# Introduction to Dijkstra's

- Gaurish Baliga

# Goal

- Why BFS won't work? ✓
- Single Source Shortest Path (Dijkstra)
- Problem Solving on Dijkstra

# Why does BFS Fail?

BFS is designed for **unweighted graphs** (or all edges having equal weights). It uses the number of edges as the metric for finding the shortest path. In the given weighted graph, BFS would incorrectly assume that the shortest path from node A = 1 to B = 10 is the path with the **fewest number of edges**, ignoring the weights.

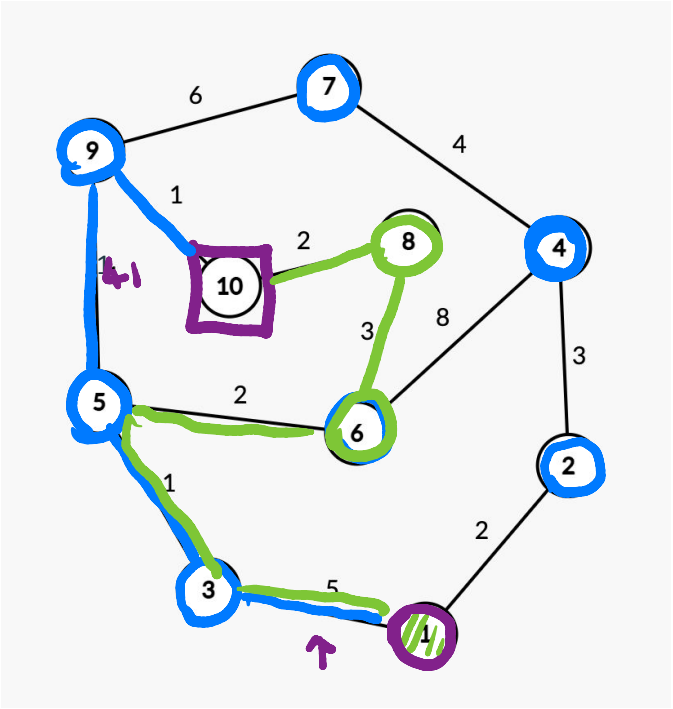
# Why does BFS Fail?

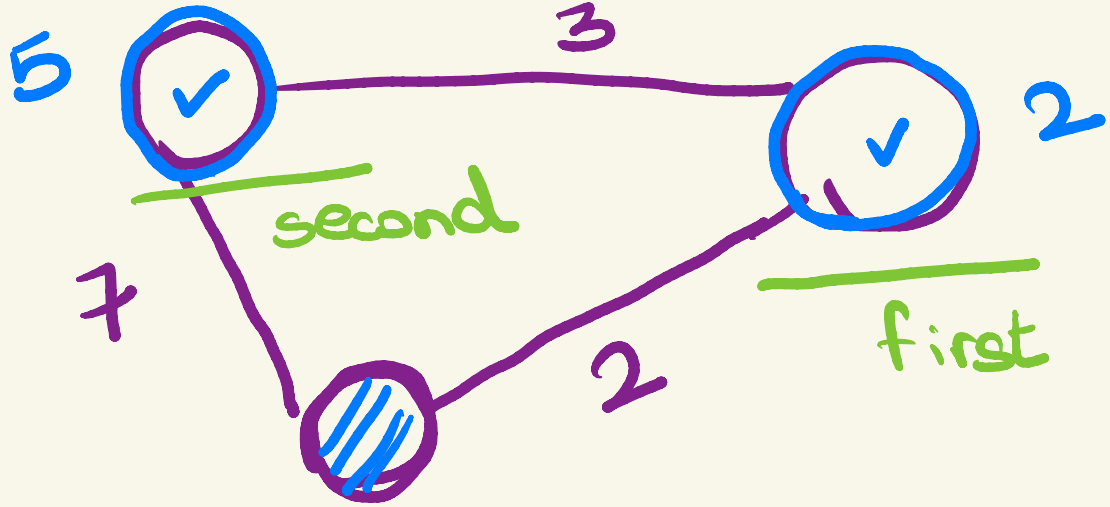
For example:

- BFS might explore the path
  1 => 3 => 5 => 9 => 10
  with a total cost of 48.

However, the true shortest path using weights is:

- 1 => 3 => 5 => 6 => 8 =>10 , with a total cost of 13 .






<dist, node>

queue ⟶ priority queue

# Dijkstra's Algorithm

Dijkstra's algorithm finds the shortest path from a source node to all other nodes in a weighted graph. It works by always expanding the node with the smallest known distance first.

Dijkstra's algorithm is a greedy algorithm that works in O((n + m)logn) time.

# Steps of Dijkstra's Algorithm

1. **Initialization:**
   - Set the distance of the source node to 0 (dist[source] = 0) and all other nodes to infinity (dist[node] = ∞).
   - Use a priority queue (min-heap) to keep track of the next node to process, based on the shortest distance.
2. **Repeat Until All Nodes Are Processed:**
   - Continue processing nodes until the queue is empty.

# Steps of Dijkstra's Algorithm

3. **Processing Nodes:**
   - Pop the node with the smallest distance from the priority queue.
   - For each neighbor of this node, calculate the potential new distance: new_dist = curr_dist + edge weight
   - If the new distance is smaller than the current recorded distance, update it and push the neighbor into the priority queue.

# Steps of Dijkstra's Algorithm

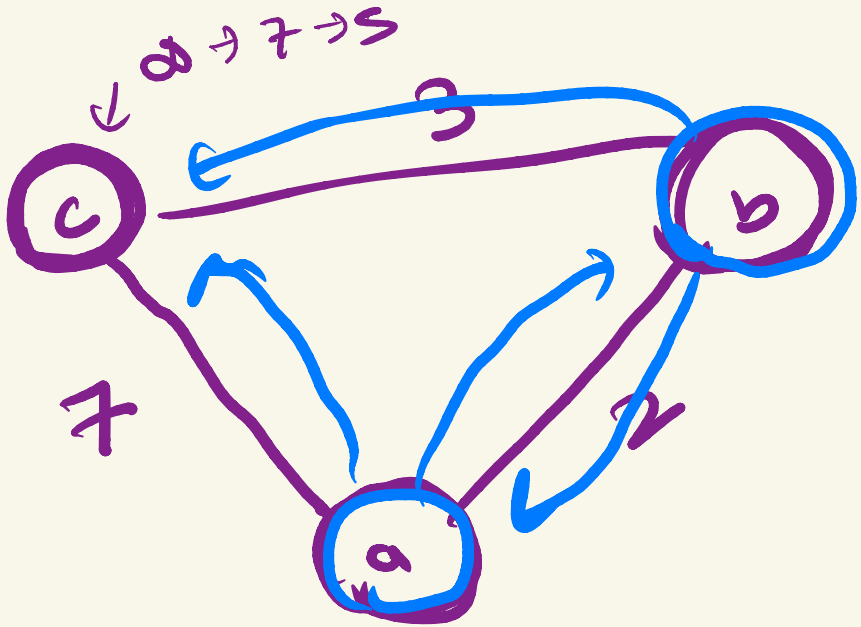
4. **Repeat Until All Nodes Are Processed:**
    - Continue processing nodes until the queue is empty

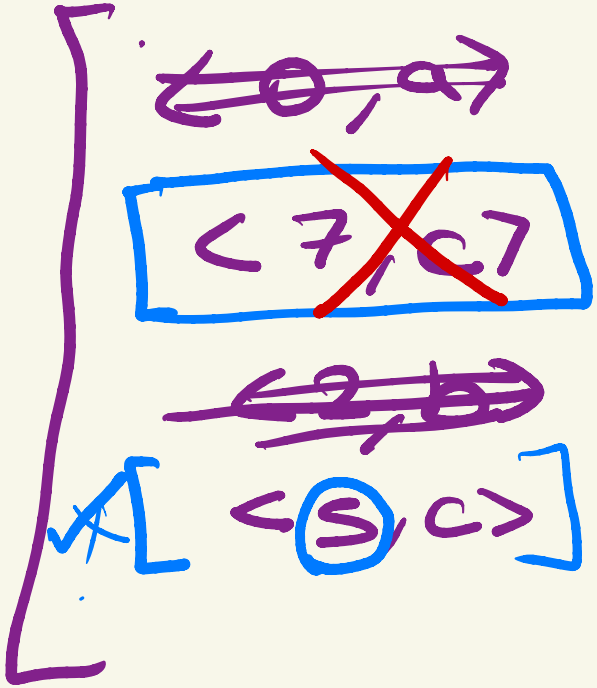
# Implementation

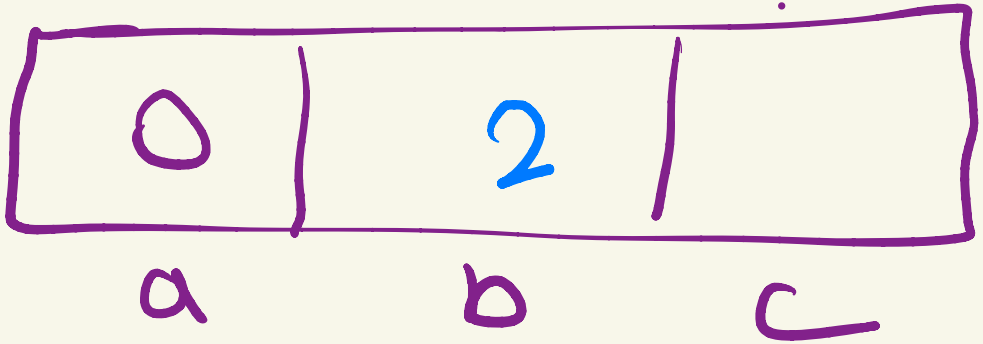
```cpp
// Function to perform Dijkstra's algorithm
vector<int> dijkstra(int n, vector<vector<pair<int, int>>>& adj, int source) {
    vector<int> dist(n + 1, INF);  // Distance array initialized to infinity
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<>> pq;

    dist[source] = 0;  // Distance to source is 0
    pq.push({0, source});  // Push source node into the priority queue

    while (!pq.empty()) {
        int currentDist = pq.top().first;
        int u = pq.top().second;
        pq.pop();

        // If the current distance is greater than the recorded distance, skip
        if (currentDist > dist[u]) continue;

        // Explore neighbors
        for (auto& neighbor : adj[u]) {
            int v = neighbor.first;      // Neighbor node
            int weight = neighbor.second; // Edge weight

            // If a shorter path is found
            if (dist[u] + weight < dist[v]) {
                dist[v] = dist[u] + weight;
                pq.push({dist[v], v});  // Push the updated distance into the pri
            }
        }
    }

    return dist;  // Return the shortest distances
}
```

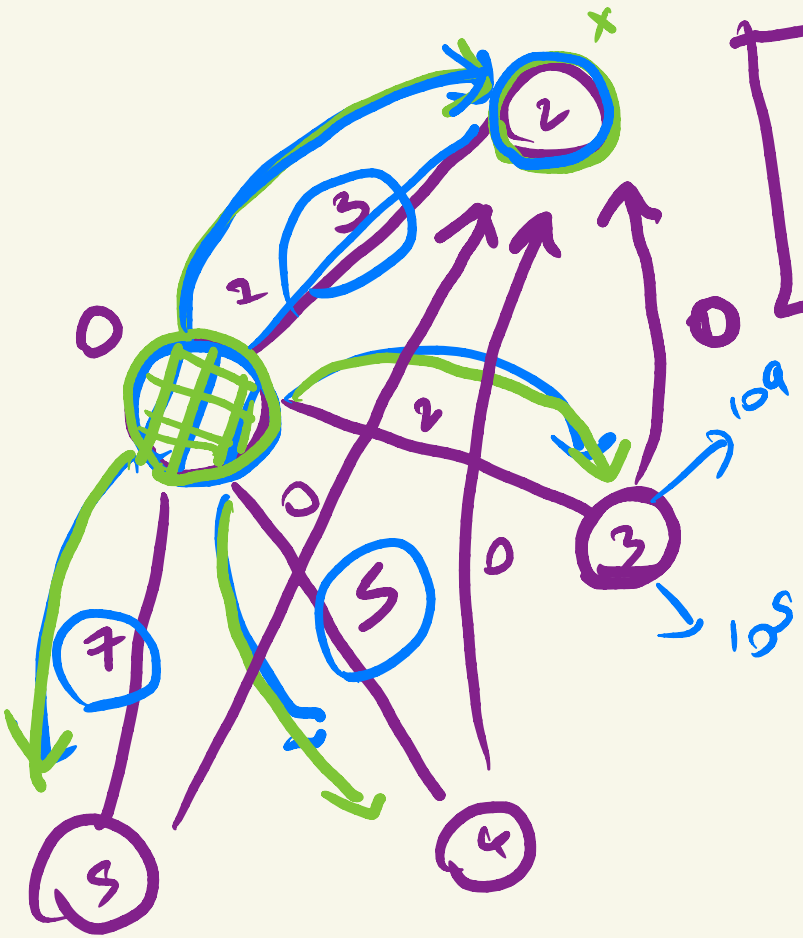
why?!

update



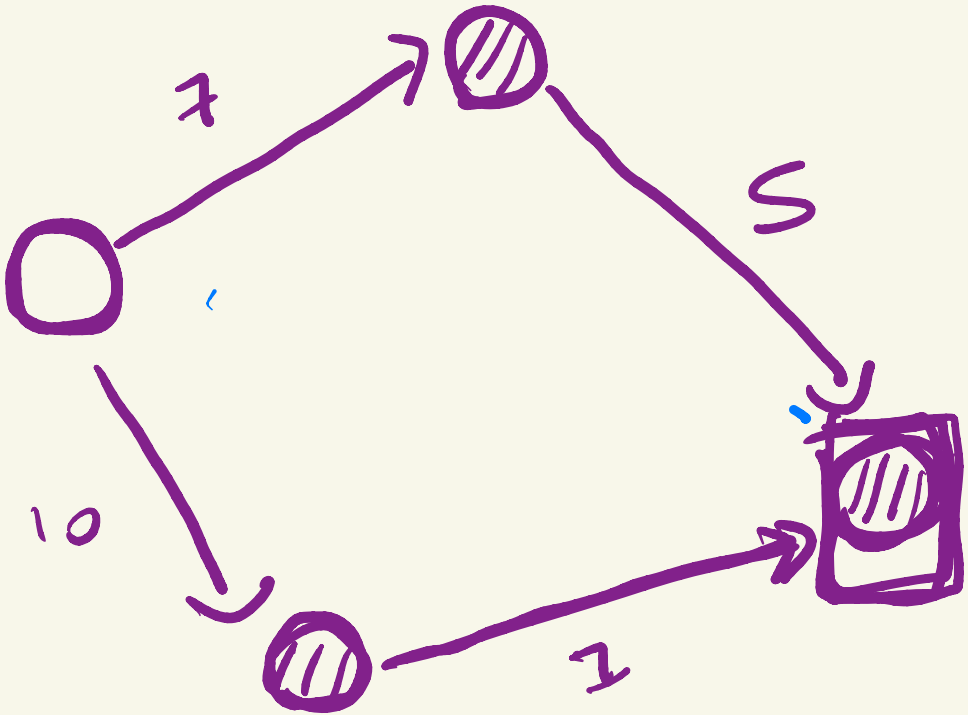






$O(n+m)$

| 0 | ∞ | ∞ | ∞ | ∞ | ∞ |
|---|---|---|---|---|---|

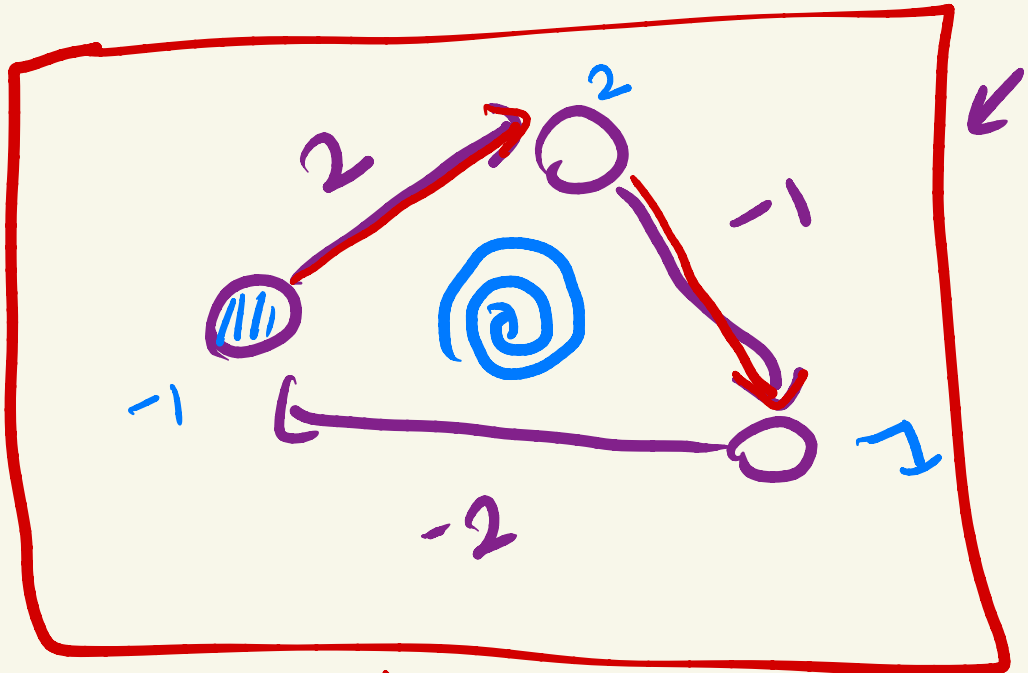
[ ⟨2, 3⟩ ]




~~<12, node>~~

<ip, node>

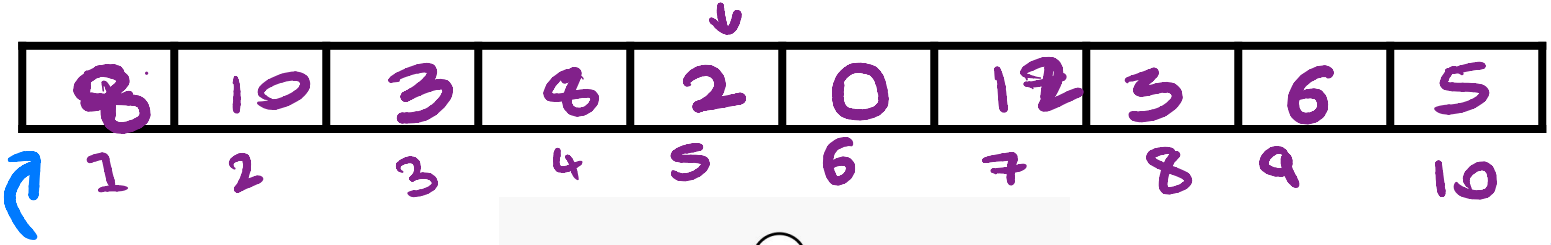
Sorted order of distances

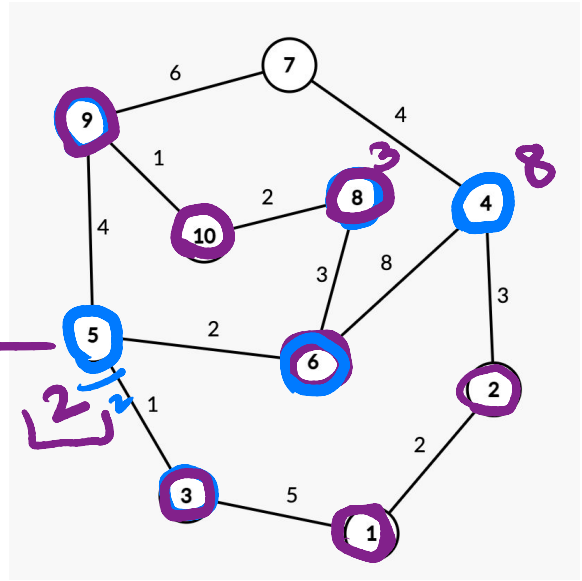



Dijkstra cannot detect this

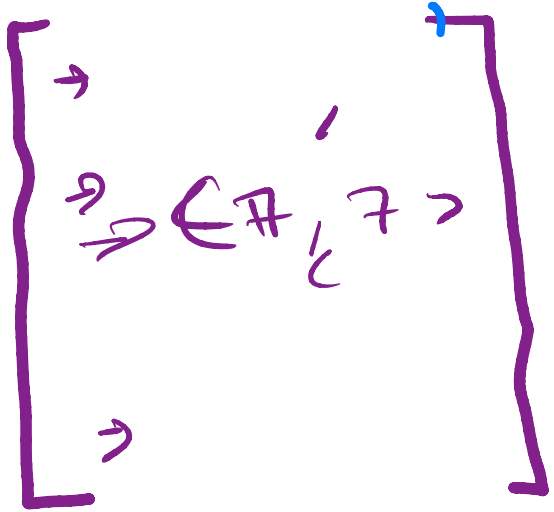
negative Cycles

# Dry Run of Dijkstra's Algorithm




dist 6 + edgew 6-5

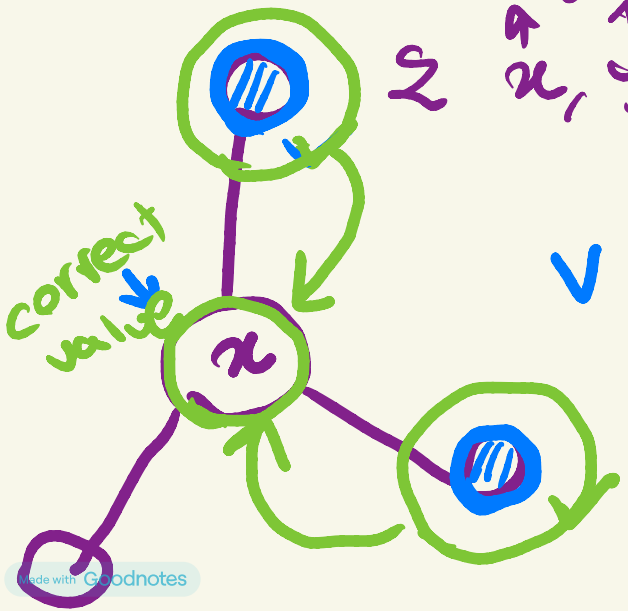


# Proof: <Contradiction>

node $x$ → smallest dist that is updated incorrectly

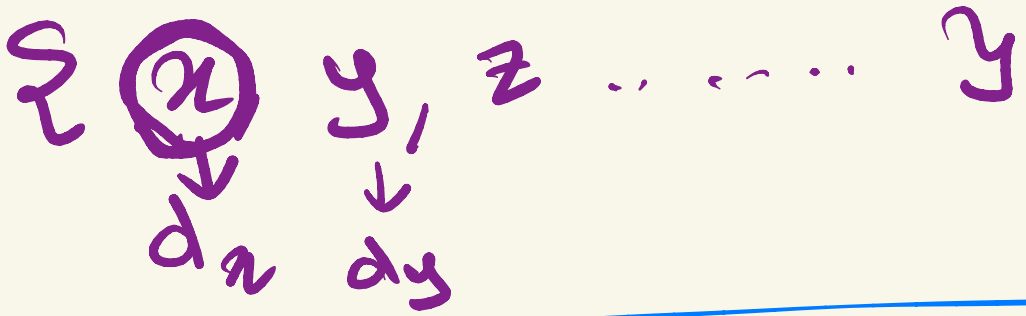
$d_x$ $d_y$ $d_z$

$x, y, z$

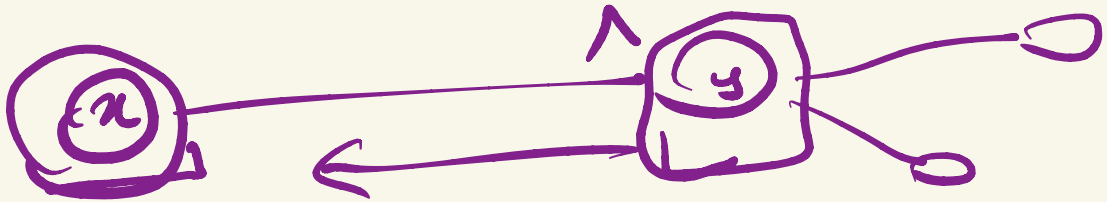



$v$ such that $d_v < d_x$

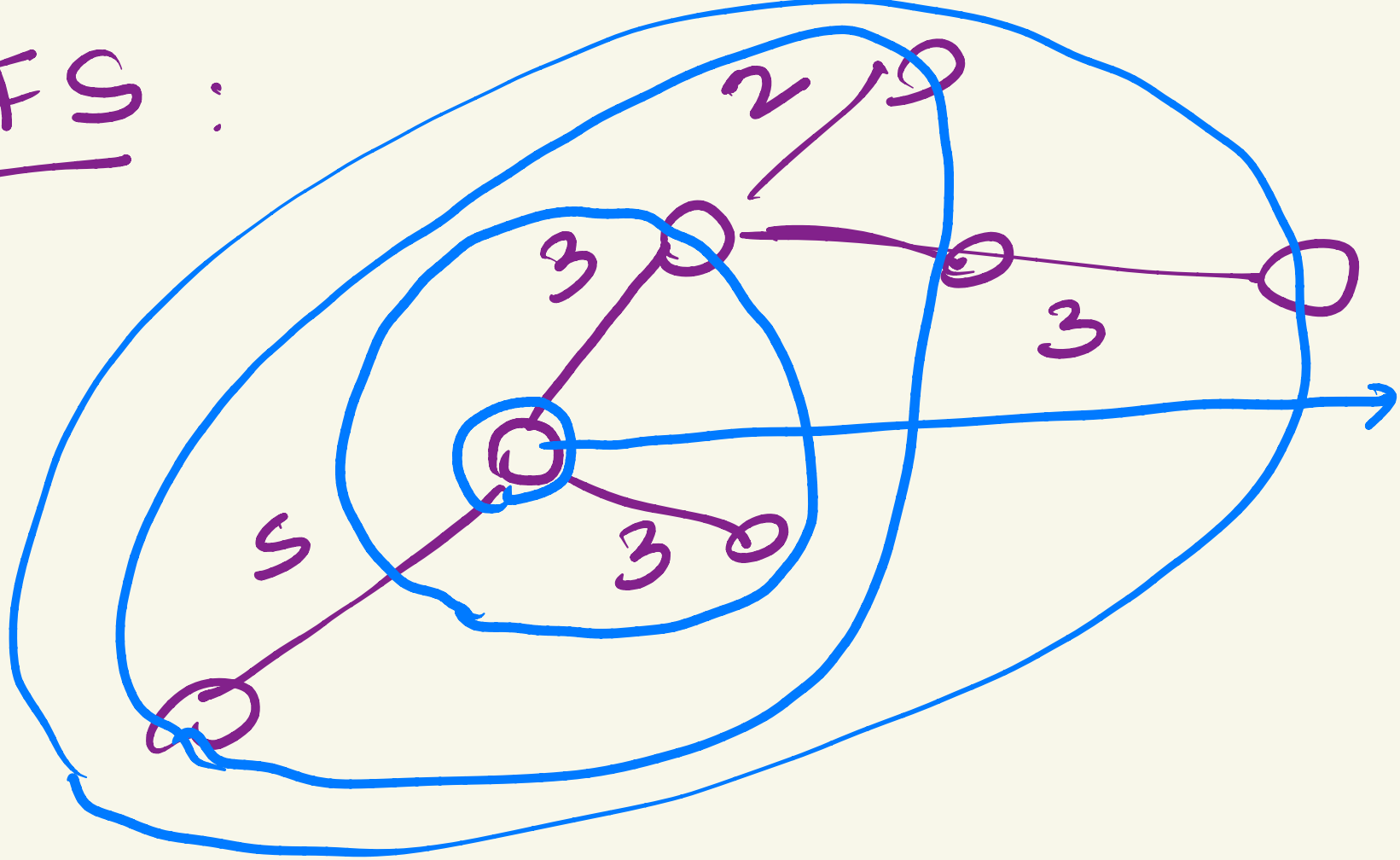
Hence, proved ✓




$\forall$ v such that $d_v < d_x$ were correctly updated



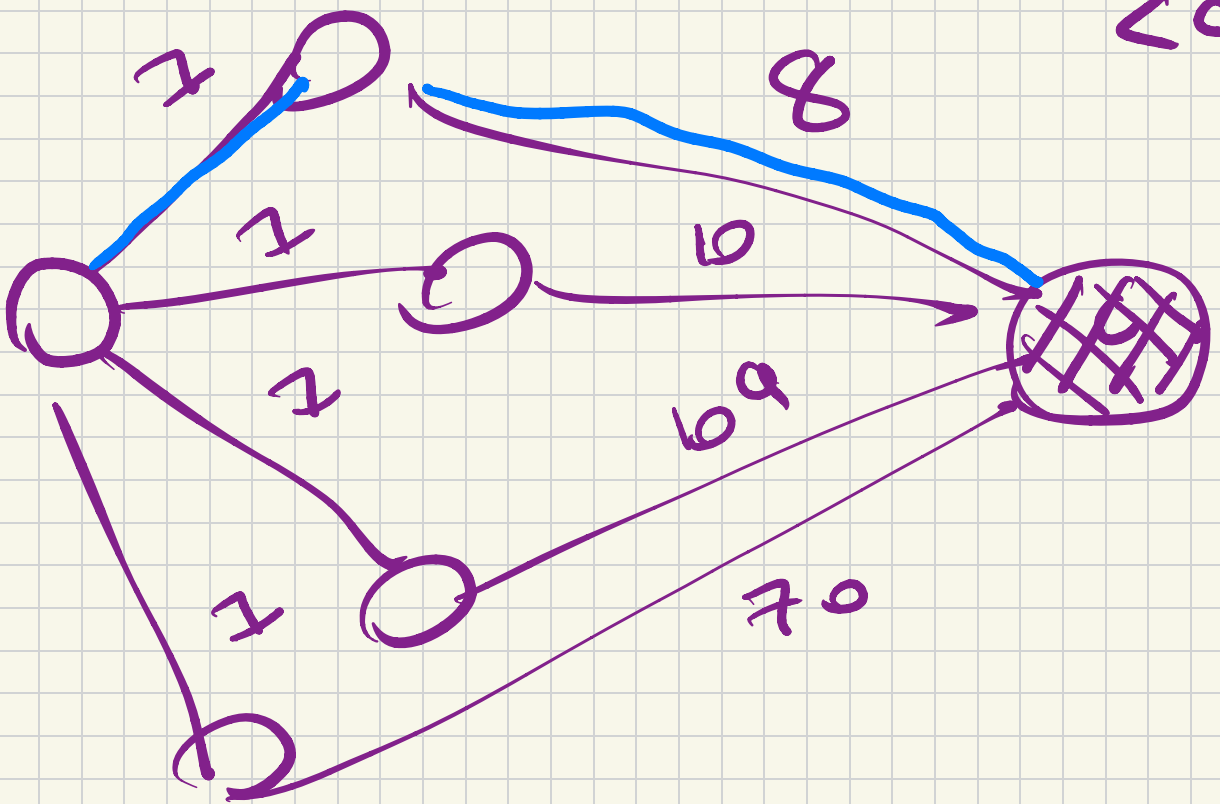
# BFS :



> $O(n^2)$

4

<dist, c>



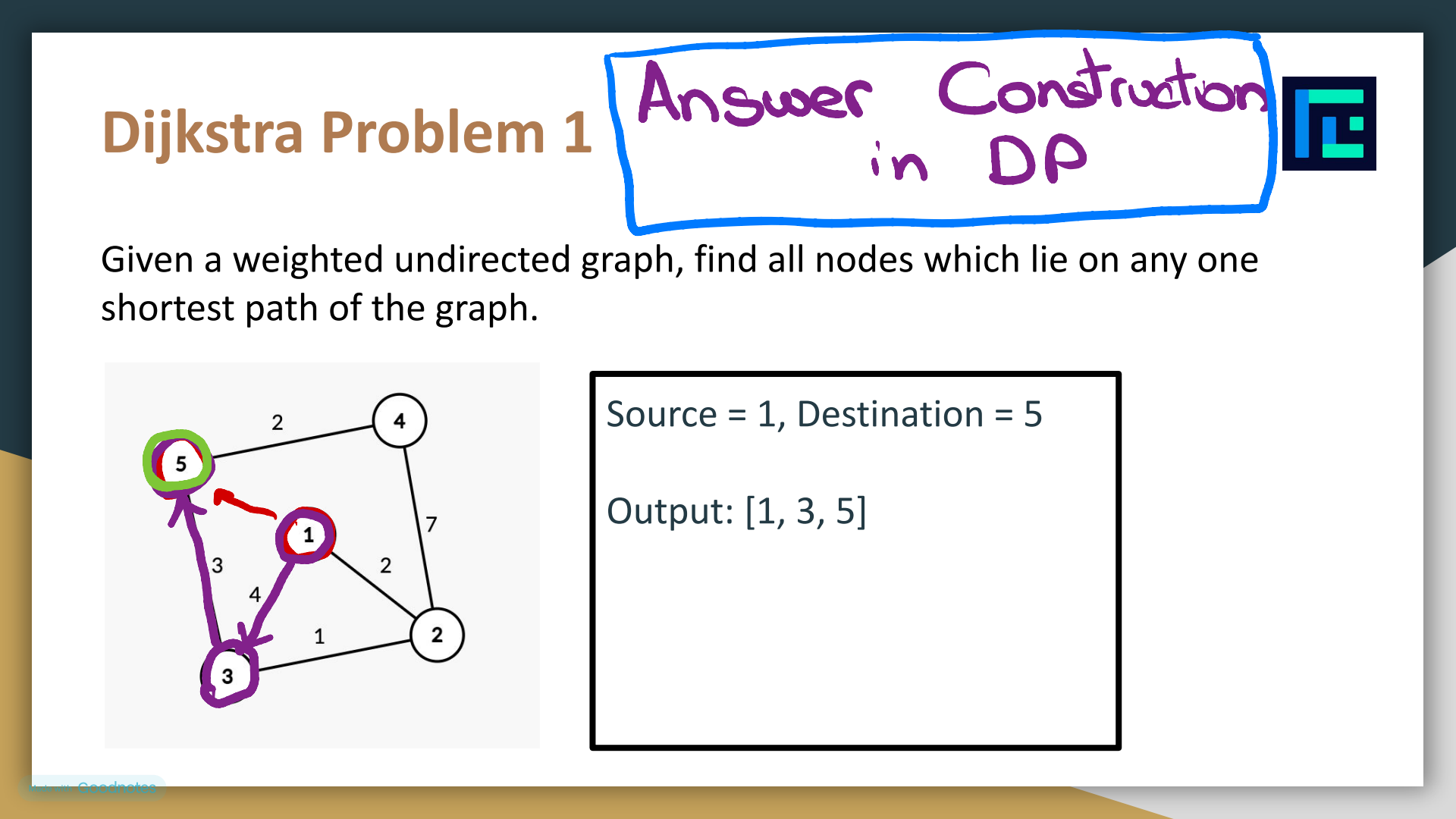
# Proof of Dijkstra's Algorithm

**Proof by Contradiction:**

1. Let x be the node with the smallest shortest distance whose shortest distance is incorrectly found using Dijkstra's Algorithm.
2. Let this node's actual shortest distance be d.
3. Now this means that all nodes with distance < d have been correctly found.
4. But if that is true, then it means that all nodes adjacent to node x with smallest distance < d had a chance to update the distance of node x.
5. Which means that the shortest distance must have been found.

# Dijkstra Problem 1

Answer Construction in DP

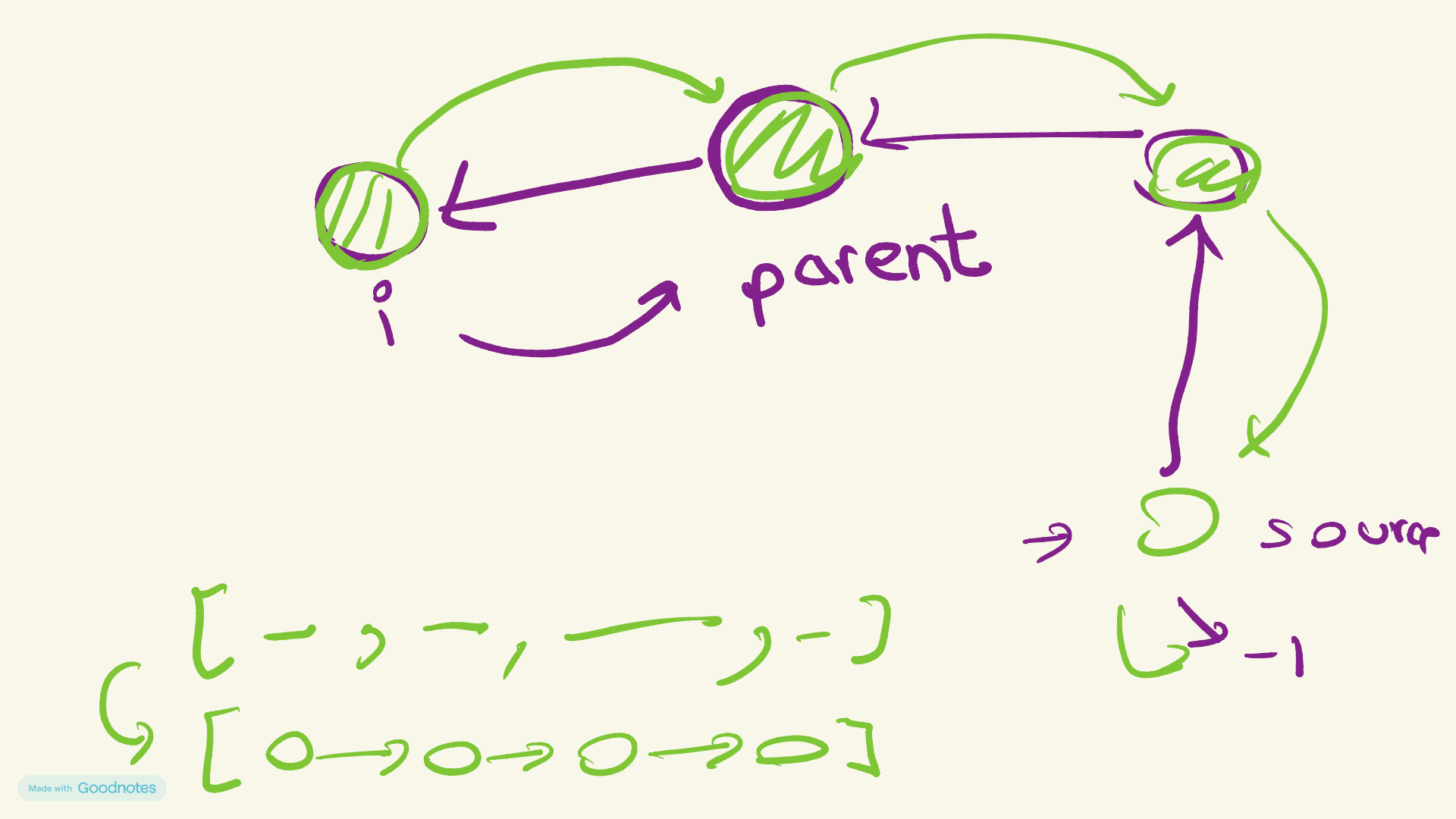


Given a weighted undirected graph, find all nodes which lie on any one shortest path of the graph.

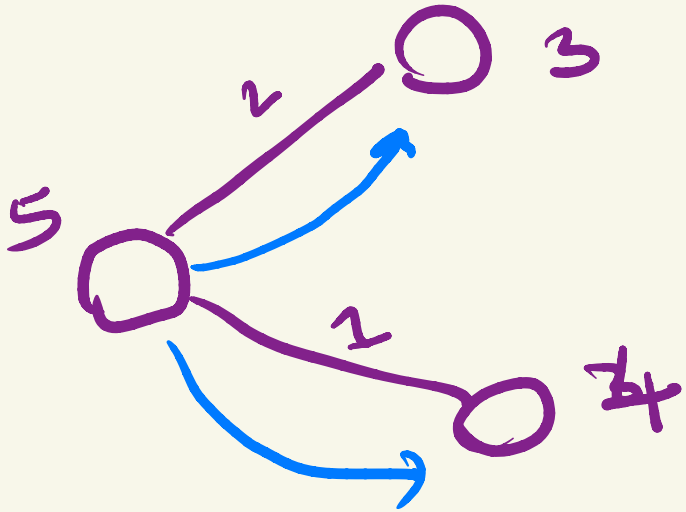



Source = 1, Destination = 5

Output: [1, 3, 5]



# There can be multiple shortest paths



# Implementation

```
vector<int> dist(n + 1, INF), parent(n + 1, -1);

priority_queue<pair<int, int>, vector<pair<int, int>>, greater<>> pq;

dist[source] = 0; pq.push({0, source});

while (!pq.empty()) {
    int currentDist = pq.top().first;
    int u = pq.top().second;
    pq.pop();

    if (currentDist > dist[u]) continue;

    for (auto& neighbor : adj[u]) {
        int v = neighbor.first;
        int weight = neighbor.second;

        if (dist[u] + weight < dist[v]) {
            dist[v] = dist[u] + weight;
            parent[v] = u;
            pq.push({dist[v], v});
        }
    }
}
```

Ans. construction.

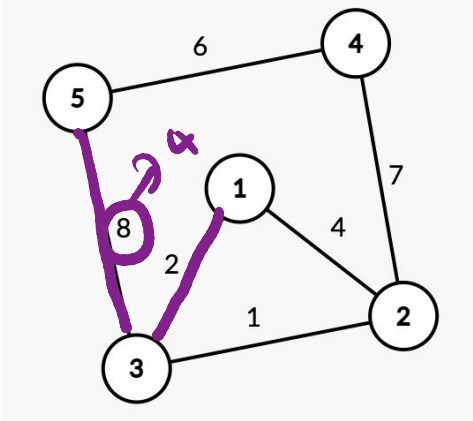
```
vector<int> path; int v = destination;
while(v != -1) {
    path.push_back(v);
    v = parent[v];
}
reverse(path.begin(), path.end());
for(auto &i : path) cout << i << " ";
}
```

# Dijkstra Problem 2

# Good Idea

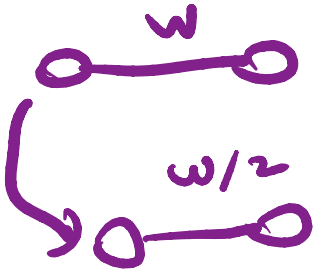


Finding shortest path from source to destination if you are allowed to reduce the weight of any one edge in the graph to 50% of its current weight.

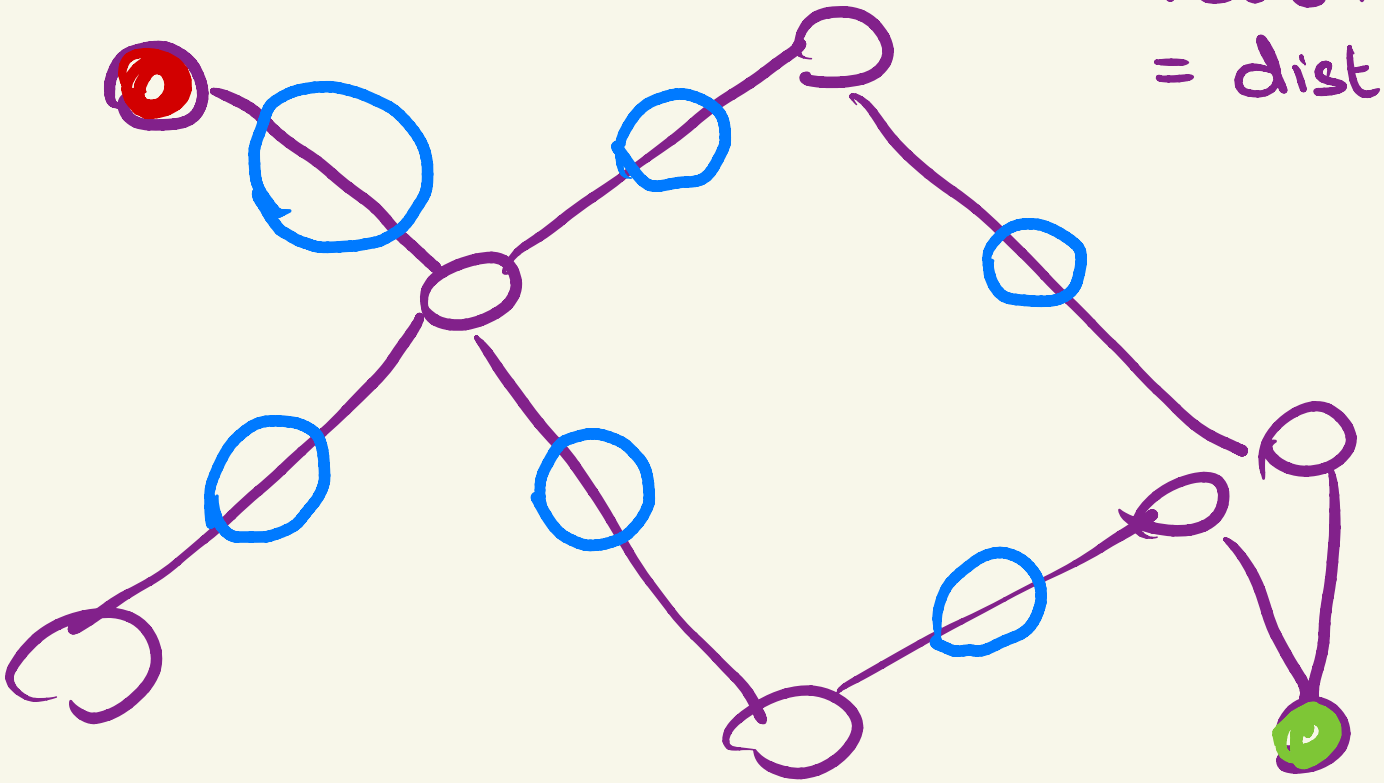



Source = 1, Destination = 5

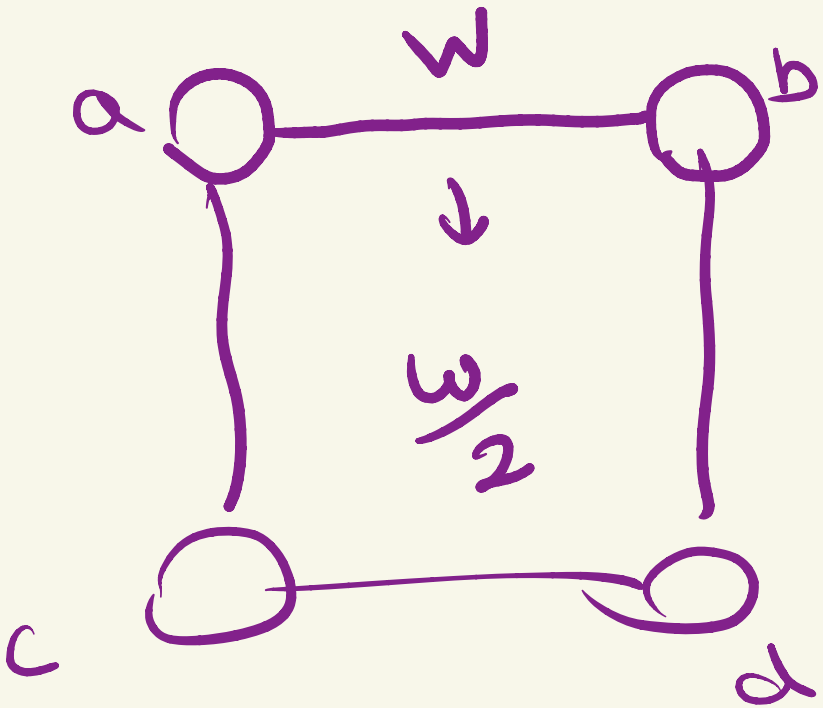
Output: 6 (2 + (8 / 2))

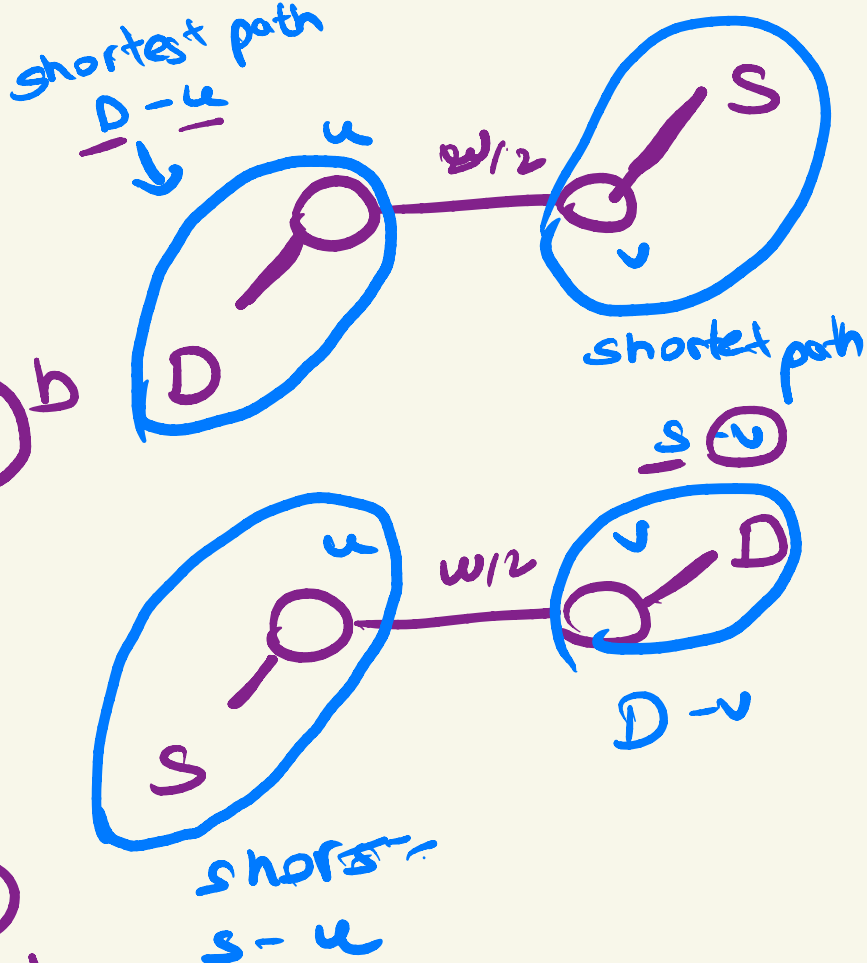


dist(s,d)
= dist(d,s)

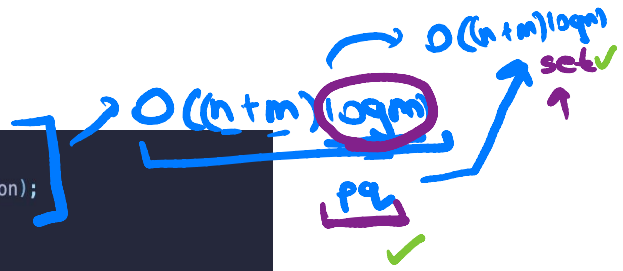


a - b - c - d

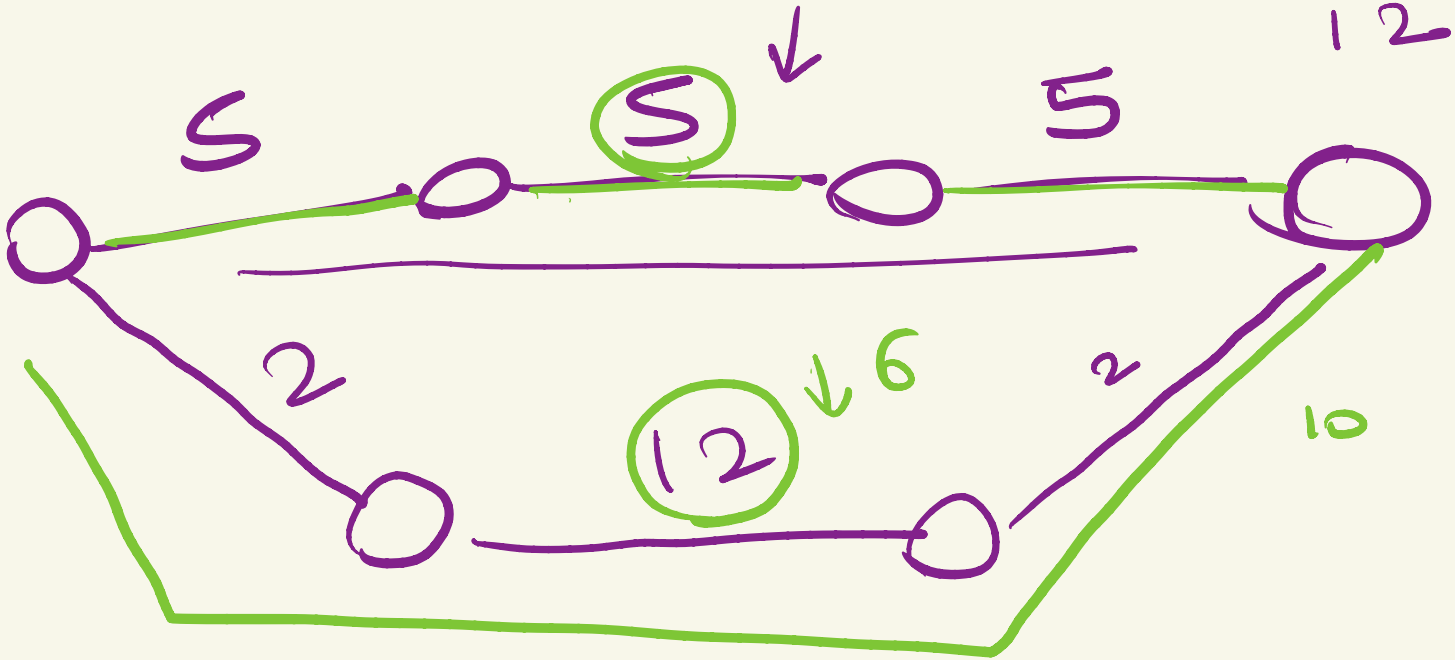


# Implementation



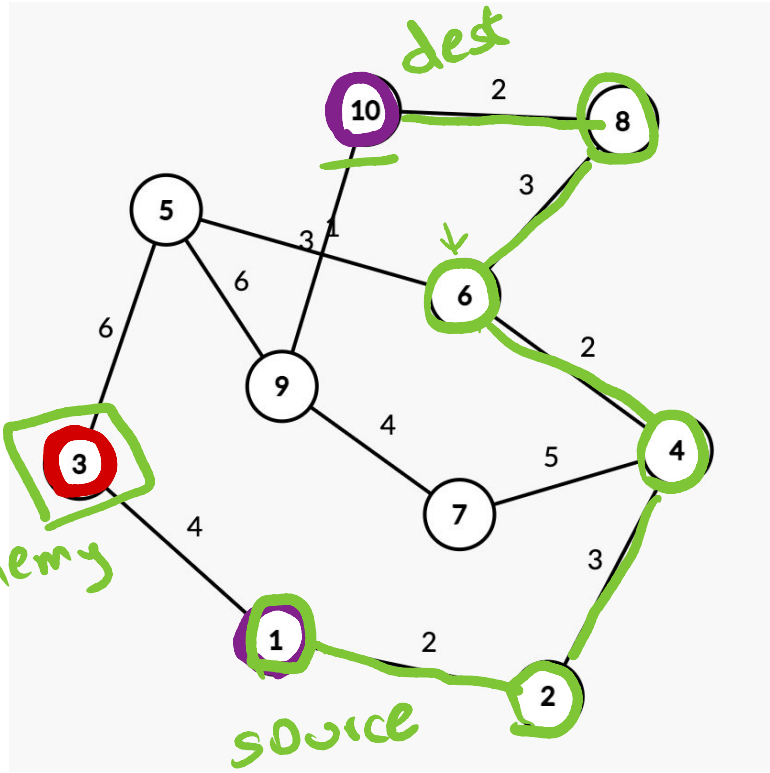

```cpp
    vector<int> distFromSource = dijkstra(n, adj, source);
    vector<int> distFromDestination = dijkstra(n, adj, destination);

    int minDist = distFromSource[destination];

    for (auto& edge : edges) {
        int u, v, w;
        tie(u, v, w) = edge;

        // Reduce the weight of this edge to 50%
        int reducedWeight = w / 2;

        // Calculate the distance if this edge is reduced
        int newDist = distFromSource[u] + reducedWeight + distFromDestination[v];
        minDist = min(minDist, newDist);

        // Also consider the edge in the opposite direction
        newDist = distFromSource[v] + reducedWeight + distFromDestination[u];
        minDist = min(minDist, newDist);
    }

    cout << "Shortest path distance with one edge reduced by 50%: " << minDist << endl;
}
```



# Dijkstra Problem 3

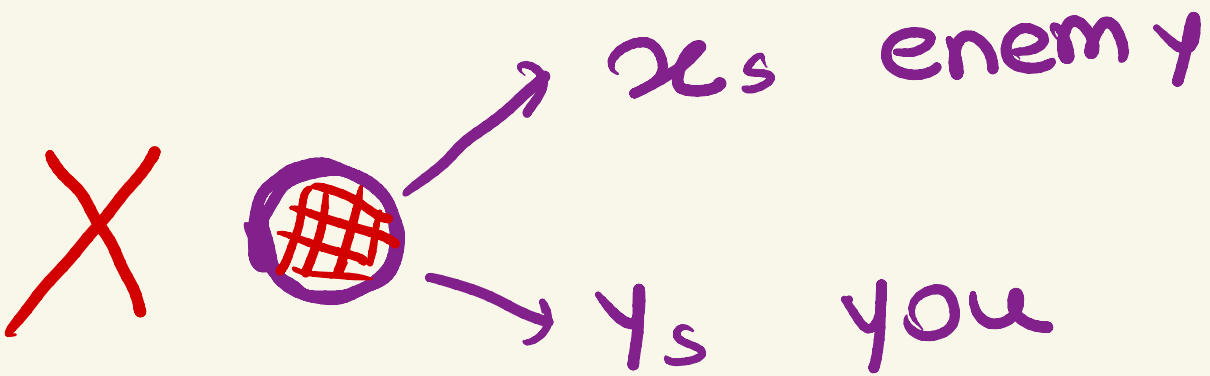
You are standing at node A in the graph and you want to reach node B. There is an enemy at node C who can block you from reaching B. Before you start your journey from A to B, you have to inform enemy about the path you're going to take. Find out if it is possible to reach B from A without getting blocked by enemy.

The enemy can block you by reaching a node in your decided path before you reach it.

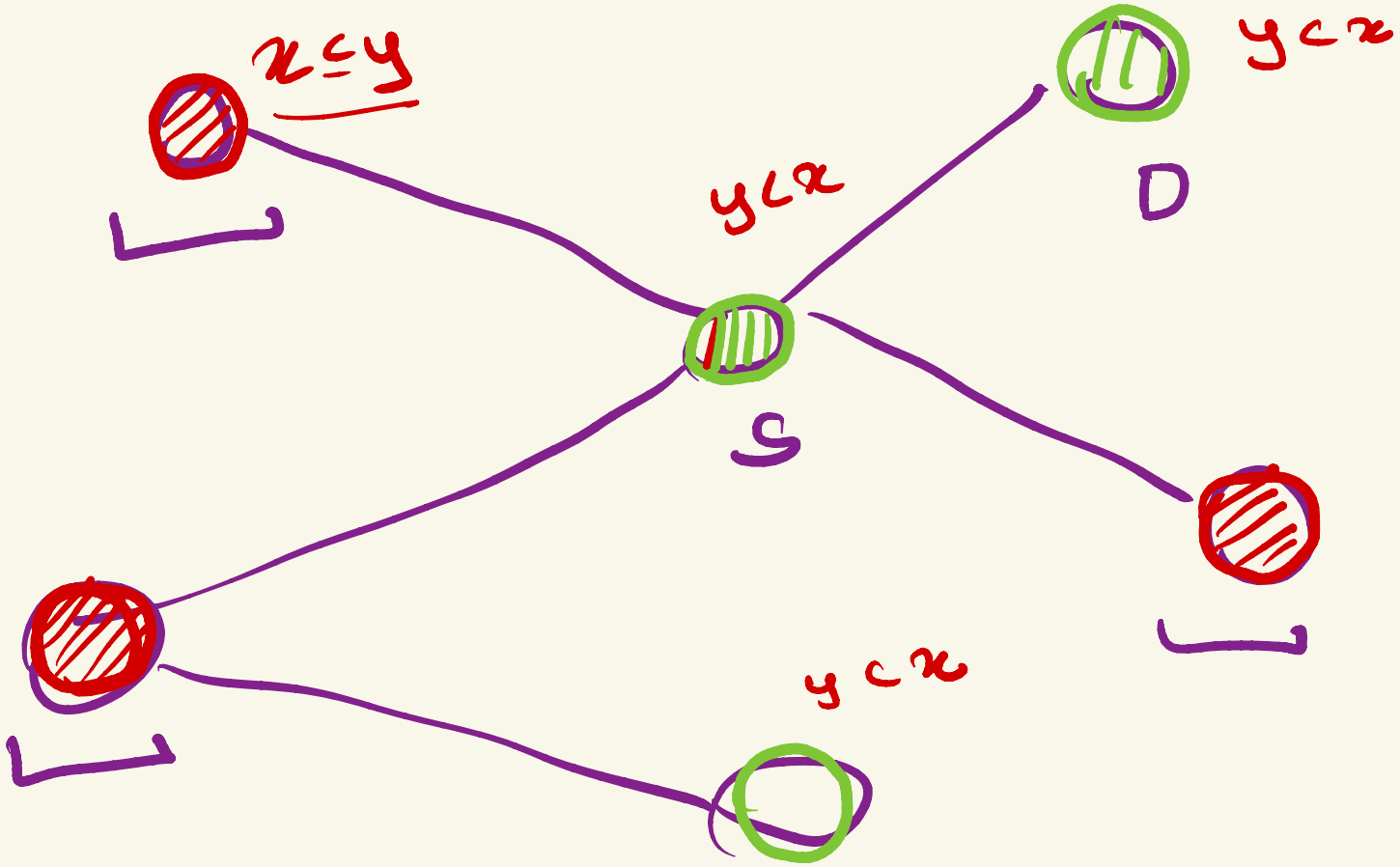



A = 1, B = 10, C = 3

Output: Yes




$$x \leq y$$



# Implementation

```
bool canReachSafely(int n, vector<vector<pair<int, int>>>& adj, int A, int B, int C) {
    vector<int> dist(n + 1, INF), distFromC = dijkstra(n, adj, C);
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<>> pq;

    dist[A] = 0; pq.push({0, A});

    while (!pq.empty()) {
        int currentDist = pq.top().first;
        int u = pq.top().second;
        pq.pop();

        if (u == B) return true; // Reached B safely

        for (auto& neighbor : adj[u]) {
            int v = neighbor.first;
            int weight = neighbor.second;

            // Only explore if A can reach v before C
            if (dist[u] + weight < dist[v] && dist[u] + weight < distFromC[v]) {
                dist[v] = dist[u] + weight;
                pq.push({dist[v], v});
            }
        }
    }
    return false; // Could not reach B
}
```