Experimental Evaluation of Blum's Maximum Matching Algorithm in General Graphs

Ahmad Dandeh[1,*], Tamás Lukovszki[1]

[1]Eötvös Loránd University (ELTE), Budapest, Hungary

Abstract

We describe an implementation and experimental evaluation of Blum's maximum matching algorithm in general graphs. Blum's algorithm finds augmenting paths in general graphs without explicitly analyzing blossoms. Although there are many implementations and performance studies of Edmonds' blossom algorithm and its variants, we are not aware of any implementation of Blum's approach. We compare three implementations: Blossom I, Blossom V, and Blum's modified depth-first search (MDFS). We extend Blum's algorithm with a preprocessing step that computes an initial random matching. This significantly reduces the number of augmenting path searches. We prepared a Java implementation of MDFS. We describe how to handle some cases that are not fully discussed in Blum's article. We created a unified experimental framework for testing the algorithms on random graphs, geometric graphs, complete graphs, DIMACS benchmark graphs, and overlapping cycles. Our experiments show that Blossom V is always faster than Blossom I and very efficient on all inputs. Blum's algorithm with preprocessing competes with Blossom V in several classes of graphs and outperforms it in some cases, which confirms the resilience of MDFS if properly initialized. These results suggest that preprocessing makes reachability-based algorithms a good alternative to blossom algorithms.

Keywords

Maximum Matching, Graph Algorithms, Algorithm Engineering, Performance Evaluation

1. Introduction

The problem of computing a maximum cardinality matching in general undirected graphs is a fundamental problem in graph theory and combinatorial optimization. A matching is a set of edges such that no two edges share a common vertex. A matching is maximum if it contains the largest possible number of such edges. The theoretical foundation for modern algorithms is Berge's theorem [1], which states that a matching in a graph is maximum if and only if there exists no augmenting path relative to that matching. Augmenting paths are sequences of edges of the graph, which alternate between edges in the matching and edges not in the matching, such that the first and last edges are not in the matching.

In bipartite graphs, augmenting paths can be found by a BFS or DFS like method, leading to efficient algorithms such as the Hopcroft–Karp algorithm for maximum matching in $O(\sqrt{n}m)$ time [2], where n and m are the number of vertices and edges, respectively. For non-bipartite (general) graphs, odd cycles pose a problem. Augmenting paths can overlap or intersect themselves, and need to be dealt with carefully to achieve correctness.

This challenge was overcome by Edmonds [3], who introduced the blossom shrinking technique to manage odd-length cycles and preserve augmenting path structure. His algorithm was the first to solve the maximum matching problem in general graphs in polynomial time. The original algorithm has a worst-case complexity of $O(n^3)$.

This bound is maintained in initial implementations like Blossom I [4]. More recently, Blossom V introduced implementation-level enhancements by Kolmogorov [5], including the use of priority queues, an auxiliary graph for managing alternating trees, and a variable dual update strategy [6]. These techniques yield significant practical speedups, particularly on large or structured inputs. However,

ICTCS 2025: Italian Conference on Theoretical Computer Science, September 10–12, 2025, Pescara, Italy

*Corresponding author.

dandehahmad@inf.elte.hu (A. Dandeh); lukovszki@inf.elte.hu (T. Lukovszki)

0009-0000-3217-648X (A. Dandeh); 0000-0001-8878-3246 (T. Lukovszki)

due to the flexibility of the dual update logic, the worst-case time complexity of Blossom V is estimated to be $O(n^2m)$ [5].

Blum suggested an effective alternative solution using graph transformation and reachability [7, 8]. Blum's MDFS algorithm avoids blossom contraction by transforming the matching problem into a reachability problem in an altered, directed bipartite graph. MDFS can be implemented in $O(n+m)$ time [8], and the maximum matching in $O(n(n+m))$ complexity time, but it is conceptually easier to comprehend and implement, as it eliminates blossom detection and shrinking. Despite this, MDFS has received limited attention in practical settings, and we are not aware of any implementations.

In this paper, we present an experimental evaluation of three algorithms for maximum cardinality matching:

- Blossom I, with a public domain Java implementation; [9]
- Blossom V, using the JGraphT library; [10]
- MDFS, derived from our implementation and optimization of Blum's algorithm.

We compare their performance on a range of graph instances and analyze the effect of providing initial matching. We also identify some cases that are not completely covered by the MDFS in [8]. We describe the necessary modifications that handle these cases correctly.
Our key contributions are the following:

- A unified experimental framework in Java for comparing Blossom I, Blossom V, and MDFS on unweighted graphs.
- An MDFS algorithm implementation of Blum's algorithm with practical robustness enhancements for specific cases that are not completely covered in [8].
- An empirical assessment with correctness behavior, and running time across a broad spectrum of graph types.

The remainder of this paper is structured as follows. Section 2 provides background and necessary algorithmic concepts, and our extension of MDFS. Section 3 presents the experimental setup and comparative results. Section 4 concludes with a discussion and future work.

2. Background: Blum's Algorithm

We use publicly available Java implementations of Blossom I and Blossom V to contrast classical blossom-based algorithms in this paper. For Blum's algorithm, we used the reformulation in [8], which framed the maximum matching problem as a reachability problem in an appropriately transformed directed bipartite graph. We call this method Modified Depth-First Search (MDFS), and it does not perform actual blossom contraction but instead searches for strongly simple augmenting paths. While Blum's paper includes a conceptual algorithm and suggestions for implementation, our experience in coding revealed some edge cases where the behavior is ambiguous or leads to the wrong traversal.

The remainder of this section gives the basic definitions used in MDFS, and then describes in detail the specific cases we discovered and how we corrected them.

2.1. Terminology and Algorithm Overview

Let $G=(V,E)$ be an undirected, and unweighted graph where V is the set of vertices and E is the set of edges $E \subseteq \{\{u,v\} \mid u,v \in V,\ u \neq v\}$. A bipartite graph is a graph whose vertex set can be partitioned into two disjoint sets U and W, and whose edges join vertices of U to those of W. A matching $M \subseteq E$ is a set of edges such that no two edges in M share a common vertex. A vertex is matched if it is incident on an edge in M; otherwise, it is free.

An augmenting path is an elementary path in G such that it starts and ends at free vertices and keeps alternating between non-matching edges and matching edges.

Following Berge's theorem [1], a matching M is maximum if and only if there is no augmenting path concerning M. Moreover, whenever an augmenting path is found, its edges can be reversed (the matched edges become unmatched and vice versa), and a new matching of one larger size than M is found.

Blum's algorithm formulates the maximum matching problem as a reachability problem in an induced directed bipartite graph G', constructed from the given undirected graph $G = (V, E)$, and a given matching $M \subseteq E$.

For each vertex $v \in V$, two vertices v_B and v_A are constructed, effectively making a copy of the set of vertices and generating a bipartition $B \cup A$. The key concept utilized in this formulation is that of the strongly simple path, a directed path in G' with the following conditions:

1. Has no pair of vertices v_B, v_A (having the same original vertex v).
2. Is simple (no repeated vertices).
3. Follows edge direction rules expressing the existing matching status.

For each undirected edge $\{u, v\} \in E$, the transformation adds directed edges to G' depending on whether the edge is part of the matching M:

- If $\{u, v\} \notin M$, two forward edges are added: $u_B \to v_A$ and $v_B \to u_A$. These represent steps along unmatched edges and are directed from B to A.
- If $\{u, v\} \in M$, two backward edges are added: $u_A \to v_B$ and $v_A \to u_B$. These represent matched edges and are directed from A to B.

This construction ensures that traversing the $A \to B$ edges corresponds to traversing the existing matching, and that the $B \to A$ edges represent candidate moves toward establishing new matches. A Modified Depth-First Search (MDFS) on G' is started from a source vertex s connecting all free vertices of side B, and attempts to find a strongly simple path to a sink vertex t connecting all free vertices of side A. Upon finding such a path, it translates to an alternating path in the original graph G that can be applied to augment the existing matching. In each augmentation that succeeds, the size of the matching is incremented by one [8]. The MDFS procedure uses strict rules on labeling to avoid visiting both v_B, v_A for any $v \in V$, thus maintaining correctness throughout the search.

In practice, the edges traversed by MDFS can be classified into five categories: tree edges, forward edges, cross edges, back edges, and weak back edges [8]. These categories determine how the search progresses and how labeling and path constraints are maintained during execution. Blum's formulation provides a structured decision process for classifying edges [8]. MDFS constructs the MDFS-Tree T. For the construction, a stack K is used that contains the vertices from the root of T to the currently visited vertex. The top element of the stack, denoted $TOP(K)$, guides the exploration. At each step, the algorithm inspects an edge of the form $(TOP(K), w_X)$, where $X \in \{A, B\}$, that has not yet been examined. Assuming an edge $(v_X, w_{\overline{X}})$ is under consideration, where the overline denotes the opposite label, i.e., $v_{\overline{A}} = v_B$ and $v_{\overline{B}} = v_A$, the edge cases are summarized as follows [8]:

Case 1: Tree edge – $X = A$ and $(v, w) \in M$

Case 2: $X = B$ and $(v, w) \in E \setminus M$

- **2.1:** Back edge; $w_A \in K$
- **2.2:** $w_A \notin K, w_B \in K$
  - **i.** weak back edge; w_A has been in K previously
  - **ii.** weak back edge; w_A has not been in K previously
- **2.3:** $w_A \notin K, w_B \notin K$
  - **i.** Forward or cross edge; w_A has been in K previously
  - **ii.** Tree edge; w_A has not been in K previously

While implementing the algorithm, whenever a vertex u_B is popped from the stack, the algorithm looks for reachable vertices v_A from where it can further build a valid strongly simple path. These vertices are then stored in the set:

$$\begin{aligned} L_{v_A} := \{u_A \in V' \mid \exists \text{ path } P = (v_A, Q, u_A) \ \wedge\ u_B \notin Q \\ \wedge\ \text{PUSH}(u_A) \text{ has never been performed} \\ \wedge\ \text{POP}(u_B) \text{ has been performed } \} \end{aligned}$$

According to Blum's [8], a vertex is pushed onto the stack in three situations: Case 1, Case 2.3.ii, and Case 2.3.i when $L_{v_A} = u_A$. In the latter case, the vertex at the top of the stack is updated by introducing an extensible vertex $TOP(K)_{[v_A]}$, and we push u_A onto the stack. Consequently, we obtain an extensible edge $(TOP(K), u_A)_{[v_A]}$.

We also rely on the following data structures introduced by Blum [8]:

$$R_{u_A} := \{v_B \in V' \mid (v_B, u_A) \text{ is a weak back edge}\}$$

$$E_{q_A} := \{v_B \in V' \mid (v_B, q_A) \text{ is a cross, forward, or back edge}\}$$

$$D_{q_A} := \{p_A \in V' \mid L_{p_A} = q_A \text{ previously}\}$$

We define the expanded MDFS-tree T_{exp} as the tree obtained from the constructed MDFS-tree T by adding all forward, back, cross, and weak back edges, together with every extensible edge. To determine the vertices in L when u_B is popped, a backward search is performed on T_{exp}. This search is carried out using a standard graph traversal, such as depth-first search, starting from vertex u_A and exploring the considered edges in reverse until u_B is reached.

To support the reconstruction of a strongly simple path when reaching t, the algorithm maintains a variable P_{v_A} that records the most recent non-tree edge terminating at vertex v_A.

2.2. Observed Cases and Modifications

While implementing Blum's MDFS algorithm [8], we identified that certain cases – particularly 2.2.i and 2.3.i when $L_{w_A} = \emptyset$ – were not handled robustly in the original formulation. Although such cases are rare in typical graphs, ignoring them may cause the algorithm to skip essential edge relationships, fail to update sets, or even lose parts of valid augmenting paths. As a result, the algorithm risks violating the constraint of a strongly simple path or failing to detect an augmenting path when one exists. To ensure correctness and stability across all graph instances, we introduced selective modifications in the traversal mechanism. The examples below illustrate the specific problems we encountered and the adaptations we employed to rectify them.

2.2.1. Case 2.2.i, Weak back edge

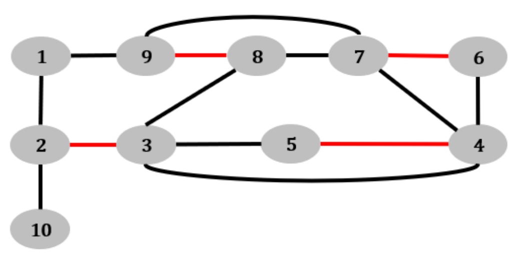
When the traversal encounters a weak back edge (v_B, u_A) – classified as Case 2.2.i – the original algorithm does not perform any action. Specifically, it does not update D_{u_A} and P_{w_A}, such that there is a path $P = w_A, Q, u_A$ and $u_B \notin Q$ has been found by the MDFS, even though the edge may play a structurally important role in the reconstruction of a strongly simple path.

Updating the value of P_{w_A} is essential for correctly reconstructing the augmenting path from s to t. Without this update, the algorithm may terminate prematurely or fail to identify an existing augmenting path, violating its correctness guarantee in these specific configurations.

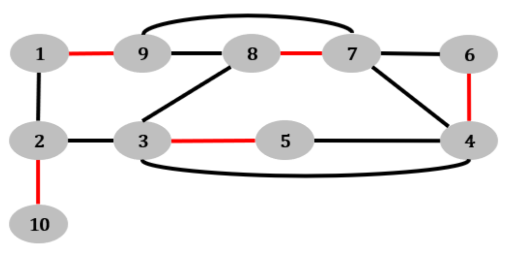
To solve this issue, whenever the algorithm encounters a weak back edge (v_B, u_A), we add v_B to R_{u_A}. This operation ensures that structurally significant relationships are preserved for future use. In order to improve the correctness and efficiency of this operation, we selectively choose which vertices are added based on the traversal history.

Specifically, we allow v_B to be added to R_{u_A} under either of the following two situations:

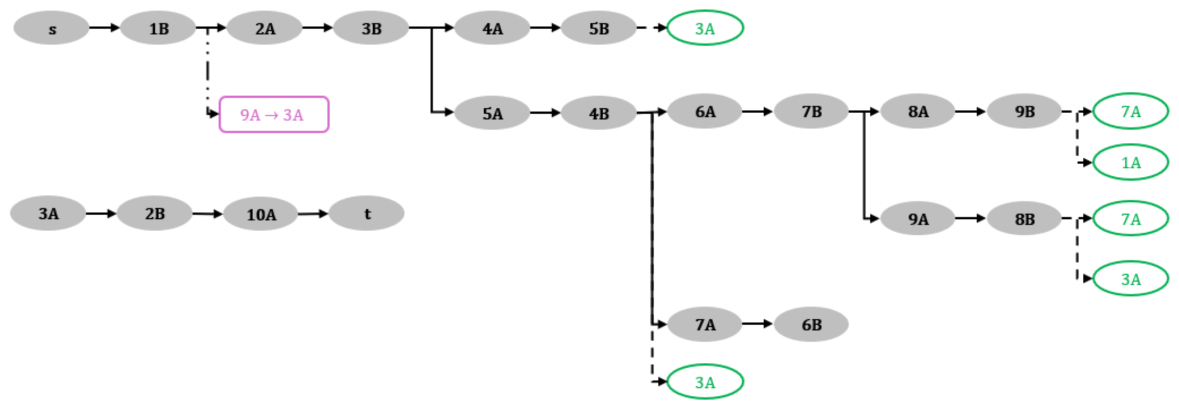
- v_A has not yet been visited by MDFS
- w_B was pushed onto the stack before v_A



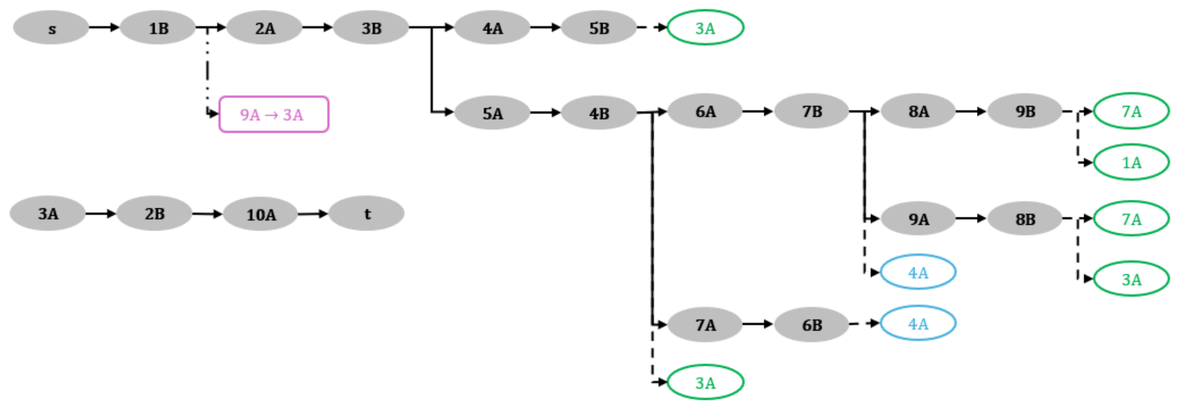
(a) Graph G with current matching (shown in red)



(b) Graph G with new matching after successful augmentation (shown in red)



(c) Tree T_{exp} built by Blum's MDFS. Edges to green vertices represent weak back edges from Case 2.2.ii, edges to purple vertices represent a forward or cross edge (Case 2.3.i) used to connect an extensible vertex via an extensible edge



(d) Corrected tree T_{exp} with our fix. Edges to green vertices represent weak back edges from Case 2.2.ii, edges to blue vertices correspond to Case 2.2.i, edges to purple vertices represent a forward or cross edge (Case 2.3.i) used to connect an extensible vertex via an extensible edge

Figure 1: Illustration of Case 2.2.1: Weak back edge. Our modification recovers a valid augmenting path missed by Blum's original MDFS.

After the execution of $POP(u_B)$, there is a backward traversal of the current expanded tree T_{exp} from the vertex u_A along edges in the opposite direction. Whenever a vertex w_A is visited through this traversal, it is added to the set D_{u_A}, but no value is given to L_{w_A}.

The behavior of this case is illustrated in Figure 1. The original graph G, along with its current matching, is presented in Figure 1a.

The tree T_{exp} constructed by Blum's original MDFS is presented in Figure 1c, When t is found then $P_{8_A} = (9_B, 7_A), P_{9_A} = (8_B, 7_A), P_{4_A} = (5_B, 3_A), P_{5_A} = (4_B, 3_A), P_{6_A} = (8_B, 3_A)$. To reconstruct the strongly simple path from s to t, we start with the last vertex t and follow the parents in T_{exp}. We obtain the path segment $3_A \to 2_B \to 10_A \to t$.

At this point, the parent of 3_A in T_{exp} is an extensible vertex $1_{B\ [9_A]}$, meaning it was added via Case 2.3.i, using an extensible edge $(1_B, 3_A)_{[9_A]}$. Now we have $P_{9_A} = (8_B, 7_A)$. So we reconstruct the path from 9_A to 8_B: $9_A \to 8_B$, next we search for P_{7_A}. Since P_{7_A} is undefined, the reconstruction process fails at this point.

In contrast, our modified version of the algorithm builds the tree T_{exp} shown in Figure 1d. First we pop 7_B, and then we get $P_{8_A} = (9_B,\ 7_A)$, $P_{9_A} = (8_B,\ 7_A)$, next we pop 4_B, and then we get $P_{6_A} = (7_B,\ 4_A)$, $P_{7_A} = (6_B,\ 4_A)$, next we pop 3_B, and we get $P_{4_A} = (5_B,\ 3_A)$, $P_{5_A} = (4_B,\ 3_A)$, after that we find t. The reconstruction of the strongly simple path from s to t starts with the last vertex t, and it follows the parents in T_{exp}. It obtains the path segment $3_A \to 2_B \to 10_A \to t$.

At this point, the parent of 3_A in T_{exp} is an extensible vertex $1_{B\ [9_A]}$, meaning it was added via Case 2.3.i, using an extensible edge $(1_B, 3_A)_{[9_A]}$. Now we have $P_{9_A} = (8_B, 7_A)$. We reconstruct the path from 9_A to 8_B: $9_A \to 8_B$. Then we have $P_{7_A} = (6_B, 4_A)$, so we reconstruct path from 7_A to 6_B: $7_A \to 6_B$. Then we have $P_{4_A} = (5_B, 3_A)$, so we reconstruct path from 4_A to 5_B: $4_A \to 5_B$. Then we get 3_A, so we continue from 1_B: $s \to 1_B$. The strongly simple path will be:

$$s \to 1_B \to 9_A \to 8_B \to 7_A \to 6_B \to 4_A \to 5_B \to 3_A \to 2_B \to 10_A \to t$$

The outcome is shown in Figure 1b, where the algorithm correctly finds an augmenting path and increases the cardinality of the matching by one. The example confirms the need to handle weak back edges differently to ensure MDFS correctness for any case.

2.2.2. Case 2.3.i, cross or forward edge

When the traversal encounters a forward edge (v_B, u_A) of Case 2.3.i with $L_{u_A} = \emptyset$, the original algorithm does nothing. This may result in the absence of valid path segments and, therefore, incomplete augmentation or reconstruction.

To address this issue, the algorithm records each forward or cross edge when $L_{u_A} = \emptyset$ and adds it to a new set WC:

$$WC_{u_A} := \{\, v_B \in V' \mid (v_B, u_A) \text{ is a forward or cross edge and } L_{u_A} = \emptyset \,\}$$

To improve the efficiency of this operation, we include v_B in WC_{u_A} subject to the condition that either of the following holds in the extended tree:

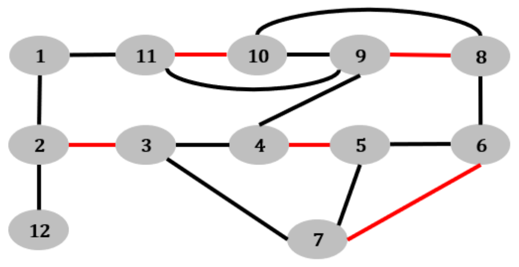
- u_B is a parent of v_A
- u_A and v_A are not on the same path

The behavior of this case is illustrated in Figure 2. The original graph G, along with its current matching, is presented in Figure 2a.

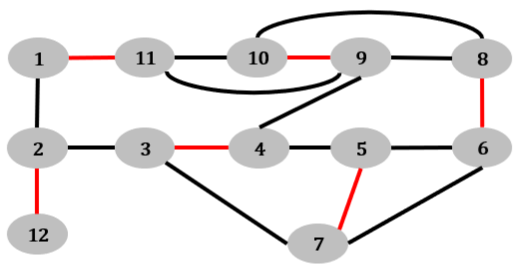
The tree T_{exp} constructed by Blum's original MDFS is presented in Figure 2c, When t is found, then $P_{10_A} = (11_B,\ 9_A)$, $P_{11_A} = (10_B,\ 9_A)$, $P_{6_A} = (7_B,\ 5_A)$, $P_{7_A} = (6_B,\ 5_A)$, $P_{5_A} = (4_B,\ 3_A)$, $P_{4_A} = (7_B,\ 3_A)$, $P_{8_A} = (9_B,\ 4_A)$. To reconstruct the strongly simple path from s to t, we start with the last vertex t and follow the parents in T_{exp}. We obtain the path segment $3_A \to 2_B \to 12_A \to t$.

At this point, the parent of 3_A is an extensible vertex $1_{B\ [11_A]}$, meaning it was added via Case 2.3.i, using an extensible edge $(1_B, 3_A)_{[11_A]}$. Now we have $P_{11_A} = (10_B, 9_A)$, so we reconstruct path from 11_A to 10_B: $11_A \to 10_B$, next we search for P_{9_A}. Since P_{9_A} is undefined, the reconstruction process fails at this point.

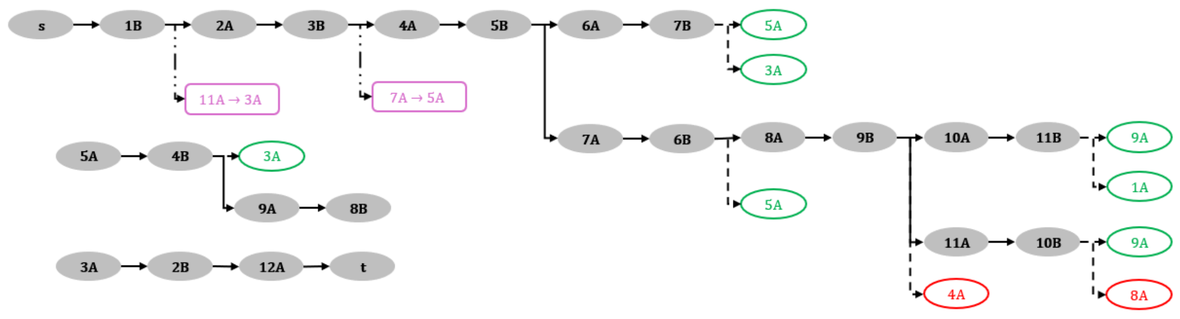
In contrast, our modified version of the algorithm builds T_{exp} as shown in Figure 2d. First we pop 9_B, and then we get $P_{10_A} = (11_B,\ 9_A)$, $P_{11_A} = (10_B,\ 9_A)$. Next we pop 5_B, and then we get $P_{6_A} = (7_B,\ 5_A)$, $P_{7_A} = (6_B,\ 5_A)$. Next we pop 3_B, and we get $P_{4_A} = (7_B,\ 3_A)$, $P_{5_A} = (4_B,\ 3_A)$, $P_{8_A} = (9_B,\ 4_A)$, $P_{9_A} = (8_B,\ 6_A)$. After that, we find t. The reconstruction of the strongly simple



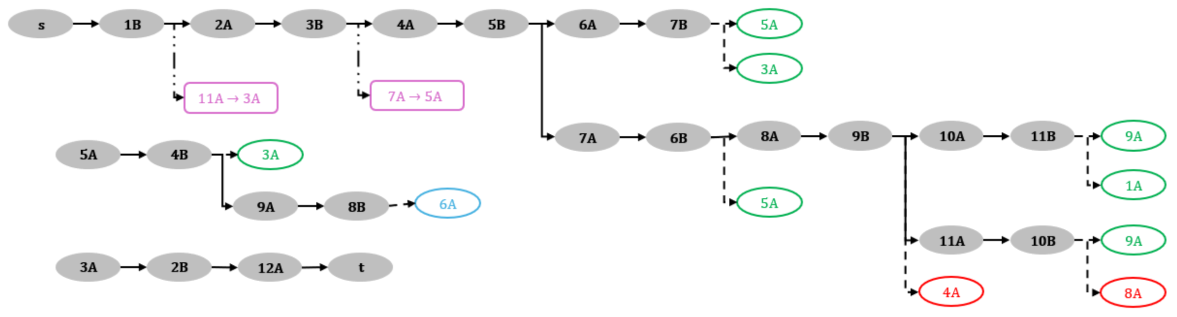
(a) Graph G with current matching (shown in red)



(b) Graph G with new matching after successful augmentation (shown in red)



(c) Tree T_{exp} built by Blum's MDFS, edges to green vertices represent weak back edges, edges to red vertices represent back, cross, or forward edges, edges to purple vertices represent a forward or cross edge (Case 2.3.i) used to connect an extensible vertex via an extensible edge.



(d) Corrected tree T_{exp} with our fix. Edges to green vertices represent weak back edges, edges to red vertices represent back, cross, or forward edges. Edges to blue vertices correspond to Case 2.3.i when $L_{u_A} = \emptyset$. Edges to purple vertices represent a forward or cross edge (Case 2.3.i) used to connect an extensible vertex via an extensible edge

Figure 2: Illustration of Case 2.3.1: Forward or cross edge when $L_{u_A} = \emptyset$. Our modification recovers a valid augmenting path missed by Blum's original MDFS.

path from s to t starts with the last vertex t and follows the parents in T_{exp}. It obtains the path segment $3_A \rightarrow 2_B \rightarrow 12_A \rightarrow t$.

At this point,the parent of 3_A is an extensible vertex $1_B\,_{[11_A]}$, meaning it was added via Case 2.3.i, using an extensible edge $(1_B, 3_A)_{[11_A]}$. Now we have $P_{11_A} = (10_B, 9_A)$, so we reconstruct path from 11_A to 10_B: $11_A \rightarrow 10_B$, then we have $P_{9_A} = (8_B,\ 6_A)$, so we reconstruct path from 9_A to 8_B: $9_A \rightarrow 8_B$. Then we have $P_{6_A} = (7_B,\ 5_A)$, so we reconstruct path from 6_A to 7_B: $6_A \rightarrow 7_B$. Next we have $P_{5_A} = (4_B,\ 3_A)$, so we reconstruct path from 5_A to 4_B: $5_A \rightarrow 4_B$. Then we get 3_A, so we continue from 1_B: $s \rightarrow 1_B$ Finally, the strongly simple path will be:

$$s \rightarrow 1_B \rightarrow 11_A \rightarrow 10_B \rightarrow 9_A \rightarrow 8_B \rightarrow 6_A \rightarrow 7_B \rightarrow 5_A \rightarrow 4_B \rightarrow 3_A \rightarrow 2_B \rightarrow 12_A \rightarrow t$$

The outcome is shown in Figure 2b, where the algorithm correctly finds an augmenting path and increases the cardinality of the matching by one. The example confirms the need to handle forward or cross edges differently when $L_{w_A} = \emptyset$ to ensure MDFS correctness for any case.

3. Experimental Results

In this section, we present a comparative analysis of the performance of three maximum matching algorithms: Blossom I, Blossom V, and Blum's MDFS algorithm. Our experiments are divided into two stages. In the first stage, we compare Blossom I and Blossom V using a diverse set of unweighted, undirected graphs. This comparison highlights the effect of algorithmic optimizations and internal data structure choices, such as queue-based event handling, on runtime performance.

In the second stage, we evaluate Blum's MDFS algorithm against Blossom V. We test two versions of Blum's algorithm: one running from scratch, and the other initialized with a random greedy matching (denoted as Blum Preparatory), which processes the edges in a random order and adds the edge to the matching iff both end vertices are unmatched. This variation allows us to assess how providing an initial matching affects the performance and convergence speed of MDFS. The initial matching is applied only to MDFS, as Blossom V already employs a greedy initialization [5]. We then compare the results across both variants and analyze how well MDFS performs relative to Blossom V in terms of total runtime under different graph topologies. In all of our experiments, all algorithms – Blossom I, Blossom V, Blum, and Blum Preparatory – consistently produced maximum matchings of the same size.

All the experiments were performed on a computer with an Intel(R) Core(TM) i7-7700HQ CPU (2.80GHz), 16GB RAM, under Windows 10 (64-bit). The code was written in Java and executed under the NetBeans 20.0 integrated development environment.

The Java runtime environment was Java 21.0.2 (LTS), provided by Oracle Corporation, under the Java HotSpot™ 64-Bit Server VM (version 21.0.2+13-LTS-58). Run times were gathered with System.currentTimeMillis(), and we only counted the time taken in the core matching function. The construction of the bipartite graph and the initialization of the random matching were included in the time measurements, while I/O accesses and graph construction were excluded.

We evaluated the algorithms on eight distinct types of problem instances to ensure diversity in both structure and complexity. Several of these instance generators incorporate randomness; in such cases, we report the average runtime over t independent runs using different random seeds. The value of t is indicated in the figure captions. All graphs are described by their number of vertices n, which are shown along the horizontal axes of the runtime plots.
Delaunay triangulations: We generated n points at random with a uniform distribution in a $2^{20} \times 2^{20}$ square and then computed the Delaunay triangulation of the point set. For constructing Delaunay triangulations, we employed a Java procedure that conformed to the key ideas and geometric ideas of Shewchuk's 2D Delaunay triangulation algorithm [11].
Erdős-Rényi Random Graphs: We generated random graphs with n vertices and m edges by randomly selecting pairs of distinct vertices. We examined two cases $m = 6n$ and $m = 80n$.
Complete Graphs: We also included full graphs in our set of benchmarks, where every graph has all the possible edges between n vertices. The number of edges in those graphs is $m = \frac{n(n-1)}{2}$.
DIMACS Benchmark Graphs: We used families of examples from the First DIMACS Implementation Challenge [12], which are widely employed to test the performance of matching algorithms in challenging conditions. We used particularly the following three generators available in the benchmark package:

- hardcard (hardcard.f): These graphs were first examined by Gabow and are proven to be challenging for Edmonds-type algorithms; they are constructed to enforce worst-case situations in blossom shrinking.
- T (t.f) and TT (tt.f): These generators create sequences of one-connected and tri-connected triangles, respectively (see [5]).

Each instance is parameterized by an integer K, which determines the size and complexity of the generated graph.

Overlapping Cycles: For testing the resilience of the algorithms, we built our own generator that generates graphs composed of overlapping cycles of odd and even size. These instances are specifically designed in a way that would challenge matching algorithms to their limits by inserting complex structures such as nested and intersecting cycles. The presence of crossing odd-length cycles is hardest for blossom-based methods, while even-length cycles test the algorithm to detect and follow alternative paths. These are structurally dense test cases that put a strain on the limits of contraction, labeling, and path-reconstruction logic and reveal a lot about the correctness and performance of the algorithm in adversarial-like instances.

3.1. Comparison of Blossom I and Blossom V

We compared Blossom I and Blossom V's run-time behavior on several graph families, namely, Delaunay triangulations, random graphs, complete graphs, overlapping cycles, and instances of the DIMACS benchmarks. In all cases, Blossom V behaves uniformly better than Blossom I.

Figure 3 summarizes the results, where the running time is plotted on a logarithmic scale to clearly illustrate performance differences of the algorithms. Blossom I exhibits explosive growth in running time with graph size, particularly for dense graphs. In contrast, Blossom V has reliable and efficient performance since it utilizes priority queues, optimized blossom management, and improved data structures.

3.2. Comparison of Blossom V, Blum, and Blum Preparatory

This section compares the performance of three algorithms: Blossom V, Blum's MDFS, and Blum Preparatory, a variant of MDFS that starts with an initial matching. Our goal is to assess both the raw performance and the practical implications of Blum's avoidance of blossom shrinking, especially when combined with a random greedy initial matching. Figure 4 summarizes the results, where the running time is plotted on a logarithmic scale. Figure 4a shows the results for Delaunay triangulations. Here, the standard Blum code is consistently slower than both competitors. Nevertheless, Blum Preparatory performs much better and often outperforms Blossom V on instances of mid-to-large size. This result shows the strength of MDFS when combined with a precomputed matching.

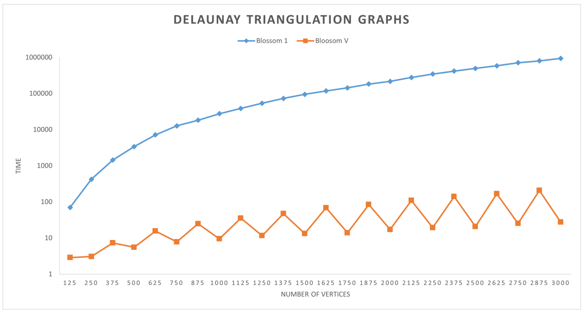
On random graphs with $m = 6n$ (Figure 4b), Blossom V is always the fastest algorithm. While in random graphs with $m = 80n$ (Figure 4c), all algorithms perform reasonably well, but the preparatory version of Blum consistently tracks or slightly outperforms Blossom V, particularly as n increases. This suggests that the combination of simplicity and a warm start is effective.

The results on complete graphs (Figure 4d) are more predictable. Blum Preparatory handles the complete graphs better overall.

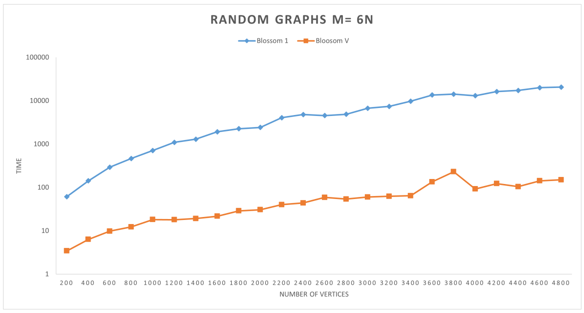
The results on overlapping cycle instances are shown in Figure 4e. Here, the performance gap between Blum and Blum Preparatory is significant, demonstrating the burden of full search in blossom-heavy graphs. Blum Preparatory is very competitive with Blossom V.

Figures 4f to 4h show the results on DIMACS graphs. As sample cases, these are chosen to be challenging, with wild blossom production. These instances also represent worst-case scenarios for MDFS due to the existence of numerous alternative paths. Blossom V performs best on the hardcard graph instances, and Blum Preparatory is the fastest on the T and TT instances.

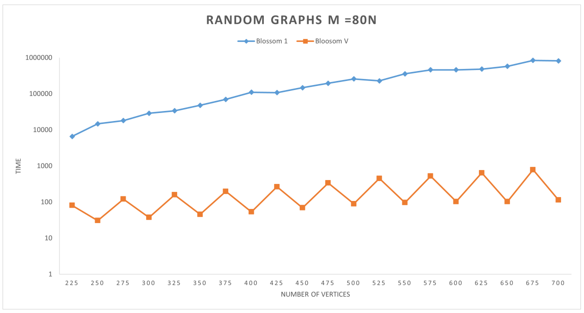
Summary: Blossom V has the best overall runtime for the most graph families. Blum Preparatory is the fastest on complete graphs, T, and TT instances of the DIMACS graphs. It is highly competitive in many structured graphs. It does not require blossom shrinking to beat or nearly match Blossom V. These results suggest that, in real-world scenarios where an approximation for a matching is known or is inexpensively computable, MDFS-based algorithms represent a good and clean alternative.



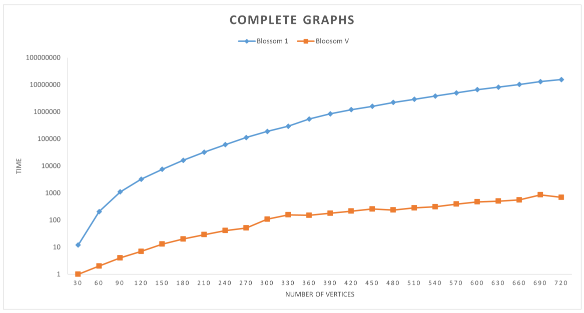
(a) Delaunay triangulation graphs ($t = 50$)



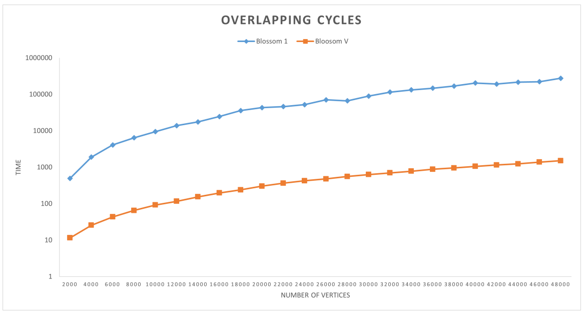
(b) Random graphs ($t = 50$ and $m = 6n$)



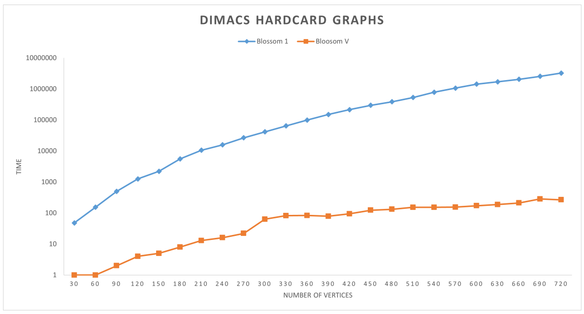
(c) Random graphs ($t = 50$ and $m = 80n$)



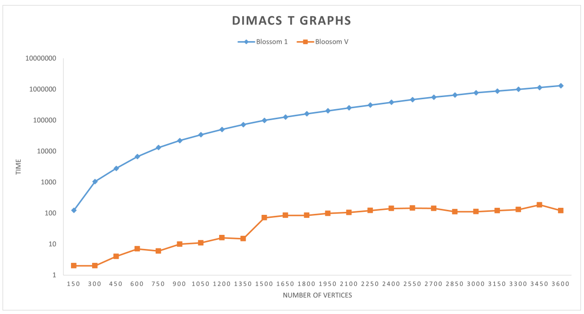
(d) Complete graphs



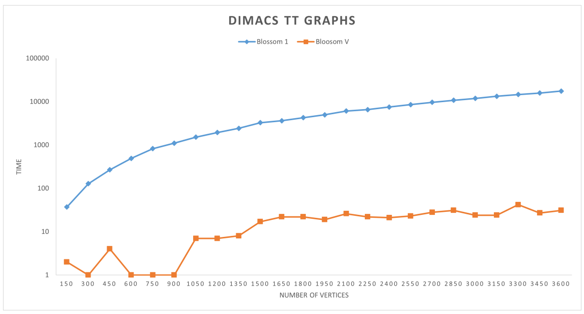
(e) Overlapping cycles ($t = 50$)



(f) DIMACS hardcard graphs



(g) DIMACS T graphs



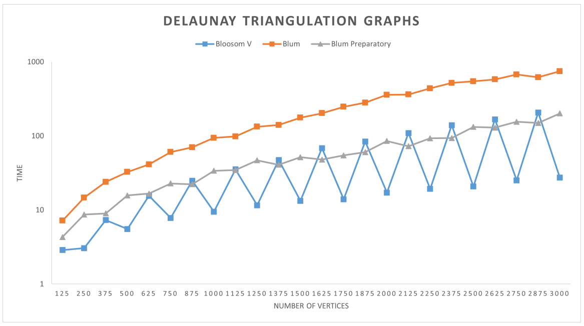
(h) DIMACS TT graphs

Figure 3: Runtime comparison between Blossom I and Blossom V across different graph families. The runtime is given in milliseconds.

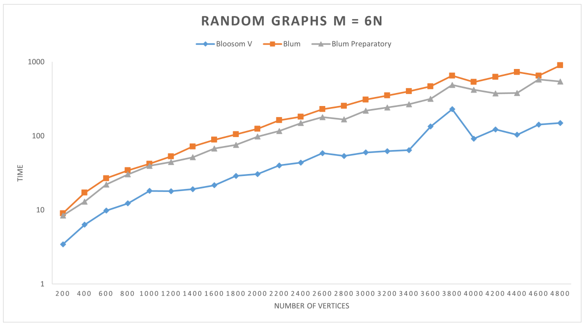
4. Conclusion

We gave a comparative analysis of the three maximum matching algorithms in unweighted, undirected graphs: Blossom I, Blossom V, and Blum's MDFS algorithm, along with a version that initializes MDFS with a random greedy matching. We compared both theoretical and experimental differences of performance over a large set of graph families: random graphs, geometric graphs, complete graphs, DIMACS benchmarks, and overlapping cycles.

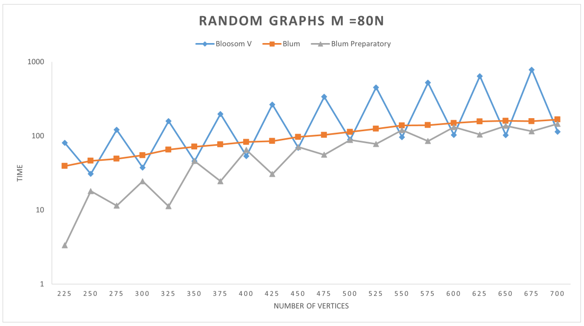
The results show that Blossom V performs substantially better than Blossom I due to improved data structures and heuristics. More importantly, our results show that Blum Preparatory, due to not



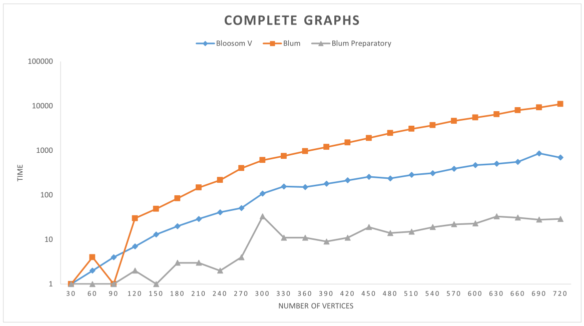
(a) Delaunay triangulations ($t = 50$)



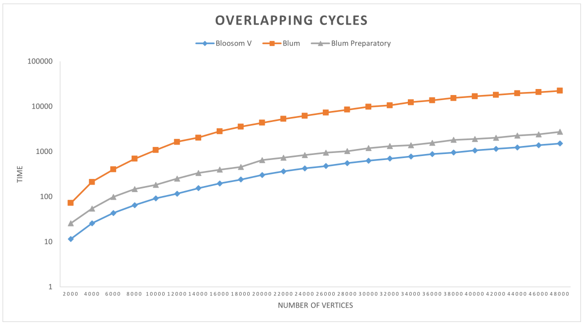
(b) Random graphs ($t = 50$ and $m = 6n$)



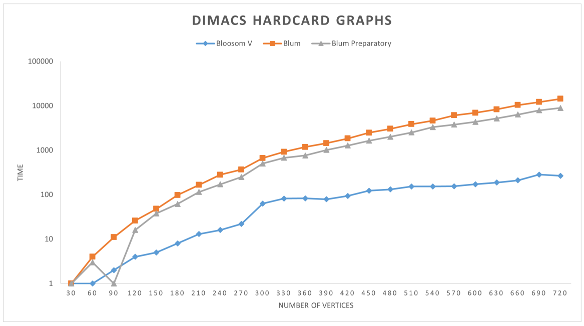
(c) Random graphs ($t = 50$ and $m = 80n$)



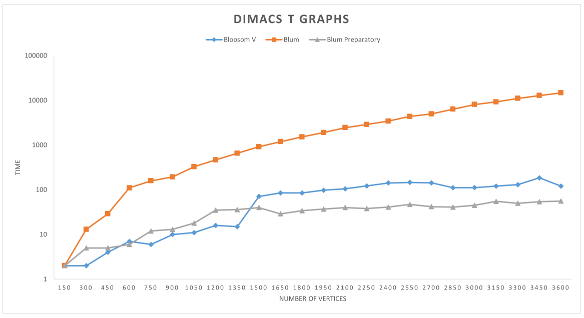
(d) Complete graphs



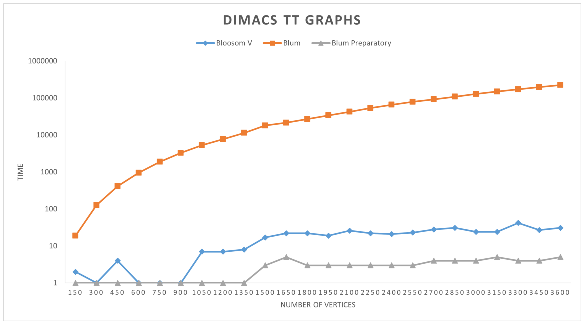
(e) Overlapping cycles ($t = 50$)



(f) DIMACS hardcard graphs



(g) DIMACS T graphs



(h) DIMACS TT graphs

Figure 4: Runtime comparison of Blossom V, Blum MDFS, and Blum Preparatory across diverse graph families. The runtime is given in milliseconds.

being complex and blossom shrinking overhead free, can achieve highly competitive performance, even beating Blossom V in some cases, particularly for structured and sparse graphs. This confirms the practical applicability of reachability-based matching when given a good initial matching.

We implemented of Blum's MDFS in Java and proposed some corrections for the previously found edge cases, rendering it correct and more versatile. Not only do these modifications enhance Blum's algorithm for real-world applications, but they also shed light on how initialization and structural characteristics of the input graph affect the efficiency of matching.

In conclusion, the paper confirms the effectiveness of augmenting path algorithms and draws out the inner strength of other techniques like MDFS. Some avenues for research work may include exploring hybrid techniques, parallel algorithms, and expanding analysis to the dynamic, weighted graphs or maximum 2-matching problems.

Declaration on Generative AI

During the preparation of this work, the author(s) used Grammarly solely for grammar and spelling correction. After using this tool, the author(s) carefully reviewed and edited the content as needed and take(s) full responsibility for the publication's content.

References

[1] C. Berge, Two theorems in graph theory, Proceedings of the National Academy of Sciences 43 (1957) 842–844.

[2] J. E. Hopcroft, R. M. Karp, An nˆ5/2 algorithm for maximum matchings in bipartite graphs, SIAM Journal on computing 2 (1973) 225–231.

[3] J. Edmonds, Paths, trees, and flowers, Canadian Journal of mathematics 17 (1965) 449–467.

[4] J. Edmonds, E. L. Johnson, S. C. Lockhart, Blossom i: a computer code for the matching problem, IBM TJ Watson Research Center, Yorktown Heights, New York 294 (1969).

[5] V. Kolmogorov, Blossom v: a new implementation of a minimum cost perfect matching algorithm, Mathematical Programming Computation 1 (2009) 43–67.

[6] W. Cook, A. Rohe, Computing minimum-weight perfect matchings, INFORMS journal on computing 11 (1999) 138–148.

[7] N. Blum, A new approach to maximum matching in general graphs, in: Automata, Languages and Programming, 17th International Colloquium, ICALP90, volume 443 of *LNCS*, Springer, 1990, pp. 586–597.

[8] N. Blum, Maximum matching in general graphs without explicit consideration of blossoms revisited, arXiv preprint arXiv:1509.04927, Updated version: https://theory.cs.uni-bonn.de/blum/papers/gmatching.pdf (2015).

[9] K. Schwarz, Edmonds' matching algorithm code, https://www.keithschwarz.com/interesting/code/?dir=edmonds-matching.

[10] JGraphT Project, Jgrapht - a java graph library, https://jgrapht.org.

[11] J. R. Shewchuk, Triangle: Engineering a 2d quality mesh generator and delaunay triangulator, in: Workshop on applied computational geometry, Springer, 1996, pp. 203–222.

[12] D. S. Johnson, C. C. McGeoch, et al., Network flows and matching: first DIMACS implementation challenge, volume 12, American Mathematical Soc., 1993.