Jagiellonian University
Department of Theoretical Computer Science

Marcin Wykpis

Algebraic algorithms in graph problems

Bachelor Thesis

Supervisor: dr hab. Krzysztof Turowski

July 2025

Contents

Chapter 1

Preliminaries

1.1 Graphs

We begin this section with some basic notations and definitions of graph theory.

Definition 1 (Graph [24]). *A graph $G = (V, E)$ consists of V, a nonempty set of **vertices** (or nodes) and E, a set of **edges**. Each edge has either one or two vertices associated with it, called its **endpoints**. An **connects** its endpoints.*

Definition 2 (Directed graph [24]). *A directed graph (or digraph) (V, E) consists of a nonempty set of vertices V and a set of **directed edges** (or **arcs**) E. Each directed edge is associated with an ordered pair of vertices. The directed edge associated with the ordered pair (u, v) is **starts** at u and **ends** at v.*

Definition 3 (Weighted graph [24]). *Graphs that have a number associated with each edge are called **weighted graphs**. Formally the graph $G = (V, E)$ is weighted if there exists an associated function $w : E \to \mathbb{R}$.*

Definition 4 (Simple graph [24]). *A simple graph $G = (V, E)$ is a subset of the set E of edges of the graph such that no two edges are incident with the same vertex.*

Definition 5 (Bipartite graph [24]). *A simple graph G is called **bipartite** if its vertex set V can be partitioned into two disjoint sets V_1 and V_2 such that every edge in the graph connects a vertex in V_1 and a vertex in V_2 (so that no edge in G connects either two vertices in V_1 or two vertices in V_2). When this condition holds, we call the pair (V_1, V_2) a **bipartition** of the vertex set V of G.*

Definition 6 (Subgraph [24]). *A **subgraph** of a graph $G = (V, E)$ is a graph $H = (W, F)$, where $W \subseteq V$ and $F \subseteq E$. A subgraph H of G is a **proper subgraph** of G if $H \neq G$.*

Definition 7 (Induced graph [24]). *Let $G = (V, E)$ be a simple graph. The **subgraph induced** by a subset W of the vertex set V is a graph $G[W] = (W, F)$, where the edge set F contains an edge in E if and only if both endpoints of this edge are in W.*

1.2 Matchings

Definition 8 (Matching [24]). ***Matching*** *M in a simple graph $G = (V, E)$ is a subset of the set E of edges of the graph such that no two edges are incident with the same vertex.*

Definition 9 (Maximum matching [24]). *A **maximum matching** in a graph G is a matching with the largest number of edges.*

Definition 10 (Perfect matching). *A **perfect matching** is a matching of exactly $\frac{|V|}{2}$ cardinality.*

We now define a decision and an optimization problems related to the above definitions.

Problem 1 (Perfect matching decision problem). *Given a graph G determine if the graph contains some perfect matching.*

Problem 2 (Maximum matching optimization problem). *Given a graph G, find its maximum matching M.*

1.3 Flows

Definition 11 (Flow [18]). *Given a directed weighted graph $G = (V, E)$, we define **flow** $f : E \to \mathbb{R}$ to be a function that assigns value $f_e := f(e)$ to each edge from G.*

The flow can pass both directions of an edge. For $e = (u, v)$ we view positive value of f_e as a flow from vertex u to v with a value of f_e. Similarly, negative flow $-f_e$ passes from v to u.

Definition 12 ([18]). *Let $E^+(v)$ (respectively $E^-(v)$) be the edges for which the flow enters (leaves) vertex v, i.e. for $e = (u, v)$, $f_e > 0$ (respectively $f_e < 0$).*

Definition 13 (Demand vector [18]). *We define a **demand vector** $\sigma : V \to \mathbb{R}^n$ and denote $\sigma_v := \sigma(v)$ for $v \in V$. Simultaneously, we define a **σ-flow**, to be a flow satisfying the flow conservation constraints:*

$$\sum_{e \in E^+(v)} f_e - \sum_{e \in E^-(v)} f_e = \sigma_v \quad \text{for each } v \in V \tag{1.1}$$

For convenience we will be focusing on a maximum flow problem for a specific demand vector $\chi_{s,t} : V \to \mathbb{R}^n$ defined as follows:

$$\chi_{s,t} = \begin{cases} -1 & \text{for } v = s, \\ 1 & \text{for } v = t, \\ 0 & \textit{otherwise.} \end{cases}$$

Definition 14 ([18]). *For each edge $e = (u, w) \in E$ we define the capacities u_e^+, u_e^- to be:*

$$u^+_{(v,w)} = u_{(v,w)} \quad \textit{and} \quad u^-_{(v,w)} = -u_{(w,v)}.$$

.

Definition 15 (Feasibility [18]). *We say that σ-flow f is **feasible** in G if and only if the capacity of each edge is not smaller than the flow running through them i.e.*

$$-u_e^- \leq f_e \leq u_e^+ \quad \textit{for all } e \in E.$$

Note that for a specific demand vector (denoted later as $F\chi_{s,t}$) with $\delta_s = F, \delta_t = -F$ and zeros elsewhere, we get the standard definition for maximum flow problem. We call such flow a s-t flow with value F.

Problem 3 (Maximum s-t flow optimization problem). *Given a directed, weighted graph G, a source vertex s and a target vertex t, find a feasible s-t flow with maximum possible value.*

Given a specific demand vector, we can also formulate a maximum flow decision problem.

Problem 4 (Maximum flow decision problem). *Given a directed, weighted graph G, and a demand vector σ, determine if there is a feasible flow satisfying the demand.*

Given a graph G, solving maximum flow problem is to find a feasible s-t flow with maximum possible value. We denote this value as F^*.

Definition 16 (Upper and lower capacities [18]). *We define **upper** and **lower** capacities to be:*

$$\hat{u}_e^+(f) := u_e^+ - f_e \quad \textit{and} \quad \hat{u}_e^-(f) := u_e^- + f_e$$

*We also define a **minimal capacity** $\hat{u}_e(f) := \min\{u_e^+(f), u_e^-(f)\}$. Note that for a feasible flow f, we have $\hat{u}_e(f) \geq 0$.*

Now let us introduce an important notion of a residual graph.

Definition 17 (Residual graph [18]). *Given a directed weighted graph $G = (V, E)$ and a weight function $w : V \to \mathbb{R}_+$ and a feasible σ-flow f, we define the **residual graph** G_f as $(V, E, \hat{u}(f))$.*

1.4 Algebraic notation

This paper focuses on maximum matching and maximum flow algorithms based on an algebraic properties of graphs. These approach differs from the purely combinatorial algorithms. It utilizes algebraic concepts such as: encoding graph into a matrix, matrix rank and determinant, linear dependency, solving linear

system of equation. This unique approach enables a creation of new, algorithms that further improves the best known time complexity for the presented problems.

Here we define notions used in the later sections:

Definition 18 (p-norm [32])**.** *Give a vector $x \in \mathbb{R}^n$ and a number $p \in \mathbb{Z}_+$ we define* ***the p-norm*** *of x to be*

$$||x||_p := \left(\sum_{i=1}^{n} |x_i|^p \right)^{1/p}.$$

We also define the ***maximum norm*** *of x to be*

$$||x||_\infty = \max\{x_i \colon i \in \{1, ..., n\}\}.$$

It is easy to prove that in fact $||x||_\infty = \lim_{p \to \infty} ||x||_p$.

Definition 19 (Laplacian matrix [30])**.** *Given a graph $G = (V, E)$, we define a* ***Laplacian matrix*** *$L \in \mathbb{R}^{n \times n}$ to be:*

$$L_{i,j} = \begin{cases} \deg_G(i) & \text{for } i = j, \\ -1 & \text{for } i \neq j \text{ and } i \text{ is adjacent to } j \text{ in } G, \\ 0 & \text{otherwise.} \end{cases}$$

Additionally, given a graph $G = (V, E)$ and its weights $w : V \to \mathbb{R}_+$ we define a ***weighted Laplacian matrix*** *to be:*

$$L_{i,j} = \begin{cases} \sum_{k \in V} w_{i,k} & \text{for } i = j, \\ -w_{i,j} & \text{for } i \neq j \text{ and } i \text{ is adjacent to } j \text{ in } G, \\ 0 & \text{otherwise.} \end{cases}$$

Definition 20 (SVD decomposition [33])**.** *Given a $m \times n$ matrix A, a diagonal $m \times n$ matrix Σ and orthonormal matrices U, V of size $m \times m$ and $n \times n$, we call a product $U\Sigma V$ the* ***SVD decomposition of*** *A if and only if $A = U\Sigma V$.*

Definition 21 (Moore-Penrose pseudoinverse [31])**.** *Given a $n \times n$ matrix A and its singular value decomposition: $A = U\Sigma V$, we define* ***Moore-Penrose inverse*** *A^+ to be:*

$$A^+ := U\Sigma^+ V,$$

where Σ^+ is defined as follows:

$$\Sigma_i^+ := \begin{cases} \frac{1}{\Sigma_i} & \text{for } \Sigma_i \neq 0, \\ 0 & \text{otherwise.} \end{cases}$$

Lemma 1.4.1 ([31])**.** *The pseudoinverse provides a least squares solution to a system of linear equations. Consequently if a system of linear equations $Ax = b$ has a solution, $x := A^+ \cdot b$ is one of them.*

Chapter 2

Perfect Matching

2.1 Introduction

Finding a maximum matching in a given a undirected graph is one of the most well-known graph problems. The research began with a polynomial algorithm for a bipartite graph by Kuhn [15], later improved by Hopcroft and Karp [12].

Finding a maximum matching in a general graphs is however far more difficult. The blossom algorithm introduced by Edmonds [5] returned a maximum matching in $O(n^4)$ time in a general case. This result was further improved by Gabow [9] to $O(n^3)$ and later by Micali and Vazirani [19] with a $O(n^{2.5})$ algorithm for a dense graph.

In Table 2.1 there is a brief summary of the best results achieved over the years:

Algorithm	**Time complexity**	**Graph type**
Kuhn 1955 [15]	$O(n^3)$	Bipartite
Edmonds 1965 [5]	$O(n^4)$	General
Hopcroft-Karp 1973 [12]	$O(n^{2.5})$	Bipartite
Gabow 1976 [9]	$O(n^3)$	General
Micali-Vazirani 1980 [19]	$O(n^{2.5})$	General
Mucha-Sankowski 2004 [20]	$O(n^\omega)$	**General**

Table 2.1: Time complexity of maximum matching algorithms for a dense graph.

In this section we will present an algorithm from [20] that improves on the previous results. Its time complexity is equal to $O(n^\omega)$, that is, the time complexity of multiplying two $n \times n$ matrices. The smallest known value of ω is about 2.371339 as of date with the results improving every few years (see Table 2.2).

Algorithm	Bound on ω
Naive algorithm	3.0
Strassen 1969 [27]	2.8074
Schönhage 1981 [25]	2.522
Coppersmith, Winograd 1990 [4]	2.3755
Williams 2012 [34]	2.3729
Alman, Duan, Williams, Xu, Xu, and Zhou 2024 [1]	2.371339

Table 2.2: Timeline of matrix multiplication exponent research.

2.2 Preliminaries

One of the key elements of the algorithm is the ***skew symmetric adjacency matrix*** encoding the graph.

Definition 22 ([20])**.** *For a graph $G = (V, E)$ where $V = \{v_1, ..., v_n\}$ we define a skew symmetric adjacency matrix $\tilde{A}(G)_{i,j}$ to be:*

$$\tilde{A}(G)_{i,j} = \begin{cases} x_{i,j} & \text{for } \{v_i, v_j\} \in E \text{ and } i < j, \\ -x_{i,j} & \text{for } \{v_i, v_j\} \in E \text{ and } i > j, \\ 0 & \text{otherwise.} \end{cases}$$

where $x_{i,j}$ is an unique variable for each edge $\{v_i, v_j \in E\}$.

We can view the determinant $\det A(G)$ as a polynomial over all $x_{i,j}$ variables. We call such polynomial a *symbolic determinant*.

Tutte [29] has shown an interesting theorem tying together the adjacency matrix and the perfect match problem:

Theorem 2.2.1 ([29])**.** *The symbolic determinant $\det \tilde{A}(G) \neq 0$ if and only if G has a perfect matching.*

This work was later generalized by Lovász [16]:

Theorem 2.2.2 ([16])**.** *The rank of the skew symmetric adjacency matrix $\tilde{A}(G)$ is equal to twice the size of maximum matching of G.*

There is however a problem with practical application of these two theorems. The symbolic determinant of $\tilde{A}(G)$ can be of an exponential length in terms of n, much to slow for what we want to achieve. Fortunately, we can avoid this problem with the use of randomness. We choose a number $R = n^{O(1)}$ (we will define the exact value of the exponent later) and replace each variable with a number uniformly taken from $\{1, ..., R\}$. We call this substituted matrix $A(G)$ an *random adjacency matrix* of G. By Lovász [16] we have

Theorem 2.2.3 ([16])**.** *The rank (i.e. the number of independent rows) of $A(G)$ is at most twice the size of maximum matching of G. The equality holds with the probability at least $1 - \frac{n}{R}$.*

This result gives us a direct algorithm for checking if a perfect matching exists in a given graph G. We can simply create the random adjacency graph and compute its determinant. We can also construct a perfect matching thanks to an observation by Rabin and Vazirani [23]:

Theorem 2.2.4 ([23]). *With high probability, $A(G)^{-1}_{i,j} \neq 0$ if and only if the graph $G \setminus \{v_i, v_j\}$ has a perfect matching.*

We call an edge $\{v_i, v_j\}$ from G *allowed* when $A(G)^{-1}_{i,j} \neq 0$. Theorem 2.2.4 shows that (with high probability) an edge e is allowed if there exists a perfect matching in G which contains e. This observation will be the key component of our algorithm Thus we can already construct a naive algorithm directly using Theorem 2.2.4. For a graph G we first construct $A(G)$ and $A(G)^{-1}$. Next we find an allowed edge (v_i, v_j) and create an induced subgraph G' on $V' := V \setminus \{v_i, v_j\}$. We then repeat the above steps until all vertices are matched. Creating the matrix in each step might by prone to accumulating the error probability with each step. The following theorem ensure that is not a problem by computing the random values only once.

Furthermore, Rabin and Vazirani [23] showed the following theorem:

Theorem 2.2.5 ([23]). *If $A(G)$ is non-singular, then for every v_i there exists a v_j such that $A(G)_j \neq 0$ and $A^{-1}_{j,i} \neq 0$, i.e. $\{v_i, v_j\}$ is an allowed edge. Moreover, the matrix $A(G)$ with rows i, j and columns j, i removed is also non-singular.*

This theorem will be sufficient to establish a bound for the probability of an error. Since most of the following claims will come across the randomness problem, we will omit the "with high probability" part from now on.

We also define a slightly modified adjacency matrix $A(G)$ for a bipartite graph:

$$\tilde{A}(G)_{i,j} = \begin{cases} x_{i,j} & \text{for } \{u_i, v_j\} \in E \text{ and } i < j \\ 0 & \text{otherwise.} \end{cases}$$

As it turns out all the previously defined theorems apply to the bipartite graphs as well.

2.3 Maximum and perfect matchings

In the previous section we presented Theorem 2.2.4 and the main idea behind finding the perfect matching. This theorem cannot be efficiently applied in the maximum matching problem. After all we do not know the order in which the allowed edges should be removed.

Fortunately we only have to worry about the perfect matching problem: Ibarra and Moran [13] (bipartite case) and Vazirani [23] (general case) showed have proven a reduction between those two problems.

Theorem 2.3.1 ([23]). *The problem if finding a maximum matching can be reduced in randomized time $O(n^\omega)$ to the problem of finding a perfect matching.*

2.4 Constructing a perfect matching

We can construct a perfect matching with this naive approach.

Algorithm 1 Naive matching

```
1: function NaiveGeneralMatching(G)
2:     Construct A(G)
3:     M ← ∅
4:     for i = 1...n/2 do
5:         Compute A(G)^{-1}
6:         Find in G an allowed edge {u, v}
7:         M ← M ∪ {{u, v}}
8:         G ← G \ {u, v}
9:     end for
10:    return M
11: end function
```

Notice that we compute the reverse $A(G)^{-1}$ in each iteration. We can calculate this matrix in the $O(n^\omega)$ time complexity. This however results in a $O(n^{\omega+1})$ algorithm. We can improve on this result by applying a clever trick that omits recalculating the matrix.

Theorem 2.4.1. *(**Elimination Theorem**, Theorem 3.1 in [20]). Let*

$$A = \begin{bmatrix} a_{1,1} & v^T \\ u^T & B \end{bmatrix} \qquad A^{-1} = \begin{bmatrix} \hat{a}_{1,1} & \hat{v}^T \\ \hat{u} & \hat{B} \end{bmatrix}$$

where $\hat{a}_{1,1} \neq 0$. then $B^{-1} = \hat{B} - \hat{u}\hat{v}^T/\hat{a}_{1,1}$.

Proof. $AA^{-1} = I$ and thus

$$\begin{bmatrix} a_{1,1}\hat{a}_{1,1} + v^T\hat{u} & a_{1,1}\hat{v}^T + v^T\hat{B} \\ u\hat{a}_{1,1} + B\hat{u} & u\hat{v}^T + B\hat{B} \end{bmatrix} = \begin{bmatrix} I_1 & 0 \\ 0 & I_{n-1} \end{bmatrix}$$

Now solving B in terms of $\hat{u}, \hat{v}, \hat{a}_{1,1}$ gives us $B\hat{B} - B\hat{u}\hat{v}^T/\hat{a}_{1,1} = I$ and finally $B^{-1} = \hat{B} - \hat{u}\hat{v}^T/\hat{a}_{1,1}$. □

Computing the matrix B is in fact a step of the Gaussian elimination. We can further modify this procedure to eliminate any row i and column j.

This theorem is sufficient to construct a simple version on the algorithm with a $O(n^3)$ time complexity. We present two variants of this algorithm. One for a bipartite graph and the second for a general case.

Algorithm 2 Simple algorithm for perfect matching in bipartite graphs

```
1: function SimpleBipartiteMatching(G)
2:     B ← A^{-1}(G)
3:     M ← ∅
4:     for c = 1...n do
5:         find a row r for which A(G)_{c,r} ≠ 0 and B_{r,c} ≠ 0
6:         eliminate r-th row an c-th column of B using Theorem 2.4.1
7:         M ← M ∪ {(r, c)}
8:     end for
9:     return M
10: end function
```

Algorithm 3 Simple algorithm for perfect matching in general graphs

```
1: function SimpleGeneralMatching(G)
2:     B ← A^{-1}(G)
3:     M ← ∅
4:     for c = 1...n do
5:         if column c is not eliminated then
6:             find a row r for which A(G)_{c,r} ≠ 0 and B_{r,c} ≠ 0
7:             eliminate r-th and c-th row and column of B
8:             M ← M ∪ {{r, c}}
9:         end if
10:    end for
11:    return M
12: end function
```

2.4.1 Matching verification

We will now present simple lazy Gaussian elimination algorithm running in $O(n^\omega)$ time. For now the elimination will not allow pivoting. This algorithm will be used to develop a matching verification algorithm. Given a matching it computes its inclusion-wise maximal allowed subset. The simple lazy Gaussian elimination will also be a foundation for a more complex lazy Gaussian elimination that will be used in the main algorithm.

As we have seen before, recomputing the whole matrix in each step of the elimination leads to a $O(n^{\omega+1})$ algorithm. To improve on this, we will perform a lazy elimination i.e. we will remember how the matrix were changed and perform these changes later, a couple at the same time. We represent such lazy eliminated row v_i and column u_i of a matrix A with three values: $l_i = (u_i, v_i, a_{i,i})$. Assuming we want to apply k changes $l_1, l_2, ..., l_2$, we have to compute

$$A \leftarrow A - \sum_{i=1}^{k} u_i v_i^T / c_i$$

We can however represent this as

$$A \leftarrow A - UV$$

where U is a matrix with columns $u_1, u_2, ..., u_k$, and V is a matrix with rows $v_1^T/a_{1,1}, v_2^T/a_{2,2}, ...{}_k^T/a_{k,k}$. We can compute the product UV with some fast matrix multiplication algorithm running in $O(n^\omega)$. For a set of rows R and a set of columns C let us define an UPDATE(R, C) function that applies accumulated changes to the $A_{R,C}$ sub-matrix.

Algorithm 4 Update function

```
1: function UPDATE(A, (r_1, r_2), (c_1, c_2), lazy[])
2:     for i = 1...lazy.length do
3:         u, v, a ← lazy[i]
4:         U_{i,c_1:c_2} ← u_{c_1:c_2} / a
5:         V_{r_1:r_2,i} ← v_{r_1:r_2} / a
6:     end for
7:     A ← A − U × V
8: end function
```

Algorithm 5 Simple lazy Gaussian elimination (no pivoting)

```
1: function SIMPLEGAUSSIANELIMINATION(A)
2:     for i = 1...n do
3:         lazy[i] ← (A_{i,*}, A_{*,i}, A_{i,i})
4:         let j be the largest integer for which 2^j | i
5:         UPDATE(A, (i + 1, i + 2^j), (i + 1, n), lazy[i − 2^j + 1 : i])
6:         UPDATE(A, (i + 2^j, n), (i + 1, i + 2^j), lazy[i − 2^j + 1 : i])
7:     end for
8: end function
```

Theorem 2.4.2 (Theorem 3.3 in [20])**.** *Iterative Gaussian elimination without row or column pivoting can be implemented in time $O(n^\omega)$ using a lazy updating scheme.*

We will prove this algorithm runs in $O(n^\omega)$ with the following lemma.

Lemma 2.4.3. *(Lemma 3.2 in [20]) The number of cell value's changes performed by Algorithm 4 in the* UPDATE *steps is at most 2^j.*

Proof. In the i-th iteration only the $i+1, ..., i+2^j$ rows and columns are updated. We defined j such that $2^j | i$ and $2^{j+1} \nmid i$ and consequently $2^{j+1} | i-2^j$. This means that the $i-2^j+1, ...i+2^j$ rows and columns were updated in the $i-2^j$ iteration. The number of changes to apply is thus at most 2^j. □

In UPDATE we multiply a $2^j \times 2^j$ and $2^j \times n$ matrices. If we split the $2^j \times n$ matrix into $n/2^j$ smaller 2^j matrices, we can compute the product in

$O(n/2^j \cdot (2^j)^\omega) = O(n(2^j)^{\omega-1})$ time. For a given j such update will be computed $n/2^j$ times. If we now sum the time complexity over all possible j we get:

$$\sum_{j=0}^{\lceil \log n \rceil} n/2^j \cdot n(2^j)^{\omega-1} = n^2 \sum_{j=0}^{\lceil \log n \rceil} (2^{\omega-2})^j = O(n^2(2^{\omega-2})^{\lceil \log n \rceil}) = O(n^\omega).$$

Using the above simple lazy elimination algorithm we construct a verification procedure returning the maximal allowed subset of a matching.

Theorem 2.4.4. *(theorem 3.4 in [20]) Let G be a graph having a perfect matching. For any matching M of G, an inclusion-wise maximal allowed (i.e. extendable to a perfect matching) submatching M' of M can be found in time $O(n^\omega)$.*

Proof. Let $M = \{\{v_1, v_2\}, \{v_3, v_4\}, ..., \{v_{k-1}, v_k\}\}$ and let $v_{k+1}, ...v_n$ be the unmatched vertices. We compute the matrix $A(G)^{-1}$ and swap columns v_{2i-1}, v_{2i} for each $i = 1, ..., \frac{k}{2}$. As a result we get a matrix with column order $v_2, v_1, \ldots v_k, v_{k-1}, v_{k+1}, ..., v_n$. We then perform the simple lazy Gaussian elimination on such matrix. Since it does not support pivoting, we skip the i-th elimination step completely if it can not be computed (i.e. $a_{i,i} = 0$). If the elimination succeed we append the row-column pair to the maximal matching M'. Since no pivoting is done we only eliminate pairs that were present in M. Thus $M' \subset M$. Each value on the matrix diagonal is equal to 0 after performing the verification algorithm. Since the only available matches were placed on the diagonal we get the maximality of M'. □

2.4.2 Perfect matching in bipartite graphs

Let $G = (V_1 \cup V_2, E)$ where $|V_1| = |V_2| = n$ be a bipartite graph with a perfect matching. Here we present an algorithm that finds such perfect matching in $O(n^\omega)$ time. Previously we have established the simple lazy Gaussian elimination that runs in the required time. However it can not be directly applied to our problem. We do not know the order of rows to eliminate. Performing it on a matrix with randomly permuted columns would result with some maximal matching that (with high probability) will not be perfect. We can however develop a more complex version of this algorithm that allows rows pivoting.

Lazy Gaussian elimination

We present the lazy Gaussian elimination based on the Hopcroft-Bunch [3] algorithm. Opposed to the recursive approach, we present it as an iterative algorithm. Such implementation emphasizes the similarity between this algorithm and simple bipartite matching.

Notice, that the only difference between this algorithm and NaiveGaussianElimination is the UpdateColumns call. To allow row pivoting we would like to update whole columns. This is however a little bit more challenging. When updating columns $c+1, c+2, ..., c+2^j$ we have to perform the lazy updates of columns $c-2^j+1, c-2^j+2, ...c$. To simplify the argument

Algorithm 6 Simple algorithm for perfect matching in bipartite graphs

function BIPARTITEMATCHING(G)
  $A \leftarrow A(G)$
  $B \leftarrow A^{-1}$
  $M \leftarrow \emptyset$
  for $c = 1...n$ **do**
    find a row r for which $A_{c,r} \neq 0$ and $B_{r,c} \neq 0$
    `lazy[c]` $= (A_{*,c},\ A_{r,c})$
    let j be the largest integer for which $2^j | c$
    UPDATECOLUMNS($A, (c+1, c+2^j)$, `lazy`)
    $M \leftarrow M \cup \{(r, c)\}$
  end for
  return M
end function

let the row elimination order be the same as the column one (this step can be avoided with a simple permutation array). In the first update we use the lazy eliminated $(c - 2^j + 1)$-th row and $(c - 2^j + 1)$-th column. To perform the second update we need to know the $(c - 2^j + 2)$-th row and $(c - 2^j + 2)$-th column. However, without performing the first update we do not know the value of the $(c - 2^j + 2)$-th row. To resolve this issue we could use the lazy updating scheme once again. We perform lazy updates on the selected columns. Below is a pseudocode of the UPDATECOLUMNS function.

Algorithm 7 UpdateColumns function

function UPDATECOLUMNS($A, (c_1, c_2)$, `lazy[]`)
  for $i = 1...j$ **do**
    let l be the largest integer for which $2^l | i$
    UPDATE($A, (0, n), (c+1, c+2^j)$, `superLazy`$[i - 2^l + 1 : i - 1]$)
    `superLazy[`i`].col`, `superLazy[`i`].val` $\leftarrow$ `lazy[`$c - 2^j + 1 + i$`]`
    `superLazy[`i`].row` $\leftarrow A_{(c-2^j+1+i),*}$
  end for
end function

Now lets analyze the time complexity of the UPDATECOLUMN operation. In order to update columns $c+1, c+2, ..., c+2^j$ we have to perform 2^j updates. To determine the time complexity of a singly iteration we will look at the values provided to the update function. Remind that the partition called on a $a \times b$ submatrix with k updates runs in $O(\frac{a}{k} \cdot \frac{b}{k} \cdot k^\omega)$. In UPDATECOLUMN we call UPDATE 2^l time for a $n \times 2^l$ submatrix. This leads to $O(\frac{n}{2^l} \cdot \frac{2^l}{2^l} \cdot 2^{l\omega}) = O(n \cdot (2^l)^{\omega-1})$ running time. Just like in simple elimination, the total time complexity of UPDATECOLUMN is now equal $O(\sum_{l=0}^{j} \frac{2^j}{2^l} \cdot n \cdot (2^l)^{\omega-1}) = O(n \cdot (2^j)^{\omega-1})$. Now summing this value for all j we get $O(\sum_{j=0}^{\lceil \log n \rceil} \frac{n}{2^j} \cdot n \cdot (2^j)^{\omega-1}) = O(n^2 \cdot (2^{\log n})^{\omega-2}) = O(n^\omega)$.

2.4.3 Perfect matching for general graphs

In order to remove some allowed edge $\{u, v\}$ in the general case we need to eliminate the u-th column and v-th row as well as the v-th row and u-th column. The lazy computation is scheme is not capable of doing this.

In the general case we will try to utilize a greedy matching algorithm along with the verification algorithm to match a number of allowed edges. If this approach does not make sufficient progress we proceed with another idea. We will use structural properties of the graph to partition it into smaller subgraphs. We can then call the general matching algorithm on each of them.

Greedy matching algorithm

Algorithm 8 Greedy matching algorithm

```
1: function GreedyMatching(G)
2:     M ← ∅
3:     for all {u, v} ∈ E do
4:         if u, v ∉ ⋃_{e∈M} then
5:             M ← M ∪ {{u, v}}
6:         end if
7:     end for
8:     return M
9: end function
```

Lemma 2.4.5. *Given a graph G with a maximum matching M^*, the greedy matching algorithm returns a matching of size at least $|M^*|/2$.*

Proof. By contradiction, let us have some matching $M = \{u_1, ..., u_k\}$ obtained through the greedy algorithm. Let $v_1, ..., v_l$ for $l > k$ be the vertices that were matching in some maximum matching M^* but were not matched under M. There are no edges $\{v_i, v_j\}$ in G. If there were, the greedy algorithm would match some of them. In G there can only be edges $\{u_i, u_j\}$ and $\{u_i, v_j\}$ thus $|M^*| \leq k = 2|M|$. □

The partition algorithm

The only part remaining is the Partition algorithm. We proceed with a sequence of definitions and theorems necessary in order to establish the correctness and the running time of the missing function.

Definition 23 ([20])**.** *Graph G is called **elementary** if it has a perfect matching and its allowed edges form a connected spanning subgraph.*

Definition 24 ([20])**.** *Given graph G we define a relation $\sim_G$. For all $u, v \in V(G)$, $u \sim_G v$ if and only if $u = v$ or $G \backslash \{u, v\}$ does not have a perfect matching. Let also $P(G)$ be the set of equivalence classes of $\sim_G$.*

Algorithm 9 Perfect matching algorithm for general graphs

```
1: function GeneralMatching(G)
2:     M_g ← GreedyMatching(G)
3:     M_m ← MatchingVerification(G, M_g)
4:     G ← G \ V(M_m)
5:     if |M_m| ≥ n/8 then
6:         return M_m ∪ GeneralMatching(G)
7:     else
8:         return M_m ∪ Partition(G)
9:     end if
10: end function
```

Lovász [17] proved the following lemma

Theorem 2.4.6 ([17]). *If G is elementary, then $\sim_G$ is an equivalence relation.*

For the sake of notation convenience, we will treat graphs as a set of vertices. For example $G[C \cup S]$ will mean a subgraph induced from G by the vertices of the subgraphs C and S.

From [17] we have also some useful structural properties of $P(G)$:

Theorem 2.4.7 ([17]). *Let G be elementary, let $S \in P(G)$ with $|S| \geq 2$ and let C be any component of $G \setminus S$. Then:*

1. *the bipartite graph G'_s obtained from G by concatenating each component of $G \setminus S$ to a single vertex and deleting edges in S is elementary;*
2. *the graph C is factor-critical, i.e. for any vertex $v \in C, C \setminus \{v\}$ has a perfect matching;*
3. *the graph C' obtained from $G[C \cup S]$ by contracting the set S to a single vertex u_c is elementary;*
4. *$P(C') = \{\{u_c\}\} \cup \{T \cap C \colon T \in P(G)\}$.*

The above theorem shows some useful structural properties derived from $P(G)$. Given a set $S \in P(G)$ we can compute a bipartite matching M_s of G'_s. For each pair (s, c) where $s \in S$ and c is some contracted component C, we will match s with its neighbor v from C. From 2. we know that that $C \setminus \{v\}$ has a perfect matching and thus we reduced the partition algorithm to computing one bipartite matching and $|S| - 1$ general matchings.

Lemma 2.4.8 (Lemma 4.3 from [20]). *The total number of vertices of graphs in which Partition finds perfect matchings by calling the BipartiteMatching algorithm or the GeneralMatching algorithm is equal to n.*

Proof. Since the GeneralMatching algorithm finds a perfect matching, we know that at least n vertices were matched. We now need to show that there

was no more than n of such vertices. The only lines were we create vertices are 7. and 16.. This does not however create additional matches. In lines 14. and 19. respectively we get a free matched vertex in place of the contracted one. □

Algorithm 10 The partition algorithm

```
1: function PARTITION(G)
2:     M ← ∅
3:     for all elementary component G_e of G do
4:         Let S ∈ P(G_e) with |S| = k ≥ 2
5:         Let C_1, C_2, ..., C_k be the connected components of G_e \ S
6:         Let C_1 be the largest component
7:         Let C'_1 be the graph G[S ∪ C_1] with S contracted to a vertex s
8:         if P(C'_1) contains a non-trivial class then
9:             M'_1 ← PARTITION(C'_1)
10:        else
11:            M'_1 ← GENERALMATCHING(C'_1)
12:        end if
13:        Let c be the vertex matched with s in M'_1.
14:        Let v ∈ S be a neighbor of c.
15:        M ← M ∪ (M'_1 \ {{s, c}}) ∪ {{v, c}}
16:        Construct graph G' from S\{v} and components C_2, ...C_k contracted
   to c_2, ...c_k
17:        M_p ← BIPARTITEMATCHING(G')
18:        for all (c_i, s) ∈ M_p do
19:            Let v ∈ C_i be a neighbor of s ∈ S.
20:            M ← M ∪ {{v, s}}∪ GENERALMATCHING(C_i \ {v})
21:        end for
22:    end for
23:    return M
24: end function
```

So far we reduced partition algorithm into bipartite matchings and general matchings on graphs a factor smaller than the graph G. BIPARTITEMATCHING might be called on large graphs (even of size $\Theta(n)$). They operate on vertices in S and $O(|S|)$ contracted components. Thus a vertex from G cannot be however a part of S in two different partition step. With Lemma 2.4.8 in place, the runtime of all bipartite matchings is $O(n^\omega)$.

This lemma shows that the cumulative run-time of all BIPARTITEMATCHING calls is $O(n^\omega)$.

Lemma 2.4.9 (Lemma 4.4 from [20])**.** *The PARTITION algorithm calls the GENERALMATCHING algorithm for graphs with $\leq \frac{7}{9}n$ vertices.*

Proof. The smaller components $C_2, ..., C_k$ have less than $\frac{1}{2}n \leq \frac{7}{9}n$ elements, since C_1 is the biggest component. As for the biggest component, we only call GENERALMATCHING on C_1^* if all of $P(C_1^*)$ classes are trivial (i.e. have one

element). Otherwise we process the component further with another partition. All we need to prove is that $|C_1^*| \leq \frac{7}{9}n$ when all of $P(C_1^*)$ classes are trivial.

When PARTITION is called from GENERALMATCHING in Algorithm 10 the verified matching M' has less than $\frac{1}{8}n$ elements. The ratio of the vertices from $\hat{M} = M \setminus M'$ to the vertices $\hat{G} := G \setminus V(M')$ passed to PARTITION is less than $\frac{1}{3}$: $\frac{2|M|-2|M'|}{n-2|M'|} < \frac{\frac{2}{4}n-\frac{2}{8}n}{n-\frac{2}{8}n} \leq \frac{1}{3}$. Let $\hat{V}$ be the endpoints of the edges in $\hat{M}$. No edge from $\hat{M}$ is allowed so both of its vertices are in the same equivalence class. This means that each edge of $\hat{M}$ goes in PARTITION to either S, the biggest component C_1 or some smaller component. Since on the last recursive PARTITION call, the biggest component does not have any edges from $\hat{M}$, it does not contain any vertices from $\hat{V}$.

Now we can create two different upper bounds for the number of vertices in C^*. On the first hand, we removed all of $\hat{V}$ from C^*. In each recursive PARTITION call we however added a single contracted vector to the biggest component. Let V_c be the set of all the added contracted vertices to the biggest components. We have $|C^*| \leq n-|\hat{V}|+|V_c|$. On the other hand, in PARTITION we reduce the number of vertices in the next PARTITION call by at least 2. Two of those vertices are in S and at least one of them goes to some smaller component. We also add a single contracted vertex to the largest component. This means that V_c has no less vertices than the recursive depth of the PARTITION calls. We can write a following equality: $|C^*| \leq n - 2\dot{|}V_c|$. Combining those two upper bounds we get: $2|C^*| + |C^*| \leq 2(n - |\hat{V}| + |V_c|) + n - 2\dot{|}V_c| = 3n - 2|\hat{V}|$. And thus: $|C^*| \leq n - \frac{2}{3}|\hat{V}| \leq n - \frac{2}{9}n = \frac{7}{9}n$. □

The above lemma introduces an important constant $\frac{7}{9} < 1$. For a graph with n vertices, GENERALMATCHING function can only be called on a graph with at most $\frac{7}{9}n$ vertices. We now can present the time complexity of the GENERALMATCHING algorithm with the following equation: $T(n) = T(\frac{7}{9}n) + f(n)$ where $f(n)$ is the time needed to perform the rest of the partition algorithm (excluding the GENERALMATCHING call).

Let us consider the time complexity of lines 4-7 in algorithm 10. The rest lines should be trivial. They are either recursive calls that we considered previously or simple transformations running in $\tilde{O}(n)$. Line 4. might run in $\tilde{\Theta}(n)$ but as explained previously with bipartite matching calls, each vertex might be in S only once. Thus the cumulative run-time of all calls of the line 4. is $\tilde{O}(n^2)$. To efficiently compute lines 5-7, we will use a dynamic connectivity data structure [11] providing these four functions each running in $\tilde{O}(n)$:

- INSERT(u, v) - inserts edge (u, v) into G;
- DELETE(u, v) - removes edge (u, v) from G;
- SIZE(u, v) - returns the size of the component of u
- CONNECTED(u, v) - returns whether u and v are in the same component

Algorithm 11 Implementation details for lines 5-7 of Algorithm 10

1: $T, D \leftarrow \emptyset, \emptyset$
2: **for all** edges (u, v) between S and $G \setminus S$ **do**
3: DELETE(u, v)
4: $T \leftarrow T \cup \{u, v\}$
5: **end for**
6: $u \leftarrow$ argmax $\{$SIZE(v)$: v \in T\}$
7: **for all** $v \in T$ **do**
8: **if not** CONNECTED(u, v) **then**
9: $D \leftarrow D \cup \{v\}$
10: **end if**
11: **end for**
12: Get $C_2, ...C_k$ running DFS from vertices in D
13: Create dummy vertex s
14: **for all** $v \in T \setminus D$ **do**
15: INSERT(v, s)
16: **end for**

Lines 2-5 and 12. run in the time proportional only to the number of edges between S and $G \setminus S$ and the number of edges of the small components. These sets of edges are disjoint between the recursive calls. The total runtime of these instructions is $\tilde{O}(n^2)$. The remaining lines operations can be trivially computed in $\tilde{O}(n)$ time. Since there are $O(n)$ partition calls, the whole PARTITION algorithm (excluding the bipartite part) runs in $\tilde{O}(n^2)$. And as a consequence GENERALMATCHING runs in $O(n^\omega)$.

Chapter 3

Maximum Flow

3.1 Introduction

In this section we will focus on a crucial optimization problem in graph theory – the maximum flow problem. he first known algorithm for finding a maximum flow in a graph is the Ford-Fulkerson method [8] running in $O((n+m)\cdot U)$ time. The algorithm was later improved by Edmonds and Karp [6] to $O(nm^2)$ making it independent from the maximum flow size U. In the following years, a number of algorithms were developed, each running in $O(n^3)$ time for a dense graph.

The work of Mądry has revitalized the research on the maximum flow problem. In this section we present a simplified version of his algorithm running in $\tilde{O}(n^{\frac{3}{2}}\log U)$ time. Since then a number of algorithms were developed based on his approach. The most recent one, published by Bernstein, Blikstad, Saranurak and Tu [2] runs in $O(n^{2+o(1)}\log U)$.

Algorithm	Time complexity
Ford, Fulkerson 1955 [8]	$O(nm\cdot U)$
Edmonds, Karp 1970 [6]	$O(nm^2)$
Push-relabel 1988 [10]	$O(n^3)$
Mądry 2016 [18]	$\tilde{O}(m^{\frac{3}{2}}\log U)$
Mądry 2016 [18]	$\tilde{O}(m^{\frac{10}{7}}U^{\frac{1}{7}})$
Bernstein, Blikstad, Saranurak, Tu 2024 [2]	$O(n^{2+o(1)}\log U)$

Table 3.1: An overview of maximum flow algorithms

3.2 Preliminaries

3.2.1 Electrical flow

We introduce a similar framework to the one defined along with the maximum flow problem.

Definition 25 ([18]). *Given a directed graph $G = (V, E)$, a potential vector $\sigma : V \to \mathbb{R}$ and a mapping from edges to non-negative values called resistances. A σ-flow $f : E \to \mathbb{R}$ is called an **electrical flow** when it minimizes the value of the following equation:*

$$\varepsilon_r(f) := \sum_{e \in E} r_e f_e^2 \tag{3.1}$$

Theorem 3.2.1 ([18]). *Flow f is an **electrical σ-flow** determined by the resistances r if and only if there exists vertex potentials $\phi \in \mathbb{R}^n$ such that*

$$f_{u,w} = \frac{\phi_u - \phi_w}{r_{u,w}} \quad \textit{for each } (u, w) \in E$$

Now we can rewrite the equation equation (3.1) as:

$$\varepsilon_r(f) := \sum_{e=(u,w)} \frac{(\phi_u - \phi_w)^2}{r_{u,w}} \tag{3.2}$$

This allows us to construct a dual characterization of the optimization problem.

The name *electrical flow* is not an accident. We can think of the graph G as an electrical network. Minimizing the equations corresponds to finding current (flow) that minimizes the energy ($\varepsilon_r(f)$). Furthermore Equation (3.2) directly corresponds to the Ohm's law.

It turns out minimizing the energy equation is relatively easy. Theorem 3.2.1 implies that electrical flow can be determined simply by calculating the potentials. Given a graph G and a vector of resistances r, we only need to find some potentials $\phi : V \to \mathbb{R}$ such that the demand σ is satisfied. Combining the Ohm's law with the system of Equation (1.1) reduces the task to solving a system of linear equations:

$$L_r \cdot \phi = \sigma,$$

where L_r is a Laplacian (see Definition 19) constructed from graph G with weights $w_e := r_e^{-1}$.

3.2.2 Laplacian solvers

The equation $L_r \cdot \phi = \sigma$ can be simply solved by calculating L_r^{-1} and calculating the following

$$\phi = L_r^{-1} \cdot \sigma$$

Note that L_r irreversible, since $\det L_r = 0$ for all laplacians. The last row can be determined from the previous rows, because the sum of values in each row

is equal to 0. Instead we can use e.g. Moore-Penrose inverse (Definition 21 and Lemma 1.4.1) to avoid this problem.

Event though this method computes electrical flow correctly, it does not meet our expectations. The time complexity needed for solving this equation is proportional to computing the inverse, that is $O(n^\omega)$. Instead we use a dedicated tool for solving such equations – a Laplacian solver.

Theorem 3.2.2 (Theorem 2.3 in [18]). *For any $\epsilon > 0$, any graph G with n vertices and m edges, any demand vector σ, and any resistances r , one can compute in $\tilde{O}(m \log \epsilon^{-1})$ time vertex potentials $\tilde{\phi}$ such that $||\tilde{\phi} - \phi^*||_L \leq \epsilon ||\phi^*||_L$, where L is the Laplacian of G, ϕ^* are potentials inducing the electrical σ-flow determined by resistances r , and $||\phi||_L := \sqrt{\phi^T L \phi}$.*

3.2.3 Primal-dual coupling

For a fixed flow value $F > 0$ let us define $\chi_\alpha := \alpha F \chi_{s,t}$, which represents an $\alpha \in [0, 1]$ fraction of the desired $F\chi_{s,t}$ demand. We also define the target demand vector $\chi := F\chi_{s,t}$. It is useful to view χ_α as progress made with routing the χ-flow.

Let F^* be the maximum flow that can be routed through the graph. For a fixed value F we would like to either route the full χ-flow or determine that such flow cannot be routed and consequently $F > F^*$. For this reason we develop a dual solution $y \in \mathbb{R}^n$. From now one we will also call the χ_α-flow f a primal solution.

Definition 26 ([18]). *For each edge $e = (u, v)$, a vector $y \in \mathbb{R}^n$ and vectors $\hat{u}_e^+(f), \hat{u}_e^-(f) \in \mathbb{R}^m$ with respect to flow f, we define*

$$\Delta_e(y) := y_v - y_u, \tag{3.3}$$

and

$$\Phi_e(f) := \frac{1}{\hat{u}_e^+(f)} - \frac{1}{\hat{u}_e^-(f)}. \tag{3.4}$$

We also establish the following coupling between primal solution f and dual solution y:

$$\Delta_e(y) = \Phi_e(f) \quad \text{for each } e \in E.$$

From a practical standpoint it will be useful to introduce a slightly weaker condition:

Definition 27 ([18]). *A primal solution f and a dual solution y are **γ-coupled** if and only if:*

$$|\Delta_e(y) - \Phi_e(f)| \leq \frac{\gamma_e}{\hat{u}_e(f)} \quad \text{for each } e \in E, \tag{3.5}$$

where $\hat{u}_e(f) := \min\{\hat{u}_e^+(f), \hat{u}_e^-(f)\}$ and for a vector $\gamma \in \mathbb{R}^{\kappa}$, called a violation vector.

Definition 28 ([18]). *A primal dual solution (f, y) is **well-coupled** when the l_2 norm of the violation vector is sufficiently small. From now on, we define a solution to be well-coupled if it holds that $||\gamma||_2 \leq 0.01$.*

The constant 0.01 is arbitrary and can be replace with any sufficiently small value.

Now we proceed with a key lemma using dual solution y to determine if the current flow can be further improved to reach the χ demand:

Lemma 3.2.3 ([18]). *Let (f, y) be a well-coupled primal dual solution with a χ_α-flow f, for some $0 \leq \alpha < 1$. If $\chi^T y > \frac{2m}{(1-\alpha)}$ then the demand χ cannot be routed in G, i.e., $F > F^*$.*

Proof. By contradiction, assume that $\chi^T y > \frac{2m}{(1-\alpha)}$ and the demand χ can be routed in G. Let f' be some flow $\chi_{(1-\alpha)}$-flow that is feasible in the residual graph G_f. We will try to get a contradiction by finding a lower and upper-bound of the value $(f')^T \Delta(y)$. First, we will show that $(f')^T \Delta(y) > 2m$.

$$\begin{aligned}(f')^T \Delta(y) = \sum_e f'_e \Delta_e(y) &\overset{(3.3)}{=} \sum_{e=(u,v)\in E} f'_e(y_v - y_u) \\ &= \sum_{v\in V} \left(\sum_{e\in E^+(v)} f'_e - \sum_{e\in E^-(v)} f'_e \right) \\ &= (1-\alpha)F(y_t - y_s) = (1-\alpha)\chi^T y > 2m\end{aligned} \tag{3.6}$$

To complete the proof we now need to show $(f')^T \Delta(y) \leq 2m$. Let us fix some edge e. We know that

$$f'_e \Delta_e(y) \overset{(3.5)}{\leq} f'_e \Phi_e(f) + \frac{\gamma_e f'_e}{\hat{u}_e(f)} \overset{(3.4)}{=} f'_e \left(\frac{1}{\hat{u}_e^+(f)} - \frac{1}{\hat{u}_e^-(f)} \right) + \frac{\gamma_e f'_e}{\hat{u}_e(f)}.$$

From the feasibility of f' we get $f'_e \leq \hat{u}_e^+(f)$. There are two cases to consider: either $\hat{u}_e(f) = \hat{u}_e^+(f)$ or $\hat{u}_e(f) = \hat{u}_e^-(f)$:

1. If $\hat{u}_e^+(f) = \hat{u}_e(f) \leq \hat{u}_e^-(f)$, then

$$f'_e \Delta_e(y) \leq f'_e \left(\frac{1}{\hat{u}_e^+(f)} - \frac{1}{\hat{u}_e^-(f)} \right) + \frac{\gamma_e f'_e}{\hat{u}_e(f)} \leq \frac{f'_e(1+\gamma_e)}{\hat{u}_e(f)} \leq (1+\gamma_e)$$

2. $\hat{u}_e^-(f) = \hat{u}_e(f) \leq \hat{u}_e^+(f)$. Since (f, y) are well-coupled we get $\gamma_e \leq ||\gamma||_\infty \leq ||\gamma||_w \leq \frac{1}{2}$.

$$\begin{aligned}f'_e \Delta_e(y) &\leq f'_e \left(\frac{1}{\hat{u}_e^+(f)} - \frac{1}{\hat{u}_e^-(f)} \right) + \frac{\gamma_e f'_e}{\hat{u}_e(f)} \\ &\leq f'_e \left(\frac{1}{\hat{u}_e^+(f)} - \frac{1-\gamma_e}{\hat{u}_e^-(f)} \right) \leq 1 - \frac{f'_e(1-\gamma_e)}{\hat{u}_e(f)} \leq 1.\end{aligned}$$

In conclusion

$$(f')^T \Delta(y) = \sum_{e \in E} f'_e \Delta_e(y) \leq \sum_{e \in E} (1 + \gamma_e) \leq m + \sqrt{m}||\gamma||_2 \leq 2m.$$

Thus we get a contradiction. □

3.3 Algorithm

We present an algorithm solving the maximum flow problem in $\tilde{O}(m^{\frac{3}{2}} \log(U^2 n))$ time. To achieve this result we fix some flow value F and use binary search to determine the maximum flow F^*. The difficult part is finding such feasible $F\chi_{s,t}$-flow for a fixed value F. This is done by RouteFlow function. For a given value F it either routes a $F\chi_{s,t}$-flow and returns **true**. Otherwise it states that $F\chi_{s,t}$ cannot be routed and returns **false**.

Algorithm 12 Maximum flow algorithm

```
1:  function MaximumFlow(G)
2:      R ← 1
3:      while RouteFlow(G, R) do
4:          R ← 2R
5:      end while
6:      L ← R/2
7:      while L < R do
8:          F ← ⌊(L + R)/2⌋
9:          if RouteFlow(G, F) then
10:             L ← F
11:         else
12:             R ← F
13:         end if
14:     end while
15:     RouteFlow(G, L)                    ▷ Route the maximum flow
16:     RoundFlow(G)
17:     return (F, G_f)
18: end function
```

3.3.1 Initialization

First we have to initiate the primal dual solutions so that the coupling constraint is satisfied. For undirected graphs, we have $u_{(v,w)} = u_{(w,v)}$ for each edge $e = (v, w)$. It means that for a zero flow f following equality holds: $\hat{u}_e^+(f) = u_e^+ = u_e^- = \hat{u}_e^-(f)$. Furthermore for $y = 0$, the pair (f, y) is *well-coupled* since $\delta_e(y) = 0 = \Phi_e(f)$ for each edge e.

This however becomes a problem for a directed graph. We can no longer use such straightforward initialization. Fortunately it is possible to reduce the problem for directed graph to one for undirected graphs in a satisfactory time.

Lemma 3.3.1 ([18]). *Let G be an instance of the maximum s-t problem with m arcs and the maximum capacity U, and let F be the corresponding target flow value. In $\tilde{O}(m)$ time, one can construct an instance G' of undirected maximum s-t flow problem that has $O(m)$ arcs and the maximum capacity U, as well as target flow value F' such that:*

(a) if there exists a feasible s-t flow of value F in G then a feasible s-t flow of value F' exists in G';

(b) given a feasible s-t flow of value F' in G' one can construct in $\tilde{O}(m)$ time a feasible s-t flow of value F in G.

As a consequence to this lemma, we will focus on solving the maximum flow problem on the undirected instances only.

3.3.2 Progress steps

The steps presented in this section will be the key part of the algorithm. Given a well coupled pair (f, y) we will try to improve the primal solution. In the augmentation step we will use the electrical network framework to compute another well-coupled pair (f^+, y^+). As it will become evident the augmentation step will violate the well-coupling. To counteract the uncoupling of the primal-dual solution, we introduce a fixing step that deals with this problem.

Algorithm 13 RouteFlow algorithm

```
1: function RouteFlow(G, F)
2:     demand ← F χ_{s,t}
3:     f, y ← {0}^m, {0}^n
4:     α ← 0
5:     while α < 1 do
6:         f^+, y^+ ← AugmentationStep(f, y)
7:         f̂, ŷ ← FixingStep(f^+, y^+)
8:         if χ^T y > 2m/(1−α) then
9:             return false
10:        end if
11:        f, y ← f̂, ŷ
12:    end while
13:    return true
14: end function
```

3.3.3 Augmentation step

First we construct an electrical χ-flow $\tilde{f}$ in $G = (V, E)$. For each edge $e \in E$ we defined resistance r_e as

$$r_e := \frac{1}{(\hat{u}_e^+(f))^2} + \frac{1}{(\hat{u}_e^-(f))^2}. \tag{3.7}$$

Algorithm 14 Electrical Flow

```
1: function ElectricalFlow(G, σ, r)
2:     w_{i,j} ← 1/r_{i,j} for each i, j ∈ V
3:     L ← weighted Laplacian of G with weights w
4:     Solve equation L · φ = σ for φ
5:     f_{i,j} ← (φ_i − φ_j)/r_{i,j} for each i, j ∈ V
6:     return f, φ
7: end function
```

For a given step size δ, we calculate the corresponding flow $\tilde{f}$ and potentials $\tilde{\phi}$ and update the primal dual solution:

$$\begin{aligned} \hat{f}_e &:= f_e + \delta \tilde{f}_e \quad \text{for each } e \in E, \\ \hat{y}_v &:= y_v + \delta \tilde{\phi}_v \quad \text{for each } v \in V. \end{aligned} \tag{3.8}$$

Since $\tilde{f}$ is a F flow, this update introduces a progress $\hat{\alpha} = \alpha + \delta$. Observe that r_e becomes very small when f_e value is near the capacity u_e. For a carefully chosen step size we preserve the feasibility of $\delta\tilde{f}$ in G_f and further of $\hat{f}$ in G.

Algorithm 15 Augmentation Step

```
1: function AugmentationStep(G, χ, f, y)
2:     for e ∈ E do
3:         r_e ← 1/(û_e^+(f))^2 + 1/(û_e^-(f))^2.
4:     end for
5:     f̃, φ̃ ← ElectricalFlow(G, χ, r)
6:     f ← f + δf̃
7:     y ← y + δφ̃
8:     return f, y
9: end function
```

To determine how small should the step size be, we introduce a notion of a *congestion vector*.

Definition 29 ([18])**.** *We define a* ***congestion vector*** $\rho \in \mathbb{R}^m$ *to be equal*

$$\rho_e := \frac{\tilde{f}_e}{\hat{u}_e(f)} \quad \textit{for each } e \in E. \tag{3.9}$$

It is useful to think of ρ as a measure of how much the flow f overflows the capacities of G. Now we can ensure $\delta\tilde{f}$ is feasible in G_f by enforcing $\delta|\rho_e| \leq \frac{1}{4}$ for each edge $e \in E$. Equivalently, in terms of L_∞ norm we require

$$\delta \leq \frac{1}{4||\rho||_\infty}. \tag{3.10}$$

The congestion vector can be also used to closely bound the energy of the flow, via the following lemma:

Lemma 3.3.2 ([18]). *For any edge $e \in E$, it holds that $\rho_e^2 \leq r_e\tilde{f}_e^2 \leq 2\rho_e^2$ and $||\rho||_2^2 \leq \varepsilon_r(\tilde{f}) \leq 2||\rho||_2^2$*

Proof. For a fixed edge e we have

$$\rho_e^2 \overset{(3.9)}{=} \frac{\tilde{f}_e^2}{\hat{u}_e(f)^2} \leq \left(\frac{1}{(\hat{u}_e^+(f))^2} + \frac{1}{(\hat{u}_e^-(f))^2}\right) = r_e\tilde{f}_e^2$$

and

$$r_e\tilde{f}_e^2 \leq \frac{2}{\hat{u}_e(f)^2}\tilde{f}_e^2 = 2\rho_e^2$$

thus

$$||\rho||_2^2 = \sum_e \rho_e^2 \leq \sum_e r_e\tilde{f}_e^2 \overset{(3.1)}{=} \varepsilon_r(\tilde{f}) \leq \sum_e 2\rho_e^2 = 2||\rho||_2^2.$$

□

To safely perform another augmentation step and consequently make progress in routing χ-flow we have to maintain several constraints:

1. The feasibility of $\hat{f}$ – we already covered that with a sufficiently small step size (3.10).

2. The well-coupling of $(\hat{f}, \hat{y})$. The augmentation step breaks the well-coupling but only to a small degree. We will develop a fixing step to counteract this divergence.

First we prove a simple lemma

Lemma 3.3.3 ([18]). *For any $u_1, u_2 > 0$, with $u = \min\{u_1, u_2\}$ and x such that $|x| \leq \frac{u}{4}$, we have that*

$$\left(\frac{1}{u_1 - x} - \frac{1}{u_2 + x}\right) = \frac{1}{u_1} - \frac{1}{u_2} + \left(\frac{1}{u_1^2} + \frac{1}{u_2^2}\right)x + x^2\varsigma$$

where $|\varsigma| \leq \frac{5}{u^3}$

Proof. We define a function $g(x) := \frac{1}{u_1 - x} - \frac{1}{u_2 + x}$ By Taylor's theorem, for some $|z| \leq |x| \leq \frac{u}{4}$ it holds that:

$$\begin{aligned} g(x) &= g(0) + g'(0)x + g''(z)\frac{x^2}{2} \\ &= \frac{1}{u_1} + \frac{1}{u_2} + \left(\frac{1}{u_1^2} + \frac{1}{u_2^2}\right)x + x^2\left(\frac{1}{(u_1 - z)^3} - \frac{1}{(u_2 + z)^3}\right) \end{aligned}$$

and thus

$$\varsigma := \left|\frac{1}{(u_1 - z)^3} - \frac{1}{(u_2 + z)^3}\right| \leq \left(\frac{4}{3u}\right)^3 + \left(\frac{4}{3u}\right)^3 \leq \frac{5}{u^3}.$$

□

Now apply this lemma to compare $\Phi_e(f)$ and $\Phi_e(\hat{f})$:

$$\begin{aligned} \Phi_e(\hat{f}) &= \frac{1}{\hat{u}_e^+(f) - \delta\tilde{f}_e} - \frac{1}{\hat{u}_e^-(f) + \delta\tilde{f}_e} \\ &= \frac{1}{\hat{u}_e^+(f)} - \frac{1}{\hat{u}_e^-(f)} + \left(\frac{1}{(\hat{u}_e^+(f))^2} + \frac{1}{(\hat{u}_e^-(f))^2}\right)\delta\tilde{f}_e + (\delta\tilde{f}_e)^2\varsigma \end{aligned}$$

And we get

$$\Phi(\hat{f}) - \Phi(f) = r_e\delta\tilde{f}_e + (\delta\tilde{f}_e)^2\varsigma_e$$

$$\Phi_e(\hat{f}) - \Phi_e(f) = r_e\delta\tilde{f}_e = \delta(\tilde{\phi}_v - \tilde{\phi}_u) = \Delta_e(\hat{y}) - \Delta_e(\hat{y}) + O(\delta^2)$$

to the first order of approximation.

The primal dual solution was left almost intact. In the following lemma we show a precise upper bound for the violation between the new pair $(\hat{f}, \hat{y})$.

Lemma 3.3.4 ([18])**.** *Let $0 < \delta \leq (4||\rho||_\infty)^{-1}$ and the primal dual solution (f, y) be γ-coupled. Then, we have that, for any arc e,*

$$\left|\Delta_e(\hat{y}) - \Phi_e(\hat{f})\right| \leq \frac{\frac{4}{3}\gamma_e + 7(\delta\rho_e)^2}{\hat{u}_e(\hat{f})}$$

Proof.

$$\begin{aligned} \left|\Delta_e(\hat{y}) - \Phi_e(\hat{f})\right| &= \left|\Delta_e(y) + \delta(\tilde{\phi}_v - \tilde{\phi}_u) - \Phi(f) - \left(\frac{1}{(\hat{u}_e^+)^2} + \frac{1}{(\hat{u}_e^-)^2}\right)\delta\tilde{f}_e - (\delta\tilde{f}_e)^2\varsigma_e\right| \\ &\overset{(3.5)}{\leq} \left|\delta\left((\tilde{\phi}_v - \tilde{\phi}_u) - r_e\tilde{f}_e\right) - (\delta\tilde{f}_e)^2\varsigma_e\right| + \frac{\gamma_e}{\hat{u}_e(f)} \\ &\overset{(3.2)}{=} \left|(\delta\tilde{f}_e)^2\varsigma_e\right| + \frac{\gamma_e}{\hat{u}_e(f)} \leq \frac{5(\delta\tilde{f}_e)^2}{(\hat{u}_e(f))^3} + \frac{\gamma_e}{\hat{u}_e(f)} \\ &\overset{(3.11)}{\leq} \frac{4}{3\hat{u}_e(\hat{f})}\left(\frac{5(\delta\tilde{f})^2}{(\hat{u}_e(f))^2} + \gamma_e\right) \leq \frac{7(\delta\rho_e)^2 + \frac{4}{3}\gamma_e}{\hat{u}_e(\hat{f})} \end{aligned}$$

Where the second last inequality comes from

$$\hat{u}_e(\hat{f}) \overset{(3.8)}{\geq} \hat{u}_e(f) - \delta\tilde{f}_e \overset{(3.9)}{=} \hat{u}_e(f) - \delta\hat{u}_e(f)\rho_e = \hat{u}_e(f)(1-\delta\rho_e) \overset{(3.10)}{=} \frac{3}{4}\hat{u}_e(f) \tag{3.11}$$

□

3.3.4 Fixing step

Above lemma gives us an upper bound on how much the primal dual solution is violated after the augmentation step. After multiple such steps this effect might accumulate and "decouple" the primal dual solution. To avoid this problem we introduce the *fixing step*. This step allows us to reduce the violation assuming it was sufficiently small to begin with. This is done by computing the electrical flow once again.

Algorithm 16 Fixing Step

```
function FixingStep(G, χ, f, y)
    for e ∈ E do
        θ_e ← ( 1/((û_e^+(g))^2) + 1/((û_e^+(g))^2) )^{-1} (Δ_e(z) − Φ_e(g))
    end for
    f ← f − θ
    for (v, w) ∈ V × V do
        χ'_v ← θ_(v,w)
    end for
    for e ∈ E do
        r_e ← 1/((û_e^+(g))^2) + 1/((û_e^+(g))^2)
    end for
    (f̃, φ̃) ← ElectricalFlow(G, χ', r)
    f ← f + δf̃
    y ← y + δφ̃
    return f, y
end function
```

For the sake of formality we introduce the following lemma:

Lemma 3.3.5 ([18])**.** *Let (g, z) be a ς-coupled primal dual solution, with g being a feasible $\chi_{\alpha'}$-flow and $||\varsigma||_2 \leq \frac{1}{50}$. In $\tilde{O}(m)$ time, we can compute a primal dual solution $(\overline{g}, \overline{z})$ that is well-coupled and in which $\overline{g}$ is still a $\chi_{a'}$ flow.*

Proof. First let us define a correction vector $\theta \in \mathbb{R}^m$. For each edge e we have

$$\theta_e := \left(\frac{1}{(\hat{u}_e^+(g))^2} + \frac{1}{(\hat{u}_e^+(g))^2}\right)^{-1} (\Delta_e(z) - \Phi_e(g)) \tag{3.12}$$

Since (g, z) is $\varsigma-$coupled we know that

$$|\theta_e| = \left(\frac{1}{(\hat{u}_e^+(g))^2} + \frac{1}{(\hat{u}_e^+(g))^2}\right)^{-1} |\Delta_e(z) - \Phi_e(g)| \overset{(3.5)}{\leq} \hat{u}_e(g)^2 \left(\frac{\varsigma_e}{\hat{u}_e(g)}\right) = \varsigma_e \hat{u}_e(g)$$

Now if we apply the correction to the primal solution

$$g'_e := g_e + \theta_e$$

then we can put forth the following upper bound

$$\begin{aligned}
|\Delta_e(z) - \Phi_e(g')| &= \left|\Delta_e(z) - \left(\frac{1}{\hat{u}_e^+(g) - \theta_e} - \frac{1}{\hat{u}_e^-(g) + \theta_e}\right)\right| \\
&= \left|\Delta_e(z) - \Phi_e(g) - \left(\frac{1}{(\hat{u}_e^+(g))^2} - \frac{1}{(\hat{u}_e^-(g))^2}\right)\theta_e - \theta_e^2 \varsigma'_e\right| \\
&= |-\theta_e^2 \varsigma'_e| \overset{(3.3.3)}{\leq} \frac{5\hat{u}_e(g)^2 \varsigma_e^2}{\hat{u}_e(g)^3} = \frac{5\varsigma_e^2}{\hat{u}_e(g)} \overset{(3.13)}{\leq} \frac{51\varsigma_e^2}{10\hat{u}_e(g')}
\end{aligned}$$

The second inequality comes from the fact that $\hat{u}_e(g')$ is closely bounded by $\hat{u}_e(g)$:

$$\frac{49}{50}\hat{u}_e(g) \leq \hat{u}_e(g') \leq \frac{51}{50}\hat{u}_e(g) \tag{3.13}$$

Furthermore we can show that primal dual solution (g', z) is $\tilde{\gamma}-$coupled. From $||\varsigma||_\infty \leq ||\varsigma||_2 \leq \frac{1}{50}$, the definition of γ-coupling and the above lemma we get

$$\begin{aligned}
||\tilde{\gamma}||_2 = \sqrt{\sum_e \tilde{\gamma_e}^2} &= \sqrt{\sum_e ((\Delta_e(z) - \Phi_e(g')) \cdot \hat{u}_e(g'))^2} \\
&\overset{3.3.4}{\leq} \sqrt{\sum_e \left(\frac{51\varsigma_e^2}{10\hat{u}_e(g')} \cdot \hat{u}_e(g')\right)^2} \\
&\overset{(3.13)}{\leq} \frac{51}{10}\sqrt{\sum_e \varsigma_e^4} \leq \frac{51}{10}||\varsigma||_2^2 \leq \frac{51}{25000} < \frac{1}{100}
\end{aligned} \tag{3.14}$$

The above fixing procedure gave us a primal dual solution (g', z). The correction however made some progress in g' which is now a $(\chi'_\alpha + \hat{\sigma})$-flow for some demand $\hat{\sigma}$. To nullify this progress we can compute an electrical $(-\hat{\sigma})$-flow denoted $\hat{\theta}$. Just like in the augmentation step, we set resistances to be equal

$$\hat{r}_e := \frac{1}{(\hat{u}_e^+(g'))^2} + \frac{1}{(\hat{u}_e^-(g'))^2}$$

and evaluate the corresponding potentials $\hat{\phi}$. Like before, we set

$$\begin{aligned}
\overline{g} &:= g' + \hat{\theta} \\
\overline{z} &:= z + \hat{\phi}
\end{aligned} \tag{3.15}$$

Now, $\overline{g}$ is once again a χ'_α-flow. All we have to do is confirm $(\overline{g}, \overline{z})$ remains well coupled. We consider the congestion vector

$$\hat{\rho} := \frac{\hat{\theta}_e}{\hat{u}_e(g')}$$

$$\begin{aligned} \varepsilon_{\hat{r}}(-\theta) = \sum_e \hat{r}_e(-\theta_e)^2 = \sum_e \left(\frac{1}{(\hat{u}^+(g')^2} + (\frac{1}{(\hat{u}^-(g')^2} \right) \theta_e^2 \\ \leq \sum \frac{2}{\hat{u}_e(g')^2} (\varsigma \hat{u}_e(g))^2 \overset{(3.13)}{\leq} \sum_e 2 \left(\frac{51\varsigma_e}{50} \right)^2 \leq \frac{21}{20} ||\varsigma||_2^2 \leq \frac{1}{2000}. \end{aligned} \tag{3.16}$$

Since $-\theta$ is a $\hat{\sigma}-$flow and $\hat{\theta}$ are the energy-minimizing potentials and thus $\varepsilon_{\hat{r}}(\hat{\theta}) \leq \varepsilon_{\hat{r}}(-\theta)$. Applying Lemma 3.3.2 we get

$$||\hat{\rho}||_2^2 \overset{3.3.2}{\leq} \varepsilon_{\hat{r}}(\hat{\theta}) \leq \varepsilon_{\hat{r}}(-\theta) \leq \frac{1}{2000}$$

Using Lemma 3.3.4 we get that $(\overline{g}, \overline{z})$ is $\overline{\gamma}-$coupled with

$$||\overline{\gamma}||_2 = \sqrt{\sum_e \overline{\gamma}_e^2} \overset{3.3.4}{\leq} \sqrt{\sum_e \left(\frac{4}{3}\tilde{\gamma}_e + 7\hat{\rho}_e^2 \right)^2} \leq \frac{4}{3}||\hat{\gamma_e}||_2 + 7||\hat{\rho}||_2^2 \leq \frac{204}{75000} + \frac{7}{2000} < \frac{1}{100}$$

This confirms that $(\overline{g}, \overline{z})$ is well-coupled. □

In order to use the fixing procedure, the primal-dual solution has to be $\frac{1}{50}$-coupled. To ensure this condition we have to introduce a more strict upper bound to the step size δ.

Lemma 3.3.6 ([18])**.** *(f^+, y^+) is a well-coupled primal dual solution with f^+ being a $\chi_{\alpha'}$-flow that is feasible in G whenever $\delta \leq (33||\rho||_4)^{-1}$.*

First we have $||\rho||_4 \geq ||\rho||_\infty$ so condition $\delta \leq \frac{1}{4||\rho||_\infty}$ is already satisfied and f^+ is $\chi_{\alpha'}-$feasible in G. Now we need to prove the well-coupling of (f^+, y^+).

To use above lemma and complete the proof we need to show that $(\hat{f}, \hat{y})$ computed in the augmentation step is $\hat{\gamma}$-coupled with $||\hat{\gamma}||_2 \leq \frac{1}{50}$.

We know that (f, y) was γ-coupled with $\gamma \leq \frac{1}{100}$. Once again, by Lemma 3.3.4 we have

$$\begin{aligned} ||\hat{\gamma}||_2 = \sqrt{\sum_e \hat{\gamma}_e^2} \overset{3.3.4}{\leq} \sqrt{\sum_e \left(\frac{4}{3}\gamma_e + 7(\delta\rho_e^2)^2 \right)^2} \\ \leq \frac{4}{3}||\gamma_e||_2 + 7\delta^2||\rho||_4^2 \leq \frac{4}{200} + \frac{7}{33^2} < \frac{1}{50} \end{aligned} \tag{3.17}$$

And thus Lemma 3.3.5 can be applied.

3.3.5 Flow rounding

Up to this point, we allowed a fractional flow passing through the edges. Most instances of maximum flow problem only consider an integral flow for which $f_e \in \mathbb{Z}$ for each edge $e \in E$. Fortunately, we can transform a fractional flow to an integral one in $O(m \log n)$ time with a flow rounding algorithm.

In order to achieve that, let us introduce an auxiliary notion of fractional cycles:

Definition 30 ([14]). *Given a flow circulation f, we call a cycle consisting of only non-integral flow to be a fractional cycle.*

The key part of flow rounding algorithm is removing all fractional cycles. The following lemma shows that once it is done, the remaining flow will be integral.

Lemma 3.3.7 ([14]). *If a circulation f contains no cycle of edges with fractional flow, then f is integral.*

Algorithm 17 RoundFlow algorithm

```
1:  function RoundFlow(G, flow)
2:      fractionalFlow ← the significant of x for each x ∈ flow
3:      dt ← DynamicTree(weights: fractionalFlow)
4:      for {u, v} ∈ E do
5:          if u and v are in the same component then          ▷ FindRoot
6:              Let S ∈ {[u, v, ..., u], [v, u, ..., v]} be a circular flow
7:              with nonnegative value                          ▷ PathSum
8:              Find minimum flow f in cycle S of the same
9:              direction as S                                  ▷ PathMin
10:             Push f through S                                ▷ PathAdd
11:         end if
12:         Add {u, v} edge if it has a fractional flow         ▷ Link
13:     end for
14: end function
```

To achieve a $O(m \log n)$ algorithm, we use a dynamic trees data structure introduced in [26]. It provides the following functions, each running in $\tilde{O}(n)$ time:

- Cut(u, v) - remove the (u, v) edge,
- FindRoot(v) - find the root of v tree; used for determining if two vertices belong to the same component,
- PathAdd(u, v, c) - add value c to all edges along the u-v undirected path
- PathMin(u, v) - find the minimum value along the u-v undirected path; consider only edges directed same as the path,
- PathSum(u, v) - find the sum of the values along the u-v undirected path

3.4 Time complexity

In this section we will determine and prove the running time complexity of the algorithm. To recall, we have constructed a primal dual solution (f, y) in which f is a χ_α flow. The dual solution y keeps track if the flow f can be further improved to route the entire target flow χ. We also introduced a coupling between those two vectors that ensures those values are linked.

The algorithm consists of multiple progress steps, each trying to improve the throughput of primal solution. The amount of progress made in a single step was denoted as δ. We have also constructed an upper-bound of the step size.

To upper-bound the time complexity of the algorithm we could find a lower-bound on the progress made in each step.

Lemma 3.4.1. *The progress made in each step of the* RouteFlow *algorithm can be lowerbounded by* $(1-\alpha)\hat{\delta}$ *for some constant value* $\hat{\delta}$*.*

We will prove this fact later in this section. For now, let's see how we can use this lemma to determine the time complexity of ElectricalFlow algorithm.

As a consequence of lemma 3.4.1, we get:

Lemma 3.4.2. *The* RouteFlow *takes at most* $O(\hat{\delta}^{-1} \log mU)$ *iterations.*

Proof. Let us consider how much progress has been done by first k steps. In the first iteration we make a progress of at least $\hat{\delta}$. In each next step we progress by $\hat{\delta}$ fraction of the remaining progress $1-\alpha$. After k step we thus made a progress of $1-(1-\hat{\delta})^k$. Let t be the the number of iterations needed before at most one unit of flow can still be routed. Formally t is the smallest number satisfying $F \cdot (1-\hat{\delta})^t \leq 1$. We compute a $\log_{1-\hat{\delta}}$ of both sides of the equation: $t \geq \log_{1-\hat{\delta}}(\frac{1}{F}) = \frac{\ln(1/F)}{\ln(1-\hat{\delta})}$. Since $\ln(1-x) \leq -x$ for every $x \geq 0$ we have $t \geq \frac{\ln(1/F)}{\ln(1-\hat{\delta})} \geq \frac{\ln(F)}{\hat{\delta}}$. Using the upper-bound for F: $F \leq mU$ we finally get $t \in O(\hat{\delta}^{-1} \ln(mU))$. □

Now we are ready to determine the time complexity behind ElectricalFlow.

Theorem 3.4.3. *The* ElectricalFlow *algorithm runs in* $\tilde{O}(\hat{\delta}^{-1} m \log U)$ *time.*

Proof. RouetFlow executes $O(\hat{\delta}^{-1} \log mU)$ iterations, each running in $\tilde{O}(m)$ time. We thus get the $\tilde{O}(\hat{\delta}^{-1} m \log U)$ running time for the RouetFlow algorithm. We also need to route at most one remaining unit of flow. This can be done in $O(m)$. Lastly, we execute the FlowRounding procedure that runs in $\tilde{O}(m)$ time. The time complexity of ElectricalFlow is thus $\tilde{O}(\hat{\delta}^{-1} m \log U)$. □

Showing a lower-bounding δ is however not easy. First of all we do not know if target flow can be fully routed through G. If it is not the case, the algorithm terminates through y. Even if the demand can be satisfied, we are still yet to prove the algorithm reaches the desired flow.

Note that in the algorithm $\hat{u}^+(f)$ and $\hat{u}^-(f)$ were always symmetric. We either used $\hat{u}(f) = \min\{\hat{u}^+(f), \hat{u}^-(f)\}$ or some other expression (e.g. resistance) symmetric in terms of $\hat{u}^+(f)$ and $\hat{u}^-(f)$. The flows computed in an electrical flow must be feasible in a symmetrized version of the residual graph.
Formally, we define symmetrization $\hat{G}_f$ of a residual graph G_f to be an undirected graph (constructed from G) with each edge holding a capacity equal to $\hat{u}(f)$.

Now we can see the first challenge in lower-bounding the progress made in a single step. Even if making a major progress in G_f is possible, the computed augmentation step in symmetrization $\hat{G}_f$ can be small. This limitation makes it impossible to lower-bound the progress without performing some extra work.

Fortunately we can slightly modify the graph G to not only compute its progress lower-bound, but also determine the number of steps taken in the original problem. We insert m undirected edges of capacity $2U$ between s and t. We call them *preconditioning arcs* and the modified graph a *preconditioning graph*. This modification will increase the flow by $2mU$ in total. Now as long as the primal dual solution (f, y) is well-coupled we can guarantee the augmentation step can push a flow with value lower-bounded by some constant. On one hand, the well-coupling prevents the preconditioning arcs to fill up to quickly. On the other hand, the preconditioning arcs assure that the progress made in each step is large enough.

Lemma 3.4.4 ([18])**.** *Let (f, y) be a well-coupled dual solution in the preconditioned graph G and let f be a χ_α-flow f' for some $0 \leq \alpha < 1$, that is feasible in G. It holds that:*

(a) there exists a $\chi_{\frac{(1-\alpha)}{10}}$-flow f' that is feasible in the symmetrization $\hat{G}_f$ of the residual graph G_f;

(b) or $\chi^T y > \frac{2m}{(1-\alpha)}$ implying that our target demand χ cannot be routed in G.

Proof. First, let us assume that $\chi^T y \leq \frac{2m}{1-\alpha}$. Otherwise case *(b)* is satisfied and the proof is concluded. We will argue that a $\chi_{\frac{(1-\alpha)}{10}}$ flow can be pushed through the preconditioning edges alone. Note that there is exactly $\frac{m}{2}$ such edges. All we need to show is that for each preconditioning edge, the following inequality is satisfied:

$$\hat{u}_e(f) \geq \frac{(1-\alpha)F}{5m} = \frac{2}{m} \cdot \frac{(1-\alpha)}{10} F,$$

The $u_e(f)$ is the capacity in the symmetrization $\hat{G}_f$ of the residual graph G_f, thus $u_e(f)$ units can be further pushed through the edge e.

We prove the above inequality by contradiction. Note that the preconditioning arcs are indistinguishable from the point of view of the algorithm. In consequence, the flow running through two preconditioning edges is the same at all times. Let us fix some preconditioning edge e and assume that the inequality does not hold. The arc capacity makes at least $\frac{2}{3}$ fraction of the maximum s-t flow F^* in G. We thus have:

$$u_e^+ = u_e^- = \frac{2}{m} \cdot \frac{2}{3} F^* = \frac{4F^*}{3m},$$

where u_e^+, u_e^- are the initial capacities of the edge. We know that $F \leq \frac{3}{2}F^*$. If $F > \frac{3}{2}F^*$ we would not be able to push at least $\frac{2}{3}F$ through the preconditioning arcs . We now have:

$$\begin{aligned} |f_e| &= \max\{u_e^+ - \hat{u}_e^+(f), u_e^- - \hat{u}_e^-(f)\} \\ &\geq \frac{4F^*}{3m} - \frac{(1-\alpha)F}{5m} \geq \frac{\frac{5}{2}F^* + \alpha F}{3m} \geq \frac{2F^*}{3m}. \end{aligned}$$

We know as well that $f_e > 0$. Otherwise there would be a t-s flow larger than $\frac{F^*}{3}$ over all the preconditioning arcs. It would be impossible to counteract this backward flow with the edges from G. We show further that

$$\hat{u}_e(f) \leq \frac{(1-\alpha)F}{5m} < \frac{(1-\alpha)F}{3m} \leq \frac{F^*}{2m} \leq \frac{1}{2}u_e^- = \frac{1}{2}(\hat{u}_e^-(f) - f_e) \leq \frac{\hat{u}_e^-(f)}{2}$$

or equivalently

$$\frac{1}{\hat{u}_e^-(f)} \leq \frac{1}{2\hat{u}_e(f)}.$$

With the addition of the well-coupling of (f, y) we get

$$\begin{aligned} \Delta_e(y) &\geq \Phi_e(f) - \frac{\gamma_e}{\hat{u}_e(f)} = \frac{1}{\hat{u}_e^+(f)} - \frac{1}{\hat{u}_e^-(f)} - \frac{\gamma_e}{\hat{u}_e(f)} \\ &\geq \frac{(1-\gamma_e)}{\hat{u}_e(f)} - \frac{1}{2\hat{u}_e(f)} = \frac{(1-2\gamma_e)}{2\hat{u}_e(f)}. \end{aligned}$$

We know that $\gamma_e \leq \frac{1}{\sqrt{m}}$. Otherwise the contribution of all the preconditioning arcs to the violation vector's l_2-norm would break the well-coupling of (f, y). We thus get

$$\Delta_e(y) \geq \frac{5m(1 - \frac{2}{\sqrt{m}})}{2(1-\alpha)F} > \frac{2m}{(1-\alpha)F}.$$

This however leads to a contradiction since

$$\Delta_e(y) = y_t - y_s = \frac{\chi^T y}{F} \leq \frac{2m}{(1-\alpha)F}.$$

□

With Lemma 3.4.4 in place, we can determine the progress lowerbound $\hat{\delta}$.

Lemma 3.4.5 ([18])**.** *Let (f, y) be a well-coupled dual solution that is feasible in G_f, for some $0 \leq \alpha < 1$. Let $\tilde{f}$ be an electrical χ-flow determined by the resistances r given by Equation* (3.7) *. We have that either:*

(a) $||\rho||_2^2 \leq \varepsilon_r(\tilde{f}) \leq \frac{200m}{(1-\alpha)^2}$, where ρ is the congestion vector defined in Equation (3.9)*,*

(b) or $\chi^T y > \frac{2m}{(1-\alpha)}$, i.e., our target demand χ cannot be routed in G.

Proof. Directly from Lemma 3.4.4 either $\chi_\alpha^T y > \frac{2m}{(1-\alpha)}$ and we fall into *(b)* or there exists a $\chi_{\frac{(1-\alpha)}{10}}$-flow f' feasible in $\hat{G}(f)$. Let us create an upper-bound for the energy of f'. Since f' is feasible in $\hat{G}_f$, i.e. $f'_e \leq \hat{u}_e(f)$ for each edge e, we have

$$\begin{aligned}\varepsilon(f') &= \sum_{e\in E} r_e (f'_e)^2 = \sum_{e\in E} \left(\frac{1}{(\hat{u}_e^+(f))^2} + \frac{1}{(\hat{u}_e^-(f))^2}\right)(f'_e)^2 \\ &\leq 2\sum_{e\in E}\left(\frac{f'_e}{\hat{u}_e(f)}\right)^2 \leq 2m\end{aligned} \tag{3.18}$$

Now if we define the flow $f'' := \frac{10}{(1-\alpha)} f'$ we get a χ-flow with the energy at most

$$\varepsilon_r(f'') = \sum_{e\in E} r_e (f''_e)^2 = \sum_{e\in E} r_e \left(\frac{10}{(1-\alpha)} f'_e\right)^2 = \frac{100}{(1-\alpha)^2}\varepsilon_r(f') \leq \frac{200}{(1-\alpha)^2} m.$$

However an electrical χ-flow $\tilde{f}$ induced by the resistances r minimizes the energy function. This implies the following inequality

$$||\rho||_2^2 \overset{3.3.2}{\leq} \varepsilon_r(\tilde{f}) \leq \varepsilon_r(f'') \leq \frac{200}{(1-\alpha)^2} m$$

□

Now we can construct an upper-bound on δ. From Lemma 3.3.6 we have

$$\delta \geq \frac{1}{33||\rho||_4} \geq \frac{1}{33||\rho||_2} \overset{3.3.6}{\geq} \frac{1}{33}\cdot\sqrt{\frac{(1-\alpha)^2}{200m}} = \frac{(1-\alpha)}{33\sqrt{200m}}$$

Taking $\hat{\delta} := (33\sqrt{200m})^{-1}$ we get the desired $\delta \geq (1-\alpha)\hat{\delta}$.

Finally, the time complexity of this algorithm is $\tilde{O}(m\cdot\hat{\delta}^{-1}\log U) = \tilde{O}(m^{\frac{3}{2}}\log U)$.

Chapter 4

Implementation and benchmarks

Both algorithms are implemented in `C++20` as a contribution to `Koala NetworKit` [28] open-source library. In order to implement these algorithms, we used three helper libraries: `NetworKit` [21], `Eigen` [7] and `NTL` [22]. The first one provides a graph class, as well as some graph related utility functions. Because of the algebraic nature of the presented algorithms, we also decided to include the `Eigen` and the `NTL` library.

4.1 Gaussian matching

4.1.1 Implementation details

We expanded on the pseudocode provided by Mucha and Sankowski in [20] to create a implementation of the Gaussian matching algorithm. The key part of the implementation is included in five files:

- `GeneralGaussianMatching.cpp` and `NaiveGaussianMatching.cpp`*
- `BipartiteGaussianMatching.cpp`
- `LazyGaussElimination.cpp` and `NaiveGaussElimination.cpp`*
- `NaiveDynamicComponents.cpp`*

The first two files represent the algorithm from [20]. The first one is capable of solving a bipartite case of the problem, while the second one computes a perfect matching for the general graphs. One of our contributions is the GaussEliminationAlgorithm. We developed both a naive version as well as a optimal LazyGaussElimination that runs in $O(n^\omega)$ time. The second version was only described in the original work of Mucha [20]. Based on the description we developed a pseudocode and a working `C++` code. The last file represents a

data structure that has only a naive implementation. Providing an implementation based on [11] will be likely to further improve the running time of the algorithm.

We first tried to use `Eigen` [7] for all matrix computations. It however turned out to be insufficient to handle finite field operations. Abandoning Z_p field for a floating point matrices introduced some issues to the implementation. Both the values of cells in the skew matrix and the determinant of the matrix itself can not be to small or to big. This would lead to a precision issues when working with a floating point data types. To avoid this issue we decided to use a number theory focused library - `NTL` [22].

4.1.2 Benchmark

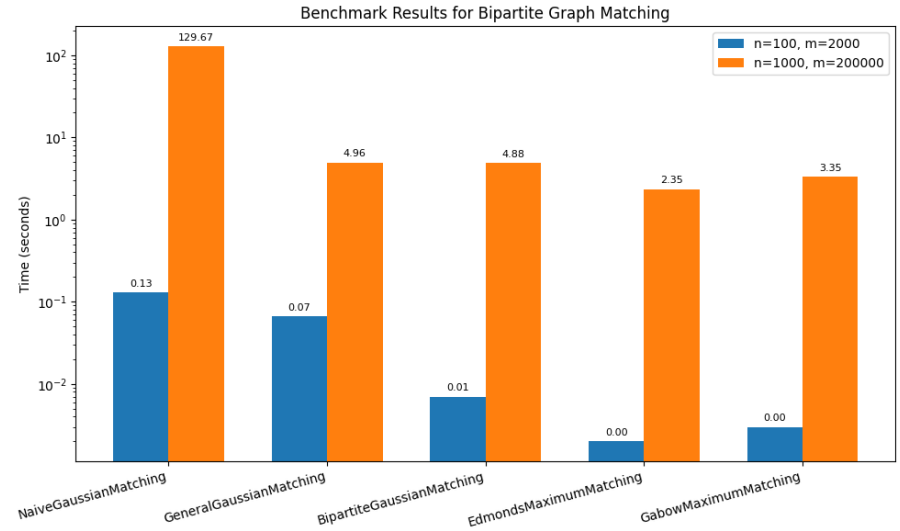
We present a benchmark comparing the running time of the implemented algorithms. For the additional context, we compare the result to some of the maximum matching algorithms already implemented in `Koala/NetworKit`. The algorithms were tested on two separate categories. The first chart regard a bipartite graphs, while the second one represents a general case. Here we present an average runtime of the algorithms, averaged over 10 different random graphs. Each graph has exactly n vertices and m edges. In addition, each graph has a perfect matching.

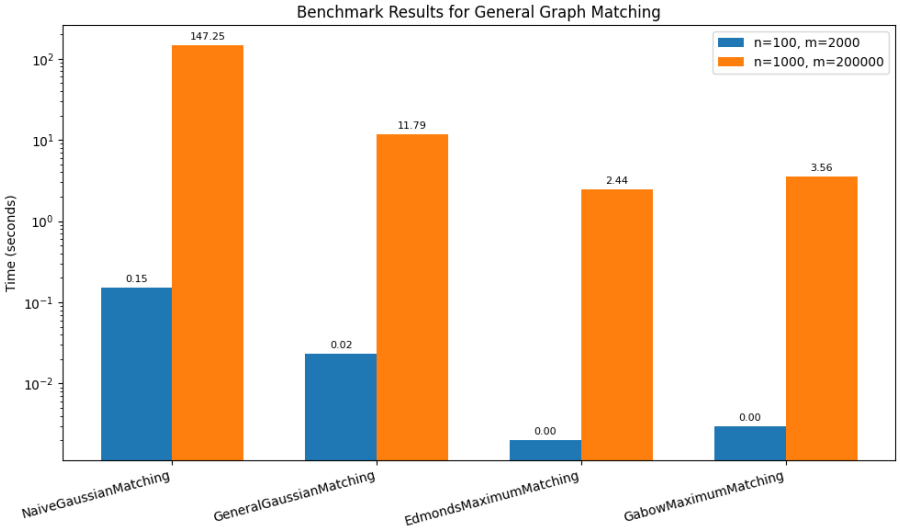
Algorithm	$n = 100$ $m = 2000$	$n = 1000$ $m = 200000$
NaiveGaussianMatching	129 ms	129.67 s
GeneralGaussianMatching	66 ms	4.96 s
BipartitegGaussianMatching	7 ms	4.88 s
EdmondsMaximumMatching	2 ms	2.35 s
GabowMaximumMatching	3 ms	3.35 s

Table 4.1: Benchmark Results for Bipartite Graph Matching

Algorithm	$n = 100$ $m = 2000$	$n = 1000$ $m = 200000$
NaiveGaussianMatching	150 ms	147.25 s
GeneralGaussianMatching	23 ms	11.79 s
EdmondsMaximumMatching	2 ms	2.44 s
GabowMaximumMatching	3 ms	3.56 s

Table 4.2: Benchmark Results for General Graph Matching



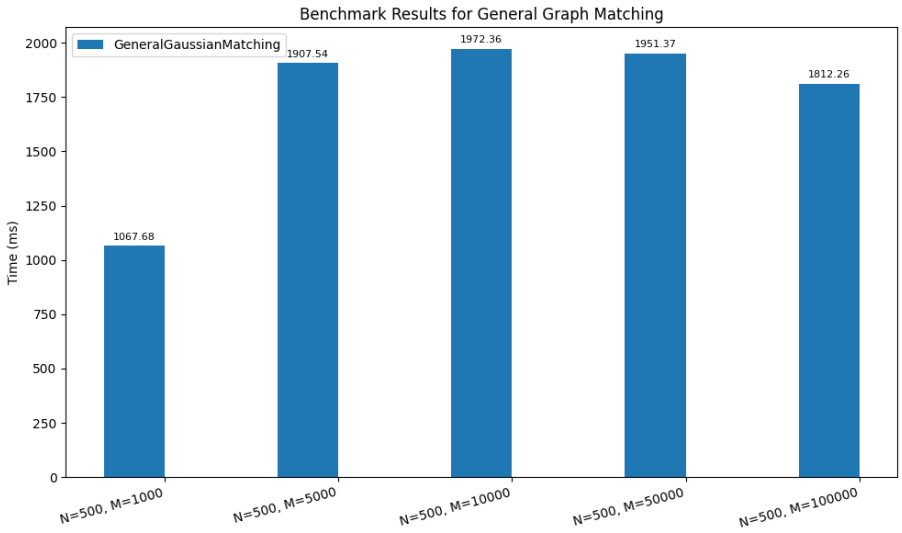


The implemented GeneralGaussianMatching and BipartiteGaussianMatching algorithms are slower than the compared algorithms. There are some improvements to be made that can improve the running time of both these algorithms. We will discuss them in the later section.

It is also worth noticing that there is a substantial improvement between these algorithms and the NaiveGaussianMatching algorithm running in $O(n^3)$.

We also consider a different benchmark. We present a graph showing how does the GeneralGaussianMatching changes for a different density graphs.

Each test case consists of 10 random graphs with 500 vertices and varying number of edges $m \in \{1000, 5000, 10000, 50000, 100000\}$. As we can see, there is little to no change between the test cases with $m = 5000$ and $m = 100000$. Furthermore, the running time decreases for the largest test. This may be caused by the fact that more edges become allowed in dense graphs. This causes the greedy algorithm to match more vertices, thus reducing the total running time of the algorithm.



4.1.3 Further work

There is a number of things that can be further done to improve the above results. Because of the limited time and a great effort put nonetheless, we left a few simplifications in the implementation. Here is a list of improvements that could be made:

- Implement fast matrix multiplication and an matrix inverse algorithm, both running in $O(n^\omega)$;
- Replace naive implementation of dynamic component with a proper, poly-logarithmic one;
- Handle separately the largest component in the PARTITION component.

4.2 Electrical flow

4.2.1 Implementation details

While the original paper by Mądry [18] does only contain a formula driven description of the algorithm, we have developed a pseudocde and a working `C++`

implementation. The main part of the algorithm consists of five files:

- `cpp/ElectricalFlow.cpp`
- `cpp/FlowNetwork.cpp`
- `cpp/ElectricalNetwork.cpp`
- `cpp/NaiveLaplaceSolver.cpp`*
- `cpp/NaiveDynamicTrees.cpp`*

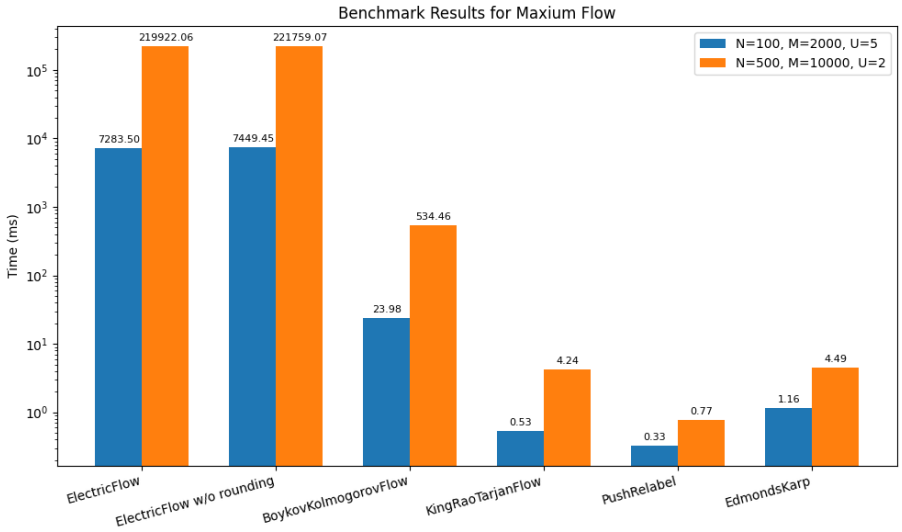
We also introduce FlowRounding implementation based on [14] that was not include in [18] explicitly. It utilizes a data structure called *dynamic trees* introduced in [26]. This data structure was implemented in a naive way that does not match the time complexity of [26]. Likewise, the `NaiveLaplaceSolver` is a simplified implementation using *Moore-Penrose pseudo-inverse*. The ElectricalFlow algorithm can be further improved by providing faster implementations of both LaplaceSolver and DynamicTrees.

4.2.2 Benchmark

We present a benchmark of the implemented algorithm. We compare its runtime to some algorithms already implemented in `Koala/NetworKit` and `NetworKit` itself. We present two test cases based on the size of n, m and $\max w$, each containing 10 random generated test.

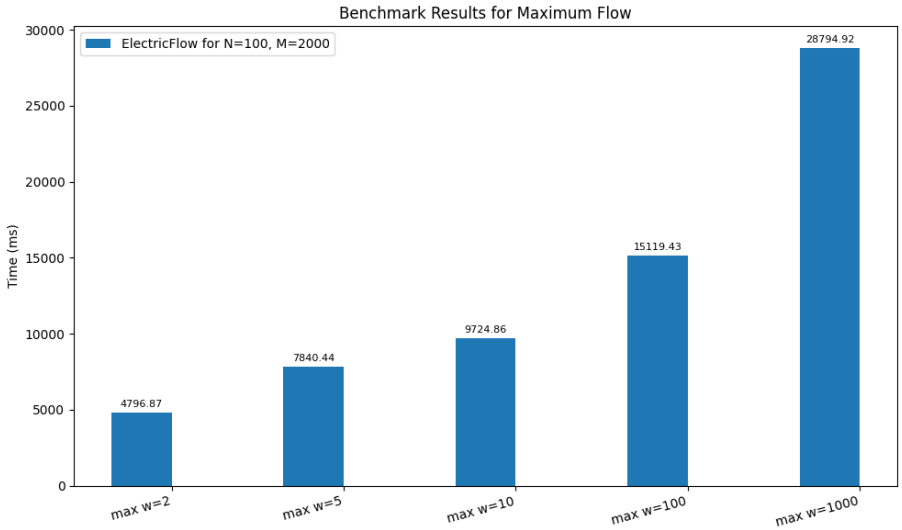
Algorithm	$n = 100$ $m = 2000$ $\max w = 5$	$n = 1000$ $m = 200000$ $\max w = 2$
ElectricalFlow	7.28 s	220 s
ElectricalFlow w/o rounding	7.28 s	222 s
BoykovKolmogorovFlow	23 ms	534 ms
KingRaoTarjanFlow	0.53 ms	4.24 ms
PushRelabel	0.33 ms	0.77 ms
EdmondsKarp	1.16 ms	4.49 ms

Table 4.3: Benchmark Results for General Graph Matching



Unfortunately, the ElectricalFlow algorithm is significantly slower than the other algorithms in the above benchmark. While there some improvements to the current implementation could be made, will not be sufficient to achieve results on par with the compared algorithms. One reason for this issue, is that the algorithm is theory-oriented. While the additional log in the time complexity might not seem a lot, calling RouteFlow multiple times increased the total runtime more than tenfold.

We present a graph showing the running time difference of ElectricalFlow algorithm for different values of $\max w$. The key difference between those results is the number of times RouteFlow method is called. In the above test cases RouteFlow was called about: $5, 6, 8, 11$ and 15 times. Based on this fact alone the $\max w = 1000$ test case runs about 1.5 times slower than the $\max w = 100$ test. A single RuouteFlow call is slower for the bigger tests as well. For these test cases we got up to $300, 400, 450, 650$ and 800 iterations per one call. The ElectricalFlow algorithm is thus very sensitive to the weights of the given graph.



4.2.3 Future work

Although we succeeded with the implementation of Mądry [18] maximum flow algorithm, there are some missing futures and improvements that could be added:

- Replace Moore-Penrose pseudo-inverse with a faster Laplacian solver;
- Add a reduction from directed graphs to the implemented undirected case;
- Implement a polylogaritmic dynamic trees data structure for the flow rounding algorithm;
- Benchmark algorithm with a preconditioned graph modification.

Bibliography

[1] Josh Alman et al. *More Asymmetry Yields Faster Matrix Multiplication*. 2024. arXiv: 2404.16349 [cs.DS]. URL: https://arxiv.org/abs/2404.16349.

[2] Aaron Bernstein, Joakim Blikstad, Thatchaphol Saranurak, and Ta-Wei Tu. *Maximum Flow by Augmenting Paths in $n^{2+o(1)}$ Time*. 2024. arXiv: 2406.03648 [cs.DS]. URL: https://arxiv.org/abs/2406.03648.

[3] James Bunch and John Hopcroft. "Triangular Factorization and Inversion by Fast Matrix Multiplication". In: *Mathematics of Computation* 28 (1974), pp. 231–236.

[4] Don Coppersmith and Shmuel Winograd. "Matrix multiplication via arithmetic progressions". In: *Journal of Symbolic Computation* 9.3 (1990), pp. 251–280.

[5] Jack Edmonds. "Paths, Trees, and Flowers". In: *Canadian Journal of Mathematics* 17 (1965), pp. 449–467.

[6] Jack Edmonds and Richard Karp. "Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems". In: *Journal of the ACM* 19.2 (1972), pp. 248–264.

[7] *Eigen: C++ Template Library for Linear Algebra*. Project website. URL: https://eigen.tuxfamily.org/.

[8] Delbert Fulkerson and George Dantzig. "Computation of maximal flows in networks". In: *Naval Research Logistics Quarterly* 2.4 (1955), pp. 277–283.

[9] Harold Gabow. "An Efficient Implementation of Edmonds' Algorithm for Maximum Matching on Graphs". In: *Journal of the ACM* 23.2 (1976), pp. 221–234.

[10] Andrew Goldberg and Robert Tarjan. "A new approach to the maximum-flow problem". In: *Journal of the ACM* 35.4 (1988), pp. 921–940.

[11] Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. "Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity". In: *Journal of the ACM* 48.4 (2001), pp. 723–760.

[12] John Hopcroft and Richard Karp. "An $n^{5/2}$ Algorithm for Maximum Matchings in Bipartite Graphs". In: *SIAM Journal on Computing* 2.4 (1973), pp. 225–231.

[13] Oscar Ibarra and Shlomo Moran. "Deterministic and probabilistic algorithms for maximum bipartite matching via fast matrix multiplication". In: *Information Processing Letters* 13.1 (1981), pp. 12–15.

[14] Donggu Kang and James Payor. *Flow Rounding*. 2015. arXiv: 1507.08139 [cs.DS]. URL: https://arxiv.org/abs/1507.08139.

[15] Harold Kuhn. "The Hungarian method for the assignment problem". In: *Naval Research Logistics Quarterly* 2.1–2 (1955), pp. 83–97.

[16] László Lovász. "On determinants, matchings and random algorithms". In: *Fundamentals of Computation Theory: Proceedings of the Conference on Algebraic, Arithmetic, and Categorial Methods in Computation Theory Held in Berlin/Wendisch-Rietz*. Ed. by Lothar Budach. Akademie-Verlag, 1979, pp. 565–574.

[17] László Lovász and Michael Plummer. *Matching Theory*. AMS Chelsea Publishing, 1986.

[18] Aleksander Mądry. "Computing Maximum Flow with Augmenting Electrical Flows". In: *Proceedings of the 57th Annual Symposium on Foundations of Computer Science (FOCS)*. 2016, pp. 593–602.

[19] Silvio Micali and Vijay Vazirani. "An O(v|v| c |E|) algoithm for finding maximum matching in general graphs". In: *21st Annual Symposium on Foundations of Computer Science (sfcs 1980)*. 1980, pp. 17–27. DOI: 10.1109/SFCS.1980.12.

[20] Marcin Mucha and Piotr Sankowski. "Maximum matchings via Gaussian elimination". In: 2004, pp. 248–255.

[21] *NetworKit: Efficient Network Analysis*. Project website. URL: https://networkit.github.io/.

[22] *NTL: A Library for doing Number Theory*. Project website. URL: https://libntl.org/.

[23] Michael Rabin and Vijay Vazirani. "Maximum matchings in general graphs through randomization". In: *Journal of Algorithms* 10.4 (1989), pp. 557–567.

[24] Kenneth Rosen. *Discrete Mathematics and its Applications*. 5th. McGraw-Hill, 2002.

[25] Arnold Schönhage. "Partial and Total Matrix Multiplication". In: *SIAM Journal on Computing* 10.3 (1981), pp. 434–455.

[26] Daniel Sleator and Robert Tarjan. "A data structure for dynamic trees". In: *Journal of Computer and System Sciences* 26.3 (1983), pp. 362–391.

[27] Volker Strassen. "Gaussian Elimination is not Optimal". In: *Numerische Mathematik* 13 (1969), pp. 354–356.

[28] Krzysztof Turowski. *Koala-NetworKit*. GitHub repository. URL: https://github.com/krzysztof-turowski/koala-networkit.

[29] William Tutte. "The Factorization of Linear Graphs". In: *Journal of the London Mathematical Society* s1-22.2 (1947), pp. 107–111.

[30] Wikipedia contributors. *Laplacian Matrix – Wikipedia, The Free Encyclopedia*. URL: https://en.wikipedia.org/wiki/Laplacian_matrix.

[31] Wikipedia contributors. *Moore–Penrose inverse — Wikipedia, The Free Encyclopedia*. URL: https://en.wikipedia.org/wiki/Moore%E2%80%93Penrose_inverse.

[32] Wikipedia contributors. *Norm (mathematics) — Wikipedia, The Free Encyclopedia*. URL: https://en.wikipedia.org/wiki/Norm_(mathematics).

[33] Wikipedia contributors. *Singular value decomposition – Wikipedia, The Free Encyclopedia*. URL: https://en.wikipedia.org/wiki/Singular_value_decomposition.

[34] Virginia Vassilevska Williams. "Multiplying matrices faster than Coppersmith-Winograd". In: *Proceedings of the 44th Annual ACM Symposium on Theory of Computing*. 2012, pp. 887–898.