You can find the codes in C++ in the corresponding Codeforces contest.

1 Definitions

There are not too many definitions to start with.

Definition 1. *Matching — the set of edges in the graph that don't have any vertices in common*

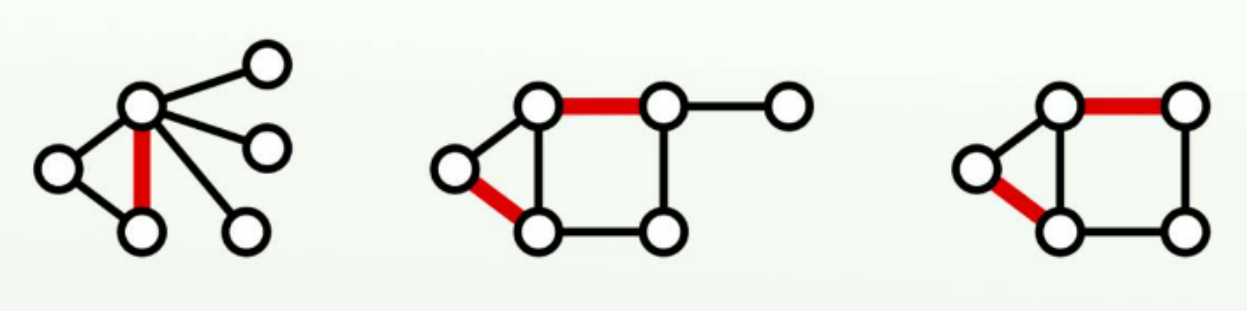


Figure 1: Matching examples

Definition 2. *Inclusion maximal matching — a matching to which no edges can be added.*
For example, on the figure 1 all matchings are Inclusion maximal matchings.

There can be more than one maximal by inclusion matching. Moreover, they can have different sizes. One of them can be found greedily. Simply iterate through all the edges and add them to the matching if the incident vertices are not covered yet.

Definition 3. *Maximum matching — the matching that has the biggest cardinality (the number of edges in it).*
The matchings 1 *and* 2 *are* ***not*** *maximum while matching* 3 *is one.*

There can be more than one maximal matching in the graph.

2 Matching in the tree

Finding the maximum matching in the arbitrary graph is out of scope of this lecture but you can take a look at it if you want here: https://en.wikipedia.org/wiki/Blossom_algorithm

Anyway, to start with, let's deal with some simple case. For example, let's find the maximum matching in the tree.

Here we will use an approach called **Dynamic Programming on trees**. Basically, similar to the fup_v you used for finding bridges we will calculate something for each vertex and its subtree. The only question is what is this *something*.

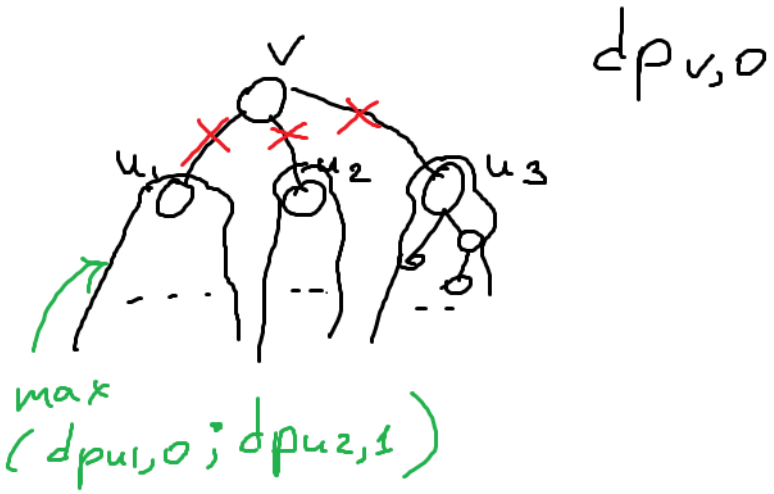
Let's calculate 2 values for each vertex:

- $dp_{v,0}$ — the maximum matching in the subtree of v that **doesn't** include vertex v.

- $dp_{v,1}$ — the maximum matching in the subtree of v that **includes** vertex v.

We can use DFS to calculate them in proper order (for each vertex we firstly calculate the answer for its sons and then for itself). After it the answer is simply $max(dp_{1,0}, dp_{1,1})$ if vertex 1 is a root (you launch DFS from it).

What's the formula for $dp_{v,0}$? Well, in this case we don't take vertex v so we can't take any edges that go from v to its sons. So, now we have a separate maximum matching problem for each son of v. The answer for each son u_i is $max(dp_{u_i,0}, dp_{u_i,1})$.



So, the total formula for $dp_{v,0}$ is:

$$dp_{v,0} = max(dp_{u_1,0}, dp_{u_1,1}) + max(dp_{u_2,0}, dp_{u_2,1}) + \ldots + max(dp_{u_k,0}, dp_{u_k,1})$$

where k is the number of sons of vertex v.

Now let's deal with $dp_{v,1}$. In this case vertex v is taken so we should select and edge which uses it. It will be one of the edges from v to its sons. Let's say it is from v to u_1. Now we have quite a similar case as before. As we can't take any other edge, we have a separate maximum matching problem for each son other than u_1. And for u_1 we have a maximum matching problem where we **can't** take u_1 as it's already taken by the edge (u_1, v). So, for u_1 we can use only $dp_{u_1,0}$ and not $dp_{u_1,1}$. Also, we shouldn't forger to add 1 for the edge (u_1, v).

Thus, the formula for this case is:

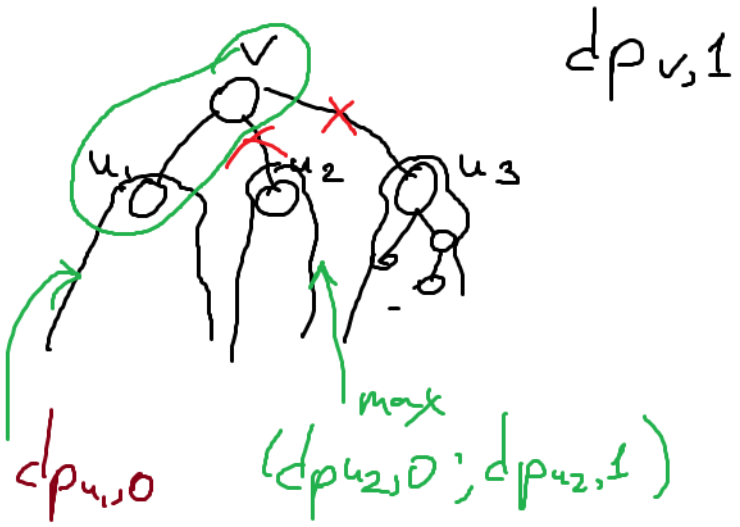
$$dp_{u_1,0} + max(dp_{u_2,0}, dp_{u_2,1}) + \ldots + max(dp_{u_k,0}, dp_{u_k,1}) + 1$$

but we can rewrite it to simplify the calculations:

$$(max(dp_{u_1,0}, dp_{u_1,1}) + max(dp_{u_2,0}, dp_{u_2,1}) + \ldots + max(dp_{u_k,0}, dp_{u_k,1})) - max(dp_{u_1,0}, dp_{u_1,1}) + dp_{u_1,0} + 1 =$$

$$= dp_{v,0} - max(dp_{u_1,0}, dp_{u_1,1}) + dp_{u_1,0} + 1$$

The only thing that is left is to notice that we can use not only u_1 but any other son, so to find $dp_{v,1}$ we simply calculate it for every son and take the maximum.



3 Bipartite graphs

The second case for which we can solve the problem of finding the maximum matching is a bipartite graph. What is it?

Definition 4. *Bipartite graph — the graph where the vertices can be splitted into* 2 *parts so that edges go from one part to another.*

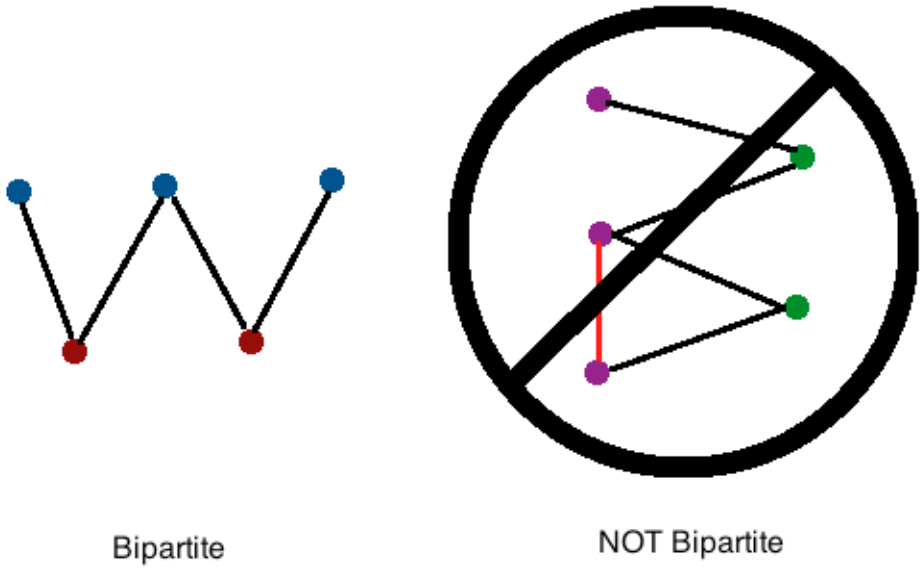


Figure 2: Bipartite and non-bipartite graph examples

How do we check if the graph is bipartite or not? Well, we can notice a very easy property: *If the vertex belongs to the part* 1 *all its neighbors should be in part* 2. So, we can implement

a DFS that traverses the graph and assigns the numbers 1 or 2 to each vertex. If there is a case that we're in vertex from part 1 and there is an edge to an already colored vertex with color 1 it means that the graph is not bipartite. The same is true for part 2.

4 Matchings in the bipartite graphs

To find the maximum matching in the bipartite graph we need to introduce another definition.

4.1 Augmenting path

Definition 5. *Imaging, we have a bipartite graph with some matching selected.*
Augmenting path — a path in this graph that:

- *Starts in the part* 1.
- *Ends in the part* 2.
- *All edges that are in this path and go from part* 1 *to part* 2 *are not in the matching.*
- *All edges that are in this path and go from part* 2 *to part* 1 *are in the matching.*
- *The vertices where the path starts and ends are not covered by the matching.*

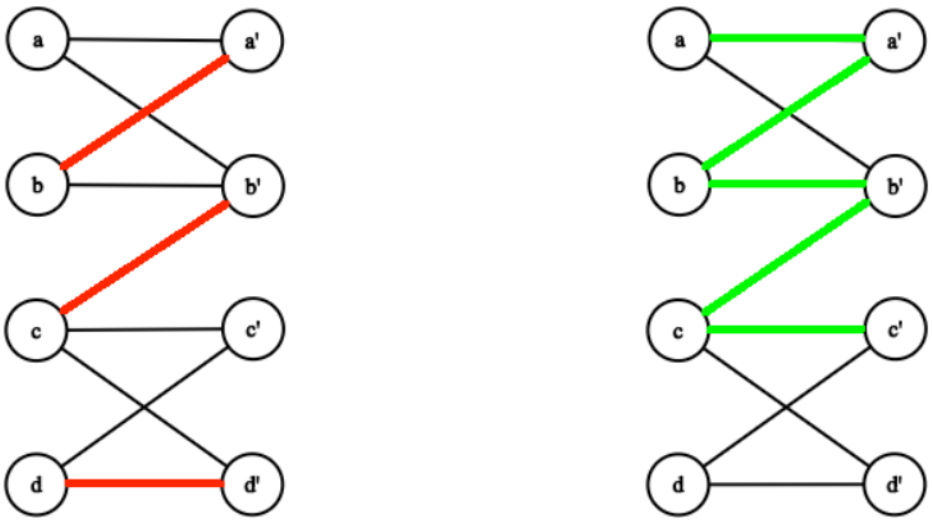


Figure 3: Example of an augmenting path

What is good about augmenting path? Well, let's take the edges from the path that are in the matching and remove them from it. Also, let's take the edges from the path that are not in the matching and add them to the matching. After this process we get a matching with a greater size than before. If we use an augmenting path from figure 3 we can get:

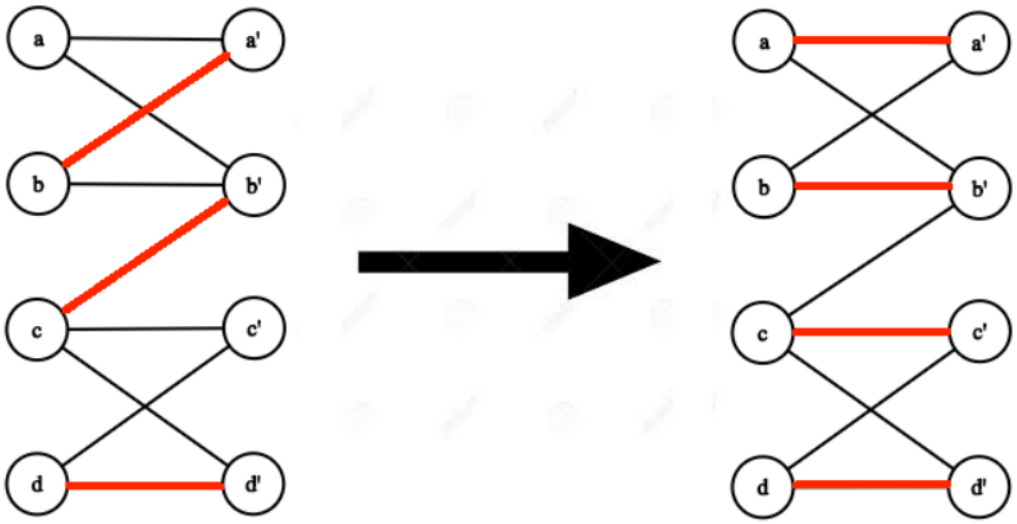


Figure 4: Example of an augmenting path usage

The key fact we'll use in the algorithm is a so-called **Berge's lemma**: *The matching in the graph is the maximum one if and only if there is no augmenting path.* Actually, we already proved that if the matching is maximum there is no augmenting paths. Technically, we also need to prove that if there is no augmenting paths then the matching is maximum. But we won't waste time on it. You can check the proof here: https://cp-algorithms.com/graph/kuhn_maximum_bipartite_matching.html#berges-lemma

4.2 Kuhn's algorithm

So, our algorithm is simply a search for augmenting paths as long as there is one. We'll go through vertices in part 1 and try to find augmenting path that starts there. Again, you can find the code in the Codeforces contest. Complexity: $O(nm)$

5 How to use this algorithm to solve problems

Let's sovle problem F from the contest.

There is a room that can be represented as a matrix of size $n \times m$. The room is covered with parquet, but it is pretty old, so some parts of the parquet are already broken.

Cells of the room where the parquet is **not broken** are marked with dots '.', and cells of the room where the parquet is **broken** are marked with stars '*'.

You have two possible opportunities: the first one is to buy a parquet piece of size 2×1 (or 1×2 if you rotate it) for a coins or buy a parquet piece of size 1×1 for b coins. Note that you can not break the parquet piece of size 2×1.

Your task is to cover **all** cells with the broken parquet for the minimum possible cost.

Figure 5: Problem statement

Let's say we have cnt broken cells. Firstly, we can notice that if $2b \leq a$ the answer is simply $cnt \cdot a$. If not, our task is basically to put as many tiles 2×1 or 1×2 as possible.

Let's create a graph where each cell is a vertex. There is an edge between two cells if they both can be covered by a tile 2×1 or 1×2. Now we can notice that solving the problem with putting as many tiles is equivalent to finding a maximum matching in this graph.

Why this graph is bipartite? Well, if you color the parket as a chessboard you will notice that in the graph the edge always connects black and white cell. So, essentially, the color in this coloring is the identifier of the part the vertex belongs to.

The interesting fact is that in this graph we have nm vertices and $4nm$ edges. So, the complexity of the solution is $O(n^2m^2)$ and if substitute the maximum values for n and m we can get something like $300^4 \approx 8 \cdot 10^9$ operations. But if you optimize it properly you will be able to fit into 500 ms. Worse implementations should work for around 2000 ms.