SCHOOL OF COMPUTER SCIENCE

An Evaluation of a State-of-the-Art Semi-Streaming Algorithm for Approximate Maximum Matchings in General Graphs

Phillip Sheng Wu Daniel

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Bachelor of Science in the Faculty of Engineering **worth 40CP**.

Friday 9th May, 2025

Abstract

The Maximum Matching (MM) problem is a problem in graph theory where the goal is to find the largest set of vertex-disjoint edges in a graph. With the rise of big data, the size of real-world graphs has grown exponentially larger, often making it infeasible to store an entire graph in memory. Instead, semi-streaming algorithms are used, which reduce the space complexity to sub-linear by processing the graph an edge at a time, thereby eliminating the need to store the entire graph in memory.

Finding an exact solution to the MM problem under the semi-streaming model has been proven to be prohibitively computationally expensive, so instead we find a $(1+\epsilon)$-approximate solution. In bipartite graphs, solving for this approximation is relatively easy, requiring $O(1/\epsilon^2)$ passes over the edge stream [ALT21]. Until recently, algorithms for computing the same problem in general graphs have had a pass complexity exponentially dependent on ϵ [McG05, Tir18]. However, a major breakthrough has improved this dependency, reducing it to a polynomial pass complexity of $O(1/\epsilon^{19})$ [FMU21]. This has since been improved to $O(1/\epsilon^6)$ by an algorithm that we will refer to as the MMSS algorithm [MMSS25].

In this work, we have implemented the MMSS algorithm in the C++ programming language, allowing us to investigate how the algorithm performs in practise, both on real-life and adversarially designed graphs. Further, we introduce optimisations made to our implementation that exploit the structure of specific graphs, resulting in a significant reduction in the number of passes required in real-world graphs. Our empirical results show us that in practise, our implementation performs significantly better than its theoretical worst-case both in pass complexity and approximation factor.

Dedication and Acknowledgements

I am deeply grateful to my supervisor, Dr. Christian Konrad, for his unwavering guidance and support throughout the duration of this project, which I have found invaluable. His advice and encouragement has been instrumental in shaping this project, and after every meeting, I felt reinvigorated to explore the project further. I would also like to thank my family and friends for their endless support and belief in me over the past three years of my undergraduate journey, which has made it so enjoyable.

Declaration

I declare that the work in this dissertation was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Taught Programmes and that it has not been submitted for any other academic award. Except where indicated by specific reference in the text, this work is my own work. Work done in collaboration with, or with the assistance of others including AI methods, is indicated as such. I have identified all material in this dissertation which is not my own work through appropriate referencing and acknowledgement. Where I have quoted or otherwise incorporated material which is the work of others, I have included the source in the references. Any views expressed in the dissertation, other than referenced material, are those of the author.

Phillip Daniel

Phillip Sheng Wu Daniel, Friday 9th May, 2025

AI Declaration

I declare that any and all AI usage within the project has been recorded and noted within Appendix A or within the main body of the text itself. This includes (but is not limited to) usage of text generation methods incl. LLMs, text summarisation methods, or image generation methods.

I understand that failing to divulge use of AI within my work counts as contract cheating and can result in a zero mark for the dissertation or even requiring me to withdraw from the University.

Phillip Daniel

Phillip Sheng Wu Daniel, Friday 9th May, 2025

Contents

1 Introduction **1**
- 1.1 Context . . . 1
- 1.2 Further Related Work . . . 4
- 1.3 Project Aims . . . 5
- 1.4 Outline . . . 6

2 Preliminaries **7**
- 2.1 Graphs . . . 7
- 2.2 Approximation Algorithms . . . 7
- 2.3 Streaming . . . 7
- 2.4 Matchings . . . 8
- 2.5 Alternating Trees and Blossoms . . . 8

3 Algorithm Overview **11**
- 3.1 Algorithm Preliminaries . . . 11
- 3.2 The MMSS Algorithm . . . 12
- 3.3 Free Vertex Structure Operations . . . 19

4 Algorithm Implementation **24**
- 4.1 Language Choice . . . 24
- 4.2 Graph Streaming . . . 24
- 4.3 Data Structures . . . 25
- 4.4 Further Algorithm Considerations . . . 30

5 Critical Evaluation **32**
- 5.1 Datasets . . . 32
- 5.2 Approximation Analysis . . . 33
- 5.3 Pass Optimisations . . . 37
- 5.4 Performance Analysis . . . 40
- 5.5 Randomised Edge Order . . . 41
- 5.6 Adversarial Graphs . . . 43
- 5.7 Operation Observations . . . 44

6 Conclusion **47**
- 6.1 Our Contributions . . . 47
- 6.2 Future Work . . . 47

A AI Prompts **51**

B Code Library **52**

C Additional Results **53**

List of Figures

1.1 An example of a simple graph structure. . . . 1
1.2 An example of the Maximum Matching problem. . . . 2
1.3 An example of an augmentation of a matching in a bipartite graph. . . . 2
1.4 An figure showing how an odd-length alternating cycle in a graph can cause problems when searching for an augmenting path. . . . 3
1.5 An example of contractions and expansions in Edmonds' blossom algorithm. . . . 4

2.1 A visualisation of an edge stream of a graph. . . . 8
2.2 An example of a graph with a regular set of blossoms and its corresponding alternating tree contraction. . . . 10

3.1 A figure demonstrating the representation of an edge using two directed arcs. . . . 11
3.2 A partial visualisation of a free vertex structure. . . . 12
3.3 A figure demonstrating the importance of vertex-disjointness between augmenting paths. . . . 14
3.4 An example showing the importance of the order of steps in the CONTRACT-AND-AUGMENT procedure. . . . 18
3.5 An example of the AUGMENT operation forming an augmenting path. . . . 19
3.6 A example of the CONTRACT operation. . . . 20
3.7 An example of Case 2 of the OVERTAKE operation. . . . 21
3.8 An example of Case 3 of the OVERTAKE operation. . . . 23

4.1 A UML class diagram illustrating the relationship between the three classes used to emulate the streaming of a graph. . . . 25
4.2 A UML class diagram illustrating the relationship the classes used in our implementation of the MMSS algorithm. . . . 26
4.3 An example of the contents of a `GraphBlossom` object. . . . 28
4.4 An example of a `FreeNodeStructure` object and the `GraphNode` objects contained within. 29

5.1 An example of the standardisation of a graph file from a `.csv` file to the format used to describe graphs in our implementation. . . . 32
5.2 A graph showing the increase in the size of the matching as the algorithm progresses. . . . 33
5.3 A magnified section of Figure 5.2, showing the change in matching size during the first scale of the algorithm. . . . 34
5.4 A magnified section of the graph in Figure 5.3, showing the change in matching size during phase one of the first scale of the algorithm for various graphs. . . . 36
5.5 A graph showing the number of free vertex structure operations that occur during each pass bundle of the first phase of scale one, using the feather-lastfm-social graph, with $\epsilon = 0.75$. . . . 36
5.6 An example showing why the early termination check may not always work. . . . 37
5.7 An example of a half graph, its adversarially ordered edge stream, and the initial matching produced. . . . 43
5.8 A figure showing the varying lengths of the augmenting paths found during the execution of the MMSS algorithm on the lexicographically ordered feather-lastfm-social graph with $\epsilon = 0.75$. . . . 45

C.1 A graph showing the change in matching size for the feather-lastfm-social during the execution of the MMSS algorithm with varying values of ϵ. . . . 54

List of Tables

1.1 A summary of the semi-streaming algorithms designed to compute a $(1+\epsilon)$-approximate maximum matching of general graphs and their pass complexities. . . . 5

5.1 A table showing relevant information about the graphs from SNAP dataset library [LK14] used to test our implementation, ordered by ascending edge count. . . . 33
5.2 A table showing the phases and scales of the algorithm during which the matching size changes. . . . 35
5.3 A table showing the point at which our implementation finishes when the early finish condition was enabled. . . . 37
5.4 A table showing number of passes required by our implementation of the MMSS algorithm before optimisation, for varying values of ϵ. . . . 38
5.5 A table showing the number of passes over the edge stream the MMSS algorithm requires to compute the maximum matching for a specified value of ϵ after the optimisations described in Section 5.3. . . . 39
5.6 A phase-by-phase breakdown of the algorithm's progress on the feather-lastfm-social graph, with $\epsilon = 0.1$ and the three skip optimisations enabled. . . . 40
5.7 Specifications of the device used for the performance analysis. . . . 40
5.8 A table showing the performance of our implementation on the real-life datasets. . . . 41
5.9 A table showing the time taken per pass-bundle for each real-life graph. . . . 42
5.10 A table showing the number of passes used when running the MMSS algorithm on graphs with randomised edge streams. . . . 42
5.11 A table showing relevant information about the half graphs tested during this section, ordered by ascending edge count. . . . 43
5.12 A table showing the number of passes used by the MMSS algorithm on the adversarially ordered half graphs. . . . 44
5.13 A phase-by-phase breakdown of the size and structure of the blossoms used during the execution of the MMSS algorithm on the lexicographically ordered edge stream of the feather-lastfm-social graph, where $\epsilon = 0.75$. . . . 46

C.1 A table showing the phases and scales of the algorithm during which the matching size changes for the remaining graphs not displayed in Table 5.2. . . . 53
C.2 A table showing a breakdown of the number of passes used by the MMSS algorithm before optimisation for different values of ϵ. . . . 54
C.3 A table showing a breakdown of the free vertex structure operations completed during the first phase of the first scale of the MMSS algorithm for the feather-lastfm-social graph, when $\epsilon = 0.75$. . . . 55
C.4 A table showing the point at which our implementation finishes when the early termination condition enabled for the remaining graphs not displayed in Table 5.3. . . . 55
C.5 A phase-by-phase breakdown of the algorithm's progress on the feather-lastfm-social graph, with $\epsilon = 0.75$ and the three skip optimisations enabled. . . . 56
C.6 A phase-by-phase breakdown of the algorithm's progress on the feather-lastfm-social graph, with $\epsilon = 0.5$ and the three skip optimisations enabled. . . . 56
C.7 A phase-by-phase breakdown of the algorithm's progress on the feather-lastfm-social graph, with $\epsilon = 0.25$ and the three skip optimisations enabled. . . . 57
C.8 A phase-by-phase breakdown of the algorithm's progress on the adversarially ordered edge stream of the halfGraph-1600 graph, with $\epsilon = 0.5$ and the three skip optimisations enabled. 58

C.9 A phase-by-phase breakdown of the size and structure of the blossoms used during the execution of the MMSS algorithm on the randomly ordered edge stream of the feather-lastfm-social graph, where $\epsilon = 0.75$. 59

Ethics Statement

This project did not require an ethical review, as determined by my supervisor, Dr. Christian Konrad.

Supporting Technologies

- The C++ programming language was used to create our implementation of the streaming model and MMSS algorithm.
- During the implementation of the MMSS algorithm, the Boost C++ library was used for its implementation of hashing for tuple-like structure (`boost::hash`). During testing and benchmarking, the same library was used for its implementation of Edmonds' blossom algorithm.
- Graphs from the Stanford Large Network Dataset Collection by the Stanford Network Analysis Project [LK14] were used to test our implementation.
- The `matplotlib` Python library (`https://matplotlib.org`) was used for graph plotting.

Chapter 1

Introduction

1.1 Context

1.1.1 Graphs and Streaming

With the emergence of big data, processing extremely large datasets consisting of millions of data points has become increasingly common. This data can often be represented through graphs, a data structure which can be used to model complex, non-linear relationships between objects, using vertices to represent the objects and edges between the vertices to represent a relationship between them. A frequently used example of this is the representation of a social network, where each vertex represents a user, and an edge represents a connection between two users, such as in Figure 1.1.

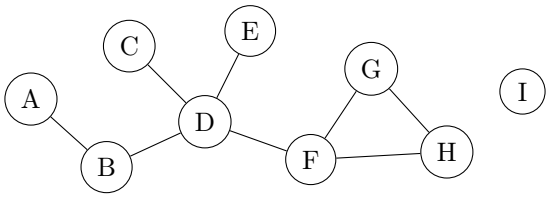


Figure 1.1: **An example of a simple graph structure.** The graph contains 8 vertices (labelled A-I) and 8 edges (straight lines). If each vertex were to represent a person and each edge represents a relationship between two people, we see that B and D are directly related as they have an edge between them, A and D are indirectly related through B, and I is not related to anyone.

These graphs can be extremely large; for example, to give a sense of scale, the popular social networking application Facebook has more than 3 billion users [SDM24] and users of the application are able to "friend" other users. If we approximate that each user is friends with 50 other users, a graph with more than 3 billion vertices and 75 billion edges would be needed to represent these relationships. The vast size of these graphs can cause difficulties when processing them, with one of the major problems being that storing the entire graph in the memory of a computer is not feasible. This has led to the idea of graph streaming, where instead of storing the graph in memory, the edges of the graph are provided sequentially, one at a time in a stream (Definition 2.4). Algorithms should be designed to process the graph in as few passes of the edge stream as possible. The semi-streaming model is one of the most widely-used models of streaming that limits algorithms to a sub-linear space complexity.

Streaming algorithms can also be incredibly useful in situations where storing the entire graph in memory is not only infeasible but also unnecessary. An example of such a situation is the analysis of a web graph [BKM+00], where each vertex represents a website, and an edge represents a hyperlink between two websites. In this case, rather than creating and storing the entire graph structure, we can just analyse it in real time as we crawl the web, traversing each website.

1.1.2 Maximum Matchings

One of the fundamental problems in graph theory is the Maximum Matching (MM) problem. The aim of MM is to find the largest possible matching in a graph, where a matching is a set of edges that do not

share common vertices, this is formally defined in Definition 2.6. The MM problem has many practical applications, such as finding an optimal pairing of individuals, as shown in Figure 1.2.

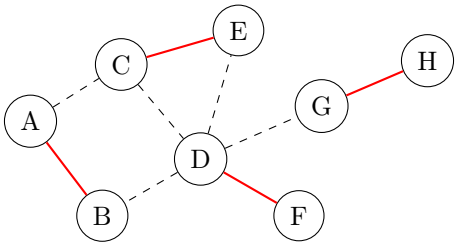


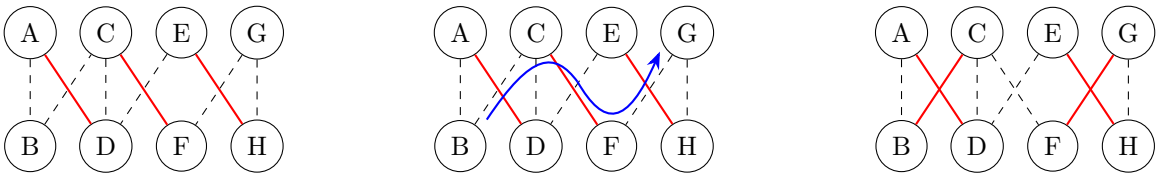
Figure 1.2: **An example of the Maximum Matching problem.** Imagine we have eight students, labelled A-H, who are connected by an edge if they work well together. We would like to group the students into pairs so that they are all working with someone they are compatible with. This problem can be solved by finding a maximum matching. In the figure, the edges marked in red are an example of a possible maximum matching of the graph.

Variants of the Maximum Matching problem also have real-life applications within the context of streaming, such as the Online Ad Assignment problem [FKM+05], where the goal is to dynamically match advertisements to relevant users as users arrive, according to predefined constraints, which can help provide more accurate targeted advertising.

1.1.3 Why Is Finding a Maximum Matching in General Graphs So Difficult?

Research into the classical, non-streaming Maximum Matching problem can broadly be split into two main categories, general graphs and bipartite graphs. General graphs refer to any possible graph, whereas bipartite graphs refer to a specific subset of general graphs whose vertices can be split into two disjoint sets such that each edge in the graph has exactly one vertex in each set.

A common strategy for finding a maximum matching is to greedily find an initial matching and then repeatedly find augmenting paths (Definition 2.9). We can then augment the matching with this path which involves removing every matched edge in the path from the matching and replacing them with the unmatched edges. Since an augmenting path starts and ends at a free vertex (Definition 2.8) and therefore must start and end with an unmatched edge, the path must contain one more unmatched edge than matched edge. This means that every augmenting path increases the size of the matching by one.



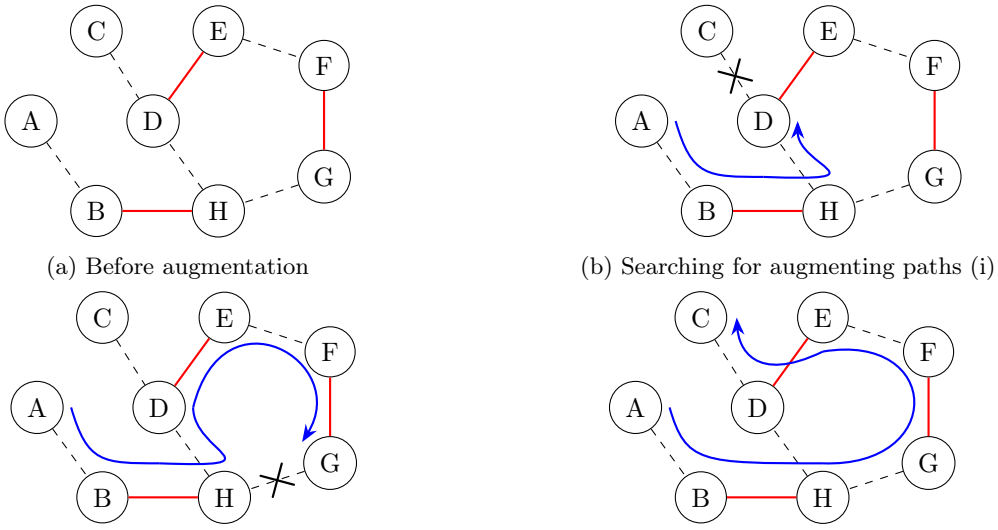
(a) Before augmentation (b) Augmenting Path (c) After augmentation

Figure 1.3: **An example of an augmentation of a matching in a bipartite graph.** Dashed black and solid red lines denote unmatched and matched edges, respectively. **(a)** The graph and its matching (of size three) before augmentation. **(b)** An augmenting path of the graph, marked with a blue arrow. The augmenting path shown is (B, C, F, G). **(c)** The graph and matching once it has been augmented with the path in (b), now with a matching of size four. This is a maximum matching.

The process of augmentation is demonstrated within a bipartite graph in Figure 1.3. Augmenting paths can be found easily in bipartite graphs using a breadth-first search because their structure guarantees that they cannot contain odd cycles. This has resulted in algorithms such as the Hopcroft-Karp algorithm [HK73], which can solve the Maximum Matching problem for an n-vertex, m-edge bipartite graphs in $O(m\sqrt{n})$ time. This runtime has since been significantly improved to give an almost-linear runtime of $O(m^{1+o(1)})$ by Chen *et al.* [CKL+22].

However, in general graphs it is possible to have odd-length cycles, and so can be problematic when searching for augmenting paths, adding extra overhead to the computation. The key difficulty is that odd-length cycles introduce the possibility of both odd and even length alternating paths existing between two vertices. Algorithms such as the Hopcroft-Karp algorithm make use of breadth-first searches to find

the shortest paths, marking vertices as visited so they are not checked multiple times. However, in a general graph, the algorithm can be tricked into believing that no augmenting paths exist depending on the direction with which the search enters the cycle. An example of this can be seen in Figure 1.4, where in Figure 1.4b whilst searching for an augmenting path, the algorithm takes the wrong direction into the cycle, making it impossible to find an augmenting path. The search then continues around the cycle until it reaches a vertex it has already visited, as shown in Figure 1.4c. Although there is a augmenting path, shown in Figure 1.4d, which can be found by backtracking and revisiting visited vertices, because we are using a shortest path algorithm, this augmenting path is discounted, as the shortest path to vertex D from A is (A, B, H, D).



(a) Before augmentation

(b) Searching for augmenting paths (i)

(c) Searching for augmenting paths (ii)

(d) Correct augmenting path

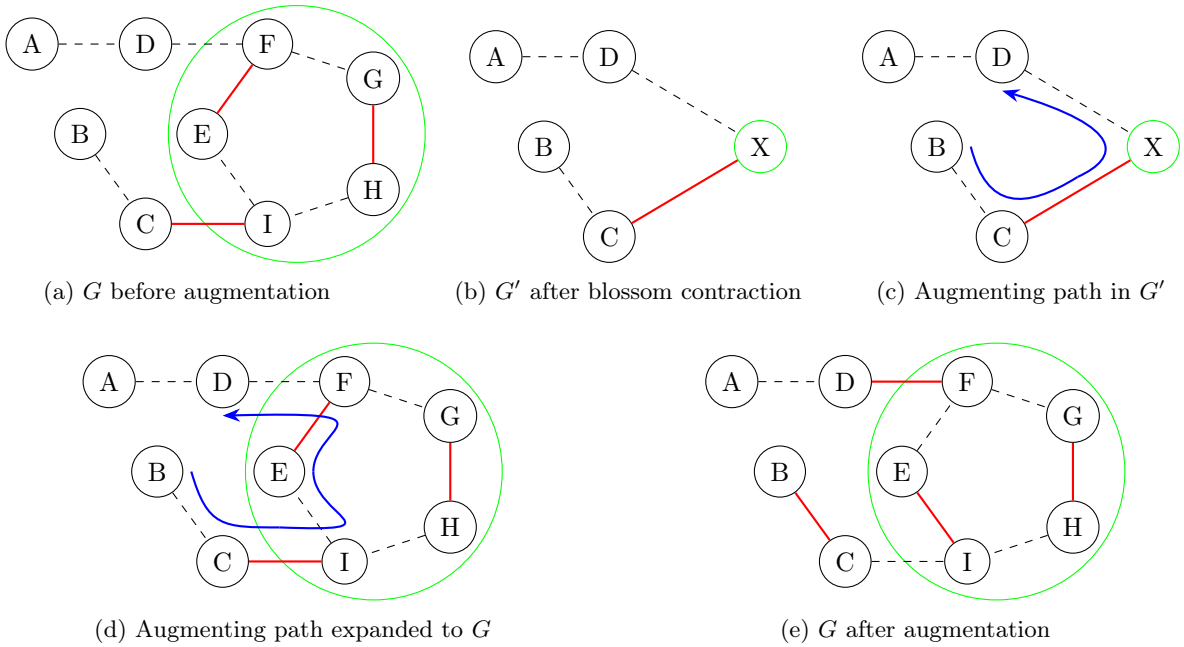
Figure 1.4: **An example showing how an odd-length alternating cycle in a graph can cause problems when searching for an augmenting path.** Dashed black and solid red lines denote unmatched and matched edges, respectively, and the solid blue arrow represents a path in the graph. **(a)** The graph and its matching before the augmentation, which contains an alternating cycle of length five, (D, E, F, G, H). **(b)** The graph mid-way through the search for an augmenting path, with the current path being (A, B, H, D). The edge (D, C) cannot be added to the path as it would then contain two unmatched edges in a row and therefore would no longer be alternating. **(c)** The path in (b) once it has been extended to (A, B, H, D, E, F, G). The unmatched edge (G, H) cannot be added to the path as the vertex H has already been visited. **(d)** The missed augmenting path, (A, B, H, G, F, E, D, C).

Because the traditional augmenting path algorithm could not be used directly, Edmonds introduced the Blossom Algorithm [Edm65], which could find augmenting paths in graphs with odd-length cycles. It was also the first algorithm to solve the Maximum Matching problem for general graphs in polynomial time. For a graph with n vertices and m edges, the runtime of the algorithm was $O(mn^2)$. This has since been improved by Micali and Vazirani [MV80, Vaz20], who produced an algorithm with $O(m\sqrt{n})$ runtime.

Edmond's blossom algorithm introduced the idea of a blossom (formally defined in Definition 2.12), which, informally, is an alternating odd-length cycle of nodes (which can be either other blossoms or individual vertices). Rather than searching for augmenting paths in the original graph, a contracted graph, denoted G', of the original graph is created by contracting each blossom into a single vertex (Definition 2.13). The process of contraction removes any odd-length alternating cycles from the contracted graph, allowing us to find augmenting paths as we can in bipartite graphs. Once an augmenting path is found in G', the path is then expanded into the original graph, where we rely on the fact that if G' has an augmenting path, there will be a corresponding augmenting path in G, Lemma 1.1. The process of contracting and expanding blossoms to find an augmenting path can be seen in Figure 1.5.

The following lemma is proved in [Edm65, Theorem 4.14].

Lemma 1.1 ([Edm65]) *Given a graph G, a matching M and a blossom B. Let G' be the graph created when contracting B in G. G' has an augmenting path if and only if there is a corresponding augmenting path in G.*



(a) G before augmentation (b) G' after blossom contraction (c) Augmenting path in G'

(d) Augmenting path expanded to G (e) G after augmentation

Figure 1.5: **An example of contractions and expansions in Edmonds' blossom algorithm.** Dashed black and solid red lines denote unmatched and matched edges, respectively. The blossom in the graph is marked in green. Augmenting paths are marked with a blue arrow. **(a)** The graph G and its matching (of size three) before augmentation. **(b)** The contracted graph of G, denoted G', where each vertex in the blossom has been contracted into a single vertex, X. **(c)** An augmenting path in the contracted graph G', (B, C, X, D). **(d)** The corresponding augmenting path from (c) expanded to the original graph, giving (B, C, I, E, F, D). **(e)** The resulting matching in G after augmentation with the path in (d), now with a matching of size four.

1.1.4 Why Do We Need to Approximate?

Computing the exact solution to a maximum matching in a streaming setting can be incredibly computationally intensive, requiring a large number of passes and a large amount of memory. Prior research on the lower bounds of space for streaming algorithms have shown that solving problems, such as Maximum Matching, on an n-vertex graph in $O(\sqrt{\log n})$ passes requires $n^{2-o(1)}$ space [CKP+21], with similar, but weaker bounds shown in [GO12]. This space requirement is significantly larger than the $O(n \operatorname{poly} \log n)$ requirement of the semi-streaming model.

Instead, we find an $(1+\epsilon)$-approximation (Definition 2.3) of the maximum matching, which gives us an answer within a factor of $(1+\epsilon)$ of the optimal solution, where we can alter the value of ϵ depending on the accuracy we require. A lower value of ϵ will give us a more accurate result but will require more computation. In the case of streaming algorithms, the smaller the ϵ value, the more passes that are required over the edge stream. The goal of these algorithms is to produce an $(1+\epsilon)$-approximation in the fewest possible passes over the edge stream.

1.2 Further Related Work

In this section, I will briefly mention some of the relevant research on the Maximum Matching problem in streaming. There are various different models of streaming, each of which differs in the amount of space algorithms that are allowed to use; however, we will be focusing on the semi-streaming model.

The semi-streaming model was first introduced in 2005 by Feigenbaum *et al.* [FKM+05] as a new model for efficiently solving graph problems where the graph is too large to be stored in memory. Rather than analysing the runtime of algorithms following this model, we instead look at the number of times we have to look through the edge stream (commonly referred to as the number of passes) to produce a solution.

One of the most common streaming algorithms for finding an initial matching is a greedy algorithm which produces a 2-approximation in a single pass of the edge stream [FKM+05], where we simply greedily

add edges to the matches when we see them, as long as the matching constraints are preserved.

The structural requirements within bipartite graphs once again make the computation easier, meaning it has been proven that a $(1+\epsilon)$-approximation for maximum matchings for bipartite graphs in the semi-streaming model can be computed with $O(1/\epsilon^2)$ passes [ALT21] of the edge stream. In contrast, work on general graphs has been shown to be more challenging, with up until recently the best pass complexity having an exponential dependence on $1/\epsilon$, i.e. $\exp(1/\epsilon)$ [McG05, Tir18] or with a polynomial dependence on $1/\epsilon$ but an additional dependence on $\log(n)$. However, significant improvements were made to this by Fischer, Mitrović and Uitto [FMU21] in 2021, who presented a semi-streaming algorithm for maximum matchings in general graphs which would output a $(1+\epsilon)$-approximation in $\text{poly}(1/\epsilon)$ passes of the graph, specifically $O(1/\epsilon^{19})$ passes. This was then improved on in 2025 by Mitrović *et al.* [MMSS25] who introduced an $O(1/\epsilon^6)$ algorithm. One of the reasons for the latest improvement in pass complexity is the use of Edmonds' blossoms [Edm65] within the algorithm.

A summary of the pass complexity of previous results can be found in Table 1.1.

Year	Passes	Deterministic?	Reference
2005	$\exp(1/\epsilon)$	No	[McG05]
2018	$\exp(1/\epsilon)$	Yes	[Tir18]
2021	$O(1/\epsilon^{19})$	Yes	[FMU21]
2025	$O(1/\epsilon^6)$	Yes	[MMSS25]

Table 1.1: **A summary of the semi-streaming algorithms designed to compute a $(1+\epsilon)$-approximate maximum matching of general graphs and their pass complexities.**

Similar research on maximum matching algorithms for weighted graphs has been completed, where the goal is to find the matching that maximises the total weight of edges within the matching; however, in this report we will be focusing on unweighted graphs.

1.3 Project Aims

With the context of our work established, we will now present the key objectives of this project. We aim to explore the most recent algorithm for finding a $(1+\epsilon)$-approximate maximum matching in general graphs in the semi-streaming model, the MMSS algorithm [MMSS25], which has a pass complexity of $O(1/\epsilon^6)$. Specifically, the goal of our project is as follows.

Investigate the performance of the MMSS algorithm on real-life graphs. Although significant work has been completed to establish the theoretical limitations of the MMSS algorithm and its predecessors, there is substantially less work detailing their practical performance. Whilst these theoretical bounds provide us with insight into worst-case scenarios, they do not necessarily reflect the algorithm's performance in real-world settings. With this in mind, the goal of our work is to explore the performance of the MMSS algorithm on real-life graphs, aiming to identify any potential optimisations that could improve this performance.

This broad goal can be broken down into several more specific objectives, which are now presented in the natural order of completion.

1. Implement the MMSS algorithm in an appropriate programming language. Included in this implementation should be code to track key performance metrics, such as execution time, memory usage, and number of passes used.

2. Design and implement optimisations to the algorithm that can reduce the number of passes required to compute a solution.

3. Gather a range of real-life and artificially created datasets for use in benchmarking. These datasets should adhere to a common format, allowing for them to be used as edge streams during testing of the algorithm.

4. Evaluate the performance of the MMSS algorithm on the benchmarking datasets, using the resulting information to demonstrate both the algorithm's real-life performance and the effectiveness of the optimisations made. This will show whether the algorithm performs as theoretically predicted or significantly better, as is often the case.

1.4 Outline

Our report is organised as follows. In Chapter 2, any technical definitions used during this report are formalised. In Chapter 3, we then provide an overview of the algorithm described in [MMSS25], followed by an explanation of our implementation in Chapter 4. Our results are then provided in Chapter 5, with our findings summarised in Chapter 6.

Chapter 2

Preliminaries

This chapter introduces definitions and the corresponding notation for the terminology used throughout this report.

2.1 Graphs

Definition 2.1 (Graphs, vertices and edges) *A graph $G = (V, E)$ consists of a set of vertices V and a set of edges E, where each edge consists of two vertices, formally, $\{u, v\}$ where $u, v \in V$.*

From this point forward, we will assume that any graphs are denoted as $G = (V, E)$, as per the definition. We will also denote the number of vertices, $|V|$, as n, and the number of edges, $|E|$, as m. When discussing multiple graphs, we will use $V(G_x)$ and $E(G_x)$ to denote the vertex and edge set of graph G_x respectively. We will also use u, v and t to represent vertices, unless otherwise stated.

Definition 2.2 (Bipartite graph) *A bipartite graph is a graph whose set of vertices, V, can be split into two disjoint subsets, V_1 and V_2, such that there is no edge between two vertices in the same subset. Formally, G is bipartite iff there exists a partition $V = V_1 \cup V_2$, where $V_1 \cap V_2 = \varnothing$, such that for every edge $\{u, v\} \in E$, either $u \in V_1$ and $v \in V_2$ or vice versa.*

A key point to note about bipartite graphs is that their structural requirement makes it impossible to have an odd-length cycle within the graph, which is why searching for augmenting paths in bipartite graphs is easier, as mentioned in Section 1.1.3.

2.2 Approximation Algorithms

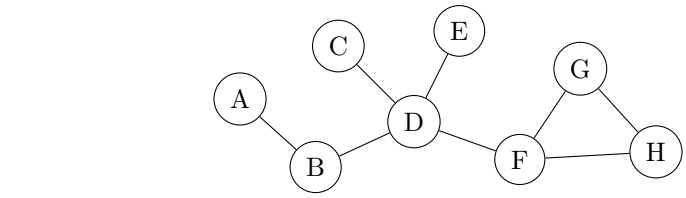
As mentioned in Section 1.1.4, rather than finding an exact solution to the Maximum Matching problem for general graphs in the semi-streaming setting, we use a-approximation algorithms, which guarantee that our solution will be within a factor a of the optimum solution.

Definition 2.3 (α-approximation algorithm) *An α-approximation algorithm, where $\alpha > 1$, for a problem P, is an algorithm that gives a solution, s, for the problem that is within a factor α of the optimal solution, Opt. For maximisation problems such as Maximum Matching, $1/(1+\epsilon) * |Opt| \leq |s| \leq |Opt|$.*

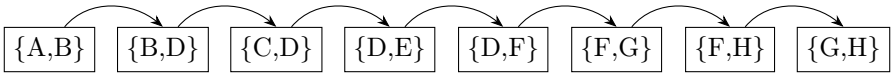
Throughout this report, we look in particular at $(1+\epsilon)$-approximation algorithms, where $\epsilon > 0$ and generally $\epsilon \leq 1$. There is typically a trade-off between the accuracy of the solution and the computation required, this is controlled by the value of ϵ. Going forward, we will be using Opt to denote the optimal solution of a problem, as per the definition.

2.3 Streaming

Definition 2.4 (Edge stream) *The edge stream S of a graph G is a sequence that contains each of the edges in G exactly once, in an arbitrary order. Formally, $S = (e_1, e_2, \ldots, e_m)$ where $\forall i \in [m], e_i \in E$ and $\forall i, j \in [m], i \neq j \implies e_i \neq e_j$. A graphical illustration of an edge stream is shown in Figure 2.1.*



(a) A simple graph



(b) A possible edge stream of the graph shown in (a)

Figure 2.1: **A visualisation of an edge stream of a graph.**

Definition 2.5 (Semi-streaming model) *An algorithm following the semi-streaming model is unable to access information in the input graph at will, but instead has access to the edge stream of the input graph. The stream can be read as many times as needed, but must be read to its entirety (also known as a pass) before it can be read again. The algorithm is also restricted to using $O(n \operatorname{poly}\log n)$ words of space.*

2.4 Matchings

Definition 2.6 (Matching, cardinality of a matching, a maximum matching) *A matching M in a graph is a set of edges $M \subseteq E$ where no two edges in M share a common vertex. The cardinality, or size, of a matching is the number of edges within the set M. A maximum matching is a matching in the graph with the maximum possible cardinality, an example can be seen in Figure 1.2.*

Definition 2.7 (Matched edge, unmatched edge) *Given a matching M, an edge $\{u, v\}$ is matched iff $\{u, v\} \in M$, otherwise $\{u, v\}$ is an unmatched edge.*

Definition 2.8 (Free vertex) *A vertex in a graph is defined as free (or unmatched) if none of the edges it is incident to are in the matching. Formally, v is a free vertex iff $v \in V$ and $\forall u \in V$ we have that $\{u, v\} \notin M$.*

Throughout the report, we will use α and β to denote free vertices, unless otherwise stated.

Definition 2.9 (Alternating path, augmenting path) *An alternating path is a path in a graph whose edges alternate between matched edges and unmatched edges. An augmenting path is an alternating path where the endpoints of the path are both free vertices. Formally, an augmenting path P for matching M is a path in G such that $P = v_0, v_1, ..., v_k$ where $\{v_i, v_{i+1}\} \in M$ for all odd i, $\{v_i, v_{i+1}\} \notin M$ for all even i and $v_0, v_k \notin \bigcup_{e \in M} e$.*

As in [MMSS25], we will use various notations to represent alternating paths. Given an alternating path $P = (u_1, v_1, \ldots, u_n, v_n)$, where $u_i, v_i \in V$ and (u_i, v_i) represents a matched arc, (v_i, u_{i+1}) represents an unmatched arc, we will often use the notation $P = (a_1, a_2, \ldots, a_n)$, where $a_i = (u_i, v_i)$, i.e. we omit the unmatched arcs. To represent alternating paths starting or ending with unmatched edges, we use the notation $P = (x, a_1, a_2, \ldots, a_n, y)$, where x and y are vertices. Finally, we will use $\overleftarrow{P}$ to denote the reverse of a path.

Definition 2.10 (Concatenation of alternating paths) *Given two alternating paths P_1 and P_2 where $P_1 = (a_1, a_2, \ldots, a_i)$ and $P_2 = (b_1, b_2, \ldots, b_j)$, we use the notation $P_1 \circ P_2 = (a_1, a_2, \ldots, a_i, b_1, b_2, \ldots, b_j)$ to describe the alternating path formed by following P_1 and then P_2. This alternating path also includes the unmatched edge between a_i and b_1.*

2.5 Alternating Trees and Blossoms

Definition 2.11 (Alternating tree, inner vertex, outer vertex) *An alternating tree, G_T, in a graph G with respect to a matching M, is a tree rooted with a free vertex where every root-to-leaf path is an*

even-length alternating path. An inner vertex of an alternating tree is a vertex with an odd-length root-to-vertex alternating path. An outer vertex is a vertex with an even-length (or 0-length) root-to-vertex alternating path.

A key note to take from this definition of an alternating tree is that every leaf in the tree is involved in a matching with its parent vertex, and every leaf (and the root) is an outer vertex.

Definition 2.12 (Blossom, as per [MMSS25]) *For a graph G, a blossom is a set of vertices B and edges E_B. We describe a trivial blossom as a blossom containing a single vertex and no edges, that is, $B = \{v\}, E_B = \emptyset$ where $v \in V(G)$. We define a blossom recursively as follows, given an odd number of vertex-disjoint blossoms $A_0, A_1, \ldots, A_k$, if for each $i \in [k]$, A_i is connected to $A_{(i+1)mod(k+1)}$ by an edge e_i, where $e_i \in [A_i \times A_{(i+1)mod(k+1)}]$ and $e_i \in M$ if and only if i is odd, then $B = \bigcup_i A_i$ is also a blossom, with a corresponding set of edges $E_B = (\bigcup_i E_{A_i}) \cup \{e_0, e_1, \ldots, e_k\}$.*

We can think of blossoms as an odd-length alternating cycle, where each vertex in the cycle is either a vertex (a trivial blossom) or a blossom itself.

When a blossom is expanded to find an augmenting path, the path within the blossom is always even, which ensures that the edge exiting the blossom is matched if and only if the edge entering the blossom was unmatched, and vice versa. This makes use of the following lemma, which is proved in [DP14, Section 2.1] and discussed in [MMSS25, Lemma 2.5].

Lemma 2.1 ([DP14]) *Given a blossom B, there exists an even-length alternating path in E_B between any two vertices in B.*

Definition 2.13 (Contraction of a Blossom) *Given a graph G and a blossom B, the contracted graph, G', is obtained by contracting all the vertices in B into a single new vertex. Formally, $G' = (V', E')$ where $V' = (V \setminus B) \cup x_B$, where x_B is a new vertex representing the blossom and $E' = (E \cap [(V \setminus B) \times (V \setminus B)]) \cup \{(x_B, w) : \exists v \in B \text{ st. } (w, v) \in E \wedge w \notin B\}$*

From this point forward, we will use the simplified expression G/B to represent the graph formed by contracting the blossom B in graph G. An example of blossom contraction can be seen in Figure 1.5a and Figure 1.5b.

The following lemma is proved in [Edm65, Theorem 4.13] and is discussed in [MMSS25, Lemma 2.7]. It is used to explain the behaviour of blossoms within alternating trees, which is used throughout the algorithm.

Lemma 2.2 ([Edm65]) *Given a graph G and an alternating tree T contained within G. If we have an edge $e \in E(G)$, which connects two outer vertices of T to one another, then there is an odd length cycle, and there exists a unique blossom B in $T \cup \{e\}$. Then G/B contains an alternating tree T/B, where B is a single outer vertex in the tree. The remaining elements of T/B are those in T which are not contained in B.*

Definition 2.14 (Laminar set of blossoms) *A set of blossoms Ω can be described as laminar if the set is a laminar family; this occurs when there is no partial overlap in the contents of any two blossoms in the set. Formally, for any $B_1, B_2 \in \Omega$ where $B_1 \neq B_2$, either $B_1 \cap B_2 = \emptyset$, $B_1 \subseteq B_2$ or $B_2 \subseteq B_1$.*

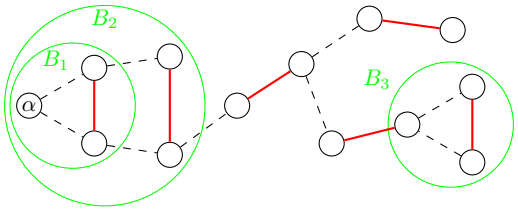
Definition 2.15 (Root Blossom) *Given a laminar set of blossoms Ω, the term root blossom describes any blossom in Ω not contained in any other blossom in Ω. Formally, B_r is a root vertex in Ω if and only if $B_r \in \Omega$ and $\forall B \in \Omega \setminus \{B_r\}, B_r \not\subseteq B$.*

As per [MMSS25], for each vertex $u \in \bigcup_{B \in \Omega} B$, we will use the notation $\Omega(u)$ to denote the unique root blossom containing u. We will use the notation G/Ω to represent the graph created by contracting G with every root node in Ω. Finally, we will use the notation M/Ω to denote the set of edges $\{\{\Omega(u), \Omega(v)\} \mid \{u, v\} \in M \text{ and } \Omega(u) \neq \Omega(v)\}$, where Ω contains all the vertices in G.

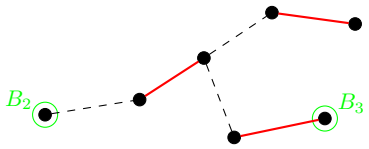
Definition 2.16 (Regular set of blossoms, as per [MMSS25]) *Given a graph G, a regular set of blossoms Ω is a set of blossoms where the following requirements are satisfied:*

- *Ω is a laminar set containing blossoms of G, including every trivial blossom in G. For any blossom $B \in \Omega$, if it is not a trivial blossom, then Ω contains all the blossoms in B.*
- *G/Ω is an alternating tree with respect to the matching M/Ω, with a root of $\Omega(\alpha)$. Each of the inner vertices of G/Ω are trivial blossoms.*

Figure 2.2 is an example of a graph G containing a regular set of blossoms Ω and its matching M. We can also see the corresponding contracted graph G/Ω and its matching M/Ω.



(a) A graph G and matching M containing a regular set of blossoms Ω, where $B_1, B_2, B_3 \in \Omega$



(b) The contracted graph G/Ω and its corresponding matching M/Ω

Figure 2.2: **An example of a graph with a regular set of blossoms and its corresponding alternating tree contraction.** Dashed black and solid red lines denote unmatched and matched edges, respectively. **(a)** A graph G with a regular set of blossoms Ω and a matching M. We can see the 3 non-trivial blossoms B_1, B_2 and B_3 marked in green, where B_2 and B_3 are root blossoms as $B1 \subseteq B_2$ and $B_2 \cap B_3 = \emptyset$. We also see a free vertex marked α, where $\Omega(\alpha) = B_2$. **(b)** The graph created by contracting G with every root node in Ω, which creates a valid alternating tree, with root $\Omega(\alpha)$. We can also see the corresponding matching M/Ω.

Chapter 3

Algorithm Overview

In this chapter, we will give an overview of the algorithm introduced in [MMSS25], which we will refer to as the MMSS algorithm. A complete proof of the correctness of the approximation factor, pass complexity, and space complexity of the algorithm can be found in [MMSS25, Chapter 5, 6]. To maintain consistency, throughout the report we will attempt to use terminology similar to those used in previous works, particularly those used by [FMU21, MMSS25].

3.1 Algorithm Preliminaries

3.1.1 Representation of an Edge

In the MMSS algorithm, we represent a single edge, $\{u, v\}$, as two directed arcs, (u, v) and (v, u). A visual representation of this can be seen in Figure 3.1. For an arc (u, v), we call u the tail and v the head of the arc. When viewing the edge stream of a graph, we will see both directed arcs corresponding to the edge in the stream one at a time.

(a) An edge (b) Directed arcs corresponding to the edge in (a)

Figure 3.1: **A figure demonstrating the representation of an edge using two directed arcs. (a)** An edge $\{A, B\}$, read as "an edge between A and B". **(b)** The two directed arcs corresponding to the edge in (a), the upper arc is (A, B), read as "an arc from A to B", and the lower arc is (B, A), which can also be written as $\overleftarrow{(A, B)}$.

Given a matching M, we say that the arcs (u, v) and (v, u) are matched if and only if the edge they correspond to is contained within the matching, that is, $\{u, v\} \in M$.

3.1.2 The Free Vertex Structure

Throughout the MMSS algorithm, we store a *structure* for each free vertex. This is used to store a tree, similar to those used in depth-first searches, rooted at a free vertex, and is used to find augmenting paths within the graph. We will refer to the free vertex structure of a free vertex simply as its structure.

Definition 3.1 (Free vertex structure, as per [MMSS25]) *Given a free vertex α in a graph G with respect to a matching M, we define the free vertex structure, S_α as a tuple $(G_a, \Omega_\alpha, w'_\alpha)$, where:*

- *G_α is a directed subgraph of G*
- *Ω_α is a regular set of blossoms in G_α*
- *The working vertex w'_α is either $\emptyset$ or an outer vertex of the alternating tree G_α/Ω_α*

Throughout the algorithm, the following properties are satisfied:

- ***Disjointness:** For any free vertex $\beta \neq \alpha$, G_α is vertex-disjoint from G_β, i.e., $V(G_\alpha) \cap V(G_\beta) = \emptyset$.*

- ***Tree Representation:*** *If G_α contains an arc (u, v) with $\Omega_\alpha(u) \neq \Omega_\alpha(v)$, then $\Omega_\alpha(u)$ is a parent of $\Omega_\alpha(v)$ in the alternating tree G_α/Ω_α, which is rooted by $\Omega_\alpha(\alpha)$. We will commonly denote the alternating tree G_α/Ω_α of S_α as T'_α.*
- ***Unique arc property:*** *For each arc $(u', v') \in E(G_\alpha/\Omega_\alpha)$, there is a unique arc $(u, v) \in E(G_\alpha)$ such that $\Omega_\alpha(u) = u'$ and $\Omega_\alpha(v) = v'$.*

An important point to note is that for a structure S_α, its subgraph G_α may not be a vertex-induced subgraph of G. That is, it is possible that G contains arcs between vertices in G_α that are not in $E(G_\alpha)$, or formally, it is possible that $\{(u, v) \in E(G) \mid u, v \in V(G_\alpha)\} \setminus E(G_\alpha) \neq \emptyset$.

Definition 3.2 (Active structure) *We say that a structure S_α is active if and only if $w'_\alpha \neq \emptyset$, otherwise we say that the structure is inactive. Inactive structures occur once we have exhausted the depth-first search from the structure's free vertex, meaning that all currently reachable vertices have been visited.*

Definition 3.3 (Active path of a structure) *Given a structure S_α, if the structure is inactive, we define the active path of the structure as $\emptyset$. Otherwise, we define the active path of the structure as the unique path in T'_α from the structure's root, $\Omega_\alpha(\alpha)$ to its working vertex, w'_α.*

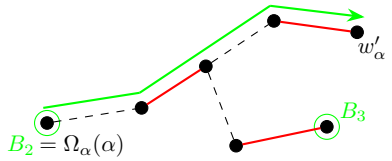


Figure 3.2: **A partial visualisation of a free vertex structure.** In the figure, we see the contracted graph $T'_\alpha = G_\alpha/\Omega_\alpha$ of a structure S_α, where G_α is the graph displayed in Figure 2.2a. Dashed black and solid red lines denote unmatched and matched edges, respectively. Non-trivial blossoms are marked with a green circle. Ω_α contains all the trivial blossoms in G, in addition to B_1 and B_2. The free vertex α is contained within the blossom B_2. Marked by a green arrow is the active path of the structure.

Throughout the algorithm, we also maintain a set Ω, where Ω contains every trivial blossom in G and every Ω_α for each free vertex $\alpha \in G$. We will use G' to denote the graph G/Ω. In addition, we will use the term "unvisited vertex" to describe any vertex that is not contained within a structure.

3.1.3 The Label of a Matched Edge

In the MMSS algorithm, we use a depth-first search-like algorithm to search for augmenting paths. In order to store the distance of visited vertices, we associate each matched edge with a *label*, as follows.

Definition 3.4 (The label of a matched edge) *Each matched edge $g \in M$ is assigned a label, denoted $\ell(g)$ where $1 \leq \ell(g) \leq \ell_{\max} + 1$, where $\ell_{\max} = 3/\epsilon$.*

In addition to the label of a matched edges, we also refer to the label of a matched arcs, which is defined similarly. For any edge, both of its corresponding arcs have the same label value as the edge. That is, for an edge $g = \{u, v\}$ and its corresponding arcs, $g_1 = (u, v)$ and $g_2 = (v, u)$, we have that $\ell(g) = \ell(g_1) = \ell(g_2)$.

Throughout the execution of the algorithm, we maintain that for any free node structure S_α, the sequence of labels along any root-to-leaf path in T'_α is always increasing. The correctness of this claim can be verified using a loop invariant over the algorithm.

3.2 The MMSS Algorithm

3.2.1 High-level Overview

In Algorithm 3.2, we can see a simple overview of the MMSS algorithm. We begin in Line 1 by obtaining an initial matching, this is done using a simple greedy algorithm, described in Section 3.2.2, which gives us a 2-approximate solution for the input graph. Similarly to other existing algorithms in both streaming and non-streaming settings, we then iteratively find augmenting paths in the graph relative to the current

matching, which we then use to augment the matching, increasing its size. We will describe the process of a single search for a set of augmenting paths and the resulting augmentation as a *phase*. We execute multiple phases, each with a *scale* parameter, which controls the maximum length of the augmenting paths found within the phase.

In Line 2, we begin this process of searching for augmenting paths. We begin with a for-loop, iterating through difference values of the scale h. In addition to controlling the maximum length of augmenting paths found, the scale also sets the number of phases run during the current scale and the number of passes over the edge stream during each phase; we will discuss this further in Section 3.2.5. As the value of the scale decreases, the number of phases run during the current scale, the number of passes per phase, and the maximum length of the augmenting paths found within a phase all increase. The size limitations of augmenting paths are an important part of the algorithm's correctness proof, which can be seen in [MMSS25, Chapter 5, 6].

In Line 3, we use a for-loop, where in each iteration we run a single phase, with the number of phases run depending on the current scale. Then, in Line 4, we execute the ALG-PHASE procedure (Algorithm 3.4), where we obtain $\mathcal{P}$, a set of vertex-disjoint augmenting paths for our existing matching M within G. We then augment M with $\mathcal{P}$ in Line 5, increasing the size of the matching by $|\mathcal{P}|$. This is done using the AUGMENT-MATCHING procedure (Algorithm 3.3). Between phases we do not store any information about the graph other than the edges involved in the current matching.

Algorithm 3.1: A high-level overview of the MMSS algorithm, a modified version of [MMSS25, Algorithm 1].

Input: A graph G and an approximation parameter ϵ
Output: A $(1+\epsilon)$-approximate maximum matching for G

```
1 M ← GREEDY-APPROX(G)                                   // (Algorithm 3.2)
2 for scale h = 1/2, 1/4, 1/8, ..., ε²/64 do
3     for phase t = 1, 2, 3, ..., 144/(hε) do
4         P ← ALG-PHASE(G, M, ε, h)                      // (Algorithm 3.4)
5         M ← AUGMENT-MATCHING(M, P)                     // (Algorithm 3.3)
6 return M
```

In the algorithm originally described in [MMSS25], the idea of removing vertices from the graph was introduced. During an execution of the ALG-PHASE procedure of the algorithm, if an augmenting path between two structures was found, all the vertices in the two structures, along with their associated edges would be removed from the graph. Upon competition of ALG-PHASE, these vertices would then be readded to the graph, this would occur between Line 4 and 5 of Algorithm 3.1. This ensured that every augmenting path found during a phase was vertex-disjoint from each other, the importance of this property is demonstrated in Section 3.2.3. In our implementation, we omit this idea, instead providing an alternative method and our reasons for this change in Section 3.2.4.

3.2.2 The GREEDY-APPROX Procedure

Algorithm 3.2: GREEDY-APPROX

Input: A graph G
Output: A 2-approximate of the maximum matching

```
1 M ← ∅
2 for each arc g = (u, v) ∈ E(G) in the edge stream do
3     if both u and v are not involved in any edges in M then
4         M ← M ∪ {u, v}
5 return M
```

In the MMSS algorithm, we begin by using a simple single-pass greedy approximation algorithm to get an initial matching, which is then iteratively improved on. Algorithm 3.2 provides a simple pseudocode for computing this greedy approximation. We begin with an empty matching, and in Line 2 we begin

an iteration over the edge stream, making a single pass. In Line 3 and 4 we check whether either of the vertices in the current arc are involved in a currently matched edge, if so, we skip to the next arc. Otherwise, we add the edge to the matching. This produces a 2-approximate matching for G that is maximal - no more edges can be added to the current matching without violating the matching properties.

3.2.3 The AUGMENT-MATCHING Procedure

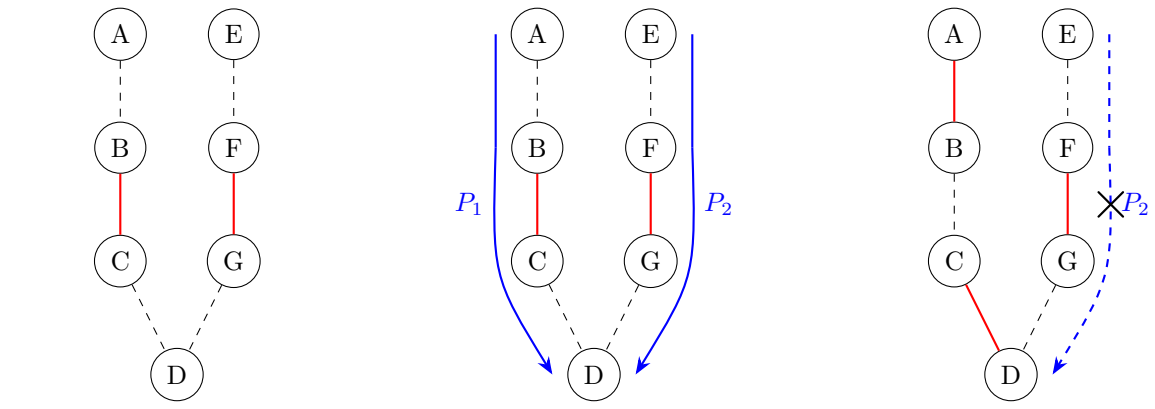
The goal of the AUGMENT-MATCHING procedure is to take a set of vertex-disjoint augmenting paths and augment the current matching with them. This is done by iterating through each of the paths within the set, then iterating through the edges of each path; if the edge is in the matching, we remove it from the matching, and otherwise we add it. Since each augmenting path contains 1 more unmatched edge than matched edges, augmentation increases the size of the matching by 1, hence $|\mathcal{P}|$ augmenting paths increase the size of the matching by $|\mathcal{P}|$, giving us a resulting matching of size $|M| + |\mathcal{P}|$. The pseudocode for the AUGMENT-MATCHING procedure can be seen in Algorithm 3.3.

Algorithm 3.3: AUGMENT-MATCHING

Input: The current matching M, a set of vertex-disjoint augmenting paths $\mathcal{P}$
Output: An augmented matching M' of size $|M| + |\mathcal{P}|$

```
1 for each path p ∈ P do
2     for each edge e = {u, v} ∈ path p do
3         if e ∈ M then
4             M ← M \ {e}
5         else
6             M ← M ∪ {e}
7 return M
```

A key point is to note that every augmenting path in the set $\mathcal{P}$ is vertex-disjoint with every other, that is, for all $P_1, P_2 \in \mathcal{P}$, $P_1 \neq P_2 \iff P_1 \cap P_2 = \emptyset$. This is important as it means that we can simultaneously augment all of the paths without them interfering with each other. If the augmenting paths were not vertex-disjoint, augmenting the matching with one path could cause another to no longer be valid. We can see an example of this in Figure 3.3, where our 2 possible augmenting paths, P_1 and P_2 are not vertex-disjoint (as $P_1 \cap P_2 = \{D\}$), meaning that once we augment the matching with one path, the other path is no longer a valid augmenting path.



(a) A graph G and matching M

(b) Two possible augmenting paths for M, P_1 and P_2

(c) M augmented with the left path, P_1

Figure 3.3: **A figure demonstrating the importance of vertex-disjointness between augmenting paths.** Dashed black and solid red lines denote unmatched and matched edges, respectively. Blue arrows represent augmenting paths within the graph. **(a)** A graph G and a matching M within it of size 2. **(b)** Two possible augmenting paths in G, these are $P_1 = (A, B, C, D)$ and $P_2 = (E, F, G, D)$. It is important to note that these paths are not vertex-disjoint. **(c)** After augmenting M with P_1, we are then unable to augment it with P_2.

3.2.4 The Alg-Phase Procedure

The Alg-Phase procedure gives an outline of the steps taken within a single phase. The pseudocode for the procedure is provided in Algorithm 3.4. During a single phase, we perform parallel depth-first searches from each of the free vertices in the graph, attempting to find augmenting paths between them.

As mentioned in Section 3.2.1, we omit the idea of removing vertices from the graph, instead replacing this with the idea of "used structures", which are structures which have been used to create augmenting paths during the current phase. At the beginning of every Alg-Phase procedure, we create a free vertex structure for each free vertex. By default, each structure is marked as *not used*, and if during the phase, a structure is involved in an augmenting path, it is marked as *used*. When we make passes over the edge stream, if either vertices of the arc are contained within a used structure, we ignore the arc and move to the next arc in the stream; this can be seen in practise in Algorithm 3.5.

We made this change to the algorithm because removing vertices from a graph stream seemed computationally expensive and difficult to manage, as it would require extra communication between the stream source and destination. Our alternative method of marking structures as *used* and *not used* achieved the same goal, without requiring any of the additional communication. Another alternative could have been storing a set of vertices which had been "removed", however, this would require more space than our method as each structure contains multiple vertices.

We begin the procedure in Line 1 by initialising our set of augmenting paths to $\emptyset$. In Line 2, we then set the label of each matched arc. Since we have not yet visited any of the matched arcs during the current depth-first search, we set each of them to a value of infinity, as there is currently no stored path to reach them from any of the free vertices. Line 3 then initialises a free vertex structure for each free vertex in the graph and mark each structure as *not used*; an explicit method for this is described in Section 4.3.4. Then in Line 4 and 5 we compute the values of limit_h and $\tau_{\max}$, which are parameters used to limit the current phase's depth-first search. These values are calculated based on the scale h of the current phase and are crucial for the proof of correctness for the algorithm, provided in [MMSS25, Chapter 5, 6]. In Line 6, we then begin a for-loop of $\tau_{\max}(h)$ iterations. We define each iteration of the loop as a single *pass-bundle*, in which we make a fixed number of passes over the edge stream (three passes in the case of our pseudocode). We can think of a pass-bundle as a single step within the parallel depth-first search of the graph.

During each pass-bundle, we complete the following steps. In Line 7 to 12 we begin by setting the status of each free node structure. We mark a structure as *on hold* if it contains more than limit_h vertices in the structure, otherwise we mark it as *not on hold*. During future stages of the pass-bundle, if a structure is *on hold*, we do not continue the depth-first search on it. This limits the size of augmenting paths, which we find later in the pass-bundle, to at most $2 \cdot \text{limit}_h$. This means that during early scales of the algorithm, we only search for short augmenting paths, and as the scale increases, we can search for longer paths. In addition, we also mark every structure as *not modified*. Later in pass-bundle, this allows us to check whether the structure has been changed during the current pass-bundle. Whenever we edit a structure, such as when we add an arc, contract a blossom, etc., we mark the structure as *modified*.

Once we have marked the status of each structure, we continue the pass-bundle by calling the Extend-Active-Path procedure (described in Section 3.2.5), which makes a single pass over the edge stream, extending any structures marked as *not on hold* if possible. We then call the Contract-And-Augment procedure (described in Section 3.2.6); this procedure makes a single pass over the edge stream, contracting any blossoms that contain the working vertex of a structure, and identifies pairs of structures where outer vertices connect to one another via an unmatched arc to form an augmenting path. Finally, we call the Backtrack-Stuck-Structures procedure (described in Section 3.2.7), which is similar to the typical backtrack step during a depth-first search. Here, if a structure is marked as both *not on hold* and *not modified*, we know that during the Extend-Active-Path procedure, there was no possible extension for the structure and during the Contract-And-Augment procedure, there was no blossom to contract. In this case, we change the working vertex of the structure to a previously visited vertex so we can carry on the depth-first search in the next pass-bundle.

3.2.5 The Extend-Active-Path Procedure

In the Extend-Active-Path procedure, we make a single pass over the edge stream, attempting to increase the size of the active pass of each free vertex structure that is not marked as "on hold". We do this by calling one of three different operations on the free vertex structures, Augment, Contract, and Overtake, each described in Section 3.3, depending on the arc we view.

Algorithm 3.4: ALG-PHASE: A modified version of [MMSS25, Algorithm 2].

Input: A graph G, the approximation parameter ϵ, the current scale h, the current matching M
Output: A set $\mathcal{P}$ containing disjoint augmenting-paths with respect to M

1 $\mathcal{P} \leftarrow \emptyset$
2 $\ell(a) \leftarrow \ell_{max} + 1$ for each arc $a \in M$
3 for each free vertex α, initialise its structure S_α and mark S_α as not used
4 $\tau_{\max}(h) = \frac{72}{h\epsilon}$
5 $\text{limit}_h = \frac{6}{h} + 1$
6 **for** *pass-bundle* $\tau = 1, 2, \ldots, \tau_{\max}(h)$ *do* **do**
7 **for** *each free vertex* $\alpha \in G$ **do**
8 **if** $|S_\alpha| > \textit{limit}_h$ **then**
9 Mark S_α as "on hold"
10 **else**
11 Mark S_α as "not on hold"
12 Mark S_α as "not modified"
13 EXTEND-ACTIVE-PATH(G, ϵ, M, S_α for each free vertex $\alpha \in G, \mathcal{P}$) // (Algorithm 3.5)
14 CONTRACT-AND-AUGMENT(G, M, S_α for each free vertex $\alpha \in G, \mathcal{P}$) // (Algorithm 3.6)
15 BACKTRACK-STUCK-STRUCTURES(S_α for each free vertex $\alpha \in G$) // (Section 3.2.7)
16 **return** $\mathcal{P}$

We provide pseudocode for the procedure in Algorithm 3.5, which works as follows. We begin by making a single pass over the edge stream. For each arc we see, we check whether we can extend on said arc, this requires the arc to be a matched arc not contained within a blossom, whose tail is the working vertex of a structure. If this arc is not extendable, we skip to the next. We make this check between Line 2 and 12. We also ignore the edge if either of the vertices belongs to used structures, which tells us that the structure has been used in an augmenting path in the current phase. This helps ensure that $\mathcal{P}$ remains disjoint, as we never modify any of the structures involved in $\mathcal{P}$. Finally, we check whether the structure is marked as *on hold* or *modified*, in which case we also skip to the next edge, since during every call of EXTEND-ACTIVE-PATH, we want to perform at most one operation on each structure.

Once we have verified that our arc can be used to extend a structure, we can continue. Given an arc $g = (u, v)$, we decide which operation to use based on 3 following cases:

- **Case 1:** $\Omega(v)$ is an outer vertex in the same structure as $\Omega(u)$, where $\Omega(v) \neq \Omega(u)$. In this case, the arc would produce a blossom in T'_α, so we use the CONTRACT operation on the arc to contract the blossom. We formally define the CONTRACT operation in Section 3.3.2.
- **Case 2:** $\Omega(v)$ is an outer vertex in a different structure to $\Omega(u)$. In this case, we can create an augmenting path by connecting the two structures with arc g. We use the AUGMENT operation to do this. We formally define the AUGMENT operation in Section 3.3.1
- **Case 3:** $\Omega(v)$ is an inner vertex in a structure S_β or unvisited vertex (contained within any structures). Since $\Omega(v)$ cannot be a free vertex, since it is an inner vertex and every free vertex must be an outer vertex, there must exist a matched arc $a = (v, t) \in E(G)$ whose tail is v. If the label of the matched arc a is greater than that of the last label on the active path of S_α we are able to add arcs g, v to S_α. If a is contained within a structure S_β, we also add the subtree rooted at v in G_β to S_α and it from S_β. We describe this operation as an "overtake" and formally define the OVERTAKE operation and it's individual cases in Section 3.3.3.

If the arc does not match any of the three cases, we skip to the next arc.

3.2.6 The CONTRACT-AND-AUGMENT Procedure

In the CONTRACT-AND-AUGMENT procedure, we repeatedly identify and contract blossoms in the graph until there exist no arcs in G connecting an outer vertex of a structure to a different outer vertex of the same structure. We then identify augmenting paths by searching for any arcs connecting the vertices of two distinct structures. The CONTRACT-AND-AUGMENT procedure is used to ensure that at the end of every pass-bundle there exists no arc in G' connecting two distinct outer vertices. This is an important

Algorithm 3.5: EXTEND-ACTIVE-PATH : A modified version of [MMSS25, Algorithm 3].

Input: A graph G, the approximation parameter ϵ, the current matching M, the structure S_α of each free vertex $\alpha \in G$, and the set of disjoint augmenting paths $\mathcal{P}$

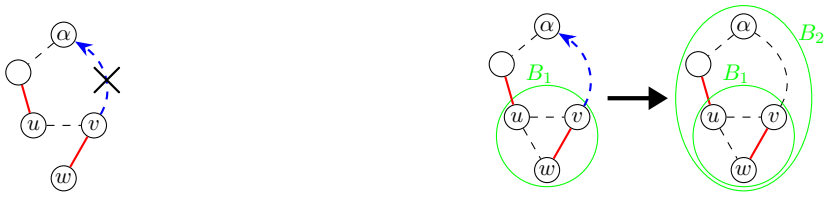
```
1  for each arc g = (u, v) ∈ E(G) in the edge stream do
2      if g ∈ M then
3          continue to the next arc
4      Let S_α and S_β denote the free node structures containing u and v respectively, or ∅ if u or v
        don't belong to a structure
5      if S_α = ∅ then
6          continue to the next arc
7      else if S_α or S_β is marked as "used" then
8          continue to the next arc
9      else if S_α = S_β and Ω(u) = Ω(v) in T'_α then
10         continue to the next arc
11     else if Ω(u) ≠ w'_α then
12         continue to the next arc
13     else if S_α is marked as "on hold" or "modified" then
14         continue to the next arc
15     if S_β ≠ ∅ and Ω(v) is an outer vertex in T'_β then
16         if S_α = S_β then
17             CONTRACT(g)                                   // (Algorithm 3.8)
18         else
19             AUGMENT(g)                                    // (Algorithm 3.7)
20     else
21         t ← the matched arc in E(G_α) whose head is Ω(u)
22         a ← the matched arc in G whose tail is v
23         if ℓ(t) + 1 < ℓ(a) then
24             OVERTAKE(g, a, l_u + 1)                       // (Algorithm 3.9)
```

invariant that is used to prove the correctness of the algorithm, which is provided in [MMSS25, Chapter 5].

We provide pseudocode for the procedure in Algorithm 3.6. It works as follows; we begin by making a pass over the edge stream. For each free vertex α, whenever we see an unmatched arc whose tail is the working vertex of the structure, w'_α, and whose head is a different outer vertex in the structure, we add it to the set A_α. Whilst there still exists an arc $g \in A_\alpha$ between w'_α and a different outer vertex in S_α, we call the CONTRACT operation, described in Section 3.3.2, on the arc g. We do this as g forms a blossom in the structure, which the CONTRACT operation then contracts. This contraction phase of the procedure occurs between Line 8 and 10.

Finally, we make another pass over the edge stream, calling the AUGMENT operation, described in Section 3.3.1, on any arc that connects the outer vertices of two distinct structures. This creates an augmenting path in the graph between the free vertices of the structures. It is important to note that we do not use AUGMENT if either of the vertices belongs to structures marked as used, this ensures that each augmenting path remains vertex-disjoint with every other. The augmentation phase of the procedure occurs between Line 11 and 17.

It is also important to note that during the contraction part of the procedure, we cannot just contract every outer vertex to outer vertex edge during a pass of the edge stream. Instead, we need to store each edge between two vertices in the same structure. This is because when we contract an arc, vertices that did not previously belong to an outer vertex of the contracted graph now do, creating more possible contractions which could be missed if we contract whilst making a pass over the stream. This can be seen in Figure 3.4. For similar reasons, we need to make two passes of the edge stream to complete the procedure rather than just one. This is because the augmentation steps has to occur after the completion of the contraction steps, otherwise new possible augmentations that were created when blossoms were contracted could be missed.



(a) Before contraction of u and v

(b) After contraction of u and v

Figure 3.4: **An example showing the importance of the order of steps in the CONTRACT-AND-AUGMENT procedure.** Dashed black and solid red lines denote unmatched and matched edges belonging to a free vertex structure S_α. Non-trivial blossoms are marked by a green ellipse. The dashed blue arrow represents an unmatched arc not in S_α. **(a)** In this figure, we cannot contract the graph on the edge $\{v, \alpha\}$, this is because in our algorithm contraction only occurs between two outer vertices and $\Omega(v)$ is an inner vertex. **(b)** Using the same graph shown in (a), if we contract on the edge $\{u, w\}$, creating the blossom B_1, containing vertices u, v and w, $\Omega(v)$ is now an outer vertex. This then allows us to contract the edge $\{v, \alpha\}$.

Algorithm 3.6: CONTRACT-AND-AUGMENT

Input: The graph G, the current matching M, the structure S_α of each free vertex $\alpha \in G$, the set of disjoint augmenting paths $\mathcal{P}$

```
1  Create a set A_α = ∅ for each free vertex α in G
2  for each arc g = (u, v) ∈ E(G) in the edge stream do
3      Let S_α and S_β denote the free node structures containing u and v respectively, or ∅ if u or v
        don't belong to a structure
4      if g ∈ M or S_α = ∅ or S_β = ∅ or S_α is marked as used or S_β is marked as used then
5          continue to the next arc
6      if S_α = S_β then
7          A_α ← A_α ∪ {g}
8  for each free vertex α ∈ G do
9      while there exists an arc g = (u, v) ∈ A_α where Ω(u), Ω(v) are distinct outer vertices of S_α,
        with w'_α = Ω(u) do
10         CONTRACT(g)                                          // (Algorithm 3.8)
11 for each arc g = (u, v) ∈ E(G) in the edge stream do
12     if u ∈ R or v ∈ R or g ∈ M or vertex u or v does not belong to a structure then
13         continue to the next arc
14     Let S_α be the structure containing u, i.e. u ∈ V(G_α)
15     Let S_β be the structure containing v, i.e. v ∈ V(G_β)
16     if S_α ≠ S_β and Ω(u), Ω(v) are outer vertices of their respective structures then
17         AUGMENT(g, R, P)                                     // (Algorithm 3.7)
```

3.2.7 The BACKTRACK-STUCK-STRUCTURES Procedure

The BACKTRACK-STUCK-STRUCTURES procedure can be thought of as a similar process to the backtrack step within a depth-first search, which occurs when a node has no more neighbours to explore. In the case of the MMSS algorithm, we backtrack any structures which have not been able to make any progress during the current pass-bundle.

The procedure works as follows; we begin by taking the structure S_α for each free vertex $\alpha \in G$ as input for the procedure. We then iterate through each structure S_α, and if the structure is marked as used, we move to the next structure; this is because we will not be using the structure again during the current pass-bundle as we have already found an augmenting path which uses it. Otherwise, we call the BACKTRACK operation on the structure, described in Section 3.3.4, which backtracks the working vertex of the structure when appropriate. Due to the simplicity of this procedure, we have not provided pseudocode for it.

3.3 Free Vertex Structure Operations

Throughout the MMSS algorithm, we perform four fundamental operations on free vertex structures that modify the contents of structures in different ways, depending on the situation. In this section, we formally define each of these operations. We also provide detailed pseudocode for each, which was not provided in the original paper.

3.3.1 The Augment Operation

The Augment operation occurs when we have an unmatched arc connecting the outer vertices of two different structures. Given a structure S_α, we know that there is a unique even-length alternating path p'_α in T'_α from the root $\Omega(\alpha)$ to any outer vertex $\Omega(u)$. Because this must be an even-length path, we know the alternating path beginnings with an unmatched edge and ending with a matched edge. If we have a second even-length alternating path p'_β from the root to an outer vertex $\Omega(v)$ in a different structure S_β and there exists an unmatched arc from $\Omega(u)$ to $\Omega(v)$ then we have an augmenting path $p'_\alpha \circ \overleftarrow{p'_\beta}$ in G'. We can then expand this augmenting path to give us the corresponding augmenting path in G using Lemma 1.1. An example of this operation can be seen in Figure 3.5.

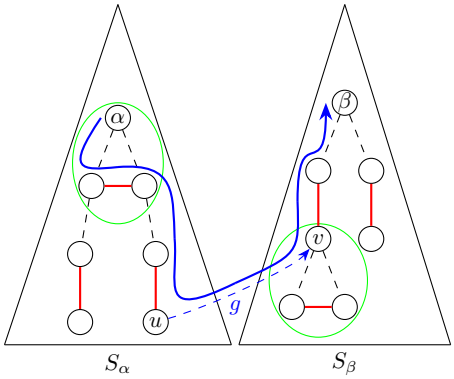


Figure 3.5: **An example of the Augment operation forming an augmenting path.** Dashed black and solid red lines denote unmatched and matched edges, respectively. Non-trivial blossoms are marked by a green ellipse. The dashed blue arrow represents the unmatched arc $g = (u, v)$ connecting the outer vertices of the free vertex structures S_α and S_β. Marked with a thick blue arrow is the augmenting path from α to β created during the Augment operation.

Pseudocode for this operation can be seen in Algorithm 3.7, where the augmentation occurs between Line 1 and 4. We ensure that the set of augmenting paths $\mathcal{P}$ remains vertex-disjoint upon completion of the procedure by marking S_α and S_β as used. As a result, both structures cannot be used in the Augment procedure, as they no longer satisfy the input conditions.

Algorithm 3.7: Augment

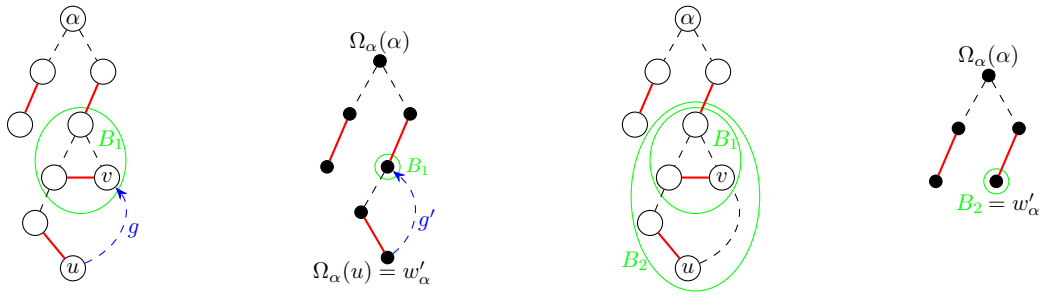
Input: An unmatched arc $g = (u, v)$, the current matching M and the set of vertex-disjoint augmenting paths $\mathcal{P}$ where:
- $g \in E(G)$
- u and v are outer vertices of structures S_α and S_β respectively, where $S_\alpha \neq S_\beta$
- S_α and S_β are marked as *not used*

1 $p'_\alpha \leftarrow$ the unique even-length alternating path from $\Omega(u)$ to $\Omega(\alpha)$ in T'_α
2 $p'_\beta \leftarrow$ the unique even-length alternating path from $\Omega(v)$ to $\Omega(\beta)$ in T'_β
3 $p' \leftarrow p'_\alpha \circ \overleftarrow{p'_\beta}$
4 $p \leftarrow$ the augmenting path in G corresponding to p'
5 $\mathcal{P} \leftarrow \mathcal{P} \cup p$
6 Mark S_α and S_β as *used*

3.3.2 The CONTRACT Operation

The CONTRACT operation occurs when we have an unmatched arc connecting two outer vertices of the same structure, which would form an odd-length cycle in the structure. We can contract this blossom into a single vertex.

Pseudocode for the operation can be seen in Algorithm 3.8. It works as follows; given an arc $g = (u, v)$, if it connects two distinct outer vertices that are in the same structure, S_α, then, by Lemma 2.2, adding the arc $(\Omega(u), \Omega(v))$ to T'_α would create a unique blossom B in the graph. In the pseudocode, we identify this blossom in Line 1 to 4. We then add the arc to the structure and contract B by adding it to Ω_α, which by definition updates T'_α to become T'_α/B. Once we have contracted B, we maintain that T'_α remains an alternating tree through Lemma 2.2. We then mark S_α as modified and update the working vertex of the structure w'_α to be B, since the previous working vertex has been contracted into B. An example of this operation can be seen in Figure 3.6.



(a) G_α and arc g before contraction

(b) T'_α and arc g' before contraction

(c) G_α after running the CONTRACT(g) operation

(d) T'_α after running the CONTRACT(g) operation

Figure 3.6: **A example of the CONTRACT operation.** Dashed black and solid red lines denote unmatched and matched edges, respectively. Non-trivial blossoms are marked with a green ellipse. **(a)** We see the subgraph G_α corresponding to the structure S_α. The dashed blue arrow represents an unmatched arc $g = (u, v)$ that is not in the structure, that is, $g \notin E(G_\alpha)$, which would create a blossom if added to S_α. **(b)** We see the contracted graph $T'_\alpha = G_\alpha/\Omega$ and the unmatched arc $g' = (\Omega(u), \Omega(v))$ shown by a dashed blue arrow. **(c)** We see the graph G_α after the CONTRACT operation has been run on the arc g. This forms a new blossom B_2 in the graph. **(d)** We see the contracted graph $T'_\alpha = G_\alpha/\Omega$ after the CONTRACT operation has been run on arc g, which created a new blossom B_2 and updated the working vertex of the structure.

Algorithm 3.8: CONTRACT

Input: An unmatched arc $g = (u, v)$, where:
- $g \in E(G)$
- u and v belong to the same structure, S_α
- $\Omega(u)$ and $\Omega(v)$ are distinct outer vertices in T'_α where $\Omega(u) = w_\alpha$

1 lca $\leftarrow$ the lowest common ancestor vertex of $\Omega(u)$ and $\Omega(v)$ in T'_α
2 $P'_u \leftarrow$ the unique path from $\Omega(u)$ to lca in T'_α
3 $P'_v \leftarrow$ the unique path from $\Omega(v)$ to lca in T'_α
4 $B \leftarrow$ the blossom in T'_α formed by the cycle $P'_u \circ P'_v$
5 $E(G_\alpha) \leftarrow E(G_\alpha) \cup \{g\}$
6 $\Omega_\alpha \leftarrow \Omega_\alpha \cup B$
7 $w'_\alpha \leftarrow B$
8 Mark S_α as modified

3.3.3 The OVERTAKE Operation

The OVERTAKE operation occurs when we can extend the active path of a structure S_α by adding a matched arc a to the structure. This happens when adding α to S_α would reduce its label value, ℓ_α. Specifically, this only occurs if $k < \ell(a)$, where k is defined as the value of the label of the matched arc

whose vertex is the current working vertex of the structure, plus one. Upon completion of the OVERTAKE operation, $\ell(a)$ is reduced to the value k.

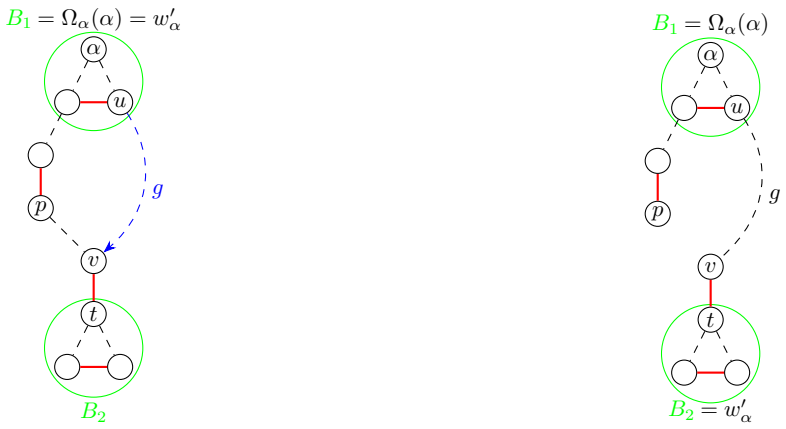
Pseudocode for the OVERTAKE operation can be seen in Algorithm 3.9. It takes an unmatched arc $g = (u, v)$ as input, where $\Omega(u) = w'_\alpha$, and a matched arc $a = (v, t)$, this means that the arcs g and a are connected by the vertex v. Finally, it takes as input an integer k, containing the label of the matched arc connected to w'_α plus one, which will be the new value of $\ell(a)$, where $k < \ell(a)$, after the operation.

We can break the operation into 3 separate cases, which one occurs depends on the structure to which the vertices of the arc a belong. These cases work as follows:

- **Case 1:** Arc a does not belong to any structure, that is, the vertices v and t are unvisited. This case is handled by Line 1 to 6. Here we simply add the arcs g and a to S_α, and their corresponding vertices to G_α, we also add each vertex to Ω_α as a trivial blossom. We then update the working vertex of the structure, w'_α to t, which is now an outer vertex in T'_α. Finally, we mark S_α as modified.

- **Case 2:** Arc a already belongs to structure S_α, that is, $u, v, t \in S_\alpha$. This case is handled by Line 7 to 12. From the conditions of this case, we know that both $g, a \in E(G_\alpha)$, and that $\Omega(v)$ is an inner vertex in T'_α and is therefore not the root. From the input requirements of the operation, we know that $\Omega(v)$ is not an ancestor of $\Omega(u)$ in T'_α. In this case, we update the parent of $\Omega(u)$ to $\Omega(v)$ in T'_α. This is done by finding the arc $(p, v) \in E(G_\beta)$ where $\Omega(p)$ is the parent of $\Omega(v)$ in T'_α, this arc is removed from $E(G_\alpha)$ and g is added instead, resulting in $\Omega(u)$ becoming the parent of $\Omega(v)$ in T'_α. Finally, we set w'_α as $\Omega(t)$ and mark S_α as modified. An example of this case can be seen in Figure 3.7.

- **Case 3:** Arc α belongs to a structure S_β, where $\alpha \neq \beta$. This case is handled by Line 13 to 25. This case is similar to Case 2, however, we also add each vertex in the subtree of v in T'_β from S_β to S_α. We start in a similar way, identifying the arc (p, v) where $\Omega(p)$ is the parent of $\Omega(v)$ in T'_β. We then remove (p, v) from G_β and add g to G_α. Then we move each vertex and arc in G_β to G_α if it is contained within the component of $\Omega(v)$ in G_β. We also move all the blossoms involved in this component from Ω_β to Ω_α. If the working vertex of S_β, w'_β, no longer exists in S_β, we set w'_α to w'_β and set w'_β to be $\Omega(p)$, otherwise we just set w'_α to be $\Omega(t)$. Finally, we mark both S_α and S_β as modified. An example of this case can be seen in Figure 3.8.



(a) Structure S_α and the unmatched arc g before the OVERTAKE(g) operation

(b) Structure S_α after the OVERTAKE(g) operation

Figure 3.7: **An example of Case 2 of the OVERTAKE operation.** Dashed black and solid red lines denote unmatched and matched edges, respectively. Non-trivial blossoms are marked in by a green ellipse. The unmatched arc $g = (u, v)$ taken as input to the CONTRACT operation is marked by a blue arrow. **(a)** We see the free vertex structure S_α and the unmatched arc g, which connects the working vertex of S_α and an inner vertex, before the overtake operation. **(b)** We see the free vertex structure S_α after the overtake operation. Prior to the overtake operation, the value of label of the arc (v, t) is 2, and after the value of the label is 1.

Algorithm 3.9: OVERTAKE

Input: An unmatched arc $g = (u, v)$, a non-blossom matched arc $a = (v, t)$ and a positive integer k, where:
- $g, a \in E(G)$ where g and a share an endpoint v
- $\Omega(u)$ is the working vertex of a structure, S_α
- $\Omega(u) \neq \Omega(v)$
- $\Omega(v)$ is an unvisited vertex or $\Omega(v)$ is an inner vertex of a structure, S_β, where S_β could be S_α
- If $S_\alpha = S_\beta$ then $\Omega(v)$ is not an ancestor of $\Omega(u)$
- $k < \ell(a)$

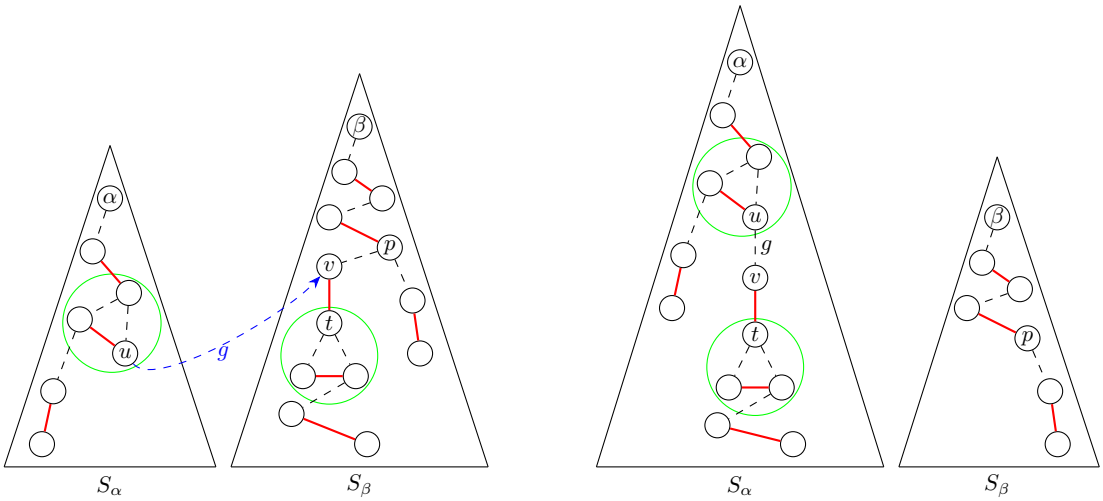
1 **if** *arc a does not belong to a structure* **then**
2 $V(G_\alpha) \leftarrow V(G_\alpha) \cup \{v, t\}$
3 $E(G_\alpha) \leftarrow E(G_\alpha) \cup \{g, a\}$
4 $\Omega_\alpha \leftarrow \Omega_\alpha \cup \{v, t\}$
5 $w_\alpha \leftarrow t$
6 Mark S_α as modified
7 **else if** *arc a belongs to S_α* **then**
8 Let p be the vertex in $V(G_\alpha)$ such that there exists an arc $(p, v) \in E(G_\alpha)$ and $\Omega(p)$ is the parent of $\Omega(v)$ in T'_α
9 $E(G_\alpha) \leftarrow E(G_\alpha) \setminus \{(p, v)\}$
10 $E(G_\alpha) \leftarrow E(G_\alpha) \cup \{g\}$
11 $w_\alpha \leftarrow \Omega(t)$
12 Mark S_α as modified
13 **else if** *arc a belongs to S_β where $S_\beta \neq S_\alpha$* **then**
14 Let p be the vertex in $V(G_\beta)$ such that there exists an arc $(p, v) \in E(G_\beta)$ and $\Omega(p)$ is the parent of $\Omega(v)$ in T'_β
15 $E_\beta \leftarrow E(G_\beta) \setminus \{(p, v)\}$
16 $E_\alpha \leftarrow E(G_\alpha) \cup \{(p, v)\}$
17 For each vertex $x \in V(G_\beta)$, if $\Omega(x)$ is in the subtree of $\Omega(v)$ in T'_β, move x from G_β to G_α
18 For each arc $(x, y) \in E(G_\beta)$, if after Line 17, $x, y \in V(G_\alpha)$, move (x, y) from G_β to G_α
19 For each blossom $B \in \Omega_\beta$, if after Line 17, B contains vertices in G_α, move B from Ω_β to Ω_α
20 **if** *$w'_\beta \notin G_\beta$ after the previous steps* **then**
21 $w'_\alpha \leftarrow w'_\beta$
22 $w'_\beta \leftarrow \Omega(p)$
23 **else**
24 $w'_\alpha \leftarrow \Omega(t)$
25 Mark S_α and S_β as modified

3.3.4 The BACKTRACK Operation

The BACKTRACK operation is used to check whether we have exhausted all the possible out-paths from the structure's working vertex. For a structure S_α, for the operation changes the working vertex of the structure, w'_α, to its closest ancestor vertex that is also an outer vertex.

Pseudocode for the BACKTRACK operation is provided in Algorithm 3.10. We begin by checking whether the structure is marked as *on hold* or *modified* in which case we end the operation immediately. Any structure marked as *on hold* means that we do not currently want to continue the depth-first search, and any structure marked as *modified* means that during the current pass-bundle we have modified the structure, by either extending the structure's active path or contracting a blossom within the structure. In these cases, we do not want to backtrack as there could be an edge to explore from the structure's working vertex. We also check whether the working vertex of the structure, w'_α, is set to $\emptyset$, if it is, then we have already exhausted the depth-first search of the free vertex α and therefore cannot backtrack any further, so we once again end the operation early. Otherwise, we know that there has not been any changes to the structure during the current pass-bundle and we need to backtrack.

Backtracking occurs between Line 3 and 6 and works as follows. For a structure S_α, we begin by checking whether the working vertex, w_α, is the root of the alternating tree, T_α, of the structure. If this is the case, and $w_\alpha = \Omega(\alpha)$, then we cannot backtrack further and the depth-first search has been



(a) Before the OVERTAKE(g) operation

(b) After the OVERTAKE(g) operation

Figure 3.8: **An example of Case 3 of the OVERTAKE operation.** Dashed black and solid red lines denote unmatched and matched edges, respectively. Non-trivial blossoms are marked with a green ellipse. The unmatched arc $g = (u, v)$ taken as input to the OVERTAKE operation is marked by a blue arrow. **(a)** We see the unmatched arc g connecting two structures S_α and S_β. **(b)** The unmatched arc g and everything in the subtree of v in structure S_β is moved to structure S_α.

completed, so we set w_α to $\emptyset$, indicating that the structure is inactive. Otherwise, we set w_α to the parent of its parent in T_α, which ensures that w_α remains an outer vertex.

Algorithm 3.10: BACKTRACK

Input: The structure S_α of a free vertex α

```
1 if S_α is marked "on hold" or S_α is marked "modified" or W'_α = ∅ then
2 |  end operation early
3 if W'_α = Ω(α) then
4 |  W'_α = ∅
5 else
6 |  W'_α ← parent of the parent of W'_α in T'_α
```

Chapter 4

Algorithm Implementation

4.1 Language Choice

We opted to implement our version of the MMSS algorithm using the C++ programming language for a couple of reasons. Firstly, the low-level nature of the language provides us with fine-grained control over features such as memory management, which high-level languages such as Java or Python do not offer. Since we are interested in both the number of passes required to compute the solution and also the performance metrics, such as runtime and memory usage, C++ is an ideal option. In addition, some of the procedures described within the MMSS algorithm required data structures such as dynamic arrays and sets, which are not supported as standard by the C programming language, which would have been another potential option. Finally, the C++ language is compatible with the Boost C++ library, an extensively used mathematics library. We used the library's implementation of non-streaming maximum matching algorithms, in particular their implementation of Edmond's Blossom algorithm, to give us optimal solution values during testing.

4.2 Graph Streaming

Throughout our implementation, we represent graphs using a `.txt` file where each line in the file represents a single edge in the graph. We represent vertices as positive integer values and edges as space-separated vertices; for example, the line `"2 5"` would represent an edge between vertices 2 and 5.

To emulate streaming a graph, we implemented a class called `StreamFromFile` that took input of the relevant graph file and would return an edge line by line from the file every time the `readStream()` function was called. Once every edge in the file has been returned, the next call to `readStream()` will return an impossible edge, `"-1 -1"`, alerting the program that it has read the entire graph. Subsequent calls will then continue outputting edges from the file, restarting from the beginning. A key point to note is that since our algorithm represents an undirected edge as two directed arcs, each edge essentially needs to be read twice; however, the graph file only needs to contain the edge once because this logic is managed by the `readStream()` function.

During our analysis, we mainly considered the number of passes of the edge stream that the algorithm requires rather than the algorithm's execution time, so we have also implemented a second class, `StreamFromMemory`. During initialisation, this class reads the entire graph file once, storing each edge in a vector. This means that instead of having to read the file each time the next edge is needed, the edge is loaded from memory, reducing the overhead required when retrieving the next edge.

It is worth noting that both of these methods only emulate the functionality of the streaming model, and in practise we would not want to be reading our graph from a text file. For this reason, we created a virtual class, `Stream`, from which both `StreamFromFile` and `StreamFromMemory` extend. Anyone wanting a different method of reading an edge stream can create their own class extending `Stream` and implement their own method for `readStream()`, which can then be used with the remaining code, allowing for custom stream readers to be designed. In Figure 4.1, a UML class diagram is provided that illustrates the relationship between these three classes.

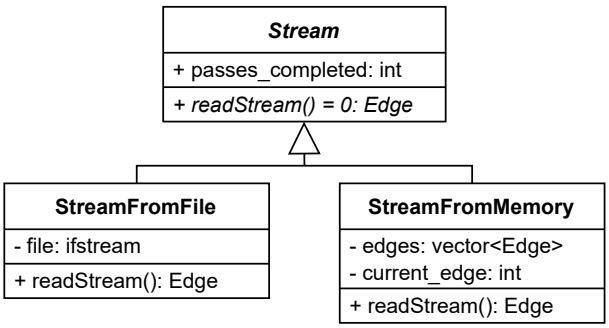


Figure 4.1: **A UML class diagram illustrating the relationship between the three classes used to emulate the streaming of a graph.**

4.3 Data Structures

Throughout the MMSS algorithm, we maintain various data structures, such as the current free vertex structures of the graph. These structures are implemented through purpose-built classes using object-orientated principles to ensure code clarity and maintainability.

In this chapter, we will use the term "vertex identifier" to describe the integer value of the vertex in the edge stream. In addition, we will use the term "node" as a general term to describe a non-trivial blossom or single vertex (trivial blossom) within a free vertex structure. Finally, given a blossom B, if B contains a second blossom B_2, we will describe B_2 as the "inner blossom" of B.

To improve the readability of our code, we also define aliases for long type definitions, which are listed below.

- `typedef std::pair<int, int> Edge` - this defines the type `Edge` as a pair of integers, where each integer represents the vertex identifier of a vertex. This definition can also be used to describe an arc, where the first vertex in the pair is the tail of the arc, and the second is the head.
- `typedef std::pair<std::vector<Edge>, std::vector<Edge>> AugmentingPath` - This type is used to represent an augmenting path in a graph. It consists of a pair of edge vectors where the first vector contains the edges in the augmenting path to match, and the second contains edges to unmatch.

A UML class diagram showing the classes used in our implementation and some of the key variables and functions contained within them is provided in Figure 4.2. Many of the classes also contain various helper functions, such as getter and setter methods used when interacting with maps, which are not shown in the diagram.

To improve debugging, overloads to the `<<` operator have also been implemented for each of our classes, which are described in the following sections. This allows relevant information about the content of an object to be outputted using `std::cout`.

4.3.1 The GraphNode and GraphVertex Classes

The `GraphNode` class is a virtual base class inherited by any class that is used to represent elements contained within the alternating tree of a free vertex structure; this includes the `GraphVertex` class, which is used to represent vertices and trivial blossoms, and the `GraphBlossom` class, Section 4.3.2, which is used to represent non-trivial blossoms. The class defines several fundamental variables that are used in its derived classes to represent these objects.

To emulate the alternating tree of each free vertex structure, created during the vertex's depth-first search, each `GraphNode` object is given `parent` and `children` variables, of types `GraphNode*` and `std::set<GraphNode*>`, respectively. These variables store pointers to the parents and children of a node, which allows us to simulate the structure of a tree. If a `GraphNode` does not have a parent, `parent` is set to `nullptr`.

If we had a non-trivial blossom B, with the current variables stored in the `GraphNode` object of B, it would not be possible to identify the specific vertex in B that connects it to its parent in the tree. For this reason, the `vertex_id` variable is introduced, which stores the identifier of the vertex in B involved in the edge connecting B and its parent. If a node is just a trivial blossom (a single vertex), `vertex_id`

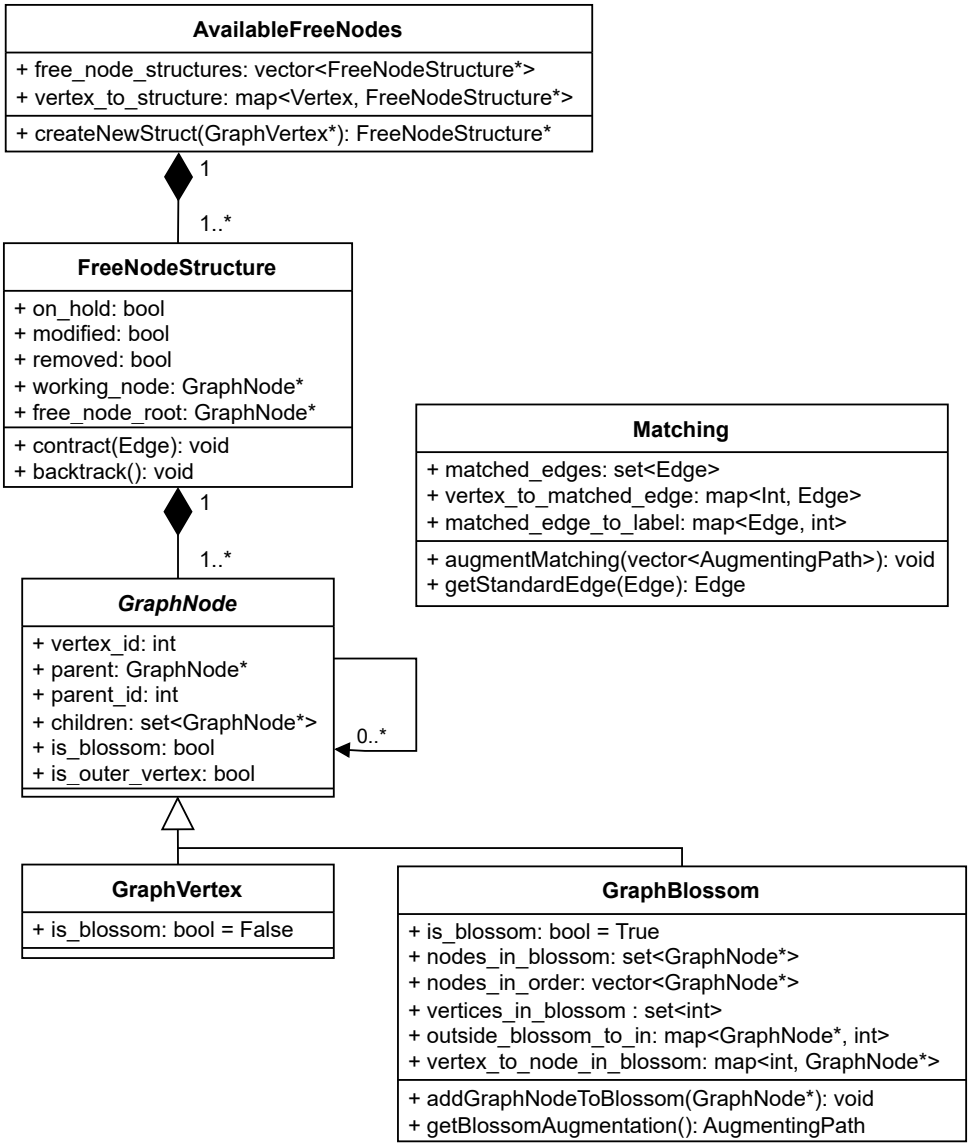


Figure 4.2: **A UML class diagram illustrating the relationship the classes used in our implementation of the MMSS algorithm.** The diagram only highlights the key functions provided by each class, with many also containing additional helper functions.

simply holds the identifier of that vertex. Similarly, if a node N has a non-trivial blossom B as its parent, it would not be possible to identify the vertex in B connected by an edge to N, so a `parent_id` variable is also stored, which holds the identifier of the vertex in B connected to N in the tree by an edge. If the parent is just a trivial blossom, `parent_id` holds the identifier of the single parent vertex, and if the node does not have a parent, an impossible vertex, `-1`, is stored instead. Finally, the `GraphNode` class stores two boolean values, `is_blossom` and `is_outer_vertex`, which allow us to identify whether a node is a blossom and whether a node is an outer vertex in its alternating tree. If a `GraphNode` is not a root blossom, i.e. the node is contained within another blossom, then `parent` is set to `nullptr`, `children` to `{}` and `is_outer_vertex` to `false`.

The `GraphVertex` class inherits the properties of `GraphNode` and is one of the simplest data structures, without any additional variables or functions other than those defined within `GraphNode`.

4.3.2 The GraphBlossom Class

The `GraphBlossom` class is used to represent a blossom structure within a graph; it inherits properties from the `GraphNode` class, Section 4.3.1, but also defines several others. These include the following variables:

- `std::set<GraphNode*> nodes_in_blossom` - This is a set containing a pointer to every `GraphNode` contained within the blossom's odd-length alternating cycle. If we have a `GraphBlossom` object describing a blossom B, within which is contained two blossoms B_1 and B_2, if B_2 is contained entirely within B_1, i.e. $B_1 \subseteq B_2$ so B_2 is an inner blossom of B_1, then only the pointer to B_1 is stored within B's `nodes_in_blossom` set, and B_2 is instead stored in B_1's `nodes_in_blossom` set.

- `std::vector<GraphNode*> nodes_in_order` - This is an array-like data type, which, similar to the `nodes_in_blossom` variable, contains each `GraphNode` in the odd-length alternating cycle of the blossom. However, the array-like structure is used to store the order with which they are seen within the blossom's cycle. Like `nodes_in_blossom`, it does not store the contents of any inner blossoms.

- `std::set<int> vertices_in_blossom` - This is a variable containing the vertex identifier of every vertex contained within the blossom and any respective inner-blossoms. We use this set throughout our implementation when iterating through each vertex contained within a blossom. Although this could be replaced by iterating through the keys of the `vertex_to_node_in_blossom` dictionary, described below, iterating through a set provides clearer and more readable syntax.

- `std::unordered_map<GraphNode*, Int> outside_blossom_to_in` - This is a dictionary data structure which is used to identify the edge in a connecting a blossom to other nodes in the structure. If there exists a node in the structure connected by an edge to the blossom, then `outside_blossom_to_in` stores the pointer to the node and the vertex identifier of the vertex in the blossom to which it is connected as a key-value pair.

- `std::unordered_map<int, GraphNode*> vertex_to_node_in_blossom` - This is another dictionary data structure used to identify the node within a blossom that contains a specified vertex in $O(1)$ time. It maps vertex identifiers in `vertices_in_blossom` to pointers to a `GraphNode` in `node_in_blossom`. An example of this variable's usage is during the AUGMENT operation where, in combination with `outside_blossom_to_in`, it identifies the vertex with which we enter and exit a blossom in an augmenting path and the `GraphNode` the vertex is contained within.

Although it would be possible to implement the `GraphBlossom` class with only the `nodes_in_order` variable and not `nodes_in_blossom`, each variable has a different purpose. The `nodes_in_blossom` variable is a set, which allows us to check whether a node is contained within the blossom in $O(1)$ time, whereas `nodes_in_order` is a list and therefore would take $O(|\texttt{nodes\_in\_order}|)$ time to complete the same task. In addition, given a node in a blossom, `nodes_in_order` allows us to find which nodes it is connected to in the blossom, but this is not possible with `nodes_in_blossom`.

A visual example of the contents of a `GraphBlossom` object is shown in Figure 4.3, with further examples available in Figure 4.4c, which provide more context on usage within a free vertex structure.

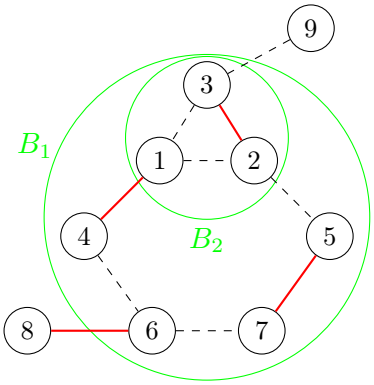
The `GraphBlossom` class also contains various functions that implement the logic of the blossom structure, we will now highlight some of the key functions.

- `void addGraphNodeToBlossom(GraphNode* node)` - This function is used during the contraction process when a new blossom is created and is called on each of the nodes in the new blossom's odd-length alternating cycle. The function takes input of the node to be added to the blossom and then updates all of the relevant variables of the blossom object to reflect the node's insertion. Finally, it updates the variables stored by the original node's parent and child objects to now point to the new blossom, as the node is now contained with it.

- `AugmentingPath getBlossomAugmentation(int in_id, int out_id, ...)` - This function finds the even-length alternating path between two vertices in a blossom, which can then be used as part of an augmenting path. It takes an input `in_id` and `out_id`, the vertex identifiers of the vertices in the blossom we would like to traverse between. Given a blossom B, and in and out vertices of x, y, the function works as follows:

  1. Identify the `GraphNode` objects that x and y belong to using the `vertex_to_node_in_blossom` variable of B.
  2. Find the position of each of these nodes in the `nodes_in_order` variable of B. Let these be i and j.

3. Find the even-length path of the nodes between position i and j in the `nodes_in_order` variable of B, either traversing backward or forward from i. We know that this path exists by Lemma 2.1.
4. If a node on this path is a blossom, B_2, we need to call the function recursively on B_2. We can identify the `in_id` and `out_id` of B_2, using B_2's `outside_blossom_to_in` variable, looking up the nodes before and after B_2 in the path in B.



(a) A graph containing blossoms B_1 and B_2

vertex_id	3
parent	V_9
parent_id	9
children	$\{V_8\}$
is_outer_vertex	true
is_blossom	true
nodes_in_blossom	$\{V_4, V_7, V_6, V_5, B_2\}$
nodes_in_order	$[V_4, B_2, V_5, V_7, V_6]$
vertices_in_blossom	$\{1, 2, 3, 4, 5, 6, 7\}$
vertex_to_node_in_blossom	$\{1 \to B_2,\ 2 \to B_2,$ $3 \to B_2,\ 4 \to V_4,$ $5 \to V_5,\ 6 \to V_6,$ $7 \to V_7\}$
outside_blossom_to_in	$\{V_8 \to 6,\ V_9 \to 3\}$

(b) Contents of the `GraphBlossom` object describing B_1

Figure 4.3: **An example of the contents of a `GraphBlossom` object. (a)** A graph containing two blossoms B_1 and B_2, where B_2 is completely contained within B_1. We use dashed black and solid red lines to denote unmatched and matched edges, respectively. Non-trivial blossoms are marked with a green ellipse. **(b)** The contents of the variables within a `GraphBlossom` object describing the blossom B_1 shown in (a). The first six variables are inherited from the `GraphNode` class. We use V_i to represent the pointer to the `GraphVertex` corresponding to vertex i and B_2 to represent the pointer to the `GraphBlossom` of blossom B_2.

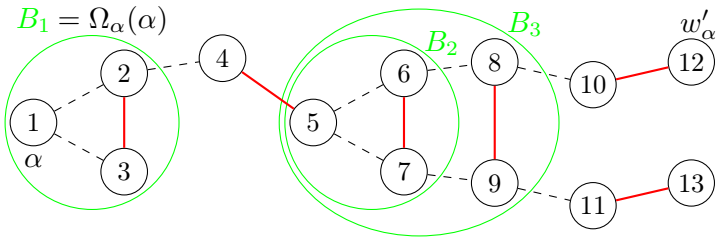
4.3.3 The FreeNodeStructure Class

The `FreeNodeStructure` class contains relevant information about a free vertex structure and its alternating tree. The class variables provide a contracted view of the tree's contents; however, the original view, where all the blossoms have been expanded, can be viewed by accessing the contents of `GraphBlossom` nodes within the tree. The class consists of the following variables:

- `bool on_hold, bool modified, bool used` - These three boolean variables are used to store whether the structure has been marked as on hold, modified or used during the current pass-bundle, as discussed in Section 3.2.4.
- `GraphNode* working_node, GraphNode* free_node_root` - These variables stores pointers to the working node and root of the structure. Since each `GraphNode` contains a pointer to its parent and children with the structure, using these two nodes, the algorithm is able to traverse the entire alternating tree of the free vertex structure.
- `std::unordered_map<int, GraphNode*> vertex_to_node` - This dictionary data structure stores a vertex identifier as the key and a pointer to the root blossom `GraphNode` which it belongs to as the value. This is frequently used when processing an arc from the edge stream to identify the `GraphNode` corresponding to the vertex's identifier. Once we know which `FreeNodeStructure` a vertex belongs to, we use this variable lookup the vertex's corresponding `GraphNode`.

Figure 4.4 shows an example of the contents of a `FreeNodeStructure`. A key point to note is that non-root blossoms (both trivial and non-trivial), such as V_1, V_6, B_2, do not store a information about their parent or children, as explained in the description of the `GraphNode` class.

The `FreeNodeStructure` class also provides the logic for the `Contract` and `Backtrack` operations, using the functions `void contract(Edge)` and `void backtrack()`, respectively. The `contract()` function



(a) An example of a free vertex structure, S_α, as defined in Definition 3.1

	$\boldsymbol{V_x}$ where $x \in \{1,2,3,5,6,7,8,9\}$	$\boldsymbol{V_4}$	$\boldsymbol{V_{10}}$	$\boldsymbol{V_{11}}$	$\boldsymbol{V_{12}}$	$\boldsymbol{V_{13}}$
`vertex_id`	x	4	10	11	12	13
`parent`	null	B_1	B_3	B_3	V_{10}	V_{11}
`parent_id`	-1	2	8	9	10	11
`children`	$\{\}$	$\{B_3\}$	$\{V_{12}\}$	$\{V_{13}\}$	$\{\}$	$\{\}$
`is_blossom`	false	false	false	false	false	false
`is_outer_vertex`	false	false	false	false	true	true

(b) The contents of each `GraphVertex` object used in the representation of S_α

	$\boldsymbol{B_1}$	$\boldsymbol{B_2}$	$\boldsymbol{B_3}$
`vertex_id`	1	5	5
`parent`	null	null	V_4
`parent_id`	-1	-1	4
`children`	$\{4\}$	$\{\}$	$\{V_{10}, V_{11}\}$
`is_blossom`	true	true	true
`is_outer_vertex`	true	false	true
`nodes_in_blossom`	$\{V_1, V_2, V_3\}$	$\{V_5, V_6, V_7\}$	$\{V_8, V_9, B_2\}$
`nodes_in_order`	$[V_1, V_2, V_3]$	$[V_5, V_6, V_7]$	$[B_2, V_8, V_9]$
`vertices_in_blossom`	$\{1,2,3\}$	$\{5,6,7\}$	$\{5,6,7,8,9\}$
`vertex_to_node_in_blossom`	$\{1 \to V_1,\ 2 \to V_2,\ 3 \to V_3\}$	$\{5 \to V_5,\ 6 \to V_6,\ 7 \to V_7\}$	$\{5 \to B_2,\ 6 \to B_2,\ 7 \to B_2,\ 8 \to V_8,\ 9 \to V_9\}$
`outside_blossom_to_in`	$\{V_4 \to 2\}$	$\{V_4 \to 5,\ V_8 \to 6,\ V_9 \to 7\}$	$\{V_4 \to 5,\ V_{10} \to 8,\ V_{11} \to 9\}$

(c) The contents of each `GraphBlossom` object used in the representation of S_α

	$\boldsymbol{S_\alpha}$
`on_hold`	false
`modified`	false
`used`	false
`working_node`	V_{12}
`free_node_root`	B_1
`vertex_to_node`	$\{1 \to B_1,\ 2 \to B_1,\ 3 \to B_1,\ 4 \to V_4,\ 5 \to B_3,\ 6 \to B_3,\ 7 \to B_3,\ 8 \to B_3,\ 9 \to B_3,\ 10 \to V_{10},\ 11 \to V_{11},\ 12 \to V_{12},\ 13 \to V_{13}\}$

(d) The contents of a `FreeNodeStructure` object representing S_α.

Figure 4.4: **An example of a `FreeNodeStructure` object and the `GraphNode` objects contained within. (a)** We see a free vertex structure S_α, where α is the vertex marked 1. In this figure, dashed black and solid read lines are used to denote unmatched and matched edges, respectively. Non-trivial blossoms are marked with a green ellipse. **(b)** We see the contents of each `GraphVertex` object. Each `GraphVertex` is used to represent a trivial blossom in the structure shown in (a). **(c)** We see the contents of each `GraphBlossom` object. Each `GraphBlossom` represents a non-trivial blossom in the structure shown in (a). `(d)` We see the `FreeNodeStructure` object representing the structure shown in (a).

works by creating a new `GraphBlossom` object, and inserting all the `GraphNode` objects that are involved in the blossom into the new object. This new `GraphBlossom` object is then inserted into the correct position within the `FreeNodeStructure`'s alternating tree, updating the `parent` and `child` variables of

surrounding nodes to ensure the remaining structure is preserved.

4.3.4 The AvailableFreeNodes Class

The `AvailableFreeNodes` class is used to keep track of all of the `FreeNodeStructure` objects that are currently in use. It stores this information in the following variables:

- `std::vector<FreeNodeStructure*> free_node_structures` - This variable maintains a list of pointers to every in-use `FreeNodeStructure` object. Since the memory space of each `FreeNodeStructure` object is dynamically allocated, this list helps to ensure correct memory management. At the end of a phase, once the structures are no longer needed, the algorithm can then iterate through the list and deallocate the memory of each structure, preventing memory leaks.
- `<std::unordered_map<int, FreeNodeStructure*>> vertex\_to\_structure` - This dictionary-like data structure allows us to identify which `FreeNodeStructure` object a vertex belongs to, in $O(1)$ time. It stores a vertex identifier and a pointer to the `FreeNodeStructure` the corresponding vertex belongs to as key-value pairs. Without this variable, we would have to iterate through each structure to check whether it contains the specified vertex.

The class also provides functions which are used to create new `FreeNodeStructure` objects and also modify their contents, for example, the class contains functions used during the `Overtake` procedure to move the contents of a subtree of one structure to another.

4.3.5 The Matching Class

In the `Matching` class, all the relevant information about the graph's current matching is stored. This includes the value of edge labels. To do this, the following variables are used:

- `std::set<Edge> matched_edges` - This variable stores a set of all the edges currently involved in a matching. The `set` data type is used over alternatives like `vector` as it gives $O(1)$ insert, lookup, and delete operations, which allows edges to be efficiently matched and unmatched during augmentation.
- `std::unordered_map<int, Edge> vertex_to_matched\_edge` - Through the algorithm, it needs to be possible to identify the matched edge in the graph with which a given vertex is involved, such as during the OVERTAKE operation. Although it would be possible to achieve the same goal without this variable by iterating through each edge in `matched_edge`, using a dictionary data structure allows us to find the corresponding edge in $O(1)$ time.
- `std::unordered_map<Edge, int, boost::hash<Edge>> matched_edge_to_label` - This dictionary variable stores the label of each matched edge in the graph, allowing us to quickly retrieve an edge's label. In standard C++, hashing an `Edge` or any similar tuple-like structure is not directly supported, instead the Boost Hashing library is used, which provides this functionality. This allows us to use an `Edge` as the key of the dictionary.

As mentioned in Section 3.1.1 and 3.1.3, when identifying whether an edge is matched, or obtaining its label value, we often use arcs, where both arcs corresponding to an edge have the same value. To ensure that both arcs of an edge are not stored in each of the variables, which would be space-inefficient and could lead to inconsistencies, before accessing any variables, the `std::pair` representing the arc is first converted to a standard form, where `pair.first` < `pair.second`. This is handled by the `Edge getStandardEdge(Edge edge)` function of the `Matching` class.

The `Matching` class also provides various other functions used for augmenting a matching and updating edge labels. It is important to note that when the matching is augmented, all of the variables within the `Matching` class are modified, not just the `matched_edge` variable. This ensures that each variable in the class reflects the information about the current matching.

4.4 Further Algorithm Considerations

The remainder of the algorithm logic is held in the `MMSS-Maximum-Matching.cpp` file. It contains the various procedures defined in the algorithm implementation, including GREEDY-APPROX, ALG-PHASE,

Extend-Active-Path, Contract-and-Augment, Backtrack-Stuck-Structures in addition to the Augment operation. Although many of these functions could have been combined into our previously defined classes, they are kept separate to minimise interdependencies between classes and instead use smaller helper functions provided by the classes; for example, the `Matching` class does not reference any of the graph structure classes, such as `FreeNodeStructure`, `GraphNode`, which ensures that it operates independently. Likewise, the graph structure classes do not rely on `Matching`. The same principles also apply to the streaming classes. This design choice ensures that classes can be modified independently of one another, which would allow us to easily change parts of our implementation if there were improvements to the algorithm. For this reason, the majority of the Overtake operation is contained in this file, as it requires access to both the matched edge labels, stored in the `Matching}` class, and information about the free vertex structures.

In our implementation of the Alg-Phase procedure, we also introduce some changes to the original pseudocode described in Algorithm 3.4. Specifically, during Line 3 of the pseudocode, every free vertex α in the graph is found and its corresponding structure S_α is initialised. This would require the algorithm to make a pass over the edge stream, where it would check if any of the vertices it sees are unmatched and free. In our implementation, we remove this pass by moving the search for and initialisation of `FreeNodeStructure` objects to inside of our implementation of the Extend-Active-Path procedure, Algorithm 3.5. This works as follows, during our pass over the edge stream in Extend-Active-Path, every time an new arc (u, v) is seen, we begin by checking whether u and v are free vertices, this is done by checking whether either of the vertices are contained within the `vertex_to_matched_edge` dictionary of our `Matching`. If either of these vertices are free nodes, we then check whether they already belong to a structure, which can be done by checking whether the vertex is contained within our `AvailableFreeNodes` object's `vertex\_to\_structure` dictionary. If the vertex does not belong to a structure, we then initialise a new one. Once this has been completed, we continue with the logic in the Extend-Active-Path procedure.

Chapter 5

Critical Evaluation

5.1 Datasets

During the analysis of our implementation, we used various open-source, undirected graphs from the SNAP dataset library [LK14]. Many of these graphs followed different formats, so we reformatted each of them to the standard format outlined in Section 4.2, where each line consists of two space-separated integers that describe an edge in the graph. The standardisation process involved the following steps:

- The graphs used various file formats, such as `.tsv` (tab-separated values file) and `.csv` (comma-separated values file). To ensure consistency, all files were converted to the `.txt` format and the delimiter between the vertices of an edge was changed to a space.
- In some graphs, the vertices were represented as non-contiguous integers, resulting in graphs with relatively few edges having a larger file size than graphs with more edges. This was standardised by coordinate compression, where the original vertices values of an n-vertex graph are mapped to values in the range $[1, n]$, whilst preserving the relative ordering. This ensured a consistent file size between the graphs.
- The order of the edges within the files varied from graph to graph, with many in lexicographical order, where edges are sorted in ascending order based on their first vertex value or by their second value if the first values were identical. We sorted the remaining files to match this order, although we also investigated alternative orders later in this chapter.

An example of a graph file before and after standardisation is shown in Figure 5.1.

Line	Contents
0	3,4
1	1,3
2	1,2
3	2,4
⋮	⋮

(a) Before standardisation

Line	Contents
0	1 2
1	1 3
2	2 4
3	3 4
⋮	⋮

(b) After standardisation

Figure 5.1: **An example of the standardisation of a graph file from a `.csv` file to the format used to describe graphs in our implementation.**

Table 5.1 shows a list of the graphs we used during the evaluation of our implementation of the MMSS algorithm, in addition to some key information on the structure of the graphs. The file size of each of these graphs was calculated after the data standardisation process. Each of these graphs was obtained from the SNAP dataset library [LK14], which contains large graph datasets depicting various real-world graphs. For testing, we selected a range of graphs with varying sizes and densities, allowing us to make comparisons between different structural properties. The chosen graphs represent a range of different networks, such as social networks and academic collaboration networks, ensuring that we test the performance of the algorithm on a wide range of real-world scenarios. In addition, we chose graphs where the majority ($\geq 99.5\%$) of edges belonged to a single weakly connected component (WCC). This ensured that the algorithm was tested on large graphs, rather than multiple trivially small components.

Dataset Name	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert\text{Opt}\rvert$	File Size (KB)
feather-lastfm-social [RS20]	7624	27806	3347	270
ego-Facebook [LM12]	4039	88234	1979	854
email-Enron [LLDM08]	36692	183831	12198	1840
ca-AstroPh [LKF07]	18772	198110	9150	2146

Table 5.1: **A table showing relevant information about the graphs from SNAP dataset library [LK14] used to test our implementation, ordered by ascending edge count.**

5.2 Approximation Analysis

5.2.1 Worst Case Approximation

From [MMSS25, Lemma 6.5], we know that at the end of a scale h, the current matching M is a $(1+4h\ell_{\max})\cdot(1+\frac{1}{\ell_{\max}})$-approximate maximum matching, where $\ell_{\max}=3/\epsilon$. This provides us with a worst-case bound on the size of the matching upon completion of a scale h:

$$\text{After scale } h\text{: } |M| \geq \frac{|\text{Opt}|}{(1+4h\ell_{\max})\cdot(1+1/\ell_{\max})}$$

That said, for small values of h, the bound is not tight, which limits its usefulness for analysis. For instance, when $\epsilon=0.5$, after the first scale, $h=\frac{1}{2}$, the bound gives that $|M|\geq\frac{6}{91}\cdot|\text{Opt}|$, however, our initial greedy approximation already guarantees $|M|\geq\frac{1}{2}\cdot|\text{Opt}|$. As h increases to sufficiently large values, the analysis becomes asymptotically tighter, eventually providing us with the $(1+\epsilon)$-approximation.

5.2.2 Approximation Observations

In Figure 5.2, we see the change in matching size of various graphs during the execution of the algorithm. Even with an approximation factor of $\epsilon=0.75$, on all graphs tested, we reached the exact maximum matching early in the execution. This showed that the MMSS algorithm performs significantly better on real-life graphs than its theoretical worst case. It is also clear that the initial greedy matching, which is the first increase in the figure, performs significantly better than its 2-approximation worst case.

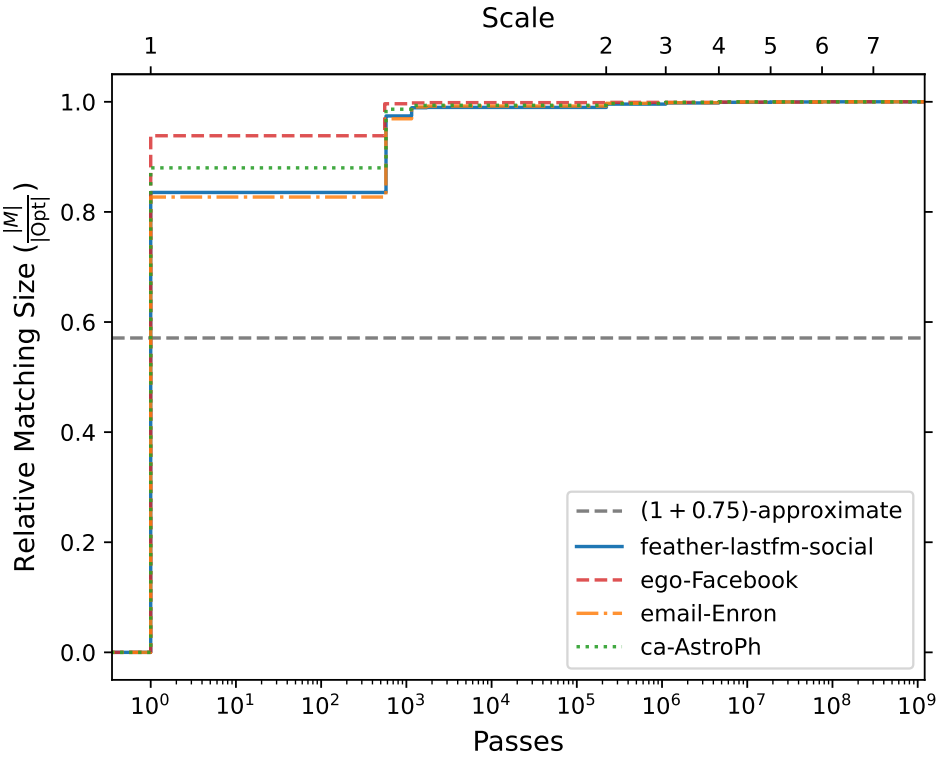


Figure 5.2: **A graph showing the increase in the size of the matching as the algorithm progresses.** The bottom x-axis shows the number of passes completed and the upper x-axis shows the current scale of the algorithm. The scale values shown correspond to $\frac{1}{2^x}$, where x is the value of the scale given. An approximation factor of $\epsilon=0.75$ is used, meaning there are seven scales in total.

Figure 5.3 is a zoomed in portion of Figure 5.2, where the first scale of the algorithm is shown, although similar results can be seen during other scales. From this figure, we can see that the matching size increases during the first few phases of the scale but does not increase during the later ones; this observation can also be seen in Table 5.2. This means that during the majority of the phases (phases 5 to 384 in this example), we made passes over the edge stream, but found no augmenting paths. However, we know that these augmenting paths still exist as they are found during the next scale, indicating that the lack of progress during the later phases is instead caused by the limitations imposed by the current scale. These limitations are the maximum size of a free vertex structure, $\text{limit}_h = \frac{6}{h} + 1$, and the number of pass-bundles in a phase, $\tau_{\max}(h) = \frac{72}{h\epsilon}$. The exact limitation is determined shortly.

Within the algorithm, each phase of a given scale performs the exact same operations, this means that if no augmenting paths are found in one phase, the same will happen in subsequent phases of the scale. This provides the basis for one of the optimisations implemented, which ends a scale early if no progress is being made. This optimisation is discussed further in Section 5.3.3.

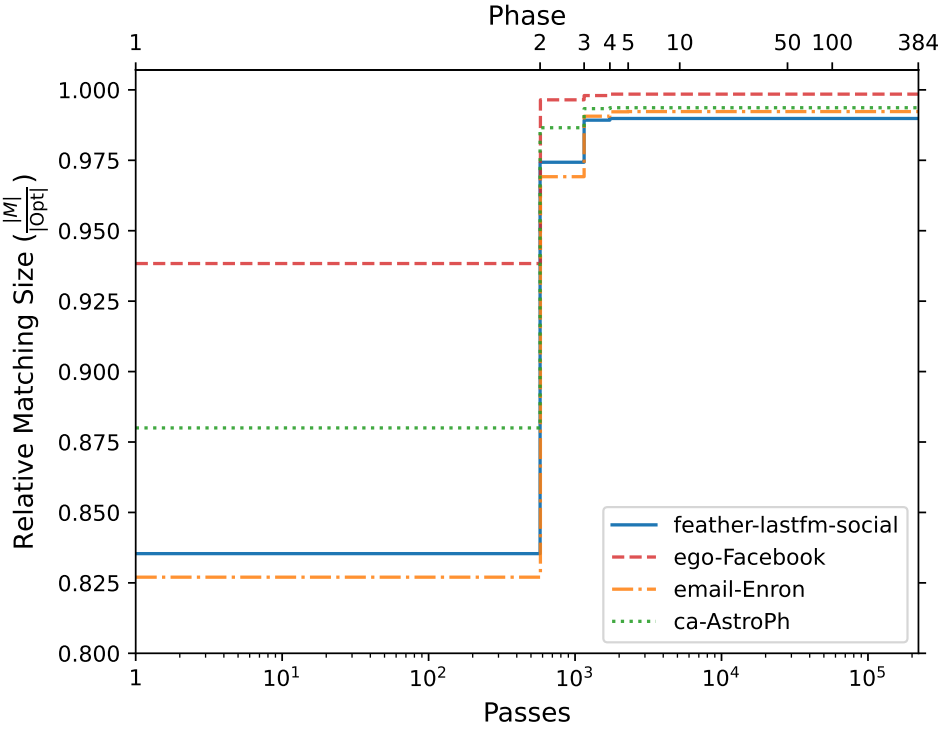


Figure 5.3: **A magnified section of Figure 5.2, showing the change in matching size during the first scale of the algorithm.** The bottom x-axis shows the number of passes completed and the upper x-axis shows the current phase the algorithm is in. An approximation factor of $\epsilon = 0.75$ is used, meaning the first scale consists of 384 phases, each using 192 pass bundles, totalling 221184 passes.

In the MMSS algorithm, the matching size remains constant during each phase and only increases at the end of a phase, during the AUGMENT-MATCHING procedure. Therefore, to understand how progress is made during a phase, we instead look at when AUGMENT operations occur during the phase, which indicate when an augmenting path is found. This can be seen in Figure 5.4, where we took a closer look at the first phase (of the first scale) of the algorithm for each graph. It shows a similar trend as that seen when looking at a single scale, with the size of the matching increasing during the initial pass bundles, but no change during later pass-bundles. There are two possible reasons for this lack of progress, the first is that the depth-first search has been completed for each free vertex at the current scale limit, meaning that every free vertex structure is either marked as *on hold* and therefore cannot be extended, or has been fully explored and is marked as *inactive*. The second possible reason is that the depth-first search is continuing until the end of the phase, but nothing is found.

We can identify which of these reasons is the case by looking at the total number of free vertex operations occurring during each pass-bundle. If there are no operations occurring, then the searches must have concluded or are no longer able to advance due to the scale's size limit. These results are presented in Figure 5.5, which shows that no operations are completed after pass-bundle 14. Similarly to our scale optimisation, each pass bundle within a given phase performs the exact same operations,

feather-lastfm-social		
Scale	Phase	Matching Size
Greedy	-	2796
1	1/384	3261
1	2/384	3311
1	3/384	3313
2	1/768	3332
2	2/768	3333
3	1/1536	3339
3	2/1536	3340
4	1/3072	3344
5	1/6144	3346
6	1/12288	3347

ca-AstroPh		
Scale	Phase	Matching Size
Greedy	-	8052
1	1/384	9027
1	2/384	9089
1	3/384	9092
2	1/768	9121
2	2/768	9122
3	1/1536	9139
3	2/1536	9141
3	3/1536	9142
4	1/3072	9147
5	1/6144	9149
6	1/12288	9150

Table 5.2: **A table showing the phases and scales of the algorithm during which the matching size changes.** The algorithm is run on each graph with an approximation factor of $\epsilon = 0.75$. The scale values shown correspond to $\frac{1}{2^x}$, where x is the value of the scale given. Each row in the table corresponds to a scale and phase of the algorithm where the matching size increased during the phase's AUGMENT-MATCHING procedure. In this table the results for the feather-lastfm-social and ca-AstroPh graphs are presented, whose maximum matchings are of size 3347 and 9150, respectively. Results for the remaining graphs can be found in Table C.1.

which means that if no operations occur during one pass bundle, the same will happen in subsequent pass bundles, as there has been no change to the graph or matching. This provides us with the basis for another optimisation that we have made to the algorithm, which ends a phase early if no operations occur during a pass bundle. More information about this optimisation is provided in Section 5.3.2, and there is further discussion about the operations completed during the execution of the algorithm in Section 5.7. Similar results can be found when looking at other phases within the algorithm.

When running the algorithm with smaller approximation factors, we obtained the same results as with $\epsilon = 0.75$, as the maximum matching for each graph had already been reached. We also observed that the specific phases during which the matching size increased, shown in Table 5.2, were identical. However, as per the algorithm's design, smaller approximation factors lead to more scales being computed, with each scale containing more phases and each phase containing more pass-bundles, resulting in more passes being used. These extra passes did not yield any improvements as each phase was already bound by the maximum size of a free node structure, limit_h, which was set independently of ϵ. Detailed results for these tests are given in Figure C.1 and Table C.2.

5.2.3 Early Termination Check

Currently, our algorithm continues its execution until completion, even if it has already met the required approximation factor. When our goal is simply to meet the approximation factor rather than surpass it, this continuation in the algorithm results in unneeded passes. To stop this, we implemented an optional early termination check that ends the algorithm early if we know that the required approximation has been reached.

The check works as follows; if, after any phase, the condition $|M| \geq \frac{1}{1+\epsilon} \cdot (|M| + \frac{1}{2} \cdot \text{num. of free vertices})$ holds, then execution can be ended early. This is valid because the size of the matching is constrained by the number of free vertices (every augmenting path increases the matching size by one and reduces the number of free vertices by two), giving us an upper bound on the size of a maximum matching. It is worth noting that because we can only calculate an upper bound on the size of the maximum matching, it is possible that even once the approximation has been met, this check may not trigger, resulting in the algorithm continuing past the required matching size. In our implementation of the algorithm, this check is disabled by default, but can be enabled by passing the `early_finish=true` parameter when calling the algorithm.

The results gathered when testing the early termination check are shown in Table 5.3. From these results, we can see that the effectiveness of the optimisation was dependent on the specific graph used, but, in general, for larger values of ϵ, the algorithm terminated earlier in its execution, during the first phase in some cases. This significantly reduced the number of passes used.

However, for smaller values of ϵ, such as $\epsilon = 0.1$, the check appeared to have no effect, and the

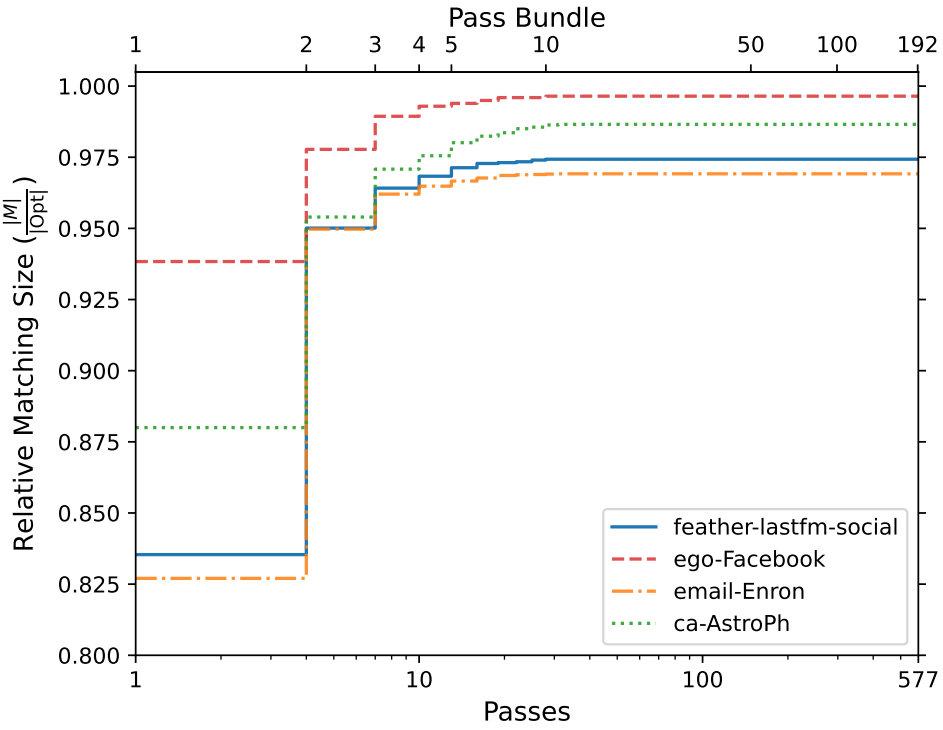


Figure 5.4: **A magnified section of the graph in Figure 5.3, showing the change in matching size during phase one of the first scale of the algorithm for various graphs.** The bottom x-axis shows the number of passes completed and the upper x-axis shows the current pass-bundle. An approximation factor of $\epsilon = 0.75$ is used, meaning that each phase in the first scale consists of 192 pass bundles, which is 576 passes in total. In the algorithm, the size of the matching is increased during the AUGMENT-MATCHING procedure at the end of each phase, however in this graph, we plot the point within the phase at which the augmenting path is found by the AUGMENT operation.

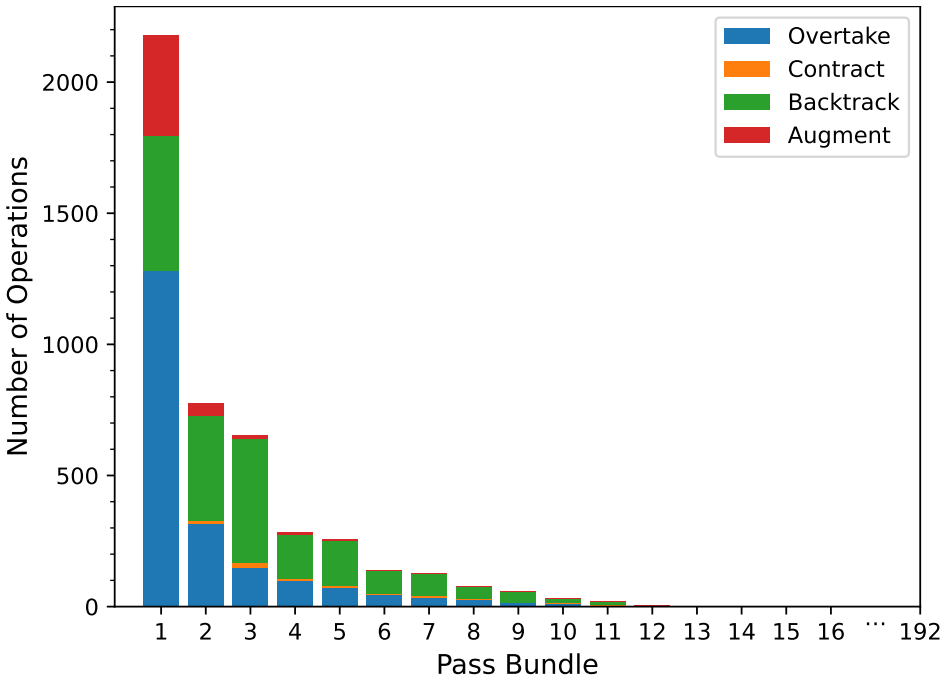


Figure 5.5: **A graph showing the number of free vertex structure operations that occur during each pass bundle of the first phase of scale one, using the feather-lastfm-social graph, with $\epsilon = 0.75$.** During pass bundles 17 to 191, which are not show in the figure, no operations are completed. The tabular form of this data can be found in Table C.3.

algorithm continued to run to completeness, even if it had found a maximum matching. To understand why this happened, we can take a closer look at the values used by the check. Taking the feather-lastfm-social graph as an example, after the GREEDY-APPROX procedure, the algorithm had a matching of size 2796 and 2032 free vertices, giving an upper bound of 3812 on the maximum matching size. This upper bound remained unchanged throughout the execution of the algorithm. The true size of the graph's maximum matching is 3347, which only corresponds to an $(1 + 0.138)$-approximate of the upper bound.

$$\frac{1}{1+\epsilon} \cdot 3812 = 3347, \ \therefore \epsilon = 0.138\ldots$$

This means that for any $\epsilon < 0.138$, even if the algorithm has already found an exact maximum matching, the early finish condition is not satisfied, and so the algorithm runs to competition, which is what was observed in our results. An example demonstrating why we only find an upper bound is shown in Figure 5.6.

This also explains the vast difference in performance of the optimisation on different graphs, where the algorithm has to run until completion for any $\epsilon \leq 0.5$ on the email-Enron graph. This is because email-Enron has a significantly larger upper bound compared to its maximum matching size. Consequently, the optimisation is very dependent on the structure of the individual graph being tested.

Graph Name	ϵ	Final Scale	Final Phase of Scale
feather-lastfm-social	0.75	1	1/384
	0.5	1	1/576
	0.25	1	1/1152
	0.15	2	1/3840
	0.1*	13	11796480/11796480
email-Enron	0.75	1	1/192
	0.5*	8	73728/73728
	0.25*	10	589824/589824
	0.15*	12	3932160/3932160
	0.1*	13	11796480/11796480

Table 5.3: **A table showing the point at which our implementation finishes when the early finish condition was enabled.** The table shows the scale and phase after which the implementation terminated for different values of ϵ when run with the early finish condition enabled. The ϵ values are marked with an * if the algorithm runs to completion. Results for the remaining graphs can be found in Table C.4.

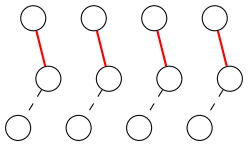


Figure 5.6: **An example showing why the early termination check may not always work.** Dashed black and solid red lines denote unmatched and matched edges, respectively. The graph shown has a maximum matching of size four, which can be seen marked. However, in this maximum matching, there are still four free vertices, which results in the upper bound used by the early termination check being six.

5.3 Pass Optimisations

In this section, we discuss the optimisations made to our implementation of the MMSS algorithm to reduce the number of passes it requires. These optimisations introduce heuristic rules which exploit the structure of specific graphs, therefore, whilst they do not reduce the number of passes in the worst case, they have led to a significant reduction in practice.

5.3.1 Unoptimised Pass Count

Using a methodology similar to that used to establish the $O(1/\epsilon^6)$ pass complexity of the MMSS algorithm in [MMSS25, Section 6.3], we can create an equation that calculates the exact number of passes required by the algorithm to produce a $(1+\epsilon)$-approximation for varying values of ϵ.

We begin by computing the number of passes completed during a fixed scale h. During scale h, we iterate over $t_{max}(h) = \frac{144}{h\epsilon}$ phases and during each phase, we have $\tau_{max}(h) = \frac{72}{h\epsilon}$ pass-bundles. Each pass-bundle consists of exactly three passes over the edge stream. Therefore, for a scale h, the total number of passes over the edge stream is:

$$t_{max}(h) \cdot \tau_{max}(h) \cdot 3 = \frac{144}{h\epsilon} \cdot \frac{72}{h\epsilon} \cdot 3 = \frac{31104}{h^2\epsilon^2}$$

During the algorithm, we complete scales for each value of $h = \frac{1}{2}, \frac{1}{4}, \ldots, \frac{\epsilon^2}{64}$, which can also be written as $h = \frac{1}{2^n}$ for each $n = 1, 2, \ldots, \lceil \log_2(\frac{64}{\epsilon^2}) \rceil$. Therefore, the total number of passes completed during all of the scales combined is:

$$\sum_{n=1}^{\lceil \log_2(\frac{64}{\epsilon^2}) \rceil} \frac{31104}{(\frac{1}{2^n})^2\epsilon^2} = \frac{31104}{\epsilon^2} \cdot \sum_{n=1}^{\lceil \log_2(\frac{64}{\epsilon^2}) \rceil} 2^{2n} = \frac{31104}{\epsilon^2} \cdot (4 \cdot \frac{4^{\lceil \log_2(\frac{64}{\epsilon^2}) \rceil}}{4-1}) = \frac{124416(4^{\lceil \log_2(\frac{64}{\epsilon^2}) \rceil} - 1)}{3\epsilon^2}$$

Finally, at the beginning of the algorithm, we make a single pass to get our initial greedy approximation, making the total number of passes for the entire algorithm:

$$1 + \frac{124416(4^{\lceil \log_2(\frac{64}{\epsilon^2}) \rceil} - 1)}{3\epsilon^2}$$

The correctness of this equation can be verified by counting the number of passes completed during the execution of our implementation, where we observe the same values. The number of passes are provided in Table 5.4 and remain constant no matter the graph used. They will serve as a baseline value to compare against our optimisations.

ϵ	Total Passes
0.75	1207885825
0.5	10871470081
0.25	695784038401
0.1	278313876633601

Table 5.4: **A table showing number of passes required by our implementation of the MMSS algorithm before optimisation, for varying values of ϵ.** Before optimisations, the total number of passes is constant for a fixed ϵ, regardless of the graph.

5.3.2 Phase Skip Optimisation

For a given phase t, if during any pass-bundle r within the phase, we do not complete any free vertex structure operations, i.e. Contract, Augment, Overtake or Backtrack, then we augment the matching with all the augmenting paths currently in $\mathcal{P}$ and then skip the remaining pass-bundles in phase t, immediately moving to the next phase.

This optimisation works as if we have not completed any operations during a pass bundle, then all of the free vertex structures must be inactive or marked as on hold. Structures are only marked as inactive once they have been fully explored, so if all the structures in the graph are inactive, there is nothing else to do in the current phase. Structures are only marked as on hold once they have reached the maximum number of explored vertices in a structure for the current scale, which means that we are unable to explore the structure further. In these two cases, the graph has been fully explored with respect to the current scale, so the current phase can be ended early, with the matching augmented with any augmenting paths found, which may create new paths that can still be found in the current scale.

5.3.3 Scale Skip Optimisation

For any given scale h, if after the competition of any phase t, within the scale, we find no augmenting paths (that is, $|\mathcal{P}| = 0$) we skip the remaining phases within the scale and move to the next scale. This optimisation works as between phases we do not store anything other than the current matching. This means that if during a phase we have not found any augmenting paths for the matching, then since the next phase uses the exact same graph and matching, it will also not find any augmenting paths as it will complete the exact same steps. Instead, we continue on to the next scale, where we may find new augmenting paths as the free vertex structure size limitation, limit_h, increases.

5.3.4 Algorithm Skip Optimisation

If after the completion of any phase t, $|\mathcal{P}| = 0$, that is no augmenting paths have been found, and every free vertex structure is inactive, with none are marked as *on hold*, we can end the algorithm early.

This optimisation works because structures are only marked as inactive once their depth-first search has been completed, and since the structures are not marked as *on hold*, we know search was not size-bound, i.e. if we completed the same search in a later scale, we would not find any more vertices. Since the depth-first search from each free vertex has been fully completed and no augmenting paths have been found, we know that there are no more augmenting paths to find. By Berge's Theorem [Ber57, Theorem 1], which states that a matching is maximum if and only if there exist no augmenting paths in the graph with respect to the matching, we know that we therefore have a maximum matching, which cannot be improved on, so we can end the algorithm early.

5.3.5 Optimised Pass Count

In our algorithm implementation, each of the three optimisations discussed are enabled by default; however, they can be disabled by passing `optimisation_level=false` when calling the algorithm.

In Table 5.5, the number of passes required by our optimised implementation of the MMSS algorithm for varying values of ϵ is provided, for each of the graphs tested. The early terminal check was disabled for each of these tests. We see a vast reduction in the number of passes required compared to the unoptimised implementation, shown in Table 5.4. In addition, we can see the algorithm skip optimisation in effect, with different values of ϵ, such as 0.25 and 0.1 for feather-lastfm-social, using the same number of passes because the remainder of the algorithm was skipped. Our results demonstrate that in real-life graphs, the number of passes required by the algorithm is significantly less than in the worst-case, although the optimisations mean that the number of passes required now differs between graphs.

Dataset Name	ϵ	Total Passes
feather-lastfm-social	0.75	7324
	0.5	11782
	0.25	30757
	0.1	30757
ego-Facebook	0.75	2173
	0.5	2173
	0.25	2173
	0.1	2173

Dataset Name	ϵ	Total Passes
email-Enron	0.75	7939
	0.5	12166
	0.25	37894
	0.1	62878
ca-AstroPh	0.75	2542
	0.5	2542
	0.25	2542
	0.1	2542

Table 5.5: **A table showing the number of passes over the edge stream the MMSS algorithm requires to compute the maximum matching for a specified value of ϵ after the optimisations described in Section 5.3.** In each of these results, the early termination check is disabled.

In Table 5.6, we give a phase-by-phase breakdown of the algorithm's progress on the feather-lastfm-social graph, with $\epsilon = 0.1$ and the three skip optimisations enabled. The table shows every phase in the algorithm and how many pass-bundles it contained, allowing us to see where the reduction in passes has been made. From this table, we can see that the phase skip optimisation is used during every phase, significantly reducing the number of pass-bundles used in each phase. In addition, each scale also uses significantly less phases than in the worst-case, with the scale skip optimisation being used within the first four phases of each scale. We can also see that the algorithm skip optimisation occurs during scale ten, which results in the final three scales being skipped.

feather-lastfm-social, $\epsilon = 0.1$

Scale	Phase	Pass-bundle	Matching Size	Optimisation	Total Passes
Greedy	-	-	2796	-	1
1	1/2800	14/1440	3261	PS	43
	2/2800	18/1440	3311	PS	97
	3/2800	18/1440	3313	PS	151
	4/2800	20/1440		PS, SS	211
2	1/5760	31/2800	3332	PS	304
	2/5760	28/2800	3333	PS	388
	3/5760	28/2800		PS, SS	472
3	1/11520	81/5760	3339	PS	715
	2/11520	57/5760	3340	PS	886
	3/11520	57/5760		PS, SS	1057
4	1/23040	135/11520	3344	PS	1462
	2/23040	94/11520		PS	1744
5	1/46080	189/23040	3346	PS	2311
	2/46080	195/23040		PS, SS	2896
6	1/92160	376/46080	3347	PS	4024
	2/92160	374/46080		PS, SS	5146
7	1/184320	726/92160		PS, SS	7324
8	1/184320	1486/73728		PS, SS	11782
9	1/368640	3066/184320		PS, SS	20980
10	1/737280	3259/368640		PS, SS, AS	30757
11	Skipped	-		-	30757
12	Skipped	-		-	30757
13	Skipped	-		-	30757

Table 5.6: **A phase-by-phase breakdown of the algorithm's progress on the feather-lastfm-social graph, with $\epsilon = 0.1$ and the three skip optimisations enabled.** The table shows each of the phases completed during the algorithm's execution, which scale they belonged to, how many pass bundles were completed during it and any optimisations that occurred during the phase. In the Optimisation column, PS, SS and AS represent the Phase Skip, Scale Skip and Algorithm Skip optimisations, respectively. We provide similar tables for the remaining values of ϵ in Table C.5, Table C.6 and Table C.7.

Another key observation that we see is that, in general, when the algorithm is run on a graph, using different values of ϵ, the number of passes required to complete each phase now remains constant. This can be seen when comparing the data shown in Table 5.6, where we use $\epsilon = 0.1$, to Table C.6, where we use $\epsilon = 0.5$; for example, in both runs, the first scale contains four phases, containing pass-bundles of size 14, 18, 18 and 20, respectively. This is because the limitation on augmenting paths that can be found is the maximum size of free vertex structures, limit_h, whose value depends solely on the current scale and is independent of the approximation factor used, meaning it doesn't change for different values of ϵ. However, for much larger values of ϵ (where $\epsilon \approx 1$), this observation is not the case as the maximum number of pass-bundles in a phase, $\tau_{max}(h)$, which is dependant on ϵ, is too small.

5.4 Performance Analysis

In this section, we discuss the performance metrics of our implementation. The results were compiled using GCC with the `-O3` optimisation flag, which enables a wide range of compiler optimisations, such as loop unrolling and instruction scheduling, to improve runtime efficiency. We also provide hardware specifications of the device used to obtain the results in Table 5.7.

Device Name	Apple MacBook Pro 14-inch
Processor	Apple M1 Pro
Operating System	macOS Sequoia 15.3.1
Installed RAM	32 GB

Table 5.7: **Specifications of the device used for the performance analysis.**

Table 5.8, shows some of the key performance metrics collected during the execution of our implementation. As expected, the execution time generally increases as ϵ decreases, since smaller values of ϵ require additional passes over the edge stream to reach the required approximation. However, this is not always the case, as each ϵ value tested on the ego-Facebook has an almost identical execution time. This is caused by the algorithm skip optimisation, which stops the algorithm once no more progress can be made, resulting in each ϵ value completing the same number of passes and therefore the same amount of computation. We see a similar trend when looking at the peak memory usage, with smaller values of ϵ requiring more memory. This is because the later scales allow each free vertex structure to explore deeper into the graph, therefore increasing the amount of memory required to store these structures.

When we compare the peak memory usage between graphs, we see that it appears to be loosely correlated with the number of vertices in the graph. This makes sense as our implementation creates a new `GraphVertex` object for each explored vertex. Therefore, graphs with more vertices require more objects to be stored, resulting in higher memory usage.

Another observation that we can make when looking at peak memory usage is that despite following the semi-streaming model, the memory usage of our implementation is greater than the file size of the original graph. One of the main reasons for this is that the original graph is stored as a plain text file, where each character uses one byte, which means that a four-digit number uses four bytes. In our implementation, we store these values as 64-bit integers, which use eight bytes regardless of the size of the actual value. This is only more space-efficient for numbers exceeding eight digits, however the graphs we have tested are not this large, resulting in an additional overhead in memory usage. Further optimisations could be made to reduce this overhead, in addition to improving the space efficiency of variables stored in our implementation.

Dataset Name	ϵ	Execution Time (s)	Peak Memory (MB)
feather-lastfm-social	0.75	129.3	5.0
	0.5	215.5	8.5
	0.25	639.4	17.9
ego-Facebook	0.75	62.4	3.7
	0.5	62.6	3.7
	0.25	62.5	3.6
email-Enron	0.75	969.9	13.1
	0.5	1507.4	14.1
	0.25	5066.0	62.3
ca-AstroPh	187.1	194.2	7.0
	0.5	186.5	6.7
	0.25	186.9	6.8

Table 5.8: **A table showing the performance of our implementation on the real-life datasets.** During these trials, the early termination check was disabled but all three skip optimisations were enabled. We round the results gathered to one decimal place.

In Table 5.9, we look in more detail at the execution time, measuring the amount of time a single pass-bundle takes for each of the graphs. When comparing these times between graphs, we see that graphs with more edges take longer, this is as expected as it will take longer to make a single pass over a longer edge stream. We also see that the average time taken per pass-bundle increases as ϵ increases. This is because the free vertex structures in later scales are larger, meaning that some operations may take longer, such as case two of Overtake which would have to move more vertices between structures. This increase is more noticeable when ϵ changes from 0.5 to 0.25 than from 0.75 to 0.5. This is because change from 0.75 to 0.5 only adds one addition scale, whereas 0.5 to 0.25 adds two.

5.5 Randomised Edge Order

Thus far, our results have been based on graphs whose edge streams have been lexicographically ordered, however, in this section we will discuss the performance of the algorithm when the edge stream order has been randomised. To evaluate this, for each graph mentioned in Table 5.1, we randomly shuffled the order of the lines in the edge file before running our implementation.

For each graph, we then ran the algorithm on three randomised edge streams, the results of which can be seen in Table 5.10, where each run describes a different randomised order. Similarly to our tests on the

Dataset Name	ϵ	Min time per pass-bundle (s)	Max time per pass-bundle (s)	Avg time per pass-bundle (s)
feather-lastfm-social	0.75	0.039	0.060	0.053
	0.5	0.038	0.080	0.054
	0.25	0.039	0.096	0.062
ego-Facebook	0.75	0.077	0.099	0.086
	0.5	0.077	0.101	0.086
	0.25	0.077	0.101	0.087
email-Enron	0.75	0.279	0.409	0.369
	0.5	0.277	0.439	0.371
	0.25	0.278	0.514	0.400
ca-AstroPh	0.75	0.181	0.313	0.209
	0.5	0.181	0.292	0.208
	0.25	0.182	0.304	0.208

Table 5.9: **A table showing the time taken per pass-bundle for each real-life graph.** During these trials, the early termination check was disabled but all three skip optimisations were enabled. We round the results gathered to three decimal places.

lexicographical ordering, every graph tested reached the maximum matching with $\epsilon = 0.75$. However, the number of passes required by each run varied, with some runs performing better than the lexicographical order, and some worse. Although these differences seem large, for example, for ego-Facebook, run one requires more than double the number of passes of run two, in comparison to the worst case, they are minimal.

Dataset Name	ϵ	Run 1	Run 2	Run 3
feather-lastfm-social	0.75	8203	7303	10579
	0.5	12487	11953	15028
	0.25	30592	31045	33721
ego-Facebook	0.75	2707	1216	2098
	0.5	2707	1216	2098
	0.25	2707	1216	2098
email-Enron	0.75	8119	7687	8905
	0.5	12145	11644	12928
	0.25	36538	35944	37375
ca-AstroPh	0.75	2695	2155	2629
	0.5	2695	2155	2629
	0.25	2695	2155	2629

Table 5.10: **A table showing the number of passes completed when running the MMSS algorithm on graphs with randomised edge streams.** In these experiments, the early termination check was disabled but the skip optimisations were enabled. As with the lexicographical ordering, each graph reached a maximum matching with $\epsilon = 0.75$.

When we compare the phase-by-phase breakdown of each run of a given graph, we see that different runs use a different number of phases within a scale, and each phase of a run contains a different number of pass-bundles to other runs. This results in the variation in the number of passes required that was observed and is caused by the different edge orderings that cause different free vertex structures to be created.

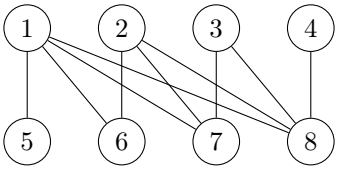
Another observation we can make from the phase-by-phase breakdown is that each run has a different initial matching, with the runs of the feather-lastfm-social graph having sizes 2577, 2559 and 2572, respectively. These initial matchings are similar to the matching found in the lexicographically ordered stream and are, once again, significantly better than the worst-case 2-approximation. These results suggest that a larger initial matching size does not necessarily lead to a reduced number of passes required. We will investigate the effect of the size of the initial matching further in the following section.

5.6 Adversarial Graphs

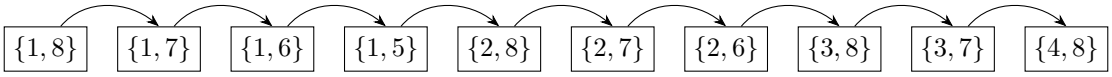
Throughout our analysis, the algorithm has provided accurate approximations using significantly fewer passes than it's theoretical worst-case pass complexity. One potential reason for this is that the initial greedy approximation has been much larger than a 2-approximation. We have therefore also tested the algorithm on graphs with adversarial edge streams, where the initial approximation provides a matching exactly half the size of the maximum matching.

The half graph [Erd84] is a particular family of bipartite graphs, where the vertices can be divided into two edge-disjoint sets V_1 and V_2 and an edge $\{u_i, v_j\}$ between $u_i \in V_1$ and $v_i \in V_2$ exists if and only if $i \leq j$. Formally, $V = V_1 \cup V_2$, where $V_1 = \{u_1, u_2, \ldots, u_{\frac{n}{2}}\}$ and $V_2 = \{v_1, v_2, \ldots, v_{\frac{n}{2}}\}$, and $E = \{(u_i, v_j) \mid i \leq j\}$. An example of a half-graph can be seen in Figure 5.7a.

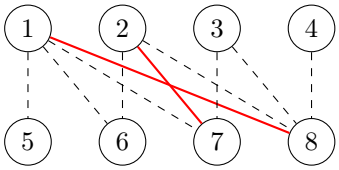
Rather than using a random or lexicographical edge arrival order, we use an edge arrival order designed to induce the worst-case scenario for the initial Greedy-Approx procedure. This results in a situation where the initial matching is exactly half the size of the maximum matching. The adversarial ordering is produced as follows; similarly to lexicographical orderings, we sort the edges in ascending order based on their first vertex. However, in the case where two edges have the same first vertex, they are sorted by their second vertex in descending order. We can see an example of the adversarially ordered stream and the resulting matching Greedy-Approx produced in Figure 5.7.



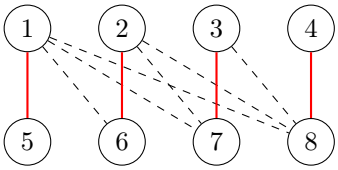
(a) An eight vertex half graph



(b) The adversarial edge arrival order of the stream for the half graph depicted in (a)



(c) The matching produced by Greedy-Approx for the graph in (a) from the edge stream shown in (b)



(d) The sole maximum matching of the graph in (a)

Figure 5.7: **An example of a half graph, its adversarially ordered edge stream, and the initial matching produced. (a)** An example of a eight vertex half graph, where $V_1 = \{1, 2, 3, 4\}$ and $V_2 = \{5, 6, 7, 8\}$. **(b)** The adversarially ordered edge stream for the half graph shown in (a). **(c)** The matching generated by the Greedy-Approx procedure on the edge stream shown in (b). **(d)** The maximum matching of the graph shown in (a). In subfigures (c) and (d) we represent matched edges with red lines and unmatched edges with dashed black lines.

In Table 5.11, we provide information about each of the half graphs used during our testing. We chose to use a range of different sizes, which would also allow us to compare the performance of the algorithm on graphs with similar structures but different sizes.

Identifier	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert$Opt$\rvert$
halfGraph-1200	1200	180300	600
halfGraph-1600	1600	320400	800
halfGraph-2400	2400	720600	1200

Table 5.11: **A table showing relevant information about the half graphs tested during this section, ordered by ascending edge count.**

In Table 5.12 we present the results obtained when running our implementation using the half graphs

as input. From these results, it can be seen that as the number of passes required by the algorithm increases as the size of the half graph increases, this is expected as more augmenting paths need to be found in the larger graphs. These graphs also require more passes than the real-life graphs tested earlier, although they still perform significantly better than the worst case in both approximation accuracy and passes required, in most cases reaching the optimal solution before the end of their execution. However, this is not the case for halfGraph-1600 and halfGraph-2400 for $\epsilon = 0.75$, which does not produce a maximum matching but still produces a result far larger than the worst-case approximation.

Identifier	ϵ	$\lvert M \rvert$	Total Passes
halfGraph-1200	0.75*	600	16333
	0.5*	600	16333
	0.25*	600	16333
halfGraph-1600	0.75	797	24535
	0.5*	800	30478
	0.25*	800	30478
halfGraph-2400	0.75	1195	32527
	0.5*	1200	47113
	0.25*	1200	47113

Table 5.12: **A table showing the number of passes used by the MMSS algorithm on the adversarially ordered half graphs.** In these experiments, the early terminal check was disabled but the skip optimisations were enabled. The * marks ϵ values where the algorithm skip optimisation occurred.

In Table C.8, we provide a phase-by-phase breakdown of the algorithm execution on halfGraph-1600 for $\epsilon = 0.5$. When compared with similar breakdowns on the real-life graphs, such as in Table C.6, the algorithm completes more phases during the later scales before the scale skip optimisation occurs. This is because some of the augmenting paths found in the half graph are considerably longer and therefore can only be found until the later scales, where the maximum free vertex structure size is larger. This is clearly illustrated when comparing the length of the longest augmenting paths, halfGraph-1600 contains a path of length 1053, whilst the longest path in feather-lastfm-social is only 149 edges long. We discuss more about the augmenting path lengths found in Section 5.7.1.

Another notable difference between the breakdowns of the half graph and real-life graphs is the algorithm's behaviour upon reaching the maximum matching. For the half graph, the algorithm skip optimisation occurred immediately, whereas in the real-life graphs we tested, the algorithm continued on, completing more scales before terminating. This is because the final matching produced by the algorithm for halfGraph-1600 is a perfect matching, meaning there are no free vertices in the graph once the maximum matching is obtained. However, in real-life graphs, perfect matchings are rare and the algorithm instead has to do additional work to confirm that the matching can not be improved.

5.7 Operation Observations

In this section, we provide a brief explanation of some of the observations we have made when looking at the key operations and structures used during the execution of the MMSS algorithm.

5.7.1 Augmentations

In Figure 5.8, we see a breakdown of the size of the augmenting paths found in each phase when the algorithm runs on the feather-lastfm-social graph. In general, the length of the path increases as the scale does. However, this is not always the case, for example, in the graph we have shown, scale two contains paths of lengths ranging between 11 and 27, but during scale three, paths of lengths 21 are found. This means that during scale two, not all the augmenting paths of length less than 27 are found. There are two reasons for this, firstly, augmenting a matching with an augmenting paths can create new augmenting paths, which can be shorter, meaning that once we find a longer augmenting path in a later phase, it enables us to then find numerous other smaller ones. This is not the case in our example, as all of the augmenting paths found in third scale are short enough to be found in the second scale, so the algorithm did not need to wait till scale three to find them. Instead, this occurs because of the inherent nature of the depth-first search algorithm, which searches as deep down a path as possible, before backtracking to

find alternative paths. Thus, when our free vertex structure conducts it's depth-first search, it reaches its maximum size, limit_h before it is able to backtrack, making other paths only accessible in later scales when limit_h is larger.

As mentioned when we discussed the performance of the half graphs in Section 5.6, the augmenting paths found by the algorithm in the real-life graphs are relatively short compared to the total number of edges, for example, in Figure 5.8 the longest path is of length 149. One of the reasons for this is that real-life graphs commonly exhibit the small world property, where every vertex can be connected to one another by a short path, whilst also remaining relatively sparse. These properties help limit the size of augmenting paths, which in turn plays a key role in enabling the algorithm's strong performance on practical graphs.

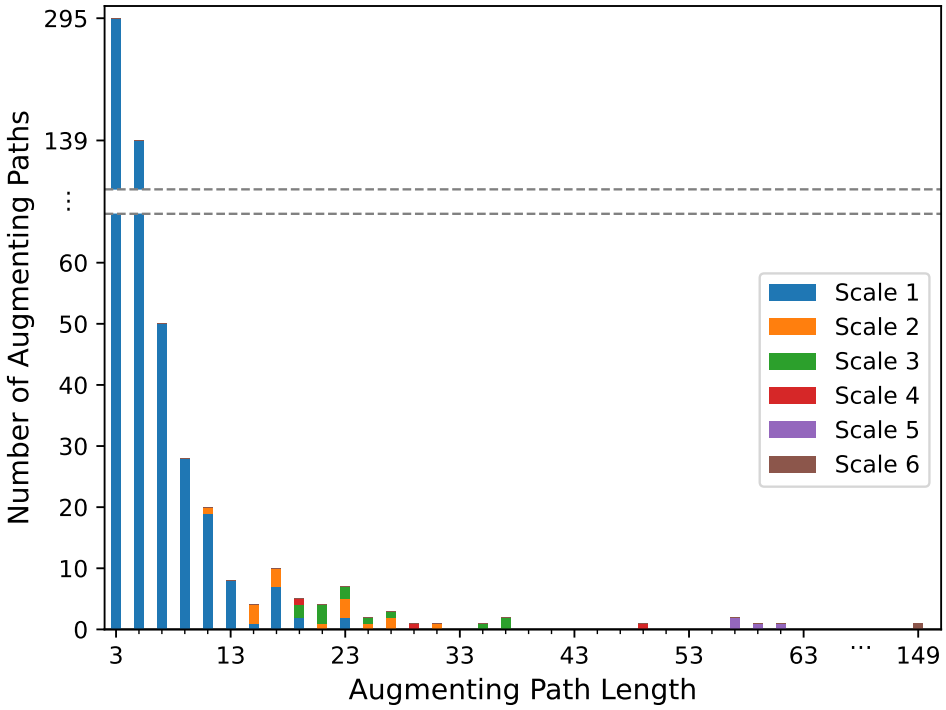


Figure 5.8: **A figure showing the varying lengths of the augmenting paths found during the execution of the MMSS algorithm on the lexicographically ordered feather-lastfm-social graph with $\epsilon = 0.75$.**

5.7.2 Blossoms and Contractions

One of the key differences between the MMSS algorithm and its precursor [FMU21] was the application of alternating trees and blossoms [Edm65]. These structures allowed for simplifications to be made to the algorithm, which contributed to the reduction in pass complexity from $O(1/\epsilon^{19})$ to $O(1/\epsilon^{6})$.

In Table 5.13, we present data on the non-trivial blossoms created during execution of the algorithm on the lexicographically ordered edge stream of the feather-lastfm-social graph, where $\epsilon = 0.75$. This data demonstrates the heavy use of the blossom structure throughout the entire algorithm, showing its importance. The first observation we can make from this data is that the root blossoms (Ω) grow increasingly deep as the algorithm progresses, causing both the average number of vertices in a root blossom and the average depth to increase. This increase corresponds with the increase in average free vertex structure size that occurs as the depth-first search is allowed to continue deeper into the graph, which results in more blossoms being identified.

Another interesting observation can be made when comparing the size of blossoms across the various different edge stream orderings. In Table C.9, we see the same information about the blossom structures formed when running the algorithm on the randomly ordered edge stream of the feather-lastfm-social, using the same approximation factor. From this data, we see that the number of nodes in a blossom (that is, the number of vertices in a blossom after contracting any interior blossoms) is noticeably larger than in the lexicographically ordered graph. This is a direct result of the lexicographical edge stream ordering,

feather-lastfm-social, $\epsilon = 0.75$, lexicographical

Scale	Phase	$\lvert\Omega\rvert$	$\frac{1}{\lvert\Omega\rvert}\sum_{B\in\Omega}\lvert V(B)\rvert$	$D_{\min}$	$D_{\max}$	D_{avg}	$\lvert\mathcal{B}\rvert$	$\frac{1}{\lvert\mathcal{B}\rvert}\sum_{B\in\mathcal{B}}\lvert V(B/\Omega_B)\rvert$
1	1/384	52	3.2	1	2	1.1	55	3.1
	2/384	31	3.1	1	2	1.0	32	3.1
	3/384	25	3.1	1	2	1.0	26	3
	4/384	25	3.1	1	2	1.0	26	3
2	1/768	33	3.6	1	5	1.3	43	3
	2/768	21	3.5	1	5	1.2	26	3
	3/768	21	3.5	1	5	1.2	26	3
3	1/1536	28	4.1	1	5	1.5	41	3.1
	2/1536	25	4.3	1	5	1.6	38	3.1
	3/1536	21	4.5	1	5	1.7	34	3.2
4	1/3072	34	5.7	1	10	1.9	63	3.5
	2/3072	25	5.4	1	10	1.9	46	3.4
5	1/6144	40	5.1	1	10	1.8	68	3.4
	2/6144	25	5.6	1	10	2	47	3.5
6	1/12288	37	5.5	1	10	1.8	66	3.5
	2/12288	24	5.9	1	10	2	47	3.5
7	1/24576	31	7.1	1	10	3	71	3.67

Table 5.13: **A phase-by-phase breakdown of the size and structure of the blossoms used during the execution of the MMSS algorithm on the lexicographically ordered edge stream of the feather-lastfm-social graph, where $\epsilon = 0.75$.** To reduce the size of this table, we use various abbreviations as follows; Ω is the set of non-trivial root blossoms in free vertex structures and $|\Omega|$ is its size, $\frac{1}{|\Omega|}\sum_{B\in\Omega}|V(B)|$ is the average number of vertices contained in a root blossom and $D_{\min}$, $D_{\max}$ and D_{avg} are the minimum, maximum and average depth of the root blossoms, respectively. We use $\mathcal{B}$ to denote the set of non-trivial blossoms (both root and non-root) in free vertex structures, with $|\mathcal{B}|$ denoting its size. Finally, we use $\frac{1}{|\mathcal{B}|}\sum_{B\in\mathcal{B}}|V(B/\Omega_B)|$ to denote the average number of nodes contained in a blossom, this is the number of vertices in a blossom after all inner blossoms have been contracted.

where all the edges incident to a particular vertex often appear consecutively in the stream, which can encourage smaller blossoms.

To illustrate this, imagine the following situation; we have a free vertex structure with a working vertex x. During the ExtendActivePath procedure, this structure is extended with the unmatched arc (x, v) and the matched arc (v, t). If there were two possible contractions involving x, (t, x) and (t, y), where y is an ancestor of x in the structure, the lexicographical order is more likely to process (t, x) before (t, y), hence contracting to create a blossom of size three containing vertices x, v, t.

Similar observations can be made when comparing the results gathered from the lexicographical and randomly ordered edge streams of other graphs.

Chapter 6

Conclusion

6.1 Our Contributions

Revisiting the core goal of our project, discussed in Section 1.3, we now summarise our contributions with respect to this goal.

"Investigate the performance of the MMSS algorithm on real-life graphs." Using our implementation of the MMSS algorithm, described in Chapter 4, we have completed a thorough empirical analysis on the algorithm using a range of real-life graphs and edge stream orderings. We began this analysis in Section 5.2 by looking at the size of the matching produced during the execution. Notably, the resulting matchings are significantly larger in practise than the worst-case of $(1+\epsilon)\cdot|\text{Opt}|$. In this section, we also introduced our first optimisation, the early termination check, and demonstrated its effectiveness in reducing the number of passes required by the algorithm, particularly when using large approximation factors. In the next section, Section 5.3, we gave a detailed breakdown of the number of passes used by the algorithm, introducing three further optimisations that skipped unneeded computation. On our test graphs, these optimisations resulted in a significant reduction (by a factor of more than 10,000) in the number of passes required. We continued our empirical analysis of the algorithm in Section 5.4, where we used profiling tools to look at the time and space usage of our implementation. We concluded our analysis by looking at random orderings and adversarial graphs in Section 5.6, which provided an understanding of the algorithm's behaviour when its initial matching was as unfavourable as possible. We observed that even on our adversarially designed edge streams, although performance was worse than the real-life graphs tested, the algorithm still performed significantly better than the theoretical worst-case.

To support any further research, we have also released our implementation of the MMSS algorithm as an open-source program, accessible from Appendix B.

6.2 Future Work

We will now briefly mention some of the possible directions that further work could take within this field. The key question being:

> *Is there an algorithm that can compute a $(1+\epsilon)$-approximate maximum matching in general graphs under the semi-streaming model with a pass complexity less than that of the MMSS algorithm, namely $O(1/\epsilon^6)$?*

Given that the pass complexity of the latest algorithm for the bipartite variant of the problem is $O(1/\epsilon^2)$, this suggests that there is certainly room for further theoretical progress on the problem. There are two possible approaches to this, which we now outline.

Our empirical analysis of the MMSS algorithm showed that, on real-world graphs, the algorithm reaches its intended approximation factor significantly before its termination. Combined with the fact that the current bound on the worst-case approximation has been shown not to be tight during the earlier stages of the algorithm, as mentioned in Section 5.2.1, this suggests that it may be possible to improve on the existing theoretical analysis to establish tighter bounds, which could then in turn reduce the pass complexity of the algorithm.

The second option would be to either modify the algorithm or redesign it entirely. The MMSS algorithm itself is an example of this, where a main reason for the progress from the $O(1/\epsilon^{19})$ pass

complexity of its predecessor was the incorporation of the concepts of blossoms and alternating trees into the algorithm. A possible redesign could be made taking inspiration from the concepts used in the latest algorithm for the bipartite variant of the problem, [ALT21]. Rather than repeatedly searching for augmenting paths, this algorithm uses an auction-based approach which iteratively builds a matching. In this approach, vertices are split into two groups, bidders and items. Bidders that are connected to an item by an edge make bids on that item, with the highest bid winning, resulting in a matched edge. An extension of this approach to general graphs could reduce the number of passes required by eliminating the need to search for augmenting paths, instead replacing it with a more efficient method.

Bibliography

[ALT21] Sepehr Assadi, S. Cliff Liu, and Robert E. Tarjan. *An Auction Algorithm for Bipartite Matching in Streaming and Massively Parallel Computation Models*, pages 165–171. 2021.

[Ber57] Claude Berge. Two theorems in graph theory. *Proceedings of the National Academy of Sciences*, 43(9):842–844, 1957.

[BKM+00] Andrei Broder, Ravi Kumar, Farzin Maghoul, Prabhakar Raghavan, Sridhar Rajagopalan, Raymie Stata, Andrew Tomkins, and Janet Wiener. Graph structure in the web. *Computer Networks*, 33(1):309–320, 2000.

[CKL+22] Li Chen, Rasmus Kyng, Yang P. Liu, Richard Peng, Maximilian Probst Gutenberg, and Sushant Sachdeva. Maximum flow and minimum-cost flow in almost-linear time, 2022.

[CKP+21] Lijie Chen, Gillat Kol, Dmitry Paramonov, Raghuvansh R. Saxena, Zhao Song, and Huacheng Yu. Almost optimal super-constant-pass streaming lower bounds for reachability. STOC 2021, New York, NY, USA, 2021. Association for Computing Machinery.

[DP14] Ran Duan and Seth Pettie. Linear-time approximation for maximum weight matching. *J. ACM*, 61(1), January 2014.

[Edm65] Jack Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17:449–467, 1965.

[Erd84] P. Erdös. Some combinatorial, geometric and set theoretic problems in measure theory. In D. Kölzow and D. Maharam-Stone, editors, *Measure Theory Oberwolfach 1983*, pages 321–327, Berlin, Heidelberg, 1984. Springer Berlin Heidelberg.

[FKM+05] Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. On graph problems in a semi-streaming model. *Theoretical Computer Science*, 348(2):207–216, 2005. Automata, Languages and Programming: Algorithms and Complexity (ICALP-A 2004).

[FMU21] Manuela Fischer, Slobodan Mitrovic, and Jara Uitto. Deterministic $(1+\epsilon)$-approximate maximum matching with poly$(1/\epsilon)$ passes in the semi-streaming model. *CoRR*, abs/2106.04179, 2021.

[GO12] Venkatesan Guruswami and Krzysztof Onak. Superlinear lower bounds for multipass graph processing. *CoRR*, abs/1212.6925, 2012.

[HK73] John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973.

[LK14] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[LKF07] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM Trans. Knowl. Discov. Data*, 1(1):2–es, March 2007.

[LLDM08] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *CoRR*, abs/0810.1355, 2008.

[LM12] Jure Leskovec and Julian Mcauley. Learning to discover social circles in ego networks. In F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[McG05] Andrew McGregor. Finding graph matchings in data streams. In *Proceedings of the 8th International Workshop on Approximation, Randomization and Combinatorial Optimization Problems, and Proceedings of the 9th International Conference on Randamization and Computation: Algorithms and Techniques*, APPROX'05/RANDOM'05, page 170–181, Berlin, Heidelberg, 2005. Springer-Verlag.

[MMSS25] Slobodan Mitrović, Anish Mukherjee, Piotr Sankowski, and Wen-Horng Sheu. Faster semi-streaming matchings via alternating trees, 2025.

[MV80] Silvio Micali and Vijay Vazirani. An $o(\sqrt{|v||e|})$ algorithm for finding maximum matching in general graphs. In *21st Annual Symposium on Foundations of Computer Science (sfcs 1980)*, pages 17–27, 10 1980.

[RS20] Benedek Rozemberczki and Rik Sarkar. Characteristic Functions on Graphs: Birds of a Feather, from Statistical Descriptors to Parametric Models. In *Proceedings of the 29th ACM International Conference on Information and Knowledge Management (CIKM '20)*, page 1325–1334. ACM, 2020.

[SDM24] We Are Social, DataReportal, and Meltwater. Most popular social networks worldwide as of april 2024, by number of monthly active users (in millions) [graph], 2024. Accessed: 2025-01-28. [Online]. Available: https://www.statista.com/statistics/272014/global-social-networks-ranked-by-number-of-users/.

[Tir18] Sumedh Tirodkar. Deterministic algorithms for maximum matching on general graphs in the semi-streaming model. In Sumit Ganguly and Paritosh Pandya, editors, *38th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2018)*, volume 122 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 39:1–39:16, Dagstuhl, Germany, 2018. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[Vaz20] Vijay V. Vazirani. A proof of the MV matching algorithm. *CoRR*, abs/2012.03582, 2020.

Appendix A

AI Prompts

I did not use any AI tools within my project.

Appendix B

Code Library

Our code library implementing the MMSS algorithm and its corresponding user guide is publicly available and can be found at:
https://github.com/phil-daniel/MMSS-Maximum-Matching

Appendix C

Additional Results

Unless otherwise specified, the results displayed in this appendix use the lexicographically ordered edge stream of the graphs they describe.

ego-Facebook, lexicographical

Scale	Phase	Matching Size
Greedy	-	1857
1	1/384	1972
1	2/384	1975
1	3/384	1976
2	1/768	1977
3	1/1536	1978
6	1/12288	1979

email-Enron, lexicographical

Scale	Phase	Matching Size
Greedy	-	10088
1	1/384	11822
1	2/384	12084
1	3/384	12103
1	4/384	12104
2	1/768	12157
2	2/768	12164
2	3/768	12165
3	1/1536	12178
3	2/1536	12179
4	1/3072	12194
5	1/6144	12195
6	1/12288	12197
6	1/12288	12198

Table C.1: **A table showing the phases and scales of the algorithm during which the matching size changes for the remaining graphs not displayed in Table 5.2.** The algorithm is run on each graph with an approximation factor of $\epsilon = 0.75$. The scale values shown in the table represent $\frac{1}{2^x}$, where x is the value provided in the scale column. Each row in the table corresponds to a scale and phase of the algorithm where the matching size increased during the phase's AUGMENT-MATCHING procedure. In this table we present the results for the ego-Facebook and email-Enron graphs, whose maximum matchings are of size 1979 and 21445, respectively.

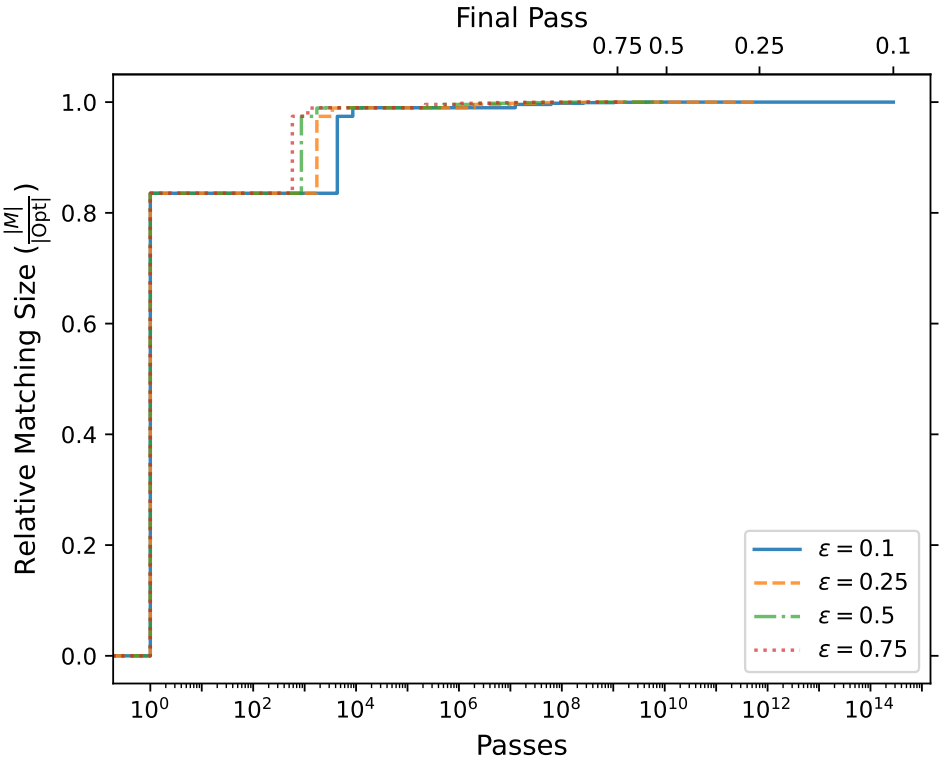


Figure C.1: **A graph showing the change in matching size for the feather-lastfm-social during the execution of the MMSS algorithm with varying values of ϵ.** During this experiment, both the early termination and skip optimisations are disabled. In the upper x-axis, we mark the final pass for each value of ϵ tested.

ϵ	Scale	Phases in Scale	Pass-bundles in each Phase	Total Passes in Scale
0.75	1	384	192	221184
	2	768	384	884736
	3	1536	768	3538944
	4	3072	1536	14155776
	5	6144	3072	56623104
	6	12288	6144	226492416
	7	24576	12288	905969644
0.5	1	576	288	497664
	2	1152	576	1990656
	3	2304	1152	7962624
	4	4608	2304	31850496
	5	9216	4608	127401984
	6	18432	9216	509607936
	7	36864	18432	2038431744
	8	73728	36864	8153726976

Table C.2: **A table showing a breakdown of the number of passes used by the MMSS algorithm before optimisation for different values of ϵ.** The scale values shown in the table represent $\frac{1}{2^x}$, where x is the value provided in the scale column. Results for other values of ϵ can be calculated in the same way, as defined in during Algorithm 3.1 and Algorithm 3.4. Before optimisations, these values are constant for a fixed ϵ, regardless of the input graph.

feather-lastfm-social, $\epsilon = 0.75$, lexicographical

Pass Bundle	1	2	3	4	5	6	7	8	9
Overtake	1280	314	149	97	71	45	36	27	16
Contract	0	11	16	9	7	4	4	3	0
Backtrack	516	404	476	168	175	90	85	44	41
Augment	384	47	14	10	5	1	1	2	1

Pass Bundle	10	11	12	13	14	15	16	...	192
Overtake	12	5	3	0	0	0	0	...	0
Contract	1	0	0	0	0	0	0	...	0
Backtrack	17	13	3	2	0	0	0	...	0
Augment	0	0	0	0	0	0	0	...	0

Table C.3: **A table showing a breakdown of the free vertex structure operations completed during the first phase of the first scale of the MMSS algorithm for the feather-lastfm-social graph, when $\epsilon = 0.75$.** During pass bundle 17 to 191, which are not shown in the table, no operations occur. This data is visualised in Figure 5.5.

Graph Name	ϵ	Final Scale	Final Phase of Scale
ego-Facebook	0.75	1	1/192
	0.5	1	1/288
	0.25	1	1/576
	0.1	1	1/2880
ca-AstroPh	0.75	1	1/192
	0.5	1	1/192
	0.25	1	1/192
	0.1	1	1/192

Table C.4: **A table showing the point at which our implementation finishes when the early finish condition enabled for the remaining graphs not displayed in Table 5.3.** The table shows the scale and phase after which the our implementation finishes for different values of ϵ when run on the test graphs with the early finish condition enabled. In both of the graphs shown in this table, the early termination check ends the algorithm after the first phase.

feather-lastfm-social, $\epsilon = 0.75$, lexicographical

Scale	Phase	Pass-bundle	Matching Size	Optimisation	Passes Completed
Greedy	-	-	2796	-	1
1	1/384	14/192	3261	PS	43
	2/384	18/192	3311	PS	97
	3/384	18/192	3313	PS	151
	4/384	20/192		PS, SS	211
2	1/768	31/384	3332	PS	304
	2/768	28/384	3333	PS	388
	3/768	28/384		PS, SS	472
3	1/1536	81/768	3339	PS	715
	2/1536	57/768	3340	PS	886
	3/1536	57/768		PS, SS	1057
4	1/3072	135/1536	3344	PS	1462
	2/3072	94/1536		PS, SS	1744
5	1/6144	189/3072	3346	PS	2311
	2/6144	195/3072		PS, SS	2896
6	1/12288	376/6144	3347	PS	4024
6	2/12288	374/6144		PS, SS	5146
7	1/24576	726/12288		PS, SS	7324

Table C.5: **A phase-by-phase breakdown of the algorithm's progress on the feather-lastfm-social graph, with $\epsilon = 0.75$ and the three skip optimisations enabled.** The table shows each of the phases completed during the algorithm's execution, which scale they belonged to, how many pass bundles were completed during it and any optimisations that occurred during the phase. In the Optimisation column, PS, SS and AS represent the Phase Skip, Scale Skip and Algorithm Skip optimisations.

feather-lastfm-social, $\epsilon = 0.5$, lexicographical

Scale	Phase	Pass-bundle	Matching Size	Optimisation	Total Passes
Greedy	-	-	2796	-	1
1	1/576	14/288	3261	PS	43
	2/576	18/288	3311	PS	97
	3/576	18/288	3313	PS	151
	4/576	20/288		PS, SS	211
2	1/1152	31/576	3332	PS	304
	2/1152	28/576	3333	PS	388
	3/1152	28/576		PS, SS	472
3	1/2304	81/1152	3339	PS	715
	2/2304	57/1152	3340	PS	886
	3/2304	57/1152		PS, SS	1057
4	1/4608	135/2304	3344	PS	1462
	2/4608	94/2304		PS	1744
5	1/9216	189/4608	3346	PS	2311
	2/9216	195/4608		PS, SS	2896
6	1/18432	376/9216	3347	PS	4024
6	2/18432	374/9216		PS, SS	5146
7	1/36864	726/18432		PS, SS	7324
8	1/73728	1486/36864		PS, SS	11782

Table C.6: **A phase-by-phase breakdown of the algorithm's progress on the feather-lastfm-social graph, with $\epsilon = 0.5$ and the three skip optimisations enabled.** The table shows each of the phases completed during the algorithm's execution, which scale they belonged to, how many pass bundles were completed during it and any optimisations that occurred during the phase. In the Optimisation column, PS, SS and AS represent the Phase Skip, Scale Skip and Algorithm Skip optimisations.

feather-lastfm-social, $\epsilon = 0.25$, lexicographical

Scale	Phase	Pass-bundle	Matching Size	Optimisation	Total Passes
Greedy	-	-	2796	-	1
1	1/1152	14/576	3261	PS	43
	2/1152	18/576	3311	PS	97
	3/1152	18/576	3313	PS	151
	4/1152	20/576		PS, SS	211
2	1/2304	31/1152	3332	PS	304
	2/2304	28/1152	3333	PS	388
	3/2304	28/1152		PS, SS	472
3	1/4608	81/2304	3339	PS	715
	2/4608	57/2304	3340	PS	886
	3/4608	57/2304		PS, SS	1057
4	1/9216	135/4608	3344	PS	1462
	2/9216	94/4608		PS	1744
5	1/18432	189/9216	3346	PS	2311
	2/18432	195/9216		PS, SS	2896
6	1/36864	376/18432	3347	PS	4024
	2/36864	374/18432		PS, SS	5146
7	1/73728	726/36864		PS, SS	7324
8	1/147456	1486/73728		PS, SS	11782
9	1/294912	3066/147456		PS, SS	20980
10	1/589824	3259/294912		PS, SS, AS	30757

Table C.7: **A phase-by-phase breakdown of the algorithm's progress on the feather-lastfm-social graph, with $\epsilon = 0.25$ and the three skip optimisations enabled.** The table shows each of the phases completed during the algorithm's execution, which scale they belonged to, how many pass bundles were completed during it and any optimisations that occurred during the phase. In the Optimisation column, PS, SS and AS represent the Phase Skip, Scale Skip and Algorithm Skip optimisations.

halfGraph-1600, $\epsilon = 0.5$, adversarial

Scale	Phase	Pass-bundle	Matching Size	Optimisation	Total Passes
Greedy	-	-	400	-	1
1	1/576	2/288	600	PS	7
	2/576	5/288	700	PS	22
	3/576	25/288		PS, SS	97
2	1/1152	18/576	750	PS	151
	2/1152	35/576		PS, SS	256
3	1/2304	84/1152	767	PS	508
	2/2304	90/1152		PS, SS	778
4	1/4608	245/2304	777	PS	1513
	2/4608	292/2304	783	PS	2389
	3/4608	320/2304		PS, SS	3349
5	1/9216	962/4608	787	PS	6235
	2/9216	614/4608	791	PS	8077
	3/9216	494/4608		PS, SS	9559
6	1/18432	674/9216	793	PS	11581
	2/18432	474/9216	794	PS	13003
	3/18432	598/9216		PS, SS	14797
7	1/36864	1275/18432	795	PS	18622
	2/36864	821/18432	796	PS	21085
	3/36864	547/18432	797	PS	22726
	4/36864	603/18432		PS, SS	24535
8	1/73728	580/36864	798	PS	26275
	2/73728	668/36864		PS	28279
	3/73728	732/36864	800	PS	30475
	4/73728	1/36864		PS, SS, AS	30478

Table C.8: **A phase-by-phase breakdown of the algorithm's progress on the adversarially ordered edge stream of the halfGraph-1600 graph, with $\epsilon = 0.5$ and the three skip optimisations enabled.** The table shows each of the phases completed during the algorithm's execution, which scale they belonged to, how many pass bundles were completed during it and any optimisations that occurred during the phase. In the Optimisation column, PS, SS and AS represent the Phase Skip, Scale Skip and Algorithm Skip optimisations.

feather-lastfm-social, $\epsilon = 0.75$, random, run 1

Scale	Phase	$\lvert\Omega\rvert$	$\frac{1}{\lvert\Omega\rvert}\sum_{B\in\Omega}\lvert V(B)\rvert$	$D_{\min}$	$D_{\max}$	D_{avg}	$\lvert\mathcal{B}\rvert$	$\frac{1}{\lvert\mathcal{B}\rvert}\sum_{B\in\mathcal{B}}\lvert V(B/\Omega_B)\rvert$
1	1/384	53	3.2	1	3	1.1	58	3
	2/384	34	3.4	1	3	1.1	38	3.1
	3/384	23	3.4	1	3	1.2	27	3.1
	4/384	22	3.2	1	3	1.1	24	3
	5/384	22	3.2	1	3	1.1	24	3
2	1/768	32	3.8	1	4	1.2	39	3.3
	2/768	28	3.6	1	4	1.1	32	3.3
	3/768	25	3.7	1	4	1.2	29	3.3
3	1/1536	36	4.2	1	4	1.2	44	3.6
	2/1536	25	4.7	1	4	1.3	32	3.9
	3/1536	25	4.7	1	4	1.28	32	3.9
4	1/3072	39	5.7	1	6	1.4	56	4.3
	2/3072	33	5.2	1	6	1.4	45	4.1
5	1/6144	35	6.9	1	8	1.7	57	4.6
	2/6144	27	7.1	1	8	1.9	48	4.5
6	1/12288	40	8.5	1	17	2.7	81	4.7
	2/12288	33	9.7	1	17	3.0	73	4.9
	3/12288	21	8.8	1	8	2.1	39	5.2
7	1/24576	26	10.5	1	11	2.6	54	5.6

Table C.9: **A phase-by-phase breakdown of the size and structure of the blossoms used during the execution of the MMSS algorithm on the randomly ordered edge stream of the feather-lastfm-social graph, where $\epsilon = 0.75$.** To reduce the size of this table, we use various abbreviations as follows; Ω is the set of non-trivial root blossoms in free vertex structures and $|\Omega|$ is its size, $\frac{1}{|\Omega|}\sum_{B\in\Omega}|V(B)|$ is the average number of vertices contained in a root blossom and $D_{\min}$, $D_{\max}$ and D_{avg} are the minimum, maximum and average depth of the root blossoms, respectively. We use $\mathcal{B}$ to denote the set of non-trivial blossoms (both root and non-root) in free vertex structures, with $|\mathcal{B}|$ denoting its size. Finally, we use $\frac{1}{|\mathcal{B}|}\sum_{B\in\mathcal{B}}|V(B/\Omega_B)|$ to denote the average number of nodes contained in a blossom, this is the number of vertices in a blossom after all inner blossoms have been contracted.