

Engineering Semi-Streaming Maximum Independent Set Algorithms

Benedikt Vidic

October 22, 2025

4738257

Master Thesis

at

Algorithm Engineering Group Heidelberg
Heidelberg University

Supervisor:

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

Co-Referee:

Prof. Dr. Felix Joos

Further advisors:

Dr. Ernestine Großmann

M. Sc. Adil Chhabra

Prof. Dr. Darren Strash

Acknowledgments

I would like to thank my advisors Dr. Ernestine Großmann and Adil Chhabra for their extremely helpful and constant support in the creation of this work. Furthermore, I would like to thank Prof. Christian Schulz for his numerous valuable suggestions and continuous generation of ideas. I would also like to thank Prof. Darren Strash for his constructive support.

Finally I would like to thank my friends and family for their support.

Hiermit versichere ich, dass ich die von mir am 22.10.2025 als PDF-Datei (mit Standard-PDF-Lesesoftware lesbar und durchsuchbar) eingereichte Master-Arbeit mit dem Titel "Engineering Semi-Streaming Maximum Independent Set Algorithms" betreut von Prof. Dr. Christian Schulz selbstständig und ohne unzulässige fremde Hilfe verfasst habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel verwendet und sämtliche Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder unveröffentlichten Werken - einschließlich solcher aus digitalen oder KI-basierten Quellen - übernommen wurden, als solche kenntlich gemacht. Ich bestätige, dass die etwaige Nutzung von KI-Tools vorab mit der Betreuerin/dem Betreuer der Abschlussarbeit abgesprochen wurde und dass die besprochenen Regelungen eingehalten wurden. Ich übernehme die volle Verantwortung für die wissenschaftliche Qualität und den Inhalt der vorliegenden Arbeit, die gewählte Methodik und den Erstellungsprozess, sowie die zitierte Literatur. Ich bin damit einverstanden, dass meine Arbeit im Rahmen universitärer Verfahren auf Plagiate sowie auf den möglichen Missbrauch automatisierter Text- und Codegenerierung überprüft wird.

Kopenhagen, October 22, 2025



Benedikt Vidic

Abstract

This thesis presents semi-streaming algorithms designed to find high-quality Maximum Independent Sets (MIS) without requiring the entire graph to be loaded into memory, addressing a gap in practical semi-streaming approaches that balance memory efficiency with solution quality. The MIS problem is a classic NP-hard problem with applications across various domains, such as computer graphics, map labeling, and information coding. As modern graphs continue to grow, the necessity for algorithms that do not require space linear in the graph size becomes increasingly important, motivating the development of semi-streaming methods that assume the nodes of a graph to fit in memory but not the edges.

We introduce two semi-streaming algorithmic approaches that are both based on graph partitioning: block-wise solving and block-wise reductions. Each of the general approaches is executed in multiple variants. The block-wise solving approach includes the Base Algorithm, Repartitioning, and Boundary Local Search variants, while the block-wise reduction approach includes the Basic Reductions and Reductions+VF (Vertex Fold) variants. All approaches incorporate a preprocessing step, applying simple reductions in a single stream.

Experimental evaluation on large instances demonstrates that our algorithms generally produce higher quality solutions than the existing competitor semi-streaming algorithms developed by Liu et al. on the majority of instances. The Reduction algorithms achieved the highest solution quality among our methods. The Base Algorithm provided the fastest running time and lowest memory footprint among the developed approaches.

However, the partitioning-based approach has two notable downsides: first, memory usage is significantly larger than competitors due to the inherent overhead of partitioning and loading block-graphs into memory. Second, result quality depends on the quality of the partitioning, where outliers with extremely large edge-cuts can lead to significantly poor solution quality.

In an evaluation on smaller instances which allows a comparison with in-memory algorithms, the Reductions algorithm achieves on average a solution quality of 98.8% compared to ReduMIS.

Contents

Abstract	v
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 Independent Sets	3
2.2.1 Local Search	4
2.2.2 Kernelization	4
2.3 Graph Partitioning	5
2.4 Computational Model	5
3 Related Work	7
3.1 In-Memory Algorithms	7
3.1.1 Exact Algorithms	7
3.1.2 Heuristic Algorithms	8
3.2 Streaming Algorithms	8
4 Main Contribution	11
4.1 Preprocessing and Partitioning	11
4.1.1 Preprocessing	11
4.1.2 Partitioning	16
4.2 Block-Wise Solving	16
4.2.1 Base Algorithm	16
4.2.2 Repartitioning	18
4.2.3 Boundary Local Search	20
4.3 Block-Wise Reductions	23
4.3.1 Basic Approach	24
4.3.2 Vertex Fold Extension	26

5	Experimental Evaluation	29
5.1	Setup	29
5.2	Parameter Study	30
5.2.1	Preprocessing	30
5.2.2	Edge Estimation Method	31
5.2.3	Block solver	33
5.2.4	Scheduling	34
5.2.5	Block Size	36
5.3	Algorithm Comparison	40
6	Discussion	49
6.1	Conclusion	49
6.2	Future Work	50
	Abstract (German)	51
	Bibliography	53

Introduction

We begin by motivating the maximum independent set problem in its general form and the usage of streaming algorithms. We continue with an overview of our contribution and the structure of this thesis.

1.1 Motivation

Given a graph $G = (V, E)$, where V is the set of vertices and E is the set of edges, the Maximum Independent Set (MIS) problem asks for a set $I \subseteq V$ of maximum cardinality such that no two vertices in I are adjacent. The MIS problem is a classic NP-hard problem [15] with applications in various domains, including computer graphics [32], map labeling [16, 33], information coding [6], and indexing for shortest path and distance queries [13]. In the map labeling problem, the goal is to maximize the number of labels displayed on a map without any overlaps. Each label is modeled as a vertex, and each overlap between two labels creates an edge between the corresponding vertices. Finding a maximum independent set on this graph is equivalent to finding a maximum set of non-overlapping labels [16, 33].

While finding optimal solutions can be difficult, in practice, high-quality solutions are often sufficient. Numerous algorithms exist for finding large independent sets [28, 2, 26, 18, 24], but most require space linear in the graph size. As graphs continue to grow larger, it is increasingly impractical to store them entirely in memory.

This motivates the development of streaming and semi-streaming algorithms. In practice, the number of vertices in a graph is often significantly smaller than the number of edges. Therefore, a reasonable assumption for large graphs is that all vertices fit in memory, but the complete edge set does not. While streaming algorithms for independent sets have been studied theoretically [21, 9, 7], there are only few practical implementations. Liu et al. [30] developed a greedy algorithm and a vertex-swap framework that work in a semi-streaming model while primarily prioritizing memory efficiency. So far, there is a gap in

practical semi-streaming algorithms that balance memory efficiency with solution quality. In this thesis, we introduce semi-streaming algorithms based on graph partitioning that aim to find high-quality independent sets without loading the entire graph into memory.

1.2 Our Contribution

This thesis presents semi-streaming algorithms for the maximum independent set problem based on graph partitioning. We develop two general approaches: block-wise solving and block-wise reductions, each implemented in multiple variants.

The core of the block-wise solving approach is the Base Algorithm, which sequentially solves blocks of a partitioning. Additionally, we present two extensions: Repartitioning, which computes a second partitioning and performs additional local search, and Boundary Local Search, which applies local search specifically to boundary regions between blocks. The block-wise reductions approach includes two variants: the basic Reductions algorithm and Reductions+VF. Both follow the same general approach but differ in implementation, with the latter supporting vertex folding reduction rules that alter graph structure in a way that requires a different handling of cut edges.

All approaches include a preprocessing phase that applies simple reductions during a single stream over the graph. This reduces graph size before computing the partitioning and executing the main algorithms.

1.3 Structure

The remainder of this thesis is organized as follows. Chapter 2 introduces the notation and general concepts relevant to understanding our algorithms. Chapter 3 provides an overview of related work on independent set algorithms and streaming graph algorithms. Chapter 4 presents the main contribution of this work: two algorithmic approaches for computing independent sets in a streaming setting. Chapter 5 evaluates the algorithms, first examining individual algorithm components to determine optimal configurations, then comparing the overall algorithm performance. Chapter 6 concludes and outlines the directions for future work.

Fundamentals

This section introduces the formal preliminaries and fundamental concepts required to understand the algorithms presented in this thesis. We begin with general graph notation, then cover independent sets and relevant solution techniques, graph partitioning, and finally the computational model used throughout this work.

2.1 General Definitions

Let $G = (V, E)$ be an undirected unweighted graph with $|V| = n$ nodes and $|E| = m$ edges. We only consider simple graphs, meaning that they do not contain multi-edges or self-loops. An edge $e = \{u, v\}$ is said to be incident to nodes u and v . The nodes u and v are then called adjacent to each other.

The *neighborhood* of a vertex $v \in V$ is defined as $N(v) = \{u \in V \mid \{u, v\} \in E\}$. The *closed neighborhood* is $N[v] = N(v) \cup \{v\}$. The *degree* of a node $v \in V$ is derived from the size of the neighborhood, $\deg(v) = |N(v)|$. A graph $G' = (V', E')$ is called a *subgraph* of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$. A subgraph $G' = (V', E')$ where $E' = \{\{u, v\} \mid \{u, v\} \in E \wedge u, v \in V'\}$ is called *induced* by V' . Induced subgraphs are denoted as $G[V']$. The *density* d of a graph is the number of edges in the graph divided by the maximum possible number of edges $\frac{2m}{n \cdot (n-1)}$.

While this thesis primarily works with unweighted graphs, we occasionally extend the model to weighted cases, $G = (V, E, c, \omega)$, where $c : V \rightarrow \mathbb{R}_{\geq 0}$ is a node-weight function, and $\omega : E \rightarrow \mathbb{R}_{> 0}$ is an edge-weight function.

2.2 Independent Sets

A set of nodes $I \subseteq V$ is called an *independent set* (IS) if $\forall u, v \in I : \{u, v\} \notin E$. An IS I is *maximal* if there is no node $v \in V \setminus I$ such that $I \cup \{v\}$ is still an independent set. A

maximum independent set (MIS) is an IS with maximum cardinality. The MIS-problem, which is finding a maximum independent set, is NP-hard [15]. In the case of weighted graphs, the weight of an independent set I is defined as $c(I) = \sum_{v \in I} c(v)$. A maximum weighted independent set (MWIS) is an independent set with maximum total weight.

2.2.1 Local Search

In general, local search is a technique that starts from an initial feasible solution and iteratively improves it through local modifications. In the context of independent sets, these are usually swaps: a node is removed from the independent set, thereby enabling the addition of one or more nodes that were previously not part of the independent set. Local search algorithms do not guarantee optimality, but can often find good solutions in practice.

In our algorithms, we mainly apply local search to improve independent sets, using the Concurrent Hybrid Iterated Local Search (CHILS) algorithm [18]. CHILS mostly makes use of simple and thereby fast local search operations. A key feature of CHILS is that it maintains multiple concurrent solutions, applying local search improvements on each of them. Then, a "difference-core" instance is computed. The difference-core is a sub-graph that gets created based on the difference of the concurrent solutions. Local search is then applied to this difference-core, and the improved partial solution is embedded back into all concurrent solutions. CHILS alternates between local search on the concurrent solutions and solving the difference-core in multiple iterations. While CHILS can be configured to run in parallel, we use it only sequentially in this work. It is also worth noting that CHILS incorporates randomness, for example in the selection of nodes considered for improvement operations. As a result, CHILS and our algorithms employing it are non-deterministic.

2.2.2 Kernelization

Another technique that is used in the process of solving the MIS problem is the application of *data reduction rules*. Reductions are transformations that simplify a problem instance into a smaller, equivalent instance. For MIS, reductions simplify the graph by removing nodes and edges or replacing local structures in the graph with simpler ones, while maintaining the ability to reconstruct a maximum independent set of the original graph from a solution on the reduced graph. This reconstruction process is called *solution lifting*. A reduced graph is called an *irreducible kernel* if no more reductions can be applied.

When a reduction preserves the solution optimality, we call it an *exact* or *optimal* reduction. When optimality is not guaranteed, we call it a *heuristic* reduction.

We use several reduction-based tools in our algorithms. The ReduMIS algorithm from the Karlsruhe Maximum Independent Sets (KaMIS) project [26] combines exact and heuristic reductions to fully solve MIS instances. Since it uses heuristic reductions, it provides no optimality guarantee. We use ReduMIS to solve the MIS problem on smaller subgraphs that fit in memory entirely.

The DataReductions Library [17] implements a collection of reduction techniques. In our algorithms, we use this library to kernelize subgraphs, and subsequently apply CHILS to the reduced instances.

ParFastKer [24] is an MIS solver that, among other elements, applies reductions in parallel in a shared-memory setting. In order to avoid conflicts when applying reductions in parallel, it implements a reduction framework that can be applied to a part of the graph while guaranteeing that no conflicts with nodes outside this part can occur. An example of a conflict would be two reductions executed in parallel that add nodes to the independent set that are adjacent to each other. Therefore, no reductions that include boundary nodes in the IS are applied. However, boundary nodes getting excluded and thereby removed from the graph is valid. While our algorithms do not run in parallel, we make use of reductions with this property in order to kernelize subgraphs while not having access to the full graph.

2.3 Graph Partitioning

For $k \in \mathbb{N}_{>0}$, a k -partitioning of a graph $G = (V, E)$ is a division of V into k pairwise disjoint sets $V_1, V_2, \dots, V_k$ such that $\bigcup_{i=1}^k V_i = V$. The induced subgraphs $G[V_i]$ are called *blocks*. A partitioning is called ϵ -balanced if for all blocks V_i , $|V_i| \leq (1 + \epsilon) \lceil \frac{n}{k} \rceil$ for some small $\epsilon \geq 0$. For weighted graphs, where the weight of a block is defined as the sum of the weights of its nodes, a partitioning is ϵ -balanced if $c(V_i) \leq (1 + \epsilon) \lceil \frac{c(V)}{k} \rceil$ for all blocks V_i .

The objective of graph partitioning is to minimize the number of *cut edges* $\{\{u, v\} \in E \mid u \in V_i, v \in V_j, i \neq j\}$. The size of the set of cut edges is also referred to as the *edge-cut*. For graphs with edge weights, the objective is to minimize the total weight of cut edges. Nodes that are incident to at least one cut edge are called *boundary nodes*. The graph obtained by contracting each block into a single node is called the *quotient graph*.

In our algorithms, we use HeiStream [12], a streaming graph partitioning algorithm that uses a buffered streaming approach. It reads batches of nodes and assigns them to blocks using the Fennel objective function [34]. This produces relatively high-quality partitionings while maintaining low memory usage. HeiStream operates in a single pass over the graph, though it can be configured to use further passes in order to improve the solution.

2.4 Computational Model

The algorithms in this thesis target the maximum independent set problem for streaming inputs. The input is provided as a stream of nodes, each accompanied by its adjacency list. Algorithms may make multiple passes over the input stream.

We assume that the graph is too large to fit entirely in memory. In the classical streaming model, memory is restricted to $O(\text{polylog}(n))$. We work in the *semi-streaming model*,

which allows $O(n \cdot \text{polylog}(n))$ memory but not $O(m)$ memory. Our algorithms also make use of external memory, placing them in the category of semi-external algorithms.

Related Work

This section surveys the state-of-the-art in algorithms for the maximum independent set problem. We begin with in-memory algorithms, including parallel approaches, then discuss streaming and semi-streaming algorithms that address memory-constrained settings.

3.1 In-Memory Algorithms

Most research on the MIS problem has focused on in-memory algorithms that assume the entire graph fits in memory.

3.1.1 Exact Algorithms

The MIS problem can be formulated as an integer linear program (ILP) and solved using generic ILP solvers [23]. This approach yields optimal solutions for small instances but becomes infeasible as instance size grows.

Branch-and-bound algorithms offer a more commonly used approach to compute exact solutions [35]. The method recursively splits the problem into subproblems (branching) while using bounds to prune subproblems that cannot improve upon the best solution found so far (bounding). This achieves optimal solutions for instances with hundreds of nodes [35].

Combining branch-and-bound with reduction rules yields the branch-and-reduce approach [1]. Lamm et al. introduced KaMIS_wB&R [27], an exact and fast solver that implements this branch-and-reduce approach. Further enhancements, like a more sophisticated selection of determining on which node to branch on next by Langedal et al. [28], have enabled solving some instances with hundreds of thousands of nodes optimally.

3.1.2 Heuristic Algorithms

Introducing heuristic elements allows solving larger instances, though without optimality guarantees. Local search algorithms iteratively improve solutions through local modifications. The ARW algorithm by Andrade et al. [2] uses iterated local search based on (1,2)-swaps: removing one node from the independent set and adding two nodes. It performs well on small to medium instances but struggles with solution quality on larger instances with millions of nodes. Combining local search with reduction rules has proven effective, as shown by Dahlum et al. in the OnlineMIS algorithm [10] that combines these two techniques.

Lamm et al. introduced EvoMIS [25], an evolutionary algorithm that is based on graph partitioning in order to split the graph into well-combinable subgraphs and then applies local search to improve combination operations. Building on this, Lamm et al. also proposed ReduMIS [26], which combines an evolutionary algorithm with branch-and-reduce techniques. This allows for the efficient computation of large independent sets and is faster than existing local search algorithms. On smaller graphs, it produces solutions comparable in quality to those found by exact algorithms, while also being applicable to very large sparse graphs with billions of edges.

A recent addition by Großmann et al. is CHILS [18], an iterated local search algorithm that maintains multiple concurrent solutions. When combined with reduction-based preprocessing, CHILS outperforms the current state-of-the-art on large instances [18]. While primarily designed for weighted independent sets, it is also applicable to the unweighted case.

Parallel Algorithms

Further scalability can be achieved through parallel algorithms. A simple parallel algorithm for MIS was introduced by Luby et al. in 1985 [31]. More recent work has focused on parallel reduction techniques. Hespe et al. [24] present ParFastKer, which makes use of the locality of many reduction rules to apply them in parallel on different graph regions. Operating in a shared-memory setting, ParFastKer has similar memory limitations as sequential algorithms while achieving comparable results in less time. Borowitz et al. [5] propose a distributed memory algorithm, mainly building on parallel reduction techniques, that solves even larger instances.

3.2 Streaming Algorithms

In-memory algorithms are limited by graph size. Streaming and semi-streaming approaches address this limitation by processing graphs that do not fit entirely in memory.

So far, streaming algorithms for independent sets have been studied primarily from a theoretical perspective, with a focus on establishing computational bounds rather than practical performance.

Cormode et al. [9] study the vertex-arrival model, where vertices arrive one by one with all edges to previously arrived vertices, as opposed to the edge-arrival model, where edges arrive one by one in arbitrary order. They show that a one-pass c -approximation algorithm for MIS requires $\Omega(\frac{n^2}{c^2})$ space on general vertex streams, even when disregarding computation time. This shows that the vertex-arrival model is not substantially easier than the edge-arrival model, where $\tilde{\Theta}(\frac{n^2}{c^2})$ space is necessary [22]. Assadi et al. [3] prove that in the edge-arrival model, any semi-streaming algorithm requires at least $\Omega(\log \log n)$ passes to find a maximal independent set with constant probability. Ye et al. [36] establish bounds for deterministic single-pass algorithms, proving that such algorithms in the semi-streaming model can only find independent sets of size $\tilde{O}(\frac{n}{\Delta^2})$ on general graphs with maximum degree Δ .

Several theoretical algorithms have been developed with provable guarantees in relation to the Caro-Wei bound $\lambda = \sum_{v \in V} \frac{1}{d(v)+1}$, a general lower bound for the size of a maximum independent set. Halldórsson et al. [20, 21] present a randomized one-pass algorithm that finds an independent set with size at least λ in expectation, using $O(n)$ space. Cormode et al. [8] provide a one-pass algorithm that $(1 \pm \epsilon)$ -approximates the Caro-Wei bound with constant success probability using $O(\epsilon^{-2} \cdot d_{\text{avg}} \cdot \log n)$ space.

Chen et al. [7] develop sublinear-space streaming algorithms for sparse graphs. Their approach greedily adds a node v to the solution given that v is earlier in a random permutation than all its neighbors. To avoid storing the permutation explicitly, they use a hash-based simulation of a random permutation. This achieves the Caro-Wei bound in expectation while using sublinear memory.

However, these works remain theoretical without practical implementation. Approaches developed with the goal of practical applicability are the semi-streaming algorithms by Liu et al. [30], which we discuss next.

Liu et al. [30] propose three semi-streaming algorithms for the MIS problem that mainly prioritize memory efficiency, a greedy and two swap-based algorithms. The greedy algorithm performs a single pass over the graph, greedily adding nodes to the independent set. It assumes nodes in the input file are sorted by degree in ascending order. Their swap algorithms perform local search like operations with one- k and two- k swaps. Only the state of each node is kept in memory, while adjacency lists are accessed through full streams over the graph. Each swap requires three passes over the input. During these three passes, as many swaps as possible are executed. The algorithm repeats iterations of three-pass swap operations as long as improvements are found.

While Liu et al.'s algorithms achieve very low memory consumption, there remains a gap in practical streaming algorithms that balance memory efficiency with solution quality.

Main Contribution

The following chapter provides a detailed explanation of the algorithms we developed. After describing the common basis of preprocessing and partitioning in Section 4.1, we explain the two general approaches to computing MIS in a streaming setting: block-wise solving and reducing to a global kernel in Section 4.2 and Section 4.3 respectively. Each approach follows a general idea executed in different configurations with varying trade-offs in terms of running time, memory consumption, and result quality.

4.1 Preprocessing and Partitioning

Each of our proposed approaches follows the same broad structure, which can be briefly summarized as follows. First, a preprocessing step identifies nodes that can be trivially removed from the graph and creates a mapping to a reduced graph that excludes these nodes. Then, a partitioning is computed on the reduced graph which is then, in an additional stream, used to copy each block to a separate file. These resulting blocks are then used by the different algorithms described in the later sections in order to compute an independent set for the reduced graph. After that, the independent set of the reduced graph is lifted to a solution for the entire graph. Algorithm 1 shows an overview of this preprocessing element that is used in the later algorithms and is explained in detail in the following.

4.1.1 Preprocessing

Before partitioning the graph, the algorithm starts by streaming over the graph once to determine which nodes to remove. The main motivation for this is that nodes with extremely large degrees can have negative impact on the partitioning quality under balance constraints, especially in the streaming partitioning setting. At the same time, it is very unlikely that any maximum independent set includes these nodes. Identifying and removing

Algorithm 1 preprocessAndPartition**Input** : Graph G , number of blocks k **Output**: Independent Set I , partitioning p , mapping to reduced graph m **Function** preprocessAndPartition(G, k):

```

 $I \leftarrow \emptyset$ 
 $I, m \leftarrow \text{streamReduce}(G)$ 
 $p \leftarrow \text{heStream}(G, m, k)$ 
extractBlocksAsFiles( $G, m, p$ )
return  $I, p, m$ 

```

these nodes beforehand avoids those problems while simultaneously offering the possibility to apply further simple reduction techniques in order to decrease the size of the graph and by that speed up any subsequent algorithms.

Reductions that can be applied while streaming over the graph are limited to those that do not require more information than the immediate neighbors of a node. These reduction rules are based on the node degree and remove nodes with especially small or large degrees.

Exact reductions. For the small degree nodes, we apply the degree-0 and degree-1 reductions, meaning marking all nodes u with $\deg(u) \leq 1$ as "to be removed" and adding them to the IS. For the degree-1 nodes, we also mark their respective neighbor as "to be removed". While counting the degree of a node, we only consider nodes that have not yet been marked as removed.

The effectiveness of this largely depends on the node order. While it is possible to keep track of the node degrees in order to recognize that removing some nodes leads to new degree-1 nodes that can possibly be reduced, we do not have access to the neighbors of past nodes and therefore would not be able to apply the reduction. Figure 4.1 illustrates an example where the node order impacts the applicability of reductions. Therefore the resulting subgraph is still potentially degree-1 reducible. Doing multiple streams over the graph to catch all of these is not worth it because the expected reduction is relatively small while each stream over the full graph is expensive.

Heuristic reductions. Heuristically removing nodes with large degrees initially aims at removing nodes whose number of incident edges is large enough such that they cause problems for the partitioning. In order to compute blocks with similar memory size, we compute a partitioning on a weighted graph, assigning each node v the weight $\deg(v) + 1$, in order to account for both the node itself and its edges, which is further explained in Section 4.1.2. Due to this weighting, any node u with $\deg(u) > \frac{|V|+2 \cdot |E|}{k}$ makes a balanced partitioning impossible while also leading to large edge-cuts. Additionally, we can optionally remove nodes that have significantly more neighbors than the average node. This is not strictly necessary, but it is generally a sensible heuristic reduction, as these nodes are unlikely to belong in any optimal IS. This is done in other MIS approaches, like online-MIS [10]. We know the average node degree to be $\frac{1}{|V|} \sum_{u \in V} \deg(u) = \frac{2 \cdot |E|}{|V|}$. Since we can not compute the standard deviation without performing a full stream over the graph

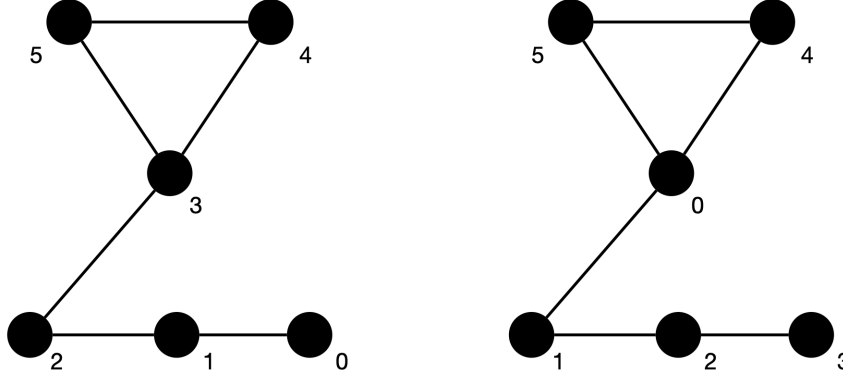


Figure 4.1: This figure shows two graphs that are identical in structure. The numbers next to the nodes indicate the order in which the nodes appear in the input stream. The extent to which the graphs get reduced during the preprocessing reductions varies depending on this ordering. After a single stream, the first graph gets completely reduced by multiple applications of the degree-1 reduction. For the second graph, the degree-1 reduction can only be applied once.

beforehand, we define significantly large node degrees as those larger than the average node degree multiplied by a constant factor. Both of these node removals are heuristic and not necessarily optimal. Therefore, any independent set computed on the resulting subgraph is potentially not maximal.

Preprocessing application. The preprocessing step only identifies which nodes should be removed and creates a mapping to a subgraph excluding these nodes. It does not write the reduced subgraph into a file, instead the mapping is applied on the fly whenever streaming over the graph. Writing the reduced graph in a new file during a second stream would be slower in most cases. However, depending on how much the graph got reduced and how often the subsequent algorithms stream over the graph, there can be some rare cases where this would be faster.

After the algorithms solving the MIS on the reduced subgraph finish, we need to map the solution for the subgraph back to the full graph. Furthermore, the degree-0 and degree-1 reductions of the preprocessing not only remove nodes from the graph, but also add them to the independent set. Therefore, as a final step, we add these to the independent set.

Size of the Reduced Graph

A challenging part of reducing the graph in a streaming setting is keeping track of the number of edges that remain in the reduced graph. We need to provide this number as an

input for HeiStream, in order to configure input parameters that are required to maintain balance. We start by explaining the issues that prevent us from obtaining the exact edge count. Then, we present three different approaches to estimate the number of edges in the reduced graph.

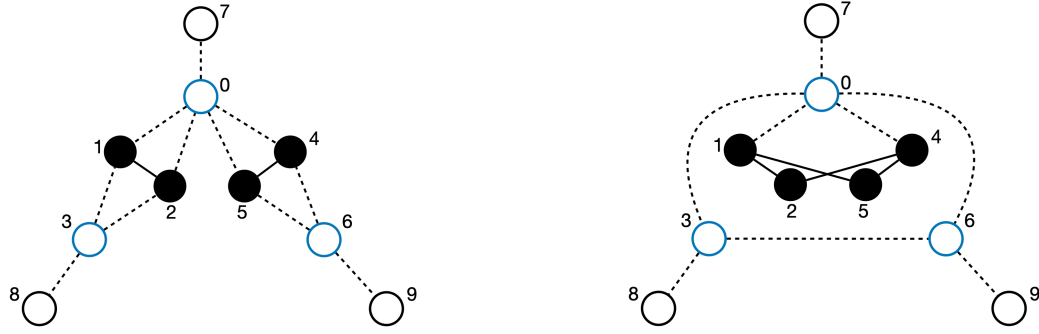
Problem description. The naive way of obtaining the precise number of remaining edges is counting during another full stream over the graph. However, streaming over the graph is relatively slow, especially for large input graphs. The alternative is maintaining an edge count while reducing the graph. The difficulty here lies in removing an edge only once, regardless of whether one or both adjacent nodes get removed from the graph. In most cases we can keep track of this by, whenever removing a node, reducing the edge count of this node to zero and reducing the degree of each neighboring node by one. There is however a scenario where this does not work. Lets say some node u has degree one and a neighbor v with $v < u$. The id of v being smaller than the id of u means that its neighborhood has been read in the past and will not appear again. We call nodes like v "post-read removed nodes". As we can not store the neighbors of all nodes, we only know how many but not which neighbors v has. Therefore, we can not directly reduce the edge count of the neighbors of v by one. Any attempts at separately accounting for these edges fail in the case where v has neighbors that are also post-read removed nodes. There is no way of detecting these cases and avoiding to possibly remove the same edge twice. Figure 4.2 illustrates how this creates situations where the knowledge about the reduced graph is not sufficient to determine the correct edge count. Therefore we need to derive our edge count in a different way.

Estimation methods. After applying the reductions, the remaining graph contains two sets of nodes: the set of nodes that are marked as kept A and the set of post-read removed nodes B . We know the degrees of each of these nodes, which gives us the two key metrics to derive an edge count for the reduced graph: $m_a = \sum_{a \in A} \deg(a)$ and $m_b = \sum_{b \in B} \deg(b)$, representing the number of edge endpoints incident to kept nodes and the number of edge endpoints incident to post-read removed nodes respectively.

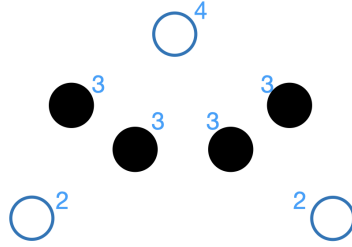
The edges in this graph can be grouped in three sets: the set of edges connecting nodes within A , $E_{AA} = \{(u, v) | u, v \in A\}$, the set of edges connecting nodes within B , $E_{BB} = \{(u, v) | u, v \in B\}$ and the set of edges connecting nodes between A and B , $E_{AB} = \{(u, v) | u \in A, v \in B\}$. By definition, $m_a = 2 \cdot |E_{AA}| + |E_{AB}|$ and $m_b = 2 \cdot |E_{BB}| + |E_{AB}|$.

The simplest way to derive an estimate for $|E_{AA}|$ from this is taking $\frac{m_a}{2}$. This is an upper bound to the actual number, as $\frac{m_a}{2} = |E_{AA}| + \frac{|E_{AB}|}{2}$, meaning the sum of degrees of the nodes in A divided by two includes every edge between the kept nodes and additionally edges between a kept node and a post-read removed node.

We can try to obtain a closer value to $|E_{AA}|$ by estimating based on the graph properties we know. We consider two different variants: a node-based and an edge-based estimation. For the node-based estimation, we assume that each edge endpoint incident to a node in A connects to a random node. Given m_a edge endpoints and $|A| + |B|$ total nodes of which $|A|$



(a) Each of the two graphs are shown in the state after applying one preprocessing-reduction stream. The numbers next to the nodes indicate their order in the input stream. Nodes appear in three possible states: *kept* (filled circles), *removed* (unfilled outlines), and *post-read removed* (blue outlines). Edges are shown as *kept* (solid lines) or *removed* (dotted lines). In both graphs, nodes 7, 8 and 9 got removed by the degree-1 reduction rule, turning each of the nodes 0, 3 and 6 into post-read removed nodes.



(b) This figure illustrates the information available after the reductions. The exact edge set is no longer known. Only the node states and their degrees after reduction are retained, with the latter shown in blue. Note that the degrees still include edges to post-read removed nodes. When these nodes were removed, their neighbors were no longer accessible, preventing degree updates for those neighbors.

Figure 4.2: Two slightly different graphs are shown. After applying the reduction process, both yield identical node states and degree information. At the point when nodes are removed, access to the neighbors of post-read removed nodes is lost. Therefore, the available information after reduction is insufficient to infer the original edge count: the first graph contains two remaining edges, while the second has four.

are kept, we get an estimation of $m_a \frac{|A|-1}{|A|-1+|B|}$ considering we exclude the possibility of self-edges.

For the edge-based estimation, we assume each edge to be placed randomly in the graph. For this we assume each endpoint of an edge to connect to a random node, considering the node degrees. We have $\frac{m_a+m_b}{2}$ edges of which each has a chance of $(\frac{m_a}{m_a+m_b})^2$ to be placed with both edge endpoints in A . This results in an estimated edge count of $\frac{m_a^2}{2 \cdot (m_a+m_b)}$.

Having established these methods for estimating edge counts, we can now proceed to the partitioning phase, which relies on an accurate edge count to compute a balanced division of the graph.

4.1.2 Partitioning

The next step computes a partitioning on the preprocessed graph to create smaller, local sections of the graph that can be processed separately. The goal is to create blocks that require a similar amount of memory when loaded.

We use HeiStream [12], a streaming graph partitioner, with slight adaptations. First, we apply the mapping to the subgraph that was computed during the preprocessing. HeiStream loads the graph into memory in batches. While reading in these batches, we apply the mapping. Nodes and edges that are marked as removed are ignored, and remaining elements are mapped to their new ids in the reduced graph. The configured batch-size refers to the reduced graph, where removed nodes and edges are not counted.

These weights indicate how much memory this node requires such that a balanced partitioning corresponds to an equal memory size of each block.

Furthermore, we add weights to the nodes. The goal is to add the weights, such that a balanced partitioning corresponds to an equal distribution of memory size per block. We approximate the distribution of the required memory for a graph by setting the total weight to $|V| + 2 \cdot |E|$ and weighting each node u with $\deg(u) + 1$. Thereby, we assume each node and incident edge to require the same amount of memory. It would also be possible to apply other weightings here, that more accurately reflect the actual memory usage of subsequent algorithms per node and edge. However, the assumption of each node and edge requiring the same amount of memory is a good approximation for most algorithms and graph storage formats.

After HeiStream completes, we perform another stream over the graph and store each block of the just computed partitioning in a separate binary file. During this process, we read the graph into a buffer while applying the mapping to the subgraph. When the buffer fills, we flush it to the corresponding binary files. These binary files enable reading each block in isolation without requiring a full stream to acquire this block.

4.2 Block-Wise Solving

The main idea of the block-wise solving approach is to find an independent set on the full graph by solving each block independently. In order to be able to treat a block as an independent subgraph, we remove boundary nodes that could potentially create conflicts with neighboring blocks. Building on this idea, we implemented three algorithms. Section 4.2.1 introduces the Base Algorithm while sections 4.2.2 and 4.2.3 introduce Repartitioning and Boundary Local Search, each an extension of the Base Algorithm.

4.2.1 Base Algorithm

After the preprocessing step, we handle the blocks sequentially, completely solving a block before reading in the next. We start by reading the block from disk and removing nodes

Algorithm 2 Structure of Base Algorithm**Input** : Graph G , number of blocks k **Output:** Independent Set I $I \leftarrow \text{preprocessAndPartition}(G, k)$ **for** $i \leftarrow 1$ **to** k **do** $I \leftarrow \text{solveBlock}(i, I, \text{false})$ **return** I **Algorithm 3** solveBlock **Input** : block number i , Independent Set I , boolean $\text{useInitialSolution}$ **Output:** Independent Set I **Function** $\text{solveBlock}(i, I, \text{useInitialSolution})$: $(V_i, E_i) \leftarrow \text{loadBlockFromFile}(i)$ $V'_i \leftarrow \text{removeNodeWithNeighborsInIS}(V_i, I)$

/* induce subgraph */

 $G'_i \leftarrow G[V'_i]$ **if** $\text{useInitialSolution}$ **then** $I \leftarrow \text{CHILS}(G'_i, I)$ **else** $I_i \leftarrow \text{runExistingSolver}(G'_i)$ $I \leftarrow I \cup I_i$ **return** I

that potentially create conflicts. A conflict occurs when two nodes from different blocks that are connected with a cut edge get added to the independent set. Therefore, first we remove all nodes that have a neighbor that is already in the independent set. Second, for boundary nodes not connected to a node in the independent set, we keep the node and remove only the incident cut edges. We do not need to remove these nodes, because they are not adjacent to any node in the current independent set and can thereby not directly create a conflict. In case this node does get added to the independent set, adjacent nodes from neighboring blocks will be removed when processing these blocks. This creates an independent subgraph that we map into a consecutive id-space and solve using non-streaming algorithms. Algorithm 2 shows an overview of this overall procedure, using a subroutine outlined in Algorithm 3.

We consider three algorithms for solving these subgraphs. The first option is ReduMIS [26], a reduction-based solver. ReduMIS applies exact and heuristic reduction rules to fully solve MIS instances, though the solution is not necessarily optimal. The second variant is CHILS [18], a fast local search algorithm that alternates between improving multiple concurrent solutions and solving the difference-core, the parts of the graph where the solutions differ. The third option, R+CHILS, combines the DataReductions [17] library with CHILS. The DataReductions library includes fast and exact reduction rules that, unlike ReduMIS, do not produce a complete solution but reduce the block-graph to a kernel. We

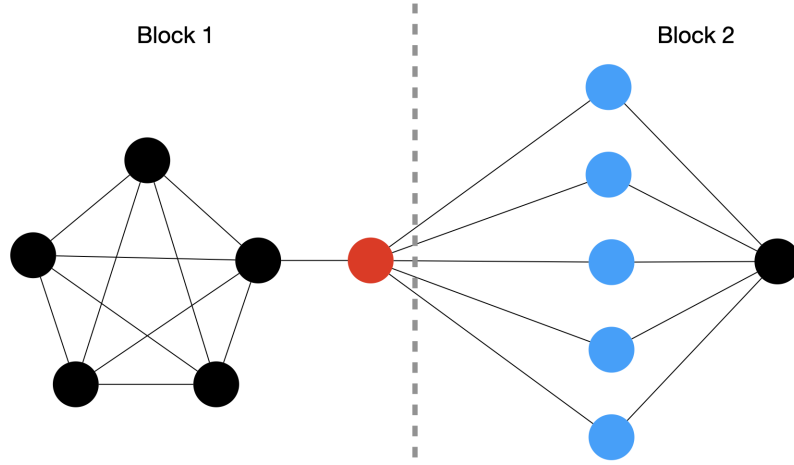


Figure 4.3: The figure shows a 2-partitioning on a graph. When being solved, the Base Algorithm removes all cut edges before solving Block 1. This leaves the red node as a deg-1 node which means it gets added to the IS. When later solving Block 2, all the blue nodes get removed because the red node has already been added to the IS and thereby excludes all its neighbors.

solve this kernel using the local search algorithm CHILS. We then lift this kernel solution to a full subgraph solution by reconstructing the stored reductions.

After solving each block, we obtain a valid solution for the full graph, though the solution usually is not optimal, leaving room for improvement especially in parts of the graph that are close to boundaries between blocks. In the process of making subgraphs independent, we greedily remove boundary nodes and cut edges and solve the simplified version of these graphs. This leads to suboptimal decisions, made because of missing context. These suboptimal decisions are considered final, and the solving of subsequent neighboring blocks is based on them. The Base Algorithm does not reconsider these decisions at any point. Consider Figure 4.3 for an example where this can lead to far from optimal solutions.

The common way to improve an existing solution is local search. The difficulty with local search in a streaming setting is obtaining a local section of the graph on which local search can be executed. In the current model, we only have access to the graph within a single block, making local search across boundaries not possible. In the following we explain approaches to improve upon solutions found by the Base Algorithm.

4.2.2 Repartitioning

The Repartitioning approach extends the Base Algorithm by repeating the procedure with a different partitioning, as outlined in Algorithm 4. The general idea is that the blocks

Algorithm 4 Structure of Repartitioning Algorithm**Input** : Graph $G = (V, E)$, number of blocks k **Output**: Independent Set I

```

 $I, p, m \leftarrow \text{preprocessAndPartition}(G, k)$ 
for  $i \leftarrow 1$  to  $k$  do
   $I \leftarrow \text{solveBlock}(i, I, \text{false})$ 
 $\omega \leftarrow \text{cutEdgeWeighting}(p)$ 
 $G' \leftarrow (V, E, \omega)$ 
 $p_{\text{new}} \leftarrow \text{heiStream}(G', m, k)$ 
 $\text{extractBlocksAsFiles}(G, m, p_{\text{new}})$ 
for  $i \leftarrow 1$  to  $k$  do
   $I \leftarrow \text{solveBlock}(i, I, \text{true})$ 
return  $I$ 

```

resulting from the second partitioning are expected to contain sections of the graph that were distributed across multiple blocks in the previous partitioning.

After applying the Base Algorithm, we compute a second partitioning with the same k . To create a partitioning that differs from the first, we assign large edge weights to the cut edges of the previous partitioning and run a partitioning that optimizes for the weighted edge-cut. To minimize overhead, we assign the edge weights during the in-memory reading of graph batches while executing HeiStream, rather than creating a weighted copy of the graph beforehand. The preprocessing is not repeated, HeiStream is called with the same map to a subgraph from the initial preprocessing.

Using this new partitioning, we repeat the procedure of the Base Algorithm. We write the blocks as binary files, handle each block individually and remove boundary nodes to create independent subgraphs. However, instead of solving subgraphs from scratch, we build upon the solution from the first iteration and use CHILS to find improvements through local search. Since the partitioning differs from the first one, the blocks now contain sections of the graph that were previously spread across multiple blocks, as illustrated in Figure 4.4. Local searching on these sections is expected to lead to the most improvement, because they were not yet considered as a coherent section by any solver.

This process could theoretically be repeated multiple times, each with new partitionings based on additional edge weights. However, the expected improvement decreases with each iteration. Local search operations work on small, local, graph regions. The first repartitioning provides the direct benefit of enabling local search on sections that were previously fragmented across different blocks. As described earlier, these sections are particularly prone to suboptimal node assignments. Additional repartitioning and local search iterations do not provide this advantage. Considering that each repartitioning iteration introduces a large I/O overhead, more than one repartitioning iteration is unlikely to achieve good results.

While this approach does achieve the goal of local searching across previous boundaries,

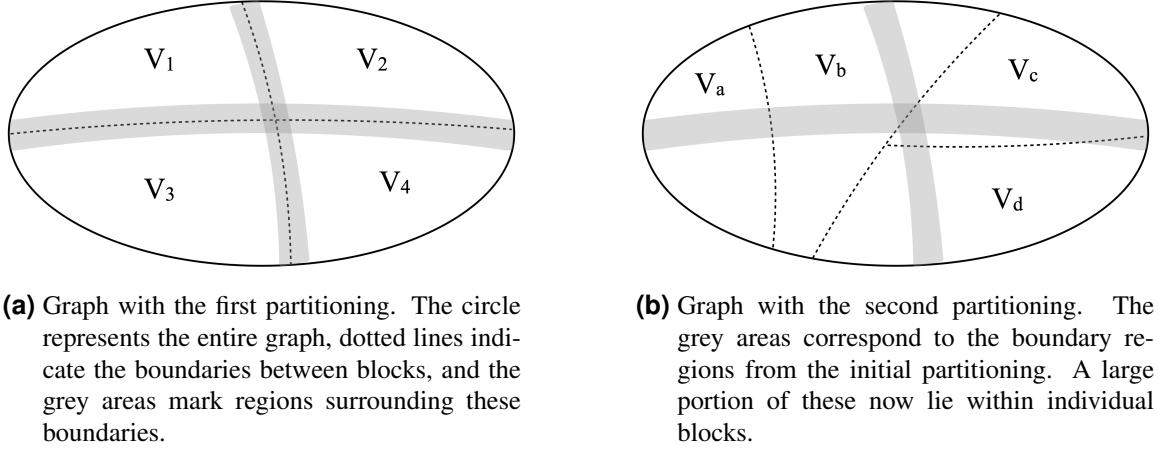


Figure 4.4: Example illustrating how Repartitioning can improve solutions in the boundary regions of the initial partitioning.

the local search is not focused on these areas. Instead we spend considerable time trying to improve solutions in other regions of the graph, that were already solved with a strong solver in the first iteration. This is not entirely ineffective since the assumptions made when creating independent subgraphs affect the solution of the entire graph. However, areas closer to cut edges of the first partitioning are where the most improvement is expected.

4.2.3 Boundary Local Search

A different approach to extending the Base Algorithm with local searching is Boundary Local Search. The main idea is to keep small sections around boundaries in memory after solving a block. When a neighboring block is processed and sections on both sides of the boundary are in memory, we perform local search on this area. Algorithm 5 offers an overview in pseudocode.

In the Boundary Local Search approach, we keep the same overall structure as the Base Algorithm: the graph gets preprocessed and partitioned, then each block gets read in and solved separately. After solving a block as described in Section 4.2, we first collect the sections around the boundaries that we later want to local search on. We start by finding all boundary nodes and grouping them by the neighboring blocks they are adjacent to. For each of these neighboring blocks we start a breadth-first search, initialized with the boundary nodes adjacent to that block, resulting in a section around the boundary. These sections only include nodes from the currently loaded block. We keep these sections in memory until reading their respective neighboring blocks.

When we collect the section on the other side of the boundary and thereby have the entire area around the boundary available in memory, we map the two sections into a subgraph. We make this subgraph independent of the rest of the graph using the same approach as for block-graphs, i.e. by removing all nodes that have an outgoing edge connected to a

Algorithm 5 Structure of Boundary Local Search Algorithm**Input** : Graph G , number of blocks k , block ordering $o : \{1, \dots, k\} \rightarrow \{1, \dots, k\}$ **Output:** Independent Set I

```

 $I, m \leftarrow \text{preprocessAndPartition}(G, k)$ 
for  $o(i)$  with  $i \leftarrow 1$  to  $k$  do
     $I \leftarrow \text{solveBlock}(i, I, \text{false})$ 
    foreach  $j \in N(i)$  do
         $V_{\text{boundary}} \leftarrow \text{getBoundaryNodes}(i, j)$ 
         $\text{section}_{i,j} \leftarrow \text{computeSection}(V_{\text{boundary}})$ 
        /* if block  $j$  already visited */
        if  $o(j) < o(i)$  then
             $G_{\text{local}} \leftarrow \text{merge}(\text{section}_{i,j}, \text{section}_{j,i})$ 
             $G'_{\text{local}} \leftarrow \text{createIndependentSubgraph}(G_{\text{local}})$ 
             $I_{\text{local}} \leftarrow \text{extractIS}(I, G'_{\text{local}})$ 
             $I'_{\text{local}} \leftarrow \text{improveIS}(G'_{\text{local}}, I_{\text{local}})$ 
             $I \leftarrow \text{update}(I, I'_{\text{local}})$ 
             $\text{free}(\text{section}_{i,j}, \text{section}_{j,i})$ 
        else
             $\text{store}(\text{section}_{i,j})$ 
return  $I$ 

```

node in the independent set. Importantly, we remove these nodes only at this point, not while initially storing sections. This is necessary because the independent set can change in the meantime, affecting which nodes should be kept or removed. We then execute local search on the boundary subgraph, initialized with the current independent set solution. Afterwards, we remove both sections from memory. Figure 4.5 visualizes the process of adding and removing sections.

Unlike Repartitioning, this approach requires no additional streams or I/O operations, and local search is executed specifically on targeted sections around boundaries. The main drawback is the memory required to keep the sections in memory until the respective neighboring block is processed. We reduce memory consumption by keeping the first-read side smaller, since it is stored longer, while the other side can be larger as it is immediately processed and removed from memory. Another improvement is reading the blocks in a specific order to reduce the number of sections that are in memory simultaneously.

Scheduling

Scheduling aims to create an ordering in which the blocks are processed such that the number of sections kept in memory simultaneously is minimized. To find this ordering, we examine the quotient graph, where each node represents one block and each edge represents a boundary section that must be stored.

The quality of a node ordering can be evaluated as follows: We initialize a counter-

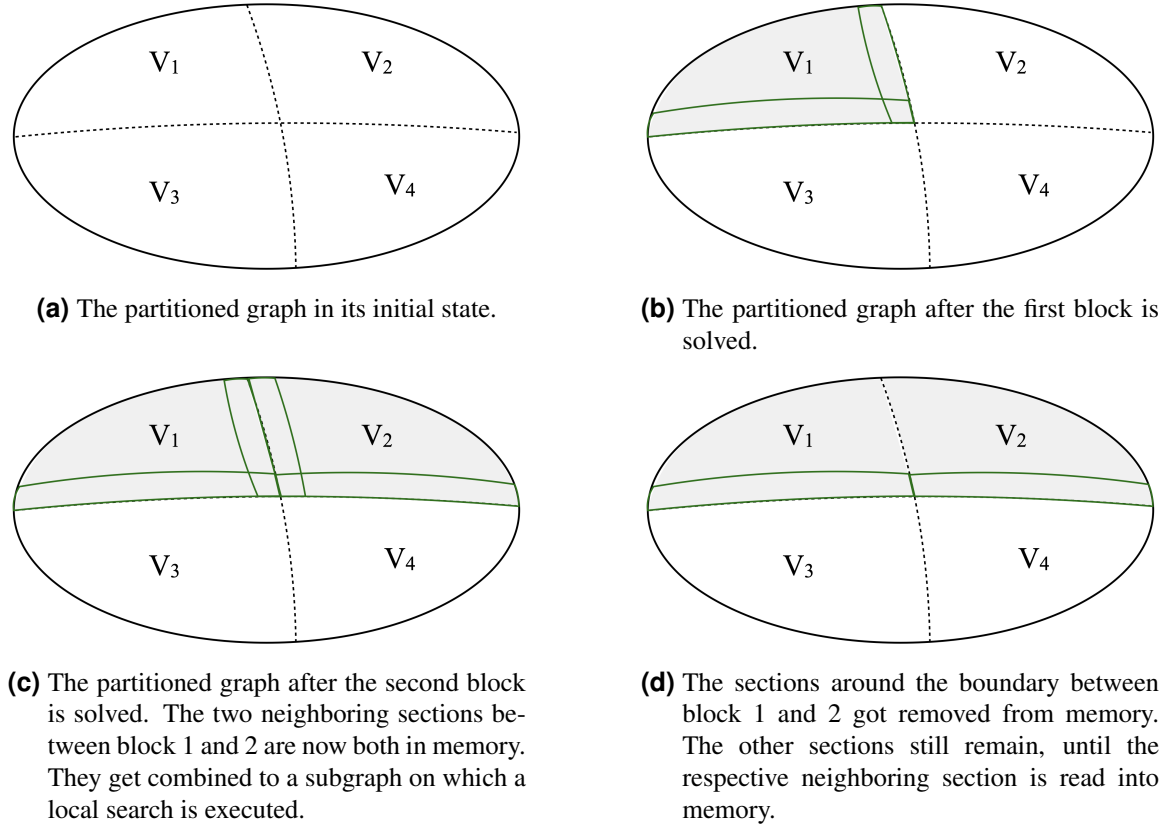


Figure 4.5: This figure visualizes the sections around boundaries that are stored in memory. A block being marked in grey indicates that it is solved. The green outlines show sections that are stored in memory.

variable $x = 0$, that represents the number of sections currently stored in memory. For each node in the ordering, we increase x for each neighbor that has a later position in the ordering and decrease it for each neighbor with an earlier position. The increase corresponds to storing a section when the respective neighboring section is not yet in memory, while the decrease corresponds to removing an already stored section of the respective neighboring block. The relevant metric is the maximum value x reaches during evaluation, as this is the number of sections that must fit in memory simultaneously. The scheduling objective is to find a node ordering that minimizes this metric. We developed several approaches for this.

Global Greedy. The first is a simple greedy strategy that always picks the node with the best immediate effect on the counter x . We maintain a score for each node indicating how it would affect the counter if picked next. Initially, we set each node's score to its degree. When adding a node to the ordering, we decrease each neighbor's score by 2, since picking that neighbor would now remove the connecting edge instead of adding it. We keep nodes with their scores in a min-priority queue and always select the node with the lowest score.

Global Greedy with Heuristics. The second idea extends the greedy idea by heuristi-

cally factoring in the degrees of the neighbors of a node. In the basic greedy approach, a node that has a small degree itself but with very high-degree neighbors would be picked relatively early. The sections added to memory in this step will probably only be removed much later. Generally nodes with small-degree neighbors should be prioritized over nodes whose neighbors all have large degrees. Based on this idea, different heuristic weightings can be applied. For example, instead of initializing the scores of a node u with the number of its neighbors $\deg(u)$, we weight each neighbor based on their respective degree, $\sum_{v \in N(u)} \frac{\deg(v)}{d_{avg}}$ with $d_{avg} = \frac{2 \cdot |E|}{|V|}$. When adding a node u to the ordering, we therefore also need to adapt the neighbor's scores by $2 \cdot \frac{\deg(u)}{d_{avg}}$, instead of the constant 2 from the simple Greedy approach.

Local Crawl. The third approach locally crawls over the graph, similar to a breadth-first search. We start with a random node and always pick the node that would remove the most sections currently stored in memory. We maintain a max-priority queue for all nodes, increasing each neighbor's priority by one when a node is added to the ordering. Compared to the greedy approaches, this achieves more localized movement over the graph.

Local Crawl with Relative Scores. This last approach follows the same local-crawling idea, but uses different priorities. The basic neighborhood crawl only considers how many currently stored sections would be removed when picking a node while disregarding how many sections get added. Handling this like in the greedy approach has the drawback of prioritizing small-degree nodes, even when they are widely distributed across the graph, which can be suboptimal overall. Therefore we score each node by the fraction of neighbors already processed and maintain them in a max-priority queue. This preserves local-crawling properties while also accounting for the degrees of the nodes.

We expect scheduling to be most helpful when the quotient graph is relatively sparse. For dense quotient graphs, node ordering matters less because most sections must be kept in memory regardless of the processing order. The drawback of every approach except basic local crawling is that they require knowledge of the quotient graph. Obtaining that requires additional work, although the overhead can be minimized when integrating this in a stream that is done anyway, e.g. when stream-copying the blocks in separate files.

4.3 Block-Wise Reductions

The second general approach to solving the MIS in a streaming setting is based on reductions. In contrast to the block-wise solving approaches that create independent subgraphs that get solved without considering the rest of the graph, this approach aims at creating a kernel of the full graph by applying globally optimal reductions on each block. This kernel can then, depending on its size, either get solved with a non-streaming MIS solver or using the streaming approach described previously.

This idea is implemented in two variants: a basic algorithm explained in Section [4.3.1](#) and a more complex version, that supports further reductions in Section [4.3.2](#).

Algorithm 6 Structure of Basic Reduction Algorithm

Input : Graph G , number of blocks k , memory threshold t **Output**: Independent Set I

```

 $I, m \leftarrow \text{preprocessAndPartition}(G, k)$ 
for  $i \leftarrow 1$  to  $k$  do
     $G_i \leftarrow \text{loadBlockFromFile}(i)$ 
     $K_i, I_i \leftarrow \text{applyReductions}(G_i)$ 
     $\text{storeKernel}(K_i, i)$ 
     $I \leftarrow I \cup I_i$ 
 $K_{\text{global}} \leftarrow \text{mergeKernels}(K_1, \dots, K_k)$ 
if  $|K_{\text{global}}| < t$  then
     $I_{\text{kernel}} \leftarrow \text{solveInMemory}(K_{\text{global}})$ 
else
     $I_{\text{kernel}} \leftarrow \text{solveStreaming}(K_{\text{global}})$ 
 $I \leftarrow \text{liftSolution}(I, I_{\text{kernel}})$ 
return  $I$ 

```

4.3.1 Basic Approach

Reduction rules are a well-established technique for solving the MIS problem, but their applicability in streaming settings is limited. Most reduction rules require access to local neighborhoods that extend beyond the immediate neighbors of a node. The preprocessing already applies reductions within the constraints of a single stream, but the selection of applicable rules remains small. Therefore, we now want to perform reductions on entire blocks of the graph. The crucial part here is to apply these reductions while considering the context of the neighboring blocks in order to ensure global optimality. Algorithm 6 outlines the overall structure of this approach.

When processing a block, the algorithm first reads all nodes stored in the corresponding binary file. This subgraph then gets extended by creating representative nodes for the boundary nodes from neighboring blocks. This is possible because the format in which the blocks are stored includes the cut edges and thereby provides the necessary context of the surrounding blocks. If multiple internal nodes connect to the same external node, only a single representative is created. External nodes that have already been removed while reducing a neighboring block get omitted. Figure 4.6 shows an example of this.

This process results in an extended subgraph consisting of two node sets: the original in-block nodes read from the file, and external nodes representing the context of the neighboring blocks. We want to apply reductions on this graph with additional constraints: only in-block nodes can be removed, and in-block boundary nodes can only be removed if they do not get added to the independent set. These constraints only determine whether a reduction can get applied, they do not change the behavior of the reductions themselves.

In order to apply the reductions we use an adapted version of ParFastKer [24], which was

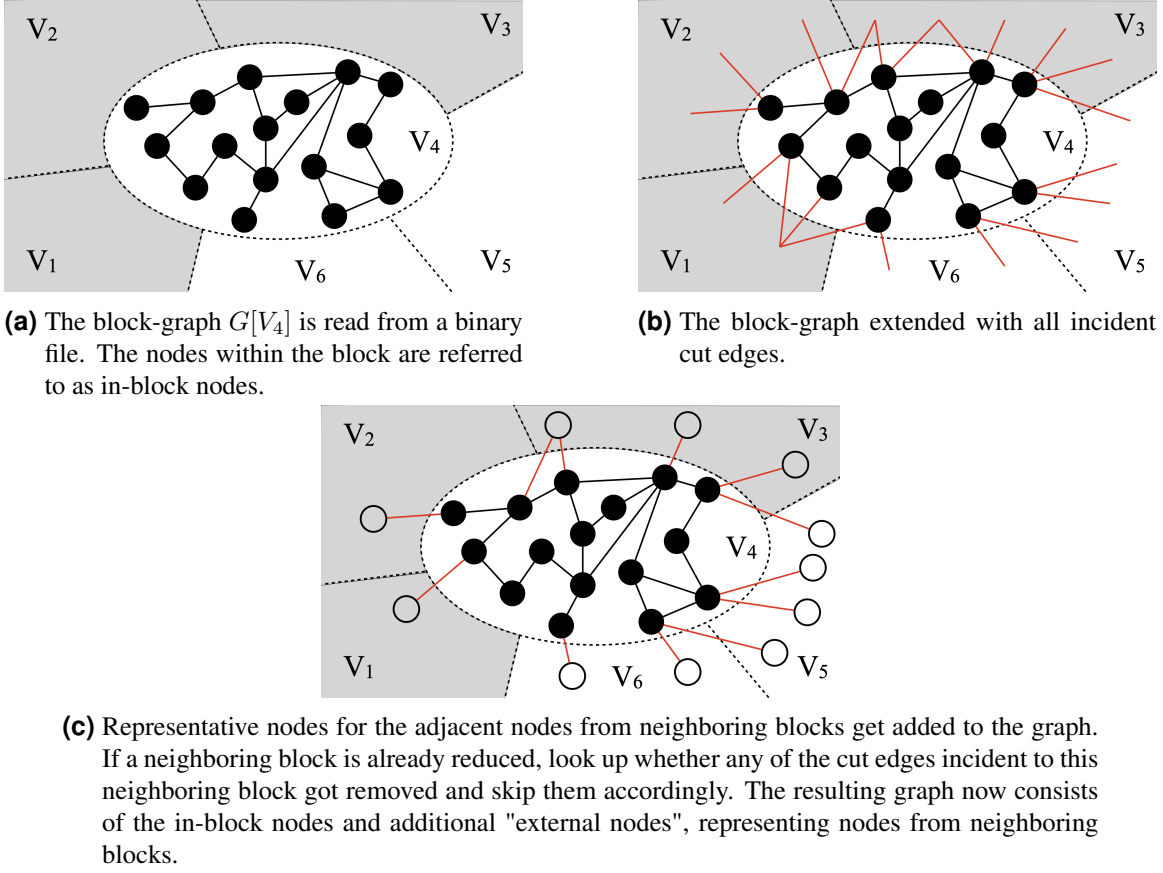


Figure 4.6: The process of adding the context from neighboring blocks to a block-graph. The different blocks are indicated with dotted lines. A section with a grey background represents a block that has already been reduced. A section with a white background represents a block that has not been reduced yet.

originally written for parallel reductions in a shared memory setting. While ParFastKer includes elements that rely on having the full graph in memory, the core approach of applying local reductions that do not cause conflicts between blocks can be reused here. The resulting kernel of a block gets stored in a file. After processing all blocks, a global kernel is created based on all partial kernels.

This global kernel is then solved using a different independent set solver, depending on its size. Small global kernels can be solved with an in-memory algorithm. If the remaining kernel is large, we need to apply a streaming algorithm like one of the block-wise solving approaches.

The implementation of this approach builds on the assumption that the applied reduction rules do not add new edges or alter existing ones. Edges can be removed but under no circumstances redirected to a different node. Moreover, using this set of reduction rules, edges are only removed when an incident node is removed. This makes it straightforward

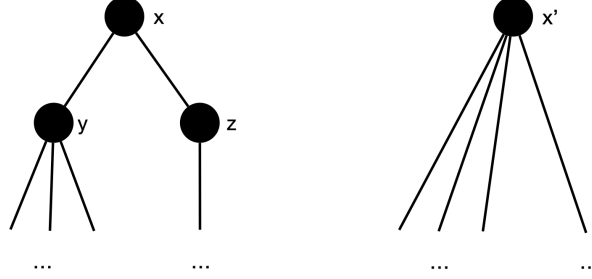


Figure 4.7: Visualization of a V-shape reduction. The nodes x, y and z get folded to a single node x' .

and memory-efficient to track whether an edge still exists, by just looking up which nodes have been removed.

4.3.2 Vertex Fold Extension

The base approach of the reductions algorithm builds on the assumption of edges not being redirected. Removing this constraint allows the usage of further reduction rules, which can further reduce kernel sizes while remaining optimality. However, this requires substantial modifications to the algorithm, leading to a new variant of the algorithm that we refer to as Reductions+VF in the following.

Vertex Folding. An example for a reduction that changes existing edges is the *V-shape* reduction [19, 27]. Let x be a node with degree two and its two neighbors y and z are non-adjacent. Then, x, y and z can be replaced with a new node x' that is adjacent to $N(y) \setminus x$ and $N(z) \setminus x$. This process of combining vertices is called *folding*. An example for this reduction is visualized in Figure 4.7.

After the remaining subgraph is solved, the solution gets lifted as follows: if x' is in the independent set of the subgraph, y and z get added to the solution. If x' is not in the independent set, x gets added to the solution. In practical implementations of this reduction, usually no new node gets added to the graph. Nodes y and z get removed and the neighborhood of x gets changed such that it resembles x' . This involves changing edges that were previously incident to y and z to now be incident to x . These edges violate assumptions we rely on, requiring extensive changes to the implementation. Previously, we relied on the fact that, when reducing a block, the set adjacent cut edges after applying reductions is a subset of the set of adjacent cut edges before applying the reductions. With vertex-fold reductions, this assumption is not met anymore.

Algorithm adaptation. Allowing edge redirections requires two main alterations of our

algorithm. First, the handling of cut edges must be adapted: both how context from neighboring blocks is incorporated during block processing and how partial kernels are stored, need to account for the possibility of edges that are not present in the original graph. Second, after solving the kernel, the full solution needs to be lifted by reversing the transformations introduced by the reductions. The following paragraphs detail these changes.

We start by explaining the change in the handling of the cut edges. Due to the possibility of changing cut edges, we now need to explicitly track the current set of cut edges. In the following, we consider neighboring blocks distinguished in two groups: blocks that already appeared in the block order and therefore have been reduced, which we call reduced neighbors and blocks that have not been reduced yet, which we call non-reduced neighbors.

Whenever we finish reducing a block, we keep all cut edges in memory, that are incident to non-reduced neighbors. When loading a new block in memory and adding the external context from the neighboring blocks, we handle the cut edges towards reduced neighbors and non-reduced neighbors differently. The non-reduced neighbors get handled in the same way as before: we add each target of the cut edges as an external node. However, cut edges connected to a reduced neighbor might have changed. Therefore we skip all of these cut edges written in the block file and instead add all cut edges from our in-memory storage that belong to this block.

Figure 4.8 visualizes this process. Keeping these cut edges in memory can create issues for large graphs. In principle, this is the same issue that occurred with the Boundary Local Search approach. Even though to a lesser extend, because we only need to store the cut edges and not a section around them as well. Computing a scheduling is again helpful to reduce the peak amount of memory used.

Furthermore, there are complications when storing the kernel of a block after it has been reduced. When writing a block-kernel to file, we need to include the cut edges. However, when writing a kernel of a block, usually not all neighboring blocks have been reduced yet. Cut edges towards non-reduced blocks might change, once that neighboring block gets reduced. Without the vertex folding reductions, this is not an issue. The set of cut edges remaining in the final kernel is a subset of the set of cut edges that is present when writing a block-kernel. Therefore, we can add all currently remaining cut edges to the block-kernel file. When combining the block-kernels to a global-kernel in the end, we can check for each cut edge present in the block-kernel files, whether they have been removed or not. In the scenario where edges change, we do not have a way of looking up the current state of each cut edge. Therefore we can only write those cut edges in our block-files that are not going to change later on. In total, this means, every cut edge is written in one block-kernel file while it is missing in another. Adding the opposite direction of the edge while combining the block-kernels to a global-kernel is not trivially possible for large global-kernels that do not fit in memory. Therefore, we need to store the cut edges in some separate data-structure. This can either be an in-memory data structure or, in case we assume that not all cut edges of the kernel fit in memory, a file where we separately store the cut edges while writing the block-kernels.

The second important addition due to the advanced reduction rules is the solution lifting

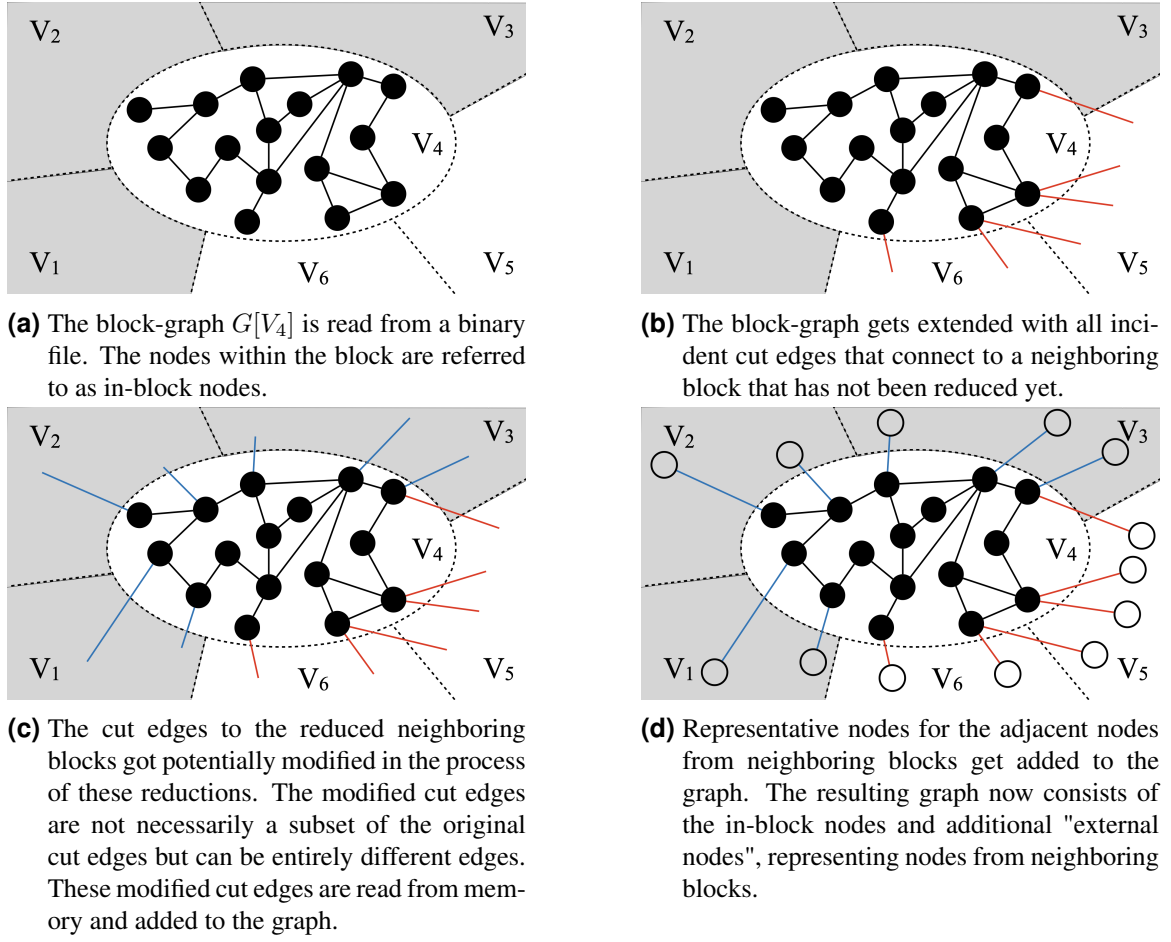


Figure 4.8: The process of adding the context from neighboring blocks to a block-graph, considering the possibility that the cut edges got modified. The different blocks are indicated with dotted lines. A section with a grey background represents a block that has already been reduced. A section with a white background represents a block that has not been reduced yet.

after solving the global-kernel. Previously, lifting was not necessary as reductions just removed nodes and edges from the graph and added nodes to the independent set, all of which can be done immediately. With vertex folding, reductions may replace multiple nodes with a single representative, and the lifting decision depends on whether this representative appears in the kernel's independent set. For example, a fold might replace three nodes a, b, c with one representative such that if the representative is in the kernel solution, nodes a and b are added to the final independent set, otherwise node c is added. To support this, the algorithm must store the inversion rules for each fold operation until after an independent set is computed on the global kernel. Then, these inversion rules are applied in reverse order of their creation to reconstruct the solution for the original graph.

Experimental Evaluation

This section evaluates the algorithms presented in this thesis. We begin by describing the experimental setup and datasets, then analyze individual algorithm components to determine optimal configurations, and finally evaluate the complete algorithms in terms of running time, solution quality, and memory consumption.

5.1 Setup

All algorithms are implemented in C++17 and compiled with g++-14 using full optimization (-O3 flag). Experiments are run on a single core of a machine with a sixteen-core Intel Xeon Silver 4216 processor at 2.1 GHz, 100 GB of main memory, 16 MB of L2 cache and 22 MB of L3 cache running Ubuntu 20.04.1.

The graph instances used in our evaluation are shown in Table 5.1. Graphs are stored in METIS format with edges sorted per node. The dataset includes social networks, web graphs, mesh graphs, and road networks collected from various sources [4, 14, 29]. We categorize the graphs into three sets for evaluation purposes: a tuning set primarily for parameter studies, a test set of large graphs for algorithm comparisons, and a set of huge graphs for large-scale experiments.

We compare our algorithms Base Algorithm, Repartitioning, Boundary Local Search, Reductions and Reductions+VF against state-of-the-art in-memory and semi-streaming algorithms for the maximum independent set problem. The in-memory solver ReduMIS [26] is used for comparison on the Test Set, while the semi-streaming vertex-swap algorithms from Liu et al. [30] are included for both the Test Set and Huge Graphs.

For the visualization of result quality, memory usage and running times of the different algorithms, we use performance profiles [11]. Performance profiles compare multiple algorithms by evaluating their results on a per-instance basis. For each test instance, the performance of an algorithm is expressed as a ratio or performance factor τ relative to the best result obtained on that instance. The x-axis of the plot shows this factor τ , while

Table 5.1: Graphs for experiments.

Graph	n	m	Type	Graph	n	m	Type
Tuning Set							
buddha-sorted	1 087 716	1 631 574	Mesh	dragonsub-sorted	600 000	900 000	Mesh
ecat-sorted	684 496	1 026 744	Mesh	turtle-sorted	267 534	401 178	Mesh
bay-sorted	321 270	800 172	Road	col-sorted	435 666	1 057 066	Road
fla-sorted	1 070 376	2 712 798	Road	ny-sorted	264 346	733 846	Road
amazon-2008-sorted	735 323	3 523 472	Social	as-skitter-sorted	554 930	5 797 663	Social
citationCiteseer-sorted	268 495	1 156 647	Social	cnr-2000-sorted	325 557	2 738 969	Social
coPapersCiteseer-sorted	434 102	16 036 720	Social	enron-sorted	69 244	254 449	social
loc-gowalla_edges-sorted	196 591	950 327	Social	web-Google-sorted	356 648	2 093 324	Social
roadNet-PA	1 088 092	1 541 898	Road	roadNet-TX	1 379 917	1 921 660	Road
Test Set							
Bump_2911	2 852 430	62 409 240	Mesh	Flan_1565	1 564 794	57 920 625	Mesh
FullChip	2 986 999	11 817 567	Circuit	ca-hollywood-2009	1 069 126	56 306 653	Road
cit-Patents	3 774 768	16 518 947	Citations	com-lj	3 997 962	34 681 189	Social
com-orkut	3 072 441	117 185 083	Social	com-youtube	1 134 890	2 987 624	Social
in-2004	1 382 908	13 591 473	Web	soc-lastfm	1 191 805	4 519 330	Social
Huge Graphs							
arabic-2005	22 744 080	553 903 073	Web	com-friendster	65 608 366	1 806 067 135	Social
it-2004-sorted	41 291 594	1 027 474 947	Web	nlpkt240	27 993 600	373 239 376	Matrix
orkut	3 072 441	117 185 082	Social	rgg_n26	67 108 864	574 553 645	Artificial
RHG-1b	100 000 000	1 000 913 106	Artificial	RHG-2b	100 000 000	1 999 544 833	Artificial
sk-2005-sorted	50 636 154	1 810 063 330	Web	twitter-2010	41 652 230	1 202 513 046	Social
uk-2007-05	105 896 555	3 301 876 564	Web	webbase-2001	118 142 155	854 809 761	Web

the y-axis represents the percentage of instances where an algorithm achieves performance within τ of the best result. Depending on whether the best value of a target metric is the largest or the smallest, the x-axis is decreasing or increasing respectively. Each algorithm appears as a separate line on the plot, producing a non-decreasing curve. Algorithms for which the respective lines rise quickly and reach higher y-values perform better overall, as they achieve near-best results on a larger fraction of instances.

5.2 Parameter Study

Before evaluating the complete algorithms, we analyze individual components to gain insights into their behavior and determine optimal configurations. The main elements studied are preprocessing, edge estimation methods, in-memory MIS algorithms used to solve block-graphs, also referred to as "block solver", scheduling strategies, and the impact of block sizes on solution quality and memory consumption.

5.2.1 Preprocessing

Preprocessing is technically optional, but skipping it can negatively impact partitioning quality as explained in Section [4.1.1](#). We evaluate the reduction achieved by preprocessing across all graphs.

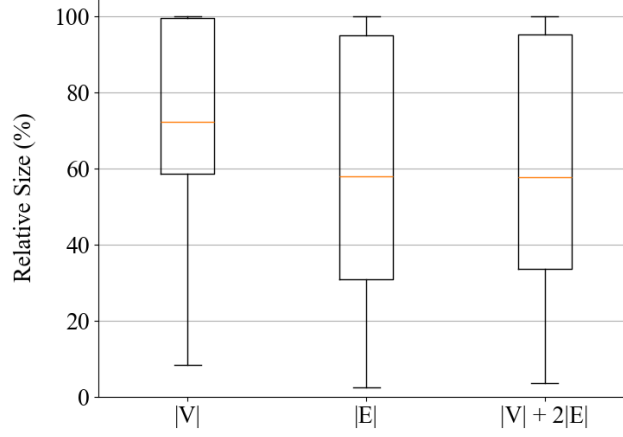


Figure 5.1: Remaining size of the graphs after preprocessing. The boxplots illustrate the fraction of nodes and edges that remain after removal across different graphs. 100% represent the size of the original graph.

Figure 5.1 shows the relative size of graphs after preprocessing. The number of nodes is reduced on average by 27.5%, and the number of edges by 42.44%. Since the size of a graph is primarily determined by edge count, the overall graph size is reduced on average by 41.45%. The standard deviations are large: 26.56% for nodes, 33.50% for edges, and 32.81% overall. This indicates that the effectiveness of the preprocessing varies considerably by graph type. Some graphs, especially mesh-type instances, are not reduced at all. Where reductions are successful, relatively more edges than nodes are removed. This occurs partly because very high-degree nodes are explicitly removed, and partly because some degree-0 and degree-1 reductions were missed due to the limited scope in which the reductions can get applied.

Despite these limitations, preprocessing is worthwhile. The cost of one additional stream is offset by reducing the graph size by over 40% on average, meaning all subsequent algorithm steps operate on substantially smaller graphs.

5.2.2 Edge Estimation Method

We compare the three edge estimation methods introduced in Section 4.1.1: the Upper Bound $\frac{m_a}{2}$, Node-Based estimation $m_a \frac{|A|-1}{|A|-1+|B|}$, and Edge-Based estimation $\frac{m_a^2}{2 \cdot (m_a + m_b)}$.

The methods are evaluated on three metrics: estimation accuracy relative to the actual edge count and edge-cut and balance of partitionings computed based on the respective estimated edge counts. Experiments are run on all graphs with $k \in \{8, 16, 32, 64, 128, 256, 512\}$. HeiStream is configured to run with a single pass and an imbalance of 3%.

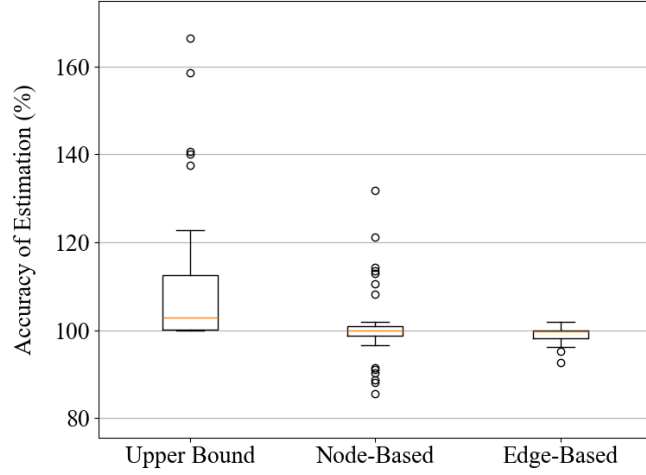


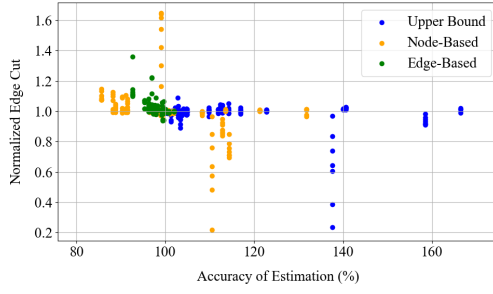
Figure 5.2: Estimation accuracy for the three methods. The y-axis shows estimated edge count divided by actual edge count. One outlier-instance is excluded for each estimation method.

In the following evaluation, one instance is excluded. For this graph, the reduced graph after preprocessing is extremely small compared to the original graph. All three estimation methods produce highly inaccurate estimates for this instance: while the actual edge count is 9 795, the Upper Bound lies at 337 145, the Node-Based estimation computes 182 594 and the Edge-Based estimation 131 108 for the edge count. We exclude this instance because its extreme reduction during preprocessing leads to negligible memory usage and makes a balanced partitioning irrelevant for this instance. Furthermore, the following evaluation is mostly based on the relative comparison between estimated and actual edge count. While the absolute inaccuracy of the estimations for this instance are not that significant considering the size of the test instances, the relative error is extremely high, thereby heavily skewing the results and figures.

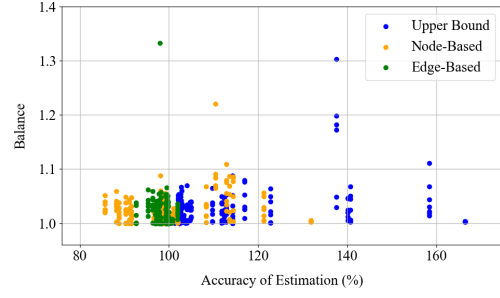
Figure 5.2 compares estimation accuracy. Node-Based and Edge-Based estimations are significantly more accurate than the Upper Bound estimation method. Edge-Based estimation is most precise, achieving a mean absolute error of 1.08%, compared to 4.99% for Node-Based and 14.68% for the Upper Bound.

However, estimation accuracy alone does not determine practical performance. The purpose of the estimations is to use them as input to HeiStream. Therefore, we evaluate the resulting partitionings in terms of edge-cut and balance, the results of which are shown in Figure 5.3. The edge-cuts of the partitionings using the estimations are normalized by the edge-cut of the partitioning using the actual edge count.

The edge-cut is primarily affected by whether the estimation underestimates or overestimates the true value. Underestimation worsens the edge-cut, as valid block assignments are rejected due to apparent balance violations. Even small underestimations have this ef-



(a) Edge-cut of partitionings. The x-axis shows the estimated edge count divided by the actual edge count. The y-axis shows the edge-cut normalized by the edge-cut of the partitioning computed with the actual edge count.



(b) Balance of partitionings. The x-axis shows the estimated edge count divided by actual edge count. The y-axis shows the balance score.

Figure 5.3: Impact of edge estimation methods on partitioning quality.

fect, as visible in Figure 5.3a. Overestimation has less of an impact. The results are similar to using the precise edge count with relaxed balance constraints, often producing better edge-cuts than the baseline. The average normalized edge-cuts are 0.9798 for the Upper Bound estimation method, 1.0006 for Node-Based estimation, and 1.0136 for Edge-Based estimation.

Balance is affected more continuously, not showing a clear cut-off between underestimation and overestimation. Larger ratios of estimated to actual edge count correlate with larger imbalances with a correlation factor of 0.23. Underestimation creates balance constraints that cannot be fulfilled, so relative estimations below 100% do not yield better balance. Overestimation naturally leads to larger imbalances. However, in most cases results are more balanced than strictly enforced: the average balance is 1.0302 for the Upper Bound estimation method, 1.0256 for Node-Based estimation, and 1.0197 for Edge-Based estimation.

The evaluation shows that avoiding underestimation has to be prioritized over achieving more accurate estimation. Even small underestimations have a large impact on the edge-cut, which directly affects subsequent algorithm performance. Therefore, we use the Upper Bound $\frac{m_a}{2}$. Although the resulting partitionings are not perfectly balanced, block sizes cannot exceed those of a balanced partitioning on the original unreduced graph, which satisfies our minimum requirements. The evaluation also shows that in practice, balance remains within reasonable bounds.

5.2.3 Block solver

We compare three in-memory MIS algorithms to solve block-graphs in the context of the Base Algorithm: CHILS, ReduMIS, and the combination of the DataReduction library and CHILS, which we refer to as "R+CHILS" in the following. The evaluation runs the Base

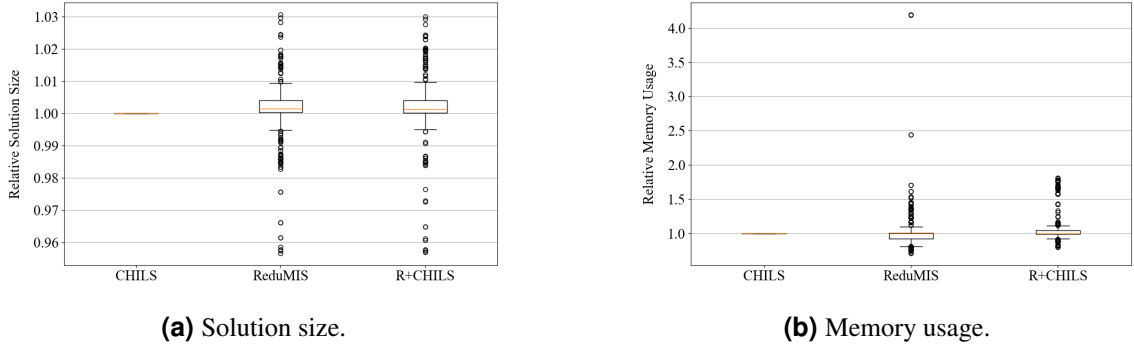


Figure 5.4: Comparison of algorithms solving the MIS on block-graphs.

Algorithm with each solver on each graph from the Tuning Set and Test Set using three time configurations: fast, medium, and slow.

By evaluating the algorithms in the context of the Base Algorithm, they are tested on blocks resulting from preprocessed and partitioned graphs. This is relevant, because we aim at comparing the algorithms in the context of our use-case, not in a general setting. Preprocessing already applies reductions, thereby potentially lowering the effectiveness of further reduction-based algorithms. Furthermore, block-graphs resulting from a streaming partitioning can have different properties than general graphs. For example, a block of a partitioning computed in the streaming setting is rarely one large connected component. Instead, it oftentimes consists of multiple connected components, each of which is disconnected from the others.

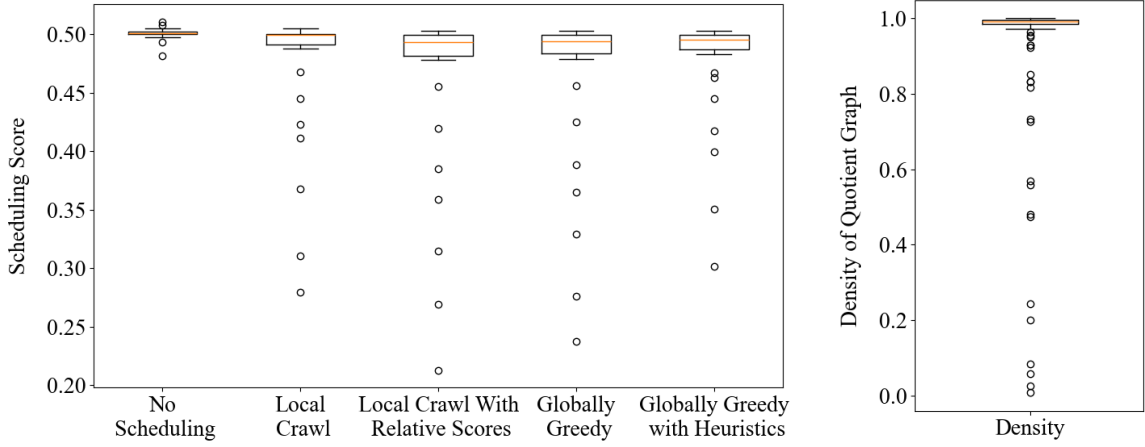
Figure 5.4 shows a comparison of the algorithms. All result quality and memory measurements are normalized using the results of CHILS as the baseline. Results are consistent across the three time configurations, so we present them aggregated instead of in separate plots.

All three algorithms produce similar results. ReduMIS and R+CHILS achieve slightly better solution quality than pure CHILS, with ReduMIS producing results 0.08% better on average and R+CHILS 0.17% better. Memory usage is also relatively similar: ReduMIS uses 2.52% more memory on average, while R+CHILS uses 6.79% more. ReduMIS exhibits larger outliers, requiring over 300% more memory than CHILS on two instances.

Based on these results, we select R+CHILS due to its good average performance, acceptable memory consumption, and the lack of extreme outliers in terms of memory.

5.2.4 Scheduling

We compare the scheduling approaches introduced in Section 4.2.3. The schedulings are computed on graphs that have been preprocessed and partitioned with HeiStream. The tests are run on the groups of graphs categorized as Test Set and Huge Graphs in Table 5.1 with $k \in \{64, 128, 256, 512\}$. For the graphs in the category Huge Graphs, we run addi-



(a) Comparison of the different scheduling approaches. A lower score relates to a better scheduling. This figure excludes all instances where the density of the respective quotient graph is 1, because all schedulings are equivalent on complete graphs.

(b) Density of Quotient Graphs.

Figure 5.5: Evaluation of key scheduling metrics.

tional tests with $k = 1024$ and $k = 2048$.

The score of a scheduling is determined as explained in Section 4.2.3 and normalized by the number of edges in the quotient graph.

We compare four approaches: Local Crawl, Local Crawl with Relative Scoring, Global Greedy, and Global Greedy with Heuristics. We compare these results in relation to No Scheduling, which refers to the default block order $1, \dots, k$. The results are visualized in Figure 5.5a.

All scheduling approaches outperform the baseline of No Scheduling. Local Crawl with Relative Scoring performs best with an average score of 0.4681, followed closely by Global Greedy at 0.4701. Local Crawl and Global Greedy with Heuristics score slightly worse at 0.4786 and 0.4804, respectively.

However, all approaches are extremely similar and relatively close to the baseline. This is primarily due to the quotient graphs of the partitionings being very dense. Figure 5.5b visualizes the distribution of the density of these quotient graphs in a boxplot. The average density is 91.76%, with 83.6% of instances having a density of at least 95%. This occurs due to limitations of partitioning in a streaming setting. HeiStream lacks global context while assigning nodes to blocks, leading to fragmented blocks. Figure 9 of the HeiStream paper [12] visualizes this fragmentation and thereby provides an intuition why streaming partitioning produces dense quotient graphs.

When the quotient graph is very dense, scheduling has minimal impact. In the extreme case of a complete graph with density 1, all orderings are equivalent. Differences between

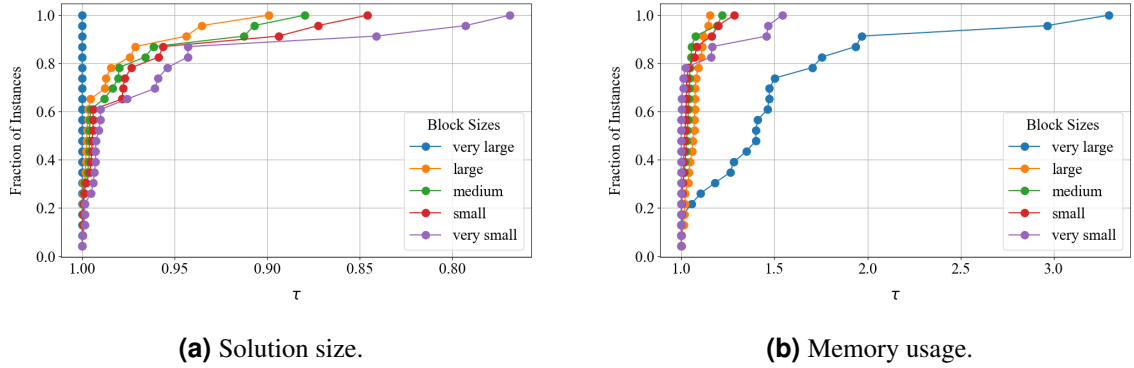


Figure 5.6: Performance profiles of key metrics of the Base Algorithm compared over multiple runs with different block sizes.

scheduling strategies only emerge for a few graphs or very large k values, which produce huge quotient graphs that are impractical in application.

Based on these results, we select Local Crawl. While it does not achieve the best scores, it performs very close to the other approaches and is the only method that does not require computing the quotient graph beforehand. Given the minimal differences, the computational savings are prioritized.

Dense quotient graphs are generally an issue for our algorithms that are employing scheduling. This is primarily a problem for the Boundary Local Search, but also for the Reductions+VF algorithm, although the scheduling is less relevant for the latter. The original aim of scheduling is to optimize the order in which blocks are handled and thereby keep the amount of information stored in memory low. For very dense quotient graphs, the block order becomes irrelevant, as the amount of memory required is high in any case. This becomes increasingly problematic for large k , as the number of edges in the quotient graph grows quadratically in relation to k for quotient graphs with density close to 1.

5.2.5 Block Size

Since all of our algorithms are partitioning-based, the number of blocks k is a key parameter. Memory consumption and solution quality are both significantly influenced by the thereby resulting block size.

We evaluate each algorithm across five block size categories: very large, large, medium, small, and very small. For large graphs taken from the Test Set, these correspond to $k \in \{8, 32, 64, 128, 512\}$. For Huge Graphs, these correspond to $k \in \{32, 128, 256, 512, 2048\}$. Note that a larger k relates to a larger number of blocks which makes each individual block smaller.

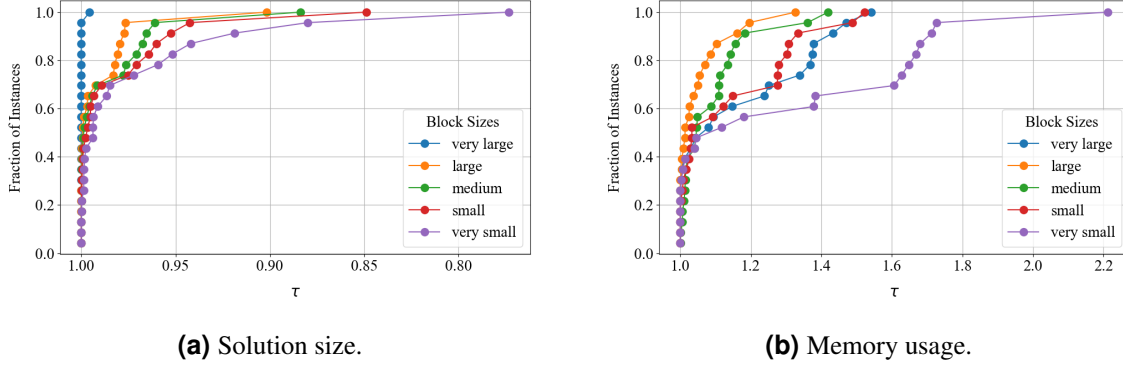


Figure 5.7: Performance profiles of key metrics of the Repartitioning algorithm compared over multiple runs with different block sizes.

Base Algorithm

As shown in Figure 5.6, the Base Algorithm shows a clear relationship between block size and solution quality: larger blocks consistently produce larger independent sets. The largest block size achieves the best result on every instance, and each block size outperforms all smaller block sizes across the majority of instances. While 80% of instances remain within 95% of the best solution across all block sizes, a few instances deteriorate significantly with smaller blocks. Two instances with very small blocks find solutions below 80% of the optimal size.

This occurs because smaller blocks create more and larger boundaries. The Base Algorithm’s approach of making subgraphs independent through greedy node and edge removals around boundaries becomes increasingly problematic with larger boundaries.

Memory usage generally decreases with smaller blocks. The more nodes and edges a block has, the larger is its size in memory. However, all block sizes except very large blocks produce relatively similar memory consumption. For some instances, very small blocks actually use more memory than large blocks. In these cases, blocks are small enough that they no longer dominate memory consumption. Instead, memory is determined by global $O(n)$ vectors that remain constant across block sizes and by HeiStream’s memory consumption, which increases for large k . Therefore, the Base Algorithm achieves lowest memory usage with roughly medium-sized blocks.

Repartitioning

Figure 5.7 shows the solution size and memory usage of the Base Algorithm across different block sizes. The Repartitioning algorithm shows a similar trend of declining solution quality with smaller blocks, but to a lesser extent than the Base Algorithm. Each block size achieves results within 1% of the best solution for 60% of instances. A few outliers with very low solution quality remain for smaller blocks. The effect of more and larger

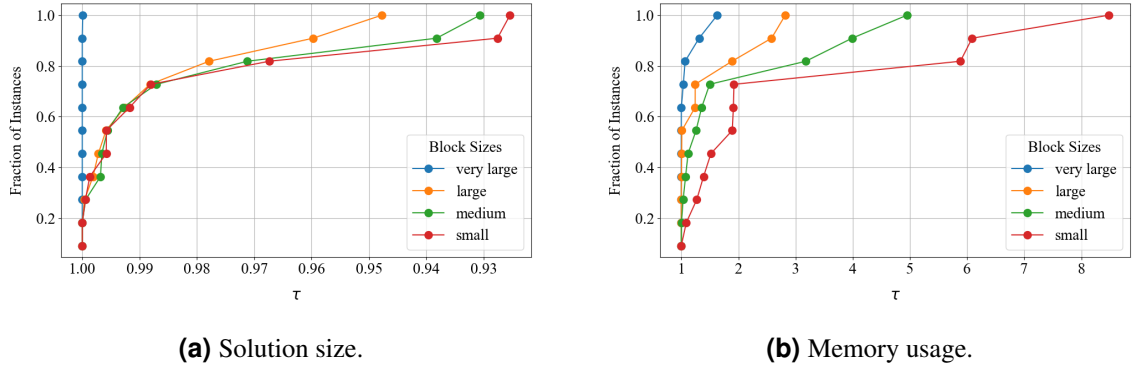


Figure 5.8: Performance profiles of key metrics of the Boundary Local Search algorithm compared over multiple runs with different block sizes.

boundaries on solution quality persists, but is partially mitigated through the further local search introduced by the Repartitioning algorithm.

Memory usage for small and very small blocks is notably higher than for larger blocks. This is again due to HeiStream requiring more memory with very large k . When blocks become small enough, their impact on memory consumption becomes less relevant, and peak memory usage is mostly dominated by HeiStream. This effect is more amplified compared to the Base Algorithm because the second partitioning uses edge weights, which requires additional memory.

Boundary Local Search

The comparison across different block sizes is more limited for Boundary Local Search. As detailed in the scheduling evaluation (Section 5.2.4), extremely dense quotient graphs make Boundary Local Search infeasible for larger k , especially on large graphs. Therefore, this evaluation includes only small graphs and excludes the very small block size category.

Figure 5.8 shows that solution quality decreases with smaller blocks. Since the algorithm is built on the Base Algorithm, it naturally inherits the trend of worse results with more blocks and larger boundaries. Boundary Local Search compensates for some of these disadvantages, but the trend persists. However, results remain relatively similar across block sizes. Even the worst result achieves a ratio of 0.925 compared to the best.

Memory usage shows the opposite relationship compared to previous algorithms. Smaller and more numerous blocks lead to higher memory consumption, as more boundary sections must be kept in memory simultaneously. For some graphs, particularly those heavily reduced during preprocessing, this effect is modest. For other graphs, the relationship is extreme due to dense quotient graphs. For quotient graphs with a density close to 1, the number of sections stored in memory grows quadratically with k , making the algorithm impractical for large numbers of blocks, especially on large graphs.

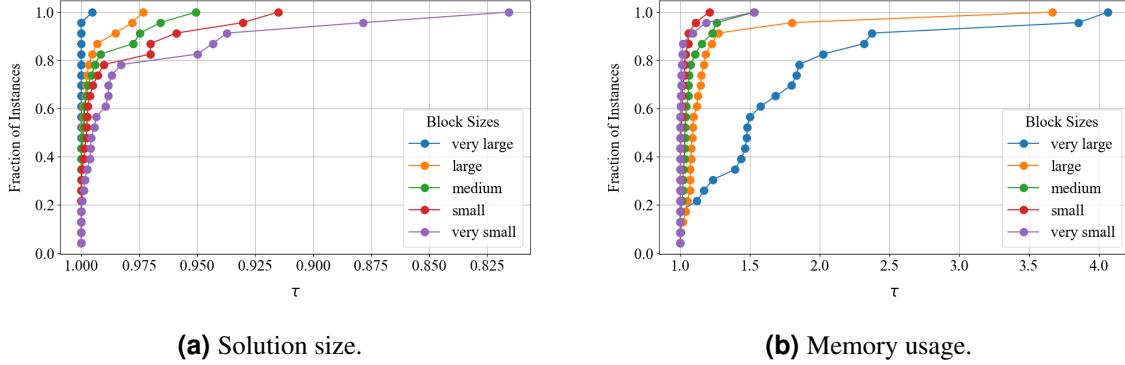


Figure 5.9: Performance profiles of key metrics of the Reductions algorithm compared over multiple runs with different block sizes.

Reductions

Figure 5.9 shows that the solution quality of the Reductions algorithm still depends on the block size but less significantly than the previous algorithms. 80% of instances remain within 97.5% of the best solution across all block sizes. Except for very small blocks, all configurations perform well on all instances. Even medium block sizes achieve at least 95% solution quality on every instance.

In contrast to the Base Algorithm and Repartitioning, the Reductions algorithm does not create entirely independent subgraphs, which was the primary cause of quality degradation with increasing k . More and larger boundaries do impact the effectiveness of reductions, as reductions rules cannot be applied across boundaries, resulting in larger kernels. Finding an independent set on a larger kernel produces worse results compared to applying more exact reductions and solving a smaller kernel. Additionally, large kernels are solved using the Base Algorithm, which inherits that algorithm’s dependency on block size.

Memory usage shows a clear trend of decreasing consumption with smaller blocks. Except for very large blocks and a few outliers, all runs have at most a factor of 1.3 difference in memory usage. The block size directly impacts memory consumption, as larger blocks require more memory to load. However, there is also a dependency on the boundary size. The reductions algorithm extends the in-memory block representation with context from boundaries. Smaller blocks create larger boundaries, which partially offsets the memory savings from smaller blocks.

Reductions + VF

Figure 5.10 shows that the impact of block size on solution quality is similar to the reductions algorithm without vertex folding. Solution size decreases with smaller blocks, but this effect is minimal for most instances. A few outliers show significant block size dependency. These are instances where reductions are less effective and large kernels remain.

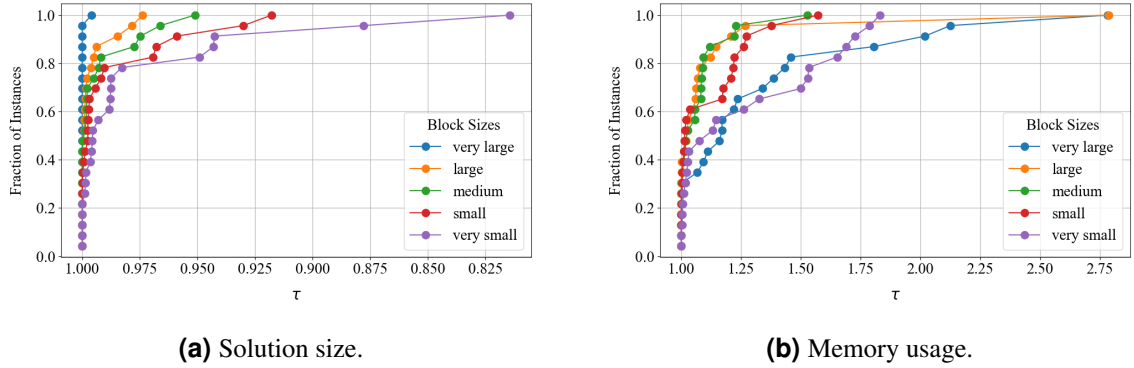


Figure 5.10: Performance profiles of key metrics of the Reductions+VF algorithm compared over multiple runs with different block sizes.

For these cases, total solution size is primarily determined by the solution computed on the kernel using the Base Algorithm, which has stronger dependency on k .

The impact on memory usage, however, differs from the reductions algorithm without vertex folding. The basic reductions algorithm shows decreasing memory usage with smaller blocks. While this trend remains for block sizes medium to very large, the vertex folding variant shows increased memory usage for small and very small blocks on a large portion of instances. Medium and large blocks are most memory efficient across the majority of instances.

This occurs because the algorithm with vertex folding requires to partially store cut edges in memory, as detailed in Section 4.3.2. More blocks create larger cuts and thereby increase memory overhead.

5.3 Algorithm Comparison

After examining individual components and properties of the algorithms, we now compare our algorithms against each other and against the state of the art. The evaluation is separated into two parts based on graph size.

First, we evaluate on graphs from the "Test Set" category (Table 5.1). These graphs are small enough that ReduMIS can still be used, which allows us to compare our streaming algorithms with an in-memory algorithm. We also compare against results from Liu et al.'s algorithms: 1-* swap and 2-* swap. Note that the original paper uses the names 1-k swap and 2-k swap which we denote differently here in order to avoid confusion with the number of blocks k . For each instance, we run ReduMIS with a time limit of 3600 seconds and configure our algorithms to use large block sizes with $k = 8$. We evaluate three metrics: solution size, memory usage, and running time.

Second, we evaluate on graphs from the "Huge Graphs" category. ReduMIS is not viable for these instances, so we compare only our algorithms against each other and against Liu

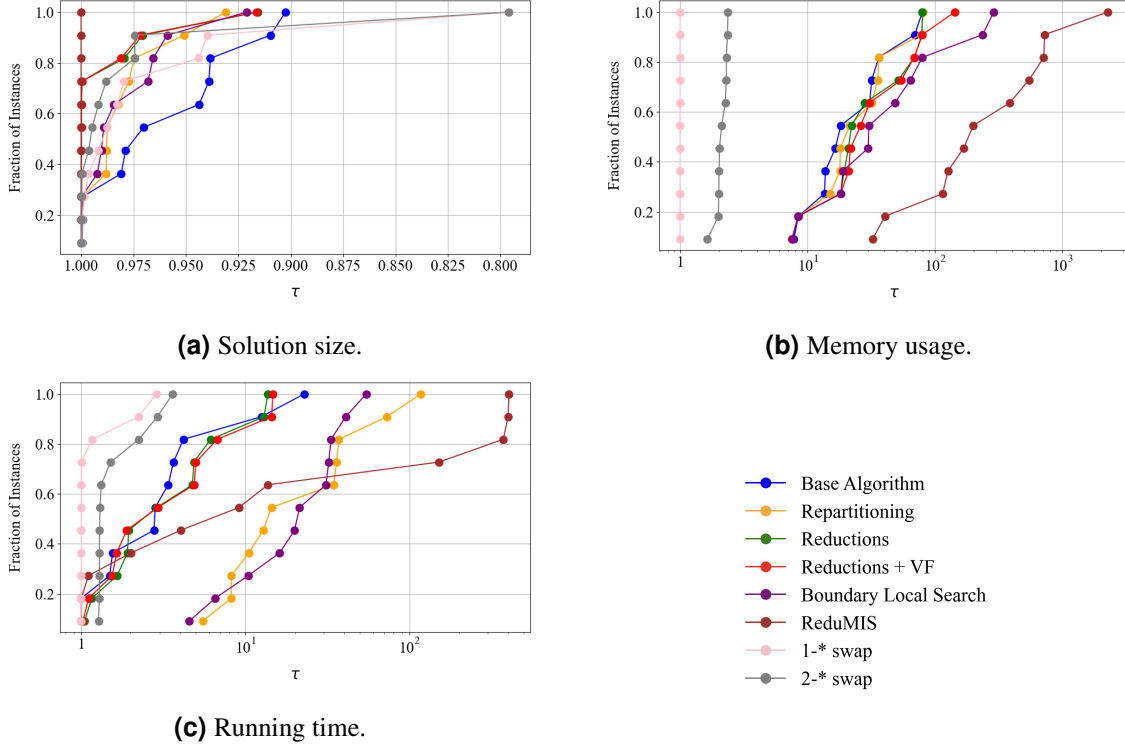


Figure 5.11: Comparison of algorithm performance on Test Set graphs. For each metric solution size, memory usage and running time, the results of the algorithms are displayed in performance profiles.

et al.’s algorithms. For these graphs, we run our algorithms twice with different block sizes: $k = 32$ (large blocks) and $k = 512$ (small blocks). Again, we evaluate solution size, memory usage, and running time.

Test Set Graphs

The Test Set contains graphs with millions of nodes and tens of millions of edges, ranging up to 117 million edges for the largest instance.

Solution size. Figure 5.11a shows a performance profile of the solution quality of the different algorithms. As expected, ReduMIS establishes the baseline. Streaming algorithms are designed to compensate for streaming constraints and allow the usage for larger instances but provide no inherent advantage over in-memory algorithms in terms of solution quality.

The Base Algorithm performs comparatively worst, achieving results close to ReduMIS on only three instances. However, every instance reaches at least 80% of ReduMIS quality. Repartitioning significantly improves upon the Base Algorithm on most instances. While overall performance remains modest with only three instances exceeding 99% quality, there

are no extreme outliers. Nine instances achieve ratios greater than 0.975, with the worst result being a ratio of 0.931. Boundary Local Search achieves performance very similar to Repartitioning, although slightly worse overall. Seven instances achieve ratios greater than 0.975 with the worst result at 0.921.

The Reduction algorithms perform best, both with and without vertex folding. Most instances achieve very good results with ratios of at least 0.99968 for eight instances. The remaining three instances perform less well, particularly one outlier with a ratio of approximately 0.915. On average, the Reduction algorithm achieves a ratio of 98.8% across all instances. The performance of these algorithms mostly depends on the effectiveness of the reductions during preprocessing and block-wise reducing. For the eight well-reducible instances, the remaining kernel averaged 3.1% of the original graph size for reductions and 2.4% for reductions with vertex folding. For the other three instances, the average remaining graph size was 46.7%, with a remaining size of even 79.1% for the instance with the worst solution quality. Vertex folding also did not prove to be effective on these instances. The remaining graph size did not significantly differ between the two variants of the Reductions algorithm.

Liu et al.’s algorithms show mixed performance. The 1-* swap algorithm performs mediocre, with most results better than 0.975. The 2-* swap algorithm performs significantly better than 1-* swap. The results for both of these algorithms include one severe outlier with solution quality below 80%. This result occurs for both on the same graph, Bump_2911, which is a mesh-type instance. This is unexpected, considering that the 1-* swap and 2-* swap algorithms are local search algorithms, while their results are evaluated in comparison to mostly reduction based approaches. Usually, reduction-based algorithms perform worse on mesh-type instances, as these are oftentimes not well-reducible. Apart from this outlier instance, the 2-* swap algorithm achieves results greater than 0.974 on every instance and is only consistently outperformed by the reduction approaches, achieving better results than all block-wise solving approaches on most instances.

Memory usage. Figure 5.11b shows a performance profile of the memory consumption across the different algorithms. Liu et al.’s algorithms use by far the lowest amount of memory, as they are not storing any parts of the graph in memory at any point. ReduMIS, as a solver that loads the entire graph in memory, naturally uses the most memory. Our algorithms fall between these results, which is to be expected for partitioning-based approaches.

Our algorithms show relatively similar memory usage. On some instances, HeiStream and $O(n)$ vectors dominate memory consumption, for example on graphs where preprocessing is highly effective. In these cases, the differences between the algorithms become negligible and the memory usage is similar across all algorithms. Generally, the Base Algorithm and Repartitioning show the lowest memory usage among our approaches, with Repartitioning requiring slightly more. Both Reduction algorithms require slightly more memory, with vertex folding adding minimal overhead. This overhead is surprisingly small considering the implementation differences. Boundary Local Search requires the most memory, exceeding all other approaches on every instance except one.

Running time. Figure 5.11c shows the running time across algorithms. Liu et al.’s swap algorithms are generally the fastest. While they require multiple read streams, they include minimal time overhead beyond this.

Partitioning-based algorithms are generally slower. They also require multiple read streams, but additionally include significant overhead due to writing blocks to binary files.

Two instances show very fast performance for the Base Algorithm and Reduction algorithms due to highly effective preprocessing. The Base Algorithm and Reduction algorithms show similar overall performance, but excel on different instances. Reduction algorithms are fast when graphs are well-reducible. When the remaining kernel is large, substantial time is required to solve it, even after already spending considerable time on reductions. The running time of the Base Algorithm is less dependent on specific graph properties.

Repartitioning and Boundary Local Search are generally slow, being the slowest on every instance except a few ReduMIS instances. They are slower by a significant margin. Repartitioning and Boundary Local Search are extensions of the Base Algorithm, thereby also extending its running time.

ReduMIS shows highly variable running times. It is very fast on a few well-reducible graphs but much slower on others where it continues attempting to improve the solution. The running times are here limited by the configured time limit of one hour.

Huge Graphs

ReduMIS is excluded from this evaluation as these graphs are too large for in-memory processing. The graphs contain tens of millions of nodes with edge counts ranging from hundreds of millions to 3.3 billion.

We present results for two block size configurations: large blocks resulting from a partitioning with $k = 32$ and small blocks resulting from a partitioning with $k = 512$. For each of the two block sizes, we present the metrics solution size, memory usage and running time, each in a performance profile. Liu et al.’s 1-* swap and 2-* swap algorithms do not depend on block size and appear identically in both groups of plots.

Boundary Local Search is excluded from the small block size evaluation. With $k = 512$, it exceeded available memory on every instance. The large number of blocks combined with dense quotient graphs (see Section 5.2.4) creates a huge number of sections that must be kept in memory simultaneously. The huge graphs also produce large boundaries, making section storage infeasible.

Before examining individual algorithms, we highlight two outlier instances: nlpkkt240 and orkut. Both graphs have highly homogeneous node degrees with almost no degree-0 or degree-1 nodes and few very high-degree nodes, making preprocessing ineffective. Furthermore, the partitioning results are poor. For large blocks, $k = 32$, the edge-cut for orkut exceeds 67 million edges which are 57.4% of the total number of edges of the original graph. For nlpkkt240, the edge-cut exceeds 270 million, which is 74.5% of the total number of edges. For small blocks, $k = 512$, the same issue occurs to an even larger

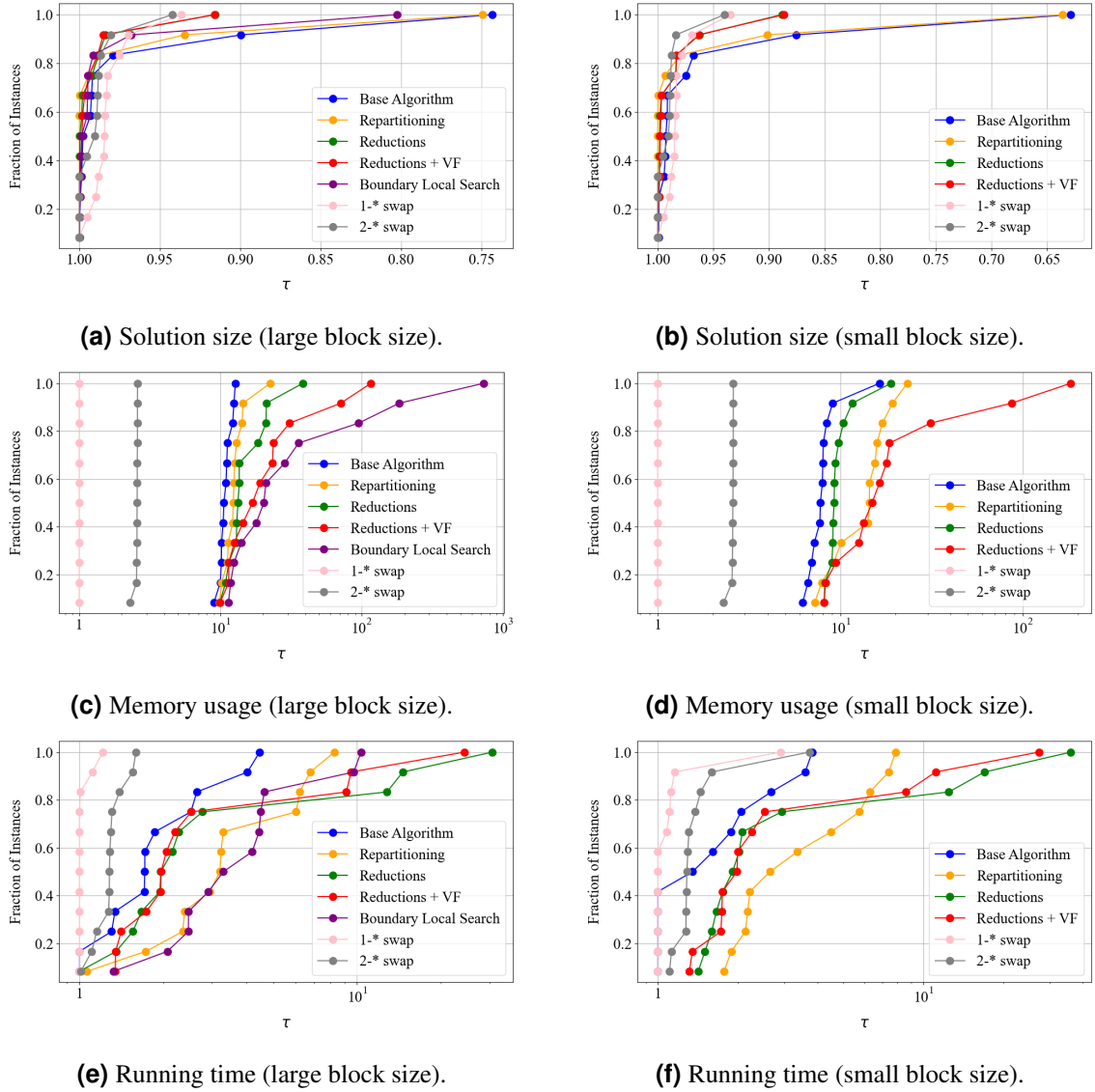


Figure 5.12: Comparison of algorithms on the Huge Graphs set. For each metric solution size, memory usage and running time, the results of the algorithms are displayed in performance profiles. The algorithms are compared in two settings: with a large block size and a small block size.

extend. The orkut edge-cut reaches approximately 90 million or 77.4% of total edges, while nlpkkt240's edge-cut exceeds 330 million or 88.9% of edges.

Large boundaries are problematic for all our algorithms. Block-wise solving approaches perform greedy, suboptimal node selection for boundary vertices. Block-wise reduction approaches are constrained because no reductions can be applied that would add boundary vertices to the independent set.

The results of this evaluation are shown in Figure 5.12. This figure contains a performance profile for large block sizes and for small block sizes for each of the two block size configurations.

Solution size. The Base Algorithm achieves decent results with large blocks on most instances, reaching at least 99% quality on 9 of 12 instances. With smaller blocks, results are slightly worse but comparable, maintaining 99% quality on 8 of 12 instances. Performance is especially poor on the two outlier instances: 89.9% on orkut and 74.3% on nlpkt240 with large blocks. With small blocks, results deteriorate further to 87.5% for orkut and 62.9% for nlpkt240.

Repartitioning shows a similar curve but achieves better results than the Base Algorithm on every instance. It reaches extremely high ratios on the majority of instances: at least 0.9997 for 8 instances. The performance on the two outlier instances is similar to the Base Algorithm with no significant improvement: 0.749 and 0.636 for nlpkt240 with large and small blocks, respectively.

Boundary Local Search produces generally good results, achieving at least 99% quality on ten instances. It performs minimally worse than Repartitioning on most instances but shows significantly better results than the Base Algorithm and Repartitioning on the two outlier instances, though performance remains noticeably poor with a ratio of only 80.3% for the nlpkt240 graph.

The Reduction algorithms produce nearly identical results with and without vertex folding. They achieve very good results on all instances except orkut, reaching at least 99% quality on 9 of 12 instances for both variants. Interestingly, performance on nlpkt240 is good, partially due to applied reductions but also because the global kernel construction changes node order. Blocks are written sequentially, so when the kernel is solved, the new partitioning operates on a graph with nodes grouped by blocks from the first partitioning. This produces a far better edge-cut when partitioning the kernel, leading to an overall good solution.

Reduction effectiveness is consistent across configurations. Without vertex folding, the remaining kernel averages 22.8% of the original graph size. This reduction in size is primarily due to edge removal as 88.1% of nodes remain in the kernel on average. With vertex folding, the kernel size is 22.6% with slightly fewer nodes, 86.8% on average. For smaller block sizes, reductions achieve similar success: 23.9% remaining kernel size without vertex folding, 23.8% with it. Solution quality is therefore very similar for large and small blocks, achieving at least 99% quality on 8 of 12 instances even with small blocks.

Liu et al.’s swap algorithms produce generally good results on all instances with no extreme outliers. The worst results are ratios of 0.934 for 1-* swap and 0.94 for 2-* swap. Excluding these, all results reach at least 0.97 quality for 1-* swap and 0.98 for 2-* swap. However, a gap remains between the best found solutions and swap solutions for most instances. The 1-* swap achieves 99% or higher quality on only 2 instances, while 2-* swap reaches this on 5 instances. These results are consistent when comparing with partitioning-based algorithms using either large or small blocks.

Memory usage. Liu et al.’s algorithms use by far the least memory, due to not storing

parts of the graph in memory. Partitioning-based approaches cannot realistically achieve comparable memory usage.

Among our algorithms, the Base Algorithm consistently uses the least memory for both large and small blocks, as expected. Repartitioning uses slightly more, which is also expected. The Reduction algorithm requires slightly more memory than the Base Algorithm, especially on instances with large edge-cuts. Reductions with vertex folding use significantly more memory than without, especially on instances with large edge-cuts, due to the need to store portions of cut edges. This effect is amplified for smaller block sizes, which lead to larger boundaries. The Boundary Local Search has the largest memory usage of all compared algorithms. For large blocks, the approach has the largest memory usage on most instance. Moreover, there are a few instances where the difference to the other approaches is very significant. This is again more apparent for the instances with large edge-cuts and those with dense quotient graphs. Each experiment with the Boundary Local Search on small block size needed to be terminated because the available memory was exceeded.

Memory usage is generally similar for large and small blocks. As detailed in the previous section, the dependency of memory usage on block size exists but is less prominent than expected. Memory consumption appears dominated by $O(n)$ vectors, indicating room for improvement.

In terms of absolute numbers, 1-* swap remains in the hundreds of megabytes. The 2-* swap requires approximately 2.5-3 times more memory, reaching up to 1.5 GB for large instances. Our algorithms use multiples of this. For large blocks, the Base Algorithm peaks at 6.2 GB, Repartitioning at 6.9 GB, and the Reductions algorithm at 7.3 GB. While Reductions+VF and Boundary Local Search show comparable memory usage on most instances, on some instances Reductions+VF reaches up to 12 GB and Boundary Local Search escalates to 58.2 GB of memory usage on one instance.

With smaller blocks, the Base Algorithm uses at most 4.7 GB and Reductions 5.1 GB. Reductions+VF uses slightly less memory overall but still reaches 11.8 GB in extreme cases. Repartitioning uses more memory compared to large blocks, reaching 7.8 GB. Boundary Local Search is infeasible on these graphs with large numbers of blocks.

Running time. On most instances, Liu et al.'s 1-* swap and 2-* swap are the fastest algorithms. Among our algorithms, the Base Algorithm is fastest. Repartitioning and Boundary Local Search are naturally slower on all instances compared to the Base Algorithm.

The reduction-based algorithms show diverse running times. On well-reducible graphs, they can be relatively fast, though still slightly slower than the Base Algorithm. On less reducible graphs, running time can increase substantially, making them the slowest algorithms.

Running time is generally consistent across different block sizes. The outlier instances nlpkkt240 and orkut become apparent again: these are the instances where most partitioning-based algorithms perform slowest compared to the non-partitioning-based 1-* swap and 2-* swap.

Summary. In total, all of our algorithms produce higher quality solutions than the competitor algorithms on the majority of instances. However, when our algorithms perform

worse, the gap can be substantial. These poor results are confined to two instances where the computed partitioning has an extremely large edge-cut. As all of our approaches are partitioning-based, this directly impacts the result of all of our algorithms. Adjusting partitioning configuration, such as using larger batch sizes or more passes, could help for these instances.

The memory usage of all our algorithms is significantly larger than the competitor algorithms. While this is partially inherent to the approach, partitioning and loading blocks into memory requires more memory, there is room for improvement in terms of memory efficiency. Still, except for Boundary Local Search and some instances of Reductions+VF, memory usage remains within reasonable bounds.

Among our algorithms, the Base Algorithm is fastest with the lowest memory requirements but produces the worst results. Repartitioning produces very good results but requires substantially more running time. The Reduction algorithms achieve the highest solution quality. Compared to the simple Reductions algorithm, Reductions+VF offers no significant improvement in solution size while using considerably more memory on some instances. Boundary Local Search produces good solutions but has large memory requirements, especially due to large boundaries and dense quotient graphs, making it impractical for large numbers of blocks.

Discussion

This chapter summarizes the contributions and findings of this thesis and outlines directions for future work. We begin with a conclusion reviewing the developed algorithms and their evaluation results, then discuss potential improvements and extensions.

6.1 Conclusion

In this thesis, we studied the MIS problem for very large graph instances. We developed multiple approaches based on partitioning a graph and processing individual blocks.

The Base Algorithm provides a simple approach that sequentially solves blocks of a partitioning. We presented two extensions of the Base Algorithm: Repartitioning and Boundary Local Search. Each of these is improving specific parts of the solution through local search. The Reductions algorithm applies reduction rules block-wise while considering context from neighboring blocks. Reductions+VF adapts this approach to enable additional reduction rules.

The experimental evaluation demonstrated that our algorithms generally produce higher quality solutions than the competitor algorithms on the majority of instances. However, on instances where the edge-cut of the computed partitioning is large, the solution quality of all of our algorithms decreases significantly.

Our algorithms use overall significantly more memory than the algorithms developed by Liu et al., which achieve very low memory consumption. This is partially inherent to our partitioning-based approach but also indicates the need for optimization in terms of memory efficiency. However, the memory usage of our algorithms remains within reasonable bounds, with the exception of the Boundary Local Search algorithm that can become infeasible for large graphs, especially for a larger number of blocks.

Among our developed algorithms, the Reduction algorithms, both with and without vertex folding, achieve the highest solution quality. The Base Algorithm produces the lowest quality results among our approaches but is also the fastest and requires the lowest memory.

Repartitioning and Boundary Local Search achieve good results but require substantially more running time.

6.2 Future Work

The algorithms presented in this thesis offer several directions for further improvement and extension.

Memory Optimization. Mainly, the algorithms can be improved in terms of memory consumption. There are multiple potential optimizations where the current implementation favors flexibility over efficiency:

Different libraries used in our algorithms employ different graph formats. DataReductions, CHILS, and ParFastKer each use their own representation. Our algorithms maintain an internal graph format for core operations, then convert to the respective library formats when needed. This conversion overhead can be eliminated by using the target format from the beginning. For example, once a block solver is selected for the Base Algorithm, blocks can be read directly into that solver’s format. Operations like creating independent subgraphs would need to be adapted to work with the specific format, but this would reduce both memory consumption and running time.

Avoidable allocation of new memory when transforming a graph also occurs during subgraph creation. When creating reduced subgraphs, for instance, after removing boundary nodes with neighbors in the independent set, a new graph is allocated. Instead, these subgraphs could be constructed in place, overwriting the pre-reduced graph and reusing existing memory.

Furthermore, the $O(n)$ vectors used throughout the algorithms can be optimized. For example, storing the independent set as a boolean array instead of integers would reduce memory usage. Additionally, for some algorithms, the vector storing the partitioning is only relevant within a limited scope and could be removed or reused for the independent set representation after that scope.

A modification to HeiStream could also improve memory efficiency. Currently, HeiStream’s batch size depends only on the node count. Basing the batch size on both node and edge count would improve the predictability of memory consumption, allowing larger batch sizes to be configured safely while accounting for less variance in actual memory usage.

Extended Reduction Rules. Apart from memory optimizations, the results of the reductions algorithms could be improved by extending with additional reduction rules. The current implementation uses only a subset of available reduction techniques. Incorporating more sophisticated reduction rules could further decrease kernel sizes, while possibly negatively impacting the running time.

Weighted Independent Sets. This work focuses exclusively on independent sets on unweighted graphs. Extending the algorithms to handle the weighted case remains an open direction for future research.

Abstract (German)

Diese Arbeit stellt Semi-Streaming-Algorithmen vor, die entwickelt wurden, um hochwertige Lösungen für das Maximum Independent Set (MIS) Problem zu finden, ohne dass der gesamte Graph in den Speicher geladen werden muss. Damit wird eine Lücke im Bereich der Semi-Streaming-MIS Algorithmen geschlossen, die einen praktisch sinnvollen Kompromiss zwischen Speichereffizienz und Lösungsqualität darstellen. Das MIS-Problem ist ein klassisches NP-schweres Problem mit Anwendungen in verschiedenen Bereichen, wie z.B. Computergrafik, Map-Labeling (Kartenbeschriftung) und Informationskodierung. Da moderne Graphen immer größer werden, wird der Bedarf an Algorithmen, deren Speichernutzung weniger als linear zur Graphengröße liegt, zunehmend relevant. Dies motiviert die Entwicklung von Semi-Streaming-Methoden, die davon ausgehen, dass die Knoten eines Graphs in den Speicher passen, die Kanten jedoch nicht.

Wir führen Semi-Streaming-Algorithmen ein, die auf Graphpartitionierung beruhen. Diese beruhen auf zwei grundsätzlichen Ansätzen entweder den Graph blockweise zu lösen oder den Graphen blockweise zu reduzieren. Basierend auf dem blockweise-lösen Ansatz haben wir die Varianten "Base Algorithm", "Repartitioning" und "Boundary Local Search" implementiert, während der blockweise Reduktionsansatz die Varianten "Basic Reductions" und "Reductions+VF" (Vertex Fold) umfasst. Alle Ansätze beinhalten einen Vorverarbeitungsschritt, bei dem einfache Reduktionen während eines einzigen Streams angewendet werden.

Experimentelle Auswertungen anhand großer Instanzen zeigen, dass unsere Algorithmen für die meisten Instanzen qualitativ hochwertigere Lösungen liefern als die bestehenden Semi-Streaming-Algorithmen von Liu et al. . Die Reduktionsalgorithmen erzielten unter unseren Methoden die höchste Lösungsqualität. Der "Base Algorithm" bot unter den entwickelten Ansätzen die schnellste Laufzeit und den geringsten Speicherbedarf.

Der partitionierungsbasierte Ansatz hat jedoch zwei nennenswerte Nachteile: Erstens ist der Speicherverbrauch deutlich höher als bei den verglichenen Algorithmen. Zweitens hängt die Qualität der Ergebnisse von der Qualität der Partitionierung ab, wobei Ausreißer mit extrem großen Edge-Cut (deutsch: Kantenschnitten) zu einer deutlich schlechteren Lösungsqualität führen können.

Bei einer Bewertung kleinerer Instanzen, die einen Vergleich mit In-Memory-Algorithmen ermöglicht, erreicht der Reduktionsalgorithmus im Durchschnitt eine Lösungsqualität von 98,8% im Vergleich zu ReduMIS.

Bibliography

- [1] Takuya Akiba and Yoichi Iwata. Branch-and-reduce exponential/fpt algorithms in practice: A case study of vertex cover. *Theor. Comput. Sci.*, 609:211–225, 2016.
- [2] Diogo Vieira Andrade, Mauricio G. C. Resende, and Renato Fonseca F. Werneck. Fast local search for the maximum independent set problem. *J. Heuristics*, 18(4):525–547, 2012.
- [3] Sepehr Assadi, Christian Konrad, Kheeran K. Naidu, and Janani Sundaresan. $O(\log \log n)$ passes is optimal for semi-streaming maximal independent set. In Bojan Mohar, Igor Shinkar, and Ryan O’Donnell, editors, *Proceedings of the 56th Annual ACM Symposium on Theory of Computing, STOC 2024, Vancouver, BC, Canada, June 24–28, 2024*, pages 847–858. ACM, 2024.
- [4] David A. Bader, Andrea Kappes, Henning Meyerhenke, Peter Sanders, Christian Schulz, and Dorothea Wagner. Benchmarking for graph clustering and partitioning. In Reda Alhajj and Jon G. Rokne, editors, *Encyclopedia of Social Network Analysis and Mining, 2nd Edition*. Springer, 2018.
- [5] Jannick Borowitz, Ernestine Großmann, and Matthias Schimek. Distributed reductions for the maximum weight independent set problem, 2025.
- [6] Sergiy Butenko, Panos M. Pardalos, Ivan Sergienko, Vladimir Shylo, and Petro Stetsyuk. Finding maximum independent sets in graphs arising from coding theory. In Gary B. Lamont, Hisham Haddad, George A. Papadopoulos, and Brajendra Panda, editors, *Proceedings of the 2002 ACM Symposium on Applied Computing (SAC), March 10–14, 2002, Madrid, Spain*, pages 542–546. ACM, 2002.
- [7] Xiuge Chen, Rajesh Chitnis, Patrick Eades, and Anthony Wirth. Sublinear-space streaming algorithms for estimating graph parameters on sparse graphs. In Pat Morin and Subhash Suri, editors, *Algorithms and Data Structures - 18th International Symposium, WADS 2023, Montreal, QC, Canada, July 31 - August 2, 2023, Proceedings*, volume 14079 of *Lecture Notes in Computer Science*, pages 247–261. Springer, 2023.
- [8] Graham Cormode, Jacques Dark, and Christian Konrad. Independent set size approximation in graph streams. *CoRR*, abs/1702.08299, 2017.

- [9] Graham Cormode, Jacques Dark, and Christian Konrad. Independent sets in vertex-arrival streams. In Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi, editors, *46th International Colloquium on Automata, Languages, and Programming, ICALP 2019, July 9-12, 2019, Patras, Greece*, volume 132 of *LIPIcs*, pages 45:1–45:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.
- [10] Jakob Dahlum, Sebastian Lamm, Peter Sanders, Christian Schulz, Darren Strash, and Renato F. Werneck. Accelerating local search for the maximum independent set problem. In Andrew V. Goldberg and Alexander S. Kulikov, editors, *Experimental Algorithms - 15th International Symposium, SEA 2016, St. Petersburg, Russia, June 5-8, 2016, Proceedings*, volume 9685 of *Lecture Notes in Computer Science*, pages 118–133. Springer, 2016.
- [11] Elizabeth D. Dolan and Jorge J. Moré. Benchmarking optimization software with performance profiles. *Math. Program.*, 91(2):201–213, 2002.
- [12] Marcelo Fonseca Faraj and Christian Schulz. Buffered streaming graph partitioning. *ACM J. Exp. Algorithmics*, 27:1.10:1–1.10:26, 2022.
- [13] Ada Wai-Chee Fu, Huanhuan Wu, James Cheng, Shumo Chu, and Raymond Chi-Wing Wong. IS-LABEL: an independent-set based labeling scheme for point-to-point distance querying on large graphs. *CoRR*, abs/1211.2367, 2012.
- [14] Daniel Funke, Sebastian Lamm, Ulrich Meyer, Manuel Penschuck, Peter Sanders, Christian Schulz, Darren Strash, and Moritz von Looz. Communication-free massively distributed graph generation. *J. Parallel Distributed Comput.*, 131:200–217, 2019.
- [15] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.
- [16] Andreas Gemsa, Martin Nöllenburg, and Ignaz Rutter. Evaluation of labeling strategies for rotating maps. *ACM J. Exp. Algorithmics*, 21(1):1.4:1–1.4:21, 2016.
- [17] Ernestine Großmann and Kenneth Langedal. Data reductions. <https://github.com/KarlsruheMIS/DataReductions>, 2025.
- [18] Ernestine Großmann, Kenneth Langedal, and Christian Schulz. Accelerating reductions using graph neural networks and a new concurrent local search for the maximum weight independent set problem. *CoRR*, abs/2412.14198, 2024.
- [19] Jiewei Gu, Weiguo Zheng, Yuzheng Cai, and Peng Peng. Towards computing a near-maximum weighted independent set on massive graphs. In Feida Zhu, Beng Chin Ooi, and Chunyan Miao, editors, *KDD '21: The 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, Singapore, August 14-18, 2021*, pages 467–477. ACM, 2021.

-
- [20] Bjarni V. Halldórsson, Magnús M. Halldórsson, Elena Losievskaja, and Mario Szegedy. Streaming algorithms for independent sets. In Samson Abramsky, Cyril Gavoille, Claude Kirchner, Friedhelm Meyer auf der Heide, and Paul G. Spirakis, editors, *Automata, Languages and Programming, 37th International Colloquium, ICALP 2010, Bordeaux, France, July 6-10, 2010, Proceedings, Part I*, volume 6198 of *Lecture Notes in Computer Science*, pages 641–652. Springer, 2010.
 - [21] Bjarni V. Halldórsson, Magnús M. Halldórsson, Elena Losievskaja, and Mario Szegedy. Streaming algorithms for independent sets in sparse hypergraphs. *Algorithmica*, 76(2):490–501, 2016.
 - [22] Magnús M. Halldórsson, Xiaoming Sun, Mario Szegedy, and Chengu Wang. Streaming and communication complexity of clique approximation. In Artur Czumaj, Kurt Mehlhorn, Andrew M. Pitts, and Roger Wattenhofer, editors, *Automata, Languages, and Programming - 39th International Colloquium, ICALP 2012, Warwick, UK, July 9-13, 2012, Proceedings, Part I*, volume 7391 of *Lecture Notes in Computer Science*, pages 449–460. Springer, 2012.
 - [23] Jan-Henrik Haunert and Alexander Wolff. Beyond maximum independent set: An extended integer programming formulation for point labeling. *ISPRS Int. J. Geo Inf.*, 6(11):342, 2017.
 - [24] Demian Hesse, Christian Schulz, and Darren Strash. Scalable kernelization for maximum independent sets. *ACM Journal of Experimental Algorithmics*, 24(1):1.16:1–1.16:22, 2019.
 - [25] Sebastian Lamm, Peter Sanders, and Christian Schulz. Graph partitioning for independent sets. In Evripidis Bampis, editor, *Experimental Algorithms - 14th International Symposium, SEA 2015, Paris, France, June 29 - July 1, 2015, Proceedings*, volume 9125 of *Lecture Notes in Computer Science*, pages 68–81. Springer, 2015.
 - [26] Sebastian Lamm, Peter Sanders, Christian Schulz, Darren Strash, and Renato F. Werneck. Finding near-optimal independent sets at scale. *J. Heuristics*, 23(4):207–229, 2017.
 - [27] Sebastian Lamm, Christian Schulz, Darren Strash, Robert Williger, and Huashuo Zhang. Exactly solving the maximum weight independent set problem on large real-world graphs. In Stephen G. Kobourov and Henning Meyerhenke, editors, *Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments, ALENEX 2019, San Diego, CA, USA, January 7-8, 2019*, pages 144–158. SIAM, 2019.
 - [28] Kenneth Langedal, Demian Hesse, and Peter Sanders. Targeted branching for the maximum independent set problem using graph neural networks. In Leo Liberti, edi-

- tor, *22nd International Symposium on Experimental Algorithms, SEA 2024, July 23-26, 2024, Vienna, Austria*, volume 301 of *LIPIcs*, pages 20:1–20:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024.
- [29] Jure Leskovec and Andrej Krevl. Snap datasets: Stanford large network dataset collection, 2014.
- [30] Yu Liu, Jiaheng Lu, Hua Yang, Xiaokui Xiao, and Zhewei Wei. Towards maximum independent sets on massive graphs. *Proc. VLDB Endow.*, 8(13):2122–2133, 2015.
- [31] Michael Luby. A simple parallel algorithm for the maximal independent set problem. In Robert Sedgewick, editor, *Proceedings of the 17th Annual ACM Symposium on Theory of Computing, May 6-8, 1985, Providence, Rhode Island, USA*, pages 1–10. ACM, 1985.
- [32] Pedro V. Sander, Diego Nehab, Eden Chlamtac, and Hugues Hoppe. Efficient traversal of mesh edges using adjacency primitives. *ACM Trans. Graph.*, 27(5):144, 2008.
- [33] Tycho Strijk, Bram Verweij, Karen Aardal, et al. Algorithms for maximum independent set applied to map labelling. *Department of Computer Science, Utrecht University*, 2000.
- [34] Charalampos E. Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. FENNEL: streaming graph partitioning for massive scale graphs. In Ben Carterette, Fernando Diaz, Carlos Castillo, and Donald Metzler, editors, *Seventh ACM International Conference on Web Search and Data Mining, WSDM 2014, New York, NY, USA, February 24-28, 2014*, pages 333–342. ACM, 2014.
- [35] Jeffrey S Warren and Illya V Hicks. Combinatorial branch-and-bound for the maximum weight independent set problem. *Relatório Técnico, Texas A&M University, Citeseer*, 9:17, 2006.
- [36] Daniel Ye. Deterministic independent sets in the semi-streaming model. In Keren Censor-Hillel, Fabrizio Grandoni, Joël Ouaknine, and Gabriele Puppis, editors, *52nd International Colloquium on Automata, Languages, and Programming, ICALP 2025, July 8-11, 2025, Aarhus, Denmark*, volume 334 of *LIPIcs*, pages 135:1–135:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025.