Union Find

Problems Discuss

< Back(/tag/union-find/discuss) Graph algorithms + problems to practice

(/A1exrebys) A1exrebys (/A1exrebys) ★ 768 Last Edit: April 5, 2022 4:28 AM 38.9K VIEWS

478

Hi folks! I created a small list of graph problems that can be useful to memorize/practice to solve more graph problems (from my point of view). We should not memorize the algorithm itself, but rather the idea and some implementation notes. So under each algorithm I put notes which can help to remember the algorithm.

P.S.: Please put comments how do you remember algorithms or ideas, and I will update the this post with interesting notes.

Update: updated with Eulearian Path, visualization of BFS, UnionFind, Dijkstra, Topological Sort and Bellman Ford.

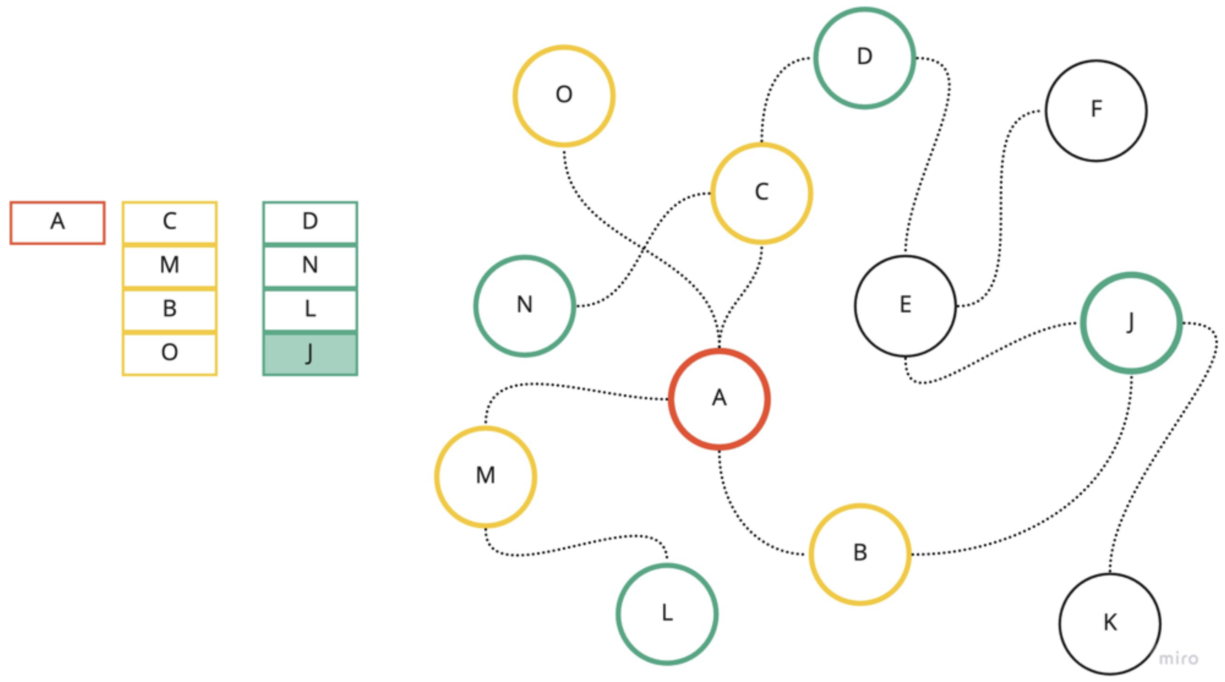
Graph problems

Description:
In computer science, graphs are used to represent networks of communication, data organization, computational devices, the flow of computation, etc. For instance, the link structure of a website can be represented by a directed graph, in which the vertices represent web pages and directed edges represent links from one page to another.

Common algorithms

BFS:
Breadth-first search (BFS) is an algorithm for searching a tree data structure for a node that satisfies a given property. It starts at the tree root and explores all nodes at the present depth prior to moving on to the nodes at the next depth level. Extra memory, usually a queue, is needed to keep track of the child nodes that were encountered but not yet explored.
Part of a solution for the problem "Evaluate Division": https://leetcode.com/problems/evaluate-division/ (https://leetcode.com/problems/evaluate-division/)

```
double bfs(unordered_map<string, vector<ip>>& adj, vector<string>& query) {
    unordered_set<string> visited;
    string start = query[0];
    string end = query[1];
    queue<ip> q;
    q.push({start, 1.0});
    visited.insert(start);
    while (!q.empty()) {
        int sz = q.size();
        for (int i = 0; i < sz; i++) {
            auto [node, cost] = q.front(); q.pop();
            if (!adj.count(node)) continue;
            if (node == end) return cost;
            for (auto& a : adj[node]) {
                if (!visited.count(a.first)) {
                    q.push({a.first, cost * a.second});
                    visited.insert(node);
                }
            }
        }
    }
    return -1.0;
}
```



Visualization of BFS with first 3 layers. A - root node/first lavel, C/M/B/O - second layer, D/N/L/J - third layer.

Notes:

1. Algorithm uses queue for implementation.
2. It checks whether a vertex has been explored before enqueueing the vertex.
3. We can track if the node was already explored by modifying the original matrix.
4. BFS algorithm can be instructed with additional array dist which can help to track the parent node of the next node. This will help to reconstruct the path by looping backward from end node.

DFS:

Is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.

```
void dfs(unordered_map<int, vector<int>>& graph, int node, unordered_set<int>& visited) {
    visited.insert(node);
    for (auto& n : graph[node]) {
        if (!visited.count(n)) {
            dfs(graph, n, visited);
        }
    }
}
// Find connected components.
int count = 0;
for (int i = 0; i < n; i++) {
    if (!visited.count(i)) {
        count++;
        dfs(graph, i, visited);
    }
}
```

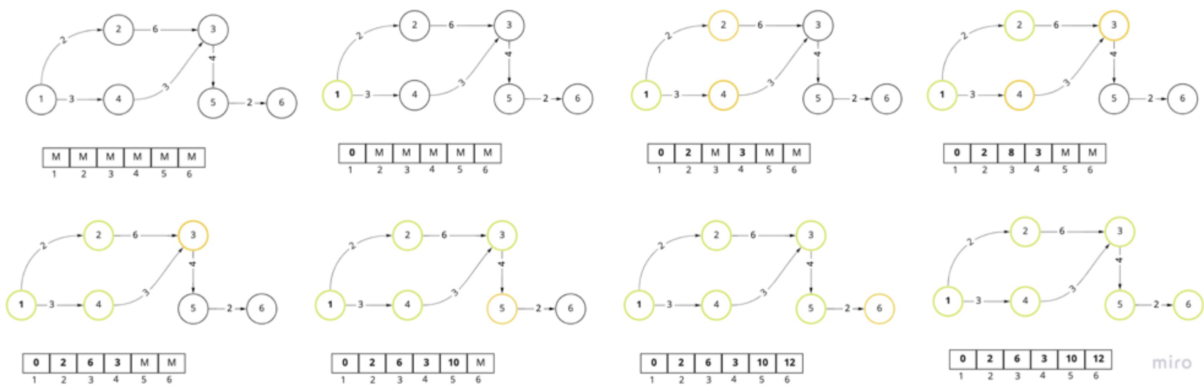
Notes:

1. Algorithm usually uses recursion implementation.
2. We mark the node as visited and will keep exploring its neighbors if there are not yet explored.
3. DFS can be useful to find connected components. We can iterate through the nodes and call dfs() to find all nodes which belongs to component.
4. We can use unordered_map<int, vector> to represent the graph.

Dijkstra's algorithm:

Is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks.

```
class Solution {
public:
    int networkDelayTime(vector<vector<int>>& times, int n, int k) {
        using ip = pair<int, int>;
        vector<vector<ip>> adj(n + 1);
        for (auto& t : times) adj[t[0]].push_back({t[1], t[2]});
        priority_queue<ip, vector<ip>, greater<ip>> pq;
        vector<int> dist(n + 1, INT_MAX);
        vector<bool> visited(n + 1, false);
        pq.push({0, k});
        dist[k] = 0;
        while (!pq.empty()) {
            auto [n_cost, node] = pq.top(); pq.pop();
            visited[node] = true;
            if (dist[node] < n_cost) continue;  // Optimization: skip node if we already find better option.
            for (auto& [next, cost] : adj[node]) {
                if (visited[next] == true) continue; // Optimization: do not re-visit nodes.
                if (dist[next] > dist[node] + cost) {
                    dist[next] = dist[node] + cost;
                    pq.push({dist[next], next});
                }
            }
        }
        int res = 0;
        for_each(dist.begin() + 1, dist.end(), [&](int d) {
            res = max(res, d);
        });
        return res == INT_MAX ? -1 : res;
    }
};
```



Visualization of traversing graph using Dijkstra algorithm and distance array. The priority queue itself is not shown there, as well as redundant nodes that we might have in priority queue. Green node - explored/current node, Yellow - node in the priority queue.

Notes:

1. We will have a set to track visited nodes.
2. We will create a distance array to track the distance to each node. Initial node will have 0, others maximum. There is a room for optimization: if the node we got from the pq has larger cost than in our dist[] array, we should not explore it as we already got a better option.
3. We will use min priority_queue to get the node with the minimum distance from the current node.
4. If we are only interested in shortes distance till some END node, we can terminate the search earlier: if (node == dst) return cost;
5. If we already find a better path we shouldn't explore it further: if (dist[node] < stops) continue;

Union-Find:

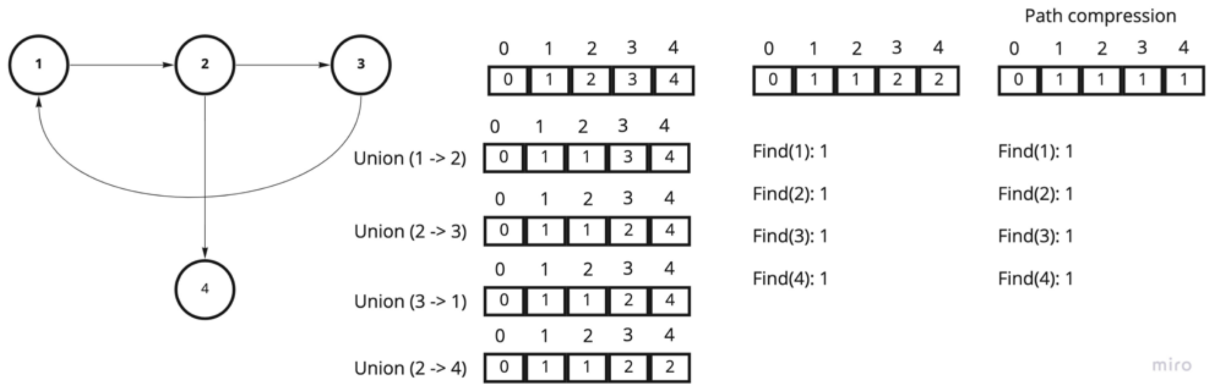
Union–find data structure or disjoint-set data structure or merge–find set, is a data structure that stores a collection of disjoint (non-overlapping) sets. Equivalently, it stores a partition of a set into disjoint subsets. It provides operations for adding new sets, merging sets (replacing them by their union), and finding a representative member of a set. Helps to find the number of connected components, and can help to find MST.

```
class UnionFind {
  public:
  UnionFind(int n) : parent(n) {
    iota(parent.begin(), parent.end(), 0);
  }

  int Find(int x) {
    int temp = x;
    while (temp != parent[temp]) {
      temp = parent[temp];
    }
    // Path compression below
    while (x != temp) {
      int next = parent[x];
      parent[x] = temp;
      x = next;
    }
    return x;
  }

  void Union(int x, int y) {
    int xx = Find(x);
    int yy = Find(y);
    if (xx != yy) {
      parent[xx] = yy;
    }
  }

  private:
     vector<int> parent;
};
```

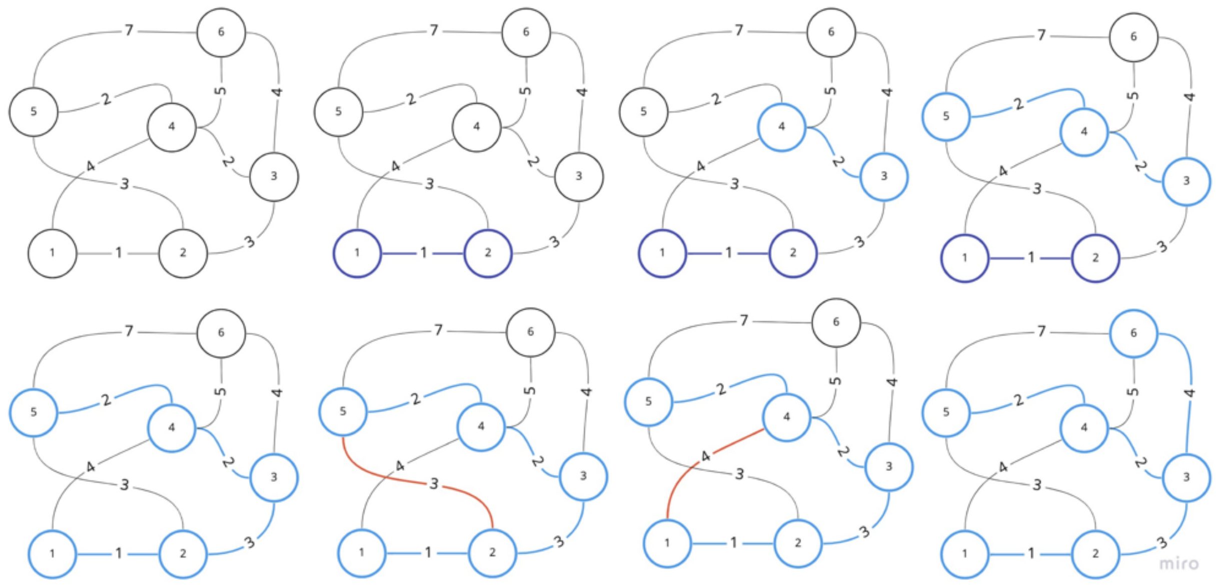


The above picture demonstrates the state of the parent array after multiple Union() calls, follows multiple Find() calls.
Notes:
1. We can use vector to hold the set of nodes or unordered_map<int, int> if we don't know the amount of nodes.
2. If the parent[id] == id, we know that id is the root node.
3. The data structure using two methods Union() - union to nodes/components, and Find() - find the root node.
4. We can do path compression, so after some number of Find() calls it will be O(1) to call Find() again.

Minimum Spanning Tree:
A minimum spanning tree (MST) is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight.
Solution for Connecting Cities With Minimum Cost: https://leetcode.com/problems/connecting-cities-with-minimum-cost/ (https://leetcode.com/problems/connecting-cities-with-minimum-cost/).

```
class UnionFind {
public:
    UnionFind(int n) : parent(n) {
        iota(parent.begin(), parent.end(), 0);
    }
    int Find(int x) {
        int temp = x;
        while (temp != parent[temp]) {
            temp = parent[temp];
        }
        while (x != temp) {
            int next = parent[x];
            parent[x] = temp;
            x = next;
        }
        return temp;
    }
    bool Union(int x, int y) {
        int xx = Find(x);
        int yy = Find(y);
        if (xx == yy) return false;
        parent[xx] = yy;
        return true;
    }

private:
    vector<int> parent;
};
int minimumCost(int n, vector<vector<int>>& connections) {
    sort(connections.begin(), connections.end(), [](const auto& lhs, const auto& rhs){
        return lhs[2] < rhs[2];
    });
    UnionFind uf(n + 1);
    int sum = 0, count = 0;
    for (auto& c : connections) {
        if (uf.Union(c[0], c[1])) {
            count++;
            sum += c[2];
        }
        if (count == n - 1) return sum; // Return earlier once graph is connected.
    }
    return -1;
}
```



Visualization of Kruskal's algorithm: we will try to union nodes if they are not connected.

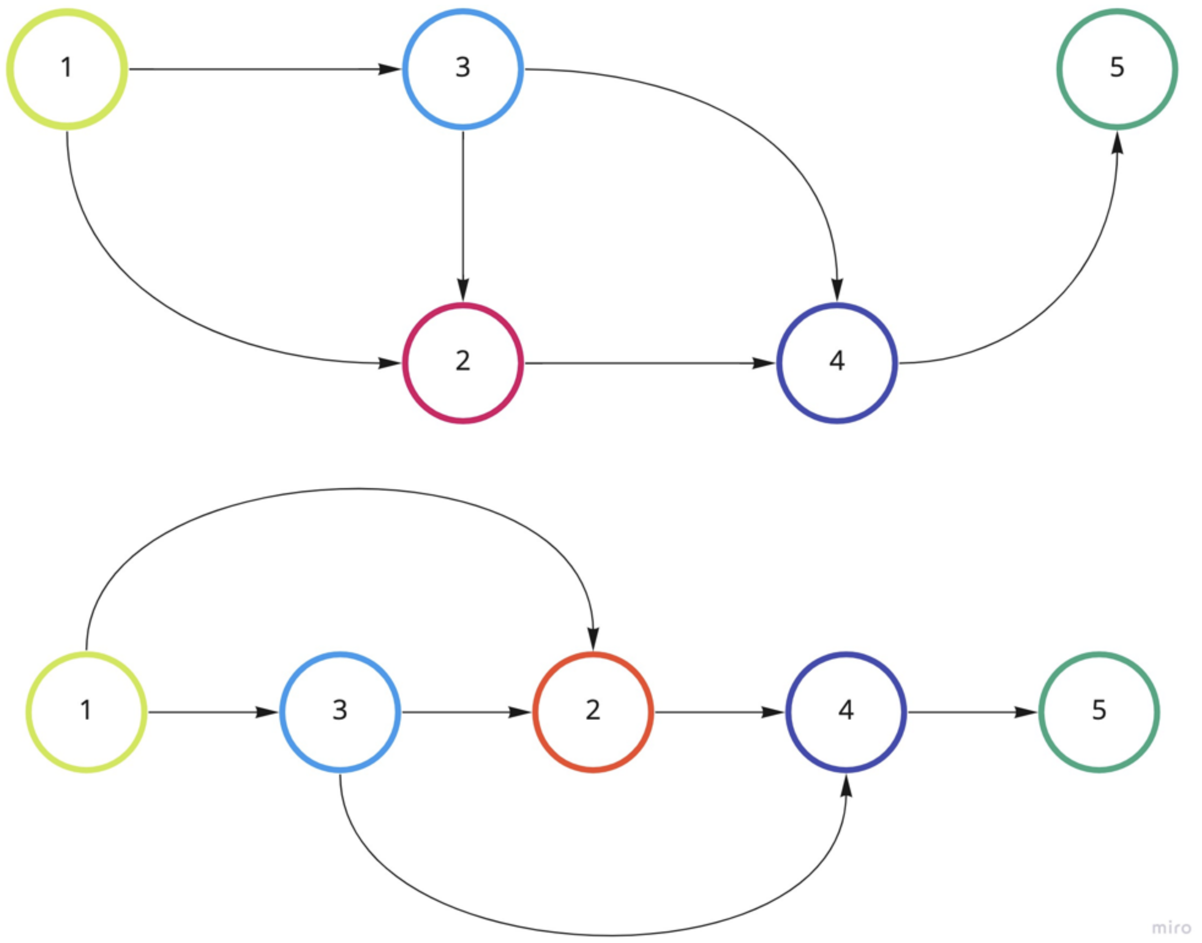
Notes:

1. One of the implementation of MST algorithm use Union Find algorithm (Kruskal's Algorithm).
2. We need to sort elements by the weight before appying the algorithm, or we can use min priority_queue.

Topological sort

Is a linear ordering of its vertices such that for every directed edge uv from vertex u to vertex v, u comes before v in the ordering.

```
// Kahn's Algorithm
vector<vector<int>> adj(numCourses);
vector<int> indegree(numCourses, 0);
for (auto& p : prerequisites) {
    indegree[p[1]]++;
    adj[p[0]].push_back(p[1]);
}
queue<int> q;
for (int i = 0; i < numCourses; i++) {
    if (indegree[i] == 0) q.push(i);
}
int prereq = 0;
while (!q.empty()) {
    int el = q.front();
    q.pop();
    prereq++;
    for (auto& next : adj[el]) {
        if (--indegree[next] == 0) {
            q.push(next);
        }
    }
}
return prereq == numCourses;
```



The above picture demonstrates linear order of the given graph. The vertices can be tasks, and edges can represent some contraints, such as U should be finished before V in (U -> V).

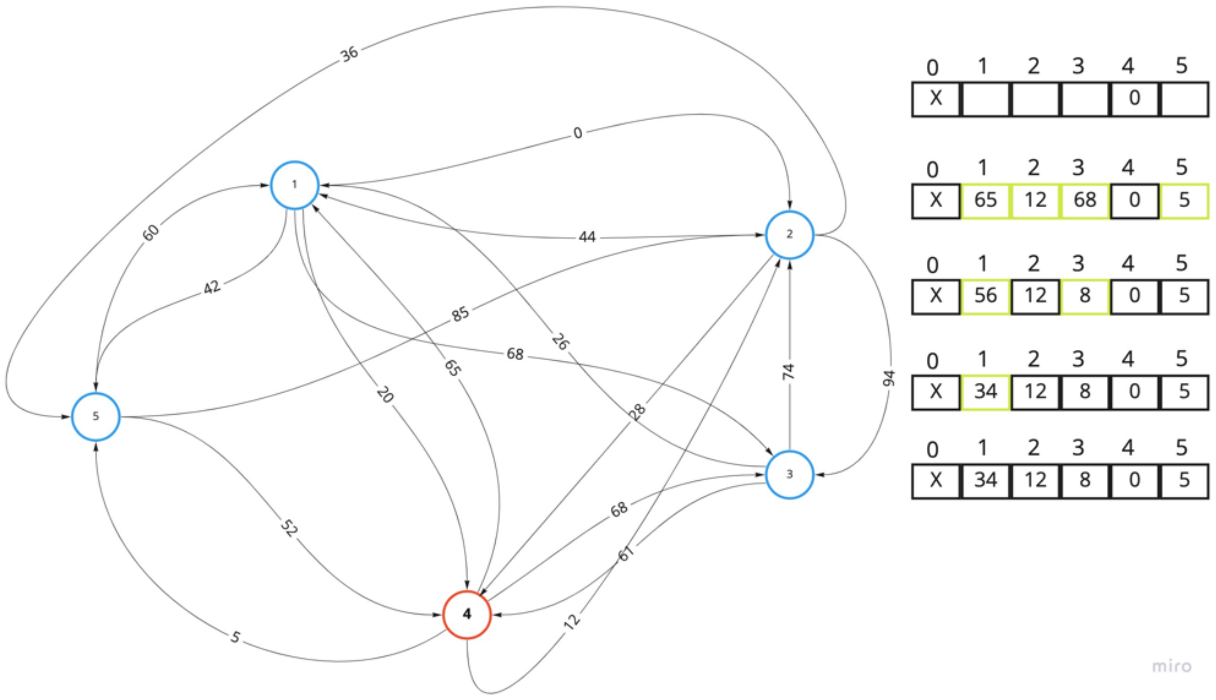
Notes:

1. We will have the indegree array to count, which nods should be visited first.
2. We will have a queue to push the nodes that don't have any dependencies.

Bellman Ford

Is an algorithm that computes shortest paths from a single source vertex to all of the other vertices in a weighted digraph.

```
class Solution {
public:
    int networkDelayTime(vector<vector<int>>& times, int n, int k) {
        vector<int> dist(n + 1, INT_MAX);
        dist[k] = 0;
        for (int i = 1; i <= n; i++) {
            for (auto& t : times) {
                if (dist[t[0]] != INT_MAX && dist[t[1]] > dist[t[0]] + t[2]) {
                    dist[t[1]] = dist[t[0]] + t[2];
                }
            }
        }
        int res = 0;
        for (int i = 1; i <= n; i++) {
            res = max(res, dist[i]);
        }
        return res == INT_MAX ? -1 : res;
    }
};
```



0	1	2	3	4	5
X				0	

0	1	2	3	4	5
X	65	12	68	0	5

0	1	2	3	4	5
X	56	12	8	0	5

0	1	2	3	4	5
X	34	12	8	0	5

0	1	2	3	4	5
X	34	12	8	0	5

The above picture demonstrates how the dist array incrementally updated with better values. If not a single value is updated during iteration - we can stop earlier.

Notes:

1. We will use the array to hold the distance between particular start node and all others.
2. We will try to improve distance n times between all nodes in the graph.

Floyd Warshall

Is an algorithm for finding shortest paths in a directed weighted graph with positive or negative edge weights.

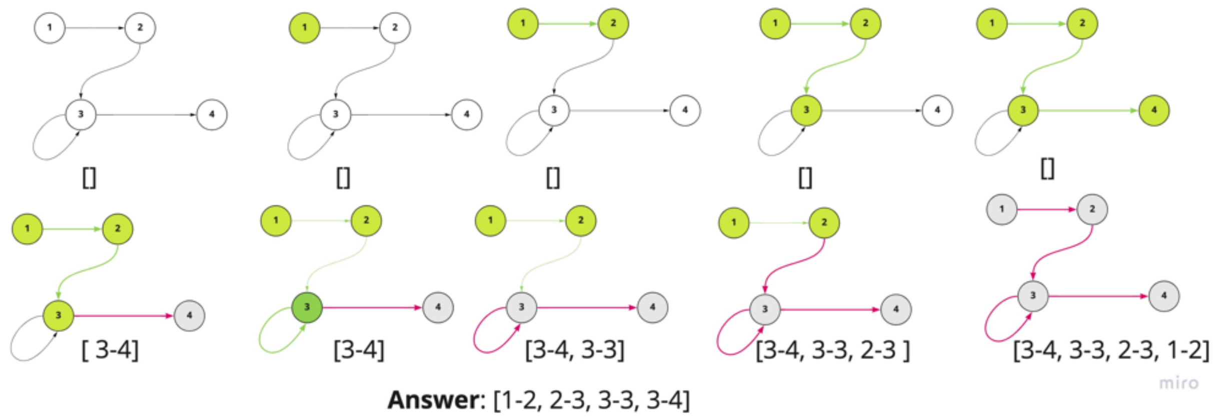
```
class Solution {
public:
    int networkDelayTime(vector<vector<int>>& times, int n, int k) {
        vector<vector<long>> dist(n, vector<long>(n, INT_MAX));
        for (auto& t : times)
            dist[t[0] - 1][t[1] - 1] = t[2];
        for (int i = 0; i < n; i++)
            dist[i][i] = 0;
        for (int k = 0; k < n; k++) {
            for (int i = 0; i < n; i++) {
                for (int j = 0; j < n; j++) {
                    dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
                }
            }
        }
        long res = INT_MIN;
        for (int i = 0; i < n; i++) {
            if (dist[k - 1][i] == INT_MAX) return -1;
            res = max(res, dist[k - 1][i]);
        }
        return (int)res;
    }
};
```

Vizualization for Floyd Warshall is slightly different from Bellman Ford, but the idea stays the same.
Notes:
1. We will use vector<vector> to keep track of distance between nodes i and j.
2. We will have a 3 loops, checks if we can improve the distance between i and j by using k node.

Eulearian Path

Is an algorithm that finds a path that uses every ***edge*** in a graph only once.
The algorithm below is a solution for the "**Reconstruct Itinerary**": https://leetcode.com/problems/reconstruct-itinerary/ (https://leetcode.com/problems/reconstruct-itinerary/)

```
class Solution {
public:
    void dfs(unordered_map<string, multiset<string>>& graph,
             vector<string>& res, string start) {
        while (graph[start].size() > 0) {
            auto next = *graph[start].begin();
            graph[start].erase(graph[start].begin());
            dfs(graph, res, next);
        }
        res.push_back(start);
    }
    vector<string> findItinerary(vector<vector<string>>& tickets) {
        unordered_map<string, multiset<string>> graph;
        for (const auto& t : tickets) graph[t[0]].insert(t[1]);
        vector<string> res;
        dfs(graph, res, "JFK");
        reverse(res.begin(), res.end());
        return res;
    }
};
```



The above picture is the visualization of eulerian path algorithm. You can observe how the result is constructed on the backtracking.

Notes:

1. The algorithm almost identical to the dfs traversal with one main instrumentation: we are building the path on the backtrack of the dfs algorithm: `res.push_back(start);`
2. That is why we should reverse the list at the end of the traversal: `reverse(res.begin(), res.end());`
3. In the above implementation we are using multiset (because of the problem), but the general implementation may use vector<> and additional vector<> to track the outgoing degrees, and use it for two main purposes: as index in the adj list, and to track how many node we not visited yet.

BFS problems

- Flood Fill: https://leetcode.com/problems/flood-fill/ (https://leetcode.com/problems/flood-fill/)
- Number of Islands: https://leetcode.com/problems/number-of-islands/ (https://leetcode.com/problems/number-of-islands/)
- Word Ladder I: https://leetcode.com/problems/word-ladder/ (https://leetcode.com/problems/word-ladder/)
- Word Ladder II: https://leetcode.com/problems/word-ladder-ii/ (https://leetcode.com/problems/word-ladder-ii/)
- Evaluate Division: https://leetcode.com/problems/evaluate-division/ (https://leetcode.com/problems/evaluate-division/)
- Get Watched Videos by Your Friends: https://leetcode.com/problems/get-watched-videos-by-your-friends/ (https://leetcode.com/problems/get-watched-videos-by-your-friends/)
- Cut Off Trees for Golf Event: https://leetcode.com/problems/cut-off-trees-for-golf-event/ (https://leetcode.com/problems/cut-off-trees-for-golf-event/)

DFS problems

- Number of Islands: https://leetcode.com/problems/number-of-islands/ (https://leetcode.com/problems/number-of-islands/)
- Flood Fill: https://leetcode.com/problems/flood-fill/ (https://leetcode.com/problems/flood-fill/)
- Longest Increasing Path in a Matrix: https://leetcode.com/problems/longest-increasing-path-in-a-matrix/ (https://leetcode.com/problems/longest-increasing-path-in-a-matrix/)
- Evaluate Division: https://leetcode.com/problems/evaluate-division/ (https://leetcode.com/problems/evaluate-division/)
- Robot Room Cleaner: https://leetcode.com/problems/robot-room-cleaner/ (https://leetcode.com/problems/robot-room-cleaner/)
- Most Stones Removed with Same Row or Column: https://leetcode.com/problems/most-stones-removed-with-same-row-or-column/ (https://leetcode.com/problems/most-stones-removed-with-same-row-or-column/)
- Reconstruct Itinerary: https://leetcode.com/problems/reconstruct-itinerary/ (https://leetcode.com/problems/reconstruct-itinerary/)
- Tree Diameter: https://leetcode.com/problems/tree-diameter/ (https://leetcode.com/problems/tree-diameter/)
- Accounts Merge: https://leetcode.com/problems/accounts-merge/ (https://leetcode.com/problems/accounts-merge/)

Connected components problems

- Number of Provinces: https://leetcode.com/problems/number-of-provinces/ (https://leetcode.com/problems/number-of-provinces/)
- Number of Connected Components in an Undirected Graph: https://leetcode.com/problems/number-of-connected-components-in-an-undirected-graph/ (https://leetcode.com/problems/number-of-connected-components-in-an-undirected-graph/)
- Number of Operations to Make Network Connected: https://leetcode.com/problems/number-of-operations-to-make-network-connected/ (https://leetcode.com/problems/number-of-operations-to-make-network-connected/)
- Accounts Merge: https://leetcode.com/problems/accounts-merge/ (https://leetcode.com/problems/accounts-merge/)
- Critical Connections in a Network: https://leetcode.com/problems/critical-connections-in-a-network/ (https://leetcode.com/problems/critical-connections-in-a-network/)

Dijkstra's problems

- Path With Maximum Minimum Valued: https://leetcode.com/problems/path-with-maximum-minimum-value/ (https://leetcode.com/problems/path-with-maximum-minimum-value/)
- Network delay time: https://leetcode.com/problems/network-delay-time/ (https://leetcode.com/problems/network-delay-time/)
- Path with Maximum Probability: https://leetcode.com/problems/path-with-maximum-probability/ (https://leetcode.com/problems/path-with-maximum-probability/)
- Path With Minimum Effort: https://leetcode.com/problems/path-with-minimum-effort/ (https://leetcode.com/problems/path-with-minimum-effort/)
- Cheapest Flights Within K Stops: https://leetcode.com/problems/cheapest-flights-within-k-stops/ (https://leetcode.com/problems/cheapest-flights-within-k-stops/)

Union Find problems

- Number of Islands: https://leetcode.com/problems/number-of-islands/ (https://leetcode.com/problems/number-of-islands/)
- Largest Component Size by Common Factor: https://leetcode.com/problems/largest-component-size-by-common-factor/ (https://leetcode.com/problems/largest-component-size-by-common-factor/)
- Most Stones Removed with Same Row or Column: https://leetcode.com/problems/most-stones-removed-with-same-row-or-column/ (https://leetcode.com/problems/most-stones-removed-with-same-row-or-column/)
- Number of Connected Components in an Undirected Graph: https://leetcode.com/problems/number-of-connected-components-in-an-undirected-graph/ (https://leetcode.com/problems/number-of-connected-components-in-an-undirected-graph/)

Minimum Spanning Tree problems

- Connecting Cities With Minimum Cost: https://leetcode.com/problems/connecting-cities-with-minimum-cost/ (https://leetcode.com/problems/connecting-cities-with-minimum-cost/)
- Min Cost to Connect All Points: https://leetcode.com/problems/min-cost-to-connect-all-points/ (https://leetcode.com/problems/min-cost-to-connect-all-points/)

Topological sort problems

- Course Schedule : https://leetcode.com/problems/course-schedule/ (https://leetcode.com/problems/course-schedule/)
- Course Schedule II: https://leetcode.com/problems/course-schedule-ii/ (https://leetcode.com/problems/course-schedule-ii/)
- Sequence Reconstruction: https://leetcode.com/problems/sequence-reconstruction/ (https://leetcode.com/problems/sequence-reconstruction/)
- Alien Dictionary: https://leetcode.com/problems/alien-dictionary/solution/ (https://leetcode.com/problems/alien-dictionary/solution/)

Floyd Warshall problems

- Find the City With the Smallest Number of Neighbors at a Threshold Distance: https://leetcode.com/problems/find-the-city-with-the-smallest-number-of-neighbors-at-a-threshold-distance/ (https://leetcode.com/problems/find-the-city-with-the-smallest-number-of-neighbors-at-a-threshold-distance/)
- Network delay time: https://leetcode.com/problems/network-delay-time/ (https://leetcode.com/problems/network-delay-time/)

Bellman Ford problems

- Network delay time: https://leetcode.com/problems/network-delay-time/ (https://leetcode.com/problems/network-delay-time/)

graph c++ bfs dfs union find topological-sort bellman ford dijkstra algorithm floyd-algorithm

Comments: 14 Best Most Votes Newest to Oldest Oldest to Newest

Type comment here... (Markdown is supported)

Post

(/parthstark) parthstark (/parthstark) ★ 39 January 19, 2022 2:03 PM

Careful, He's a hero!

38 Reply

(/hello_world_22) hello_world_22 (/hello_world_22) ★ 1828 May 8, 2022 7:38 AM

Leetcode should add **Save Post Feature**.. So we can save this types of amazing post into our list.
I raised ticket for it. Please upvote it so we can get this feature soon :: https://leetcode.com/discuss/feedback/1986862/requesting-save-post-feature (https://leetcode.com/discuss/feedback/1986862/requesting-save-post-feature)

29 Show 1 reply Reply

(/milapanwani) milapanwani (/milapanwani) ★ 34 Last Edit: January 19, 2022 2:24 PM

List to clone and use : https://leetcode.com/list/9t2q1i9e (https://leetcode.com/list/9t2q1i9e) (Does not contain the listed problems that are premium)

17 Show 4 replies Reply

(/rhythmjayee) rhythmjayee (/rhythmjayee) ★ 158 July 14, 2021 5:30 PM

Amazing Dude!!🙌🙌🔥

3 Show 1 reply Reply

(/gjha133) gjha133 (/gjha133) ★ 95 August 2, 2022 6:41 PM

Great Post!

2 Reply

(/dr_kruger) dr_kruger (/dr_kruger) ★ 105 June 10, 2022 8:12 AM

Adding Complexities will enhance this post :)
Thanks!

2 Reply Share Report

(/2019ucp1900) 2019ucp1900 (/2019ucp1900) ★ 38 July 12, 2021 8:02 AM

Hi there hats off to your efforts! It'll be great if you can put these problems in a public, cloneable list so that one can use it

▲ 1 ▼ Show 2 replies Reply

(/lkyei) lkyei (/lkyei) ★ 1 July 12, 2021 4:56 AM

how do you memorize algorithms??

▲ 1 ▼ Show 1 reply Reply

(/Aryan3017) Aryan3017 (/Aryan3017) ★ 0 13 hours ago

Dude You are Something else.

▲ 0 ▼ Reply

(/ccchan0109) ccchan0109 (/ccchan0109) ★ 22 Last Edit: September 11, 2022 12:23 PM

Great list! Bro!

▲ 0 ▼ Reply

‹ 1 2 ›