ICC4101 - Algorithms and Competitive Programing

Javier Correa

Graphs

A graph is a structure composed of nodes/vertexes (V) which are related between each other with edged ($(u, v) \in E$):

$$G = (V, E).$$

Adjacency Matrix

A possible representation of a graph is using an Adjacency Matrix. A graph is represented with a matrix M such as, if nodes v_i and v_j are connected, e.g. $(v_i, v_j) \in V$, then $M[i, j] \neq 0$, otherwise $M[i, j] = 0$.

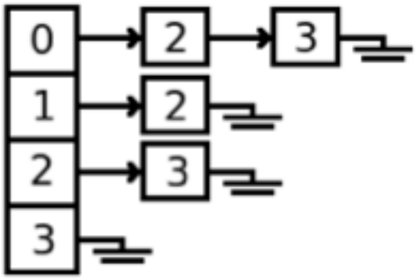
- For an undirected graph, the matrix M is symmetric.
- Edge weights can be stored in entries of the adjacency matrix.

Adjacency list

A different representation of a graph is by adjacency lists.

- An array of lists $L[]$, with the number of elements equal the number of nodes in the graph
- The ith entry of the list corresponds to a list of all node j such that $(v_i, v_j) \in V$

$$L[i] = list(\dots, j, \dots) \Rightarrow (v_i, v_j) \in E$$



Adjacency matrix vs Adjacency lists

1. Matrix

    - Check if two vertexes are connected takes $O(1)$ time.
    - Store the graph requires $O(|V|^2)$ memory space.

2. Lists

    - Check if two nodes are connected takes $O(|V|)$ time.
    - Store the graph requires $O(|V| + |E|)$ memory space.

Implicit Graph

In cases where the graph is too big or even infinite, it is convenient to calculate the neighbors of a node "on-the-fly" rather than storing them directly.

Other case of implicit graph is when the nodes and edges can be calculated following some problem specific rule.

Dijkstra Algorithm

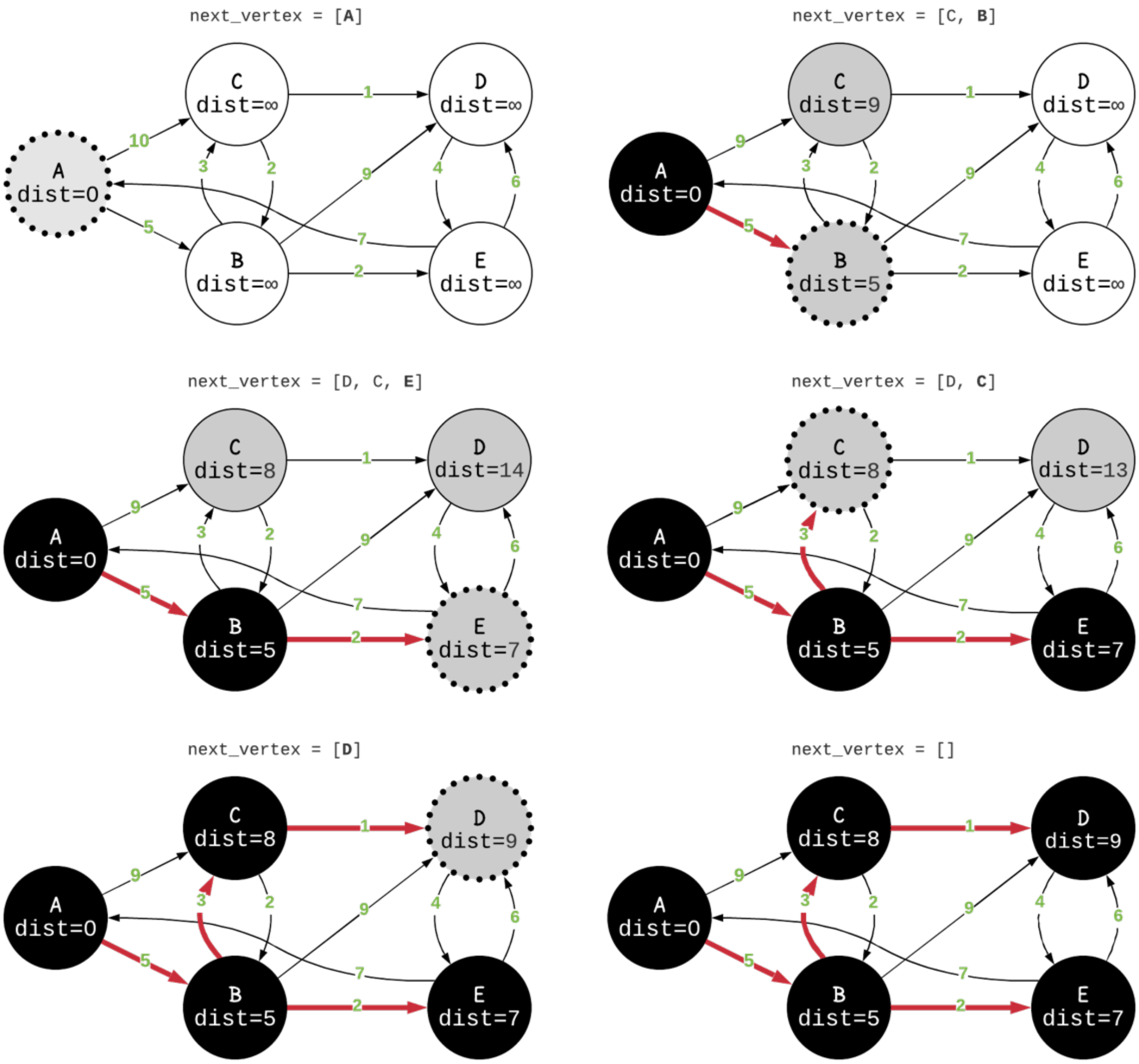
Dijkstra is an algorithm to find the shortest path between two nodes of a (weighted) graph.

Dijkstra Algorithm

```python
def dijkstra(G, s):
    s.dist = 0
    next_vertexes = [s]
    while len(next_vertexes) > 0:
        u = next_vertexes.pop()
        for (v, w) in G.neighbors(u):
            if not hasattr(v, 'dist') or v.dist > u.dist + w:
                v.dist = u.dist + w
                v.parent = u # optional, in case we want to reconstruct the "best" path
            next_vertexes.append(v)
        next_vertexes.sort(key=lambda x: -x.dist)
```

If using dictionaries change the `x.dist` and `x.parent` with `x["dist"]` and `x["parent"]`.

Dijkstra Example



Shortest distance between all nodes

If we want to find the shortest distance between all nodes, calling Dijkstra for every starting node is too expensive.

A better solution is to use Floyd-Warshal algorithm, which calculates the distance between all points using dynamic programing.

This algorithm calculates the matrix $D^{(k)}[i, j]$ which is the smallest cost between nodes i y j using nodes numbered k or less (assuming some arbitrary numbering of the nodes).

Floyd-Warshall Algorithm (I)

With this definition of the matrix D in mind, it is not hard to see that the following recursion holds:

$$D^k[i,j] = \begin{cases} w_{i,j} & , k = 0 \\ \min(D^{k-1}[i,j], D^{k-1}[i,k] + D^{k-1}[k,j]) & \text{, in other cases} \end{cases}$$

where $w_{i,j}$ is the weights/cost between nodes i and j.

Floyd-Warshall Algorithm (II)

or using `defaultdict`:

```
from collections import defaultdict
from math import inf

def floyd_warshall(G):
    D0 = defaultdict(lambda: inf)
    D1 = defaultdict(lambda: inf)

    for (i, j, w) in G.edges():
        D0[i, j] = w

    for k in range(len(G.vertex)):
        for i, j in product(range(len(G.vertex)), repeat=2):
            D1[i, j] = min(D0[i, j],
                           D0[i, k] + D0[k, j])
        D0, D1 = D1, D0
    return D0
```

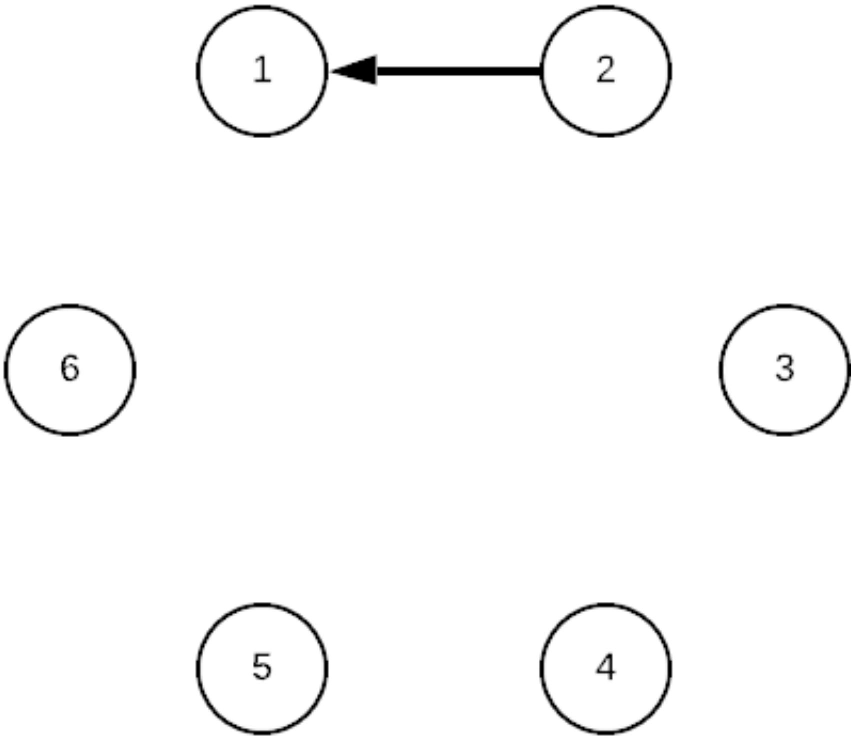
Connected components in undirected graphs

An alternative to exploring using DFS or BFS is to use the Union-Find Disjoint Set data structure. This datastructure is capable of very efficiently merge elements marked as *equal*.

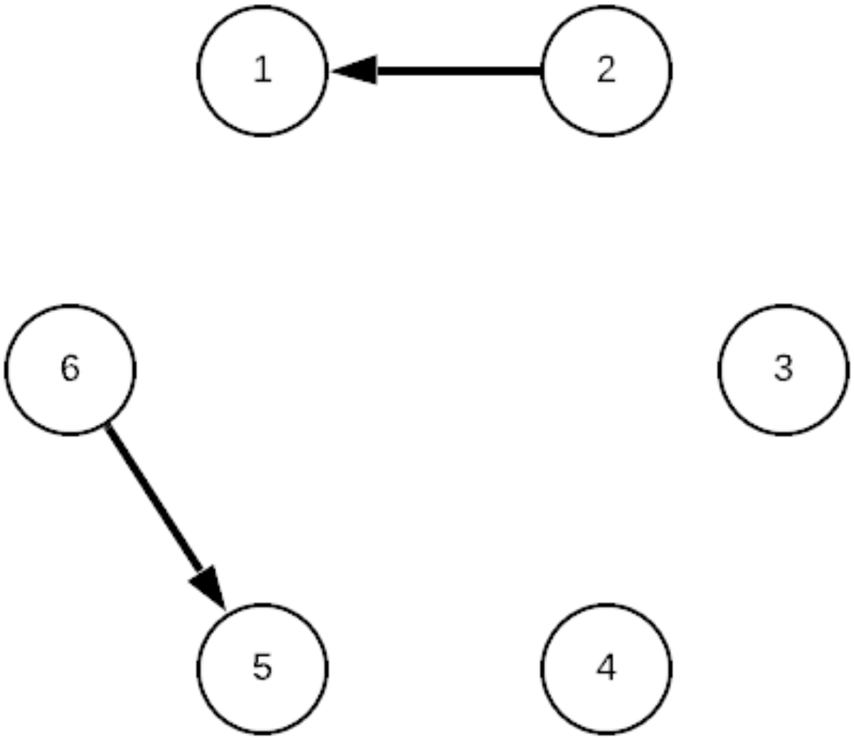
The main advantage is that the complexity of asking if something belongs to the same components (equivalently, asking if two elements are *equal*) is $O(1)$.

```
class disjoint_set:
    def __init__(self):
        self.D = dict()

    def make_set(self, x):
        if x not in self.D:
            self.D[x] = [x, 0]

    def find(self, x):
        if x not in self.D:
            self.make_set(x)
        if self.D[x][0] != x:
            self.D[x] = self.find(self.D[x][0])
        return self.D[x]

    def is_same(self, x, y):
        return self.find(x) == self.find(y)

    def union(self, x, y):
        x_ = self.find(x)
        y_ = self.find(y)

        if x_[1] > y_[1]:
            y_[0] = x_[0]
        else:
            x_[0] = y_[0]
            if y_[1] == x_[1]:
                y_[1] += 1
```

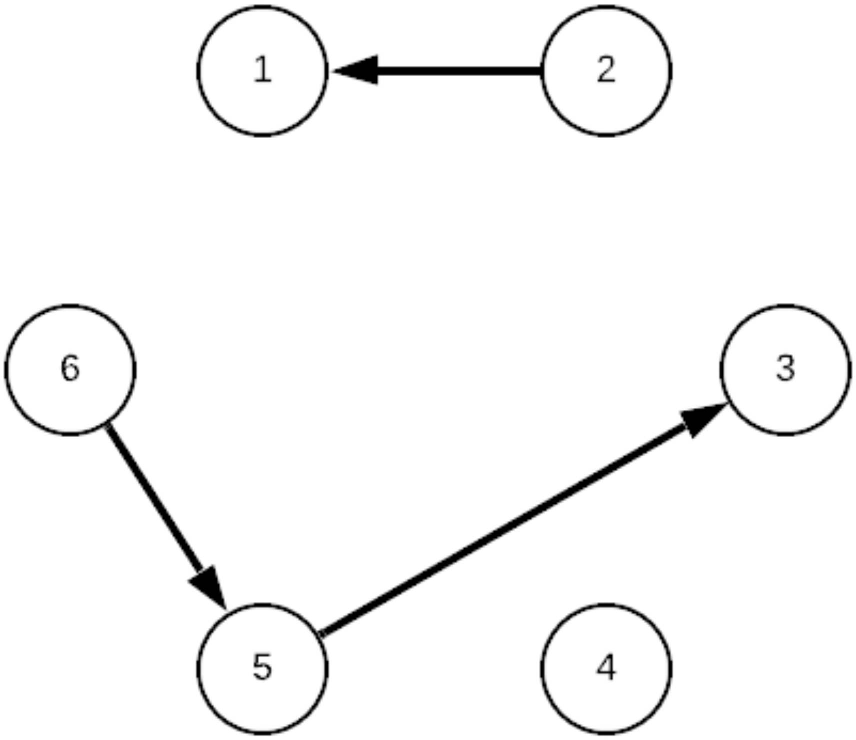
Union(1, 2)



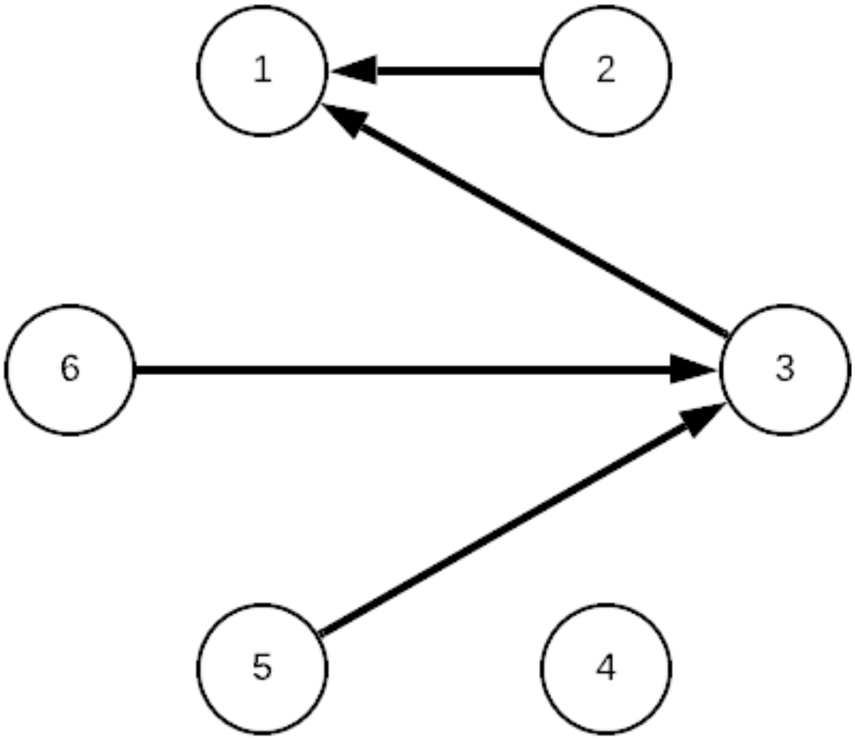
Union(5, 6)



Union(3, 6)



Union(6, 2)



Minimum Spanning Tree (MST)

By running DFS or BFS, we can build a tree that *touches* all vertices of the graph. If the edges have weights, we are interested in finding the tree that minimizes the total weights of the edged of the tree.

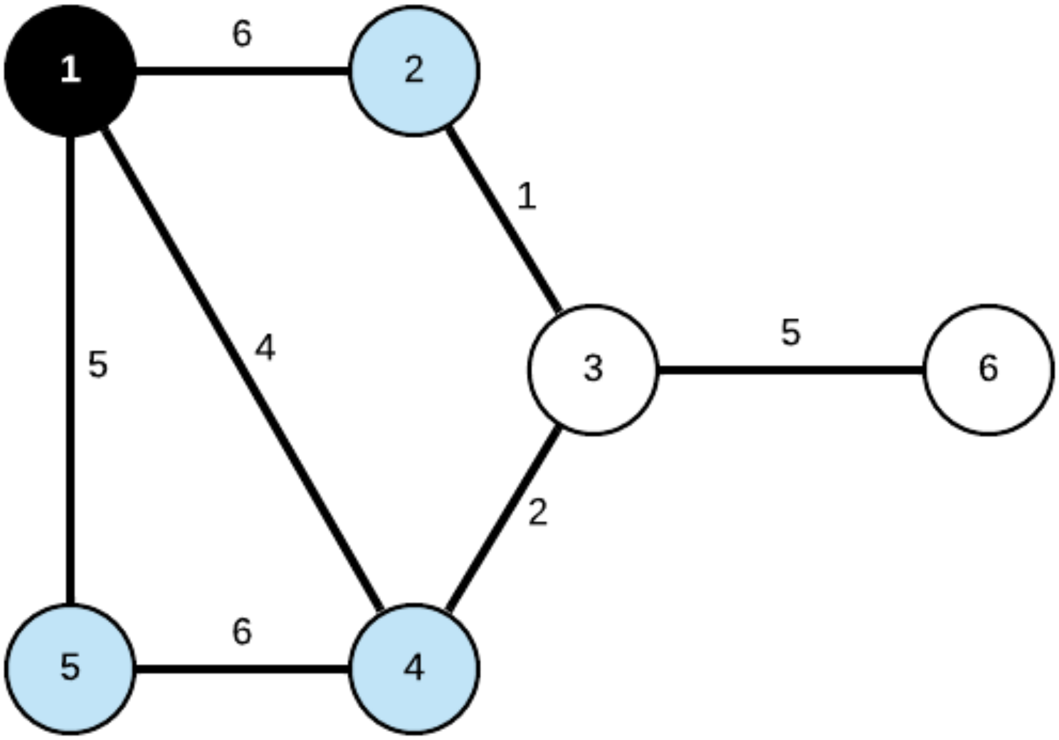
Two classical algorithms solve this problems:

- Prim's algorithm
- Kruskal's algorithm

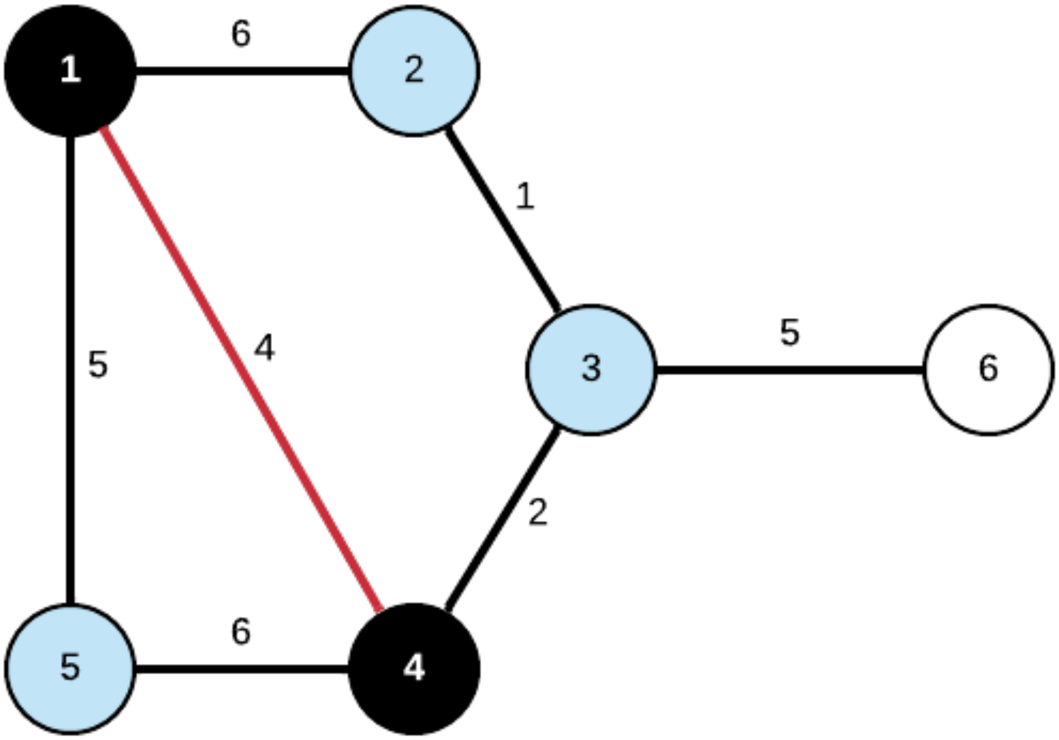
Prim's algorihtm.

```
def prims(i, neighbors, nodes):
    next_vertex = [(i, 0)]
    u = nodes[i]
    u["color"] = color.Grey
    u["parent"] = None
    while len(next_vertex) > 0:
        i,_ = next_vertex.pop()
        v = nodes[i]
        v["color"] = color.Black
        for j, w in neighbors[i]:
            u = nodes[j]
            if u["color"] == color.White:
                u["color"] = color.Grey
                next_vertex.append((j, w))
                u["parent"] = i
            elif u["color"] == color.Grey:
                nv = next(nv for nv in next_vertex if nv[0] == j
and nv[1] > w)
                nv[1] = w
                u["parent"] = i
        next_vertex.sort(key=lambda x: x[1], reverse=True)
    return parents
```

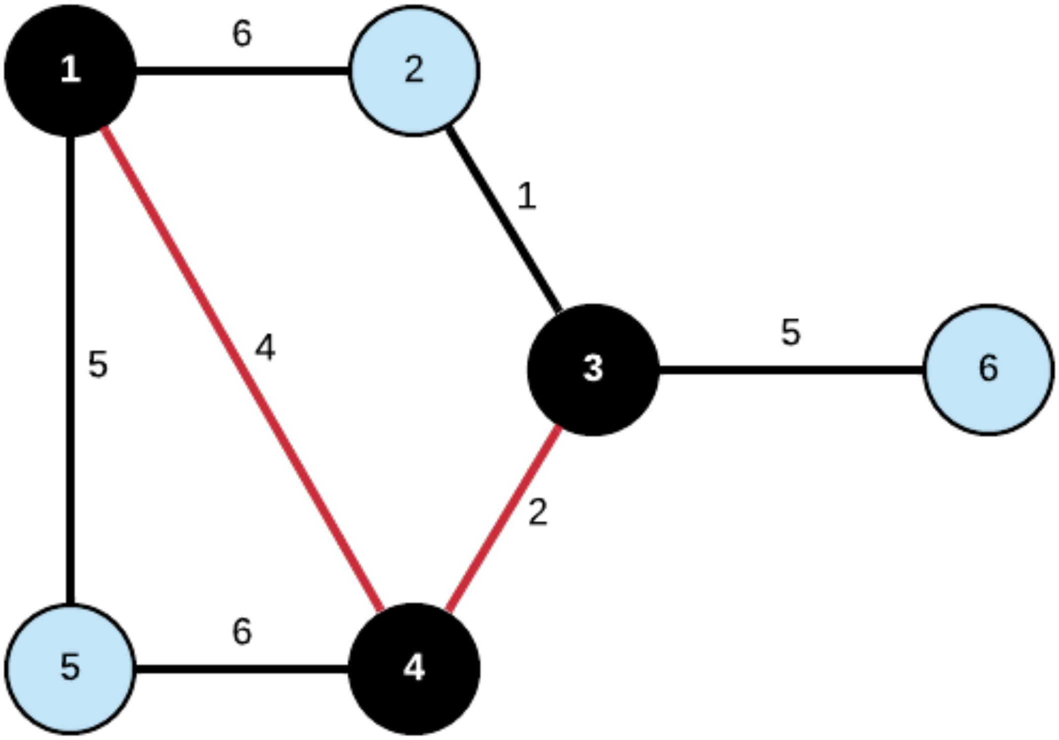
```
next_vertex = [(2, 6), (5, 5), (4, 4)]
```



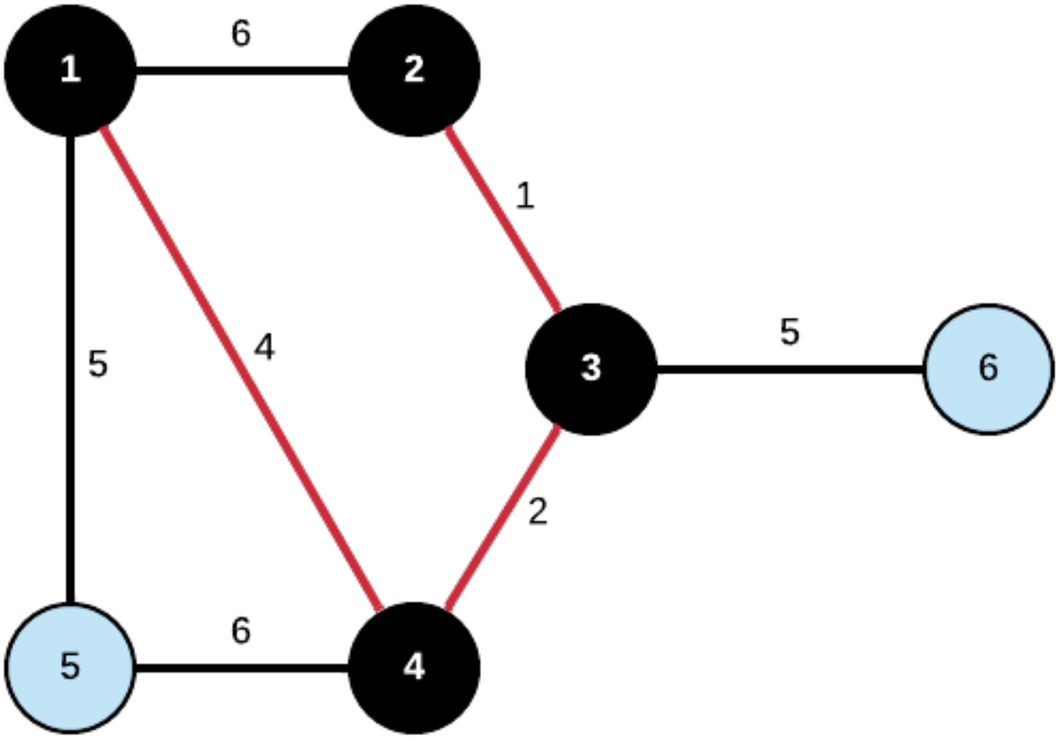
```
next_vertex = [(2, 6), (5, 5), (3, 2)]
```

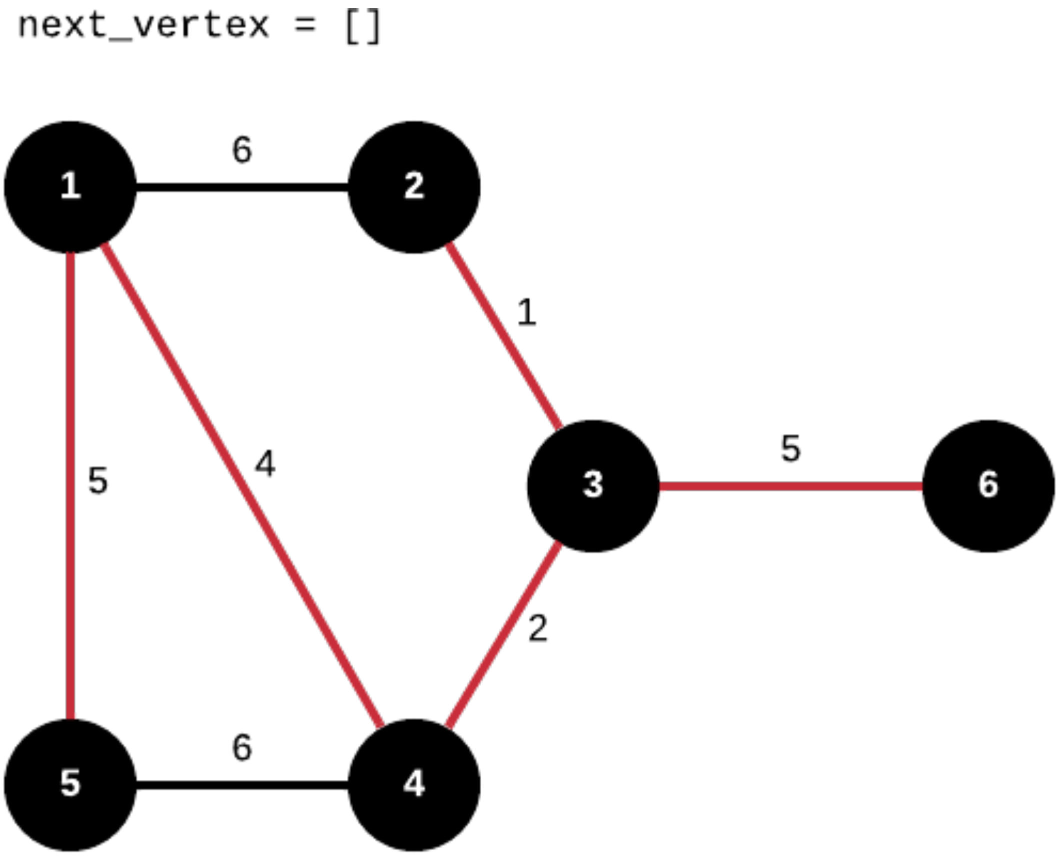


```
next_vertex = [(6, 5), (5, 5), (2, 1)]
```



```
next_vertex = [(6, 5), (5, 5)]
```





```
next_vertex = []
```

Kruskal algorithm

Kruskal algorithm iterates over edges insted of vertices, and uses disjoin-set-union data structure to avoid creating loops.

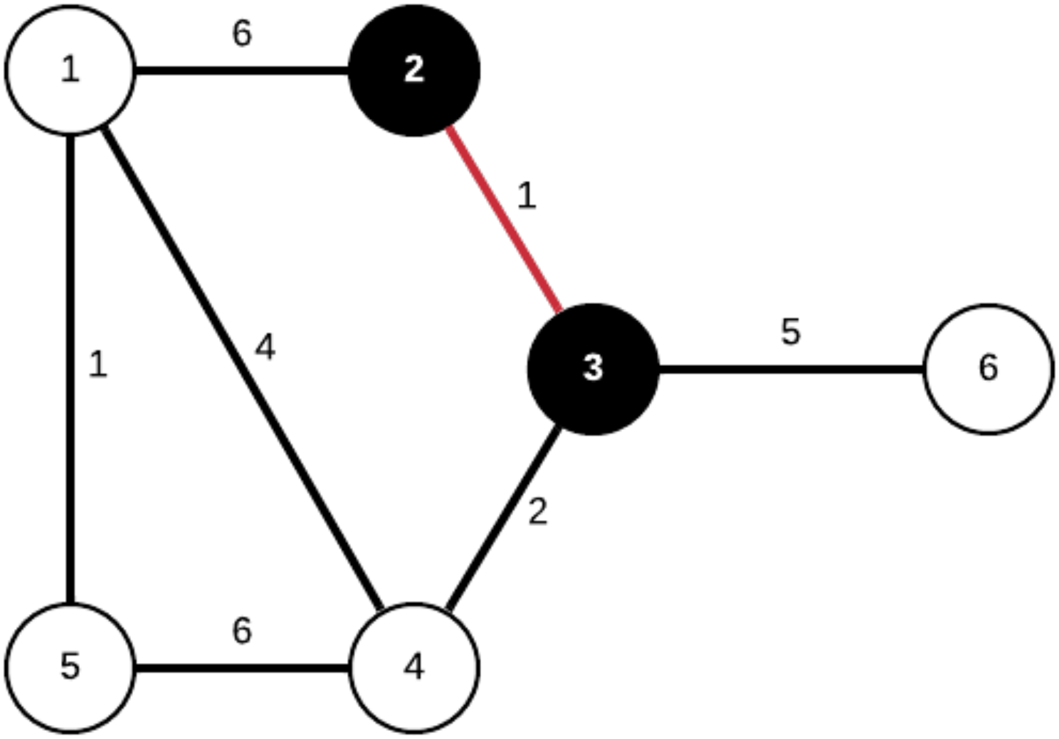
In [ ]:
```
neighbors = {1: [(2, 6.0), (4, 4.0), (5, 1.0)], 2: [(1, 6.0), (3, 1.0)], ...

edges = list((u, v, w) for u in neighbors
                       for (v, w) in neighbors[u]) # equivalente G.edges()
edges = [(1, 2, 6.0), (1, 4, 4.0), (1, 5, 1.0), ..,]

sorted_edges = sorted(edges, key=lambda x: x[2], reverse=True)
```

In [ ]:
```
try:
    input()
except:
    break
```

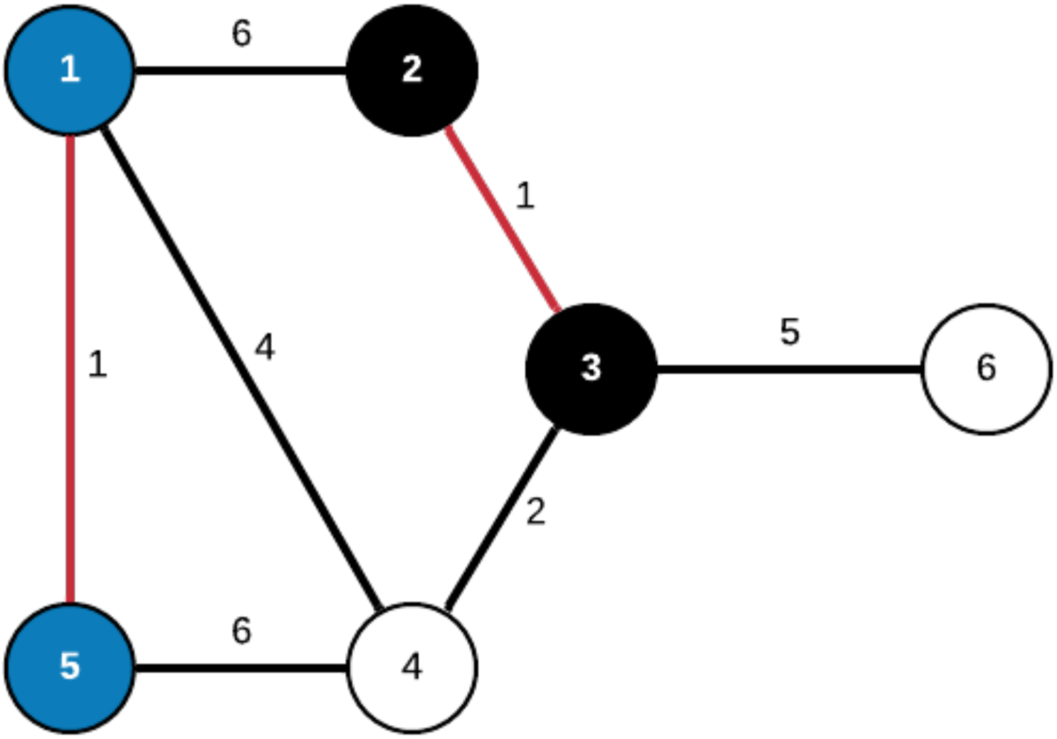
And Kruskal algorithms:

```python
def kruskal(neighbors):
    sorted_edges = sorted(((u, v, w) for u in neighbors
                                     for (v, w) in neighbors[u]),
                          key=lambda x: x[2],
                          reverse=True)
    djs = disjoint_set()

    V = list()
    while len(sorted_edges) > 0:
        u, v, _ = sorted_edges.pop()
        if not djs.is_same(u, v):
            V.append((u, v))
            djs.union(u, v)
    return V
```

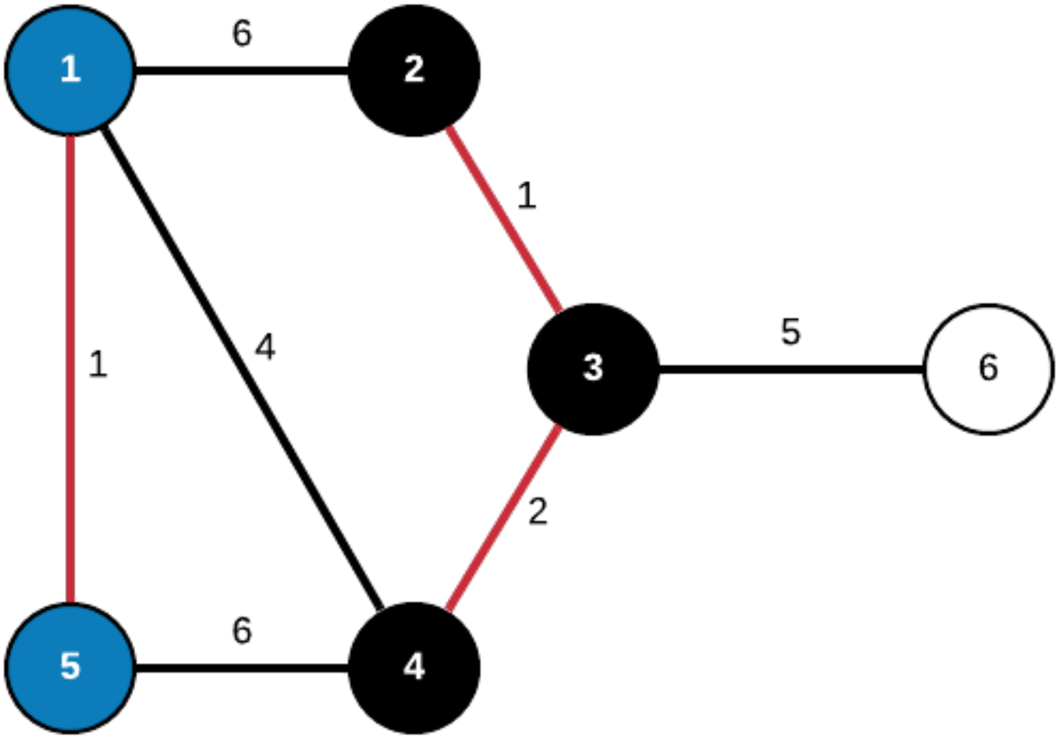
```
sorted_edges = [(1,5), (3,4), (1,4),
                (3,6), (1,2), (4,5)]
V = [(2,3)]
```



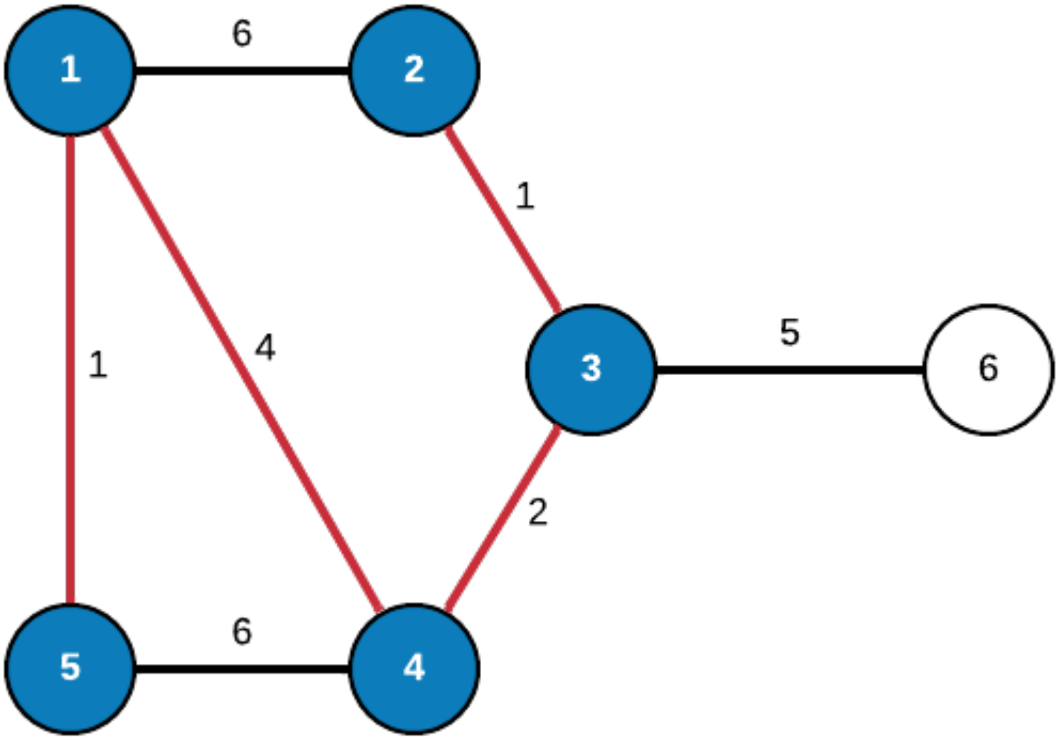
```
sorted_edges = [(3,4), (1,4), (3,6),
                (1,2), (4,5)]
V = [(2,3), (1,5)]
```



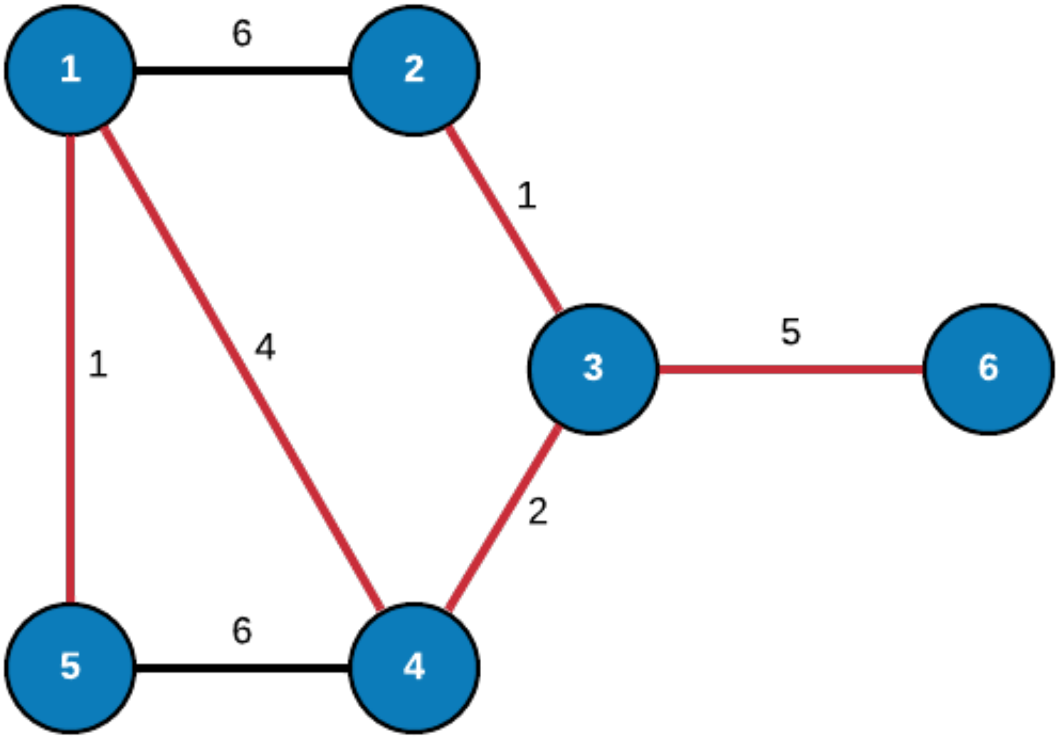
```
sorted_edges = [(1,4), (3,6), (1,2),
                (4,5)]
V = [(2,3), (1,5), (3,4)]
```

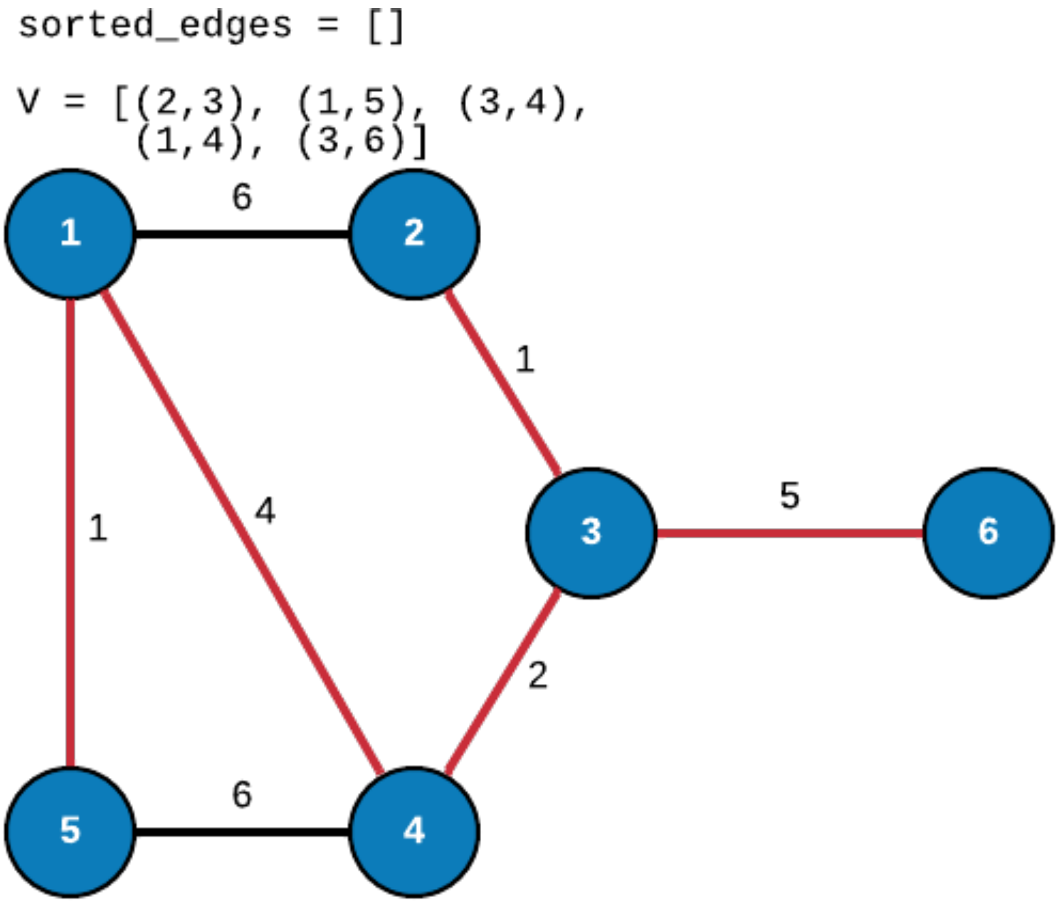


```
sorted_edges = [(3,6), (1,2), (4,5)]

V = [(2,3), (1,5), (3,4), (1,4)]
```



```
sorted_edges = [(1,2), (4,5)]

V = [(2,3), (1,5), (3,4),
     (1,4), (3,6)]
```





```
sorted_edges = []

V = [(2,3), (1,5), (3,4),
    (1,4), (3,6)]
```

Variations

1. Max spanning tree
2. Min spanning subgraph, what to do when we are forces to take some edges?
3. Second best spanning tree (or in general n-th best spanning tree)

Max flow problems

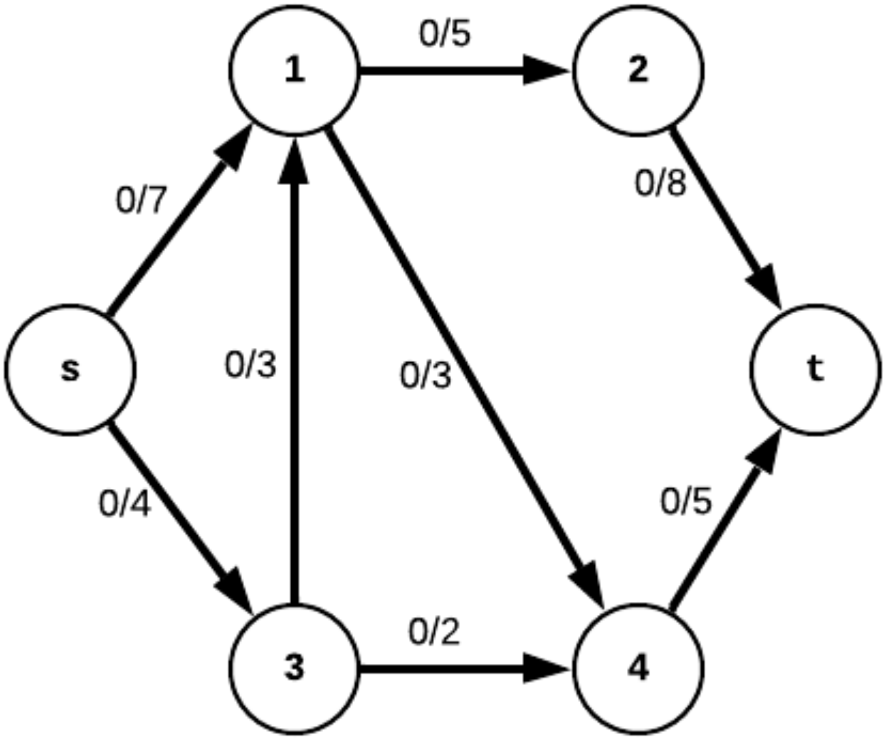
Given a directed graph `G` with weighted edges that represent the flow capacity between nodes, find the maximum flow that can be send from a source node `s` to a sink node `t`.

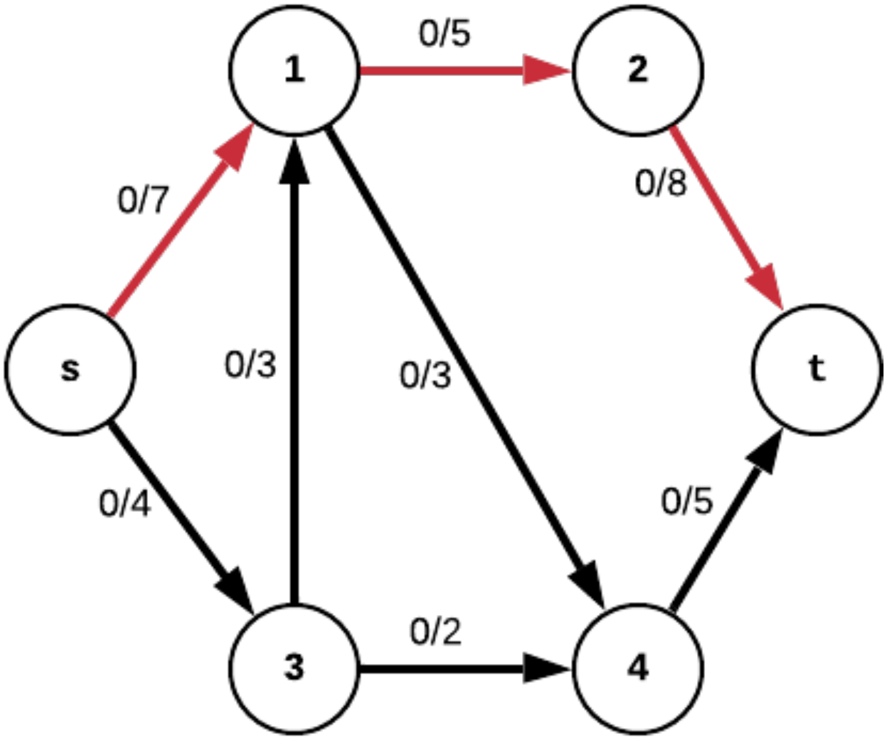
The idea to solve this problem is:

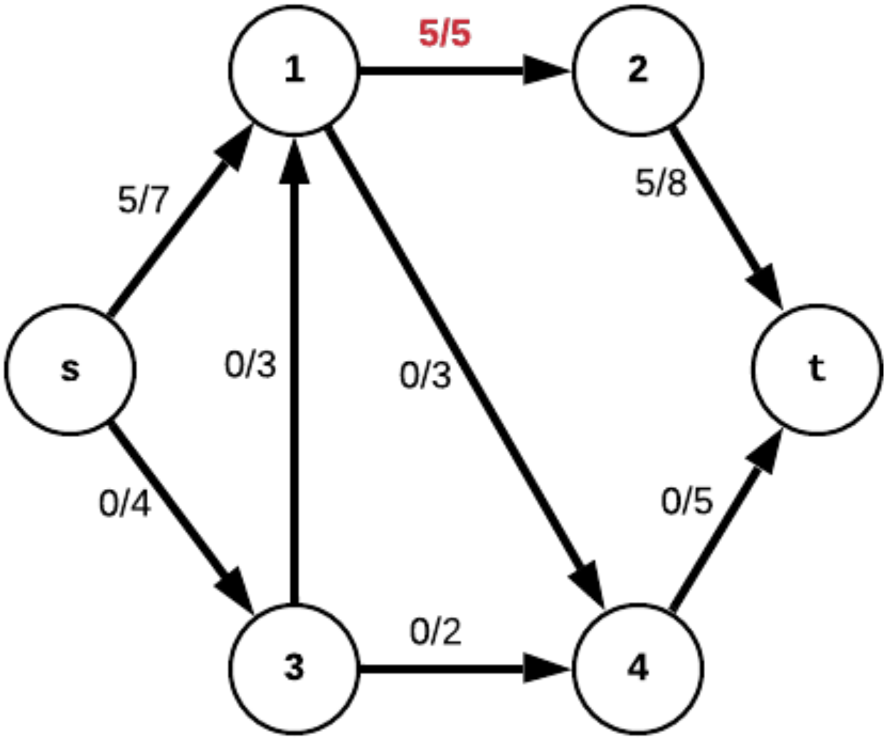
1. Run BFS from the `s` node until the `t` node is found and record the path that was followed.
2. Send the maximum possible flow through that path and update the "remaining capacity" of the edges of the path.
3. Iterate until no new flow can be sent from `s` to `t`.

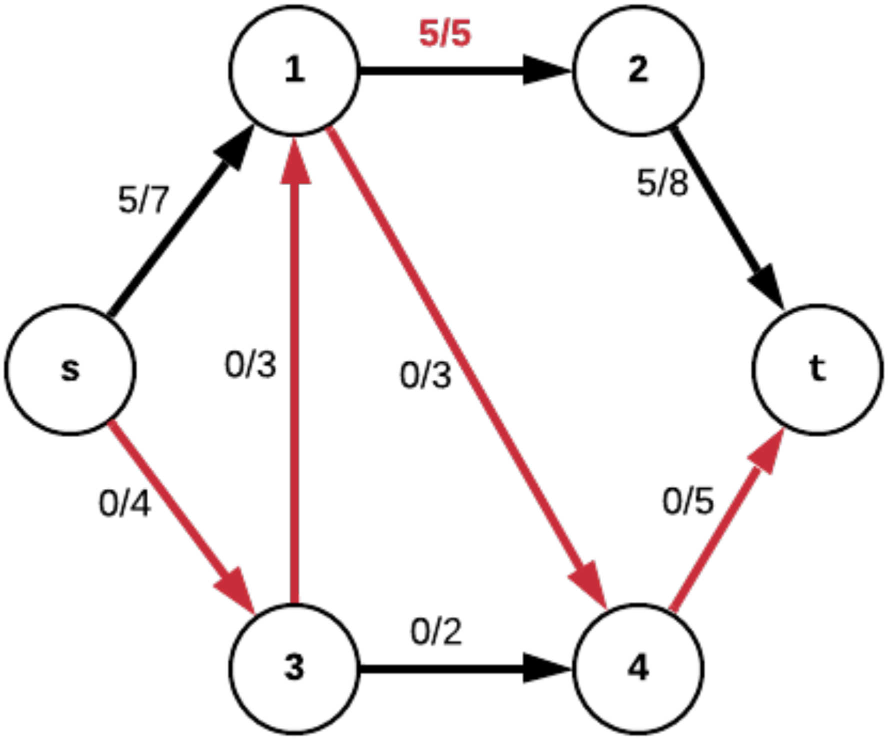
This is the Edmon-Karp Algorithm.

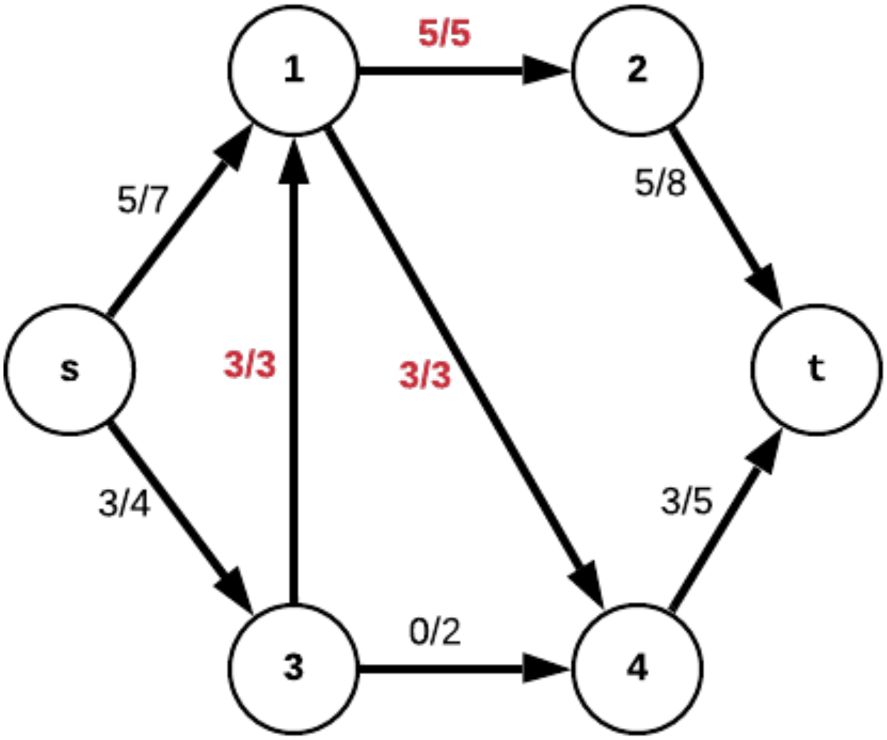
```python
def maxflow(G, s, t, edge_weights):
    def bfs(s, t):
        parent.clear()
        parent[s] = tuple() # something different than None
        next_node = [(s, 1E10)]

        # Find a path between s and t and calculate the
        # total flow that can be sent in that path
        while len(next_node) > 0:
            u, w = next_node.pop()

            for v in G[u]:
                if parent[v] is None and edge_weights[u, v]:
                    parent[v] = u;
                    new_flow = min(w, edge_weights[u, v])
                    if v == t:
                        return new_flow
                    next_node.append((v, new_flow))

        return 0

    flow = 0
    parent = defaultdict(lambda: None)

    while True:
        new_flow = bfs(s, t)
        if new_flow == 0:
            break
        flow += new_flow
        cur = t;
        # update the remaining capacity of the path
        while cur != s:
            prev = parent[cur]
            edge_weights[prev, cur] -= new_flow
            edge_weights[cur, prev] += new_flow
            cur = prev
    return flow
```

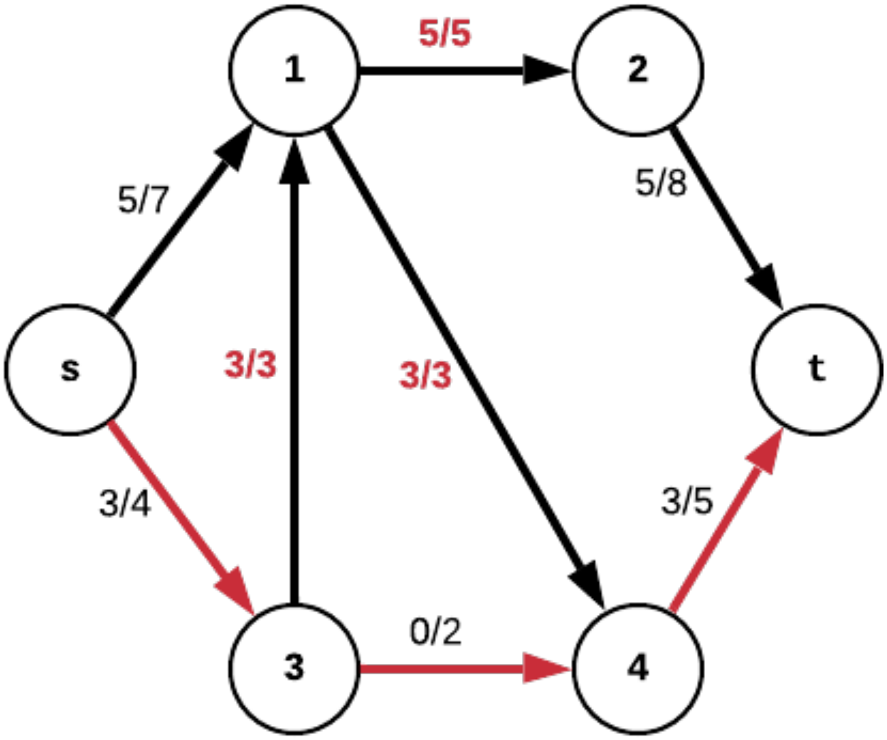

1
2
0/5
0/7
0/8
s
0/3
0/3
t
0/4
0/5
0/2
3
4

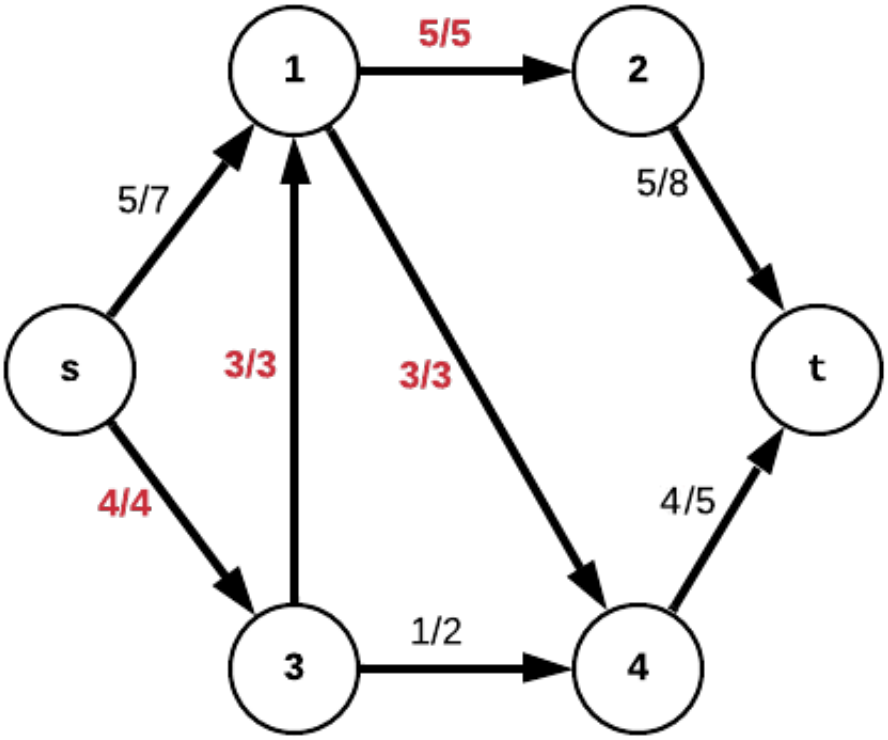

1
2
0/5
0/7
0/8
s
0/3
0/3
t
0/4
0/5
0/2
3
4

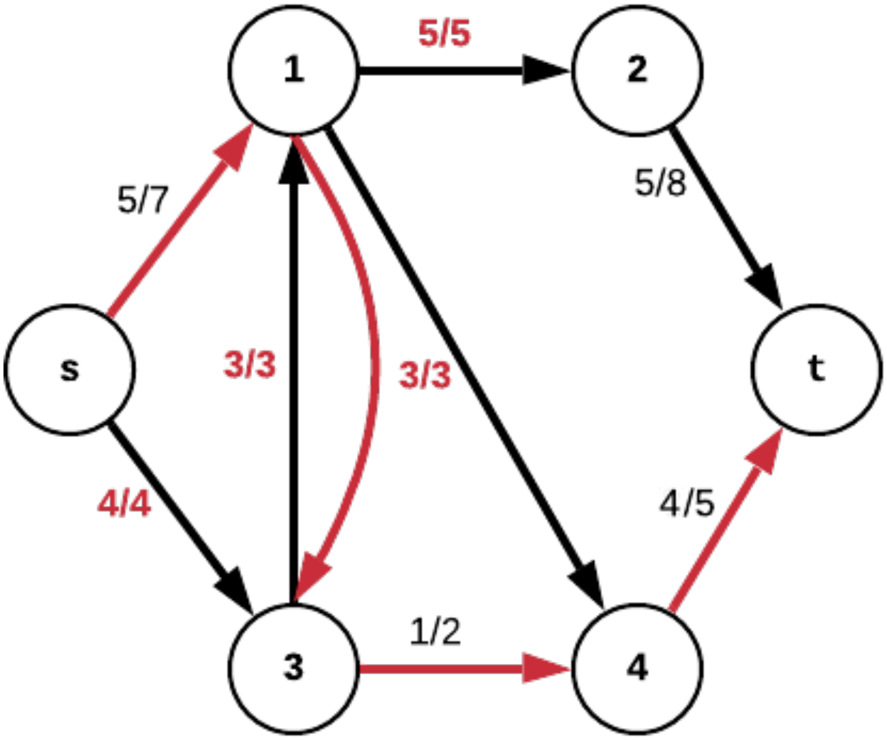

1
5/5
2
5/7
5/8
s
0/3
0/3
t
0/4
0/5
0/2
3
4

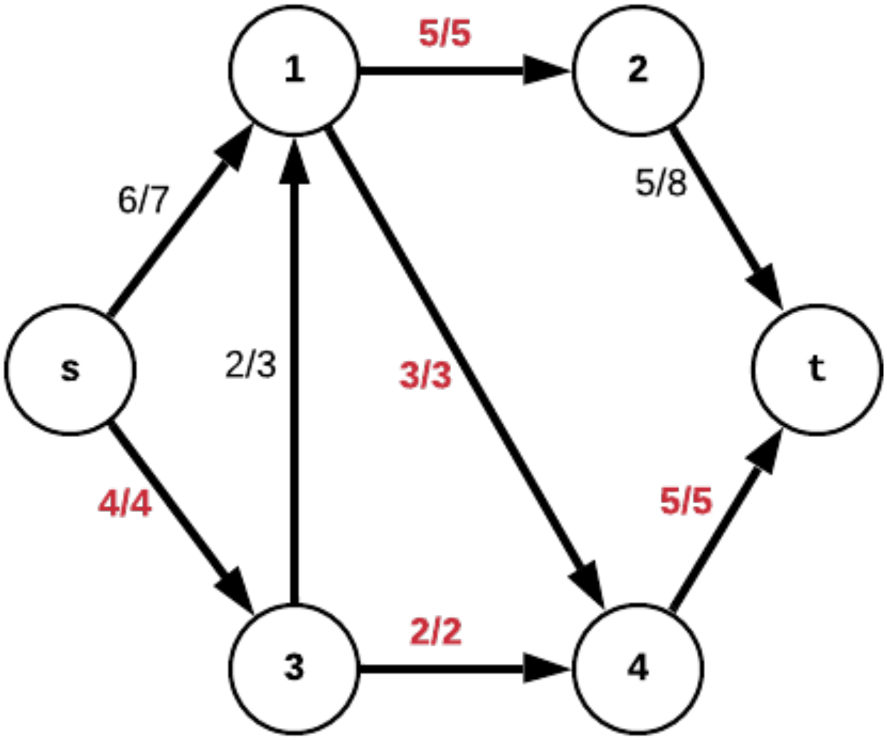

1
5/5
2
5/7
5/8
s
0/3
0/3
t
0/4
0/5
0/2
3
4


1
5/5
2
5/7
5/8
s
3/3
3/3
t
3/4
3/5
3
0/2
4


1
5/5
2
5/7
5/8
s
3/3
3/3
t
3/4
3/5
3
0/2
4


1
5/5
2
5/7
5/8
s
3/3
3/3
t
4/4
4/5
1/2
3
4


1
5/5
2
5/7
5/8
s
3/3
3/3
t
4/4
4/5
1/2
3
4



```
In [ ]:
```