Competitive Programming Algorithms

David del Val

August 11, 2021

Contents

1 Basic range queries **3**
- 1.1 Types of range queries . . . 3
  - 1.1.1 LCA using RMQ . . . 4
- 1.2 Sparse table . . . 5
  - 1.2.1 Example . . . 6
  - 1.2.2 Sparse Table (sort of) with $\mathcal{O}(n)$ memory . . . 8
- 1.3 Fenwick tree (BIT) . . . 12
- 1.4 Segment Trees . . . 14
  - 1.4.1 Lazy propagation . . . 15
  - 1.4.2 Range update and query . . . 16

2 Graphs **19**
- 2.1 Dijkstra's . . . 19
  - 2.1.1 Dijkstra's with cost and distance . . . 21
- 2.2 Bellman Ford's . . . 22
- 2.3 Warshall's . . . 23
- 2.4 DFS . . . 25
  - 2.4.1 Articulation points and bridges . . . 25
  - 2.4.2 Trajan's algorithm for strongly connected components . . . 28
- 2.5 Kosaraju's . . . 31
- 2.6 Kruskal's . . . 36
  - 2.6.1 UFDS . . . 36
  - 2.6.2 Kruskal's . . . 37
- 2.7 Flows and cuts . . . 38
  - 2.7.1 Edmonds-Karp's (Max Flow) . . . 38
- 2.8 Directed Acyclic Graphs (DAG) . . . 40
  - 2.8.1 Topological sort . . . 40
- 2.9 Bipartite Graphs . . . 43
  - 2.9.1 Algorithms to find MCBM . . . 45
  - 2.9.2 Assignment problem . . . 48

3 Mathematics **48**
- 3.1 General calculations . . . 48
  - 3.1.1 Binary exponentiation . . . 48
- 3.2 Modular arithmetic . . . 49

3.2.1 Inverses 49
3.3 Catalan numbers 49
3.4 Permutations 51
3.4.1 Converting one permutation into another 51
3.4.2 Converting one permutation into another using adjacent swaps . . . 52

4 Geometry **54**
4.1 Area of polygon 54

5 Ad-hoc **56**

Chapter 2

Graphs

2.1 Dijkstra's

Shortest path from `orig` node to `dest` (or to every node) in a graph that does not contain negative edges. It chooses the best path greedily in each iteration and, therefore, it only works on graphs without negative weights.

```
20 int dijkstra(int orig, int dest, vector<vector<pii>> &graph) {
21     vi dists(graph.size(), INT_MAX / 10);
22     priority_queue<pii, vector<pii>, greater<pii>> q;
23
24     q.push({0, orig});
25     dists[orig] = 0;
26     while (!q.empty()) {
27         auto a = q.top();
28         q.pop();
29         if (a.se == dest) { // If we already reached the destination
30             return a.fi;
31         }
32         if (dists[a.se] != a.fi) { // If we have found a better option
33             continue;
34         }
35         for (auto b : graph[a.se]) { // Else iterate through the edges
36             if (dists[b.fi] > dists[a.se] + b.se) {
37                 dists[b.fi] = dists[a.se] + b.se;
38                 q.push({dists[b.fi], b.fi});
39             }
40         }
41     }
42     return dists[dest];
43 }
```

Running time: $\mathcal{O}((E + V) \log V)$
(V = vertices, E = edges)

Remarks

- If we ignore the check in line 30, we can return the distances vector, which will contain the shortest distance from `dist` to every other node.
- If we are doing some kind of pruning it is imperative that we prune as many branches as possible in the main loop. That is to say, we should introduce as many `if` statements in line 36 to make sure that we run the `for` loop as few times as possible.

  An example of this approach is problem `UVA-11635`. In that problem, we add a lot of branches to the queue (we may run the `for` loop twice in some nodes) but we prune them in the main loop. Thus the running time is still acceptable.
- However, pruning the branches with a higher cost can be quite complicated when we have to optimize several factors (see section 2.1.1). Therefore, we will omit the check in line `33` if we think it might discard relevant options.

2.1.1 Dijkstra's with cost and distance

In this problem we are going to define two values for each edge, cost and distance:

- *Cost*. This is the number that we want to minimize. It is the equivalent to the usual cost in a normal Dijkstra problem
- *Distance*. We define the distance of a path as the sum of the distances of the edges in it. And the distance of the path from source to destination cannot exceed a given limit (`B`)

This problem can be solved with a slightly modified version of Dijkstra's:

```
20 int B;
21 using pipii = pair<int, pii>;
22 vector<vector<pipii>> adyList; //destination, cost, distance
23 ll dijkstra(int orig, int dest) {
24     vi distances(adyList.size(), iinf);
25     priority_queue<pipii, vector<pipii>, greater<pipii>> q; //cost,node,dist
26 
27     distances[orig] = 0;
28     q.push({orig, {0, 0}});
29 
30     while (!q.empty()) {
31         auto a = q.top();
32         q.pop();
33 
34         if (a.se.fi == dest) {
35             return a.fi;
36         }
37         distances[a.se.fi] = a.se.se;
38 
39         for (auto b : adyList[a.se.fi]) {
40             if ((a.se.se + b.se.se < distances[b.fi]) ) {
41                 q.push({a.fi + b.se.fi, {b.fi, a.se.se + b.se.se}});
42             }
43         }
44     }
45     return -1;
46 }
```

Running time: $\mathcal{O}\big(B \cdot (V + E \log E)\big)$
(V = vertices, E = edges, B = max_distance)

In this algorithm, we will process each node at most `B` times. Furthermore, if we process a node with a cost c and a distance d, we know that this is the best possible cost for that distance d since all the nodes in the queue have a higher cost already. Therefore, we can close the node for that particular distance value (line 37).

An example of this algorithm is `SWERC-19_20-A`

2.2 Bellman Ford's

Shortest parth from `orig` to every other node. It is slower than Dijkstra but it works on graphs with negative weights.

This algorithm works by trying to relax every edge $V-1$ times. If there are no negative cycles, after $V-1$ iterations, we must have found the minimum distance to every node. Therefore if after these iterations, we run another iteration and the distance to a node decreases, we must have a negative cycle.

```
20 vi bellmanFord(int orig, vector<pair<pii, int>> edges, int n) {
21     vi dist(n, INT_MAX / 10);
22     dist[orig] = 0;
23     for (int i = 0; i < n - 1; ++i) {
24         for (auto e : edges) {
25             dist[e.fi.se] = min(dist[e.fi.se], dist[e.fi.fi] + e.se);
26         }
27     }
28     // dist[i] contains the shortest path from 0 to i
29 
30     //We can now check for a negative cycle
31     bool negativeCycle = false;
32     for (auto e : edges) {
33         if (dist[e.fi.se] > min(dist[e.fi.se], dist[e.fi.fi] + e.se)) {
34             negativeCycle = true;
35         }
36     }
37     return dist;
38 }
```

Running time: $\mathcal{O}(VE)$
(V = vertices, E = edges)

Remarks

- If we keep track of the distance that decrease when we check for a negative cycle, we will get at least one node of every negative cycle present in the graph.

  We can use this, for instance, to check if we can reach a node with a cost smaller than a given bound. If it is connected to a node in a negative cycle, it's distance will be as small as we want it to be (by looping in the cycle).

  This can be seen at play in `UVA-10557`

- If we modify slightly the main loop, iteration i will be the result of considering paths of at most $i+1$ edges:

```
for (int i = 0; i < n - 1; ++i) {
        for (auto e : edges) {
            dists2[e.fi.se] = min(dists2[e.fi.se], dists[e.fi.fi] + e.se);
        }
    dists = dists2;
}
```

This can be seen at play in `UVA-11280`

2.3 Warshall's

Warshall's algorithm solves the APSP (All pairs shortest path) using DP. Each iteration of the outer loop tries to add a node (the k-th node in particular) to the path between all pairs of nodes. We can think of the comparison as:

Is the path from i to j shorter if we first move from i to k and, then, from k to j?

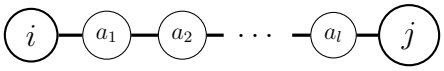
It is important to note that we must use an adjacency matrix in this implementation.

```
void warshall(int n) {
    for (int k = 0; k < n; ++k) {
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j) {
                adyMat[i][j] = min(adyMat[i][j], adyMat[i][k] + adyMat[k][j]);
            }
        }
    }
}
```

Running time: $\mathcal{O}(V^3)$
(V = vertices, E = edges)

Obtaining the path

To get the explicit path, we will store the last vertex in the path that the algorithm found to go from i to j. That is to say, after a given iteration the optimal path from i to j is: Then



we should store that the last node in the path that goes from i to j is a_j. The code to do so is the following:

```cpp
ll lastInPath[lim][lim];

void warshall(int n) {
    for (int k = 0; k < n; ++k) {
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j) {
                if (adyList[i][j] > adyList[i][k] + adyList[k][j]) {
                    adyList[i][j] = adyList[i][k] + adyList[k][j];
                    lastInPath[i][j] = lastInPath[k][j];
                }
            }
        }
    }
}

// Give an empty vector
void getPath(int i, int j, vector<int> &path) {
    if (i != j) {
        getPath(i, lastInPath[i][j], path);
        path.push_back(j);
    }
}
```

2.4 DFS

2.4.1 Articulation points and bridges

These algorightms can be used in undirected graphs, and we will use the following definitions:

- **Articulation point**. A node whose removal would increase the number of connected components of the graph. That is to say that it "splits" a connected component.
- **Bridge**. An edge whose removal increases the number of connected components in the graph.

We will use a modified version of DFS to solve this problem. We mainly introduce two new properties for every node:

- `num`. Time at which the node was first explored by DFS
- `low`. Earliest node that can be found in the DFS spanning tree that starts from this node

When we visit a node, for every edge, there are two options:

- **Tree edge**. This edge point to a node that has not been discovered yet. As such we explore it (calling `dfs`) and, we update the value of `low` for the current node.

  After the update, we can now process the child since we will not visit it again and it's DFS tree has been fully explored
- **Back edge**. This edge points to a node that has already been visited. Therefore, it will have a relatively low `num` and we use it to update the `low` value of the current node

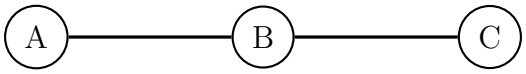
```cpp
vector<vi> adyList; // Graph
vi num, low;        // num and low for DFS
int cnt;            // Counter for DFS
int root, rchild;   // Root and number of (DFS) children
vi artic;           // Contains the articulation points at the end
set<pii> bridges;   // Contains the bridges at the end

void dfs(int nparent, int nnode) {
    num[nnode] = low[nnode] = cnt++;
    rchild += (nparent == root);

    for (auto a : adyList[nnode]) {
        if (num[a] == -1) { // Tree edge
            dfs(nnode, a);
            low[nnode] = min(low[nnode], low[a]);

            if (low[a] >= num[nnode]) {
                artic[nnode] = true;
            }

            if (low[a] > num[nnode]) {
                bridges.insert((nnode < a) ? mp(nnode, a) : mp(a, nnode));
            }
        } else if (a != nparent) { // Back edge
            low[nnode] = min(low[nnode], num[a]);
        }
    }
}
void findArticulations(int n) {
    cnt = 0;
    low = num = vi(n, -1);
    artic = vi(n, 0);
    bridges.clear();

    for (int i = 0; i < n; ++i) {
        if (num[i] != -1) {
            continue;
        }
        root = i;
        rchild = 0;
        dfs(-1, i);
        artic[root] = rchild > 1; //Special case
    }
}
```

Running time: $\mathcal{O}(V + E)$

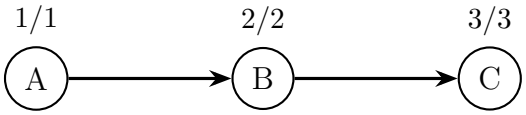
(V = vertices, E = edges)

Explanation

The first graph that we will consider is the following: After applying DFS on A we get the



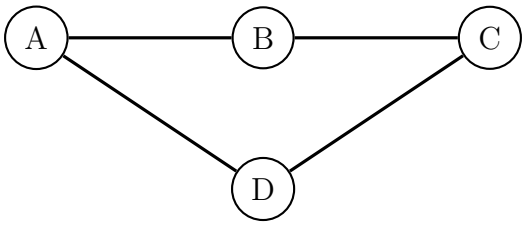
following DFS spanning tree. Above every node, we have included `num / low`.



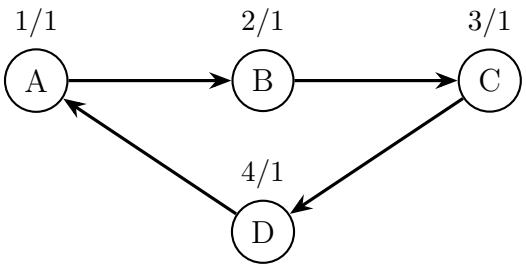
This is a rather simple graph and it's only articulation point is B. This is because it has a child whose `low` value is greater than or equal to B's `num` value. Therefore, there is no connection from that child (namely C) to a node explored before B that does not go through B.

However, we can already see that the root has to be treated as a separate case. The root will be an articulation point iff it has more than one child in its DFS tree. It is important to note that the children of the DFS tree need not be the same as the children of the root in the initial graph.

Let's look at another example



As before, we show the DFS spanning tree:



Now we have no articulation point since there is no node that has a child with a `low` greater than or equal than the parent's `num`. This difference is caused by the fact that now there is an edge that makes C accessible through a path that does not involve traversing B.

In this case, we can see how the root has only one child in the spanning tree but two in the initial graph. This reflects the fact that those numbers need not coincide.

Finally, the condition for a bridge is: `low[child]>num[parent]`. This is equivalent to stating that the child has no other way of reaching either the parent or a node that was explored before the parent

2.4.2 Trajan's algorithm for strongly connected components

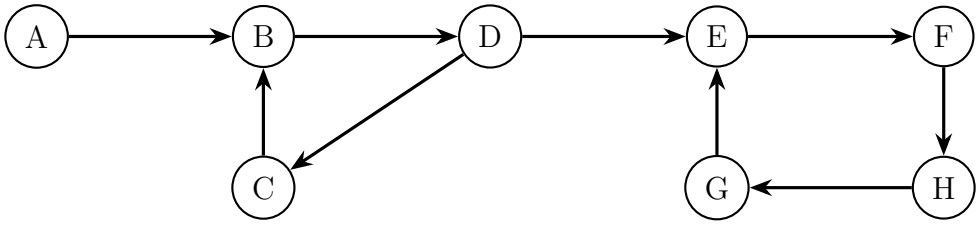
In a directed graph, we say that a subset of vertices comprises a strongly connected component if every vertex is reachable from every other vertex in this subset.

We will now present an algorithm that divides the graph into strongly connected parts that are as large as possible. It will use some of the same concepts as in the previous section.

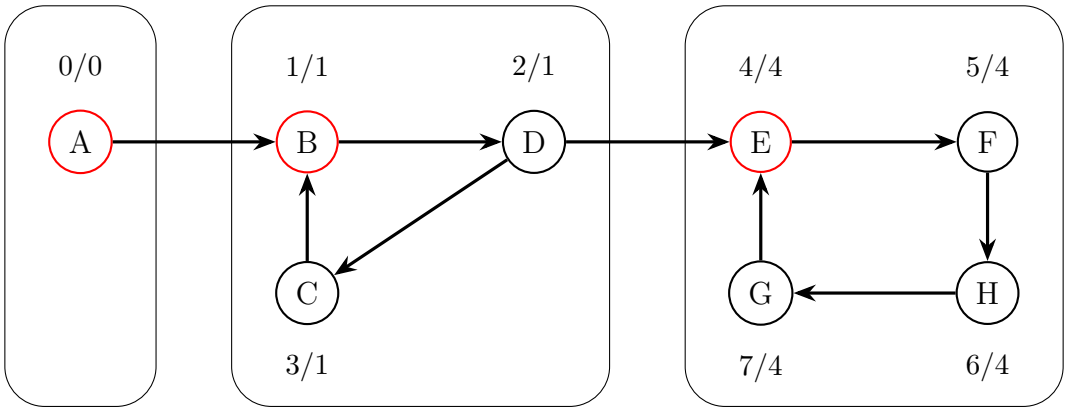
Explanation

The main idea is the following: if when we are done with a node, we have not been able to reach a node that was further back, this one is the root of a SCC.

Let's explain this reasoning with an example:



And, after running DFS, we would get the following spanning tree with three SCCs.



Let's look at the moment in which we close node F. As we can see the `low` value is lower than the `num`. Therefore, there is a connection from F to another node that was visited before and it will be a part of the SCC "generated" by that node.

However, when we close E, we can see that the `low` and the `num` are equal. As a result, there is no way to get to a node that has a number lower than E's through the spanning tree of E. Therefore, there is no way to have a bigger SCC that contains E.

To get the nodes that are part of the SCC, we need to get all the nodes in the stack before the one that "generates" the SCC (g). This is because the stack only contains the

nodes that belong to the spanning tree of g and do not belong to any other SCC. Therefore, they must belong to the one generated by g.

Furthermore, all those nodes will have the same `low` value since the `low` any node with a higher value must have been processed before. Therefore, g is accessible from all those nodes and, clearly, all those nodes are accessible from g. Thus, they fulfill the definition of an SCC.

```cpp
vector<vi> adyList; // Graph
vi low, num;        // num and low for DF
int cnt;            // Counter for DFS
stack<int> st;
vi inStack;      // Position in the stack + 1
vector<vi> sccs; // Contains the SCCs at the end

void dfs(int u) {
    low[u] = num[u] = cnt++;
    st.push(u);
    inStack[u] = st.size();
    for (auto a : adyList[u]) {
        if (num[a] == -1) {
            dfs(a);
        }
        if (inStack[a]) {
            low[u] = min(low[u], low[a]);
        }
    }
    if (low[u] == num[u]) { // Root of a SCC
        vi v;
        // Add all the nodes till u (included)
        int lim = inStack[u];
        while (st.size() && st.size() >= lim) {
            v.push_back(st.top());
            inStack[st.top()] = false;
            st.pop();
        }
        sccs.push_back(v);
    }
}
void TarjanSCC(int n) {
    low = num = vi(n, -1);
    inStack = vi(n, 0);
    st = stack<int>();
    cnt = 0;
    sccs = vector<vi>();

    for (int i = 0; i < n; ++i) {
        if (num[i] == -1) {
            dfs(i);
        }
    }
}
```

Running time: $\mathcal{O}(V + E)$

(V = vertices, E = edges)

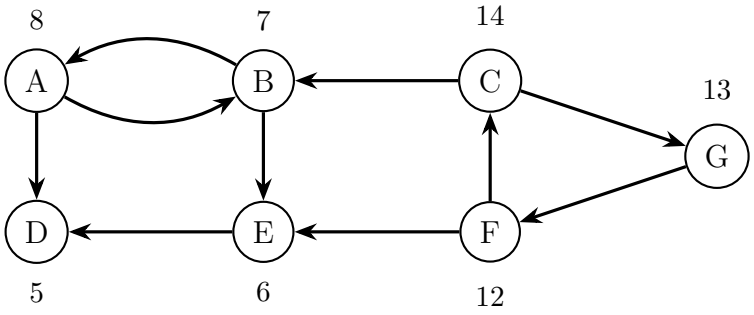
2.5 Kosaraju's

Kosaraju's algorithm is a slightly simpler method for finding the SCC of a graph. On the other hand, it is also somewhat slower than Trajan's approach since it will require running DFS on the graph twice. The algorithm follow this procedure:

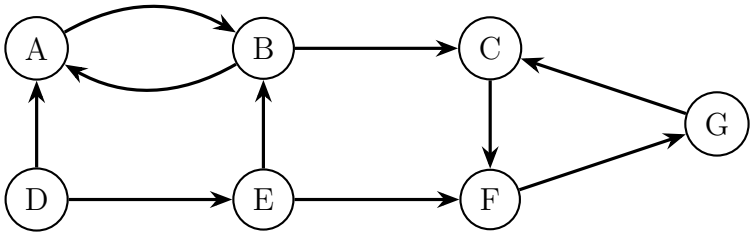
1. Run DFS on the graph and store the vertices in postorder in a list called `postorder`.
2. Reverse the graph
3. Loop through the nodes in `postorder` starting from the one that was closed the last. For each of them check if it has already been added to a SCC (visited). If it hasn't, run DFS from that node and create a SCC with all the nodes that this DFS visits and hadn't been visited before.

Example

Let's look at an example graph where we have run DFS starting first in node A and, since it didn't explore the entire graph, we also run it from node C. We have included the time at which each node was closed:



Now we reverse the graph and we get the following:

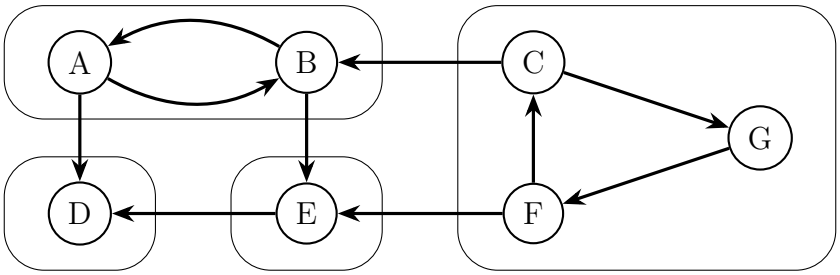


Finally, we can loop through the nodes in postorder, starting with the node that was close the last:

$$\text{postorder} = \{C, G, F, A, B, E, D\}$$

1. We run DFS on node C and we get the SCC:
$$s_1 = \{C, F, G\}$$
2. We skip nodes G and F in the list since they are already in a SCC and we run DFS on node A, getting the SCC:
$$s_2 = \{A, B\}$$
3. We skip node B since it is already in a SCC and we run DFS on node E, getting the SCC:
$$s_3 = \{E\}$$
4. We finally run DFS on node D, and we get the last SCC:
$$s_4 = \{D\}$$

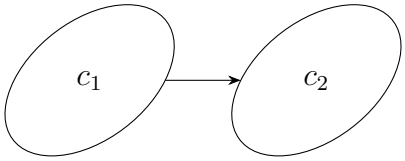
Now, returning to the initial graph, we have found the following SCCs:



Explanation

Let's look at why this algorithm works.

Firstly, we have to take into account that the SCCs of a graph G are preserved when we reverse all the edges and get G^t. The only relevant issue is the order in which we process the SCCs. Let's assume we have a graph that has two SCC's: We have three options:



- *They are not connected.* In this case, it does not matter whether we explore one or the other first since DFS will not "leak" from one of them to the other
- *They are connected only in one direction.* This is the case that the figure shows and it is the most important one. If we explore c_2 before exploring c_1, DFS will first explore the entirety of c_2 and then, when it is exploring c_1 it will not "leak" to c_2 because those nodes are already marked as visited.
- *They are connected in both directions.* This can never happen since, if they were to be connected bidirectionally, they would form a single SCC, not two.

Therefore, the correctness of this algorithm simply depends on exploring the SCCs in the right order.

Let's demonstrate the following claim:

In the previous setting, the maximum closing time of the nodes in c_1 will be greater than the maximum closing time of the nodes in c_2

We just have to distinguish two cases:

- If we started exploring c_2 before c_1, there is no way to get to c_1 from c_2. Therefore, we will start exploring c_1 when we have already closed c_2, which means that all nodes in c_2 will be closed before the nodes in c_1 are even "opened".
- If we started exploring c_1 before c_2, at some point, DFS will leak to c_2 and it will explore it entirely before returning to c_1. Therefore, the closing time of every node in c_2 will be lower than the closing time of the node where DFS was started in c_1.

This completes the proof of that statement. Let's now apply it to check the correctness of this algorithm using induction.

- *Base case:*

  After reversing the graph, we start with the node that was closed the last (a). Let's call it's SCC s_1. Let's assume that there is an edge from s_1 to another SCC (s_α) in G^t, which would make DFS leak to that SCC. However, if that was the case, there would be an edge from s_α to s_1 in G and, therefore, we can apply the previous claim.

  In that scenario, the maximum closing time of s_α would be greater than the maximum closing time of s_1. This is a contradiction because we have stated that $a \in s_1$ is the node that was closed the last.

  Therefore, there cannot be any edge from s_1 to another SCC.

- *Inductive step:*

  Let's now assume that we have already processed n SCCs. We now choose a node b, which is the node with the highest closing time such that $b \notin s_i,\ i = 1, \dots n$.

  We will call the SCC of this node s_{n+1}. As before, we want to prove that s_{n+1} is not connected to any s_β that hasn't been explored yet in G^t.

  We will assume that there is an edge from s_{n+1} to a $s_\beta \notin \{s_1 \dots s_n\}$. Therefore, in G, there is an edge from s_β to s_{n+1}, which implies that the maximum closing time of s_β is higher than the maximum closing time of s_{n+1}.

  Let's define $x_\beta :=$(the node with the maximum closing time of s_β). We just stated that the closing time of x_β is greater than the closing time of b (there cannot be a node with a greater closing time in s_{n+1}).

  However, this is a contradiction. Therefore, s_{n+1} cannot be connected to an SCC that has not been explored and, thus, DFS will not leak.

□

```cpp
vector<vi> adyList;  // Graph
vector<int> visited; // Visited for DFS
vector<vi> sccs;     // Contains the SCCs at the end

void dfs(int nnode, vector<int> &v, vector<vi> &adyList) {
    if (visited[nnode]) {
        return;
    }
    visited[nnode] = true;
    for (auto a : adyList[nnode]) {
        dfs(a, v, adyList);
    }
    v.push_back(nnode);
}

void Kosaraju(int n) {
    visited = vi(n, 0);
    stack<int> s = stack<int>();
    sccs = vector<vi>();

    vector<int> postorder;
    for (int i = 0; i < n; ++i) {
        dfs(i, postorder, adyList);
    }
    reverse(all(postorder));

    vector<vi> rAdyList = vector<vi>(n, vi());
    for (int i = 0; i < n; ++i) {
        for (auto v : adyList[i]) {
            rAdyList[v].push_back(i);
        }
    }

    visited = vi(n, 0);
    vi data;
    for (auto a : postorder) {
        if (!visited[a]) {
            data = vi();
            dfs(a, data, rAdyList);
            if (!data.empty())
                sccs.pb(data);
        }
    }
}
```

Running time: $\mathcal{O}(V + E)$

(V = vertices, E = edges)

Remarks

- The SCCs of any graph form a DAG

2.6 Kruskal's

Kruskal's algorithm finds a minimum spanning forest of the given graph. To do so, it uses UFDS to keep track of which nodes are already connected.

2.6.1 UFDS

UFDS (Union-find data structure) is a data structure that stores a partition of the vertices into sets such that all vertices in the same set are connected. Each node u will have two properties:

- *Parent*. It is the representative of the partition that contains u.
- *Rank*. It is an upper bound of the height of the "tree" that starts on u.

```
20 const int lim = 200000;
21 int parent[lim];
22 int rankk[lim];
23
24 void initialize(int n) {
25     for (int i = 0; i < n; ++i) {
26         rankk[i] = 0;
27         parent[i] = i;
28     }
29 }
30 int find(int x) {
31     if (parent[x] == x)
32         return x;
33     else
34         return parent[x] = find(parent[x]);
35 }
36
37 void Union(int a, int b) {
38     int pa = find(a);
39     int pb = find(b);
40     if (pa == pb) {
41         return;
42     }
43     if (rankk[pa] > rankk[pb]) {
44         parent[pb] = pa;
45     } else if (rankk[pa] < rankk[pb]) {
46         parent[pa] = pb;
47     } else {
48         parent[pa] = pb;
49         rankk[pb]++;
50     }
51 }
```

Running time: $\mathcal{O}(E \log V)$
(V = vertices, E = edges)

Explanation

There are two main optimizations that are applied in this implementation:

- *Union by rank*. When we join two partitions, we have to choose a node to represent the new partition. In order to choose between the two parents, we use their rank.

  Our goal is to minimize the height of the trees that start at every node. Therefore, we pick the node with the highest rank as the parent. This choice ensures that the rank of both parents will not increase.

  However, if both parents have the same rank, we can choose either of them.

- *Path shortening*. The `find` function updates the value of the parent of each node so that it does not point to it's "immediate" parent but to the highest possible ancestor. This difference increases the efficiency of subsequent executions of the `find` routine.

2.6.2 Kruskal's

Using UFDS, the implementation of Kruskal is trivial. We just edges that connect vertices that are not connected already until all vertices are connected.

```
20 ll kruskal(vector<pair<pii, int>> &edgeList) {
21     int cost = 0;
22
23     auto cmp = [](pair<pii, int> a, pair<pii, int> b) { return a.se < b.se; };
24     sort(all(edgeList), cmp);
25
26     for (auto a : edgeList) {
27         if (find(a.fi.fi) ≠ find(a.fi.se)) {
28             //Not connected already → connect them
29             Union(a.fi.fi, a.fi.se);
30             cost += a.se;
31         }
32     }
33     return cost;
34 }
```

Running time: $\mathcal{O}(\max\{E, V\} \log(\max\{E, V\}))$
(V = vertices, E = edges)

2.7 Flows and cuts

In this section we will consider graphs as networks of pipes and the "weights" of the edges will be their capacity.

2.7.1 Edmonds-Karp's (Max Flow)

The first problem that we have to consider is how to find the maximum flow from one source node to a sink node. Edmonds-Karp's is an implementation of Ford-Fulkerson's that solves this problem.

This algorithm has some major caveats: it requires the use of an adjacency matrix and it is not as fast as other options.

To improve its running time, this implementation uses both an adjacency matrix and adjacency list. However, we have to be careful when creating the adjacency list since for every forward edge, we need a backward edge that will start with a capacity of 0 but its capacity may increase.

Therefore, the code for adding edges would be the following:

```
// Add a directed edge
void addedgeUni(int orig, int dest, ll flow) {
    adjList[orig].pb(dest);
    adjMat[orig][dest] = flow;
    adjList[dest].pb(orig);   //Add edge for residual flow
}
// Add a bidirectional edge
void addEdgeBi(int orig, int dest, ll flow) {
    adjList[orig].pb(dest);
    adjList[dest].pb(orig);
    adjMat[orig][dest] = flow;
    adjMat[dest][orig] = flow;
}
```

```cpp
vector<vector<ll>> adjList;
vector<vector<ll>> adjMat;

void initialize(int n) {
    adjList = decltype(adjList)(n);
    adjMat = decltype(adjMat)(n, vector<ll>(n, 0));
}

map<int, int> p;
bool bfs(int source, int sink) {
    queue<int> q;
    vi visited(adjList.size(), 0);
    q.push(source);
    visited[source] = 1;
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        if (u == sink)
            return true;
        for (auto v : adjList[u]) {
            if (adjMat[u][v] > 0 && !visited[v]) {
                visited[v] = true;
                q.push(v);
                p[v] = u;
            }
        }
    }
    return false;
}
int max_flow(int source, int sink) {
    ll max_flow = 0;
    while (bfs(source, sink)) {
        ll flow = inf;
        for (int v = sink; v != source; v = p[v]) {
            flow = min(flow, adjMat[p[v]][v]);
        }
        for (int v = sink; v != source; v = p[v]) {
            adjMat[p[v]][v] -= flow; // Decrease capacity forward edge
            adjMat[v][p[v]] += flow; // Increase capacity backward edge
        }
        max_flow += flow;
    }
    return max_flow;
}
```

Running time: $\mathcal{O}(VE^2)$

(V = vertices, E = edges)

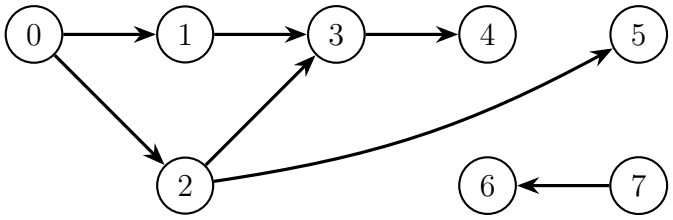
Remarks

- If we have multiple sources $s_1, \ldots, s_n$ and multiple sinks $t_1, \ldots t_n$, we can use the same algorithm. We just have to create a source s that connects to s_i with edges of infinite capacity and a sink t such that s_i is connected to t with edges of infinite capacity.
- If we have vertex capacities, we can split the vertex v into two vertices: v_{in} and v_{out}, connected with an edge that has the vertex capacity as a capacity. Then we connect the incoming edges to v_{in} and the out-coming edges to v_{out}

2.8 Directed Acyclic Graphs (DAG)

2.8.1 Topological sort

A topological sort of a directed graph is a linear ordering of its vertices such that for every directed edge u $\rightarrow$ v, u comes before v in the ordering. A topological sorting is possible if and only if the graph has no directed cycles, that is, it has to be a DAG. As an example:



0	1	2	3	4	5	6	7
0	1	2	3	4	5	7	6

Regarding the algorithm to obtain the topological sorting, there are two alternatives:

- A modified version of DFS
- Kahn's algorithm

Using DFS

The key observation is:

> *When DFS closes a node, all nodes that are accessed from that node are already closed*

Therefore if we order the vertices regarding the DFS closing time, we can be sure that if there exists an edge u $\rightarrow$ v, v will be placed before u since it must have been closed before v was closed. Now we should realize that to get the topological sorting, we only need to reverse this ordering.

```
vi topoSort;
vi visited;
vector<vi> adyList;
void dfs(int node) {
    if (visited[node]) {
        return;
    }
    visited[node] = true;
    for (auto a : adyList[node]) {
        dfs(a);
    }
    topoSort.push_back(node);
}
void topologicalSort() {
    int n = adyList.size();

    visited = vi(n, 0);
    topoSort.clear();

    for (int i = 0; i < n; ++i) {
        if (!visited[i]) {
            dfs(i);
        }
    }
    reverse(all(topoSort));
}
```

Running time: $\mathcal{O}(E + V)$
(V = vertices, E = edges)

Kahn's algorithm

Kahn's algorithm is a very simple algorithm. It simply keeps track of the in-degree of every vertex and only adds to the topological sort list vertices that do not have any incident edges.

Furthermore, this algorithm can be easily adapted to store all possible topological orderings by using a recursive function and running through all possibilities at each step

```
vi topoSort;
vi visited;
vector<vi> adyList;
void dfs(int node) {
    if (visited[node]) {
        return;
    }
    visited[node] = true;
    for (auto a : adyList[node]) {
        dfs(a);
    }
    topoSort.push_back(node);
}
void topologicalSort() {
    int n = adyList.size();

    visited = vi(n, 0);
    topoSort.clear();

    for (int i = 0; i < n; ++i) {
        if (!visited[i]) {
            dfs(i);
        }
    }
    reverse(all(topoSort));
}
```

Running time: $\mathcal{O}(E + V)$
(V = vertices, E = edges)

2.9 Bipartite Graphs

A graph is bipartite if its vertices can be split into two different so that no two nodes in the same set are connected by an edge.

Definitions

It is important to note that in this section a path (unless specified otherwise) will be a simple path (not containing repeated vertices or edges).

- *Matching*. Set of pairwise non-adjacent edges in a graph. That is to say, no more than one edge from the set should be incident to any vertex of the graph.
- *Saturated* (a vertex with respect to some matching). A saturated vertex is a vertex that is an endpoint of an edge contained in the matching.
- *Alternating path* (with respect to some matching). Path in which the edges alternately belong / do not belong to the matching.
- *Augmenting path* (with respect to some matching M). Alternating path whose initial and final vertices are unsaturated (they do not belong to the matching). That is to say, if we represent the edges already included in M as I and the new ones as U, it follows this pattern:

  $$U - I - U - \cdots - U - I - U$$

  We can also notice that there will be $\left\lfloor \frac{|P|}{2} \right\rfloor$ edges already included and $\left\lceil \frac{|P|}{2} \right\rceil$ new edges.
- *Independent set*. Set of vertices in which no two vertices represent an edge in the original graph.
- *Vertex coverage*. Set of vertices C that fulfills the following property: Every vertex of G is incident on, at least, one vertex of C.

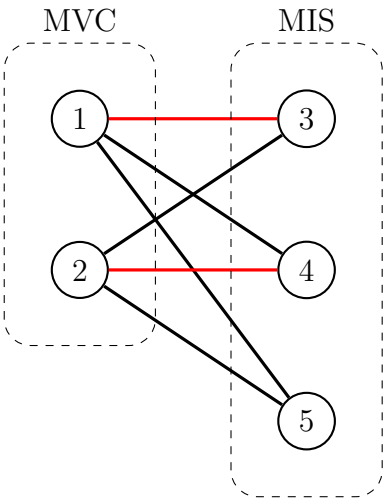
From these definitions, we can derive the following features in a bipartite graph:

- *Maximum cardinality bipartite matching (MCBM)*. Matching that contains the maximum number of edges.
- *Maximum independent set (MIS)*. Independent set that fulfills that if any other vertex was to be added to it, there would be an edge between two of the vertices of the set.
- *Minimum vertex coverage (MVC)*. Vertex coverage with the minimum number of vertices.

Furthermore, it can be proved that even though these "sets" may not be unique, their cardinality is. Additionally, they are closely related among them:

$$|\text{MIS}| = |\text{V}| - |\text{MCBM}| \qquad |\text{MCBM}| = |\text{MVC}|$$

Example



In this example we have a bipartite graph where nodes 1 and 2 belong to the first set of vertices and nodes 3, 4 and 5 to the second set.

- Since there are only two vertices in the first set, the MCBM has to be at most two. We have found a matching of size two (in red), so that must be the solution.
- The MIS has to be one of the two sets of vertices since all vertices from one set are connected to all vertices from the second set. We pick the maximum one.
- Finally, the MVC has to be the smallest set for the same reasons.

Finally, we can also notice how the MCBM is not unique even though its cardinality is unique.

Berge's lemma

A matching M is maximum $\iff$ there is no augmenting path relative to the matching M

Proof

We can prove it by contradiction:

$\implies$ Let there be an augmenting path P relative to M. This path will necessarily be of odd length (it has one more edge not in M than in M). We can now create a new matching $M' = M \oplus P = (M \setminus P) \cup (P \setminus M)$.

Now we have to see that this matching is valid. In order for it not to be valid, there must be a vertex such that two edges in M' are adjacent to. Furthermore, one of those edges must belong to $P \setminus M$ while the other one belongs to M. However, the second one must then also belong to P (P alternates edges) and will not be in $M \setminus P$. Therefore, it cannot be in M' and we reached a contradiction

Finally, we can see that there are $|M| + 1$ edges in M' since:

- By subtracting P from M we delete $\left\lfloor \frac{|P|}{2} \right\rfloor$ edges
- By adding $P \setminus M$, we add $\left\lceil \frac{|P|}{2} \right\rceil$ edges

Therefore, M could not be a maximum matching

$\Longleftarrow$ Let there be a matching M' of greater cardinality than M. We can consider the subgraph $Q = M \oplus M'$. Any vertex in this subgraph has a maximum degree of 2, which means that all connected components in it are either:

- An isolated vertex
- a (simple)path whose edges are alternate from M and M'
- A cycle of even length whose edges are alternate from M and M'

However, since there are more edges from M' than from M, there must be at least one component with more edges from M' than from M. Since such path must be alternating, it will have initial and final vertices unsaturated by M, making it an augmenting path for M.

□

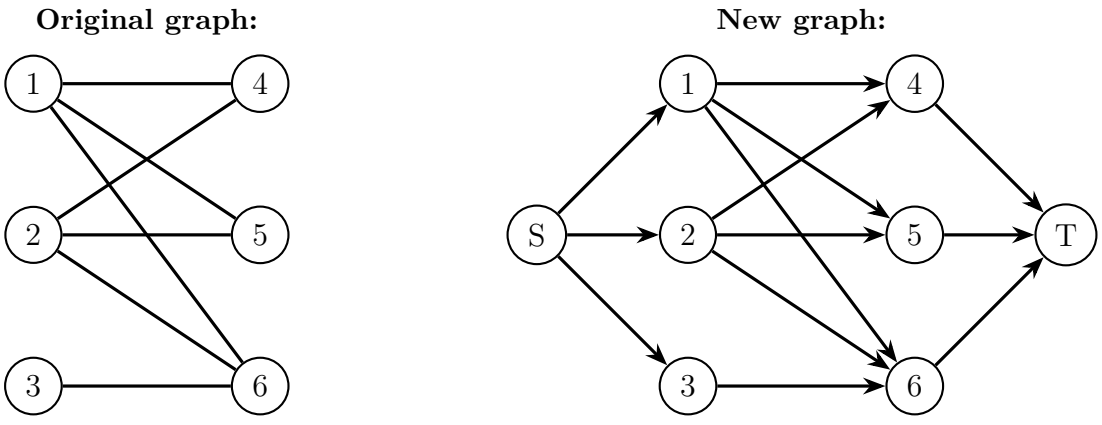
2.9.1 Algorithms to find MCBM

Reduce to Max-Flow

The bipartite matching problem can be easily reduced to a max-flow problem. We only have to:

- Make the edges between the two components directed from the first component to the second (all edges must go in the same direction)
- Connect the source to all nodes in the first component.
- Connect the nodes in the second component to the sink.
- Set the capacities of all edges to 1.

For instance:



After this process, we can run the normal max-flow algorithm and the maximum flow between the source and the sink will be the result. This will have a time complexity of $\boldsymbol{\mathcal{O}(VE^2)}$ using Edmonds-Karp's or $\boldsymbol{\mathcal{O}(V^2E)}$ using Dinic's

Kuhn's Algorithm

On a high level, the algorithm implements the following pseudocode (where M is the matching):

```
M ← ∅
while (able to find augmenting path P){
        M ← P ⊕ M
}
```

This is a direct application of Berge's lemma; however, if we were to implement it naïvely, we could end up with a time complexity of $\mathcal{O}(V^2E)$ since there will be at most V augmenting paths and to find each of them, we would need to check for paths that start on each node and then run DFS to find the actual path. To make it faster, it can be implemented using the following pseudocode (where n_1 is the size of the first set in the graph):

```
M ← ∅
for (i = 0 ... n1){
        P ← augmenting path that starts on i
        if(P ≠ ∅){
                M ← P ⊕ M
        }
}
```

To understand why this implementation should work, it is necessarily to make a few observation:

- The augmenting path found in each iteration starts on a vertex in the first set, then traverses some number of vertices that are already saturated and it ends on a vertex of the second set that is not saturated.
- When we update M with the new P, no vertices that were saturated before will be unsaturated now. Since all saturated vertices in P are connected by two adjacent edges, one of them will still be present in P $\oplus$ M since it could not be present on M before.
- If there is no augmenting path that starts from i in the current iteration, it will be impossible to find one in a some future iteration. This is because no vertex can go from being saturated to being unsaturated in this process.

With all of this in mind, we can obtain a c++ implementation that runs on $\mathcal{O}(EV)$. Furthermore, if we make sure that the first set of the bipartite graph is the one with less elements, the runtime will be faster since the iterations of the outer loop will decrease.

As we can see, the search for an augmenting path from a node will try all options to pair that node to a node in the second set. For each one of them, if they are already matched to some node in the first set, it will try to create an augmenting path from there. The algorithm will stop when it finds an unsaturated vertex or runs out of vertex to check.

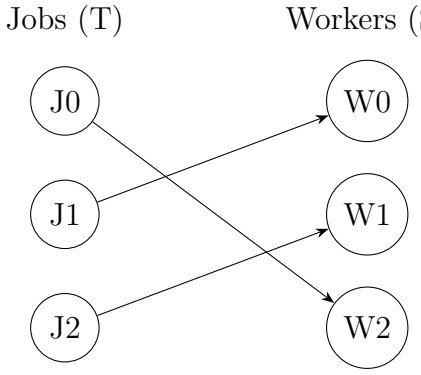
```cpp
// List of vertices in the 2nd component connected to
// the i-th vertex in the first component
vector<vi> adyList;

// Index of the vertex in 1st component connected to the
// i-th vertex in the second component
vector<int> match;
vector<bool> used;
bool augmentPath(int current) {
    if (used[current]) {
        return false;
    }
    used[current] = 1;
    for (int to : adyList[current]) {
        if (match[to] == -1 || augmentPath(match[to])) {
            match[to] = current;
            return true;
        }
    }
    return false;
}
int calculateMCBM(int n1, int n2) {
    match = vi(n2, -1);
    int res = 0;
    for (int i = 0; i < n1; ++i) {
        used.assign(n1, false);
        res += augmentPath(i);
    }
    return res;
}
```

Running time: $\mathcal{O}(EV)$
(V = vertices, E = edges)

2.9.2 Assignment problem

There are n workers that should complete n jobs. Each assignment (worker, job) has a given cost and each worker can only perform one job (and vice versa). The goal is to minimize the total cost of performing all jobs.

This problem can be modeled as a bipartite graph where the jobs and workers are represented by vertices. In such graph, we can encode the assignment of a job to a worker by adding an edge between those two vertices. For example:



Costs			
	W0	W1	W2
J0	1	3	5
J1	1	10	20
J2	10	3	10

Definitions

- A complete matching M is a set of edges such that all jobs are paired with exactly one worker.
- A perfect matching M is a complete matching such that the added cost of all the edges is maximum.
- The set of all nodes that represent jobs is T.
- The set of all nodes that represent workers is S.
- The cost of adding an edge between two vertices v, w ($v \in S \wedge w \in T$ or $v \in T \wedge w \in S$) is represented as $c(v, w)$.
- A function $y : (S \cup T) \to \mathbb{R}$ is a potential if $y(v) + y(w) \leq c(v, w)$
- The value of a potential y is: $V_y = \sum_{v \in S \cup T} y(v)$
- An edge $v - w$ is tight with respect to a potential y if $c(v, w) = y(v) + y(w)$

Assumptions

- All costs in the cost matrix are positive. This is not required in the final implementation but simplifies the proof. Moreover, it is trivial to make the costs positive by adding a large positive value to all of them.
- There is the same number of both workers and jobs. This simplifies the proof but is not required. If we have less jobs than workers, we can always add more jobs with cost 0 to all workers.

Analysis of the algorithm

The goal of the algorithm is to find a perfect matching. In order to do so, we can observe that:

The cost C of a perfect matching M is greater or equal to the value of any potential y

We can easily prove this statement. Since M pairs each job with one worker, we can enumerate identify the worker assigned to the job v as $w(v)$:

$$V_y = \sum_{u \in S} y(u) + \sum_{v \in T} y(v) = \sum_{v \in T} y(v) + y(w(v)) \leq \sum_{v \in T} c(v, w(v)) = C$$

From that claim, we can deduce that if the cost of a complete matching M equals the value of a potential, M is perfect Furthermore, if we analyze the previous inequality, we can observe that in a perfect matching, there must be a potential y such that all edges belonging to the matching M are tight with respect to y. Thus:

A complete matching M formed by tight edges with respect to some potential y is a prefect matching

Using this insight we can code a constructive algorithm that produces complete matchings that only consist of tight edges. On a high level, this algorithm performs the following operations:

```
for each job i in [0,n):
        run O(n^2) Dijkstra to calculate the "closest"
        not matched worker

        update the current matching and potential to obtain a
        matching that matches all [0, i] jobs using tight edges
```

The implementation of this algorithm required the following arrays:

- `matchedJob[i]`: Job matched to the i-th worker
- `jobPotential[i]`: Value of the potential evaluated in the i-th job. We will see in the implementation that this is always a negative value (it is only decreased during the algorithm)
- `workerPotential[i]`: Value of the potential evaluated in the i-th worker. We will see in the implementation that this is always a positive value (it is only increased during the algorithm)
- For each iteration of the outer loop:
  - `dist[i]`: Minimum distance from the visited
  - `from[i]`: Worker from which we found the current distance to the i-th worker.
  - `seen[i]`: Whether the i-th worker has been visited

Pseudocode of the implementation

```
int hungarian(costMatrix):
        matchedJob[*] ← -1
        jobPotential[*] ← 0
        workerPotential[*] ← 0

        for maxJob in [0,n):  # outer loop
                currentWorker = helperNode
                matchedJob[helperNode] = maxJob
                dist[*] ← ∞
                from[*] ← -1
                seen[*] ← 0

                while (currentWorker not matched):  # Dijkstra loop
                        seen[currentWorker] = 1
                        currentJob = matchedJob[currentWorker]
                        delta = ∞

                        for worker in [0,n):
                                if seen[worker]: continue
                                candidateDistance = costMatrix[currentJob][worker]
                                candidateDistance -= jobPotential[currentJob]
                                candidateDistance -= workerPotential[worker]
                                if candidateDistance < dist[worker]:
                                        dist[worker] = candidateDistance
                                        from[worker] = currentWorker
                                if dist[worker] < delta:
                                        delta = dist[worker]
                                        nextWorker = worker

                        for worker in [0, n):
                                if seen[worker]:
                                        jobPotential[matchedJob[worker]] += delta
                                        workerPotential[worker] -= delta
                                else:
                                        dist[worker] -= delta

                        currentWorker = nextWorker

                while currentWorker ≠ helperNode:
                        worker = from[currentWorker]
                        matched[currentWorker] = matched[worker]
                        currentWorker = worker

        return workerPotential[helperNode]
```

Outer loop

The outer loop has two invariants that we have to maintain. At the beginning of the k-th iteration, the following properties must hold true:

1. We have a matching consisting of k tight edges that is complete with respect to the first $k-1$ jobs.
2. The potential function given by arrays `jobPotential` and `workerPotential` fulfills the potential property.

In each iteration of the outer loop, we have to alter the already formed matching to add exactly one more job and one more worker. The job that we have to add is given by the `maxJob` loop counter.

However, deciding which worker to add is more challenging since the optimal solution might involve changing some of the previous assignment. This is the problem that this modified Dijkstra aims to solve.

The cost function for Dijkstra's

During the search process, the cost of an edge that goes from v to w will be considered to be $c(v,w) - y(v) - y(w)$. This is the increment in the cost of the matching if we add that edge to it. If we analyze it case by case:

- When there are no edges in the matching that touch either vertex, the potencial of the vertices will be 0 and the cost will not be altered.
- When the job was already touched by an edge, the cost of this edge must be decreased since if we add this edge, we will eliminate the previous paring for the job, which already had a cost. This is the effect of subtracting the potential of the job (which is a positive value).
- When the worker was already picked, the cost of this edge must be increased since adding this edge will imply deleting the previous pairing and adding an additional one that pairs the job that no longer is paired with another worker. Since the potential value of a worker is negative, subtracting it will increase the cost of the edge.

Using this cost definition enables the algorithm to search for the minimal increase to the cost of the matching.

Modified $\mathcal{O}(n^2)$ Dijkstra's

To start the Dijkstra, we use a helper node that will be matched to the new job. This helper node has two main uses: it simplifies the code of the algorithm by making the first iteration more similar to the following ones and it's potential value stores the cost of the entire matching.

Now the algorithm can start. In each iteration, the algorithm will have a `currentJob` and a `currentWorker` value, where the `currentJob` value is the job matched to the current worker. It should be noted that for all intensive purposes, either node can be considered the current node since they are joint by an edge of "cost" 0.

After the innermost for loop, the `dist` array will be updated to contain the lowest cost from the explored jobs to each worker. Additionally, the `from` array might also change to reflect the changes in the distance vector. Finally, the `nextWorker` and the `delta` variables will be updated to the "closest" worker and the cost of reaching that worker.

Now we have to update the potential values. Since it takes at least `delta` to reach any other worker, we can increase the potential of all visited jobs by `delta`. However, all those nodes are connected to a worker with a tight edge since they are already paired. Therefore, in order to maintain the potential property, we have to decrease the potential at the other end of those tight edges.

Finally, since we are updating the potentials, we also have to change the distances accordingly. Since the potentials of all visited jobs have been increased by `delta`, the minimum distances to all workers are decreased by the same amount. This update is the price we have to pay to be able to update the "cost" of the edges of the graph in the middle of the search.

To finish the iteration we have to check if the current worker is already matched to some job. If it is, we move to the worker and conduct at least another iteration. Otherwise, we can stop the algorithm since we have obtained the worker that can be matched with a job while increasing the cost of the matching by the minimum amount.

If we look back at the invariants of the outer loop, we can see that the second invariant already holds true after this process. However, to make the first one true, we still have to update the `matchedJob` array. In order to do so, we can simply backtrack from the last worker using the `from` array.

C++ implementation

This implementation follows the previous pseudocode but there are some aspects that should be kept in mind:

- The input matrix must have the following format:

  `matrix[i][j]` $\equiv$ Cost of having the i-th job done by the j-th worker

- If the objective is the maximum cost, the entries of the cost matrix should be multiplied by -1.
- The algorithm can handle cases where there are more workers than jobs to get done

```cpp
// The rows are jobs, the columns are workers
pair<ll, vl> hungarian(vector<vl> &matrix) {
    int n = matrix.size(), m = matrix[0].size();
    vl jobP(n), workerP(m + 1), matched(m + 1, -1);

    vl dist(m + 1, inf);
    vi from(m + 1, -1), seen(m + 1, 0);

    for (int i = 0; i < n; ++i) {
        int cWorker = m;
        matched[cWorker] = i;
        std::fill(all(dist), inf);
        std::fill(all(from), -1);
        std::fill(all(seen), false);

        while (matched[cWorker] != -1) {
            seen[cWorker] = true;
            int i0 = matched[cWorker], nextWorker = -1;
            ll delta = inf;

            for (int worker = 0; worker < m; ++worker) {
                if (seen[worker])
                    continue;
                ll candidateDistance = matrix[i0][worker];
                candidateDistance += -jobP[i0] - workerP[worker];

                if (candidateDistance < dist[worker]) {
                    dist[worker] = candidateDistance;
                    from[worker] = cWorker;
                }
                if (dist[worker] < delta) {
                    delta = dist[worker];
                    nextWorker = worker;
                }
            }
            for (int j = 0; j <= m; ++j) {
                if (seen[j]) {
                    jobP[matched[j]] += delta;
                    workerP[j] -= delta;
                } else {
                    dist[j] -= delta;
                }
            }
            cWorker = nextWorker;
        }
        while (cWorker != m) {
            int prevWorker = from[cWorker];
```

```
            matched[cWorker] = matched[prevWorker];
            cWorker = prevWorker;
        }
    }
    ll ans = -workerP[m];
    vl rowMatchesWith(n);
    for (int j = 0; j < m; ++j) {
        if (matched[j] != -1) {
            rowMatchesWith[matched[j]] = j;
        }
    }
    return {ans, std::move(rowMatchesWith)};
}
```

Remarks

- Sometimes, it might be useful to add workers to represent actions that may occur sequentially.