Kattis

Graphs Part 2

Atli Fannar Franklín

October 10, 2025

School of Computer Science
Reykjavík University

Today we're going to cover

- Minimum spanning tree
- Shortest paths
  - Single source
  - Negative weights
  - All pairs

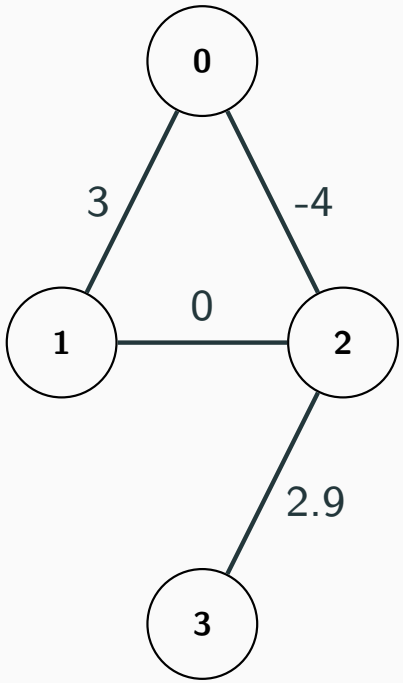
Weighted graphs

- 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 {
    // Depending on context,
    // having both endpoints
    // can be overkill
    int u, v;
    int weight;
    // Sometimes weight has to
    // be a double, or something else

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



Weighted graphs

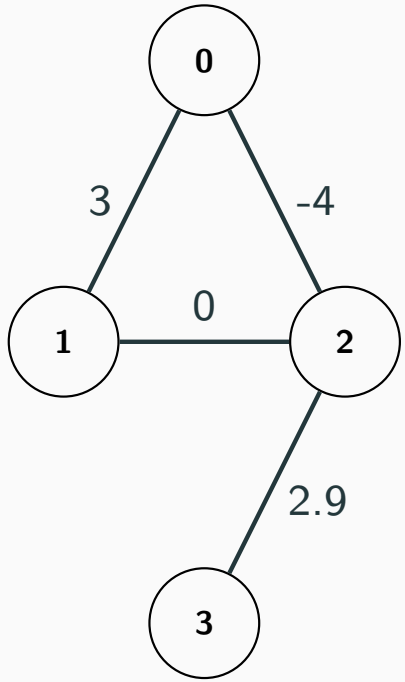
```
vector<edge> adj[4];

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 trees

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

- This can model many things, the cheapest way to connect a set of objects is one of them
- For some graph problems we can remove everything but a spanning tree to simplify calculations
- Note however that it does not have to be unique, but the minimum weight is unique

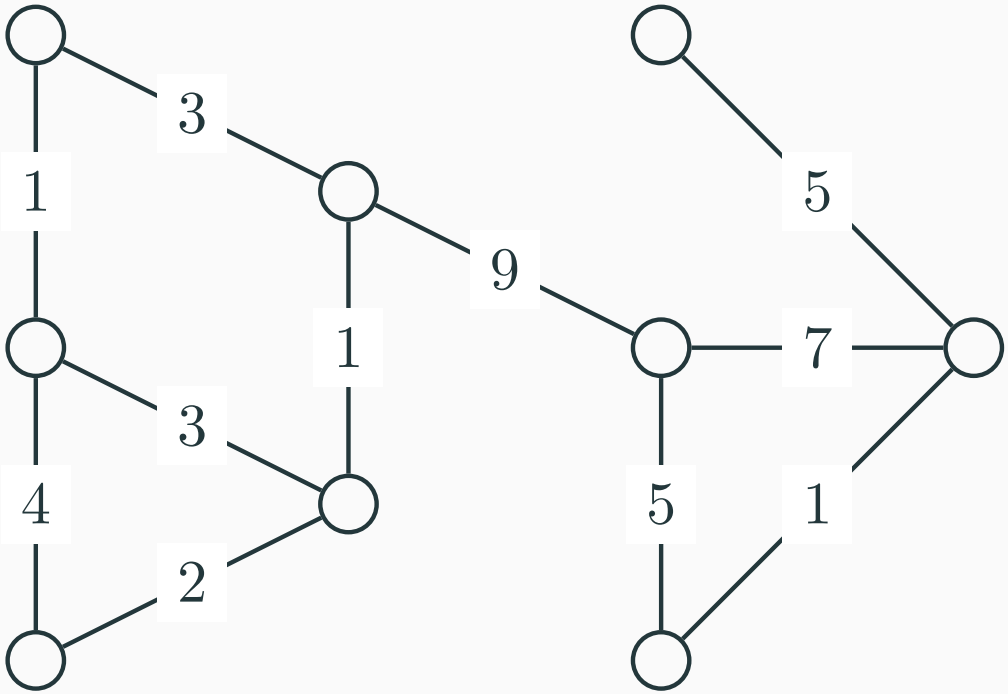
Minimum spanning tree

- Several greedy algorithms work

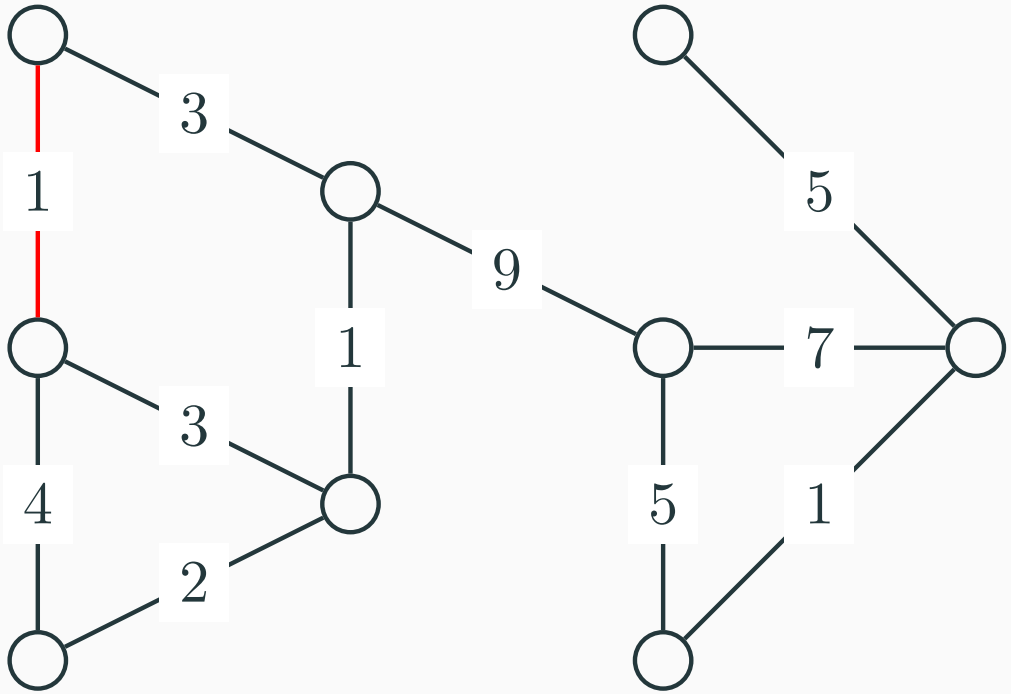
- Go through the edges in the graph in increasing order of weight
- Greedily pick an edge if it doesn't form a cycle (Union-Find can be used to keep track of when we would get a cycle)
- When we've gone through all edges, we have a minimum spanning tree

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

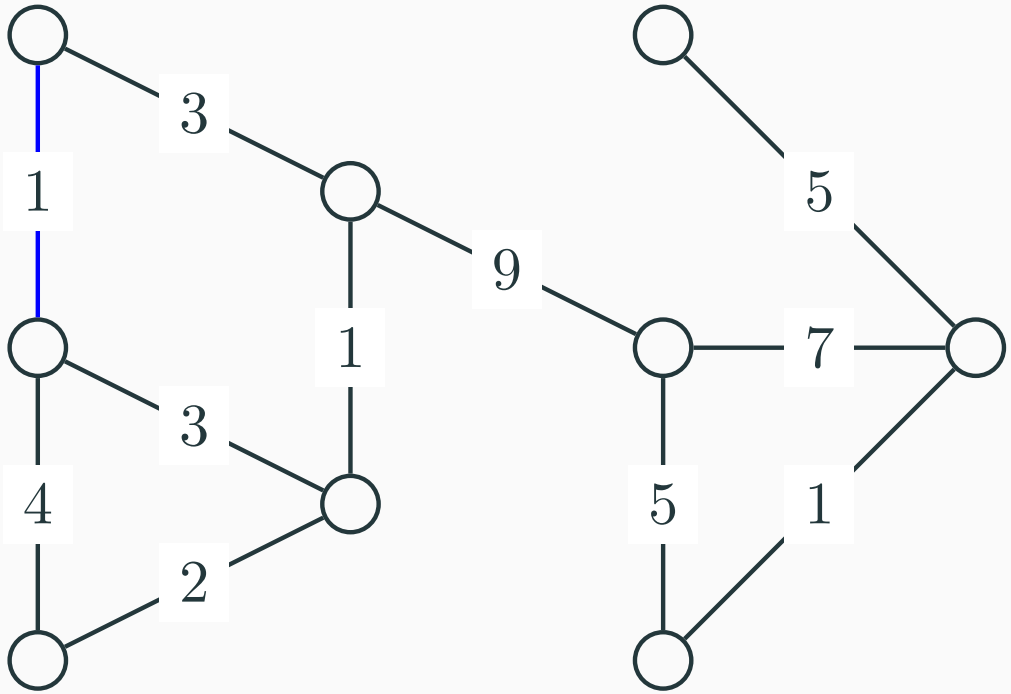
Example



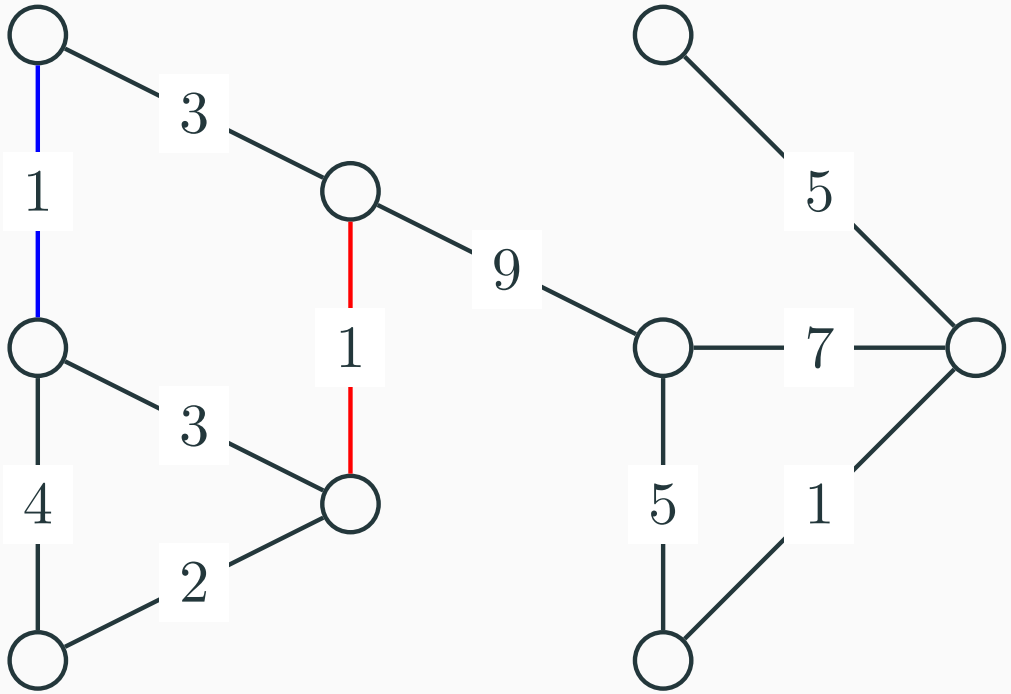
Example



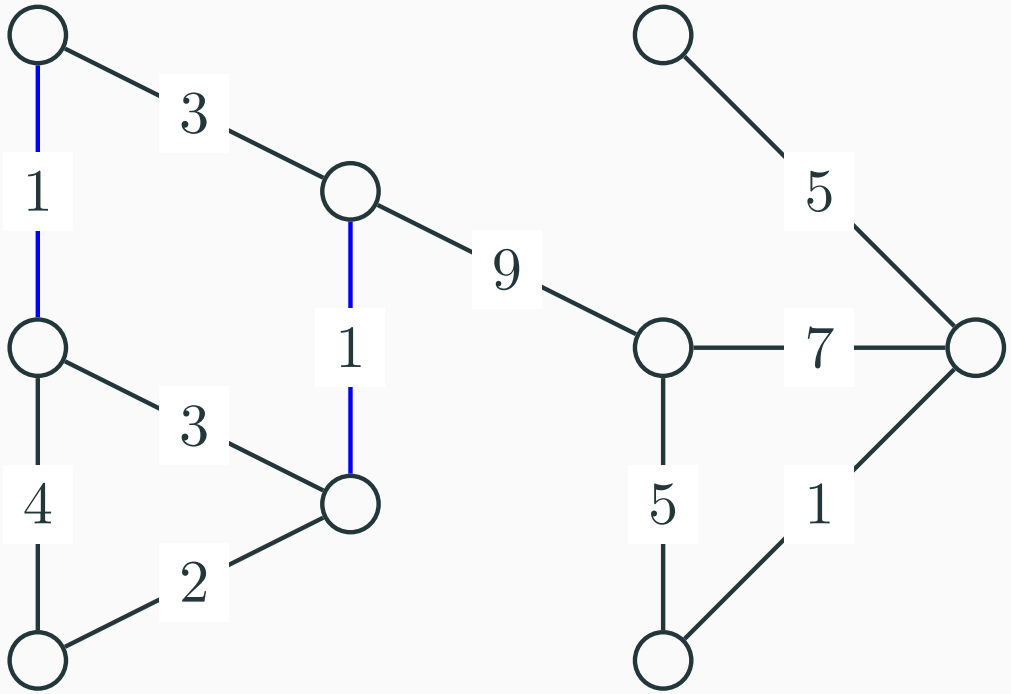
Example



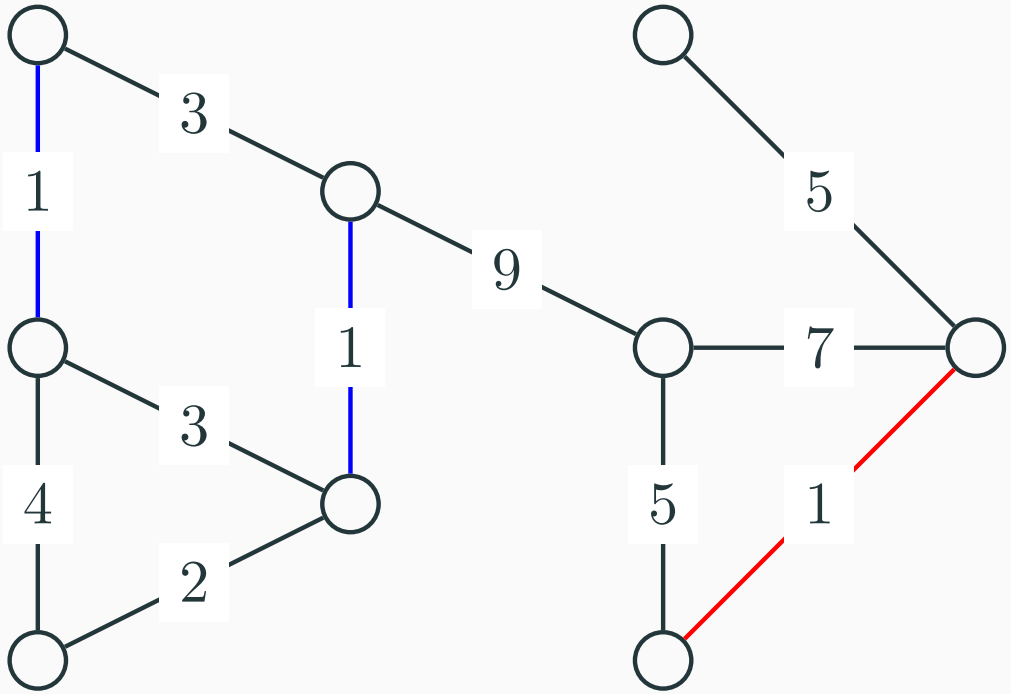
Example



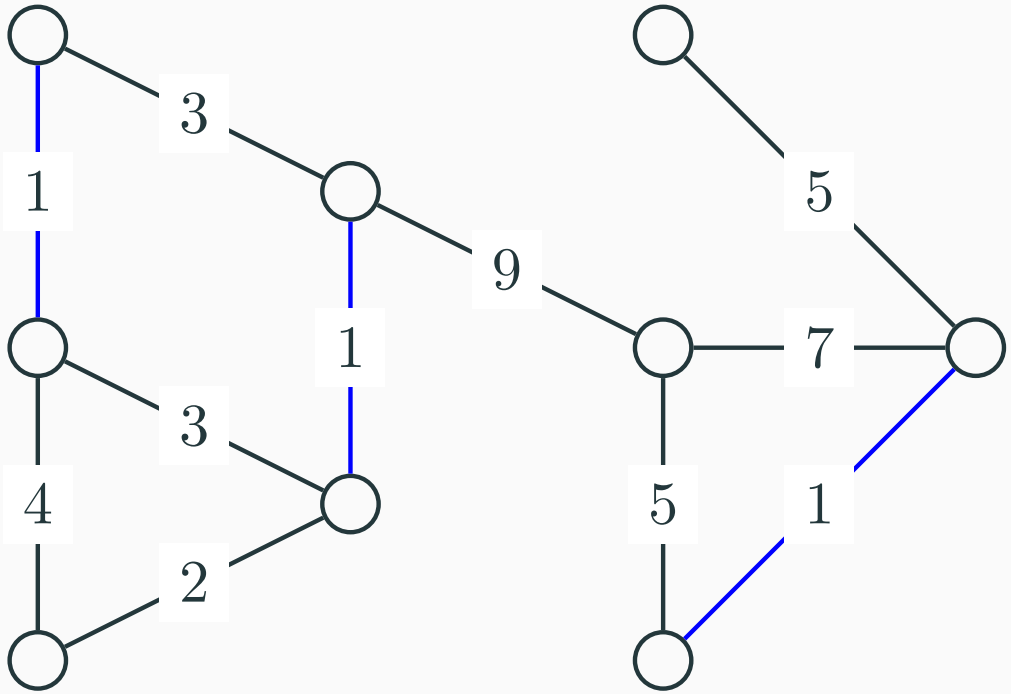
Example



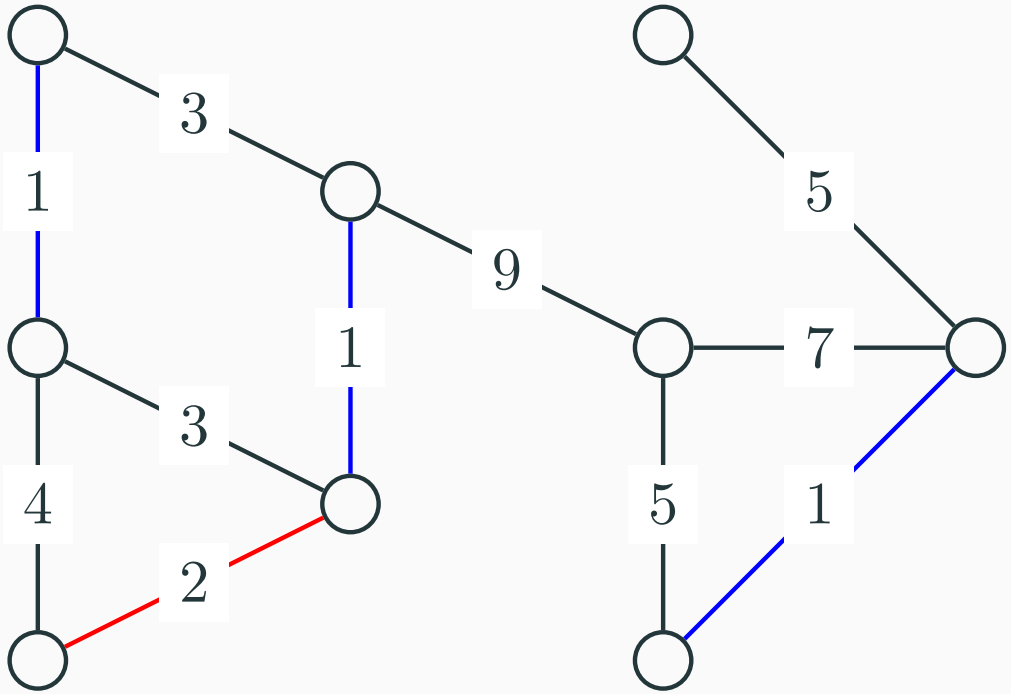
Example



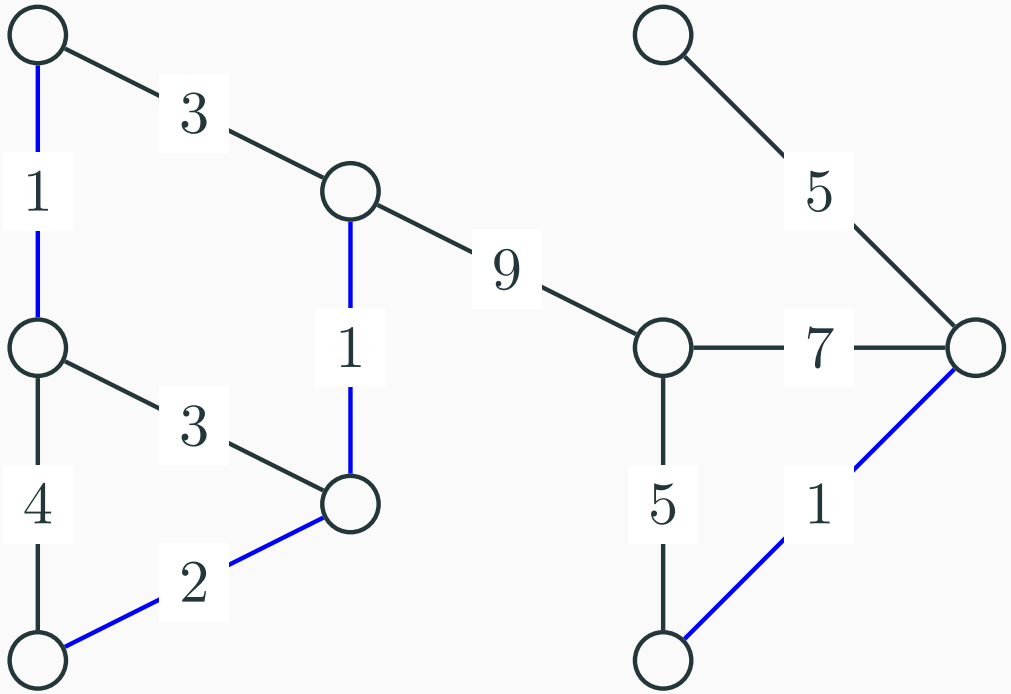
Example



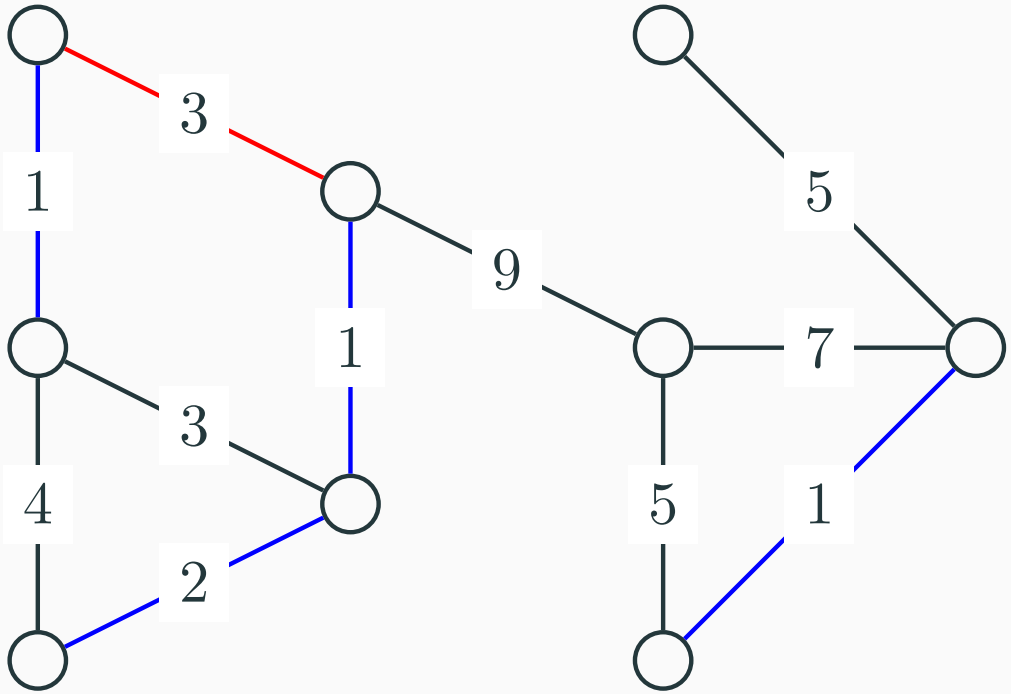
Example



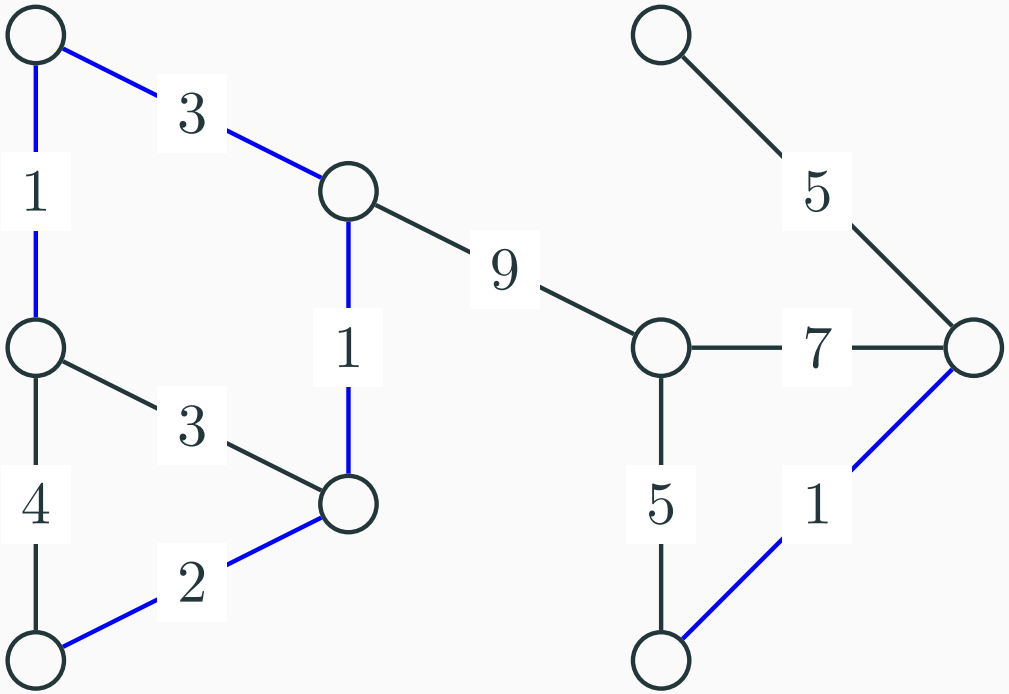
Example



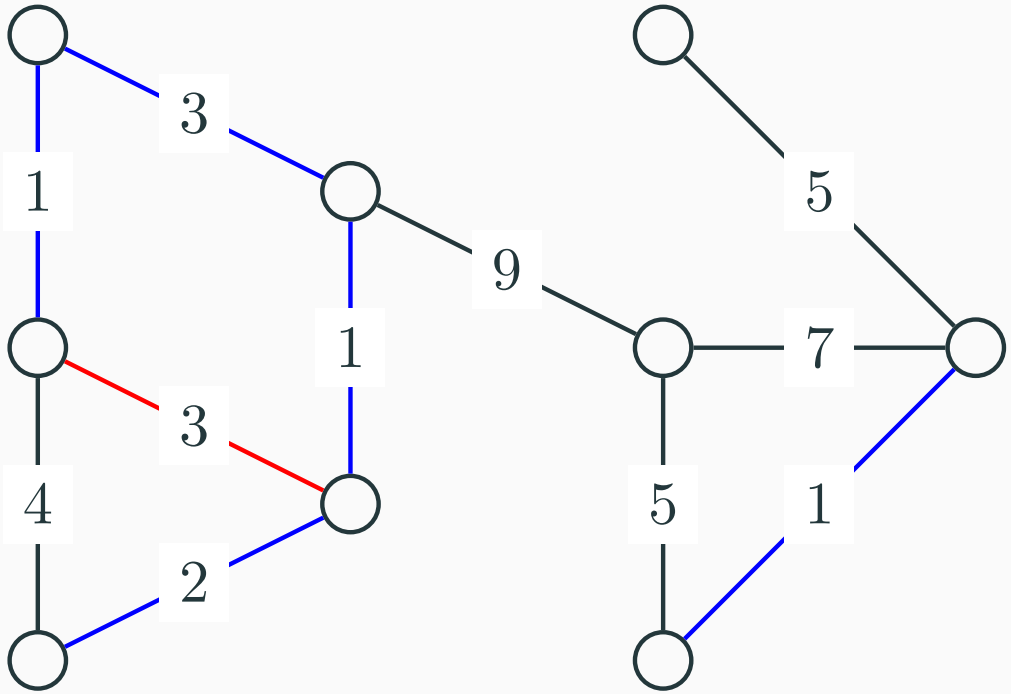
Example



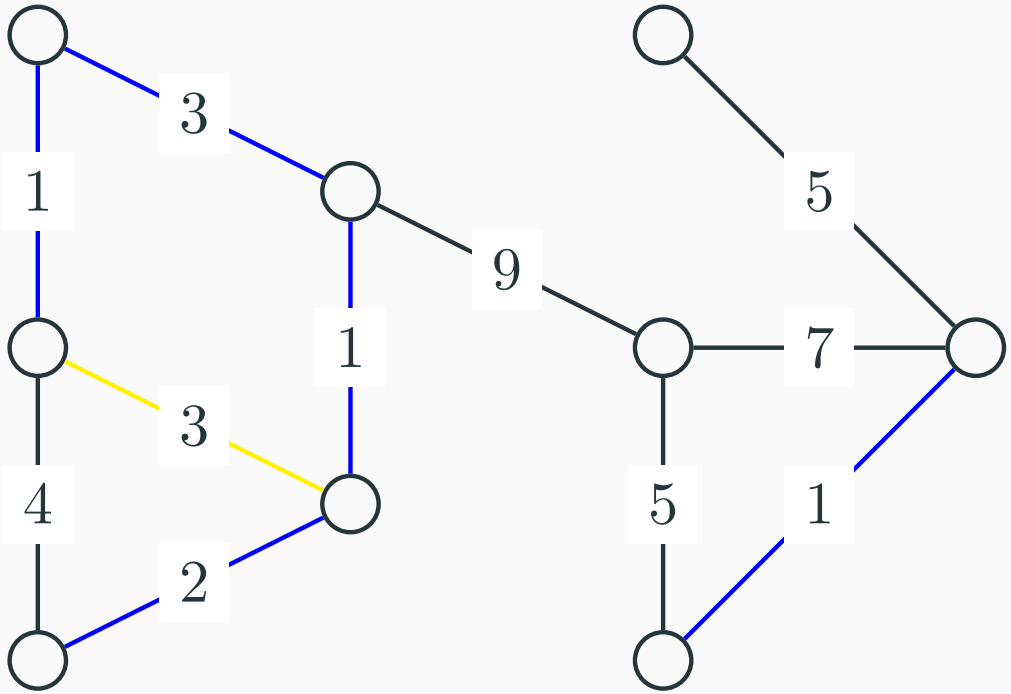
Example



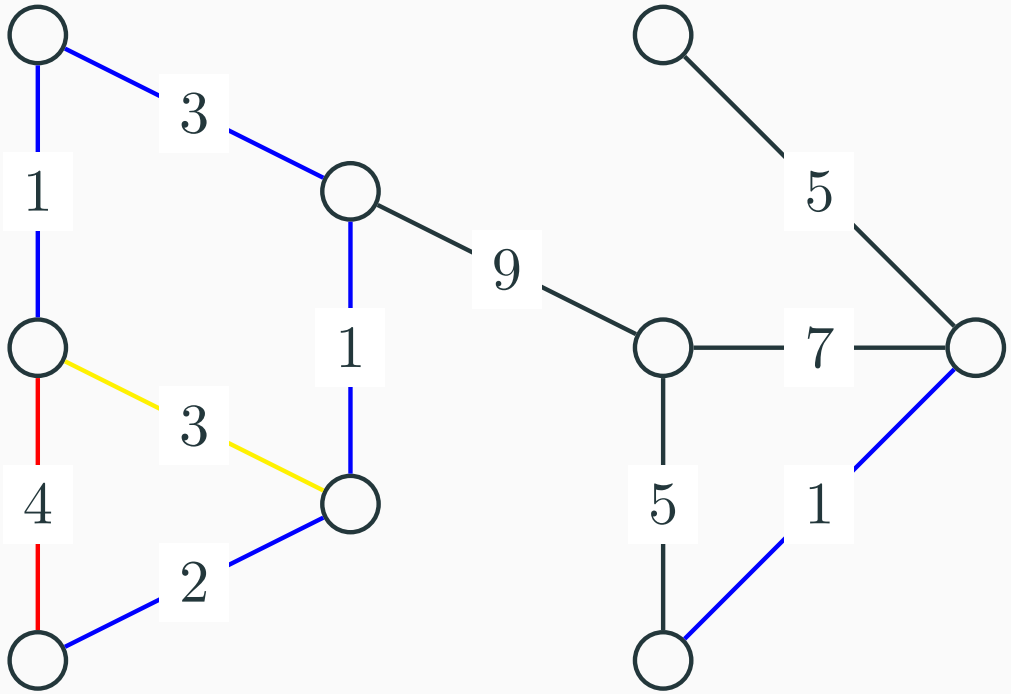
Example



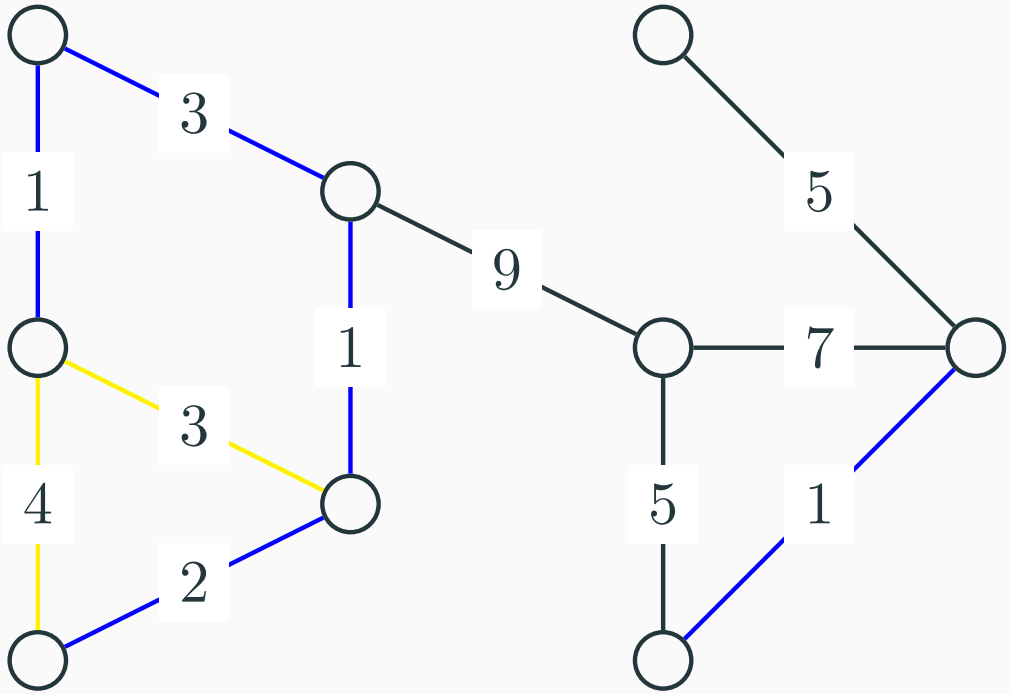
Example



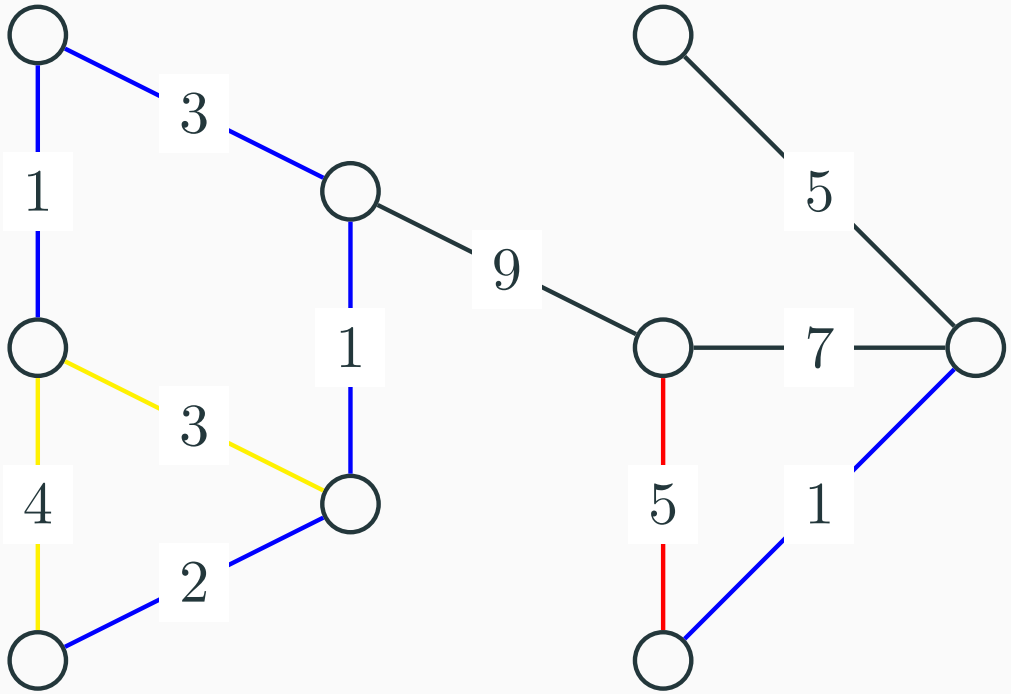
Example



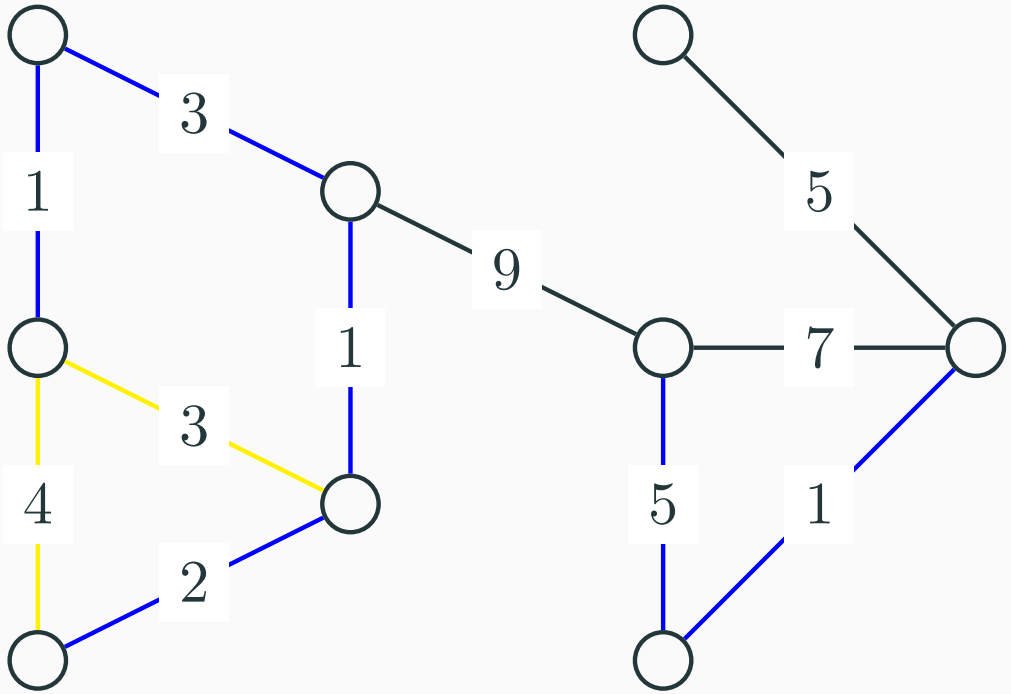
Example



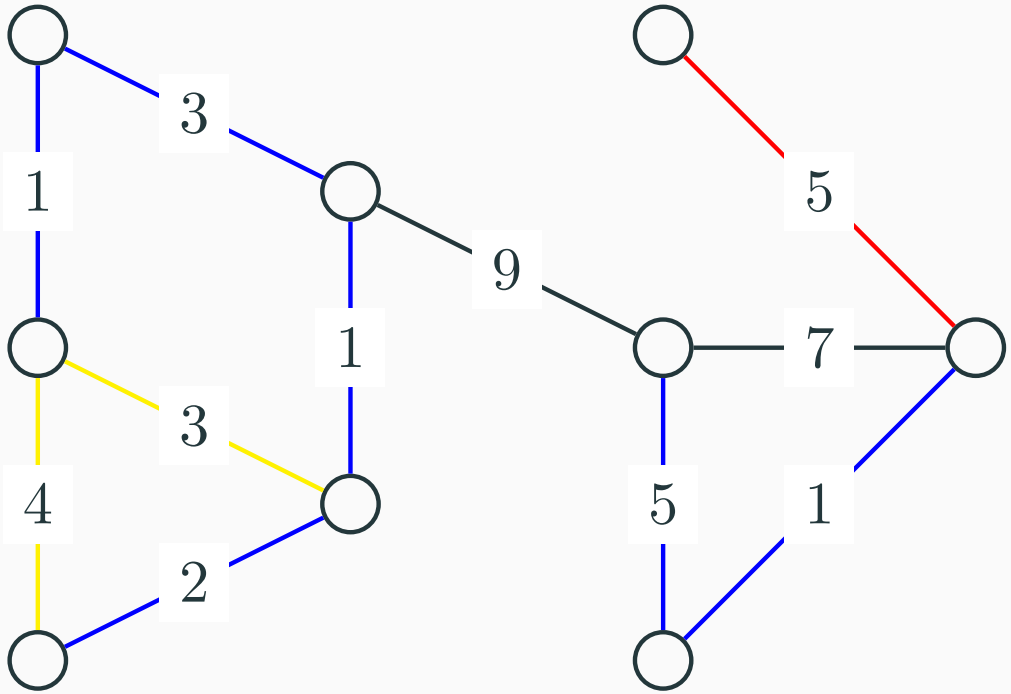
Example



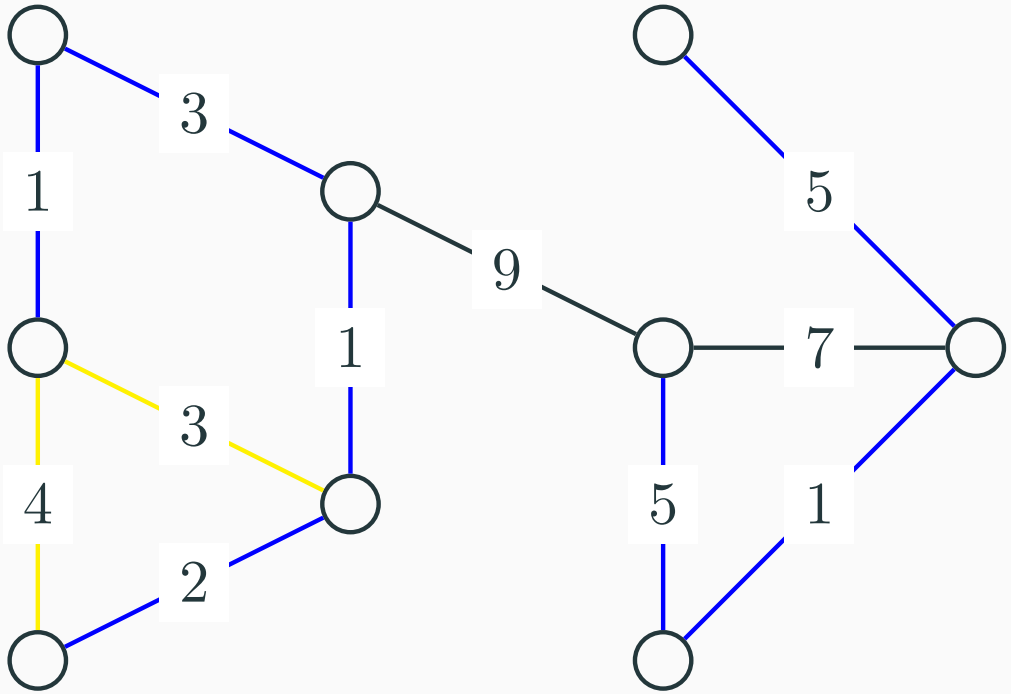
Example



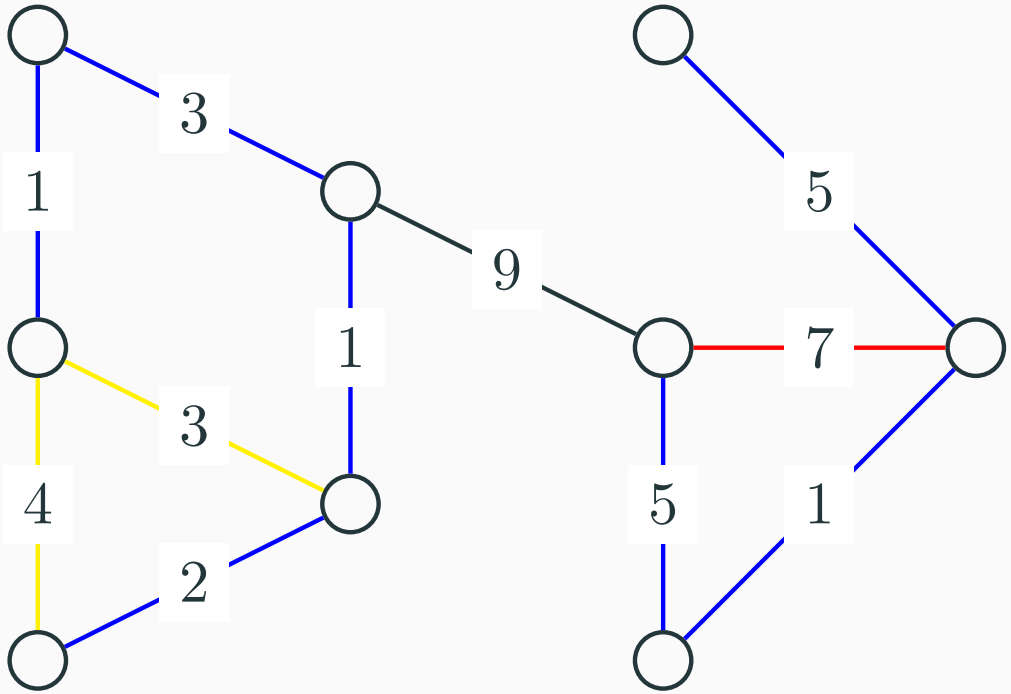
Example



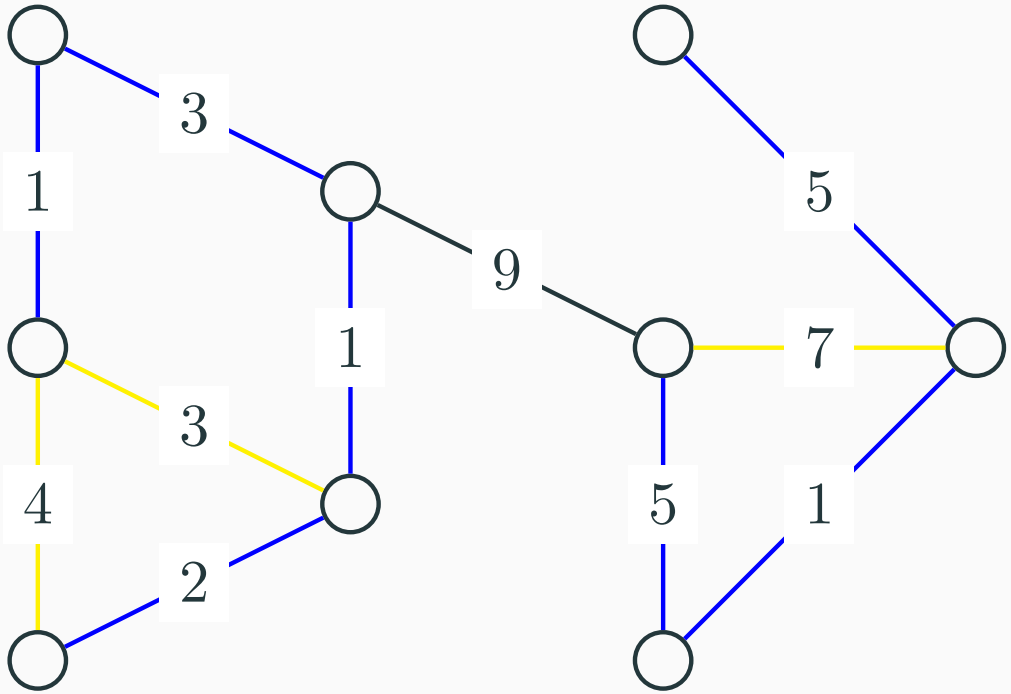
Example



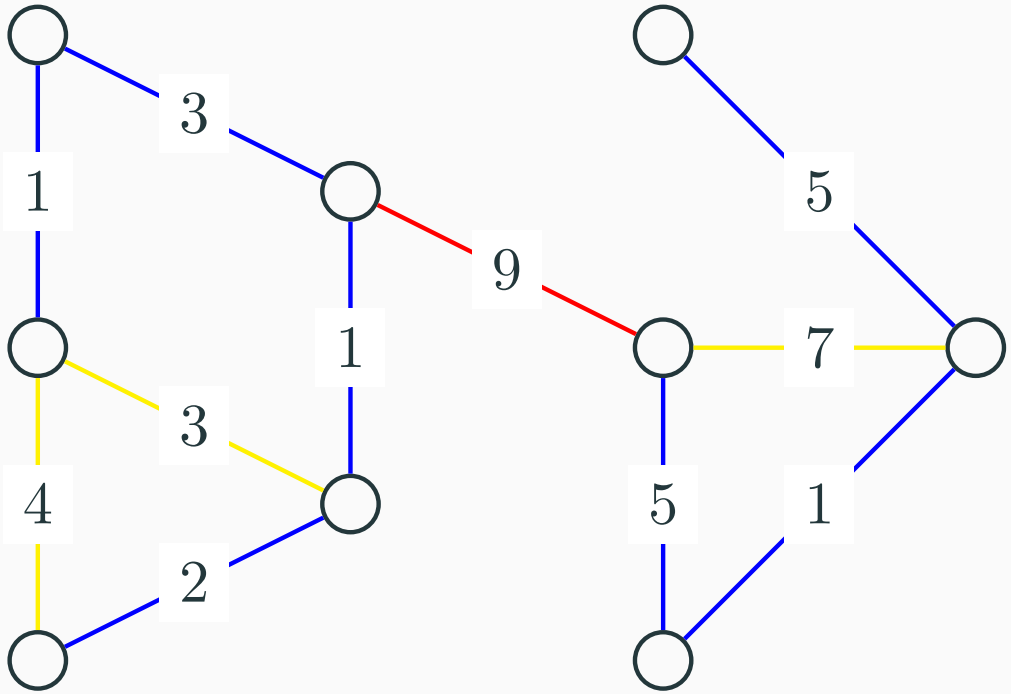
Example



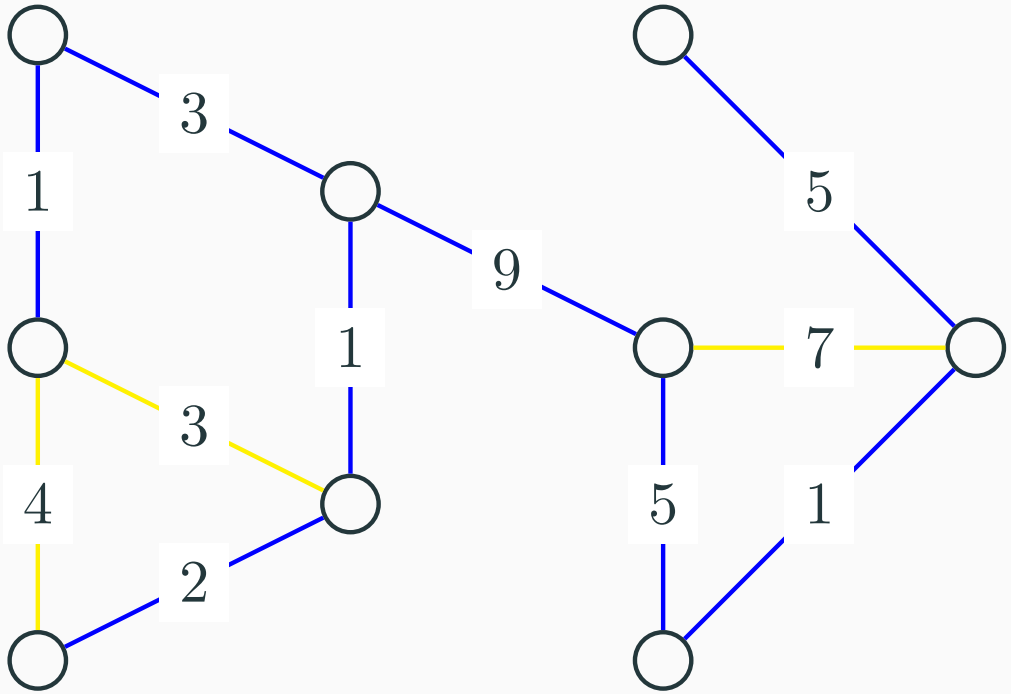
Example



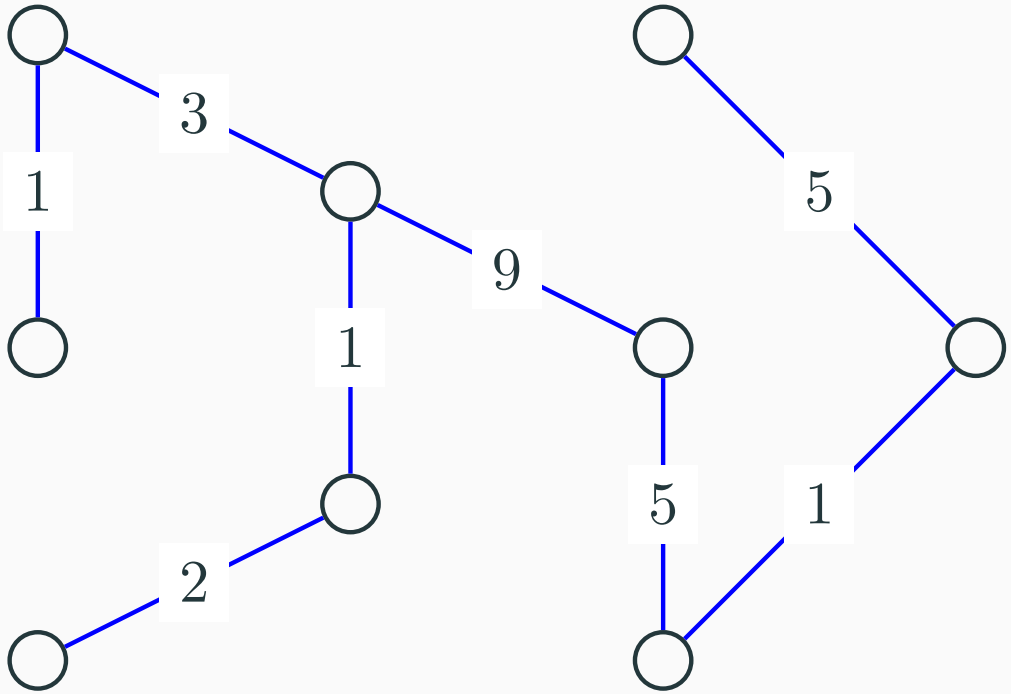
Example



Example



Example



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;
}
```

Minimum spanning tree

- Another one that works is Prim's algorithm
- We start with vertex 0, then grow the tree one vertex at a time
- Each time we take the cheapest edge that connects our tree to something not in the tree
- This can also be done in $\mathcal{O}(E \log(E))$
- For our purposes Kruskal will suffice

Single source shortest path

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 (or 0/1), 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
- We will look at Dijkstra's algorithm, the Bellman-Ford algorithm and the Floyd-Warshall algorithm

Shortest paths

- We will start with the simplest case, where no weights are negative and we want the distance from a particular vertex to the other ones
- Why no negative weights? Well if there is a cycle with negative weight on the way to our destination there is no shortest path! You can always go a "shorter" way by cycling around the negative cycle an extra time
- In principle Dijkstra is not so dissimilar from BFS

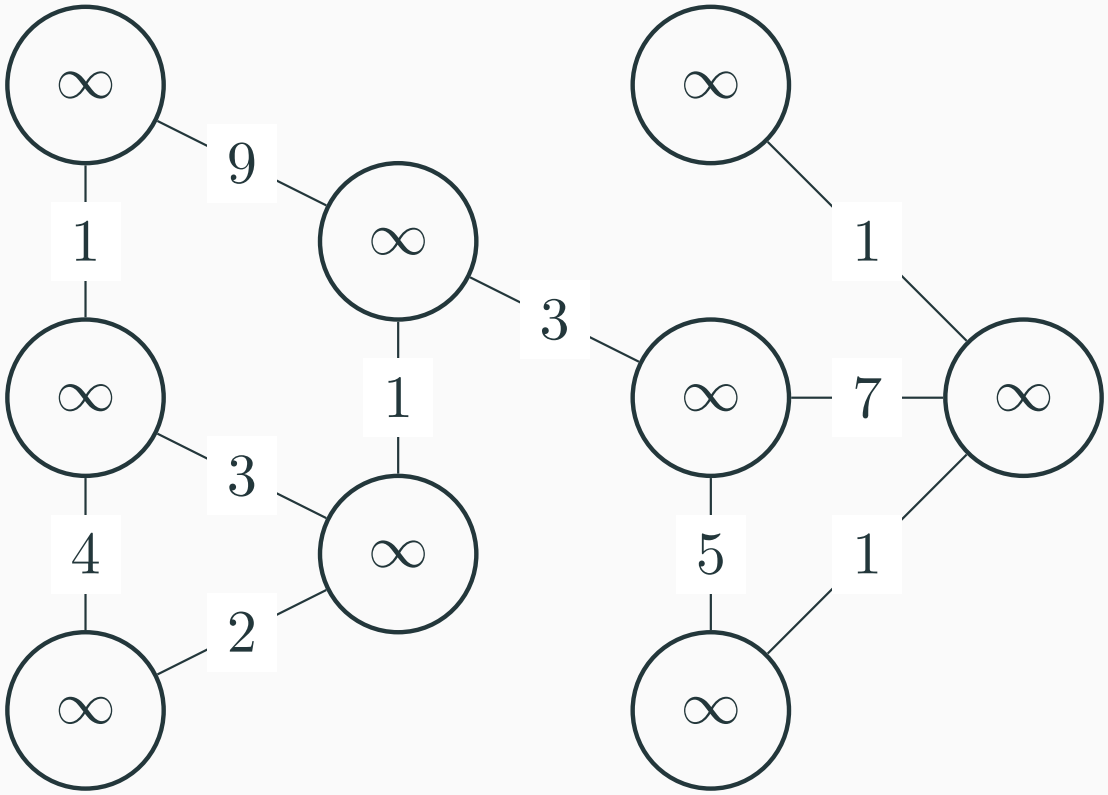
Shortest paths

- In BFS the next item in our queue was always the closest one to the source vertex we had not processed yet
- In Dijkstra's algorithm we have to do some extra work to ensure this is the case
- We use a min-heap, so that the next vertex we pick is always the closest unprocessed one

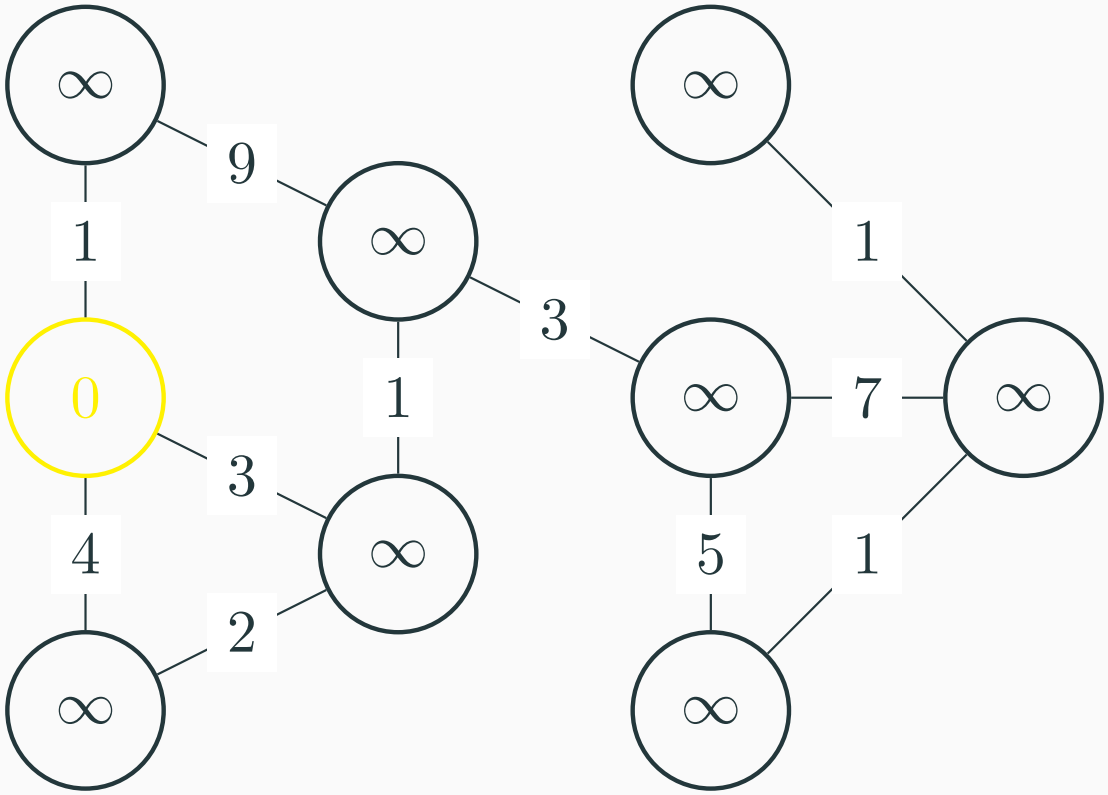
Shortest paths

- We initialize the source vertex to distance 0, others to ∞
- We then at any point take the closest vertex v that we haven't processed
- We then check all edges adjacent to v and update the distance to the neighbours
- We then mark v as done and don't check it again
- Note that if all weights are one this will just do the same things as BFS

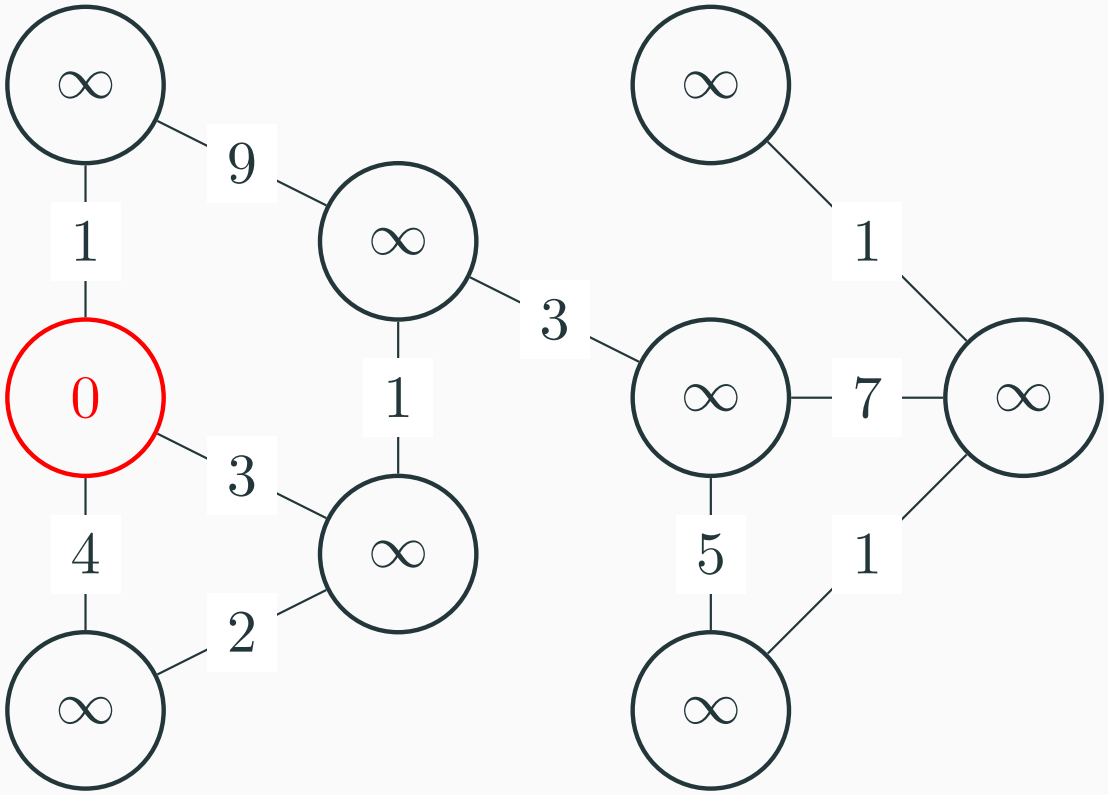
Example



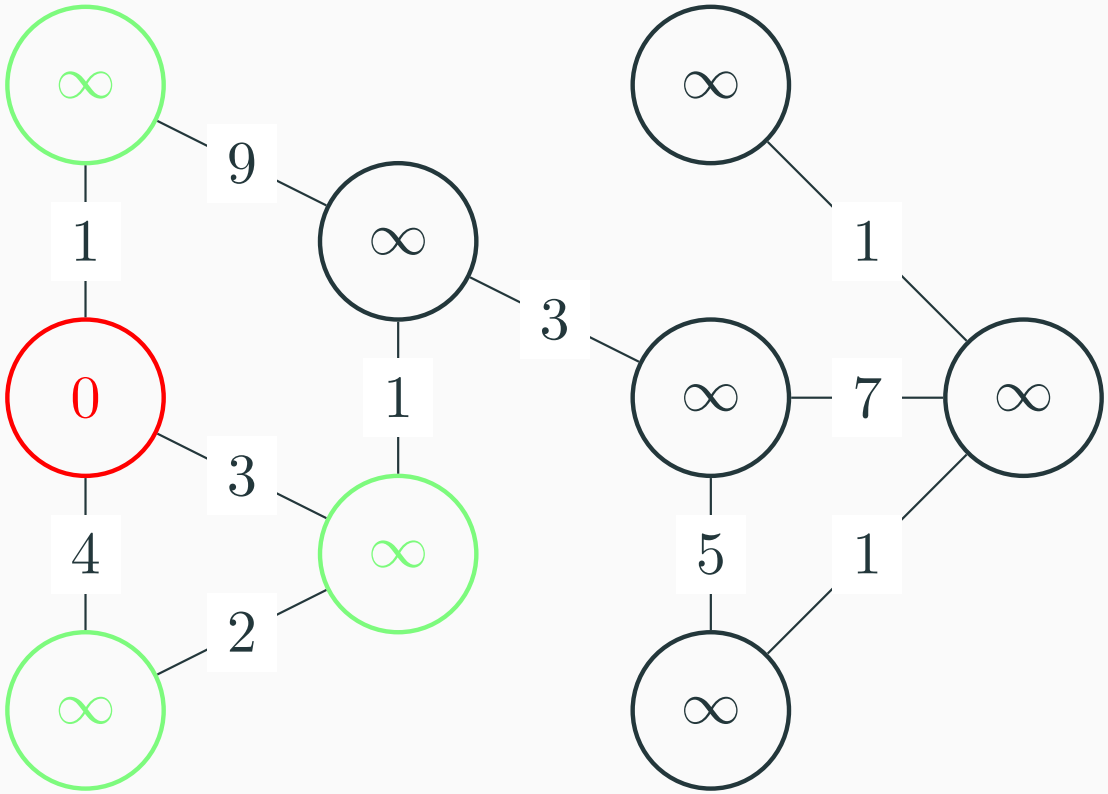
Example



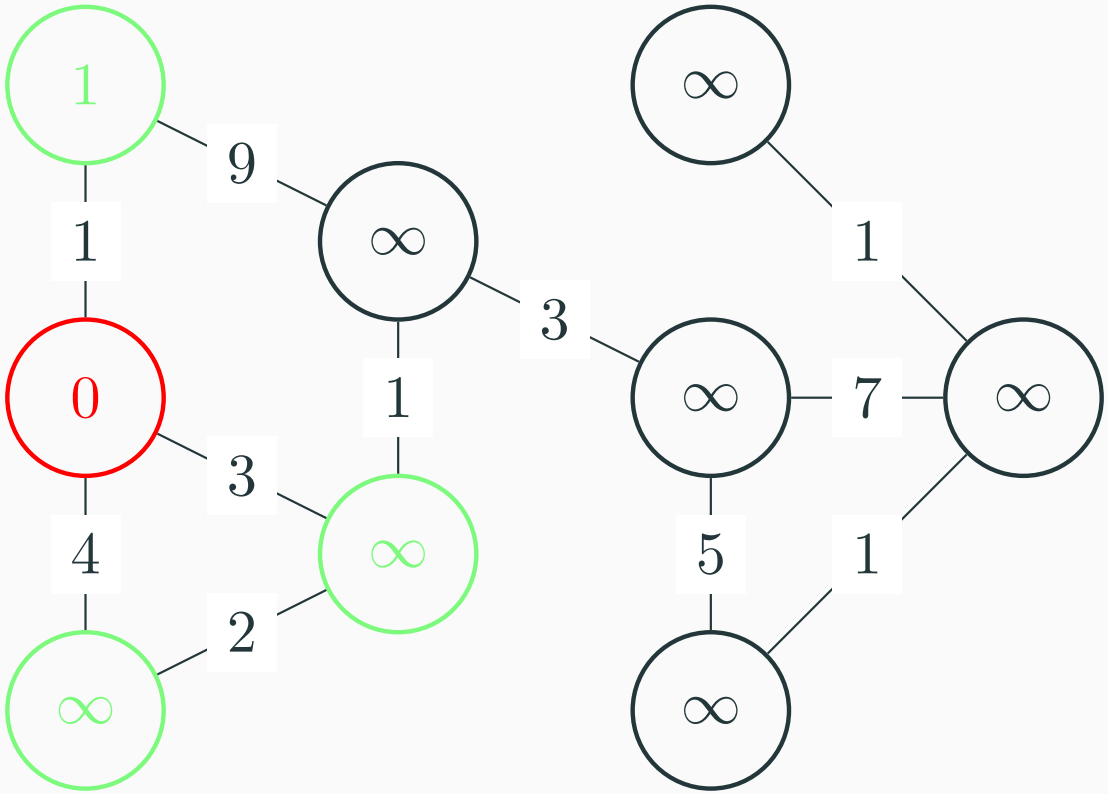
Example



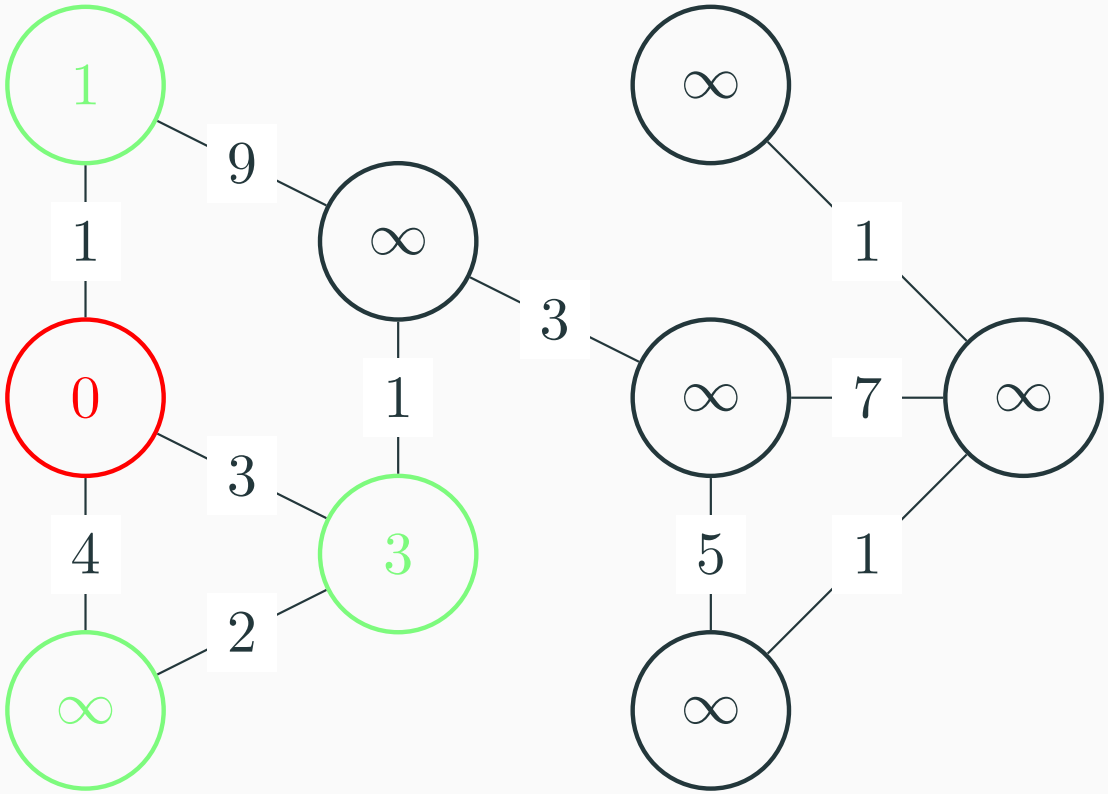
Example



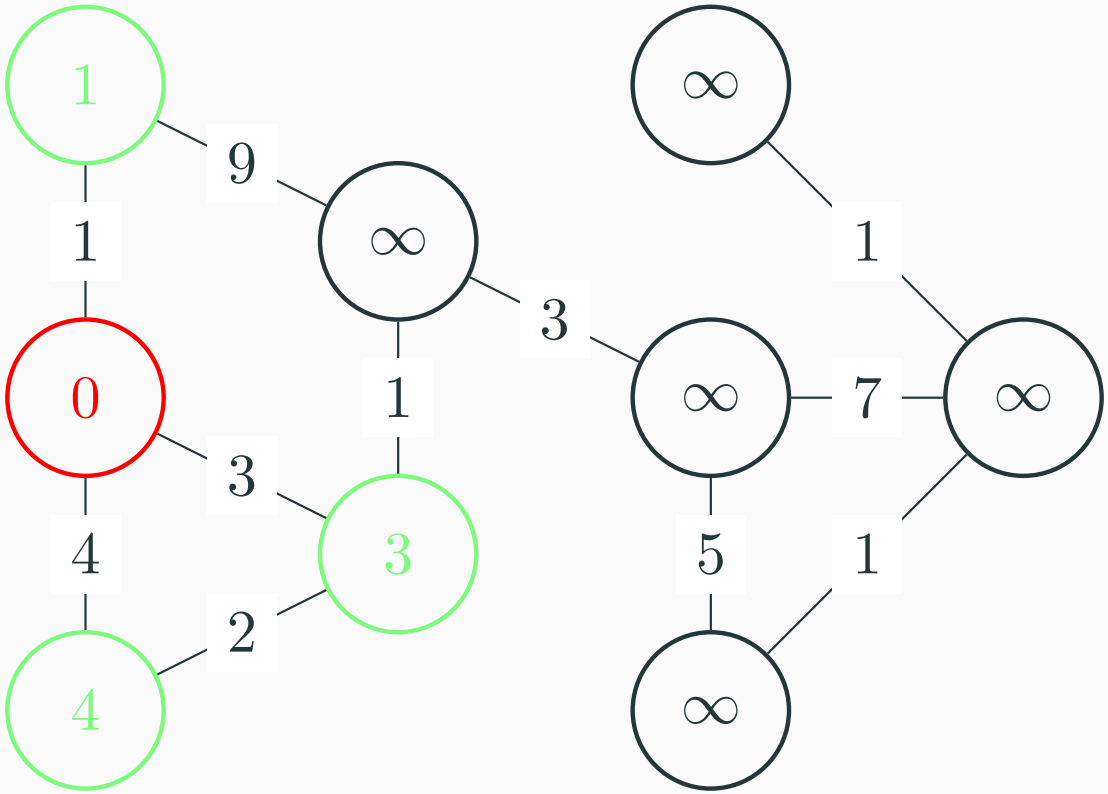
Example



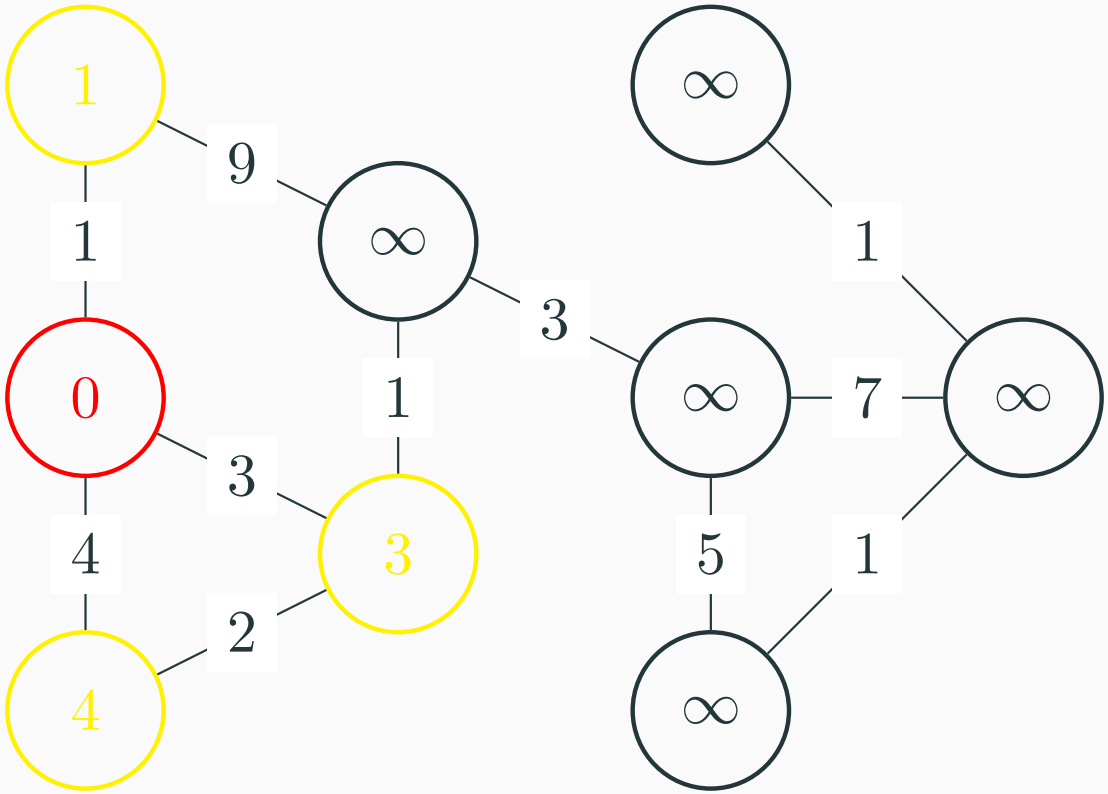
Example



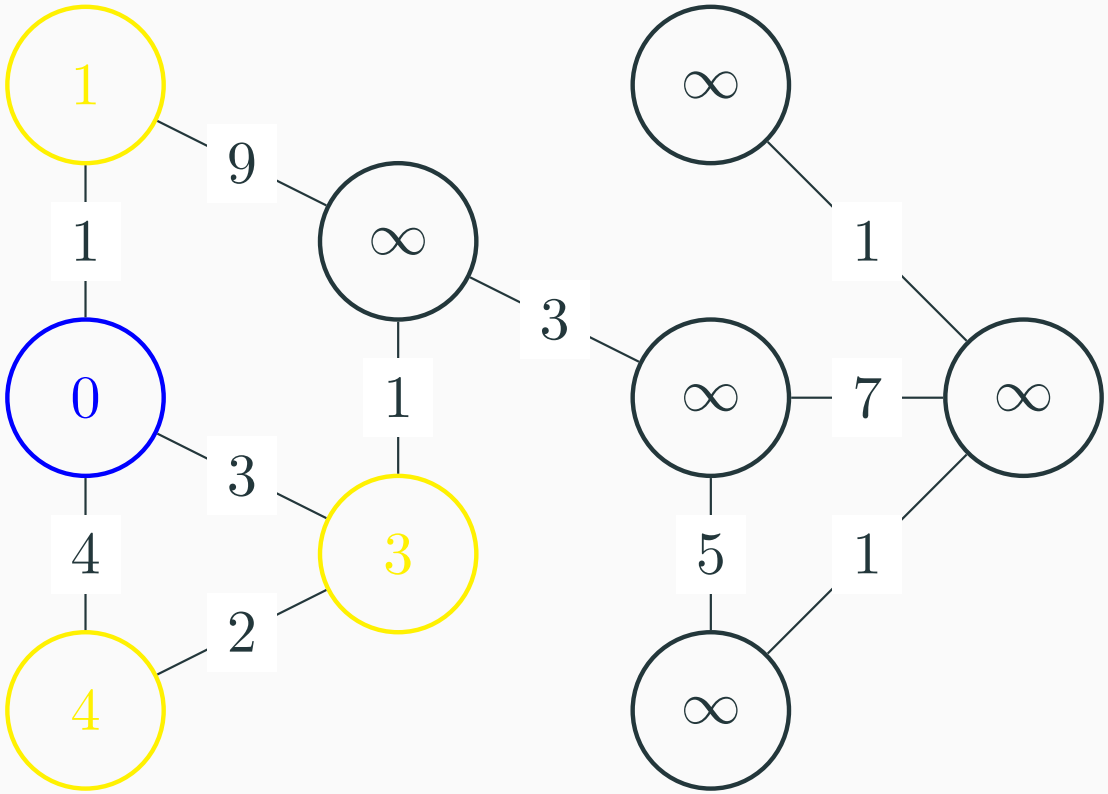
Example



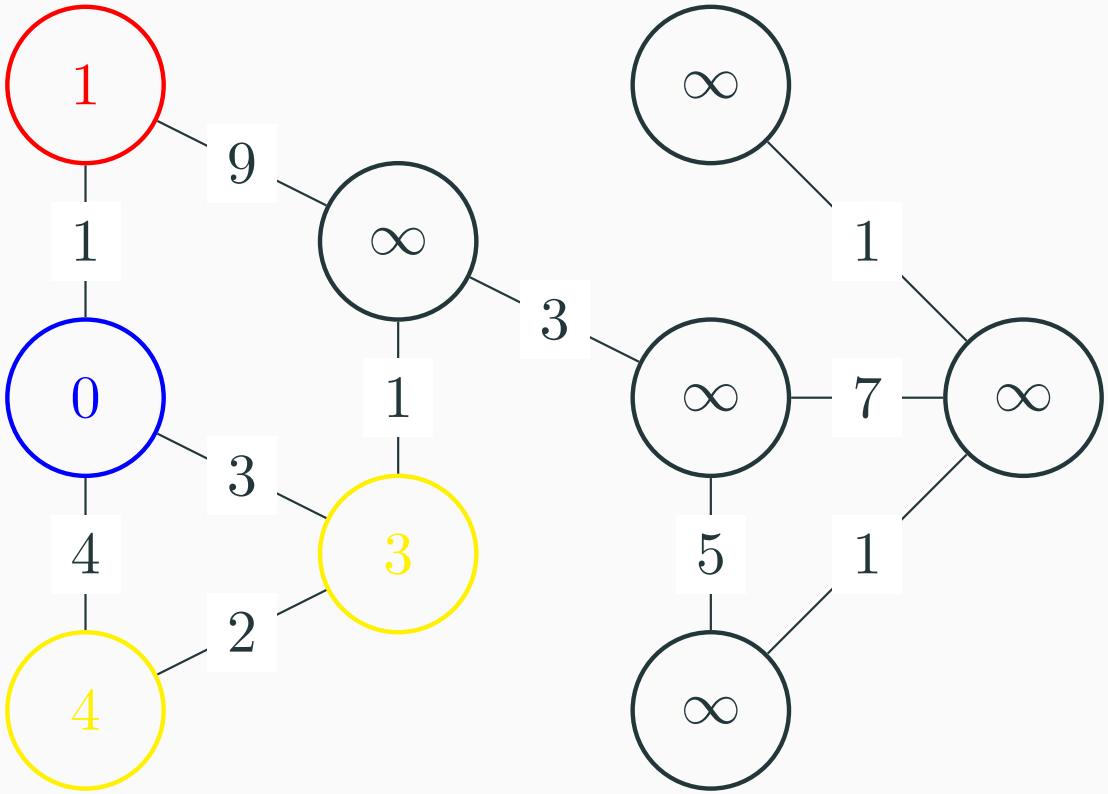
Example



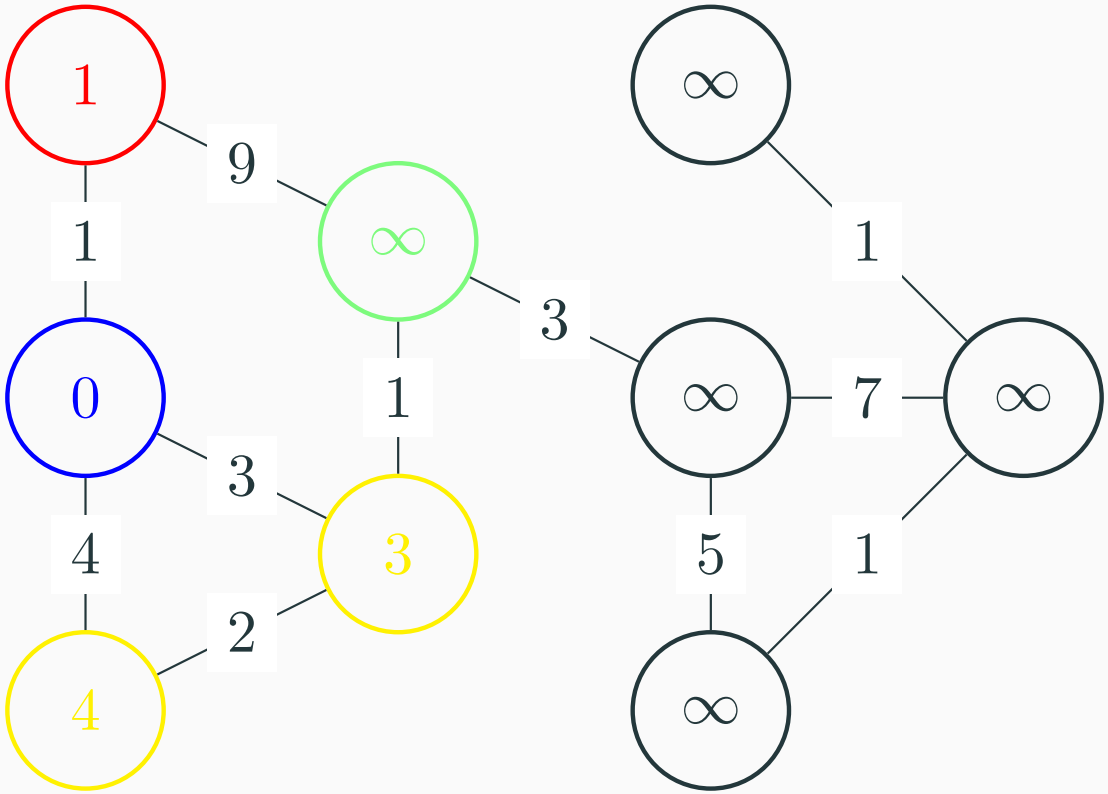
Example



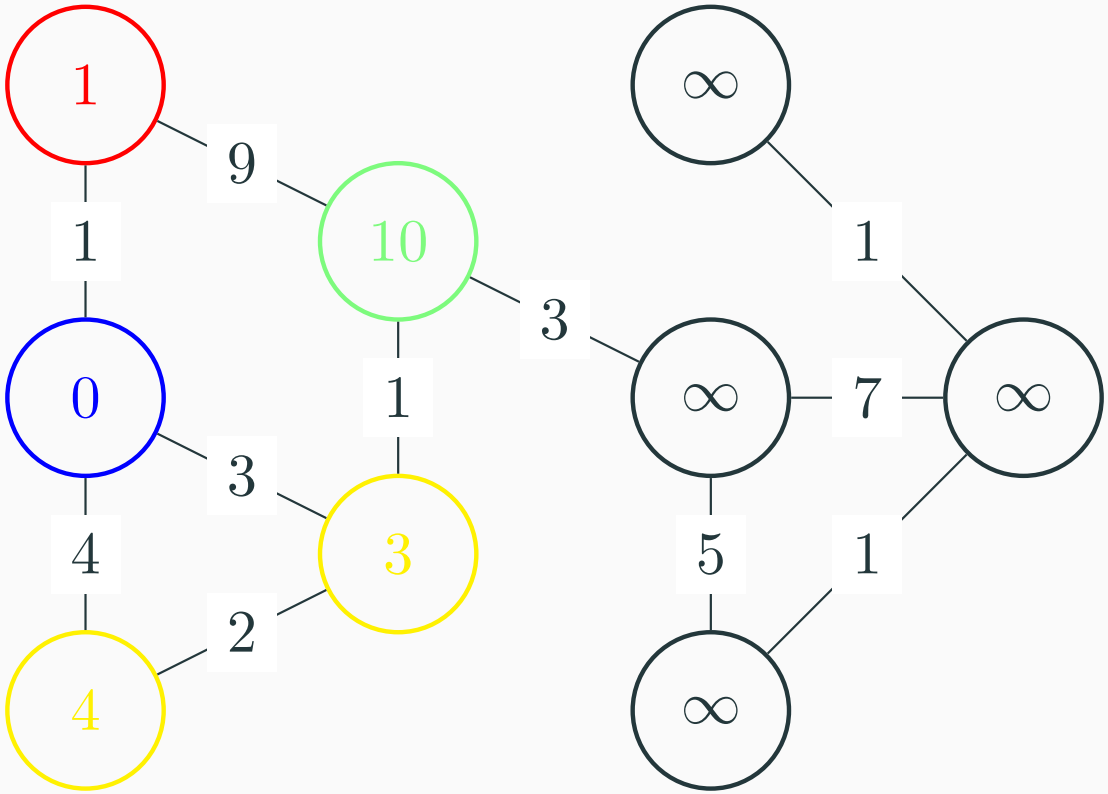
Example



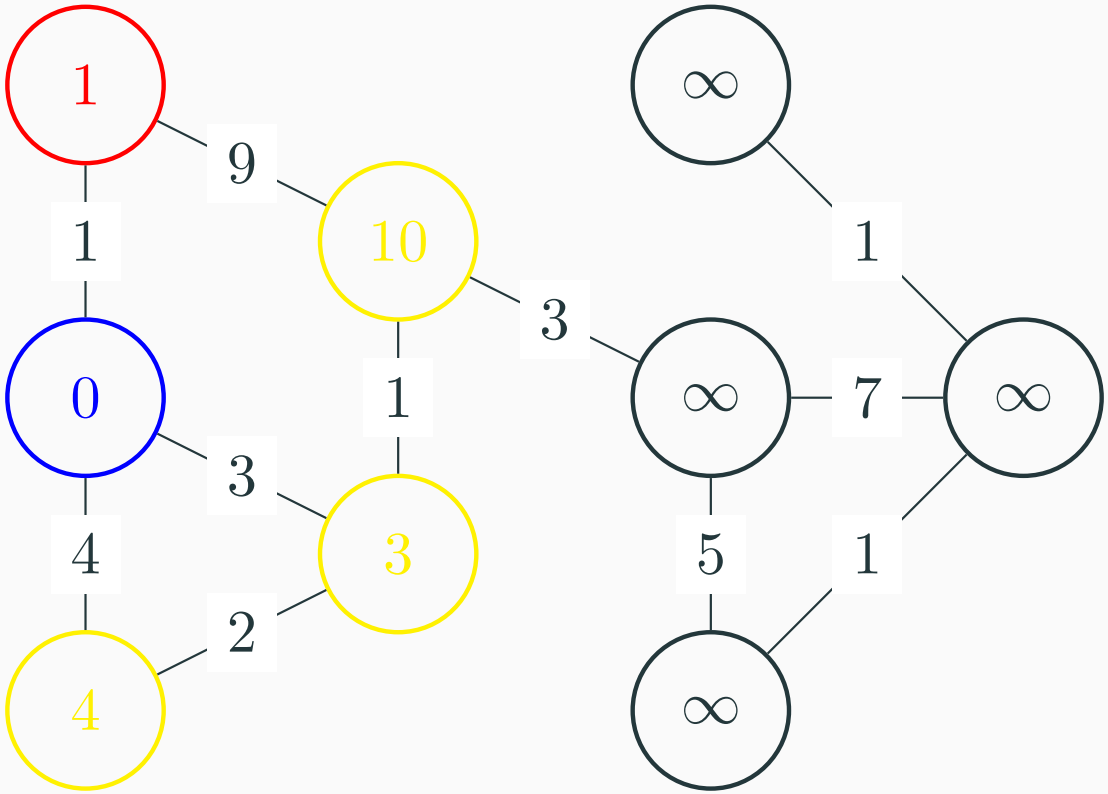
Example



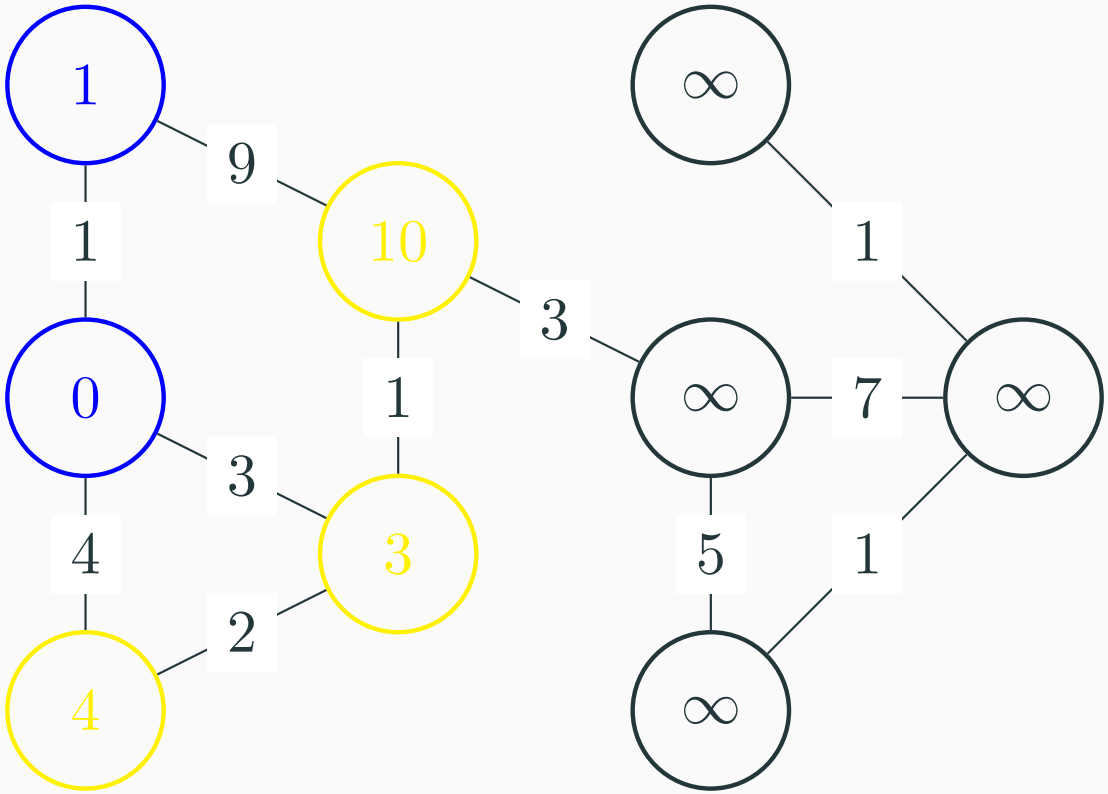
Example



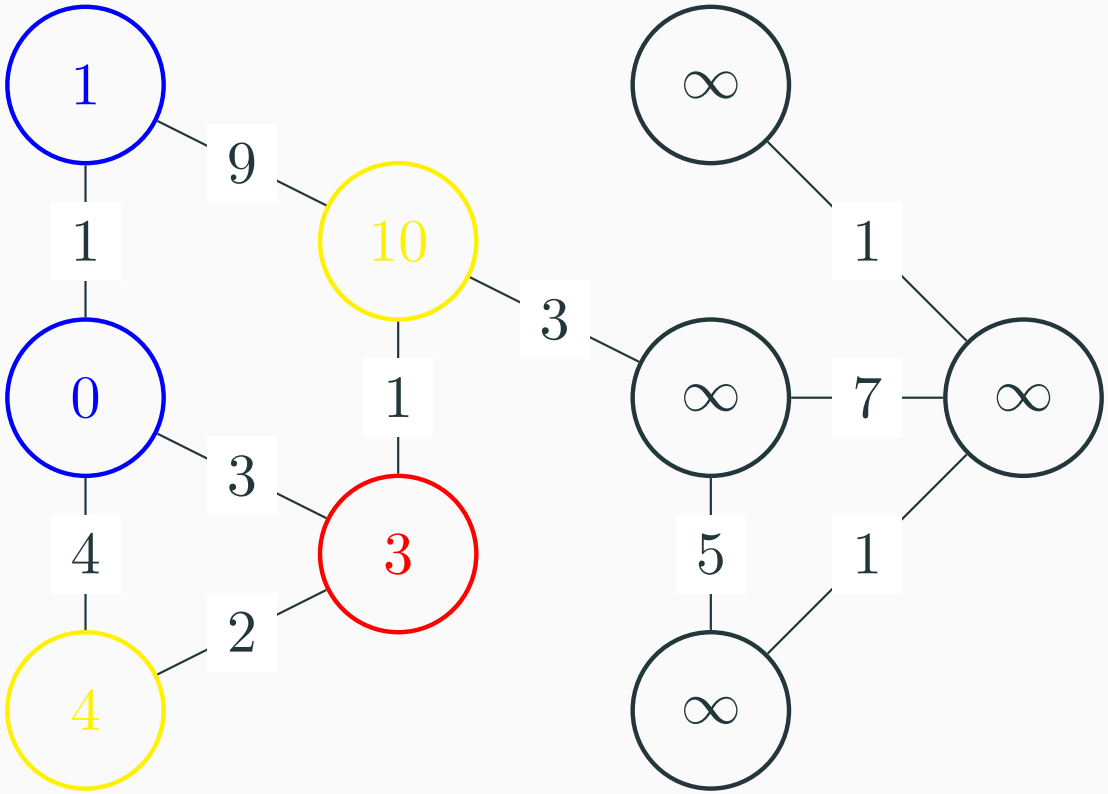
Example



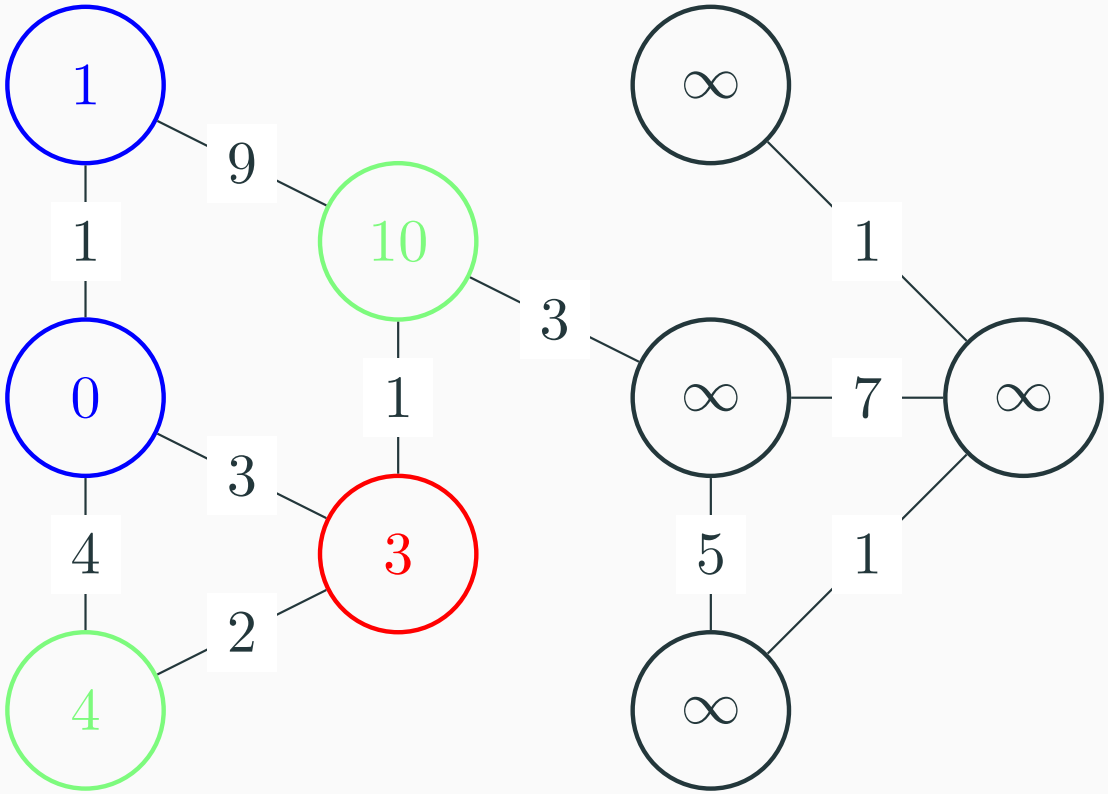
Example



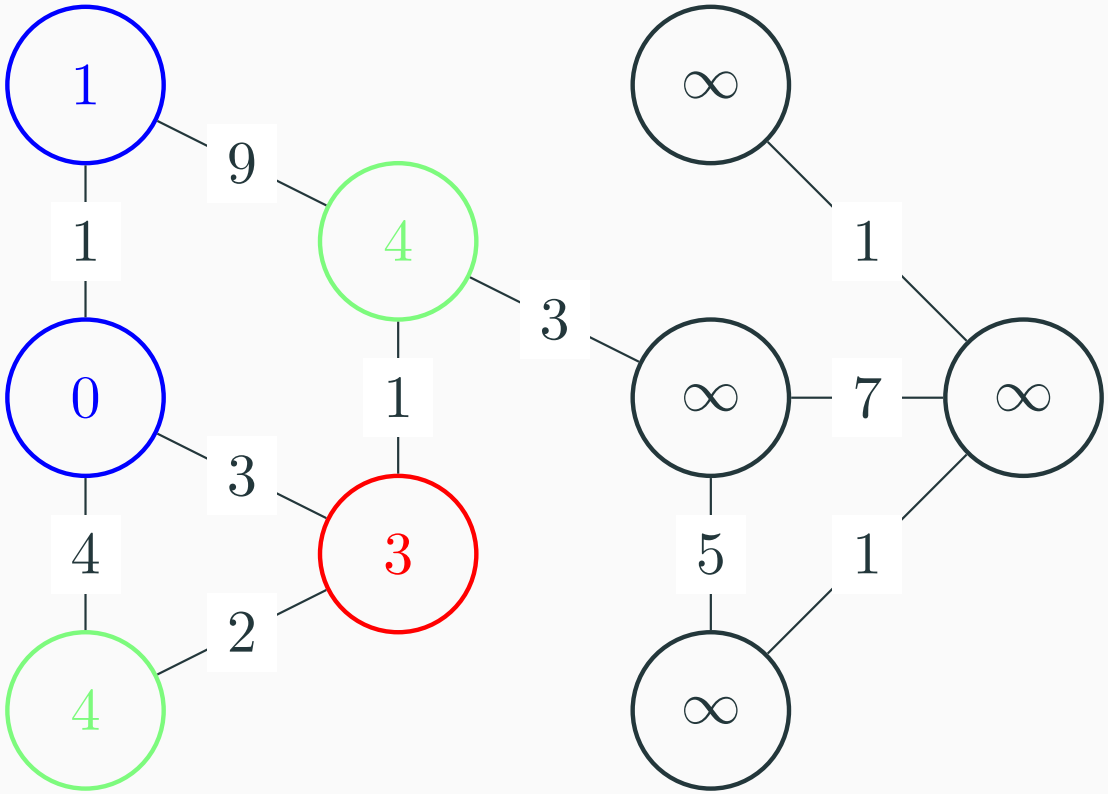
Example



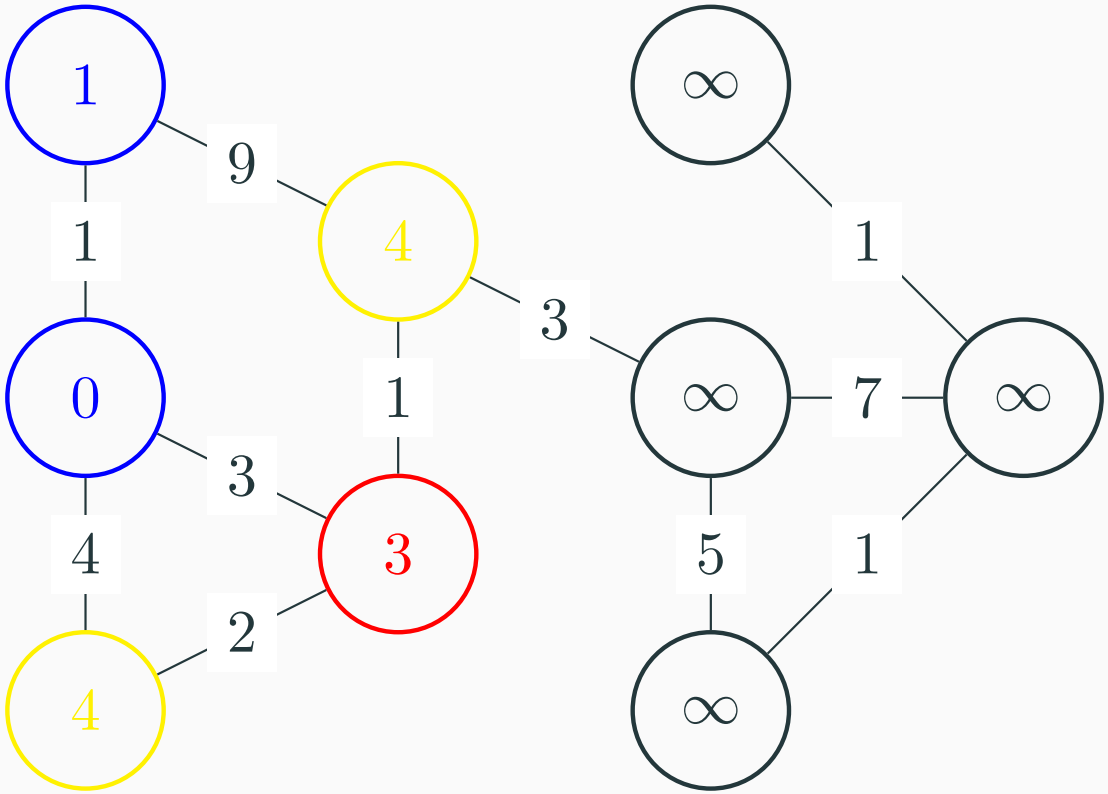
Example



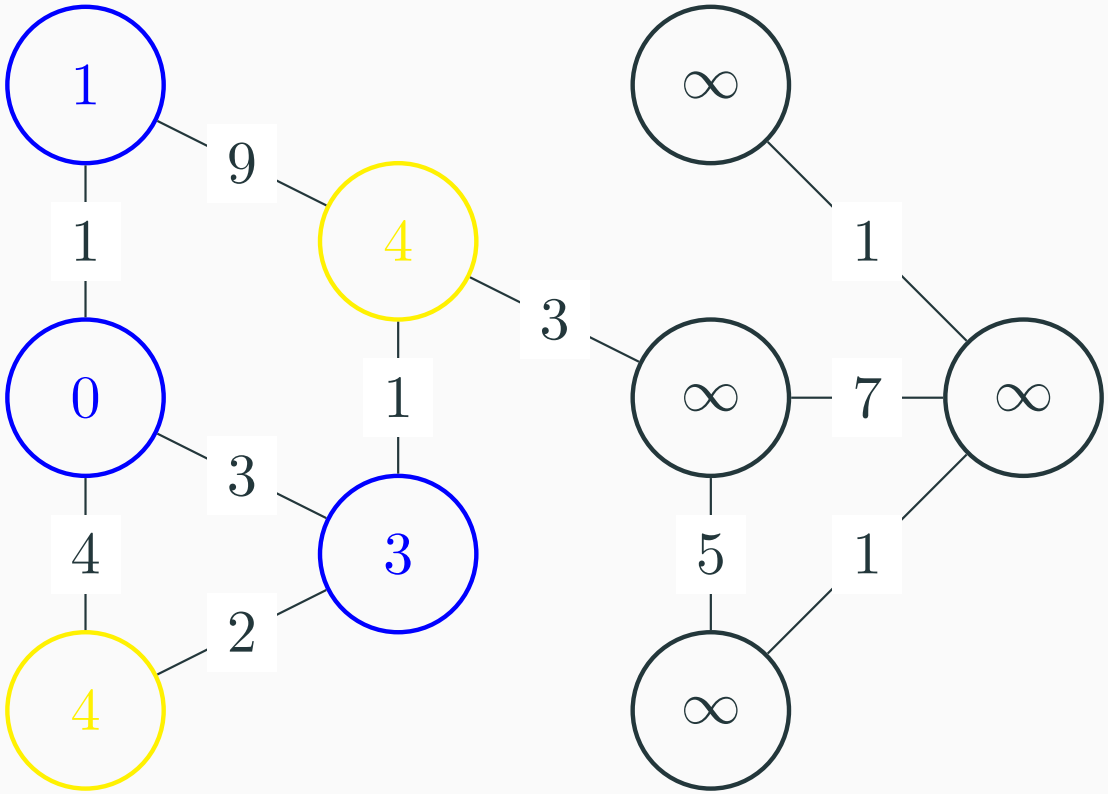
Example



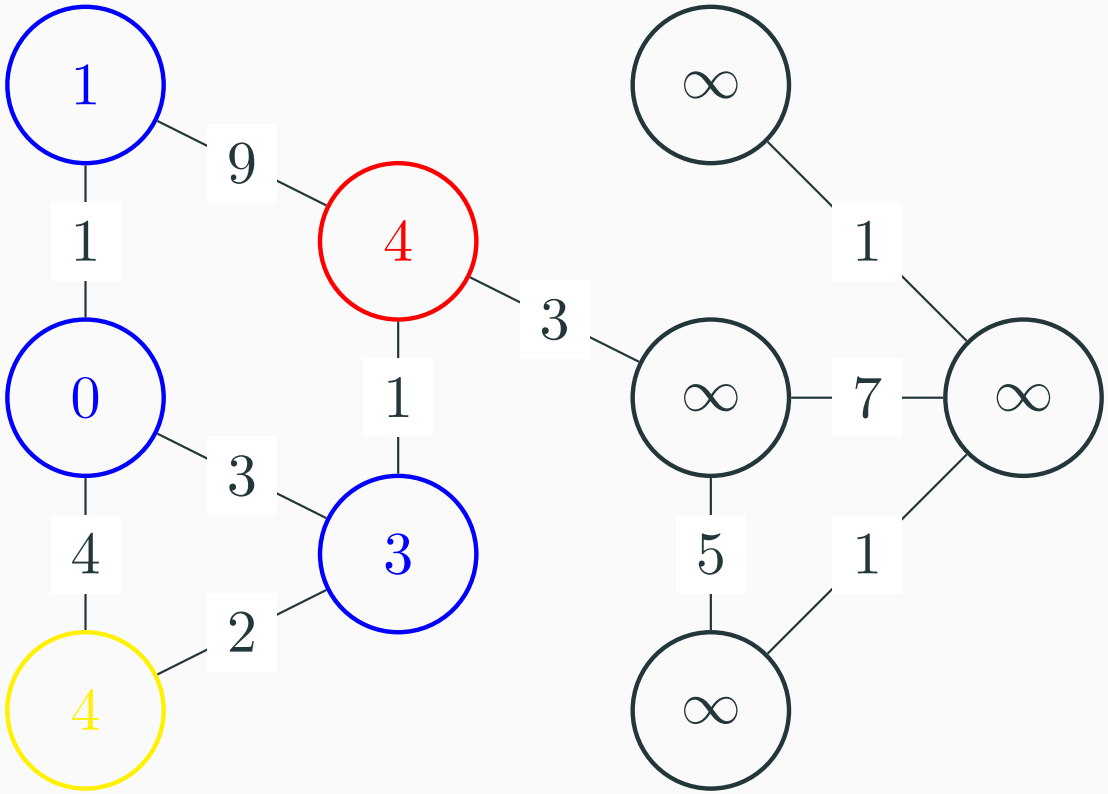
Example



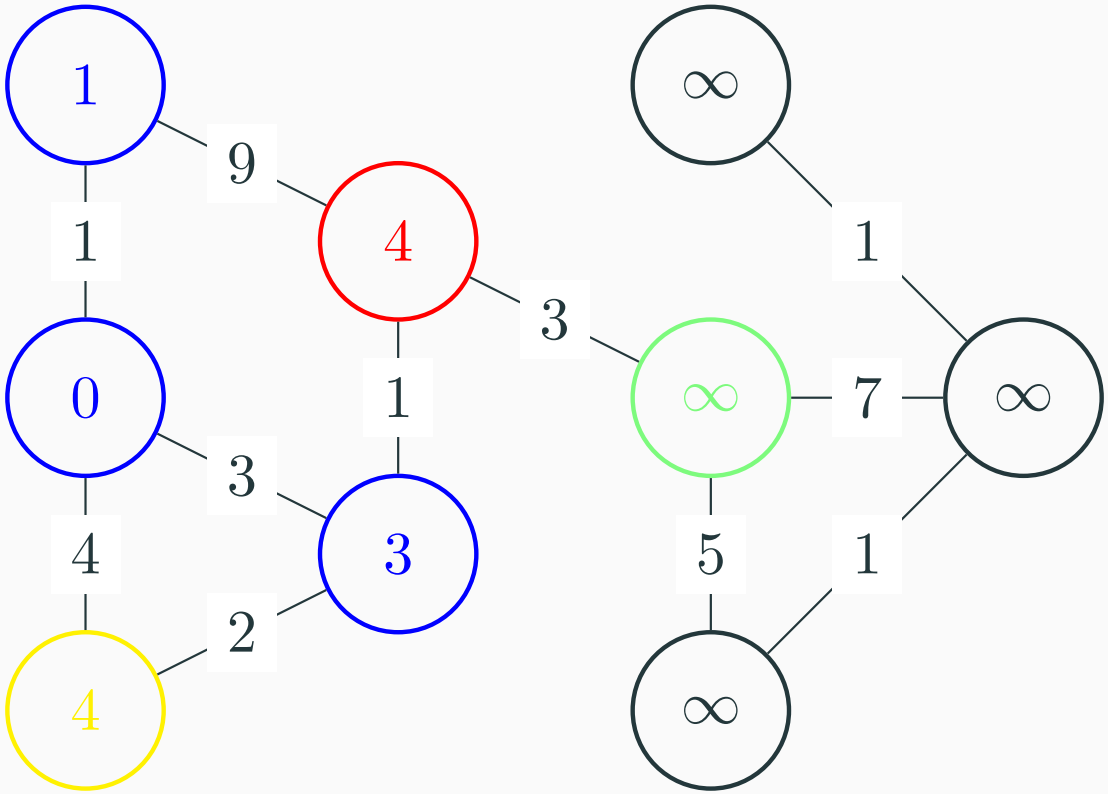
Example



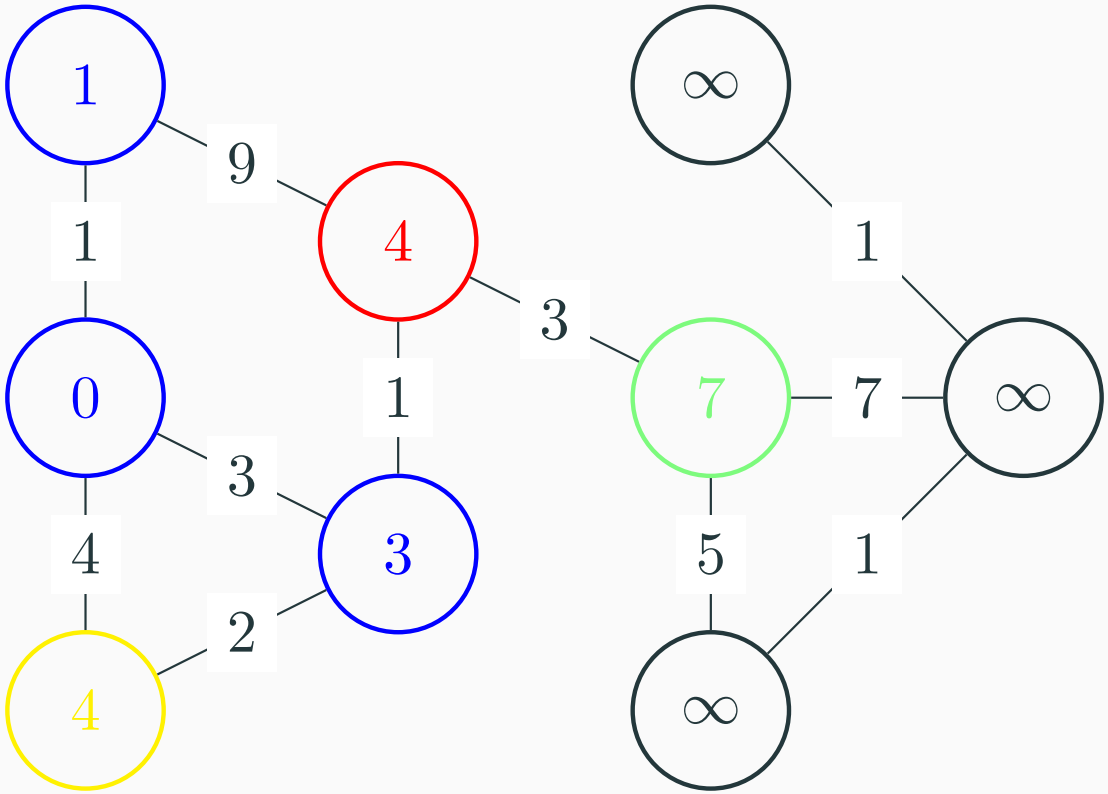
Example



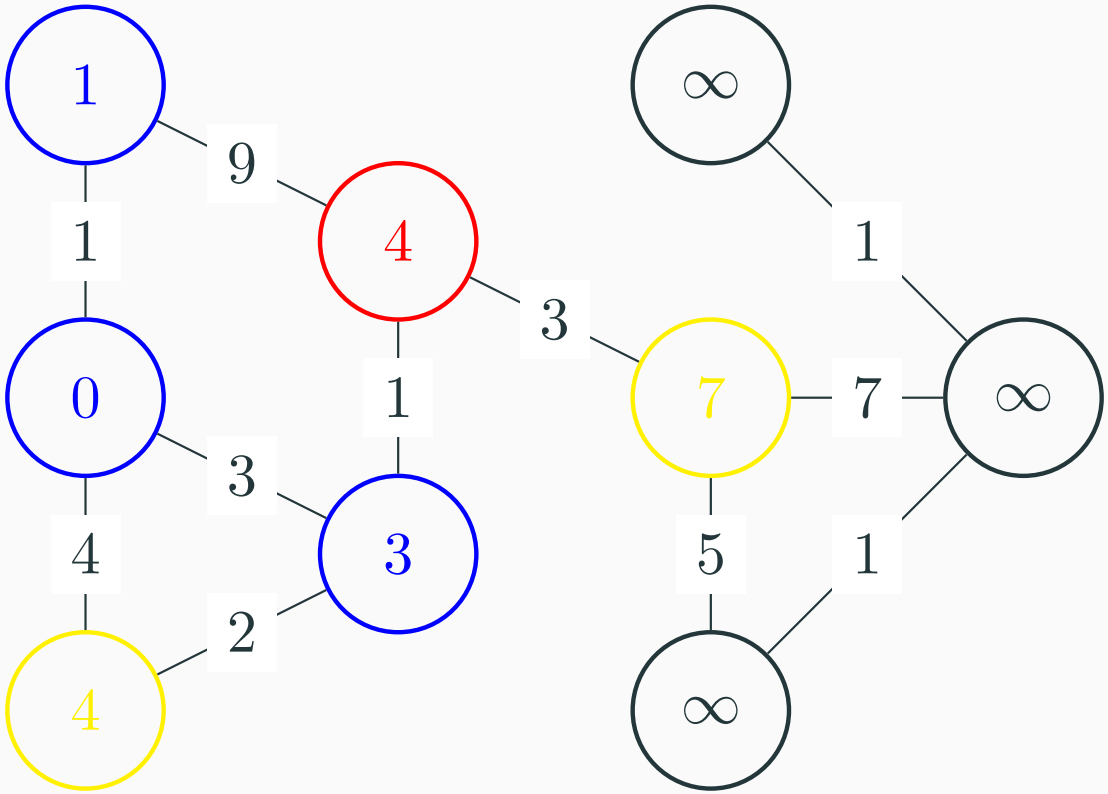
Example



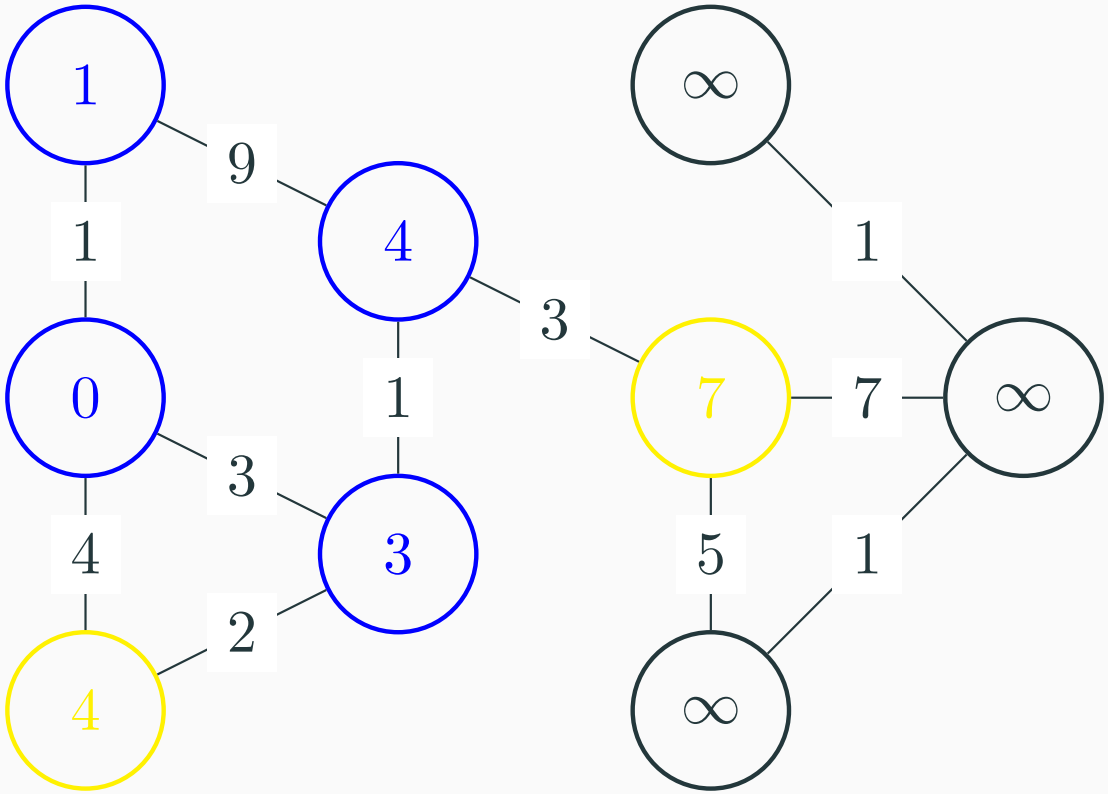
Example



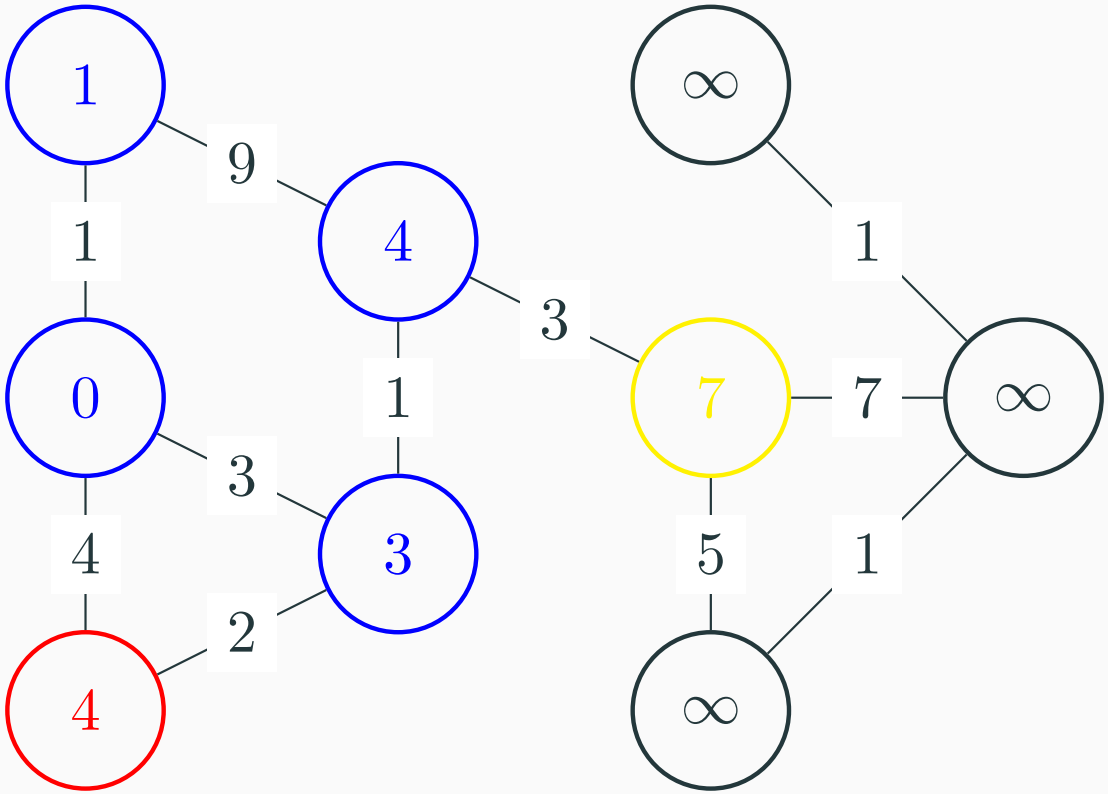
Example



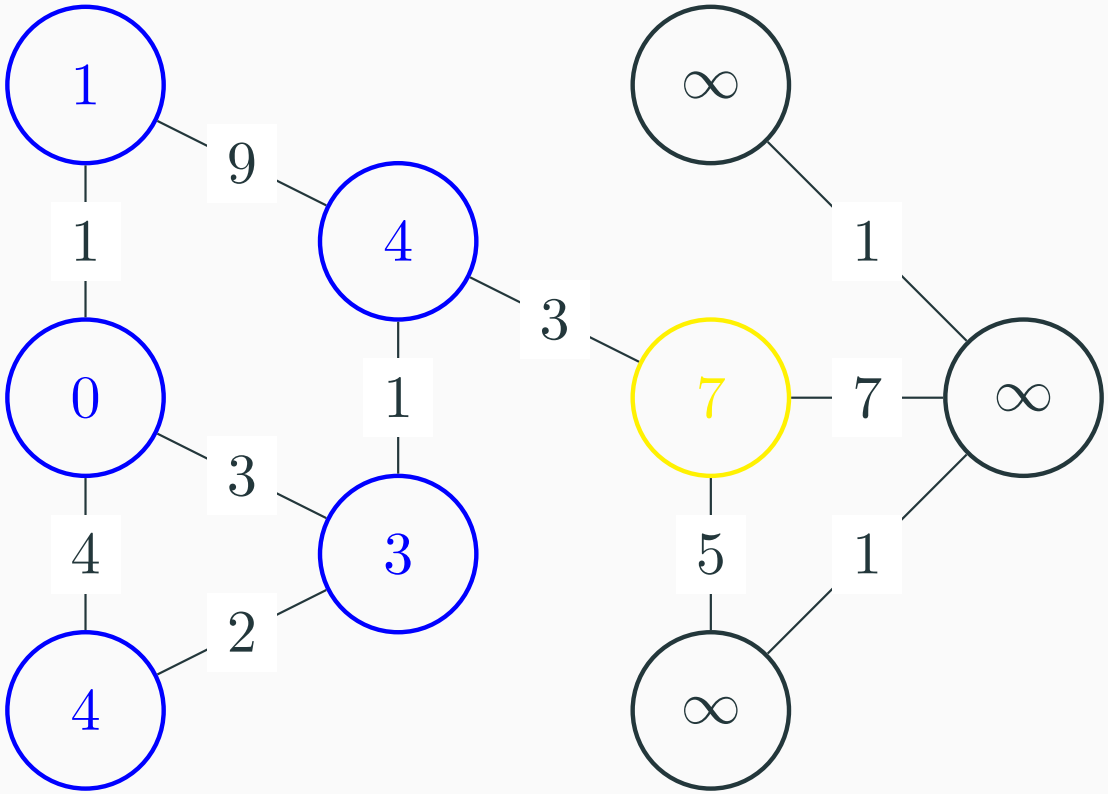
Example



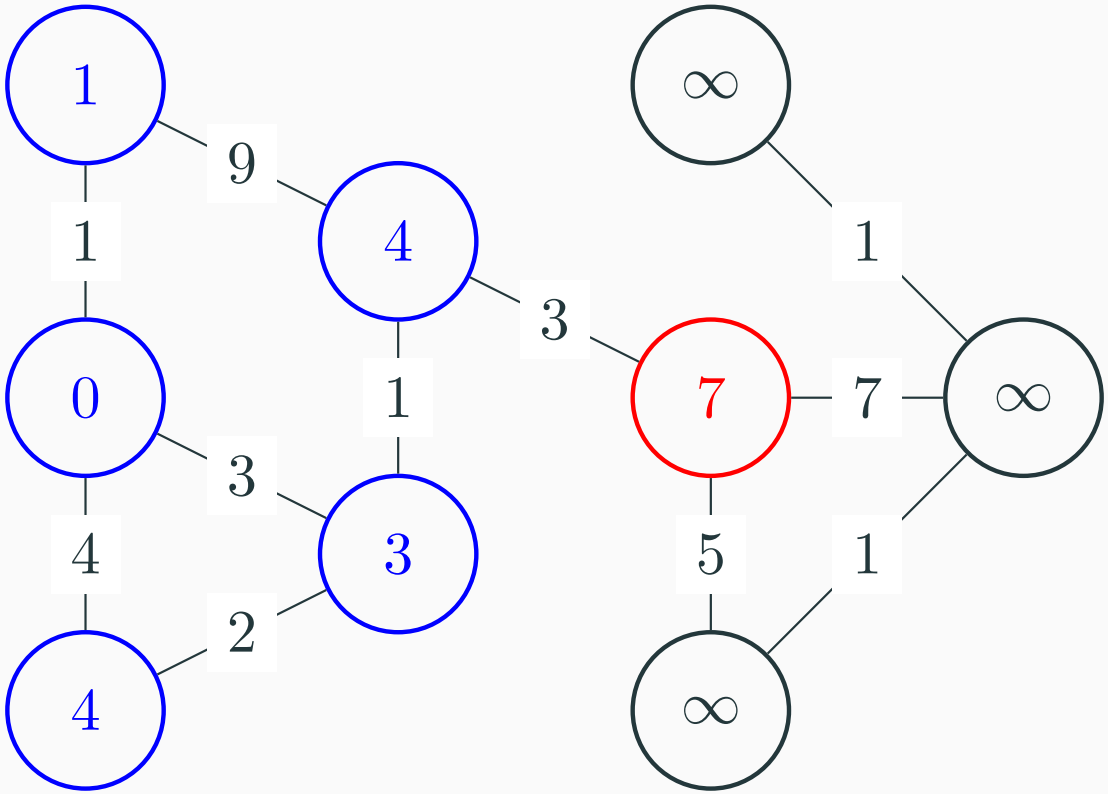
Example



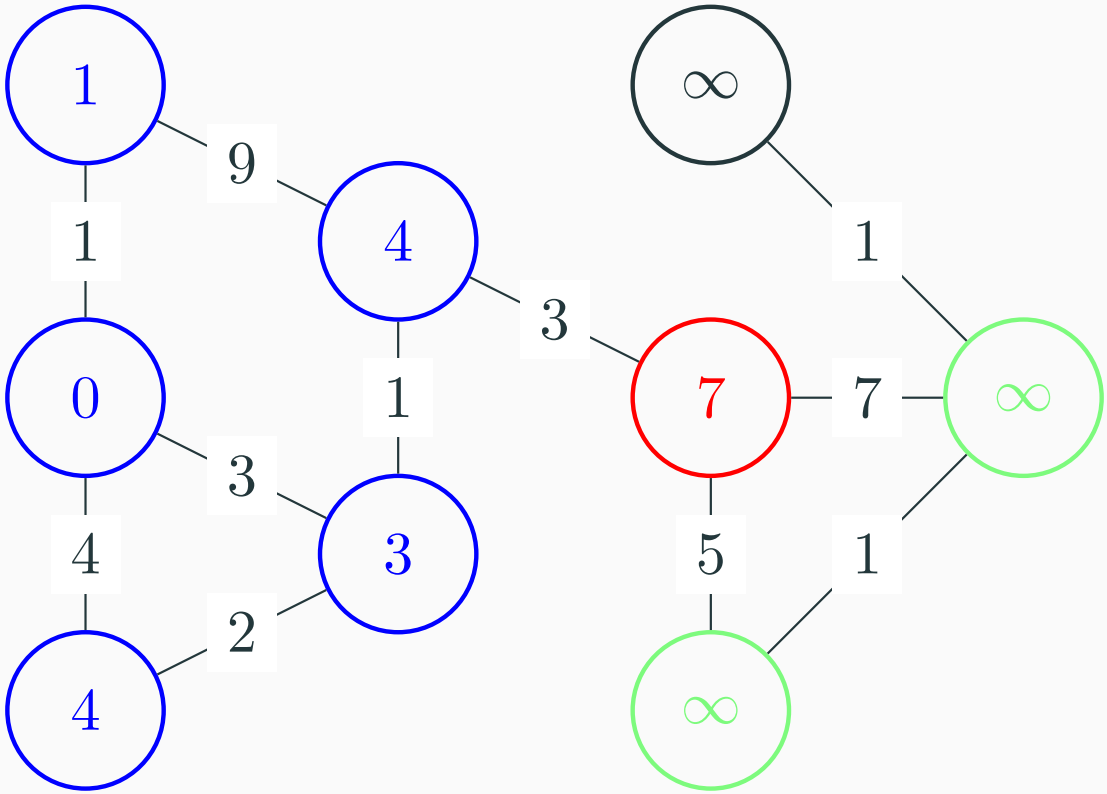
Example



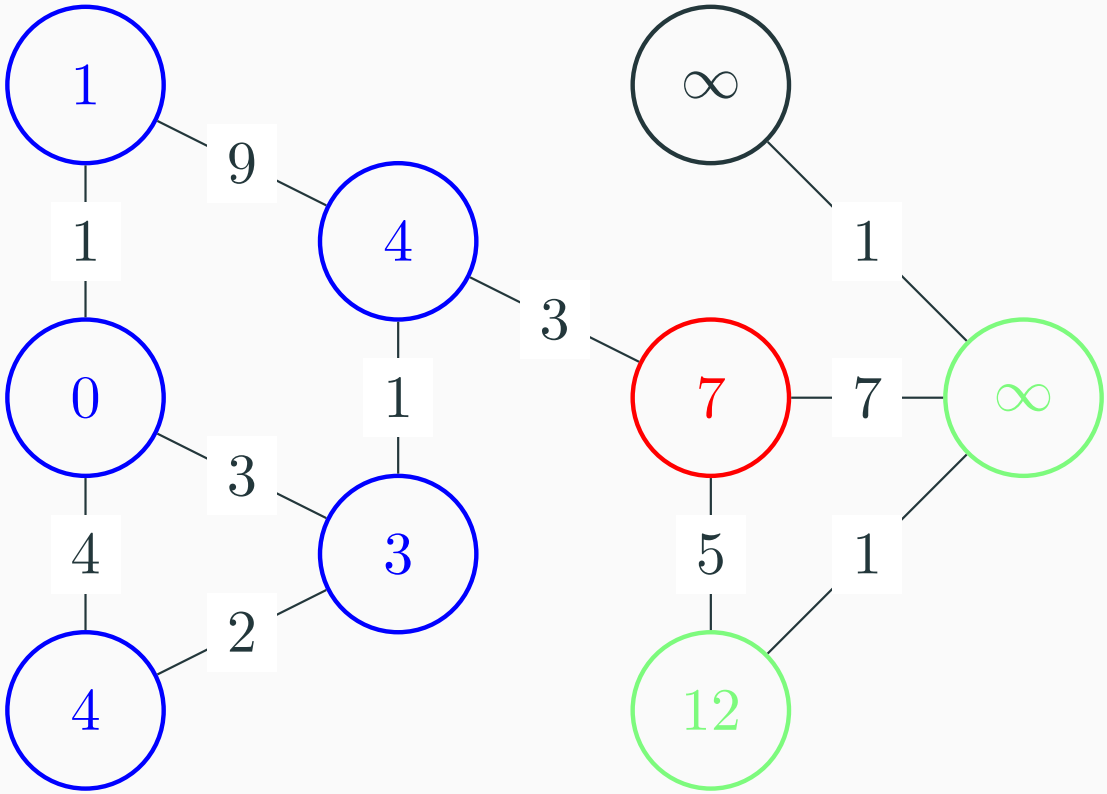
Example



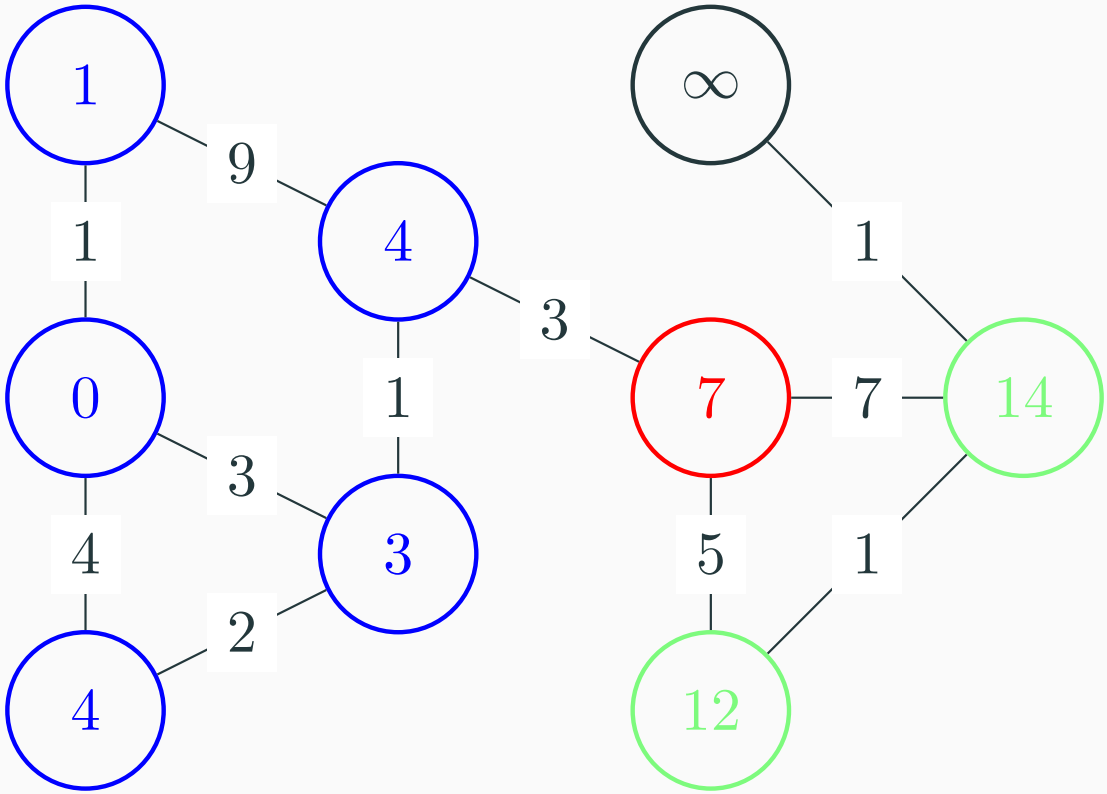
Example



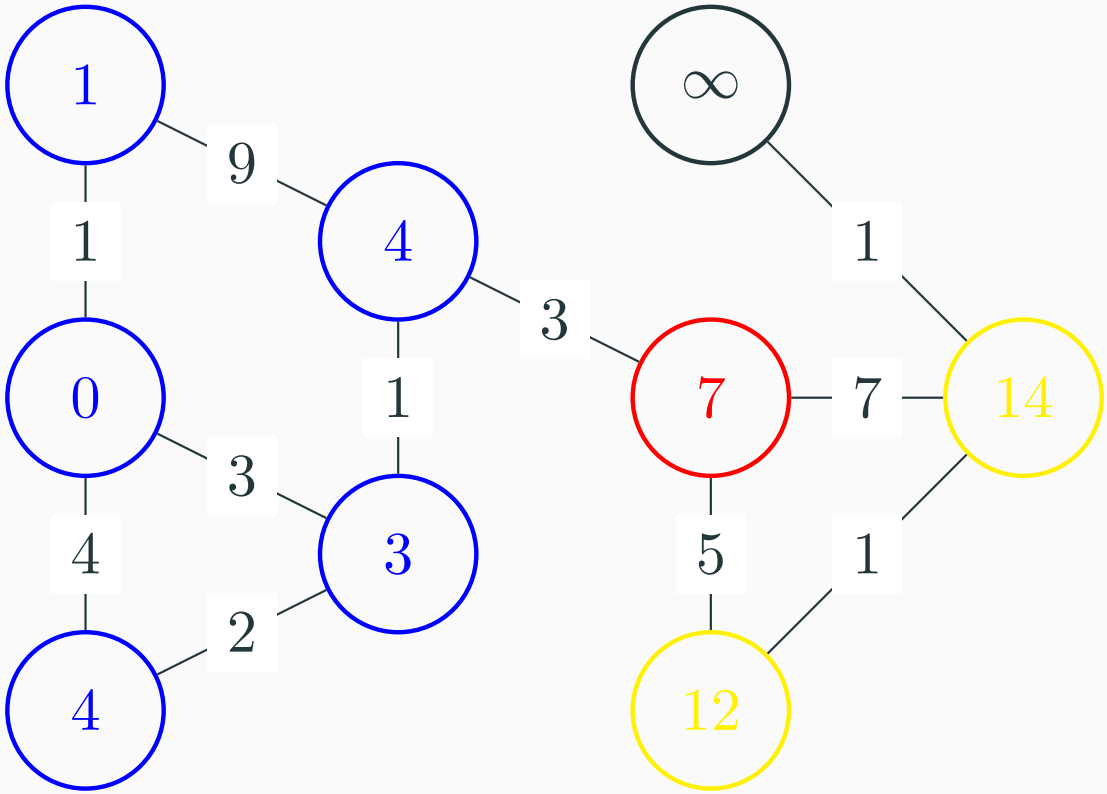
Example



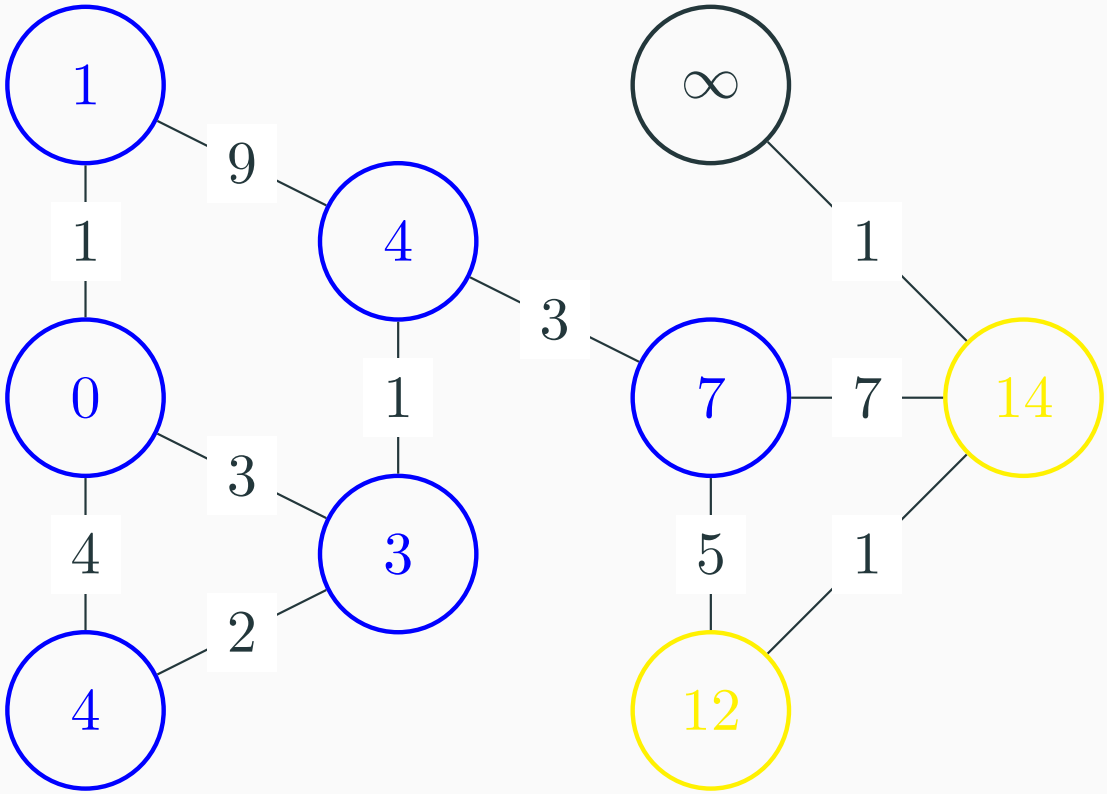
Example



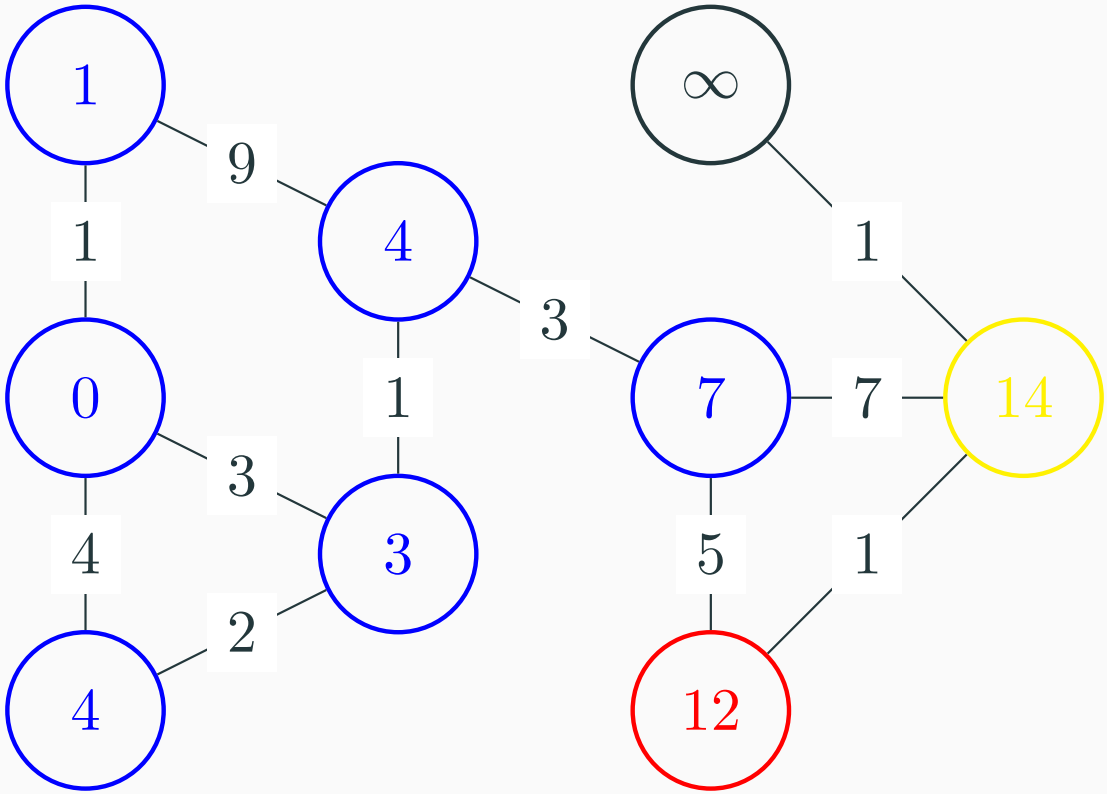
Example



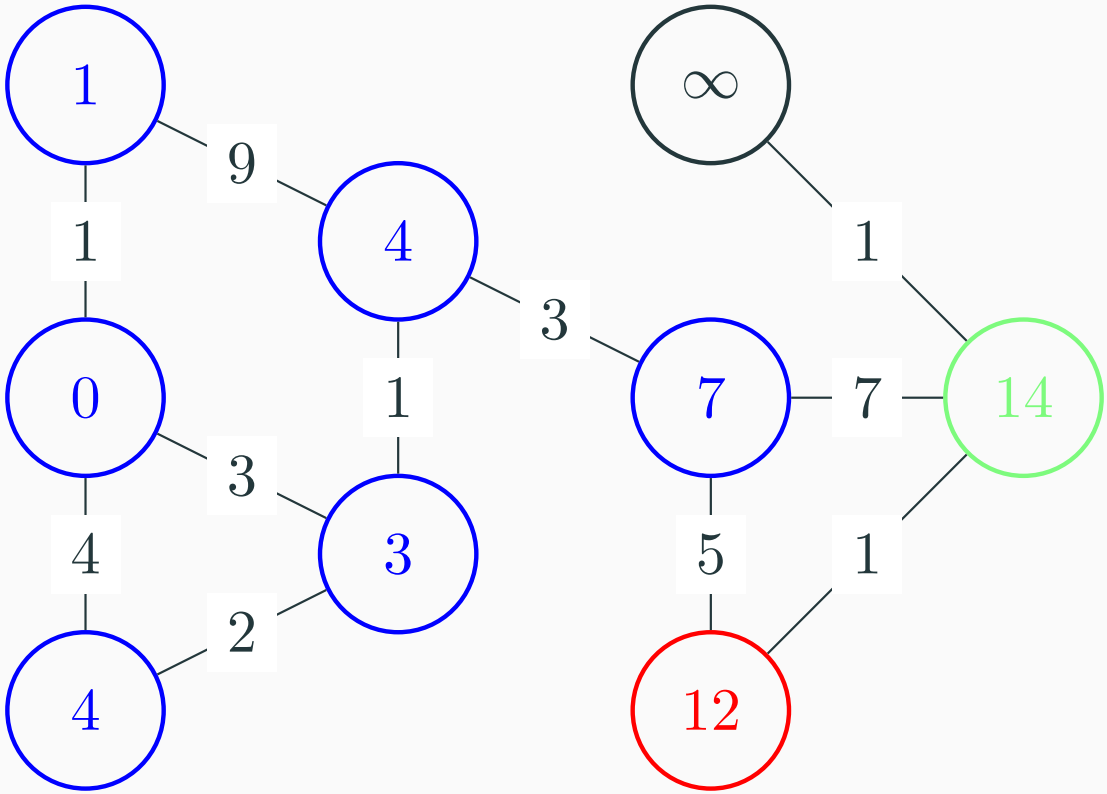
Example



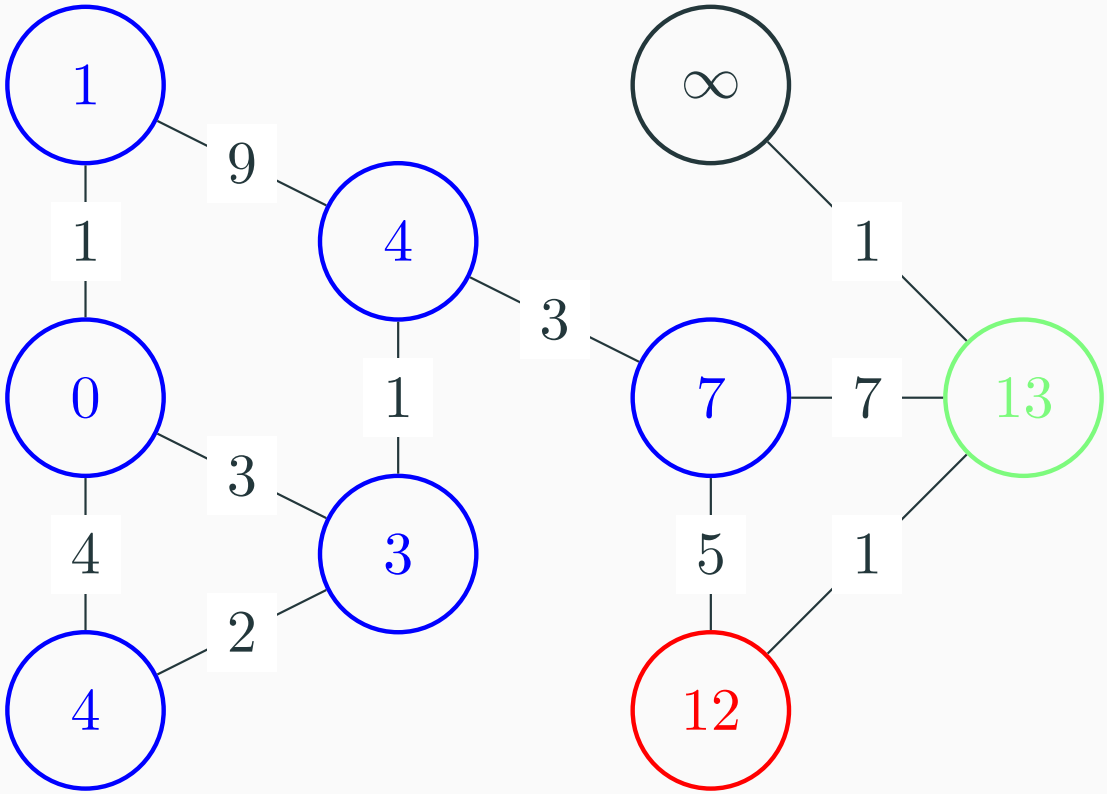
Example



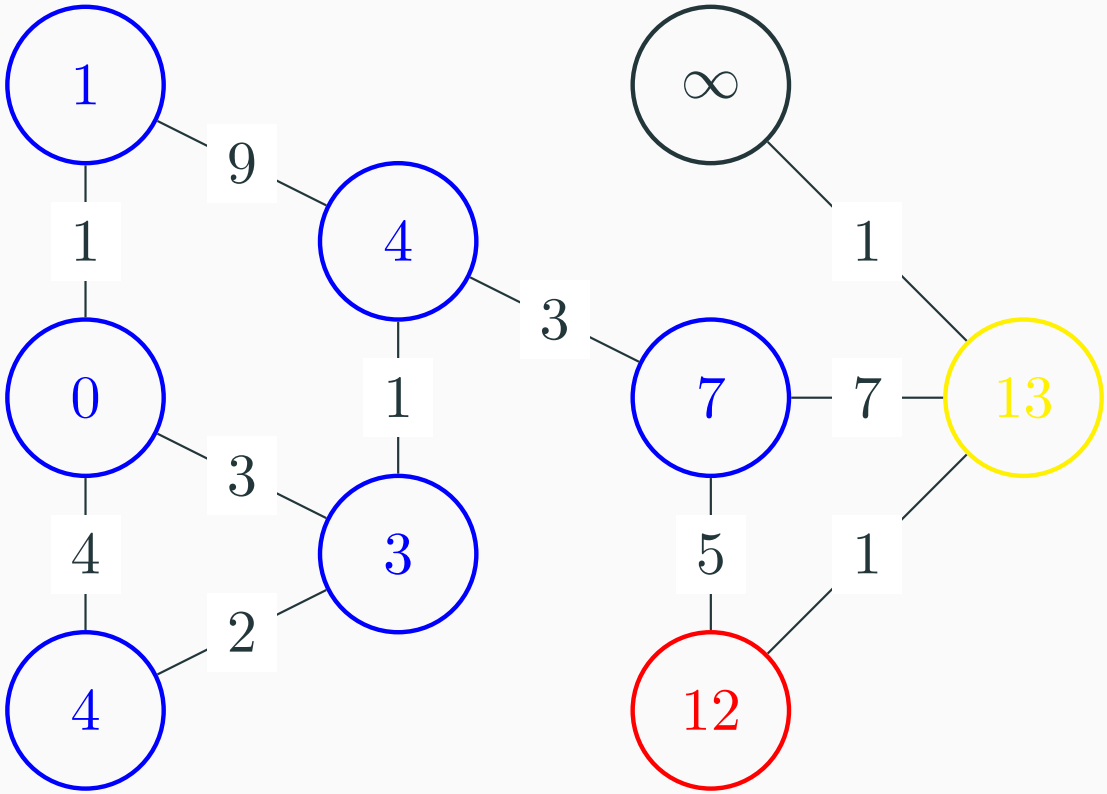
Example



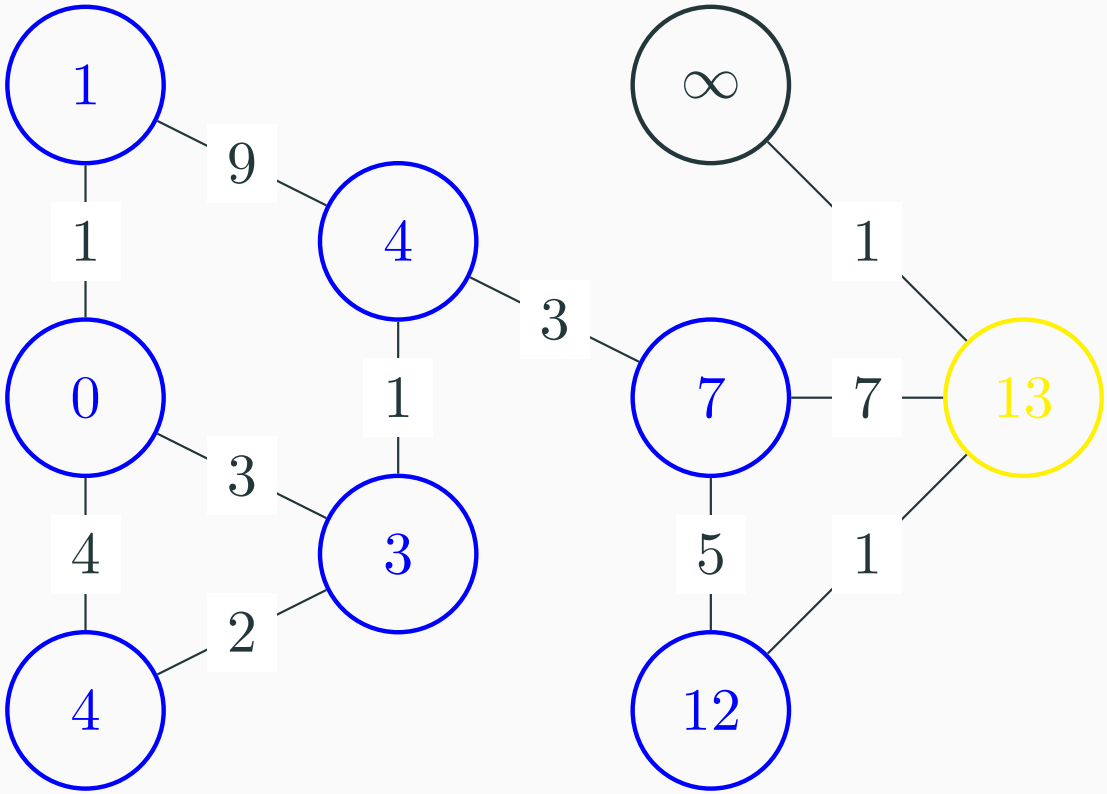
Example



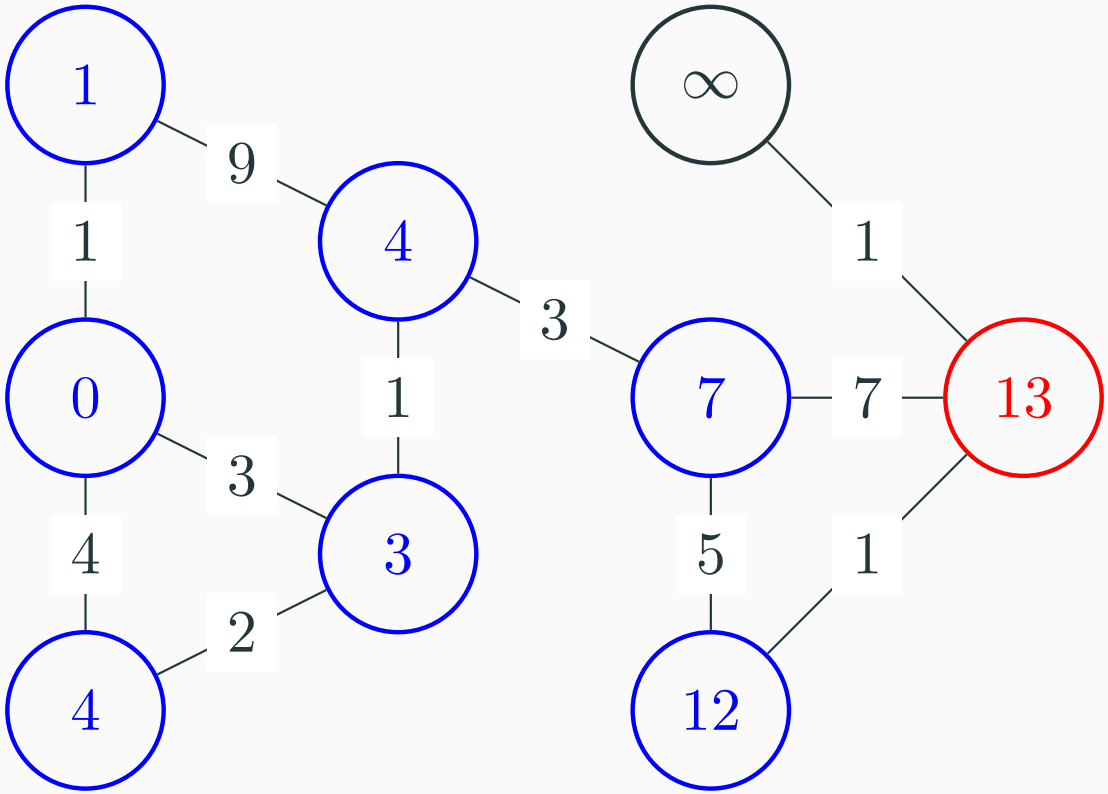
Example



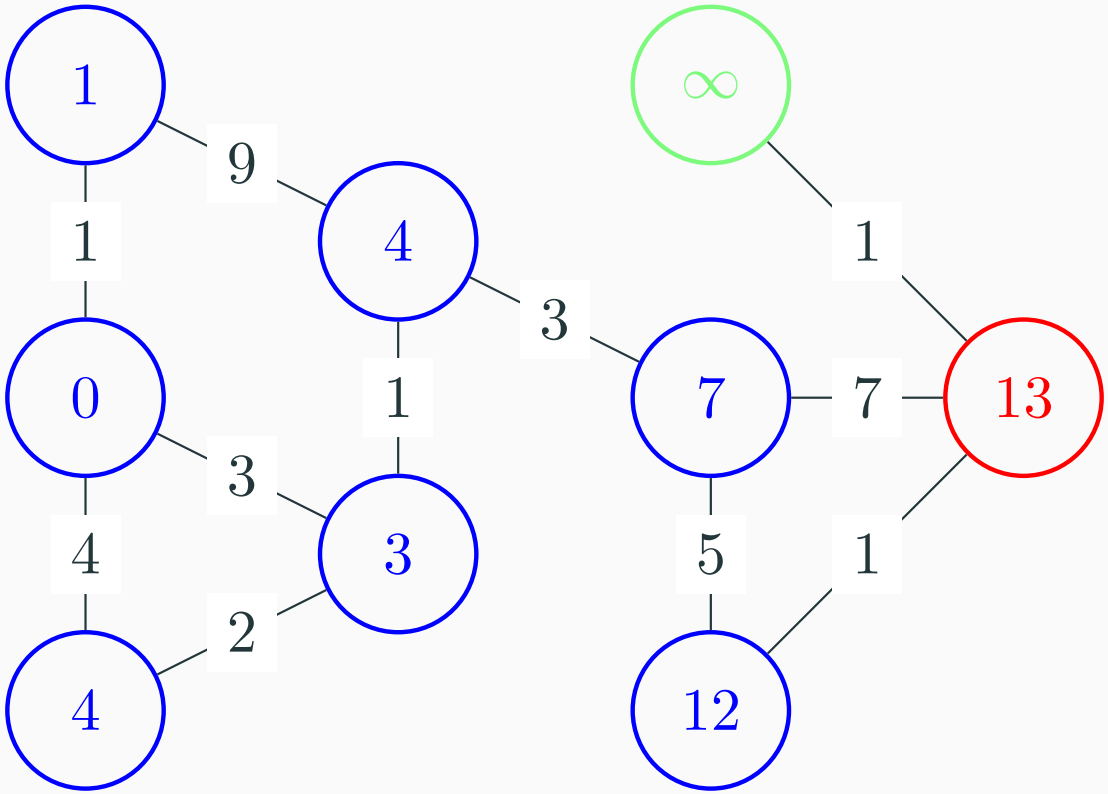
Example



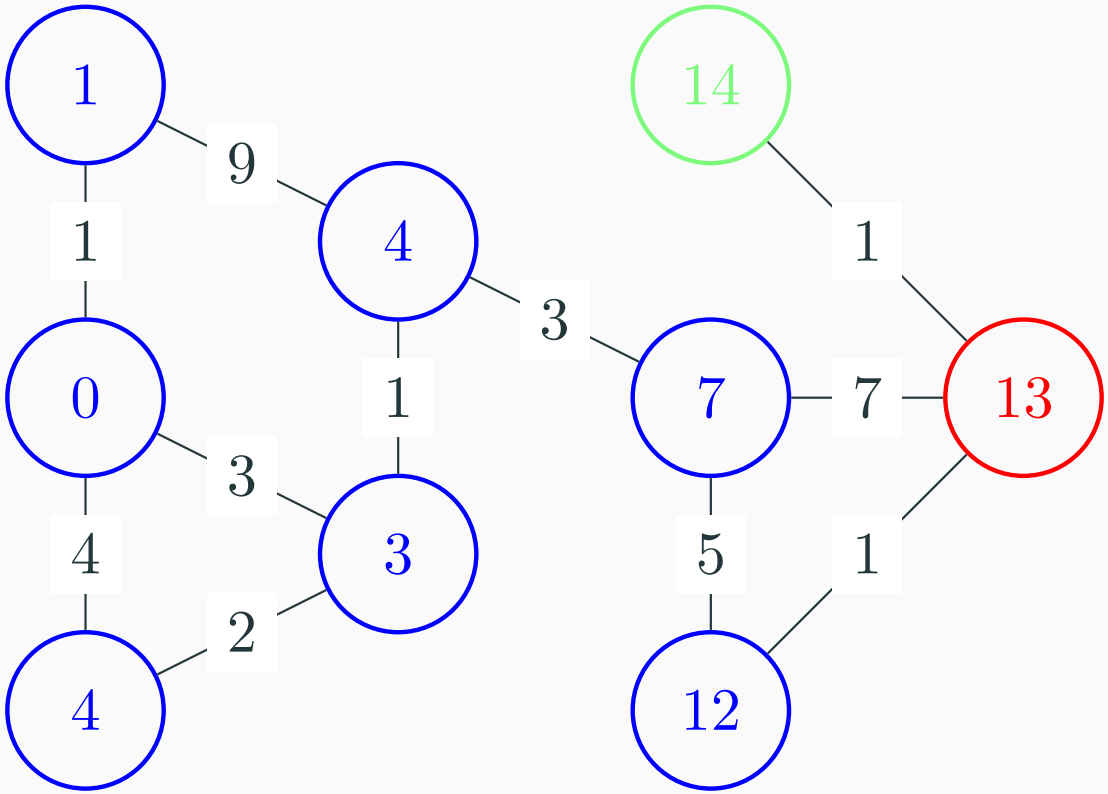
Example



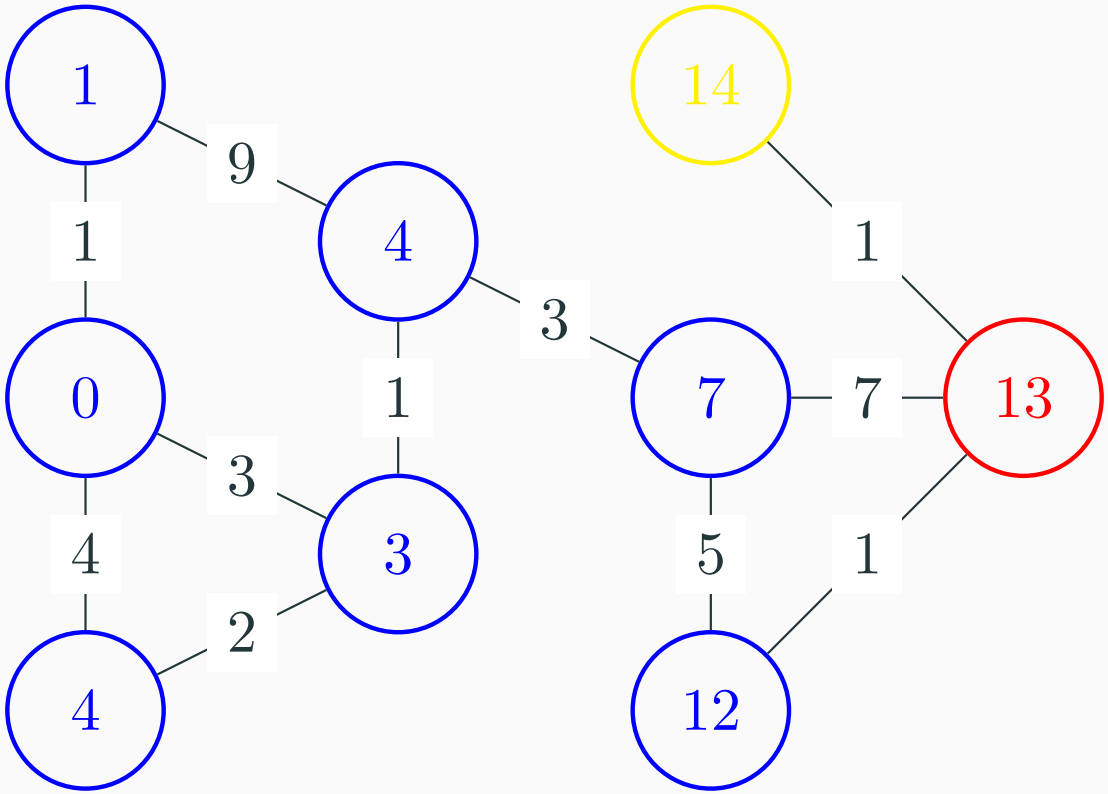
Example



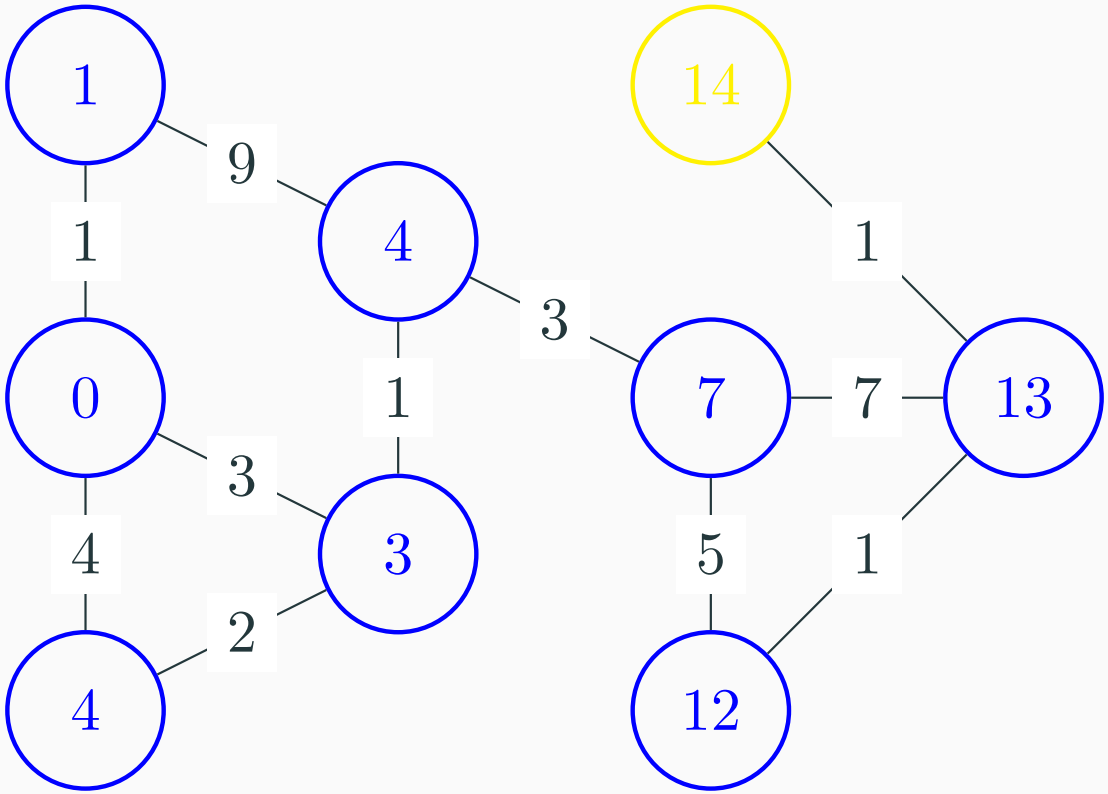
Example



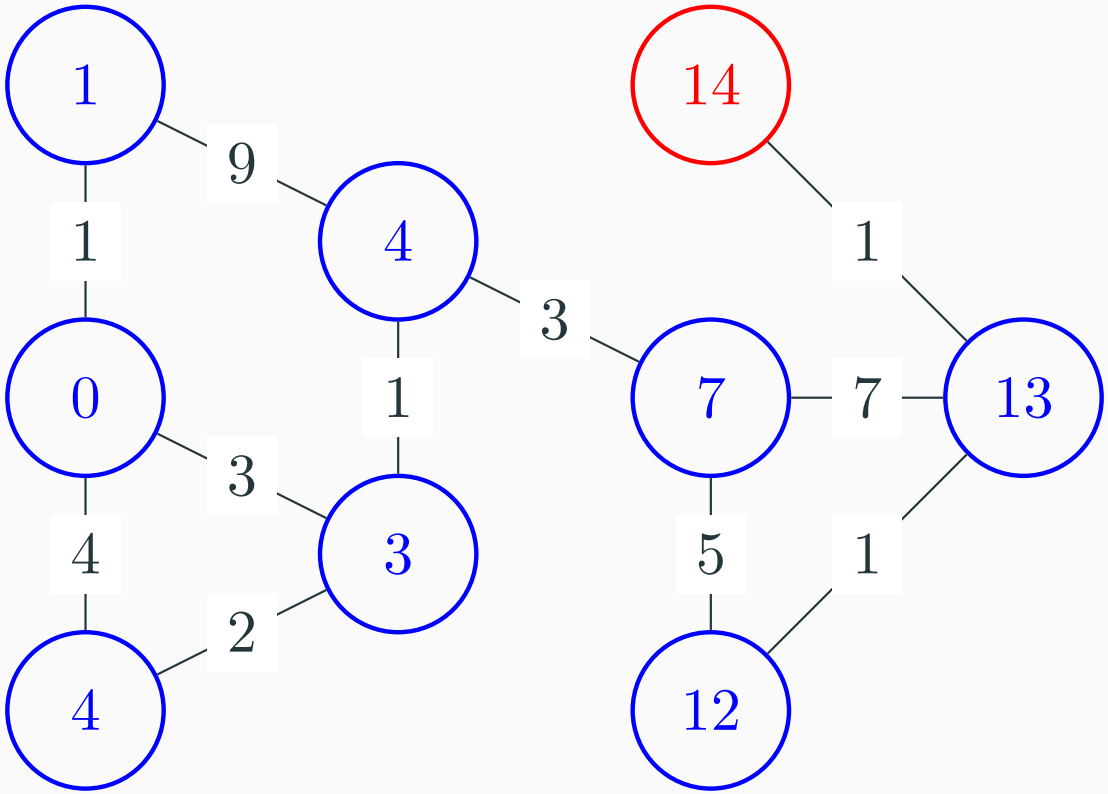
Example



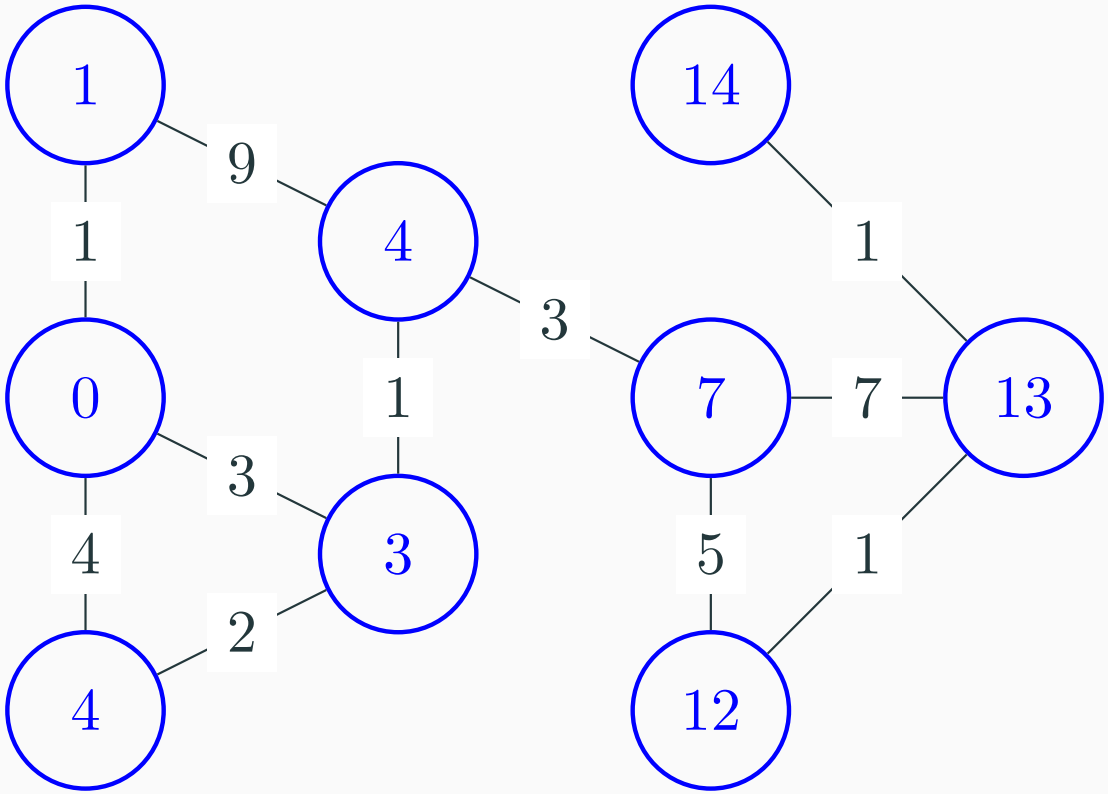
Example



Example



Example



Implementation detail

- There is one thing to watch out for
- The original version of the algorithm uses a heap where you can update the values as you change them
- This is not possible in most standard libraries, so we use a trick instead
- We push a new entry into the heap each time we update the weight to a lower one
- It is **very important** to only process an entry on the heap if the weight is correct to avoid repeated calculation when doing this!

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;
        int w = pq.top().first;
        pq.pop();
        if(dist[u] != w) continue;

        for (int i = 0; i < adj[u].size(); i++) {
            int v = adj[u][i].v;
            int 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

- For each edge in the graph we might have to push to the heap
- We visit each vertex at most once and iterate over its neighbours
- Thus the time complexity is $O((V + E) \log E)$

- Note that this only works for non-negative weights
- The correctness relies on the fact that once we are done with a vertex we never have to think about it again, which is not true when we have negative weights

Dealing with negative weights

Negative weights

- What do we do then when we have negative weights?
- We can use the Bellman-Ford algorithm, at the cost of a worse time complexity

Bellman-Ford

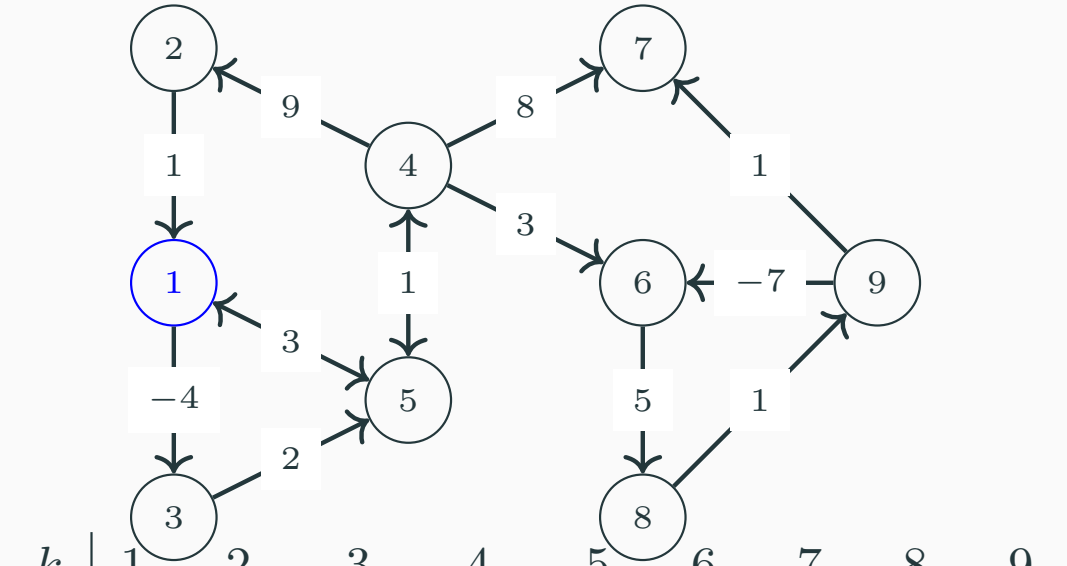
- It is based on dynamic programming, let $f(v, k)$ be the shortest path from the source u to a vertex v that doesn't use more than k vertices to get there
- Then as base cases we get $f(u, 0) = 0$ and $f(v, 0) = \infty$ for $v \neq u$
- Otherwise we can look at $f(w, k-1)$ for some w, and add a single extra edge, giving the recurrence

$$f(v, k) = \min\left(f(v, k-1), \min_{x \in N^-(v)} (w(x, v) + f(x, k-1))\right)$$

Bellman-Ford

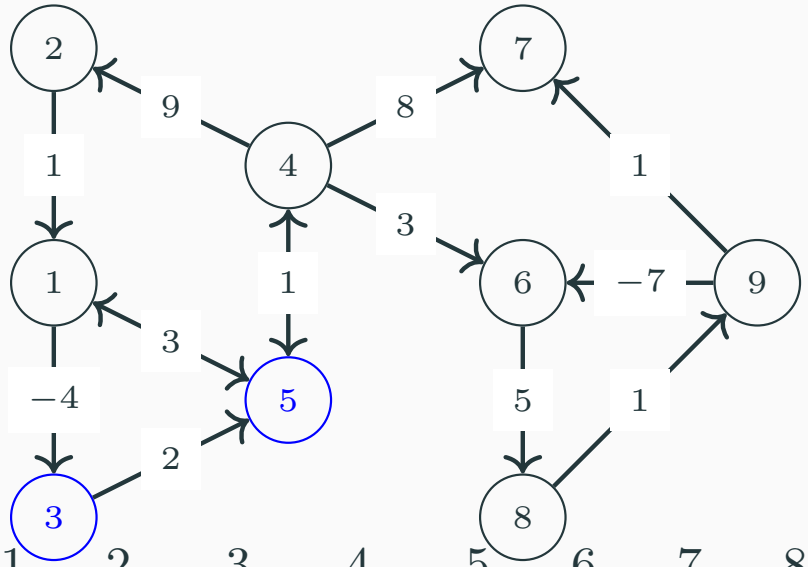
- To implement this efficiently we will make some changes from just a straight-forward DP implementation (but that would work!)
- Each $f(v, k)$ depends only on $f(w, k-1)$ for some different w, so we only have to store our current row k and the last one
- This essentially gets us the usual implementation of Bellman-Ford

Example



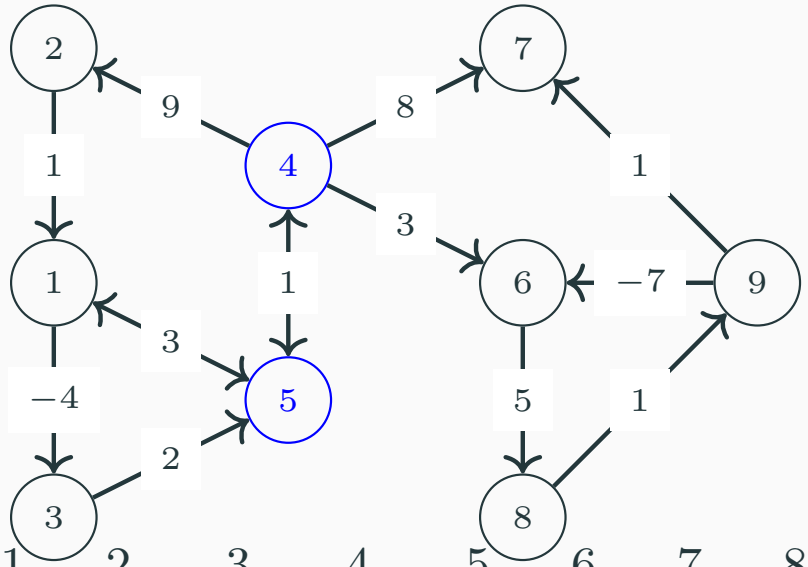
k	1	2	3	4	5	6	7	8	9
0	0	∞	∞	∞	∞	∞	∞	∞	∞

Example



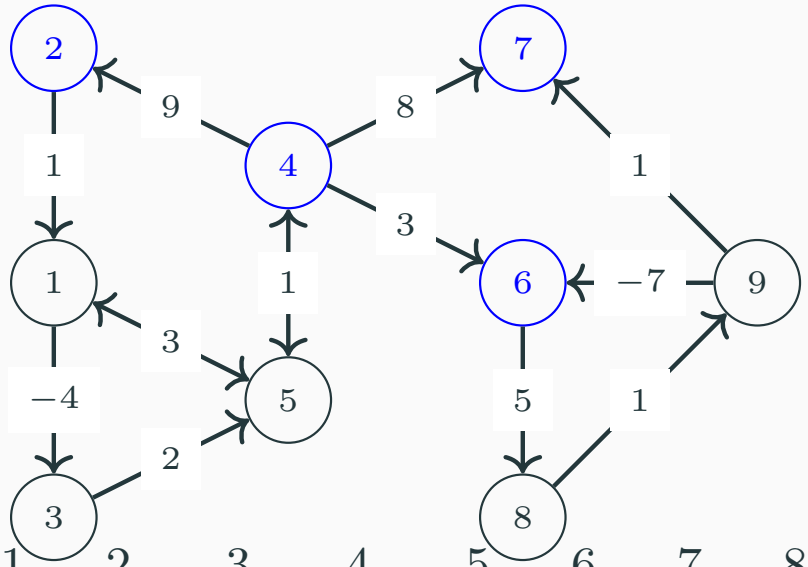
k	1	2	3	4	5	6	7	8	9
0	0	∞	∞	∞	∞	∞	∞	∞	∞
1	0	∞	-4	∞	3	∞	∞	∞	∞

Example



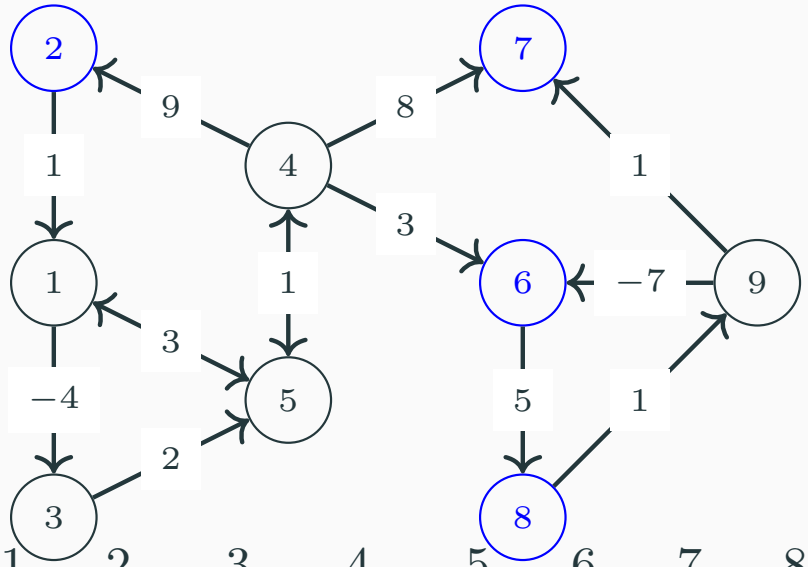
k	1	2	3	4	5	6	7	8	9
0	0	∞	∞	∞	∞	∞	∞	∞	∞
1	0	∞	-4	∞	3	∞	∞	∞	∞
2	0	∞	-4	4	-2	∞	∞	∞	∞

Example



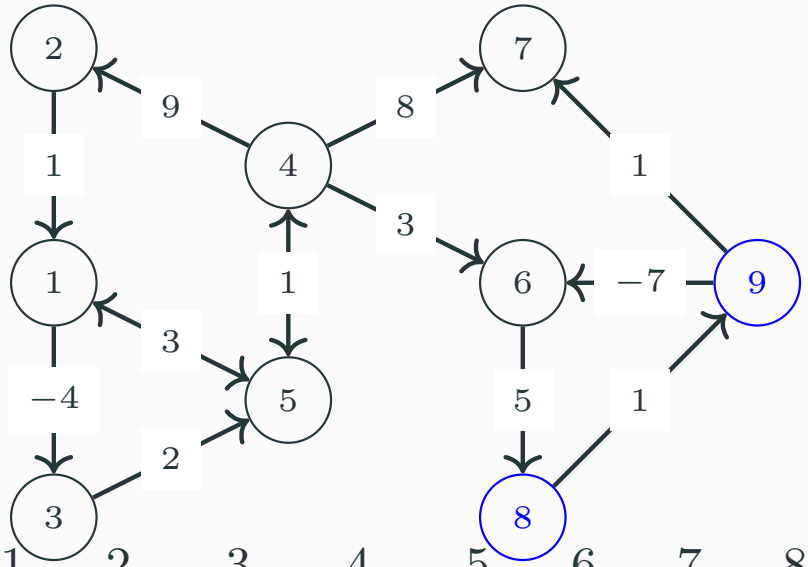
k	1	2	3	4	5	6	7	8	9
0	0	∞	∞	∞	∞	∞	∞	∞	∞
1	0	∞	-4	∞	3	∞	∞	∞	∞
2	0	∞	-4	4	-2	∞	∞	∞	∞
3	0	13	-4	-1	-2	7	12	∞	∞

Example



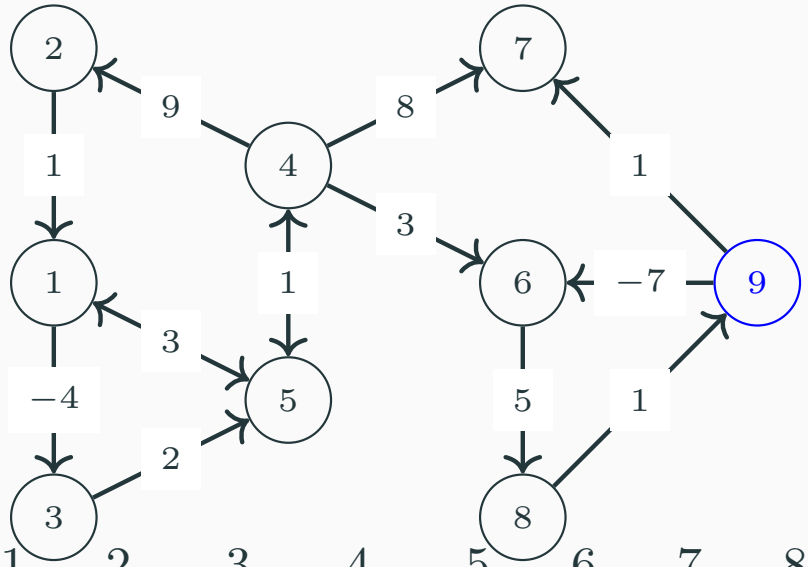
k	1	2	3	4	5	6	7	8	9
0	0	∞	∞	∞	∞	∞	∞	∞	∞
1	0	∞	-4	∞	3	∞	∞	∞	∞
2	0	∞	-4	4	-2	∞	∞	∞	∞
3	0	13	-4	-1	-2	7	12	∞	∞
4	0	8	-4	-1	-2	2	7	12	∞

Example



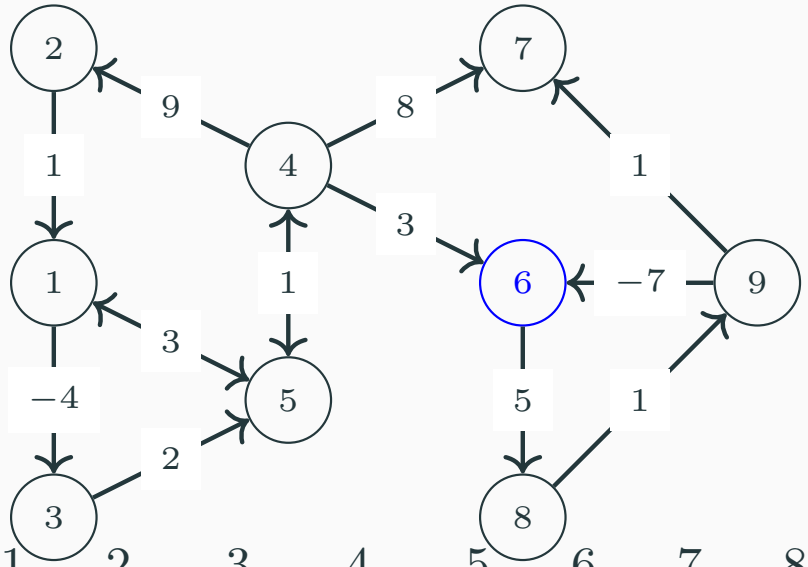
k	1	2	3	4	5	6	7	8	9
0	0	∞	∞	∞	∞	∞	∞	∞	∞
1	0	∞	-4	∞	3	∞	∞	∞	∞
2	0	∞	-4	4	-2	∞	∞	∞	∞
3	0	13	-4	-1	-2	7	12	∞	∞
4	0	8	-4	-1	-2	2	7	12	∞
5	0	8	-4	-1	-2	2	7	7	13

Example



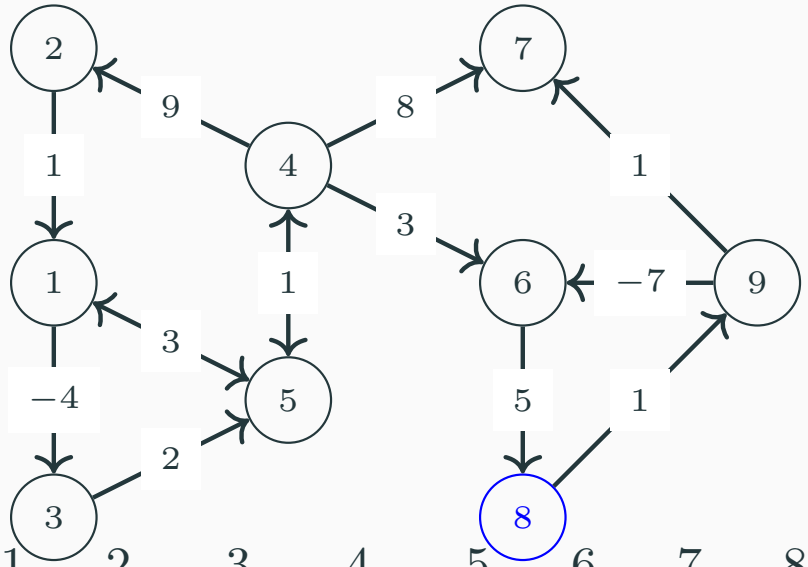
k	1	2	3	4	5	6	7	8	9
0	0	∞	∞	∞	∞	∞	∞	∞	∞
1	0	∞	-4	∞	3	∞	∞	∞	∞
2	0	∞	-4	4	-2	∞	∞	∞	∞
3	0	13	-4	-1	-2	7	12	∞	∞
4	0	8	-4	-1	-2	2	7	12	∞
5	0	8	-4	-1	-2	2	7	7	13
6	0	8	-4	-1	-2	2	7	7	8

Example



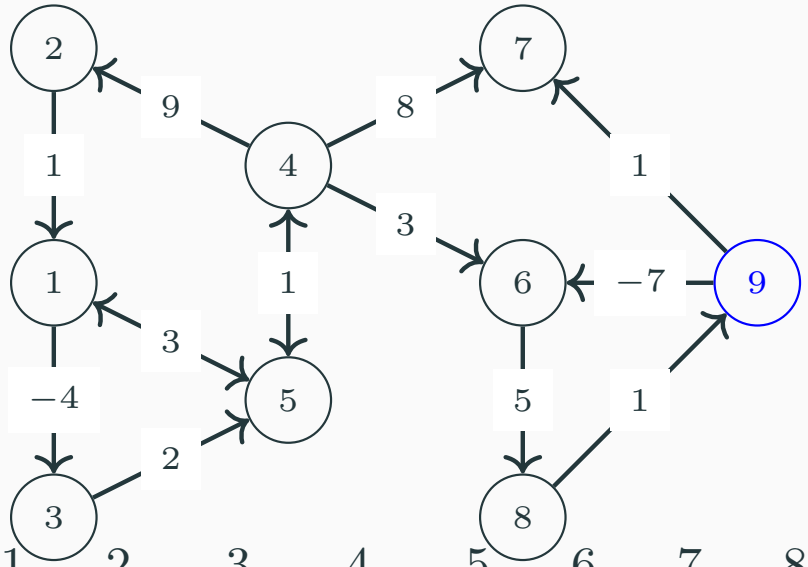
k	1	2	3	4	5	6	7	8	9
0	0	∞	∞	∞	∞	∞	∞	∞	∞
1	0	∞	-4	∞	3	∞	∞	∞	∞
2	0	∞	-4	4	-2	∞	∞	∞	∞
3	0	13	-4	-1	-2	7	12	∞	∞
4	0	8	-4	-1	-2	2	7	12	∞
5	0	8	-4	-1	-2	2	7	7	13
6	0	8	-4	-1	-2	2	7	7	8
7	0	8	-4	-1	-2	1	7	7	8

Example



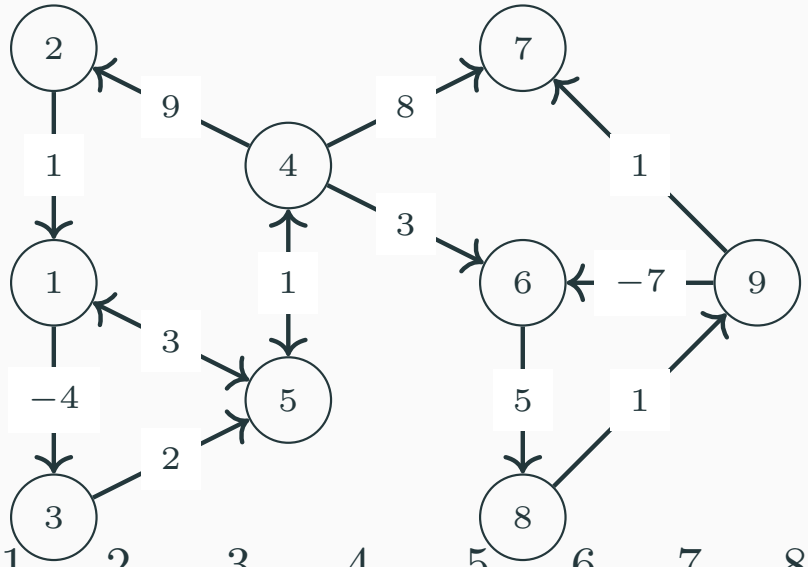
k	1	2	3	4	5	6	7	8	9
0	0	∞	∞	∞	∞	∞	∞	∞	∞
1	0	∞	-4	∞	3	∞	∞	∞	∞
2	0	∞	-4	4	-2	∞	∞	∞	∞
3	0	13	-4	-1	-2	7	12	∞	∞
4	0	8	-4	-1	-2	2	7	12	∞
5	0	8	-4	-1	-2	2	7	7	13
6	0	8	-4	-1	-2	2	7	7	8
7	0	8	-4	-1	-2	1	7	7	8
8	0	8	-4	-1	-2	1	7	6	8

Example



k	1	2	3	4	5	6	7	8	9
0	0	∞	∞	∞	∞	∞	∞	∞	∞
1	0	∞	-4	∞	3	∞	∞	∞	∞
2	0	∞	-4	4	-2	∞	∞	∞	∞
3	0	13	-4	-1	-2	7	12	∞	∞
4	0	8	-4	-1	-2	2	7	12	∞
5	0	8	-4	-1	-2	2	7	7	13
6	0	8	-4	-1	-2	2	7	7	8
7	0	8	-4	-1	-2	1	7	7	8
8	0	8	-4	-1	-2	1	7	6	8
9	0	8	-4	-1	-2	1	7	6	7

Example



k	1	2	3	4	5	6	7	8	9
0	0	∞	∞	∞	∞	∞	∞	∞	∞
1	0	∞	-4	∞	3	∞	∞	∞	∞
2	0	∞	-4	4	-2	∞	∞	∞	∞
3	0	13	-4	-1	-2	7	12	∞	∞
4	0	8	-4	-1	-2	2	7	12	∞
5	0	8	-4	-1	-2	2	7	7	13
6	0	8	-4	-1	-2	2	7	7	8
7	0	8	-4	-1	-2	1	7	7	8
8	0	8	-4	-1	-2	1	7	6	8
9	0	8	-4	-1	-2	1	7	6	7

Bellman-Ford

- How do we read negative cycles from this though?
- In a graph with no negative cycles a shortest path visits each vertex at most once
- Thus we can simply do $V - 1$ more values of k to see if anything gets shorter
- If the distance to v gets shorter then there is a negative cycle on the way to v

Bellman-Ford algorithm

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

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

- This gets us a $\mathcal{O}(VE)$ algorithm, which is quite a bit slower than Dijkstra
- A possible improvement is to store every vertex that received an update for the last k value in a queue, and only checking things they could affect at the next k
- This is still worst case $\mathcal{O}(VE)$, but in the average case much much faster
- This is generally known as the SPFA algorithm (Shortest Path Faster Algorithm)

All pairs shortest path

Bellman-Ford

- What if we want to get the distance from every vertex to every vertex?
- If there are no negative cycles running Dijkstra's from every vertex works pretty well
- But for negative cycles running Bellman-Ford V times is pretty slow
- Luckily we can get a bulk discount in this case!
- This is known as the Floyd-Warshall algorithm

Floyd-Warshall algorithm

- We use dynamic programming again.
- 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: $\mathrm{sp}(-1, i, j) = \mathrm{weight}[i][j]$ 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
- Though we can do better! By combining everything we did so far and mixing in an algorithm known as Johnson's algorithm, Floyd-Warshall can be beat. But this won't be in the homework (maybe the bonus problems!)