এখন আবার যোগ হচ্ছে, মোট দুবার)।

মজার ব্যাপার হচ্ছে আমরা যদি সর্বোচ্চ weight আর সর্বনিম্ন weight এর edge যোগ-বিয়োগ করার শর্তটি হালকা পরিবর্তন করে বলি, যেকোনো edge এর weight একবার বাদ দিতে হবে, আর যেকোনো

একটি edge এর weight যোগ করতে হবে, তবে আসলে প্রব্লেমটির উত্তর পরিবর্তন হয় না! কেন? কারণ আমরা চাই Cost যত সম্ভব ছোট করতে, তাই যেকোনো দুটি edge যোগ-বিয়োগ করা যাবে শর্ততেও আমরা যদি Cost কে ছোট করতে চাই, তবে নিশ্চয়ই S থেকে যার weight সবচেয়ে বেশি তা বিয়োগ করব, আর যার weight সবচেয়ে কম তাকে যোগ করব। তাই পরিবর্তিত প্রব্লেম এর অপটিমাল উত্তর বের করলে আসলে আসল প্রব্লেম এর উত্তর ও বের হয়ে যাচ্ছে!

এবার কাজ অনেক সহজ হয়ে গিয়েছে, আমাদের Path এর সর্বোচ্চ সর্বনিম্ন edge নিয়ে মাথা ঘামাতে হবে না। এবার আমরা কোনো নোড কে ৩টি State দিয়ে প্রকাশ করতে পারি - (নোডের index, নোডে আসার পথে কোন edge বিয়োগ করা হয়েছে কিনা, নোডে আসার পথে কোনো edge যোগ করা হয়েছে কিনা)। কাজেই $dist[u][flag_1][flag_2]$ হচ্ছে 1 থেকে u তে আসার সর্বনিম্ন Cost, যেখানে $flag_1 = 1$ হবে যদি u তে আসার পথে কোনো edge এর weight বিয়োগ করা হয়ে থাকে, আর $flag_2 = 1$ হবে কোনো edge এর weight যোগ করা হয়ে থাকে। যেমন $dist[u][0][1]$ এর অর্থ হচ্ছে, u তে আসার পথে S থেকে এখনো কোনো edge এর weight বিয়োগ করা হয় নি, তবে একটি edge এর weight যোগ করা হয়েছে। এবার এটির উপর Dijkstra চালালেই আমাদের কাজ হয়ে যাচ্ছে!

```cpp
....
while(!pq.empty()) {
    //pq contains a tuple of 4 integers, denoting :
    //the cost of the path, index of the node, flag1 and flag2
    ....
    ....
    for(auto [v, c] : adj[u]) {
        if(!flag1) {
            //Now we consider subtracting the weight of this edge from S
            //This means this edge will have no contribution in the cost of the path
            if(dist[v][1][flag2] > w) {
                dist[v][1][flag2] = w;
                pq.push({dist[v][1][flag2], v, 1, flag2});
            }
        }
        if(!flag2) {
            //Now we consider adding the weight of this edge to S
            //So the weight of this edge will be counted twice in the total cost
            if(dist[v][flag1][1] > w + 2 * c) {
                dist[v][flag1][1] = w + 2 * c;
                pq.push({dist[v][flag1][1], v, flag1, 1});
            }
        }

        //Now we consider the case when -
        //the current edge will neither be added nor subtracted
        if(dist[v][flag1][flag2] > w + c) {
            dist[v][flag1][flag2] = w + c;
            pq.push({dist[v][flag1][flag2], v, flag1, flag2});
        }

    }
}
cout << min(dist[n][0][0], dist[n][1][1]) << endl;
```

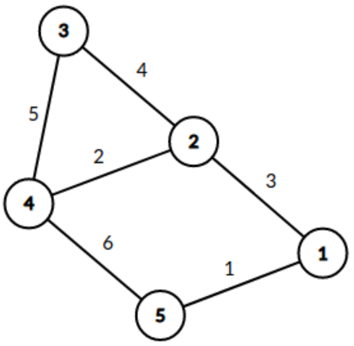
এমন ও হতে পারে যে একই edge আমরা যোগ ও করেছি, বিয়োগ ও করেছি, তখন আসলে Cost হচ্ছে S, তাই শেষে $dist[n][0][0]$ (যার মান ই আসলে S) এবং $dist[n][1][1]$ এর মধ্যকার মিনিমাম প্রিন্ট করেছি।

2.2.3 Paired Payment

Statement

একটি দেশে n টি শহর এবং শহরগুলোর মধ্যে m টি রাস্তা আছে ($n \le 10^5, m \le 10^5$)। রাস্তা এবং শহর গুলো একটি undirected weighted গ্রাফ তৈরি করে, এখানে প্রতিটি edge এর weight $w_i \le 50$। সেই দেশের সরকার হঠাৎ এক আজব নিয়ম করে বসল - কেও রাস্তা দিয়ে চলতে চাইলে তাকে পর পর দুটি রাস্তা ব্যাবহার করতে হবে! অর্থাৎ সে যদি এখন নোড a তে থাকে, তবে সে a থেকে b তে গিয়ে থামতে পারবে না, তাকে আরেকটি রাস্তা দিয়ে b থেকে অন্য শহর c তে যেতেই হবে, আর তার মোট খরচ হবে $(w_{ab} + w_{bc})^2$, এখানে w_{ab} ও w_{bc} হচ্ছে যথাক্রমে (a, b) ও (b, c) edge এর weight। তুমি নোড 1 থেকে নোড n এ যেতে চাও, সর্বনিম্ন কত খরচে যেতে পারবে?

নিচের গ্রাফটি দেখা যাক।



এখানে 1 থেকে 5 এ আমরা সরাসরি যেতে পারব না (কেনো না পরপর দুটি রাস্তা ব্যাবহার করতে হবে!) তাই আমরা $1 \to 2 \to 3$ এ যাব, খরচ হবে $(3 + 4)^2 = 49$, তারপর $3 \to 4 \to 2$ এ যাব, আরও খরচ হলো $(5 + 2)^2 = 49$, এবার $2 \to 1 \to 5$ এ যাব, আরও খরচ হবে $(2 + 3)^2 = 25$, কাজেই মোট খরচ দাঁড়াচ্ছে $49 + 49 + 25 = 114$, এটি ই সর্বনিম্ন খরচ 1 থেকে 5 এ যাওয়ার!

এখানে দুটি edge এর weight এর যোগফল এর উপর বর্গ আছে, তাই কাজটি সাদামাটা Dijkstra দিয়ে করা যাবে না। আচ্ছা লক্ষ্য করেছ কি যে সবগুলো edge এর weight সর্বোচ্চ 50 হতে পারে? বিনা কারণে নিশ্চয়ই weight এর সর্বোচ্চ মান এত কম দেয় নি!

এবার আমরা যা করব সেই আইডিয়াটি খুব সুন্দর। আমরা প্রতি নোড u এর $u_0, u_1, ..., u_{50}$ পর্যন্ত 51 টি copy বানাব! এবার আমরা এই নতুন নোড গুলোকে দিয়ে একটি নতুন গ্রাফ বানাব। এমন ভাবে বানানোর চেষ্টা করব যেন এই নতুন গ্রাফ এর উপর dijkstra চালালেই আমাদের কাজ হয়ে যায় (একটু চিন্তা করে দেখতে পার নতুন গ্রাফটা কিভাবে construct করা যায়)!

আসল গ্রাফের w weight এর প্রতিটি edge (u, v) এর জন্য আমরা নতুন গ্রাফে $u_0 \to v_w$ তে একটি 0 weight এর edge দিব। আর $c > 0$ এর জন্য প্রতি u_c এবং আসল গ্রাফে w weight এর প্রতি edge (u, v) এর জন্য নতুন গ্রাফে $u_c \to v_0$ তে $(c + w)^2$ weight এর একটি edge দিব! এবার এই গ্রাফ এর উপর 1_0 থেকে dijkstra চালালেই n_0 তে যেই Cost পাব সেটি ই আমাদের উত্তর! এটি কেনো কাজ করছে? নিজে খানিকক্ষণ চিন্তা করে দেখতে পার!

ধরা যাক আমরা আসল গ্রাফে u থেকে একটি edge দিয়ে v তে গিয়েছি, তারপর v থেকে আরেকটি edge দিয়ে x এ গিয়েছি, অর্থাৎ $u \to v \to x$ পথে গিয়েছি। (u, v) edge এর weight p আর (v, x) edge এর weight q হলে আমাদের খরচ হচ্ছে $(p + q)^2$। এবার আমাদের নতুন গ্রাফ এ এই কাজটা কিভাবে হচ্ছে তা দেখি। শুরুতে আমরা u_0 নোড থেকে 0 weight এর একটি edge দিয়ে আমরা $u_0 \to v_p$ তে যাচ্ছি (কারণ (u, v) edge এর weight p), এবার v_p থেকে $(p + q)^2$ weight এর edge দিয়ে $v_p \to x_0$ তে

যাচ্ছি! কাজেই মোট খরচ হচ্ছে $0 + (p+q)^2 = (p+q)^2$। অর্থাৎ আমরা যদি কখনো কোনো নোড x_0 তে এসে পৌছাই, তার অর্থ হচ্ছে আসল গ্রাফ এ আমরা আমরা কোনো নোড u থেকে পর পর দুটি edge ব্যাবহার করে x তে এসে পৌঁছেছি (অর্থাৎ মাঝের নোড টি v হলে $u \to v \to x$ তে এসেছি), আসল গ্রাফ এ আসতে যতখানি খরচ হত, এখানেও ঠিক ততখানি ই হয়েছে! আমরা যদি u_0 থেকে কোনো নোড v_p তে যাই, তার অর্থ হচ্ছে আসল গ্রাফে p weight এর একটি edge দিয়ে $u \to v$ তে গিয়েছি। তারপর v থেকে x এ যেতে হবে আসল গ্রাফে, সেই কাজটির জন্য নতুন গ্রাফ এ $v_p \to x_0$ তে যাচ্ছি! দুই গ্রাফেই u থেক x এ যাওয়ার খরচ সমান, পরপর দুটি edge ব্যাবহার এর শর্ত ও পূরণ হচ্ছে, কাজেই নতুন গ্রাফ এ 1_0 থেকে n_0 তে যাওয়ার খরচ আর আসল গ্রাফ এ 1 থেকে n এ যাওয়ার খরচ একই!