Bipartite graphs

Bipartite check, MCBM, MVC, MIS

beOI Training

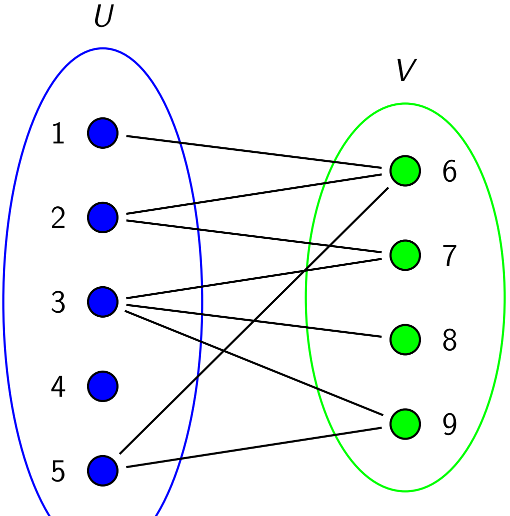
OLYMPIADE BELGE D'INFORMATIQUE
BELGISCHE INFORMATICA-OLYMPIADE

Reminder about graphs

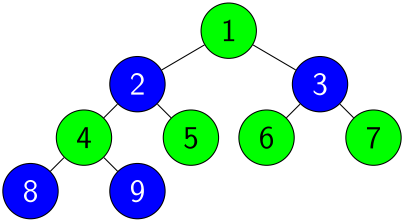
See 06-graph-basics

Bipartite graphs

A graph is bipartite if it can be separated into two sets U and V such that nodes in U are only connected to nodes in V, and conversely.



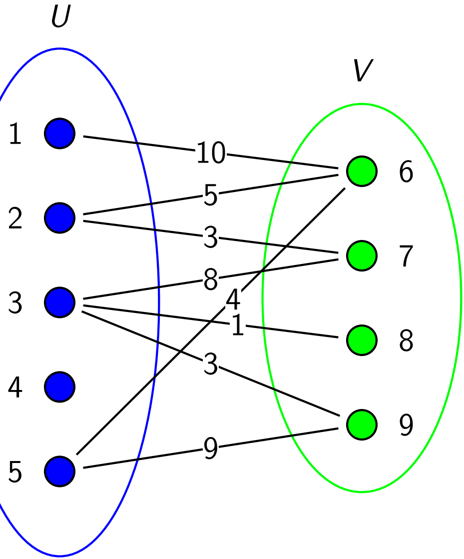
A tree is also a bipartite graph



Here $U = \{1, 4, 5, 6, 7\}$ and $V = \{2, 3, 8, 9\}$

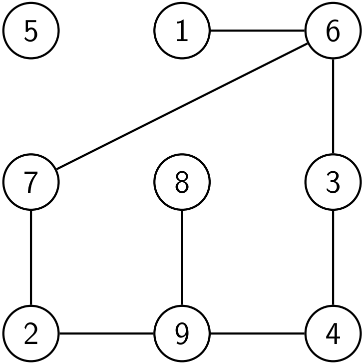
Weighted bipartite graphs

The same, but with weights:



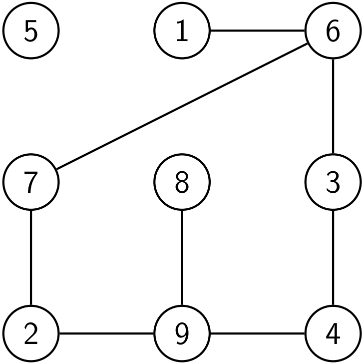
Bipartite check

How to test if a particular graph is a bipartite one?



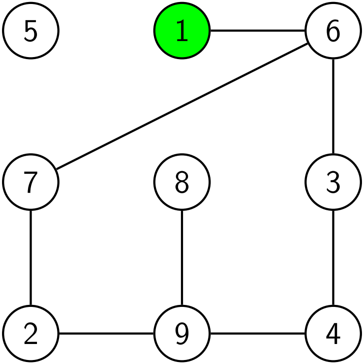
Bipartite check

How to test if a particular graph is a bipartite one?

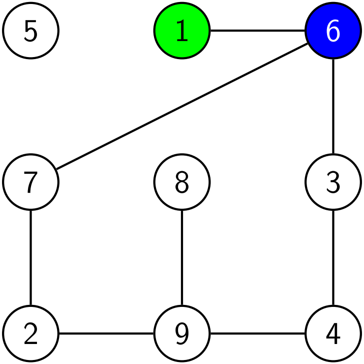


DFS with coloration!

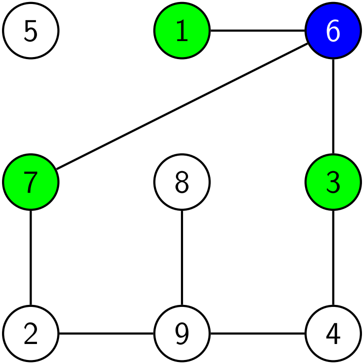
Bipartite check



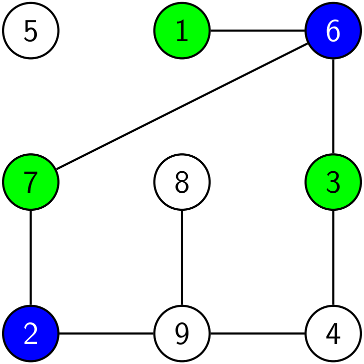
Bipartite check



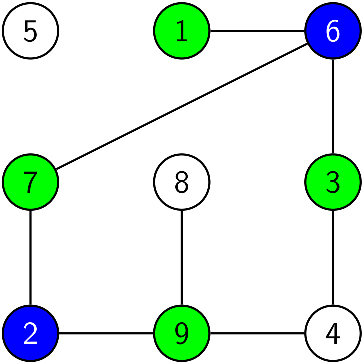
Bipartite check



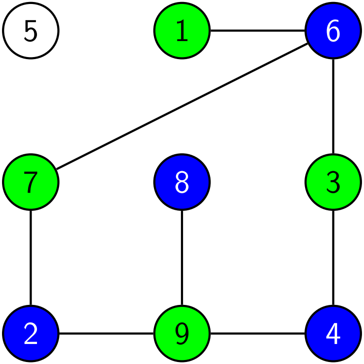
Bipartite check



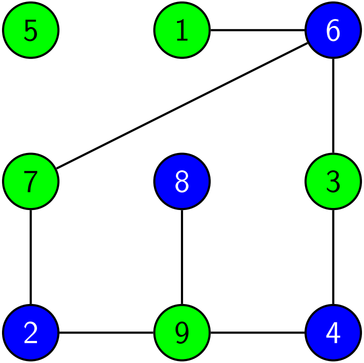
Bipartite check



Bipartite check



Bipartite check



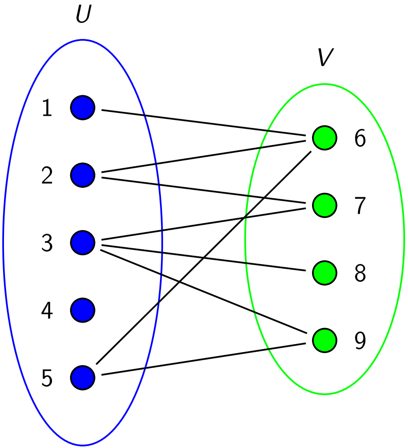
Ok!

Algorithm

```
bool visit(int, const vector<vector<int>>&, vector<int>&, int); // defined below

bool isBipartite(int n, const vector<vector<int>>& successors) {
    //successors[i] contains the successors of node i
    vector<int> color(n, -1); // initialized to -1

    for (int i = 0; i < n; i++) {
        if (color[i] == -1)
            if (!visit(i, successors, color, 1))
                return false;
    }
    return true;
}

bool visit(int node, const vector<vector<int>>& successors, vector<int>& color,
     int parentColor) {
    if (color[node] == parentColor) // failure
        return false;

    if (color[node] != -1) // avoid infinite looping
        return true;

    color[node] = (parentColor + 1) % 2;
    for (int next : successors[node])
        if (!visit(next, successors, node, color[node]))
            return false;
    return true;
}
```

Maximum Cardinality Bipartite Matching

Given a set U of men and a set V of women, and a list of "compatibilities" between men and women, we obtain this:



Can you create a maximum number of couples?

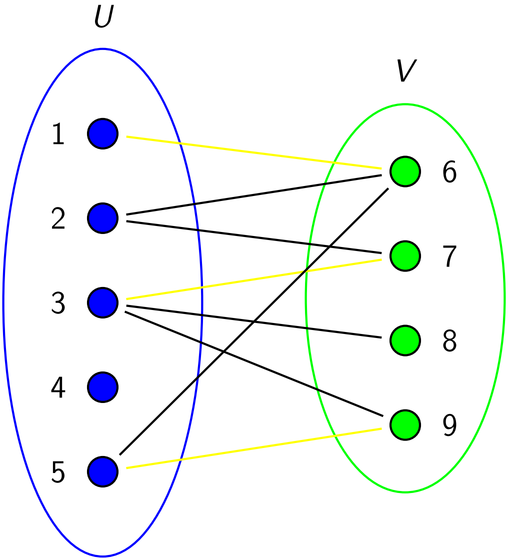
A bit of theory

$M \in E$ is a **matching** if each node is used at most once by the edges in M.

A **maximum cardinality matching** is a matching that has the maximal number of edge/node possible.

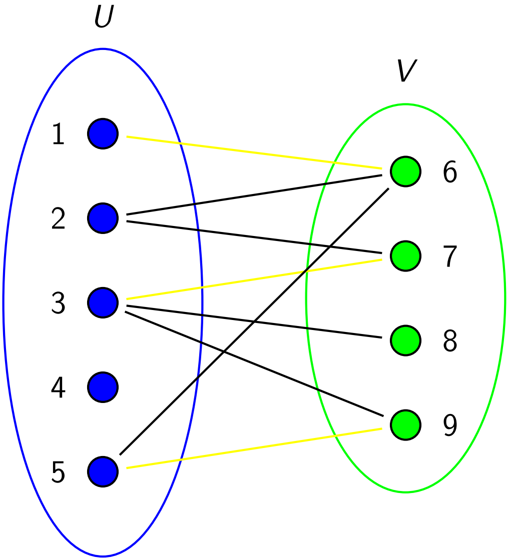
A node which is not used by any edge in a matching is said **free**. The others are said **non-free**.

An example of matching



It is maximal?

An example of matching

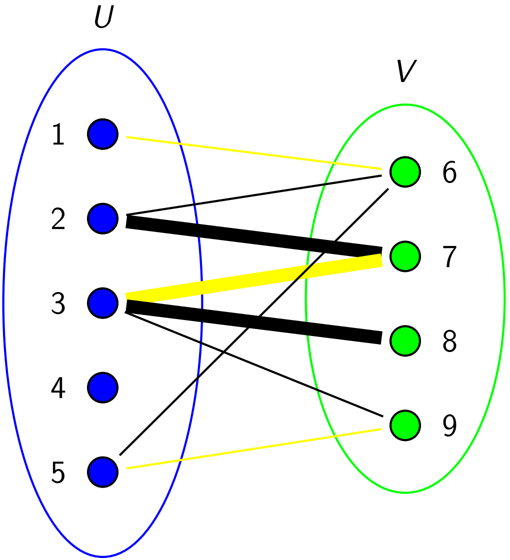


It is maximal? How to improve it?

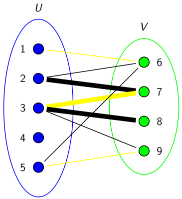
More theory

An augmenting path for a matching M is a path starting and ending at free nodes, and alternating between matched and unmatched edges.

An example of augmenting path

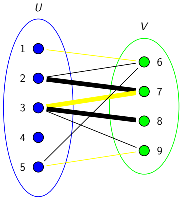


Let's think about augmenting paths



How can we use such paths? Let's find some useful properties

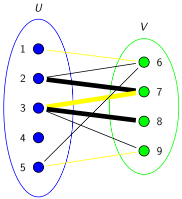
Let's think about augmenting paths



How can we use such paths? Let's find some useful properties

- By definition, an aug. path always starts and ends at a free node

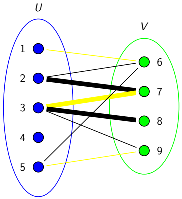
Let's think about augmenting paths



How can we use such paths? Let's find some useful properties

- By definition, an aug. path always starts and ends at a free node
- Thus, the first and last edge are not in M

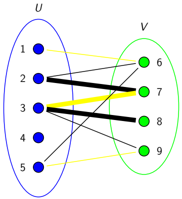
Let's think about augmenting paths



How can we use such paths? Let's find some useful properties

- By definition, an aug. path always starts and ends at a free node
- Thus, the first and last edge are not in M
- The size of an aug. path is always odd

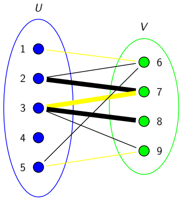
Let's think about augmenting paths



How can we use such paths? Let's find some useful properties

- By definition, an aug. path always starts and ends at a free node
- Thus, the first and last edge are not in M
- The size of an aug. path is always odd
- There is always one "free edge" more than the number of "taken edges" in an aug. path

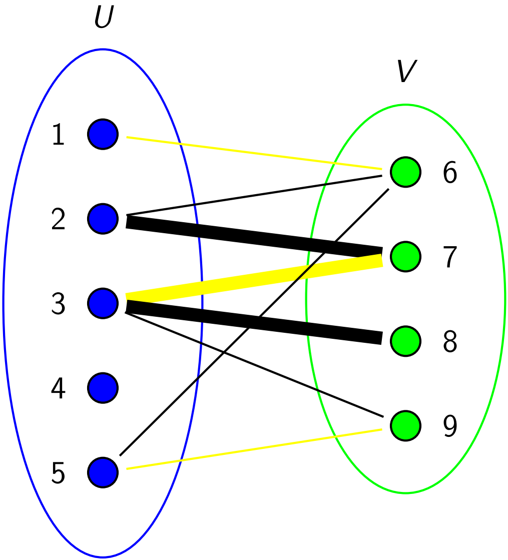
Let's think about augmenting paths



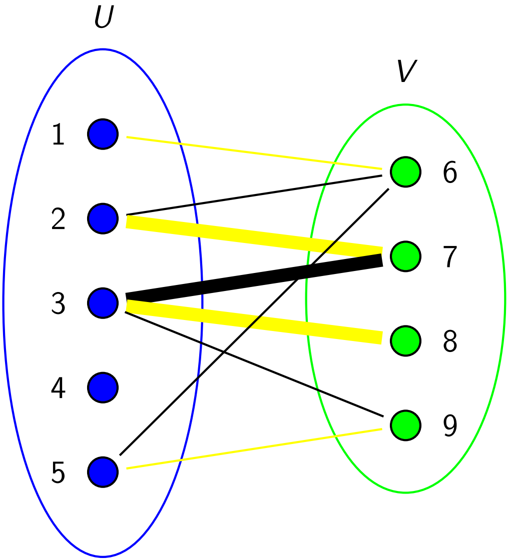
How can we use such paths? Let's find some useful properties

- By definition, an aug. path always starts and ends at a free node
- Thus, the first and last edge are not in M
- The size of an aug. path is always odd
- There is always one "free edge" more than the number of "taken edges" in an aug. path
- What if we inverse the edges?

An augmenting path



Inversing the augmenting path



Augmenting path = good?

- It is always possible to inverse an augmenting path
- It always increase the size of the matching by 1!

Augmenting path = good!

Given a matching M in a bipartite graph such that there exist no augmenting path, then M is of **maximal cardinality**.

Solving the MCBM

1. Find an aug. path. If no such path exists, return the matching, it is maximal
2. Inverse the path found
3. Repeat from 1

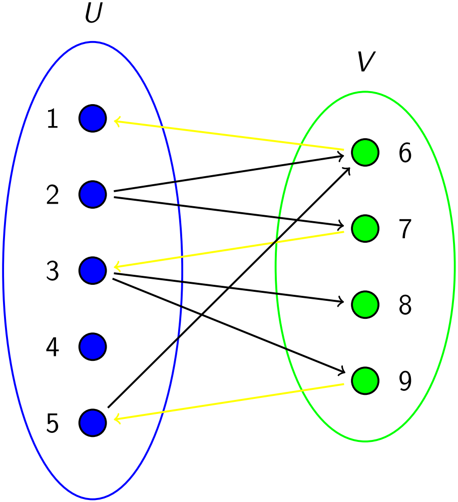
Simple, isn't it?

Let's see two algorithms

- The first one is based on the "flow" representation of the graph, and is very similar to a max flow. Simple to understand, but longer.
- The second is recursive so more difficult to understand but shorter.

Finding an aug. path

An alternative representation:



In this representation, an aug. path is a path starting at a free node in U and ending at a free node in V.

Finding an aug. path

```
// succ[i] contains the nodes that can be reached from i
// initially, all the edges are from U->V
// inU[i] is true iff i is in U
bool findAndReverse(int n, vector<vector<int>>& succ, vector<bool>& inU) {
    vector<int> pred(n);
    vector<bool> visited(n, false);
    stack<int> todo;

    // Find free nodes in U
    vector<bool> isFree(n, true);
    for (int i = 0; i < n; i++)
        if (!inU[i])
            for (int s : succ[i])
                    isFree[s] = false;

    for (int i = 0; i < n; i++) {
        if (inU[i] && isFree[i]) {
            todo.push(i);
            pred[i] = -1;
        }
    }
```

Finding an aug. path (cont.)

```
    // Run the DFS
    int found = -1;
    while(!todo.empty()) {
        int node = todo.top(); todo.pop();
        if(visited[node])
            continue;
        visited[node] = true;
        //if we are at a free node in V
        if(!inU[node] && succ[node].size() == 0) {
            found = node;
            break;
        }
        else {
            for(int next: successors[node]) {
                if(!visited[next]) {
                    pred[next] = node;
                    todo.push(next);
                }
            }
        }
    }

    // Reverse the nodes
    if(found != -1) {
        while(predecessors[found] != -1) {
            succ[pred[found]].erase(found);
            succ[found].push_back(pred[found]);
            found = pred[found];
        }
        return true;
    }
    return false;
}
```

Final algorithm

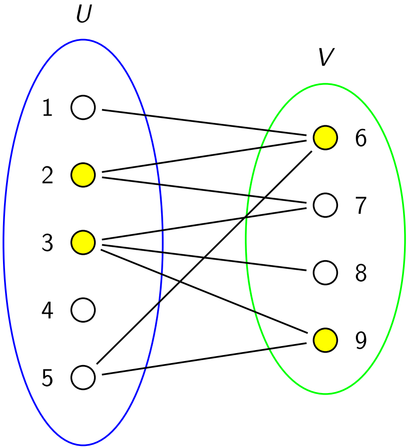
```
void getMCBM(int n, vector<vector<int>>& succ, vector<bool>& inU) {
    while(findAndReverse(n, succ, inU)) {}
    //MCBM == edges from nodes in V (in succ)
}
```

Another (shorter) algorithm

```
vector<vi> AdjList;
vi match, vis;

int Aug(int l) { // return 1 if an augmenting path is found
    if (vis[l]) return 0; // return 0 otherwise
    vis[l] = 1;
    for (int j = 0; j < (int)AdjList[l].size(); j++) {
        int r = AdjList[l][j];
        if (match[r] == -1 || Aug(match[r])) {
            match[r] = l;
            return 1; // found 1 matching
        }
    }
    return 0; // no matching
}

int MCBM = 0;
match.assign(V, -1); //V = total number of vertices
for(int l = 0; l < n; l++) { //n = size of the left set
    vis.assign(n, 0);
    MCBM += Aug(l);
}
```

Minimum vertex cover (in bipartite graph)

A **Vertex cover** K is a set of nodes from G such that each edge of G is incident to at least one node of K.
A **Minimal vertex cover** is a vertex cover of minimal size.



Minimum vertex cover

Given a maximum matching M

Minimum vertex cover

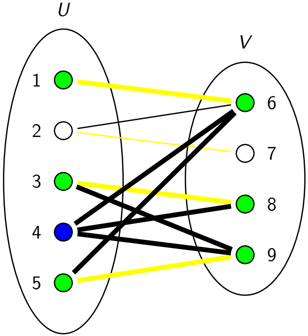
Given a maximum matching M, if we construct a minimum vertex cover of size $|M|$, it must be minimal.

Minimum vertex cover

Given a maximum matching M, if we construct a minimum vertex cover of size $|M|$, it must be minimal.
Let U_f be the set of free nodes in U, and let Z be the set of vertices in U_f or connected to U_f using alternating paths.

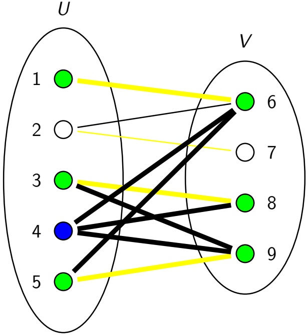
Minimum vertex cover

Given a maximum matching M, if we construct a minimum vertex cover of size $|M|$, it must be minimal.
Let U_f be the set of free nodes in U, and let Z be the set of vertices in U_f or connected to U_f using alternating paths.



Minimum vertex cover

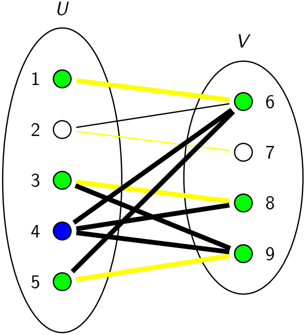
Given a maximum matching M, if we construct a minimum vertex cover of size $|M|$, it must be minimal.
Let U_f be the set of free nodes in U, and let Z be the set of vertices in U_f or connected to U_f using alternating paths.

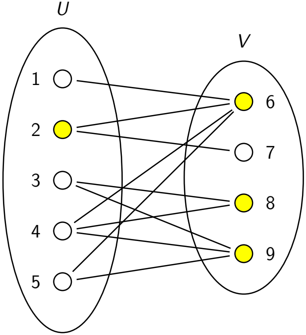


Then $K = (U \backslash Z) \cup (V \cap Z)$ is a minimum vertex cover

Minimum vertex cover

Given a maximum matching M, if we construct a minimum vertex cover of size $|M|$, it must be minimal.
Let U_f be the set of free nodes in U, and let Z be the set of vertices in U_f or connected to U_f using alternating paths.





Then $K = (U \setminus Z) \cup (V \cap Z)$ is a minimum vertex cover

Maximum independant set (in bipartite graph)

An **independant set** of a bipartite graph $G = < V, E >$ is a set of nodes MIS such that there is no edges between nodes in MIS.

Said in another way, $\nexists v_1 \in MIS, v_2 \in MIS$ s.t. $(v_1, v_2) \in E$.

A **maximum independant set** is an independant set whose size is maximal (...)

Maximum independant set (2)

Given a Minimum vertex cover K on a graph with the set of nodes $S = U + V$, then

$$MIS = S - K$$

is a maximum independant set.