Today we're going to cover

- Minimum spanning tree
- Shortest paths
- Some known graph problems
- Special graphs
  - Trees
  - Directed acyclic graphs
  - Bipartite graphs

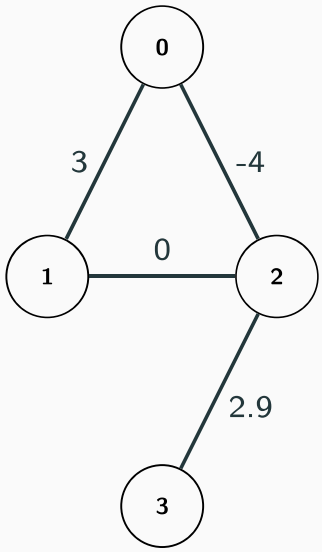
Weighted graphs

- Now the edges in our graphs may have weights, which could represent
  - the distance of the road represented by the edge
  - the cost of going over the edge
  - some capacity of the edge
- We can use a modified adjacency list to represent weighted graphs

Weighted graphs

```
struct edge {
    int u, v;
    int weight;

    edge(int _u, int _v, int _w) {
        u = _u;
        v = _v;
        weight = _w;
    }
};
```



Weighted graphs

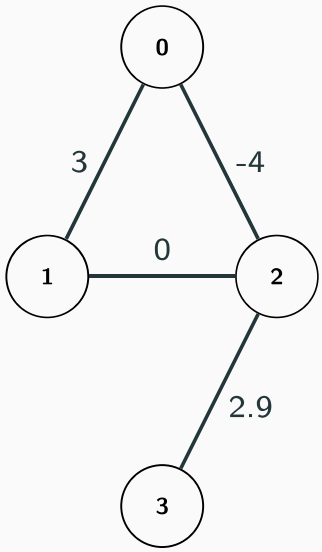
```
vector<edge> adj[4];

adj[0].push_back(edge(0, 1, 3));
adj[0].push_back(edge(0, 2, -4));

adj[1].push_back(edge(1, 0, 3));
adj[1].push_back(edge(1, 2, 0));

adj[2].push_back(edge(2, 0, -4));
adj[2].push_back(edge(2, 1, 0));
adj[2].push_back(edge(2, 3, 2.9));

adj[3].push_back(edge(3, 2, 2.9));
```



Minimum spanning tree

- We have an undirected weighted graph
- The vertices along with a subset of the edges in the graph is called a spanning tree if
  - it forms a tree (i.e. does not contain a cycle) and
  - the tree spans all vertices (all vertices can reach all other vertices)
- The weight of a spanning tree is the sum of the weights of the edges in the subset
- We want to find a minimum spanning tree

Minimum spanning tree

- Several greedy algorithms work

- Go through the edges in the graph in increasing order of weight
- Greedily pick an edge if it doesn't form a cycle (Union-Find can be used to keep track of when we would get a cycle)
- When we've gone through all edges, we have a minimum spanning tree

- This is Kruskal's algorithm
- Time complexity is $O(E \log E)$

Minimum spanning tree

```
bool edge_cmp(const edge &a, const edge &b) {
    return a.weight < b.weight;
}

vector<edge> mst(int n, vector<edge> edges) {
    union_find uf(n);
    sort(edges.begin(), edges.end(), edge_cmp);

    vector<edge> res;
    for (int i = 0; i < edges.size(); i++) {
        int u = edges[i].u,
            v = edges[i].v;

        if (uf.find(u) != uf.find(v)) {
            uf.unite(u, v);
            res.push_back(edges[i]);
        }
    }

    return res;
}
```

Shortest paths

- We have a weighted graph (undirected or directed)
- Given two vertices u, v, what is the shortest path from u to v?

- If all weights are the same, this can be solved with breadth-first search
- Of course, this is usually not the case...

Shortest paths

- There are many known algorithms to find shortest paths
- Like breadth-first search, these algorithms usually find the shortest paths from a given start vertex to all other vertices
- Let's take a quick look at Dijkstra's algorithm, the Bellman-Ford algorithm and the Floyd-Warshall algorithm

Dijkstra's algorithm

```
vector<edge> adj[100];
vector<int> dist(100, INF);

void dijkstra(int start) {
    dist[start] = 0;
    priority_queue<pair<int, int>,
                   vector<pair<int, int> >,
                   greater<pair<int, int> > > pq;
    pq.push(make_pair(dist[start], start));

    while (!pq.empty()) {
        int u = pq.top().second,
            d = pq.top().first;
        pq.pop();
        if (d > dist[u]) continue;
        for (int i = 0; i < adj[u].size(); i++) {
            int v = adj[u][i].v,
                w = adj[u][i].weight;
            if (w + dist[u] < dist[v]) {
                dist[v] = w + dist[u];
                pq.push(make_pair(dist[v], v));
            }
        }
    }
}
```

Dijkstra's algorithm

- Time complexity is $O(V \log E)$
- Note that this only works for non-negative weights

Bellman-Ford algorithm

```
void bellman_ford(int n, int start) {

    dist[start] = 0;

    for (int i = 0; i < n - 1; i++) {
        for (int u = 0; u < n; u++) {
            for (int j = 0; j < adj[u].size(); j++) {
                int v = adj[u][j].v;
                int w = adj[u][j].weight;
                dist[v] = min(dist[v], w + dist[u]);
            }
        }
    }
}
```

Bellman-Ford algorithm

- Time complexity is $O(V \times E)$
- Can be used to detect negative-weight cycles

Floyd-Warshall algorithm

- What about using dynamic programming to compute shortest paths?
- Let $\mathrm{sp}(k, i, j)$ be the shortest path from i to j if we're only allowed to travel through the vertices $0, \ldots, k$
- Base case: $\mathrm{sp}(k, i, j) = 0$ if $i = j$
- Base case: $\mathrm{sp}(-1, i, j) = \mathrm{weight}[a][b]$ if $(i, j) \in E$
- Base case: $\mathrm{sp}(-1, i, j) = \infty$
- $\mathrm{sp}(k, i, j) = \min \begin{cases} \mathrm{sp}(k-1, i, k) + \mathrm{sp}(k-1, k, j) \\ \mathrm{sp}(k-1, i, j) \end{cases}$

Floyd-Warshall algorithm

```
int dist[1000][1000];
int weight[1000][1000];

void floyd_warshall(int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            dist[i][j] = i == j ? 0 : weight[i][j];
        }
    }

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

Floyd-Warshall algorithm

- Computes all-pairs shortest paths
- Time complexity is clearly $O(n^3)$
- Very simple to code

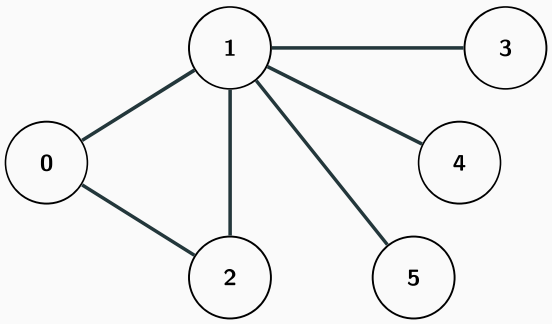
Known graph problems

- The problems we're dealing with very often ask us to solve some well known graph problem
- But usually it's well hidden in the problem statement

- Let's take a look at a few examples

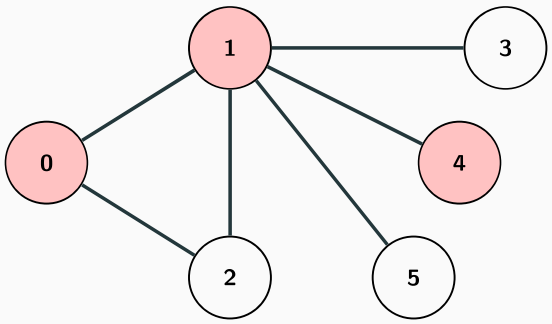
Minimum vertex cover

- We have an unweighted undirected graph
- A vertex cover is a subset of the vertices S, such that for each edge (u, v) in the graph, either u or v (or both) are in S



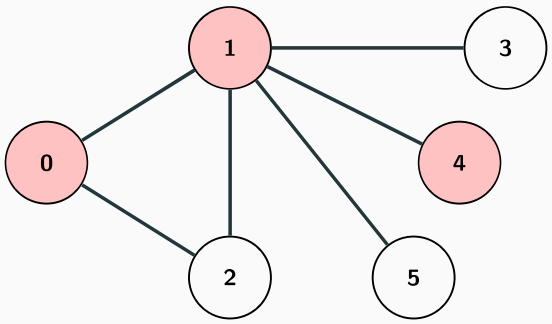
Minimum vertex cover

- We have an unweighted undirected graph
- A vertex cover is a subset of the vertices S, such that for each edge (u, v) in the graph, either u or v (or both) are in S



Minimum vertex cover

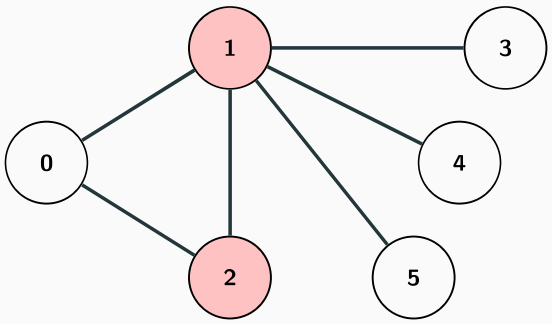
- We have an unweighted undirected graph
- A vertex cover is a subset of the vertices S, such that for each edge (u, v) in the graph, either u or v (or both) are in S



- We want to find a vertex cover of minimum size

Minimum vertex cover

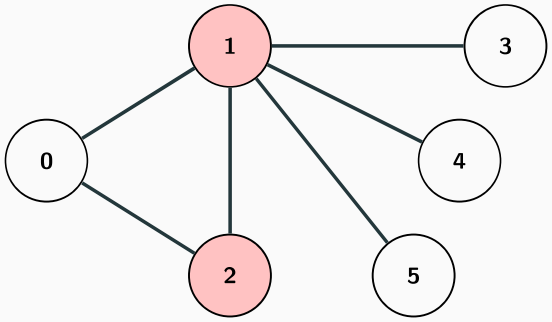
- We have an unweighted undirected graph
- A vertex cover is a subset of the vertices S, such that for each edge (u, v) in the graph, either u or v (or both) are in S



- We want to find a vertex cover of minimum size

Minimum vertex cover

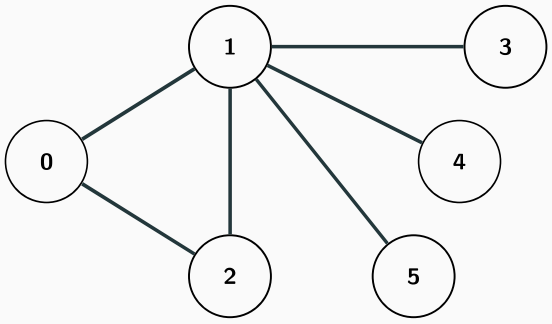
- We have an unweighted undirected graph
- A vertex cover is a subset of the vertices S, such that for each edge (u, v) in the graph, either u or v (or both) are in S



- We want to find a vertex cover of minimum size
- NP-hard problem in general graphs

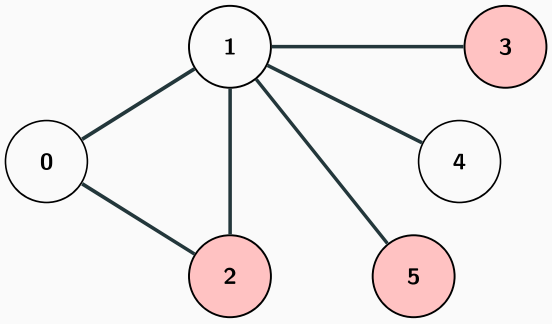
Maximum independent set

- We have an unweighted undirected graph
- An independent set is a subset of the vertices S, such that no two vertices u, v in S are adjacent in the graph



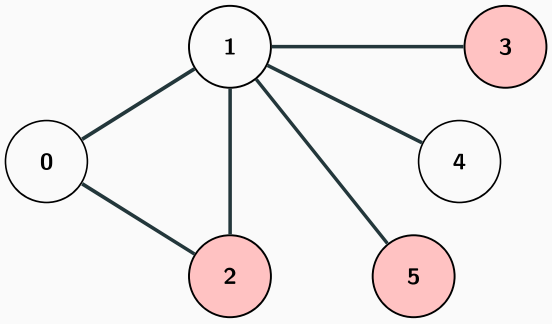
Maximum independent set

- We have an unweighted undirected graph
- An independent set is a subset of the vertices S, such that no two vertices u, v in S are adjacent in the graph



Maximum independent set

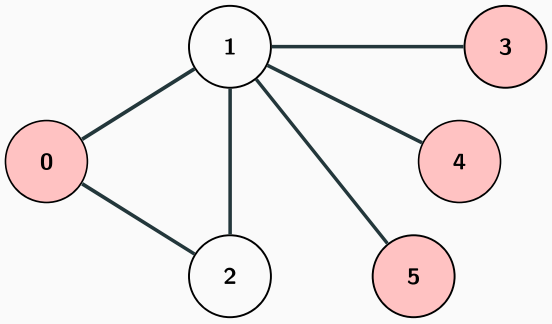
- We have an unweighted undirected graph
- An independent set is a subset of the vertices S, such that no two vertices u, v in S are adjacent in the graph



- We want to find an independent set of maximum size

Maximum independent set

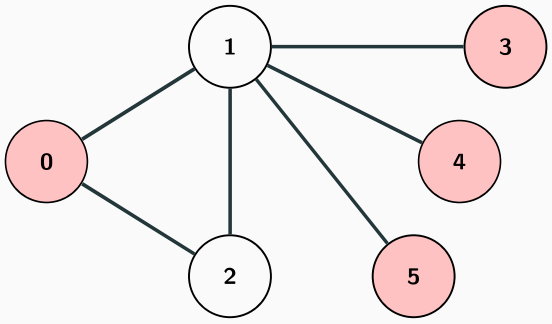
- We have an unweighted undirected graph
- An independent set is a subset of the vertices S, such that no two vertices u, v in S are adjacent in the graph



- We want to find an independent set of maximum size

Maximum independent set

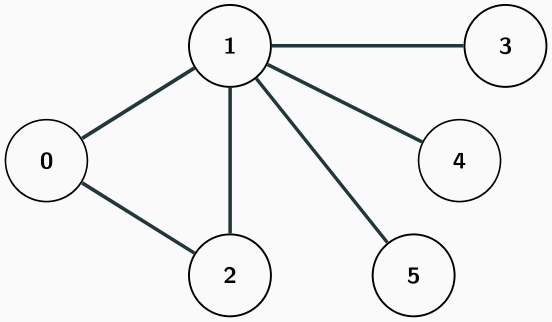
- We have an unweighted undirected graph
- An independent set is a subset of the vertices S, such that no two vertices u, v in S are adjacent in the graph



- We want to find an independent set of maximum size
- NP-hard problem in general graphs

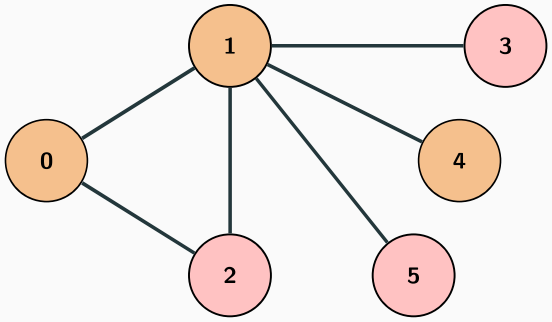
Relation between MVC and MIS

- The previous two problems are very related
- A subset of the vertices is a vertex cover if and only if the complement of the set is an independent set



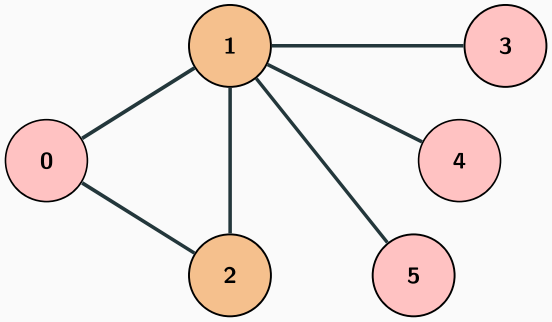
Relation between MVC and MIS

- The previous two problems are very related
- A subset of the vertices is a vertex cover if and only if the complement of the set is an independent set



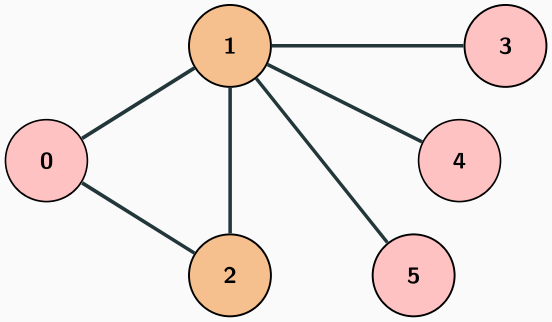
Relation between MVC and MIS

- The previous two problems are very related
- A subset of the vertices is a vertex cover if and only if the complement of the set is an independent set



Relation between MVC and MIS

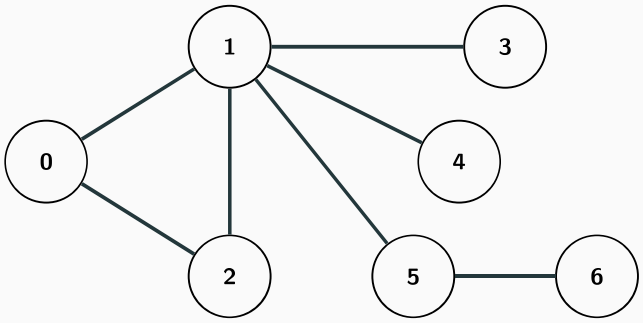
- The previous two problems are very related
- A subset of the vertices is a vertex cover if and only if the complement of the set is an independent set



- The size of a minimum vertex cover plus the size of a maximum independent set is equal to the number of vertices

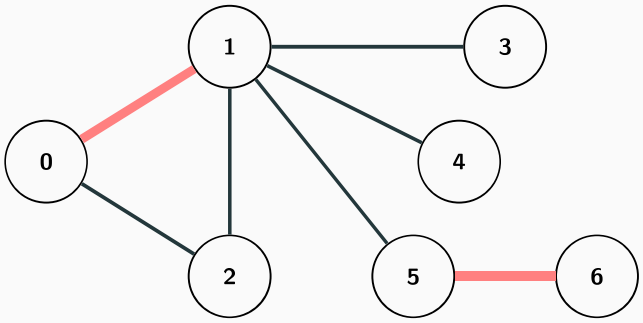
Maximum matching

- We have an unweighted undirected graph
- A matching is a subset of the edges such that each vertex is adjacent to at most one edge in the subset



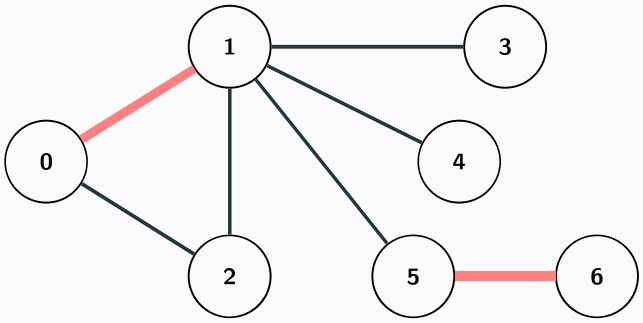
Maximum matching

- We have an unweighted undirected graph
- A matching is a subset of the edges such that each vertex is adjacent to at most one edge in the subset



Maximum matching

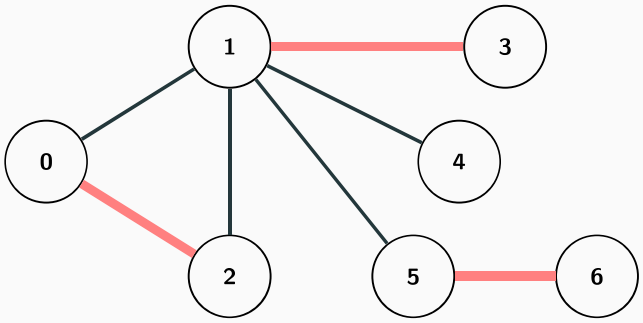
- We have an unweighted undirected graph
- A matching is a subset of the edges such that each vertex is adjacent to at most one edge in the subset



- We want to find a matching of maximum size

Maximum matching

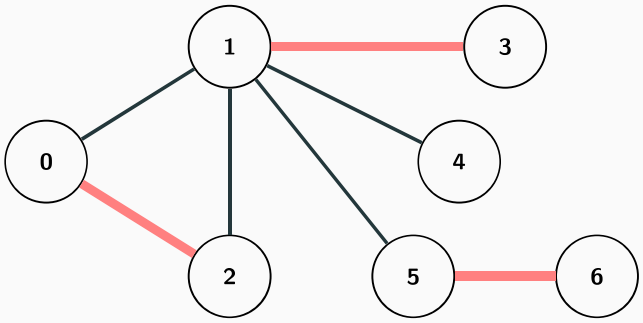
- We have an unweighted undirected graph
- A matching is a subset of the edges such that each vertex is adjacent to at most one edge in the subset



- We want to find a matching of maximum size

Maximum matching

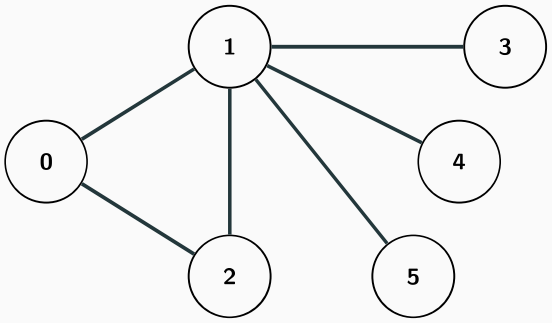
- We have an unweighted undirected graph
- A matching is a subset of the edges such that each vertex is adjacent to at most one edge in the subset



- We want to find a matching of maximum size
- There exists an $O(V^4)$ algorithm for general graphs, but is pretty complex

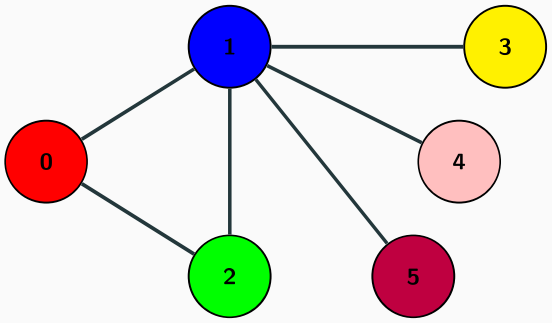
Graph coloring

- We have an unweighted undirected graph
- A coloring of the graph is an assignment of colors to the vertices such that adjacent vertices have different colors



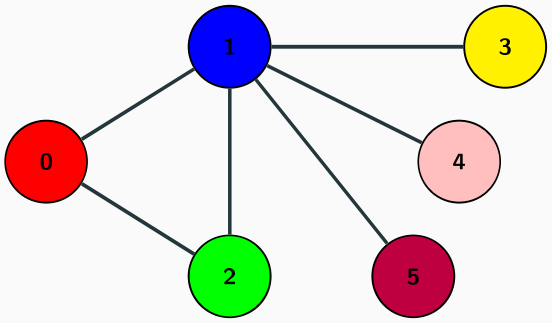
Graph coloring

- We have an unweighted undirected graph
- A coloring of the graph is an assignment of colors to the vertices such that adjacent vertices have different colors



Graph coloring

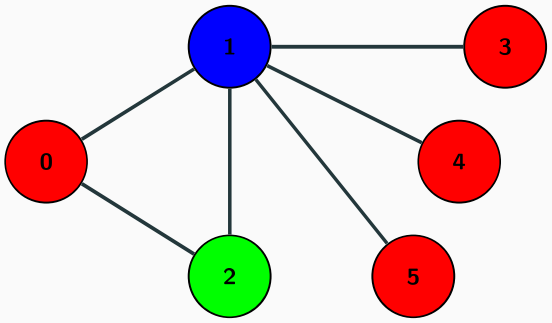
- We have an unweighted undirected graph
- A coloring of the graph is an assignment of colors to the vertices such that adjacent vertices have different colors



- We want to find a coloring that uses the minimum number of distinct colors

Graph coloring

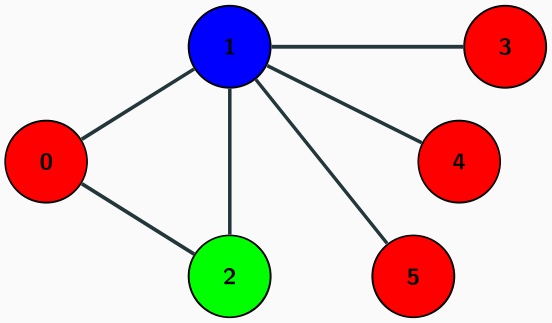
- We have an unweighted undirected graph
- A coloring of the graph is an assignment of colors to the vertices such that adjacent vertices have different colors



- We want to find a coloring that uses the minimum number of distinct colors

Graph coloring

- We have an unweighted undirected graph
- A coloring of the graph is an assignment of colors to the vertices such that adjacent vertices have different colors



- We want to find a coloring that uses the minimum number of distinct colors
- NP-hard in general graphs

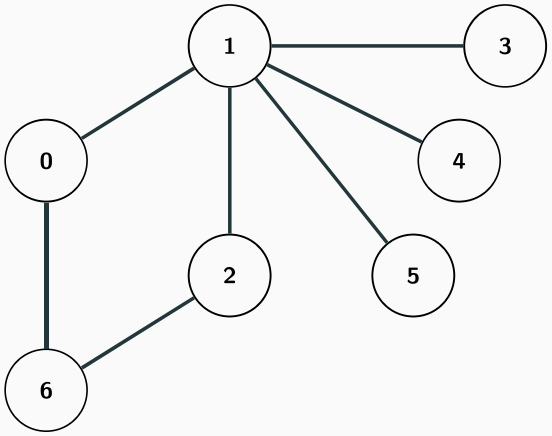
Special graphs

- All of these problems are hard (in some sense) in general graphs
- But what if we're working with special kinds of graphs?

- Let's look at a few examples

Bipartite graphs

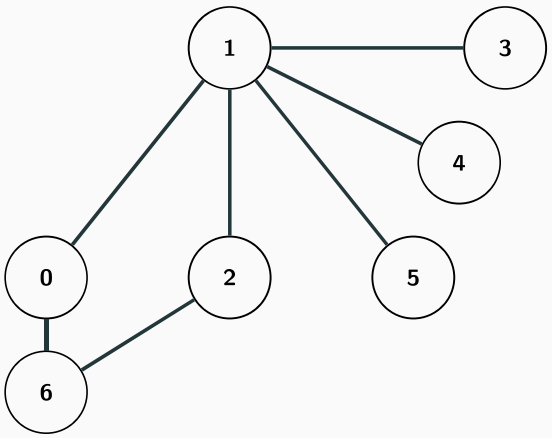
- A graph is bipartite if the vertices can be partitioned into two sets such that for each edge (u, v) u and v are in different sets



- How do we check if a graph is bipartite?

Bipartite graphs

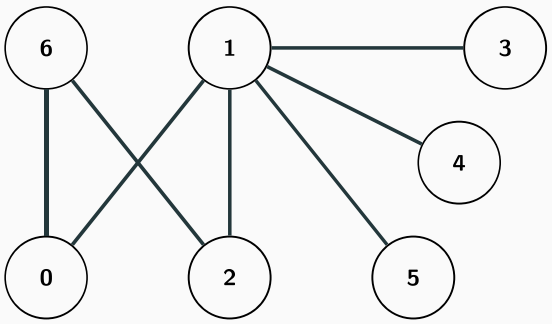
- A graph is bipartite if the vertices can be partitioned into two sets such that for each edge (u, v) u and v are in different sets



- How do we check if a graph is bipartite?

Bipartite graphs

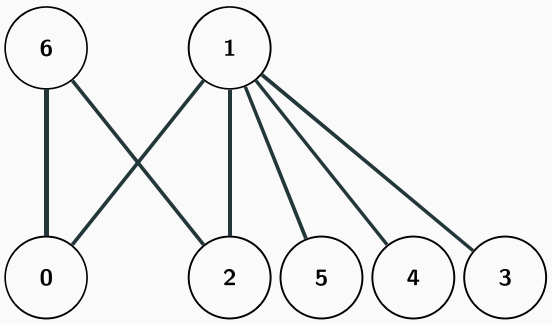
- A graph is bipartite if the vertices can be partitioned into two sets such that for each edge (u, v) u and v are in different sets



- How do we check if a graph is bipartite?

Bipartite graphs

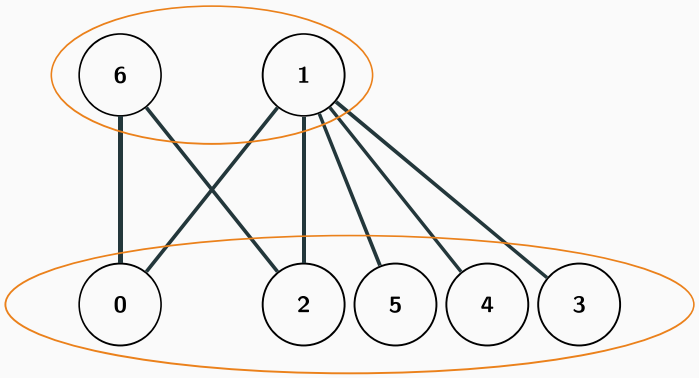
- A graph is bipartite if the vertices can be partitioned into two sets such that for each edge (u, v) u and v are in different sets



- How do we check if a graph is bipartite?

Bipartite graphs

- A graph is bipartite if the vertices can be partitioned into two sets such that for each edge (u, v) u and v are in different sets



- How do we check if a graph is bipartite?

Bipartite graphs

- We want to check if we can split the vertices into these two groups
- Take any vertex, and assume that it's in the first group
- Then all of his neighbors must be in the second group
- And then all of their neighbors must be in the first group
- And so on...

- We can do this with a simple depth-first search
- If we ever find a contradiction (i.e. a vertex must both be in the first and second set), then the graph is not bipartite

Bipartite graphs

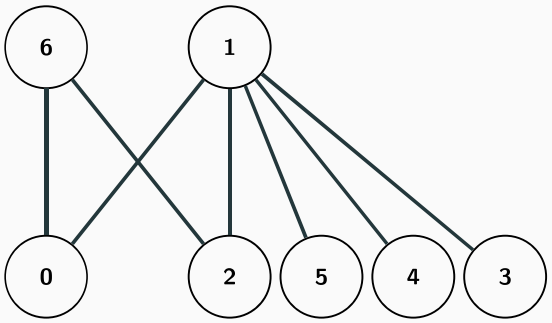
```
vector<int> adj[1000];
vector<int> side(1000, -1);
bool is_bipartite = true;

void check_bipartite(int u) {
    for (int i = 0; i < adj[u].size(); i++) {
        int v = adj[u][i];
        if (side[v] == -1) {
            side[v] = 1 - side[u];
            check_bipartite(v);
        } else if (side[u] == side[v]) {
            is_bipartite = false;
        }
    }
}

for (int u = 0; u < n; u++) {
    if (side[u] == -1) {
        side[u] = 0;
        check_bipartite(u);
    }
}
```

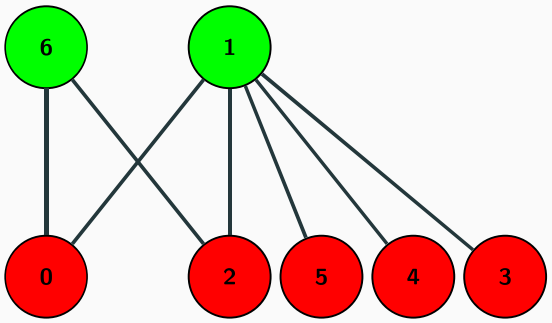
Coloring bipartite graphs

- What if we want to find the minimum graph coloring of a bipartite graph?



Coloring bipartite graphs

- What if we want to find the minimum graph coloring of a bipartite graph?



- Simple, one side can be colored with one color, and the second side can be colored with a second color

Bipartite matching

- Finding a maximum matching in bipartite graphs is very common
- *see example*

- Soon we'll see an efficient algorithm for finding the maximum matching in a bipartite graph