Universiteit Utrecht

Approximating the maximum weight matching problem using the GraphBLAS standard

Master Thesis

Jetske Kemper

Mathematical sciences

Supervisors:

Prof. Dr. Rob H. Bisseling
Utrecht University

Dr. Ivan V. Kryven
Utrecht University

March 24, 2022

Abstract

The maximum weight matching problem for general graphs is a well-studied problem with a variety of approximation algorithms already existing. Yet, many of them are hard to parallelize. Therefore, we propose an approximation algorithm completely built on matrix operations in GraphBLAS using different semirings. The parallelization of these matrix operations is also well studied, and the idea is that parallelization is therefore more straightforward.

Our algorithm is based on the idea of positive-gain k-augmentations. We provide algorithms to search for these k-augmentations for $k = 1, 2, 3$ and we describe how to flip them if they exist. These algorithms can be performed fast since the runtime is linear in the size of the graph. Repeating these searching and flipping methods until no positive-gain k-augmentation exists, gives a guaranteed lower bound of $k/(k+1)$ times the optimal weight. This means that the best lower bound is equal to 3/4 times the optimal weight, meaning the algorithm is a 3/4-approximation algorithm.

We provide some numerical results for $k = 1, 2$, indicating that the quality of the matchings is indeed as expected. In addition, we analyse the runtime of a few different setups to find the fastest.

Contents

1 Introduction

Nowadays, graphs are widely used in a variety of applications and scientific areas due to their natural way of describing many real-life situations. Examples are road networks, social networks, as well as the structure of complicated molecules. Each situation gives rise to some specific questions and problems. One of these problems is the maximum weight matching problem: consider an undirected weighted graph $G(V, E)$ where V and E contain the vertices and the edges respectively and where $|V| = n$ and $|E| = m$ denote the number of vertices and edges. The goal is to find a set of edges such that the total weight is maximized and all edges are vertex disjoint. This problem occurs in numerous real-life applications where the goal is to assign some task, object or person to another entity, where each item can only be linked to one other item. Examples are: assigning tasks to employees, making a schedule for a tournament, and assigning donor organs to patients. The problem also has a more scientific application. When performing a sparse Gaussian elimination (LU-decomposition), it is beneficial to permute the original matrix such that the diagonal contains heavy weights. The pivoting approach by Olschowka and Neumaier [34] uses a perfect matching with maximum product of matched elements. The implementation by Duff and Koster [15], using sparse bipartite graphs is widely used nowadays.

The maximum weight matching problem is a well-studied problem. The Blossom-algorithm by Edmonds [16] is the first algorithm that computes a maximum matching in polynomial time. The implementation of this algorithm by Gabow [17] is the fastest known implementation, using $O(m)$ space and $O(nm+n^2 \log n)$ time. For some applications, it is not necessary to find an exact solution. A matching close enough to the maximum matching is often sufficient. Approximation algorithms are therefore often a great alternative. In comparison to heuristics, approximation algorithms guarantee a lower bound on the weight of the obtained matching in terms of the exact matching. A variety of these algorithms have been developed for the maximum weight matching problem. Many of them guarantee a lower bound of $1/2$ times the maximum weight. Examples of these algorithms are Greedy, LAM [37], PGA [12], PGA' [12], GPA [33] and Suitor[30]. These algorithms often return a matching which is far better than the $1/2$ guarantee [33, 36]. However, a better lower bound cannot be proved for general graphs. Another group of approximation algorithms guarantees a better bound. The quality of the bound as well as the runtime of the algorithm depend on a variable ϵ. Known lower bounds are $(2/3 - \epsilon)$ [10, 35], $(3/4 - \epsilon)$ and $(4/5 - \epsilon)$ [19, 13] and $(1 - \epsilon)$ [13, 14].

With the rise of parallel computing, the question arises how to compute a maximum weight matching in parallel. Many of the previous mentioned algorithms are inherently difficult to parallelize due to their sequential nature. However, some are suitable for parallelization, such as the local domination algorithm [37] and the Suitor algorithm [30]. For the distributed memory models, a few algorithms have been developed based on the concepts and ideas of these algorithms [26, 21, 29, 5].

In recent years, there has been a (renewed) interest in the duality between graphs and matrices. A community effort to standardize the building blocks of this duality has led to GraphBLAS [31, 22, 7]. GraphBLAS is a standard in which graph operations can be seen as matrix-vector operations using different semirings. An advantage of using this duality is that parallelization comes almost for free, since parallelization of (sparse) matrix operations is widely studied [23].

The aim of this thesis was therefore to develop an approximation algorithm for the maximum weight matching problem using GraphBLAS in reasonable time. Since there already exist many $1/2$-approximations, the additional goal was to achieve a lower bound of at least $2/3$. The algorithm is based on performing different improvements called augmentations and a lemma which guarantees

a lower bound if no positive-gain augmentations of a certain type can be found which improve the matching.

This thesis is structured as follows: Chapter 1 introduces the problem, states the preliminaries for the augmentations and introduces the used concepts of GraphBLAS. In addition, some existing algorithms are discussed in more depth. Chapter 2 describes our algorithm based on different augmentations. The first section focuses on the three simplest augmentations, whereafter the step is made to more complicated augmentations. Chapter 3 shows some results and finally Chapter 4 draws conclusions.

1.1 Problem description and notation

In this section, some basic notation and definitions considering graphs used in this thesis are described. Consider an undirected weighted graph $G(V, E)$ where V and E contain the vertices and the edges respectively. We will denote $|V| = n$ and $|E| = m$. Two vertices are adjacent if they are linked by an edge. Two edges are called adjacent or incident if they have a vertex in common. If a vertex is on an edge, we say that the vertex is incident to this edge. The weights of the edges are denoted by $w(e)$ for all edges $e \quad E$. In this thesis, we will assume that the weights are positive. If edge $e \ / \ E$, the weight of the edge is zero. The total weight of a set of edges S is given by

$$w(S) = \sum_{e \ S} w(e).$$

A matching M is a set of edges such that every edge does not have a vertex in common with another edge. This means that every vertex has either one incident edge or zero incident edges in M. An edge in a matching is called a matched edge and an edge not in a matching an unmatched edge. Similarly, a vertex that is incident to an edge from the matching is called a matched vertex and a vertex not incident to a matched edge an unmatched vertex. A set of edges that forms a matching and cannot be extended by adding an unmatched edge is called a maximal matching. A matching that contains the maximum possible number of edges is called a maximum matching. Note that a maximum matching is a maximal matching but a maximal matching does not have to be a maximum matching. When all vertices in a graph are matched, the matching is called a perfect matching. Note that this can only occur when the number of vertices is even.

Given a set of edges S, we will use a solid line in figures to indicate when an edge is in S. When an edge is not in S, it will be depicted with a dashed line. Vertices incident to an edge in S will be depicted by a solid black circle, whereas a vertex with no edge from S incident to it will be an open circle. Using this representation, Figure 1a shows a matching whereas Figure 1b does not since one vertex has two edges incident to it.

The maximum weight matching problem looks for a matching M such that the sum of the weights is maximized. Note that a maximum weight matching does not have to be a maximum matching. However, if the weights are positive, the matching must be maximal. If not, we could add another unmatched edge to the matching which will increase the total weight. This is a contradiction since we assumed to have a maximum weight matching.

There are some special cases of the maximum weight matching problem. The first is when all weights of the graph are one. In this situation, the maximum weight matching problem is equivalent to the maximum cardinality matching problem, i.e., find a matching with as many edges as possible. The second case is when the vertices can be split into two different sets such that for every vertex, it cannot be adjacent to a vertex from the same set. This kind of graph is called a bipartite graph.

(a) A matching (b) Not a matching

Figure 1: Two simple graphs with each a set of edges S depicted. Edges in S are depicted as a solid line and vertices incident to them are denoted by a solid black circle. The dashed lines depict edges not in S and if a vertex has no adjacent edge in S, it is depicted as an open circle.

For both special cases, there are specific algorithms that only work for that case and not for a general graph. For the remaining part of this thesis, we will assume we have a general graph.

1.2 Augmentations

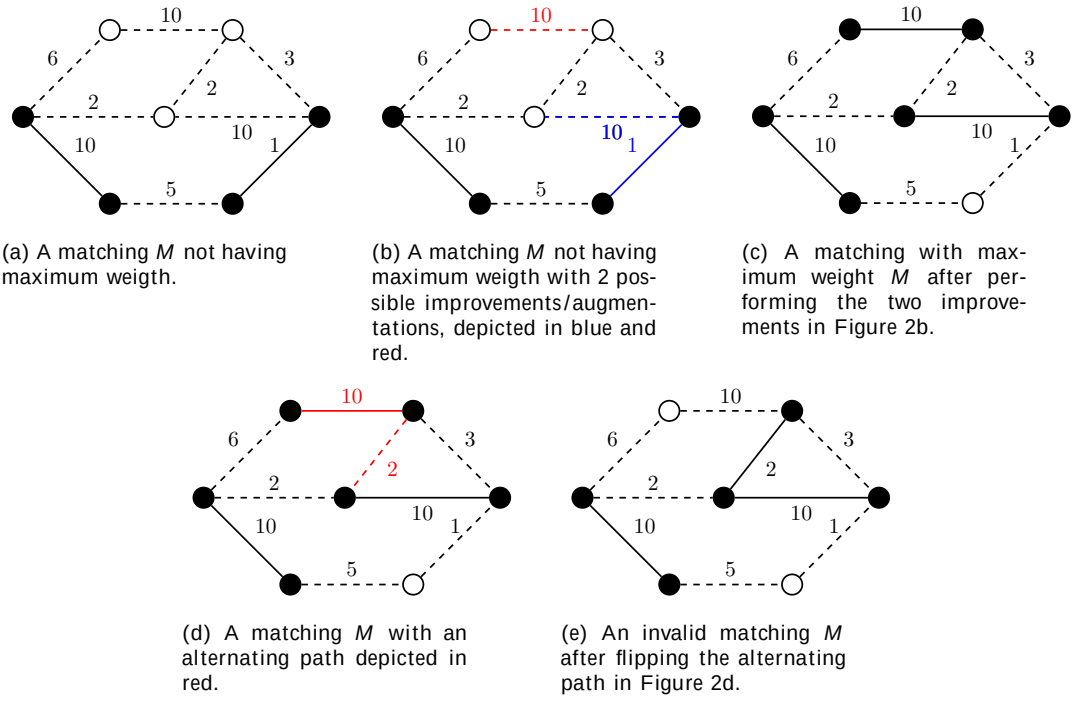
In this section, we discuss how we can improve an existing matching M, define the term augmentations and state an important lemma guaranteeing a lower bound based on these augmentations. The theory in this section is based on [36] by Pothen, Ferdous and Manne.

An existing matching M can be improved in many ways. Take, for example, the matching M from Figure 2a. The red unmatched edge in Figure 2b is the most straightforward edge to add to the matching. This unmatched edge is not incident to matched vertices and therefore, adding this edge gives a valid matching with a higher weight. If at least one of the adjacent vertices is matched and the edge would be added to M, M becomes an invalid matching. However, it could be that the weight of the unmatched edge is higher than the adjacent matched edge, as can be seen in the blue path in Figure 2b. Therefore, deleting the matched edge from M and adding the unmatched edge also improves the matching. Figure 2c shows the improved matching where these improvements are performed. The idea of deleting matched vertices from M and adding adjacent unmatched vertices is the main idea for augmentations. To define them properly, we need the notion of an alternating path or cycle:

Definition 1.1. Let M be a matching. Then, a path or a cycle is alternating if the edges are alternately drawn from the matching M and all edges not in the matching: $E \setminus M$.

An alternating path or cycle forms the basis for the improvements of matching M. Each improvement must be an alternating path or cycle. If two unmatched edges are allowed to be adjacent to each other in an improvement, the matching becomes invalid when the improvement is performed. However, an alternating path is not sufficient to guarantee a valid matching. To see this, consider Figure 2d. The red path is an alternating path. However, performing this improvement yields an invalid matching, see Figure 2e. Therefore, we define an augmentation as those alternating paths which give a valid matching.

Definition 1.2. Let $M \quad P = (M \setminus P) \quad (P \setminus M)$ be the symmetric difference of two sets and let P be an alternating path or cycle. Then P is called an augmentation with respect to M if $M \quad P$ is also a matching. If it is clear from the context which matching M is used, P will be just called an augmentation.



(a) A matching M not having maximum weigth.

(b) A matching M not having maximum weigth with 2 possible improvements/augmentations, depicted in blue and red.

(c) A matching with maximum weight M after performing the two improvements in Figure 2b.

(d) A matching M with an alternating path depicted in red.

(e) An invalid matching M after flipping the alternating path in Figure 2d.

Figure 2: A simple graph to show di□erent situations involving augmentations.

The symmetric difference of P and M is the mathematical description of deleting the matched edges in the alternating path and adding the unmatched edges. We will call this process the flipping of an alternating path. An augmentation is therefore an alternating path which gives a valid matching if the path is flipped. In fact, an alternating path can only be an augmentation if the matched edges adjacent to an unmatched edge in the alternating path are included in the same path. Note that finding an augmentation does not automatically yield a better matching. Therefore, we will speak of a positive-gain augmentation if the total weight of a matching is increased when M is replaced by $M \quad P$. The gain of an alternating path or cycle P is denoted by

$$g(P) = w(P \setminus M) - w(P \cap M).$$

Intuitively, it is clear that a matching M has maximum weight if and only if no positive-gain augmentation can be found. If a matching has maximum weight and there is a positive-gain augmentation, the matching could not have had maximum weight. Furthermore, if a matching does not have maximum weight, there must be a path or cycle P with respect to M which is a positive-gain augmentation with respect to M. To see this, let M be a matching with maximum weight and define the following variables: $S = M \cap M$, $R = M \setminus S$ and $R = M \setminus S$. Then

$$w(M) = w(S) + w(R) < w(S) + w(R\,) = w(M\,)$$

yielding $w(R) < w(R\,)$. By assumption, M and M are both matchings and thus are R and R . Therefore, the matched vertices in the union of R and R have a degree of at most two. As a result,

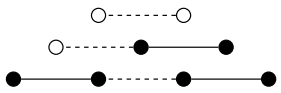


Figure 3: All possible 1-augmentations.

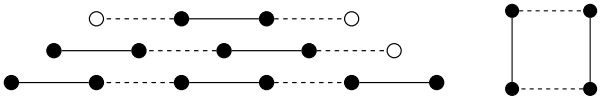


Figure 4: All types of augmentations with exactly 2 edges not in M.

R R consist of cycles of even length and paths with edges alternately drawn from R and R . Since $w(R) < w(R\ )$, for at least one of these paths or cycles the total weight of the edges in R must be higher than those in R. Call one of such paths or cycles P. Since P is an alternating path or cycle, augmenting M with P leads to a new matching with a higher weight, which proves the statement.

Note that there are many augmentations of different lengths. To distinguish between those augmentations, we introduce k-augmentations. A k-augmentations is an augmentation with at most k edges not in M. Figure 3 shows all possible types of 1-augmentations and Figure 4 shows all possible types of augmentations with exactly 2 edges not in the matching M. Together with the 1-augmentations, they form all types of the 2-augmentations.

In matching with maximum weight, all k-augmentations must be negative or zero gain augmentations for all k. When using approximation algorithms, this does most likely not hold. However, the approximation lemma states that when no k-augmentations occur, a certain lower bound can be proved.

Lemma 1 (Approximation Lemma)**.** Let M be a matching on G and k be an integer greater than 1. If furthermore, M does not admit any positive-gain $(k - 1)$-augmentations, then

$$\frac{k-1}{k} w(M\ ) \leq w(M),$$

where M is the matching with maximum weight.

Drake and Hougardy proved the result for $k = 3$ in 2003 [11]. The general statement as stated in Lemma 1 is proved in [36] by Pothen, Ferdous and Manne.

Using positive-gain 2-augmentations to improve the outcome of a 1/2-approximation algorithm was suggested by Drake and Hougardy in [11]. They state that a maximal set of pairwise vertex disjoint 2-augmentations can be found in linear time. They numerically show that performing a number of these augmentations increases the quality of a matching found by using a 1/2-approximation considerably. However, the guaranteed lower bound is not increased since they only perform a fixed number of such improvements, and they cannot guarantee that they considered all positive-gain 2-augmentations.

1.3 Known approximation algorithms

In this section, we give a short overview of the existing algorithms for the maximum weight matching problem. Despite the existence of an exact polynomial runtime algorithm, many approximation algorithms have been developed. Solutions obtained from these algorithms have, unlike heuristics, a guaranteed lower bound. Let M be the matching obtained by such an algorithm and let M be the matching with maximum weight. Then, the following must hold:

$$\alpha \cdot w(M\ ) \leq w(M) \leq w(M\ ),$$

i.e., the total weight of the approximation is at least α times the maximum weight. The idea is that by approximating the solution, one can complete the algorithm in less time than by using an exact algorithm. Furthermore, approximation algorithms can be much simpler than exact algorithms and therefore they are often easier to understand and implement.

The simplest approximation algorithm is the Greedy algorithm. The Greedy algorithm starts with an empty matching and considers all edges by nonincreasing weight. If the considered edge has two unmatched vertices incident to it, the edge is added to the matching. If not, the edge is discarded. It can be shown that the Greedy algorithm results in a 1/2-approximation. Due to the sorting of the edges, the time complexity of the Greedy algorithm is $O(m \log n)$.

The first linear-time approximation algorithm was given by Preis in 1999 [37] and is called LAM. It uses the idea of the Greedy algorithm but instead of using the global heaviest edge each iteration, it considers a local heaviest edge. A local heaviest edge e is an edge such that the weight of e is at least the weight of all adjacent edges of e. In other words: if $e = (u, v)$ is a local heaviest edge then $w((u, v)) \geq w((a, b))$ for all $(a, b) \quad E$ such that $a = u$ or $b = v$. The algorithm adds such an edge to the matching and removes all edges adjacent to e and itself from E. This is repeated until E is empty. This algorithm is also called the Locally Dominant Edge algorithm since each iteration, it searches for the edge that is dominating over all its adjacent edges. In his paper, Preis shows that LAM is a 1/2-approximation algorithm and that it has runtime $O(m)$. However, to obtain the linear runtime, a Depth-First-Search is used. This makes parallelizing the algorithm directly hard. Although the algorithm itself is not suitable, it provides useful ideas for other algorithms that are parallelizable.

In 2003, Drake and Hougardy published another approach for a 1/2-approximation algorithm [12]. This algorithm is known as the Path Growing Algorithm (PGA). The algorithm starts at a single vertex v with a degree of at least 1. The goal is to construct a path P_1 starting at this vertex. Each step, the heaviest edge incident to the endpoint of the already existing path is added to the path and all other edges incident to the endpoint are removed. This is repeated until no edge can be added to the path. Then, if there are still edges left, a new not yet visited starting point v is chosen and the procedure starts again. The algorithm terminates when no new starting point can be found. The result is a collection of l paths $\{P_1, P_2, \ldots, P_l\}$. During the process, each edge that is added to a path is alternately put into the sets M_1 and M_2. Then, by construction, M_1 and M_2 are matchings. The matching with the largest weight is returned. The runtime of this algorithm is $O(m + n)$. In [12], it is proved that this is indeed a 1/2-approximation algorithm. In the same paper, some additional improvements are stated. For example, the obtained matching can be a non-maximal matching. Therefore, the matching can be improved easily by extending the found matching to a maximal matching. Furthermore, instead of just putting an edge alternately into M_1 or M_2, one can determine the best matching for each P_i. This can be done via dynamic programming and this algorithm is often denoted by PGA . Each of these improvements do not increase the asymptotic runtime. Since this algorithm constructs long paths, it is most likely not suitable for parallelization.

In 2007, Maue and Sanders proposed the Global Paths Algorithm (GPA) [33]. This algorithm combines the ideas of Greedy and PGA . It uses the path creating approach from PGA but this time it considers all edges in nonincreasing order. If an edge is applicable, it is added to a set of edges E . An edge is applicable if adding it to E does not yield cycles of odd length or vertices with more than 2 adjacent edges. When all edges are considered, E consist of paths and cycles of even length. Then, for each such path or cycle, a maximum matching is found using the same dynamical programming technique as in PGA . The final result is the union of all found maximum

matchings. Maue and Sanders also propose an alternative post-processing step. Instead of extending the algorithm to a maximal matching by just picking the heaviest weights, they perform GPA again on the edges which have two unmatched vertices incident to it. Although this algorithm focuses more on finding heavier paths, the guarantee is still 1/2. The runtime is $O(m + \text{sort}(m))$ which in most cases is equal to $O(m \log n)$.

Up to this point, all algorithms are 1/2-approximations. Although these algorithms perform quite well in practice, a better bound cannot be guaranteed for general cases. Drake and Hougardy proposed the first $(2/3 - \epsilon)$-approximation [10] where both the lower bound and the runtime depend on ϵ. Given a fixed ϵ, the runtime is linear in the size of the graph. The algorithm uses the idea of using 2-augmentations to improve an already existing maximal matching. Sometime later, Pettie and Sanders propose a similar but simpler algorithm [35]. In this paper, they propose two different algorithms: one based on randomization and a deterministic algorithm, both with a lower bound of $(2/3 - \epsilon)$. The lower bound and the runtime for the algorithm based on randomization is expected. Although all three algorithms are linear in time and the lower bound can get arbitrarily close to 2/3, they are often more complicated than the previously mentioned algorithms. Similarly, using the ideas of augmenting paths and cycles, algorithms have been developed that guarantee a $(3/4 - \epsilon)$ or even a $(4/5 - \epsilon)$ lower bound [19], [13]. In 2010, Duan and Pettie propose a near linear-time $(1 - \epsilon)$-approximation [13] which they improve in 2014 to linear runtime given a fixed ϵ [14]. Although these lower bounds can get as good as one wants, these algorithms are often quite complicated to understand, analyse and implement.

Parallel algorithms

In the past decades, computers advanced considerably, increasing the computational speed. One of these improvements is the change from one processor to multiple ones. By having multiple processors, more work can be done at the same time in parallel, speeding up the process. This development gave rise to a new problem: how to distribute the work over different processors. The answer is often not as simple as taking a sequential algorithm and distributing the work evenly. Processors need to communicate and synchronize at some point in time to share their progress with the other processors which can be very costly. To analyse these costs, many models have been developed. One of these models is the bulk synchronous parallel model (BSP) [42]. The model consists of different supersteps (communication, computation, or both). At the end of each superstep, each processor synchronizes with the other ones. This means that if one processor takes more time than the others, some processors stay idle for a moment. After the synchronization, the processors continue with their work.

For many problems, special algorithms need to be developed for the parallel case. The matching problem is one of these problems. Many of the sequential algorithms mentioned in the previous section are inherently sequential, and the gain of parallelizing them is very limited. There are already numerous parallel algorithms for the (bipartite) weighted matching problem. However, most of them are developed for shared memory models. We will focus on algorithms for distributed memory models and general graphs.

One of the first algorithms for distributed memory models is an algorithm based on the ideas behind the local domination algorithm and is developed by Manne and Bisseling [29]. It is based on the parallel algorithm by Hoepman [21] which uses the idea of LAM by Preis [37].

In 2014, Manne and Halappanavar [30] presented an algorithm called Suitor based on [29] which also works well when used sequential. Each vertex u makes a bid to match with one of its neighbours

equal to the weight of the edge between the two vertices. Vertex u proposes to the vertex v which yields the highest weight under the requirement that v has not received a higher offer from another vertex. Let Suitor(v) denote the adjacent vertex with the highest edge weight. If at some moment in time such a higher offer occurs, the bid by u is withdrawn and u has to propose to another vertex if possible. The algorithm terminates when for all u, u is a suitor or u has no vertex to propose to. The edges (u, v) with $u = \text{Suitor}(v)$ and $v = \text{Suitor}(u)$ are the edges in the matching. Note that it is important that each edge weight is unique. If not, three vertices with edges between them of equal weight can result in a stalemate where no matches can be made. If the weights are all unique, this is not a problem, but in general graphs this does not hold. To overcome this, one can add a second weight criterion by, for example, considering the sum of the vertex numbers of an edge. If a tie occurs, the edge with the highest sum is the one with the highest weight. Since only edges adjacent to each other are compared, the edges differ only one vertex and therefore it is sufficient to compare these two vertices. When tie-breaking is done consistently, the Suitor algorithm gives the same outcome as the Greedy algorithm. Therefore, it gives a 1/2-approximation.

This algorithm is extended in [5] by Bisseling using the ideas of the Suitor algorithm [30]. New in this algorithm is the partial sorting added to find preferences faster.

1.4 GraphBLAS

In most situations, graphs are denoted by vertices and edges. However, each graph can also be represented as a matrix. Through the past years, this duality between graphs and matrices has been stated multiple times and has been proved to be useful [20, 24]. For many known problems, algorithms are developed in terms of matrix operations. Examples are a Breadth-First-Search (BFS) algorithm [6], Shortest Path algorithms [40] and PageRank algorithms [38]. Likewise, it is also used for Clustering [4] and Deep Neural Networks [25].

In 2013, Mattson et al. [31] proposed to define a standard which defines some primitive building blocks for graph operations based on linear algebra. The authors of the manifest were worried that without such a standard, progress would be held back. Algorithms with different building blocks are more difficult to compare, and it also makes it harder to discuss the topic. From this point onwards, GraphBLAS is developed as a community effort. This resulted in a description of the mathematical foundations by Kepner et al. in 2016 [22] and a C binding called the GraphBLAS C API [7] in 2017. Nowadays, there are multiple implementations of the GraphBLAS standard. One of these implementations is the SuiteSparse:GraphBLAS library by Timothy Davis [8]. In the recent versions, it includes (shared-memory) parallelism [28] and a MATLAB interface. Other implementations include GraphBLAST [43] for the GPU and the implementation of Yzelman et al. for C++[44].

These days, the focus shifts to developing algorithms which are built on top of GraphBLAS [32]. Some algorithms completely built on GraphBLAS can be found in [41]. Up till now, no algorithm for the maximum weight matching problem for general graphs is developed in terms of matrix operations. However, for some special cases, there have been some developments. For the bipartite case, an algorithm for the cardinality matching problem is developed by Azad and Buluç [1, 2] and a few years later an algorithm for the maximum weight perfect matching problem was developed [3].

In the next sections, we will introduce the basic ideas behind GraphBLAS, introduce additional functions and operations and give a short remark about sparse matrix operations in GraphBLAS.

Main idea: matrix-vector multiplication

In this subsection, we will introduce the idea behind GraphBLAS and give some notation. This notation is based on [22].

As stated before, the main idea behind GraphBLAS is that each graph can be represented as a matrix. The two most used representations are the adjacency matrix and incidence matrix. We will use only the adjacency matrix, and therefore we will only describe this representation. An adjacency matrix A is often a square $n \times n$-matrix which has a nonzero at $A(i,j)$ if there is a directed edge from vertex i to vertex j. This nonzero is equal to one if the graph is unweighted, and is equal to its weight if the graph is weighted. If the graph is undirected, the adjacency matrix is symmetric.

GraphBLAS uses this matrix representation to express graph operations in terms of linear algebra. The simplest example is the matrix-vector multiplication Ae_1. This operation gives all vertices adjacent to vertex 1. This can be extended to $A^k e_1$ which gives the vertices after k hops in the graph starting from vertex 1. This example uses the standard matrix-vector multiplication which can be written as:

$$(Af)(i) = \sum_{k=1}^{n} A(i,k) f(k) = \bigoplus_{k=1}^{n} A(i,k) \quad f(k)$$

where $= +$ and $= \times$ are the standard addition and multiplication.

There are many possible choices for the operators and and each combination can be a graph operation. However, there are some restrictions on the possible combinations. Each combination of addition and multiplication needs to form a (GraphBLAS) semiring. The definition of a GraphBLAS semiring and a monoid are given below.

Definition 1.3 (Commutative monoid)**.** Given an operator $: D \times D \to D$. The algebraic structure $D, \ , 0$ is called a commutative monoid if for all $a, b, c \ D$ the following holds

- $a \quad b = b \quad a$ (commutative)
- $(a \quad b) \quad c = a \quad (b \quad c)$ (associative)
- $a \quad 0 = a = 0 \quad a$ (identity)

Definition 1.4 (GraphBLAS semiring)**.** The algebraic structure $D, \ , \ , 0$ is called a GraphBLAS semiring if

- $D, \ , 0$ is a commutative monoid with the addition operator
- the multiplication operator is a closed binary operator where multiplication distributes over addition.

Note that the definition of a GraphBLAS semiring is slightly different from the conventional mathematical definition of a semiring [18, 27]. The latter one requires to be a monoid. For the GraphBLAS semiring this is not obligatory. This extra freedom gives more possibilities when designing an algorithm. The matrix-vector multiplication between matrix A and vector b over a semiring $D, \ , \ , 0$ will be denoted by $A \ . \ b$. Some example GraphBLAS semirings and their applications are:

- The semiring using the standard addition and multiplication: $\mathbb{R}, +, \times, 0$. The standard matrix-vector multiplication uses this semiring.
- The semiring $\mathbb{R} \quad \{+\infty\}, \min, +, +\infty$. This semiring can be used to determine all lengths of shortest paths from a starting source s.
- The GraphBLAS semiring $\mathbb{N} \quad \{+\infty\}, \min, \text{first}, +\infty$ with $\text{first}(x, y) = x$. Note that this GraphBLAS semiring is not a conventional semiring since the function first is not a (commutative) monoid. This semiring is used in performing a BFS and finding a BFS-parent of each vertex.

Other operations and functions

GraphBLAS offers plenty more possibilities. When performing a matrix-vector multiplication, it is possible to add some extra options. One of these is applying a mask. A mask prevents the adjusting of certain elements in an output matrix C. This output can be the result of a matrix multiplication or some other operation. A mask can be expressed as a boolean matrix M whose size is the same as the output. If $M(i, j)$ is true, then the result of the main operation is written at $C(i, j)$. If it is false, the mask prevents the output to be modified. The notation of applying a mask is $C \; M \; = Z$ where C is the output, M the mask and Z the result of some operation. It is also possible to use the complement of mask M as a mask, i.e., $C \; \neg M \; = Z$.

Another option is the use of an accumulator option. It is denoted by $C = C \quad T$ or $C \quad = T$ and accumulates entries of T into existing C entries. For example, $C = C + AB$. This operator is controlled by a descriptor object. The option replace is also controlled by such a descriptor object. It does what it says, it deletes all previous entries in C before setting the new values. It is possible to use replace in combination with a mask.

Another possibility is to perform an element-wise addition and multiplication. Similar to changing the operators in the matrix-matrix multiplication, the operator used in these calculations can be changed into any binary operator. An element-wise addition gives a different outcome than an element-wise multiplication even when the same binary operator is used. An element-wise addition between A and B has a nonzero at position (i, j) if at least one of $A(i, j)$ or $B(i, j)$ is nonzero. The operation is defined as follows:

$$[A \quad B](i,j) = \begin{cases} A(i,j) & & \text{if only } A(i,j) \text{ is nonzero} \\ B(i,j) & & \text{if only } B(i,j) \text{ is nonzero} \\ A(i,j) & B(i,j) & \text{if both } A(i,j) \text{ and } B(i,j) \text{ are nonzero} \\ 0 & & \text{otherwise.} \end{cases}$$

Let a pattern of a matrix represent the nonzeros of a matrix. It is clear that an element-wise addition represents the union of the patterns of A and B. This in contrary to an element-wise multiplication. An element-wise multiplication between A and B has a nonzero at position (i, j) if both $A(i, j)$ and $B(i, j)$ are nonzero. Therefore, it represents the intersection of the patterns of A and B.

In Table 1 the notation of the operations described above is mentioned as well as some other often used operations such as the reduction of a matrix to a vector. Note that most of the operations are described as matrix operations, but there is often also a vector variant. Table 2 shows some GraphBLAS functions that are used in this thesis.

Mathematical notation	Name	Examples
$c = A \quad . \quad b$	Matrix-vector multiplication	max.first, $+.\times$
$C = A \quad . \quad B$	Matrix-matrix multiplication	
$C \ M \ = Z$	Mask operation. M is the boolean matrix of the same size as C.	
$C = C \quad T$ or $C \ = T$	Accumulator	$C = C + T$
$C = A^{\intercal}$	Transpose	
$C = A \quad B$	Element-wise addition	
$C = A \quad B$	Element-wise multiplication	
$v = \bigoplus_j S(:, j)$	Reduce a matrix to a vector by summing up all elements in a row.	
$D, \quad , \quad , 0$	GraphBLAS semiring	

Table 1: Table with the most important notation when using GraphBLAS.

Function name	Input	Explanation
GrB.prune	Matrix or vector and an additional scalar (not obligatory).	Deletes all explicit zeros. When an additional scalar is given, it deletes all explicit entries equal to the scalar.
GrB.select	Matrix or vector and an additional string.	Select specified elements of the input matrix, determined by the input string. Examples are: 'positive', 'tril'. Sometimes an additional input entry is necessary.
GrB.emult	Two matrices of the same size and a binary operator	Element-wise multiplication using a specified binary operator. The standard operator is $\times$, although any binary operator can be used.
GrB.eadd	Two matrices of the same size and a binary operator	Element-wise addition using a specified binary operator. The standard operator is $+$, although any binary operator can be used.

Table 2: Table with functions in GraphBLAS used in this thesis.

Sparse matrices in GraphBLAS

Often, the adjacency matrix A is a sparse matrix, i.e, A has many zeros. In this data structure, only the nonzeros are stored to prevent unnecessary storage use. When using GraphBLAS, the element that is not stored depends on the monoid used as the addition operation [7]. This is important to realize, since the zero element depends on the used semiring, and it is likely to change the semiring during an algorithm. This means that it is not possible to assume the value of a not stored element since there might be multiple different options. Stored values will be called explicit values or entries, whereas non-stored values are called implicit values. In addition to the definition of implicit values, it is important to know what this means for matrix operations in GraphBLAS. Let us consider the following matrix-matrix multiplication:

$$C(i,j) = \bigoplus_{k=1}^{n} A(i,k) \quad B(k,j).$$

The operation $A(i,k) \quad B(k,j)$ is only performed if $A(i,k)$ and $B(k,j)$ are both explicit. If we allow one of the two entries to be implicit, this could lead to unwanted situations if the semiring is changed during the algorithm. Note that this approach requires more work if we want to perform $A(i,k) \quad B(k,j)$ where $B(k,j)$ is equal to zero, and we want $A(i,k) \quad B(k,j)$ to be used into the $\bigoplus$ operation. This is only possible when $B(k,j)$ is stored explicitly. This in contrast to the normal sparse matrix operation, where the assumption can be made that the implicit value is equal to zero. This is important to keep in mind since this situation will occur later on in this thesis. When talking about an element being zero, we will mean that this zero is stored implicit unless stated otherwise. The same applies when talking about nonzeros. A nonzero means an explicit stored value, unless stated otherwise.

This implementation of a sparse matrix-matrix multiplication makes the multiplication between a permutation matrix P and a sparse matrix A masked with sparse matrix B very fast. To see this, consider a single element of output matrix:

$$[P \quad . \quad A](i,j) = \bigoplus_{k=1}^{n} P(i,k) \quad A(k,j).$$

The operation $P(i,k) \quad A(k,j)$ is performed only once due to P being a permutation matrix. Therefore, the operation is only performed on a single explicit value and thus, to compute $[P \quad . \quad A](i,j)$ only two operations are needed. This implies that the complexity of the matrix-matrix computation only depends on the number of nonzeros in matrix B. Only for elements (i,j) where $B(i,j)$ is nonzero, the matrix-matrix multiplication is performed. If the number of nonzeros in matrix B is bounded by the number of nonzeros in adjacency matrix A, the runtime is linear in the number of edges m.

2 Design of the algorithm

Our approximation algorithm is based on finding a vertex-disjoint set of positive-gain k-augmentations, after which these augmentations are flipped simultaneously. When this is repeated until no positive-gain k-augmentation can be found, a lower bound of at least $\frac{k}{k+1}$ times the optimal solution can be guaranteed as a result of the approximation lemma. By using a vertex disjoint set, we ensure that the new matching remains valid when multiple augmentations are flipped at the same time, whereas searching for multiple augmentations and flipping them at once is a good base for parallelization. This idea of finding positive-gain k-augmentations and flipping them is used in many algorithms already [10, 11, 12, 13, 14, 19, 35].

The pseudocode for our main algorithm can be found in Algorithm 1. The structure is as follows: each iteration, a set of vertex disjoint positive-gain k-augmentations is found by the function *FindkAug*(A, k, M). This step is significantly different for each k. After this step, the found augmentations are flipped and the process is repeated until no positive-gain k-augmentations can be found. The flipping phase *Flipping*(M, Aug, k) also depends on parameter k, but the function is very similar for each k.

Algorithm 1: Main algorithm

Input : Adjacency matrix A, parameter k.
Output: A matching M with a weight of at least $\frac{k}{k+1}w(M^*)$.

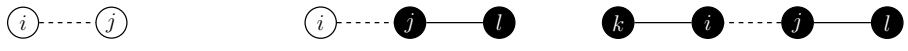
```
1 M = ∅ ;
2 while a positive-gain k−augmentations can be found in matching M;
3 do
4 |  Aug = FindkAug(A, k, M);      // Find a vertex disjoint set of positive-gain
  |   k−augmentations named Aug
5 |  M = Flipping(M, Aug, k) ;     // Augment matching M with Aug: M = M ⊕ Aug
```

In Section 2.1 we describe the details of Algorithm 1 for $k = 1$, providing two different approaches for the function *FindkAug*(A, k, M). We also give a simple method to detect if there exists a positive-gain 1-augmentation in an existing matching without finding the path itself. We also provide how to flip the found augmentations within GraphBLAS. In section 2.2 we then generalize the ideas of $k = 1$ to $k = 2$ with the emphasis on *FindkAug*(A, k, M) and in 2.3 we do the same for $k = 3$.

2.1 1-augmentations

In this section, we discuss how the algorithm works for $k = 1$, including how to search for (positive-gain) 1-augmentations and how to flip them. Two different approaches are discussed for finding the 1-augmentations. We also give a method which checks for termination only. All three methods have in common that they treat all 1-augmentations within the same operation. The intuition behind this is that these augmentations are very similar to each other. Each 1-augmentation can be seen as an unmatched edge extended with either zero, one or two adjacent matched edges. Therefore, taking this unmatched edge as the starting point, the hope is that these augmentations can be treated simultaneously. At last, the method for flipping is described for $k = 1$ and some remarks are made for general k.

Before we continue, we set some notation. In this section we will use the labelling of the vertices and edges as in Figure 5. In all three augmentations, the vertices incident to the unmatched edge are



(a) Augmentation 1a. (b) Augmentation 1b. (c) Augmentation 1c.

Figure 5: All three 1-augmentations with labelled vertices.

labelled i and j. Starting with this edge, augmentation 1a is extended to 1b by adding a matched edge adjacent to vertex j. The matched vertex adjacent to this edge is called l. Augmentation 1b can be extended to augmentation 1c by adding another matched edge adjacent to vertex i. The vertex adjacent to this edge is called vertex k. We will say that each augmentation is centred around edge (i, j). We will also say that the augmentation with unmatched centre edge (i, j) has vertex i as starting vertex or starts at i and that vertex j is the end vertex or the augmentation ends at vertex j. With this notation, the gain of the three augmentations can be expressed as follows:

$$\begin{aligned} &\text{gain augmentation 1a:} && w(i,j), \\ &\text{gain augmentation 1b:} && w(i,j) - w(j,l), \\ &\text{gain augmentation 1c:} && w(i,j) - w(j,l) - w(i,k). \end{aligned}$$

With the use of the indicator function

$$\mathbb{1}_E(i,j) = \begin{cases} 1 & \text{if } (i,j) \quad E \\ 0 & \text{if } (i,j) \ / \ E, \end{cases}$$

where E is a set of edges, we can express all three gains in a single expression:

$$w(i,j) - w(j,l)\mathbb{1}_M(j,l) - w(i,k)\mathbb{1}_M(i,k).$$

The matrix $[A \setminus M]$ only contains the edges in the graph which are not in the matching M. Vector m contains the index of the adjacent matched vertex if vertex i is matched and zero otherwise, and vector m_w contains the weight of the matched edge if vertex i is matched and zero otherwise. Note that for the algorithm to work, the zeros in m_w needs to be stored explicitly, as will be shown later in this section. Table 3 shows the most important variables used in this section.

2.1.1 Termination method

The method discussed in this subsection can be used to check if there exists a positive-gain 1-augmentation without knowing where the corresponding path is in the graph. It is therefore unsuitable for performing a flip, but it can be used as a termination criterium. Additionally, the first approach for searching uses similar ideas to this method, and therefore we discuss it.

The idea behind this method is as follows: each row i of matrix $[A \setminus M]$ represents the unmatched edges adjacent to vertex i. The position of the edge in the row indicates the index of the adjacent vertex j. This vertex j can be adjacent to a matched edge, but it does not have to. Let each unmatched edge and the possible matched edge form a path starting at i. If this path contains a matched edge, the path is part of either an augmentation 1b or 1c, depending on whether i is matched or not. If it does not contain a matched edge, it is part of either augmentation 1a or 1b. All four situations are depicted in Figure 6. The idea is to select the highest gain from all possible paths starting at i. Then, if i is matched, subtract the weight of the matched edge adjacent to i and

A	Adjacency matrix.
M	Matrix with $M(i,j) = w(i,j)$ if edge (i,j) is matched and $M(i,j) = 0$ (implicit) otherwise.
$[A \setminus M]$	Matrix without the edges in M, e.g. $[A \setminus M](i,j) = w(i,j)$ if (i,j) is not matched and implicit otherwise.
gain	Scalar which is equal to the maximum gain of all possible 1-augmentations, or a vector where $gain(i)$ is equal to the maximum gain of the 1-augmentations starting at vertex i if i is unmatched and implicit otherwise.
Gain	Matrix which contains the gains of the 1-augmentations centred around unmatched edge (i,j). Only has explicit values at $[A \setminus M]$.
$Gain_+$	Equal to *Gain* but only shows those elements greater than 0.
m_w	Vector with $m_w(i) = w(i,j)$ if vertex i is matched with vertex j and $m_w(i) = 0$ if vertex i is not matched. Note that $m_w(i) = 0$ is stored explicitly.
m	Vector where $m(i) = j$ if vertex i is matched to vertex j and implicit otherwise.
Aug	Matrix that contains the edges that are the centre of a 1-augmentation that can be used for improving the matching M.

Table 3: Table with names of different important variables used in this section.

take the maximum over all possible gains over all i. The result is a scalar *gain* which contains the maximum gain over all the possible 1-augmentations. Since we first take the maximum gain over all suitable paths adjacent to i and then over the corrected gain for all i, we are indeed guaranteed to find the maximum gain of the 1-augmentations, meaning that if *gain* is positive, there exist a positive-gain 1-augmentation. If *gain* is zero or negative, no positive-gain 1-augmentation exists and there is no need to search for them. Furthermore, a lower bound of 1/2 can be guaranteed.

How can this be done in GraphBLAS? This method relies on a matrix-vector and a vector-vector multiplication, both using the same semiring $\max.-$. Assume we have adjacency matrix A and the matching matrix M to our disposal. Matrix M contains the weight of edge (i,j) if edge (i,j) is matched and is zero totherwise. The first step is to make vector m_w with the weights of the current matched edges. This can be obtained by summing over all elements of row i in matrix M. Matrix M only contains one element per row, and therefore $m_w(i)$ will be equal to $M(i,k)$ if i is matched. Matrix $[A \setminus M]$ can be obtained by subtracting M from A and making the explicit zeros implicit, or more formally perform an element-wise addition using the binary operator - and pruning the explicit zeros. The selection of the best path starting at vertex i for each i can be done via the matrix-vector multiplication: $c = [A \setminus M]\max.- m_w$. To see that this works, consider

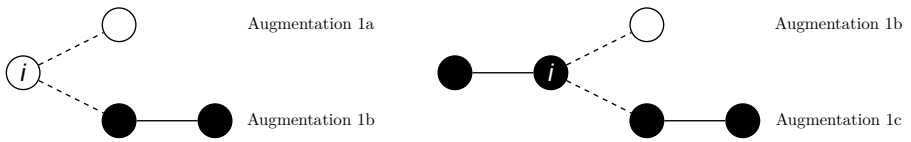


Figure 6: The different possible 1-augmentations when considering a starting vertex i adjacent to an unmatched edge.

the calculation for a single i:

$$\begin{aligned} c(i) &= ([A \setminus M] \max.- m_w)(i) \\ &= \max_j([A \setminus M](i,j) - m_w(j)) \\ &= \max_j(w(i,j) - w(j,l)\mathbb{1}_M(j,l)). \end{aligned}$$

The element $c(i)$ is indeed the maximum gain of all the paths starting at i existing of either a single unmatched edge if j is unmatched or an unmatched edge followed by a matched edge if j is matched. The second step is a vector-vector multiplication: $gain = c \max.- m_w$ which is equal to:

$$\begin{aligned} gain = c \max.- m_w &= \max_i(c(i) - m_w(i)) \\ &= \max_i[\max_j[[A \setminus M](i,j) - m_w(j)] - m_w(i)] \\ &= \max_i[\max_j[w(i,j) - w(j,l)\mathbb{1}_M(j,l)] - w(k,i)\mathbb{1}_M(k,i)]. \end{aligned}$$

This means that *gain* calculates indeed the correct maximum gain of all possible 1-augmentations.

Note that vector m_w needs to be a full vector where the zeros are stored explicit, implying more storage is needed. If the zeros would be stored implicitly, the computation $[A \setminus M](i,j) - m_w(j)$ would not be performed for those j for which $m_w(j) = 0$. This is due to the implementation of the sparse matrix-matrix multiplication. This situation can lead to an incorrect value of *gain*. To see this, consider augmentation 1a. In this case, vertex j is unmatched and therefore, augmentation 1a will not be treated in computation $[A \setminus M](i,j) - m_w(j)$. Since both vertices are unmatched, the edge (i,j) is completely ignored by the algorithm, while it might be that this edge is the only positive-gain augmentation. Therefore, it is crucial that m_w is a full vector with explicit zeros. If we want m_w to be full, we need to adjust the operation $m_w = \bigoplus_j M(:,j)$. Since matrix M is sparse, m_w will only have nonzeros at matched vertices. A solution to this is to add $\bigoplus_j M(:,j)$ to a vector with explicit zeros. For the remainder of this thesis, m_w will be a full vector with explicit zeros.

The complete algorithm can be found in Algorithm 2 and the runtime of Algorithm 2 is $O(m+n)$. To see this, note that the creation of vector m_w and the multiplication between c and m_w takes $O(n)$ time. The creation of matrix $[A \setminus M]$ takes $O(m)$ time and the multiplication of $[A \setminus M]$ by m_w can be performed in $O(m)$ due to matrix $[A \setminus M]$ being sparse and containing at most $2m$ nonzeros. Therefore, the number of operations performed in the matrix-vector multiplication is bounded by m. This gives a total time complexity of $O(m+n)$.

Algorithm 2: Termination method for 1-augmentations

Input : Adjacency matrix A, Matching M, vector z with explicit zeros
Output: A scalar *gain* equal to the maximum gain of all 1-augmentations

1 $m_w = \bigoplus_j M(:,j) + z$;
2 $[A \setminus M] = A - M$;
3 $[A \setminus M]$=GrB.prune($[A \setminus M]$);
4 $c = [A \setminus M] \max.- m_w$;
5 $gain = c \max.- m_w$;

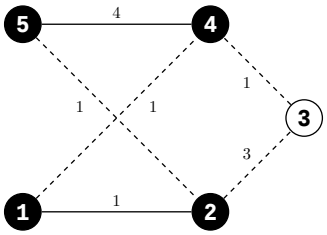


Figure 7: Example graph to illustrate the different methods in this section.

	1		1	
1		3		1
	3		1	
1		1		4
	1		4	

Figure 8: Matrix A belonging to the example graph of Figure 7. The gray squares represent the elements in $[A \setminus M]$.

2.1.2 Searching method: approach 1

In this section, we describe the first method to search for augmentations to flip. It heavily relies on the previous described termination method. It uses the same ideas, but instead of finding one number with the maximum gain, we find the maximum gain over all 1-augmentations starting from a vertex adjacent to an unmatched edge. The problem of the termination method is that the path with the maximum gain is not known. If the unmatched edge (i, j) of a 1-augmentation would be known, the whole path is determined due to the knowledge of vertices i and j being matched or unmatched. The method in this section solves the problem of the previous section by replacing line 5 of algorithm 2 by the operation $gain = c - m_w$ which gives:

$$\mathit{gain}(i) = \max_j([A \setminus M](i, j) - m_w(j)) - m_w(i).$$

The result is a vector *gain* with for each element i the maximum gain of all three 1-augmentations starting at vertex i. To select only those augmentations that are positive, only the elements are selected that have positive gain. These are stored in the vector $gain_+$. If the maximum of $gain$ is positive, a positive-gain augmentation exist. Figures 7 and 8 show a very simple graph and its corresponding adjacency matrix, respectively. This graph will be used to visualize the different steps in the algorithm. Figures 9 and 10 visualize the steps up to obtaining vector $gain$. We see that $gain(2)$ and $gain(3)$ are both positive. Since these values correspond to the same and only augmentation, we know vertices i and j and therefore could perform the flipping.

The minor change of line 5 in Algorithm 2 can give us possibly more paths with positive-gain than just the one we found in the example, since there might be multiple positive-gain 1-augmentations starting at different vertices. This might lead to an invalid matching if all are executed. We will say that a conflict occurs, if after performing an augmentation, the matching becomes invalid. In the example it is clear what index j is because there is only one positive-gain augmentation. Due to the symmetry of A, both augmentations starting at i and j are found and therefore i and j must belong together. However, if we find multiple augmentations, we cannot say anything about which vertex j belongs to i in the general case. However, this information is needed for the flipping phase. This means more computations are needed to find this vertex j and to avoid conflicts. Fortunately, both tasks coalesce quite nicely. First, we focus on finding j whereafter we say something about the conflicts.

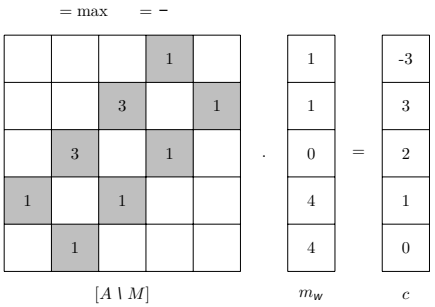


Figure 9: Visualization of $[A \setminus M]$ max. − m_w.

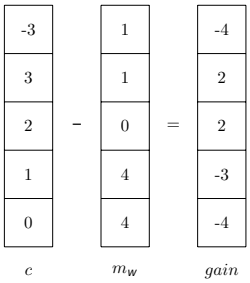


Figure 10: Visualization of $gain = c - m_w$.

The first idea that comes to mind for finding j is to use the argmax function instead of the max function. However, this is not allowed since argmax itself is not a monoid, and therefore it cannot be chosen as the operator. Therefore, we need to do more work to find vertex j. Since only the weight of the maximum-gain augmentation starting at i is known, we need to reconstruct the process of finding this path. One approach is as follows: construct a temporary matrix C which is defined as

$$C(i,j) = [A \setminus M](i,j) - m_w(j) \tag{1}$$

or in words: the entry at (i,j) is equal to the gain of the augmentation centred at (i,j) without the subtraction of the matched edge adjacent to vertex i, if applicable. Note that $[A \setminus M]$ is a matrix and m_w a vector and therefore computing C by how it is defined is not possible. To solve this issue, take the outer product between a full vector b and m_w using = second where second$(x,y) = y$. This operation makes a matrix of vector m_w such that the element at (i,j) is equal to $m_w(j)$. The choice for is less important since it is not used in the actual calculation. It is however necessary to provide , and therefore we will use = +. Since only the unmatched vertices are of interest, matrix $[A \setminus M]$ must be used as a mask to obtain a valid solution and avoid unnecessary computations. The operation b +.second m_w^T is made visible in Figure 11 and the visualization of Equation (1) can be found in Figure 12.

With this matrix C it is possible to select the elements of row i of matrix C that are equal to $c(i)$. For this, define diag(c) as a diagonal matrix with vector c on the diagonal. Then the correct elements can be selected by performing diag(c) any.eq C where eq$(x,y) = 1$ if $x = y$ and zero otherwise and where any$(x,y) = 1$ if at least one of the entries is nonzero. To see that this works consider the outcome of a single element (i,j):

$$[\text{diag}(c) \text{ any.eq } C](i,j) = \text{any}_{k \ 1:n} \text{ eq}\left([\text{diag}(c)](i,k), C(k,j)\right)$$

Note that $[\text{diag}(c)](i,k)$ is only nonzero when $i = k$. Therefore, the any function only has to consider one nonzero, which is equal to $\text{eq}\left([\text{diag}(c)](i,i), C(i,j)\right) = \text{eq}\left(c(i), C(i,j)\right)$. This function is only nonzero if $c(i) = C(i,j)$. To improve the operation, note that only for the elements $c(i)$ for which $gain_+(i) > 0$ the position in C needs to be found, meaning the number of operations is reduced. Note that it might be possible that there are multiple positions where $C(i,j) = c(i)$, meaning that for some j and j : $C(i,j) = C(i,j\ )$. In this thesis, ties are broken by selecting the element with the highest sum of the two indices. In this case, it means the element with the highest index j since i is the same for both positions. The highest index j can be found by using the max .secondi semiring

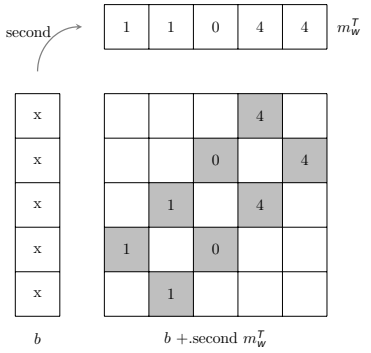


Figure 11: Visualization of the operation b +.second m_w^T with the matrix $[A \setminus M]$ used as a mask. The values of b are not used and therefore, they can be anything. This is represented by the x symbol in this vector.

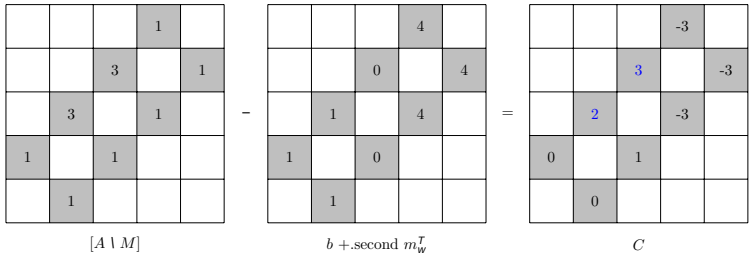


Figure 12: The visualization of Equation (1). The gray squares represent the locations of the nonzeros in mask $[A \setminus M]$. The blue numbers in matrix C represent the positions in the matrix for which the following holds: $C(i, j) = c(i)$ and $gain(i) > 0$.

in a matrix-vector multiplication between the new matrix and a full vector. Here, secondi(x, y) gives the index belonging to entry y. The above-mentioned procedure gives vertex j belonging to the best 1-augmention starting at vertex i. In fact, it calculates the argmax of matrix C. This procedure will be used again later on in this thesis when the argmax has to be computed again.

Now that vertex j can be determined, it is possible to select a set of augmentations which do not lead to conflicts. For the 1-augmentations, conflicts only occur when a vertex is part of multiple 1-augmentations. Examples can be seen in Figure 6 where performing all 1-augmentations will yield an invalid matching in both examples. These conflicts can happen in three different situations. For the first one, two different augmentations are performed starting both at vertex i. In this section using approach 1, this conflict cannot occur due to the construction of the algorithm. The second way is if there are multiple augmentations ending in the same vertex j. This seems similar to the first conflict, however this conflict is not yet dealt with. To avoid it, it is necessary to select the augmentation with the highest gain out of the augmentations ending at the same vertex. If there are ties, the augmentation with the highest index i is chosen. The last conflict occurs when a vertex i is a starting vertex and an end vertex for two different augmentations. To avoid this conflict, an augmentation centred around (i, j) can only be added to the set if the augmentation is the highest gain augmentation starting at i and ending at i. This procedure is similar to the local domination idea where an unmatched edge (i, j) is added to the matching if vertex j is the preference of vertex i and vice versa. [37]

To avoid these conflicts using GraphBLAS, consider matrix D. This matrix contains the gains of the highest (positive-)gain augmentations per row obtained from the first part of the algorithm. If there are ties, they are broken, meaning there is only one explicit value per row which avoids conflict one. The second conflict occurs when a column has multiple nonzeros. To avoid conflict two, select the highest gain augmentations for each column and break ties correctly. The matrix now also contains one element per column. Then check whether both augmentation (i, j) and (j, i) are present in the remaining matrix. If so, then the third conflict is avoided. If not, there is a

possibility that conflict three occurs and these augmentations must not be used. This checking can be done by taking the element-wise product of matrix D and D^T which takes the intersection of both matrices resulting in a matrix where if (i, j) is present, so is (j, i). The result of the intersection is a (symmetric) matrix Aug with a nonzero at the position (i, j) when this edge is a centred unmatched edge which belongs to a 1-augmentation that can be flipped.

Observe that checking for the second conflict is not strictly necessary as this is resolved by solving the third conflict. Since D only contains one augmentation per row, the one with the highest gain starting at the corresponding vertex, matrix D^T only contains one augmentation per column, the one with the highest gain ending at the corresponding vertex. Now strictly speaking D^T is not equal to matrix D after the selection procedure of the columns. However, it is still only possible to get at most one element in a row and column. Furthermore, we still select the highest-gain augmentation ending in vertex i. Assume we would not and let (i, j) be the augmentation with the highest gain in row i. Then there would be another augmentation centred around $(j\ , i)$ that is not picked with higher gain than the augmentation centred around (j, i). But then the gain of the augmentation centred around $(i, j\ )$ also must be higher than the gain of the augmentation centred around (i, j). This is not possible since (i, j) is in the original D matrix, and therefore is the augmentation with the highest gain augmentation starting at vertex i. If there are ties, the same contradiction can be deduced. Therefore, it is sufficient to look at the intersection of D and D^T and the column selection is not necessary. As a result of this, it is not necessary to let matrix D contain the gains of the augmentations around (i, j) at this point in the algorithm. Only the position of the elements are of interest and not the value of these elements.

Note that we said we are looking for a vertex-disjoint set of augmentations. This is not entirely true. It is not necessary that vertices k and l are only in a single augmentation. To see this, note that both k and l can only be in at most two different 1-augmentations. If they were in more, we would either have conflict number 1 or 2. This means that we only have to look at the situation where they are in two 1-augmentations meaning vertex l and k are also adjacent to an unmatched edge. Let i be the other vertex adjacent to such an unmatched edge. If we first flip the augmentation around (i, j), edge (j, l) or (i, k) becomes unmatched. This influences the other not yet flipped augmentation. However, this does not lead to a problem since this augmentation changes from either an augmentation 1c to an augmentation 1b or from an augmentation 1b to an augmentation 1a. Since the previously matched edge would be subtracted from the total gain of the last augmentation, this means that the new gain of this augmentation is even higher than before without this edge and more important, it is positive. Therefore, performing both augmentations does not lead to conflicts and therefore this situation is not excluded. This result can be extended for general k. When one of the two outer edges is matched, this edge can be in two different augmentations without problems.

The full pseudocode can be found in Algorithm 3. The first half of the algorithm (up to line 5) is similar to the termination method and the runtime of this part is therefore $\mathcal{O}(m + n)$. The second half of the algorithm also takes $\mathcal{O}(m + n)$. Checking whether vector $gain_+$ contains an element larger than zero takes $\mathcal{O}(n)$ since the number of nonzeros in $gain_+$ is bounded from above by n. The runtime of the creation of matrix C is bounded from above by the number of nonzeros in $[A \setminus M]$ which also can be at most $2m$. The same holds for the creation of matrix D, which despite the use of a matrix-matrix multiplication has runtime $\mathcal{O}(m+n)$. One of the matrices in the matrix-matrix multiplication is a permutation matrix. Therefore, since the number of nonzeros in C is again bounded from above by m, the multiplication is still linear in m. The other operations used to create D can be performed in $\mathcal{O}(m+n)$. Lastly, the creation of matrix Aug is also $\mathcal{O}(m+n)$

leading to a total runtime of $O(m+n)$.

Algorithm 3: Searching method 1: potential $Aug = FindkAug(A, 1, M)$

Input : Adjacency matrix A, Matching M,vector z with explicit zeros
Output: A matrix Aug which contains nonzeros at the edges which are the centre of the augmentations that can be flipped to improve the matching

$m_w = \bigoplus_j M(:,j) + z$;
$[A \setminus M] = A - M$;
$[A \setminus M]$=GrB.prune($[A \setminus M]$);
$c = [A \setminus M]$ max.$-$ m_w;
$gain = c - m_w$;
$gain_+ =$ GrB.select($gain$, positive);
if $\max_i(gain_+(i)) > 0$ **then**
  $C\ [A \setminus M] = [A \setminus M] -$ [z +.second m_w^T];
  Select $C(i,j)$ such that $gain_+(i) > 0$, $C(i,j) = c(j)$ and j is the highest index when there are ties. Store the values (i,j) in matrix D;
  $Aug =$ GrB.emult(second, D^T, D);

2.1.3 Searching method: approach 2

In this section, we describe the method behind approach 2 of searching 1-augmentations. This method first calculates the gains of all 1-augmentations, after which the best (positive-gain) augmentations are selected. This in comparison to the first method, which during the searching of the augmentations selects the best 1-augmentation. The advantage of approach 2 is that it is much easier to know vertices i and j. This approach constructs a new matrix $Gain$ where the element at (i,j) contains the gain of a possible 1-augmentation centred around edge (i,j). Or written more formally:

$$Gain(i,j) = \begin{cases} w(i,j) - w(j,l)\mathbb{1}_M(j,l) - w(i,k)\mathbb{1}_M(i,k) & \text{if } (i,j) \text{ is unmatched} \\ 0 & \text{otherwise.} \end{cases}$$

To obtain this using GraphBLAS, note again that

$$Gain(i,j) = \begin{cases} A(i,j) - (m_w(j) + m_w(i)) & \text{if } (i,j) \text{ is unmatched} \\ 0 & \text{otherwise.} \end{cases}$$

The previous expression cannot be calculated directly since A is a matrix and m_j a vector. To overcome this problem, we can use a similar method as for the computation of matrix C in approach 1. The expression $m_w(j) + m_w(i)$ needs to be transformed to a matrix which can be done by calculating the outer product of m_w with itself using $\quad = +$ and $\quad = +$. It is not necessary to compute the whole matrix m_w +.+ m_w^T since this information is only relevant where (i,j) is unmatched. Therefore, the matrix $[A \setminus M]$ can be used as a mask to avoid unnecessary computations. The visualization of m_w +.+ m_w^T can be seen in Figure 13 and with this computation, we get

$$Gain\ [A \setminus M] = [A \setminus M] - (m_w +.+\ m_w^T).$$

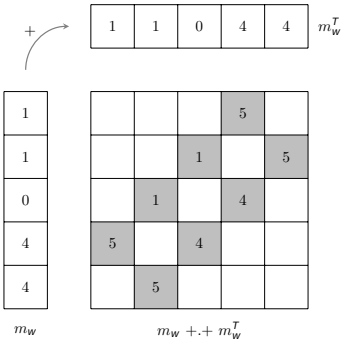


Figure 13: The visualization of $m_w +.+ m_w^T$.

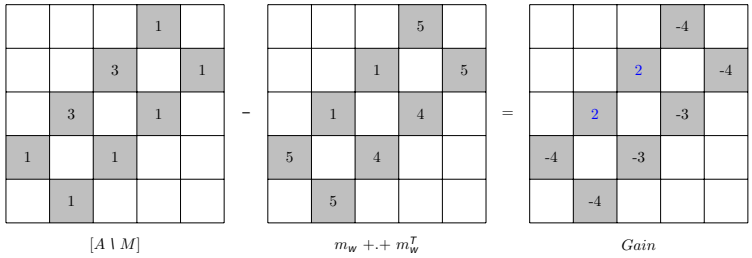


Figure 14: Visualization of $Gain = [A \setminus M] - m_w +.+ m_w^T$. The blue numbers in the matrix $Gain$ are the positive elements and represent matrix $Gain_+$.

The resulting matrix $Gain$ is a symmetric matrix which contains the gains for every 1-augmentation viewed from an unmatched edge. If there is a positive-gain 1-augmentation, it can be found. Since $Gain$ is symmetric, it contains all 1-augmentations twice. Note that to obtain the positive-gain augmentations, only the positive elements of $Gain$ need to be used. The matrix $Gain_+$ only contains these positive elements, see Figure 14 for the visualization of $Gain$ and $Gain_+$.

The next step is selecting a set of 1-augmentations such that conflicts are avoided. The same conflicts can occur in this situation as in approach 1, although the first conflict is not dealt with yet. The selection can be made by using the same approach as in searching approach 1. To construct matrix D, first calculate the maximum value per row of matrix $Gain_+$ and determine the highest index which contains this value by using a similar approach as in Section 2.1.2 when finding the argmax of matrix C. This solves conflict 1. Then proceed as in approach 1 in Section 2.1.2 to solve the other conflicts. Although the selection involves more work due to there being more 1-augmentations to consider, it also has an advantage. Since we have access to all 1-augmentations, more positive-gain augmentations can be added. After the first selection round, a second round can be performed where the edges adjacent to already selected edges are deleted from $Gain_+$. This can be repeated until no more augmentations can be added. The access to all possible 1-augmentations also gives the possibility to use a selection method based on the Suitor idea. The mean idea of how to implement Suitor in GraphBLAS and how it can be adjusted to select 1-augmentations is described in subsection 2.1.4.

The full pseudocode of searching method 2 can be found in Algorithm 4. The runtime of this algorithm is again similar to the previous two algorithms when choosing for the standard selection procedure. Line 4 can be performed in $O(m)$ whereas the selection of the augmentations can be done in $O(m+n)$. The analysis of the selection is similar to the selection of searching approach 1. Checking whether matrix $Gain_+$ contains elements larger than zero can also be done in $O(m)$. This leads to a total runtime of $O(m+n)$.

2.1.4 Suitor in GraphBLAS

In this section we describe how the idea of Suitor [30] can be implemented in GraphBLAS. This method is not described into full detail, but gives an idea of how it can be described in terms of

Algorithm 4: Searching method 2: potential $Aug = FindkAug(A, 1, M)$

Input : Adjacency matrix A, Matching M, vector z with explicit zeros
Output: A matrix Aug which contains nonzeros at the edges which are the centre of the augmentations that can be flipped to improve the matching

$m_w = \bigoplus_j M(:, j) + z$;
$[A \setminus M] = A - M$;
$[A \setminus M]$=GrB.prune($[A \setminus M]$);
$C\ [A \setminus M]\ = m_w +.+ m_w^T$;
$Gain\ [A \setminus M]\ = [A \setminus M] - C$;
$Gain_+ =$ GrB.select($Gain$, positive);
if $\max_{i,j} Gain_+(i, j) > 0$ **then**
    select correct augmentations by using either the Suitor method or the same selection procedure of approach 1 ;

GraphBLAS. In comparison to the sequential Suitor implementation, it first calculates all preferences and then determines the suitor values from all current preferences. Then it checks which edges are dead, and the process is repeated.

At the start of the algorithm, all edges in A are alive meaning they can be used to add to the matching. The edges that are alive are stored in matrix A . Let $\mathit{pref}(i)$ be the preferred partner of vertex i. This is the vertex adjacent to vertex i with the highest edge weight that is currently alive. To calculate pref, first perform a matrix-vector multiplication between matrix A and vector z which is a full vector, using the max .first semiring. This computation gives the maximum element of each row stored in the vector pref_w. Then, the positions of these maximum elements can be found by computing the argmax of matrix A whereafter the ties can be broken by picking the element with the highest index. This can be done in the same way as in Section 2.1.2. After these operations, the preferences of all vertices are known.

Now let $suitor(j)$ be the vertex adjacent to j that prefers j and has the highest edge weight of all other vertices that prefer j. All preferences are known and therefore $suitor$ can be easily computed. Let Pref be the matrix where $\mathit{Pref}(i, j)$ is equal to $A(i, j)$ if vertex i prefers vertex j. Then $suitor$ can be computed by finding the argmax over each column of matrix Pref. To see this, each column of Pref represents a vertex j that is a preference of some vertex i. If column j has no elements, vertex j is not a preference of all other vertices. By taking the maximum of each column, we find the highest weight of all edges that prefer vertex j which is what the suitor value is. Therefore, taking the argmax of the column j results in $suitor(j)$.

If $suitor(i) = \mathit{pref}(i)$, the vertex with the highest edge weight that prefers vertex i is the same as the vertex that is the preference of vertex i and therefore the edge $i, suitor(i)$ can be added to the matching. Checking whether $suitor(i) = \mathit{pref}(i)$ can be done via an element-wise multiplication, using the function eq as the binary operator.

The next step is to determine which edges will become dead edges. An edge is dead if making a proposal using this edge results in an immediate rejection. Proposing such an edge is meaningless since there is already an edge with higher weight adjacent to the vertex proposed to. An immediate rejection of edge (i, j) occurs when $w(i, suitor(i)) > w(i, j)$. Vertex j can never become a preference of vertex i since there already exist another vertex $suitor(i)$ which prefers vertex i with a higher edge weight. To find the edges for which $w(i, suitor(i)) > w(i, j)$ holds, consider the operation $B =$

diag($suitor_w$) max. $> A$ where $suitor_w(i)$ is equal to the weight of edge $(i, suitor(i))$ and implicit if this edge does not exist. The analysis of this calculation goes similar to that of diag(c) any.eq C in Section 2.1.2. The result is a matrix B which contains a one at (i, j) if $w(i, suitor(i)) > w(i, j)$ and zero otherwise. The operation can be masked by matrix A . The matrix can be used to remove the dead edges from A . Note that when edge (i, j) dies, edge (j, i) also dies. Therefore, edges in B^T also need to be removed from A .

If two vertices can be matched, it is unnecessary to compute the preferences and suitors of these vertices again in an upcoming iteration. Therefore, $pref$, $suitor$ and $suitor_w$ can be masked by $\bar{m}$ where $\bar{m}$ is the complement of vector m. This prevents unnecessary computations.

The Suitor algorithm described above describes one iteration of the standard Suitor algorithm. The algorithm terminates if $pref$ is empty. Since $pref$ is masked by $\bar{m}$, this means that every unmatched edge is not able to set a new preference, meaning there are no alive edges that can be added. The algorithm can be used for selecting 1-augmentations. To do this, change matrix A in matrix $Gain_+$. The resulting algorithm is very similar to the standard selection procedure, but now with the dead edges removed after every iteration.

2.1.5 Searching method 1 vs 2

In Sections 2.1.2 and 2.1.3 we described two different methods to search for positive-gain 1-augmentations. Both methods are very similar to each other and take $O(m + n)$ time. Therefore, determining which one to use cannot be determined by the time complexity. In this section, we describe some advantages and disadvantages of both methods.

The main advantage of approach 2 is that all 1-augmentations are known and that therefore a larger selection of augmentations can be made to flip within a single main iteration. When the selection procedure ends after one selection iteration, both methods find the same set of 1-augmentations. This can easily be seen since both methods calculate the best 1-augmentation starting at vertex i for all i before selecting the best 1-augmentations ending at vertex j. Approach 1 does this immediately and approach 2 searches for the best 1-augmentation starting at vertex i after calculating all possible 1-augmentations. Both methods do not differ much in this case. Both approaches have similar runtime since they use very similar operations. The outcome of approach 2 needs more storage since all 1-augmentations are stored whereas in approach 1 the outcome is only the best augmentation starting at each vertex. Approach 2 is slightly more elegant in the sense of that vertex j can be determined quite easily whereas approach 1 might be more intuitive in terms of the operations used. Another advantage of approach 2 is that the second-best augmentation can be found easily. In approach 1, the algorithm needs to be repeated with the best centre edges removed from A whereas in approach 2 the second-best augmentation can be found by consulting matrix $Gain_+$ directly. This observation is of importance for when the 2-augmentations and 3-augmentations are considered later on in this thesis.

The two methods do differ when more selection iterations are used in approach 2. Each main iteration, the set of augmentations can be extended in approach 2. When considering only 1-augmentations, this will lead to the same final matching. To see this note that positive-gain augmentations 1b and 1c are never found. When starting with an empty matching, the first iteration will always only consists of augmentation 1a. Let edge (i, j) be such an augmentation 1a. Then all edges adjacent to vertex i and j except edge (i, j) have a lower or equal weight to edge (i, j). This is due to the construction of the algorithm. Edge (i, j) was the edge with the highest weight starting at vertex i and j, otherwise it could not have been added to the matching. Due to

this observation, every edge $(i, i\ )$ will have $w(i, i\ ) \leq w(i, j)$ and therefore every augmentation 1b or 1c starting at vertex i centred around an unmatched edge $(i, i\ )$ will always have negative or zero gain. This means that these augmentations do not occur in matrix $Gain_+$ meaning that $Gain_+$ only consist of augmentations 1a. This means that calculating matrix $Gain_+$ again in a new main iteration only means deleting the edges adjacent to an already matched edge from the matrix. This is exactly what happens when performing more selection iterations but without calculating matrix $Gain_+$ again. Therefore the two methods give the same outcome. As a consequence of this, when only performing 1-augmentations it is expected to be faster to repeat the selection iterations until no new augmentation can be found.

The same statement does not hold when combining 1-augmentations with higher k-augmentations. The order in which 1-augmentations and augmentations with k edges not in M are flipped, influences the final matching. All that can be said about this final matching is that it fulfils the lower bound if no k-augmentations are found.

Finding a larger set of augmentations to flip each iteration might suggest needing fewer iterations overall. Especially when the graph has the structure of a chain or when many vertices prefer the same end vertex, it can be beneficial to perform a second (or more) selection iteration. This prevents calculating the same best augmentations over en over again without using them due to there being better augmentations that end in the end vertex. However, it is not known how many selection iterations are needed to find a maximal set of augmentations. This number is upper bounded by the number of vertices and therefore, the main iteration would lose its linear runtime if the selection iterations stop when no augmentations can be added. It is possible to fix the number of selection iterations in each main iteration to hopefully get a better matching sooner.

Overall, both methods are very alike. Approach 2 has the slight advantage in terms of using it for finding the second-best augmentation, although this comes at the cost of needing more storage. Since this is needed for 2-augmentations and 3-augmentations and in combination with the possibility to extend the set of augmentations without performing the main iteration again, this approach is the preferred approach theoretically.

2.1.6 Flipping phase

In this section the flipping phase of the found and selected 1-augmentations is described. For general k-augmentations, the method is very similar and therefore, some remarks will be made for general k such that it is not needed to describe this method for higher k.

The result of the two searching methods are the (unmatched) edges (i, j) which are the centre edges of the 1-augmentations that need to be flipped. If i or j are matched vertices, the edges (i, l) and (j, k) are known and available in vector m. This vector can be obtained by performing M max.secondi z where secondi gives the index of the second entry and z is a full vector. Using this semiring gives the highest index of the explicit values per row. In the situation of the 1-augmentations, flipping means adding edge (i, j) to the matching and if applicable deleting edges (i, l) and (j, k). The edges that need to be added are available in matrix Aug and if $Aug(i, j)$ is equal to $w(i, j)$ we can just perform $M + Aug$ to add the new matched edges. To achieve that $Aug(i, j)$ is equal to $w(i, j)$, we can perform an element-wise matrix multiplication between Aug and A using the binary operator second where Aug is used as a mask to prevent unnecessary and incorrect calculations. Now, let vector aug contain vertex j at position i if the augmentation around edge (i, j) needs to be flipped and zero otherwise. This vector can be obtained by Aug max.secondi z. Then, the intersection between vector aug and m represents the matched edges which are adjacent

to a centre edge in Aug and thus it gives which edges need to be deleted from M with the remark that also edges (k, j) and (l, i) need to be removed from M. The pseudocode for $k = 1$ can be found in Algorithm 5.

Algorithm 5: Flipping method: $M = Flipping(M, Aug, 1)$

Input : Matrix Aug with the needed augmentations, Matching M, vector z with explicit zeros.

Output: An updated matching M with higher gain.

```
1 m = M max .secondi z;
2 aug = Aug max .secondi z;
  // The foreach loop can be performed for all i at the same time
3 Foreach i in V do
4 |   if m(i) and aug(i) explicit then
5 |   |   Remove edges (i, m(i)) and (m(i), i) from M;
6 Aug = GrB.emult(second, Aug, A);
7 M = Aug + M;
```

Now some remarks for general k. The k-augmentations consist of the $(k - 1)$-augmentations and the augmentations with exactly k edges not in M. Assuming we know how to flip $(k - 1)$-augmentations, only the augmentations with exactly k edges not in M are of interest. The difference between the flipping for general k and $k = 1$ is not big. Assume that we know the whole path by knowing the centre edge and which unmatched edges are in an augmentation. The centre edge is either a matched or unmatched edge which depends on k being even or odd. If k is odd, the centre edge is unmatched and this edge needs to be added to the matching. The adjacent edges to the centre edge are matched and need to be removed, the edges adjacent to these matched edges need to be added until the whole path is treated. If k is even, the centre edge is matched, and thus it must be removed and the adjacent unmatched edges in the augmentation need to be added etc. The adding and deleting procedure can be done in a similar way as for $k = 1$ given that the unmatched edges are provided. It does however depend on how this information is stored.

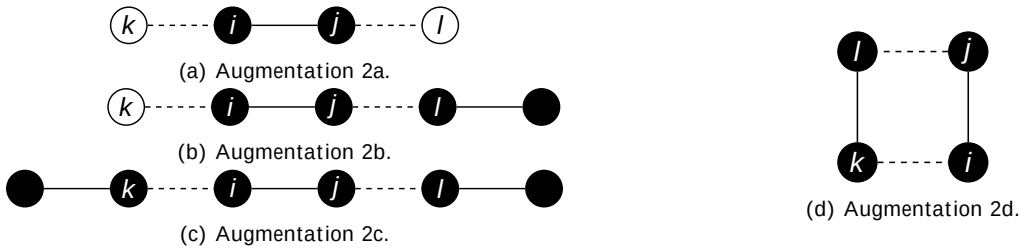


Figure 15: The four augmentations 2 with labelled vertices.

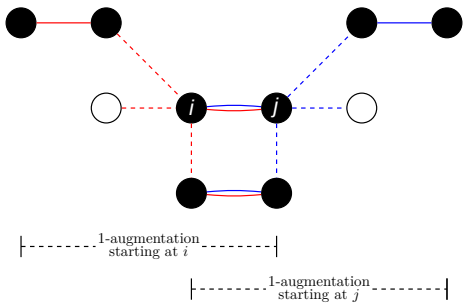


Figure 16: Visualization of a matched edge (i, j) being part of two 1-augmentations. The edges belonging to a possible 1-augmentation starting at vertex i are depicted in red and those starting at vertex j are blue. A combination of two 1-augmentations yields an augmentation 2 and all augmentations 2 can be created. When there are two edges between two vertices, this means that this edge is in both the 1-augmentations.

2.2 2-augmentations

In this section, the method for detecting positive-gain 2-augmentations is described. A 2-augmentation is either a 1-augmentation or an augmentation with exactly two unmatched edges not in M. To shorten the name of the augmentations with exactly two unmatched edges not in M, they will be called augmentations 2. Since the 1-augmentations are treated in section 2.1, the focus in this section lies on finding augmentations 2. The four augmentations 2 and the labelling used in this section are shown in Figure 15. Each augmentation is centred around a matched edge (i, j) and has an unmatched edge adjacent to both vertex i and j. The difference between the four augmentations is whether these unmatched edges have another matched edge adjacent to them or not. Note that augmentation 2d is a special case of augmentation 2c where the first and last matched edge of augmentation 2c are the same.

The main idea behind the algorithm for finding (positive-gain) augmentations 2 is to use the observation that each augmentation 2 is centred around a matched edge (i, j) and that both vertices i and j have a path of at least length one adjacent to them. These paths including edge (i, j), are equal to either an augmentation 1b or 1c. Therefore, an augmentation 2 consists of two 1-augmentations with an overlapping matched edge. A visualization of this can be seen in Figure 16. If we find the best augmentation 1b or 1c for the two vertices i and j, the gain of the best augmentation 2 centred around matched edge (i, j) can be found by combining the gains of these 1-augmentations and making a correction for matched edge (i, j). This correction is needed since edge (i, j) is subtracted twice from the total gain, and therefore it needs to be added again. The best 1-augmentation can

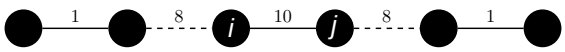


Figure 17: Example of a positive-gain augmentation 2 with gain 4. The gain of both 1-augmentations starting at i and j is -3. When the negative-gain 1-augmentations would have been ignored, this augmentation would not be found.

have negative or zero gain and in contrast to the previous section, these cannot be discarded. The reason for this is to make sure no positive-gain augmentation 2 exists if none are found. To see this, take for example the augmentation 2 in Figure 17. Both 1-augmentations have a negative gain of -3. However, the total gain of the augmentation 2 is positive. This augmentation would not have been detected if the negative-gain 1-augmentations had been discarded. Another observation is that a 1-augmentation starting at vertex i can only be part of at most one (best) augmentation 2. This is due to the centred edge (i, j) being matched and therefore the choice of vertex i also determines vertex j. This observation implies that changing one of the two 1-augmentations only has effect on one augmentation 2. This will be important when conflicts are solved.

Finding the best augmentation 1b or 1c can be done by using a slightly adapted algorithm for finding 1-augmentations. Instead of finding the best 1-augmentation starting at all vertices i, only the matched vertices i need to be considered. This can be obtained by using vector m as a mask. Both searching methods for finding 1-augmentations can be used, both without the full selection procedure. It is sufficient to know the best 1-augmentation starting at each matched vertex i even if there might occur conflicts when multiple are used at the same time. These conflicts are treated and solved later on in this section. To calculate the gain of the best augmentation 2 centred around (i, j), we need to access the gains of the best 1-augmentation which are stored in vector aug_{gain}. For the flipping phase, the end vertices of the best 1-augmentations starting at vertex i are also needed and are therefore stored in vector aug. These end vertices are vertices k and l in Figure 15. With the information stored in vector aug_{gain} and matrix M, the maximum gain for each centred matched edge (i, j) can be calculated in matrix $Gain2$:

$$Gain2(i,j) = \begin{cases} aug_{gain}(i) + aug_{gain}(j) + M(i,j) & \text{if } (i,j) \quad M \\ 0 & \text{otherwise.} \end{cases}$$

Here, the matrix $Gain2$ can be constructed in the same way as the $Gain$ matrix in searching method 2 and therefore it can be written as:

$$Gain2\ M\ = (aug_{gain} +.+\ aug_{gain}^{T}) + M.$$

Matrix $Gain2$ contains the maximum gain for each centred matched edge (i, j). However, it might be that a conflict occurs within a single augmentation 2 or that the gain is not calculated correctly. We first consider the first situation. The only conflict within a single augmentation 2 that can occur is when the two 1-augmentations starting at vertex i and j have the same end vertex. This means $k = l$ or in terms of vector aug: $aug(i) = aug(j)$ and the situation can be seen in Figure 18. When this conflict occurs, it cannot simply be ignored without searching for another augmentation 2 centred around edge (i, j). It might be that there exist another augmentation 2 centred around this edge which also has positive gain and if this is the only positive-gain augmentation left in the current matching, it is not found and the algorithm fails. This conflict can be solved by keeping one of the two 1-augmentation fixed and adjusting the other 1-augmentation. By doing this, the situation cannot occur again. There are two possible choices to do this: keep the 1-augmentation

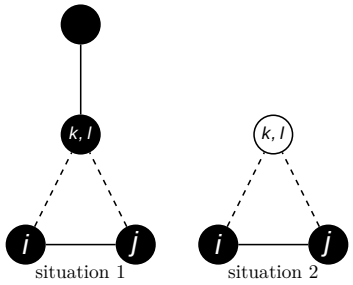


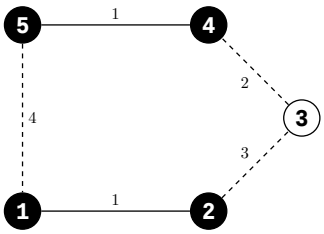
Figure 18: Invalid augmentation after selecting the best 1-augmentations for vertices i and j. Vertices k and l are the same. In situation 1 vertex $k = l$ is matched and in situation 2 vertex $k = l$ is unmatched.

starting at i fixed and adjust the 1-augmentation starting at j or fix the 1-augmentation starting at j and adjust the 1-augmentation starting at i. Both situations give a different augmentation 2. To ensure that a positive-gain augmentation can be found if it exists, both possibilities need to be included in the algorithm. This can be done by first adjusting the 1-augmentation belonging to the smallest index ($i < j$). If after computing $Gain2$, no positive-gain augmentation is found, the procedure is repeated for the highest index vertices ($i > j$). If again no positive-gain augmentation is found, no augmentation 2 exist centred around (i, j). This approach will not always lead to the best possible augmentation centred around edge (i, j). However, finding a positive-gain augmentation 2 is sufficient at this point.

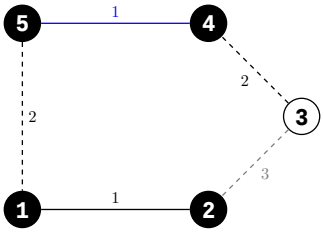
The second problem occurs when augmentation 2d is found. In the current procedure, edge (k, l) is subtracted twice from the total gain and therefore a correction needs to be made. This situation can be seen in Figure 16 where the two matched edges are both counted twice in augmentation 2d. For edge (i, j) an adjustment is already made in the calculation of $Gain2$. The adjustment for edge (k, l) can be done similarly. First, compute the best 1-augmentation for each vertex i and check for conflicts within a single augmentation. Then, if it holds that $m(k) = l$, augmentation 2d occurs and the weight of edge (k, l) needs to be added to $Gain2(i, j)$. This can be done by adding the weight of edge (k, l) to the gain of one of the two 1-augmentations which is stored in vector aug_{gain}. In our algorithm we adjust the gain of the 1-augmentation with the highest starting index.

These two situations are the only problems that can occur within a single augmentation 2. Both adjustments can be performed before the calculation of matrix $Gain2$. The only exception to this is when the first conflict occurred and $Gain2$ has no positive entries. Then matrix $Gain2$ needs to be computed again using the adjusted vector aug_{gain}. The matrix $Gain2_+$ now only contains the augmentations centred around matched edges with positive gain. If $Gain2_+$ does not contain any nonzeros, there does not exist an augmentation 2 in the graph with the current matching M. If there exists multiple augmentations 2, a number of augmentations need to be selected which do not lead to conflicts if they are performed together. This means that vertices i, j, k and l can only be part of one and the same augmentation 2. Note that vertices $m(k)$ and $m(l)$ do not need to be included in this statement due to the same reasoning as in section 2.1.

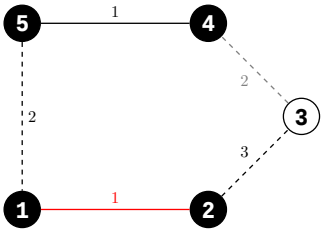
To select a set of valid augmentations, we use a similar but slightly different approach to the selection phase of the 1-augmentations. For each start vertex i of a positive-gain augmentation 2, find vertices j, k and l of the best positive-gain augmentation. These indices of the best augmentation 2 starting at i can be found by using matrix M and vector aug. Then, a new matrix $Aug2$ is constructed which contains the gain of the best augmentation 2 starting at i at positions (i, i), (i, j),



(a) Simple example graph.



(b) Augmentation 2 centred around edge $(4, 5)$. The centred edge is blue and the edge not belonging to the augmentation 2 is coloured gray. The gain of this augmentation 2 is 2.



(c) Augmentation 2 centred around edge $(1, 2)$. The centred edge is blue and the edge not belonging to the augmentation 2 is coloured gray. The gain of this augmentation 2 is 3.

Figure 19: Simple example graph for the visualization of matrix *Aug*2 and the two possible augmentations 2 in the example graph highlighted.

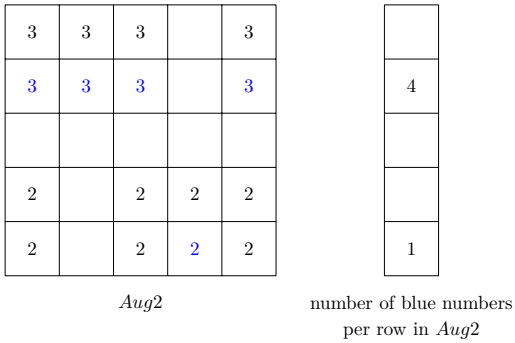


Figure 20: Visualization of matrix *Aug*2 and the vector with the number of nonzeros per row belonging to the example of Figure 19. The blue numbers in matrix *Aug*2 are the maximum values per column and if there are ties, the value with the highest index is chosen.

(i, k) and (i, l). It can now be checked if a vertex is in multiple best augmentations 2 and a selection can be made. To do this, select the highest value in each column and in case of ties, select the one with the highest index. This corresponds to selecting the highest gain augmentation 2 for each vertex. If after this selection procedure a row still contains four nonzeros, the augmentation can be used in the flipping phase. If not, the augmentation cannot be used since at least one of the four vertices is part of a better augmentation 2 which will be used. To visualize this approach, consider Figure 19. This figure shows a simple graph and the two augmentations 2 in the graph. Flipping both augmentations gives a conflict at vertex 3. Figure 20 visualizes matrix $Aug2$ and shows the number of nonzeros after the selection of the columns. In this example, the augmentation centred around edge $(2, 1)$ can be flipped, which has indeed the highest gain of the two augmentations. Note that it is not necessary to add a row for both vertices i and j. Adding one of the rows is sufficient since the best augmentation 2 starting at vertices i and j is the same by symmetry of matrix $Gain2_+$. Therefore, rows i and j will both contain elements i, j, k and l and are therefore the same. The selection procedure can be repeated to extend the set of augmentations, but this is

not necessary. If it is repeated, the rows with an already used vertex in it must be removed from $Aug2$.

The pseudocode for the described algorithm for finding and selecting augmentations 2 can be found in Algorithm 6. A single iteration of finding and selecting takes $O(n+m)$. To see this, note that the algorithm relies on the algorithm for finding 1-augmentations which also has $O(n+m)$. This algorithm is performed at most 3 times. The first time is to find matrix Aug and the other times are to find the second-best 1-augmentations when a conflict occurs. Checking whether this conflict or an augmentation 2d occurs can be done in $O(n)$ by first permuting a vector and then comparing two vectors element-wise. Constructing matrix $Gain2$ takes $O(|M|)$ which can be bounded by $O(m)$. Lastly, creating matrix $Aug2$ and selecting the highest elements in each column can also be done in $O(n+m)$. Therefore, the total runtime is linear in the size of the graph.

The augmentations 2 found can be flipped using the approach described in section 2.1.6. To ensure that no positive-gain 2-augmentations remain, all 1-augmentations and augmentations 2 need to be negative or zero gain. However, at the moment it is not possible to flip them at the same time, since there is no check to avoid conflicts between 1-augmentations and augmentations 2. This check could be done by using the same selection procedure of augmentation 2. For each starting vertex i, determine the best 1-augmentation and add i and $aug(i)$ to matrix $Aug2$ in row i. Note that row i might already contain elements for an augmentation 2 since the 1-augmentations found in Section 2.1 can also be part of an augmentation 2. If so, chose the augmentation with the highest gain. If after the column selection, there are two nonzero elements in a row of a 1-augmentation, this 1-augmentation can be flipped safely. Overall, this selection procedure is more complicated in comparison to just repeating the two separate algorithms for 1-augmentations and augmentations 2, although the set of found augmentations might be smaller in the latter case.

Algorithm 6: Searching method for augmentations 2

Input : Adjacency matrix A, matching matrix M, vector m_w and a vector z with explicit zeros

Output: A set of paths determined by $i, m(i), aug(i)$ and $aug(m(i))$ which can be flipped

1 $Aug = FindkAug\ (A, 1, M)$; `// FindkAug is an adapted version of` $FindkAug$

2 $aug = Aug+.\text{secondi}\ z$;

3 $aug_{gain} = \bigoplus_j Aug(:, j)$;

`// All if and for statements can be performed for all` i `at the same time`

4 **if** $aug(i) = aug(m(i))$ **then**

5 | **for** $i < m(i)$ **do**

6 | | Repeat lines 1 till 3 to find the second-best 1-augmentation starting at vertex i;

7 **if** $m(aug(i)) = aug(m(i))$ **then**

8 | **for** $i > m(i)$ **do**

9 | | $aug_{gain}(i) = aug_{gain}(i) + m_w(aug(i))$;

10 $Gain2\ M\ = (aug_{gain} +.+\ aug_{gain}^T) + M$;

11 $Gain2_+ = \text{GrB.select}(Gain2, \text{positive})$;

12 **if** $Gain2_+$ *empty and for some vertex* i*:* $aug(i) = aug(m(i))$ **then**

13 | **for** $i > m(i)$ **do**

14 | | Repeat lines 1 till 3 to find the second-best 1-augmentation starting at vertex i;

15 | | Repeat lines 9 till 15;

16 **if** $Gain2_+$ *not emtpy* **then**

17 | Create matrix $Aug2$;

18 | Select highest value in each column of $Aug2$ and remove others from $Aug2$;

19 | **if** *Row* $Aug2(i, :)$ *contains four nonzeros* **then**

20 | | Add $i, m(i), aug(i)$ and $aug(m(i))$ to the set for flipping;

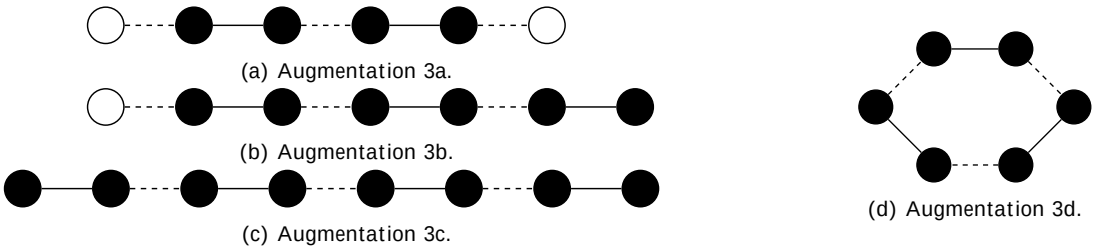


Figure 21: The four augmentations 3.

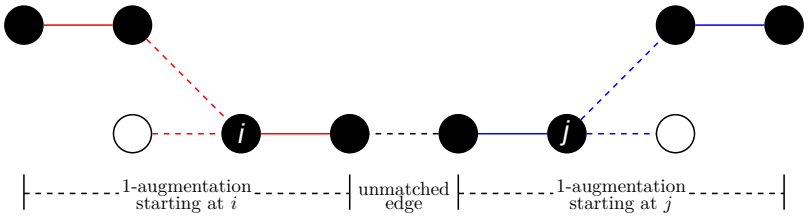


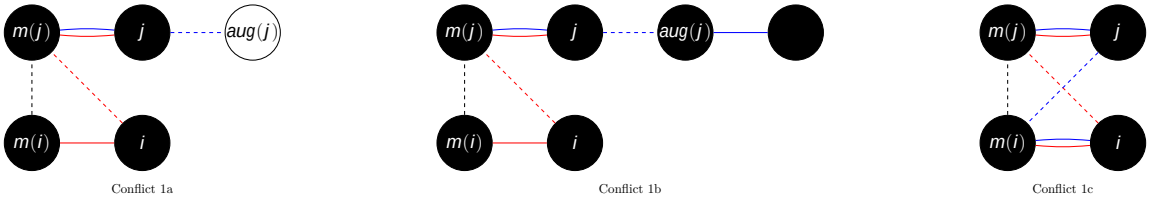
Figure 22: Visualization of the construction of augmentations 3. Each augmentation 3 can be constructed from two 1-augmentations starting at matched vertices i and j and an unmatched edge $(m(i), m(j))$. The red edges correspond to the edges in the 1-augmentation starting at i and the blue edges to the edges in the 1-augmentation starting at j. Augmentation 3d is not depicted in this figure, but it can be constructed by letting the two outer matched edges overlap.

2.3 3-augmentations

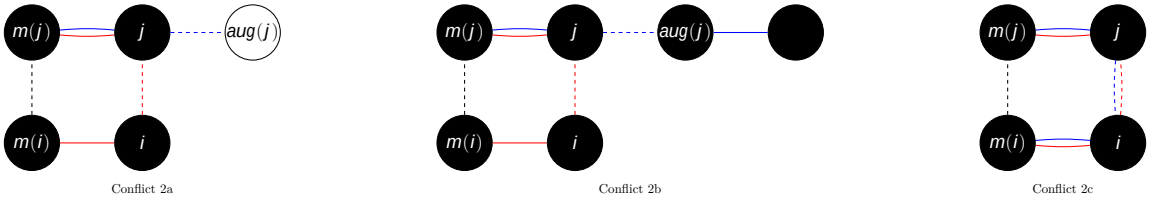
In this section we describe an idea to treat the 3-augmentations. This method is not described in full detail. It does however give a good overview of how these augmentations can be treated knowing how to find 1-augmentations. Similar to the 2-augmentations, the 3-augmentations exist of 1-augmentations, augmentations 2 and the augmentations with exactly 3 unmatched edges not in the matching M. The latter ones will be called augmentations 3 and are the ones that are treated in this section. All augmentations 3 can be seen in Figure 21.

Each augmentation 3 can be seen as two augmentations 1b or 1c connected with an unmatched edge. This unmatched edge is adjacent to two matched edges belonging to the 1-augmentations and is the centre edge of an augmentation 3. A visualization of this can be seen in Figure 22. Matched vertices i and j are both starting vertices of an augmentation 1b or 1c and therefore the unmatched centre edge of augmentation 3 is equal to $(m(i), m(j))$. The procedure for finding the best augmentation 3 centred around each unmatched edge is as follows: first, calculate the best 1-augmentation starting at each vertex. Then, construct an augmentation 3 centred around edge $(m(i), m(j))$ with the best 1-augmentations starting at i and j. The total gain of this augmentation 3 is equal to the sum of the gain of the two 1-augmentations and the weight of the unmatched edge and is by construction the best possible augmentation 3 centred around $(m(i), m(j))$. The gains of all augmentations 3 are stored in matrix $Gain3$ which is defined as:

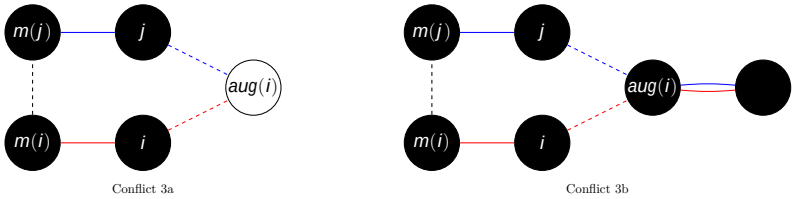
$$Gain3(m(i), m(j)) = \begin{cases} aug_{gain}(i) + aug_{gain}(j) + A(m(i), m(j)) & \text{if } (m(i), m(j)) \quad E \\ 0 & \text{otherwise.} \end{cases}$$



(a) The three possible categories of conflict 1 where $aug(i) = m(j)$.



(b) The three possible categories of conflict 2 where $aug(i) = j$.



(c) The two possible categories of conflict 3 where $aug(i) = aug(j)$.

Figure 23: All possible conflicts for the augmentations 3. The red edges belong to the 1-augmentation starting at vertex i and the blue edges to the 1-augmentation starting at vertex j. When there are two edges between two vertices, this means that this edge is in both the 1-augmentations.

The construction of matrix $Gain3$ is very similar to that of matrices $Gain$ and $Gain2$ with the difference that vector aug_{gain} needs to be permuted first such that the value at position $m(i)$ is equal to $aug_{gain}(i)$. Note that a single 1-augmentation starting at vertex i can be in multiple augmentations 3. If there exists a 1-augmentation starting at matched vertex j and edge $(m(i), m(j))$ exists, there are two augmentations 3 containing the 1-augmentation starting at vertex i. This is in contrary to augmentations 2, where a single 1-augmentation starting at vertex i could only be in a single augmentation 2. This observation has consequences for how the conflicts within a single augmentation 3 can be resolved as will be seen later on.

An augmentation 3 is longer than an augmentation 2, meaning it has more possibilities to intersect with itself which leads to more conflicts. The conflicts can be divided into three main categories, each with a number of subcategories. All possible conflicts can be seen in Figure 23. The red edges belong to the 1-augmentation starting at vertex i and the blue ones to the 1-augmentation starting at j. Conflict 1 occurs when $aug(i) = m(j)$ and therefore can only happen when the 1-augmentation starting at vertex i is an augmentation 1c. In conflict 1a, the 1-augmentation starting at vertex j is an augmentation 1b, whereas in conflict 1b this is an augmentation 1c. Conflict 1c is a special case of conflict 1b where it also holds that $aug(j) = m(i)$. Conflict 2 occurs when $aug(i) = j$. This conflict can only happen when the 1-augmentation starting at vertex i is an augmentation 1c since j must be matched. In conflict 2a, the 1-augmentation starting at vertex j

is an augmentation 1b, whereas in conflict 2b this is an augmentation 1c. Conflict 2c is a special case of conflict 2b where it also holds that $aug(j) = i$. For conflict 3, $aug(i) = aug(j)$ must hold. In conflict 3a, $aug(i)$ is unmatched whereas in conflict 3b $aug(i)$ is matched.

To solve the conflicts within a single augmentation, we use a similar approach to the augmentations 2 algorithm. First, we calculate the best 1-augmentations starting at each vertex. Then, we check if one of the three conflicts occur. If so, the 1-augmentation with the lowest index is changed into the second best 1-augmentation and the one with the highest index stays the same. In the algorithm for augmentations 2, the next step would be to adjust vectors aug and aug_{gain} and make matrix $Gain3$. This is not possible for these augmentations 3. A single 1-augmentation starting at vertex i can be in multiple augmentations 3 and we only want to change an augmentation 3 if a conflict occurs. When this is not the case, the augmentation must be kept the same. This distinction depends on vertex j. To solve this issue, the idea is to first make matrix $Gain3$ with the original vector aug_{gain}, then adjust aug_{gain} as would be done in augmentations 2 and adjust $Gain3$ for those centred edges for which a conflict occurs.

To find the centred edges for which a conflict occurs, a mask can be used. This mask is a matrix which contains a one at position $(m(i), m(j))$ if one of the three conflicts occurs and zero otherwise. The next part will give a outline of how this mask can be made and the casualties that occur. This is explained for the first conflict but can be used for the others. The creating of the mask relies on the operation aug +.eq m^T which gives the position (i, j) for which $aug(i) = m(j)$. Note that it gives the position (i, j) where we want to know the position $(m(i), m(j))$. This can be solved by permuting the obtained matrix. Another important observation is that position $(m(j), m(i))$ is not found, as $aug(j) = m(i)$ in the general case. However, the augmentation centred around (j, i) does need to be adjusted since it still gives a conflict. Another problem is that the index of the end vertex of the augmentation starting at i ($aug(i)$) depends on vertex j. Therefore, the current way of storing this information is not sufficient since it currently can store only one value. Fortunately, all these problems can be solved with a little extra work, meaning the conflicts can be dealt with relatively easily.

As mentioned all three conflicts can be checked and adjusted for at the same time. However, after one round of adjustments, it is necessary to check for conflicts again. There are multiple conflicts that can happen, and therefore it might be that another conflict occurs after adjusting for the first. This takes at most three adjustments since we can obtain each conflict at most once per centred edge $(m(i), m(j))$ if the other 1-augmentation is kept fixed. When no positive-gain augmentation 3 is found, the same procedure needs to be repeated with the 1-augmentation with the highest index changed into the second-best 1-augmentation and where the augmentation with the smallest index stays the same.

Similar to augmentation 2d, the gain of augmentation 3d is not calculated correctly. After checking for conflicts within a single augmentation, an adjustment can be made for those augmentations. This augmentation occurs when $aug(i) = m(aug(j))$ holds. This adjustment can be made in a similar way as the adjustment for the conflicts described above.

Now that matrix $Gain3$ contains only valid augmentations with the correct gains, a selection of augmentations 3 has to be made such that no conflicts between multiple augmentations occur. Of course, only the augmentations 3 with positive gain need to be considered and when there are no augmentations 3, there exist no augmentations 3 with the current matching M. The selection can be done in the same way as described in Section 2.2 for augmentations 2. This time, a matrix is formed with the best augmentations 3 starting at vertex i for all i. This means that the constructed matrix contains the gain of the best augmentation 3 starting at i at positions

$(i,i),(i,j),(i,m(i)),(i,m(j)),(i,aug(i))$ and $(i,aug(j))$. An augmentation can be added to the flipping phase if after the selection of the columns there are six nonzeros in each row. Special care needs to be taken for the value of $aug(i)$, since this now depends on j.

The time complexity of the augmentations 3 is $O(m+n)$ and its analysis is similar to that of augmentations 2. The main components of the algorithms are the same such as first calculating the best 1-augmentations and combining them to augmentations 3. There are a couple of main differences. The first is that there are more centred edges since the centre edge is unmatched. This can result in more edges for which $Gain3(m(i),m(j))$ needs to be computed. Furthermore, in many of the stages of the algorithm, a vector needs to be permuted to get the correct solution. Also, there are more conflicts to take care of, taking at most three rounds of adjustments. Lastly, the selection matrix to select a set of multiple augmentations 3 contains more nonzeros than the same matrix for augmentations 2. However, these changes do not influence the time complexity, and therefore it is still $O(m+n)$.

3 Experimental results

The approximation algorithm has been tested on performance in terms of runtime and quality of the obtained matching. A distinction has been made between the results of the 1-augmentations algorithm and the 2-augmentations algorithm. The graphs used for testing are obtained from the SuiteSparse collection [9] and their properties are shown in Table 4. If the adjacency matrix of the graph was nonsymmetric, the lower trianguler part of the matrix was used in the calculations. All used matrices have positive explicit values.

All experiments were run on the same computer with an Intel(R) Core(TM) i7-10750H processor at 2.60GHz using 16 GM of RAM, running Windows 10. We used MATLAB version R2020b using the MATLAB interface of SuiteSparse:GraphBLAS version 4.0.3 [8].

To test the quality of the matching, we define the gap to optimality. This gap indicates the difference between the weight of the maximum matching and the weight of the approximation and is defined as

$$\left(1 - \frac{w(M)}{w(M\ )}\right) \cdot 100.$$

Calculating this gap can give problems since computing the exact optimal matching can become very time-consuming when the graphs get bigger. However, if we want to show that our algorithm obtains a 1/2- or 2/3-approximation, we need to know the exact maximum matching M . For this reason, a few smaller graphs have been added for which the optimal solution could be computed in reasonable time using [39]. For these graphs, the exact gap to optimality can be computed. For the other test graphs, the following upper bound [5] can be used to estimate the weight of the maximum matching:

$$w(M\ ) \leq \frac{1}{2} \sum_{i\ \ V} \max\{w(i,j) : j \quad V \text{ and } (i,j) \quad E\}.$$

If this upper bound is used in the gap to optimality to replace the maximum weight, the estimated gap to optimality will be an upper bound of the exact gap to optimality. This means that if the estimated gap implies a 1/2-approximation, the algorithm is indeed a 1/2-approximation. However, if it does not, this does not mean the algorithm fails automatically since the estimated gap to optimality might be too pessimistic. Despite this detail, it can be used to say something about the difference in quality between the different matchings obtained from the 1-augmentations and 2-augmentations algorithm.

3.1 1-augmentations

In this section we discuss the results obtained from the algorithms for 1-augmentations. The main results can be seen in Tables 5 and 6. We tested a few different setups. Both searching method approach 1 and searching method approach 2 were tested. Since both methods give the same matching, the gap to optimality is only mentioned once. The main advantage of approach 2 is the ability to extend the set of augmentations used in the flipping phase in a single main iteration. Therefore, we included the results of approach 2 where some additional selection iterations were performed in each main iteration. If it was possible to extend the set of augmentations with the available augmentations, additional selection iterations were performed, up to a total of two, five or ten selection iterations. These methods are denoted as approach 2-2, approach 2-5 and approach

Matrix name	Number of vertices	Number of edges	Description
orsirr_2	886	2542	Computational Fluid Dynamics Problem
saylr4	3564	9376	Computation Fluid Dynamics Problem
Binaryalphadigs_10NN	1404	9696	Undirected Weighted Graph
G22	2000	19990	Undirected Random Graph
MISKnowledgeMap	2427	28511	Undirected Weighted Graph
cond-mat	16726	47594	Undirected Weighted Graph
foldoc	13356	59693	Directed Weighted Graph
hi2010	25016	62063	Undirected Weighted Graph
har_10NN	10299	75868	Undirected Weighted Graph
astro-ph	16706	121251	Undirected Weighted Graph
sd2010	88360	205361	Undirected Weighted Graph
cage11	39082	260320	Directed Weighted Graph
appu	14000	919552	Directed Weighted Random Graph
kron_g500-logn16	65536	2456071	Undirected Multigraph
gupta3	16783	4653322	Optimization Problem

Table 4: The used test matrices sorted by the number of edges. All are obtained from the SuiteSparse collection [9].

2-10 respectively. In the next two sections, the runtime and the quality of the matching and its behaviour in time are discussed in more detail.

Quality of the matching

In this section we discuss the quality of the matching after performing the algorithm for 1-augmentations. First, we start with the analysis of the results of the graphs for which an exact gap to optimality could be calculated. Table 6a shows these gaps to optimality for five relatively small graphs. All five gaps are less than 50% and therefore the found matchings are indeed 1/2-approximations as should be by design of the algorithm. Notable is that all gaps are quite small. The quality of the matchings is therefore much higher than the guaranteed lower bound. The matching of graph saylr4 is of very high quality, having a gap of 0.0007733 after just finding and flipping all positive-gain 1-augmentations. The graph MISKnowledgeMap has the lowest quality matching, having a gap to optimality of 6.1457. Still this is far better than the guaranteed upper bound of 50%.

Next, we discuss the results of table 6b which shows the estimated gaps to optimality of ten bigger graphs. These numbers are higher than the exact gaps to optimality. Notable is the gap belonging to graph kron_g500-logn16. This gap is 66.7609 which is higher than the upper bound of 50 which would have been expected if the exact gap to optimality was used. However, since this gap is an estimated gap, this does not automatically mean that the algorithm fails. The estimated gap is an upper bound of the exact gap and therefore, the exact gap to optimality might be lower than 50. This cannot be said with certainty until the exact matching is calculated. Due to the same reasoning, it cannot be said that the quality of the matchings in this set of graphs is worse

	app 1		app 2		app 2-2		app 2-5		app 2-10	
Matrix	It	Time	It	Time	It	Time	It	Time	It	Time
orsirr_2	9	0.1468	9	0.1482	5	0.0806	3	0.0508	2	0.0378
saylr4	17	0.6112	17	0.6055	9	0.3239	4	0.1628	3	0.1315
Binaryalpha digs_10NN	7	0.1365	7	0.1236	4	0.0726	2	0.0406	2	0.0417
G22	29	0.5604	29	0.5668	15	0.3102	6	0.1623	4	0.1332
MISKnow- ledgeMap	10	0.2491	10	0.2418	6	0.1514	3	0.0917	2	0.0739
cond-mat	9	4.7242	9	4.6321	5	2.6448	3	1.6693	2	1.1442
foldoc	1062	405.6532	1062	415.2144	532	197.2991	178	70.2070	98	42.2087
hi2010	10	12.0129	10	12.0537	5	5.9528	3	3.6447	2	2.4695
har_10NN	8	1.9259	8	2.0382	5	1.2349	3	0.7739	2	0.5379
astro-ph	24	13.3852	24	12.7739	13	6.9778	5	2.8309	4	2.3410
sd2010	11	157.7922	11	163.5784	6	87.2879	3	45.3147	2	29.2746
cage11	17	49.0224	17	48.9057	9	26.0803	4	11.9939	3	8.8428
appu	14	7.4022	14	6.9004	8	4.1419	4	2.5826	3	2.1618
kron_g500- logn16	39	270.9712	39	279.5976	20	143.4536	7	52.7207	5	39.0580
gupta3	164	135.4841	164	148.7698	82	93.9225	28	58.6497	16	51.0913

Table 5: Runtimes of the 1-augmentation algorithms on all fifteen graphs. The time is measured in seconds, and it represents the number of main iterations used in the algorithm. App 1 and app 2 represent searching methods approach 1 and 2 respectively. App 2-2, app 2-5 and app 2-10 represent the methods where searching methods approach 2 is used with 2, 5 or 10 iterations of selecting 1-augmentations.

Matrix	%
orsirr_2	0.0254000
saylr4	0.0007733
Binaryalphadigs_10NN	1.3617000
G22	3.1000000
MISKnowledgeMap	6.1457000

(a) The exact gap to optimality of 10 graphs.

Matrix	%
cond-mat	23.8647
foldoc	6.9908
hi2010	26.2439
har_10NN	10.3633
astro-ph	23.4335
sd2010	21.1847
cage11	1.8381
appu	16.9548
kron_g500-logn16	66.7609
gupta3	1.2691

(b) The estimated gap to optimality of 10 graphs.

Table 6: Test results of the gap to optimality. The % column contains the exact gap to optimality (Table 6a) and the estimated gap to optimality (Table 6b).

than the quality of the matchings in the first set. The exact matching should be calculated to say something about this.

We described the results of quality of the matching at the end of the algorithm. Since the algorithm makes a new matching each main iteration, it can be interesting to see what happens with the gap to optimality after each iteration. Therefore, the gap to optimality was calculated after each main iteration. These results can be seen in Figure 24. For the five small graphs, the exact gap to optimality is used. For the 10 bigger graphs this is the estimated gap to optimality.

A number of observations can be made. The first is that the first gap to optimality differs for each matrix. There are test matrices where approach 2 without extra selection iterations achieves a gap of less than 40%. Examples of these matrices are orsirr_2, Binaryalphadigs_10NN and cage 11. There are also graphs for which the matching after the first main iteration is of very poor quality. Examples are sarylr4 and G22. The matchings for foldoc, kron_g500-logn16 and gupta3 also have very high estimated gaps after the first iteration. These gaps might be smaller in practice since they are upper bounds. The second observation involves the rate of convergence. Some graphs approach the final gap very closely after a few iterations whereafter the gap only decreases very little for the remaining iterations. Examples are appu, cage 11 and astro-ph. This in contrary to matrices foldoc and G22 for which the gap decreases steadily as the number of iterations increases. The third category consist of matrices gupta3 and kron_g500-logn16. These matrices need a few iterations before the gap starts to decrease fast. The difference in rate of converges between the three used methods depends on the rate of convergence of approach 2. When the decrease of the gap using approach 2 in the first iterations is small, approach 2-2 and approach 2-5 will start with a higher start gap than when the decrease of the gap using approach 2 is larger. This is expected since approach 2-2 gives the same result as two main iterations and 2-5 as five main iterations. The differences between the matrices were expected since the performance of the algorithm depends heavily on the structure of the graphs and the weights of the edges. Since each matrix has its own structure, it is likely that the behaviour is different.

Runtime

In this section we discuss the runtime of the two different approaches and their variants shown in Table 5. First, we discuss the runtimes of the different methods for the same test matrix whereafter we say something about the difference in runtime between multiple test matrices.

We start with noting that the difference between approach 1 and approach 2 without selection is very small. Both methods always use the same number of iterations and the runtime is very similar. In most test matrices, the runtime only differs 1%. This is very plausible since the two methods use the same kind of operations. When comparing approach 2 without extra selection and approach 2 with extra selection, a larger difference can be observed. For almost every test matrix, the runtime and the number of main iterations decrease by a factor between 1.5 and 2.5 as the number of selection iterations increases. For some test matrices, this decrease is less dramatic. For example graph Binaryalphadigs_10NN shows no improvement from approach 2-5 to approach 2-10. An explanation is that the maximum number of selection iterations needed is already achieved in approach 2-5. Therefore, method 2-5 and approach 2-10 are the same. This explanation is strengthened by the observation that both methods use two main iterations. In the first iteration, all positive-gain augmentations are found and flipped. In the second main iteration, the *Gain* matrix is calculated again and is found empty and therefore the algorithm terminates after two iterations. This means that only one flipping phase has happened implying the maximal number

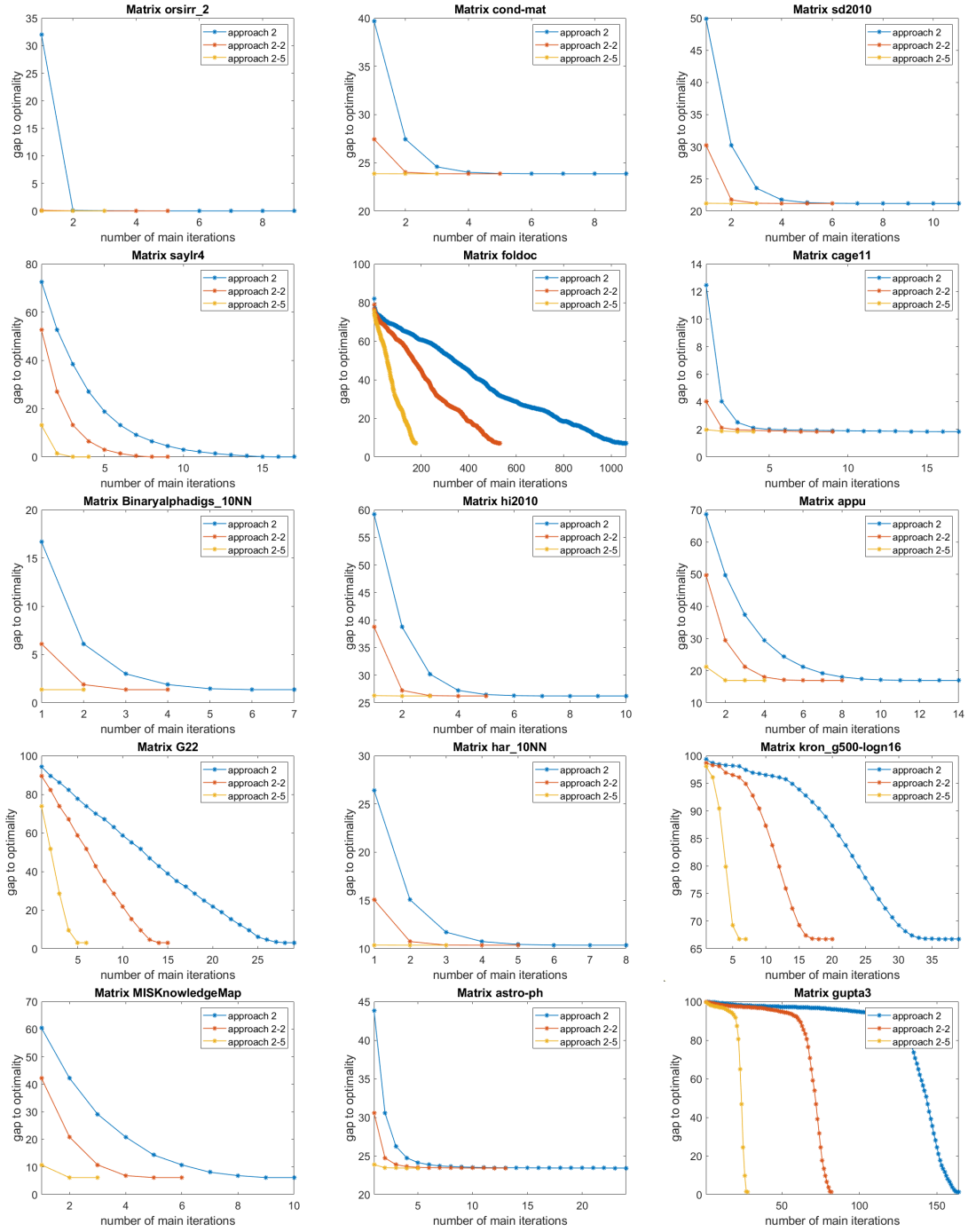


Figure 24: The gap to optimality after each main iteration for the methods approach 2, approach 2-2 and approach 2-5. For the five matrices in the left column, this is the exact gap to optimality. For the other ten matrices this is the estimated gap to optimality. The gap to optimality is given as a percentage.

of selection iterations is achieved in approach 2-5. For test matrices as gupta3, astro-ph and appu, the difference between 2-5 and 2-10 is also less than the 1.5 time speed up. In these matrices, the decrease of main iterations is less than half times the time of approach 2-5.

To investigate the dependence of the runtime on the number of selection iterations, we performed a second experiment. The number of selection iterations was set to the number of vertices, which is an upper bound for the maximum number of selection iterations that can be performed within a single main iteration. When the maximum number is reached, i.e., there are no more edges that can be added, the main iteration stops. To keep the experiments the same and make them comparable, an additional iteration was done to check if matrix *Gain* is indeed empty. This method was called approach 2-max. Table 7 shows the runtime of this experiment. It also shows the maximum number of selection iterations needed. When comparing the results from this table with Table 5, two observations can be made. The first is that the number of selection iterations in this experiment and the number of main iterations in approach 2 in the previous experiment are the same. The second is that the runtimes of this experiment are faster or the same in comparison with approach 2-10. For some matrices, the decrease in runtime is large (matrix foldoc and kron_g500-logn16) whereas for others it is not. The two mentioned matrices needed more than 10 main iterations in approach 2-10, meaning a lot of time could be saved. For many other matrices, the number of main iterations was already low. These results prove the hypothesis that performing a single main iteration with a maximum number of selection iterations is faster than performing the main iteration multiple times with or without a fixed number of selection iterations when only considering 1-augmentations of type 1a.

Matrix name	Iterations	Runtime
orsirr_2	9	0.0348
saylr4	17	0.1063
Binaryalphadigs_10NN	7	0.0374
G22	29	0.0992
MISKnowledgeMap	10	0.0608
cond-mat	9	1.1281
foldoc	1062	9.5462
hi2010	10	2.4992
har_10NN	8	0.5671
astro-ph	24	1.3586
sd2010	11	29.3817
cage11	17	6.1640
appu	14	1.7680
kron_g500-logn16	39	20.0479
gupta3	164	47.0488

Table 7: Results of method approach 2-max. The number of iterations represents the maximum number of selection iterations needed. The runtime is in seconds.

Another interesting question about the runtime is how it relates to the size of the test matrices. This question is harder to analyse, since it depends on the number of edges, the number of vertices and the structure of the graph. Additional to this, the algorithm needs a different number of

iterations for each test matrix. This make comparing multiple graphs with each other difficult. To make an attempt to analyse it, we calculated the average runtime per iteration. Table 8 shows these average runtimes for each method, where the matrices are sorted by the number of vertices. The average runtime increases as the number of vertices increases for almost all methods and all test matrices implying that the number of vertices has a major influence on the average runtime. An increase in the number of vertices means almost always a higher average runtime per iteration despite the number of edges. This statement is strengthened by comparing matrix foldoc with hi2010. These matrices have almost the same number of edges but a different number of vertices, 59693 and 62063 edges vs 13356 and 25016 vertices respectively. When comparing the average runtime per iteration, hi2010 has an average runtime of around 3 times the average runtime of foldoc whereas the number of vertices has increased by a factor 1.5. The same holds for matrices cage11 and sd2010. The difference in vertices is around a factor 2.2, but the difference in average runtime is a factor 5. For small matrices Binaryalphadigs_10NN and saylr4 this difference is less. Again, both matrices have almost the same number of edges and the number of vertices of saylr4 are 2.5 times the number of vertices of Binaryalphadigs_10NN. The difference in average runtime is a factor 2.

The number of edges seems to be of less importance. Sorting Table 8 by the number of edges does not show any direct relationship between the test matrices. To see strengthen this further, consider matrices cond-mat en astro-ph. These matrices have a very similar number of vertices: 16726 vs 16706. The number of edges differs more: 47594 vs 121251. The runtime for cond-mat is slightly less for all methods, but they are very close to each other. A similar observation can be made when comparing foldoc and appu. The number of vertices is 13356 and 14000 respectively, whereas the number of edges is equal to 59693 and 919552 respectively. The runtime of appu is at most 2 times the runtime of foldoc whereas for the number of edges this is 15 times. These results suggest that the runtime increases as the number of edges is increased and the number of vertices is kept fixed. However, this effect is less dramatic than when the number of vertices is increased.

The above analysis indicates that the number of vertices plays an important role in the runtime of the algorithm. To get a better idea of how the runtime depends on the number of edges and vertices, additional experiments should be done with graphs that have a similar structure or have the same number of vertices and a different number of edges or vice versa.

Matrix	app 1	app 2	app 2-2	app 2-5	app 2-10
orsirr_2	0.0163	0.0165	0.0161	0.0169	0.0189
Binaryalpha-digs_10NN	0.0195	0.0177	0.0182	0.0203	0.0209
G22	0.0193	0.0195	0.0207	0.0271	0.0333
MISKnow-ledgeMap	0.0249	0.0242	0.0252	0.0306	0.0370
saylr4	0.0360	0.0356	0.0360	0.0407	0.0438
har_10NN	0.2407	0.2548	0.2470	0.2580	0.2690
foldoc	0.3820	0.3910	0.3709	0.3944	0.4307
appu	0.5287	0.4929	0.5177	0.6457	0.7206
astro-ph	0.5577	0.5322	0.5368	0.5662	0.5853
cond-mat	0.5249	0.5147	0.5290	0.5564	0.5721
gupta3	0.8261	0.9071	1.1454	2.0946	3.1932
hi2010	1.2013	1.2054	1.1906	1.2147	1.2348
cage11	2.8837	2.8768	2.8978	2.9985	2.9476
kron_g500-logn16	6.9480	7.1692	7.1727	7.5315	7.8116
sd2010	14.3447	14.8708	14.5480	15.1049	14.6373
Normalized geometric mean	1.00	1.01	1.03	1.16	1.25

Table 8: Average runtime of all methods applied to the test matrices where the matrices are sorted by number of vertices. The bottom line summarizes the results of each column as a normalized geometric mean.

3.2 2-augmentations

In this section we discuss the results of the algorithm for 2-augmentations. We investigated four setups. In the first, we start with a maximal matching obtained with searching approach 2 using the maximum number of selection iterations needed. After this step, the algorithms for 1-augmentations without additional selection and augmentations 2 are alternated. The difference between setup 1, setup 2 and setup 3 is that for setup 2 the normal 1-augmentation algorithm is replaced by approach 2-2 and for setup 3 that the normal 1-augmentation algorithm is replaced by approach 2-max. In the fourth setup, searching approach 2 without selection is alternated with augmentations 2 directly, meaning it does not start with a head start. A single iteration contains both a search for 1-augmentations and a search for augmentations 2. The results can be found in Tables 9 and 10.

Quality of the matching

In this section we discuss the quality of the matchings of the four setups found in table 9. The first thing to notice is that almost all gaps are bounded from above by 33%, implying the algorithm is indeed a 2/3-approximation. The only exception is again kron_g500-logn16 with an estimated gap to optimality of 64.1061 and 64.3815. This graph was also an outlier in the algorithm for 1-augmentations. Again, it could be that this estimated gap is too pessimistic and that the actual gap is less than 33%, but this has to be checked by computing the exact solution. Secondly, the gaps to optimality for setup 1, 2 and 3 are all the same. There is no difference between these setups. The gap to optimality of setup 4 does differ from the other setups. For most test matrices, the gap

Matrix	setup1	setup2	setup3	setup4
orsirr_2	0.0123	0.0123	0.0123	0.0016
saylr4	0.0005	0.0005	0.0005	5.0729
Binaryalphadigs_10NN	0.3964	0.3964	0.3964	0.5922
G22	0.4000	0.4000	0.4000	0.3000
MISKnowledgeMap	2.1134	2.1134	2.1134	2.6510
cond-mat	22.9142	22.9142	22.9142	23.0433
foldoc	6.2962	6.2962	6.2962	6.5663
hi2010	23.8503	23.8503	23.8503	24.1537
har_10NN	8.9824	8.9824	8.9824	9.1869
astro-ph	22.1311	22.1311	22.1311	22.2450
sd2010	18.1422	18.1422	18.1422	18.3245
cage11	1.5875	1.5875	1.5875	1.8666
appu	16.0739	16.0739	16.0739	17.4218
kron_g500-logn16	64.1061	64.1061	64.1061	64.3815
gupta3	0.6971	0.6971	0.6971	0.7091

Table 9: The gap to optimality when the algorithm is finished for four di□erent setups. The gaps of the first 5 matrices are exact, the others are estimated.

	setup1		setup2		setup3		setup4	
Matrix	It	Time	It	Time	It	Time	It	Time
orsirr_2	5	0.2933	5	0.2968	5	0.2960	7	0.3780
saylr4	2	0.2958	2	0.2836	2	0.2808	7	0.7719
Binaryalphadigs_10NN	6	0.4072	6	0.4132	6	0.4101	11	0.7202
G22	5	0.4640	5	0.4402	5	0.4397	15	0.9925
MISKnowledgeMap	8	0.6947	8	0.6417	8	0.6468	13	0.9642
cond-mat	5	6.7992	5	6.8848	5	6.7530	7	9.0253
foldoc	4	14.5512	4	14.2818	4	14.1753	21	22.4479
hi2010	8	29.4262	8	29.2034	8	29.6515	11	39.2547
har_10NN	12	9.3207	12	9.3800	12	9.7724	18	14.3847
astro-ph	5	7.8531	5	8.4249	5	8.0560	13	18.9385
sd2010	10	462.3908	10	464.6600	10	467.6958	15	684.3732
cage11	4	36.7353	4	36.1937	4	37.4641	13	119.9034
appu	21	34.4274	21	34.0722	21	33.9948	22	34.4894
kron_g500-logn16	6	118.5570	6	117.1152	6	119.3278	15	249.9642
gupta3	24	102.6813	24	101.4183	24	102.0405	103	247.9074

Table 10: The runtimes in seconds and the number of iterations of the four di□erent setups for the 15 test matrices.

belonging to setup 4 is higher than the gap belonging to the other setups. The only two exceptions are orsirr_2 and G22. The difference between the setup 4 and the others seems not very large. Only for the five small test matrices and cage11, this difference is a little larger. The difference for matrix saylr4 is the most remarkable. Setup 4 obtains a gap to optimality of 5.0729% whereas the gap of the other setups is 0.0005%. The gap to optimality of setup 4 is even larger than the gap to optimality after just performing all 1-augmentations. This behaviour can be expected. Since setup 4 does not start with a maximal matching obtained via the 1-augmentations, it is more likely to end with a different final matching. This matching can be better or worse than the matching found with the algorithm for 1-augmentations or setup 1,2 or 3. The only thing that can be said about the matching is that it obtains a gap of at most 33%. There are more test matrices for which the matching of setup 4 obtains a larger gap to optimality than the 1-augmentations algorithm. These matrices are cage11 and appu. The difference between the lowest estimated gap to optimality of the 2-augmentations and the 1-augmentation is not very large. The highest difference is achieved for matrix gupta3, which has decreased its gap by a factor 0.5. For the other matrices, this factor lies between 0.97 and 0.86. For the exact gaps to optimality, this difference is larger. Matrix orsirr_2 improves its gap by a factor 0.06 and the others improve by a factor between 0.1 and 0.59. An explanation for these low improvements could be that the gaps to optimality obtained by the 1-augmentations algorithm are already quite low. Therefore, decreasing the gap is more difficult to achieve.

Runtime

In this section, we discuss the results of the runtime of the algorithm for 2-augmentations as shown in Table 10. The most important observation is that setup 4 is always slower than the other setups, up to three times. The number of iterations is also higher, up to 5 times. Setup 1, 2 and 3 perform again very alike. The number of iterations for these methods is the same as is the runtime. An explanation for this is that after flipping augmentations 2 only a limited number of edges become unmatched and available for 1-augmentations again. Therefore, the number of positive-gain 1-augmentations after performing a round of augmentations 2 is not as high as the number of positive-gain 1-augmentations starting with an empty matching. Therefore, there probably will be fewer positive-gain 1-augmentations starting at the same vertex and therefore fewer conflicts with other 1-augmentations occur. Therefore, an additional selection iteration has less effect. Another explanation is that after flipping the augmentations 2, there are still no positive-gain 1-augmentations and a new round of augmentations 2 is performed. In the current implementation, no extra selection iteration is added for the augmentations 2. In future work, this could be added and the influence of this extra selection could be investigated.

4 Conclusion

In this thesis, we presented an approximation algorithm for the maximum weight matching problem for general graphs. In each iteration of the algorithm, the algorithm searches for positive-gain k-augmentations and flips them. With the use of an approximation lemma, a lower bound of $\frac{k}{k+1}$ times the maximum weight can be guaranteed if no positive-gain k-augmentations exist. In this thesis, we provided searching methods for k equal to 1, 2 and 3 as well as a general description of how to perform these augmentations. For $k = 1$ we even provided two different approaches and a method to detect if a positive-gain 1-augmentation exists. This all means that theoretically, we obtained a 3/4-approximation algorithm. The algorithm is developed in GraphBLAS, meaning it is completely built with matrix operations. Each iteration of searching for augmentations and flipping them can be done in linear time depending on the size of the graph. This suggests that the algorithm can be fast, although the overall number of iterations depends on the graph and is not known beforehand. Additionally, we provided a brief description of how the Suitor idea can be implemented in terms of GraphBLAS.

We implemented the algorithm for $k = 1$ and $k = 2$. For this we used the Matlab interface of the SuiteSparse implementation [8] of GraphBLAS. With this implementation we checked the quality of the obtained matching and the runtime of the algorithm for both $k = 1$ and $k = 2$.

The obtained matchings were compared with the maximum weight if this matching could be computed in reasonable time. If not, an upper bound of the maximum weight was used. For both $k = 1$ and $k = 2$, the algorithm achieved the expected quality when comparing them with the exact maximum weight. For the estimated maximum weight, there was one matrix that did not achieve the required quality. Since the estimated maximum weight is an upper bound of the exact maximum weight, it cannot be concluded that the algorithm fails. To ensure this, the exact matching must be determined. For all other test matrices, the matchings were of sufficient quality. In most cases, the quality was far better than the guaranteed lower bound.

We compared the runtime and number of iterations needed for a different number of setups. The result was that for $k = 1$, the algorithm with a maximal number of selection iterations was the fastest. Setting the number of selection iterations to maximal means losing the linear runtime of each main iteration. However, the results showed that even when the number of selection iterations is not maximal, it is beneficial to perform a number of them. Therefore, performing a fixed number of these selection iterations decreases the runtime and keeps the linear runtime. For $k = 2$ it was fastest to start with a maximal matching obtained by running the algorithm for $k = 1$ until no positive-gain augmentations are found. Then switch between the algorithm for $k = 1$ and the searching method for augmentation 2. Both the runtime as the number of main iterations was lower in these cases in comparison to not starting with a maximal matching.

To conclude, our algorithm achieves our goal to develop an 2/3-approximation algorithm for the maximum weight matching problem in GraphBLAS. Theoretically, we even obtained a 3/4-approximation. Theoretically, each iteration has linear time complexity, which also fulfils the goal to find an algorithm that performs in reasonable time.

4.1 Future work

Although the algorithm fulfils our goals, more work remains to be done. The algorithm has been developed in GraphBLAS such that it can easily be parallelized. This has not been tested and it is interesting to see if it indeed performs well if used with multiple processors. Furthermore, the algorithm is not implemented for $k = 3$. This could be added in further work. The same holds for better selection methods for augmentations 2 and 3.

Furthermore, more experiments could be done to investigate the runtime of each iteration and the influence of the number of vertices and edges and the structure of the graph on it. Now, it was hard to investigate the claim of linear runtime since the test matrices all had different structures and different sizes.

Lastly, in this thesis, we focussed on the development of a single main iteration. Questions as: how many of them are needed? or: what order is best to use when repeating augmentations 2 and 1-augmentations in the algorithm for $k = 2$? are not investigated theoretically. This could be of interest. It might be that some smart order of alternating the augmentations 2 and 1-augmentations exists to achieve a faster runtime.

References

[1] Ariful Azad and Aydın Buluç. Distributed-memory algorithms for maximum cardinality matching in bipartite graphs. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 32–42. IEEE, 2016.

[2] Ariful Azad and Aydın Buluç. A matrix-algebraic formulation of distributed-memory maximal cardinality matching algorithms in bipartite graphs. *Parallel Computing*, 58:117–130, 2016.

[3] Ariful Azad, Aydın Buluç, Xiaoye Li, Xinliang Wang, and Johannes Langguth. A distributed-memory algorithm for computing a heavy-weight perfect matching on bipartite graphs. *SIAM Journal on Scientific Computing*, 42(4):C143–C168, 2020.

[4] Ariful Azad, Georgios A. Pavlopoulos, Christos A. Ouzounis, Nikos C Kyrpides, and Aydın Buluç. HipMCL: a high-performance parallel implementation of the Markov clustering algorithm for large-scale networks. *Nucleic acids research*, 46(6):e33, 2018.

[5] Rob H. Bisseling. *Parallel Scientific Computation: A Structured Approach Using BSP*. Oxford University Press, USA, second edition, 2020.

[6] Aydın Buluç and Kamesh Madduri. Parallel breadth-first search on distributed memory systems. In *2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 1–12. Association for Computing Machinery, 2011.

[7] Aydın Buluç, Tim Mattson, Scott McMillan, José Moreira, and Carl Yang. Design of the GraphBLAS API for C. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 643–652. IEEE, 2017.

[8] Timothy Davis. Algorithm 1000: SuiteSparse:GraphBLAS: Graph algorithms in the language of sparse linear algebra. *ACM Transactions on Mathematical Software*, 45:1–25, 2019. See `http://suitesparse.com`.

[9] Timothy A. Davis and Yifan Hu. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.*, 38(1), dec 2011.

[10] Doratha E. Drake and Stefan Hougardy. Improved linear time approximation algorithms for weighted matchings. In *Approximation, Randomization, and Combinatorial Optimization.. Algorithms and Techniques*, pages 14–23, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[11] Doratha E. Drake and Stefan Hougardy. Linear time local improvements for weighted matchings in graphs. In *Experimental and E cient Algorithms*, pages 107–119, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[12] Doratha E. Drake and Stefan Hougardy. A simple approximation algorithm for the weighted matching problem. *Information Processing Letters*, 85(4):211–213, 2003.

[13] Ran Duan and Seth Pettie. Approximating maximum weight matching in near-linear time. In *2010 IEEE 51st Annual Symposium on Foundations of Computer Science*, pages 673–682. IEEE, 2010.

[14] Ran Duan and Seth Pettie. Linear-time approximation for maximum weight matching. *Journal of the ACM*, 61(1):1–23, 2014.

[15] Iain S Duff and Jacko Koster. On algorithms for permuting large entries to the diagonal of a sparse matrix. *SIAM Journal on Matrix Analysis and Applications*, 22(4):973–996, 2001.

[16] Jack Edmonds. Maximum matching and a polyhedron with 0, 1-vertices. *Journal of research of the National Bureau of Standards B*, 69(125-130):55–56, 1965.

[17] Harold N. Gabow. Data structures for weighted matching and extensions to b-matching and f-factors. *ACM Transactions on Algorithms (TALG)*, 14(3):1–80, 2018.

[18] Jonathan S Golan. *Semirings and their Applications*. Springer Science & Business Media, 2013.

[19] Sven Hanke and Stefan Hougardy. *New approximation algorithms for the weighted matching problem*. Research report 101010, Research Institute for Discrete Mathematics, University of Bonn, 2010.

[20] Frank Harary. *Graph theory*. Addison-Wesley Publishing Company, 1969.

[21] Jaap-Henk Hoepman. Simple distributed weighted matchings. *ArXiv*, cs.DC/0410047, 2004.

[22] Jeremy Kepner, Peter Aaltonen, David Bader, Aydın Buluç, Franz Franchetti, John Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, et al. Mathematical foundations of the GraphBLAS. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–9. IEEE, 2016.

[23] Jeremy Kepner, David Bader, Aydin Buluç, John Gilbert, Tim Mattson, and Henning Meyerhenke. Graphs, matrices, and the GraphBLAS: Seven good reasons. *Procedia Computer Science*, 51, 04 2015.

[24] Jeremy Kepner and John Gilbert. *Graph Algorithms in the Language of Linear Algebra*. Society for Industrial and Applied Mathematics, 2011.

[25] Jeremy Kepner, Manoj Kumar, José Moreira, Pratap Pattnaik, Mauricio Serrano, and Henry Tufo. Enabling massive deep neural networks with the GraphBLAS. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–10. IEEE, 2017.

[26] Arif Khan, Alex Pothen, Md. Mostofa Ali Patwary, Mahantesh Halappanavar, Nadathur Rajagopalan Satish, Narayanan Sundaram, and Pradeep Dubey. Designing scalable b-matching algorithms on distributed memory multiprocessors by approximation. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 773–783. IEEE, 2016.

[27] M Lothaire. *Applied combinatorics on words*, volume 105. Cambridge University Press, 2005.

[28] Mohsen Mahmoudi Aznaveh, Jinhao Chen, Timothy Davis, Bálint Hegyi, Scott Kolodziej, Tim Mattson, and Gábor Szárnyas. Parallel GraphBLAS with OpenMP. In *2020 Proceedings of the SIAM Workshop on Combinatorial Scientific Computing*, pages 138–148. SIAM, 2020.

[29] Fredrik Manne and Rob H. Bisseling. A parallel approximation algorithm for the weighted maximum matching problem. In *International Conference on Parallel Processing and Applied Mathematics*, pages 708–717. Springer, 2007.

[30] Fredrik Manne and Mahantesh Halappanavar. New effective multithreaded matching algorithms. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 519–528. IEEE, 2014.

[31] Tim Mattson, David Bader, Jon Berry, Aydın Buluç, Jack Dongarra, Christos Faloutsos, John Feo, John Gilbert, Joseph Gonzalez, Bruce Hendrickson, et al. Standards for graph algorithm primitives. In *2013 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–2. IEEE, 2013.

[32] Tim Mattson, Timothy A. Davis, Manoj Kumar, Aydın Buluç, Scott McMillan, José Moreira, and Carl Yang. LAGraph: A community effort to collect graph algorithms built on top of the GraphBLAS. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 276–284. IEEE, 2019.

[33] Jens Maue and Peter Sanders. Engineering algorithms for approximate weighted matching. In *International Workshop on Experimental and E cient Algorithms*, pages 242–255. Springer, 2007.

[34] Markus Olschowka and Arnold Neumaier. A new pivoting strategy for gaussian elimination. *Linear Algebra and its Applications*, 240:131–151, 1996.

[35] Seth Pettie and Peter Sanders. A simpler linear time 2/3 - ε approximation for maximum weight matching. *Information Processing Letters*, 91(6):271–276, 2004.

[36] Alex Pothen, S.M. Ferdous, and Fredrik Manne. Approximation algorithms in combinatorial scientific computing. *Acta Numerica*, 28:541–633, 2019.

[37] Robert Preis. Linear time 1/2 - approximation algorithm for maximum weighted matching in general graphs. In *Proceedings of the 16th Annual Conference on Theoretical Aspects of Computer Science*, STACS'99, page 259–269, Berlin, Heidelberg, 1999. Springer-Verlag.

[38] Nadathur Satish, Narayanan Sundaram, Md. Mostofa Ali Patwary, Jiwon Seo, Jongsoo Park, M. Amber Hassaan, Shubho Sengupta, Zhaoming Yin, and Pradeep Dubey. Navigating the maze of graph analytics frameworks using massive graph datasets. In *2014 ACM SIGMOD international conference on Management of data*, SIGMOD '14, pages 979–990, 2014.

[39] Daniel Saunders. Weighted maximum matching in general graphs, 2022. MATLAB Central File Exchange. Last accessed March 9, 2022, (`https://www.mathworks.com/matlabcentral/fileexchange/42827-weighted-maximum-matching-in-general-graphs`).

[40] Edgar Solomonik, Aydın Buluç, and James Demmel. Minimizing communication in all-pairs shortest paths. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 548–559. IEEE, 2013.

[41] Gábor Szárnyas, David A. Bader, Timothy A. Davis, James Kitchen, Timothy G. Mattson, Scott McMillan, and Erik Welch. LAGraph: Linear algebra, network analysis libraries, and the study of graph algorithms. *arXiv preprint arXiv:2104.01661*, 2021.

[42] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[43] Carl Yang, Aydin Buluç, and John D. Owens. Graphblast: A high-performance linear algebra-based graph framework on the GPU. *CoRR*, abs/1908.01407, 2019.

[44] A. N. Yzelman, D. Di Nardo, J. M. Nash, and W. J. Suijlen. A C++ GraphBLAS: specification, implementation, parallelisation, and evaluation. Preprint, 2020.