Ivan Franko National University of Lviv

University of L'Aquila

Double-Degree Master's Programme - InterMaths
Applied and Interdisciplinary Mathematics

Master of Science **Applied Mathematics**	**Master of Science** **Mathematical Engineering**
Ivan Franko National University of Lviv (IFNUL)	**University of L'Aquila (UAQ)**

Master's Thesis

Generalized maximum flow problem and its application

Supervisor
Assoc Prof. Vasyl Biletskyj

Candidate
Oleh-Andrii Zhuk

Student ID (UAQ): 274485
Student ID (IFNUL): 2720354

Academic Year 2021/2022

Contents

Introduction 3

Chapter 1. Flow network 4

1.1. Flow network definition. 4

1.2. Maximum flow problem 5

1.3. Generalized maximum flow problem. 6

Chapter 2. Maximum flow problem algorithms 8

2.1. Augmenting path algorithm 8

2.2. Algorithm of canceling flow generating cycles. 9

2.3. Greedy approach to maximum flow problem. 11

Chapter 3. Software implementation 12

Conclusions 12

Appendix A. Bellman-Ford algorithm. 14

Appendix B. Algorithm of canceling all cycles with negative weights. 16

Appendix C. Algorithm of flow propagation through the cycle. 17

Appendix D. Computation of potentials of vertices algorithm. 18

Appendix E. Canceling flow generating cycles algorithm. 19

Appendix F. Maximum flow computation algorithm. 19

Introduction

Logistics and planning are the basis of conducting any business that has gone beyond local. So, for example, a skillful solution to logistical issues allows one to optimize the work of the delivery service, build an efficient infrastructure and minimize transportation costs.

Often, one can interpret such and similar problems as finding the maximum flow in the network, finding the minimum cut, or finding the maximum flow of minimum cost. These and similar problems consider that the flow maintains its power during transportation. However, practice shows that this is not always the case — old pipes often leak, liquids evaporate, and power lines lose some of their energy during long-distance transportation. Thus, there is a need to generalize the maximum flow problem for networks where the edges lose a share of the flow.

Therefore, this work aims to study algorithms for solving the generalized maximum flow problem and applying such algorithms.

Based on its goal, the paper has the following tasks:

1. Formulate a generalization of the maximum flow problem;
2. Investigate existing algorithms for solving the problem and evaluate their effectiveness;
3. Implement the algorithm for solving the problem;
4. Create an application to demonstrate the operation of the algorithm on a given graph;

The first section, “Flow network”, describes the problem of finding the maximum flow and its generalization and formulates the condition of validity and optimality of the found solution.

The second section, “Maximum flow problem algorithms”, reviews existing algorithms for solving the problem and compares their performance.

The third section, “Software implementation”, summarizes the author's implementation of the algorithm and the web application for its visualization.

Chapter 1.

Flow network

1.1. Flow network definition.

A *flow network* is a graph in which a carrying capacity is defined for each edge - a limit on the amount of flow it can pass along the edge. We will assume that the flow starts from a particular source vertex and goes to the sink vertex. In the problems of finding the flow in the network, it is allowed to control how much flow can go through a distinct edge. This amount should not exceed the given capacity of the edge.

All the problems that we will consider in this paper concern the directed graph (V, E), where V is the set of vertices, and E is the set of directed edges of the graph. Note that the number of vertices in the graph is n, and the number of edges is m.

Among all the vertices, we distinguish two vertices: the source vertex s and the sink t. A *source vertex* is a vertex that generates an infinite amount of flow that goes to a *sink vertex*. The maximum flow problem aims to find the maximum possible amount of flow reaching t.

We will assume that the capacity of the arc is a constant value that limits the maximum amount of flow that can pass through it. Therefore, we denote by $u: E \to \mathbb{R}$ the function that characterizes the capacity of edges. For example, let the pair (v, w) be the edge from the vertex v to w, then the edge's capacity is $u(v, w)$.

Thus, a flow network is a directed graph G with predefined capacities for each edge. We can say $G = (V, E, u)$.

We denote the *flow function* $f: E \to \mathbb{R}$ as the one that characterizes the flow of the graph and satisfies all the constraints of *capacity* (1.1), *antisymmetry* (1.2), and *flow preservation* (1.3):

$$\forall (v, w) \in E: f(v, w) \leq u(v, w) \tag{1.1}$$

$$\forall (v, w) \in E: f(v, w) = -f(w, v) \tag{1.2}$$

$$\forall v \in V \backslash \{s, t\}: \sum_{w \in V} f(v, w) = 0 \tag{1.3}$$

Thus, for the flow function *f*, it is possible to determine *residual capacity function*: $u_f: E \to \mathbb{R}$ such that satisfies equality (1.4):

$$u_f(v,w) = u(v,w) - f(v,w) \tag{1.4}$$

The residual capacity of the network generates the so-called *residual network* $G_f = (V, E, u_f)$.

For example $u(v,w) = 7, u(w,v) = 0, f(v,w) = -f(w,v) = 4$, then in the residual network G_f the capacity of the edge $u_f(v,w) = 7 - 4 = 3$, while $u_f(w,v) = 0 - (-4) = 4$.

For a residual network, an *augmenting path* is a path from vertex s to vertex t that passes only along edges with a positive capacity value: $(v,w) \in E$, such that:

$$u_f(v,w) > 0$$

1.2. Maximum flow problem

The maximum flow problem aims to deliver the maximum possible amount of flow from the source node to the sink of the flow network $G = (V, E, u)$. Since the network flow function f must satisfy the flow conservation condition (1.3) for all vertices except s and t, the *value of the flow function* can be defined as follows:

$$|f| = \sum_{v \in V} \sum_{(v,t) \in E} f(v,t) \tag{1.5}$$

Therefore, the goal of the problem is to find the *flow function* which maximizes its value $|f|$.

It is clear that if there is an augmenting path in the network G_f, then we can increase the value of the flow function by propagating flow along this path. However, the converse statement also holds. It was shown in the works of Ford and Fulkerson [6]. This fact allows us to formulate the optimality condition of the flow f: the flow f is maximal if and only if there is no increasing path in the residual network G_f.

1.3. Generalized maximum flow problem.

The maximum flow problem can be interpreted for the supply network. For example, it can be a network of oil pipelines on the scale of the country. However, this problem formulation does not consider the shortcomings of such a network - old pipes can leak and lose some amount of the flow during transportation. Therefore, we will describe the generalization of the maximum flow problem for such imperfect flow networks.

Consider the problem of finding the maximum flow. On top of it, we define a positive coefficient for each edge that adjusts the amount of flow passing through this edge. Let us define by $\gamma: E \to \mathbb{R}$, a function that describes a flow amplification factor for each edge. We call it a *gain function*.

For example: let 100 units of the flow arrive at the vertex v; $\gamma(v,w) = \frac{1}{4}$, then after its transportation through the edge (v,w), $100 \cdot \gamma(v,w) = 25$ units of flow will reach the vertex w.

Without limiting the generality, we will assume that the function γ is symmetric, such that $\gamma(v,w) = \frac{1}{\gamma(w,v)}$.

For the generalized flow network $G = (V, E, u, \gamma)$ we define the generalized flow function $g: E \to \mathbb{R}$ as such that satisfies the capacity restriction (1.1), flow conservation (1.3), and generalized antisymmetry (1.6):

$$\forall (v,w) \in E: g(v,w) = -\gamma(w,v) g(w,v) \qquad (1.6)$$

Similarly, it is possible to calculate the value of the generalized flow $|g|$:

$$|g| = \sum_{v \in V\ (v,t) \in E} \gamma(v,t) g(v,t) \qquad (1.7)$$

With regards to the generalized flow the residual capacity of the network is determined as follows:

$$u_g(v,w) = u(v,w) - g(v,w) \qquad (1.8)$$

Then $G_g = \left(V, E, u_g, \gamma\right)$ is the residual network.

From now on, when we talk about the problem of finding the maximum flow and its derived definitions, we will refer to its generalized version.

For the flow network $G = (V, E, u, \gamma)$, we define a *labeling function* $\mu: V \to \mathbb{R}^+ \cup \{\infty\}$ such that $\mu(t) = 1$. The idea of the relabeled network was first introduced by F. Glover and D. Klingman in their works [4].

According to the following labeling function, the *relabeled capacities* (1.9) and the relabeled gains (1.10) are determined as follows:

$$u_\mu(v, w) = u(v, w)/\mu(v) \quad (1.9)$$

$$\gamma_\mu(v, w) = \gamma_\mu(v, w)\mu(v)/\mu(w) \quad (1.10)$$

According to such transformations, we will form a *relabeled network* $G_\mu = (V, E, u_\mu, \gamma_\mu)$. Then the following statement holds:

For any labeling function μ, g is a generalized flow in the network G if and only if $g_\mu(v, w) = g(v, w)/\mu(v)$ is a generalized flow in the network G_μ and also $|g| = |g_\mu|$.

In other words, the relabeling of the network does not affect the solution of the maximum flow problem.

At the same time, we call the labeling function μ *canonical* if, for each vertex v, $\mu(v)$ is equal to the inverse of the maximum value of the generalized flow reaching the sink vertex t, provided that one unit of flow leaves the vertex v.

For example, let $\gamma(v, w) = \frac{1}{4}$, $\gamma(w, q) = \frac{6}{5}$, and $\gamma(q, t) = 3$, as well as the value of the flow capacity of these edges is greater than 1. Let us propagate one unit of flow from vertex v. Then, exactly $\gamma(v, w)\gamma(w, q)\gamma(q, t) = \frac{1}{4} \cdot \frac{6}{5} \cdot 3 = \frac{18}{20}$ units of flow will reach the sink. Thus, the canonical labeling $\mu(v) = \frac{20}{18}$.

One can compute the canonical labeling using the algorithm for finding the minimum path in a graph with edges of length $l(v, w) = -\log \gamma(v, w)$.

Chapter 2.

Maximum flow problem algorithms

2.1. Augmenting path algorithm

In this subsection, we consider Trumper's algorithm. It is a kind of generalization of the Ford-Fulkerson algorithm for solving the maximum flow problem. The main idea of Trumper's algorithm is to find the most profitable path in the network, along which a unit of flow reaches the sink with the maximum possible value, and to push the maximum possible flow along such a path.

Using the canonical coloring of the network, we describe Trumper's algorithm:

1. Let the optimal flow be $g \equiv 0$.
2. Let the initial labeling of the network be $\mu \equiv 1$
3. As long as there is an increasing path in G_g, repeat:
    a. Let μ be the canonical labeling of the graph G_g.
    b. Let g_μ be the maximum flow in the graph $G_{g,\mu}$ passing only along edges (v, w) such that $\gamma_\mu(v, w) = 1$.
    c. Update the optimal flow:
    $\forall (w, v) \in E: g(w, v) \coloneqq g(w, v) + \mu(v) g_\mu(w, v)$.
4. End of the algorithm.

The computational complexity of finding the canonical labeling of the graph is equal to the complexity of finding the shortest path in the graph. At any cycle iteration, one can find the canonical labeling using the coloring found in the previous step. Thus, it is possible to guarantee that the value of the function $\gamma \leq 1$ in the graph G_g for all edges. This fact allows using the Dijkstra algorithm with Johnson potentials [1].

Since the complexity of finding the shortest path is asymptotically faster than the complexity of finding the maximum flow in the network, we can assume that the complexity of loop iterations (a)-(c) directly depends on the complexity of finding the maximum flow in the graph.

It is possible to show that the number of iterations of the cycle (a)-(c) is finite, but it can be of exponential order.

2.2. Algorithm of canceling flow generating cycles.

We consider the algorithm proposed by A. Goldberg, S. Plotkin and E. Tardos [2].

Let us define the function $\pi: V \to \mathbb{R}$, which describes the value of the *potential* for each vertex. Let the function $c: E \to \mathbb{R}$ describe the weight of the edges in the initial graph: $c(v,w) = -\log_b(\gamma(v,w))$, for some constant b. Then the *potential weight* of the edge $c_\pi(v,w)$ can be determined as follows:

$$c_\pi(v,w) = c(v,w) - \pi(v) + \pi(w) \qquad (2.1)$$

The idea of the cycle processing algorithm is to find potential weights such that the following condition (2.2) is fulfilled for some variable ϵ:

$$c_\pi(v,w) \geq -\epsilon \qquad (2.2)$$

The algorithm suggests removing all cycles such that the potential weights of all its edges are negative. After that, we can update the flow function and decrease the value of ϵ. Such an iterative method allows one to find and process all cycles that generate a flow in the initial graph in polynomial time.

We formulate the algorithm for the initial graph G as follows:

1. Let $g \equiv 0, \pi \equiv 0, \mu \equiv 0, \epsilon \coloneqq \underset{(v,w)\in E}{-min}\, c(v,w)$.
2. While G_g has flow generating cycle, repeat:
   a. Calculate the values of potentials π that satisfy (2.2)
   b. Delete all cycles on edges with negative weights c_π and update the flow g afterward.
   c. $\epsilon \coloneqq \epsilon \cdot \frac{n-1}{n}$
3. Optionally we can define the labeling $\mu(v) \coloneqq b^{\pi(v)-\pi(t)}$
4. End of the algorithm.

It is important to denote that if $\epsilon = 0$ and the potentials of the vertices satisfy (2.2), we can guarantee that there is a labeling function μ such that $\gamma_\mu \leq 1$, which means that there is no flow generating cycles in the relabeled graph.

Let us estimate the complexity of such an algorithm.

Deleting all cycles (step 2.b.) can be done using the depth first search algorithm in $O(m^2)$. However, if, in the process, we mark the vertices that are no longer part of the cycles, the complexity will be $O(nm)$. We can improve it up to $O(mlog(n))$ using Sleator and Tarjan dynamic trees [3]. One can find the $O(nm)$ implementation of this step in the appendices.

Step 2.a. can be computed in $O(m)$ using the topological sorting algorithm for given values of potentials computed in the previous step.

Checking the existence of flow generating cycles (step 2) can be done using the Bellman-Ford algorithm. Its complexity is $O(nm)$. However, it can be done with a lower frequency - every n iterations of the loop, so it does not dominate the overall complexity of the algorithm. See the implementation of step 2 and step 2.a. in the appendices.

Thus, the complexity of one iteration of the cycle entirely depends on the complexity of finding negative cycles in the graph.

The algorithm claims that the number of iterations of the loop 2 depends on the values of the weights of the edges in graph G. Assuming we can represent them as a fraction of integers in the interval from 1 to C, then the number of iterations of the loop 2 is $O\left(n^2 log(C)\right)$. Hence, the total complexity of the algorithm is $O(mn^3 \log(C))$. With the use of Sleator and Tarjan dynamic trees, one can find the solution in $O(mn^2 log(n) \log(C))$. We provide the implementation of the algorithm without using dynamic trees in the appendices.

2.3. Greedy approach to maximum flow problem.

Consider the greedy version of the algorithm described in 2.1. At each step, we find the path with maximum flow value and propagate the flow along that path.

Note that we can use a similar approach to solve the maximum flow of the minimum value problem.

Unlike the previous algorithm, we cannot guarantee at each step that there will not be flow-generating cycles. Therefore, we should find and cancel such cycles at each iteration of the algorithm.

Let us describe this algorithm:

1. Let the optimal flow be $g \equiv 0$
2. While G_g has augmenting path, repeat:
    a. $g' \coloneqq CancelCycles(G_g)$
    b. $g \coloneqq g + g'$
    c. $P \coloneqq$ The path with the maximum flow in G_g
    d. Update the optimal flow g by propagating through P
3. End of the algorithm.

We can find the path with the maximum flow using a similar principle as in the previous algorithm - Dijkstra's algorithm with Johnson potentials[1] or the Bellman-Ford algorithm. We used the Bellman-Ford algorithm in our implementation in the appendices.

According to 2.2. the function that processes and cancels cycles in the graph works with complexity $O(mn^2 log(n)\log(C))$.

Loop 2 is claimed to do $O\big(mlog(|f|)\big)$ iterations. Thus, the total complexity of such an algorithm is $O\big(m^2n^2log(n)log(\mathrm{C}n)\big)$.

The appendices provide an example of this algorithm's implementation without using Sleator and Tarjan dynamic trees with its total complexity of $O\big(m^2n^3log(\mathrm{C}n)\big)$.

Chapter 3.

Software implementation

The software implementation consists of two main parts:

1. Implementation of the maximum flow algorithm described in 2.3.
2. Implementation of the web application for visualization of the algorithm's work.

One can find the main components of the author's implementation of the algorithm in the Rust programming language in the appendix section. The implemented algorithms use a standard set of libraries and data structures.

The web application uses the following technologies and libraries: typescript, React, mui, webpack, react-force-graph (graph visualization), and wasm-tool (integration of the compiled Rust library).

The source code of the software implementation can be viewed and downloaded from the online repository below the link: https://github.com/andrii-zhuk/generalizedFP.

The web application for visualization of the algorithm's work is accessible by the link https://graph.algotester.com.

Conclusions

This work described the theoretical principles of the generalization of the maximum flow problem and estimated the complexity of the algorithms that allow finding the optimal solution to the problem.

We considered an algorithm with potentially exponential execution time, as well as an algorithm with polynomial execution time, which finds an optimal solution with complexity $O(m^2n^2log(n)log(C)log(Cn))$ – section 2.3. We propose the author's software implementation of this algorithm in the Rust programming language in the paper. In addition, we created a web application to demonstrate how the algorithm works.

Appendix A. Bellman-Ford algorithm.

```
pub fn bellman_ford(
    graph: &DirectedGraph,
    edge_value: fn(edge: &Edge) -> Option<f64>,
    mode: Mode,
    start_id: usize,
    reverse: bool,
) -> (Vec<Option<f64>>, Vec<Option<usize>>, Option<Vec<usize>>) {
    let compare = match &mode {
        Mode::Min => |x: f64, y: f64| -> bool { x - y < -EPSILON },
        Mode::Max => |x: f64, y: f64| -> bool { x - y > EPSILON },
    };
    let current_node = if reverse == true {
        |edge: &Edge| edge.to_id
    } else {
        |edge: &Edge| edge.from_id
    };
    let next_node = if reverse == true {
        |edge: &Edge| edge.from_id
    } else {
        |edge: &Edge| edge.to_id
    };
    let mut parents: Vec<Option<usize>> = vec![None; graph.n()];
    let mut dist: Vec<Option<f64>> = vec![None; graph.n()];
    dist[start_id] = Some(0.0);
    for i in 0..graph.n() {
        for (edge_id, edge) in graph.edges_list.iter().enumerate() {
            if dist[current_node(edge)] == None
                || !graph.reachable_from_source(current_node(edge))
                || !graph.reachable_from_source(next_node(edge)) {
                continue;
            }
            let value = edge_value(edge);
            let value = match &value {
                None => continue,
                Some(value) => value,
            };
            if dist[next_node(edge)] == None
                || compare(
                    dist[current_node(edge)].unwrap() + value,
                    dist[next_node(edge)].unwrap(),
                ) {
```

```
                if i == graph.n() - 1 {
                    let mut cycle: Vec<usize> = vec![];
                    let mut node_id = current_node(edge);
                    cycle.push(node_id);
                    while (cycle.len() == 1 || node_id != cycle[0]) && cycle.len() <= 2 *
graph.n() {
                        let edge_id = parents[node_id]
                            .expect("Negative cycle retrieving error: Undefined parent.");
                        cycle.push(edge_id);
                        node_id = current_node(&graph.edges_list[edge_id]);
                        cycle.push(node_id);
                    }


                    cycle.reverse();
                    while cycle.len() > 1 && cycle.first().unwrap() != cycle.last().unwrap() {
                        cycle.pop();
                        cycle.pop();
                    }
                    if reverse {
                        cycle.reverse();
                    }
                    return (dist, parents, Some(cycle));
                }
                dist[next_node(edge)] = Some(dist[current_node(edge)].unwrap() + value);
                parents[next_node(edge)] = Some(edge_id);
            }
        }
    }


    return (dist, parents, None);
}
```

Appendix B. Algorithm of canceling all cycles with negative weights.

```
fn remove_negative_cycles(graph: &mut DirectedGraph, potentials: &Vec<f64>) -> Option<f64> {
    let mut status = vec![0; graph.n()];
    let mut cycle: Vec<usize> = Vec::<usize>::new(); cycle.reserve(graph.n());
    fn dfs(node_id: usize, graph: &DirectedGraph, potentials: &Vec<f64>, status: &mut
Vec<i32>, cycle: &mut Vec<usize>) {
        status[node_id] = 1;
        for &edge_id in &graph.adj_lists[node_id] {
            let edge = &graph.edges_list[edge_id];
            if edge_value(edge, potentials) == None || status[edge.to_id] == 2 {
                continue;
            }
            if status[edge.to_id] == 0 {
                dfs(edge.to_id, graph, potentials, status, cycle);
            }
            if status[edge.to_id] == 1 || cycle.len() > 0 {
                if cycle.len() == 0 || cycle.len() % 2 == 0 {
                    cycle.push(edge.to_id); cycle.push(edge_id);
                    if cycle[0] == edge.from_id {
                        cycle.push(edge.from_id); cycle.reverse();
                    }
                }
                status[node_id] = 0;
                return;
            }
        }
        status[node_id] = 2;
    }
    let mut cumulative_excess = 0.0;
    for node_id in 0..(graph.n()) {
        if !graph.nodes[node_id].reachable_from_source || status[node_id] == 2{
            continue;
        }
        dfs(node_id, &graph, potentials, &mut status, &mut cycle);
        if cycle.len() == 0 { continue; }
        let cycle_edges = get_path_edge_ids(Some(cycle.clone()));
        let excess = propagate_cycle(graph, cycle_edges).unwrap_or(0.0);
        cumulative_excess += excess;
        cycle.clear();
    }
    if cumulative_excess < EPSILON {
        return None;
    }
    Some(cumulative_excess)
}
```

Appendix C. Algorithm of flow propagation through the cycle.

```
pub fn propagate_cycle(
    graph: &mut DirectedGraph,
    cycle: Option<Vec<usize>>,
) -> Option<f64> {
    if cycle == None {
        return None;
    }
    let cycle = cycle.unwrap();
    let mut flow: Option<f64> = None;
    for &edge_id in cycle.iter() {
        let edge = &graph.edges_list[edge_id];
        let available = edge.capacity - edge.flow;
        if available < EPSILON {
            return None;
        }
        flow = match flow {
            None => Some(available * edge.amplification),
            Some(value) => Some(value.min(available) * edge.amplification),
        };
    }
    let mut flow = match flow {
        None => return None,
        Some(value) => value,
    };
    let flow_in_destination = flow;

    let &cycle_start = cycle.first().unwrap();
    let cycle_start = graph.edges_list[cycle_start].from_id;

    graph.nodes[cycle_start].excess += flow_in_destination;

    for &edge_id in cycle.iter().rev() {
        let reverse_edge = graph.reverse_edge_ids[edge_id];
        let reverse_edge = &mut graph.edges_list[reverse_edge];

        reverse_edge.flow -= flow;
        flow *= reverse_edge.amplification;

        let edge = &mut graph.edges_list[edge_id];
        edge.flow += flow;
    }

    return Some(flow_in_destination);
}
```

Appendix D. Computation of potentials of vertices algorithm.

```
fn recalculate_potentials(graph: &DirectedGraph, potentials: &mut Vec<f64>, barrier: f64) {
    let mut stack: Vec<usize> = vec![];
    let mut status = vec![0; graph.n()];

    fn dfs(
        node_id: usize,
        graph: &DirectedGraph,
        status: &mut Vec<i32>,
        potentials: &Vec<f64>,
        stack: &mut Vec<usize>,
    ) {
        status[node_id] = 1;
        for &edge_id in &graph.adj_lists[node_id] {
            let edge = &graph.edges_list[edge_id];
            if edge_value(edge, &potentials) == None || status[edge.to_id] != 0 {
                continue;
            }
            dfs(edge.to_id, graph, status, potentials, stack);
        }
        stack.push(node_id);
    }

    for node_id in (0..graph.n()).rev() {
        if status[node_id] != 0 {
            continue;
        }
        dfs(node_id, graph, &mut status, &potentials, &mut stack)
    }

    for (topological_order, node_id) in stack.iter().rev().enumerate() {
        potentials[*node_id] += (topological_order as f64) * barrier / (graph.n() as f64);
    }
}
```

Appendix E. Canceling flow generating cycles algorithm.

```
pub fn cancel_cycles(graph: &mut DirectedGraph) {
    let mut counter = 0;
    let mut potentials = vec![0.0; graph.n()];
    let edge_value = |edge: &Edge| -> Option<f64> {
        if edge.capacity - edge.flow < EPSILON {
            None
        } else {
            Some(-(edge.amplification).log(E))
        }
    };
    let mut barrier: Option<f64> = None;
    for edge in &graph.edges_list {
        let cost = match edge_value(&edge) {
            None => continue,
            Some(value) => value,
        };
        barrier = Some(barrier.unwrap_or(-cost).max(-cost));
    }
    if let Some(mut barrier) = barrier {
        while has_cycles(graph, &mut counter) {
            remove_negative_cycles(graph, &potentials);
            recalculate_potentials(graph, &mut potentials, barrier);
            barrier *= 1.0 - 1.0 / (graph.n() as f64);
        }
    }
}
```

Appendix F. Maximum flow computation algorithm.

```
pub fn find_flow(graph: &mut DirectedGraph) -> f64 {
    let mut result = 0.0;
    let mut step = 0;
    while has_augmenting_path(&graph) != None {
        cancel_cycles(graph);
        let augmenting_path = get_augmenting_path(graph);
        let augmenting_path_edges = get_path_edge_ids(augmenting_path);

        let flow = propagate_path(graph, augmenting_path_edges);
        result += flow.unwrap_or(0.0);
    }
    result
}
```

References

[1] Andrew V. Goldberg An Efficient implementation of a scaling minimum-cost flow algorithm - Journal of Algorithms, 1997

[2] A. V. Goldberg, S. A. Plotkin, and E. Tardos. Combinatorial algorithms for the ´ generalized circulation problem. Mathematics of Operations Research, 16:351– 379, 1991

[3] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. Journal of Computer and System Sciences, 26:362–391, 1983

[4] F. Glover and D. Klingman. On the equivalence of some generalized network flow problems to pure network problems. Mathematical Programming, 1973.

[5] K. D. Wayne. Generalized maximum flow algorithms, 1991

[6] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. Canadian Journal of Mathematics, 1956

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Introduction to algorithms. — Third edition: K. I. C., 2019

[8] V. Chvatal. Linear programming. W. H. Freeman, New York, 1983