Hungarian Algorithm

Ananya Appan - IMT2017004
Rathin Bhargava - IMT2017522

April 30, 2021

Contents

1 Algorithm Overview

1.1 Definitions

- A graph $G(V, E)$ is **bipartite** if V can be partitioned into two sets X and Y such that, without loss of generality, any edge $e(x, y) \in E$ goes from some vertex $x \in X$ to some vertex $y \in Y$.
- A **matching** $M \subseteq E$ is a set of edges such that $\forall v \in V$, at most one edge $e \in M$ is incident on v. A matching of maximum size is a maximum matching.
- In a weighted bipartite graph, a **maximum weighted matching** is matching whose sum of edge weights is maximum.
- An **alternating path** is a path which alternates between edges in a matching M, and edges not in the matching.
- An **augmenting path** is an alternating path which starts and ends in edges not in the matching M. By swapping the edges of an augmenting path, one can increase the size of the matching by 1.
- For a graph $G(V, E)$, the **neighborhood** of a vertex $v \in V$ is the set $N(v) = \{u \in V : e(u, v) \in E\}$.

1.2 Problem Statement

Given a bipartite graph $G(V, E)$ the Hungarian algorithm solves the assignment problem, i.e, to find the **maximum weighted matching** in G.

This problem is called the assignment problem because it helps solve a similar problem of assigning jobs to workers - Given n workers and m jobs such that any worker can take up only one job and that an job can be taken up by only one worker, find an optimal assignment of workers to jobs with minimum cost. One can think of the n workers as a partition X in a bipartite graph, and the m jobs as the other partition Y, and the cost to be paid to a worker for a specific job to be the weight of the edge between them.

We will be looking at maximum weighted matching in complete bipartite graphs. For any graph $G(V, E)$ which isn't complete bipartite, the same problem can be solved in a complete bipartite graph $G'(V, E')$ where $\forall e(x, y) \in E'$ such that $e(x, y) \notin E$, $w'(e(x, y)) = 0$.

Note that one can find the **minimum weighted matching** using the same algorithm by negating the edge weights.

1.3 Algorithm

1.3.1 The intuition behind it all

The main intuition behind the algorithm is as follows. Suppose a worker decides to reduce his price in order to land more jobs. He decreases the his price for all jobs which are performed by him by the same amount x. Then, solving the assignment problem in this scenario will yield the same answer as solving the assignment problem earlier. This is because the price has been reduced uniformly across *all* jobs. Here, we decrease the weight of all edges which are outgoing from the node corresponding to that worker. In order to save time, we assign a label to each vertex which indicates the amount reduced. Formally, we have the following definitions.

- A **vertex labelling** is a function $l : V \to \mathcal{R}$.
- A **feasible vertex labelling** is one such that $\forall x \in X, y \in Y,\ l(x) + l(y) \geq w(x, y)$.
- An **Equality Graph** with respect to a vertex labelling l , denoted as G_l, is a graph with edges $e(x, y)$ such that $l(x) + l(y) = w(x, y)$.

It is easy to see that a perfect matching in G_l will be a maximum weighted matching in G. Hence, throughout our algorithm, we keep track of edges in G_l, and try to find a perfect matching.

1.3.2 Alternating Trees

In order to find a perfect matching, we want to match all vertices in both X and Y. Hence, if there is any unmatched vertex $x \in X$, we want to match it. To do this, we construct an alternating tree rooted at x. This is a tree where all paths from the x are alternating paths with respect to a matching M. In order to extend the matching and match x, all we need to do is find an augmenting path in this alternating tree. Since x is not matched yet, it is assured that after flipping the edges of the augmenting path, M continues to remain a matching. Note that these paths have to contain edges only in G_l, since we want to find a perfect matching in G_l.

1.3.3 Sets S and T

Suppose we have an alternating tree rooted at $x \in X$. $S \subseteq X$ is the set of vertices which belong to the alternating tree from X, and $T \subseteq Y$ the set of vertices which belong to the alternating tree from Y.

1.3.4 Building the Alternating Tree

In order to find all vertices in the alternating tree, we check if there is any vertex $y \in Y$ such that $e(x', y) \in E(G_l)$ where y is matched. If we find such a y, then we can add x' to S and y to T and build the alternating tree. This can continue until we find an augmenting path, or no more vertices can be added.

1.3.5 Improving the Labelling

Suppose we are in a situation where, while building an alternating tree rooted at x, we are neither able to find an augmenting path nor able to add more edges. In this case, the neighborhood of S becomes equal to T. At this time, we have to add more edges to our equality graph. In order to do this, we improve the labelling. We do this as follows.

- Find $\forall x \in S, y \in Y \setminus T, min(l(x)) + l(y) - w(x, y))$. Say this is α_l.
- Suppose the new labelling is l'. Define this as follows :

$$l'(x) = \begin{cases} l(x) - \alpha_l & x \in S \\ l(x) + \alpha_l & x \in T \\ l(x) & otherwise \end{cases}$$

By doing this, we include the edge for which $l(x)) + l(y) - w(x, y)$ was minimum. Because we consider edges from nodes in S to nodes outside of T, this will ensure that a node outside of T is added to the equality graph and that our alternating tree can be expanded. Since we add α_l to vertices in T while subtracting α_l from vertices in S, edges which are a part of the alternating tree will continue to remain in the new equality graph.

1.3.6 An Initial Feasible Labelling

An easy way to take an initial feasible labelling is to define l as follows by taking the weight of the maximum weighted edge incident to any vertex in X as $l(x)$, and by taking $l(y)$ to be 0.

$$l(x) = \begin{cases} \forall y \in N(x)\ max(w(x, y)) & x \in X \\ 0 & y \in Y \end{cases}$$

1.3.7 An Initial Feasible Matching

The initial feasible matching can be taken as the maximum weighted outgoing edge $e(x, y)$ from each vertex $x \in X$, if y is not matched already.

1.3.8 Putting Everything Together

1. Start with an initial feasible vertex labelling l and an initial feasible matching M.
2. If M is perfect return. Else, there is an unmatched vertex $x \in X$. Initialize S to $\{x\}$ and T to ϕ.
3. In order to build alternating tree, check if $N(S) \neq T$. If $N(S) = T$, then improve labelling.

4. Now, $N(S) \neq T$. If there is an unmatched $y \in T$, then find augmenting path. Otherwise, build alternating tree and go to step (3).

5. Go to step (2).

1.4 Time Complexity

Let's go through all the major steps of the algorithm and see what the time complexity of each step is. Here $n = |V|$, the total number of vertices in the bipartite graph.

In each phase of the matching, the cardinality of the matching $|M|$ increases by one. Since it can go to at most $O(n)$, we need to show that the complexity of each phase is $O(n^2)$.

Note: For each phase, we are keeping track of $slack_y = min_{x \in S}(l(x) + l(y)) - w(x, y)$.

- Initialize all slacks - $O(n)$
- If $N(S) = T$, then we have to update the labels for all the nodes. Calculating $\alpha_l = min_{y \in T}(slack_y)$ takes $O(n)$ time. We have $O(n)$ nodes to update which takes $O(n)$ time as well.
- We have to update all slacks when a vertex gets added to the set S.
  - A vertex can never be removed from S in a particular phase. Hence, this happens at most n times.
  - It takes $O(n)$ to update all slacks once a vertex is moved.

  Hence, overall this step takes $O(n^2)$ in each phase.
- Finally, when $N(S) \neq T$, we can find an augmenting path. We can do so by using BFS which takes $O(V + E)$ time, where in this case takes $O(n^2)$ time (since we are dealing with a bipartite graph). Note that this only happens once per phase, since after which, the cardinality of our matching increases by one and we move on to the next phase

Hence, we can see that overall, each phase takes $O(n^2)$. Since there are $O(n)$ phases, we get an overall time complexity of $O(n^3)$.

In the case when we have a $K_{n,m}$ instead of a $K_{n.n}$, the algorithm takes $O(min(n, m) * nm)$ time.

2 Applications

2.1 Chess Match Fixing

2.1.1 Problem Statement

This problem statement has been taken from here.
We have two teams of n players who want to play against each other. You are from the first team and you know the ratings of each and every player from both teams.

The scenario is simple, if player A has a higher rating than player B, player A will win the match and their team gets 2 points. If the ratings are equal, both opponents get 1 point each.

As the manager of team A, you have the special privilege of choosing who can play against whom. Formally, for every player p_A from your team you can choose one player p_B from team B. Every player can only be chosen once.

How do you maximise your score?

2.1.2 Solution

With a little bit of analysis, we can reduce this problem to the assignment problem. Since every player plays exactly one match, we just need to assign the players in a way to maximise our score.

Let's make the cost matrix. Given player A_i from team A and player B_j from team B, we determine the cost matrix. Let the rating of a player p be denoted as $r(p)$.

$$mat[i][j] = \begin{cases} 2 & r(A_i) > r(B_j) \\ 1 & r(A_i) = r(B_j) \\ 0 & else \end{cases}$$

It's easy to see how this correctly models our situation. Finding the maximum matching using the Hungarian Algorithm gives us our optimal answer.

2.2 Wine Bottle Couriers

2.2.1 Problem Statement

This problem has been adapted from here.

We have n wine bottles scattered around in a city. They need to be delivered to a restaurant R. There are m courier workers who can deliver the bottles to the restaurant.

The cost of travelling between two points A and B is the **Manhattan Distance** between them. Given the coordinates of the wine bottles, couriers and the restaurant, we need to find the minimum cost to deliver the wine bottles to the restaurant.

Note: At one time, one courier worker can only carry one wine bottle. So if the courier has to go for the second bottle, they have to travel from the restaurant to the bottle and back.

2.2.2 Solution

If we start with the assumption that one courier can only transport one wine bottle overall, and that $n = m$, then this reduces to our standard assignment problem where the cost $mat[i][j]$ is defined as the Manhattan distance from the j^{th} courier to the i^th bottle added to the distance from the bottle to the restaurant.

However, since a courier can transport multiple wine bottles in multiple trips, we might have to do some extra work.

At this point, an observation helps us. If a courier is going for the second bottle, the cost to get the second bottle is simply twice the distance from the bottle to the restaurant (to go to the bottle and come back). Hence, the identity of the courier doesn't really matter just as long there's someone who can do it.

To make our life simpler, we could add n bogus couriers, all who are located at the restaurant, and run the algorithm. The original m couriers will be henceforth termed as "legitimate". Now, every wine bottle can be taken on a courier's first trip (by the courier themselves) or on a second trip of some courier (by the bogus restaurant courier). In this way, we would also be able to capture the multiple trips issue we had earlier. However, there's a catch.

What if all the bottles are only taken by bogus couriers? We need at least one legitimate courier to reach the restaurant before carrying out the round trips for the wine bottles. There's a simple fix really. That means that at least one wine bottle will be taken on it's first trip. So we don't need n bogus couriers, $n-1$ will do.

By making this change, we are ensuring that at least one bottle will get to a legitimate courier. All the others have options to choose from the bogus courier or the legitimate one.

Formally, $mat[i][j], j < m$ is defined as the Manhattan distance from the j^{th} courier to the i^{th} bottle added to the distance from the bottle to the restaurant (just how it was earlier). And for our bogus carriers, the costs are defined like $mat[i][j], m \leq j \leq n+m-1$ where the cost is defined as twice the distance from the restaurant to the i^{th} wine bottle.

Note, that since we're dealing with costs, and the Hungarian Algorithm gives us a maximum weighted matching, we negate all the costs before running the algorithm.

3 Demo

3.1 Input Output Format

Input

The first line of the input contains the integers n and m - the size of partitions X and Y.
This followed by n lines containing m integers - the weights of the edges in the graph.

Output

The output consists of one line - an integer representing the weight of the maximum weighted matching in the graph.

Example

Input
3 3

1 4 5
5 7 6
5 8 8
Output
18

3.2 Testing

We tested our code in two ways. We took an algorithm of the net (and named in Internet-Code.cpp). Then we generated random testcases and checked if both codes gave the same output.

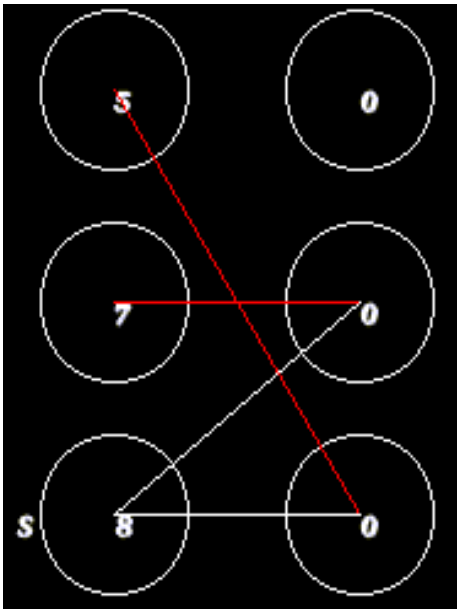
Then on the problems we've described above(applications), we submitted the code for those problems and have gotten an "Accepted" verdict. The online judge runs our algorithm on their testcases and generally try to cover all tests.

Hence, we are confident our code is 100% correct.

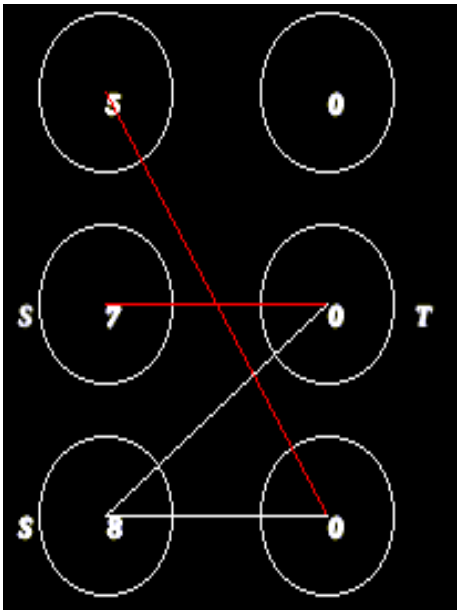
3.3 Visualization

To explain the algorithm better, we implemented visualization using the graphics library in c++. We will explain the algorithm using the example given while explaining the input output format.

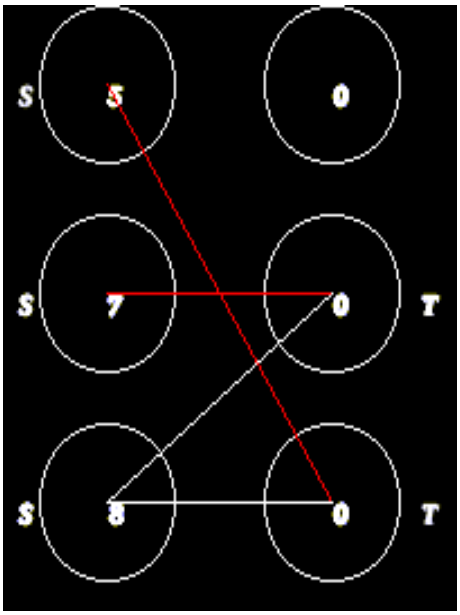
1. We start with an initial feasible vertex labelling and initial matching as described in the algorithm. Based on this, we also build the equality graph and initialize S to a vertex in X which has not been matched yet.



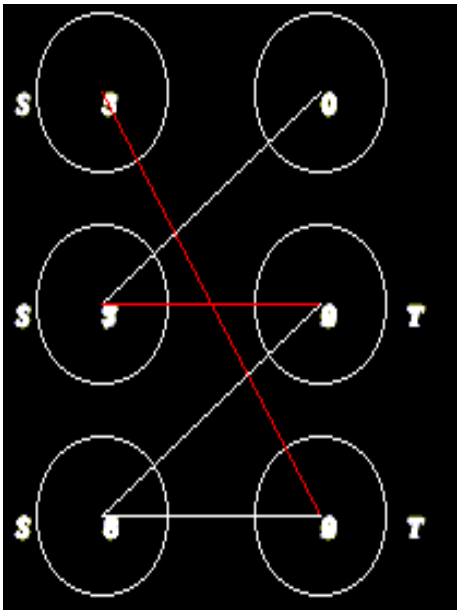
2. We build the alternating tree by adding edges from a vertex $y \in Y$ which has already been matched to some vertex $x' \in X$. We add x' to S and y to T.



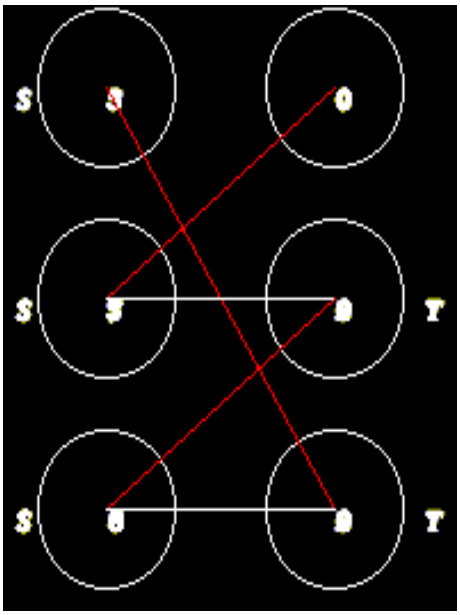
3. We further build the alternating tree.



4. Now, $N(S) = T$ and we cannot build the tree further. Hence, we update labelling ($\alpha_l = 2$) to include other edges in the equality graph.



5. An augmenting path has been found! Flip the edges in the augmenting path to obtain a perfect matching.



4 References

- Notes from UC Saint Barbara
- Brilliant.org
- Translated version of emaxx.ru
- Tutorial on TopCoder