Engineering Buffered Algorithms for NeighbourCover

Felix Wörner

October 1, 2023

4016508

Bachelor Thesis

at

Algorithm Engineering Group Heidelberg
Heidelberg University

Supervisor:
Univ.-Prof. PD. Dr. rer. nat. Christian Schulz

Co-Supervision:
Marcelo Fonseca Faraj
Ernestine Großmann
Henrik Reinstädtler

Hiermit versichere ich, dass ich die Arbeit selbst verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und wörtlich oder inhaltlich aus fremden Werken Übernommenes als fremd kenntlich gemacht habe. Ferner versichere ich, dass die übermittelte elektronische Version in Inhalt und Wortlaut mit der gedruckten Version meiner Arbeit vollständig übereinstimmt. Ich bin einverstanden, dass diese elektronische Fassung universitätsintern anhand einer Plagiatssoftware auf Plagiate überprüft wird.

Heidelberg, October 1, 2023

Felix Wörner

Abstract

In [19] Veldt introduces the NeighbourCover algorithm and shows that it is a probabilistic 2-approximation for the NP hard vertex cover problem, in both the weighted and unweighted case. Furthermore, they extend this approach to design probabilistic 2-approximation algorithms for 3 additional problems, namely minimum delete to matching, DAG edge deletion and edge-colored hypergraph clustering. In this work we introduce buffered versions of NeighbourCover for all the stated problems. Moreover, we introduce further buffered algorithms for these problems by modifying existing offline algorithms. We show that buffered NeighbourCovers solution quality is on average 15% better for minimum weighted vertex cover, 5% better for DAG edge deletion and 24% better for colored edge clustering, than the compared algorithms.

Contents

Abstract iii

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Our Contribution . . . 2
- 1.3 Structure . . . 2

2 Fundamentals 3
- 2.1 General Definitions . . . 3
- 2.2 Problem Definitions . . . 6

3 Related Work 7
- 3.1 NeighbourCover Algorithms . . . 7
  - 3.1.1 NeighborCoverVC Algorithm . . . 7
  - 3.1.2 NeighbourCoverMinD2M . . . 8
  - 3.1.3 NeighbourCoverDED2 . . . 9
  - 3.1.4 NeighbourCoverColorEC . . . 10
- 3.2 Competing Algorithms . . . 11

4 Buffered Approach To NeighbourCover 13
- 4.1 Buffered NeighbourCoverVC . . . 13
- 4.2 Buffered NeighbourCover variants for MinD2M, DED2 and ColorEC . . . 15

5 Competing Algorithms 17
- 5.1 Original Algorithms . . . 17
- 5.2 Modifications on Competing Algorithms . . . 17

6 Experimental Evaluation 21
- 6.1 Setup . . . 21
- 6.2 Determining Parameters for VC Algorithms . . . 24
- 6.3 VC Algorithms Comparison . . . 26
- 6.4 Determining Parameters for Matching Algorithms . . . 27

6.5 Matching Algorithms Comparison 30
6.6 Determining Parameters for DED2 Algorithms 31
6.7 DED2 Algorithms Comparison 32
6.8 Determining Parameters for ColorEC Algorithms 33
6.9 ColorEC Algorithms Comparison 34

7 Discussion **35**
7.1 Conclusion 35
7.2 Future Work 35

Abstract (German) **37**

Bibliography **39**

CHAPTER 1

Introduction

Graphs can be used as a model in a plethora of different use cases. Some examples that can be represented by a graph are street networks, social networks, data dependency in applications or the design of application specific integrated circuits. Since the information we gather tends to increase, graphs can often be huge and require large amounts of memory. This is why calculating a solution requires an effective algorithm that not only takes into account the specific problem, but also the machine it is computed on.

1.1 Motivation

When designing an algorithm, you have to consider the three factors running time, solution quality and memory usage. There are several models like the offline, buffered streaming and streaming algorithms that prioritize different factors. The offline model loads the entire graph into memory. This provides algorithms which return high quality solutions, but when working on huge graphs they also need a lot of memory. Therefore, if working on a machine with enough memory is not possible, an offline algorithm is not an option. Instead, a buffered streaming or streaming approach is necessary, because they only have a fraction of the graph in memory at a time. They require less memory and usually execute faster at the cost of solution quality.
The NeighbourCover algorithm was introduced by [19] from Veldt. It is an offline probabilistic 2-approximation algorithm for both NP hard problems of unweighted and weighted vertex cover. In the same paper Veldt also adapts the algorithm to design offline probabilistic 2-approximation algorithms for minimum delete to matching, directed acyclic graph (DAG) edge deletion and edge-colored hypergraph clustering. Since all of these algorithms are offline algorithms, they require a machine with a lot of memory, for large graphs. Most people do not have access to expensive machines, which is why versions of the algorithms which need less memory are of interest. For this purpose, we introduce a buffered streaming version for each of the NeighbourCover algorithms.

1.2 Our Contribution

We expand the original paper [19] by Veldt by converting the NeighbourCover algorithms from an offline, to a buffered streaming approach. Furthermore, we introduce other streaming or buffered streaming algorithms for the same problems, which are based on offline algorithms. We then implement all the algorithms and compare them on a benchmark set of graphs. This allows us to find out how well the NeighbourCover algorithms translate to a buffered approach in comparison to other algorithms. In our experiments we show that for minimum vertex cover, DAG edge deletion and colored edge clustering the buffered NeighbourCover algorithms will achieve a higher quality than their competitors, at the cost of a higher running time. In particular their solution quality is on average 15% better for minimum weighted vertex cover, 5% better for DAG edge deletion and 24% better for colored edge clustering, than the compared algorithms.

1.3 Structure

The remainder of this thesis is organized as follows. Chapter 2 contains the basic terms and it explains the problems the algorithms work on. Chapter 3 explains all algorithms in their basic form. The explanation of how the algorithms were modified can be found in Chapter 4. Lastly, Chapter 6 explains the setup of our experiments and shows our results. Lastly in Chapter 7 we discuss our findings and present our conclusion.

CHAPTER 2

Fundamentals

This chapter is meant to introduce the fundamental concepts needed to understand this thesis. It is split into two separate sections. Section 2.1 gives definitions needed for later chapters. Section 2.2 defines the different problems the algorithms are designed for.

2.1 General Definitions

This section provides the definitions for nontrivial concepts needed to understand this thesis.

Offline Algorithm. An offline algorithm loads the entire graph into memory. This is different to online algorithms which at any given point only have a fraction of the graph in memory. Online models are for example streaming, semi streaming and buffered streaming.

Buffered Streaming Algorithm. A buffered algorithm has a buffer of a fixed size. It reads in the graph sequentially and stores it in the buffer until it is full. Then it can use the stored data for computation. After it is done with the current batch, the batch is cleared and the next part of the graph is loaded. A buffered streaming algorithm is limited to $O(n)$ space.

Streaming Algorithm. A streaming algorithm sequentially receives only one node and its neighbourhood at a time. It is limited to $O(n)$ space.

Semi Streaming Algorithm. A semi streaming algorithm sequentially receives only one node and its neighbourhood at a time. It is limited to $O(n \text{ polylog } n)$ space. This model was proposed by Feighenbaum et al. [10].

Approximation Algorithm. An approximation algorithm has an approximation factor ρ. For each possible input I the solution provided by the algorithm $f(I)$ is within factor ρ of the optimal solution OPT. Formally this can be defined as follows:

$$f(I)/OPT \geq \rho, \quad \text{if } \rho > 1$$

$$f(I)/OPT \leq \rho, \quad \text{if } \rho < 1$$

A probabilistic approximation algorithm is a generalization of an approximation algorithm, where only the expected value of the algorithm has to be within factor ρ of OPT.

Undirected Graph. A graph $G = (V, E)$ is defined by a finite set of nodes V and its edges E. An intuitive way to think about an edge is as a line connecting two nodes. Formally the set of edges can be defined as :

$$E \subseteq V \times V$$

Line Graph. The line graph $L(G)$ can be constructed out of an unweighted graph G in the following way: In the line graph create a corresponding node for every edge of G. Two nodes of the line graph are connected by an edge, if the edges in G they correspond to, have a node in common.

Directed Graph. A directed graph $D(V, A)$ is defined by its set of nodes V and its directed edges A. The difference to an undirected graph is that the edges now have a direction. An intuitive way to think about an edge in a directed graph is as an arrow starting at one node and pointing to another. This means the edges are defined as an ordered pair of nodes, instead of a set. We will commonly refer to the first node as start node and the second as target node. Formally, the set of edges of a directed graph can be defined as:

$$A \subseteq \{(x, y) | x, y \in V\}$$

Weighted Graph. An undirected graph $G = (V, E)$ can be node weighted, edge weighted or both. It is node weighted if it has a weight function $W_v : V \to \mathbb{N}$ that assigns every node $v \in V$ a weight w_v. Similarly, in an edge weighted graph a weight function $W_e : E \to \mathbb{N}$ assigns every edge $e \in E$ a weight w_e. The concept of a weighted graph can be defined analogously for directed graphs and hypergraphs.

Directed Paths and Directed Trails. In a directed graph $D(V, A)$ a sequence of edges $e_1, e_2, ..., e_{n-1} \in A$ is a directed path of length $(n-1)$, if there is a sequence of distinct nodes $v_1, v_2, ..., v_n \in V$ such that the following holds:

$$\forall\ i \in \{1, ..., n-1\} : e_i = (v_i, v_{i+1})$$

A directed trail of length $(n - 1)$ is a relaxed version of a path, where the edges $e_1, e_2, ..., e_{n-1} \in A$ need to be distinct, but the nodes $v_1, v_2, ..., v_n \in V$ do not. In this work we will usually refer to directed paths or directed trails as paths or trails.

Directed Cycles. In a directed graph $D(V, A)$ if there is a trail $e_1, e_2, ..., e_n \in A$ with node sequence $v_1, v_2, ..., v_n, v_1 \in V$ and the only repeating node is v_1, then we call it a directed cycle. In this work we will usually refer to directed cycles as cycles.

Directed Acyclic Graph (DAG). A directed graph which does not contain any directed cycles is called a directed acyclic graph or in short DAG.

Hypergraph. A hypergraph $H(V, \mathcal{E})$ is defined by its set of nodes V and its hyperedges $\mathcal{E}$. Unlike a normal edge that contains exactly two nodes, a hyperedge can contain an arbitrary amount of pairwise distinct nodes. Formally, the set of hyperedges of a hypergraph can be defined as:

$$\mathcal{E} \subseteq \{h | h \subseteq V\}$$

Colored Hypergraph. Let $H(V, \mathcal{E}, l, k)$ be a colored hypergraph. The number of different colors is denoted by k and $l : \mathcal{E} \to \{1, 2, ..., k\}$ is the function that assigns each edge one of those k colors.

Weighted Shuffle. A weighted shuffle is performed on a set of n elements $X = \{x_1, ..., x_n\}$ with weights $w_i > 0 : \forall i = 1, ..., n$ assigned to them. It generates a permutation of the elements by repeating a weighted sample without replacement n times. Let U be the set of elements which were not sampled yet and s the sequence which will become our permutation. We start with U containing all elements and s being empty. The weighted sample picks one of the undecided elements in U with probability $p(x_i) = w_i / (\sum_{j:x_j \in U} w_j)$, removes it from U and adds it to the end of sequence s. After repeating the weighted sample n times there are no undecided nodes left and the sequence s is a permutation of all elements in X.

Independent Set On an undirected Graph $G = (V, E)$ a subset of nodes $I \subseteq V$ is an independent set, if no two nodes in I are connected by an edge. Formally this can be defined as:

$$\neg\exists\, e = \{u, v\} \in V : u \in I \land v \in I$$

2.2 Problem Definitions

This section provides a definition for the 4 different problems we designed our algorithms for. Namely minimum vertex cover, minimum delete to matching, DAG edge deletion with parameter k and colored edge clustering.

Minimum Vertex Cover (VC). Given an undirected graph $G = (V, E)$, a set of the nodes $C \subseteq V$ is a vertex cover if:

$$\forall uv \in E \Rightarrow u \in C \vee v \in C$$

That means for every edge, there is at least one endpoint that is contained in C. In the case of an unweighted graph, a minimum vertex cover is a cover C, with the lowest possible cardinality. If the graph is node weighted, a minimum vertex cover is a cover C, where the sum of all node weights in C is minimized. The complement of a vertex cover is always an independent set.

Minimum Delete to Matching (MinD2M). Given an undirected graph $G = (V, E)$, a set of edges $M \subseteq E$ is a matching, if all edges $e = \{u, v\}$ are disjoint. Formally, this can be defined as:

$$\forall e_i, e_j \in M : \ i \neq j \Rightarrow e_i \cap e_j = \emptyset$$

MinD2M is an abbreviation for Minimum Delete to Matching. Given an undirected edge weighted graph $G = (V, E, W_e)$, the goal is to find a minimum weight subset of edges $\mathcal{D} \subseteq E$ to delete, such that the remaining edges in the graph form a matching. Conversely, we can formulate the goal as maximizing the weight of the edges in the matching.

DAG Edge Deletion with Parameter k (DED-k). Let $D(V, A, W_e)$ be an edge weighted DAG. The goal of DED-k is to find a minimum weight subset of edges $\mathcal{D} \subseteq A$ to delete, such that in the remaining graph there are no paths of length k left. In our case we will always choose parameter $k = 2$.

Colored Edge Clustering (ColorEC). Let $H(V, \mathcal{E}, l, k, W_e)$ be a colored edge weighted hypergraph. We can assign each node a color. An edge is considered unsatisfied if not every node in the edge has the same color as the edge. The objective is to minimize the weight of unsatisfied edges.

CHAPTER 3

Related Work

This chapter is split into two sections. Section 3.1 explains the NeighbourCover algorithms from Veldt [19]. Section 3.2 summarizes the algorithms we compare NeighbourCover against.

3.1 NeighbourCover Algorithms

In [19] Veldt introduces the NeighbourCover algorithm. It is a probabilistic 2-approximation algorithm for the minimum weight vertex cover problem. Furthermore, Veldt proposes algorithms for minimum delete to matching, directed acyclic graph edge deletion with the parameter k=2 and colored hypergraph edge clustering. All of these algorithms are based on the NeighbourCover algorithm and are probabilistic 2-approximations for their respective problems. Every NeighbourCover version uses a weighted shuffle as a sub step. When implementing the weighted shuffle for our algorithms we used the version from Wong and Easton [21].

3.1.1 NeighborCoverVC Algorithm

The NeighbourCover algorithm works on an undirected node weighted graph. It is a 2-approximation algorithm for minimum vertex cover (2.2).

Algorithm. Given an undirected node weighted graph $G = (V, E, W_v)$. The NeighbourCoverVC algorithm first assigns all nodes to the undecided node set U while leaving the independent set I and cover C empty. Algorithm 1 provides pseudocode.

Algorithm 1 NeighbourCoverVC

$C \leftarrow \emptyset, I \leftarrow \emptyset, U \leftarrow V$
nodePermutation $\leftarrow$ WEIGHTEDSHUFFLE$(w_1, w_2, ..., w_{|V|})$
for v *in nodePermutation* **do**
  if $v \in U$ **then**
    $U \leftarrow U \backslash (\{v\} \cup N(v))$
    $I \leftarrow I \cup v$
    $C \leftarrow C \cup N(v)$
return C

3.1.2 NeighbourCoverMinD2M

NeighbourCoverMinD2M works on an undirected edge weighted graph. It is a probabilistic 2-approximation algorithm for minimum delete to matching (2.2). The idea of the algorithm is, that the MinD2M objective has the same results as finding a maximum independent set of the line graph of G. This means, forming the line graph and then using NeighbourCover on it, already leads to a 2-approximation algorithm. A line graph L can be constructed out of the original graph $G(V, E, W_e)$ in the following way: For all edges $e \in E$ of the original graph create a vertex in L. Two vertices in L are connected, if their corresponding edges in G share a vertex. An example of forming the line graph out of the original graph can be found in Figure 3.1. However, this approach takes $O(|E|^2)$ time, since it is necessary to explicitly form the line graph. NeighbourCoverMinD2M improves on this idea, by implicitly iterating through all the vertices of the line graph and keeping track of the nodes V_M, which have already been matched in the original graph. Thereby, it achieves a time complexity of $O(|E| \log |E|)$ in the weighted case and $O(|E|)$ in the unweighted case.

Algorithm. Given an undirected edge weighted graph $G = (V, E, W_e)$, the algorithm keeps track of all deleted edges $\mathcal{D}$ and the vertices of the matched edges V_M. Both of the sets $\mathcal{D}$ and V_M start empty. Algorithm 2 provides pseudocode.

Algorithm 2 NeighbourCoverMinD2M

$\mathcal{D} \leftarrow \emptyset, V_M \leftarrow \emptyset$
edgePermutation $\leftarrow$ WEIGHTEDSHUFFLE$(w_1, w_2, ..., w_{|E|})$
for $e = \{u, v\}$ *in edgePermutation* **do**
  if $v \in V_M$ *or* $u \in V_M$ **then**
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{e\}$
  else
    $V_M \leftarrow V_M \cup \{u, v\}$
return $\mathcal{D}$

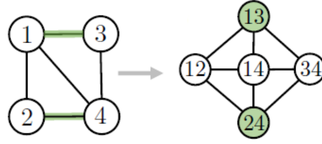


Figure 3.1: Transformation graph to line graph. Reprinted from [19].

3.1.3 NeighbourCoverDED2

NeighbourCoverDED2 works on a DAG. It is a probabilistic 2-approximation algorithm for DED-2 (2.2). We can transform the given edge weighted DAG $D(V, A, W_e)$ into an undirected node weighted graph $G(V, E, W_v)$. This is done by transforming each directed edge e in D into a vertex v_e in G. Two vertices v_e and v_f in G are connected, if their corresponding edges e and f in D form a directed path. An example of such a transformation can be found in Figure 3.2. Finding a maximum independent set on G has the same results as solving DED-2 on D. This means, forming G and then using NeighbourCover on it leads to a 2-approximation. NeighbourCoverDED2 improves on this idea by only implicitly iterating over the vertices in G and keeping track of the nodes V_{head} and V_{tail}, which are head/tail vertices of kept edges respectively. This allows for a time complexity of $O(|A| \log |A|)$ in the weighted case and $O(|A|)$ in the unweighted case.

Algorithm. Given an edge weighted DAG $D = (V, A, W_e)$. The algorithm keeps track of all deleted edges $\mathcal{D}$ and the head and tail vertices of kept edges V_{head} and V_{tail}. All the sets $\mathcal{D}, V_{head}$ and V_{tail} start empty. Algorithm 3 provides pseudocode.

Algorithm 3 NeighbourCoverDED2

```
D ← ∅, V_head ← ∅, V_tail ← ∅
edgePermutation ← WEIGHTEDSHUFFLE(w_1, w_2, ..., w_|E|)
for e = (u, v) in edgePermutation do
    if u ∈ V_head or v ∈ V_tail then
        D ← D ∪ {e}
    else
        V_head ← V_head ∪ {v}
        V_tail ← V_tail ∪ {u}
return D
```

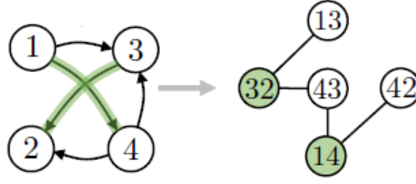


Figure 3.2: Transformation DAG to undirected node weighted graph. Reprinted from [19].

3.1.4 NeighbourCoverColorEC

NeighbourCoverColorEC works on a colored edge weighted hypergraph. It is a probabilistic 2-approximation for colored edge clustering (2.2). We can transform the given colored edge weighted hypergraph $H(V, \mathcal{E}, l, k, W_e)$ into an undirected node weighted graph $G = (V, E, W_v)$. This is done by transforming each edge $e \in \mathcal{E}$ into a vertex in G. Two vertices v_e and v_f in G are connected, if their corresponding edges $e, f \in \mathcal{E}$ overlap at one or more vertices $v \in V$. An example of such a transformation can be found in Figure 3.3. The idea is, that finding a maximal independent set in G, is equivalent to finding a maximal set of satisfied edges in H. This means forming G and using NeighbourCover on it, leads to a 2-approximation. NeighbourCoverColorEC uses an improved version of this approach by only iterating through G implicitly. This yields a time complexity of $O(|\mathcal{E}| * \log|\mathcal{E}| + \sum_{e \in \mathcal{E}} |e|)$ in the weighted case and $O(\sum_{e \in \mathcal{E}} |e|)$ in the unweighted case.

Algorithm. Given a colored edge weighted hypergraph $H(V, \mathcal{E}, l, k, W_e)$. The algorithm keeps track of all the colors of the nodes. All nodes start as uncolored. Algorithm 4 provides pseudocode.

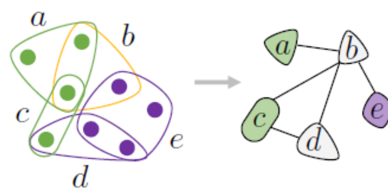


Figure 3.3: Transformation from colored edge weighted hypergraph to undirected node weighted graph. Reprinted from [19].

Algorithm 4 NeighbourCoverColorEC

```
nodeColor ← array of size |V| filled with zeros (uncolored)
edgePermutation ← WEIGHTEDSHUFFLE(w_1, w_2, ..., w_|E|)
for e in edgePermutation do
    edgeColor ← l(e)
    edgeSatisfiable ← true
    for node in e do
        if nodeColor[node] != edgeColor or nodeColor[node] != 0 then
            edgeSatisfiable ← false
            break
    if edgeSatisfiable then
        for node in e do
            nodeColor[node] = edgeColor
If a node still has nodeColor zero, assign it a random color
return nodeColor
```

3.2 Competing Algorithms

This section contains other algorithms which work on the same problems as the Neighbour-Cover algorithms. These are the basic versions of the algorithms. Any adaptations made before the evaluation can be found in Section 5.2.

MatchingVC. Both Gavril and Yannakakis are credited with the development of the MatchingVC algorithm (see [11] and [17]). It iterates through the edges of the graph. If both endpoints of the current edge are not covered yet, MatchingVC puts them in the cover.

PittVC. The algorithm from Pitt [16] iterates through the edges of a node weighted graph. If it encounters an edge where both endpoints are not covered, it decides to cover one of them with a random decision weighted by the node weights.

LocalRatioVC. The LocalRatioVC algorithm from Bar-Yehuda and Even [4, 5] keeps track of the residual weight of each node. In the beginning of the algorithm the residual node weight is equal to the normal weight of the node. Then, the algorithm iterates through the edges of the graph. For each edge the algorithm checks which endpoint has the lower residual weight. It then reduces the residual weight of both nodes by that value. At the end of the algorithm all nodes with residual weight zero are in the vertex cover.

GreedyMIS. The GreedyMIS algorithm [19] generates a random uniform node permutation and all nodes start as undecided. Then, it iterates through the nodes in the order of the permutation. If the current node is undecided the algorithm assigns it to the independent set and all its undecided neighbours to the vertex cover.

GreedyMatching The GreedyMatching algorithm [8] picks the edge with the highest edge weight in the graph. Then, it puts it into the matching and deletes all adjacent edges. It repeats this until all edges are either in the matching or deleted.

SemiStreamingMatching. The SemiStreamingMatching algorithm from Paz and Schwartzman [18] for constant $\alpha > 1$, is a single-pass (2α) approximation for maximum weight matching (MWM). It adapts the local-ratio MWM algorithm to the Semi-Streaming model. For the exact workings of the algorithm we refer to the original paper [18]. By using the Semi-Streaming model, it may use $O(n \text{ poly} \log n)$ space instead of being restricted to $O(n)$ space like normal streaming algorithms.

LocalRatio. The LocalRatio Algorithm [15] for DED2 is a polynomial time 2-approximation. While there are still paths of length 2 remaining it repeats the following: Find a path of length 2 and then reduce the weight of all of its edges by the smallest edge weight in the path. Should an edge be reduced to zero by this, it is deleted.

CHAPTER 4

Buffered Approach To NeighbourCover

When trying to convert the NeighbourCover algorithms [19] to a buffered approach, we did not find a way to keep both the quality guarantee and the complexity the same as the original. Instead, bNeighbourCoverVC keeps the quality guarantee the same, but it restreams the graph, which leads to a worse theoretical time complexity.
For the remaining algorithms bNeighbourCoverMinD2M, bNeighbourCoverDED2 and bNeighbourCoverColorEC we use another strategy, ensuring that the time complexity of the algorithms is better than the offline approach. The drawback is that the quality is no longer a 2-approximation of the solution. These two strategies will be explained in more detail in the following sections.

4.1 Buffered NeighbourCoverVC

Pseudocode for bNeighbourCoverVC can be found in Algorithm 5. This approach is designed to deliver exactly the same solution as NeighbourCoverVC. Since it delivers the same solution, naturally the quality guarantee stays the same. The first step is to get a permutation of all nodes in the graph via a weighted shuffle. We then assign a priority to all nodes depending on their place in the permutation. If a node comes first in the permutation, it has a higher priority than a node that comes later in the permutation.
Normally, NeighbourCoverVC [19] would visit and assign the nodes in the order decided by the weighted shuffle. Since we only have access to a limited part of the graph, strictly following this order is inefficient. Often, we would have to restream the graph an additional time, to find the batch which contains the exact node we are looking for. This is why we use another strategy which lets us make decisions quicker, while still providing the same result.
For each batch we look through all the undecided nodes in order of their priority, from high to low. Then, if a nodes priority is bigger than all the undecided nodes in its neighbourhood, it can be assigned to the independent set I and all its undecided neighbours are assigned to C. This is exactly how offline NeighbourCoverVC would assign the nodes if it visited

an undecided node. We can do this decision out of the normal permutation order, because the higher priority of our current node guarantees, that it also would be visited before its neighbours in the permutation order. At most, it is possible that a neighbouring node would already be in C, when this decision is normally made. This makes no difference since the neighbours are assigned to C either way. By going through the nodes in the batches in descending order, we allow for the chance of an earlier decision in the batch to make another decision in the same batch possible. If we passed through the entire graph and there are still undecided nodes left, the graph is restreamed, and we go through all batches again.

In the worst case the algorithm needs to be restreamed n times which leads to a complexity of $O(n \log(n) + n^2)$. For this worst case to happen though, the nodes have to be arranged in a very particular way in accordance to the permutation created by the weighted shuffle. Also, in that case the performance could be improved by running the algorithm with another seed. In practice, the performance was never close to the worst case for our benchmarks. At most a graph was restreamed four times, which includes one stream to get all the node weights.

Algorithm 5 bNeighbourCoverVC

```
C ← ∅, I ← ∅, U ← V
go through the all batches once and save the node weights w_1, w_2, ..., w_|V|
restartStream()
permutation ← WEIGHTEDSHUFFLE(w_1, w_2, ..., w_|V|)
nodePriority[v] ← numberOfNodes - permutation.index_of(v)
while U ≠ ∅ do
    //Stream the graph. Restart stream if necessary.
    if currentBatch = lastBatch then
        restartStream()
    currentBatch ← loadNextBatch()
    //Go through the undecided nodes in the batch from hightest to lowest priority.
    currentBatch.sortByPriority()
    for currentNode in currentBatch do
        localMax ← true
        for x ∈ N(currentNode) ∩ U do
            if nodePriority[currentNode] < nodePriority[x] then
                localMax ← false
        if localMax then
            remove the currentNode from U and add it to I
            remove all undecided neighbours of the currentNode from U
            assign all neighbours of the currentNode to C
return C
```

4.2 Buffered NeighbourCover variants for MinD2M, DED2 and ColorEC

Pseudocode for these variants can be found in Algorithm 6. The NeighbourCover algorithms for MinD2M, DED2 and ColorEC all calculate a permutation of the edges of the input graph. Since we are restricted to only use $O(n)$ space, this is not feasible for our buffered approach. Instead, we opt to only permutate the order of the edges in each batch via a weighted shuffle. Then, we use the permutation of the batch to execute the standard logic of the respective NeighbourCover adaptation.
For each subgraph induced by a batch, this method produces an expected 2-approximation. For the overall graph this approximation does not hold. Only the edges in the batches are shuffled. This means unlike in the original NeighbourCover algorithms not every edge permutation is possible. Therefore, it is possible that for one or more decisions, there are only permutations that lead to a suboptimal choice. Thus, the algorithm is lower bounded by the permutation allowing the worst choice to be taken at every decision.
On the other hand, the time complexity is reduced with a smaller buffer size. For NeighbourCoverMinD2M and NeighbourCoverDED2, the complexity is $O(|E| \log |E|)$ and $O(|E| \log |E| + \sum_{e \in E} |e|)$ for NeighbourCoverColorEC. The term of $|E| \log |E|$ stems from the weighted shuffle over all edges. The rest of the algorithm only takes $O(|E|)$ time or rather $O(|E| + \sum_{e \in E} |e|)$ for NeighbourCoverColorEC. Since the buffered NeighbourCover algorithms only shuffle each buffer, the overall time needed for shuffling is $O(|E| \log \frac{|E|}{b})$, where b is the amount of batches. That means the new time complexity for bNeighbourCoverMinD2M and bNeighbourCoverDED2 is $O(|E| \log \frac{|E|}{b})$ and $O(|E| \log(\frac{|E|}{b}) + \sum_{e \in E} |e|)$ for bNeighbourCoverColorEC.

Algorithm 6 General Algorithm Structure
(bNeighbourCoverMinD2M, bNeighbourCoverDED2, bNeighbourCoverColorEC)

while *!lastBatchReached()* **do**
  currentBatch ← loadNextBatch()
  $\sigma \leftarrow$ WEIGHTEDSHUFFLE($w_e | e \in currentBatch$)

  for *currentEdge*← $e_{\sigma(1)}, e_{\sigma(2)}, ..., e_{\sigma(bufferSize)}$ **do**
    //Apply the logic for the specific problem here
    solution ← problemSpecificLogic()
return solution

CHAPTER 5

Competing Algorithms

This chapter is split into two sections. Section 5.1 contains a simple greedy algorithm, which to the best of our knowledge was not formalized by any other paper. Section 5.2 lists the modifications to the already existing algorithms specified in Section 3.2, to convert them to a streaming or buffered streaming approach.

5.1 Original Algorithms

This section contains a simple greedy algorithm, which to the best of our knowledge was not formalized by any other paper. The prefix “s” signifies that it is a streaming algorithm.

sGreedyColorEC. The sGreedyColorEC algorithm streams through all hyperedges of the graph. If it encounters a hyperedge which contains no node of a color other than its own, it is deemed satisfiable. In this case the algorithm satisfies it by assigning each contained node the color of the hyperedge. To the best of our knowledge no other paper has explicitly formalized this algorithm.

5.2 Modifications on Competing Algorithms

The algorithms in this section are a modification of previous work. The authors and a short description of the original algorithms can be found in Section 3.2. We add prefixes to their original names, in order to easily distinguish buffered streaming and streaming approaches. Buffered streaming algorithms have “b” as prefix and streaming algorithms have “s” as prefix.

sMatchingVC. MatchingVC can easily be used as a streaming approach since it only makes decisions for one edge at a time. When streaming the edges of the graph, no modification is necessary. The algorithm runs in $O(E)$ time and is a 2-approximation for unweighted VC.

sPittVC. We need to know the weight of the nodes to make the random weighted decisions. For a streaming approach, this makes it necessary to stream through the graph once in the beginning. Otherwise, it would be possible that the required node weights are not known yet. In the re-stream all weights are known. During the re-stream no additional modification is necessary. The algorithm runs in $O(E)$ time and is a randomized 2-approximation for weighted VC.

sLocalRatioVC. When converting the algorithm to a buffered approach, it is necessary to stream over the graph once in the beginning, to set all residual node weights. In the restream when the residual node weights are known, no additional modification is necessary. The algorithm runs in $O(E)$ time and is a 2-approximation for weighted VC.

bGreedyMIS. When trying to convert GreedyMIS to a buffered approach iterating through the vertices in order of the permutation poses a problem. Because of that we do not shuffle the order of all nodes and instead just shuffle each batch. The rest of the algorithm can be applied to each batch normally. The algorithm runs in $O(E)$ time.

bGreedyMatching. The bGreedyMatching algorithm remembers for every node if an adjacent edge has been matched. It iterates through each batch from highest to lowest edge weight. If it encounters an edge where neither endpoint has an adjacent matched edge, it adds it to the matching. The endpoints are then marked as next to a matched edge. The algorithm runs in $O(E)$ time and is a 2-approximation for maximum unweighted matching.

SemiStreamingMatching. The SemiStreamingMatching was not modified.

bLocalRatioDED2. When trying to convert LocalRatioDED2 into a buffered approach we encounter the following problem: We are restricted to $O(n)$ memory, but we would need $O(m)$ memory to keep track of the residual edge weights of all edges. This is why for each node, we choose to keep track of the summed up weights of outgoing edges instead. We then iterate through the edges as follows: For each edge reduce the outgoing edge weights of the endpoints by MIN(outgoingEdgeWeights(startNode), outgoingEdgeWeights(targetNode), edgeWeight). If the edge weight was the minimum, delete the edge immediately. After iterating through all edges, delete all edges that have a start node with outgoing edge weight zero.

Theorem 1
The bLocalRatioDED2 algorithm returns a valid solution for the DED2 problem.

Proof: A solution is correct if no edge in the remaining graph can be extended to a path of length 2. In the following we will show that the algorithm guarantees this for each edge. When looking at an edge there are three possibilities:

1. The outgoing edge weight of the start node is the lowest. That means it will be reduced to zero. Since that means we delete all its outgoing edges, the current edge will be deleted and therefore cannot be extended to a path of length 2.
2. The outgoing edge weight of the target node is the lowest. That means it will be reduced to zero. Since that means we delete all of its outgoing edges, the target node no longer has outgoing edges. Therefore, the current edge cannot be extended to a path of length 2.
3. The edge weight of the current edge is lowest. That means we delete the current edge. Therefore, it can not be extended to a path of length 2.

At the end of the algorithm it has visited each edge once, which means there is no way to extend any edge to a path of length 2. □

Theorem 2
The bLocalRatioDED2 algorithm has a time complexity of $O(|E|)$.

Proof: The algorithm needs to stream the graph three times. The first stream is done to set the weights of outgoing edges for each node, which can be done in $O(|E|)$. The second stream reduces these weights and deletes edges if necessary which also takes $O(|E|)$ time. The third stream deletes all edges which start at a node with outgoing edge weight zero, which again takes $O(|E|)$ time. This means overall the algorithm has a time complexity of $O(|E|)$. □

Theorem 3
The bLocalRatioDED2 algorithm is a 2-approximation algorithm for DED2.

Proof: This proof is done analogous to the quality proof of LocalRatioDED2 in [15]. Let $Opt \subseteq E$ be an optimal solution and $S \subseteq E$ the solution of bLocalRatioDED2. An edge is only deleted and therefore in S in the following cases:

1. Its edge weight is smaller than the outgoing edge weights of its start and target node. In this case the outgoing edge weights of both endpoints get reduced by the weight of the node.

2. The outgoing edge weights of the start point have been reduced to zero.

This means the weight of the deleted edges is upper bounded by the total reduction of the outgoing edge weight of nodes. Each iteration of the buffered approach is equal to looking at paths of length 2 starting with the current edge $e = (u, v)$ and reducing the outgoing edge weight of u and v by the minimum weight edge, until there is no path left starting with e. For each of these implicit paths the reduction is at most equal to 2 times the minimum weight edge. Since in each path there is at least one edge d that is in OPT we can charge the weight reduction to the weight of d. Then, the weight of d decreases by at least the factor $1/2$ of what is charged, and cannot decrease beyond 0. This is why the weight of S is within a factor 2 of the weight of OPT. □

CHAPTER 6

Experimental Evaluation

6.1 Setup

The machine used for all tests, has an Intel-Core I7-7500U CPU @2.70GHz and 8GB of main memory. The operating system is Ubuntu 22.04.3 LTS and the kernel version is 5.15.0-83-generic. For the evaluation regular graphs were chosen from the 10th DIMACS implementation challenge [3]. Hypergraphs were taken from the DAC2012 benchmark suite [20] and the ISPD98 Circiut benchmark suite [2, 13]. More information on the graphs can be found in Table 6.1 and Table 6.2.
These graphs are all undirected and unweighted. Since the minimum vertex cover problem requires node weighted graphs and all other problems require edge weighted graphs these values are generated before applying the algorithms. This was done by assigning each node a weight equal to (ID modulo 100) + 1. Edge weights are assigned uniformly at random between 1 and 100, to avoid the weight of edges incident to a node to be in a continuous interval. For ColorEC each edge was assigned one of ten colors, by setting the edge color equal to (ID modulo 10) + 1. Lastly DED2 works on DAGs. In this case the graphs were converted by generating a random uniform permutation of the node ID's and assigning them to the nodes as direction ID's. Then, we remove all edges whose start node has a higher direction ID than their target node. All algorithms were compared in terms of running time and solution quality. For the purpose of our experiments the time to read in the graphs or to generate additional graph data, like weight, color or direction is not counted towards the runtime.
In the experiments we will use the following methodology: The algorithms which are dependent on another criterion than just the input data are tweaked on a subset of all the test graphs first. After we have compared the algorithms against themselves with different criterions and have found the versions we want to use, we move on to the next step. Here, we use the full set of test graphs. We then compare all algorithms which solve the same problem. Additionally, we run all non-deterministic algorithms ten times with different

seeds. The resulting score will be the arithmetic mean of all results on the same test graph. The quality of the algorithms is determined as follows: The VC algorithms are compared in terms of the weight of the cover, meaning lower is better. The matching algorithms are compared on the weight of the matching, meaning higher is better. The DED2 algorithms are compared in terms of the weight of deleted edges, meaning lower is better. Lastly, the ColorEC algorithms are compared in terms of weight of satisfied edges, meaning higher is better.
We present our results as performance profiles [9]. A performance profile compares different algorithms on a set of instances. For every instance it determines the best solution provided by one of the algorithms. Then, for factor f it plots the fraction of instances on which a given algorithm is only factor f away from the best compared solution. This means an algorithm is better the higher its fraction within factor f of the best algorithm.

Type	Name	n	m	sources
Citation Network	CoAuthorsCiteseer	227 320	814 134	[20, 12]
	CoPapersCiteseer	434 102	16 036 720	[20, 12]
	citationCiteseer	268 495	1 156 647	[20, 12]
	CoAuthorsDBLP	299 067	977 676	[20, 12]
	CoPapersDBLP	540 486	15 245 729	[20, 12]
Clustering Instance	cnr2000	325 557	2 738 969	[20, 7, 6]
	eu2005	862 664	16 138 468	[20, 7, 6]
	in2004	1 382 908	13 591 473	[20, 7, 6]
Delanauy Graphs	delanauy_n19	524 288	1 572 823	[20, 14]
	delanauy_n20	1 048 576	3 145 686	[20, 14]
	delanauy_n18	262 144	786 396	[20, 14]
	delanauy_n21	2 097 152	6 291 408	[20, 14]
Random Geometric	rgg_n_2_19_s0	524 288	3 269 766	[20, 14]
	rgg_n_2_20_s0	1 048 576	6 891 620	[20, 14]
	rgg_n_2_18_s0	262 144	1 547 283	[20, 14]
	rgg_n_2_21_s0	2 097 152	14 487 995	[20, 14]
Street Network	belgium.osm	1 441 295	1 549 970	[20, 1]
	netherlands.osm	2 216 688	2 441 238	[20, 1]
	great-britain.osm	7 733 822	8 156 517	[20, 1]
	italy.osm	6 686 493	7 013 978	[20, 1]

Table 6.1: Information for graphs from 10th DIMACS implementation challenge. Graphs with **bold** names are only used in the extended benchmark set.

Type	Name	n	m	sources
ASIC Design	superblue3	3 110 509	898 001	[20]
	superblue7	4 935 083	1 340 418	[20]
	superblue9	2 898 853	833 808	[20]
	superlbue14	2 049 691	619 815	[20]
	supberblue16	2 280 931	697 458	[20]
	superblue19	1 714 351	511 685	[20]
	superblue6	3 401 199	1 006 629	[20]
	superblue11	3 071 940	935 731	[20]
	superblue12	4 774 069	1 293 436	[20]
	ISPD98_ibm14	147 605	152 772	[2, 13]
	ISPD98_ibm15	161 570	186 608	[2, 13]
	ISPD98_ibm16	183 484	190 048	[2, 13]
	ISPD98_ibm17	185 495	189 581	[2, 13]
	ISPD98_ibm18	210 613	201 920	[2, 13]

Table 6.2: Information for hypergraphs from DAC2012 benchmark suite. Graphs with **bold** names are only used in the extended benchmark set.

6.2 Determining Parameters for VC Algorithms

bNeighbourCoverVC. For bNeighbourCoverVC we only consider running time (see 6.1), because the quality stays exactly the same regardless of buffer size. The offline version of the algorithm is the fastest. This is expected, since there is some overhead introduced by the need to sort the nodes of each batch by priority. We can also see that with lower buffer size the algorithm generally performs faster. This can be explained by the fact that it is easier to sort multiple small batches instead of fewer large batches. For the smallest buffer size of $10\,000$ one graph takes longer than in all other compared versions. Likely this is due to an additional restream that is necessary on that graph with the smaller buffer size. Due to this data we will be using the bNeighbourCoverVC of buffer size $50\,000$ to compare with the other algorithms. It is faster than the larger buffer sizes while avoiding additional restreams which could happen with lower buffer sizes.

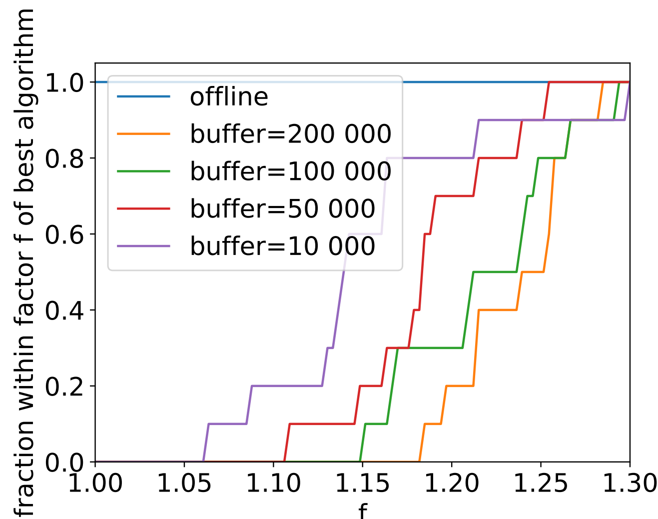
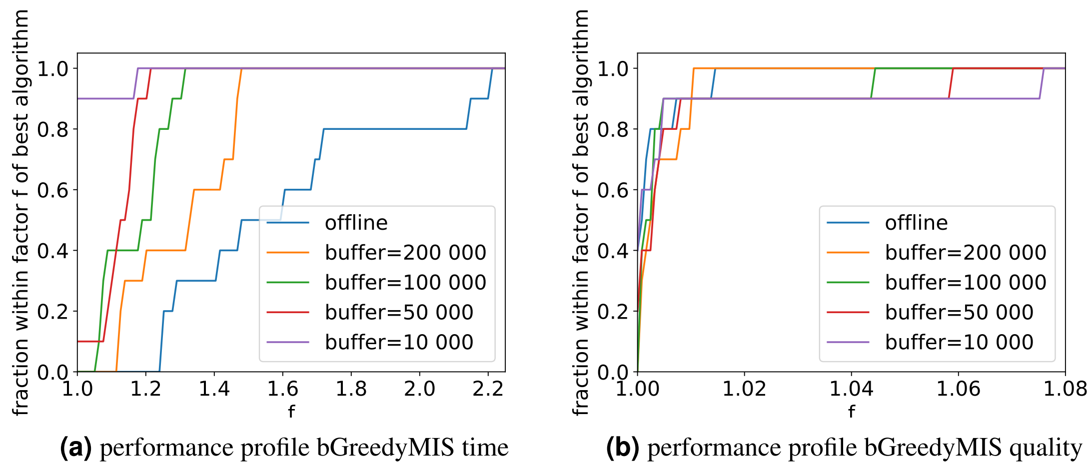


Figure 6.1: performance profile bNeighbourCoverVC time

bGreedyMIS. When comparing the running time for different buffer sizes of bGreedyMIS (see 6.2a), one can observe that a smaller buffer correlates with a faster running time. The biggest buffer size of $100\,000$ is at least 10% slower than buffer size $10\,000$ on all instances. Unlike for bNeighbourCoverVC we shuffle each batch separately, which means there is no additional overhead for traversing each batch. On the contrary, only having to shuffle smaller batches is less complex and reduces running time. In most cases the quality of the algorithm does not change significantly with varying buffer size. 90% of all instances do not differ more than 1% in quality, regardless of buffer size. But for the remaining graphs, an additional loss in quality between roughly 1% to 4% occurs each time the buffer size is lowered. For this reason we have chosen the buffer size of $100\,000$ to compare against the other algorithms. It provides a reasonable speedup for running time, while still being within 5% of the best quality of compared algorithms.



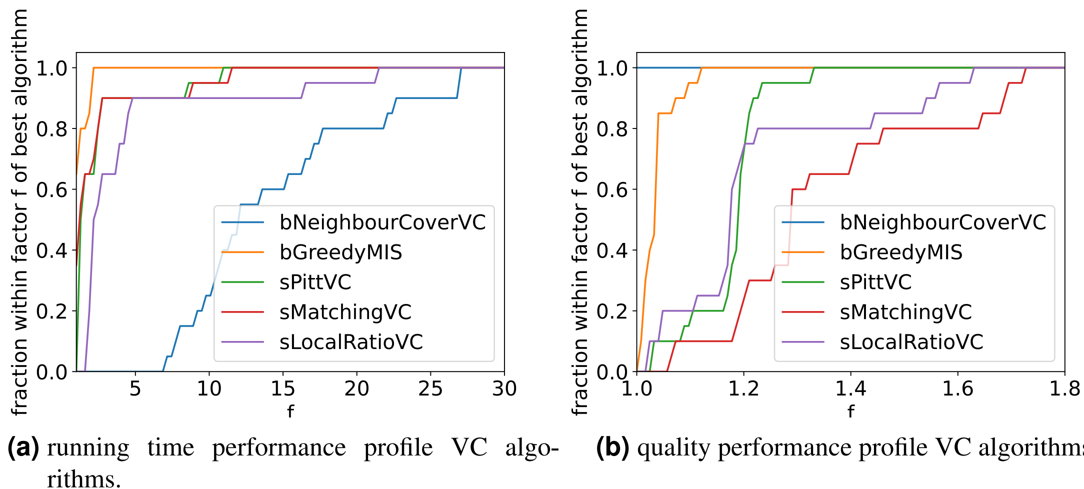
(a) performance profile bGreedyMIS time

(b) performance profile bGreedyMIS quality

Figure 6.2

6.3 VC Algorithms Comparison

Figure 6.3a compares the running time of all algorithms we implemented for minimum weighted vertex cover. The figure shows that bNeighbourCover is the slowest amongst all compared algorithms. In the best case it takes roughly 7 times longer than the fastest algorithm and in the worst case it can even take about 28 times longer. The bGreedyMIS algorithm achieves the lowest running time in 65% of all instances and sMatchingVC is the fastest in all other cases. It is noteable that bGreedyMIS seems to be the most consistent, being at most a factor of 2 away from the best time in all instances.
Figure 6.3b compares the solution quality for the different algorithms. In this category bNeighbourCoverVC performs best in all instances. Second best is bGreedyMIS, at worst being 11% worse than bNeighbourCover. Despite its good performance in terms of time, sMatchingVC has the worst results in quality. In some cases it can even have a solution that is more than a factor of 1.7 away from bNeighbourCoverVC.
Overall, bGreedyMIS seems to perform best with a good running time and a comparably good quality. On the other hand, bNeighbourCover consistently provides a better quality. If the solution quality is critical, it might be a better choice in spite of its longer running time.



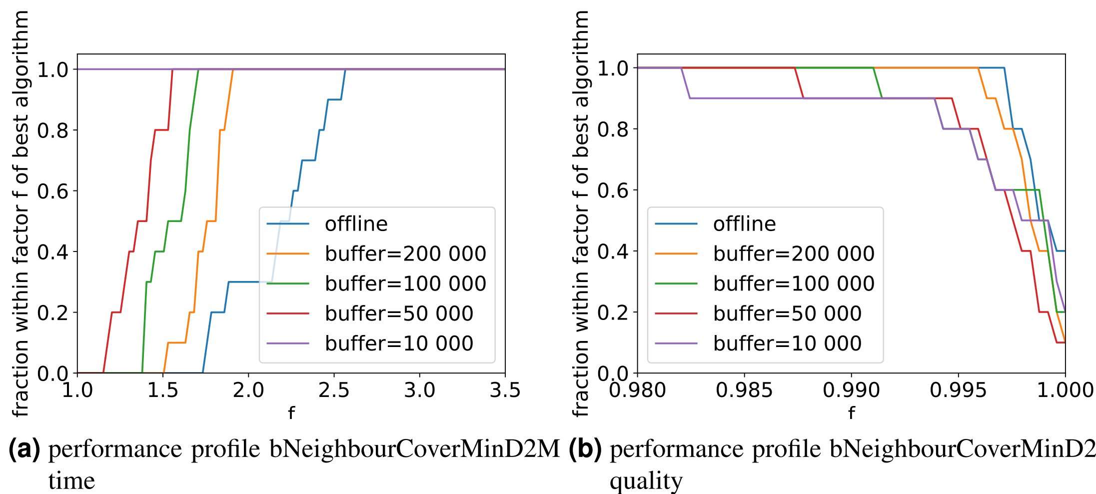
(a) running time performance profile VC algorithms.

(b) quality performance profile VC algorithms

Figure 6.3

6.4 Determining Parameters for Matching Algorithms

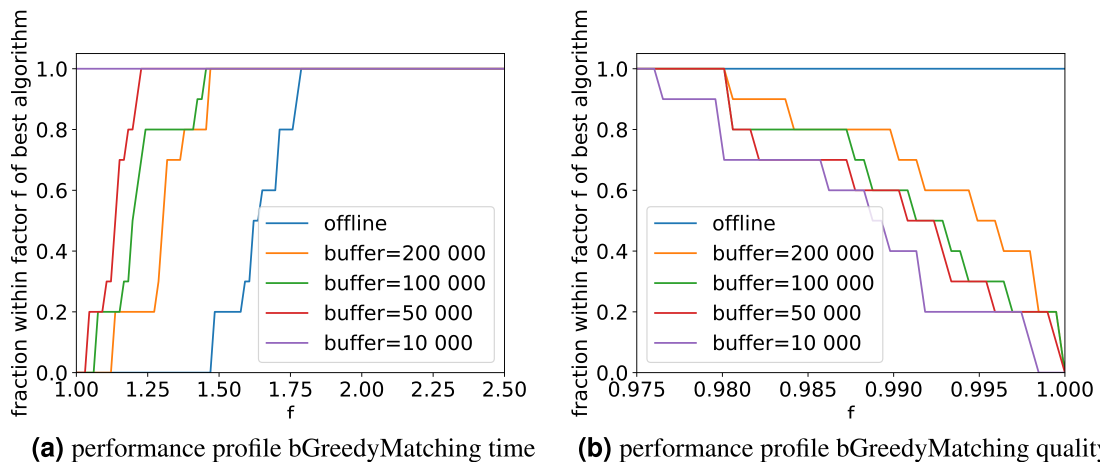
bNeighbourCoverMinD2M. Figure 6.4a compares the running time of NeighbourCoverMinD2M and bNeighbourCoverMinD2M for varying buffer sizes. Generally, for a lower buffer size the algorithm executes faster. For the buffer size of $10\,000$ NeighbourCoverMinD2M is slower than bNeighbourCoverMinD2M by at least a factor of 1.7. In one case the factor even grows to 2.5.
Figure 6.4b compares the solution quality of NeighbourCoverMinD2M and bNeighbourCoverMinD2M for varying buffer sizes. Here, the results are more similar. The biggest gap of quality is less than 2%. We chose the buffer size of $10\,000$ to compare against the other algorithms, because it provides a significant speedup for a small loss in quality.



(a) performance profile bNeighbourCoverMinD2M time **(b)** performance profile bNeighbourCoverMinD2M quality

Figure 6.4

bGreedyMatching. In Figure 6.5a you can see that bGreedyMatching executes faster for lower buffer sizes. The bGreedyMatching algorithm sorts the edges of each batch by weight. As already mentioned, it is easier to sort multiple smaller batches, which leads to the faster running times for smaller buffers.
The difference in solution quality (see 6.5b) is generally very low for different buffer sizes. For all buffer sizes the quality never differs more than 2.5%. Because the quality seems stable even for small sizes, we have chosen the buffer size of $10\,000$ to compare against other algorithms. On all instances it has the fastest running time, while still providing a comparatively good quality.

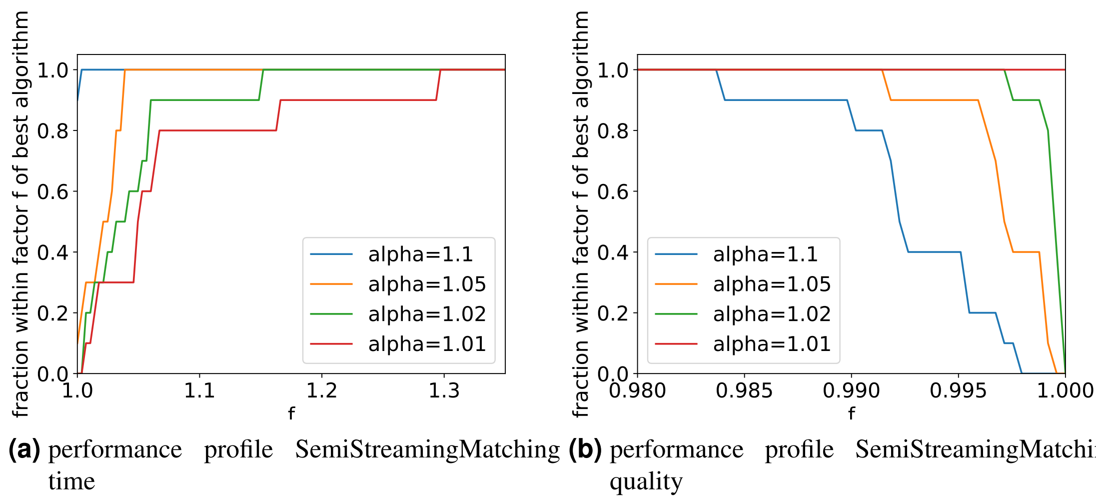


(a) performance profile bGreedyMatching time

(b) performance profile bGreedyMatching quality

Figure 6.5

SemiStreamingMatching. Figure 6.6a compares the running time of SemiStreaming-Matching for different constants $\alpha > 1$. For $\alpha = 1.1$ the algorithm executes the fastest followed by $\alpha = 1.05$ which is only about 3% slower in all cases. When setting α even lower we experience a more significant running time increase for some instances. In one particular instance the algorithm is approximately 27% slower for $\alpha = 1.01$.
Figure 6.6b compares the solution quality for the different values for α. The best quality is achieved by $\alpha = 1.01$ and the lowest quality by $\alpha = 1.1$. Even for these the maximum difference in quality is under 1.75%. Because the quality is so similar for all α we choose $\alpha = 1.05$. This ensures a good quality while avoiding the more volatile running times that coincide with lower values of α.



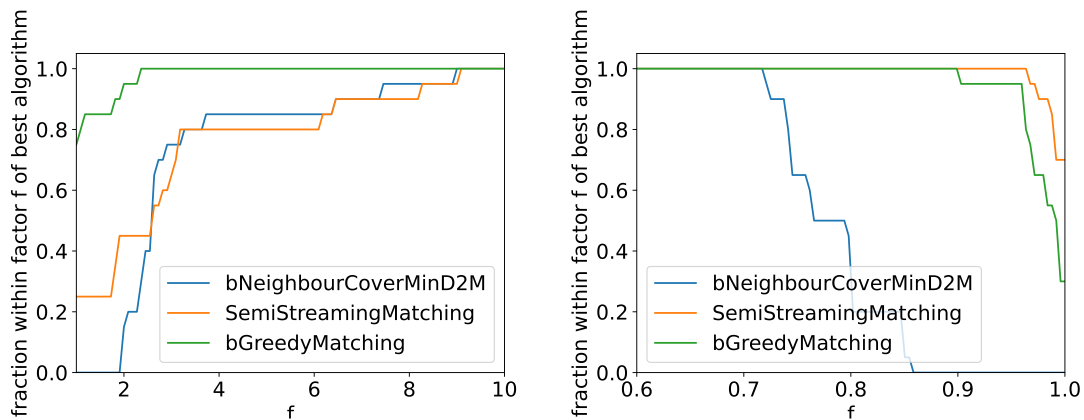
(a) performance profile SemiStreamingMatching time

(b) performance profile SemiStreamingMatching quality

Figure 6.6

6.5 Matching Algorithms Comparison

Figure 6.7a compares the running time of all algorithms we implemented for matching. In 75% of all instances bGreedyMatching performs best and SemiStreamingMatching performs best in the remaining cases. The running time of bNeighbourCoverMinD2M is never the fastest, but in 55% of all instances it performs very similar to SemiStreaming matching, being equally good or even slightly better.
Figure 6.7b compares the solution quality for the different algorithms. In 70% of all instances SemiStreamingMatching provides the best quality and bGreedyMatching provides the best quality in the remaining cases. The results of bNeighbourCoverMinD2M are always more than 15% away from the best compared algorithm.
Overall, bGreedyMatching seems like the best algorithm with both good running times and quality. Although, bNeighbourCoverMinD2M is often close in running time to SemiStreaming, it loses in solution quality.



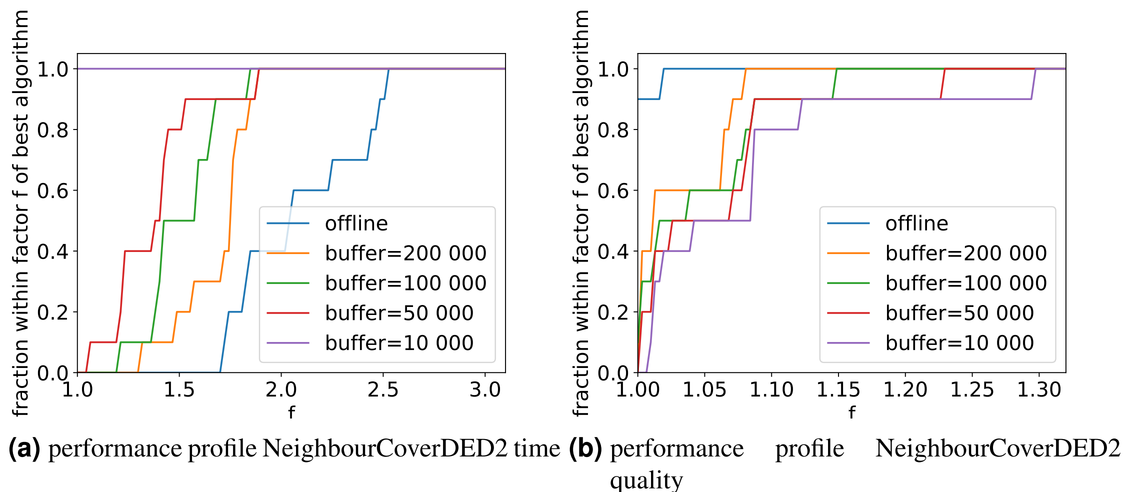
(a) running time performance profile matching algorithms

(b) quality performance profile matching algorithms

Figure 6.7

6.6 Determining Parameters for DED2 Algorithms

bNeighbourCoverDED2. Figure 6.8a compares the running time of NeighbourCoverDED2 and bNeighbourCoverDED2 for varying buffer sizes. The algorithm executes faster for lower buffer sizes. When comparing bNeighbourCoverDED2 with buffer size $10\,000$ with NeighbourCoverDED2 (offline), the running time of NeighbourCoverDED2 is higher by a factor of 1.6 on all instances.
When comparing the quality of the solutions in Figure 6.8b we can see that the versions with a higher buffer size also produce higher quality solutions. For up to 80% of the tested graphs, the solution quality of all versions is within a factor of 1.1. The remaining graphs have a more volatile reaction to a decrease in buffer size. In one instance, the algorithm with buffer size equal to $10\,000$ has a 30% lower solution quality than the best algorithm. In order to achieve a stable quality across all graphs we choose the biggest buffer size of $200\,000$.

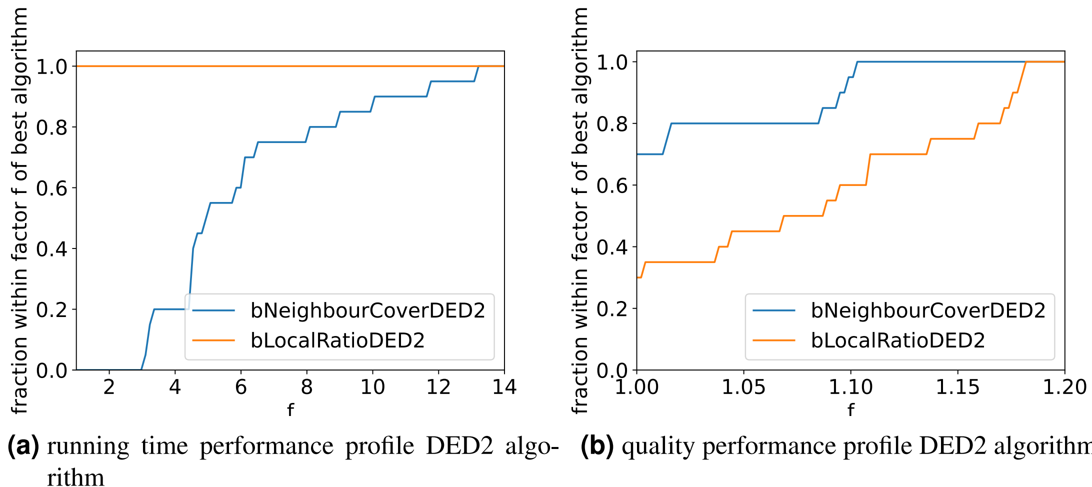


(a) performance profile NeighbourCoverDED2 time **(b)** performance profile NeighbourCoverDED2 quality

Figure 6.8

6.7 DED2 Algorithms Comparison

Figure 6.9a compares the running time of both algorithms we implemented for DED2. Here, bNeighbourCoverDED2 is slower than bLocalRatioDED2. The difference in runtime ranges from factor 2 up to factor 13.
Figure 6.9b compares the solution quality for the algorithms. Here, bNeighbourCoverDED2 has the best solution in 70% of all instances. At worst bNeighbourCoverDED2 is about 10% away from the best compared solution. Its competitor bLocalRatioDED2 can be up to roughly 17.5% worse.
Overall, there is a trade-off between speed and quality between the two algorithms. While bLocalRatioDED2 wins in speed, bNeighbourCoverDED2 provides the best solution more often. Even in the cases in which it does not provide the best solution they are less than 10% apart.

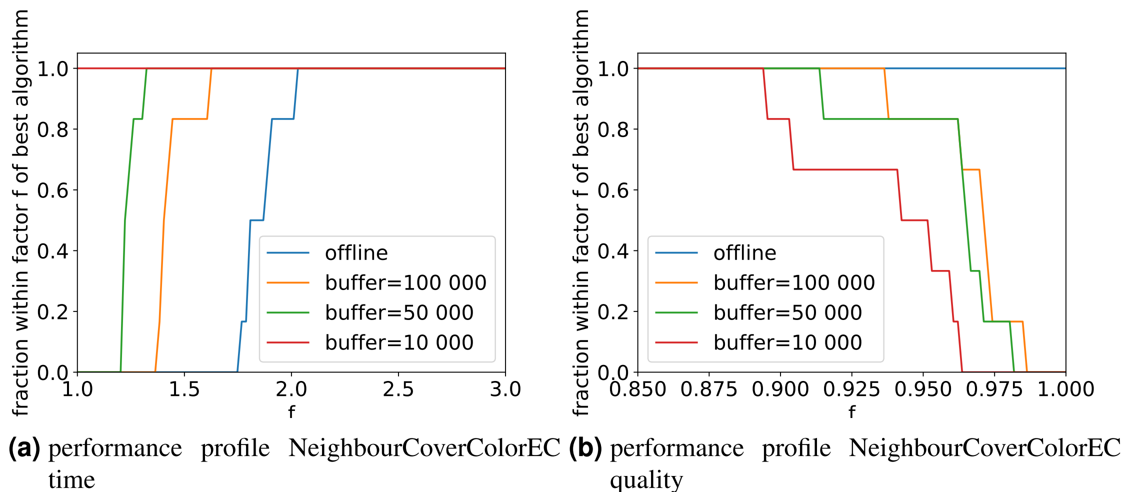


(a) running time performance profile DED2 algorithm **(b)** quality performance profile DED2 algorithms

Figure 6.9

6.8 Determining Parameters for ColorEC Algorithms

NeighbourCoverColorEC. We did not test for buffer size $200\,000$ because some graphs in the extended benchmark have less than $200\,000$ edges. Figure 6.10a compares the running time of NeighbourCoverColorEC and bNeighbourCoverColorEC for varying buffer sizes. Generally, the algorithm executes faster for lower buffer sizes. Therefore, the fastest version has the lowest buffer size of $10\,000$. NeighbourCoverColorEC (offline) is slower by at least the factor of 1.7 on all instances.
When comparing the quality of the solutions in Figure 6.10b, we observe that algorithms with a higher buffer size also produce solutions of a better quality. The lowest quality is achieved by the buffer size of $10\,000$. For 60% of the graphs the solution quality is only 6% lower than the solution of the best algorithm. But for the remaining graphs it is worse by 10% to 12%. We choose the version with buffer size $100\,000$, because it has decent results in both running time and solution quality.

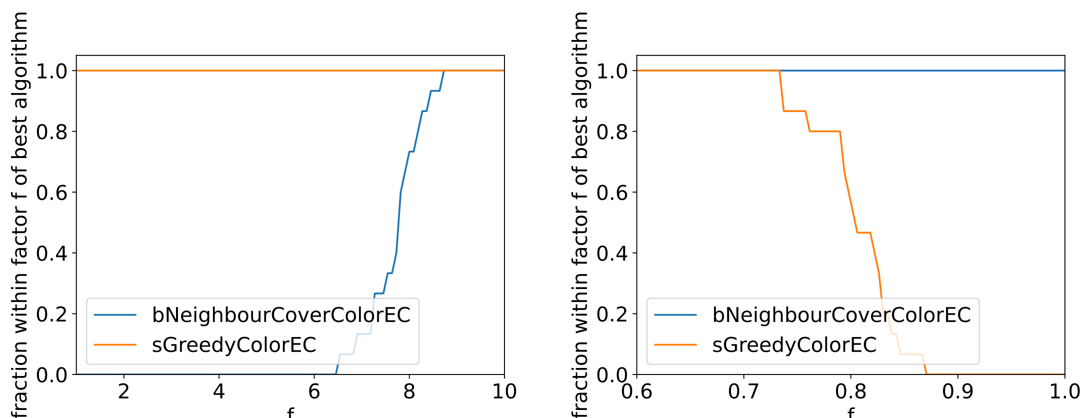


(a) performance profile NeighbourCoverColorEC time **(b)** performance profile NeighbourCoverColorEC quality

Figure 6.10

6.9 ColorEC Algorithms Comparison

Figure 6.11a compares the running time of both algorithms we implemented for ColorEC. In this category bNeighbourCover is worse, being slower by a factor between roughly 6.5 and 8.5.
Figure 6.11b compares the solution quality for the algorithms. In this category bNeighbourCoverColorEC provides the best solution on all instances. The solution of sGreedyColorEC is worse by at least 12% and at most 27%.
Overall, it is very clear which algorithm you should pick, depending on the metric that is most important to you. If running time is more important sGreedyColorEC is superior but for solution quality bNeighbourCoverColorEC is better.



(a) running time performance profile ColorEC algorithms **(b)** quality performance profile ColorEC algorithms

Figure 6.11

CHAPTER 7

Discussion

7.1 Conclusion

In this thesis we engineer buffered streaming approaches for the NeighbourCover algorithms introduced in [19] by Veldt. Furthermore, we modify other algorithms solving the same problems, so that they can be used in a streaming or buffered streaming approach too. We then move on and test on our graph set which parameters work best for these algorithms. Finally, we test all algorithms on the extended graph set and compare their runtime and solution quality for their respective problems.
For minimum weighted vertex cover and colored edge clustering the buffered NeighbourCover algorithms produce the best quality solution on all instances. For DAG edge deletion with parameter $k = 2$ bNeighbourCoverDED2 delivers the best solution quality on 70% of all instances. Conversely, in terms of running time the bufferedNeighbourCover algorithms are the slowest. Overall, they offer a trade-off between solution quality and running time. Only for the matching problem bNeighbourCoverMinD2M does not perform well in terms of quality and is also slightly worse in terms of running time.

7.2 Future Work

Because bNeighbourCoverMinD2M performs poorly for the matching problem, it could be of interest to look into ways to further improve the algorithm. One idea in particular could be to use the semi streaming model. With a memory constraint less strict than the streaming approach, an increase in solution quality could be feasible.
Another point of interest may be the weighted shuffle as it is responsible for the majority of the running time. A more efficient implementation could benefit the running times.

Zusammenfassung

In [19] führt Veldt den NeighbourCover Algorithmus ein und zeigt, dass dieser sowohl im gewichteten, als auch im ungewichteten Fall, eine probabilistische 2-approximation für das NP harte vertex cover Problem ist. Zusätzlich weiten sie die Idee aus, um probabilistische 2-approximationen für minimum delete to matching, DAG edge deletion und edge colored hypergraph clustering, zu erstellen. In dieser Arbeit stellen wir buffered streaming Versionen von NeighbourCover für alle genannten Problemstellungen vor. Desweiteren erstellen wir weitere buffered Algorithmen, indem wir existierende offline Algorithmen modifizieren. Wir zeigen, dass die Qualität der buffered NeighbourCover Algorithmen, im Vergleich zu den anderen implementierten Algorithmen im Durchschnitt 15% besser ist für minimum weighted vertex cover, 5% besser für DAG edge deletion und 24% besser für colored edge clustering.

Bibliography

[1] Geofabrik gmbh, https://download.geofabrik.de/.

[2] Charles J. Alpert. The ispd98 circuit benchmark suite. In *Proceedings of the 1998 International Symposium on Physical Design*, ISPD '98, page 80–85, New York, NY, USA, 1998. Association for Computing Machinery.

[3] D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, editors. *Graph Partitioning and Graph Clustering - 10th DIMACS Implementation Challenge Workshop, Atlanta, Georgia/USA, February 13-14, 2012*, volume 588 of *Contemporary Mathematics*. American Mathematical Society, 2013.

[4] Reuven Bar-Yehuda and Shimon Even. A linear-time approximation algorithm for the weighted vertex cover problem. *J. Algorithms*, 2(2):198–203, 1981.

[5] Reuven Bar-Yehuda and Shimon Even. A local-ratio theorem for approximating the weighted vertex cover problem. In Manfred Nagl and Jürgen Perl, editors, *Proceedings of the WG '83, International Workshop on Graphtheoretic Concepts in Computer Science, June 16-18, 1983, Haus Ohrbeck, near Osnabrück, Germany*, pages 17–28. Universitätsverlag Rudolf Trauner, Linz, 1983.

[6] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th international conference on World Wide Web*. ACM Press, 2011.

[7] Paolo Boldi and Sebastiano Vigna. The WebGraph framework I: Compression techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 595–601, Manhattan, USA, 2004. ACM Press.

[8] Argyrios Deligkas, George B. Mertzios, and Paul G. Spirakis. On the complexity of weighted greedy matchings. *CoRR*, abs/1602.05909, 2016.

[9] Elizabeth D. Dolan and Jorge J. Moré. Benchmarking optimization software with performance profiles. *Math. Program.*, 91(2):201–213, 2002.

[10] Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. On graph problems in a semi-streaming model. *Theor. Comput. Sci.*, 348(2-3):207–216, 2005.

[11] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[12] Robert Geisberger, Peter Sanders, and Dominik Schultes. Better approximation of betweenness centrality. In J. Ian Munro and Dorothea Wagner, editors, *Proceedings of the Tenth Workshop on Algorithm Engineering and Experiments, ALENEX 2008, San Francisco, California, USA, January 19, 2008*, pages 90–100. SIAM, 2008.

[13] Lars Gottesbüren, Tobias Heuer, Nikolai Maas, Peter Sanders, and Sebastian Schlag. Scalable high-quality hypergraph partitioning, 2023.

[14] Manuel Holtgrewe, Peter Sanders, and Christian Schulz. Engineering a scalable high quality graph partitioner. In *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA, 19-23 April 2010 - Conference Proceedings*, pages 1–12. IEEE, 2010.

[15] Sreyash Kenkre, Vinayaka Pandit, Manish Purohit, and Rishi Saket. On the approximability of digraph ordering. *Algorithmica*, 78(4):1182–1205, 2017.

[16] Yale University. Department of Computer Science and L.B. Pitt. *A Simple Probabilistic Approximation Algorithm for Vertex Cover*. Technical report. Yale University, Department of Computer Science, 1985.

[17] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, 1982.

[18] Ami Paz and Gregory Schwartzman. A (2+ϵ)-approximation for maximum weight matching in the semi-streaming model. *ACM Trans. Algorithms*, 15(2):18:1–18:15, 2019.

[19] Nate Veldt. Growing a random maximal independent set produces a 2-approximate vertex cover. *CoRR*, abs/2209.04673, 2022.

[20] Natarajan Viswanathan, Charles Alpert, Cliff Sze, Zhuo Li, and Yaoguang Wei. The dac 2012 routability-driven placement contest and benchmark suite. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, page 774–782, New York, NY, USA, 2012. Association for Computing Machinery.

[21] C. K. Wong and M. C. Easton. An efficient method for weighted sampling without replacement. *SIAM Journal on Computing*, 9(1):111–113, 1980.