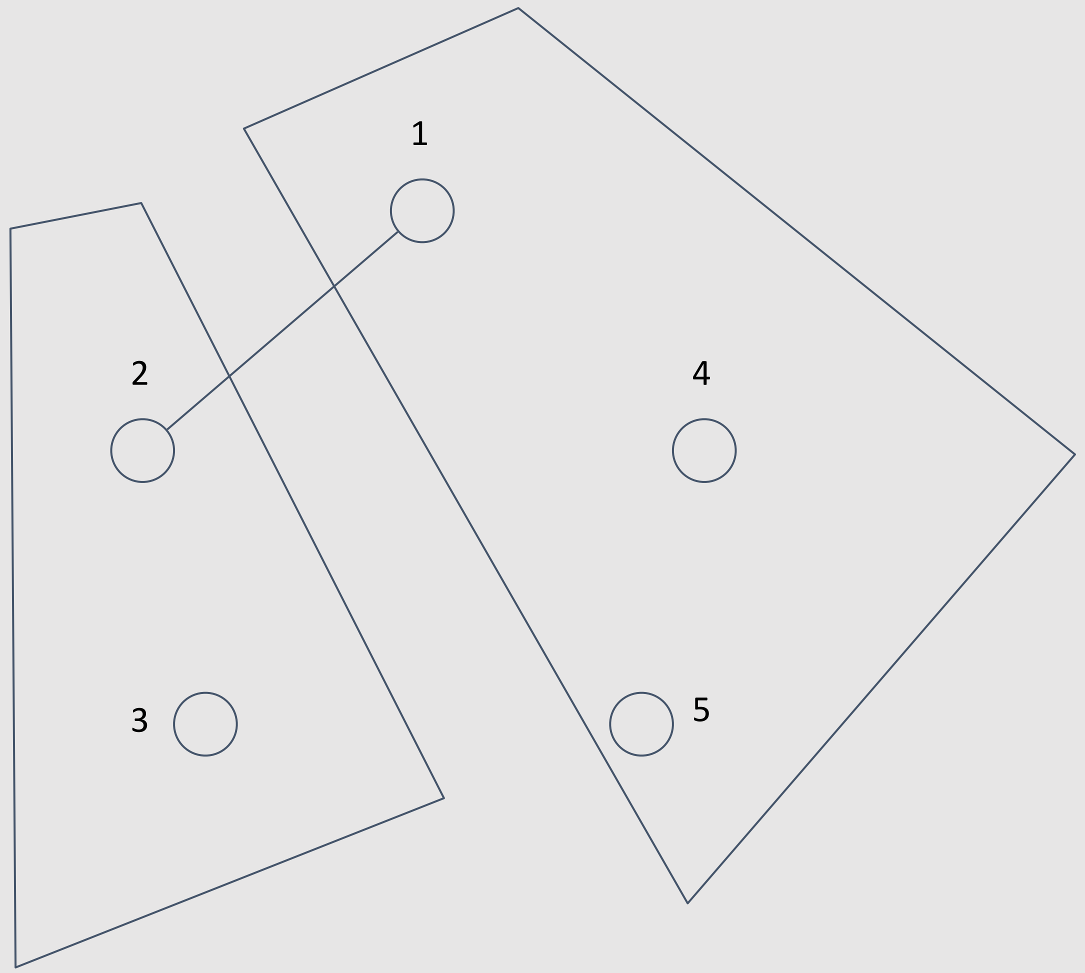
Matching

Bipartite graph

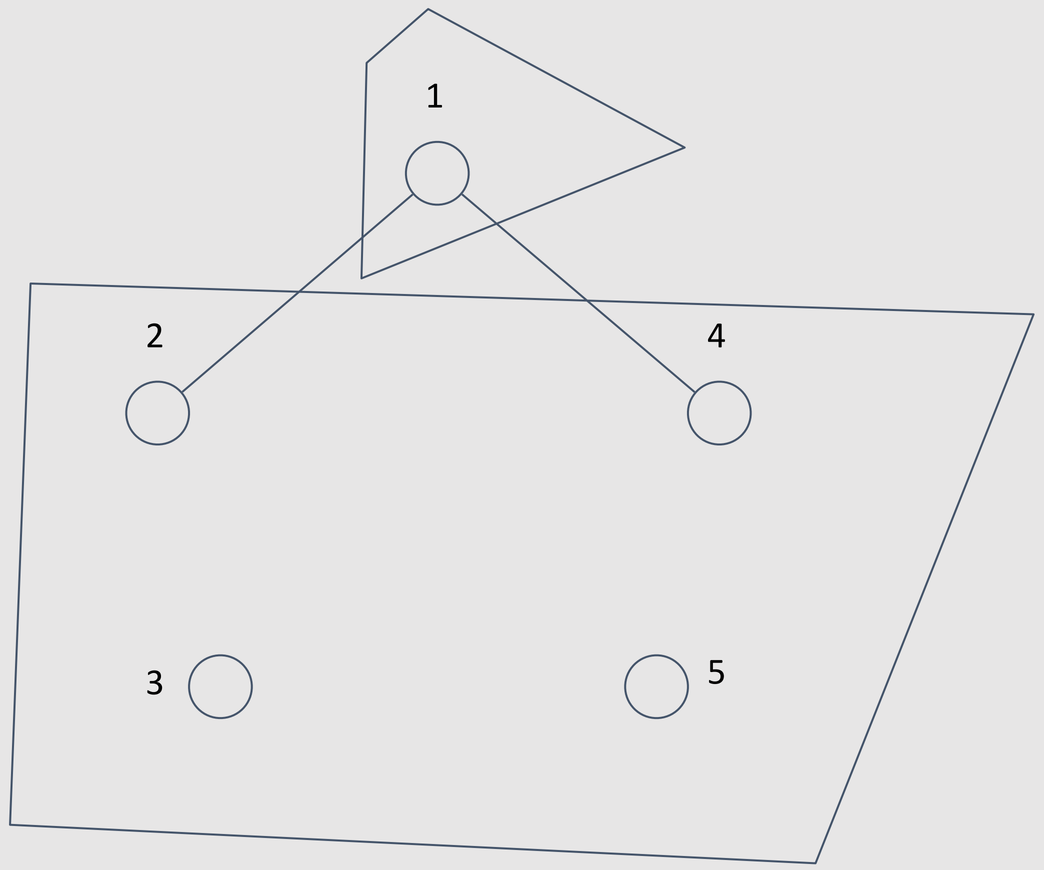
We have already mentioned bipartite graphs, but let's repeat just in case what they are.

A bipartite graph is a graph that can be divided into two sets such that there are no edges between the vertices of one set.

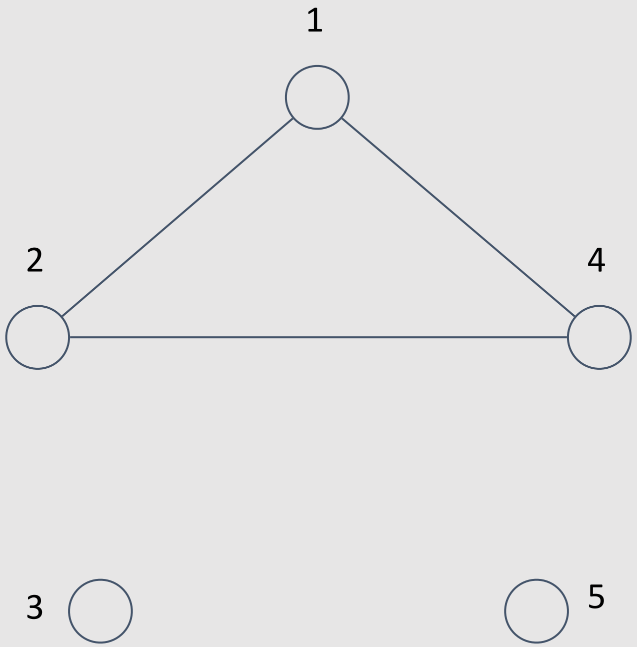
YES



YES



NO



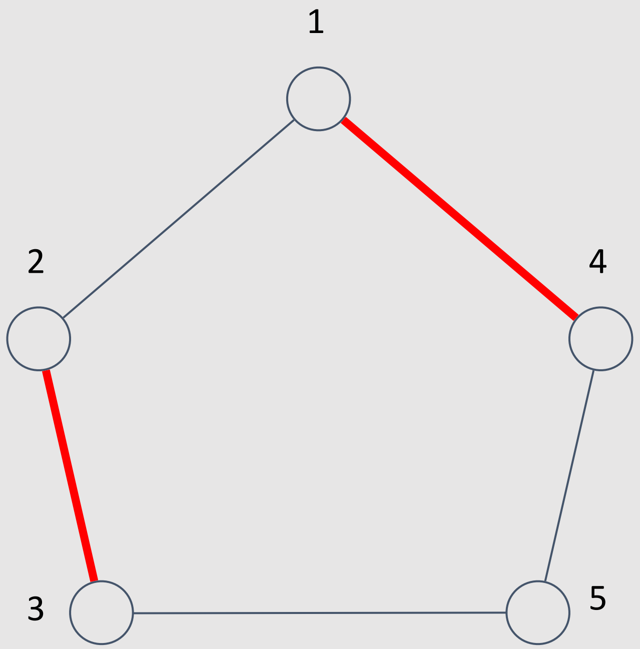
Real task

At the table, there are n people and m dishes, and for each person, it is known which dishes they would like to eat. It is required to select the maximum number of pairs (person; dish that they like).

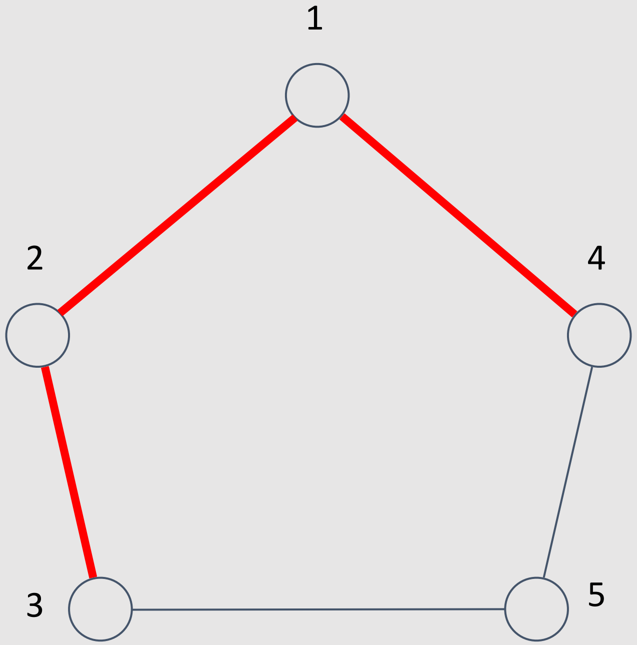
Matching

A matching M is called a set of edges in a graph that are pairwise non-adjacent (in other words, no vertex in the graph should be incident to more than one edge from M).

Yes



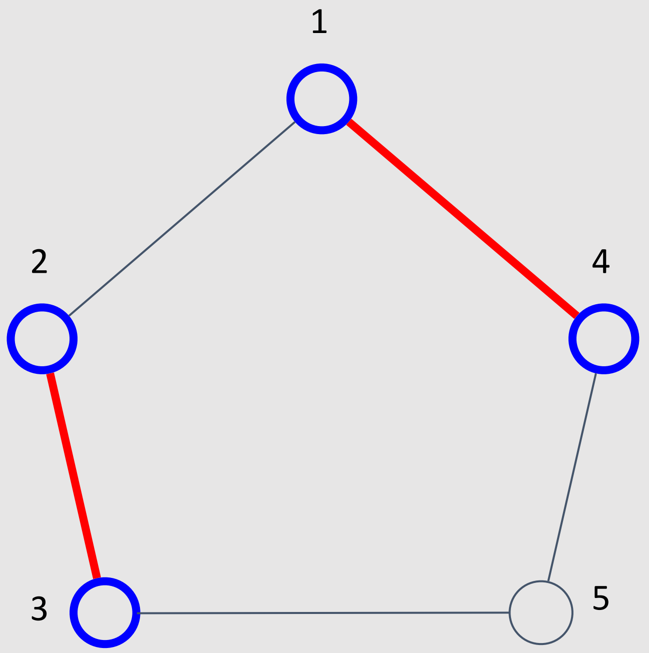
No



Saturated vertex.

All vertices that have an adjacent edge from the matching (i.e., which have a degree of exactly one in the subgraph formed by M) are called saturated by this matching.

saturated



Matching

At the moment, we will be looking for matchings only in bipartite graphs.

Maximal matching

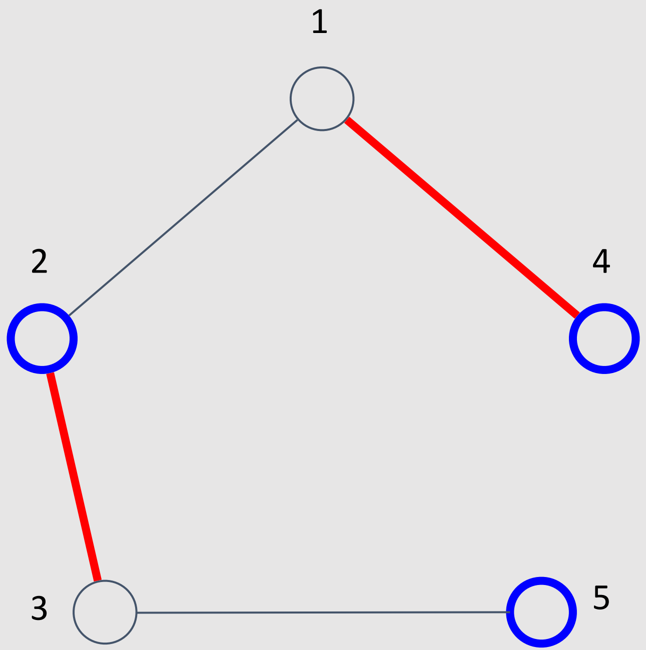
The size of a matching is called the number of edges in it.

A maximum matching is a matching whose size is the largest among all possible matchings in the given graph.

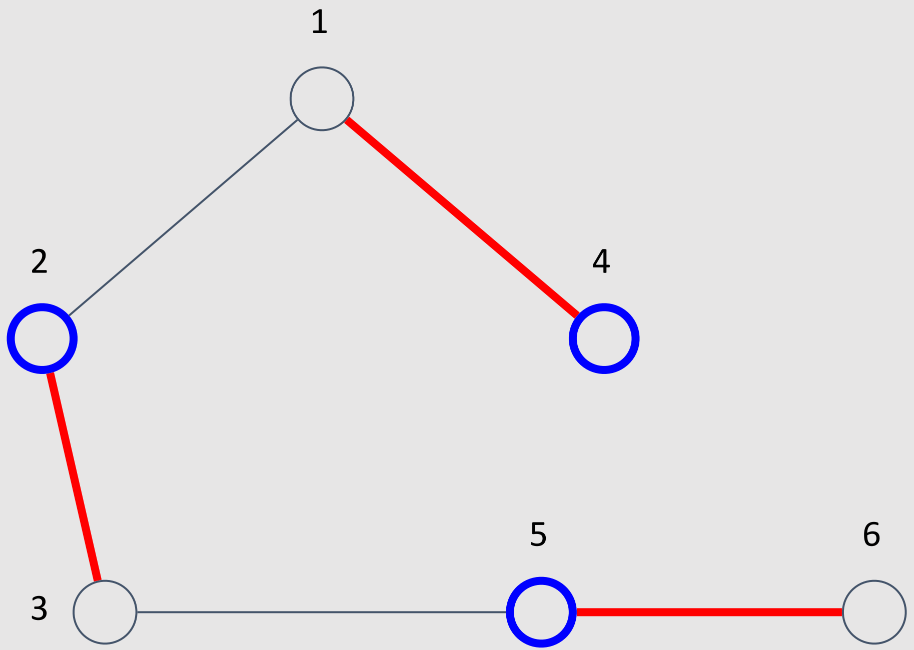
A maximal matching is a matching M of a graph G that is not a subset of any other matching.

Perfect — where all vertices of the left set are saturated by it.

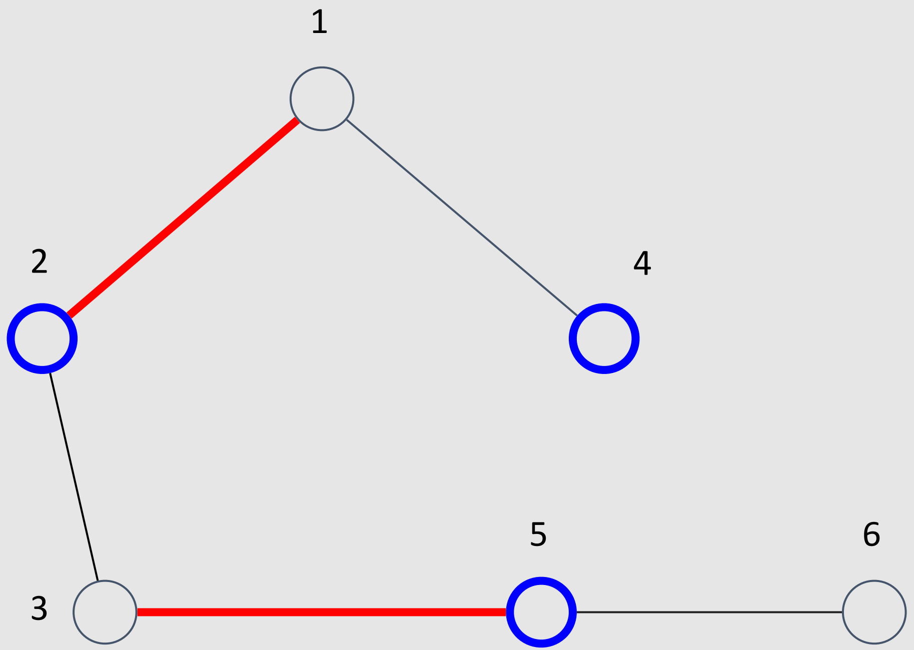
|M| = 2, maximal and maximum matching, non-perfect



|M| = 3, maximal, maximum, perfect matching



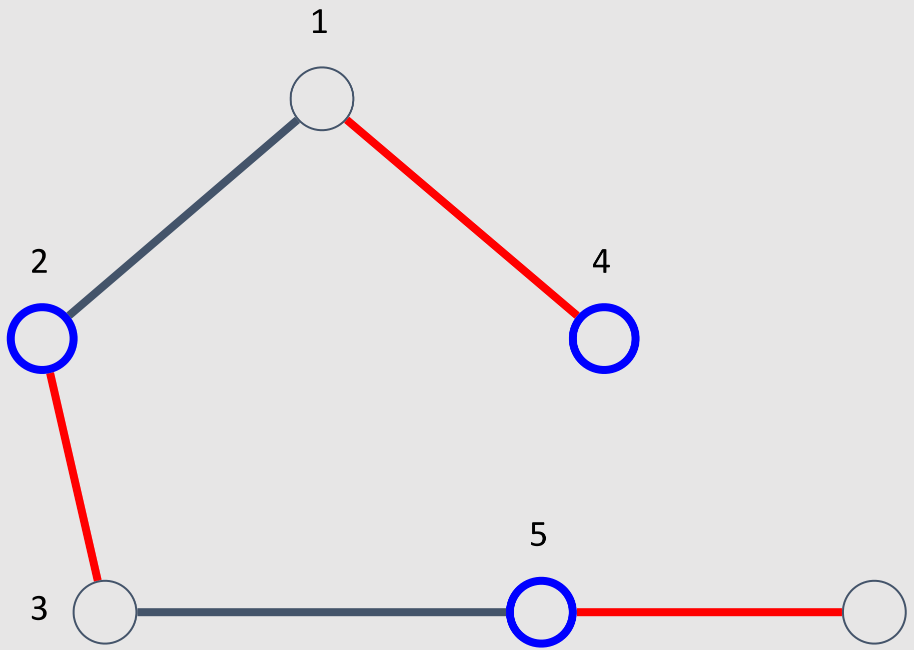
|M| = 2, maximal matching



Matching

A chain of length k is called a simple path (i.e., one that does not contain repeating vertices or edges) that contains exactly k edges

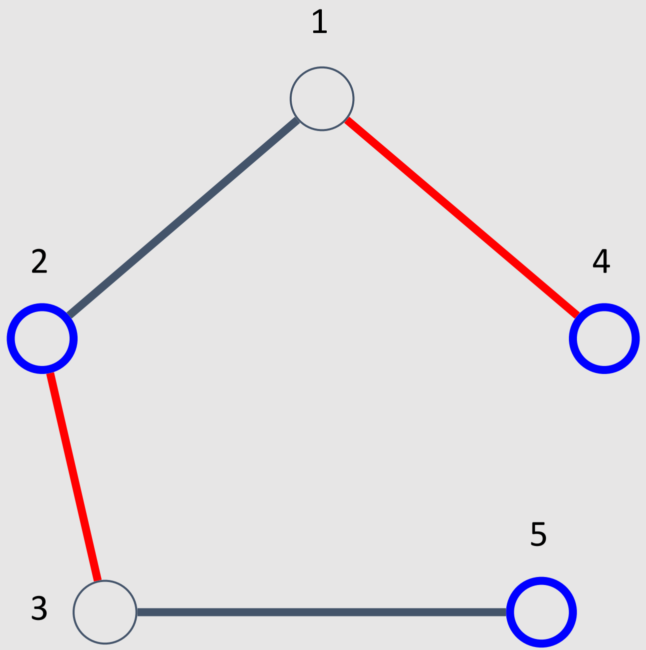
|chain| = 5



Matching

An alternating path with respect to a certain matching is called a simple path of length k in which the edges alternately belong/do not belong to the matching, with the first edge not belonging to the matching.

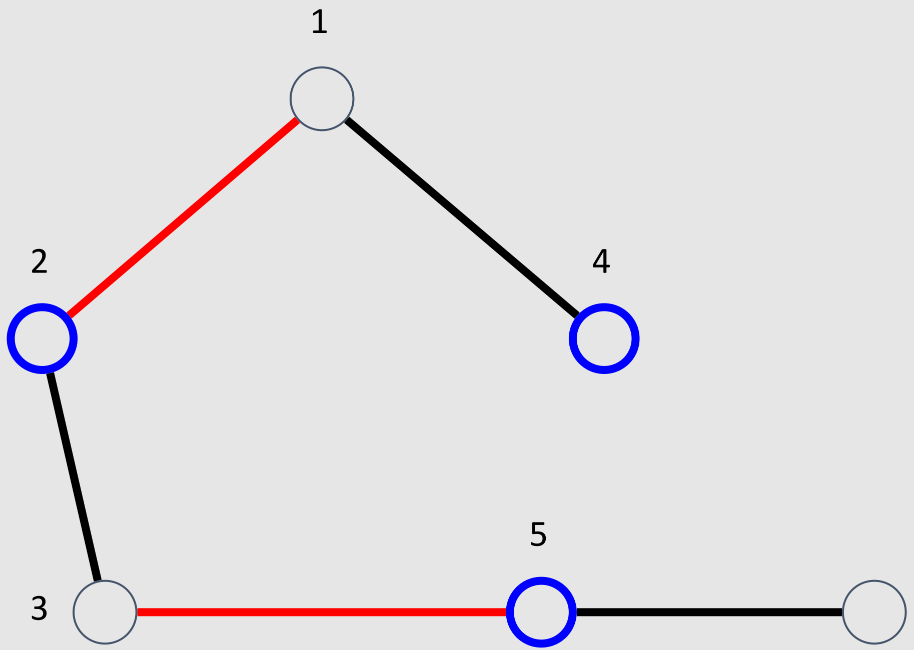
An alternating path from 5-th vertex



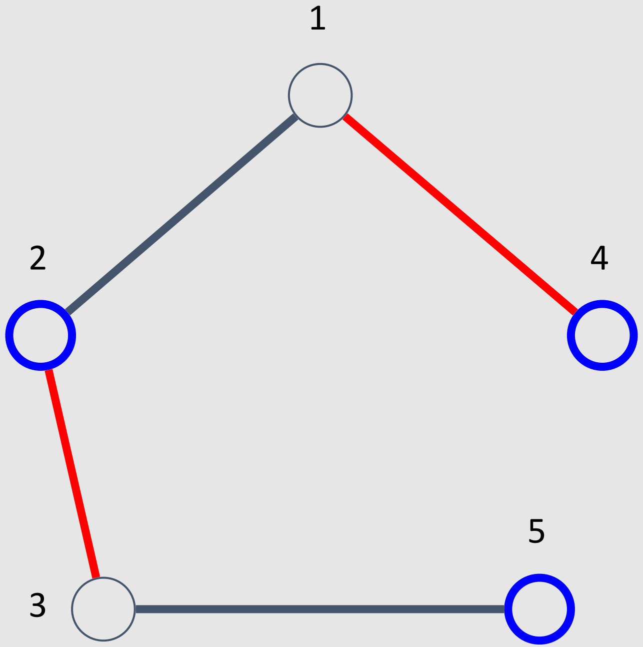
Matching

An augmenting path with respect to a certain matching is called an alternating chain whose initial and final vertices do not belong to the matching.

An augmenting path



No



Berge's theorem

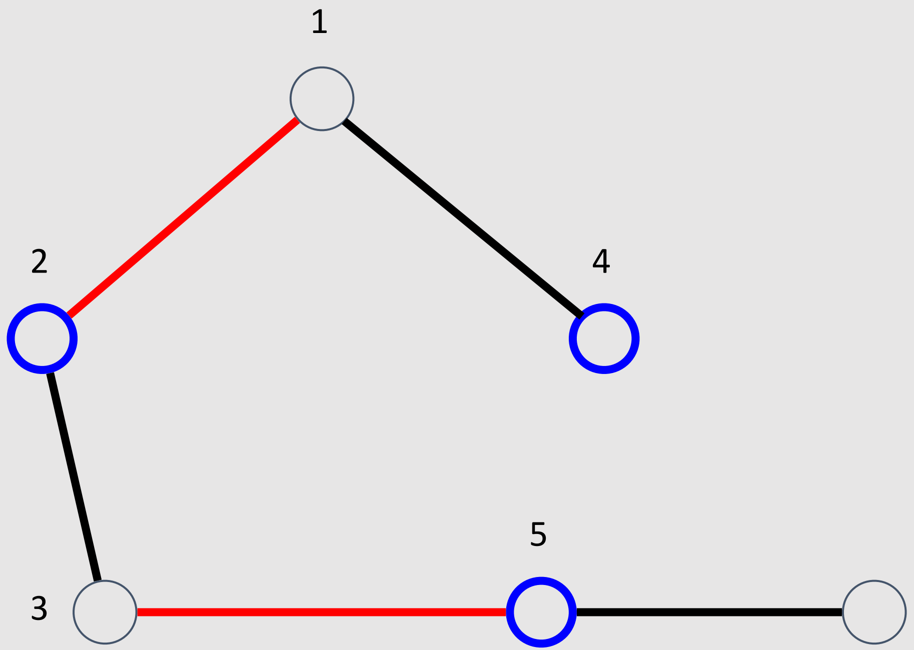
matching hasn’t augmenting path $\leftrightarrow$ matching is maximum.

Proof

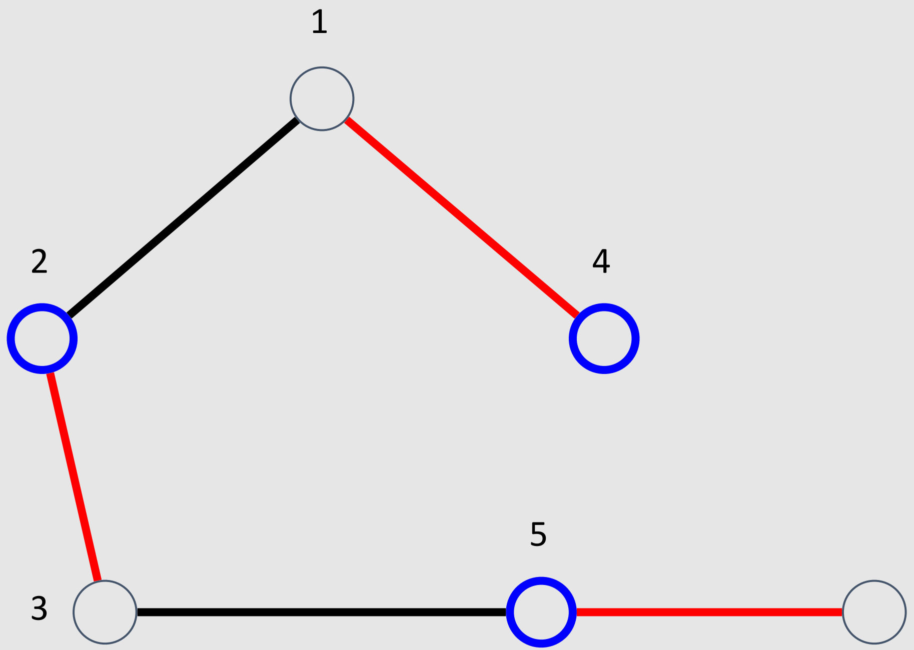
1) matching hasn’t augmenting path <- matching is maximum.

OR if matching has augmenting path, then matching is not maximum.

With augmenting path



Reverse



Proof

We can simply reverse path and we will get M’, such as lM’l = lMl + 1

Proof

2) matching hasn’t augmenting path -> matching is maximum.

OR if matching hasn’t augmenting path, then matching is maximum.

Proof

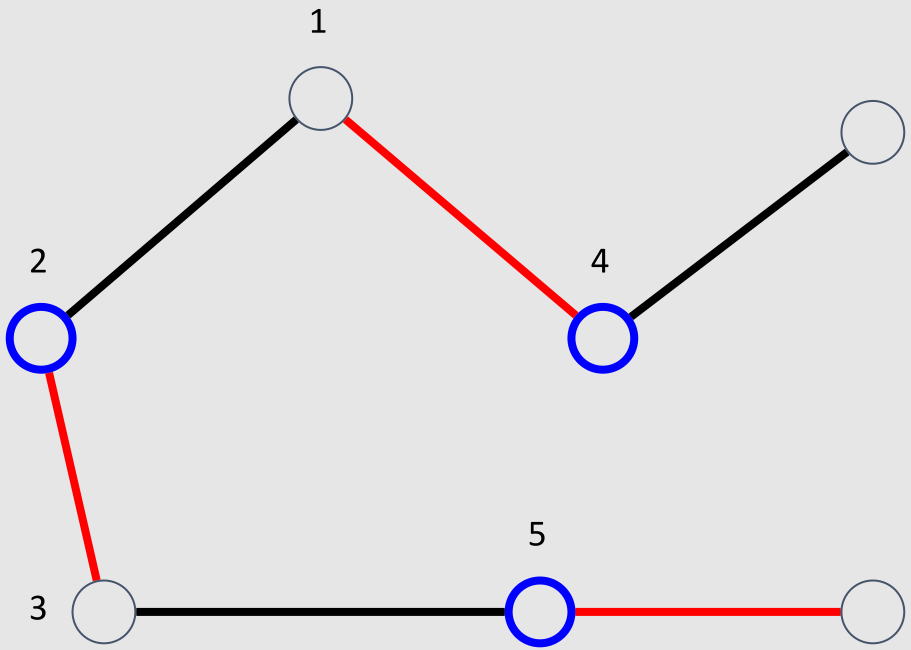
M = matching without augmenting path

M’ = maximum matching.

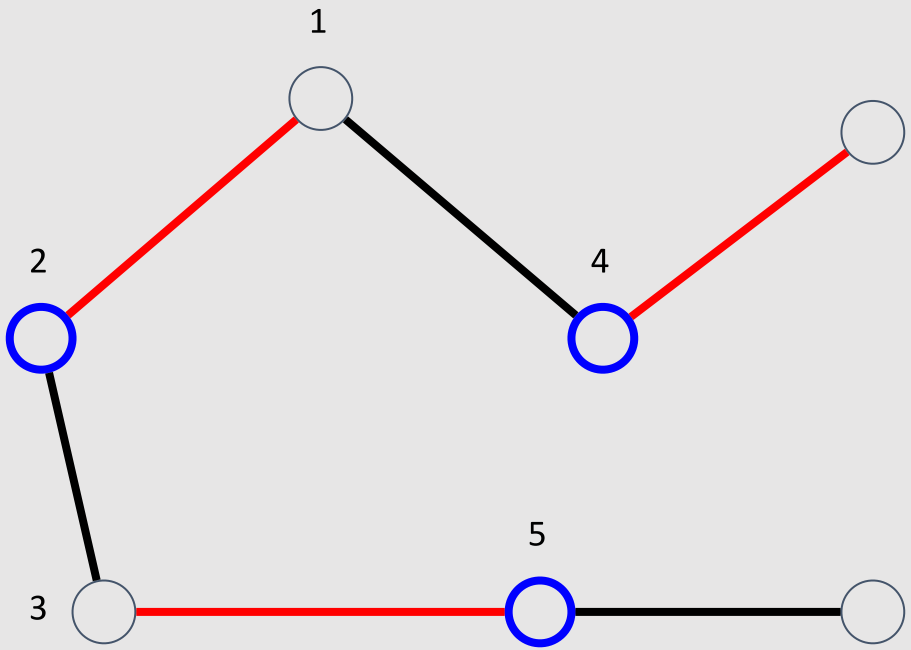
Proof

Let’s take M^M’

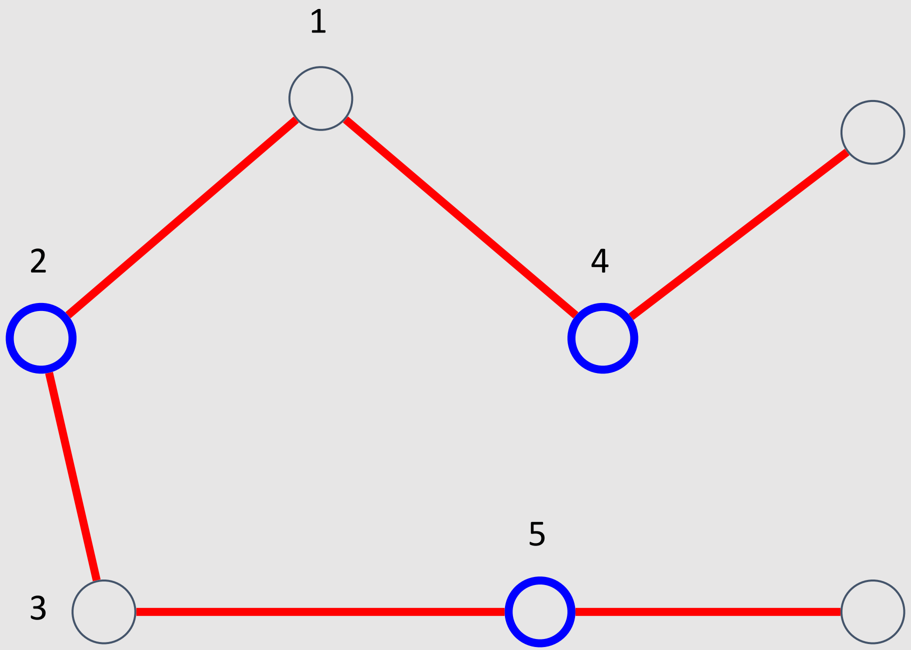
M’



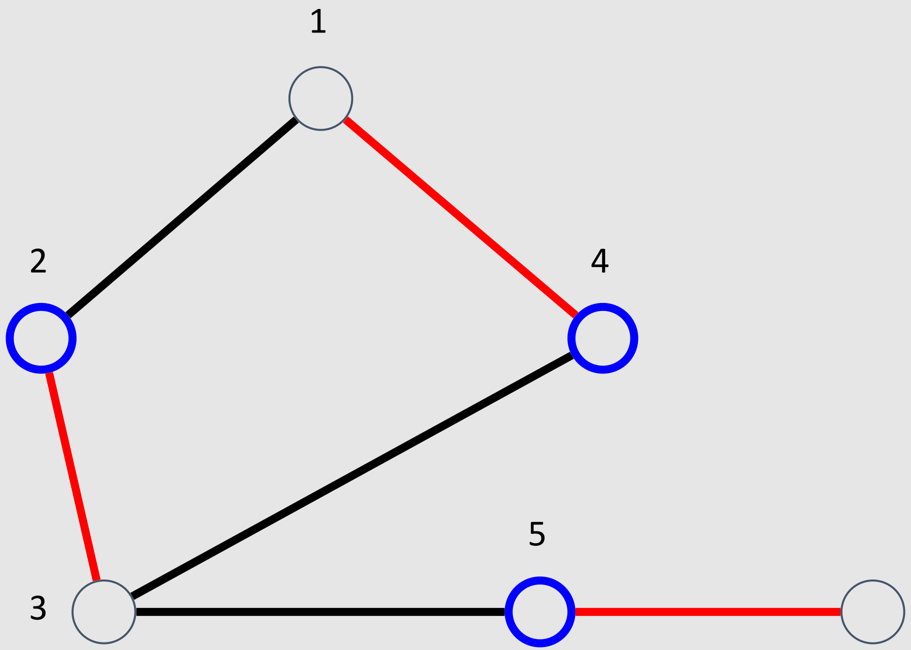
M



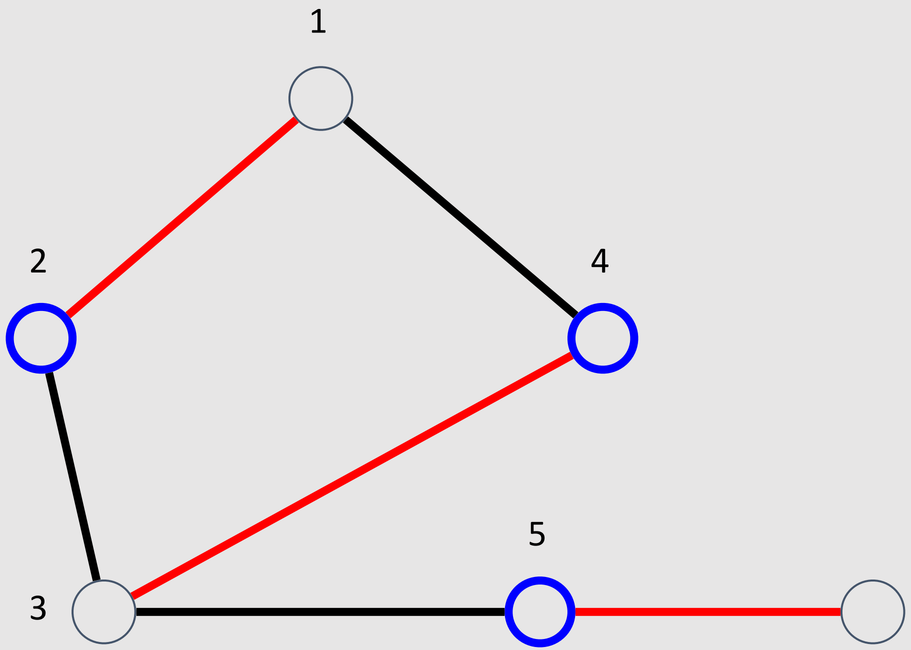
M’^M



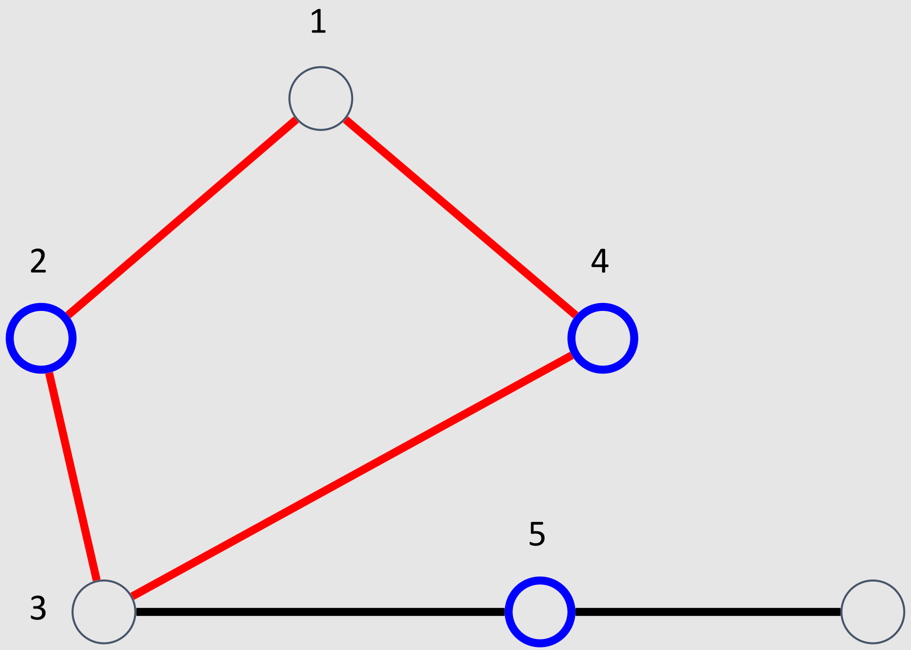
M’



M



M'^M



Proof

Let’s take $X = M^{M'}$

We want to prove $|M| = |M'|$, we will think about X.

Proof

What can we see inside?

1) Vertex

2) Path

3) Cycle

Vertex

1) Vertex

Just ignore it, it doesn’t give you any edges in X

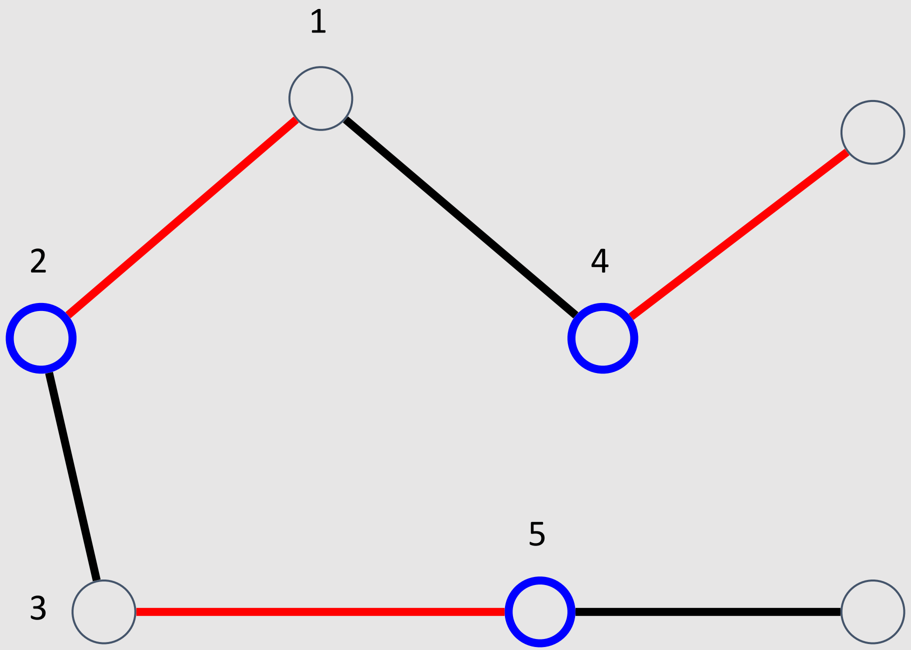
Path

2) Path

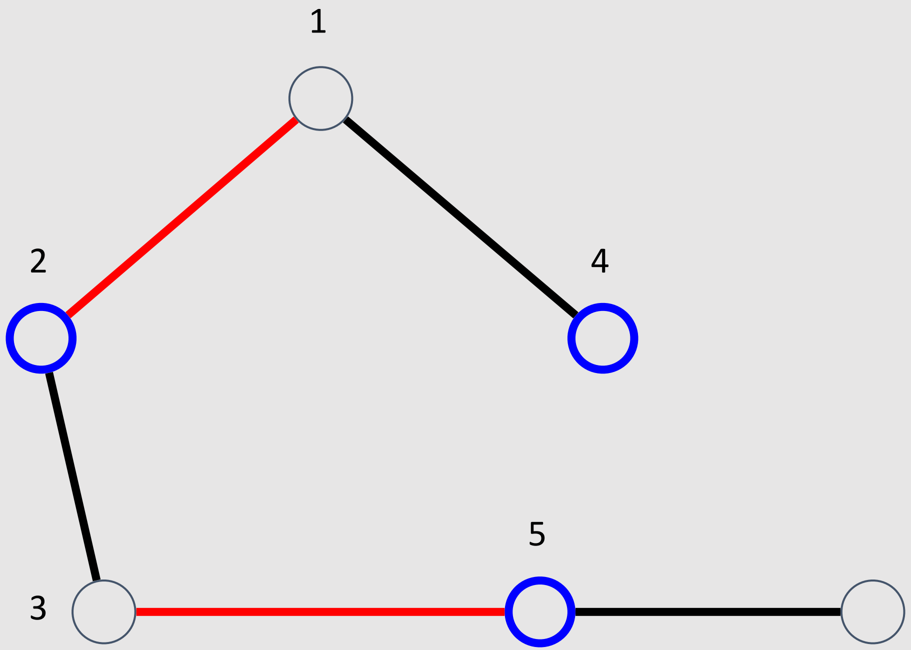
Path can only contain an even number of edges.

This is because if it contains an odd number of edges, it means that there is an augmenting path either in M or in M'. However, this cannot be the case, as M by definition does not contain an augmenting path, and M', being a maximum matching, cannot contain one either, as already proven.

Path, black from M, red from M’



Impossible path, black from M, red from M’

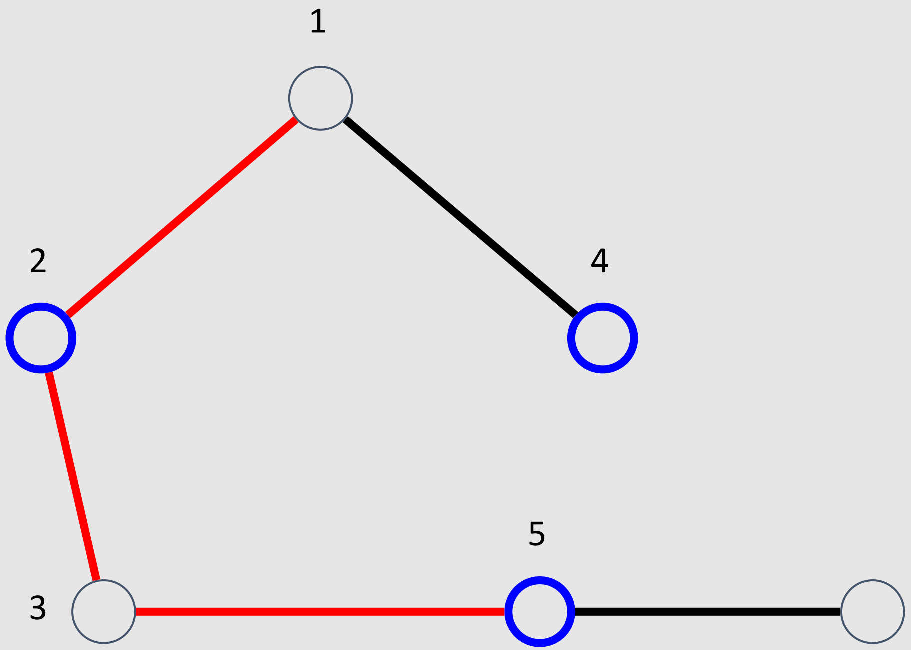


Path

Moreover, in this path, the edges from M' and M alternate, because otherwise, we would not have a matching.

Therefore, the paths contain an equal number of edges from M and M'.

Impossible path



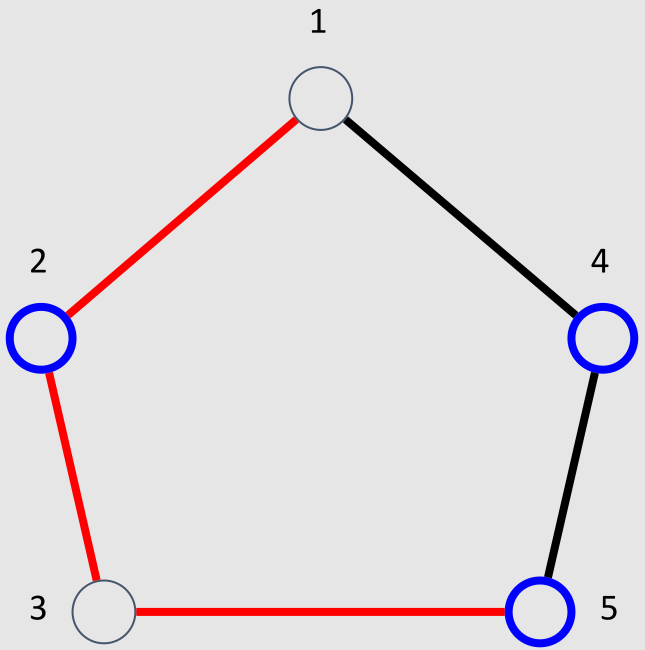
Proof

3) Cycle

A cycle can only contain an even number of edges.

This is because if it contains an odd number of edges, it means that we have an edge from one set to the same set, which would not make it a bipartite graph.

Impossible cycle



Result

By the same logic, the edges from M' and M alternate, because otherwise, we would not get a matching.

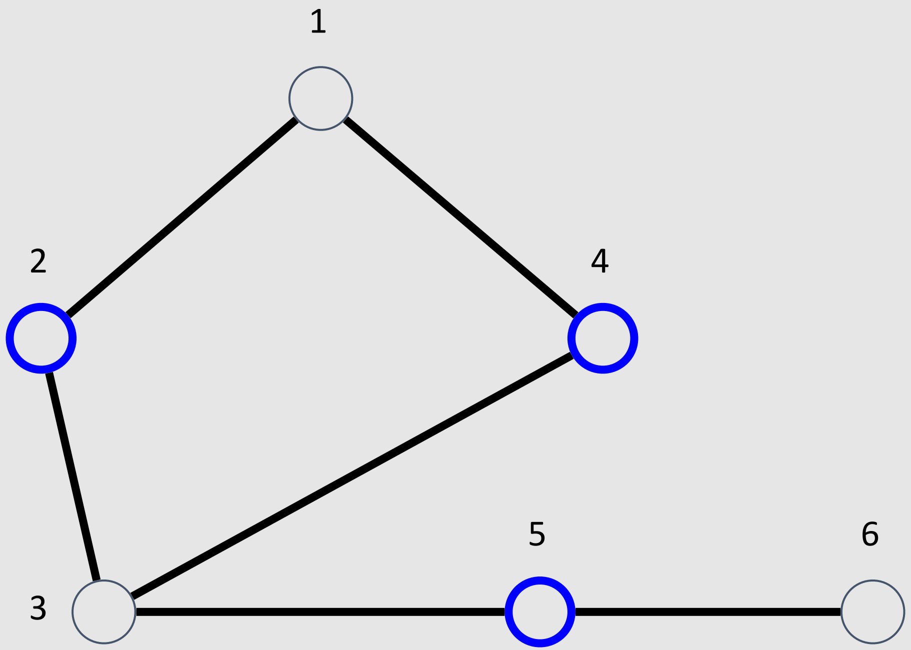
Consequently, cycles also contain an equal number of edges from M and M'.

Therefore, there is an equal number of edges from M' and M in X, and since the XOR operation only removes identical edges, there is an equal number of edges in both M' and M.

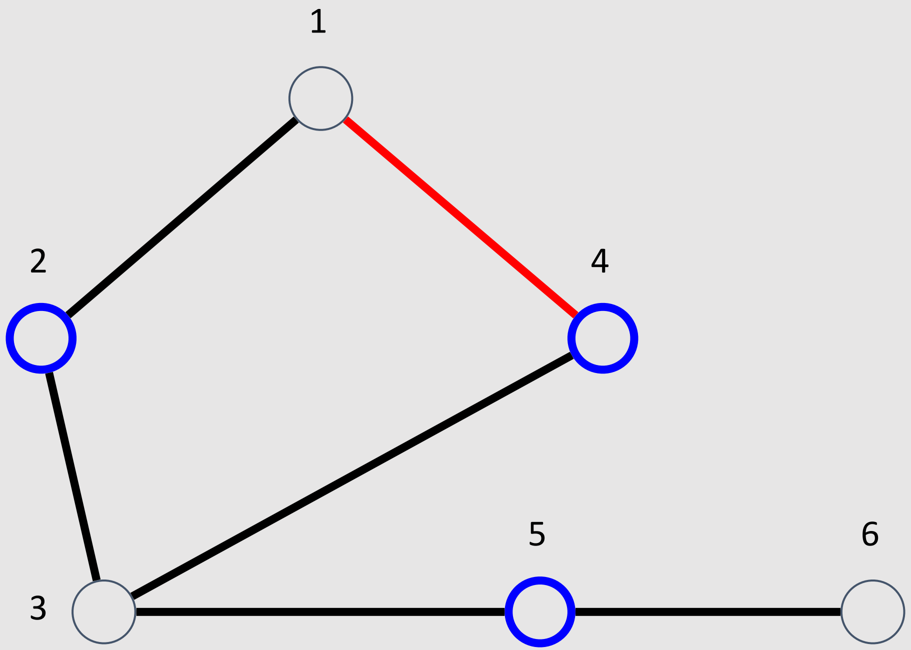
And what?

So what have we obtained? Well, now that we can find matchings, let’s just look for augmenting paths as long as they exist. Even the most naive implementation is possible in O(n^2 * m), because we can look for a matching using dfs from each vertex up to n times (the maximum number of edges in max matching).

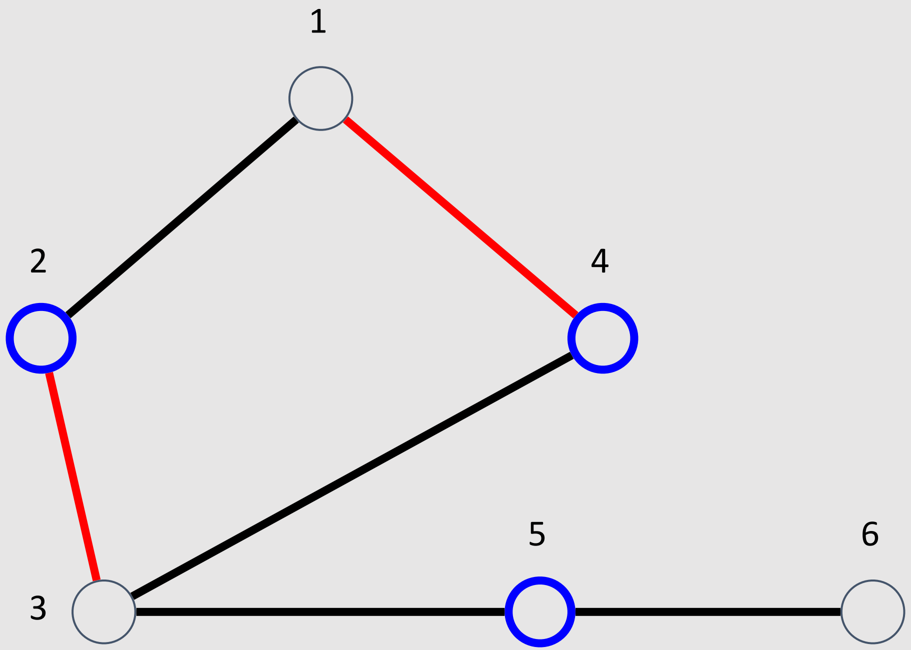
G



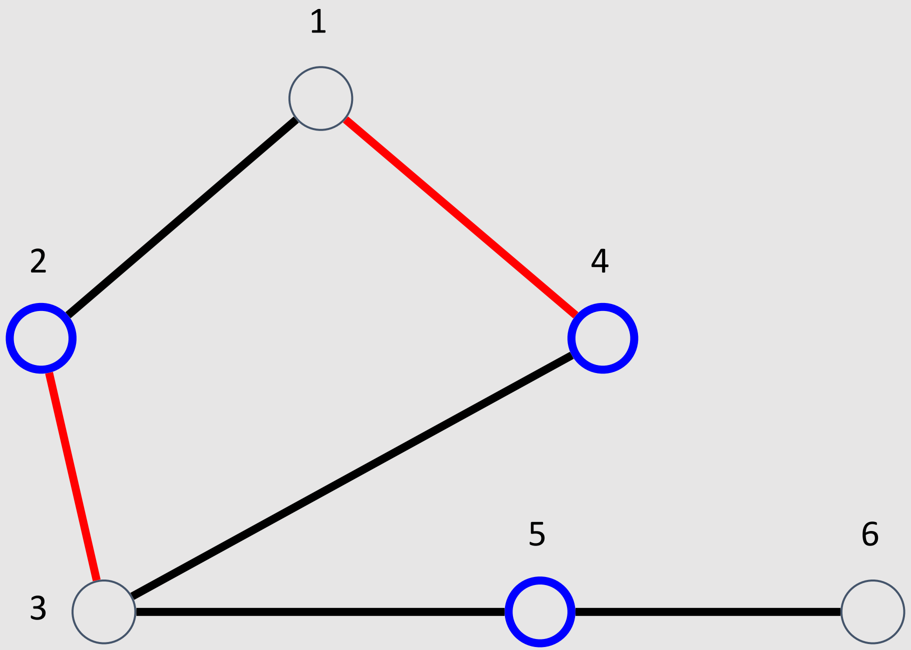
1

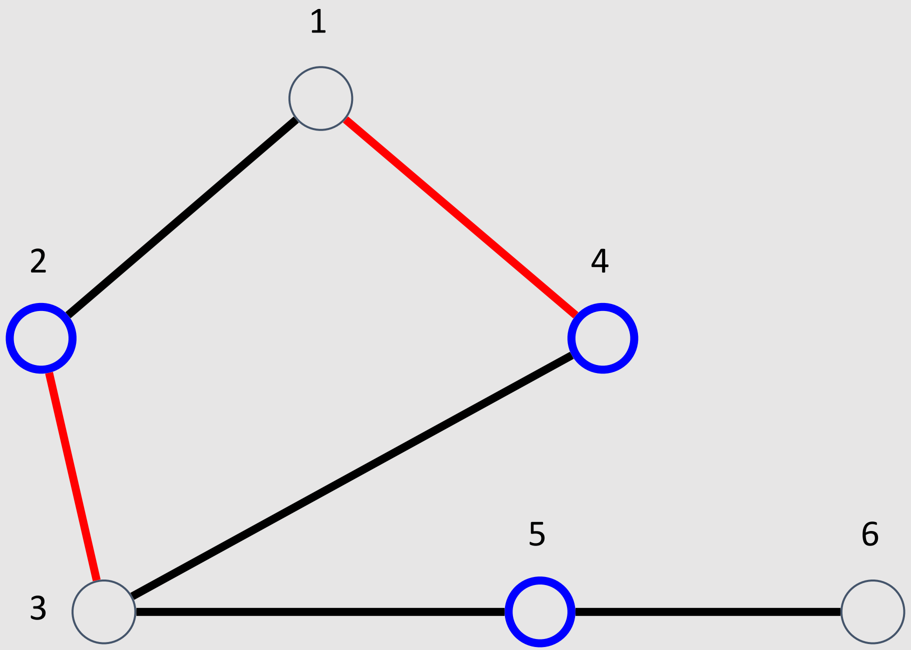


2

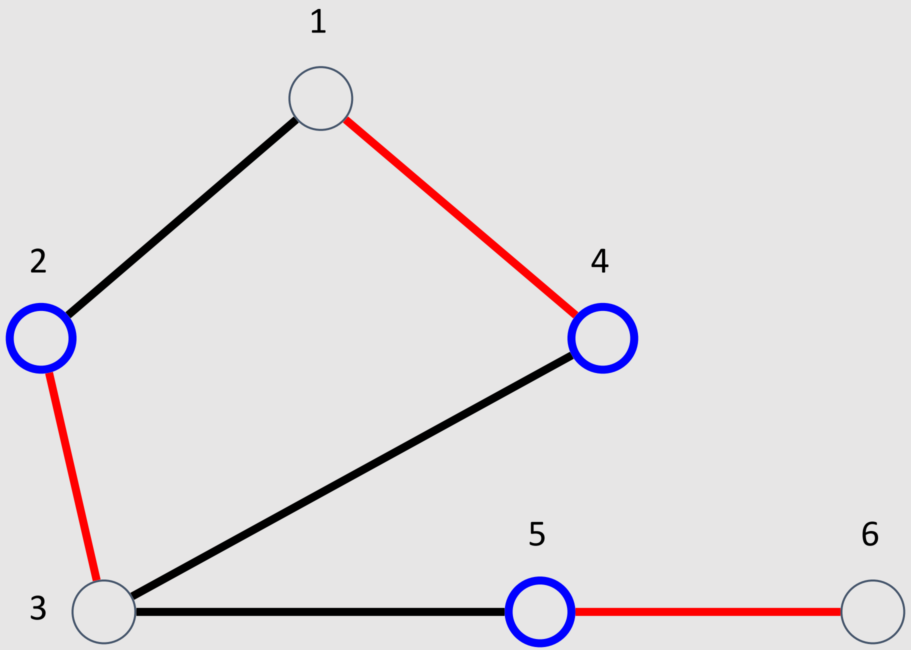


3

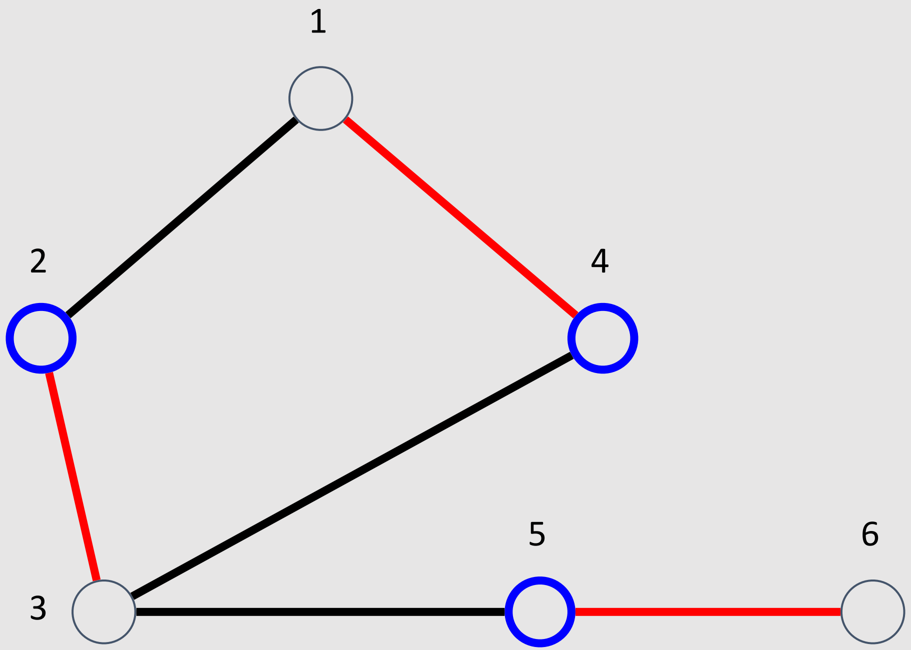



1
2
4
3
5
6

5



6



Optimization

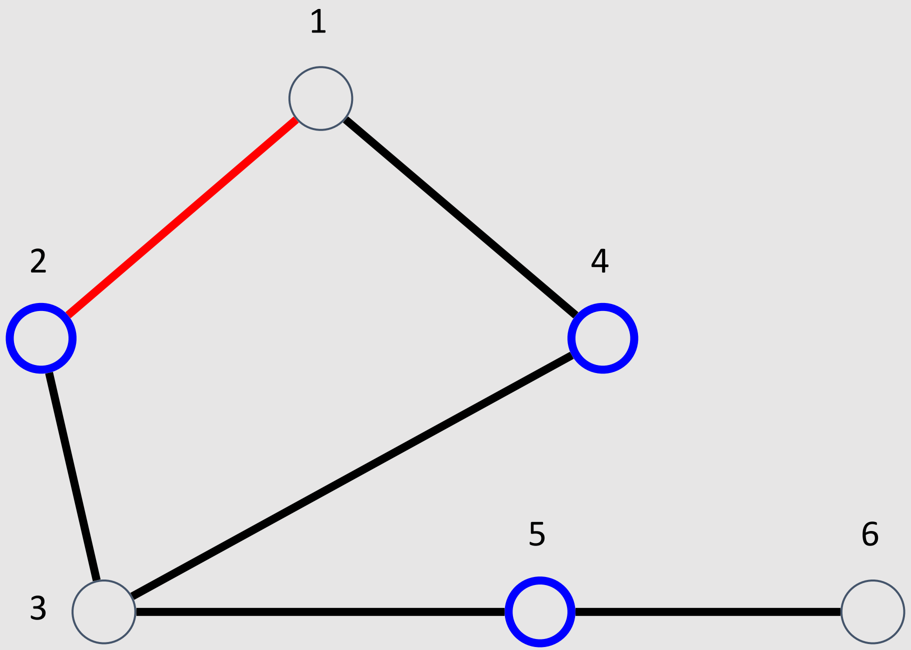
Can it be done faster?

Of course, let's just look for an augmenting path from each vertex just once.

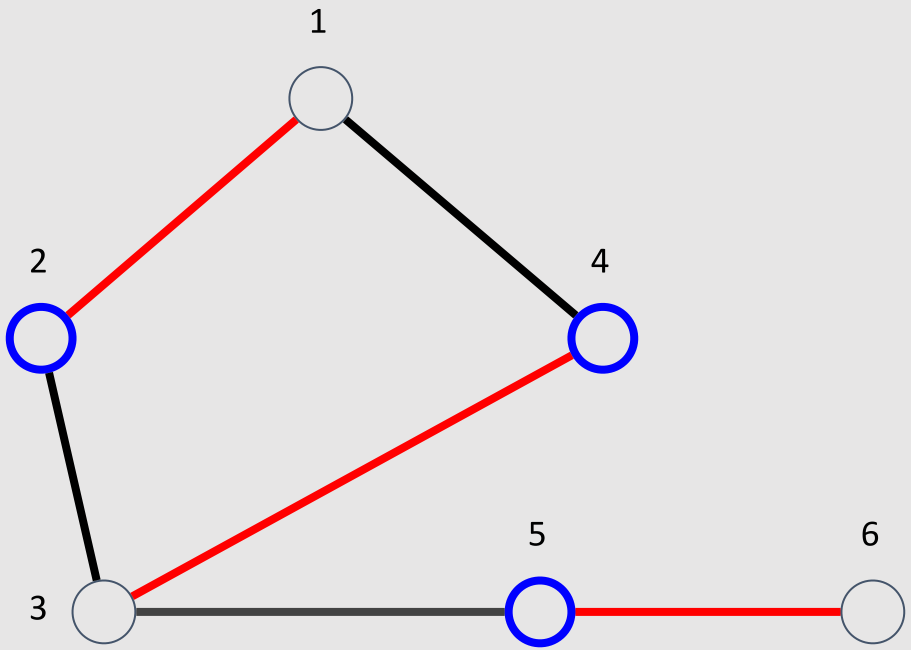
Why?

Well, the answer is actually quite simple. If we have already checked a vertex, then an augmenting path can only reappear there if we have inverted some other augmenting path, but then we will find at least one more vertex that has not been checked yet, which starts the augmenting path, because our already checked vertex cannot start the path.

some submatching



Let’s imagine we already checked [1, 2, 3], we check 4



Why?

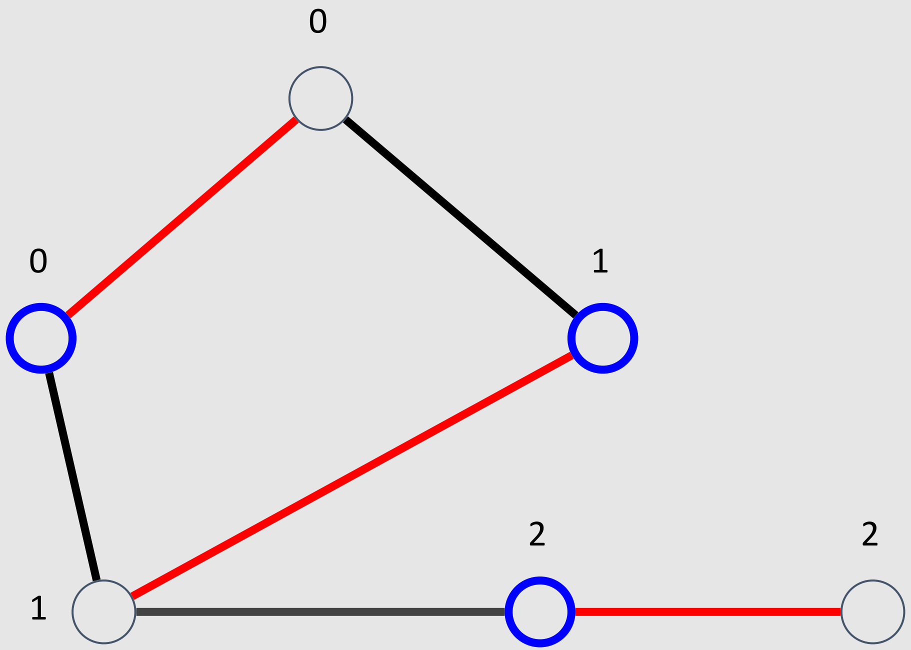
If something new appeared from vertex 3, then we could invert it together with vertex 4.

Code

Let's write the code.

For simplicity of implementation, let's assume that we are given a graph in the form of size_l (the number of vertices in the first partition), size_r (the number of vertices in the second partition), and m (the number of edges).

We will also assume that the edges are defined from the first partition to the second.

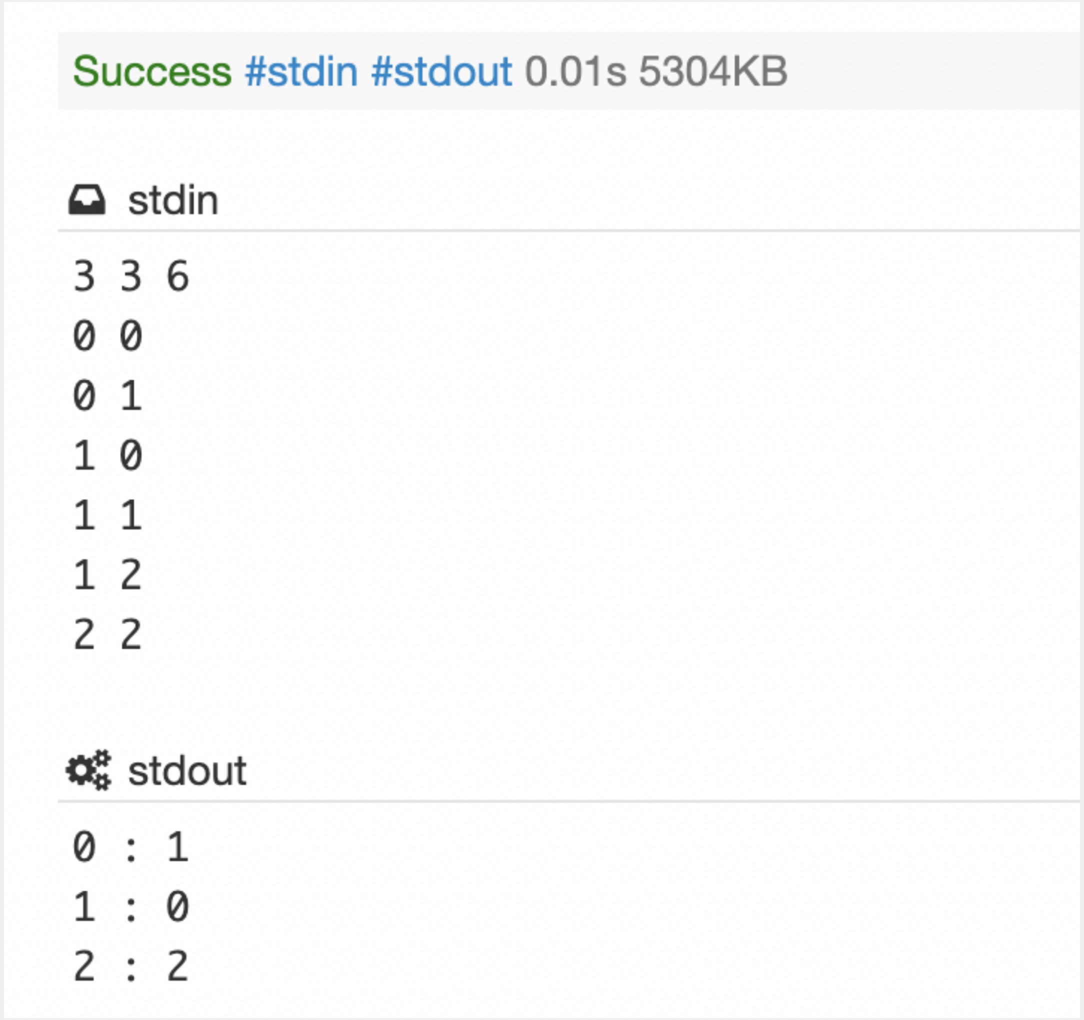

0
0
1
1
2
2

```
// dfs returns whether it is possible to find an augmenting path from vertex v
// to some vertex in the right partition
// if possible, it also establishes this path
bool dfs(int v, vector<int> &matched, vector<bool> &used, const vector<vector<int>> &graph) {
    if (used[v]) {
        return false;
    }
    used[v] = true;
    for (int u : graph[v]) {
        // if the vertex is free, then it can be matched immediately
        // if it is occupied, it can only be matched if
        // an alternate vertex can be found for its current pair
        if (matched[u] == -1 || dfs(matched[u], matched, used, graph)) {
            matched[u] = v;
            return true;
        }
    }
    return false;
}
```

```
int main() {
    int size_l, size_r, m;
    cin >> size_l >> size_r >> m;
    vector<vector<int>> graph(size_l); // we will store only edges from the left partition to the right
    for (int i = 0; i < m; i++) {
        int from, to;
        cin >> from >> to;
        graph[from].push_back(to);
    }

    vector<int> matched(size_r, -1); // which vertex in the right partition is matched with (-1 if none)
    for (int i = 0; i < size_l; i++) {
        vector<bool> used(size_l, false); // auxiliary array for path searching with dfs
        dfs(i, matched, used, graph);
    }

    for (int i = 0; i < size_r; i++) {
        cout << i << " : " << matched[i] << "\n";
    }
}
```



Success #stdin #stdout 0.01s 5304KB

stdin

```
3 3 6
0 0
0 1
1 0
1 1
1 2
2 2
```

stdout

```
0 : 1
1 : 0
2 : 2
```

And what?

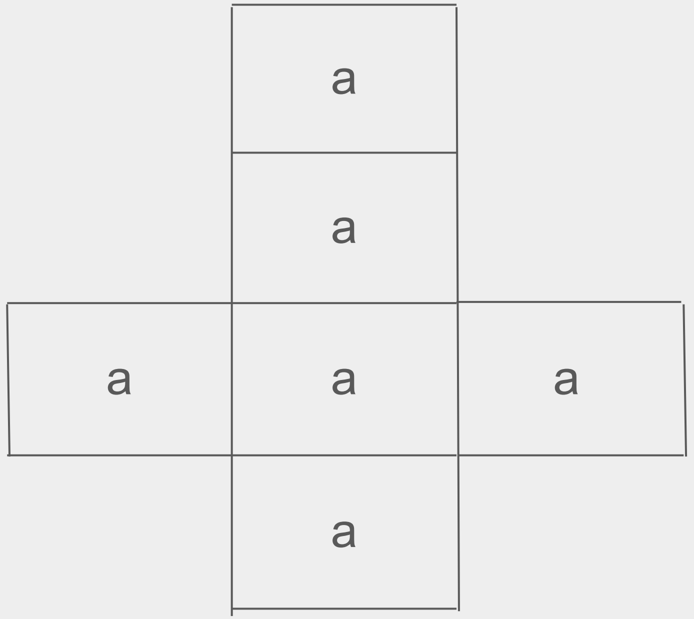
Well, we've learned how to find a matching, so what? In fact, this is called the Kuhn algorithm, and it can solve a lot of things.

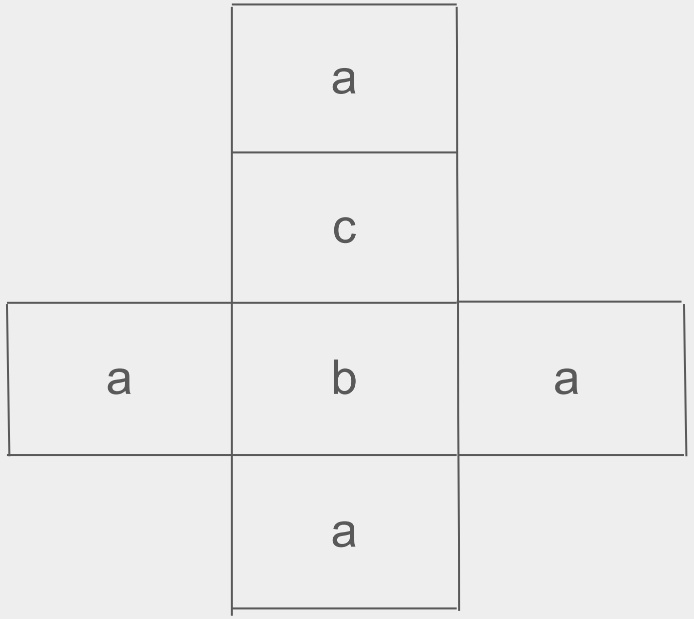
Let's learn how to solve tasks using this method.

Cubes

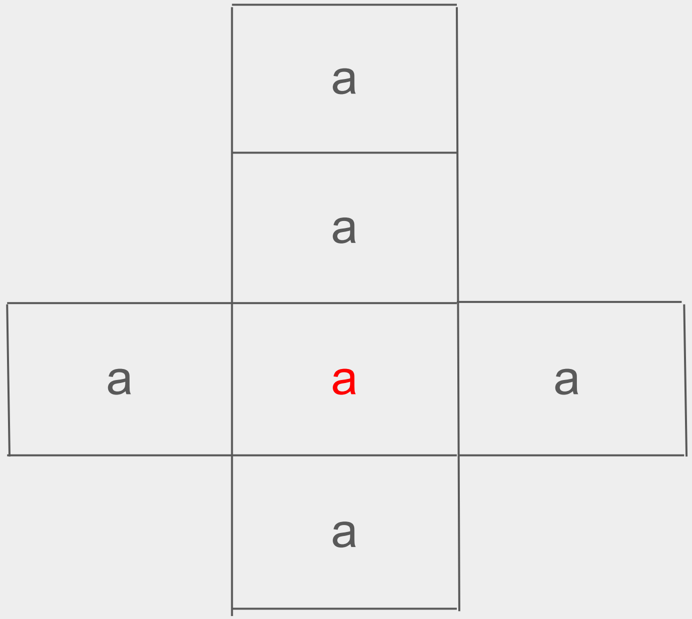
Given n cubes, each with 6 faces, and on each face, there is a letter. Given a word s, it is required to assign a unique cube to each letter of the word s so that we can rotate this cube and obtain the letter we need.

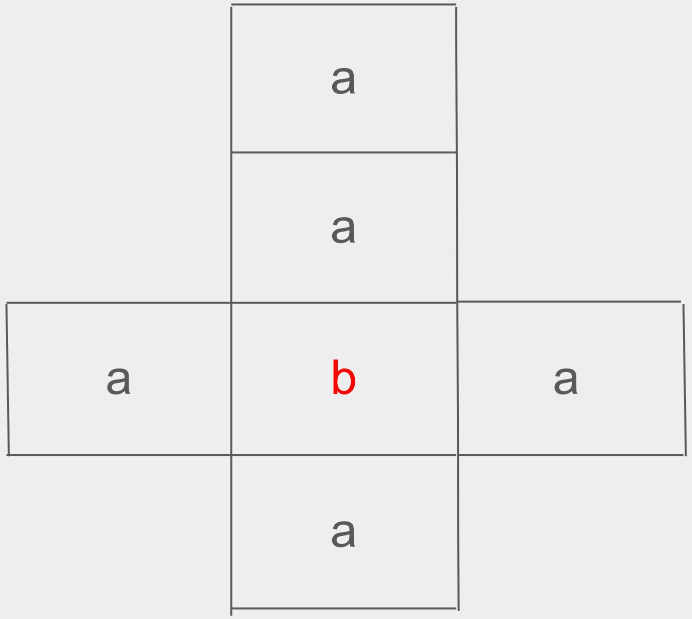
bc? - No





ab? - Yes





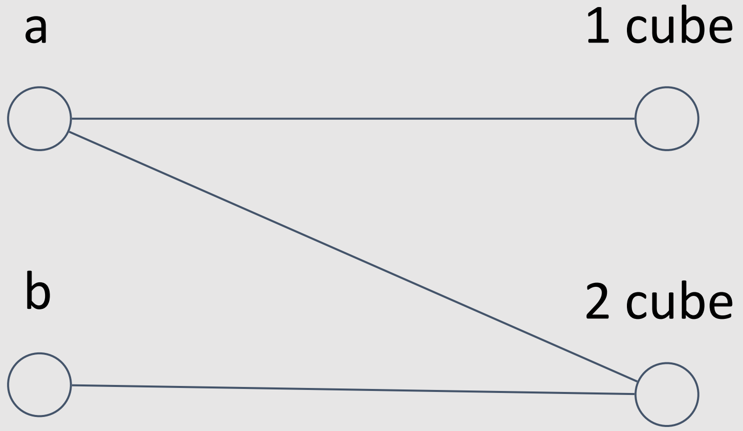
How to solve?

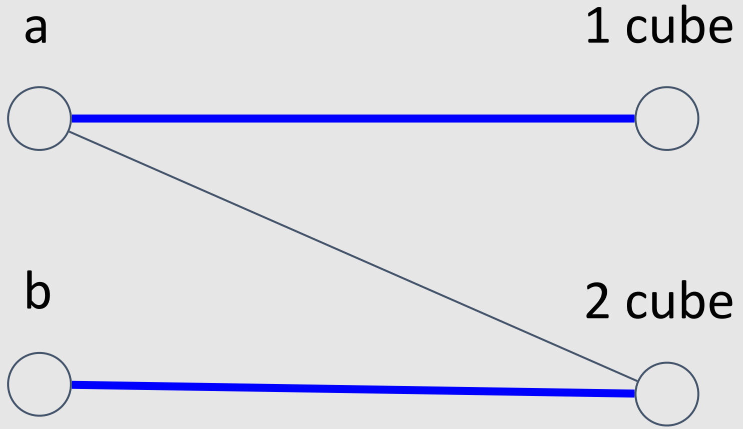
In tasks involving matching in bipartite graphs, it's important to identify what represents the first and second parts and under what circumstances an edge is drawn between them.

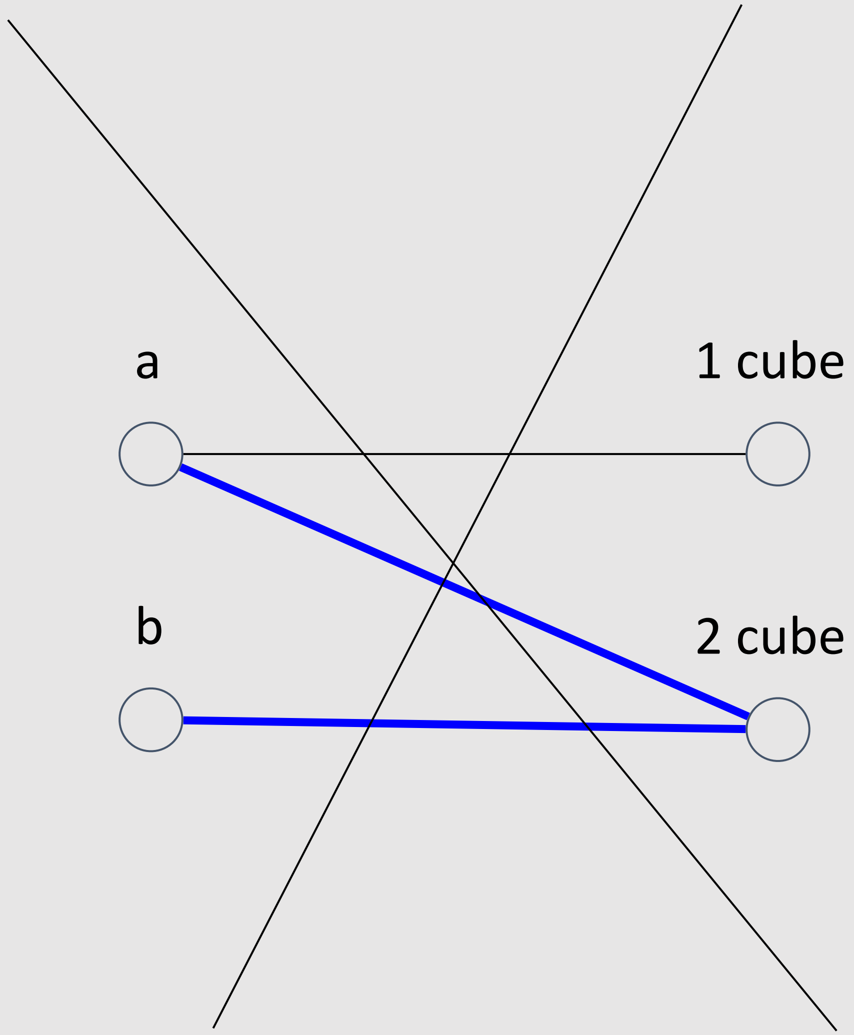
Let's consider the first part to be the letters of the string.

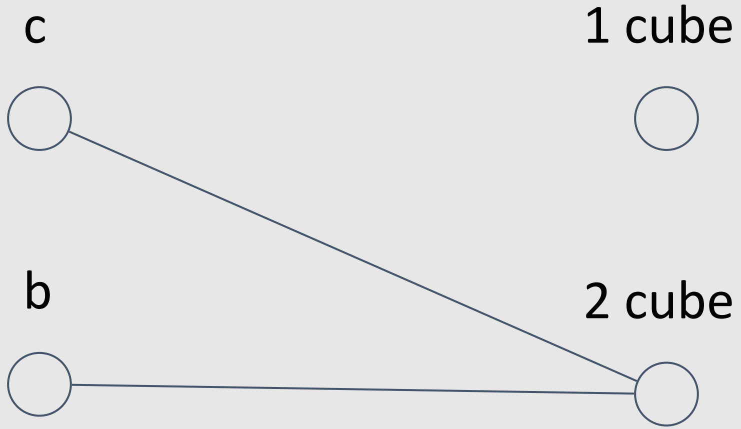
The second part will be the indices of the cubes.

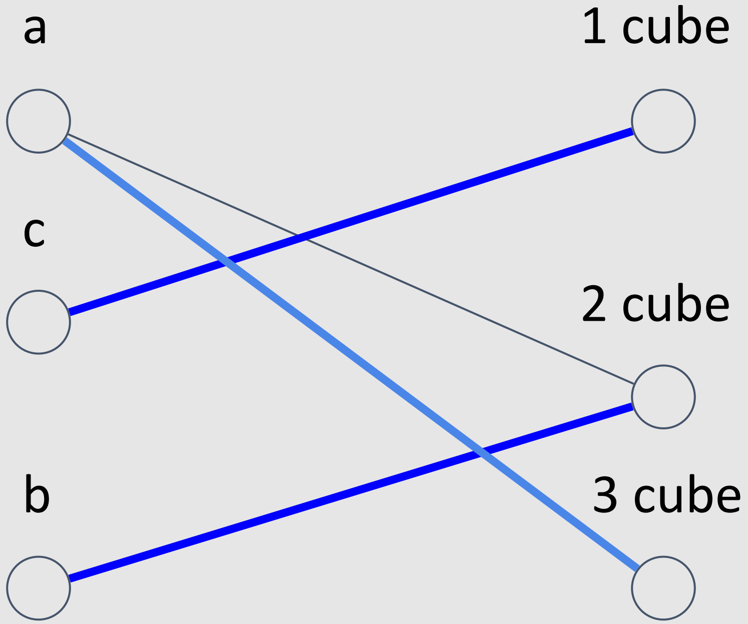
An edge (i, j) will indicate that the j-th cube can be used to display the i-th letter.


a
1 cube
b
2 cube

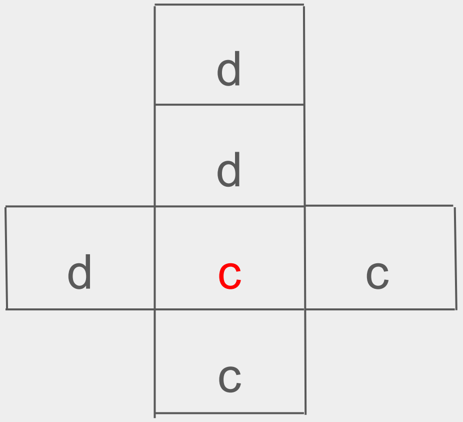

a
1 cube
b
2 cube

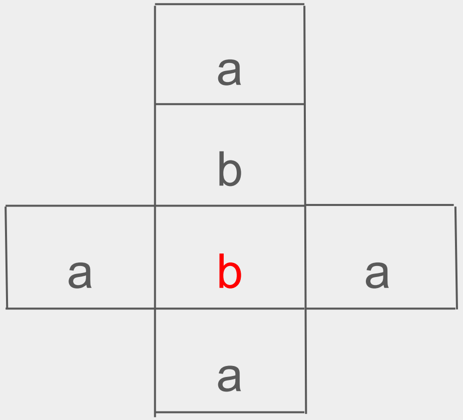

a
1 cube
b
2 cube

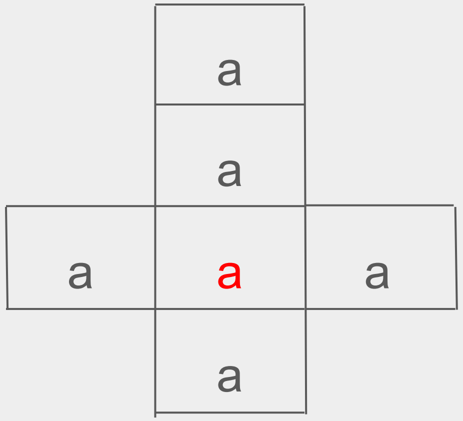

c
1 cube
b
2 cube


a
c
b
1 cube
2 cube
3 cube

acb?







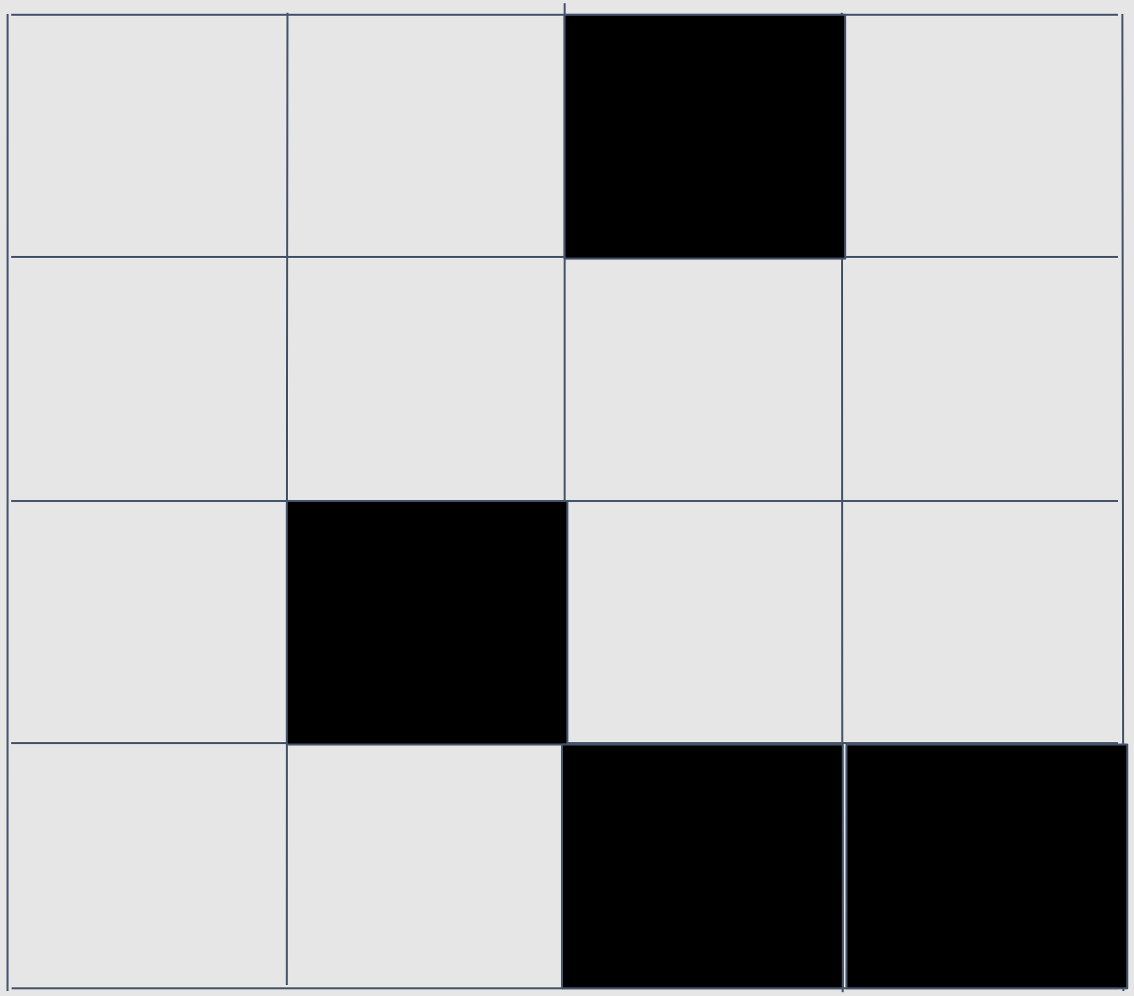
How to solve?

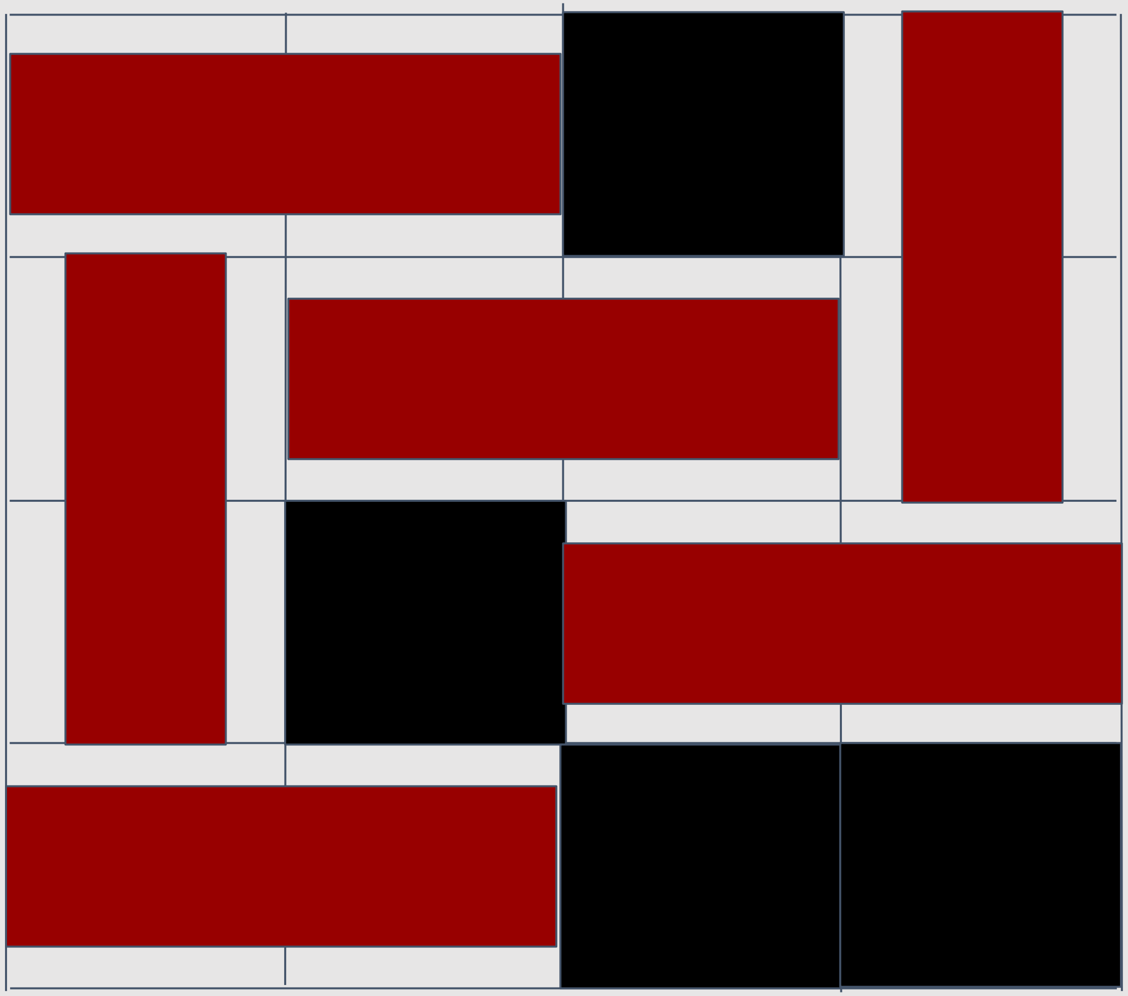
Then, in such a graph, a matching is exactly what we need.

Because it uniquely associates each cube with a letter of the string.

Dominoes

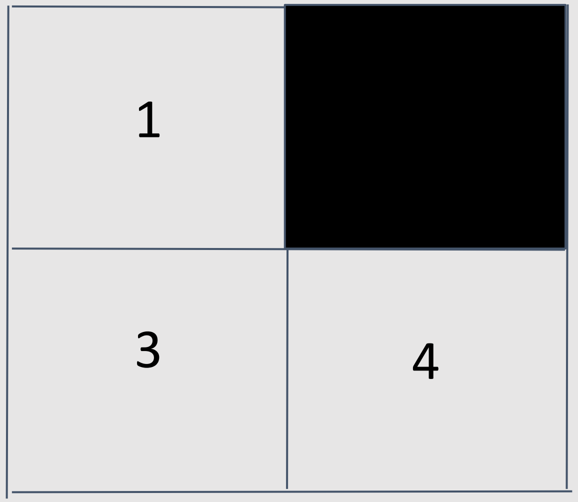
There is a rectangular field of size n×m that contains some punched-out cells. The task is to place as many dominoes (1×2 rectangles) on this field as possible, with the condition that nothing should be placed over the punched-out cells.

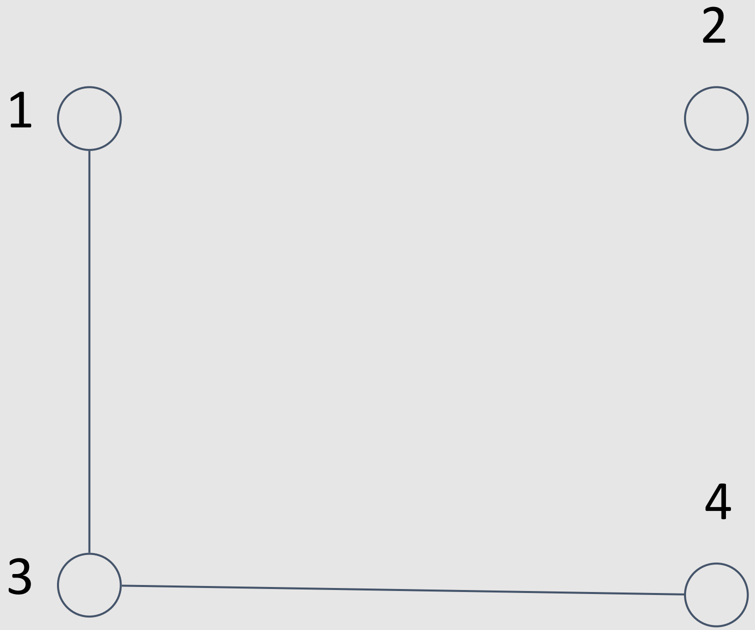




Dominoes

Let's construct a graph in which each cell is a vertex, and an edge between two vertices represents the possibility of placing a domino between these two cells.


1
3
4


1
2
3
4

Dominoes

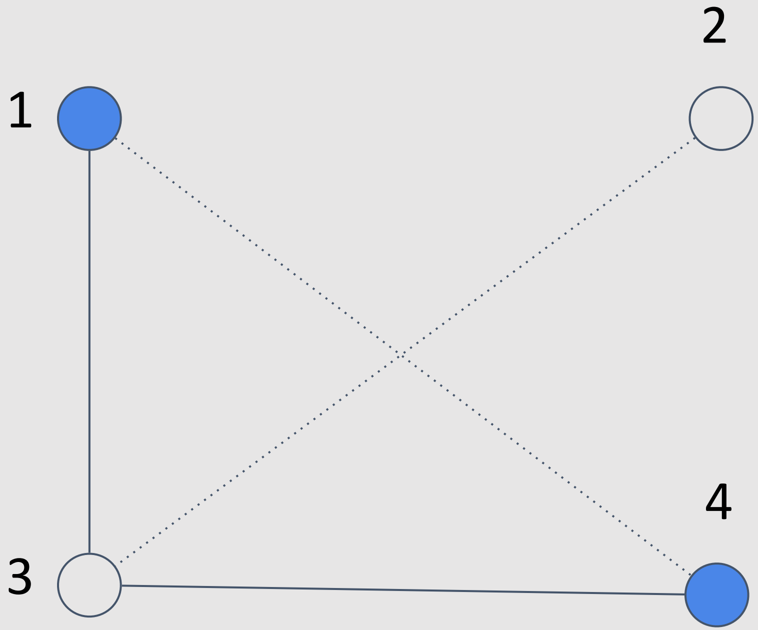
A matching in such a graph will answer the question of whether it is possible to tile the field with dominoes, because if we take an edge, it means these two cells are covered by one domino piece.

Dominoes

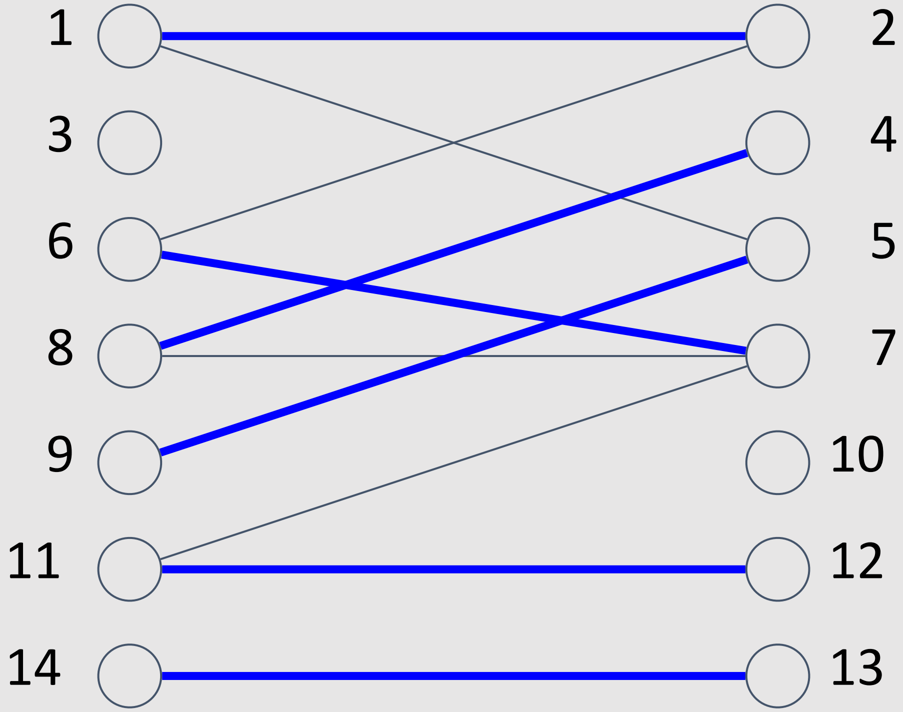
But why is this a bipartite graph?

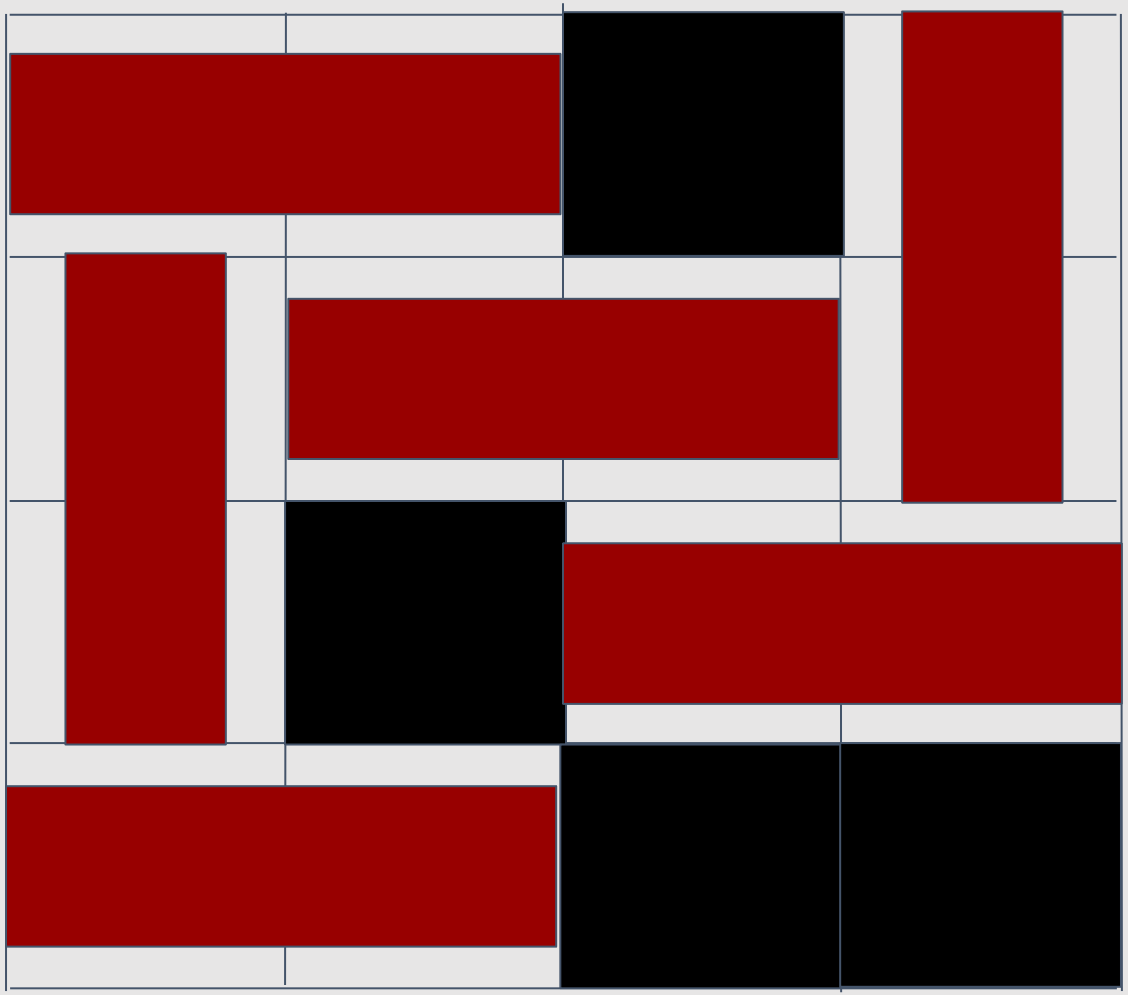
To prove that the graph is bipartite, we will show that we can explicitly divide the vertices into two sets.

The vertices of the first set are those with an even sum (row number plus column number). The second set comprises vertices with an odd sum. Edges can only go from vertices of the first type to the second, therefore the graph is bipartite.


1
2
3
4

1	2	■	4
		7	
9	■		12
	14	■	■


1
2
3
4
6
5
8
7
9
10
11
12
14
13



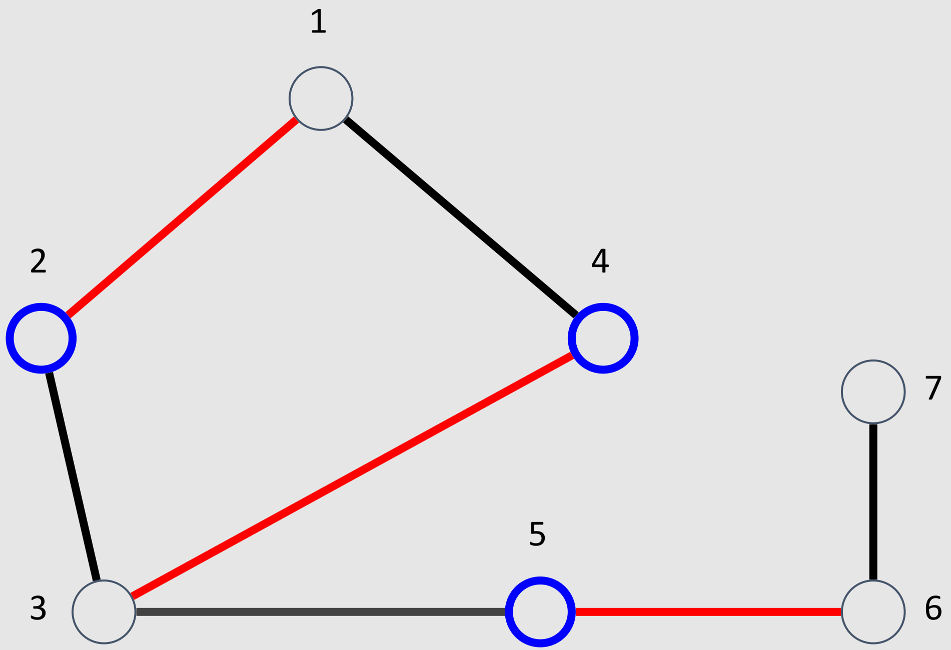
Minimum vertex cover

Let's define a vertex cover V as a set of vertices such that each edge of the graph is incident to at least one vertex from the set. It is necessary to find the vertex cover of the smallest size.

We solve only for bipartite graphs.

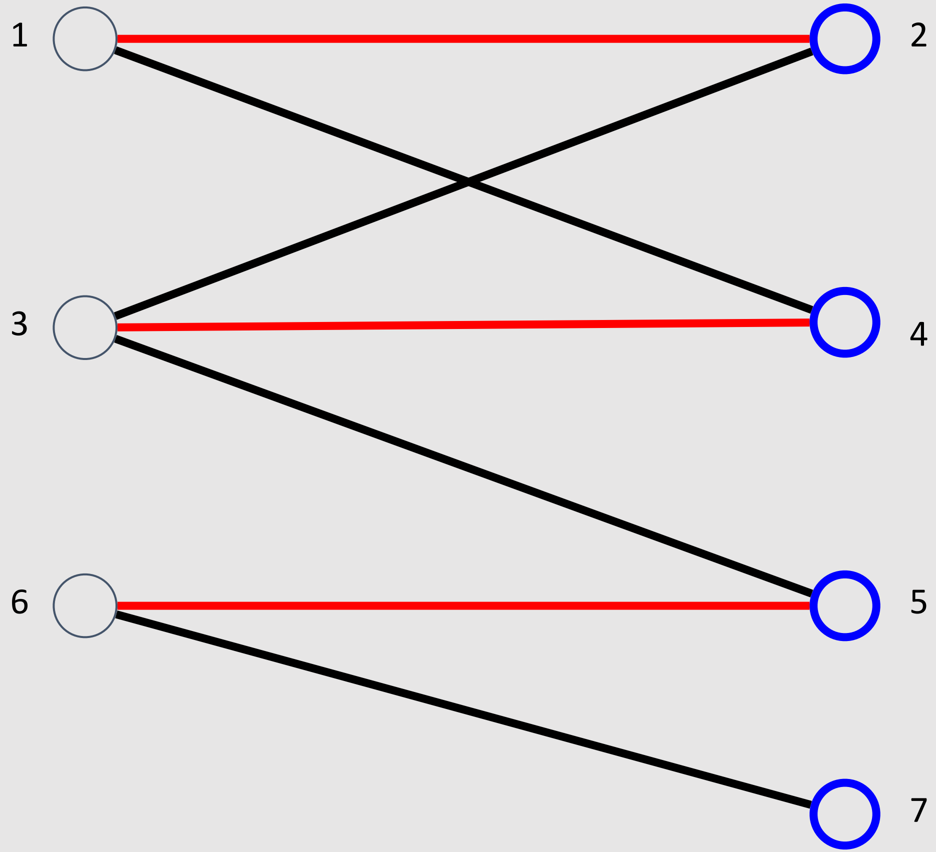
Minimum vertex cover

The size of |V_min| >= |M|, because M is a set of independent edges, and therefore, in any case, we need to take at least one endpoint of each of the edges.


1
2
3
4
5
6
7

Minimum vertex cover

To now prove that IV_minl equals M, let's figure out how to construct the Minimum Vertex Cover from a matching.

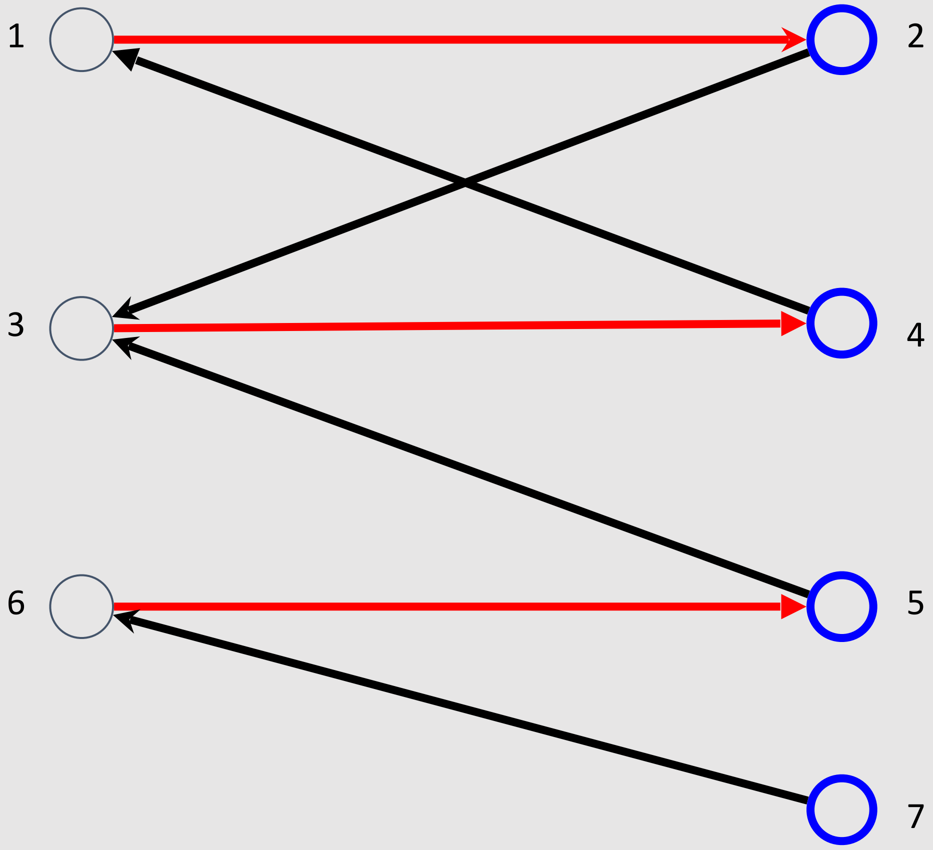

1
2
3
4
6
5
7

Minimum vertex cover

Let's orient the edges.

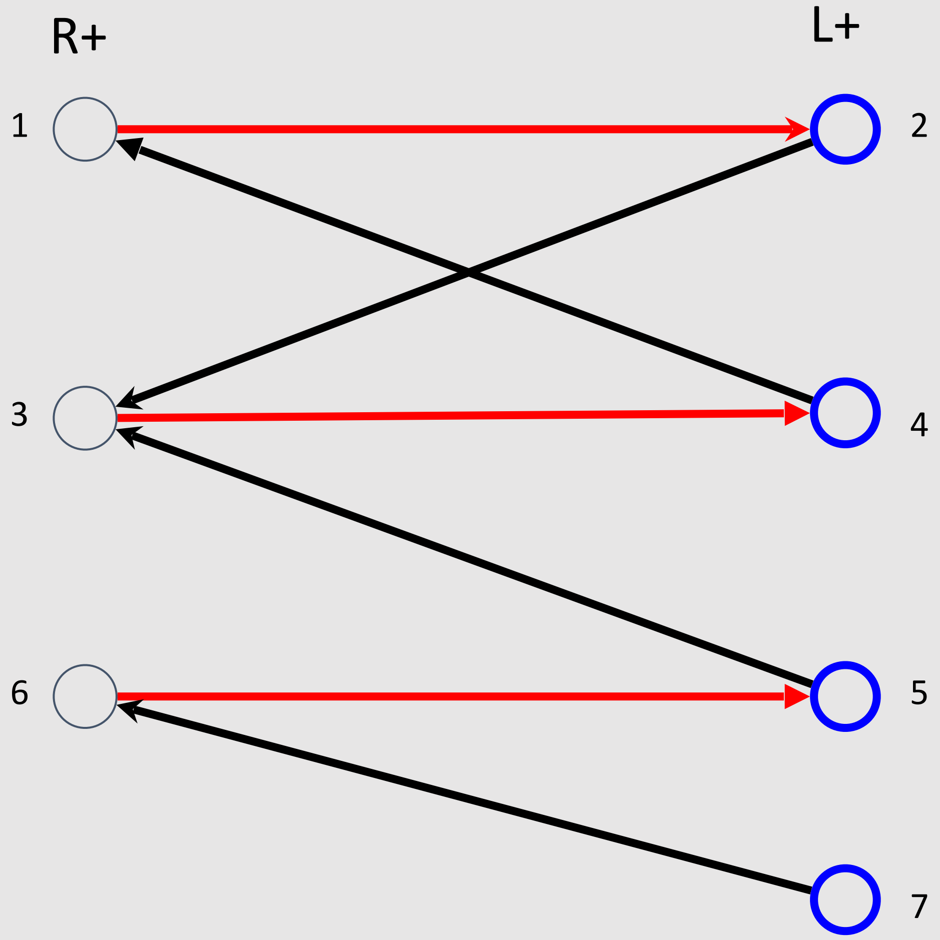
We will orient the edges of the matching from the right part to the left, and the non-matching edges from the left to the right.

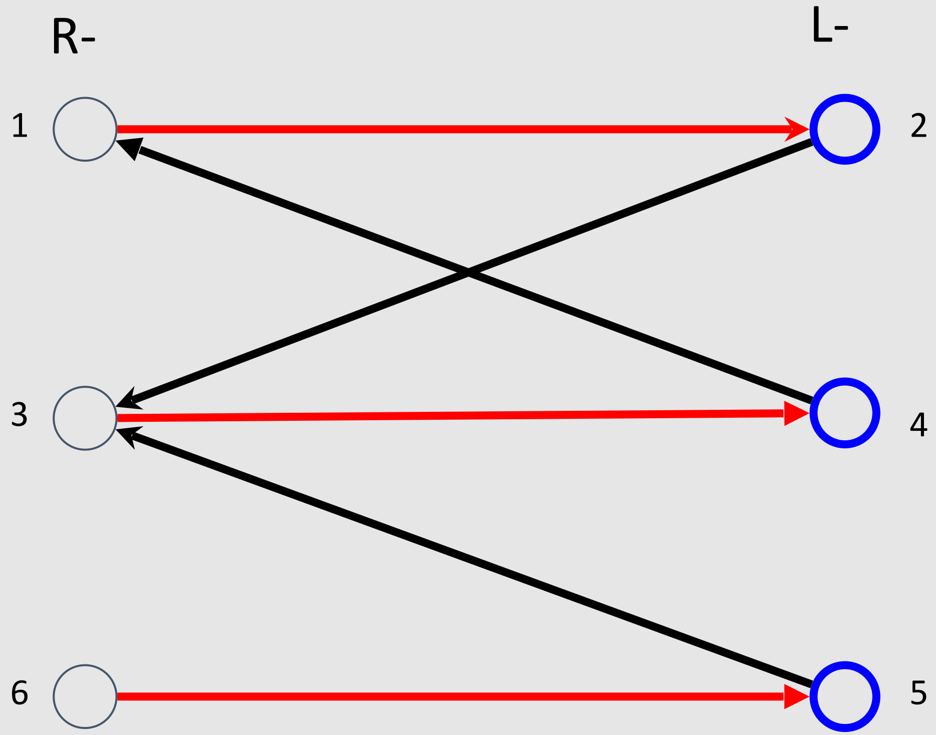
Now, let's run DFS from all vertices that are not Saturated.


1
2
3
4
6
5
7

Minimum vertex cover

Note that the graph has been divided into several sets: L+, R+, L-, R-, where the "plus" sets are the sets of vertices visited during the traversal.

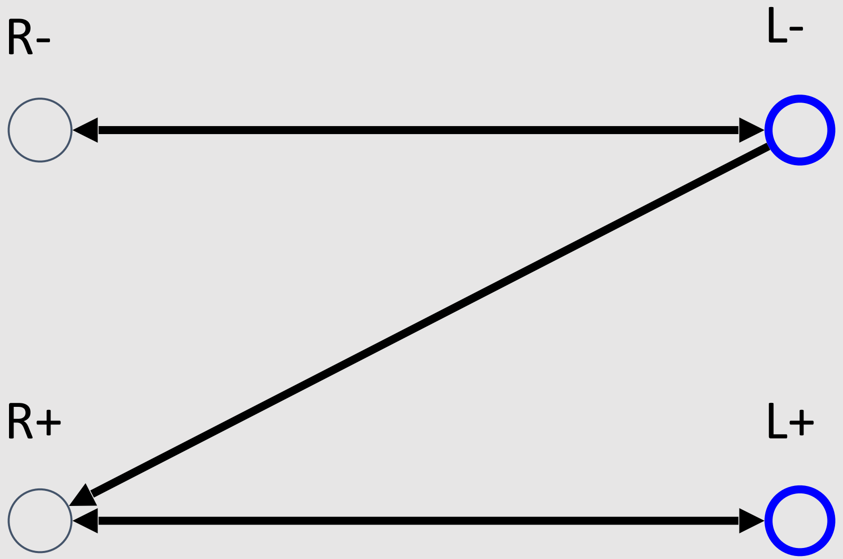

R+
L+
1
2
3
4
6
5
7


R-
L-
1
2
3
4
6
5

Minimum vertex cover

In a graph of this type, there are no edges from L+ to R- or from R+ to L-, because otherwise, they would have been traversed by DFS.

There are no edges from L- to R+ because then we would have had to include this edge in the matching, and therefore the matching would not be Minimum.


R-
L-
R+
L+

Minimum vertex cover

Take L- U R+ as the minimum vertex cover.

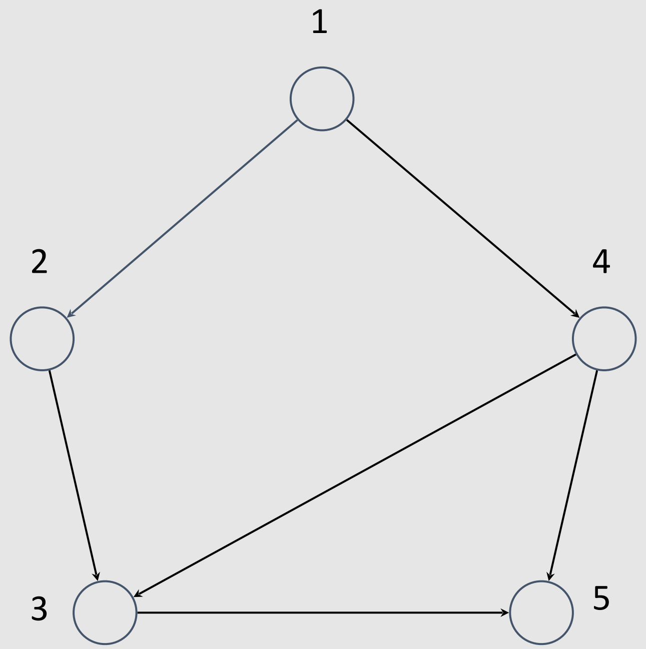
Since vertices in R+ cover vertices in R+ and L+.

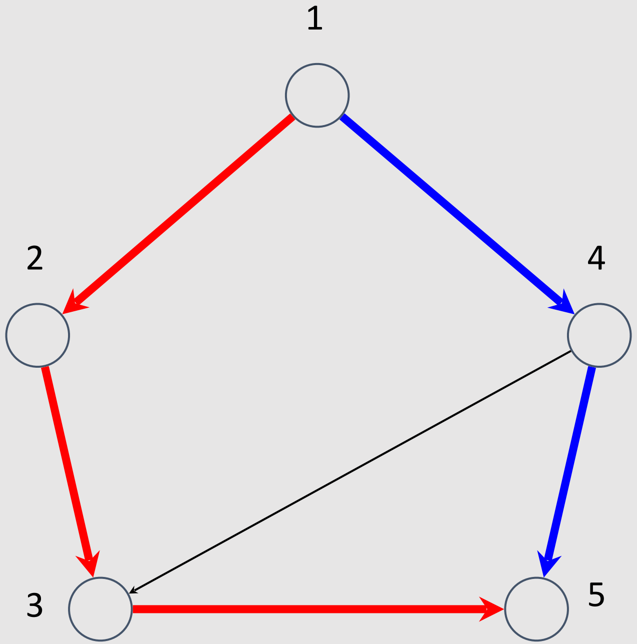
And vertices in L- cover vertices in R- and L-.

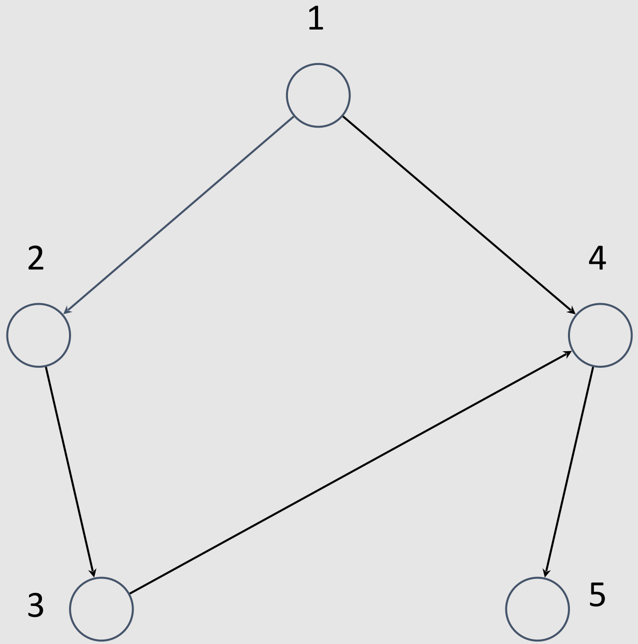
At the same time, edges from L- to R+ are impossible, so this is indeed the minimum vertex cover.

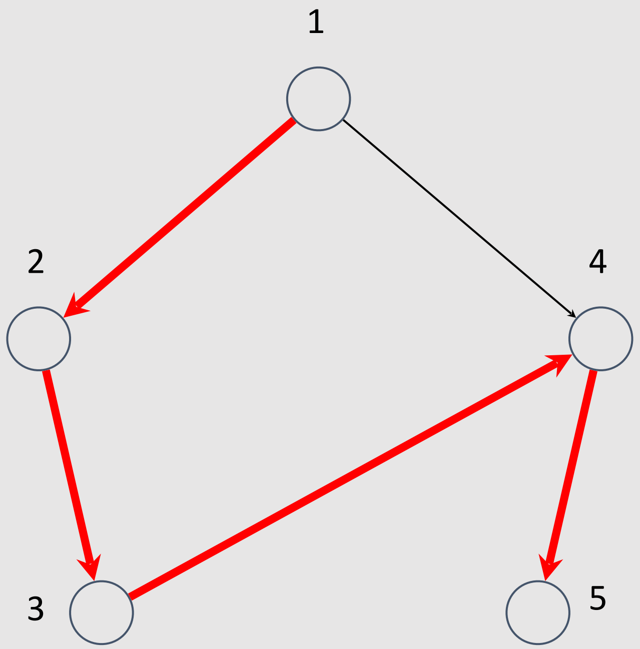
vertex-disjoint paths

There is a DAG (directed acyclic graph), and the task is to cover it with the minimum number of vertex-disjoint paths.


1
2
4
3
5

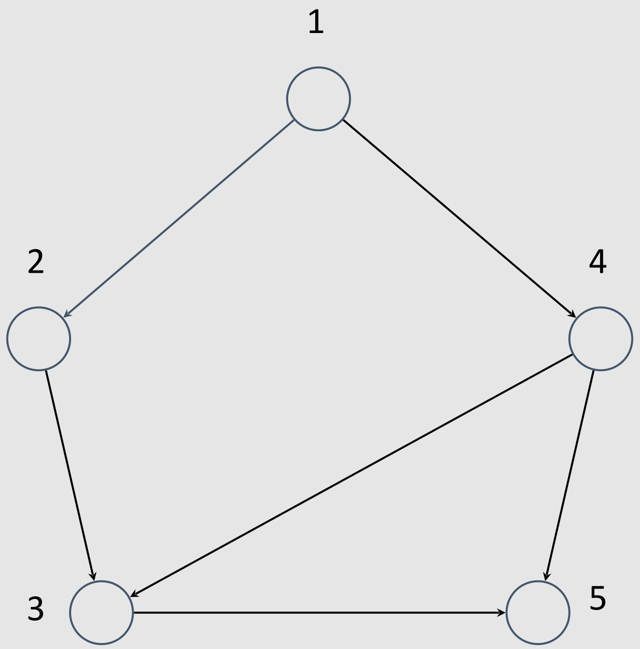

1
2
4
3
5

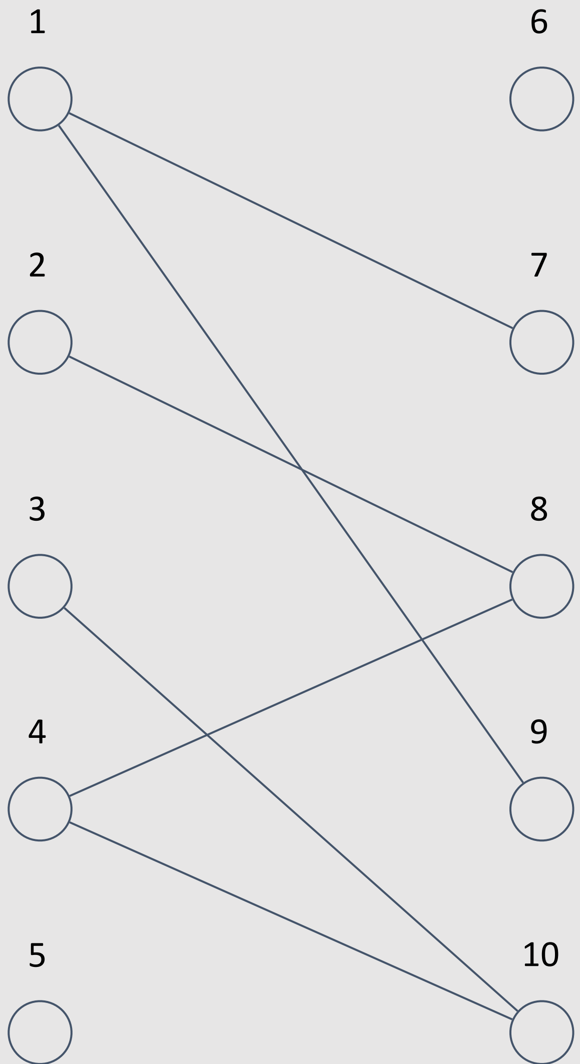

1
2
4
3
5


1
2
3
4
5

vertex-disjoint paths

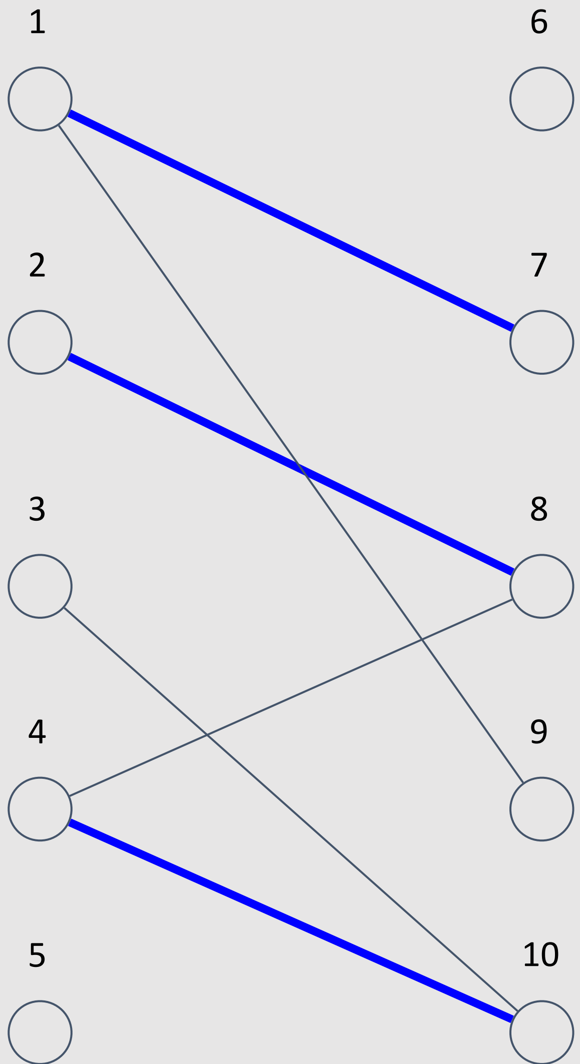
We will construct a bipartite graph where vertex i of the original graph turns into vertex i and i + n in the first and second parts, respectively. If there is an edge (i, j) in the original graph, then we will add an edge (i, j + n) in the bipartite graph.


1
2
4
3
5


1
2
3
4
5
6
7
8
9
10

vertex-disjoint paths

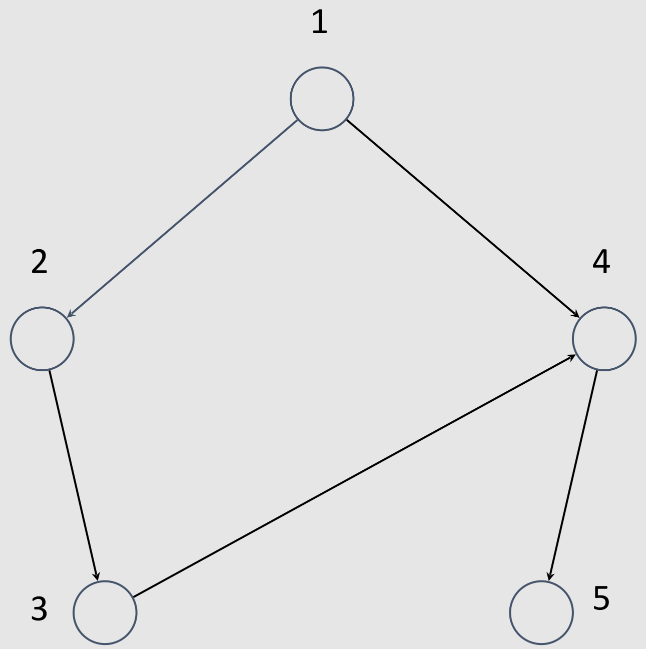
Let's take a close look at the matching in this graph.

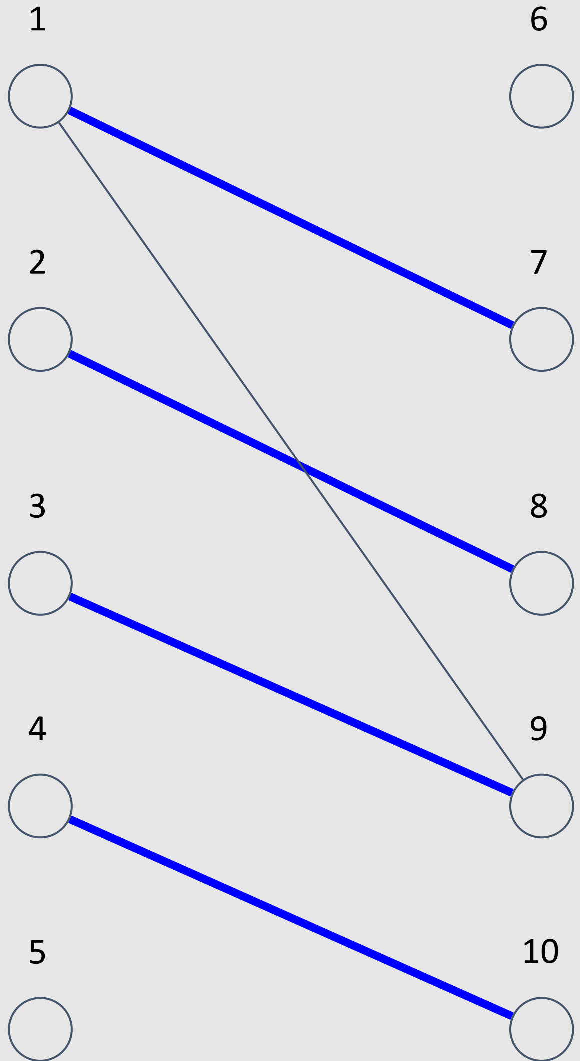

1
2
3
4
5
6
7
8
9
10

vertex-disjoint paths

Look, we have a matching of the form (1-2, 2-3, 4-5) and we also have paths 1-2-3, 4-5.

This can be explained as follows: if we have included an edge in the matching, it means that we can extend a path using it.


1
2
4
3
5


1
2
3
4
5
6
7
8
9
10

Task 1

Why doesn't the same matching search algorithm work in a non-bipartite graph?

Task 2

Why can't a matching be found greedily?

All codes

matching - https://ideone.com/quZuXs