CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Greedy Matching Algorithms in Semi-Streaming Environments for Large Bipartite Graphs

Master's thesis in Computer Science and Engineering

Mattias Djärv
Gustav Ewing

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2025

Master's thesis 2025

Greedy Matching Algorithms in Semi-Streaming Environments for Large Bipartite Graphs

Mattias Djärv
Gustav Ewing

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2025

Greedy Matching Algorithms in Semi-Streaming Environments for Large Bipartite Graphs

Mattias Djärv
Gustav Ewing

Supervisor: Romaric Duvignau, Department of Computer Science and Engineering
Examiner: Peter Damaschke, Department of Computer Science and Engineering

Master's Thesis 2025
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2025

Greedy Matching Algorithms in Semi-Streaming Environments for Large Bipartite Graphs

Mattias Djärv
Gustav Ewing
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

The maximum weight bipartite matching and the related assignment problem become computationally challenging for large-scale graphs due to the high memory and time demands of optimal algorithms. This thesis investigates the viability of greedy matching algorithms within a semi-streaming environment as a practical approach for such scenarios. Several variations of greedy algorithms, including the ℓ-Local Greedy and the ℓ-Local Double Greedy, were implemented and adapted for the semi-streaming model. Their performance, in terms of solution quality, execution time, and memory requirements, were evaluated against a naive random matching algorithm and the optimal solution. Experiments were mainly conducted using generated bipartite graphs of varying sizes, with a focus on how dividing the data into chunks or shards for semi-streaming impacts performance. The results indicate that greedy algorithms, particularly when adapted for semi-streaming, offer near-optimal solutions with significantly reduced resource consumption. This means the greedy algorithms are viable for large bipartite graphs where optimal algorithms are infeasible. In conclusion, the ℓ-Local Double Greedy algorithm and especially the ℓ-Local Greedy algorithm are robust, scalable, and give near-optimal solutions for maximum weight bipartite matching in a semi-streaming environment, making them highly valuable tools for large-scale data processing and analysis.

Keywords: Greedy Algorithms, Assignment Problem, Matching, Semi-Streaming, Bipartite Graphs, Single-Pass, Multi-Pass.

Acknowledgements

We want to thank our supervisor Romaric Duvignau and our examiner Peter Damaschke at Chalmers University of Technology for their valuable feedback, insights, and guidance during this thesis. Also, a big thank you to Chalmers and the CSE department for the opportunity to pursue this thesis.

Mattias Djärv, Gustav Ewing, Gothenburg, 2025-06-24

Contents

List of Figures

List of Tables

Glossary

approximation ratio The ratio of the theoretical worst case result of an algorithm when compared to the optimal solution. 8, 11, 29

chunk A subset of a graph that always contains the entire neighborhood for any producer present in it. Therefore, a chunk can never be smaller than a neighborhood but it can contain several different neighborhoods. These are created before any algorithm is run by the creator of the graph. 15, 18, 27

neighbor Another node which connects to the origin node via an edge. 13, 19, 29, 35

neighborhood The collection of all neighbors for a specific node. 9, 14, 16, 18, 19, 31, 40, 41

practical approximation ratio The ratio of the practical results of an algorithm when compared to the optimal solution. 27, 32, 39

shard Also a subset of a graph just like a chunk but these are instead created by a specific preprocessor. Another difference is that there can also be shards that contain consumer neighborhoods in addition to the shards that contain only producer neighborhoods. 19, 35, 40

space complexity The worst case memory usage for a specific algorithm. The asymptotic upper bound of this memory usage is described using Big O notation. For example, $\mathcal{O}(n)$. 1, 5, 8, 17, 25

time complexity The worst case execution time for a specific algorithm. The asymptotic upper bound of this execution time is described using Big O notation. For example, $\mathcal{O}(n)$. 2, 3, 7, 22, 25

1

Introduction

The widespread use of large-scale network data in the information era comes with significant computational challenges. From web linkages and social media connections to biological networks and logistical systems, graphs provide a powerful abstraction for modeling complex relationships. With graphs growing larger, a problem in graph theory and combinatorial optimization becomes increasingly prevalent: the matching problem, specifically the maximum weight bipartite matching problem. This problem is inherently related to the classical assignment problem, as it can be viewed as the inverse of that problem.

The maximum weight bipartite matching problem involves finding the maximum total weight sum by pairing nodes in a weighted bipartite graph. Specifically, within a graph consisting of two distinct sets of nodes connected by edges, the objective is to find a matching. A matching is defined as a subset of these edges where no two selected edges share a common node, and the goal is to maximize the total sum of the edges' weights. Crucially, unlike the classic assignment problem, a maximum weight bipartite matching does not require a perfect matching. This means there is no requirement for all nodes in either set to be matched, making the problem suitable for scenarios where some nodes may remain unmatched.

The maximum weight bipartite matching problem involves finding the maximum total weight sum when matching nodes into pairs in a weighted bipartite graph. That is, in a graph that consists of two distinct sets of nodes connected by edges, match the nodes between the sets into pairs. This is a collection of distinct pairs of nodes, each consisting of one node from each set, that maximizes the total edge weight. Crucially, unlike the classic assignment problem which requires a perfect matching where all nodes must be part of a pair, a maximum weight bipartite matching does not have this requirement. This also makes it suitable for scenarios where a perfect matching is unfeasible.

One way to solve the matching problem is to adapt one of the optimal algorithms designed to solve the assignment problem, either by tweaking the algorithm or the dataset. Some suitable algorithms for this are the Hungarian algorithm [1] or one of its alternatives such as the Edmonds-Karp algorithm [2], the Dinitz's algorithm [3], or the Jonker-Volgenant algorithm [4]. However, when dealing with large graphs, it is no longer possible to have the entire graph in memory, which results in a space complexity of $\mathcal{O}(N^2)$, when applying a matching algorithm to it. This large memory requirement, in conjunction with the fact that these algorithms

typically have quadratic or higher time complexity, results in poor scaling on such large graphs. While it is possible to utilize parallelization to improve execution times for such large datasets, as shown in [5], this implementation requires specific hardware to operate and still does not achieve a linear time complexity. While this approach demonstrates significant improvement in handling large graphs more efficiently, it still faces challenges related to scalability, hardware dependency, and memory optimization.

In order to address the challenges posed by large graphs, this thesis will implement and tune a variety of greedy algorithms for the matching problem using a semi-streaming approach. Notably, this class of algorithms typically has a linear time and space complexity, which is crucial when working with large graphs. By also leveraging the Jonker-Volgenant algorithm as a baseline, the results of these algorithms will be compared to the optimal solution. This comparison will provide insight into when greedy approaches are viable, their limitations, and the trade-offs involved in using them over optimal solutions.

1.1 The Matching Problem

The problem addressed is a specific type of matching problem in weighted bipartite graphs. Such a graph $G = (P \cup C, E)$ consists of two disjoint sets of nodes, P (often referred to as producers) and C (often referred to as consumers), which together form the set of all nodes N. A set of edges E connects nodes, where each edge explicitly links a node in P to a node in C. Notably, each of these edges also has an associated weight which is always strictly positive, if the edge's weight is zero or negative the edge is considered to not exist. Crucially, this accommodates sparse graphs, rather than only dense ones.

The objective is to find a matching $M \subseteq E$, which is defined as a subset of edges where no two edges share a common node. The goal is to maximize the sum of weights of the edges in M. That is, to seek the set of matches that yields the highest possible total weight sum. For graphs of a moderate size, this problem is well-understood and efficiently solvable using optimal algorithms, which were originally designed for the assignment problem, since these algorithms always find the optimal solution with the absolute maximum total weight, if tuned correctly.

These types of weighted bipartite matching problems arise in various real-life scenarios where efficient resource allocation and pairing are desired. For instance, in telecommunication applications, finding the optimal configuration for the Border Gateway Protocol (BGP) can be formulated as such a problem, where egress edge links and prefixes are seen as producers and consumers, respectively [6]. Another application is within the field of distributed computer systems, where the allocation of a central database's partitions among processors can be optimized for communication cost while adhering to processing, storage, and communication capacities [7].

This thesis builds upon previous work on using matching algorithms in peer-to-peer energy sharing, as presented in [8]. This involves households that generate energy through solar power (called producers) and sharing the excess energy with

households that consume more energy than they generate (called consumers). An edge exists between a producer and a consumer if they can share energy, and the edge's weight represents the potential mutual cost savings from that specific assignment (as illustrated conceptually in Figure 1.1). The goal is to assign producers to consumers in a one-to-one fashion to maximize overall cost savings.

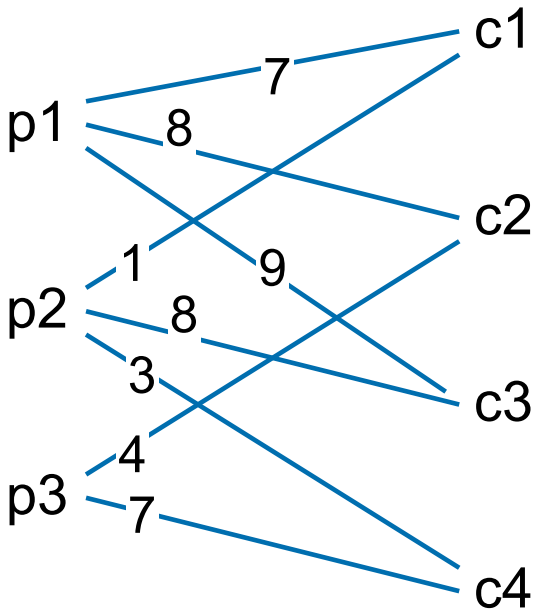

Figure 1.1: Example of a weighted bipartite graph that is used for the assignment problem.

1.2 Challenges with Large Graphs

However, these optimal matching algorithms are not as feasible when considering massive datasets. Graphs representing global social networks, the entire web, or large-scale transactional data can easily contain billions of nodes and edges. Applying the Hungarian method or the Jonker-Volgenant algorithm to such graphs encounters two primary challenges:

- **Computational Complexity**: The polynomial time complexity ($\mathcal{O}(N^3)$ for the Jonker-Volgenant algorithm) of optimal algorithms scales poorly, making computations infeasible within practical time limits for very large graphs.
- **Memory Requirements**: Perhaps more critically, optimal algorithms typically require random access to the entire graph structure, requiring it to reside in RAM. When the graph size exceeds available RAM, which is a common scenario with massive datasets, algorithms become memory-bound. Relying on slower secondary storage leads to excessive I/O, page faults, and severely degraded performance. Standard algorithms like the Hungarian method require that the entire graph is accessible, making them impractical both theoretically

(due to time complexity) and practically (due to memory limits) on very large graphs.

These limitations of optimal algorithms on massive graphs require an alternative method of processing the data. The semi-streaming model offers a practical compromise by allowing algorithms to process the input graph, which is typically read as a stream of edges [9], in multiple sequential passes. This allows algorithms to process graphs that would normally not fit into memory by dividing the graph into chunks of edges that can be processed separately.

By using this memory-constrained model, finding the optimal solution is generally infeasible. Therefore, the focus shifts towards approximation algorithms and the central goal of the semi-streaming maximum weight matching problem can be redefined as follows: output a matching whose total weight is as close as possible to the maximum weight, while minimizing time complexity and memory usage relative to the number of nodes. This involves an inherent trade-off between the quality of the solution and the computational resources required.

Greedy algorithms can be considered to be good candidates for this setting. Typically, these algorithms process edges from the stream in some order (for example, by weight) and select an edge if it does not conflict with any of the edges already selected. This means they make locally optimal choices without requiring global graph knowledge. In addition to this simplicity, their potential for linear time complexity and low memory usage aligns well with semi-streaming requirements.

1.3 Aim

The aim of this thesis is to implement multiple variations of the greedy algorithm to achieve maximum weight matching using the semi-streaming model. These can then be evaluated in terms of their practical approximation ratio while also taking real-world performance into account. This can be summarized by the following research question:

Research Question: Is it viable to utilize the greedy algorithms on very large bipartite graphs, and how do their results compare to the naive and optimal solutions?

1.3.1 Goals

This subsection outlines the goals of the thesis that are required to address the research question.

- **Acquire test graphs**: Generate random graphs and find external real-world graphs.
- **Implement a naive algorithm**: Implement a random matching algorithm that can serve as a comparison point.
- **Implement an optimal algorithm**: Implement an optimal algorithm so that it can serve as a baseline.
- **Implement greedy algorithms**: Implement the greedy algorithms without semi-streaming.
- **Adapt to semi-streaming**: Adapt all the greedy algorithm implementations to be semi-streaming.
- **Tune for semi-streaming**: Tune the algorithm implementations for better performance in semi-streaming.
- **Perform experiments**: Execute the algorithms on the generated and external graphs to acquire varied results.
- **Evaluate algorithms**: Evaluate the algorithms within the semi-streaming context based on the results.

1.3.2 Limitations

This thesis compares the implemented local greedy algorithms to the optimal solution as well as a naive random matching algorithm. Consequently, only the implemented variations of the local greedy algorithm will be evaluated. For this comparison, a variation of the Jonker-Volgenant algorithm will be used. However, this limits the size of the graphs on which such comparisons can be performed due to the time and space complexity of the Jonker-Volgenant algorithm. Therefore, only the local greedy matching algorithms will be evaluated on the larger graphs.

1.4 Thesis Outline

This thesis explores different approaches to solving the maximum weight bipartite matching problem on very large bipartite graphs using the semi-streaming model. We will implement, tune, and evaluate a variety of greedy matching algorithms specifically designed for, or adapted to this model. To assess their performance, the results (in terms of solution quality, memory usage, and execution time) will be benchmarked against the optimal solution from the Jonker-Volgenant algorithm, when run on smaller graphs where it is feasible to load the entire sparse dataset into memory. This comparative analysis aims to examine the viability of greedy

semi-streaming approaches and evaluate the trade-offs between solution quality, memory usage, and execution time for large-scale matching problems.

2

Background

Since this project compares and evaluates greedy algorithm implementations in a semi-streaming environment, background knowledge in optimal algorithms, semi-streaming, and the specific greedy algorithms in use is necessary. Therefore, this chapter introduces the Jonker-Volgenant algorithm, which is the optimal algorithm used for comparison in the context of the maximum weight bipartite matching problem, as well as the semi-streaming model and the Local Greedy algorithms. The Matrix Market format is also briefly described, since this format is used by our graph generator and by other third parties when storing graphs on disk.

2.1 Jonker-Volgenant Algorithm

The assignment problem is a computationally expensive task to solve. The first well-known algorithm to solve this problem, the Hungarian algorithm written by Kuhn in 1955, has a time complexity of $\mathcal{O}(N^4)$ [1]. While this enabled the assignment problem to be solved automatically, it takes an increasingly longer time to compute as the dataset grows larger. Multiple algorithms were proposed in the following decades that improved the time complexity slightly [2][3], and in 1987, Jonker and Volgenant proposed their algorithm using a shortest augmenting path approach [4]. This approach, previously proposed for the assignment problem by Tomizawa in 1971 [10], contains multiple optimizations when compared to previous work in the field.

The initialization phase of the Jonker-Volgenant algorithm consists of performing column and row reduction. To do this, the dataset first has to be represented in the format of a cost matrix. In a cost matrix, the element $M[p][c]$ represents the associated cost or weight of assigning node p to node c. As subtracting a constant from the weight of all elements in a row or column does not change the optimal assignment, costs are reduced such that at least one zero can be found in each row. After this, when the matrix now contains zeros, a search for a perfect matching using only these zero-cost edges is attempted. This is done using a greedy approach where, if there is a unique zero-cost edge for each assignment, this is guaranteed to be the optimal solution. If no perfect matching can be made, Dijkstra's shortest path algorithm is used to find augmenting paths from unassigned rows to unassigned columns. After all reachable columns have been visited and no zero-cost edges are found, a new round of cost reduction begins. This approach, while having a worst case

time complexity of $\mathcal{O}(N^3)$, which scales poorly for larger graphs, often performs much better in practice, making it a clear improvement compared to previous approaches.

Jonker and Volgenant also proposed a version of their assignment problem algorithm using the same data structure as before but adapted for sparse graphs [4]. This version also performed well in testing compared to contemporary algorithms for sparse graphs. Notably, they mentioned a problem with memory usage for certain algorithms when the graph exceeded a size of 400 nodes with a 20% density. As previously discussed in Section 1.2, this is still one of the main challenges of solving large-scale matching problems with optimal algorithms, where the graph grows very large.

2.2 Semi-streaming

The semi-streaming model proposed by Feigenbaum et al. [11], as mentioned earlier, has a space complexity of $\mathcal{O}(N \log(N))$ (or $\mathcal{O}(N)$ depending on the specific algorithm). This complexity represents a middle ground between offline algorithms, which can process the entire graph at once, and the more constrained streaming model, which typically operates with significantly less memory, often $\mathcal{O}(\log N)$ or even $\mathcal{O}(1)$ space. The semi-streaming model differs from the streaming model, often called the online model, by allowing batch processing of nodes instead of making decisions on a per-node basis. This batch processing allows algorithms to use a buffer of nodes or edges to make more informed decisions based on locally relevant information within the graph. Without processing the entire graph at once, massive datasets that are infeasible to fit into memory can be processed by approximate algorithms to achieve close to optimal results.

Multiple papers implementing algorithms using a semi-streaming model exist, such as [12] and [13], which achieve approximation ratios of $4.91 + \epsilon$ and $2 + \epsilon$, respectively, where the ϵ is a parameter that allows the user to tune the algorithm for either better solution quality or faster execution times. Here, the approximation ratio metric represents the ratio of the guaranteed worst-case result compared to the optimal solution. Similarly, a practical approximation ratio can instead be calculated experimentally using the ratio between the optimal result and the result of the approximate algorithm.

Algorithms using the semi-streaming model can be run using a single pass over the input, reading each chunk of data into memory only once when processing. Alternatively, multiple passes over the input can be made to improve solution quality with the downside of a more complex algorithm. Both single-pass and multiple-pass approaches are viable options when implementing greedy algorithms in a semi-streaming context.

2.3 Greedy Algorithms

The local greedy algorithms are a set of algorithms that apply a greedy approach on the scale of neighborhoods. That is, all the edges that connect a producer to consumers, or vice versa. These algorithms have previously been described and evaluated by Duvignau et al. [14]. Because greedy algorithms only make decisions based on local neighborhoods within the graph, the semi-streaming model is a good approach for running these algorithms on massive datasets. As neighborhoods are subsets of the whole graph, they can be contained within the data chunks that are loaded into memory using semi-streaming, allowing a greedy algorithm to run normally without requiring the entire graph to be in memory.

2.3.1 The Local Greedy Algorithm

The simplest of these algorithms is the local greedy algorithm, which utilizes a greedy approach to the matching problem by picking the edge with the highest weight within each neighborhood [14]. This method could possibly block a better match later for another producer. Therefore, it usually does not find the optimal solution. Figure 2.1 shows an example execution of this algorithm. It is important to note in this execution that the first match in p1 blocked p2's optimal match, which in turn caused p2 to select a suboptimal match and also blocked p3's optimal match. Therefore, the sum of these greedy decisions led to a suboptimal result.

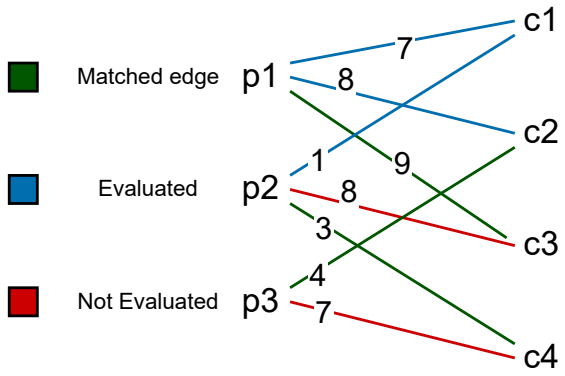

Figure 2.1: Example execution of the local greedy algorithm.

2.3.2 The ℓ-Local Greedy Algorithm

One possible modification to the previous algorithm is the addition of the ℓ parameter. This parameter restricts how much of the neighborhood is examined when performing the greedy matching by limiting the number of processed edges to $\ell + 1$. However, we instead limit the number of processed edges to ℓ in our later implementations. The example in Figure 2.2 shows the difference compared to the previous algorithm. In this case, $\ell = 1$ was chosen, which causes some edges, such as edge p1-c3, not to be evaluated. This example also shows an anomaly that sometimes occurs for greedy algorithms, where considering fewer edges can improve the result.

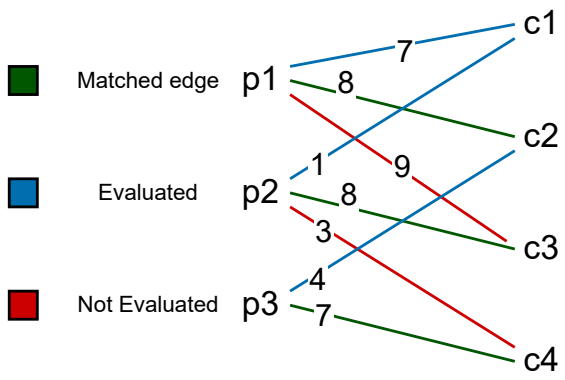

Figure 2.2: Example execution of the ℓ-local greedy algorithm.

2.3.3 The ℓ-Local Double Greedy Algorithm

Another extension to these algorithms introduced in [15] is the ℓ-Local Double Greedy algorithm, which builds greedy paths and then finds the optimal within those paths in order to match nodes. These greedy paths are constructed by matching a producer to its best consumer, then matching that consumer to its best producer. After that, the procedure repeats in the producer that the consumer matched to and the process continues. A path terminates when a node has no valid edge, which can occur either due to a lack of edges or due to all edges connecting to nodes that are unavailable or already on the path. Figure 2.3 shows an example execution of this algorithm. In this case, the ℓ value was set to 2, which means that the entire neighborhood is always considered. Note that this path terminates in c2 as there are no more valid edges due to p1 already being a part of the path.

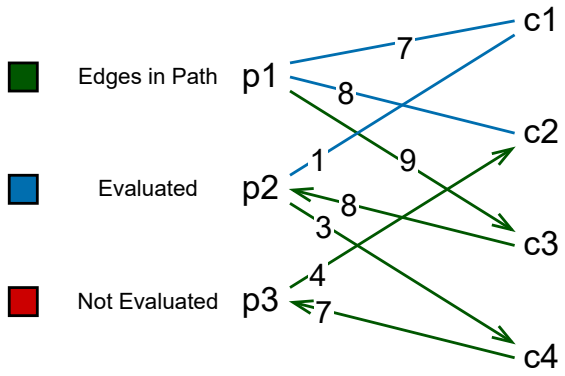

Figure 2.3: Example execution of the ℓ-local double greedy algorithm. Here the arrows indicate the order in which the algorithm explores the graph.

The algorithm then needs to extract the matching from the constructed greedy path. A naive approach involves simply selecting all edges in the path that originate from a producer, which yields a viable, albeit potentially suboptimal, partial matching. A more effective option, in terms of solution quality, is to identify the subset of edges within the path that maximizes their total weight. This alternative is, however, more computationally expensive, thus presenting a trade-off between solution quality and

computational cost.

2.4 Approximation Ratios

Duvignau et al. [15] also determined the approximation ratios for the different local greedy algorithms under some different assumptions. Table 2.1 is a reproduction of a table, from the improved paper that introduced the ℓ-Local Double Greedy algorithm, which shows the approximation ratio for each corresponding assumption.

Table 2.1: Approximation ratios for the different Local Greedy algorithms, simplified and adapted from [15]. Here Local is the Local Greedy algorithm while ℓ-Local is the ℓ-Local Greedy algorithm and lastly Double is the ℓ-Local Double Greedy algorithm.

<table>
<tr><th rowspan="2">Algorithm</th><th colspan="4">Approximation Ratio</th></tr>
<tr><th>Optimal</th><th>Local</th><th>ℓ-Local</th><th>Double</th></tr>
<tr><td>No assumptions</td><td rowspan="4">1</td><td colspan="3">∞</td></tr>
<tr><td>Strong β & no γ</td><td>$1+\beta$</td><td colspan="2">∞</td></tr>
<tr><td>No β & strong γ</td><td>$1+\gamma$</td><td colspan="2">∞</td></tr>
<tr><td>Strong β & strong γ</td><td>min{1+β, max{1,$\beta+\gamma$}}</td><td>max{1,$\beta+\gamma$}</td><td>2*max{1,$\beta+\gamma$}</td></tr>
</table>

2.4.1 β and γ Assumptions

The two assumptions used in Table 2.1 are the β and γ assumptions, with both of these assumptions each consisting of a strong and a weak variant. However, since only the strong assumptions are utilized in this thesis, the weak assumptions will not be presented.

For the greedy algorithms considered here, producers are often processed in ascending producer ID order for simplicity. However, the theoretical applicability of the β and γ assumptions does not strictly depend on this specific global ordering. Indeed, within the neighborhood of any given producer or consumer, the order in which edges are examined can be arbitrary, provided that the weight-ratio assumptions detailed below hold.

Specifically, the strong γ assumption states that for any producer p, if the algorithm processes an edge (p, c_i) before another edge (p, c_j) in its neighborhood, then the weight of the later-processed edge (p, c_j) cannot be more than γ times the weight of the earlier-processed edge (p, c_i). More formally, for any two edges (p, c_i) and (p, c_j) which connect p to its neighbors c_i and c_j, if (p, c_i) occurs before (p, c_j) in the processing order, then $w(p, c_j) \leq \gamma \cdot w(p, c_i)$. Figure 2.4 shows an example where this γ assumption both holds and does not hold.

The similar strong β assumption applies symmetrically from the perspective of the consumer. It states that for any consumer c, if the algorithm processes an edge (p_i, c) before another edge (p_j, c) in its neighborhood, then the weight of the later-processed edge (p_j, c) cannot be more than β times the weight of the earlier-processed edge (p_i, c). Essentially, for any two edges (p_i, c) and (p_j, c) originating from consumer c, if (p_i, c) occurs before (p_j, c) in the processing order, then $w(p_j, c) \leq \beta \cdot w(p_i, c)$. Therefore, the β and γ assumptions are essentially symmetric, which is shown in Figure 2.5.

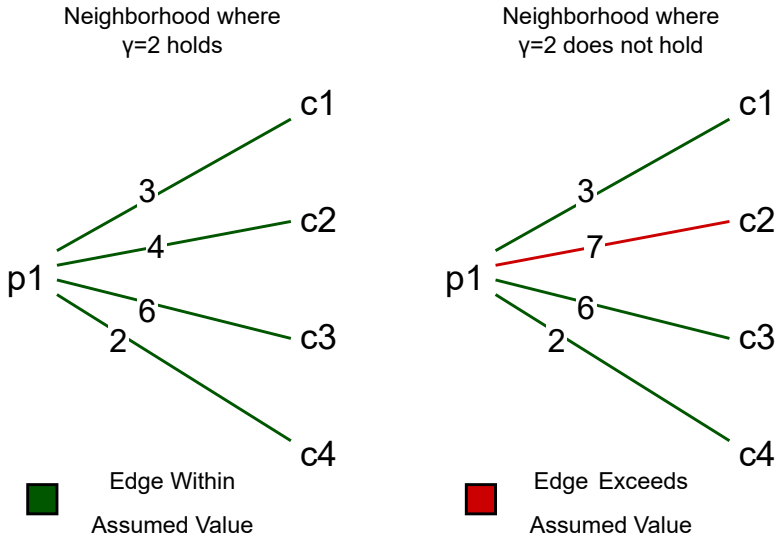

Figure 2.4: Example of the γ assumption holding and not holding.

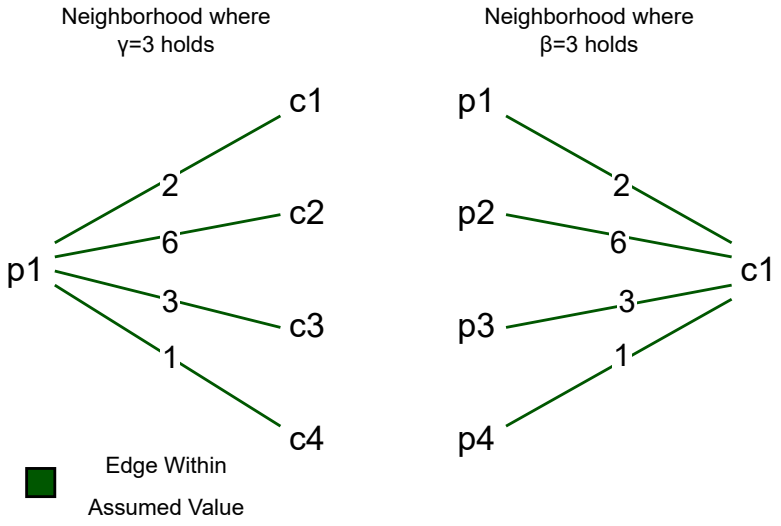

Figure 2.5: Example of how the γ and β assumptions are essentially symmetrical

Notably, only the approximation ratios for the case when there is both a strong β assumption and a strong γ assumption are relevant for this thesis as our generator only generates graphs where both the β and γ assumptions hold for some specific β and γ values.

2.5 Matrix Market

When storing and exchanging graphs, there are a number of different options available. One option is the *Matrix Market* format, which is used for the *Matrix Market* graph exchange headed by NIST [16]. This format supports both coordinate and array format, which are more suitable for sparse and dense graphs, respectively. Due to the project's focus on sparse graphs, only the coordinate format is relevant here. Apart from being sparse, this format is also human-readable, allowing for easy validation and simple parsing and graph creation. Since this is also one of the preferred formats of the *SuiteSparse Matrix Collection*, it also allows for the utilization of the many graphs found in that database [17].

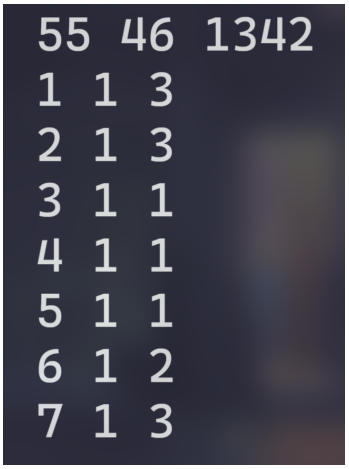

Figure 2.6: Excerpt from a sample Matrix Market file. Here the metadata as well as the values for the first 7 edges can be seen.

The *Matrix Market* format can be seen in Figure 2.6. The first entry in the file contains the metadata of the graph. This includes the number of producers, consumers, and edges that the full graph is built of. Each subsequent entry represents a single edge, the producer-consumer pair which it connects, as well as an optional field containing the edge's weight.

2.6 Neighborhood Locality in File Formats

An important observation that we have made for graph file formats like *Matrix Market* is that for a specific producer, all of its neighbors are typically stored contiguously. This is likely because the simplest way to create the file writing functionality is with a simple for loop that sequentially writes all edges for each producer. Notably, the order of the consumers within these contiguous producer segments can also be sequentially ordered or it can in some cases be more random. Such randomness would then typically be caused by, for example, iterating over entries in a hash map instead of iterating over all consumers with a for loop and performing lookups. This first approach, iterating over the hash map, will typically be more efficient with sparse graphs. Therefore, in sparse graphs, it is not necessarily safe to assume that

consumers are sequentially ordered, while it is generally safe to assume producers are.

At first glance, the previous Matrix Market example in Figure 2.6 does not seem to follow this pattern, since it only lists one neighbor for each producer instead of the entire neighborhood, as would be assumed. However, this is actually because it is an atypical dataset where the producer set is larger than the consumer set. Therefore, it is reasonable to swap the two sets in the bipartite graph so that consumers become producers while producers become consumers. After performing such a swap, it can then be observed that it follows the observed pattern. The implication of this is that it might be necessary to swap the sets in some cases to achieve this property.

In essence, it is typically easy to find a producer's neighborhood if one of its neighbors has been found. However, for the consumers, this is not the case and instead it is typically hard to find the rest of the consumer's neighborhood if only one neighbor has been found. Also if there are more producers than consumers, one should consider swapping the sets.

3

Method

This chapter first describes the implementation of the graph generator which was used to generate graphs used during the evaluation. Then, it describes the implementation of the different matching algorithms which includes a Random Matching algorithm, the Jonker-Volgenant algorithm, the ℓ-Local Greedy algorithm, and the ℓ-Local Double Greedy algorithm. Lastly, it describes how the algorithms were evaluated.

3.1 Graph Generation

For this project, bipartite graphs with a larger size than what can fit into memory are needed to test the potency of the greedy algorithms in a semi-streaming environment. To do this, a deterministic graph generation tool was created. This tool can generate bipartite graphs with several adjustable parameters:

- **Producers:** The total number of producers in the graph.
- **Consumers:** The total number of consumers in the graph. This is usually a multiple of the number of producers.
- **Sparse factor:** This determines how sparse the graph will be, defined by the probability to create an edge between a producer and a consumer.
- **Seed:** The seed that is used for all randomization, which allows for the graph generation to be deterministic so that graphs can be reproduced.
- γ **value:** This value determines how the weights of a producer's edges will scale as more are generated. A new edge will not have a weight higher than γ multiplied by the lowest weighted edge of that producer so far.
- β **value:** Similar to the γ value, the β value determines how the weights of a consumer's edges will scale as more are generated. A new edge will not have a weight higher than β multiplied by the lowest weighted edge of that consumer so far.
- **Chunk size:** This value limits the number of edges saved in each generated chunk file.

The graph generation works by simulating a check to create an edge with a probability defined by the *sparse factor* for each possible edge to a consumer that a producer can have. This is generated using a binomial distribution, using the *sparse factor*

as the probability function. By doing this, the number of edges each producer will have is determined. The consumers these edges should be connected to can then be randomized. The weight of the edges is randomized with the criterion that the weight of each subsequent edge generated is lower than γ multiplied by the lowest weight generated so far for that producer. In addition, the weight of the generated edge must also be lower than β multiplied by the lowest weighted edge of that consumer so far. When the total number of edges generated is larger than what is defined in the *chunk size* parameter, the edges are written to disk in a chunk file, and generation continues into a new chunk. Each chunk then contains all edges connected to the producers within that chunk, i.e., the producer's neighborhood, as visualized in Figure 3.1. This means that from a consumer's perspective, its edges can be spread out over multiple chunks, while a producer's edges are contained within a single chunk. In addition, by using a specific *seed* for the generated graph, the graphs can be easily re-generated on separate machines to allow for reproduction of tests.

The only thing that needs to be stored in memory between chunks is what the current maximum allowed weight for a consumer's edge, which scales with the β-value. This is because consumer neighborhoods are spread out over multiple chunks, as opposed to producer neighborhoods, which are contained solely within one chunk. All other data structures are temporary and can be cleared for each chunk, which limits memory usage to being dependent on chunk size.

The data format used when storing each chunk in a file is *Matrix Market*, as described in Section 2.5. This was chosen due to its simple, human-readable structure, ease of parsing, and widespread usage.

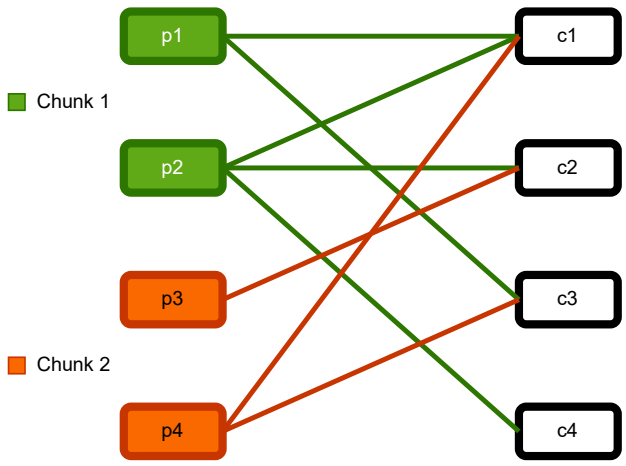

Figure 3.1: Visualization of a generated bipartite graph divided into chunks.

3.2 Naive Solution: Random Matching Algorithm

This Python solution utilizes randomness in order to pick an available edge for each producer for the matching. Every edge in the graph is stored in a hash map where

the producer and consumer pair connected by the edge is the key in the form of a tuple representing the $(producer, consumer)$ pair, and the weight of the edge is the value. This requires a space complexity of $\mathcal{O}(E)$. Every matched pair of producers and consumers is also stored, which in the worst case has a space complexity of $\mathcal{O}(P)$. This results in a total space complexity that scales linearly with the maximum of the number of edges and the number of producers, denoted as $\mathcal{O}(E + P)$.

A list of all consumers connected to each producer is stored in a dictionary. A random value from this list can then be selected and considered for a match. Since the matching between the producer and consumer needs to be exclusive, the algorithm also checks that the consumer on the randomly selected edge has not already been matched. By using an unordered set to keep track of the consumers that have a matching, an instant lookup can be made for each randomly selected edge to check for availability on the consumer side. If the consumer is already matched with another producer, the corresponding edge is discarded for that producer, and the random selection begins anew. If every possible match for a producer is to an already matched consumer, no match for that producer is made.

3.3 Optimal Solution: Jonker-Volgenant Algorithm

For finding an optimal solution to the matching problem, this thesis initially considered using the standard `linear_sum_assignment` implementation from *SciPy* [18]. This function, which implements a generalized version of the Jonker-Volgenant algorithm [19], typically operates on dense cost matrices. Consequently, it requires the construction of a matrix containing all edge weights, leading to a space complexity of $\mathcal{O}(P \cdot C)$, where P is the number of producers and C is the number of consumers. Due to this significant memory requirement for large and sparse graphs, this initial approach was identified as unsuitable for the scale of graphs needed in this thesis.

To overcome the memory bottleneck and enable computation on larger graphs, this thesis transitioned to using the `min_weight_full_bipartite_matching` function, also available within the *SciPy* library. This version of the Jonker-Volgenant algorithm allows for the use of sparse arrays for graph representation, meaning it does not reserve memory for non-existent edges. This significantly reduces the space complexity, allowing for optimal matching computations on larger graphs than previously possible. However, while space is no longer the primary limitation, the algorithm still suffers from a time complexity of $\mathcal{O}(N^3)$ [4], where $N = \max(P, C)$. This cubic complexity results in impractical computation times when the number of nodes exceeds approximately 10 million. Therefore, this optimal Jonker-Volgenant implementation is primarily used as a verification tool for the correctness of other algorithms and for comparisons on smaller graphs, such as the 100,000 $\times$ 500,000 dataset, as it is not adapted for semi-streaming environments and is always executed on graphs contained within a single chunk.

3.4 ℓ-Local Greedy Algorithm Implementation

The ℓ-Local Greedy algorithm, as mentioned in Section 2.3, uses the ℓ-value to determine how many consumers in each producer's neighborhood are considered for a matching. Using semi-streaming, this algorithm works by sequentially loading each chunk of the graph into memory. It then processes each producer neighborhood contained in that chunk. For each producer, the algorithm considers up to ℓ of its available connected consumers, prioritizing them by their order of appearance in the chunk. Among these ℓ considered consumers, it selects the one with the highest weighted edge that is currently unmatched. This selected consumer is then matched with the producer. If all consumers in a producer's neighborhood are already matched and thus not eligible for a new matching, no new match is made for that producer.

This C++ implementation uses a single pass over each chunk of the graph to create matchings. This is in comparison to using a multi-pass approach, which would require chunks to be loaded into memory multiple times during the algorithm's runtime, which increases the execution time. The single-pass approach only works if an assumption can be made about the data structure of the graph. That is, each producer's neighborhood is fully contained within a single chunk. This allows the algorithm to fully load the neighborhood from a single chunk and perform the matching without storing partial neighborhoods between chunks. If this assumption cannot be guaranteed, multiple passes through every chunk are required for each producer neighborhood, until an ℓ number of consumers has been loaded into memory to allow for the greedy matching to begin. Since the graphs that are generated using the graph-generator described in Section 3.1 ensure that this assumption holds, a single-pass approach for this algorithm is a viable option.

3.4.1 Naive and Normal Local Greedy Algorithms

By adjusting the ℓ-value, the ℓ-Local Greedy algorithm can also be used as both a Normal Local Greedy algorithm and a Naive-Local Greedy algorithm. The naive variant works by only matching with the first available consumer in each neighborhood. This can be achieved by setting the ℓ-value to 1, as this limits the number of consumers to consider for the match, ensuring the first available consumer is always chosen.

The Normal Local Greedy algorithm considers the entire neighborhood of a producer for a match. This can be achieved in this implementation by setting the ℓ-value to a larger value than the size of the largest neighborhood. Since this algorithm considers the entire neighborhood for a match, the best possible matching will be made as long as the consumer has not been matched previously. If that is the case, the next best matching in the neighborhood will be made. This allows for good matchings to be made for the producers that are found in early chunks, while the producers which are found in later chunks may receive worse matchings. As more and more consumers are matched, fewer eligible options exist for later producers. As a consequence of this, how the edges of the graph are sorted may impact the result of this algorithm.

3.5 ℓ-Local Double Greedy Algorithm Implementation

The increased complexity of the Double ℓ-Local Greedy algorithm compared to the other greedy algorithms necessitates a more complex implementation. Specifically, it requires both the producer and the consumer neighborhoods during the path construction. However, as observed in Section 2.6, it is easy to find a producer neighborhood, while it is hard to find a consumer neighborhood. Therefore, a multi-pass approach was instead chosen for the ℓ-Local Double Greedy algorithm. This warranted the creation of a sharding system that can load and unload parts of the graph during the execution and preprocessing of the graph. As with the ℓ-Local Greedy algorithm, this is a C++ implementation.

3.5.1 Sharding System

The sharding system consists of different shards that can be loaded into main memory from the disk in order to vary the memory usage of the system. Each shard then stores the neighborhoods assigned to it in an internal hash map, which maps producer and consumer IDs to their respective neighborhoods. This then allows the system to unload specific parts of the graphs without impacting the other parts of it. Therefore, it essentially enables random access to different neighborhoods of the graph but with possible I/O overhead.

To access a required producer shard, the system has a simple internal method, `getProducerShard`, which first checks whether the producer shard is already loaded and returns it immediately if so. If not, it determines whether another producer shard must be evicted in order to free up space. The possible eviction is handled using a simple Least Recently Used (LRU) strategy. Finally, if the requested producer shard already exists on disk, it is loaded. Otherwise, the system constructs a new producer shard in its place. Consumer shards instead have the almost identical `getConsumerShard` for the same purposes.

To add a new neighbor to a producer neighborhood, the system has the public method `addProducer`. This method first determines which shard the producer in question belongs to by determining its hash, and then it accesses that shard through `getProducerShard`. When it has the shard, it simply inserts the new neighbor into the preexisting producer neighborhood, whether it exists or not. Since an empty producer neighborhood simply leads to an implicit construction of that producer neighborhood. Just as before, there is corresponding `addConsumer` with the same functionality as `addProducer`.

Then, to read the neighborhood of a specific producer the public method `getProducerNeighborhood` is used. This method also accesses the shard through the `getProducerShard` method just as before. When the neighborhood contains neighbors, as expected, this method simply returns them. However, in the case that the neighborhood does not exist, that is the producer has no neighbors, it instead returns a neighborhood containing only an edge connecting to consumer 0

with the weight 0. This is, of course, an invalid value that needs to be detected and handled on the receiving end. The choice of returning specifically consumer 0 in this case is due to the fact that consumer 0 and producer 0 signal the end of a greedy path in the implementation of the Double ℓ-Local Greedy algorithm. As usual, there is also a corresponding method for the consumer neighborhoods called `getConsumerNeighborhood`.

Finally, the actual storing of shards onto the disk is handled by three internal methods: `loadShard`, `saveProducerShard`, `saveConsumerShard`. The first of these, `loadShard` reads and deserializes the shard from the disk. Meanwhile, `saveProducerShard` serializes and stores producer shards, while `saveConsumerShard` serializes and stores consumer shards. As usual, these are essentially identical except for the fact that they read from separate data structures and write to different filenames. For all three methods, the serialization library *Cereal* is used. This library was chosen due to it being a native C++ header-only library, which made it simple to integrate it into the system. However, it could be the case that another serialization library performs better for this application, but that would require more implementation time and testing to verify.

One important thing to note is that it is possible to tune the size of the shards, in terms of neighborhoods, as well as how many shards can be in memory at any time. This is handled through four variables: `maxCacheSizeProducer`, `shardSizeProducer`, `maxCacheSizeConsumer`, and `shardSizeConsumer`. The usual separation of different values for producers and consumers allows for some interesting tests and variations later on. Improper selection of these values will allow the system to consume more memory than is available. Currently, the user has to manually calculate the memory usage of these values or find viable values through testing. Therefore, a possible improvement could be to either find these values automatically or to enable the user to select some sort of configuration profile or specify their maximum available main memory for the task. Either of these options would likely be easier for an end user to understand than the current indirect values.

3.5.2 Graph Preprocessing

In order to use the sharding system, a new extra preprocessing step which loads the data into the sharding system has been added. During this step, it is generally simple to construct the shards containing producer neighborhoods, since these are typically accessed in producer order, as mentioned earlier. This leads to a sequential access pattern for the preprocessing step if only producer neighborhoods are required. However, for the consumers, this is not the case, as they are essentially entered in random order, which leads to a random access pattern when constructing the consumer shards. Therefore, when preparing shards for the ℓ-Local Double Greedy algorithm, the access pattern is not entirely sequential.

Due to the addition of the sharding system and the preprocessing step, the way the ℓ-value is handled has been changed compared to the original algorithm. Specifically, it is generally unnecessary to add more than ℓ neighbors to a producer's or consumer's neighborhood since the algorithm only considers the first ℓ neighbors for any producer

or consumer. Therefore, the preprocessor keeps track of how big each neighborhood is and if it exceeds the ℓ-value then no more neighbors are added.

However, this approach is not perfect, since with a low enough ℓ-value, it is possible that all the ℓ first neighbors of a producer or consumer have already been matched with another producer or consumer. This then results in that specific producer or consumer not matching at all, which substantially reduces the value of the overall solution. In such a case, it would be ideal to return to the preprocessing stage and add more neighbors to the neighborhood. However, this is expensive since finding a specific producer or consumer in the original graph files requires sequential reading of the files in order to guarantee it is found. For producers, it could be possible to only consider part of the files during such a reprocessing stage, since they typically are stored in order in the graph files. Currently, no such methods exist, and the producer or consumer is simply left unmatched.

3.5.3 Algorithm Implementation Specifics

As described in [15], the matching step is performed by the ℓ-Local Double Greedy algorithm. Its implementation relies on the function `doubleGreedy` and its sub-routine `nextEdge`. The function `doubleGreedy` leads the execution by managing the sets `producers` and `consumers` which track the availability of producers and consumers, it builds and stores the greedy paths, and instructs `nextEdge` which node to explore next. Meanwhile, `nextEdge` explores the node supplied from `doubleGreedy` by requesting its neighborhood with `getProducerNeighborhood` or `getConsumerNeighborhood`. Then, it determines which of these neighbors are available by using the sets `producers` and `consumers`. Lastly, it picks the best available neighbor. If there is no valid neighbor to select, it returns producer 0 or consumer 0, which causes `doubleGreedy` to terminate the active greedy path.

This special usage of the consumer 0 and producer 0 forces usage of one-indexing instead of zero-indexing, which means that caution must be taken to properly convert outside values before using them. However, the alternative of using negative values for these purposes is worse, since it forces the use of signed integers for simply handling this one specific case. Therefore, all producers and consumers have their ID value incremented by one when read from external files in order to ensure a producer 0 or consumer 0 does not exist in the data set. The risk of this causing an integer overflow is minimal, since it would require the amount of producers or consumers to be equal to the maximum value for an unsigned 32-bit integer ($4,294,967,295$). At that point, the better choice would be to switch to a 64-bit implementation. Using one-indexing was also useful for the `getProducerNeighborhood` method discussed earlier, since it can now simply treat producer 0 and consumer 0 as invalid.

Since the Double ℓ-Local Greedy algorithm has essentially random access behavior, it would seem that a uniform distribution of producers to shards would be desired. For example, $S = P\%N$, where P is the producer's ID, S is the shard P is placed in, and N is the amount of shards, would be such a uniform solution. However, during the preprocessing stage, producers generally arrive in order, which favors a more sequential solution. Therefore, $S = P/A$, where A is the size of each shard, was

chosen since it leads to less frequent evictions of producers during the preprocessing stage. Possibly, different hashing approaches for producers and consumers could be beneficial, but a uniform hash was chosen for simplicity.

3.5.4 Alternative Standard Greedy Version

During the implementation, the standard ℓ-Local Greedy algorithm has also been adapted so that it can be used for the matching step. This was mainly done in order to confirm the correctness of the sharding system and to directly compare the algorithms without the sharding system overhead only impacting one of the algorithms. That extra overhead of the sharding system impacts the performance of the ℓ-Local Greedy algorithm, but its more sequential access pattern makes it easier to handle even with the sharding system. The ℓ-Local Greedy algorithm with sharding should handle graphs with out-of-order producers better than its non-sharding version. This is because it only needs to store the producers' neighborhoods, which means it should be faster than the Double ℓ-Local Greedy algorithm as well. However, since the generator does not generate out-of-order graphs, it is hard to test this specific case, but it likely is the better option in some real-world scenarios.

3.6 Evaluation

This section describes the metrics, system hardware, and experiments used to evaluate the implemented algorithms.

3.6.1 System Hardware

The system used to perform the experiments was a consumer-grade workstation. Specifically, this system is equipped with the following components:

- **CPU**: AMD Ryzen 7 5700X3D, Base: 3.0GHz, Boost: 4.1GHz
- **System Memory**: 16 GB DDR4 at 3000 MHz with CL 16
- **Secondary Storage**: Kingston NVMe SSD, read speed: 3500 MB/s, and write speed: 2800 MB/s

3.6.2 Datasets for Benchmarks without Semi-Streaming

To determine if the proposed algorithms are viable and are able to produce good results in a reasonable amount of time, benchmarks on graphs of different sizes were gathered. These graphs range in size from 1,000 producers and 5,000 consumers up to 1,000,000 producers and 5,000,000 consumers. Above this size, the limiting factor for running further experiments becomes the time complexity of the Jonker-Volgenant algorithm. To be able to evaluate the quality of the results of these algorithms, the graphs generated for these experiments were saved into a single chunk. This was done to be able to make a fair comparison to the optimal solution, which was gathered from the Jonker-Volgenant algorithm. The following datasets were used

for each experiment without semi-streaming, in the order that is presented in the results section.

Table 3.1: The dataset used for the practical approximation ratio and execution time comparison without semi-streaming. In this table and the following table, 'k' stands for $\cdot 10^3$ and 'M' stands for $\cdot 10^6$.

Practical Approximation Ratio and Execution Time Comparison Dataset	
Producers	{1k, 5k, 10k, 25k, 50k, 100k, 1M}
Consumers	{5k, 25k, 50k, 125k, 250k, 500k, 5M}
Sparse Factor	$3 * \log(\|N\|)/\|N\|$
γ**-value**	2
β**-value**	4
Seed	1234

Table 3.2: The dataset used for the γ-value comparison without semi-streaming.

γ**-value Comparison Dataset**	
Producers	100k
Consumers	500k
Sparse Factor	$3 * \log(\|N\|)/\|N\|$
γ**-value**	{2, 3, 4, 8}
β**-value**	2
Seed	1234

Table 3.3: The dataset used for the β-value comparison without semi-streaming.

β**-value Comparison Dataset**	
Producers	100k
Consumers	500k
Sparse Factor	$3 * \log(\|N\|)/\|N\|$
γ**-value**	2
β**-value**	{2, 3, 4, 8}
Seed	1234

Table 3.4: The dataset used for the comparison on real-world data without semi-streaming. Since this is a third-party graph, multiple parameters are unknown.

Dataset for the Comparison on Real-World Data	
Producers	100
Consumers	315
Edges	7518
Sparse Factor	UNKNOWN
γ**-value**	UNKNOWN
β**-value**	UNKNOWN

3.6.3 Semi-Streaming Results Dataset

For the semi-streaming experiments, the ℓ-Local Greedy and the Double ℓ-Local Greedy algorithms were run and evaluated. Firstly, since the Jonker-Volgenant algorithm is unable to run on these large-scale graphs, an estimated optimal solution was calculated based on the findings in the practical approximation ratio comparison in Section 4.1.1. This estimated optimal solution was then used to compare the performance in terms of practical approximation ratio and execution time when running the greedy algorithms with a range of parameters. For these experiments, the standard graph used was generated with 10,000,000 producers, 50,000,000 consumers, 447,917,438 edges, and γ and β-values of 2 and 4, respectively. The following datasets were used for each experiment with semi-streaming, in the order that is presented in the results section.

Notably, two differently sized datasets were used to perform the ℓ-value comparisons due to the excessive execution times of the ℓ-Local Double Greedy algorithm when executed on the larger dataset. Preferably, both algorithms would have been evaluated on the same dataset, but the excessive execution times of the ℓ-Local Double Greedy algorithm, in conjunction with time constraints, prevented this.

Table 3.5: The dataset used for the ℓ-value comparison for the ℓ-Local Greedy algorithm with semi-streaming.

ℓ-value Comparison Dataset Big	
Producers	10M
Consumers	50M
Sparse Factor	$3 * \log(\|N\|)/\|N\|$
γ-value	2
β-value	4
Seed	1234

Table 3.6: The dataset used for the ℓ-value comparison for the ℓ-Local Double Greedy algorithm with semi-streaming.

ℓ-value Comparison Dataset Small	
Producers	10M
Consumers	50M
Sparse Factor	$3 * \log(\|N\|)/\|N\|$
γ-value	2
β-value	4
Seed	1234

3.6.4 Experiments

These tests are performed with and without semi-streaming in order to evaluate the impact of utilizing semi-streaming for the greedy algorithms. To assess the effective-

Table 3.7: The dataset used for the impact of shard parameters evaluation for both ℓ-Local Greedy and ℓ-Local Double Greedy algorithms with semi-streaming.

Impact of Shard Parameters Dataset	
Producers	100k
Consumers	500k
Sparse Factor	$3 * \log(\|N\|)/\|N\|$
γ-value	2
β-value	4
Seed	1234

ness of the implemented greedy algorithms in both contexts, multiple performance metrics are evaluated:

- **Practical Approximation Ratio:** The ratio of the optimal solution's value to the greedy algorithm's obtained practical solution value. This evaluates the solution quality.
- **Execution Time:** What is the actual execution time of the algorithm? This metric is influenced not only by the time complexity and space complexity but also by factors such as the number of passes over each data item and the algorithm's memory access pattern.

However, as approximate algorithms allow for processing of larger graphs than the Jonker-Volgenant algorithm, comparisons of the practical approximation ratio can only be made on graph sizes that both approaches are able to handle. Specifically, the Jonker-Volgenant algorithm can only process graphs up to 6 million nodes within a reasonable execution time. This sets the upper bound of the graph size used in the experimental comparisons that include the Jonker–Volgenant algorithm.

The ℓ-Local Greedy algorithm and the ℓ-Local Double Greedy algorithm are tested with ℓ-values varying from $\ell = 1$, which implies a Naive-Local Greedy algorithm, to $\ell = MAX$, where ℓ is large enough to cover all edges in every neighborhood, which implies it is the basic Local Greedy algorithm. Here the quality of the results, as well as the execution time of different ℓ-values are evaluated and compared.

When not varying the ℓ-value it is set to MAX for consistency and in order to acquire good solution quality during the comparisons. Notably, for some evaluations, such as the β and γ value comparison in Section 4.1.3, it might have been more appropriate to evaluate more ℓ-values, as it could have affected the result of these evaluations. However, this would have required substantially more experiments to be performed within this single evaluation and it was therefore deemed unnecessary, but perhaps this should be looked at in future work.

4

Result

In this chapter, results and comparisons for all implemented algorithms are presented. Firstly, as the limitations of the Jonker-Volgenant algorithm do not support semi-streaming, a performance comparison of all implemented algorithms is made with the use of a single chunk for the graph. This use of a single chunk functions as a fully offline model, as the entire graph can be stored in memory at once. However, this limits the size of the datasets used. After this, evaluation of the specific algorithms is presented. This includes evaluation of the performance in terms of practical approximation ratio and execution time for larger graphs with the use of semi-streaming. It also presents evaluation of the performance based on different parameters set for each specific algorithm implementation.

4.1 Benchmarks without Semi-Streaming

As described in Section 3.6.2, these results are based on non-semi-streaming benchmarks using graphs of increasing size and are compared to the optimal solution provided by the Jonker-Volgenant algorithm.

4.1.1 Practical Approximation Ratio Comparison

In Table 4.1, the practical approximation ratios of all the implemented algorithms are compared on different graph sizes. Since the Jonker-Volgenant algorithm always finds the optimal solution, it has a static practical approximation ratio of 1. The practical approximation ratio of the other algorithms can then be calculated by dividing their respective solution values by the optimal solution value obtained from the Jonker-Volgenant algorithm. This results in a ratio greater than or equal to 1, where values closer to 1 indicate higher solution quality. This method of calculation aligns with the previously discussed approximation ratios, which allows for comparison with the approximation ratios presented in Section 4.1.3. The results show that both the ℓ-Local Greedy and the Double ℓ-Local Greedy algorithms produce close to optimal results at every tested graph size, with average practical approximation ratios of 1.033 and 1.034, respectively. The naive Random Matching algorithm performed far worse, with an average practical approximation ratio of 4.736. This, however, is because of the weight scaling that is used in the generated graphs with the help of γ and β-values, explained in Section 2.4.1. Later, in Section 4.1.4, results from

running the Random Matching algorithm on a graph with a more linearly scaling edge weights achieved better practical approximation ratios.

Table 4.1: Comparison of practical approximation ratio between different algorithms at different dataset sizes without semi-streaming. See Table 3.1 for the dataset used.

Total Nodes	JV	Greedy	Double	Random
6,000	1	1.034	1.035	4.275
60,000	1	1.024	1.037	4.663
150,000	1	1.040	1.034	4.839
300,000	1	1.036	1.031	4.860
600,000	1	1.037	1.033	4.869
6,000,000	1	1.030	1.036	4.909
Average	1	1.033	1.034	4.736

4.1.2 Execution Time Comparison

In addition to gathering the practical approximation ratio for all the algorithms, the execution time was also measured. In Table 4.2, the execution times in seconds for all algorithms are presented, using the same dataset as for the practical approximation ratio comparison.

The results indicate that the Jonker-Volgenant algorithm is consistently the slowest algorithm across all tested problem sizes. This gap in performance becomes wider as the size of the graph grows larger. For the largest graph size of 6,000,000 nodes, the execution time of the Jonker-Volgenant algorithm increases significantly to 3406.60 seconds, while the ℓ-Local Greedy and the Double ℓ-Local Greedy algorithms remain at a more reasonable 50.70 and 80.34 seconds, respectively. The naive approach of the Random Matching algorithm has the lowest execution times across the board, although relative to its practical approximation ratio, this is to be expected.

The difference in execution time between the Jonker-Volgenant algorithm and the implemented algorithms, as the graph grows larger, highlights the challenges of finding the optimal solution for large-scale graphs. On the other hand, the implemented algorithms prove to be far more efficient while still producing close to optimal results, as shown in Section 4.1.1.

Table 4.2: Comparison of execution time in seconds between different algorithms at different dataset sizes without semi-streaming. See Table 3.1 for the dataset used.

Total Nodes	JV	Greedy	Double	Random
6,000	0.09 s	0.02 s	0.04 s	0.02 s
60,000	2.25 s	0.25 s	0.67 s	0.22 s
150,000	8.25 s	0.71 s	1.78 s	0.61 s
300,000	16.50 s	1.54 s	3.92 s	1.37 s
600,000	45.26 s	3.60 s	9.49 s	3.04 s
6,000,000	3406.60 s	50.70 s	80.34 s	45.22 s

4.1.3 γ and β Value Comparison

Figure 4.1 and Figure 4.2 show the impact of different γ and β values on the practical approximation ratio for the ℓ-Local Greedy algorithm and the ℓ-Local Double Greedy algorithm. Especially noteworthy here is that the γ-value impacts the practical approximation ratio more than the β-value does. This is likely due to the fact that there are five times more consumers than producers in this dataset, and therefore each producer has about five times as many neighbors. Thus, it is reasonable that the γ-value has more of an effect than the β-value due to it affecting more neighbors for each producer.

By using Table 2.1 to calculate the approximation ratios for these β and γ values used in the evaluation, we get Table 4.3. This table then allows for comparison between the practical results and the theoretical worst case. When comparing these, it becomes clear that the practical results are significantly better than the Approximation Ratios would suggest. For example, β and γ values of 2 and 4, respectively, result in an approximation ratio of 6 for the ℓ-Local Greedy algorithm and an approximation ratio of 12 for the ℓ-Local Double Greedy algorithm, as opposed to the practical values of 1.021 and 1.019, respectively.

Notably, in this evaluation, the practical approximation ratios improved as the β and γ values increased, instead of worsening as the theory would suggest. The main cause for this is that an increase in the β and γ values leads to more high values edges in the graph. This then means that there are more good edges that the algorithm could select, which reduces the negative impact of the algorithm's greediness. However, this evaluation was only performed on a generated dataset with $\ell = MAX$, as described in Section 3.6.4, which means that the entire neighborhood was always evaluated. This means that the practical results for the ℓ-Local Greedy algorithm should actually be compared with the Approximation Ratios of the Local Greedy algorithm. When that comparison is performed instead, the difference becomes smaller but is still significant.

Table 4.3: Calculated approximation ratios for the different β and γ values that were used in the practical approximation tests. Here Local is the Local Greedy algorithm while ℓ-Local is the ℓ-Local Greedy algorithm and Double is the ℓ-Local Double Greedy algorithm. See Table 3.2 and Table 3.3 for the datasets used.

Approximation Ratios				
β	γ	Local	ℓ-Local	Double
2	2	3	4	8
2	3	3	5	10
2	4	3	6	12
2	8	3	10	20
2	2	3	4	8
3	2	4	5	10
4	2	5	6	12
8	2	9	10	20

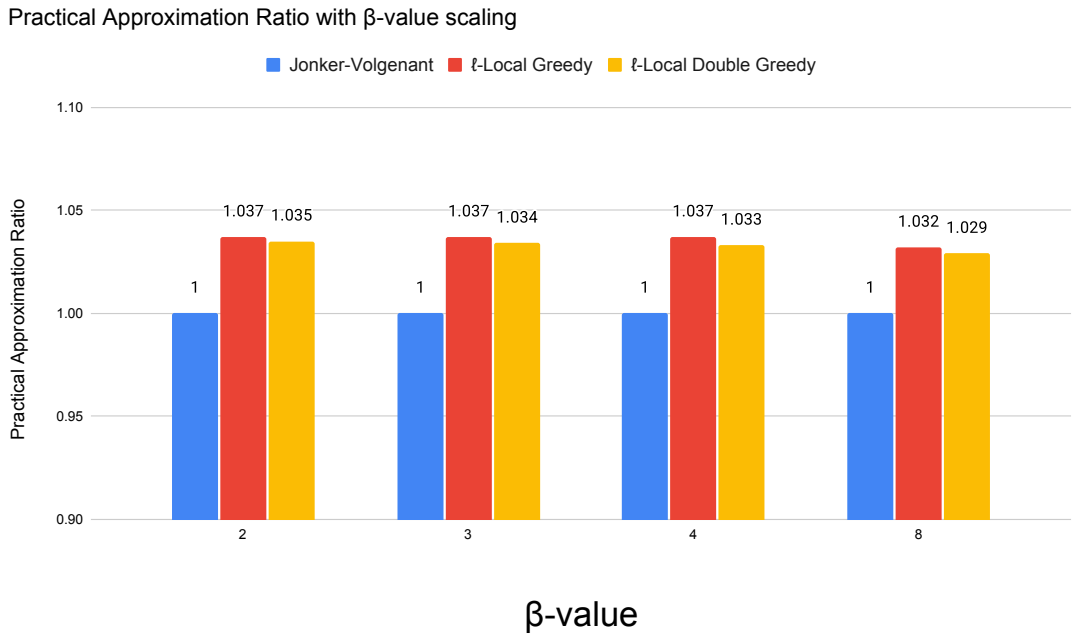

Figure 4.1: Practical approximation ratio for the ℓ-Local Greedy and Double ℓ-Local Greedy algorithms using $\ell = MAX$, with a scaling β-value and a fixed γ-value of 2. See Table 3.3 for the dataset used.

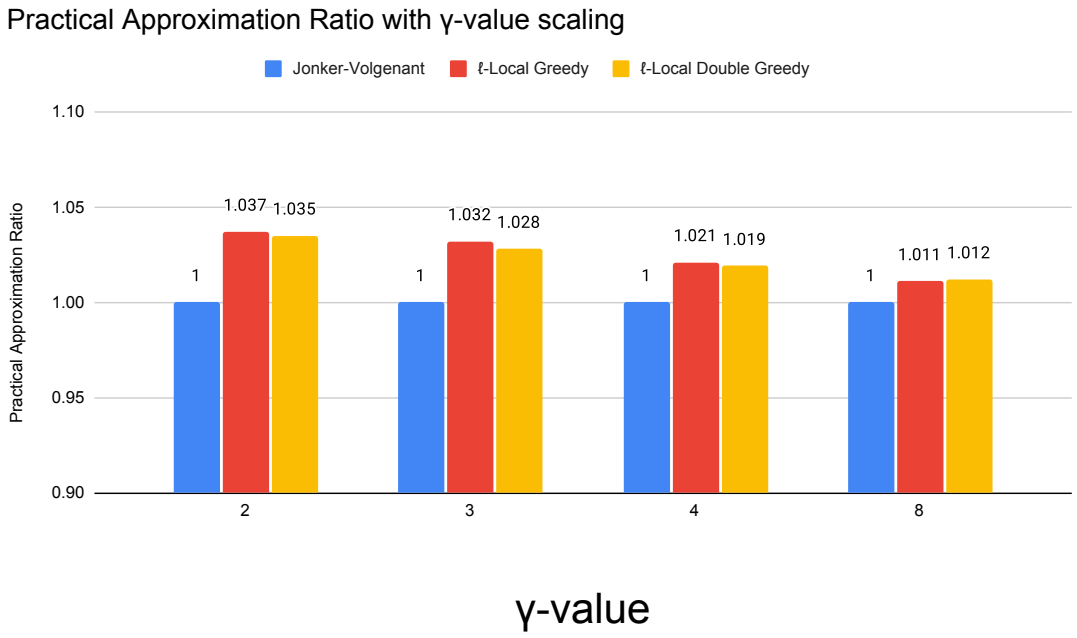

Figure 4.2: Practical approximation ratio for the ℓ-Local Greedy and Double ℓ-Local Greedy algorithms using $\ell = MAX$, with scaling γ-value and a fixed β-value of 2. See Table 3.2 for the dataset used.

4.1.4 Comparison on Real-World Data

From the *SuiteSparse Matrix Collection*, only two weighted bipartite graphs could be found. The largest of them, named *WorldCities*, has a size of 315 by 100 nodes and a total of 7518 edges. Since this is the largest weighted bipartite graph of real-world data in the collection, it was downloaded in the Matrix Market format and run on all implemented algorithms to compare and evaluate performance. As shown in Table 4.4, the small size of the graph makes the execution time almost instant for all algorithms, and the differences at this timescale could be caused by a number of factors outside the algorithms themselves. Notably, this experiment shows that the different variants of the greedy algorithms perform well even on graphs not generated specifically for this thesis.

Although not conclusive because of the small size of the dataset, it shows that greedy algorithms have potential to produce near-optimal results on graphs based on real-world data. Another notable detail is that the Random Matching algorithm performed noticeably better using the *WorldCities* graph compared to using a generated one. This is because the weight distribution of the edges differs between the two graphs. In the *WorldCities* graph, the weights are more linearly distributed, meaning that the average weight of all edges in a neighborhood is closer to the highest possible weight, than what can be found in the generated graphs. As a more aggressive scaling for the edge weights is used when generating graphs, the average weight of a neighborhood is low compared to the highest possible weight, thus resulting in the Random Matching algorithm performing worse.

Table 4.4: Comparison of practical approximation ratio and execution time between different algorithms using the *WorldCities* graph from the *SuiteSparse Matrix Collection* [17]. The algorithms considered here were Jonker-Volgenant(JV), ℓ-Local Greedy, ℓ-Local Double Greedy, and Random Naive algorithms. See Table 3.4 for the dataset used.

WorldCities	**JV**	**Greedy**	**Double**	**Random**
Approx. Factor	1	1.078	1.097	1.683
Execution Time	15 ms	7 ms	9 ms	10 ms

4.2 Semi-Streaming Results

As described in Section 3.6.3, the semi-streaming experiments were conducted using a large-scale synthetic dataset. The performance results for ℓ-Local Greedy and ℓ-Local Double Greedy under varying parameters are summarized below.

4.2.1 ℓ-value Comparison

This section presents the results from testing the ℓ-Local Greedy and ℓ-Local Double Greedy algorithms using ℓ-values ranging from 1 to MAX, where MAX is a high enough value to cover all edges in every neighborhood. A comparison of how the

practical approximation ratio and execution time scale with the ℓ-value is presented, as well as discussing how these relate to each other.

Practical Approximation Ratio with ℓ-value Scaling

In Figure 4.3, the practical approximation ratio for the ℓ-Local Greedy algorithm, depending on the ℓ-value, is presented. Using an ℓ-value of 1, the algorithm works by naively matching with the first available edge in every neighborhood. This produces a practical approximation ratio of 3.06, which is closer to the performance of the naive Random Matching algorithm. This is expected, as both are naive approaches whose performance depends heavily on edge ordering and matching sequence. When increasing the ℓ-value, performance increases accordingly, producing near-optimal results as the ℓ-value approaches MAX.

Since the optimal solution is only an estimation for these large-scale graphs, the main takeaway from these results is that increasing the ℓ-value produces significantly better results compared to keeping the ℓ-value to a minimum. Lower ℓ-values produce a solution that is closer to the naive Random Matching algorithm, while ℓ-values approaching MAX produce significantly better solutions. The important question is whether this drastic difference in solution quality is reflected in the execution time while using the different ℓ-values.

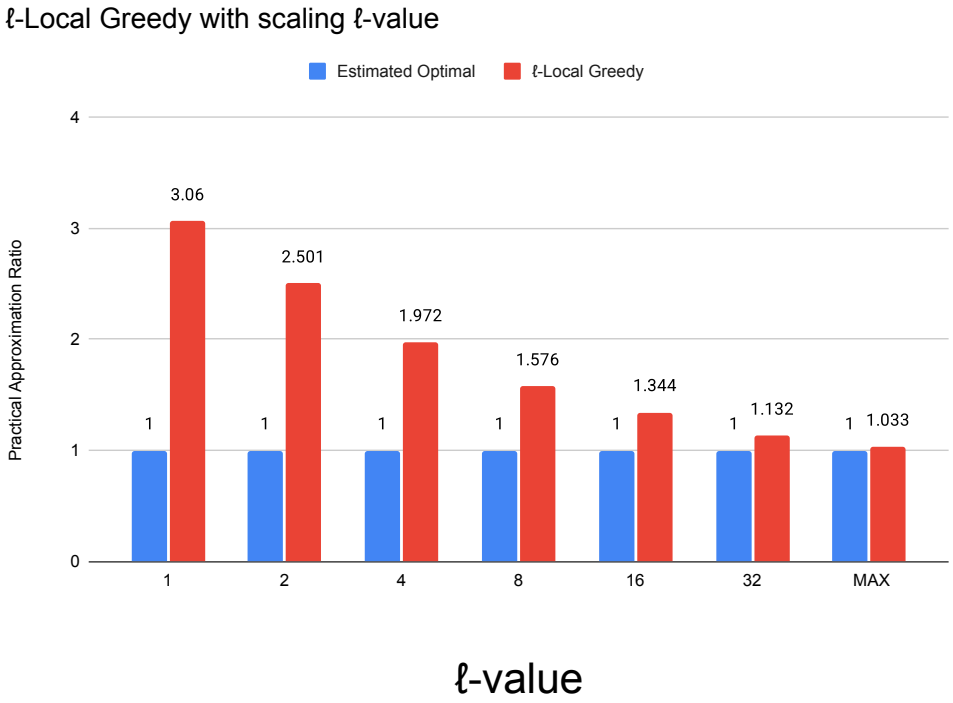

Figure 4.3: Practical approximation ratio for the ℓ-Local Greedy algorithm using different ℓ-values, with a graph-size of 10,000,000 producers and 50,000,000 consumers. See Table 3.5 for the dataset used.

Execution Time with ℓ-value Scaling

In Figure 4.4, the execution time of the ℓ-Local Greedy algorithm with a scaling ℓ-value is presented. The result shows that the execution time gradually increases as higher ℓ-values are used. However, the difference in execution time while using an ℓ-value of 1 and an ℓ-value of MAX, is not large, with only an increase from 363 to 477 seconds. This suggests that using a higher ℓ-value is not excessively detrimental to the execution time, while producing significantly better solutions, as shown in earlier in this section.

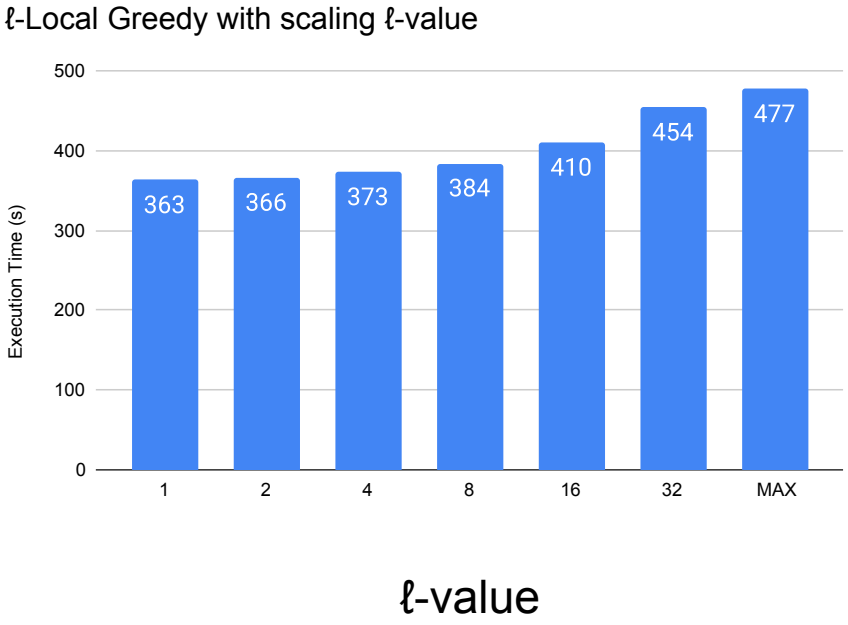

Figure 4.4: The execution time in seconds for the ℓ-Local Greedy algorithm using different ℓ-values, with a graph-size of 10,000,000 producers and 50,000,000 consumers. See Table 3.5 for the dataset used.

The reason for this relatively small increase in execution time is caused by how the algorithm processes the edges from each chunk. As the edges within a chunk can arrive in any order, the entire chunk needs to be loaded into memory. This is because the algorithm cannot determine beforehand which neighborhoods are contained within a specific chunk. Therefore, the entire neighborhood is always loaded into memory, independent of which ℓ-value is being used. The reason for the lower execution times when using smaller ℓ-values is that only a portion of the neighborhood currently in memory is evaluated during matching. This is already a fast operation compared to the slower I/O reads which leads to a small reduction in execution time. A possible optimization here that could improve the execution time further would be to make the ℓ-value impact how many I/O reads are performed. However, this requires the algorithm to have prior knowledge of the neighborhoods of each chunk, specifically which part of the chunk to read into memory and when a chunk has finished matching. Considering this, large fundamental changes to the

algorithm's logic would be necessary for this to work, and were therefore considered outside the scope of this thesis.

4.2.2 ℓ-value Comparison Double Greedy

These tests were carried out in order to determine the impact of the ℓ-value on the ℓ-Local Double Greedy algorithm. In this case, the smaller 1M*5M dataset was used instead of the larger 10M*50M dataset which was used earlier. As described in Section 3.6.3 the reason for why this smaller dataset was chosen was due to the substantially worse execution time of the ℓ-Local Double Greedy algorithm.

Practical Approximation Ratio with ℓ-value Scaling

Figure 4.5 shows the impact of the ℓ-value on the practical approximation ratio. However, in this case, the impact of the ℓ-value is significantly larger, about twice, when compared to the standard ℓ-Local Greedy algorithm. This is reasonable since the ℓ-value now is applied to both the producer and consumer neighborhoods during the execution of the algorithm. As described in Section 3.6.3, the evaluation of the ℓ-value for the ℓ-Local Double Greedy algorithm was performed on a smaller dataset. This somewhat limits the possible comparisons to the ℓ-Local Greedy algorithm, which was evaluated on a larger dataset.

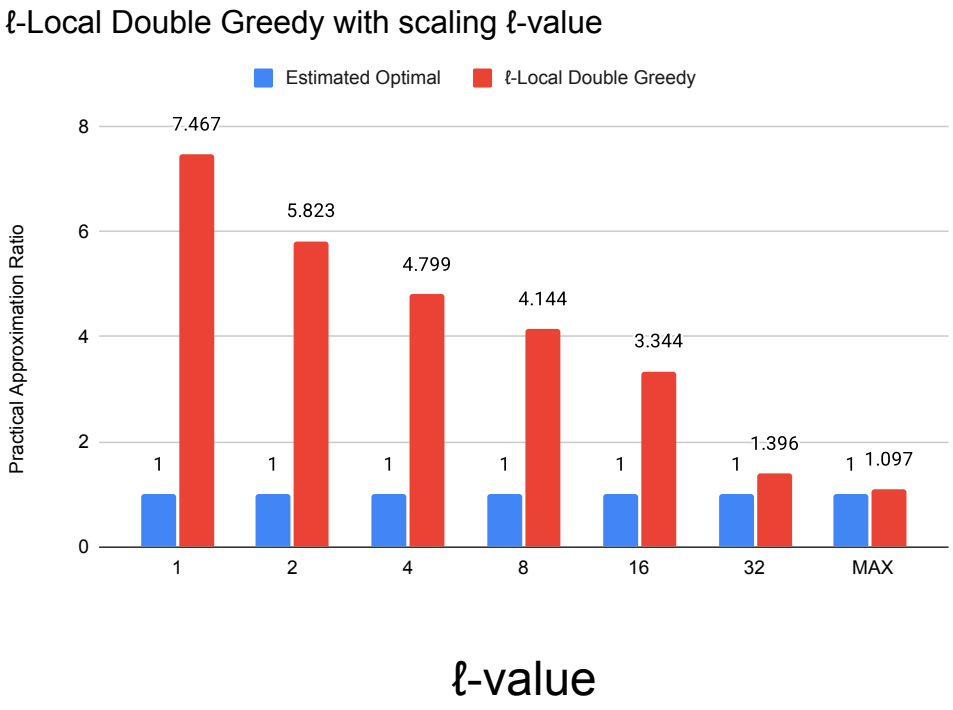

Figure 4.5: Practical approximation ratio for the Double ℓ-Local Greedy algorithm using different ℓ-values, with a graph-size of 1,000,000 producers and 5,000,000 consumers. See Table 3.6 for the dataset used.

Execution Time with ℓ-value Scaling Double Greedy

The same pattern as before holds for the execution time where, as can be seen in Figure 4.6, the ℓ-value once again has a greater impact on the ℓ-Local Double Greedy algorithm than on the standard ℓ-Local Greedy algorithm. Another reason for the greater impact in this case is that the ℓ-value restricts the size of the neighborhoods in the shards, which restricts how many neighbors are added to each shard. This is an expensive operation, so it is reasonable that the execution time decreases quite significantly. Also, as mentioned earlier, this test was performed on the smaller 1M*5M dataset, so it is not unlikely that the impact of the ℓ-value becomes even more significant as the dataset grows larger.

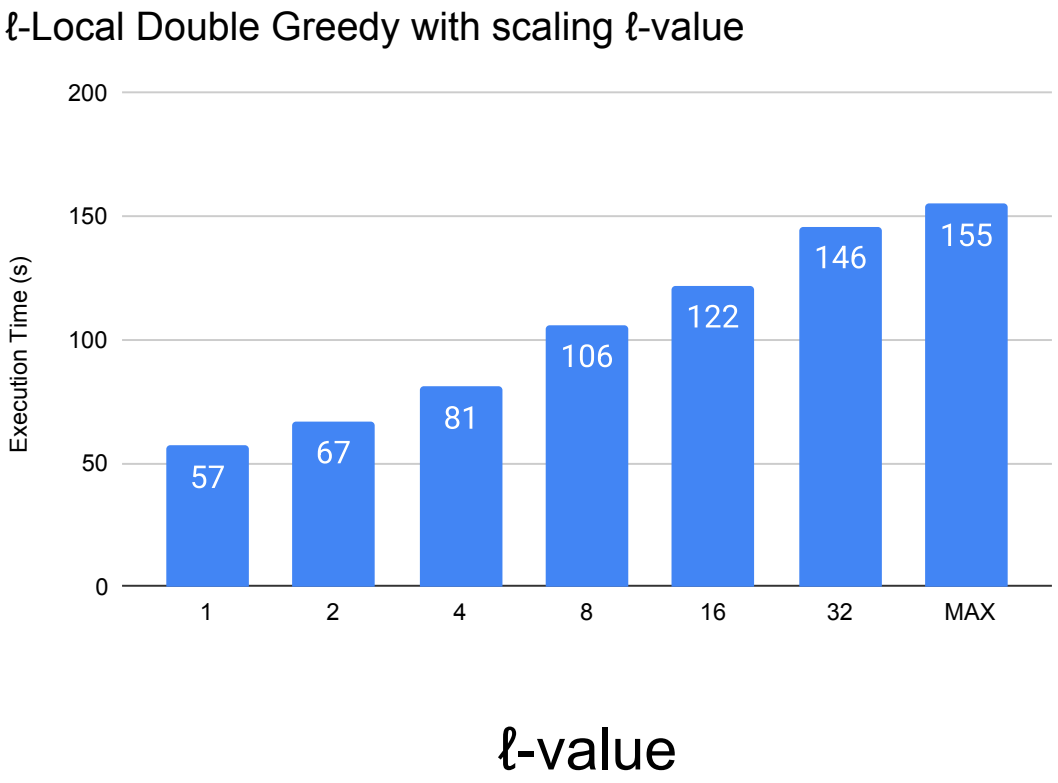

Figure 4.6: The execution time in seconds for the ℓ-Local Double Greedy algorithm using different ℓ-values, with a graph-size of 1,000,000 producers and 5,000,000 consumers. See Table 3.5 for the dataset used.

4.2.3 Impact of Shard Parameters

Table 4.5 shows tests where the different sharding parameters were varied in order to determine whether many small shards or fewer large shards are better for the ℓ-Local Double Greedy algorithm when utilizing the sharding system. All of these tests were performed on the 100k*500k dataset. This smaller dataset was chosen in this case to reduce the overall execution time, which is especially important for the

poor parameter configurations. To make the comparison consistent, all the different variations were limited to storing at most 60,000 neighborhoods in main memory at any time.

Table 4.5: Comparison of how the different values for the shard parameters affect the total execution time as well as the execution time per stage for the ℓ-Local Double Greedy algorithm. Here, NPS is the number of producer shards allowed in memory at any time, while PPS is the amount of producers per producer shard. Similarly, NCS is the number of consumer shards allowed in memory at any time while CCS is the amount of consumers per consumer shard. Notably, all configurations included here are standardized to use the same amount of memory. See Table 3.7 for the dataset used.

ℓ-Local Double Greedy Algorithm							
Parameters					**Execution Time**		
Config	**NPS**	**PPS**	**NCS**	**CCS**	**Total**	**Preprocessing**	**Matching**
1	100	100	500	100	782 s	723 s	59 s
2	100	100	5000	10	463 s	418 s	45 s
3	100	100	50000	1	425 s	380 s	45 s
4	1000	10	50000	1	199 s	185 s	14 s
5	10000	1	50000	1	185 s	174 s	11 s

Initially, it was also intended to include two more test cases for NPS and PPS, as well as two more for NCS and CCS. Specifically, for NPS and PPS these were NPS = 1 and PPS = 10,000 and NPS = 10 and PPS = 1,000. However, these parameters performed so poorly that it was decided to abandon these tests, as the results would be terrible. For example, NPS = 1 and PPS = 10,000 was left running for over an hour and never even finished loading the first chunk. With some limited testing, similar behavior was observed when NCS and CCS were set to 1 or 10, and 10,000 and 1,000, respectively. This, combined with the findings in Table 4.5, suggests that the impact of these parameters is somewhat exponential.

The ideal scenario, where all neighborhoods fit in memory, would obviously outperform all of these listed configurations. However, that is only feasible for graphs with less than about 1M*5M nodes with 16 GB of RAM. Therefore, for graphs larger than that configuration number 5 becomes the clear choice since it performs the best with a limited memory budget. This aligns with the intuitive idea that many smaller shards should perform better than fewer larger shards when the memory access pattern is essentially random. Note that more consumer shards are kept in memory for all these configurations. This was decided since the graph has five times more consumers than producers, so it is less likely that each consumer shard will be accessed when compared to the producer shards. However, more testing would need to be carried out in order to verify this.

The same test was then performed again, but the ℓ-Local Double Greedy algorithm was replaced with the ℓ-Local Greedy algorithm in order, to determine if there are any differences in how the sharding parameters affect these algorithms. However, this time the parameters for consumer shards were ignored because this algorithm only

requires producer neighbors and therefore does not utilize consumer neighborhoods at all. Based on the findings from the previous test, only 3 configurations were tested, as it seemed unlikely for the large shard configurations to perform well. As can be seen in Table 4.6, the configuration focusing on very many minimal shards once again had the best total execution time. However, this time it had a longer execution time for the matching step than the other configurations. Specifically, as can be seen in Table 4.6, the execution time for the matching step seems to scale inversely with the amount of shards. However, this poor scaling is outweighed by the fact that the execution time for the preprocessing step, which is overall longer, seems to scale with the amount of shards.

Table 4.6: Comparison of how the different values for the shard parameters affect the total execution time as well as the execution time per stage for the ℓ-Local Greedy algorithm. Here NPS is the number of producer shards allowed in memory at any time while PPS is the amount of producers per producer shard. Notably, all three configurations use the same amount of memory here. See Table 3.7 for the dataset used.

ℓ-Local Greedy Algorithm with Sharding					
Parameters			**Execution Time**		
Config	**NPS**	**PPS**	**Total**	**Preprocessing**	**Matching**
1	100	100	257.0 s	246.7 s	0.9 s
2	1000	10	55.3 s	53.9 s	1.4 s
3	10000	1	42.7 s	36.8 s	5.8 s

These findings then suggest that many minimal shards are also the preferred solution here. However, this would also need to be verified on larger datasets, as these larger datasets are typically more matching-bound than preprocessing-bound, based on our observations. Another implication of this result, as well as the result from the previous test, is that using the sequential hash function described in Section 3.5.3 to determine which shard a producer or consumer belongs to is unlikely to be beneficial at all. Since if only one neighborhood is stored in each shard, it will no longer place sequential producers in the same shard. Therefore, it might be better to instead select the other hash function option mentioned, which was $S = P/N$, or to possibly explore a more advanced third-party solution in case the current solution is causing a performance loss.

5

Summary and Conclusion

This chapter presents a summary of the results for the experiments run on the implemented algorithms for the maximum weight bipartite matching problem. The algorithms ℓ-Local Greedy, ℓ-Local Double Greedy, and Naive Random Matching algorithm, were benchmarked against the optimal Jonker-Volgenant algorithm. The evaluation covers both fully offline (single chunk) and semi-streaming models, assessing practical approximation ratios and execution times. It also discusses future possible improvements to the graph generation and greedy implementations.

5.1 Benchmarks without Semi-Streaming

- **Practical Approximation Ratio:** The ℓ-Local Greedy and Double ℓ-Local Greedy algorithms consistently achieved near-optimal practical approximation ratios, around 1.03, across graphs with sizes varying from 6,000 to 6,000,000 nodes, which indicates a high solution quality.

- **Execution Time:** Jonker-Volgenant was consistently the slowest algorithm, with its execution time increasing significantly for larger graphs. For example, 3406 seconds with 6,000,000 nodes. In contrast, the ℓ-Local Greedy and Double ℓ-Local Greedy algorithms remained significantly more efficient with, for example, 50 to 80 seconds of execution time for 6,000,000 nodes, which highlights their scalability advantages.

- **γ- and β-Value Comparison:** The γ-value had a greater impact on the practical approximation ratio than the β-value, likely due to the graph's structure, which has five times more consumers than producers. A key finding was that the practical approximation ratios achieved by the greedy algorithms were substantially better than their theoretical worst-case guarantees.

- **Real-World Data:** On a small real-world graph (*WorldCities*), the greedy algorithms still produced near-optimal results, with practical approximation ratios of 1.08 to 1.09. Also, the Random Matching algorithm performed much better than on generated data, with a practical approximation ratio of 1.68, which is likely due to a more linear edge weight distribution in this real-world dataset.

5.2 Semi-Streaming Results

- **ℓ-value Comparison:**
  - **Practical Approximation Ratio:** For the ℓ-Local Greedy algorithm, increasing the ℓ-value from 1, a naive solution, to MAX significantly improved the practical approximation ratio from 3.06 to near-optimal. A similar trend was observed for the Double Greedy, especially when the ℓ-values were high, greater than 32.
  - **Execution Time:** For the ℓ-Local Greedy algorithm, increasing the ℓ-value caused only a moderate increase in execution time. For example, from 363 seconds for $\ell = 1$ to 477 seconds for $\ell = MAX$. This is because the primary bottleneck remains I/O reads to load full chunks, not the in-memory processing of neighborhoods. The impact on execution time was more significant for the Double Greedy due to its preprocessing stage.
- **Impact of Shard Parameters:** Analysis of shard parameters for the Double Greedy and Greedy algorithms revealed that using many smaller shards (for example, one producer neighborhood per producer shard) significantly improved overall execution time by optimizing memory access patterns, especially for the preprocessing stage. This approach proved more efficient than using fewer but larger shards when memory is limited.

5.3 Conclusion

Research Question: Is it viable to utilize the greedy algorithms on very large bipartite graphs, and how do their results compare to the naive and optimal solutions?

This thesis has evaluated the performance and scalability of ℓ-Local Greedy and Double ℓ-Local Greedy algorithms for solving the maximum weight bipartite matching problem, particularly focusing on their viability in semi-streaming environments for large-scale graphs.

The results clearly demonstrate that these greedy algorithms offer a good balance between solution quality and computational efficiency. They consistently achieve near-optimal practical approximation ratios of around 1.03 while being orders of magnitude faster than the optimal Jonker-Volgenant algorithm for large graphs. This makes them highly practical for scenarios where finding a near-optimal solution is sufficient.

Crucially, the semi-streaming implementation enables these algorithms to handle graphs that far exceed available memory. Parameter evaluations highlighted that increasing the ℓ-value significantly boosts solution quality with only a marginal impact on execution time for the ℓ-Local Greedy algorithm, as I/O operations dominate the runtime. For the ℓ-Local Double Greedy, the impact of the ℓ-value was more significant. Furthermore, optimizing shard parameters by favoring numerous smaller

shards was shown to drastically reduce execution times by optimizing memory access patterns, especially during the preprocessing stage.

A significant observation was that the practical approximation ratios achieved by these greedy algorithms were substantially better than their theoretical approximation ratios, which demonstrates their viability for large-scale graphs. While direct comparison to an optimal solution was not feasible for the largest semi-streaming graphs, the consistent and strong performance across varied datasets and with different parameter settings further validates the viability of these greedy approaches.

In conclusion, the ℓ-Local Greedy and ℓ-Local Double Greedy algorithms are robust, scalable, and near-optimal solutions for maximum weight bipartite matching in a semi-streaming environment, which makes them highly valuable tools for large-scale data processing and analysis. However, when comparing the two algorithms, the ℓ-Local Greedy algorithm becomes the better choice due to its lower memory usage and lower execution times while still achieving excellent solution quality.

5.4 Future Work

Currently, the implementation uses 32-bit unsigned integers in order to conserve memory. This leads to issues when graphs contain edges in the billions. The best possible solution here would be to allow the user to select which data type to use for the producer IDs, consumer IDs, and weights. This then also opens up the option of using smaller data types when possible in order to save memory as well as using larger ones such as, for example, 128-bit floating-point numbers. However, implementing this would require significant changes to the source code.

Another possibility is to perform low-level cache optimization, since the performance of the ℓ-Local Double Greedy algorithm was significantly improved by optimizing its high-level memory behavior. Specifically, if the access patterns could be better understood or fixed in some manner, then it might be possible to reduce the I/O load for this algorithm substantially.

This also highlights one of the main challenges for the ℓ-Local Double Greedy algorithm, which is the observed neighborhood locality in Section 2.6. Suppose that instead a different type of graph file format existed that better accommodated the ℓ-Local Double Greedy algorithm's locality needs. In that case, it would likely achieve better execution times and possibly even become a proper alternative to the ℓ-Local Greedy algorithm.

Since the ℓ-Local Greedy algorithm works locally within each neighborhood and traverses the graph sequentially, it could possibly be parallelized. In order to achieve this, a shared data-structure would be needed to keep track of all available consumers. Due to the limited memory usage of the Local Greedy algorithm, this could be a substantial optimization and possibly reduce the execution time significantly.

Parallelization could also be possible for the ℓ-Local Double Greedy algorithm. However, as mentioned earlier, it takes an essentially random path through the graph, which makes it hard to predict how different threads could affect each other. It

would also need another shared data-structure to keep track of which producers are matched. Therefore, we would suggest working with the standard ℓ-Local Greedy algorithm instead.

Bibliography

[1] H. W. Kuhn, "The hungarian method for the assignment problem," *Naval Research Logistics Quarterly*, vol. 2, no. 1-2, pp. 83–97, 1955. DOI: https://doi.org/10.1002/nav.3800020109. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/nav.3800020109. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/nav.3800020109.

[2] J. Edmonds and R. M. Karp, "Theoretical improvements in algorithmic efficiency for network flow problems," *J. ACM*, vol. 19, no. 2, pp. 248–264, Apr. 1972, ISSN: 0004-5411. DOI: 10.1145/321694.321699. [Online]. Available: https://doi.org/10.1145/321694.321699.

[3] Y. Dinitz, "Dinitz' algorithm: The original version and even's version," Jan. 2006, pp. 218–240, ISBN: 978-3-540-32880-3. DOI: 10.1007/11685654_10.

[4] R. Jonker and T. Volgenant, "A shortest augmenting path algorithm for dense and sparse linear assignment problems," *Computing*, vol. 38, pp. 325–340, 1987. [Online]. Available: https://api.semanticscholar.org/CorpusID:7806079.

[5] K. Date and R. Nagi, "Gpu-accelerated hungarian algorithms for the linear assignment problem," *Parallel Computing*, vol. 57, pp. 52–72, 2016, ISSN: 0167-8191. DOI: https://doi.org/10.1016/j.parco.2016.05.012. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S016781911630045X.

[6] T. Bressoud, R. Rastogi, and M. Smith, "Optimal configuration for bgp route selection," in *IEEE INFOCOM 2003. Twenty-second Annual Joint Conference of the IEEE Computer and Communications Societies (IEEE Cat. No.03CH37428)*, vol. 2, 2003, 916–926 vol.2. DOI: 10.1109/INFCOM.2003.1208929.

[7] Gavish and Pirkul, "Computer and database location in distributed computer systems," *IEEE Transactions on Computers*, vol. C-35, no. 7, pp. 583–590, 1986. DOI: 10.1109/TC.1986.1676799.

[8] R. Duvignau, V. Gulisano, and M. Papatriantafilou, "Efficient and scalable geographical peer matching for p2p energy sharing communities," in *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, ser. SAC '22, Virtual Event: Association for Computing Machinery, 2022, pp. 187–190, ISBN: 9781450387132. DOI: 10.1145/3477314.3507203. [Online]. Available: https://doi.org/10.1145/3477314.3507203.

[9] M. Ghaffari and D. Wajc, *Simplified and space-optimal semi-streaming for $(2+\epsilon)$-approximate matching*, 2018. arXiv: 1701.03730 [cs.DS]. [Online]. Available: https://arxiv.org/abs/1701.03730.

[10] N. Tomizawa, "On some techniques useful for solution of transportation network problems," *Networks*, vol. 1, no. 2, pp. 173–194, 1971. DOI: https://doi.org/10.1002/net.3230010206. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/net.3230010206. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/net.3230010206.

[11] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang, "On graph problems in a semi-streaming model," *Theoretical Computer Science*, vol. 348, no. 2, pp. 207–216, 2005, Automata, Languages and Programming: Algorithms and Complexity (ICALP-A 2004), ISSN: 0304-3975. DOI: https://doi.org/10.1016/j.tcs.2005.09.013. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0304397505005323.

[12] L. Epstein, A. Levin, J. Mestre, and D. Segev, "Improved approximation guarantees for weighted matching in the semi-streaming model," *SIAM Journal on Discrete Mathematics*, vol. 25, no. 3, pp. 1251–1265, 2011. DOI: 10.1137/100801901. eprint: https://doi.org/10.1137/100801901. [Online]. Available: https://doi.org/10.1137/100801901.

[13] A. Paz and G. Schwartzman, "A (2+)-approximation for maximum weight matching in the semi-streaming model," *ACM Trans. Algorithms*, vol. 15, no. 2, Dec. 2018, ISSN: 1549-6325. DOI: 10.1145/3274668. [Online]. Available: https://doi.org/10.1145/3274668.

[14] R. Duvignau and R. Klasing, "Greediness is not always a vice: Efficient discovery algorithms for assignment problems," *Procedia Computer Science*, vol. 223, pp. 43–52, 2023, XII Latin-American Algorithms, Graphs and Optimization Symposium (LAGOS 2023), ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2023.08.212. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050923009857.

[15] R. Duvignau, N. Gillet, and R. Klasing, *Greediness is not always a vice: Efficient discovery algorithms for assignment problems*, 2024. arXiv: 2410.09434 [cs.DS]. [Online]. Available: https://arxiv.org/abs/2410.09434.

[16] NIST, *Matrix market*, https://math.nist.gov/MatrixMarket/, Accessed: 2025-05-08, 2007.

[17] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, Dec. 2011, ISSN: 0098-3500. DOI: 10.1145/2049662.2049663. [Online]. Available: https://doi.org/10.1145/2049662.2049663.

[18] P. Virtanen, R. Gommers, T. E. Oliphant, *et al.*, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020. DOI: 10.1038/s41592-019-0686-2.

[19] D. F. Crouse, "On implementing 2d rectangular assignment algorithms," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 52, no. 4, pp. 1679–1696, 2016. DOI: 10.1109/TAES.2016.140952.

A

Appendix - AI Usage

During the thesis work, a few different LLMs have been utilized. For example, different versions of GPT and Gemini were used as an aid when debugging and optimizing code. For proofreading the thesis, both ChatGPT and Gemini, as well as LanguageTool were used in order to help correct grammar and spelling mistakes.