Universiteit Utrecht

Faculty of Science

A 3/4-approximation of the maximum weight matching problem using the GraphBLAS standard

Master Thesis

David D. de Best

Mathematical Sciences

Supervisor:

Prof. Dr. Rob H. Bisseling
Utrecht University

Second reader:

Dr. Ivan V. Kryven
Utrecht University

June 11, 2023

Abstract

The maximum weight matching problem is a well-studied and central problem in graph theory, for which a variety of exact and approximation algorithms exist. Parallelisation of these algorithms is often hard, as it is for many graph problems. By designing an algorithm using the GraphBLAS standard, we gain easy access to parallelism in executing the sparse matrix operations that constitute the solution procedure.

We propose an approximation algorithm based on searching and applying a set of positive-gain k-augmentations. In this method, searching for sets of 1-, 2- and 3-augmentations can be performed in linear time in terms of the size of the input graph. By performing a series of these searches and applying the positive-gain augmentations until none are available, we can guarantee a 3/4-approximation of the true maximum weight matching.

We present several numerical results of experiments investigating the runtime of the algorithm and quality of the obtained results.

Contents

Chapter 1

Introduction

The maximum weight matching problem is a well-known and fundamental problem in graph theory. A matching in a graph $G(V, E)$ is a subset of pairwise non-adjacent edges of E. In a weighted graph, the solution to the maximum weight matching problem is a matching where the combined weight of its edges is the maximum of the weights of all matchings.

The GraphBLAS standard stems from the deep connection between graphs and matrices and the observation that many graph algorithms can be expressed in the same basic building blocks of linear algebra operations. When the state of the art of constructing these algorithms was mature enough, there came a need for a standard set of these primitive building blocks such that research and communication on these graph algorithms would not be hindered by overlapping of different standards [11]. A community effort was launched to develop this standard that became the GraphBLAS, leading to a mathematical definition in [9]. Now that this definition and several implementations exist, the focus is shifted to creating a large set of graph algorithms that use the standard, as is done in [12]. This thesis is aimed at contributing to that effort.

In this thesis, we propose an approximation algorithm solving the maximum weight matching (MWM) problem in general graphs using the GraphBLAS standard. We continue the work in the thesis of Kemper [8], where an algorithm is proposed that makes use of positive-gain k-augmentations with $k = 1$ and $k = 2$. The algorithm using 1-augmentations guarantees a $1/2$-approximation of the maximum weight matching, and the algorithm using 2-augmentations is a good basis for achieving a $2/3$-approximation, although it cannot yet be guaranteed. Some ideas are also proposed for the use of 3-augmentations which would result in a $3/4$-approximation. Finding positive-gain 2- and 3-augmentations using the linear algebra methods of GraphBLAS is notably more difficult than doing this for $k = 1$, and it will be the main focus of this thesis. We will combine all methods into one algorithm that will efficiently find a $3/4$-approximation for general graphs and analyse this algorithm in terms of runtime and quality of its results. This algorithm will consist of a series of iterations that search for a set of augmentations and for it to be usable in practice, a theoretical linear runtime of each iteration is required. We will thus provide a detailed runtime analysis of the iterations in this thesis.

A great motivation for following the GraphBLAS standard is that the resulting algorithm will be suited well for the use of parallelism. Graph algorithms are typically hard to parallelise, but to perform computations on the largest data sets in a reasonable amount of time, this can become essential. The use of GraphBLAS allows for easy access to parallel computation of the matrix operations that make up the algorithm. Therefore we will also study the improvements in runtime when executing the algorithm on multiple parallel threads.

In chapter 2 we define the problem and discuss any preliminary knowledge. We will also briefly describe the use of the GraphBLAS standard and introduce the needed notation. The chapter concludes with a set of basic and often-used procedures and their GraphBLAS implementation. The approximation algorithm is described in chapter 3. The application of augmentations is discussed in section 3.1 and our method for finding 1-augmentations is detailed in section 3.2. These sections also function as a practical case for developing graph algorithms for readers unfamiliar with GraphBLAS. An elaborate description of our methods for finding 2-augmentations is given in section 3.3, and we do the same for 3-augmentations in section 3.4. Each part builds on the previous and includes a detailed runtime analysis. We discuss experiments performed using a C implementation of our algorithm in chapter 4, examining the runtimes and quality of the obtained approximations. Finally, we conclude in chapter 5 and discuss the limitations of these results and give an outlook on future work in chapter 6.

Chapter 2

Preliminaries

In the following sections we introduce the maximum weight matching problem, the GraphBLAS standard and all relevant notation, revising the preliminary knowledge for this thesis. A more elaborate introduction, as well as a discussion of existing approximation algorithms that solve the maximum weight matching problem, either sequentially or in parallel, can be found in the thesis of Kemper [8], on which this text is based. We start by describing some basic definitions and notation of the necessary graph theory.

We consider a general graph $G(V, E)$, with vertex set V and edge set E, without double edges and self loops. Edges are adjacent to each other when they share a vertex and vertices are adjacent to each other when they are connected by an edge. A vertex is incident to an edge if it is one of the endpoints of that edge. Each edge $(v_1, v_2) \in E$ has a weight $w(v_1, v_2)$, where in this thesis we assume $w(v_1, v_2) \in \mathbb{R}_+$ for every edge. The weight of a set of edges S is the combined edge weight, so $w(S) = \sum_{s \in S} w(s)$. When we assume $|V| = n$ and $|E| = m$, G can be modeled as its $n \times n$ adjacency matrix A containing m nonzeros $a_{v_1 v_2} = w(v_1, v_2)$.

A matching M is a subset of E where no two edges are adjacent. It reflects a selection of pairs of vertices that are 'matched' to each other. Edges that are an element of M are called matched, and the vertices that are endpoints of matched edges are called matched as well. Other edges or vertices are unmatched. In a maximal matching M, none of the edges in E can be added to M with M remaining valid. A maximum(-cardinality) matching has the highest number of edges possible in any matching for G. This is referred to as a perfect matching if every vertex in V is incident to a matched edge.

To solve the maximum weight matching problem, we search for a matching M of G such that there does not exist a matching M' with $w(M') > w(M)$, so the weight of M is maximal. Naturally, the resulting matching will be a maximal matching, otherwise we could easily add another edge to the matching and increase the combined weight. Note the obvious ambiguity in the use of the terms maximum and maximal. We refer to the number of edges when explicitly using maximum or maximal as an adjective for the matching, but we refer to the weight of the matching when using it in any other way, like 'maximum weight matching' or 'a matching of maximal weight'.

Our method of obtaining the maximum weight matching will come down to improving an existing matching. We might start with a matching that is just an empty set and add a number of edges. In trying to improve the matching, we require the definition of an alternating path or cycle and an augmentation.

Definition 2.0.1 (Alternating path or cycle)**.** Let M be a matching. A path or cycle of edges is alternating if those edges are alternately drawn from the matching M and the rest of the graph

$E \setminus M$.

Definition 2.0.2 (Augmentation)**.** Let $M \triangle P = (M \setminus P) \cup (P \setminus M)$ be the symmetric difference of two sets, with matching M and alternating path or cycle P. Then P is called an augmentation with respect to M if $M \triangle P$ is a valid matching. An augmentation with at most k unmatched edges is called a k-augmentation.

If it is clear from the context which matching is used, in this thesis always matching M, P will just be called an augmentation. The symmetric difference of M and P is the result of adding the unmatched edges and removing the matched edges in the augmentation. We will refer to this procedure as flipping or applying the augmentation. Our interest is in augmentations that improve the current matching, which is not guaranteed. Augmentations have a gain value that is defined as

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

If $g(P)$ is positive, P is a positive-gain augmentation, and applying P will increase the weight of the matching. Our algorithm will search and apply all positive-gain k-augmentations for k up to a certain number. Using the approximation lemma as proven by Pothen, Ferdous and Manne [15], we can guarantee a lower bound on the weight of the matching obtained by the algorithm. The approximation lemma is stated below.

Lemma 2.0.1 (Approximation Lemma)**.** *Let M be a matching of G and k an integer greater than 1. If no positive-gain $(k-1)$-augmentations can be applied to M, then*

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

where $M^\star$ is a matching with maximum weight of G.

Using this lemma for our own purposes, we can conclude to have found a 3/4-approximation of the maximum weight matching when we have applied all possible 3-augmentations to our matching.

Building on the theory above, we will design the approximation algorithm in chapter 3, after describing the GraphBLAS standard in the following section.

2.1 GraphBLAS

We will summarise the main concepts from [8] on the GraphBLAS standard here and introduce relevant notation.

Any graph can be represented fully by its adjacency matrix A. It is an $n \times n$-matrix with nonzeros a_{ij} that indicate a directed edge from vertex i to j. Its values are $a_{ij} = w(i,j)$ for weighted graphs and $a_{ij} = 1$ for unweighted graphs. An undirected graph has a symmetric adjacency matrix. Most of the elements in A will be zero. Like many of the matrices that we will be working with, it is often quite sparse. The locations of nonzeros in the matrix as a whole will be referred to as its *sparsity pattern*, a concept that we will use often in comparing matrices to each other. Because of this representation of graphs as (sparse) matrices, many graph algorithms can be expressed in terms of linear algebra operations, which is a field of research that has emerged and expanded in the last twenty years [11]. The work by Kepner and Gilbert [10] contains a thorough explanation of this linear algebraic approach to graph algorithms, suitable for readers new to this topic, covering many practical examples. To help advance this research and support communication in this field, the authors of [11] proposed a community effort to define a standard for linear algebra building

blocks to build these graph algorithms. This led to a mathematical definition of foundations for the GraphBLAS standard by Kepner et al. in 2016 [9]. The next step was an implementation of the standard, for which a definition for a C API was given by Brock et al. [2], and the SuiteSparse:GraphBLAS C implementation by Davis [3] is what we use in this thesis. Currently, a number of GraphBLAS libraries exist and the focus is shifted to creating a large set of graph algorithms following the GraphBLAS standard, as done in the LAGraph library [12], for example.

The GraphBLAS standard defines a set of sparse matrix and vector operations using many different semirings. The specific GraphBLAS semiring is defined in the following way, using a commutative monoid. We also require an addition operator $\oplus$ and multiplication operator $\otimes$.

Definition 2.1.1 (Commutative monoid)**.** Given an operator $\oplus : D \times D \to D$, the algebraic structure $\langle D, \oplus, 0 \rangle$ is called a commutative monoid if the following holds for all $a, b, c \in D$

- Commutativity: $a \oplus b = b \oplus a$
- Associativity: $(a \oplus b) \oplus c = a \oplus (b \oplus c)$
- Identity element: $a \oplus 0 = a = 0 \oplus a$.

Definition 2.1.2 (GraphBLAS semiring)**.** The algebraic structure $\langle D, \oplus, \otimes, 0 \rangle$ is called a GraphBLAS semiring if

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

Matrix multiplication of matrices B_1 and B_2 using the semiring $\langle D, \oplus, \otimes, 0 \rangle$ is denoted as $B_1 \oplus.\otimes B_2$. Similarly, matrix-vector multiplication of B and vector $\mathbf{b}$ is denoted as $B \oplus.\otimes \mathbf{b}$. Of course, standard matrix multiplication uses the $\langle \mathbb{R}, +, \times, 0 \rangle$ semiring. Using this semiring for matrix-vector multiplication of an adjacency matrix B with unit vector e_1 results in a vector containing nonzero values for all neighbouring vertices of vertex 1. This can be interpreted as one step in a Breadth-First Search algorithm on a graph, and thus we find a connection between a basic step in a graph algorithm and a linear algebra operation. However, the true strength of GraphBLAS lies in the use of different operators for addition and multiplication, of which we will see a number of examples in practice in section 3.2 and sections thereafter. It allows for a wide array of operations on graphs via linear algebra.

GraphBLAS allows for more types of operations, of which we will mention a number of essential ones here. Element-wise multiplication and addition of matrices B_1 and B_2 follows standard conventions in GraphBLAS. The main difference between the two are the effect of zero elements, where $b \oplus 0 = b$ and $b \otimes 0 = 0$ for any element b of B_1. For the resulting matrix as a whole, multiplication produces an intersection of the sparsity patterns of B_1 and B_2 and addition produces a union of the sparsity patterns. Using any binary operator $\circ$, element-wise multiplication is denoted as $B_1 \, (\otimes, \circ) \, B_2$, for B_1 and B_2 either both matrices or both vectors. Similarly, we denote element-wise addition as $B_1 \, (\oplus, \circ) \, B_2$. We make an exception for standard element-wise addition and subtraction, usually formulated as $B_1 + B_2$ and $B_1 - B_2$.

Masking of a matrix B with mask M is denoted as $B\langle M \rangle$. This requires M to be a boolean matrix of the same size as B. If $M(i, j)$ is true then $B\langle M \rangle(i, j) = B(i, j)$ and otherwise $B\langle M \rangle = 0$. When using a mask M that is not a boolean matrix, we abuse the notation $B\langle M \rangle$ and imply a masking operation with boolean matrix M' where $M'(i, j)$ is false if $M(i, j)$ is zero and true otherwise.

The number of operations with zero elements will be substantial with the use of sparse matrices. With efficiency in runtime and storage in mind, the GraphBLAS standard distinguishes between implicit and explicit zeros, which we will do in this thesis from now on as well. In the GraphBLAS data structures, implicit zeros are not stored, but only the locations and values of explicit values in the matrix. If such an element is 0, we refer to it as an explicit zero. This greatly reduces the amount of storage needed for storing sparse matrices. Since the value of implicit zeros is not stored, their actual value is determined by the zero element in the semiring used for that operation, and it might not be 0. In designing graph algorithms, we need to pay attention to the effects of the used operators and zero element to ensure the correct outcome. The value of implicit zeros will always be assumed to be 0 in this thesis, unless mentioned otherwise.

2.2 Notation and functions

Here we will mention some other notation used in the thesis. The complement of a boolean matrix B or vector $\mathbf{b}$ is denoted as $\overline{B}$ or $\overline{\mathbf{b}}$, respectively. We use the $:=$ operator for assignment of an existing matrix or vector with new values. An entirely implicit matrix, often returned by a function that finds no result, is denoted as $\mathcal{M}_\emptyset$. Vectors are written in boldface, and we assume that every vector is a column vector, unless mentioned otherwise. Every transposed vector will therefore be a row vector.

One often-used operation in this thesis is permutation of rows or columns of a matrix. For some matrix B that has only one nonzero per row, we denote P^B as the permutation matrix corresponding to B that has the same pattern of nonzeros, but each nonzero has value 1. We will only do this when P^B is in fact a valid permutation matrix, although we allow some rows of B and P^B to be all zeros, when this is inconsequential to the result. The inverse of a permutation matrix is often the same matrix, in which case we will just use the original matrix without notice.

We use a number of standard GraphBLAS functions in pseudocode listings. The function $\texttt{select}(B, \texttt{criterion})$ creates a matrix of the same size as input matrix B and only copies the explicit values of B that adhere to the given criterion. The $\texttt{reduce}(B, \circ)$ function creates a vector from the input matrix B by applying the given binary operator to all values of each row of B. The same function exists for vectors and we omit the operator input when the input matrix only has one explicit value per row.

A list of the functions used in our algorithms can be found in table 2.2. Matrix-matrix multiplication and functions like `reduce` use binary operators $\circ$ to produce a result. Apart from the standard $+, -$ and $\times$-operators, the used binary operators are listed in table 2.1.

The SuiteSparse:GraphBLAS implementation gives useful bounds for the time complexity of each step, summarized in [3]. We list the time complexity of the functions that are used in table 2.2. To determine these values, the storage format of the matrices is essential. Matrices are stored in the compressed-sparse row (CSR) format by default, but other options, like compressed-sparse column (CSC) or structures for hypersparse matrices, are also possible. We assume the CSR format unless specified otherwise. Selection and reduction by row have straightforward implementations using the CSR format and take $O(n+k)$ time when applied to an $n \times n$-matrix with k nonzeros. Element-wise operations are handled either in a set-union or in a set-intersection manner, the element-wise addition and multiplication respectively. Their implementations differ slightly, but are both guaranteed to have a time complexity of $O(n + k_1 + k_2)$ when using two matrices with respectively k_1 and k_2 nonzeros. Matrix-matrix and matrix-vector multiplication is handled in one of three ways, which we discuss further in the next section. In short, the creation of an $n \times n$-matrix from two matrices with a constant number of explicit values per row, like two $n \times 1$-vectors, using a mask with k explicit

values can be done in $O(n + k)$ time. Matrix-vector multiplication where the matrix has k explicit values has the same time bound.

Binary operator $\circ$	Result of $x \circ y$
max	Return the maximum of x and y
min	Return the minimum of x and y
iseq	Return 1 when $x = y$ and 0 when $x \neq y$
pair	Return 1 when x and y are both explicit values and 0 otherwise
any	Return either x or y, decided by the program
first	Return x
second	Return y
firsti	Return the index of x
secondi	Return the index of y

Table 2.1: List of used binary operators.

2.3 Basic procedures

In this section, we cover a number of basic procedures that will be reused often in the algorithms in chapter 3. They are listed in table 2.2 with their theoretical time complexity.

Fast matrix-matrix multiplication with a mask

In general, matrix-matrix multiplication is a slow operation that requires more than linear time. Even when multipying a (possibly full) column and row vector of length n, to an $n \times n$-matrix, which we will do often in this thesis, the runtime can be more than linear. However, we might not be interested in all values of the result matrix, but only need explicit values indicated by some mask matrix. If this mask has $O(m)$ explicit values, we should theoretically be able to perform the operation in $O(n + m)$ time. Suppose we have column vector $\mathbf{v}$, row vector $\mathbf{w}$ and a mask matrix B_{mask} with $O(m)$ explicit values, and calculate

$$B = (\mathbf{v} +.+ \mathbf{w})\langle B_{mask} \rangle.$$

The addition operator in the used semiring does not influence the result here, so without any consequence we choose to use the +-operator. The result B is an $n \times n$-matrix with $O(m)$ explicit values, ideally requiring $O(n + m)$ time to calculate.

As indicated by the GraphBLAS C API, the SuiteSparse:GraphBLAS implementation actually only applies the mask after calculating the intermediary result $(\mathbf{v} +.+ \mathbf{w})$ in what is called the mask/replace phase (see [2], [3] and [4]). In this situation, it is highly inefficient and we need SuiteSparse:GraphBLAS to take the mask into account during the multiplication of the vectors. Matrix-matrix multiplication relies on three algorithms in SuiteSparse:GraphBLAS: Gustavson's algorithm, a heap-based method and a dot-product formulation [3]. The implementation chooses the method it deems most suitable for each situation, but it can be forced to use a method selected by the user. For the masked matrix-matrix multiplication that we want to perform here, it is required to always use the dot-product formulation method. It is intended for this purpose and uses the mask exactly as we need it.

Function	Result	Time complexity
B_1 $(\oplus, \circ)$ B_2, `eAdd`$(B_1, B_2, \circ)$	Element-wise addition with binary operator $\circ$	$O(n + k_1 + k_2)$
B_1 $(\otimes, \circ)$ B_2, `eMult`$(B_1, B_2, \circ)$	Element-wise multiplication with binary operator $\circ$	$O(n + k_1 + k_2)$
$B_1\langle B_2\rangle$	Masking operation on B_1 with mask B_2	$O(n + k_1)$
$B_1 \oplus.\otimes\ B_2^T\langle B_3\rangle$	Matrix-matrix multiplication with a mask, where B_1 and B_2 have a constant number of explicit values per row	$O(n + k_3)$
$B_1 \oplus.\otimes\ \mathbf{b}$	Matrix-vector multiplication	$O(n + k_1)$
`select`$(B_1,$ `criterion`$)$	Copy each explicit value of B_1 to the result that adheres to `criterion`	$O(n + k_1)$
`reduce`$(B_1, \circ)$	Apply binary operator $\circ$ to all values in each row of B_1 to obtain a result vector	$O(n + k_1)$
`selectRows`$(B_1, \mathbf{b})$	Copy the explicit values in row i of B_1 to the result, if $\mathbf{b}(i)$ is explicit	$O(n + k_1)$
`selectColums`$(B_1, \mathbf{b})$	Copy the explicit values in column i of B_1 to the result, if $\mathbf{b}(i)$ is explicit	$O(n + k_1)$
`maxPerRow`(B_1)	Copy only the maximum explicit value in each row of B_1 to the result	$O(n + k_1)$
`maxPerColumn`(B_1)	Copy only the maximum explicit value in each column of B_1 to the result	$O(n + k_1)$

Table 2.2: List of used GraphBLAS functions. Let B_1, B_2 and B_3 be three $n \times n$-matrices with k_1, k_2 and k_3 nonzeros, respectively, and let $\mathbf{b}$ be a column vector of length n.

Taking this into account, we can do the same with two $n \times p$-matrices, C_1 and C_2 with $O(1)$ explicit elements per row, and calculate

$$B = (C_1 \oplus.\otimes\ C_2^T)\langle B_{mask}\rangle$$

in $O(n + m)$ as well, independent of p.

Select rows and columns of a matrix

A very common operation is keeping only the explicit values from a matrix B that are in some given set of rows or columns. To do this for keeping given rows, we require a vector $\mathbf{v}$ which has explicit values at exactly the indices corresponding to the rows that should be kept in the result. Then we can create a mask B_{mask} by multiplying $\mathbf{v}$ with a full vector $\mathbf{z}$ and using B as a mask,

$$B_{mask} = (\mathbf{v} +.+\ \mathbf{z}^T)\langle B\rangle.$$

Note the masked multiplication as described above in this section. The result $B^\star$ is obtained by applying the new mask, so $B^\star = B\langle B_{mask}\rangle$. These steps can both be performed in $O(n + m)$ time. In pseudocode we write $B^\star =$ `selectRows`$(B, \mathbf{v})$, or $B^\star =$ `selectColums`$(B, \mathbf{v})$ for the analogous method for selecting the given columns in a matrix.

Select the maximum value per row

A procedure that we perform often is selecting the maximum value in each row or column of a matrix. Suppose we want to only select the maximum value in each row of matrix B. In case of ties, we choose the element with the highest column index. We aim to create a matrix B_{mask} that indicates exactly those values and we apply it as a mask. Using a reduction over the rows of B using the max-operator, we obtain vector $\mathbf{b}$ with the maximum of row i of B at $\mathbf{b}(i)$. We can multiply $\mathbf{b}$ with the transpose of a full vector $\mathbf{z}$ using the $+.\mathsf{first}$ operation to obtain an $n \times n$-matrix where each column is the same as $\mathbf{b}$. Here we will use B as a mask to restrict the number of operations, only obtaining the relevant explicit values. We choose to provide $\mathbf{z}$ to the algorithm at the start as a full vector of explicit values of 1, requiring $O(1)$ storage in the SuiteSparse:GraphBLAS implementation. An element-wise multiplication with B using the 'is equal' (iseq) operation results in a matrix C indicating the positions of all 1-augmentations with the maximum gain in each row. So,

$$C = B \ (\otimes, \mathsf{iseq}) \ (\mathbf{b} \ +.\mathsf{first} \ \mathbf{z}^T)\langle B \rangle.$$

Here C has multiple explicit values in a row if there were any ties, otherwise there is only one explicit value in a row. Now we need to solve those ties and only keep the value in each row with the highest column index. Instead of a reduction, we now use a matrix-vector multiplication of C and a full vector using the max.secondi operation, and obtain a vector $\mathbf{c}$ containing the highest column index of the explicit values in each row of C. We also use a matrix K with explicit values $K(p,q) = q$ if $C(p,q)$ is explicit, so each value in K reflects its own column index. Now we take an approach analogous to the previous step, calculating

$$\mathbf{c} = C \ \mathsf{max.secondi} \ z^T$$
$$B_{mask} = K \ (\otimes, \mathsf{iseq}) \ (\mathbf{c} \ +.\mathsf{first} \ \mathbf{z}^T)\langle C \rangle.$$

After removing the zeros from B_{mask} it indicates exactly the required explicit values in B, so we apply it as a mask to B. A pseudocode listing for the procedure can be found in algorithm 1, with details on the $O(n+m)$ time complexity of the method. An analogous approach can be used for selecting the highest value per column.

Algorithm 1 `maxPerRow`: Select maximum value per row

Input: Input matrix B, a full vector $\mathbf{z}$ of explicit values 1
Output: Result matrix $B^\star$ with only the maximum value per row of B, using the highest column index in case of ties

1: $\mathbf{b} = \texttt{reduce}(B, \texttt{max})$ ▷ Reduction; $O(n+m)$
2: $C = \texttt{eMult}(B, (\mathbf{b} \ +.\mathsf{first} \ \mathbf{z}^T)\langle B \rangle, \mathsf{iseq})$ ▷ Masked m×m-mult, e-wise mult; $O(n+m)$
3: $\mathbf{c} = C \ \mathsf{max.secondi} \ z^T$ ▷ m×v-mult; $O(n+m)$
4: Create matrix K with $K(p,q) = q$ for $C(p,q)$ explicit ▷ $O(n+m)$
5: $B_{mask} = \texttt{eMult}(K, (\mathbf{c} \ +.\mathsf{first} \ \mathbf{z}^T)\langle C \rangle, \mathsf{iseq})$ ▷ Masked m×m-mult, e-wise mult; $O(n+m)$
6: $B_{mask} = \texttt{select}(B_{mask}, \texttt{nonzeros})$ ▷ Selection; $O(n+m)$
7: $B^\star = B\langle B_{mask} \rangle$ ▷ Masking; $O(n)$

Chapter 3

Algorithm

This chapter contains a detailed description of our algorithm, listed in algorithm 2, and its main components.

Algorithm 2 MWM approximation algorithm - main loop

Input: Adjacency matrix A of a weighted undirected graph $G(V, E)$, an iteration strategy `STRAT`

Output: Matrix M reflecting a 3/4-approximation of the MWM of G

1: $M = \mathcal{M}_\emptyset$
2: $k = 1$
3: **while** any positive-gain k-augmentation exists for $k = 1, 2, 3$ **do**
4: $\quad \mathcal{A} =$ `SearchAugmentations`(A, M, k)
5: $\quad M =$ `FlipAugmentations`$(A, M, \mathcal{A})$
6: $\quad k =$ `SelectStrategy`$(\texttt{STRAT}, k)$
7: **end while**

We take the adjacency matrix A of a weighted undirected graph $G(V, E)$. Starting with a completely empty matching, we set parameter $k = 1$ and start the main loop of the algorithm which consists of searching a set of augmentations, applying them to the matching and selecting a new value for k. Based on the passed parameter k, method `SearchAugmentations` uses one of three distinct methods to obtain a set of positive-gain k-augmentations. However, each method has a similar structure:

1. Find all potential locations where a k-augmentation might exist, which we refer to as its center.
2. Calculate the gain of the best k-augmentation for each center. This ensures that we find at least one positive-gain augmentation if one exists. These values are saved in matrix G_k. Under some conditions a center might not have any k-augmentation available, and the method should then automatically produce an implicit value for this center in G_k. Information on the vertices that an augmentation contains is saved in a matrix D_k.
3. Select only positive values in G_k, since we are only interested in applying positive-gain augmentations. If G_k is now completely empty, terminate the algorithm early and return $\mathcal{A} = \mathcal{M}_\emptyset$.

4. Solve conflicts between the found set of augmentations, as it is most likely that not all can be applied at the same time without issues. We solve the conflicts by reducing the set of found augmentations, while making sure the set will retain some of its elements, striving to keep this number large.

5. Create matrix $\mathcal{A}$ from D_k that indicates the edges that can be added to the matching to apply the resulting set of augmentations.

After finding the set of augmentations, we apply them to the matching using the flipping algorithm in `FlipAugmentations`. Then we decide the new value for parameter k, either $1, 2$ or 3, using a predetermined strategy and start a new iteration. These strategies are discussed in chapter 4.

In section 3.1, we give an algorithm for applying a set of augmentations for general k, introducing the methodology of the GraphBLAS standard in a practical setting. The searching methods for $k = 1, 2, 3$ are described separately and in full detail in sections 3.2 to 3.4, each building on the previous section.

3.1 Flipping algorithm

Each method for searching the different types of k-augmentations results in an augmentation matrix $\mathcal{A}$ indicating with its explicit values all edges that should be added to the matching. Therefore we require a method that uses this matrix to efficiently add the correct edges to the matching and that removes any previously matched edges adjacent to the newly added ones. A useful observation is the possibility to use an element-wise multiplication with the second-operator of $\mathcal{A}$ and adjacency matrix A and add the result element-wise to M to easily add the newly matched edges to the matching. We will use a similar method to remove the correct edges.

The edges $(v_1, v_2) \in E$ that need to be removed are currently in the matching, so $M(v_1, v_2)$ and $M(v_2, v_1)$ are explicit values. They only need to be removed if (v_1, v_3) or (v_2, v_4) will be added to the matching for some $v_3, v_4 \in V$. In other words, we need to remove (v_1, v_2) if $\mathcal{A}(v_1, v_3)$ or $\mathcal{A}(v_2, v_4)$ are explicit. Since both M and $\mathcal{A}$ have at most one element per row, we can check for these conditions by reducing the matrices by row to vectors $\mathbf{m}$ and $\mathbf{a}$ respectively, and finding the vector $\mathbf{r}$ that is the intersection of their sparsity patterns. The intersection is found using an element-wise multiplication with the pair operator, which returns $\mathsf{pair}(\mathbf{m}(v), \mathbf{a}(v)) = 1$ when both $\mathbf{m}(v)$ and $\mathbf{a}(v)$ are explicit values and zero otherwise. The indices of the explicit values in this vector $\mathbf{r}$ mark the vertices whose currently matched edge needs to be removed. We can find the correct elements in M by multiplying $\mathbf{r}$ with a transposed full vector $\mathbf{z}$ of values 1 and using M as a mask. To the resulting matrix R we add its own transpose, to make sure we remove both $M(v_1, v_2)$ and $M(v_2, v_1)$. A standard element-wise product of R and M results in a matrix that allows us to remove the correct elements by an element-wise subtraction from M. Now the complete update of the matching is done by

$$M := M + \mathcal{A}\ (\otimes, \mathsf{second})\ A - R\ (\otimes, \times)\ M.$$

We end with removing any explicit zeros from the result.

A full pseudocode for this method of flipping the correct edges for general k-augmentations can be found in algorithm 3. All steps are operations like reductions, element-wise multiplications and additions, and masked matrix-multiplications, but only on matrices with $O(n)$ explicit elements. This leads to a time complexity of $O(n)$, detailed further in the pseudocode.

Algorithm 3 `FlipAugmentations`: Flipping algorithm for general k-augmentations

Input: Augmentation matrix $\mathcal{A}$, matching M, a full vector $\mathbf{z}$ of explicit values 1

Output: An updated matching M with improved weight

1: $\mathbf{m} = \texttt{reduce}(M)$ ▷ Reduction; $O(n)$
2: $\mathbf{a} = \texttt{reduce}(\mathcal{A})$ ▷ Reduction; $O(n)$
3: $\mathbf{r} = \texttt{eMult}(\mathbf{m}, \mathbf{a}, \mathsf{pair})$ ▷ Element-wise mult; $O(n)$
4: $R = (\mathbf{r} +.\times\ \mathbf{z}^T)\langle M \rangle$ ▷ Masked m×m-mult; $O(n)$
5: $R = R + R^T$ ▷ Element-wise add; $O(n)$
6: $M = M + \texttt{eMult}(\mathcal{A}, A, \mathsf{second}) - \texttt{eMult}(R, M, \times)$ ▷ Element-wise add + mult; $O(n)$
7: $M = \texttt{select}(M, \texttt{nonzeros})$ ▷ Selection; $O(n)$

3.2 Finding 1-augmentations

A 1-augmentation is an unmatched edge (i, j) in a graph and possibly a matched edge (i, k) adjacent to i and another matched edge (j, l) adjacent to j. We define i as its start vertex and j as its end vertex, and say that a 1-augmentation is centered around (i, j). For each vertex v in the graph, we would like to select the best possible positive-gain 1-augmentation with v as its start vertex. After properly removing any occurring conflicts, we are left with a set of 1-augmentations that we can use to improve our matching, that will not be empty if any positive-gain 1-augmentation exists. To this end, we calculate the matrix $[A \setminus M]$ by performing an element-wise addition using the minus-operator of A and M, and selecting the remaining non-zeros a_{ij} from the result. Each explicit element in $[A \setminus M]$ contains the weight of the unmatched edge centered in a potential 1-augmentation. For all of these centers there exists a 1-augmentation, which means that only for start vertices that are matched and that are not incident to any unmatched edges there does not exist a 1-augmentation.

Our aim is to transform this matrix to contain elements reflecting the gain of each of these 1-augmentations. We start by calculating the vector $\mathbf{m}_w$ containing the weight of a matched edge adjacent to vertex v at its corresponding index, if v is indeed in the current matching. This is done by summing over each row in M and thus reducing the matrix to a vector. The gain of any 1-augmentation is expressed as

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

where we use the indicator function corresponding to some edge set E

$$\mathbb{1}_E(v_1, v_2) = \begin{cases} 1 & \text{if } (v_1, v_2) \in E \\ 0 & \text{if } (v_1, v_2) \notin E. \end{cases}$$

Therefore, finding each correct gain value of the 1-augmentations in $[A \setminus M]$ requires subtracting the weights of the possible matched edges adjacent to i and j. We do this by calculating matrix $(\mathbf{m}_w +.+\ \mathbf{m}_w^T)$ with a mask $[A \setminus M]$, resulting in the value $\mathbf{m}_w(i) + \mathbf{m}_w(j)$ at the relevant positions (i, j). This requires $\mathbf{m}_w$ to be a fully explicit vector, as the multiplication of $\mathbf{m}_w(i)$ and $\mathbf{m}_w(j)$ will result in an implicit value if either one is implicit, even when we use the addition operator for multiplication. Performing another element-wise subtraction gives the desired result

$$G_1 = [A \setminus M] - (\mathbf{m}_w +.+\ \mathbf{m}_w^T)\langle [A \setminus M] \rangle.$$

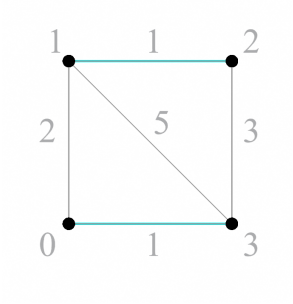

Figure 3.1: An example graph containing three possible 1-augmentations.

We can perform these calculations for the instance described in fig. 3.1, where

$$(\mathbf{m}_w \mathbin{+.+} \mathbf{m}_w^T)\langle [A \setminus M] \rangle = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \begin{bmatrix} & 2 & & \\ 2 & & & 2 \\ & & & 2 \\ & 2 & 2 & \end{bmatrix}, \quad \begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix}$$

And so,

$$G_1 = \underset{[A \setminus M]}{\begin{bmatrix} & 2 & & \\ 2 & & & 5 \\ & & & 3 \\ & 5 & 3 & \end{bmatrix}} - \underset{(\mathbf{m}_w \mathbin{+.+} \mathbf{m}_w^T)\langle [A \setminus M] \rangle}{\begin{bmatrix} & 2 & & \\ 2 & & & 2 \\ & & & 2 \\ & 2 & 2 & \end{bmatrix}} = \begin{bmatrix} & 0 & & \\ 0 & & & 3 \\ & & & 1 \\ & 3 & 1 & \end{bmatrix}.$$

If all values in G_1 are nonpositive, we cannot improve our matching using 1-augmentations and the algorithm can be terminated. First we will select the elements whose value is greater than zero in G_1 and then check the number of remaining explicit values of G_1, which is done in $O(1)$ time in SuiteSparse:GraphBLAS. If there are no explicit values in G_1, we terminate the method and return an empty set of augmentations.

In order to obtain the desired set of 1-augmentations to improve the matching, we aim to create a matrix with the values in G_1 but selecting only the highest positive-gain 1-augmentation for each row, and choose the one with the highest column index in the case of ties. So we can use the algorithm from section 2.3 to select these values, resulting in matrix D_1. The explicit values of D_1 indicate at most one positive-gain 1-augmentation for each starting vertex with the highest possible gain for that vertex.

Now we will adapt D_1 to indicate a set of 1-augmentations that do not conflict with each other. Any 1-augmentation centered around (i, j) is completely determined by its start and end vertex. The only conflicts between two 1-augmentations centered around (i, j) and (i', j') that can occur are: the sharing of a start vertex $i = i'$; the sharing of an end vertex $j = j'$; or a start vertex being the end vertex of the other augmentation $i = j'$. Since we have selected at most one 1-augmentation for each start vertex, the first conflict cannot occur here. We choose to solve the other two types of conflicts by an element-wise multiplication of D_1 and its own transpose, resulting in $\mathcal{A}$. We do this using the pair operator, which returns $\mathsf{pair}(D_1^T(p,q), D_1(p,q)) = 1$ when both $D_1^T(p,q)$ and

$D_1(p, q)$ are explicit values and zero otherwise. Calculating the transpose of D_1 is fast enough since it contains $O(n)$ elements. Going back to the instance in fig. 3.1, the calculation is

$$\overset{D_1^T}{\begin{bmatrix} & & \\ & & 3 \\ 3 & 1 & \end{bmatrix}} \overset{}{(\otimes, \mathsf{pair})} \overset{D_1}{\begin{bmatrix} & & 3 \\ & & 1 \\ 3 & & \end{bmatrix}} = \overset{\mathcal{A}}{\begin{bmatrix} & & 1 \\ & & \\ 1 & & \end{bmatrix}}.$$

The conflict of sharing an end vertex results in two explicit values in column j of D_1. Since we have at most one value per row, at least one of the two values in column j will disappear after the multiplication. When we have $i = j'$, entry $D_1(j, i)$ will not be an explicit value and so $\mathcal{A}(i, j)$ will be zero after the multiplication. We will refer to $\mathcal{A}$ as the augmentation matrix. One can easily check that if two 1-augmentations share different vertices than mentioned above, both can be applied without issues.

A full pseudocode listing for this method of searching for 1-augmentations can be found in algorithm 4. Being a straightforward combination of selection, reduction, element-wise multiplication and addition and previously described methods, it has a time complexity of $O(n + m)$, as detailed per step in the pseudocode comments.

Algorithm 4 `SearchAugmentations(A, M, 1)`: Searching for 1-augmentations

Input: Adjacency matrix A, matching M, a full vector $\mathbf{z}$ of explicit values 1
Output: Augmentation matrix $\mathcal{A}$ with explicit values indicating a set of 1-augmentations suitable for improving the matching

1: $[A \setminus M] = \texttt{select}(A - M, \texttt{nonzeros})$ ▷ Selection, e-wise add; $O(n + m)$
2: $\mathbf{m}_w = \texttt{reduce}(M)$ ▷ Reduction; $O(n)$
3: $\mathbf{m}_w = \texttt{fullvector}(\mathbf{m}_w)$ ▷ $O(n)$
4: $G_1 = [A \setminus M] - (\mathbf{m}_w +.+ \mathbf{m}_w^T)\langle[A \setminus M]\rangle$ ▷ Masked m×m-mult, e-wise add; $O(n + m)$
5: $G_1 = \texttt{select}(G_1, \texttt{positive})$ ▷ Selection; $O(n + m)$
6: **if** $G_1 \neq \mathcal{M}_\emptyset$ **then** ▷ $O(1)$
7: $\quad D_1 = \texttt{maxPerRow}(G_1)$ ▷ Max per row algorithm; $O(n + m)$
8: $\quad \mathcal{A} = \texttt{eMult}(D_1^T, D_1, \mathsf{pair})$ ▷ E-wise mult, transposition; $O(n + m)$
9: **else**
10: $\quad \mathcal{A} = \mathcal{M}_\emptyset$ ▷ $O(1)$
11: **end if**

3.3 Finding 2-augmentations

A 2-augmentation is either a 1-augmentation or an augmentation consisting of exactly two unmatched edges. We will refer to the latter type as a *proper* 2-augmentation. These proper 2-augmentations are centered around a matched edge (i, j) and have two adjacent unmatched edges (i, k) and (j, l) where vertices k and l possibly are matched as well. We use the notation as tuple (i, j, k, l) for such a proper 2-augmentation. Here we will describe an algorithm for searching a set of positive-gain proper 2-augmentations. We will only discuss proper 2-augmentations in this section,

so any 2-augmentation is assumed to be proper here. Taking each edge in the matching as the center of a proper 2-augmentation, we try to find the highest positive-gain augmentation for each matched edge. After removing conflicts, we are left with a set of augmentations that will not be empty if any positive-gain augmentation exists.

The crucial observation is to see that a proper 2-augmentation consists of two 1-augmentations sharing either one or two matched edges. We will refer to these as *path* and *cycle* augmentations, respectively. Generally, we will obtain the best proper 2-augmentation centered around (i, j) if we find and combine the best 1-augmentations with i and j as start vertices. Its gain is the sum of the gain of both 1-augmentations and the weight of the center matched edge. Note that two negative-gain 1-augmentations can be combined into a positive-gain 2-augmentation because of this correction to the gain with the weight of the center. The method of calculating the gain this way fails when it finds a path augmentation, when in truth a cycle augmentation is optimal. The cause is the true gain of that cycle augmentation being the sum of the gains of its 1-augmentations where a correction must be made for both matched edges. We choose to calculate both the best cycle and path proper 2-augmentations separately and select the best one for each matched edge afterwards. A path augmentation will always exist for a matched edge if either endpoint has an available 1-augmentation, but a cycle augmentation is not guaranteed. The two algorithms below will reflect this fact and the selection procedure will also handle this correctly, as described in each corresponding section.

For each center, we require the following results from both the path and cycle augmentation method if a corresponding positive-gain augmentation exists: the center edge (i, j), the gain of the highest-gain augmentation and the vertices k and l. We will collect the gain values in matrices G_2^c and G_2^p at locations (i, j), for cycle and path augmentations, respectively. We can combine them with an element-wise addition using the max operator, effectively selecting the best of both augmentations per center edge, by

$$G_2 = G_2^c \ (\oplus, \mathsf{max}) \ G_2^p.$$

If G_2 is empty because neither method has found any positive-gain augmentations, we can return $\mathcal{A}$ as an empty matrix.

Vertices k and l are stored in matrices D_2^c for cycle augmentations and D_2^p for path augmentations, by one explicit value per row at (i, k) and (j, l). We need to combine these matrices into D_2 according to the selection of path and cycle augmentations made for G_2. We will create a vector $\mathbf{c}$ that indicates which rows of D_2^c we need, and from D_2^p we can then take the rows indicated by the complement of $\mathbf{c}$. We create matrix C by

$$C = (\mathbf{z} \ +.\times \ \mathbf{z}^T)\langle G_2^c \rangle,$$

so C has values 1 where a positive-gain cycle augmentation is possible. Then we apply the complement of G_2^p as a mask, now keeping only the values where there does not exist a path augmentation and thus we need the cycle to be selected there. Now we multiply G_2^c and G_2^p element-wise using the greater-than operator to find those centers where a cycle is optimal, and add the result to C. Reducing C to a vector $\mathbf{c}$ gives us the vector we require.

In handling conflicts, issues and edge cases that occur only at a subset of vertices, we would like to implement a type of conditional for vertices, where we check if a certain situation arises at a vertex and then take some action if this is the case. If we can construct a vector with explicit values at indices corresponding to exactly those vertices, we can use it to select specific rows and columns in matrices and values in vectors that will be modified. Vector $\mathbf{c}$ is an example of this, as is using vector $\mathbf{m}$ to only work on vertices that are matched. By taking intersections or unions we can create a vector that combines different conditionals.

A situation that occurs often, is the need to operate on just one vertex of a matched pair (i, j). We might make the choice to only operate on values in a matrix corresponding to matched edges where $i < j$. We can find these indices by reducing the upper triangular part of matching M to a vector $\mathbf{m}_U$, since it only contains explicit values at (i, j) in rows where $i < j$. This means $\mathbf{m}_U$ has an explicit value at index i but not at index j. Equivalently, $\mathbf{m}_L$ is a reduction of the lower triangular part of M, indicating the highest indices of matched pairs.

3.3.1 Cycle augmentations

This section discusses the algorithm to find the best cycle proper 2-augmentation for each matched edge in A. The steps are given in pseudocode in algorithm 5. We aim for a matrix G_2^c that has explicit values $G_2^c(i, j)$ containing the gain of the best positive-gain cycle 2-augmentation (i, j, k, l), and a matrix D_2^c with $D_2^c(i, k)$ and $D_2^c(j, l)$ explicit.

A matched edge (i, j) can certainly be the center of multiple cycle augmentations. We will calculate the gain value of each and select the best one. Without loss of generality, we will assume in the following description of our algorithm that (i, j) is the center of two cycle augmentations (i, j, k, l) and (i, j, k', l'), where $k \neq k'$ and $l \neq l'$. Figure 3.2 describes this situation.

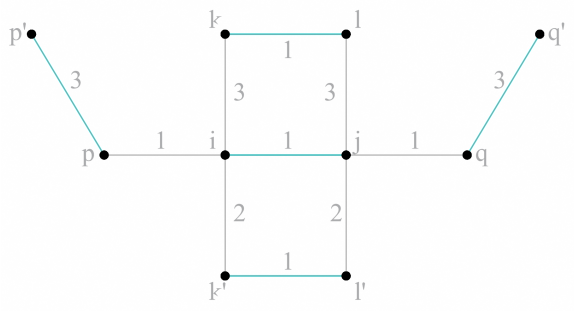

Figure 3.2: Two cycle augmentations with the same center (i, j).

In all cycle augmentations, every vertex is matched. Thus we select only the rows and columns of A of matched vertices using vector $\mathbf{m}$, producing $A_{\mathbf{m}}$. This will also allow us to use the permutation matrix corresponding to M for permuting the rows and columns of $A_{\mathbf{m}}$. For a matched edge (i, j), we need to check if one of the unmatched connections of i is matched to one of the unmatched connections of j. If this is the case, a cycle augmentation exists. We permute the rows of A with P^M and thus swap rows i and $m(i)$, so row i of the result will contain values indicating all edges connected to $j = m(i)$. Permuting the columns of the result, gives a matrix $A_{\mathbf{m}}^M$ that indicates in row i the vertices that are matched to vertices connected to j. Naturally, these two steps can be performed in any order, so we choose

$$A_{\mathbf{m}}^M = P^M A_{\mathbf{m}} P^M.$$

Now $A_{\mathbf{m}}(i, k)$ and $A_{\mathbf{m}}^M(i, k)$ contain the weight of edges (i, k) and (j, l), respectively. For unmatched edges incident to i or j that are not part of a cycle 2-augmentation at least one of these values is implicit. By element-wise multiplication using the $+$ operator we add the weights of both edges or get an implicit zero, so

$$C = A_{\mathbf{m}} \ (\otimes, +) \ A_{\mathbf{m}}^M.$$

Illustrating this using the instance in fig. 3.2 with disappearing explicit values highlighted in red, this calculation is

$$
\begin{array}{c|cccccccccc}
 & i & j & k & l & k' & l' & p & q & p' & q' \\
i & & 1 & 3 & & 2 & & 1 & & & \\
j & 1 & & & 3 & & 2 & & 1 & & \\
k & 3 & & & 1 & & & & & & \\
l & & 3 & 1 & & & & & & & \\
k' & 2 & & & & & 1 & & & & \\
l' & & 2 & & & 1 & & & & & \\
p & 1 & & & & & & & & 3 & \\
q & & 1 & & & & & & & & 3 \\
p' & & & & & & & 3 & & & \\
q' & & & & & & & & 3 & &
\end{array}
\quad (\otimes, +) \quad
\begin{array}{c|cccccccccc}
 & i & j & k & l & k' & l' & p & q & p' & q' \\
i & & 1 & 3 & & 2 & & & & & 1 \\
j & 1 & & & 3 & & 2 & & & 1 & \\
k & 3 & & & 1 & & & & & & \\
l & & 3 & 1 & & & & & & & \\
k' & 2 & & & & & 1 & & & & \\
l' & & 2 & & & 1 & & & & & \\
p & & & & & & & & & 3 & \\
q & & & & & & & & & & 3 \\
p' & & 1 & & & & & 3 & & & \\
q' & 1 & & & & & & & 3 & &
\end{array}
=
$$

$A_{\mathbf{m}}$ $\qquad\qquad$ $A^M_{\mathbf{m}}$

$$
\begin{array}{c|cccccccccc}
 & i & j & k & l & k' & l' & p & q & p' & q' \\
i & & 2 & 6 & & 4 & & & & & \\
j & 2 & & & 6 & & 4 & & & & \\
k & 6 & & & 2 & & & & & & \\
l & & 6 & 2 & & & & & & & \\
k' & 4 & & & & & 2 & & & & \\
l' & & 4 & & & 2 & & & & & \\
p & & & & & & & & 6 & & \\
q & & & & & & & & & 6 & \\
p' & & & & & & & 6 & & & \\
q' & & & & & & & & 6 & &
\end{array}
\,.
$$

C

Only the unmatched edges (i, p) and (j, q) are not part of a cycle augmentation, so the corresponding elements disappear in C. Of course, $A_{\mathbf{m}}(i, j)$ and $A^M_{\mathbf{m}}(i, j)$ both contain $w(i, j)$ so for those elements the multiplication leads to $C(i, j) = 2w(i, j)$. Suppose a matched edge is not part of any cycle augmentation, then $C(i, j)$ will be the only explicit value in row i.

The goal here is to have $C(i, k)$ be the gain of cycle 2-augmentation (i, j, k, l). We need to subtract the weight of matched edges (i, j) and (k, l) to achieve this. We create an $n \times n-$matrix where each element contains the sum of the weight of these edges by $(\mathbf{m}_w +.+ \mathbf{m}_w^T)\langle C\rangle$ and subtract the matrix from C, resembling the step in calculating 1-augmentations. Clearly, $C(i, j)$ is now an explicit zero.

Now row i of C contains explicit values at $C(i, k)$ and $C(i, k')$. We will select the maximum value per row and the value with the highest column index in case of ties. However, supposing $C^\star(i, k)$ is selected, we have no guarantee that $l' < l$ and that $C(j, l)$ is selected as well. To work around this issue, we select only the rows of matched edges (i, j) where $i < j$ using $\mathbf{m}_U$. The result is D_2^c. Only the positive-gain augmentations are relevant, so we select these. For any center that is not part of a cycle augmentation, all explicit values will have been removed from the corresponding rows at this point. If D_2^c is completely empty, we can return both G_2^c and D_2^c as empty matrices, otherwise we need to perform these last few steps. With $D_2^c(i, k)$ being the only explicit value in that row as needed, we will also set $D_2^c(j, l) = D_2^c(i, k)$ as the only explicit value in row j, which is currently

fully implicit. This is done by a permutation of rows and columns with P^M as seen above and then adding it to the original

$$D_2^c := D_2^c + P^M D_2^c P^M.$$

The desired result G_2^c is derived directly from matrix D_2^c. The only explicit value in row i of G_2^c should be $G_2^c(i,j) = D_2^c(i,k)$, so the result $\mathbf{r}$ of a reduction by row of D_2^c allows us to put the explicit values at the correct location by calculating $G_2^c = (\mathbf{r} +.\mathsf{first}\ \mathbf{z}^T)\langle M\rangle$.

Algorithm 5 `SearchCycle2Augs`: Searching for cycle 2-augmentations

Input: Adjacency matrix A, matching M, vectors $\mathbf{m}, \mathbf{m}_w, \mathbf{m}_U, \mathbf{m}_L$, a full vector $\mathbf{z}$ of explicit values 1

Output: Matrices G_2^c and D_2^c

1: $A_{\mathbf{m}} = \texttt{selectColums}(A, \mathbf{m})$ ▷ Row/Column selection; $O(n+m)$
2: $A_{\mathbf{m}} = \texttt{selectRows}(A_{\mathbf{m}}, \mathbf{m})$ ▷ Row/Column selection; $O(n+m)$
3: $A_{\mathbf{m}}^M = \texttt{permuteRows}(A_{\mathbf{m}}, P^M)$ ▷ Permutation; $O(n+m)$
4: $A_{\mathbf{m}}^M = \texttt{permuteColums}(A_{\mathbf{m}}^M, P^M)$ ▷ Permutation; $O(n+m)$
5: $C = \texttt{eMult}(A_{\mathbf{m}}^M, A_{\mathbf{m}}, +)$ ▷ Element-wise mult; $O(n+m)$
6: $C = C - (\mathbf{m}_w +.+\ \mathbf{m}_w^T)\langle C\rangle$ ▷ Masked m×m, e-wise add; $O(n+m)$
7: $D_2^c = \texttt{selectRows}(C, \mathbf{m}_U)$ ▷ Row/Column selection; $O(n+m)$
8: $D_2^c = \texttt{maxPerRow}(D_2^c)$ ▷ Max per row algorithm; $O(n+m)$
9: $D_2^c = \texttt{select}(D_2^c, \texttt{positive})$ ▷ Selection; $O(n)$
10: **if** $D_2^c \neq \mathcal{M}_\emptyset$ **then** ▷ $O(1)$
11: $D_2'^c = \texttt{permuteRows}(D_2^c, P^M)$ ▷ Permutation; $O(n)$
12: $D_2'^c = \texttt{permuteColums}(D_2'^c, P^M)$ ▷ Permutation; $O(n)$
13: $D_2^c = D_2^c + D_2'^c$ ▷ Element-wise add; $O(n)$
14: $\mathbf{r} = \texttt{reduce}(D_2^c)$ ▷ Reduction; $O(n)$
15: $G_2^c = (\mathbf{r} +.\mathsf{first}\ \mathbf{z}^T)\langle M\rangle$ ▷ Masked m×m-multiplication; $O(n)$
16: **else**
17: $G_2^c = \mathcal{M}_\emptyset$ ▷ $O(1)$

3.3.2 Path augmentations

This section discusses the algorithm to find the best path augmentation for each matched edge (i,j). The steps are given in pseudocode in algorithm 6. As stated before, the best path proper 2-augmentation is a combination of the best 1-augmentations for (i,k) and (j,l). These 1-augmentations can be found using the algorithm in section 3.2, where they are stored in matrix D_1, which contains the best 1-augmentation for each starting vertex. The gain values of all 1-augmentations are stored in G_1. We are only interested in 1-augmentations for starting vertices that are matched, so we can select these rows using vector $\mathbf{m}$ from D_1, resulting in matrix D_2^p. A multiplication with the $+.\mathsf{secondi}$ operator of D_2^p and a full vector $\mathbf{z}$ puts in vector $\mathbf{a}$ the end vertex of each 1-augmentation, and a reduction per row of D_2^p leads to a vector $\mathbf{a}_g$ containing the gain of each augmentation. Here we aim to construct the matrix G_2^p that has explicit values $G_2^p(i,j)$ that are the gain of the best path augmentation with center (i,j). This gain consists of the gain of both 1-augmentations stored

in D_2^p and the weight of the center matched edge, which has been corrected for twice in calculating the gain of the 1-augmentations, and thus

$$\begin{aligned} G_2^p(i,j) &= D_2^p(i,k) + D_2^p(j,l) + M(i,j), \\ &= \mathbf{a}_g(i) + \mathbf{a}_g(j) + M(i,j). \end{aligned}$$

Similar to the construction of G_1, we build G_2^p by calculating

$$G_2^p = (\mathbf{a}_g \ +.+ \ \mathbf{a}_g^T)\langle M \rangle + M.$$

We now have a matrix containing the gain of the best 2-augmentation per matched edge used as the centre, with at most one element per row. We select only the positive-gain augmentations. Suppose one of either endpoints i or j does not have any available 1-augmentations, then $G_2^p(i,j)$ would become implicit since multiplication with any implicit value, even using the addition operator, will not produce an explicit value.

Ideally, we would now like to make the selection between path and cycle augmentations and remove any conflicts. However, we need to remove certain invalid path augmentations. These can only exist if G_2^p is not empty, otherwise we can return G_2^p and D_2^p as empty matrices.

Our method of combining 1-augmentations might lead to one type of invalid 2-augmentation, where $k = l$, as illustrated in fig. 3.3.

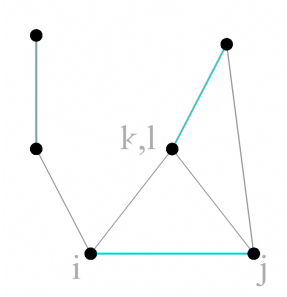

Figure 3.3: An invalid path augmentation found in an example graph where $k = l$. Changing either of the 1-augmentations leads to a valid 2-augmentation.

These invalid augmentations cannot be applied so they must be removed from the set, but (i,j) might be the center of another positive-gain proper 2-augmentation that is valid. To meet the maximum weight lower bound we need to search for it. Describing this case from the viewpoint of vertex i, we have $\mathbf{a}(i) = \mathbf{a}(\mathbf{m}(i))$. As before, we would like to solve this issue at every occurrence at once using matrix operations, so we will construct a vector $\mathbf{c}$ indicating with the indices of its explicit values the vertices i where $\mathbf{a}(i) = \mathbf{a}(\mathbf{m}(i))$. Now using P^M for permutation of the values in vectors, we can see that calculating $\mathbf{v} = P^M\mathbf{a}$ gives $\mathbf{v}(i) = \mathbf{a}(\mathbf{m}(i))$. A simple element-wise multiplication $\mathbf{c} = \mathbf{a}\ (\otimes, \mathsf{iseq})\ \mathbf{v}$ leads to the desired vector $\mathbf{c}$ representing the conditional $\mathbf{a}(i) = \mathbf{a}(\mathbf{m}(i))$ for each matched vertex i.

We choose to try to create a new 2-augmentation using the second-best 1-augmentation starting at i, assuming $i < j$. The new 2-augmentation will be valid, since $k = l$ cannot occur now. This might not lead to a positive-gain 2-augmentation, so we also attempt using the second-best 1-augmentation at j, which must also result in a valid 2-augmentation. These two steps are performed independent of each other so the order does not matter. We choose to operate first on the vertices that are

indicated by both $\mathbf{c}$ and $\mathbf{m}_U$, so the pair operator can be used in an element-wise multiplication to obtain the intersection of both vectors. We choose to remove the best 1-augmentations for i from G_1 and calculate the rest of the algorithm for 1-augmentations again, now leading to G^p_{2u} and D^p_{2u}. The same is done for vertices indicated by $\mathbf{c}$ and $\mathbf{m}_L$, leading to matrices G^p_{2l} and D^p_{2l}. These matrices are then combined into the final G^p_2 and D^p_2, choosing the best 2-augmentation for each center. The specific steps for removing the best 1-augmentations from G_1 are omitted, but consist of straightforward combinations of element-wise additions and multiplications of matrices with the right rows selected.

Combining the two best 1-augmentations might also result in a cycle 2-augmentation. In this case, the gain that is calculated will be incorrect; its value is too low. We choose to leave this as is. The cycle algorithm will find the same or a better cycle augmentation, and the algorithm will prefer it over the incorrect one.

Algorithm 6 `SearchPath2Augs`: Searching for path 2-augmentations

Input: Adjacency matrix A, matching M, vectors $\mathbf{m}, \mathbf{m}_w, \mathbf{m}_U, \mathbf{m}_L$, a full vector $\mathbf{z}$ of explicit values 1

Output: Matrices G^p_2 and D^p_2

$G_1, D_1 = \texttt{SearchAugmentations}(A, M, 1)$ ▷ $O(n+m)$
$D^p_2 = \texttt{selectRows}(D_1, \mathbf{m})$ ▷ Row/Column selection; $O(n)$
$\mathbf{a} = D^p_2$ +.secondi $\mathbf{z}$ ▷ m×v-multiplication; $O(n)$
$\mathbf{a}_g = \texttt{reduce}(D^p_2)$ ▷ Reduction; $O(n)$
$G^p_2 = (\mathbf{a}_g$ +.+ $\mathbf{a}^T_g)\langle M\rangle + M$ ▷ Masked m×m-mult, e-wise add; $O(n)$
$G^p_2 = \texttt{select}(G^p_2, \texttt{positive})$ ▷ Selection; $O(n)$
if $G^p_2 \neq \mathcal{M}_\emptyset$ **then** ▷ $O(1)$
    $\mathbf{v} = \texttt{permuteVector}(\mathbf{a}, P^M)$ ▷ Permutation; $O(n)$
    $\mathbf{c} = \texttt{eMult}(\mathbf{a}, \mathbf{v}, \text{iseq})$ ▷ Element-wise multiplication; $O(n)$
    $\mathbf{c}_u = \texttt{eMult}(\mathbf{c}, \mathbf{m}_U, \text{pair})$ ▷ Element-wise multiplication; $O(n)$
    $\mathbf{c}_l = \texttt{eMult}(\mathbf{c}, \mathbf{m}_L, \text{pair})$ ▷ Element-wise multiplication; $O(n)$
    Remove best 1-augmentations from G_1 in rows of $\mathbf{c}_u$ and calculate G^p_{2u} and D^p_{2u} ▷ $O(n+m)$
    Remove best 1-augmentations from G_1 in rows of $\mathbf{c}_l$ and calculate G^p_{2l} and D^p_{2l} ▷ $O(n+m)$
    Combine G^p_{2u} and G^p_{2l} with G^p_2, and D^p_{2u} and D^p_{2l} with D^p_2 to obtain the correct positive-gain path 2-augmentations. ▷ $O(n+m)$
else
    $D^p_2 = \mathcal{M}_\emptyset$ ▷ $O(1)$

3.3.3 Conflicts

Now we have obtained the best 2-augmentation for each matched edge and we will select a subset of these that do not have any conflicts. Two distinct proper 2-augmentations (i, j, k, l) and (i', j', k', l') cannot share any of their vertices if applied, since all the vertices will be matched after flipping, which in all but one case leads to an invalid matching. The one exception occurs when $k = j', i =$

$l', j = \mathbf{m}(l')$ (or any analogous situations, which effectively are 3-augmentations), which is allowed, but for simplicity of the method below we will remove one of the augmentations anyway. From any set of conflicting augmentations, only the one with the highest gain will be selected. We set up a matrix Γ where the gain of each augmentation is an explicit value at $(i,i),(i,j),(i,k)$ and (i,l), so at four locations in the row corresponding to the start vertex. The gain of each augmentation is the result of the reduction of G_2 to $\mathbf{a}_g$. Matrix Γ can be obtained by the following operations

$$\Gamma_{mask} = diag(\mathbf{m}) + M + D_2 + P^M D_2,$$
$$\Gamma = (\mathbf{a}_g \ +.\mathsf{first}\ \mathbf{z}^T)\langle\Gamma_{mask}\rangle.$$

Each of the terms adds one of the necessary explicit values per row to Γ_{mask}, which is then used to assign the correct gain value at the right indices in each row. If a column of Γ has more than one explicit value, that vertex is part of multiple proper 2-augmentations, which we do not allow. We choose to select the maximum per column and use the highest column index to break ties. Next, the number of explicit values per row is calculated using a $+.second$ multiplication of Γ with a full vector of values 1. Now we select only the values that are equal to four, as these values reflect the 2-augmentations that 'survived' the selection process. The result is vector $\mathbf{v}$. Matrix D_2 contains exactly the unmatched edges that need to be flipped for applying each 2-augmentation. If we select the rows indicated by $\mathbf{v}$ from D_2 and add to it its own transpose, we get the desired matrix $\mathcal{A}$ that can be passed on to the flipping procedure.

The algorithms for searching cycle and path 2-augmentations and combining them are listed as separate methods in pseudocode in algorithms 5 to 7. Every step consists of basic steps like selection, reduction to a vector, (element-wise) matrix multiplication and addition, or an algorithm seen earlier in the thesis. Often we work with matrices with only one element per row/column, which in this case leads to a runtime of $O(n)$ for algorithm 7 after the first two steps. Algorithms 5 and 6 have a runtime of $O(n+m)$. This is detailed further in each pseudocode listing.

3.4 Finding 3-augmentations

A 3-augmentation is either an augmentation containing exactly three unmatched edges or a (possibly proper) 2-augmentation. Like before, we will refer to 3-augmentations with exactly three unmatched edges as *proper* 3-augmentations and this section will focus on finding this type. The 3-augmentations are centered around an unmatched edge (i,j), and vertices i and j are both matched, to k and l respectively. Now k and l are incident to unmatched edges (k,p) and (l,q), and p and q are possibly matched vertices as well. We denote a 3-augmentation by the tuple (i,j,k,l,p,q). Again, we can make a distinction between *path* and *cycle* augmentations. For a valid path augmentation, all vertices i,j,k,l,p,q and possibly $m(p)$ and $m(q)$ are pairwise distinct. This can easily be verified by considering each possible instance, but intuitively one can see that each vertex is incident to an unmatched edge, and an alternating path where two vertices coincide will have two adjacent unmatched edges or it must be a cycle augmentation. The former is an invalid augmentation of course. In a valid cycle augmentation, we have either $m(p) = q$ and $m(q) = p$, a *hexagonal* cycle augmentation, or $k = q$ and $l = p$, which is a *square* cycle augmentation. Like before, we deal with each type of augmentation separately.

We recognize that a 3-augmentation (i,j,k,l,p,q) is a center unmatched edge (i,j) and two 1-augmentations with start vertices k and l. We refer to these 1-augmentations as the k-arm and l-arm, respectively. A basic 3-augmentation is shown in fig. 3.4. Naively, its gain would be $g(i,j,k,l,p,q) = g(k,p)+g(l,q)+w(i,j)$. For path augmentations this is indeed the case, and as such we will structure our method for finding path 3-augmentations around combining the best 1-augmentations around

Algorithm 7 `SearchAugmentations(A, M, 2)`: Searching for 2-augmentations

Input: Adjacency matrix A, matching M, vectors $\mathbf{m}, \mathbf{m}_w, \mathbf{m}_U, \mathbf{m}_L$, a full vector $\mathbf{z}$ of explicit values 1

Output: Augmentation matrix $\mathcal{A}$ with explicit values indicating a set of 2-augmentations suitable for improving the matching

1: $G_2^c, D_2^c = \texttt{SearchCycle2Augs}(A, M, \mathbf{m}, \mathbf{m}_w, \mathbf{m}_U, \mathbf{m}_L)$ ▷ $O(n+m)$
2: $G_2^p, D_2^p = \texttt{SearchPath2Augs}(A, M, \mathbf{m}, \mathbf{m}_w, \mathbf{m}_U, \mathbf{m}_L)$ ▷ $O(n+m)$
3: $G_2 = \texttt{eAdd}(G_2^c, G_2^p, \mathsf{max})$ ▷ Element-wise add; $O(n)$
4: **if** $G_2 \neq \mathcal{M}_\emptyset$ **then** ▷ $O(1)$
5: $\mathbf{a}_g = \texttt{reduce}(G_2)$ ▷ Reduction; $O(n)$
6: $C = (\mathbf{z} \ +.\times \ \mathbf{z}^T)\langle G_2^c \rangle$ ▷ Masked m×m-multiplication; $O(n)$
7: $C = C\langle \overline{G_2^p} \rangle + \texttt{eMult}(G_2^c, G_2^p, >)$ ▷ Element-wise multiplication, $O(n)$
8: $C = \texttt{select}(C, \texttt{nonzeros})$ ▷ Selection; $O(n)$
9: $\mathbf{c} = \texttt{reduce}(C)$ ▷ Reduction; $O(n)$
10: $D_2 = \texttt{selectRows}(D_2^c, \mathbf{c}) + \texttt{selectRows}(D_2^p, \overline{\mathbf{c}})$ ▷ E-wise add, row/col selection; $O(n)$
11: $\Gamma_{mask} = \texttt{diag}(\mathbf{m}) + M + D_2 + \texttt{permuteRows}(D_2, P^M)$ ▷ E-wise add, permutation; $O(n)$
12: $\Gamma = (\mathbf{a}_g \ +.\mathsf{first} \ \mathbf{z}^T)\langle \Gamma_{mask} \rangle$ ▷ Masked m×m-multiplication; $O(n)$
13: $\Gamma = \texttt{maxPerColumn}(\Gamma)$ ▷ Max per column algorithm; $O(n)$
14: $\mathbf{v} = \Gamma \ +.\mathsf{second} \ \mathbf{z}$ ▷ m×v-multiplication; $O(n)$
15: $\mathbf{v} = \texttt{select}(\mathbf{v}, [=4])$ ▷ Selection; $O(n)$
16: $\mathbf{v} = \mathbf{v} + \texttt{permuteVector}(\mathbf{v}, P^M)$ ▷ E-wise add, permutation; $O(n)$
17: $\mathcal{A} = \texttt{selectRows}(D_2, \mathbf{v})$ ▷ Row/Column selection; $O(n)$
18: $\mathcal{A} = \mathcal{A} + \mathcal{A}^T$ ▷ E-wise add, transposition; $O(n)$
19: **else**
20: $\mathcal{A} = \mathcal{M}_\emptyset$ ▷ $O(1)$

the center, in an algorithm based on the method by Duan and Pettie [6]. The challenge lies in finding the best k- and l-arm whose vertices are pairwise distinct in order to obtain a valid augmentation while preserving the linear runtime of the algorithm. As it turns out, we can find the best path 3-augmentation in constant time per center for an $O(m)$ number of centers. This method is described in section 3.4.1.

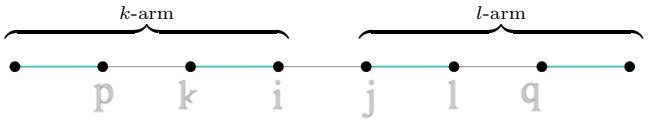

Figure 3.4: A basic 3-augmentation.

Square cycle augmentations are proper 2-augmentations in practice, and we choose to let the 2-augmentation algorithm deal with these and completely ignore them here. This choice requires us to pay attention in structuring of the full algorithm to solve this correctly.

Searching the optimal hexagonal cycle augmentation per center poses a serious obstacle. In a similar issue as before, a naive calculation of the gain will return a value that is too low, and it would need to be corrected by adding $w(p, q)$. We have very little information a priori on the effect of this correction on the gain. The following example shows that we cannot create an algorithm that considers all cycle augmentations in $O(n + m)$ time. Consider the complete graph K_n on n points with a perfect matching. In fig. 3.5 the graph is shown with its vertices split in two groups of $n/2$ points, both arranged in circles stacked on top of each other. The pairs of vertices that are aligned vertically are matched. We have $\frac{n(n-2)}{2}$ possible centers for cycle augmentations, namely all unmatched edges. If we choose one of these edges, then we have the vertices i and j of the 3-augmentation, and k and l as well as they are the matches of i and j. To complete the cycle, we need to choose p and q as the endpoints of the k- and l-arms and all of the $n - 2$ remaining vertices are an option. To find the best cycle augmentation, each k-arm and l-arm must be combined, while correcting for $w(p, q)$, to calculate the gain. Choosing this combination means choosing one of the $n - 2$ remaining matched edges in the graph, since p and q are matched. This means any method that tries to search the best hexagonal cycle augmentation for all $\Theta(n^2)$ centers will have at least a $\Theta(n^3)$ runtime, even when taking symmetries into account, and thus a linear runtime is not feasible.

This fact is also recognized by Duan and Pettie [6], who show that we do not need to consider all hexagonal cycle augmentations to obtain a set of positive-gain augmentations that will help us improve the matching to a 3/4-approximation of the maximum weight matching. Crucial is the theorem from Pettie and Sanders [14], formulated below as in [6].

Theorem 3.4.1 (See [6] and [14])**.** *Let M be a matching in a graph and $M^\star$ be a maximum weight matching. Let $g_k(a) = \frac{k}{k+1} w(a \setminus M) - w(a \cap M)$ if a is a k-augmenting cycle and $g_k(a) = g(a)$ if a is a k-augmenting path. There is a set A_k of vertex disjoint k-augmentations such that*

$$\sum_{a \in A_k} g_k(a) \geq \frac{k+1}{2k+1} \left(\frac{k}{k+1} w(M^\star) - w(M)\right).$$

This can be interpreted as follows. Giving each augmentation an alternative gain value where cycles are slightly *devalued*, there still exists a set of vertex disjoint k-augmentations where the sum

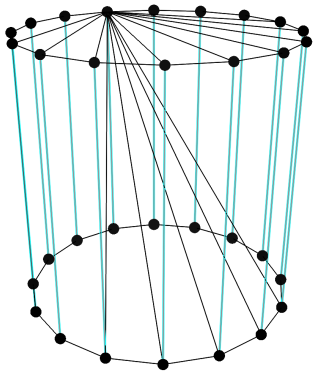

Figure 3.5: The complete graph K_n shown as K_{32}. Not all edges are drawn for clarity.

of these alternative gain values will improve the matching sufficiently. Duan and Pettie show that their method finds a set of k-augmentations $A_k^\star$ whose true gain is at least as high as that sum, so

$$\sum_{a \in A_k^\star} g(a) \geq \sum_{a \in A_k} g_k(a),$$

and as such improves our matching sufficiently as well.

In the algorithm, which does have an $O(n+m)$ runtime, only path augmentations are calculated and cycle augmentation are ignored unless we happen to find one by accident. To emphasize this essential fact, it means we are still able to achieve the desired 3/4-approximation by considering a set of augmentations that does not necessarily contain all positive-gain cycle augmentations, in $O(n+m)$ time. Our algorithm based on [6] for calculating path augmentations but adapted to the GraphBLAS standard is given in the next section.

3.4.1 Finding path 3-augmentations

In this section we give an algorithm for finding path 3-augmentations based on the algorithm by Duan and Pettie [6] for computing a $(3/4 - \epsilon)$-approximate maximum weight matching, adapted to the use of the GraphBLAS standard. We will first summarise the relevant part of the algorithm and then provide a method for implementing the algorithm in terms of matrix operations.

Basic Algorithm

The aim is to find the proper path 3-augmentation with the highest gain per possible center and create a set of positive-gain vertex disjoint augmentations from these by removing any conflicts. This requires the following four steps.

1. We search all possible centers for 3-augmentations (i, j), which are all unmatched edges where both endpoints are matched vertices. These centers also determine the vertices k and l. The $O(m)$ centers should be found in $O(m)$ time.

2. For computing the best 3-augmentations, we require a list of the four best arms per vertex k and l of each center. The creation of each of these lists should have a runtime in the order of

the degree of the corresponding start vertex. Given an average degree of d in the entire graph, this step should have a total runtime of $O(nd) = O(m)$.

3. Per center, we need to combine an arm from each list, so we choose the best k- and l-arm that are vertex disjoint and have the highest combined gain. This should be done in $O(1)$ time per center. We are certain to obtain a valid 3-augmentation when only considering the top four k/l-arms since at most three k-arms can conflict with $(i, j) \cup (j, l, q, m(q))$ for some l-arm, namely when $k = j, k = l$ or $k = q$. Any other k-arm must be valid. The analogous statement for l-arms holds as well. As such, comparing the top four k/l-arms results in the highest-gain path 3-augmentation. If we were to use the fifth best k-arm and some l-arm, a better option that is also valid with the l-arm is found in the top four, leading to a higher total gain. Note that this statement is certainly not true for cycle augmentations. However, combining k- and l-arms from the top four best arms can result in a hexagonal cycle 3-augmentation by accident. The calculated gain will be too low, so if the augmentation gets selected we allow it and the matching might be improved even more. Square cycle 3-augmentations are considered invalid by this algorithm, which is acceptable because these will be found by the 2-augmentation algorithm.

4. We take only the positive-gain augmentations and remove conflicts similar to the method for 2-augmentations. This should take $O(m)$ time.

We find a total runtime of $O(m)$, but we will accept an $O(n + m)$ runtime due to the nature of the GraphBLAS implementation of matrix operations.

Implementation

We will provide a GraphBLAS implementation for each of the steps mentioned in the previous section. As before, we work towards a matrix G_3 that contains at (i, j) the gain of the best path 3-augmentation for the corresponding center. After solving conflicts within this set of augmentations, the result should again be a matrix $\mathcal{A}$ with explicit values indicating the edges that need to be added to the matching, which for each 3-augmentation are the edges (i, j), (k, p) and (l, q).

Since we need to know the vertices i, j, p and q – other vertices are deduced from the current matching – and the gain per augmentation, we have a lot information to keep track of. The GraphBLAS standard allows for the use of matrices containing elements of a user-defined type, and the algorithm below will make use of tuples (v, x_1, x_2) where v is a floating-point value and x_1 and x_2 are both integers. We can work with these tuples by defining semirings and operators that work specifically for these elements. We will denote these operators with a †-symbol. For example, we define the binary $\mathsf{max}^\dagger$-operator working on two tuples t_1 and t_2 as

$$t_1 \ \mathsf{max}^\dagger \ t_2 = \begin{cases} t_1 & \text{if } t_1.v > t_2.v \\ t_2 & \text{otherwise.} \end{cases}$$

A pseudocode listing is found in algorithm 8.

1. Finding all centers (i, j) is a simple step of calculating $[A \setminus M]$ and selecting the rows and columns indicated by $\mathbf{m}$, thus obtaining all unmatched edges where both endpoints are matched. All steps are done in $O(n+m)$ time. The result $[A \setminus M]^\star$ is symmetric, and contains each center twice. The following steps would preserve that symmetry and we would do the necessary work twice. Therefore, we can use the upper triangular part $[A \setminus M]^\star_U$ as indication of the centers without problems.

2. To create the lists of four best arms per center endpoint, we require matrix G_1 from our 1-augmentation algorithm to provide all necessary data. We select only the rows corresponding to vertices that are matched, since the start vertex of a k/l-arm should be matched. Selecting the best arms per augmentation comes down to selecting the best 1-augmentation from G_1 per start vertex, which we can do by using the `maxPerRow` algorithm. The precise steps are omitted here, but it is a straightforward combination of steps that we have seen before. The results are matrices K_{x_1}, L_{x_1} and vectors $\mathbf{k}_{x_1}$ and $\mathbf{l}_{x_1}$ for $x_1 = 1, 2, 3, 4$. These two vectors will have values of our user-defined type. The explicit elements are the following for each k-arm in the top four per augmentation center: $K_{x_1}(i,i), K_{x_1}(i,k)$ and $K_{x_1}(i,p_{x_1})$ explicit and $\mathbf{k}_{x_1}(i) = (g(i,k,p_{x_1},m(p_{x_1})),p_{x_1},0)$. For each l-arm in the top four, we have: $L_{x_1}(j,j), L_{x_1}(j,l)$ and $L_{x_1}(j,q_{x_1})$ explicit and $\mathbf{l}_{x_1}(j) = (g(j,l,q_{x_1},m(q_{x_1})),0,q_{x_1})$. Now K_{x_1} and L_{x_1} contain the vertices of each x_1th best k/l-arm as explicit values and will be used to test for conflicts between the arms. Vectors $\mathbf{k}_{x_1}$ and $\mathbf{l}_{x_1}$ contain the gain and p and q vertices per k/l-arm and will be used to combine and create the path 3-augmentation. All steps are done in $O(n+m)$ time.

3. Per center, we need to combine each k-arm in the top four with each l-arm in the top four, which we do in a double for-loop, iterating over x_1 and x_2, with $x_1, x_2 = 1, 2, 3, 4$.

   This step revolves around two matrices, F and $C^\star$. Matrix $C^\star$ functions as a mask, it indicates all locations where we will combine a k- and l-arm. Combining arms happens in two steps. In the first step we *try* to combine two arms, and see at which locations a conflict occurs. In the second step we actually combine the arms at all locations where we now know a conflict will not occur.

   At the beginning of each x_1-iteration, this is done at all possible centers, so we set $C^\star = [A \setminus M]^\star_U$. In the first x_2-iteration, we will try to combine the best k- and l-arm for each center. In some cases this will lead to an invalid augmentation, which is where matrix C comes in. By matrix multiplication $C = (K_{x_1} \ \mathsf{any.pair} \ L^T_{x_2})\langle C^\star\rangle$, we obtain an explicit value in matrix C at (i,j) if any of the vertices in the k- and l-arm overlap, meaning they are incompatible. These found centers are not suitable in this iteration, so we take them away by masking $C^\star$ with the complement of C.

   The steps above represented *trying* to combine arms. Now that we know where we actually can combine them – at each center saved in $C^\star$ – we will do so and save the result in matrix F. Combining the arms is done by multiplication $F = (\mathbf{k}_{x_1} +^\dagger.+^\dagger \mathbf{l}^T_{x_2})\langle C^\star\rangle$, such that

   $$F(i,j) = (g(i,k,p_{x_1},m(p_{x_1}) + g(j,l,q_{x_2},m(q_{x_2}))),p_{x_1},q_{x_2}).$$

   In the next x_2-iteration, we will try to combine arms at exactly those locations where a conflict occurred in the current iteration, so we set $C^\star := C$. Note the interplay between $C^\star$ and C each iteration. In each iteration, $C^\star$ denotes the locations where we *can* combine two arms without conflicts, and C denotes the locations where we *cannot*. These locations where we cannot are exactly the centers where we will try a new combination in the next iteration.

   In the new x_2 iteration, we want to save the new combinations in F. It still contains explicit values from the previous iteration, and the new values occur at different locations in the matrix, so we can set $F := F + (\mathbf{k}_{x_1} +^\dagger.+^\dagger \mathbf{l}^T_{x_2})\langle C^\star\rangle$ without any issues. When we have performed the steps for $x_2 = 4$, we save the result of F in G_3 by $G_3 = G_3 \ (\oplus, \mathsf{max}^\dagger) \ F$, keeping always the current best combination of k- and l- arms in G_3.

4. To find the correct gain values per augmentation, we need to add the weight of the center edge to the combined gain of the arms, so we add $[A \setminus M]^\star_U$ to G_3. We select only the positive-gain

augmentations from G_3, or return an empty matrix if none have been found. Now we add the transpose of G_3 to itself, since we started the algorithm with only the upper triangular part of $[A \setminus M]^\star$. Since G_3 now has $O(n)$ explicit values, this operation is cheap in terms of runtime.

We can solve any conflicts between the found augmentations, by first recognizing that for any centers (i, j_1) and (i, j_2) only one of the augmentations can be applied because of the shared vertex. From G_3 we select the element in each row with the highest gain value by using the standard method but using operators adapted to dealing with the tuple elements. Next, we implement the same method for solving conflicts in the 2-augmentation algorithm by setting up a matrix Γ, here with six explicit values per row, namely $(i,i),(i,j),(i,k),(i,l),(i,p),(i,q)$, all containing the gain of the single 3-augmentation saved in row i of G_3. We select the highest value per column of Γ and any row that remains with six explicit values represents an augmentation that has passed the selection procedure. We select these rows from G_3 and then create matrix D_3 with three explicit values of 1 for each remaining tuple in G_3. These locations are $(i,j),(k,p)$ and (l,q). By adding to D_3 its own transpose, we have the desired result $\mathcal{A}$ that can be passed on to the flipping procedure. All steps are done in $O(n+m)$ time.

Algorithm 8 `SearchAugmentations(A, M, 3)`: Searching for 3-augmentations

Input: Adjacency matrix A, matching M, vector $\mathbf{m}$, a full vector $\mathbf{z}$ of explicit values 1

Output: Augmentation matrix $\mathcal{A}$ with explicit values indicating a set of 3-augmentations suitable for improving the matching

$[A \setminus M] = \texttt{select}(A - M, \texttt{nonzeros})$ ▷ Selection, e-wise add; $O(n+m)$
$[A \setminus M]^\star = \texttt{selectRows}([A \setminus M], \mathbf{m})$ ▷ Row/Column selection; $O(n+m)$
$[A \setminus M]^\star = \texttt{selectColums}([A \setminus M]^\star, \mathbf{m})$ ▷ Row/Column selection; $O(n+m)$
$G_1 = \texttt{SearchAugmentations(A, M, 1)}$ ▷ $O(n+m)$

for $x_1 = 1, 2, 3, 4$ **do** ▷ $O(n+m)$
    From G_1, create:
    K_{x_1} with explicit values at $(i,i), (i,k)$ and (i, p_{x_1})
    $\mathbf{k}_{x_1}$ with values $\mathbf{k}_{x_1}(i) = (g(i, k, p_{x_1}, m(p_{x_1})), p_{x_1}, 0)$
    L_{x_1} with explicit values at $(j,j), (j,l)$ and (j, q_{x_1})
    $\mathbf{l}_{x_1}$ with values $\mathbf{l}_{x_1}(j) = (g(j, l, q_{x_1}, m(q_{x_1})), 0, q_{x_1})$
end for

$G_3 = \mathcal{M}_\emptyset$
for $x_1 = 1, 2, 3, 4$ **do** ▷ $O(n+m)$
    $F = \mathcal{M}_\emptyset$
    $C^\star = [A \setminus M]^\star_U$
    for $x_2 = 1, 2, 3, 4$ **do**
        $C = (K_{x_1}\ \mathsf{any.pair}\ L^T_{x_2})\langle C^\star \rangle$ ▷ Masked m×m-multiplication; $O(n+m)$
        $C^\star = C^\star \langle \overline{C} \rangle$ ▷ Masking; $O(n+m)$
        $F = F +^\dagger (\mathbf{k}_{x_1} +^\dagger .+^\dagger \mathbf{l}^T_{x_2})\langle C^\star \rangle$ ▷ Masked m×m-mult, E-wise mult; $O(n+m)$
        $C^\star = C$
    end for
    $G_3 = \texttt{eAdd}(G_3, F, \mathsf{max}^\dagger)$ ▷ Element-wise add; $O(n+m)$
end for

$G_3 = \texttt{eAdd}(G_3, [A \setminus M]^\star_U, +^\dagger)$ ▷ Element-wise add; $O(n+m)$
$G_3 = \texttt{select}^\dagger(G_3, \texttt{positive})$ ▷ Selection; $O(n+m)$

if $G_3 \neq \mathcal{M}_\emptyset$ **then** ▷ $O(1)$
    $G_3 = G_3 +^\dagger G_3^T$ ▷ Element-wise addition; $O(n)$
    $G_3 = \texttt{maxPerRow}^\dagger(G_3)$ ▷ Max per row algorithm; $O(n+m)$
    Create matrix Γ where $(i,i), (i,j), (i,k), (i,l), (i,p), (i,q)$ contain gain per augmentation starting at i
    $\Gamma = \texttt{maxPerColumn}(\Gamma)$ ▷ Max per column algorithm; $O(n)$
    $\mathbf{v} = \Gamma +.\mathsf{second}\ \mathbf{z}$ ▷ m×v-multiplication; $O(n)$
    $\mathbf{v} = \texttt{select}(\mathbf{v}, [=6])$ ▷ Selection; $O(n)$
    Create matrix D_3 with 1 at $(i,j), (k,p), (l,q)$ in rows of $\mathbf{v}$ ▷ $O(n)$
    $\mathcal{A} = D_3 + D_3^T$ ▷ Element-wise add; $O(n)$
else
    $\mathcal{A} = \mathcal{M}_\emptyset$ ▷ $O(1)$

Chapter 4

Experiments

In this chapter we perform several experiments using an implementation of the algorithm described in the previous chapter. This C implementation can be found at `https://github.com/DavidDieudeBest/ApproximatingMWMGraphBLAS`, and makes use of SuiteSparse:GraphBLAS [3]. Instructions for use of the code and program can be found in the *readme* file attached to the code and in appendix A. All tests are performed on a MacBook Pro (2020) with 8 GB 2133 MHz LPDDR3 RAM memory and a 1.4 GHz Quad-Core Intel Core i5 processor. The graphs used for these experiments are obtained through the SuiteSparse Matrix Collection [5]. We load these graphs as Matrix Market data (`.mtx`) using the algorithm from the LAGraph library [12], a collection of algorithms that use GraphBLAS. The graphs are listed in table 4.1. All are undirected graphs without double edges. The *human_gene*1 and *mouse_gene* graphs contain self loops, which we remove in a preprocessing step before calculating any matching.

Graph name	n	m	Matching weight upper bound
Binaryalphadigs_10NN	1 404	9 696	91.5
G22	2 000	19 990	1 000
MISKnowledgeMap	2 427	28 511	965.9
cond-mat	16 726	47 594	9 968.5
hi2010	25 016	62 063	1 103 986 256.5
har_10NN	10 299	75 868	488.2
astro-ph	16 706	121 251	8 261.2
sd2010	88 360	205 361	7 681 459 041.5
fl2010	484 481	1 173 147	15 108 221 779.5
tx2010	914 231	2 228 136	39 547 303 682.5
human_gene1	22 283	12 323 680	1 121
mouse_gene	45 101	14 461 095	1 553.4

Table 4.1: A selection of graphs from the SuiteSparse Matrix Collection [5] that we use in experiments for our algorithm, sorted by number of edges in the graph.

We restate that the main structure of our algorithm consists of a loop where each iteration consists of searching proper k-augmentations, flipping the found augmentations and determining a new k value. This is shown in algorithm 9. Integer k determines what type of augmentations are searched in an iteration, so choosing the value for k can come to involve some strategy. To ensure

the 3/4-approximation result, all strategies must end with three iterations, for $k = 1, 2, 3$, where method `SearchAugmentations` does not find any positive-gain augmentations.

A straightforward strategy might be to search 1-augmentations until no positive-gain augmentations are found, then switch to proper 2-augmentations. When no positive-gain proper 2-augmentations are found, we check if new 1-augmentations have become available. If neither 1- or proper 2-augmentations can be found with positive gain, we switch to proper 3-augmentations, and finish when no positive-gain augmentations for $k = 1, 2, 3$ are left.

We show results for different strategies in section 4.3. First we look at the quality of obtained approximations in section 4.1 and the runtimes of each of the methods in section 4.2. There we will also discuss the runtime improvements when using the parallelism provided by SuiteSparse:GraphBLAS.

Algorithm 9 MWM approximation algorithm - main loop

Input: Adjacency matrix A of a weighted undirected graph $G(V, E)$, an iteration strategy `STRAT`

Output: Matrix M reflecting a 3/4-approximation of the MWM of G

1: $M = \mathcal{M}_\emptyset$
2: $k = 1$
3: **while** any positive-gain k-augmentation exists for $k = 1, 2, 3$ **do**
4: $\quad \mathcal{A} =$ `SearchAugmentations`(A, M, k)
5: $\quad M =$ `FlipAugmentations`$(A, M, \mathcal{A})$
6: $\quad k =$ `SelectStrategy`$(\texttt{STRAT}, k)$
7: $\quad$ **end while**

4.1 Quality

We will test the quality of maximum weight matching (MWM) approximations obtained through the algorithm to see if we can indeed get 3/4-approximations of the true MWM. However, calculating the MWM for larger graphs in the data set is not feasible in a reasonable amount of time using even the optimal $O(nm + n^2 \log n)$ algorithm by Gabow [7]. Alternatively, we can quite easily find an upper bound for the weight of the MWM using lemma 4.1.1, as described in [1].

Lemma 4.1.1 (Upper bound on matching weight [1])**.** *For any graph $G(V, E)$, let $M^\star$ be a matching for G with maximum weight, then an upper bound on the matching weight is given as*

$$w(M^\star) \leq \frac{1}{2} \sum_{v \in V} \max\{w(u, v) \mid (u, v) \in E\}.$$

Intuitively, we can see that this bound holds by viewing the edge set as a set of half-edges, dividing each edge into two half-edges, either connecting to one of the endpoints. Now each vertex contributes the weight of its heaviest connected half-edge to the total weight, which is at most half of the weight of the heaviest edge connected to that vertex. In table 4.1, this upper bound on the weight is given for each of the graphs in the used data set.

When only performing iterations for searching 1-augmentations, we will theoretically obtain a 1/2-approximation of the MWM. Results for searching for 1-augmentations until no positive-gain

augmentations are left are shown in fig. 4.1. For each graph, we show the calculated weight as a fraction of the upper bound of the MWM as given by lemma 4.1.1. We see that we obtain an approximation that is far better than a 1/2-approximation for each network, and even often find a 3/4-approximation. In truth the approximations might even be better since we used an upper bound, and therefore these values reflect a lower bound on the approximation.

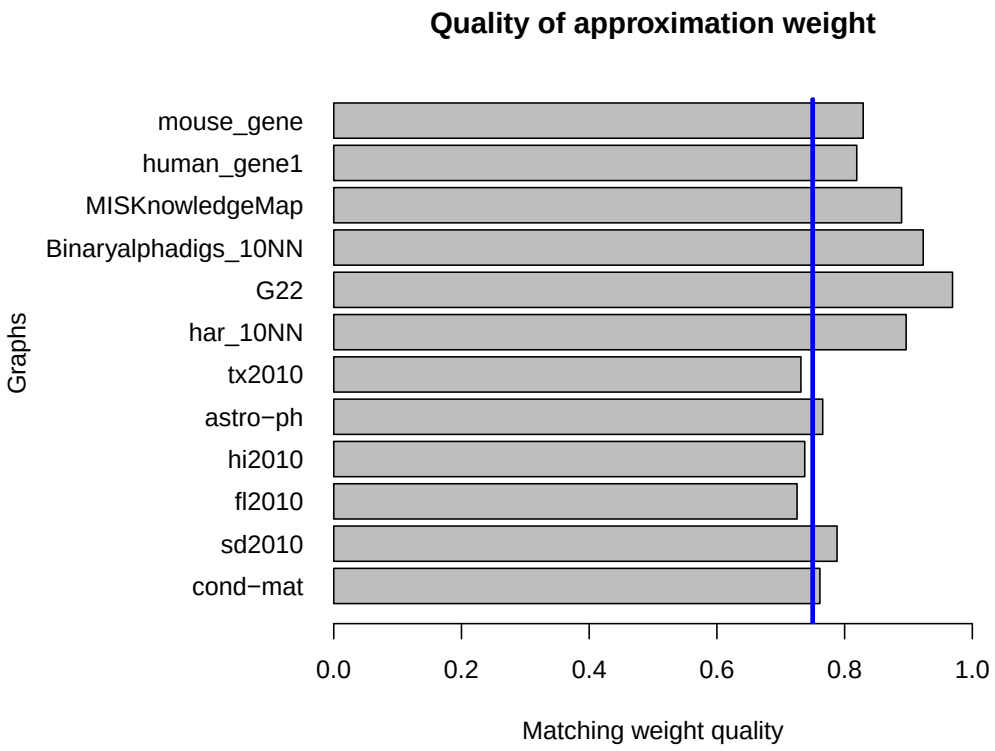

Figure 4.1: The weight of found MWM approximations for each graph as a fraction of the MWM upper bound as in table 4.1. The MWM approximations are calculated using only 1-augmentations. Bars passing the blue line indicate at least a 3/4-approximation of the true MWM.

For a better comparison between obtained approximations, we can use the *gap-to-optimality* metric. A matching M has a gap-to-optimality value of

$$\frac{w(M^\star) - w(M)}{w(M^\star)},$$

where $M^\star$ is a matching with maximum weight.

Using the strategy described in the previous section, we can run the algorithm three times. Once for searching only 1-augmentations, once for 1- and 2-augmentations, and once for all types of augmentations. Results on the gap-to-optimality to the upper bound of its MWM of obtained approximations are shown in fig. 4.2. We see that adding 2-augmentations improves the gap-to-optimality somewhat, and then also adding 3-augmentations gives a much smaller improvement. For some graphs, like $hi2010$ and $fl2010$, adding 2- and 3-augmentations does allow us to guarantee a 3/4-approximation. For most other graphs this was already obtained using only 1-augmentations.

This might also have been the case for $hi2010$ and $fl2010$, but we cannot be certain since we only have the upper bound of the MWM and not the true MWM.

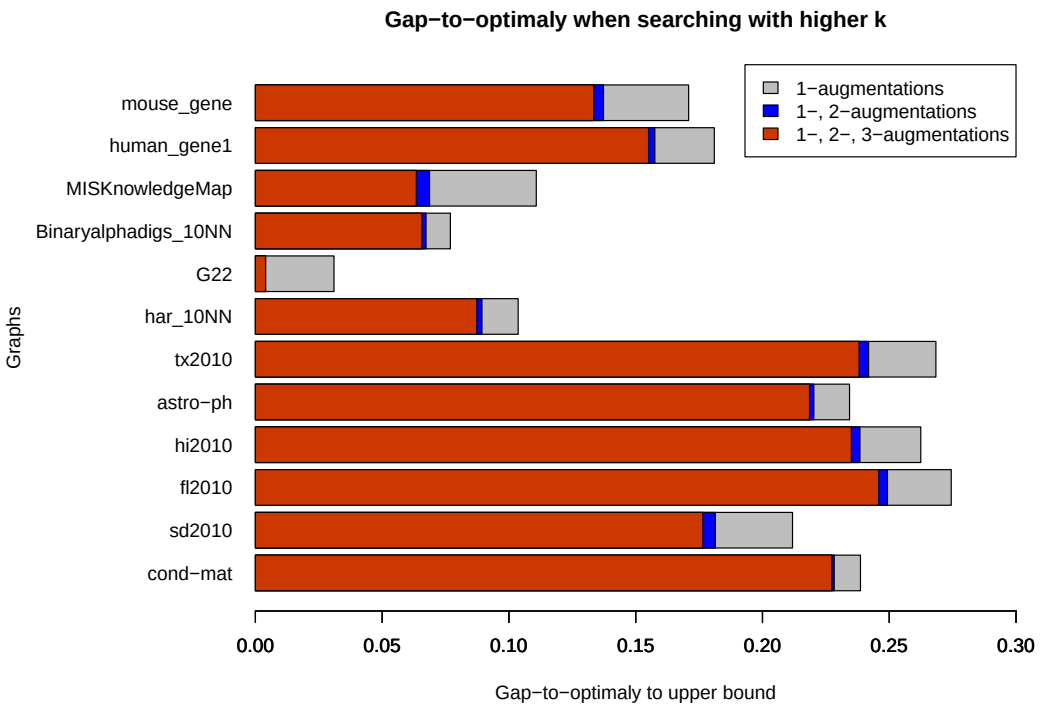

Figure 4.2: The gap-to-optimality to the MWM upper bound of found approximations for each graph when searching for only 1-augmentations, both 1- and 2-augmentations, and 1-, 2- and 3-augmentations.

4.2 Runtime

We can analyse the runtimes of different iterations – referring to either searching for 1-, 2- or 3-augmentations or flipping augmentations – to get some insight into useful strategies for the order of execution of iterations. In table 4.2, we show the mean runtime of each type of iteration for each graph. Matrix $G22$ turned out not have any possible centers for 3-augmentations.

Graph name	1-Aug Search	2-Aug Search	3-Aug Search	Flipping
Binaryalphadigs_10NN	0.001	0.006	0.011	0.0002
G22	0.002	0.013	Not performed	0.0002
MISKnowledgeMap	0.002	0.015	0.022	0.0003
cond-mat	0.007	0.068	0.170	0.0018
hi2010	0.010	0.084	0.133	0.0027
har_10NN	0.008	0.060	0.080	0.0010
astro-ph	0.010	0.099	0.133	0.0027
sd2010	0.034	0.304	0.451	0.0080
fl2010	0.168	1.772	2.698	0.0492
tx2010	0.349	3.515	5.527	0.0974
human_gene1	1.486	9.083	10.447	0.0024
mouse_gene	1.699	10.650	11.579	0.0050

Table 4.2: Mean runtime in seconds per iteration type for each graph.

We see that flipping augmentations is the fastest action, and that searching 2- and 3-augmentations is notably more expensive in terms of runtime than searching 1-augmentations. These results are to be expected from the algorithm theory, since searching 2- and 3-augmentations requires more steps in the algorithm in comparison to the other methods. We can conclude that a strategy for obtaining fast runtimes, would perform as few 2- and 3-augmentation searches as possible, even if it means replacing these types of iterations with multiple 1-augmentation searches and flipping iterations, since these are relatively cheap. The values from table 4.2 can be plotted for each graph against the size of the graph as given by m. Noting that for each graph we have $m > n$, we would like to see that these runtimes indeed indicate a runtime of each iteration that is linear in terms of the size of the graph, as concluded in the theoretic analysis. The fitting of a least squares regression line on the log values of each set of data points indeed results in a line with a slope of 1.048, 1.035 and 0.970 for searching 1-, 2- and 3-augmentations, respectively. The error bounds fall between 0.04 and 0.07, further confirming the theoretical linear runtime. Since the flipping algorithm has an $O(n)$ runtime and the *human_gene*1 and *mouse_gene* graphs have a small number of vertices, their mean runtimes for flipping are relatively small in comparison to the other larger graphs.

The runtimes mentioned so far have all been obtained by single-threaded execution of the program. Using SuiteSparse:GraphBLAS with OpenMP allows us to use its internal parallelism without any extra work in programming. Executing the algorithm with more processors results in different runtimes, shown in fig. 4.4. Using more processors will involve extra work that needs to be performed, and there will often not be an even distribution of work among the processing units. We see that the smallest graphs do not have any faster runtimes when using more processors, likely due to the extra work outweighing the gain in processing power. This data set indicates that using more processors becomes more beneficial when dealing with larger graphs. We see that for the largest graphs runtimes can become more than twice as fast with computation on four processors.

We note that the MacBook Pro used for these experiments has only four physical cores and

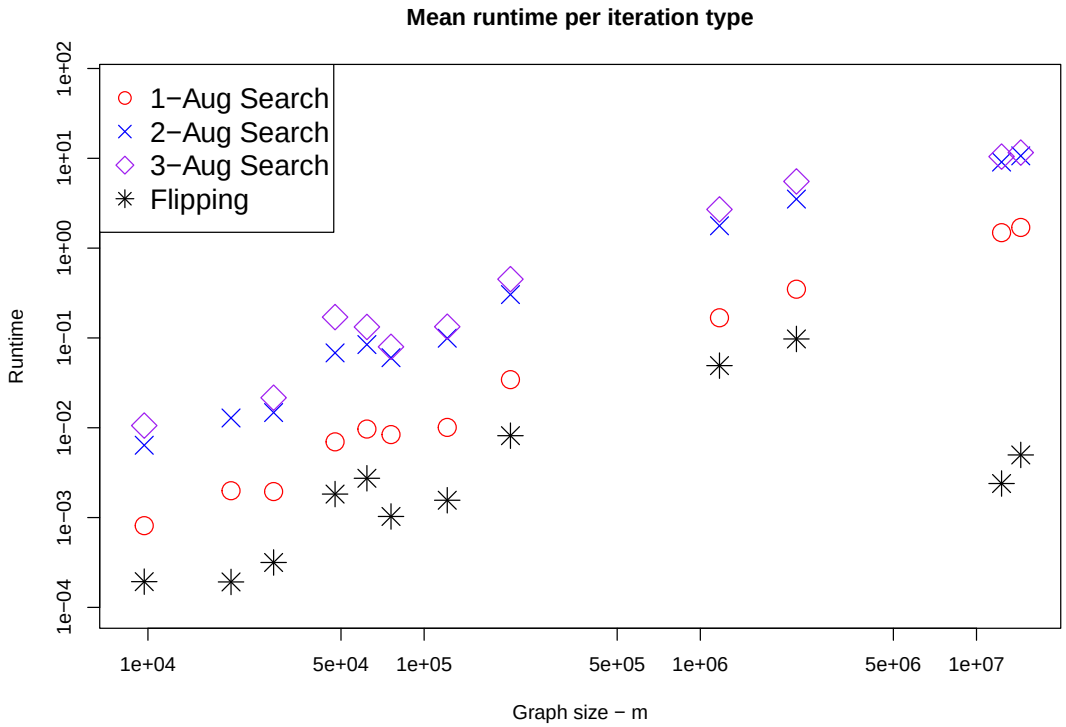

Figure 4.3: Mean runtime in seconds per iteration type for each graph against its size in terms of m, plotted on log-log scale from the values in table 4.2.

achieves eight parallel threads via hyperthreading. We can therefore not expect to obtain the results that compare to using eight physical cores. Moreover, we see that using eight threads does not result in any runtime improvement in almost all cases in this data set.

Since 2- and 3-augmentation searches are quite expensive in terms of runtime, we might want to improve the number of augmentations that can be applied each iteration after solving conflicts. In table 4.3 we show the mean value of the percentage of positive-gain k-augmentations that are applied after solving conflicts. We can see that for graphs with a high average degree like *human_gene*1 and *mouse_gene* only a small percentage of found augmentations get applied, which is to be expected for such denser graphs. Coincidentally, the *human_gene*1 and *mouse_gene* graphs take the most runtime in our dataset because of their size, and if we would want to improve here, we might be able to achieve the most by revising the conflict solving in each algorithm.

4.3 Strategy

In this section, we will compare the basic strategy (`BASIC`) discussed at the start of this section to some other options. Using `BASIC`, we obtained the runtimes for different graphs as given in section 4.2. The 2- and 3-augmentation searching algorithms require a lot more runtime than the 1-augmentation algorithm. This fact, together with flipping iterations being very cheap, encourages the use of more 1-augmentation iterations if it saves us using the other iterations. The strategy

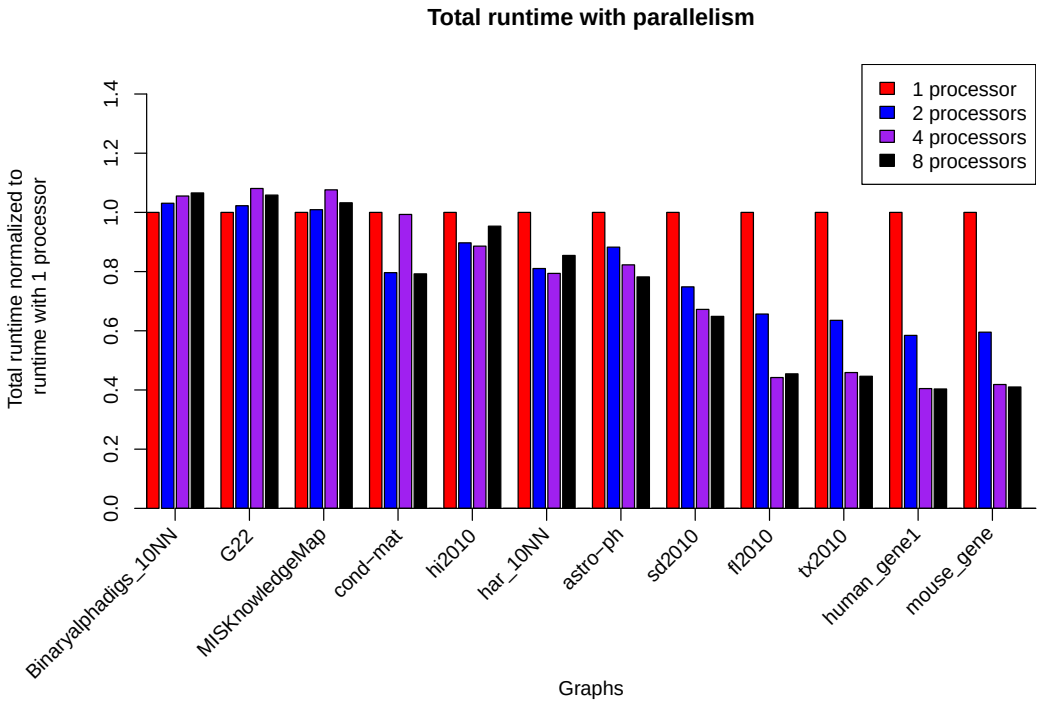

Figure 4.4: Total runtime of the algorithm per graph using a varying number of processors, normalized to the runtime using one processor. The graphs are sorted by increasing number of edges.

Graph name	1-Aug Search	2-Aug Search	3-Aug Search
Binaryalphadigs_10NN	29.0	25.4	28.7
G22	7.9	12.7	Not performed
MISKnowledgeMap	19.2	20.0	7.3
cond-mat	30.3	39.0	42.5
hi2010	32.5	36.1	39.3
har_10NN	33.3	29.5	29.1
astro-ph	16.0	28.9	29.2
sd2010	31.2	33.2	40.0
fl2010	26.3	38.7	39.8
tx2010	29.3	36.5	39.4
human_gene1	6.0	13.7	12.7
mouse_gene	8.9	15.5	10.2

Table 4.3: Mean percentage of found positive-gain k-augmentations that are applied after solving conflicts per iteration type for each graph.

that always prefers 1-augmentation iterations (`ONEAUG_PREFERENCE`) starts by performing as many 1-augmentation iterations as possible, then attempts a 2-augmentation iteration and afterwards switches back to as many 1-augmentations iterations as possible. This is repeated until both 1-

and 2-augmentation iterations do not give any positive-gain augmentations. Then we attempt a 3-augmentation iteration, and switch back to as many 1-augmentations as possible. We finish when no positive-gain 1-, 2- or 3-augmentations are found. Another strategy is just alternating each type of iteration by increasing k (`ALTERNATING`). In both previously mentioned strategies, we find as many 1-augmentations as possible before moving on to larger augmentations. This creates a lot of possible centers and therefore these iterations might require a long runtime. By using this `ALTERNATING` strategy, the first iterations of 2- and 3-augmentations might not take very long since there are not as many possible centers.

Each strategy is able to produce an MWM approximation for each of the input graphs. Results of the total runtime per strategy for each graph in our dataset are presented in fig. 4.5. The graphs are listed in order of increasing size. `ONEAUG_PREFERENCE` will sometimes give an improvement in the runtime, but not always. Strategy `ALTERNATING` can sometimes be a lot slower, almost twice as slow for graph $MISKnowledgeMap$, but for larger graphs seems to give an improvement as well. We can use the normalized geometric mean values to give more insight. The geometric mean of p values $v_i > 0$, $i = 1, ..., p$, is defined as

$$m_{geom}(v_1, v_2, ..., v_p) = (v_1 \cdot v_2 \cdot ... \cdot v_p)^{\frac{1}{p}}.$$

We write the set of runtimes obtained by a strategy `STRAT` as $R_{\texttt{STRAT}}$. We can normalize the geometric mean values of the runtimes per strategy to the geometric mean for the `BASIC` strategy, resulting in

$$m_{geom}(R_{\texttt{BASIC}}) = 1.000, \quad m_{geom}(R_{\texttt{ALTERNATING}}) = 1.022, \quad m_{geom}(R_{\texttt{ONEAUG\_PREFERENCE}}) = 1.011.$$

The differences here are small, but would point to the `BASIC` strategy being slightly preferable to the other two. From these experiments, it seems not one strategy is an obvious best one based on the graphs from this dataset in terms of runtime.

The differences in quality of the results of different strategies are very minor. Shown in table 4.4 is the largest difference from the mean obtained weight approximation for each graph as a percentage. Each strategy produces a matching of very similar quality, with differences well below 1%, and therefore a user most likely does not have to take this into consideration when choosing the best strategy. If it is important to have an approximation that is as good as possible and the graph is not too large, the user could try each strategy and see what result is best.

When a user needs a MWM approximation without the guarantee of a 3/4-approximation but with a fast runtime, they can use only 1-augmentation searching and likely obtain a good approximation anyway. When we do need the guarantee, we cannot say with certainty from these tests which strategy is best for the runtime.

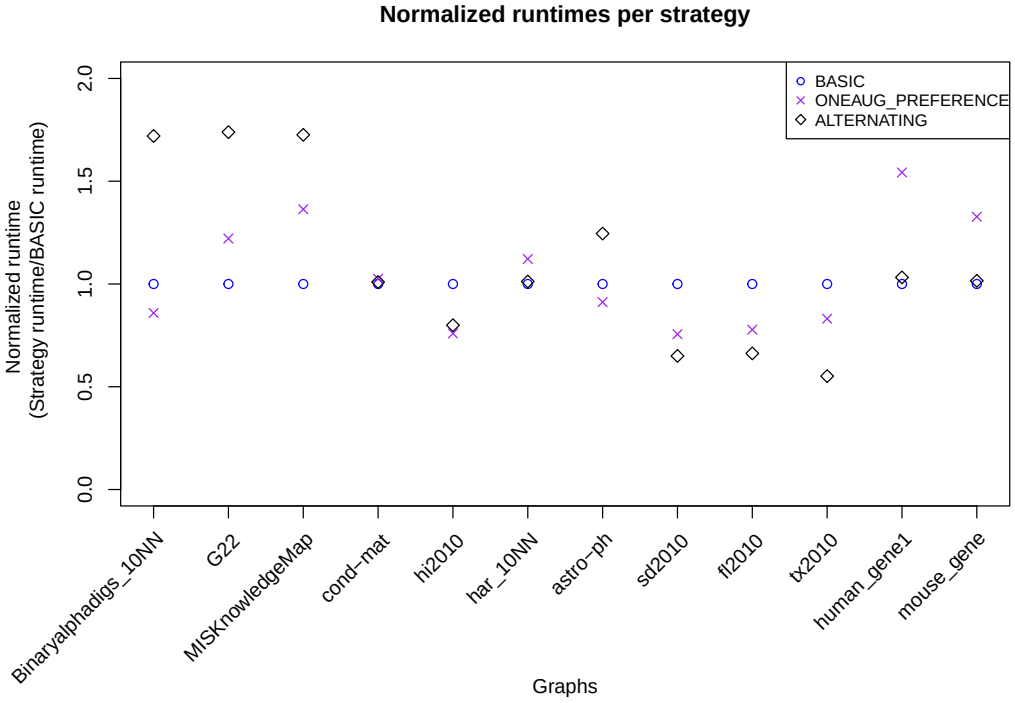

Figure 4.5: The total runtime of each strategy per graph, normalized with the runtime of the `BASIC` strategy.

Graph name	Difference in %
Binaryalphadigs_10NN	0.017
G22	0.067
MISKnowledgeMap	0.081
cond-mat	0.002
hi2010	0.015
har_10NN	0.013
astro-ph	0.008
sd2010	0.022
fl2010	0.013
tx2010	0.006
human_gene1	0.018
mouse_gene	0.022

Table 4.4: Largest difference of any strategy from the mean obtained MWM weight approximation as a percentage.

Chapter 5

Conclusions

We have proposed an approximation algorithm consisting of a series of linear-time iterations to solve the maximum weight matching problem using the GraphBLAS standard. Making use of the application of k-augmentations for up to $k = 3$, we can guarantee a 3/4-approximation of the true maximum weight matching.

Using our C implementation of the algorithm, we performed several experiments testing quality and runtime on a dataset of several large graphs. The guaranteed quality was indeed achieved for all graphs and often greatly exceeded, even when compared to an upper bound instead of the exact weight. In our implementation, each part of the algorithm follows the theoretical runtime. These results have been achieved in other algorithms tackling this problem, but because of our use of the GraphBLAS standard and its SuiteSparse:GraphBLAS C implementation, we have easy access to its internal parallelism for all matrix operations. Application of the algorithm to the larger graphs in the data set showed good improvements in runtime when using multiple processors for execution, being sometimes twice as fast when using four processors, compared to sequential execution. This improvement was smaller or not existent for the smallest graphs that were tested. We have implemented three strategies for the order of selecting algorithm iterations, but did not find a single preferable strategy in terms of runtime or quality of the result.

To conclude, we have achieved the desired approximation algorithm using the GraphBLAS standard that produces a 3/4-approximation using only linear-time iterations, adding to the larger community goal of expanding the set of graph algorithms using the GraphBLAS standard.

Chapter 6

Future work

The experiments performed in chapter 4 demonstrate the qualities of the algorithm that we seek, but they are also limited. We could get more insight into the performance of the algorithm when executing on a system with a much larger number of processors, preferably on a broader data set containing even bigger graphs, hopefully showing the advantages of using the GraphBLAS standard to a greater extent.

We currently have no guarantees or a priori indications on the number of iterations needed to achieve a certain quality. Focusing on this aspect and investigating more strategies can be important to further test the practical use of the algorithm. For improvements in its mechanisms, the methods of conflict solving could be the first section to enhance. Furthermore, an iteration currently does not take into account the work of previous iterations. This is a feature that could be added to possibly improve runtimes. When the previous iteration has not found any positive-gain augmentations in a large part of the graph and no augmentations have been flipped in its neighborhood, the next iteration will not find any in that area either, and the algorithm does not need to search for them, saving some amount of work.

Extending the algorithm to search for 4-augmentations to guarantee a better quality of the approximation might not be possible or generally useful. We have seen that adding in 3-augmentations often made only a slight difference in the obtained quality in practice, and we would expect an even smaller contribution from 4-augmentations. Additionally, due to a similar reasoning as in section 3.4, there exist too many possible centers for path augmentations and options for cycle augmentations to consider all of them in linear time. In many social networks it might not even be useful because of the six-degrees-of-separation concept, which means a 3-augmentation already might span the entire diameter of the network.

For improving the implementation of the algorithm, gains in runtime performance could be obtained by better exploiting the systems of the SuiteSparse:GraphBLAS framework. An example is the use of the different storage formats like structures designed for hypersparse matrices. The framework is still being developed further, so the algorithm might be able to make use of coming updates, like in the treatment of user-defined types which is currently still considered slow [4]. Another option would be to attempt an implementation using a different GraphBLAS library, for example using C++ [13, 16].

Taking a wider scope, future work in this area could involve developing similar graph algorithms using the GraphBLAS standard, furthering the community effort to create a large set of graph algorithms with implementations.

Chapter 7

Acknowledgements

Many people contributed greatly to the creation of this thesis and I would like to acknowledge each of those contributions here. Thank you very much to:

- Rob Bisseling, for a year of great supervision and guidance, and luckily only a single stressful video call when all work seemed to have been for nothing, although it was not.
- Ivan Kryven, for being second reader.
- Janne Looman, for proofreading this entire thesis.
- My study group *'Hok'* and *'KantNoor'*, without whom this process would certainly not have been so much fun. The games of Ligretto, summer and winter walks, not-so-serious discussions and general support have made this one of my favourite years studying mathematics.
- Jetske Kemper, for creating a master thesis that was concise, well-written and a great basis to work from.
- André van Ginkel, for spending his Friday afternoon helping me with a dysfunctional LaTeX bibliography.
- My parents, for their genuine and quite successful attempt to understand the problem and solution in this research.
- And a general thanks to the people working for Utrecht University that were involved in this process.

Bibliography

[1] Rob H. Bisseling. *Parallel Scientific Computation; A Structured Approach Using BSP*. Second edition. USA: Oxford University Press, 2020. ISBN: 9780198788348.

[2] Benjamin Brock, Aydın Buluç, Timothy Mattson, Scott McMillan, and José Moreira. *The GraphBLAS C API Specification*. `https://graphblas.org/`. Technical Report. 2021.

[3] Timothy A. Davis. "Algorithm 1000: SuiteSparse: GraphBLAS: Graph algorithms in the language of sparse linear algebra". In: *ACM Transactions on Mathematical Software (TOMS)* 45.4 (2019). DOI: 10.1145/3322125.

[4] Timothy A. Davis. *User Guide for SuiteSparse:GraphBLAS*. `https://github.com/DrTimothyAldenDavis/GraphBLAS/blob/stable/Doc/GraphBLAS_UserGuide.pdf`. Accessed: 9-12-2022.

[5] Timothy A. Davis and Yifan Hu. "The University of Florida Sparse Matrix Collection". In: *ACM Transactions on Mathematical Software 38* 38.1 (2011). DOI: 10.1145/2049662.2049663.

[6] Ran Duan and Seth Pettie. "Approximating Maximum Weight Matching in Near-Linear Time". In: *2010 IEEE 51st Annual Symposium on Foundations of Computer Science*. 2010, pp. 673–682. DOI: 10.1109/FOCS.2010.70.

[7] Harold N. Gabow. "Data Structures for Weighted Matching and Extensions to B-Matching and f-Factors". In: *ACM Trans. Algorithms* 14.3 (June 2018). DOI: 10.1145/3183369.

[8] Jetske Kemper. *Approximating the maximum weight matching problem using the GraphBLAS standard*. MSc Thesis. Utrecht University. 2022.

[9] Jeremy Kepner, Peter Aaltonen, David Bader, Aydın Buluç, Franz Franchetti, John Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, Scott McMillan, Carl Yang, John D. Owens, Marcin Zalewski, Timothy Mattson, and José Moreira. "Mathematical foundations of the GraphBLAS". In: *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. 2016, pp. 1–9. DOI: 10.1109/HPEC.2016.7761646.

[10] Jeremy Kepner and John Gilbert. *Graph Algorithms in the Language of Linear Algebra*. Ed. by Jeremy Kepner and John Gilbert. Society for Industrial and Applied Mathematics, 2011. DOI: 10.1137/1.9780898719918.

[11] Tim Mattson, David Bader, Jon Berry, Aydın Buluç, Jack Dongarra, Christos Faloutsos, John Feo, John Gilbert, Joseph Gonzalez, Bruce Hendrickson, Jeremy Kepner, Charles Leiserson, Andrew Lumsdaine, David Padua, Stephen Poole, Steve Reinhardt, Mike Stonebraker, Steve Wallach, and Andrew Yoo. "Standards for graph algorithm primitives". In: *2013 IEEE High Performance Extreme Computing Conference (HPEC)*. 2013, pp. 1–2. DOI: 10.1109/HPEC.2013.6670338.

[12] 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)*. 2019, pp. 276–284. DOI: 10.1109/IPDPSW.2019.00053.

[13] Scott McMillan, Jonathan Chu, Samantha Misurda, John Matty, Andrew Mellinger, Andrew Lumsdaine, Marcin Zalewski, Eric Holk, Peter Zhang/Aaltonen, Oren Wright, and Jason Larkin. "GraphBLAS Template Library (GBTL) 3.0". June 2020. URL: https://github.com/cmu-sei/gbtl.

[14] Seth Pettie and Peter Sanders. "A simpler linear time 2/3-e approximation for maximum weight matching". In: *Information Processing Letters* 91.6 (2004), pp. 271–276. ISSN: 0020-0190. DOI: 10.1016/j.ipl.2004.05.007.

[15] Alex Pothen, S. M. Ferdous, and Fredrik Manne. "Approximation algorithms in combinatorial scientific computing". In: *Acta Numerica* 28 (2019), pp. 541–633. DOI: 10.1017/S0962492919000035.

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

Appendix A

Code readme

```
# Maximum weight Matching Approximation Algorithm Using The GraphBLAS Standard

David D. de Best, 2023.

This C program implements a method of approximating the maximum weight matching
(MWM) for general graphs via positive-gain k-augmentations using the GraphBLAS
standard. It is the companion for the master thesis 'A 3/4-approximation of the maximum
weight matching problem using the GraphBLAS standard', David D. de Best,
supervised by prof. dr. Rob H. Bisseling, Utrecht
University, 2023.

For an in-depth explanation of the procedures implemented, please see the thesis
and provided comments in de code files.

--------------------------------------------------------------------------------
## Software Dependencies

This program relies on:
 -  The SuiteSparse:GraphBLAS package by Timothy A. Davis, which is a full
    implementation of the GraphBLAS standard, see http://graphblas.org and
    https://people.engr.tamu.edu/davis/GraphBLAS.html.
 -  The LAGraph library, a collection of high level graph algorithms based on
    the GraphBLAS C API, see https://lagraph.readthedocs.io/en/latest/ and
    https://github.com/GraphBLAS/LAGraph.

To compile this C code, one needs to compile both SuiteSparse:GraphBLAS and
LAGraph and add their folders to the folder containing main.c.

--------------------------------------------------------------------------------
## Compilation/Use

Requires compiled SuiteSparse:GraphBLAS and LAGraph folders in the same folder as
```

main.c. Compilation using some C compiler using the -llagraph and -lgraphblas flags, for example on MacOS:

```
clang -o program main.c -llagraph -lgraphblas
```

Then either run the program from terminal straight as:

```
./program
```

Or provide input and output folders as first and second arguments, respectively:

```
./program /SomePath/Input /SomePath/Output
```

When not providing these folders, the program will ask for them in the terminal. Next, the program will search for all graphs in Matrix Market format (.mtx) in the input folder and perform the MWM algorithm one-by-one on each graph and write all output metadata to a single .csv file named 'MWM_Log.csv' in the output folder.

Use of parallelism with OpenMP might cause problems during compilation on MacOS with the clang compiler. Try using the GCC compiler and use the -fopenmp flag, for example:

```
gcc main.c -o program -llagraph -lgraphblas -fopenmp
```

One might need to link the libraries via the terminal, something like:

```
gcc main.c -o program -fopenmp -L /usr/local/lib -lgraphblas -llagraph
    -I /usr/local/include
```

Parallelism only works when SuiteSparse:GraphBLAS is compiled with OpenMP. For more on this and other options on OpenMP parallelism, see the GraphBLAS User Guide.

Only compilation on MacOS has been tested, there are no guarantees for Linux/Windows systems but with GraphBLAS, LAGraph and OpenMP installed compilation should likely be possible.

Calculation of the MWM approximation is done via strategy of selecting iterations. The standard strategy is set to BASIC, but can be set to ALTERNATING or ONEAUG_PREFERENCE in the generalheader.h file by changing the line:

```
#define STRATEGY BASIC
```

to either one of the following:

```
#define STRATEGY ALTERNATING
#define STRATEGY ONEAUG_PREFERENCE
```

See the thesis for a description of each strategy.

```
---
Other options on what output is given in the terminal, debugmode and number of
threads used can be changed in the generalheader.h file.

--------------------------------------------------------------------------------
## Error messages

The program throws warnings when using GraphBLAS 8.0.0 with the latest version of
LAGraph about the GraphBLAS JIT Kernel and user defined types like: "error:
unknown type name 'VX1X2Struct'". These can be ignored, have no effect on the
program outcome and are hopefully solved with the coming version of LAGraph.

--------------------------------------------------------------------------------
```