Jagiellonian University
Department of Theoretical Computer Science

Rafał Kilar

Comparison of maximum weight matching algorithms on general graphs

Master's Thesis

Supervisor: dr hab. inż. Krzysztof Turowski

September 2024

Contents

1 Introduction **2**

2 Algorithms **5**
2.1 Blossom Algorithm 5
2.1.1 Edmonds' algorithm 7
2.1.2 Gabow's algorithm 13
2.1.3 Galil, Micali & Gabow algorithm 14
2.2 Scaling algorithms 18

3 Computational results **39**

Chapter 1

Introduction

The graph matching problem is among the most extensively researched topics in combinatorial optimization. Initial studies into the subject were motivated by practical issues including minimization of transportation costs [21] or optimally assigning personnel to tasks [25]. Over time, matching algorithms have found their use in scheduling, approximation algorithms and network switching among other problems. They play a crucial role in various other optimization algorithms including undirected shortest paths [22], planar max cut [20], metric traveling salesman [2], and Chinese postman tours [8].

Graph matching algorithms are mainly divided into four groups based on the classes of graphs they operate on: whether they're weighted or unweighted and whether they're bipartite or non-bipartite. We will focus on the solutions to the case of weighted matching on general graphs.

To discuss further, we first define some terms. Consider a weighted graph $G = (V, E)$ with edge weight function w. We use $n = |V|$ and $m = |E|$ unless otherwise specified. For a graph $G' = (H, E')$ induced by a vertex set $H \subseteq V$ we denote $n_H = |H|$ and $m_H = |E'|$.

A *matching* is a vertex-disjoint subset of the graph's edges. We call a matching *perfect* if all vertices belong to exactly one edge of the matching. We define the *weight* of a matching as the sum of its edges' weights and denote it as $w(M) = \sum_{e \in M} w(e)$.

In this work, we consider following variants of the maximum matching problem:

- Maximum Cardinality Matching (MCM) Find a matching in a graph G with maximum number of edges.
- Maximum Weight Matching (MWM) Find a matching in a weighted graph G with maximum weight.
- Maximum Weight Perfect Matching (MWPM) Find a perfect matching in a weighted graph G with maximum weight.

The MWM and MWPM are reducible to each other. For an instance $G = (V, E)$ of MWM, define a new graph $G' = (V', E')$ where $V' = V_1 \cup V_2$ and V_1, V_2 are two copies of V. The edges E' consist of copies of edges in E as well as one zero-weight edge between each corresponding pair of vertices in V_1 and V_2. A maximum weight perfect matching M' on G' can be used to obtain a maximum weight matching M on G by restricting the matching to only edges contained in V_1. If a vertex in V_1 is matched to its copy in V_2, it is unmatched in M. It's easy to see that M is a maximum weight matching on G as a matching with higher weight could be used to create a perfect matching on G' with weight higher than M'. In the other direction, let a graph $G = (V, E)$ with weight function w be an instance of MWPM. Construct a weight function $w'(e) = w(e) + nN$. A maximum weight matching on the graph $G' = G$ with weight function w' must have the maximum possible number of edges as the nN term in the definition w' ensures that any matching with more edges has a higher weight.

The first polynomial algorithm for maximum weight matching in general graphs was given by Edmonds in [6]. It uses the primal-dual method and relies on his previous work on an algorithm for maximum cardinality matching [7]. It is called the blossom algorithm after blossoms, which are certain odd length cycles, a feature that notably doesn't appear in bipartite graphs and is a major source of complexity in algorithms for general graphs. The running time given by Edmonds was $O(n^m)$. It was then independently improved to $O(n^3)$ by [22] and [18]. Many implementations of Edmonds' blossom algorithm followed with improved theoretical time complexity. The current best runs in $O(nm+n^2 \log n)$ due to [13], which is in some sense optimal. The blossom algorithm works in $O(n)$ phases, a single of which can be used to sort n numbers and therefore requires $\Omega(m + n \log n)$ time in an appropriate model of computation. Other algorithms use an approach called scaling to solve the problem for graphs with integer edge weights. The weights are exposed one bit a time. In the ith scale, the algorithm computes the optimal solution for weights w_i consisting of i significant bits of w. The solution is then used to more efficiently solve the next scale. The first algorithm based on this approach was presented in [15] and runs in time $O(n^{3/4}m)$. This was later improved to $O(m\sqrt{n\alpha(m,n)} \log n \log(nN))$ in [16] and lately $O(m\sqrt{n} \log(nN))$ in [5]. An algebraic randomized algorithm by [3] runs in $O(n^\omega N)$ with high probability for the matrix multiplication exponent ω.

A summary of existing algorithms is presented in table 1.

We will take a closer look at a selection of maximum weight matching algorithms, namely Edmonds' original $O(n^2m)$ algorithm, Gabow's $O(n^3)$ algorithm, Galil, Micali & Gabow's $O(nm \log n)$ algorithm and Gabow's first scaling algorithm running in $O(n^{3/4}m \log N)$. We implement all of them in C++20 as part of the open-source KOALA NetworKit library [27]. In chapter 2 we describe how they work and how to implement them. In chapter 3, we present results of computational tests and compare the performance of our implementations.

Author	Year	Running time
Edmonds	1965	n^2m
Gabow	1974	n^3
Lawler	1976	n^3
Gabow	1985	$n^{3/4}m \log N$
Galil, Micali & Gabow	1986	$nm \log n$
Gabow, Galil & Spencer	1989	$nm \log\log\log_{2+m/2} + n^2 \log n$
Gabow	1990	$nm + n^2 \log n$
Gabow & Tarjan	1991	$m\sqrt{n\alpha(m,n)\log n}\log(nN)$
Cygan, Gabow & Sankowski	2012	$n^\omega N$
Duan, Pettie & Su	2018	$m\sqrt{n}\log(nN)$

Table 1: Summary of maximum weight matching algorithms

Chapter 2

Algorithms

2.1 Blossom Algorithm

Consider a matching M in G. We define some terms.

An edge is *matched* if it belongs to the matching and *unmatched* otherwise. A vertex is *matched* if it belongs to one of the edges of the matching, otherwise we call it *exposed*. For a matched vertex u with corresponding edge $(u, v) \in M$, the vertex u is called a *mate* of v.

A path $e_0, e_1, \dots e_n$ is *alternating* if it visits each vertex at most once and $e_i \in M \Leftrightarrow e_{i+1} \notin M$, meaning its edges are and aren't matched in alternating turns. If an alternating path starts and ends in exposed vertices we call it an *augmenting* path.

We take special consideration of specific sets of vertices of odd size which we will refer to as *blossoms*. We define blossoms recursively. Each vertex by itself is a *trivial* blossom. Denote by $\mathcal{O}$ the set of all odd-sized subsets of V with more than 1 vertex.

An edge sequence $e_0, e_1, \dots, e_{n-1}$ where $e_i = (u_i, v_i)$ is alternating for blossom $B_0, B_1, \dots, B_n$ if $u_i \in B_i$ and $v_i \in B_{i+1}$ and $e_i \in M \Leftrightarrow e_{i+1} \notin M$. Similarly, it is *augmenting* if $e_0, e_{n-1} \notin M$.

Consider a sequence of blossoms $B_0, B_1, \dots, B_n$ where n is odd, $B_0 = B_n$ with an alternating path of odd length $e_0, e_1, \dots e_{n-1}$ where $e_0, e_n \notin M$. The blossom $B_1, \dots, B_n$ combine to form a new blossom B and are called its *sub-blossoms*.

For each blossom we define its *base*. When a blossom is trivial its sole vertex is the base. For a new blossom B defined as above its base is the base of B_n.

Consider a blossom B with base vertex b, subblossoms $B_1, B_2, \dots, B_n$ and edges $e_0, e_1, \dots e_{n-1}$. We observe some useful facts about blossoms:

(1) Any vertex $c \neq b$ of B is matched to another vertex in B other than b.

(2) If b is matched, its mate is outside B

(3) For every subblossom B_i where $0 < i < n$, the sequences $e_0, e_1, \ldots, e_{i-1}$ and $e_n, e_{n-1}, \ldots, e_i$ are alternating paths from B_0 to B_i. One of them is of odd length and the other one is of even length.

(4) There exists an even length alternating path between b and any vertex $c \in B$. The proof is by induction. If B is trivial the path is empty. If it's nontrivial, find the subblossom B_i of B such that $c \in B$ and choose the even length path from those described in (3). For any subblossom on the path B_i exactly one of e_{i-1} or e_i is in M, and it's adjacent to the base b_i of B_i according to (1) and (2). The other of the two edges is adjacent to some $d \in B_i$. By induction, there is an even length alternating path from b_i to d. Inserting said path between e_{i-1} and e_i preserves the fact that the path is of even length and alternating. Doing so for each subblossom on the alternating path yields the desired path between b and c.

The structure of a blossom B can be represented by a tree T_B. The root of T_B corresponds to B. When B is nontrivial, the children of T_B's root are trees corresponding to subblossoms $B_1, \ldots, B_n$ of B. The leaves of T_B correspond to individual vertices that comprise B. We refer to T_B as B's *structure tree*.

The order of leaves in B structure tree implies an order on B's vertices. We call the list $L(B)$ of vertices of B in the order of their corresponding leaves in T_B the blossoms blossom list. What's important is that a blossom list for any of the subblossoms of B or other any of their descendants is a substring of $L(B)$.

During the execution of the algorithm we will find and construct new blossoms and shrink all nodes and edges of the blossom into a single node. We might also expand blossoms and return their subblossom to the state they were in before the expanded blossom was created. We call the blossoms which aren't currently subblossoms of any other blossom *proper*. At any point in the algorithm we refer to the graph whose vertices correspond to proper blossoms as the *current graph*. For each vertex of the original graph we refer to the unique proper blossom it belongs to as its *current blossom*.

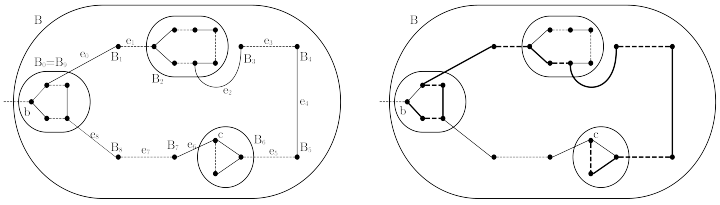

Figure 1: Example of a blossom B with subblossom list $(B_1, e_1), \ldots, (B_8, e_8), (B_9, e_0)$ and a base b. Edges in the matching are indicated with a dashed line. On the right an alternating path of even length between B's base b and an example vertex c is highlighted.

2.1.1 Edmonds' algorithm

Edmonds' maximum weight matching algorithm from [6] is based on the primal-dual method [22].

The maximum weight matching problem in a graph $G = (V, E)$ can be expressed as an integer linear program with variables x_e for each edge $e \in E$

$$\begin{aligned} \text{maximize } & \sum_{e \in E} x_e w(e) & \\ \text{subject to } & \sum_{v} x_{uv} \leq 1 & \text{for each } u \in V \\ & x_e \in \{0, 1\} & \text{for each } e \in E \\ & x_e \geq 0 & \text{for each } e \in E \end{aligned}$$

In order to get rid of the integer constraint, Edmonds introduced an exponential number of constraints for all odd sized vertex subsets belonging to $\mathcal{O}$. He proved that each optimal solution to the new problem consists only of vectors x with values 0 and 1 which correspond to matchings in G.

$$\begin{aligned} \text{(MWM)} \qquad \text{maximize } & \sum_{e \in E} x_e w(e) & \\ \text{subject to } & \sum_{v} x_{uv} \leq 1 & \text{for each } u \in V \\ & \sum_{u,v \in B} x_{uv} \leq \left\lfloor \frac{1}{2} n_B \right\rfloor & \text{for each } B \in \mathcal{O} \\ & x_e \geq 0 & \text{for each } e \in E \end{aligned}$$

The above primal linear program has a following dual program with variables y_v for each vertex $v \in V$ and z_B for each odd sized vertex subset $B \in \mathcal{O}$:

$$\begin{aligned} \text{(}\overline{\text{MWM}}\text{)} \qquad \text{minimize } & \sum_{v \in V} y_v + \sum_{B} z_B \left\lfloor \frac{1}{2} n_B \right\rfloor & \\ \text{subject to } & y_u + y_v + \sum_{u,v \in B} z_B \geq w(uv) & \text{for each } uv \in E \\ & z_B \geq 0 & \text{for each } B \in \mathcal{O} \\ & y_v \geq 0 & \text{for each } v \in V \end{aligned}$$

For each edge $e = (u, v) \in E$ we introduce a value we call *slack* defined as follows:

$$slack(e) = \pi_e = y_u + y_v - w(e) + \sum_{u,v \in B} z_B$$

We say that an edge e is *tight* if $slack(e) = 0$. We say that dual weights y and z are *dominating* if $slack(e) \geq 0$ for each $e \in E$.

Theorem 2.1.1. *given dual variables y and z, a matching M is a maximum weight matching if the following conditions hold:*

(1) $y_v, \pi_{uv}, z_B \geq 0$ *for each* $v \in V, uv \in E, B \in \mathcal{O}$

(2) $\pi_{uv} = 0$ *for each edge* $uv \in M$

(3) $y_i = 0$ *for each exposed vertex* i

(4) $|\{e|e \subseteq B, e \in M\}| = \left\lfloor \frac{1}{2} n_B \right\rfloor$ *for each blossom* B *where* $z_B > 0$

Proof. Consider an arbitrary matching M'. We show that M has a higher weight:

$$\begin{aligned}
\sum_{uv \in M'} w(uv) &= \sum_{uv \in M'} \left(y_u + y_v + \sum_{u,v \in B} z_B \right) && \text{from } \pi_{uv} \geq 0 \\
&\leq \sum_{v \in V} y_v + \sum_{B \in \mathcal{O}} z_B \left\lfloor \frac{1}{2} n_B \right\rfloor && \text{from } y_v, z_B \geq 0 \\
&= \sum_{uv \in M} \left(y_u + y_v + \sum_{u,v \in B} z_B \right) && \text{from (2) and (3)} \\
&= \sum_{uv \in M} w(uv) && \text{from (1)}
\end{aligned}$$

□

The algorithm starts with a matching M and dual weights y and z that fulfill (1), (2) and (4) and continually decreases the number of exposed vertices v with $y_v > 0$ by either matching them or decreasing their dual variable to 0. We maintain that $z_B > 0$ only for blossoms B of the current graph.

We start with an empty matching M_0 and $y_v = \max_{e \in E} w(e)/2$ and no blossoms, values which trivially fulfil the desired constraints.

The algorithm works in phases we call *stages*. Each stage finds an augmenting path between two exposed blossoms comprised of solely tight edges and augments the matching along the path. The execution ends when there are no more exposed vertices or all dual variables y_v for all exposed vertices v have been reduced to 0.

2.1.1.1 Search algorithm

The main part of the algorithm is concerned with finding an augmenting path between two exposed blossoms in the current graph. We will build trees comprised of alternating paths rooted in exposed blossoms which we will call *search trees*. Every blossom can be labeled with one of three values: *even*, *odd* or *free*. Blossoms labeled with *free* are outside the search trees, while those that

have been reached are labeled based on whether the alternating path has an even or odd length. We refer to blossoms labeled with even as *even blossoms* and all their vertices *even vertices* with analogous names for the remaining labels. At the beginning, all exposed blossoms are marked as even with the rest of blossoms marked as free.

Besides the label, for each blossom, we remember the edge by which the blossom has been reached we call its *backtrack edge*. We refer to the collection of search trees as the *search structure*.

In order to expand the search structure we will look for so-called *useful edges*, which are defined as tight edges either between even and free vertices or between two even vertices contained in distinct proper blossoms. Based on which of the two conditions hold and whether the search trees of the two even blossoms are distinct, one of three steps can take place: *grow*, *blossom* or *augment*.

Grow step In the case when the useful edge connects an even vertex u to a free vertex v, the grow step takes place. Let B_v be the blossom of v. We mark B_v as odd and set its backtrack edge to uv. Let p be the base of B_v. Because B_v was previously free, p has to be matched to some vertex q and its blossom B_q. We mark B_q as even and set its backtrack edge to pq. Vertices of B_q become even for the first time so some of their adjacent edges might become useful.

Backtrack procedure When considering a tight edge $e = (u, v)$ between two even blossom B_u and B_v we backtrack using the previously described backtrack edges. We either find an augmenting path between two exposed blossoms at the roots of B_u and B_v's search trees and perform the augment step or the two blossoms are in the same search tree, in which case we execute the blossom step. During the backtracking procedure we have to be careful not to unnecessarily go back to the root of the search tree when the two paths converge at some earlier point. To do that we advance simultaneously along the two paths one step at a time and mark the blossom as visited while checking if the paths have met.

Augment step If an augmenting path between two exposed blossoms is found, the augment step is performed and the current stage comes to an end. The augmentation consists of swapping the edges along the found path in the current matching M. We have previously described how to recursively find alternating paths in blossoms. After such an augmentation the bases of blossoms along the paths change and with them the order of subblossom lists, which get shifted cyclically to put the blossom containing the new base at the end. Notice we don't actually need to perform the augmentation inside the blossoms and instead can do it lazily. The only information about blossom we need are which vertices belong to it and what its base is. Instead of swapping edges along an even alternating path inside the blossoms along the augmenting path, we can simply change their bases, which saves time with repeated augmentations. The

augmentation is only done when the blossom is expanded. We do so by swapping edges along the even path from the blossom's initial base to its current base. We again only do so for the blossom edges and only update the bases of subblossoms. At the end of the algorithm we expand all blossoms to reveal the final matching. The lazy augmentation makes it harder to maintain the matching. We store it by remembering each vertex's mate and for each edge whether it's part of the matching. When expanding the blossom, besides swapping edges along the even path we also fix the mate array for remaining edges.

Blossom step When the two paths meet at some blossom B_q, a new blossom B is formed. The base of the new blossom is the base q of B_q. As the paths split in B_q, it has to be an even blossom which is why we also label B as such. The backtrack edge for B is B_q's backtrack edge, which is also the edge matched to q, or it's empty if B_q is at the root of a search tree. Notice that vertices in odd subblossoms of B become even for the first time in the current stage, so some of their adjacent edges might turn useful.

Dual weight adjustment step Whenever there are no more useful edges, a dual weight adjustment step is performed. For a chosen $\delta > 0$ we adjust the values of dual variables as follows:

- $y_v \leftarrow y_v - \delta$ for all even vertices v
- $y_v \leftarrow y_v + \delta$ for all odd vertices v
- $z_B \leftarrow z_B + 2\delta$ for all even blossoms B
- $z_B \leftarrow z_B - 2\delta$ for all odd blossoms B

Such an adjustment strictly decreases the dual objective function getting us closer to the optimal weights. Simple calculation shows that the dual objective function changes by $\delta(o - e)$ where e is the number of even blossoms and o the odd ones. Each tree in the search forest starts with an even blossom at the root and by the grow step, all odd blossom lead to a unique even blossom connected to it with an edge in the current matching, meaning there are more even blossoms than odd ones and the dual objective function decreases.

We choose a maximum value of δ that preserves the (1), (2) and (4) constraints. In order to do that we compute 4 candidate values $\delta_1, \delta_2, \delta_3$ and δ_4 and choose $\delta = \min\{\delta_1, \delta_2, \delta_3, \delta_4\}$. We define them as follows:

- $\delta_1 = \min_u y_u$ for even vertices u – preserves the constraint $y_u \geq 0$
- $\delta_2 = \min_{uv} \pi_{uv}$ for edges $uv \in E$ where u is even v is free – preserves $\pi_{uv} \geq 0$
- $\delta_3 = \min_{uv} \frac{1}{2}\pi_{uv}$ for edges $uv \in E$ where u and v are even – preserves $\pi_{uv} \geq 0$

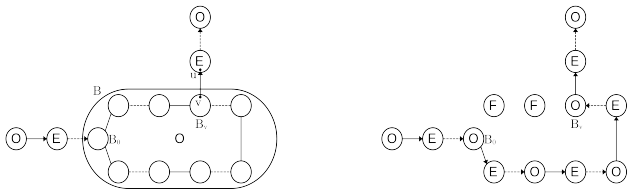

Figure 2: An example odd blossom B before and after expansion. The arrows indicate backtrack edges for each blossom. Blossoms are marked based on their label: E for even, O for odd and F for free blossoms.

- $\delta_4 = \min_B \frac{1}{2} z_B$ for odd blossoms B – preserves $z_B \geq 0$

Based on which of the above values we choose for δ a few different things can happen.

When $\delta = \delta_1$, all even vertices v have their dual variables reduced to $y_v = 0$. This includes all exposed vertices which fulfill the constraint (3). With all optimality constraints satisfied, the current matching M is maximum and the algorithm is done.

When $\delta = \delta_2$ or $\delta = \delta_3$ the edges for which the minimum was achieved become tight as their slack reaches 0, meaning they're now useful and we can continue with the search.

When $\delta = \delta_4$ we expand all odd blossoms B for which $z_B = 0$ after the dual weight adjustment. Let b be the base of B and uv be B's backtrack edge with B_v being the subblossom of B that contains v. We find an alternating path of even length over subblossoms of B from B_0 to B_v and label the subblossom on the path even and odd in turns while setting their backtrack edges accordingly. The remaining subblossoms become free. As new vertices become even, some edges might become useful in this process.

At the end of the stage all even blossoms B with $z_B = 0$ are expanded to preserve the constraint (4).

2.1.1.2 Complexity

All new blossoms are even and even blossoms are only expanded at the end of the stage, meaning all even blossoms correspond to some node in a structure tree of one of the proper blossoms at the end of the stage. Similarly, all odd blossoms correspond to nodes in structure trees at the beginning of the stage. Overall, there's at most $O(V)$ different blossoms during a stage.

We count the number of dual weight adjustments. We get $\delta = \delta_1$ at most once as it leads to the end of the algorithm. Whenever $\delta = \delta_2$, the newly useful edges lead to grow steps during which new blossoms are labeled as odd and even. When $\delta = \delta_3$, either the blossom step is performed and a new even blossom is added or the augment step is executed and the stage is finished. Finally, each time $\delta = \delta_4$, an odd blossoms is expanded. We see that there's at most $O(n)$

dual weight adjustments in a single stage as their number is bounded by the number of unique blossoms. We call the section of the algorithm between each dual weight adjustment a *substage*.

The Edmond's original algorithm spends most of it's time calculating δ for each dual weight adjustment and finding new useful edges. It maintains a doubly linked lists of vertices for each proper blossom, which are split and concatenated whenever blossoms are created or expanded. For each vertex its current blossom is maintained in a simple array and updated together with the vertex lists. As there are $O(n)$ unique blossoms, it takes $O(n)$ time to maintain vertex lists and $O(n^2)$ to update the current blossom array in each stage. We also maintain a doubly linked list of all proper blossoms which we can update in constant time each time a new blossom appears by storing a pointer to its position in said list for each blossom.

Once a vertex becomes even, it stays that way, so we only need to scan its edges once either at the start of the stage, after a grow step or after an odd blossom expansion. All of that takes $O(m)$ time per stage. We calculate δ by iterating over all vertices and edges. If $\delta = \delta_2$ or $\delta = \delta_3$ we also check which edges achieved the minimum and become useful. Each dual weight adjustment step takes $O(n + m) = O(m)$ time for a total cost of $O(nm)$ per stage. With $O(n)$ stages the total time complexity of Edmond's algorithm comes down to $O(n^2m)$ or $O(n^4)$ when bounding m by n^2.

2.1.1.3 Numerical accuracy

The algorithm is defined for real numbers, but for practical reasons we would like to use integer data types. This is complicated by the presence of division which we do at two points – when calculating the starting values of y and when finding δ_3. In [18], it is shown that if the weights $w(e)$ are even for all edges $e \in E$ all of these calculations yield integer values.

At the start of the algorithm, we initialize the dual weight y_v for all v to the same value $\max_{e \in E} \frac{1}{2} w(e)$, which is obviously an integer when all weights are even.

Remember that we defined

$$\delta_3 = \min_{\substack{uv \in E \\ u,\, v \text{ even}}} \frac{1}{2}\pi_{uv} = \min_{\substack{uv \in E \\ u,\, v \text{ even}}} \frac{1}{2} \left(y_u + y_v - w(e) + \sum_{u,v \in B} z_B \right)$$

As dual variables z_B start at 0 and are changed by 2δ during dual weight adjustment, we can ignore them along with the edge weights. It is then sufficient to prove that weights y_v are of the same parity for all even vertices v. All exposed vertices start with the same dual weights which are adjusted by $-\delta$ with each dual weight adjustment, so their dual weight are all equal. It is enough for us to show that for any even $v \in V$ the parity of y_v is the same as y_u where u is the base of the exposed blossom at the root of v's search tree.

Let B_v and B_u be u and v's current blossoms. There exist an even length alternating path P between v and u. This path can be obtained by finding the

even length path from B_v to B_u that follows backtrack edges. The path is then expanded in similar way to how finding an even length path from a vertex to its blossom base. Let M be the current matching. We can swap edges along P and obtain a new matching M'. All edges in P are tight, so we can write the weight of M as

$$\sum_{uv \in M} w(uv) = \sum_{uv \in M} y_u + y_v + \sum_{u,v \in B} z_B$$

We can ignore the weights z_B as they're all even. The sums for M and M' share all vertices except for u and v so the difference between M and M''s weights is of the same parity as $y_u - y_v$. As all edge weights are even, so are the weights of the two matchings meaning that $y_u - y_v$ is even and y_u and y_v are of the same parity.

2.1.2 Gabow's algorithm

Most of the time in Edmonds' original implementation is spent calculating δ_2 and δ_3 as well as finding useful edges during each dual weight adjustment. These particular issues are addressed by Gabow in his implementation from [18].

For each non-even vertex v we maintain $edge(v)$ which we define as the edge (v, u) with the smallest possible slack that connects v to an even vertex u.

For each even blossom B we maintain a list $edges(B)$ consisting of edges $(v_1, u_1), \ldots, (v_n, u_n)$ sorted such that $u_1 < u_2 < \cdots < u_n$ where for each $i = 1, \ldots, n$: $v_i \in B$, $u_i \notin B$, u_i is even and v_i became even after u_i and (v_i, u_i) has the smallest slack among such edges (v, u_i) where $v \in B$. We also remember the one edge in $edges(B)$ with the smallest slack and denote it $edge(B)$.

With those values we can perform a dual weight adjustment in time $O(V)$ by calculating:

$$\delta_2 = \min_{\substack{uv \in E \\ u \text{ even},\ v \text{ free}}} slack(uv) = \min_{u \text{ even}} slack(edge(u))$$

$$\delta_3 = \min_{\substack{uv \in E \\ u,\ v \text{ even}}} \frac{1}{2} slack(uv) = \min_{B \text{ even}} \frac{1}{2} slack(edge(B))$$

We can find new useful edges after a dual weight adjustment in $O(n)$ by checking which edges achieved the minimum. Notice that the minimum edge in $edge(v)$ and $edge(B)$ doesn't change with a weight adjustment as all relevant edges have their slack changed by the same amount.

Whenever a blossom B becomes even either at the start of a stage, after a grow step or during odd blossom expansion, we check all edges adjacent to vertices of B. For each $v \in B$ we iterate over its neighbors u in ascending order. If u is not even we update $edge(u)$ if (u, v) has smaller slack. If u is even we merge it into $edges(B)$ if it has smaller slack than any other edge (v', u) in the list. To do this efficiently we take advantage of the fact that the list and neighbors are sorted by utilizing a pointer to an element of $edges(B)$ that we

reset to the beginning for each $v \in B$ and move along the list according to the value of u. We only scan each vertex once during a stage, so we spend at most $O(n^2)$ time per stage doing this.

The only other time the lists *edges* change is when a new blossom B is found. We first perform the previously described scan for all odd subblossoms of B and then merge all the lists together. We perform at most $O(n)$ merges per stage which we can each do in $O(n)$ using the classic sorted list merge algorithm costing us $O(n^2)$ time per stage.

To sum up we spend $O(n^2)$ time maintaining the new values and $O(n^2)$ doing dual weight adjustment per stage. All the other calculations stay the same as in Edmond's algorithm and cost $O(n^2)$ per stage giving us a final running time of $O(n^3)$.

2.1.3 Galil, Micali & Gabow algorithm

Implementations of the blossom algorithm described above spend most of their time doing dual weight adjustments and finding useful edges. The authors of [23] showed that it is possible to perform dual weight adjustment in constant time by taking advantage of the fact that large groups of weights are changed by the same amount. We start by describing the specialized priority queues they've devised to do so and we show how they're used in the search stage of the blossom algorithm.

2.1.3.1 Data structures

We define a specialized priority queue we call pq_1 which operates on integer elements from $\{0, \ldots, n-1\}$. Elements start outside the queue and can be inserted or deleted from it. Elements inside the queue are called *current*. Each current element has an associated priority. We support following operations:

- insert(i, p) insert an element i with priority p or update its priority if p is smaller than the current one in time $O(\log n)$
- delete(i) delete the element i in time $O(\log n)$
- findmin() find the element with the lowest priority in time $O(1)$
- decrease(δ) decrease the priorities of all current elements by δ in time $O(1)$

Let Δ be the sum of all priority decreases. We make use of a priority queue Q which supports insertion, deletion, finding minimum and updating priority which in our case is implemented with an array heap. We don't store elements' exact priorities in Q instead we use their *modified priorities*. An element's modified priority is calculated at the moment of insertion into the priority update. When an element's priority is set to p we set its modified priority to the value $p + \Delta$ and the time of the change and store it as such in Q. Additionally, we store the modified priority in an array to support checking an element's current priority. Any element's current priority can be calculated

by adding Δ to its modified priority. The order of elements' modified priorities is the same as the order of their real priorities as they're all just shifted by Δ.

We will also make use of what we call *concatenable queues*. Each queue contains a list of elements in a specific order. Each element has a corresponding priority. The queues can be concatenated together or split at a specified element. Elements are referenced using handles which are preserved by split and concatenate operations. An elements handle allows one to find its current queue.

- init() create an empty queue
- append(q, i, p) append an element i with priority p to the end of the queue q in $O(\log n)$
- delete(i) delete the element i from its queue in time $O(\log n)$
- findmin(q) find the element in queue q with the lowest priority in time $O(1)$
- concat(q_1, q_2) concatenate queues q_1 and q_2 in time $O(\log n)$
- split(q, i) split q into two new queues: one that contains all elements in q up to i and one that contains all elements after i in q in time $O(\log n)$
- findqueue(i) return i's current queue in time $O(\log n)$

We implement concatenable queues using 2–3 trees which support splitting and concatenation as described in [1]. Elements are stored in the leaves of the tree with their order determined by the order of the leaves. Each inner node stores the element with the lowest priority among its descendants along with said priority, meaning the minimum can be easily accessed by checking the root. The elements are referenced using pointers to their corresponding leaves. We take care to maintain said pointers during splits and concatenations. The root of the tree stores a pointer to it's queue which allows us to implement findqueue with a simple walk from its corresponding leaf. To differentiate between queues each one stores an ID in its root which can be set during init, concat or split operations.

The last data structure we need is priority queue pq_2. Just like pq_1 it operates on elements from a predetermined universe $\{0, \ldots, n-1\}$ which are divided into groups. The elements in a given group have an order and priority. Each group can be either *active* or *nonactive*. We call elements active if they're inside an active group. We support modifying the priorities of all active elements by a provided value. A group's status can be changed at any time and their elements can be split to create two new groups. Lastly we can retrieve the active element with the smallest priority. To summarize we support operations:

- creategroup() create a new empty group

- append(g, i, p) append an element i with priority p to the end of the group g in $O(\log n)$
- update(i, p) update i's priority to p if it's smaller than the current one in time $O(\log n)$
- findmin() find the active element with the lowest priority in time $O(1)$
- split(g, i) split g into two new groups: one that contains all elements in q up to i and one that contains all elements after i in q in time $O(\log n)$
- change(g, s) change g's status in time $O(1)$
- delete(g) delete the group g
- decrease(δ) decrease the priorities of all active elements by δ

We store the sum of changes from the decrease function in a variable Δ.

For each group g we sum up all the changes to its elements' priorities in a variable Δ_g. This value is updated each time we refer to g. To do this we also store the value Δ_{last} which corresponds to the value of Δ last time we looked at g. If during that time g was active, we increase Δ_g by $\Delta - \Delta_{last}$. We then update Δ_{last} to the current value of Δ regardless of g's status. The elements of g are stored in a concatenable queue Q_g. The priorities of the elements in Q_g again use modified priorities similarly to pq_1 but using Δ_g instead.

In order to find a minimum active element we maintain a pq_1 called $Q_{\min}$ which for each active group stores its smallest element according to priority. Whenever a new element is added or a priority is changed in an active group we check if we need to update its corresponding entry in $Q_{\min}$. When a group is set to nonactive or deleted we remove said entry. Similarly, during a split if the group is active we need to replace its entry with entries for the two resulting groups. We need to make sure to store the elements' actual priorities in $Q_{\min}$. The decrease function simply calls decrease on $Q_{\min}$.

2.1.3.2 Search procedure

We use concatenable queues to maintain the current blossoms for each vertex. For a given blossom, its vertices are stored in a separate concatenable queue with the same order as the blossom's vertex list $L(B)$. We perform the appropriate split and concatenation operations whenever blossoms are created or expanded. Maintaining the queues costs us $O(n \log n)$ per stage. The queues' ids store references to their corresponding blossoms allowing us to retrieve a given vertex's current blossom in $O(\log n)$.

To maintain dual weights y_v we make use of two queues pq_1 we call y_{even} and y_{odd} which we use to maintain values y_v for correspondingly even and odd vertices. The values for free vertices are stored in an array y_{free}. Similarly, for weights z_B two queues pq_1 z_{even} and z_{odd} for even and odd blossoms are used. Then B is free, its dual weight z_B is stored int a field in the blossom struct. To

retrieve the current value of y_v or z_B for a vertex v or blossom B (mainly for the purpose of calculating an edge's slack) the algorithm checks their label and refers to the appropriate queue.

In order to calculate the value of δ_3, we employ a pq_1 called Q_{good} in which we store edges uv between two different even blossoms which we refer to as *good edges*. Each edge's priority in Q_{good} corresponds to its current slack. Over the course of the algorithm, as new even blossoms are created, some of those edges might become contained within a single even blossom. We don't have enough time to detect it at the time it happens, so instead we remove them in a lazy manner each time we want to check the minimum by removing the smallest edge as long as it's no longer good.

To calculate the value of δ_2 efficiently we make use of a pq_2 called Q_{even}. Each non even blossom B has a corresponding group g_B. The elements of g_B are vertices of B stored in blossom order. The priority of $v \in B$ in Q_{even} corresponds to the minimum slack of an edge $uv \in E$ such that u is even. We also remember the edge for which the minimum is achieved. Group g_B is active when B is free and nonactive when it's odd. Free blossoms may become odd and odd blossoms might be expanded into even, odd or free subblossoms, so we need to change a group's status and perform splits whenever necessary.

Beginning of a stage For free blossom we initialize groups in Q_{even} and set all priorities to inf. For even blossoms we scan outgoing edges to update Q_{even} and Q_{good} and start tracking the dual variables for even vertices using y_{even}.

Grow step Two blossoms are added to the search structure, one even and one odd. We scan edges adjacent to the newly even vertices to update Q_{good} and Q_{even}. To maintain the y weights we add the blossoms' vertices to y_{even} and y_{odd} according to their label.

Blossom step We concatenate all vertex queues for all subblossoms. For each odd subblossom we scan its outgoing edges to update Q_{good} and Q_{even}, move its vertices from y_{odd} to y_{even} and delete its corresponding group in Q_{even}. We remove the subblossoms from z_{even} and z_{odd} and insert the new blossom into z_{even}.

Odd blossom expansion Let B be the odd blossom that is being expanded. We split B's group in Q_{even} and assign the new groups to corresponding subblossoms. We delete the groups for even subblossom as they're no longer in use. For non-even ones we set the status according to the label – active for free subblossoms and nonactive for odd. We also swap the way we track y according to the new labels.

Dual weight adjustment step We can calculate δ using our $findmin$ functions for corresponding queues:

- y_{even} to calculate δ_1,
- Q_{good} to calculate δ_2,
- Q_{even} to calculate δ_3,
- z_{odd} to calculate δ_4.

To adjust dual weights we use the decrease functions to change the priorities in all the queues:

- by δ in y_{even},
- by $-\delta$ in y_{odd},
- by -2δ in z_{even},
- by 2δ in z_{odd},
- by 2δ in Q_{good},
- by δ in Q_{even}.

End of a stage After the search finishes we clear all the queues after first moving the current priorities from queues y_{even}, y_{odd}, z_{even} and z_{odd} into the y_{free} array and the z field to keep the dual weights up to date for the next stage.

2.1.3.3 Complexity

There are $O(n)$ stages in the algorithm. During each stage we consider each edge at most twice. Each time we may make a constant number of calls to various queues, each of which takes $O(\log n)$ time. Maintaining queues for blossoms by executing splits and concatenations when necessary takes $O(n \log n)$ time per stage. The $O(n)$ dual adjustments each take constant time. Single stage takes $O((n + m) \log n) = O(m \log n)$ time for a total running time of $O(nm \log n)$.

2.2 Scaling algorithms

One of the limiting factors in the running time of algorithms based on the Blossom algorithm is the fact that they find augmenting path one at a time. The most efficient maximum cardinality matching algorithms do so in batches. One technique that has been successfully used to achieve that is *scaling* first introduced in [9] and has been applied to among others to weighted bipartite matching in [17] and various geometric problems in [14].

The idea of the scaling approach is to recursively solve the problems for reduced weights, representing i most significant binary digits, and use the returned solution to more efficiently solve the original instance. It has been first

used in [11] to achieve a running time of $O(n^{3/4}m \log N)$. The algorithm works in $O(\log N)$ phases each of which finds the maximum weight perfect matching. Later work from [16] showed that it is enough to find $\pm O(n)$-approximate maximum weight perfect matching provided $O(\log nN)$ scales are executed, each of which runs in $O(m\sqrt{n\alpha(m,n)\log n})$ time. The algorithm of [5] further reduces the requirements by finding near-optimal and near-perfect matching in each scale to achieve a running time of $O(m\sqrt{n}\log nN)$. We take a closer look and describe an implementation of the $O(n^{3/4}\log N)$ algorithm from [11].

2.2.0.1 The algorithm

All the existing scaling algorithms solve the maximum weight perfect matching problem. The problem can be expressed by a linear program similar to the one for the non-complete version. The only difference is the equality constraint which forces each vertex to be matched:

$$\begin{aligned}
\text{(MWPM)} \qquad \text{maximize } & \sum_{e\in E} x_e w(e) \\
\text{subject to } & \sum_{v} x_{uv} = 1 && \text{for each } u \in V \\
& \sum_{u,v\in B} x_{uv} \le \left\lfloor \frac{1}{2} n_B \right\rfloor && \text{for each } B \in \mathcal{O} \\
& x_e \ge 0 && \text{for each } e \in E
\end{aligned}$$

Naturally, the dual program is also very close to the one for MWM. The only difference is that the equality constraint causes the corresponding dual variables y_v to be unrestricted:

$$\begin{aligned}
\text{(}\overline{\text{MWPM}}\text{)} \qquad \text{minimize } & \sum_{v\in V} y_v + \sum_{B} z_B \left\lfloor \frac{1}{2} n_B \right\rfloor \\
\text{subject to } & y_u + y_v + \sum_{u,v\in B} z_B \ge w(uv) && \text{for each } uv \in E \\
& z_B \ge 0 && \text{for each } B \in \mathcal{O}
\end{aligned}$$

We define a function $yz(S)$ for a subset of vertices $S \subseteq V$ as follows:

$$yz(S) = \sum_{v\in S} y_v + \sum_{B\subset S} z_B \left\lfloor \frac{1}{2} n_B \right\rfloor + \sum_{B\supseteq S} z_B \left\lfloor \frac{1}{2} n_S \right\rfloor$$

Notice that $yz(V)$ corresponds to the dual program's objective function and $yz(e) = slack(e)$.

The algorithm works using recursion. Given even weights $w(e)$, we first calculate smaller weights $w'(e) = 2\lfloor w(e)/4 \rfloor$ that remain even. We then perform for w' which we call M', dual weights y and z along with a blossom tree T'. The root of the tree T' is G which is not a blossom and has no weight. We than use the returned values to help us solve the problem for the original weights $w(e)$.

Function *scale*(w)

1. If $w(e) = 0$ for all $e \in E$ then return perfect matching M, dual weights $y_v = 0$ and no blossoms
2. Calculate $w'(e) \leftarrow 2\lfloor w(e)/4 \rfloor$ for all $e \in E$
3. Call *scale*(w') to get dual weights y' and z' and old blossom tree T'
4. Calculate dual weights

$$\begin{aligned} y^0_v &\leftarrow 2y'_v + 1 && \text{for all } v \in V \\ z^0_B &\leftarrow 2z'_B && \text{for all old blossoms } B \in T' \end{aligned}$$

5. Call *match*(y^0, z^0, T') to calculate a perfect matching M
6. Return M along with corresponding dual weights y and z and blossom tree T

The depth of recursion is $O(\log N)$. The formula $y^0_v \leftarrow 2y'_v + 1$ accounts for the bit lost in calculating w'. The weights y^0 and z^0 are feasible and close to optimal, which is show by a following fact:

Theorem 2.2.1. *For any maximum weight perfect matching N:*

$$y^0 z^0(V) \geq w(N) \geq y^0 z^0(V) - n \tag{2.1}$$

Proof. The first equality follows from complementary slackness and the second one from the fact that the duals returned from the recursive call were optimal:

$$\begin{aligned} y^0 z^0(V) &= 2yz(V) + n && \text{from the definition of } y^0 \text{ and } z^0 \\ &= 2w'(M) + n && \text{from the recursive call } yz(V) = w'(M) \\ &= 2 \sum_{e \in M} 2 \left\lfloor \frac{w(e)}{4} \right\rfloor + n && \text{from the definition of } w' \\ &\leq \sum_{e \in M} w(e) + n && \text{from the fact that } 4 \left\lfloor \frac{x}{4} \right\rfloor \leq x \\ &\leq w(N) + n && \text{from the definition of } N \end{aligned}$$

□

The problem is that the edges in blossoms in T' are not tight for the new weights y^0 and z^0. We want to make use of the new weights while getting rid of the old blossoms. We say a blossom has *dissolved* when its dual weight has been reduced to 0. One way to do that is to *distribute* the dual weight z_B of an old blossom B. The distribution consists of decreasing z_B by a value δ while

increasing y_v by $\delta/2$ for all $v \in B$. While the distribution maintains dominance it also increases the dual objective function by $\delta/2$.

The match procedure starts with an empty matching and weights y^0 and z^0 and builds a *current matching* with corresponding blossoms. We work with two types of blossoms. We refer to the ones inherited from the recursive call as *old blossoms* and blossoms created during this scale as *current blossoms*. When calculating the $yz(S)$ function we now take into account both old and current blossoms.

In order to efficiently dissolve old blossoms the algorithm uses heavy path decomposition. Let B be a blossom in a blossom tree T. We call a subblossom C of B its *heavy child* if $n_C > \frac{n_B}{2}$. Any blossom has at most one heavy child. We can partition T into *heavy paths* which are maximal paths in T in which the next blossom on the path is the previous blossom's heavy child. If a blossom is not its parent's heavy child (or it's the root of the tree) it is a root of its own heavy path.

The *match* function works by partitioning the old blossom tree T' into heavy paths and then calling $path(B)$ on roots B of heavy path in postorder.

Additionally, in order to maintain the dominance on the edges outgoing from the currently processed old blossom, the algorithm maintains the following invariant for all vertices $v \in G$.

$$y_v \geq y_v^0 \tag{2.2}$$

2.2.0.2 Path procedure

We first define the notion of a *shell*. Consider a perfect structured matching with blossom B and its descendant C we call the graph induced by $B \setminus C$ a shell.

The procedure $path(B)$ takes as an input a root B of a heavy path P_B. All old blossoms contained in B that aren't a part of P_B have been dissolved prior to calling $path(B)$.

Two consecutive blossoms on P_B form a shell. All the undissolved blossoms on path P_B form a sequence of shells. We also include the deepest blossom C in P_B which corresponds to a shell $C \setminus \emptyset$. This is not a shell as we have previously defined them and as such has different properties.

The procedure dissolves all old blossoms on P_B while building up the current matching and associated blossoms and maintaining the optimality constraints. Over time as the old blossoms are dissolved the shells merge together. One exception is when $B = G$, in which case the top shell $G \setminus C$ never dissolves. The procedure ends when all the old blossoms are dissolved or when $B = G$ and the current matching is perfect.

Function $path(B)$

Repeat until all blossoms in P_B have been dissolved or if $B = G$ when the current matching is perfect:

1. Build a graph G' by shrinking current blossoms and using only tight edges contained within a single shell. Utilizing the current matching M as the

starting point, find the maximum cardinality matching in G' using the MV algorithm. Use the found matching to augment M.

2. Sort the shells by the number of exposed vertices in descending order.

3. Call *shellsearch* on the shells of P_B in the calculated order if they haven't been dissolved or searched already.

To perform path augmentations we make use of a maximum cardinality matching algorithm. In our case it is specifically the Micali-Vazirani algorithm with time complexity $O(\sqrt{n}m)$ introduced in [23] which we will refer to as the MV algorithm. To efficiently contract the blossoms and create a new graph we need some information about the shells and blossoms. For each shell we maintain a list of proper blossom contained within it and a list of vertices. These lists get concatenated as old blossoms dissolve. At the start of the iteration we record for each vertex its current shell and blossom in an array to be able to efficiently check it. We don't delete the structure for the root of the path even after it was dissolved, we just mark it as such. At the end of the *path* procedure, after all the old blossoms have been dissolved it will contain a list of all blossoms and vertices in the path. These lists are than concatenated to the lists of B's parent in the blossom tree, so that it contains the information about current blossoms.

We also need to efficiently calculate a slack of an edge to check whether it's tight. Notice that we only need to check an edge's slack if it's contained in a single shell and connects vertices in different blossoms. If we want to calculate the slack of such an edge uv, we only need to know y_u, y_v, $w(uv)$ and the sum of dual weights of all old blossoms that contain it. For each blossom B on the current path we define $z_{path}(B)$ as a sum of dual weights z_P for all ancestors P of B on the current path including B. Before starting $path(B)$ we also calculate the value z_{outer} which is the sum of dual weights for all old blossoms containing B in the old blossom tree. We do that while performing the heavy path decomposition by summing up the values along the way. Let B be the smallest old blossom one P_B that contains uv. We can calculate the slack of uv as

$$slack(uv) = y_v + y_u - w(uv) + z_{path}(B) + z_{outer}$$

We explicitly build the shrunk graph G' before passing it to the MV algorithm. To do that, we iterate over all blossoms and associate them with consecutive integers. When adding edges, we remember which original edge they come from. The MV algorithm returns the matching as a mapping from a vertex to its mate. We iterate over the returned matching and augment our current matching according to it. If a blossom B was matched to C by the MV algorithm, we first iterate over B's adjacency list in G' to find the original edge through which the two were matched. Just like in the Blossom algorithm, the augmentation is lazy – we only change the bases of B and C (along with some other indicators that describe the matching).

The main part of the procedure consists of calls to the $shellsearch(B \setminus C)$ procedure, which takes as an input a shell and executes a variant of the blossom

algorithm search. We search for an augmenting path over tight edges in the shell defined by two consecutive undissolved old blossoms. We call these two old blossoms the *boundaries* of the searched shell and refer to them as *outer* or *inner* where the outer blossom contains the inner one. During the search some of these old blossoms might get dissolved. When that happens the boundaries of the shell change and the searched graph grows. When searching the innermost shell the inner boundary doesn't exist. Apart from blossom dissolution the high level search algorithm is almost identical.

The main difference is in the dual weight adjustment step. When calculating δ we no longer use δ_1 as the dual variables y_v are unrestricted in the perfect matching version of the dual linear program. We do have to take into the account the weight corresponding to the inner and outer blossom of the shell. When adjusting dual weights by δ we distribute 2δ units for the two boundaries of the shell. The two exceptions are when the inner blossom doesn't exist and when the outer blossom is the whole graph G for which we don't perform any distributions. When calculating δ we make sure that the distribution doesn't decrease the dual weight of a boundary below 0.

After dual weight adjustment the dual weight corresponding to one of the boundaries might be brought down to 0. When that happens that old blossom is dissolved and the boundaries are moved. When the outer blossom is dissolved we move the boundary to its parent in the path P_B and when it's the inner one, we move the boundary to its child. We say that the current shell *dissolves into* the shell defined by the dissolved boundary and the new boundary. If said shell has already been involved in a previous shell search during this iteration, we finish our search. When the outer boundary is dissolves, and it happens to be the outermost non-dissolved blossom in P_B, we also halt our search. If none of these cases happen the search finishes when it finds an augmenting path over tight edges.

The details of implementing the *shellsearch* procedure differ in major ways from the previously described approaches and are described in the next section.

Lemma 2.2.1.1. *The shellsearch procedure maintains the dominance on all edges, tightness on blossom edges and the property 2.2.*

Proof. The dominance and tightness on blossom edges are ensured by the blossom search algorithm.

The distributions on C and D maintain the dominance and tightness on edges contained within $C \setminus D$ as the decrease of z_C cancels out the increase of y.

For edges uv such that $u \in C \setminus D$ and $v \in D$ only dominance needs to be maintained, as they are not part of the matching. The slack of such an edge uv changes by -2δ from the change to z_C, by δ and 2δ from distributions to y_u and y_v and at the worst by $-\delta$ from a dual adjustment to y_u, depending on u's label. This means that the slack of uv doesn't decrease and remains non-negative. □

Similar investigation of dual adjustments shows how the value of the yz function changes with each adjustment:

Lemma 2.2.1.2. *If $C \setminus D$ is not the last shell of B, a dual adjustment of δ decreases $yz(B)$ by $\delta(f-2)$ and $yz(C \setminus D)$ by δf. In the search of the last shell $C \setminus \emptyset$, both $yz(B)$ and $yz(C)$ are decreased by $\delta(f-1)$.*

2.2.0.3 Shell search

The authors of the algorithm leave out the details of the *shellsearch* procedure's implementation except for key data structures used. We've relied on a description presented in [5] with some additions.

To take advantage of integer weights instead of employing tree based priority queues that work in logarithmic time we can make use of an array based queue. It is convenient for us to reframe the dual adjustment in terms of *time*. We call steps of the search algorithm *events*. We say that an event happens at time t if happens after the sum of dual adjustments reaches t. A simple implementation of an array queue stores events in an array of linked lists and supports scheduling events at set times. It stores the current time and a pointer to current event. To retrieve the next event it moves the pointer to the next event in the current list or moves the time counter until the corresponding list is non-empty.

There are 4 types of events that happen during the search:

- $grow(w, e)$ – happens when the blossom B_v containing the vertex v is added to the search structure through edge e. The value of e might be empty, which happens when B_v is exposed and it becomes the root of a new search tree,
- $blossom(e)$ – happens when a new blossom is created after the addition of edge e to the search structure,
- $dissolve(B)$ – happens when an odd blossom B expands after its corresponding dual weight z_B reaches 0,
- $dissolveShell(B)$ – happens when one of the boundaries of the search dissolves after its weight reaches 0.

Another problem we need to solve is maintaining vertices' membership to blossoms. We have previously used concatenable queues which we split and concatenated as blossom were created and expanded. We can do better by taking advantage of the order blossoms change. Notice that once a blossom becomes even it can no longer expand and stays even, it can only become a part of a new even blossom. Only odd blossoms expand after which some of their subblossoms become even, odd and free. A free blossom can either become even or odd as it gets added to the search structure. We can model this behavior using two data structures – one for splitting and one for joining sets of vertices.

The first data structure used is the well known $union-find$, which operates on a universe of elements from $\{0, 1, \ldots, n-1\}$. The elements are divided into sets. Each set additionally has an ID. It supports the following operations:

- $init(u, id)$ – initializes u's set to a singleton $\{u\}$ with the provided ID,

- $join(u, v)$ – joins the sets containing u and v into a new set with the provided ID,
- $find(u)$ – returns the ID of u's set.

The last data structure we need is the $splitfindmin$ data structure also called the splitting list data structure introduced in [11]. In the next section we described in detail its implementation. The splitting list data structure operates on a universe of elements from $\{0, 1, \ldots, n\}$. Every element x can be contained in at most one list $L(x)$ and has an associated cost $c(x)$ which can be infinite. The cost $c(L)$ of a list is the smallest cost of an element in L. Every list has an id. It supports the following operations:

1. $initialize(x_1, \ldots, x_l)$ initializes a list of elements
2. $decreasecost(x, d)$ update $c(x)$ to $\min(d, c(x))$
3. $split(x)$ split $L(X)$ into two lists L_1 and L_2 where L_1 contains all elements of $L(x)$ until and including x and L_2 contains the remaining elements in the order they appear in $L(x)$
4. $findmin(L)$ return $c(L)$ and the element of L which achieves the minimum
5. $findlist(x)$ return x's current list

With these two data structure we can keep track of which blossom each vertex belongs to. Each even blossom will correspond to a set in the $union-find$ data structure. Meanwhile, each non-even blossom will correspond to a list in the $splitfindmin$ data structure. When a blossom is no longer proper $L(v)$ will correspond to the last proper blossom x was a part of before it became even.

When a blossom is labeled as even through a grow step or after becoming a root of a search tree, we iterate over its vertices to join them into a single set with ID pointing to the blossom. After a new blossom with base b is created we iterate over all newly even vertices v and call $join(b, v)$. For all even subblossoms with base c we call $join(b, c)$. After these join operations all vertices of the new blossom are in a single set with ID corresponding to the blossom. It is important for us to change the label of odd subblossoms to even. At the beginning of the search and after the searched graph expands we create lists in the $splitfindmin$ for all blossoms with their vertices in the blossom order. When an odd blossom expands we call $split$ to divide its list into lists corresponding to the new blossoms. In order to find which blossom a vertex v belongs to we first call $findlist(v)$. If the blossom corresponding to the returned list is non-even it is v's blossom. If the blossom is even we need to call $find(v)$. Additionally, we will store costs associated with all non-even vertices which we will describe later.

During the search we maintain the following values:

- t_{now} – the current time, maintained by the array queue

- $z_0(B)$ – the value of z_B at the time it became a proper blossom
- $t_{proper}(B)$ – the time B became a proper blossom
- $t_{even}(B)$ – the time B became an even blossom
- $t_{odd}(B)$ – the time B became an odd blossom
- $\Delta(B)$ – the sum of dual weight adjustment experienced by the vertices of B when they were a part of an odd blossom before B became proper or even
- $y_0(v)$ – the value of y_v before all searches
- $t_{search}(v)$ – the time the vertex v was added to the searched graph
- $\Delta(v)$ – the sum of distributions to vertex v before it was added to a search graph
- t_{outer} – the time the current outer boundary became the boundary
- t_{inner} – the time the current inner boundary became the boundary
- t_{whole} – the time the outer boundary became G – a special case
- $z_{boundary}$ – the sum of weights of old blossom on the path containing the outer boundary (including itself) at the time it became the boundary

For each shell we maintain a list of blossoms contained inside it to allow us to iterate over them. Each proper blossom has a pointer to its position in said list. We remove and add blossoms from said list whenever dissolve and blossom events happen. When a shell dissolves into another we join their lists.

With these values we can calculate the values of y_v and z_B at a time t_{now}. Let

$$D(v) = \Delta(v) + \max(0, \min(t_{now}, t_{whole}) - t_{search}(v))$$

be the sum of old blossom distributions to v since the beginning of the iteration – consisting of distributions before it was added to search graph $\Delta(v)$ and distributions from the outer boundary after v was added to the search graph. Assuming $B_v = findlist(v)$, we can calculate

$$y(v) = y_0(v) + D(v) + \begin{cases} \Delta(B) + t_{now} - t_{odd}(B) & \text{if } B_v \text{ is odd} \\ \Delta(B) - (t_{now} - t_{even}(B)) & \text{if } B_v \text{ is even} \\ \Delta(B) & \text{if } B_v \text{ is free} \end{cases}$$

$$z(B) = z_0(B) + \begin{cases} -2(t_{now} - t_{odd}(B)) & \text{if } B \text{ is odd} \\ 2(t_{now} - t_{even}(B)) & \text{if } B \text{ is even} \\ 0 & \text{if } B \text{ is free} \end{cases}$$

When calculating the slack of an edge in the search graph not contained within a blossom we need to include the weights of old blossoms containing it z_{old}. W

$$z_{old} = z_{outer} + \begin{cases} 0 & \text{if } G \text{ is the boundary} \\ z_{boundary} - 2(t_{now} - t_{outer}) & \text{otherwise} \end{cases}$$
$$slack(uv) = y(u) + y(v) - w(uv) + z_{old}$$

During each iteration, the shells are searched in descending order based on the number of exposed vertices they contain. Distributions from old blossom higher in the path influence the vertices of all lower blossoms. To calculate $\Delta(v)$ we need to know the sum of all distributions from old blossom containing v in previous searches. We use a data structure $addprefixsum$ capable of calculating a prefix sum of an array and adding a value to an element at a specified index. In our case it is implemented as a Fenwick tree [10], which accomplishes both of these operations in $O(\log n)$ for an array of size n.

Before performing any searches we index undissolved shells with consecutive integers $0, 1, \ldots$ which we use as indices in the $addprefixsum$ data structure. We denote the number of all distributions done to B and its ancestors on the current path since the start of the current iteration as $dist(B)$.

The costs in the $splitfindmin$ data structure will help us maintain the slack of edges between even and non-even vertices. We want to keep the invariant that for each non-even vertex v the value $c(v)$ is equal to $\min_{u\text{even}} slack(vu)$ shifted by some offset shared by all vertices of the blossom $B = L(v)$. Specifically, for each non-even vertex u in blossom $B = L(v)$ we maintain the following invariant:

$$\min_{u\text{even}} slack(vu) = \begin{cases} c(v) - (t_{odd}(B) - \Delta(B)) & \text{if } B \text{ is odd} \\ c(v) - (t_{now} - \Delta(B)) & \text{if } B \text{ is free} \end{cases}$$

When v is odd the slack of edges connecting it to even vertices doesn't change with dual adjustments and when v is free it decreases by 1 with each dual adjustment, meaning that the invariant is maintained after dual adjustment without making changes to $c(v)$. We additionally remember which edge is responsible for the current cost $c(v)$.

We show how to update all the counters and data structures throughout the search starting with the beginning of the search.

At the start of the search First we calculate $z_{boundary}$. Let B be the outer boundary. We already know the sum of the dual weight before any searches and it is equal to $z_{path}(B)$. To include the distributions that happened to B's ancestors in any searches that took place since then, we query the $addprefixsum$ data structure and call the result Δ. The value of $z_{boundary}$ is equal to $z_{path}(B) - 2\Delta$.

We iterate over all blossom B in the searched graph to initialize their lists in the $splitfindmin$ data structure and set $\Delta(B) \leftarrow 0$. If B is exposed we schedule a $grow(v, -)$ event at time 0 for the base v of B. For all vertices v we set $t_{search}(v) \leftarrow 0$ and $\Delta(v) \leftarrow \Delta$. For each boundary B we queue up the $dissolveShell(B)$ event at time $z_B/2$ as each dual adjustment decreases the value of z_B by 2. We don't do that if the inner boundary does not exist or the outer boundary is the whole graph G. This is also when we set $t_{whole} = 0$, which is otherwise initialized to ∞.

The $grow(v, e)$ **event** Let B be v's current blossom. If B is not free, it is already in the search structure, and we have nothing else to do. We proceed based on whether e is in the current matching.

If e is not provided or if it's currently part of the matching, B becomes even. We set e to be B's backtrack edge and assign

$$t_{root}(B), t_{even}(B) \leftarrow t_{now}$$

In the $unionfind$ data structure, we link all vertices of B to v, which is the base of B according to our definition of the grow event. For every $B \in u$ we call $schedule(u)$ to queue up event associated with these vertices.

If e is known and it is not matched, B becomes odd and the values become

$$t_{root}(B), t_{odd}(B) \leftarrow t_{now}$$

After which we call $schedule(u)$ where u is the base of B.

The $schedule(u)$ **procedure** This function schedules events associated with a vertex u which is either even or a base of an odd blossom. Let B_u be the blossom containing u.

If u is odd, we schedule the $dissolve(B_u)$ event at time $t_{now} + z(B_u)/2$, which is when z_B reaches 0. As B_u is odd, it's not exposed and as such u is matched to some vertex v belonging to B_v. We can continue the search by scheduling either a $grow(v, uv)$ or $blossom(uv)$ event based on whether v is free or even.

Now assume that u is even. We iterate over all neighbors v of u. Let B_v be v's current blossom. If v is even we schedule an event $blossom(uv)$ at time $t_{now} + slack(uv)/2$ as that is when uv becomes tight. If v is odd, we update the cost $c(v)$ in the $splitfindmin$ by calling $decreasecost(v, slack(uv) + (t_{odd}(B_v) - \Delta(B_v)))$. When v is free, we try to decrease $c(v)$ to $slack(uv) + (t_{now} - \Delta(B_v))$ and if we're successful, we schedule $grow(v, uv)$ at $t_{now} + slack(uv)$ as that is when uv becomes tight and the search tree can be extended to B_v.

The $blossom(e)$ **event** We follow the backtrack edges the same way we did in the blossom algorithm by simultaneously moving two pointers until we either visit the same blossom twice or reach two distinct roots. If we reached different roots, an augmenting path is found and the $shellsearch$ procedure is finished. If not, a new blossom is created.

Let B be the new blossom and $B_1, \ldots, B_k$ its subblossoms. For each subblossom B_i we store its current dual weight which will be its starting weight when it becomes proper by setting $z_0(B_i) \leftarrow z(B_i)$. For previously odd subblossoms B_i we set $t_{even}(B_i) \leftarrow t_{now}$ and update the number of dual adjustment while its vertices were odd $\Delta(B_i) \leftarrow \Delta(B_i) + (t_{now} - t_{odd}(B_i))$. We link all vertices of B_i to its base and the base to the new base of B in the $unionfind$ data structure. For all vertices v of B_i, which become even for the first time, we call $schedule(v)$ after updating all counters and data structures.

The $dissolve(B)$ **event** We split the lists in the $splitfindmin$ data structure so that they correspond to the vertices of B's subblossoms. The subblossoms are label and incorporated into the search tree the same way as we've done in the blossom algorithm search. Let C be a subblossom of B that now becomes proper. We set $t_{proper}(C) \leftarrow t_{now}$ and update the count $\Delta(C) \leftarrow \Delta(B)+(t_{now}-t_{odd}(B))$ to account for the dual adjustments while B was a proper odd blossom. We set t_{odd} or t_{even} to t_{now} if b was labeled accordingly.

If C has become odd, we schedule a $dissolve(b)$ event at $t_{now} + z_0(b)/2$. If it was labeled as even, we link all vertices v of C to its base in the $union - find$ data structure and call $schedule(v)$ to queue event associated with the newly even vertices. We do that at the end for all even vertices after updating counters and data structures for all new proper blossoms. If C has become free we find the edge with the smallest slack connecting a vertex v of C to some even vertex u using the $splitfindmin$ data structure. According to our invariant we can calculate $slack(uv) = c(v) - (t_{now} - \Delta(C))$. With that we can schedule a $grow(v, uv)$ event at time $t_{now} + slack(uv)$.

The $dissolveShell(B)$ **event** Assume that B is the outer boundary. The steps for the inner boundary are analogous except for a few corner cases. We start by recording the distributions to B in the $addprefixsum$ data structure by adding $t_{now} - t_{outer}$ at B's index.

The old blossom B is marked as dissolved, so it can be deleted at the end of the iteration. We check if B is the outermost undissolved old blossom. To be able to do that we maintain a pointer to said blossom. If B matches that pointer, the $shellsearch$ procedure is done. We follow the heavy children along the current path to find the new highest undissolved blossom and update our pointer.

Assume now that B has an undissolved ancestor C in the path. If the shell $C \setminus B$ has already been searched, we finish the search. If it's not the case, we expand the searched graph by adding vertices of $C \setminus B$. For each vertex $v \in C \setminus B$, we set $t_{search}(v) \leftarrow t_{now}$ and $\Delta(v) \leftarrow dist(C)$.

For all blossom D contained in $C \setminus B$ we set $\Delta(D) \leftarrow 0$ and create a corresponding list in the $splitfindmin$ data structure.

Each exposed vertex v of $C \setminus B$ results in a $grow(v, -)$ event that happens at current time and establishes v's blossom as a new root in the search tree.

Next we scan all new edges vu where $v \in B \setminus C$ and u is already a part of

the searched graph. If u is even we schedule a $grow(v, uv)$ event that happens at time $t_{now} + slack(uv)$, which is when uv becomes tight.

If C is not equal to G, we schedule a $dissolveShell(C)$ event at time $t_{now} + z_C/2$. Otherwise, we set $t_{whole} \leftarrow t_{now}$. We set $t_{outer} \leftarrow t_{now}$ and recalculate $z_{boundary}$ as described above.

No matter which of the above cases happens, we append B's list of shell vertices and blossoms to C's. We also remove B from the path by reassigning heavy child/parent pointers.

At the end of the search For each vertex v, we store the current value of y_v by setting $y'(v) \leftarrow y(v)$. This value might no longer be accurate by the end of the iteration, as more distributions take place in subsequent searches. To take that into the account we store the current value $\Delta(v) \leftarrow (\min(t_{now}, t_{whole}) - t_{search}(v))$. When all the searches have finished we can calculate the final value $y_v \leftarrow y'(v) + dist(B) - \Delta(v)$, where B is the lowest old blossom containing v at the start of the iteration.

For each blossom B we delete its corresponding list in the $splitfindmin$ data structure and store the final value $z_B \leftarrow z(B)$. If a blossom's dual weight z_B is equal to 0, we expand it.

We update the dual weights for the current boundaries to reflect the distributions that took place. For the current outer boundary B, we update $z_B \leftarrow z_B - 2(t_{now} - t_{outer})$ and add $t_{now} - t_{outer}$ at B's index in the $addprefixsum$ data structure. We do the same for the inner boundary.

Running time analysis The $splitfindmin$ data structure can perform m $decreasecost$ operations in $O(m\alpha(m, n))$ time as we will show in 2.2.3.

The running time of the array priority queue with k events and maximum time $t_{\max}$ is trivially $O(k + t_{\max})$. It can be shown that a search of shell $C \setminus D$ decreases the value $yz(C \setminus D)$ by $n_c - n_D$ with a proof similar to that of 2.2.4.2. Together with 2.2.1.2 this shows that $t_{\max} = O(n_C - n_D)$.

The time needed to maintain the $addprefixsum$ data structure during a single iteration of $path(B)$ is $O(n_B \log n_B)$, which is enough for the desired time bound. This time is not included in the original analysis [14] suggesting the author had a different implementation in mind. I suspect such an algorithm most likely takes advantage of the fact that a distribution of an old blossom keeps the slack of edges it contains the same while increasing the slack of edges that cross between the inside of the blossom and outside. Each shell is only searched once during an iteration and during a single execution of the $shellsearch$ procedure, we only check the slack of edges within some set of consecutive shells. Instead of calculating how the exact value of y changes as a result of distributions, we can calculate the slack of an edge by checking how much time it spent connecting a vertex of the current graph and one outside it. We can do that using the t_{search} values while also accounting for the case when G is the outer boundary. Personally, I've found my approach using $addprefixsum$ easier to follow.

With that in mind the searches of all shells in a single iteration of $path(B)$ except for the innermost shell $C \setminus \emptyset$ can be done $O(m_B \alpha(m_B, n_B))$.

All that remains is the time needed to search the innermost shell $C \setminus \emptyset$. When it has at least 2 exposed vertices the same array priority queue is sufficient. When that number is equal to one, similar calculations as in 2.2.4.2 show the blossom C dissolves after n_C dual adjustments. As the number of old blossoms can reach $O(n_B)$, it results in quadratic running time in the worst case. When $m = \Omega(n^{5/4})$ this is enough for our desired time bound. In general, it is enough to perform the search of the $C \setminus \emptyset$ shell in $O(\sqrt{n} m \alpha(m, n))$ which can be with search procedures of various Blossom algorithm variants. For simplicity, our implementation omits this case and uses the same simple array priority queue in all searches.

2.2.0.4 Splitting list

We now present an implementation of the previously used splitting list data structure as described by [11]. Remember that the splitting list data structure operates on elements from a universe $\{0, 1, \ldots, n\}$. Every element x can be contained in at most one list $L(x)$ and has an associated cost $c(x)$, which can be infinite. The cost $c(L)$ of a list is the smallest cost of an element in L. It supports following operations:

1. $initialize(x_1, \ldots, x_l)$ initializes a list of elements,
2. $decreasecost(x, d)$ update $c(x)$ to $\min(d, c(x))$,
3. $split(x)$ split $L(X)$ into two lists L_1 and L_2 where L_1 contains all elements of $L(x)$ until and including x and L_2 contains the remaining elements in the order they appear in $L(x)$,
4. $findmin(L)$ return $c(L)$ and the element of L which achieves the minimum,
5. $findlist(x)$ return x's current list.

For our analysis, we define the Ackermann's function $A(i, j)$ along with inverse functions $a(i, n)$ and $\alpha(m, n)$ as follows:

$$
\begin{aligned}
A(i, 0) &= 2 && \text{for } i \geq 1 \\
A(1, j) &= 2^j && \text{for } j \geq 1 \\
A(i, j) &= A(i-1, A(i, j-1)) && \text{for } i \geq 2 \text{ and } j \geq 1 \\
a(i, n) &= \max\{j : 2A(i, j) \leq n\} && \text{for } n \geq 4 \\
\alpha(m, n) &= \min\left\{i : A\left(i, \left\lfloor \frac{m}{n} \right\rfloor\right) \geq n\right\} && \text{for } m \geq n
\end{aligned}
$$

It is handy for us to be able to calculate $A(i, j)$ in constant time. We only need to calculate values below the n. Cases when $j = 0$ or $i = 1$ can be

calculated on the fly by simply returning 2 or calculating 2^j using a bit shift operation. For the remaining cases, we store all values $A(i, j) < 10^9$ as there are only 5 such cases.

The splitting list data structure works recursively. Each list has a set *level* $i \in \{1, 2, \dots\}$. A list L is divided into a *head* and *tail*, where the head stores a starting fragment of L and tail the stores the remaining elements. Either of the fragments can be empty.

Both the head and tail are divided into *superelements*. A superelement of *rank* $j \geq 0$ in a list L of level i consists of $2A(i, j)$ consecutive elements of L. A superelement e of the head has a maximum rank $a(i, n_e)$ where n_e is the number of elements from the beginning of the head to the last element of e. This means that the head consists of superelements of non-decreasing ranks and at most three remaining elements at the start of the head which do not belong to any superelement which we call *singletons*. We call a maximal sequence of superelements of the same rank a *sublist* of L. We say that a sublist containing superelements of rank j is of that rank. A sublist L' of a level i list L is represented by a list of level $i-1$. The tail is partitioned similarly with the exception that the ranks of elements are non-increasing.

Along with the size n of the universe of elements, when initializing the splitting list data structure we provide the chosen level of lists used to store the original elements which we denote as $i_{\max}$. Performing operations on the splitting list data structure consists of interacting with these top level lists, which then refer to their lower level sublists.

For every list we store:

- the lists ID for top level lists
- its level
- the list of all its elements
- its current cost $c(L)$
- two lists of sublists for the head and tail
- two lists of singletons without superelement in the head and tail
- pointer to a sublist if it corresponds to one

For every sublist we store:

- its rank
- pointer to its list
- the splitting list containing its superelements
- pointer to the position in its list's sublist list

In order to index arrays using superelements we associate them with one of their elements (in the head it is their last element and in the tail the first one). We make use of two-dimensional arrays of size $i_{\max} \times n$. For each element x at its corresponding level we store:

- the value $e(x)$ of its superelement if it belongs to one or -1 if x is a singleton
- the cost $c(x)$ of the element – for superelement this is the minimum cost of one of its elements
- the pointer to x's list $L(x)$ if x is a singleton
- a doubly-linked list of all elements comprising the superelement if x corresponds to one

We maintain the value of $c(L)$ so that it is always correct, allowing us to execute $findmin(L)$ in constant time.

We now describe how to implement the splitting list operations. The function $findlist(x)$ for an element of a level i list checks if x is a singleton in which case it has a pointer to its list. If x is part of a superelement we recursively call $findlist(e(x))$ at level $i-1$. The returned list has a pointer to $e(x)$'s sublist which itself points to x's list. Total time taken is $O(i_{\max})$.

The function $decreasecost(x, d)$ works similarly to $findlist(x)$ – it returns the list containing x. If x is a singleton it simply updates $c(x)$ and $c(L(x))$. When x is a part of a superelement $e(x)$ it calls $decreasecost(e(x), d)$ recursively and uses the returned pointer to $L(x)$ to update the costs. The time complexity is again $O(i_{\max})$.

To initialize a list L one of two internal functions can be used – *initializehead* or *initializetail*. The function *initializehead* works by scanning elements from the right to left and dividing them into superelements of maximum rank. These superelements are than divided into sublists which are initialized by recursively calling *initializehead* at the lower level. The scan at one level can be done in linear time. If the list has l elements than all of its sublists have no more than $\frac{l}{4}$ elements which means the total time for $initializehead(L)$ is $O(l)$. The *initializetail* function works similarly with the exception that the scan proceeds from left to right.

The last operation to implement is $split(x)$. If x is a singleton, we first check if it's in the head or tail. Assume x is a head singleton. We split the head singleton list at x and create a new list which consists solely of head singletons up to x. Similarly, if x is a tail singleton, we create a new list with just tail singletons.

Assume now that x is not a singleton in a level i list and belongs to a superelement $e(x)$ in sublist S inside the list L. We perform two splits on S to divide it into three parts: S_1 containing superelements before $e(x)$, S_2 containing elements after $e(x)$ and a sublist containing solely $e(x)$ which we can discard. Assume S is in the head, the case when S is part of the tail is analogous. We

create two new lists L_1 and L_2. The head of L_1 consists of the sublists in L's head before S along with S_1. The tail of L_1 is created by calling *initializetail* on elements of $e(x)$ until x. The tail of L_2 is just the tail of L and its head is initialized with a call to *initializehead* on elements of $e(x)$ after x. We than update the cost of L_1 and L_2 by checking costs of all sublists and singletons. It is easy to see that the new lists are partitioned consistently with the previously described rules.

Theorem 2.2.2. *The time taken to perform all split operations is* $O(na(i,n))$

Proof. First we estimate the time at the top level. A single list has at most $a(i,n)$ sublists as each has a stores superelements of different ranks and $a(i,n)$ is the maximum rank possible. The number of singletons in a list is at most 6. Using the maintained pointers splits on sublist lists, element lists take constant time. Updating pointers in sublists and calculating cost for new lists takes time $a(i,n)$. All that remains is the time spent in *initializehead* and *initializetail* and on splitting lists of elements comprising superelements. We perform the split in linear time by finding x in $e(x)$'s list. The initialization also takes linear time. Each time an element x is in an initialization the rank of its superelement decreases or it becomes a singleton meaning time needed for all initialization calls is $O(na(i,n))$.

We show by induction that time needed for splits on level i lists on a universe of k elements is $O(ka(i,k))$. For $i = 1$ all sublists contain at most one superelement, so every recursive split takes constant time, meaning the total time for all splits is $O(ka(1,k))$.

Suppose $i > 1$. A rank j sublist of a level i list contains at most

$$2A(i,j+1)/2A(i,j) \leq A(i,j+1)$$

as there are fewer than $2A(i,j+1)$ elements remaining at the time of partition as otherwise a rank $j+1$ elements would have been created. The time spent in recursive calls per a level j superelement is

$$\begin{aligned} a(i-1,A(i,j+1)) &= a(i-1,A(i-1,A(i,j))) \\ &= \max\{j : 2A(i-1,j) \leq A(i-1,A(i,j))\} \\ &< A(i,j) \end{aligned}$$

There are $n/2A(i,j)$ rank j superelements at level i, so the total time spent on them is $O(n)$. As the maximum rank of a superelement is $a(i,n)$, the total time complexity is $O(na(i,n))$.

□

Theorem 2.2.3. *Assuming we know the number $m \geq n$ of decreasecost operations on universe of n elements in advance, we can choose the value $i_{\max}$ such that the total running time of the splitting list data structure is $O(m\alpha(m,n))$.*

Proof. Choose $i_{\max} = \alpha(m,n)$. Total time needed is $O(mi_{\max} + na(i_{\max}, n))$. From the definition of α, we know that $A(\alpha(m,n), \lfloor \frac{m}{n} \rfloor) \geq n$. This means that $a(i_{\max}, n) = \max\{j : 2A(\alpha(m,n), j) \leq n\} < \lfloor \frac{m}{n} \rfloor$ and the final complexity is $O(m\alpha(m,n))$. □

Our splitting list data structure implementation differs from the one described here in that it stores an additional value associated with a cost that can be specified during a *decreasecost* operation and is returned in $findmin$ instead of the element. In our case the value corresponds to some edge and the cost is determined by its slack.

Solutions to the $splitfindmin$ problem with better theoretical running time exist. The data structure from [24] works in $O(m \log \alpha(m,n) + n)$ and [26] solves it in $O(m+n)$ for integer costs.

2.2.0.5 Complexity

Theorem 2.2.4. *The time complexity of match is $O(n^{3/4}m)$.*

Proof. We first show the time complexity for $path(B)$. To do that we need a few lemmas with the first one describing a special property of shells.

Lemma 2.2.4.1. *(Shell lemma) Assume we have a complete structured matching with dual weights y and z. Let $B \setminus C$ be a shell.*

The maximum weight perfect matching in a shell $B \setminus C$ exists and has weight $yz(B) - yz(C)$. If C is a subblossom of B then $yz(B) - yz(C) = yz(B \setminus C)$.

Proof. First assume that C is a trivial blossom for vertex v. Construct a graph B' by adding a vertex v' to B connected with an edge vv' with zero weight.

The structured matching implies that there exists a complete matching on $B \setminus v$ which can be obtained by changing B's base to v. We can extend it to B' by adding vv'. Call the resulting matching M'.

We construct weights y' and z' which are identical to y and z with the exception that:

$$y'_{v'} = -y_v$$
$$z'_B = \sum_{B \subseteq D} z_D$$

The new duals are tight and dominating and with M' constitute a complete structured matching on B', meaning that M' is a maximum weight perfect matching on B' with weight $y'z'(B') = yz(B) - y_v$. By removing vv' from M' we obtain a complete matching M on $B \setminus v$ which is a maximum weight perfect matching (otherwise we could find a complete matching on B' which weighs more than M').

In case when C is non-trivial start by choosing a vertex $v \in C$. We can again obtain a complete matching on $B \setminus v$ by using the blossom structure. The resulting matching is also complete on $C \setminus v$ and $B \setminus C$ which can be seen by

examining the matching after changing B's base to v – no matched edges cross between C and $B \setminus C$ and v is not matched to any of vertices in B.

The resulting complete matching on $B \setminus C$ weighs

$$(yz(B) - y_v) - (yz(C) - y_v) = yz(B) - yz(C)$$

A larger matching would give a larger matching on $B - v$.

If C is a child of B then

$$\begin{aligned} yz(B) - yz(C) = &\left(\sum_{v \in B} y_v + \sum_{S \subset B} z_S \left\lfloor \frac{n_S}{2} \right\rfloor + \sum_{S \supseteq B} z_S \left\lfloor \frac{n_B}{2} \right\rfloor \right) - \\ &\left(\sum_{v \in C} y_v + \sum_{S \subset C} z_S \left\lfloor \frac{n_S}{2} \right\rfloor + \sum_{S \supseteq C} z_S \left\lfloor \frac{n_C}{2} \right\rfloor \right) \\ = &\sum_{v \in B \setminus C} y_v + \sum_{S \subset B \setminus C} z_S \lfloor 7 \rfloor \frac{n_S}{2} + z_C \left\lfloor \frac{n_C}{2} \right\rfloor + \\ &\sum_{S \supseteq B \setminus C} z_S \left\lfloor \frac{n_{B \setminus C}}{2} \right\rfloor - z_C \left\lfloor \frac{n_C}{2} \right\rfloor \\ = &yz(B \setminus C) \end{aligned}$$

□

Lemma 2.2.4.2. *During the execution of path(B) the value of $yz(B)$ decreases by at most n_B.*

Proof. If $B = G$, it follows from the fact that the duals are almost optimal as shown by 2.1. Assume now that B is an old blossom. The dual objective never increases according to 2.2.1.1. At the start of *path*(B) the value of $yz(B)$ is at most $y^0 z^0(B)$. It's sufficient for us to show that at the end of *path*(B) the dual weight satisfy:

$$yz(B) \geq y^0 z^0(B) - n_B$$

We choose a vertex $v \in B$. Let M^* be a maximum weight perfect matching on the shell $B \setminus v$. We get

$$w(M^*) \geq y^0 z^0(B) - y_v^0 - n_B$$

which follows from the 2.2.4.1 and reasoning analogous to proof of 2.1. By constructing dual weights similarly to the proof of 2.2.4.1 it can be shown that

$$yz(B) - y_v \geq w(M^*)$$

Using the fact that $y_v \geq y_v^0$ we get

$$yz(B) - y_v \geq w(M^*) \geq y^0 z^0(B) - y_v^0 - n_B \geq y^0 z^0(B) - y_v - n_B$$

By adding y_v to both sides we obtain the desired inequality. □

Lemma 2.2.4.3. *For any $\varepsilon > 0$ the number of iterations in path(B) with at least n_B^ε exposed vertices after the first step is $O(n_B^{1-\varepsilon})$.*

Proof. Assume B is a blossom. Let f be the number of exposed vertices after the first step of an iteration. We call a shell *small* when it has at most 2 exposed vertices and *big* otherwise. We consider an iteration with $f \geq n_B^\varepsilon$.

If at least $\frac{1}{2}f$ exposed vertices are in big shells. There are no augmenting paths in any of the shells after the first step. This means that any search that finds an augmenting path performs at least one dual adjustment. This adjustment decreases the dual weight of all exposed vertices. A shell might not have the dual weights of its exposed vertices adjusted if another shell dissolved into after the same dual adjustment that found an augmenting path. Because we search the shell in the order of decreasing number of exposed vertices, such a shell has to be adjacent to a shell with at least as many exposed vertices that has been found to contain an augmenting path. This means that at least one third of exposed vertices in big shells had their dual weights adjusted by at least 1. The big shells where dual adjustments took place contain at least $\frac{1}{6}f$ exposed vertices. By 2.2.1.2, the function $yz(B)$ decreases by at least $\frac{1}{12}f \geq \frac{1}{12}n_B^e$. By 2.2.4.2, the number of such iterations is at most $O(n_B^{1-e})$.

In the remaining case, at lest $\frac{1}{2}f$ exposed vertices are in small shells – those with at most 2 exposed vertices. Assume that we're not considering the first iteration. Any shell that had exposed vertices at the start of the iteration had to contain an augmenting path by the stopping condition of *shellsearch*. The procedure ends when an augmenting path is found or the shell dissolves either into another one that has already been searched or the outer boundary is the outermost undissolved old blossom and the shell ceases to exist. This means that all the shells with exposed vertices had their number of exposed vertices decreased by at least 2. There are no less than $\frac{1}{4}f$ such shells. This means that at least $\frac{1}{2}f$ were matched in the first step. Meaning the number of exposed vertices was multiplied by a number less than $\frac{2}{3}$. This means that the number of such iterations is $O(\log n_B)$. □

We can now show how much time is needed for *path*(B).

Lemma 2.2.4.4. *The time complexity of path(B) is $O(n_B^{3/4}m_B)$.*

Proof. The algorithm executes $O(\sqrt{n_B})$ iterations. From the 2.2.4.3 there are $O(\sqrt{n_B})$ iterations with at least $\sqrt{n_B}$ exposed vertices. Since each iteration matches at least one edge there are $O(\sqrt{n_B})$ iterations with fewer than $\sqrt{n_B}$ exposed vertices. As we've discussed before, all executions of the *shellsearch* procedure can be accomplished in $O(m_B\alpha(m_B, n_B))$ per iteration except the last shell, which itself can be dissolved in desired time.

We now focus on the augmentation step. The contraction of the graph and augmentation can be done in $O(n_B + m_B)$ the way we've described. According to 2.2.4.3 there are $O(n_B^{1/4})$ iterations with at least $n_B^{3/4}$ exposed vertices. With the MV algorithm's running time of $O(\sqrt{n}m)$ this means that we spend $O(n_B^{3/4}m_B)$ time in these iterations. In the iterations with less than $n_B^{3/4}$ exposed

vertices less than $n_B^{3/4}$ new vertices are matched. We make use of the fact that the MV algorithm works in phases each of which takes $O(m)$ time and matches at least 1 new vertex. By providing a partial matching consistent with current matching, we can complete augmentation in these cases in $O(n_B^{3/4} m_B)$. □

Knowing that the execution of $path(B)$ takes $O(n_B^{3/4} m_B)$ time we show that the *match* procedure runs in $O(n^{3/4}m)$ time.

Let i be a non-negative integer. Consider the set $\mathcal{B}_i$ of the roots B of major paths in T' such that $\frac{n}{2^{i-1}} > n_B \geq \frac{n}{2^i}$.

There are no two roots C and D in $\mathcal{B}_i$ such that D is a descendant of C. Assume otherwise and let P_C be the heavy path starting in C. D has to be a descendant of a non-heavy child of some blossom C' in P_C so $n_D \leq \frac{n_{C'}}{2} \leq \frac{n_C}{2} < \frac{n}{2^i}$ which contradicts the definition of $\mathcal{B}_i$.

This means that any vertex v belongs to at most one blossom in $\mathcal{B}_i$ and any edge is contained in at most one of such blossoms so $\sum_{B \in \mathcal{B}_i} m_B \leq m$. We sum up the time spent in $path(B)$ for all $B \in \mathcal{B}_i$:

$$\sum_{B \in \mathcal{B}_i} n_B^{3/4} m_B < \sum_{B \in \mathcal{B}_i} \frac{n^{3/4}}{2^{3(i-1)/4}} m_B = \frac{n^{3/4}}{2^{3(i-1)/4}} \sum_{B \in \mathcal{B}_i} m_B \leq \frac{n^{3/4} m}{2^{3(i-1)/4}}$$

By summing over i we obtain the final complexity $O(n^{3/4}m)$.

□

Chapter 3

Computational results

All the algorithms described in the previous chapter have been implemented as part of the KOALA library [27] built on top of NetworKit(TODO cite).

The implementations are contained in following files:

- `include/matching/MaximumMatching.hpp` contains declarations of classes corresponding to maximum matching algorithms, namely:
  - `MaximumWeightMatching` – the base class for MWM and MWPM algorithms,
  - `BlossomMaximumMatching` – an abstract base class for implementations of the blossom algorithm. Responsible for the common operations shared by all implementations – building the search tree, creating blossoms, augmenting paths, expanding blossom and others. It relies on call virtual methods implemented by children classes to perform implementation-specific parts of the algorithm. One such part is calculating the δ_i values during dual weight adjustment. Another one is finding useful edges to perform the next step. After each step of the algorithm, specific virtual methods are called to allow the implementations to update their specific data structures.
  - `EdmondsMaximumMatching` – the Edmond's original $O(n^m)$ version of the blossom algorithm [6],
  - `GabowMaximumMatching` – the Gabow's $O(n^3)$ implementation of the blossom algorithm [18],
  - `GalilMicaliGabowMaximumMatching` – the $O(nm \log n)$ implementation of the blossom algorithm of [19],
  - `GabowScalingMatching` – the $O(n^{3/4} m \log N)$ scaling algorithm [14],
  - `MaximumCardinalityMatching` – base class for MCM algorithms,
  - `MicaliVaziraniMatching` – the class for the implementation of the Micali-Vazirani maximum cardinality matching algorithm [23].

- `include/matching/structures` directory contains the implementations of various data structures utilized across the implemented algorithms:
  - `ArrayPriorityQueue.hpp` – contains the implementation of a simple array priority queue used by the scaling algorithm,
  - `ConcatenableQueue.hpp` – contains the implementation of concatenable queues,
  - `FenwickTree.hpp` – contains an implementation of the $addprefixsum$ data strucutre based the Fenwick trees,
  - `HeapWithRemove.hpp` – contains a simple implementation of an array based heap with ability to remove elements, used by the pq_1 priority queue implementation,
  - `PriorityQueues.hpp` – contains the implementations `PriorityQueue1` and `PriorityQueue2` of the pq_1 and pq_2 priority queues,
  - `SplitFindMin.hpp` – contains the splitting list implementation of the $splitfindmin$ data structure,
  - `UnionFind.hpp` – contains the implementation of the $unionfind$ data structure.
- `cpp/matching` contains the implementation of the classes defined in the `MaximumMatching.hpp` header file,
- `test/testMaximumMatching.cpp` contains simple unit tests for the implemented algorithms,
- `benchmark/benchmarkMaximumMatching.cpp` contains the program used to benchmark the various algorithms on provided graphs with various options.

All tests were conducted on a personal computer equipped with an Intel Core i7-6700HQ processor running at a clock speed of 2.60 GHz, using the Ubuntu 22.04.4 LTS operating system.

We've made use of test generators provided as part of the 1st DIMACS implementation challenge [4].

Random instances

The first class of graphs we've tested are random sparse and dense graphs generated using the `random.c` generator from the DIMACS challenge written by C. McGeoch. The program generates a random graph with a provided number of nodes and edges with weights uniformly chosen from a specified range $[1, N]$. It is also possible to specify a seed used to generate random numbers. For all of our test we've set the maximum cost at 2^{16}. For each pair of values n and m we've generated 3 different graphs using random seeds and then average the running time.

Our set of sparse graphs consists of graphs with varying values of n and $m = an$ where $a \in \{2, 4, 8\}$. The results are presented in table 2. We use the letter E to refer to our implementation of Edmonds' algorithm, G to Gabow's blossom algorithm, M to Galil, Micali and Gabow's algorithm and S to Gabow's scaling algorithm. We will refer to them using those letters. As our set of dense graphs we've chosen random graphs with $n \in \{1000, 2000, 3000, 4000\}$ and containing approximately 20%, 40% and 60% possible edges with results presented in table 3. In both cases the scaling algorithm performs the best. Among our implementations of the blossom algorithm, M runs fastest for some smaller test, but gets overtaken by E for larger instances. The G implementation is consistently the slowest for sparse graphs and on par with M for dense instances.

We plot the running time of tested algorithms in relation to n for sparse graphs with $m = 4n$ and dense graphs containing roughly 20% of possible edges. For sparse graphs, the running time of blossom algorithm implementation is roughly quadratic, indicating that that the G and E variant perform better than their theoretical worst case complexity would suggest. Meanwhile, the running time of the scaling algorithm S seems to align with its $n^{7/4}$ complexity. For dense instances, gauging the asymptotic running time seems to be harder with the blossom implementations taking roughly n^3 time.

Perfect matching

We repeat the same tests, this time running the perfect matching versions of our implementations. The tests are generated similarly as in the previous section, with the difference that we make sure that the generated instances contain perfect matchings by running our implementation of the Micali-Vazirani maximum cardinality algorithm. The approach was infeasible for some larger sparse instances, in which case we've used our own random test generator that ensures that the generated graph contains a perfect matching by first generating a random permutation of vertices $[1, n]$ $v_1, v_2, \ldots v_n$ and adding edges $v_1v_2, v_3v_4, \ldots, v_{n-1}v_n$, with the rest of edges chosen randomly.

We've tested our blossom algorithm implementation with both the empty and greedy initialization strategies. The results using the greedy initialization are marked with a G subscript. Additionally, all algorithms are marked with *

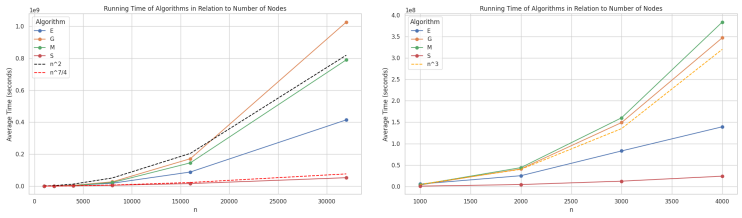

Figure 3: Running time of tested algorithms on sparse and dense random instances.

n	m	E	G	M	S
1000	2000	0.15	0.17	**0.11**	0.20
1000	4000	0.28	0.25	**0.19**	0.25
1000	8000	0.57	0.39	0.33	**0.29**
2000	4000	0.59	0.76	**0.43**	0.44
2000	8000	1.17	1.22	0.77	**0.59**
2000	16000	2.14	1.75	1.37	**0.69**
4000	8000	2.39	3.33	1.86	**1.29**
4000	16000	4.54	5.35	3.47	**1.72**
4000	32000	8.61	8.15	6.56	**2.73**
8000	16000	9.48	14.91	11.64	**4.05**
8000	32000	18.30	27.09	22.72	**5.41**
8000	64000	34.24	50.38	41.95	**8.77**
16000	32000	43.03	93.19	73.50	**11.72**
16000	64000	88.41	171.60	145.75	**16.00**
16000	128000	149.67	294.83	264.03	**33.65**
32000	64000	232.85	569.88	410.78	**44.63**
32000	128000	414.49	1026.40	790.05	**52.79**
32000	256000	611.80	1652.81	1355.59	**125.09**

Table 2: Results on random sparse graphs

n	m	E	G	M	S
1000	100000	5.65	3.51	4.08	**0.82**
1000	200000	7.35	7.26	6.87	**1.62**
1000	300000	8.04	10.26	10.32	**2.26**
2000	400000	25.09	40.97	43.96	**4.47**
2000	800000	31.16	60.80	55.41	**8.14**
2000	1200000	40.67	87.33	87.49	**12.77**
3000	900000	82.91	149.13	160.21	**12.36**
3000	1800000	84.38	205.11	214.92	**17.59**
3000	2700000	102.76	286.73	278.40	**50.79**
4000	1600000	139.25	346.93	383.64	**23.99**
4000	3200000	151.17	505.38	503.95	**38.57**
4000	4800000	233.07	700.78	775.19	**99.25**

Table 3: Results on random dense graphs

to indicate that the perfect matching variant is used.

The results of our experiments on random sparse and dense graphs containing perfect matchings are presented in table 4 and table 5 respectively. As in the case of non-perfect matchings, the scaling algorithm S comes out on top. The margin is even larger most likely to the fact it is designed to find maximum perfect matching and finding a non-perfect matching requires a reduction that doubles the number of nodes and edges. For all blossom algorithm implementations, the greedy initialization resulted in a shorter running time. The speedup ranged from around 30% for the E algorithm and as much as 2-4 times for the G and M implementations.

The influence of edge weight ranges

The previous tests have neglected to account for one of the main contributing factors in the running time of the scaling algorithm - the maximum edge weight N. To test how different values of N affect the running time of the S algorithm, we've conducted a series of tests on random sparse graphs with $m = 4n$ and values of N ranging from 1 to 10^7. We've compared both the perfect and non-perfect variant of S to the E algorithm.

The results can be found in table 6. We can observe that the running time of the blossom algorithm implementation E is influenced slightly by the value of N but seems to level off at higher values of N. Intuitively, we speculate that the initial increase in weight range causes a rise in complexity, but as the weights get larger, especially compared to the number of edges, the higher weight range doesn't meaningfully increase the complexity. As we expect from its time complexity, the running time of the S algorithm seems to scale with $\log N$, which can be seen in figure 4.

DIMACS instances

We next turn to more structured instances. The tests were generated using the following programs:

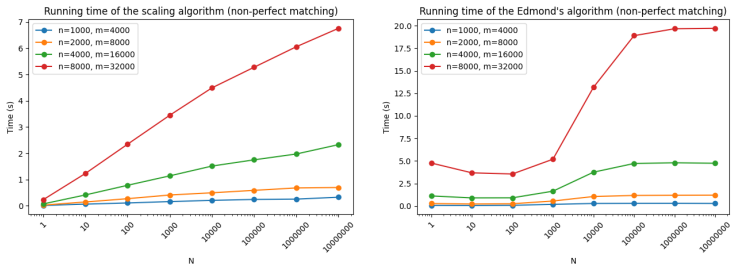

Figure 4: Running time of the scaling algorithm and the Edmond's algorithm in relation to the maximum edge weight N.

n	m	E^*	E^*_G	G^*	G^*_G	M^*	M^*_G	S^*
1000	4000	0.32	0.22	0.27	0.13	0.19	**0.08**	0.13
1000	8000	0.54	0.36	0.36	0.16	0.33	**0.13**	**0.13**
2000	8000	1.36	0.90	1.25	0.65	0.80	0.33	**0.29**
2000	16000	2.11	1.43	1.63	0.74	1.35	0.51	**0.30**
4000	16000	5.52	3.91	5.66	3.07	3.55	1.38	**0.73**
4000	32000	9.15	5.91	7.75	3.42	6.61	2.45	**0.93**
8000	32000	21.34	15.33	29.52	13.81	22.47	8.85	**2.24**
8000	64000	35.71	23.37	48.09	16.29	42.31	15.19	**3.31**
16000	64000	100.73	74.28	184.69	87.47	147.05	54.72	**7.66**
16000	128000	156.35	106.55	298.43	107.97	266.19	93.96	**11.38**
32000	128000	512.00	354.36	1166.83	594.43	767.65	283.84	**26.70**
32000	256000	707.58	542.16	1655.12	727.57	1381.63	528.16	**21.80**

Table 4: Results on random sparse graphs when looking for a perfect matching

n	m	E^*	E_G^*	G^*	G_G^*	M^*	M_G^*	S^*
1000	100000	4.79	3.75	3.54	0.74	4.21	1.27	**0.41**
1000	200000	6.49	4.47	7.11	1.67	6.60	2.11	**0.70**
1000	300000	8.14	5.68	10.98	2.72	10.29	3.32	**0.89**
2000	400000	28.21	22.42	41.09	10.21	44.07	13.51	**1.93**
2000	800000	33.37	26.92	62.93	16.37	56.31	18.27	**2.83**
2000	1200000	40.55	30.74	86.72	22.38	84.48	26.57	**4.20**
4000	1600000	134.92	110.14	347.76	87.46	389.26	114.10	**10.89**
4000	3200000	157.18	111.22	494.36	127.89	504.29	147.19	**16.26**
4000	4800000	218.07	143.22	718.58	180.23	786.43	227.36	**32.61**

Table 5: Results on random dense graphs when looking for a perfect matching

n	m	N	E	S	E^*_G	S^*
1000	4000	1	0.07	**0.01**	**0.01**	**0.01**
1000	4000	10	0.06	0.07	**0.03**	0.04
1000	4000	100	0.08	0.11	0.07	**0.06**
1000	4000	1000	0.19	0.16	0.17	**0.09**
1000	4000	10000	0.28	0.21	0.23	**0.11**
1000	4000	100000	0.29	0.24	0.24	**0.12**
1000	4000	1000000	0.30	0.26	0.29	**0.15**
1000	4000	10000000	0.29	0.33	0.22	**0.16**
2000	8000	1	0.27	0.03	**0.02**	**0.02**
2000	8000	10	0.23	0.15	0.11	**0.09**
2000	8000	100	0.25	0.27	0.22	**0.15**
2000	8000	1000	0.55	0.41	0.54	**0.24**
2000	8000	10000	1.05	0.50	0.85	**0.25**
2000	8000	100000	1.17	0.59	0.90	**0.29**
2000	8000	1000000	1.19	0.68	1.02	**0.33**
2000	8000	10000000	1.20	0.70	1.06	**0.36**
4000	16000	1	1.12	0.07	0.09	**0.04**
4000	16000	10	0.90	0.42	0.44	**0.26**
4000	16000	100	0.91	0.78	0.68	**0.36**
4000	16000	1000	1.64	1.14	1.65	**0.50**
4000	16000	10000	3.75	1.51	3.17	**0.63**
4000	16000	100000	4.72	1.75	3.96	**0.75**
4000	16000	1000000	4.79	1.97	4.04	**0.83**
4000	16000	10000000	4.74	2.33	3.82	**0.87**
8000	32000	1	4.76	0.24	0.36	**0.12**
8000	32000	10	3.68	1.23	1.90	**0.73**
8000	32000	100	3.55	2.35	2.67	**1.07**
8000	32000	1000	5.19	3.45	6.93	**1.58**
8000	32000	10000	13.16	4.49	13.62	**1.70**
8000	32000	100000	18.89	5.28	14.74	**2.05**
8000	32000	1000000	19.66	6.06	16.81	**2.36**
8000	32000	10000000	19.70	6.76	16.62	**2.75**

Table 6: Results on random graphs with varying maximum weights for both perfect and non-perfect variants

k	n	m	E	G	M	S
100	300	399	**0.00**	0.01	0.01	**0.00**
200	600	799	0.02	0.02	0.03	**0.00**
400	1200	1599	0.06	0.10	0.09	**0.01**
1000	3000	3999	0.35	0.60	0.50	**0.04**
2000	6000	7999	1.38	2.40	2.31	**0.12**
4000	12000	15999	6.01	10.38	14.08	**0.39**
k	n	m	E^*_G	G^*_G	M^*_G	S^*_G
100	300	399	**0.00**	**0.00**	**0.00**	**0.00**
200	600	799	**0.00**	**0.00**	0.01	**0.00**
400	1200	1599	**0.01**	0.02	0.04	**0.01**
1000	3000	3999	0.05	0.10	0.16	**0.02**
2000	6000	7999	0.19	0.40	0.74	**0.06**
4000	12000	15999	0.74	1.76	4.29	**0.20**

Table 7: Results on the `t.f` DIMACS instances with varying k

- `t.f` written by N. Ritchey and B. Mattingly. For a parameter k it generates a sequence of k triangles. Consecutive triangles are connected with a single edge. The configuration tends to generate a lot of blossoms according to the comments provided with the generators.
- `tt.f` written by N. Ritchey and B. Mattingly. Similar to `t.f` with the difference that each pair of consecutive triangles is connected with 3 separate edges.
- `hardcard.f` written by B. Mattingly. Generates graphs that have been shown in [12] to be hard for Edmond's maximum cardinality matching algorithm.

All of these instances contain perfect matching, so we've run both variants of the algorithms. For perfect matching we've opted to only test with greedy initialization. We ran each algorithm 3 times and average the time spent.

The results for `t.f` are presented in table 7. Once again, S is by far the fastest. Of the blossom algorithm implementations E is the clear winner with G and M following. The results are similar for `tt.f` with the exception that the greedy initialization tends to find the perfect matching immediately.

In the case of `hardcard.f`, as observed in table 8, the S algorithm once again comes out on top. What is unusual is that G outperforms both E and M, while in most of the previous tests it was consistently the slowest.

k	n	m	E	G	M	S
50	300	20000	0.11	0.08	0.10	**0.02**
100	600	80000	0.90	0.60	0.75	**0.06**
200	1200	320000	8.60	5.56	7.32	**0.24**
400	2400	1280000	68.91	44.53	63.66	**0.97**
800	4800	5120000	529.36	351.33	595.68	**4.08**
k	n	m	E^*_G	G^*_G	M^*_G	S^*
50	300	20000	0.03	**0.01**	0.04	**0.01**
100	600	80000	0.21	0.07	0.25	**0.03**
200	1200	320000	2.55	0.69	2.56	**0.13**
400	2400	1280000	19.67	5.43	23.45	**0.55**
800	4800	5120000	139.87	41.25	244.50	**2.28**

Table 8: Results on the `hardcard.f` DIMACS instances with varying k

Bibliography

[1] Alfred V Aho and John E Hopcroft. *The design and analysis of computer algorithms*. Pearson Education India, 1974.

[2] Nicos Christofides. "Worst-case analysis of a new heuristic for the travelling salesman problem". In: *Operations Research Forum*. Vol. 3. 1. Springer. 2022, p. 20.

[3] Marek Cygan, Harold N Gabow, and Piotr Sankowski. "Algorithmic applications of baur-strassen's theorem: Shortest cycles, diameter, and matchings". In: *Journal of the ACM (JACM)* 62.4 (2015), pp. 1–30.

[4] *DIMACS :: Implementation Challenges — dimacs.rutgers.edu*. `http://dimacs.rutgers.edu/programs/challenge/`. [Accessed 29-08-2024].

[5] Ran Duan, Seth Pettie, and Hsin-Hao Su. "Scaling algorithms for weighted matching in general graphs". In: *ACM Transactions on Algorithms (TALG)* 14.1 (2018), pp. 1–35.

[6] Jack Edmonds. "Maximum matching and a polyhedron with 0, 1-vertices". In: *Journal of research of the National Bureau of Standards B* 69.125-130 (1965), pp. 55–56.

[7] Jack Edmonds. "Paths, trees, and flowers". In: *Canadian Journal of mathematics* 17 (1965), pp. 449–467.

[8] Jack Edmonds and Ellis L Johnson. "Matching, Euler tours and the Chinese postman". In: *Mathematical programming* 5 (1973), pp. 88–124.

[9] Jack Edmonds and Richard M Karp. "Theoretical improvements in algorithmic efficiency for network flow problems". In: *Journal of the ACM (JACM)* 19.2 (1972), pp. 248–264.

[10] Peter M. Fenwick. "A new data structure for cumulative frequency tables". In: *Software: Practice and Experience* 24 (1994), pp. 327–336. URL: `https://api.semanticscholar.org/CorpusID:7519761`.

[11] Harold N Gabow. "A scaling algorithm for weighted matching on general graphs". In: *26th Annual Symposium on Foundations of Computer Science (sfcs 1985)*. IEEE. 1985, pp. 90–100.

[12] Harold N Gabow. "An efficient implementation of Edmonds' algorithm for maximum matching on graphs". In: *Journal of the ACM (JACM)* 23.2 (1976), pp. 221–234.

[13] Harold N Gabow. "Data structures for weighted matching and nearest common ancestors with linking". In: *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*. 1990, pp. 434–443.

[14] Harold N Gabow, Jon Louis Bentley, and Robert E Tarjan. "Scaling and related techniques for geometry problems". In: *Proceedings of the sixteenth annual ACM symposium on Theory of computing*. 1984, pp. 135–143.

[15] Harold N Gabow, Zvi Galil, and Thomas H Spencer. "Efficient implementation of graph algorithms using contraction". In: *25th Annual Symposium onFoundations of Computer Science, 1984.* IEEE. 1984, pp. 347–357.

[16] Harold N Gabow and Robert E Tarjan. "Faster scaling algorithms for general graph matching problems". In: *Journal of the ACM (JACM)* 38.4 (1991), pp. 815–853.

[17] Harold N Gabow and Robert E Tarjan. "Faster scaling algorithms for network problems". In: *SIAM Journal on Computing* 18.5 (1989), pp. 1013–1036.

[18] Harold Neil Gabow. *Implementation of algorithms for maximum matching on nonbipartite graphs.* Stanford University, 1974.

[19] Zvi Galil, Silvio Micali, and Harold Gabow. "An O(EV\logV) algorithm for finding a maximal weighted matching in general graphs". In: *SIAM Journal on Computing* 15.1 (1986), pp. 120–130.

[20] Frank Hadlock. "Finding a maximum cut of a planar graph in polynomial time". In: *SIAM Journal on Computing* 4.3 (1975), pp. 221–225.

[21] Frank L Hitchcock. "The distribution of a product from several sources to numerous localities". In: *Journal of mathematics and physics* 20.1-4 (1941), pp. 224–230.

[22] Eugene L Lawler. *Combinatorial optimization: networks and matroids*. Courier Corporation, 2001.

[23] Silvio Micali and Vijay V Vazirani. "An O (sqrt(V)E) algorithm for finding maximum matching in general graphs". In: *21st Annual Symposium on Foundations of Computer Science (sfcs 1980)*. IEEE. 1980, pp. 17–27.

[24] Seth Pettie. "Sensitivity analysis of minimum spanning trees in sub-inverse-Ackermann time". In: *International Symposium on Algorithms and Computation*. Springer. 2005, pp. 964–973.

[25] Robert L Thorndike. "The problem of classification of personnel". In: *Psychometrika* 15.3 (1950), pp. 215–235.

[26] Mikkel Thorup. "Equivalence between priority queues and sorting". In: *Journal of the ACM (JACM)* 54.6 (2007), 28–es.

[27] Krzysztof Turowski. *KOALA NetworKit*. 2024. URL: `https://github.com/krzysztof-turowski/koala-networkit`.