Karlsruher Institut für Technologie

Master thesis

Scalable Kernelization for the Maximum Independent Set Problem

Demian Hespe

Date: 30. Januar 2017

Supervisors: Prof. Dr. Peter Sanders
Dr. rer. nat. Christian Schulz
Dr. Darren Strash

Institute of Theoretical Informatics, Algorithmics
Department of Informatics
Karlsruhe Institute of Technology

Abstract

The NP-hard *maximum independent set* problem has applications in many real-world domains, such as coding theory [13], computer graphics [42], computational biology [15, 21] and route planning [29]. While the problems arising from these areas are not always in the form of graphs, they can be transformed into one and then handled by an independent set algorithm independent from the domain.

A technique that has been part of exact maximum independent set algorithms [2, 25, 48] for a long time and recently has been discovered to be beneficial for inexact algorithms [18, 32] is *kernelization*. Kernelization is the process of reducing the size of a graph without losing the information required to compute a maximum independent set of the original graph. Algorithms for finding a kernel are much faster than algorithms for directly finding a maximum independent set. Intuitively, a kernel is the part of a graph that makes the maximum independent set problem hard. Exact *branch-and-reduce* algorithms find a kernel of the input graph, then branch at carefully chosen vertices and compute kernels of the new graphs after branching. Recently, finding a kernel before or during *local search* and *evolutionary* algorithms has also be found to be beneficial.

In the past, extensive work has been done to lower the theoretical worst-case time complexity of finding a maximum independent set. Furthermore, there is a growing focus on algorithms that find large independent sets, but not necessarily a maximum independent set. However, very little work has been done parallelizing maximum independent set algorithms. As kernelization plays an important role in both these approaches, we develop a fast parallel algorithm for finding a relaxed version of a kernel. This relaxed kernel is sufficiently small but might still have potential to be reduced to an even smaller size by sequential algorithms. Beside parallelism we also employ a technique to prune vertices that cannot be used to further decrease the graph size at the current state of the algorithm.

In this thesis we give a detailed explanation of how our kernelization algorithm works. Besides a theoretical view on the parallel application of kernelization techniques and the pruning of vertices, we also employ an extensive experimental evaluation of our implementation. Our results show that sequentially, we can find a kernel a factor of up to 4 faster than previous results and a factor of up to 11 faster using parallelism.

Zusammenfassung

Das NP-schwere Problem der *größten unabhängigen Menge* findet Anwendung in vielen Domänen, wie Kodierungstheorie [13], Computergrafik [42], Bioinformatik [15, 21] und Routenplanung [29]. In einigen dieser Bereiche liegt das Problem nicht in Form eines Graphen vor. Indem die relevanten Teile der Problemstellung in einen Graphen konvertiert werden, können sie von einem Algorithmus gelöst werden, der unabhängig von der ursprünglichen Domäne entwickelt werden kann.

Eine Technik, die bereits seit Langem für exakte Algorithmen zur Findung von größten unabhängigen Mengen [2, 25, 48] verwendet wird und kürzlich auch für inexakte Algorithmen [18, 32] als hilfreich entdeckt wurde, ist die *Kernfindung*. Die Kernfindung bezeichnet einen Prozess, in dem die Größe des Eingabegraphen verkleinert wird, ohne dabei die Informationen zu verlieren, die zur Berechnung einer größten unabhängigen Menge benötigt werden. Die hierfür verwendeten Algorithmen sind deutlich schneller als Algorithmen, die direkt eine größte unabhängige Menge finden. Anschaulich ist ein Kern der Teil eines Graphen, der das Problem der größten unabhängigen Menge schwer macht. Exakte *branch-and-reduce*-Algorithmen berechnen einen Kern des Eingabegraphen, verzweigen dann an sorgfältig ausgewählten Knoten und berechnen erneut Kerne der neuen Graphen, die beim Verzweigen entstanden sind. Kürzlich wurde die Kernfindung auch erfolgreich vor oder während *lokaler Suche-* und *evolutionären* Algorithmen eingesetzt.

In der Vergangenheit gab es bereits weitgehende Forschung mit dem Ziel die worst-case Laufzeit von exakten Algorithmen für größte unabhängige Mengen zu senken. Ebenfalls gibt es ein größer werdendes Interesse an inexakten Algorithmen, die zwar eine große unabhängige Menge berechnen, allerdings nicht zwingend eine größtmögliche. Ein Bereich, in dem es allerdings überraschend wenig Ergebnisse in der Literatur gibt, ist die Parallelisierung solcher Algorithmen. Da die Kernfindung eine entscheidende Rolle in sowohl exakten als auch inexakten Algorithmen spielt, wird ein schneller paralleler Algorithmus zur Findung einer relaxierten Version eines Kerns entwickelt. Dieser relaxierte Kern ist ausreichend klein um ihn für weiterführende Algorithmen zu verwenden, könnte aber mit sequenziellen Algorithmen noch weiter verkleinert werden. Neben Parallelismus wird auch eine Technik angewendet, durch die Knoten zeitsparend ausgeschlossen werden können, die im aktuellen Zustand nicht zur weiteren Verkleinerung des Graphen beitragen können.

Die vorliegende Arbeit erklärt detailliert, wie der entwickelte Kernfindungsalgorithmus funktioniert. Neben einer theoretischen Sicht auf die parallele Anwendung von Kernfindungstechniken und die Ausschließung von Knoten, wird auch eine ausführliche experimentelle Evaluation durchgeführt. Die dabei erzielten Ergebnisse zeigen, dass ein Kern in einer Laufzeit gefunden werden kann, die um einen Faktor von bis zu 4 schneller ist als bisherige Ergebnisse. Durch die Verwendung von Parallelismus kann ein Kern sogar um einen Faktor 11 schneller gefunden werden.

Acknowledgments

I would like to thank my supervisors Dr. Christian Schulz and Dr. Darren Strash as well as Prof. Dr. Peter Sanders for the opportunity to work on such an interesting topic. I had countless helpful discussions with my supervisors and got valuable feedback from them. I felt very well cared for under their supervision.

I would also like to thank all the other students in the student lab for the valuable conversations with them about my thesis project as well as interesting talks about their work. The working environment in the student pool made the challenging work on my thesis project much more pleasant.

Contents

1 Introduction

1.1 Motivation

Independent sets, i.e. sets of pairwise nonadjacent vertices in a graph, have applications in many areas, including coding theory, computer graphics and route planning. Most applications require large independent sets or even one of maximum cardinality (or equivalently a *minimum vertex cover* or a *maximum clique*). In coding theory [13], a maximum independent set is used to find a code with a minimum hamming distance of h. In the respective graph, a vertex is added for every possible code word and an edge between two vertices is added if the hamming distance between the associated code words is less than h. A maximum independent set in this graph represents a maximum cardinality code with minimum hamming distance h. In computer graphics [42] a small vertex cover is used to find a set of triangles that cover all edges in a triangle mesh. In computational biology [15] maximum cliques are used to find maximally complementary sets of donor/acceptor pairs for protein docking. Other examples of maximum cliques in computational biochemistry can be found in [21]. Independent sets are also used for map labeling [22, 23] by building a graph where each label is represented by a vertex and and edge is added when a label would overlap with some other label. Every independent set then represents a set of labels that can be drawn without overlapping. In route planning [29], in particular contraction hierarchies, independent sets can be used to find vertices that can be contracted in parallel.

A technique that has proven to be efficient in both theory [25, 48] and practice [1, 2, 18, 32] is to modify the input graph and decrease its size such that it is possible to obtain a maximum independent set of the input graph from a maximum independent set of the smaller graph. We call this smaller graph the *kernel* of the graph and obtain it by applying a set of reduction rules. After finding a maximum independent set of the kernel we can undo the reduction rules to get a maximum independent set of the original graph.

Finding the kernel often has a huge impact on the running time of algorithms. In most algorithms that use a kernel, a significant part of the running time is spend on kernelization. For example, the evolutionary algorithm by Lamm et al. [32] finds a very large independent set very fast after finding a kernel. However, kernelization is a bottleneck for their algorithm.

1.2 Contribution

We aim to reduce the long kernelization times by designing a parallel kernelization algorithm in combination with other techniques to speed up the computation of a kernel. We compute a kernel by partitioning the input graph in blocks and processing each block in a separate thread, while trying to apply as many reductions as possible. We also introduce a technique for selecting vertices that are candidates for further reductions.

We evaluate our algorithm on a set of large network graph instances and compare our results to previous work. Sequentially, we can find a kernel a factor of up to 4 faster than previous work and a factor of up to 11 faster using parallelization. We believe that we can speed up maximum independent set algorithms significantly by using our fast kernelization as a preprocessing step.

1.3 Structure of Thesis

We start by giving an overview of the notation and definitions used throughout the thesis in Chapter 2. We then survey related work on independent sets and related problems in Chapter 3. In Chapter 4 we describe the reductions used for our algorithm. In Chapter 5 we introduce a way to restrict attention to areas of the graph where reductions can be applied and explain the parallel application of reductions in Chapter 6. Chapter 7 then discusses our experimental evaluation and we close in Chapter 8.

2 Preliminaries

2.1 General Definitions

In this section we cover general definitions required to follow the rest of the thesis.

A graph $G = (V, E)$ consists of a set of vertices V and a set of edges $E \subset V \times V$ connecting vertices. In this thesis we assume undirected graphs where when an edge (u, v) is in E then (v, u) must also be in E, so edges can be written as unordered pairs $\{u, v\}$ rather than ordered pairs. When there is an edge between two vertices u and v, we say that u and v are *adjacent* and u is v's neighbor. For an edge $e = \{u, v\}$ we say that e and u (and e and v) are incident.

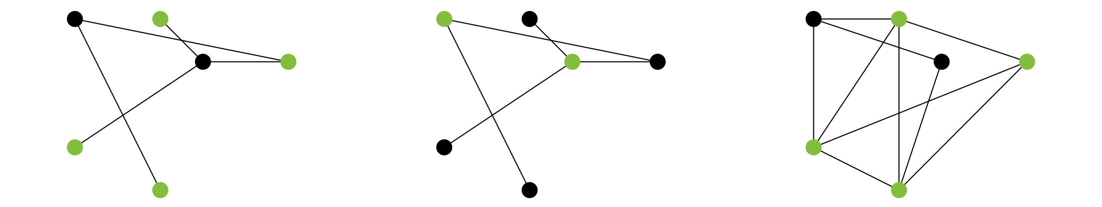
We call the set $N_G(v)$ consisting of all of v's neighbors in a graph G the *open neighborhood* of v and $|N_G(v)|$, which is the number of edges incident to v, the degree of v. We also define the *closed neighborhood* of a vertex: $N_G[v] = N_G(v) \cup \{v\}$. These definitions can be extended for sets of vertices $S \subseteq V$ to $N_G(S) = \bigcup_{v \in S} N_G(v)$ and $N_G[S] = \bigcup_{v \in S} N_G[v]$. We omit the subscript G when the graph used is clear.

A graph $G' = (V', E')$ with $V' \subset V$ and $E' \subset E \cap (V' \times V')$ is called a subgraph of G and $G'' = (V', E'')$ with $E'' = E \cap (V' \times V')$ is called the subgraph *induced* by V'. A graph $G = (V, E)$ is called a *clique* if all vertices in G are pairwise adjacent. For simplicity we also use the term clique for a set of vertices that induce a clique.

2.2 Graph Partitioning

The graph partitioning problem is the problem of dividing a graph $G = (V, E)$ into blocks $V_1 \cup \cdots \cup V_k = V$ with $V_i \cap V_j = \emptyset$, $\forall i \neq j$ while optimizing some cost function, typically the number of edges that have end points in different blocks. Additionally, a balance constraint is applied, demanding that the blocks have about equal size with respect to the number of vertices or the sum of some weights associated with the vertices. Vertices that are adjacent to vertices in other blocks and edges that cross block boundaries are of special interest. We call these *boundary vertices* and *cut edges*, respectively. We use the notation $b[v]$ to denote the block that v belongs to, so $\forall v \in V_i : b[v] = V_i$.

Graph partitioning is a technique commonly used to solve problems in parallel. Usually a vertex represents some unit of work. Units of work have dependencies, usually the need to communicate results, represented by edges between the respective vertices. By assigning



(a) A maximum independent set of G **(b)** A minimum vertex cover of G **(c)** A maximum clique of the complement graph $\bar{G}$

Figure 2.1: The maximum independent set problem and the related problems minimum vertex cover and maximum clique for a graph G. The green vertices are in the respective set.

each block of the partition to a processor, we have an equal amount of work on each processor (due to the balance constraint) and the communication required between processors is minimized (due to the cost function).

2.3 The Maximum Independent Set Problem

An *independent set* (IS) in a graph $G = (V, E)$ is a set of vertices $I \subset V$ such that $\forall u, v \in I : \{u, v\} \notin E$. That is, all vertices in I are pairwise nonadjacent. A *maximal* independent set is an independent set I that is not a proper subset of any other independent set. The *maximum* independent set problem (MIS) is the problem of finding an independent set that has maximum cardinality. That is, no other independent set contains more vertices. We denote the size of the maximum independent set of a graph G by $\alpha(G)$, also called the independence number of G. Figure 2.1a shows a maximum independent set in an example graph. It is well known that finding the maximum independent set of a graph is NP-hard.

Finding a maximal/maximum independent set is equivalent to finding a minimal/minimum vertex cover: A vertex cover is a set of vertices $C \subset V$ such that C contains either u or v for every edge $\{u, v\} \in E$. Given a maximal/maximum independent set I, $C = V \setminus I$ is a minimal/minimum vertex cover in G. Figure 2.1b shows a minimum vertex cover in an example graph.

The *maximum clique* problem is the problem of finding a maximum cardinality set of vertices in G that form a clique. We can transform it to the maximum independent set problem by constructing the complement graph: $\bar{G} = (V, \bar{E}), \bar{E} = \{u, v \in V \mid u \neq v \wedge \{u, v\} \notin E\}$. A clique in $\bar{G}$ is a maximum independent set in G. Figure 2.1c shows a maximum clique in an example graph.

2.4 Kernelization for the Maximum Independent Set Problem

One approach to finding a maximum independent set or at least a large independent set is to reduce the size of the graph by applying reduction operations before running algorithms to find these independent sets. Reduction operations are functions r that transform a graph $G = (V, E)$ into a new graph $r(G) = G' = (V', E')$ with $|V'| < |V|$. We can then find a maximum independent set of G' and revert the reductions applied to get a maximum independent set of G. We repeatedly apply a set of reductions to the graph. When none of these reductions can be applied to the graph anymore, we call the resulting graph a *kernel*. There are three general patterns for reducing the graph size used in this thesis:

- Prove that there is at least one maximum independent set that does *not* contain a certain vertex and remove it from the graph.
- Prove that there is at least one maximum independent set that *does* contain a certain vertex and fix it to be in an independent set I. Note that the neighborhood of this vertex cannot be in I, so we can remove the vertex and its neighborhood from the graph.
- Remove a set of vertices and replace it with a smaller set of vertices. After finding a maximum independent set of the kernel, this operation has to be reversed in order to find the maximum independent set of the original graph. This can be done when the larger set of vertices show a structure that can be expressed as a smaller set of vertices.

3 Related Work

Kernelization is widely used for solving MIS and related problems in theory and practice. We give a short overview on previous work focusing on reductions or using reduction operations as part of their algorithm.

3.1 Kernelization

Guo and Niedermeier [25] give a theoretical overview on kernelization for different problems, including k vertex cover, k independent set and bounds on the kernel size for the decision variants of the problems. The decision problem k independent set is, given a parameter k, is there an independent set of size at least k (or a vertex cover of size at most k for the k vertex cover problem). Here, a more restrictive definition of the term *kernel* is used where the size of the resulting graph must be bound by a function $g(k)$ only depending on the parameter k, not the size of the input graph. One important result in literature is that, given the parameter k for the vertex cover problem, the optimal kernel size achievable by polynomial algorithms is greater than $1.36k$ (Dinur and Safra [19]) unless $P = NP$ and there are kernelization algorithms that achieve a kernel size of $2k$ (Nemhauser and Trotter [38]).

Abu-Khzam et al. [1] use a collection of reduction rules to kernelize a collection of four small graphs that are complements of maximum clique instances from computational biology. In particular, they use the isolated clique reduction (described in Section 4.2) for cliques of size three or less, the vertex fold reduction (described in Section 4.1), the crown reduction (not used in this thesis) and two versions of the LP reduction (described in Section 4.6): One solving the linear program directly and one using a maximum matching of the bi-double graph. As they assume a given maximum size k of the vertex cover, they can also remove every vertex of degree $\geq k$ from the graph as it must be in all minimum vertex covers.

3.2 Exact Algorithms

Applying reduction rules to instances of NP-hard problems is used in algorithms that implement the *branch-and-reduce* paradigm where the instance is reduced by a set of polynomial time reduction rules followed by a branching into two or more sub problems. Branch-and-reduce algorithms for MIS use special algorithms for low degree graphs (with a degree bounded by some integer i) to obtain better worst case running times. For example, the vertex fold reduction (explained in Section 4.1) in combination with a special case of the isolated clique reduction (explained in Section 4.2) are used to solve MIS for graphs with a maximum degree of two.

The first important algorithm for MIS is due to Tarjan and Trojanowski [45] which uses the branch-and-reduce paradigm and runs in time $\mathcal{O}(n^{n/3})$. Later, Fomin et al. [20] introduced a new technique for analyzing branch-and-reduce algorithms called *measure & conquer* and show its effectiveness by proving a running time of $\mathcal{O}^*(2^{0.287n})$[1] for a simple algorithm for MIS. To the best of our knowledge the currently fastest exact polynomial space algorithm for MIS is by Xiao and Nagamotchi [48] and runs in $\mathcal{O}^*(1.1996^n)$, analyzed by the measure & conquest method. For their algorithm they consider three reduction rules: The first rule they use is removing so called *unconfined* vertices, which is also part of our algorithm (described in Section 4.4). The second rule is folding so called *complete k-independent sets* for $k \leq 2$. This reduction is also part of our algorithm: For $k = 2$ this corresponds to the vertex fold reduction (Section 4.1) and for $k = 1$ it is a special case of the isolated clique reduction (described in Section 4.2). The last reduction is used for a better worst case analysis and removes so called line graphs of four-regular graphs.

Akiba and Iwata [2] design and implement an exact branch-and-reduce algorithm for the minimum vertex cover problem where they use a superset of the kernelization rules covered in this thesis for the reduction step. They show that kernelization is actually an important step of their algorithm in order to solve large instances by testing their algorithm with different sets of reduction rules. We believe that by improving scalability of kernelization, even larger instances can be solved.

3.3 Local Search Algorithms

As all exact algorithms for MIS and the related problems run in exponential time, unless $P = NP$, heuristic algorithms have been studied extensively. *Local search* algorithms try to improve one (or many) initial solution and alter it in order to improve its quality. While these algorithms often do not give a guarantee for the quality of their solution, they can often find high quality solutions significantly faster than exact algorithms.

[1]The $\mathcal{O}^*$ notation hides polynomial factors, so for functions f and g, $f(n) \in \mathcal{O}^*(g(n))$ if $f(n) \in \mathcal{O}(g(n) \cdot poly(n))$ for some polynomial $poly(n)$.

There have been several local search algorithms published for the maximum clique problem. Grosso et al. [24] develop an algorithm called *Deep Adaptive Greedy Search (DAGS)*. In a first step they find large maximal cliques, starting at every vertex by a greedy search and swapping vertices that result in an equal sized clique. In the second step they then pick the most promising vertices from the first step and try to find large cliques including these vertices by running multiple iterations that add feasible vertices to the clique with a probability proportional to a vertex weight. This weight is decreased whenever a vertex is added to a clique in an iteration.

Pullan and Hoos [41] present *Dynamic Local Search - Max Clique (DLS-MC)* which alternately applies a greedy expansion of the current clique and a plateau search (that is, swapping vertices in the current clique with other vertices while keeping the clique size). In order to add some diversity, they employ a penalty to vertices that are added to the clique that decays over the iterations. Pullan [39] further improves this algorithm by using different phases, each with a different strategy of choosing vertices that are added (or swapped) into the clique, adding random vertices to the clique to increase diversity and adaptively choosing the rate at which the penalty decays. In [40] Pullan adapts this algorithm for maximum independent set and minimum vertex cover.

Cai et al. [16] introduce NuMVC, a local search algorithm for the minimum vertex cover problem. Their main idea is to remove a vertex from a candidate set and then search for a new vertex to add to the candidate set in a second step. This differs from most previous approaches that try to find a vertex in the candidate set and another one outside of the candidate set which are then exchanged. They also employ a vertex weighting scheme that pushes edges that have been uncovered for many iterations into the candidate set. In order to take into account recent iterations more than older iterations, they scale the current edge weights down if the average weight surpasses some threshold.

Andrade et al. [3] present a local search algorithm, *iterated local search*, that is often referred to as ARW, for finding large independent sets by repeatedly removing a vertex from a maximal independent set and replacing it with two other vertices, increasing the IS size by one each time they perform this operation. Using appropriate data structures, finding such a *swap* is possible in time linear in the number of edges. In order to find subsequent swaps, they maintain a list of candidate vertices that may be removed during a swap which ensures that a vertex is only considered for a swap again if there is a certain change in its neighborhood. To add diversity they randomly force vertices into the solution, causing other vertices to be removed from the solution.

There are many other local search algorithms for MIS and the related problems. For this thesis, however, we want to draw attention to a result that combines local search algorithms with kernelization techniques from exact algorithms. Dahlum et al. [18] improve the ARW algorithm by removing high degree vertices from the graph and using the isolated clique reduction, which we also use in this work, online during local search. In particular, whenever they add a vertex to the independent set using the ARW algorithm, they check whether that vertex is simplicial (which means that the isolated clique reduction can be applied to it) and has degree less than three and apply the reduction if possible. They report very high speedups for this algorithm (*OnlineMIS*) compared to other approaches including the basic ARW algorithm. They also present an algorithm (*KerMIS*) using a kernelization step, high degree vertex removal and the ARW algorithm which, in many cases, computes the largest independent set they find. Their results show that kernelization is an important preprocessing step to enable local search algorithms to solve even very large instances. However, kernelization is a slow preprocessing step. For example, on a graph with about $50 \cdot 10^6$ vertices $2 \cdot 10^9$ edges, KerMIS was only able to output a first independent set after 10^4 seconds, making OnlineMIS the more appealing algorithm. It is therefore an interesting research topic to speed up the kernelization step in order to make KerMIS competitive with OnlineMIS in terms of execution time as it gives higher quality results.

3.4 Evolutionary Algorithms

Evolutionary algorithms are inspired by nature's concept of *survival of the fittest*, *recombination* and *mutation*. They usually hold some set of candidate solutions, called the population, and combine them using crossover operations to produce offspring or mutate them to add diversity to the population.

There have been several evolutionary algorithms for MIS and the related problems. Algorithms like the ones described by Bäck and Khuri [6], Singh and Gupta [43] or Borisovsky and Zavolovskaya [12] use very little domain knowledge and thus do not achieve results as good as most local search algorithms.

Balas and Niehaus [8] describe an evolutionary algorithm for the maximum clique problem. In contrast to the algorithms mentioned above, they use domain knowledge for their crossover operation: They generate an offspring of two parent cliques, by searching for the largest clique in the subgraph induced by the union of the parent cliques. By using bipartite maximum matching algorithms this is possible in polynomial time.

Lamm et al. [31] develop an evolutionary algorithm (*EvoMIS*) for computing high cardinality independent sets by using graph partitioning for crossovers. They compute a node separator $V_1 \cup V_2 \cup S$ and combine two individuals by taking the vertices in the independent set in V_1 from one of the parents and in V_2 from the other parent. Using a node separator ensures that there are no edges between V_1 and V_2, hence the offspring is always a valid independent set. To make the offspring a maximal independent set they greedily add the vertices in S to the IS and then improve the result using the ARW algorithm by Andrade et al. [3]. In [32] Lamm et al. use EvoMIS as a subroutine of an algorithm (*ReduMIS*) that repeatedly computes the graphs kernel, runs EvoMIS and forces low degree vertices from the current independent set into the final solution. In their experiments, ReduMIS always finds an exact maximum independent set when its size is known and consistent sizes when the independence number is unknown. They use the same kernelization algorithm as KerMIS, so again, kernelization is a bottleneck.

3.5 Parallel Algorithms

As our main technique for speeding up kernelization is parallelization, we give a short overview of parallel algorithms in the scope of MIS and related problems. Probably the most popular parallel algorithm for computing maximal independent sets (not maximum) is the algorithm by Luby [34]: In each iteration a set X of vertices is randomly chosen from all vertices remaining in the graph. If any two vertices in this set are adjacent, the vertex with the lower degree is removed from X. X is then added to the intermediate independent set and $N[X]$ is removed from the graph. The algorithm terminates when there are no vertices remaining in the graph. Because it is independently decided whether a vertex is added to X, every step in this algorithm can be performed in parallel.

Xiang et al. [46] present a parallel branch and bound algorithm for the maximum clique problem in the MapReduce framework. They recursively partition the graph into subgraphs that they can solve independently. They stop the recursion when the estimated time to solve a subgraph falls under a predefined threshold. They prune a subgraph when an upper bound for the maximum clique in this subgraph, based on graph coloring, is below the largest clique found so far in addition to some other, simpler pruning rules. Using their algorithm they were able to solve maximum clique for the instance *C4000.5* from the second DIMACS implementation challenge [27] which is a randomly generated graph with 4000 vertices and and approximately $4 \cdot 10^6$ edges. Other algorithms found a clique of the same size for this instance before but they were the first to prove its optimality. The computations for this took 39 hours on a highly parallel system and they estimate a sequential running time of over one year.

4 Reductions

We now describe the reductions we use in this thesis. In this section we focus on the definitions and proofs of several key reductions for self containment and to help a better understanding of the techniques developed in this thesis. Later sections will explain how to apply them in parallel and other techniques to speed up kernelization. We cover the vertex fold reduction by Chen et al. [17] in Section 4.1, the isolated clique reduction found in the paper by Butenko et al. [14] in Section 4.2, the twin and unconfined reductions by Xiao and Nagamochi [47] in Sections 4.3 and 4.4, respectively, the diamond reduction by Akiba and Iwata [2] in Section 4.5 and the LP reduction by Nemhauser and Trotter [38] in Section 4.6. All proofs of the reductions can be found in the respective papers and are included here for completeness and deeper understanding.

4.1 Vertex Fold Reduction

If a vertex v has degree two and its neighbors u and w are nonadjacent, either v or both u and w can be in a maximum independent set of the graph but never v in combination with either one of u or w. The vertex fold reduction by Chen et al. [17] uses this property by combining these three vertices into one vertex, reducing the size of the graph by two.

Theorem 4.1. *In a graph $G = (V, E)$, let v be a vertex with $N_G(v) = \{u, w\}$ and let u and w be nonadjacent. We can build a new graph $G' = (V', E')$ with $V' = (V \setminus \{u, v, w\}) \cup \{x\}$, $E' = \{\{a, b\} \in E \mid a, b \in V'\} \cup \{\{x, a\} \mid a \in (N_G(u) \cup N_G(w)) \setminus \{v\}\}$. If x is in a maximum independent set I' of G', then $I = (I' \setminus \{x\}) \cup \{u, w\}$ is a maximum independent set of G. If it is not, then $I = I' \cup \{v\}$ is a maximum independent set of G.*

Proof. Both cases imply that $|I| = |I'| + 1$ so we have to prove that $\alpha(G) = \alpha(G') + 1$ and that the construction actually produces an independent set. If v is in a maximum independent set I of G then $I \setminus \{v\}$ is an independent set of G'. If it is not then at least one of v's neighbors is in I (since it is of maximum cardinality). If only one of v's neighbors $a \in N_G(v)$ is in I, $I \setminus \{a\}$ is an independent set of G'. Otherwise, both u and w are in I. In this case $I \setminus \{u, w\} \cup \{x\}$ is an independent set of G. This shows that $\alpha(G') \geq \alpha(G) - 1$.

If x is in a maximum independent set I' of G', then $(I' \setminus \{x\}) \cup \{u, w\}$ is an independent set of G. Otherwise, $I' \cup \{v\}$ is an independent set of G. It follows that $\alpha(G) \geq \alpha(G') + 1$. $\square$

Figure 4.1 illustrates the vertex fold reduction.

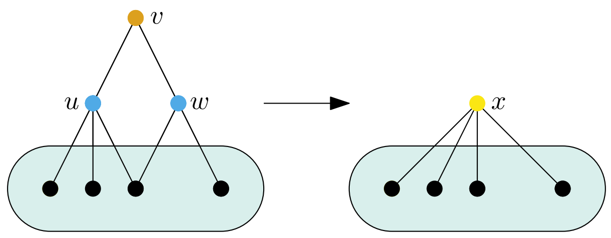


Figure 4.1: The vertex fold reduction: The vertices u, v and w are removed and a new vertex x is inserted.

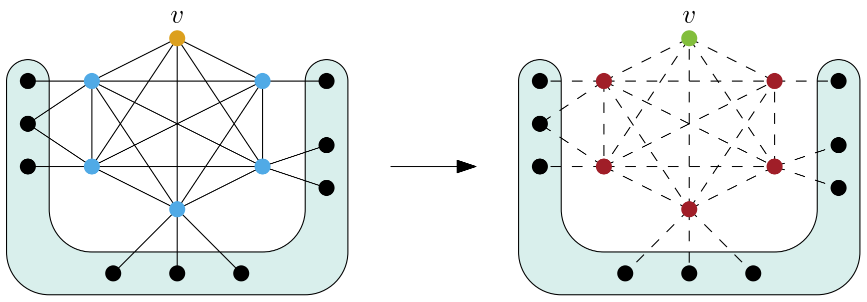


Figure 4.2: The isolated clique reduction: All of v's neighbors are pairwise adjacent, so v is inserted into the independent set and removed from the graph along with its neighbors. Green and red vertices as well as dashed edges are removed from the graph. Green vertices are added to, and red vertices are excluded from the independent set.

4.2 Isolated Clique Reduction

Butenko et al. [14] have shown the effectiveness of removing so called *isolated* cliques from the graph: If the subgraph induced by a vertex v and all its neighbors is a clique then there is always a maximum independent set containing v. Hence we can add v to the independent set and remove it and all its neighbors from the graph. We call v a *simplicial* vertex. More formally:

Theorem 4.2. *In a graph G, let v be a vertex with $\forall u \in N_G(v) : N_G[v] \subseteq N_G[u]$. Let G' be the graph obtained by removing $N_G[v]$ from G. For a maximum independent set I' of G', $I = I' \cup \{v\}$ is a maximum independent set of G.*

Proof. As none of v's neighbors is in G', it is easy to verify that $I' \cup \{v\}$ is an independent set of G for any maximum independent set I' of G', so $\alpha(G) \geq \alpha(G') + 1$. Similarly, $I \setminus \{v\}$ is an independent set of G' for any maximum independent set I of G, so $\alpha(G') \geq \alpha(G) - 1$. □

Figure 4.2 shows an example of the isolated clique reduction.

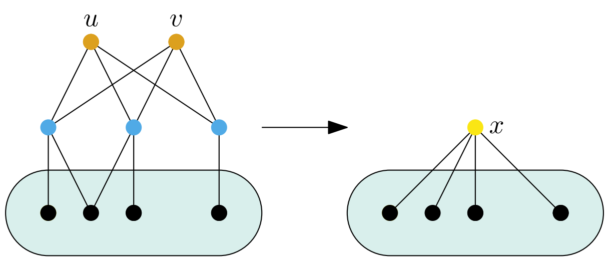


Figure 4.3: Twin reduction case 1: The three neighbors of u and v are not adjacent to each other, so we remove u, v and their neighbors and insert a new vertex x.

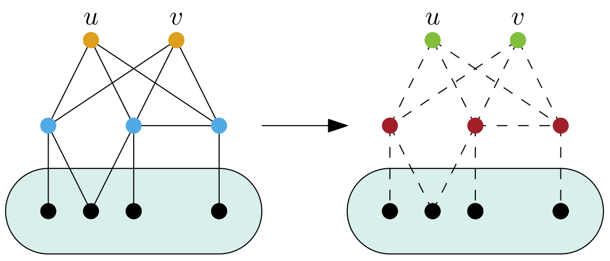


Figure 4.4: Twin reduction case 2: Two of the neighbors of u and v are adjacent so we add u and v to the independent set and remove $N_G[u] \cup N_G[v]$ from the graph. Green and red vertices as well as dashed edges are removed from the graph. Green vertices are added to, and red vertices are excluded from, the independent set.

4.3 Twin Reduction

The twin reduction by Xiao and Nagamochi [47] is a generalization of the vertex fold reduction. Two vertices u, v with degree three that have the same three neighbors are called twins. The reduction of twins is split up into two cases:

(i) If no pair of vertices in the neighborhood of v and u are adjacent either u and v can be in the independent set or $N(u)$ but never u or v in combination with any vertex in $N(v)$. We capture this in the reduction by replacing all five vertices with a new vertex.

(ii) If at least two neighbors of v and u are adjacent, any independent set can contain at most two of the vertices in $N(v)$ so we can just add u and v to the independent set and remove all five vertices from the graph.

We split the proof for these two cases:

Theorem 4.3. *In a graph $G = (V, E)$, let u, v be two nonadjacent vertices of degree three with $N_G(u) = N_G(v)$ and let the vertices in $N_G(u)$ be pairwise nonadjacent. We can build a new graph $G' = (V', E')$ with $V' = (V \setminus \{u, v\} \setminus N_G(u)) \cup \{x\}$ and $E' = \{\{a, b\} \in E \mid a, b \in V'\} \cup \{\{x, a\} \mid a \in N_G(N_G(u)) \setminus \{u, v\}\}$. Let I' be a maximum independent set of G'. If $x \in I'$, then $I = (I' \setminus \{x\}) \cup N_G(u)$ is a maximum independent set of G. Otherwise, if $x \notin I'$, $I = I' \cup \{u, v\}$ is a maximum independent set of G.*

Proof. As both cases imply $|I| = |I'| + 2$, we have to prove that $\alpha(G) = \alpha(G') + 2$ and the construction actually results in a valid independent set.

Let I be a maximum independent set of G. If $u, v \in I$, then $I' = I \setminus \{u, v\}$ is an independent set of G'. Note that if u/v is in some maximal independent set, then v/u is also in this set. If for some proper subset S of $N(u)$ (so $|S| \leq 2$), $S \subseteq I$ and $N_G(u) \setminus S \notin I$, then $I \setminus S$ is an independent set of G'. Otherwise, $N_G(u) \subseteq I$. In this case $(I \setminus N_G(u)) \cup x$ is an independent set of G'. It follows that $\alpha(G') \geq \alpha(G) - 2$.

Let I' be a maximum independent set of G. If $x \in I'$, then $(I' \setminus \{x\}) \cup N_G(u)$ is an independent set of G. Otherwise, $x \notin I'$. In this case $I' \cup \{u, v\}$ is an independent set of G. It follows that $\alpha(G) \geq \alpha(G') + 2$. □

Figure 4.3 illustrates this case.

Theorem 4.4. *In a graph $G = (V, E)$, let u, v be two nonadjacent vertices of degree three with $N_G(u) = N_G(v)$ and let some of the vertices in $N_G(u)$ be adjacent. We can build a new graph $G' = (V', E')$ with $V' = V \setminus N[v] \setminus \{u\}$, $E' = E \cap (V' \times V')$. Let I' be a maximum independent set of G', then $I = I' \cup \{u, v\}$ is a maximum independent set of G.*

Proof. Let I be a maximum independent set of G, then $I \setminus (\{u, v\} \cup N_G(v))$ is an independent set of G'. Any maximal independent set that contains u also contains v and does not contain any vertex from $N_G(v)$. Furthermore, if an independent set contains any vertex $w \in N_G(v)$, it cannot contain v or u and at most two vertices from $N_G(v)$ can be in any independent set. It follows that $\alpha(G') \geq \alpha(G) - 2$.

Let I' be a maximum independent set of G'. Because none of u's and v's neighbors is in G', $I' \cup \{u, v\}$ is an independent set of G. It follows that $\alpha(G) \geq \alpha(G') + 2$. □

We illustrate this case in Figure 4.4.

4.4 Unconfined Reduction

The unconfined reduction by Xiao and Nagamochi [47] proves that there is some maximum independent set that does not contain a given vertex v.

Definition 4.5 (Child). *A child of a set S is a vertex $u \in N(S)$ with $|N(u) \cap S| = 1$. The unique vertex $s \in S$ with $\{s, u\} \in E$ is called the parent of u.*

Theorem 4.6. *Let S be a set of vertices contained in* every *maximum independent set of a graph $G = (V, E)$. Then every maximum independent set of G contains at least one vertex from $N(u) \setminus N[S]$ for every child u of S.*

Algorithm 1: IsUnconfined

```
Input: A vertex $v$ and a graph $G = (V, E)$
Output: Whether $v$ is unconfined
1  $S \leftarrow \{v\}$
2  while True do
3      $u \leftarrow u \in N(S)$ such that $|N(u) \cap S| = 1$ and $|N(u) \setminus N[S]|$ is minimized
4      if $u =$ None then
5          return False                                   // See Figure 4.5b
6      if $N(u) \setminus N[S] = \emptyset$ then
7          return True                                    // See Figure 4.5c
8      if $N(u) \setminus N[S] = \{w\}$ then
9          $S \leftarrow S \cup \{w\}$                     // See Figure 4.5a
10     else
11         return False                                   // See Figure 4.5d
```

Proof. We prove the contrapositive: We assume there is a maximum independent set that does not contain any vertex from $N(u) \setminus N[S]$ for some child u of S and then conclude that there is some other maximum independent set which does not completely contain S.

Assume that there is a maximum independent set I of G with $I \cap (N(u) \setminus N[S]) = \emptyset$ for some child u of S. Then for the parent s of u, $I' = (I \setminus \{s\}) \cup \{u\}$ is a maximum independent set of G because $N(u) \cap (N[S] \setminus \{s\}) = \emptyset$. This contradicts that S is contained in every independent set of G because $s \in S$. □

We can use this theorem by starting with $S = \{v\}$ and assuming that S is a subset of every maximum independent set of G. We then repeatedly add vertices w with $N(u) \setminus N[S] = \{w\}$ for children u of S to S because they must also be in every maximum independent set of G. If we find a child u of S with $N(u) \setminus N[S] = \emptyset$, the assumption was false and we can remove v because there is a maximum independent set that does not contain v. We then say that v is *unconfined*. Algorithm 1 decides whether a vertex is unconfined. If a vertex v is unconfined, there always exists a maximum independent set that does not include v so we remove it from the graph.

4.5 Diamond Reduction

The diamond reduction by Akiba and Iwata [2] is an extension to the unconfined reduction. It uses the set S that is constructed for the unconfined reduction. Akiba and Iwata use it in their implementation but it is not explained in their paper. The definition and proof were kindly provided by Yoichi Iwata (slightly modified to be in the scope of MIS):

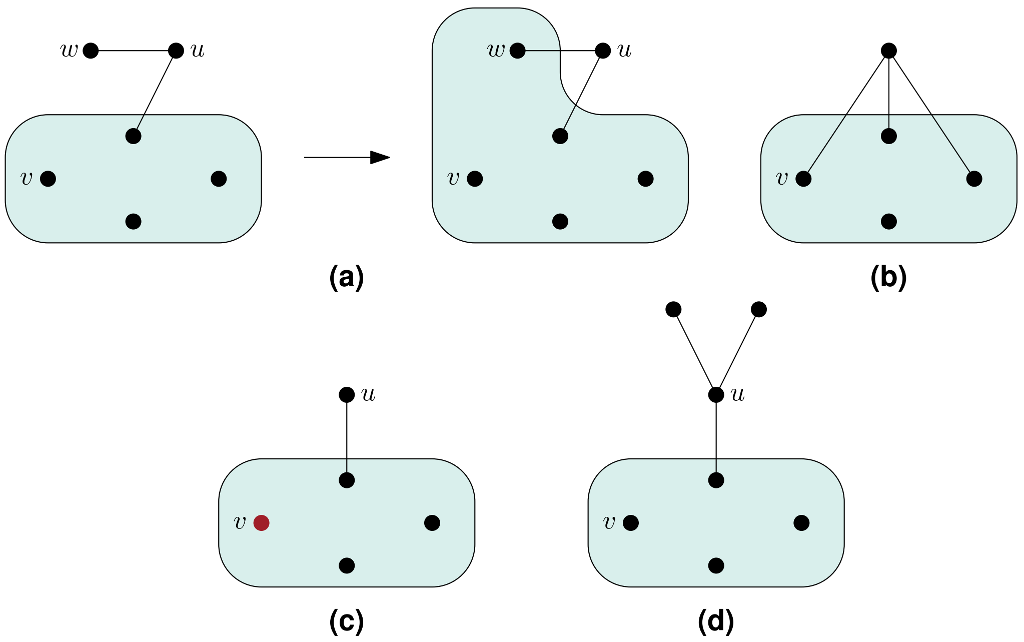


Figure 4.5: The unconfined reduction. The green box represents the set S. (a) There is a vertex $u \in N(S)$ such that $|N(u) \cap S| = 1$ and $N(u) \setminus N[S] = \{w\}$, so we add w to S. (b) There is no vertex $u \in N(S)$ such that $|N(u) \cap S| = 1$, so we cannot conclude that u is unconfined. (c) There is a vertex $u \in N(S)$ such that $|N(u) \cap S| = 1$ and $N(u) \setminus N[S] = \emptyset$. It follows that S is not contained in *every* maximum independent set, so u is unconfined and we can remove it from the graph. (d) There is a vertex $u \in N(S)$ such that $|N(u) \cap S| = 1$ but $[N(u) \setminus N[S]] > 1$. In this case we cannot conclude that v is unconfined.

Theorem 4.7. *Let S be the set constructed in the unconfined reduction for a vertex v that is not unconfined. If there are nonadjacent vertices u_1, u_2 in $N(S)$ such that $N(u_1) \setminus N(S) = N(u_2) \setminus N(S) = \{v_1, v_2\}$, then there exists a maximum independent set of G that does not contain the vertex v. (Note that the condition implies $v_1, v_2 \in S$)*

Proof. Assume that every maximum independent set of G contains v. Then, from the construction of S, it holds that $S \subseteq I$ for any maximum independent set I of G. Let $I' = (I \setminus \{v_1, v_2\}) \cup \{u_1, u_2\}$. This is also a maximum independent set and does not satisfy the condition $S \subseteq I$, which is a contradiction. Thus, there exists a maximum independent set that does not contain v. $\square$

Figure 4.6 illustrates a simple example of the diamond reduction.

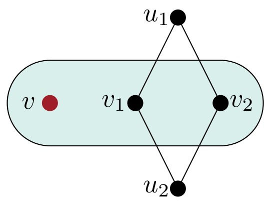


Figure 4.6: The diamond reduction. If a maximum independent set I contains the set S, then there is another maximum independent set $I' = (I \setminus \{v_1, v_2\}) \cup \{u_1, u_2\}$. It follows that there is a maximum independent set that does not completely contain S, so we can remove v from the graph.

4.6 Relaxed Integer Linear Program

The problem of finding a maximum cardinality independent set can be written as an integer linear program (ILP) as follows:

$$\begin{aligned} \text{minimize} \sum_{v \in V} x_v \text{ such that} && (4.6.1) \\ x_v \in \{0,1\} \text{ for } v \in V, && (4.6.2) \\ x_u + x_v \geq 1 \text{ for } \{u,v\} \in E, && (4.6.3) \\ x_v \geq 0 \text{ for } v \in V. && (4.6.4) \end{aligned}$$

If x_v is 0 in a solution then v is in the independent set.

This can be relaxed to a linear program (LP) by replacing constraint 4.6.2 with $0 \leq x_v \leq 1$.

It can be shown that

- There exists a solution such that each variable takes a value in $\{0, \frac{1}{2}, 1\}$ (Nemhauser and Trotter [37]).
- If a variable x_v takes a value in $\{0,1\}$, there always exists an integer solution where x_v takes the same value (Nemhauser and Trotter [38]).

Nemhauser and Trotter [38] also show that a solution with values in $\{0, \frac{1}{2}, 1\}$ can be found by reducing the LP to the bipartite matching problem as follows: We construct the bi-double graph $B(G) = \{L_V \cup R_V, E'\}$ with:

$$\begin{aligned} L_V &= \{l_v | v \in V\}, \\ R_V &= \{r_v | v \in V\}, \\ E' &= \{\{l_u, r_v\} | \{u,v\} \in E\}. \end{aligned}$$

From a minimum vertex cover C' of $B(G)$, we can find a solution with values in $\{0, \frac{1}{2}, 1\}$ for the LP:

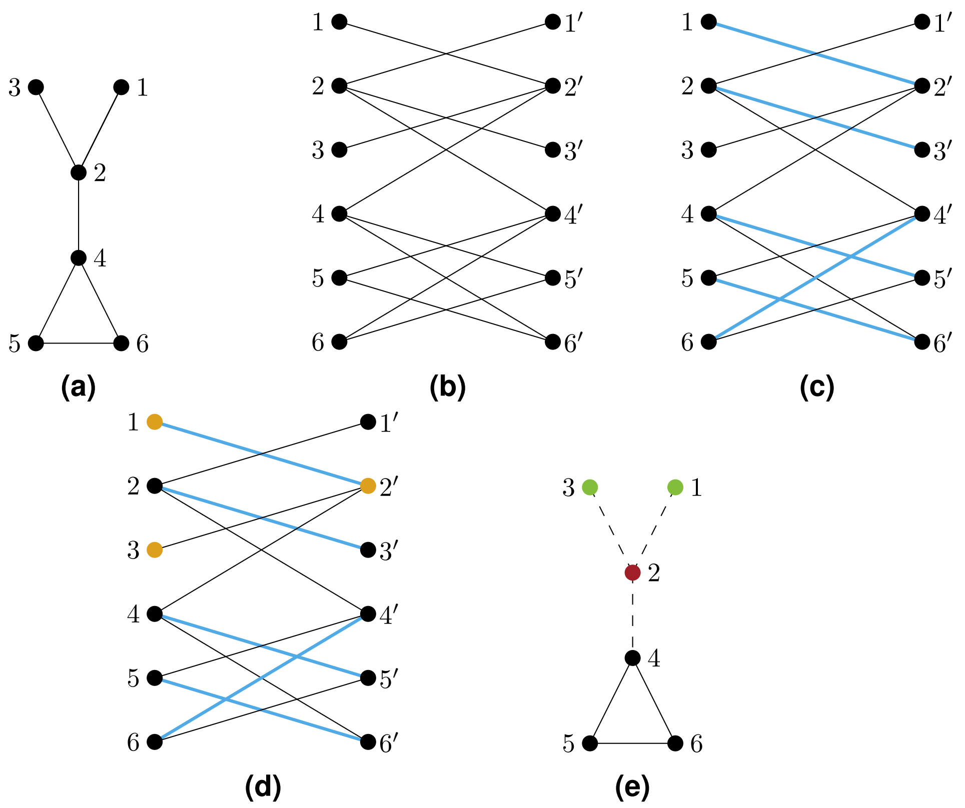


Figure 4.7: The LP reduction: (a) We start with an input graph G, (b) construct the bi-double graph $B(G)$ and (c) compute a maximum cardinality matching on it. (d) Next, we mark all vertices reachable on alternating paths starting at unmatched vertices on the left side. (d) We then use these markings to determine whether a vertex is in the independent set or not. As none of the vertices corresponding to $4, 5$ and 6 are reachable in the bipartite graph, they remain in the kernel. Green and red vertices as well as dashed edges are removed from the graph. Green vertices are added to, and red vertices are excluded from the independent set.

$$x_v^* = \begin{cases} 0 & \text{if } l_v, r_v \notin C', \\ 1 & \text{if } l_v, r_v \in C', \\ \frac{1}{2} & \text{otherwise.} \end{cases}$$

Note that we can find C' in linear time after finding a maximum cardinality bipartite matching on $B(G)$ by finding vertices that are reachable by alternating paths starting at unmatched vertices in L_V (König's theorem).

Algorithm 2 explains the LP reduction as pseudocode. Marking all vertices that are reachable on alternating paths is possible in linear time using a depth first search that stops when reaching a vertex that has already been visited, so the running time is dominated by the matching algorithm. A maximum matching in a bipartite graph can be computed by the

Hopcroft Karp algorithm [26] in $\mathcal{O}((m+n)\sqrt{n})$. An example application of this algorithm is shown in Figure 4.7.

Algorithm 2: The LP reduction

```
Input: A graph $G = (V, E)$
Output: A reduced graph $G'$ and an independent set $I$ (such that
        $I \cup MaximumIndependentSet(G')$ is a maximum independent set of $G$)
1  $M \leftarrow MaximumMatching(B(G))$
2  $R \leftarrow \emptyset$
3  for $l_v \in G \mid l_v$ is unmatched in $M$ do
4      $R \leftarrow R \cup VerticesOnAlternatingPaths(l_v, B(G), M)$   // Vertices reachable by
       alternating paths in $B(G)$ (with matching $M$) starting at vertex $l_v$
5  $G' \leftarrow G$
6  $I \leftarrow \emptyset$
7  for $v \in V$ do
8      if $l_v \in R \wedge r_v \notin R$ then
9          $I \leftarrow I \cup \{v\}$
10         $G' \leftarrow G' \setminus \{v\}$          // Remove $v$ from $V$ and all incident edges from $E$
11     else if $l_v \notin R \wedge r_v \in R$ then
12         $G' \leftarrow G' \setminus \{v\}$          // Remove $v$ from $V$ and all incident edges from $E$
13 return $G', I$
```

5 Dependency Checking

In this section we introduce a technique to prune vertices from the kernelization algorithm if the part of the graph close to it did not change recently. By this we reduce the amount of work for kernelization, especially in the later stages of the algorithm, where only a few reductions can be applied, scattered across the graph.

To compute a kernel, Akiba and Iwata [2] apply their reductions $r_1, \ldots, r_j$ as in Algorithm 3. The implementation by Strash [44] uses a scheme for checking dependencies between reductions for the isolated clique and vertex fold reductions. After unsuccessfully trying to apply a reduction to a vertex, one only has to consider this vertex again for reduction after its neighborhood has changed. So when the neighborhood of a vertex v has changed while applying a reduction to another vertex w, v is added to a set of vertices that should be considered for reduction. Initially all vertices in the graph are added to this set so that every vertex is considered at least once for reduction. We use this set for the vertex fold, isolated clique, and twin reductions.

In the following proofs we assume that none of the reductions adds or removes edges between vertices that are already in the graph. Put more formally: For every reduction r and every graph $G = (V, E)$, in $r(G) = G' = (V', E')$, $\forall u, v \in V \cap V' : \{u, v\} \in E \Leftrightarrow \{u, v\} \in E'$. This assumption is true for all reductions used in this thesis: The unconfined, isolated clique and LP reductions only remove edges by removing vertices from the graph, and the vertex fold and twin reductions remove edges by removing vertices from the graph and add edges to newly inserted vertices.

Theorem 5.1. *Let G be a graph and G' the graph after applying some reduction to G. Let v be a vertex that cannot be removed by a vertex fold reduction in G. If $N_G(v) = N_{G'}(v)$, the vertex fold reduction cannot be applied to v in G'.*

Proof. Assume that the vertex fold reduction can be applied to v in G', then $N_G(v) = N_{G'}(v) = \{u, w\}$. Furthermore, there must be an edge $\{u, w\} \in E'$ which implies $\{u, w\} \in E$. It follows that we can apply the vertex fold reduction to v in G which is a contradiction. □

Theorem 5.2. *Let G be a graph and G' the graph after applying some reduction to G. Let v be a vertex that is not simplicial in G. If $N_G(v) = N_{G'}(v)$, then v is not simplicial in G'.*

Proof. Assume that v is simplicial in G', then for all neighbors $u \in N_{G'}(v)$, $N_{G'}[v] \subseteq N_{G'}[u]$. However, this contradicts to v not being simplicial in G because $N_{G'}[v] = N_G[v]$ and we assume that no edge $\{u, w\}$ can be added for $w \in N(v)$. □

Algorithm 3: Kernelization as in [2]

```
Input: A graph $G = (V, E)$
Output: A reduced graph $G'$
$i \leftarrow 1$
do
    $G \leftarrow r_i(G)$                         // Apply $r_i$ to all vertices
    if $G$ changed by applying $r_i$ then
        $i \leftarrow 1$
while $i \leq j$
return $G$
```

Theorem 5.3. *Let G be a graph and $G' = r(G)$ the graph after applying some reduction to G. Let v, u be vertices that are not twins in G. If $N_G(v) = N_{G'}(v)$, v and u are not twins in G'.*

Proof. Assume that v and u are twins in G', then $N_{G'}(v) = N_{G'}(u)$. As this implies $N_G(v) = N_G(u)$, u and v are also twins in G. It follows that u and v cannot be twins in G'. □

For the unconfined and diamond reductions, this is not applicable because it might be possible to apply the unconfined reduction to a vertex even if its direct neighborhood did not change, but another vertex further away did. We can, however, add the neighbors of vertices removed by the unconfined reduction to the dependency checking set for better performance of the reductions that benefit from it. For the LP reduction, dependency checking is also not applicable because we have to consider the entire graph for finding a maximum bipartite matching. We can add vertices to the dependency checking set as in the unconfined and diamond reductions so that other reductions can benefit from it.

A possible speedup for our algorithm is to consider only cliques of small size (say, at most k) for the isolated clique reduction. This is done in the algorithm by Akiba and Iwata [2] for $k = 3$. In this case we can prune all vertices with degree greater than $\min(3, k)$ (we need vertices of degree two for the vertex fold reduction and vertices of degree three for the twin reduction) from the dependency checking set.

5.1 Reduction Order

The dependency checking scheme proposed in this section (and the parallelization introduced in the next section) changes the order in which reductions are applied to the vertices compared to other implementations like the one by Akiba and Iwata [2]. We want to investigate the impact the change of the order has on the kernel size. In a kernelization algorithm that applies just the isolated clique reduction, we can see that for every simplicial vertex v, $N[v]$ will always be removed from the graph independent from the order in which the reductions are applied. The reason is that v will always be simplicial, even if a vertex $u \in N(v)$ is removed by some other isolated clique reduction.

For other reductions, the case is not that simple. In Section 5.1.1 we will investigate a kernelization algorithm that uses just the isolated clique and vertex fold reductions, for which the order of reductions does not seem to change the kernel. In Section 5.1.2 we show that the order in which the vertex fold and unconfined reductions are applied *does* lead to different kernels.

5.1.1 Vertex Fold and Isolated Clique Reductions

We investigate the result of kernelization using just the isolated clique reduction and the vertex fold reduction. Empirical evidence suggests that independent from the order in which we apply the reductions, the size of the kernel does not change. We modify the code by Strash [44] to choose vertices from the dependency checking set at random and observe that the reductions are applied in a different order in every run of the algorithm. We perform ten runs on each graph from a set of ten graphs and the results of these experiments show that for all sequences of reductions that occurred, the kernel size was the same. The results can be found in Table A.1 in the appendix. We conjecture something stronger. That is, all kernels reachable by reordering the reductions are isomorphic.

Conjecture 5.4. *Let K_1, K_2 be two kernels reachable by different sequences of vertex fold and isolated clique reductions, then K_1 and K_2 are isomorphic.*

We make some first steps towards solving Conjecture 5.4 by considering local changes around reductions. We investigate each combination of two reductions that "conflict" with each other and show that isomorphic graphs *can* be reached after applying either of the two. However, in some of the cases, the application of either of the reductions does not immediately result in isomorphic graphs. In these cases another reduction is possible and has to be applied for the resulting graphs to be isomorphic (see for example Theorem 5.8). Before these additional reductions are applied, the graph could be changed by other reductions, which makes a complete proof of Conjecture 5.4 difficult - we do not prove it fully.

First, we consider a vertex v in a graph G on which we can apply the vertex fold reduction. Vertex v is not simplicial as the vertex fold reduction requires v's neighbors u and w to be nonadjacent. For the other two vertices that get removed by the vertex fold reduction (i.e., u and w), there are several cases which are identical for u and w, so w.l.o.g. we only consider u here.

Theorem 5.5. *Let v be a vertex that we can apply the vertex fold reduction on and u, w its neighbors. Let u have degree one (so u is simplicial). Let G_1 be the graph resulting from applying the vertex fold reduction to v and G_2 the graph resulting from applying the isolated clique reduction to u, then G_1 and G_2 are isomorphic.*

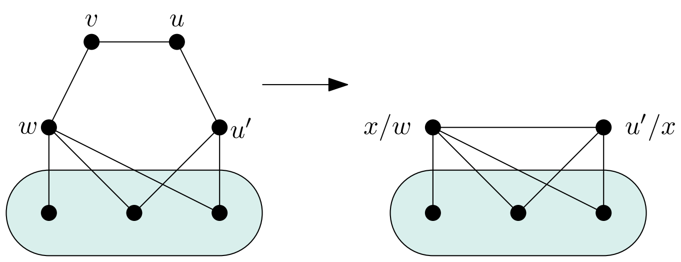


Figure 5.1: Applying conflicting vertex fold reductions. The reduced graphs when folding v or u first, are isomorphic.

Proof. Applying the isolated clique reduction would remove v and u from the graph and leave w in the new graph G_2. When applying the vertex fold reduction on v, the resulting graph after the reduction does not contain any of the vertices v, u, w and a new vertex x is inserted which is connected to the same neighbors as w in G_2. □

Next, we are going to consider graphs where a neighbor u of v has degree two. Note that u is not simplicial because $v \in N_G(u)$ cannot be adjacent to the other vertex in $N_G(u)$. We are going to consider the case that we apply the vertex fold reduction to u. The case where u's other neighbor is simplicial is covered in Theorem 5.7.

Theorem 5.6. *Let v be a vertex that we can apply the vertex fold reduction on and u, w its neighbors. Let u have degree two with $N_G(u) = \{u', v\}$. Let G_1 be the graph resulting from applying the vertex fold reduction to v and G_2 the graph resulting from applying the vertex fold reduction to u, then G_1 and G_2 are isomorphic.*

Proof. Let x_1 and x_2 be the new vertices in G_1 and G_2, respectively. Consider the graph G_1 obtained from applying the vertex fold reduction to v. Here, $N_{G_1}(x_1) = (N_G(w) \cup \{u'\}) \setminus \{v\}$ and $N_{G_1}(u') = (N_G(u') \setminus \{u\}) \cup \{x_1\}$. In G_2, $N_{G_2}(x_2) = (N_G(u') \cup \{w\}) \setminus \{u\}$ and $N_{G_2}(w) = (N_G(w) \setminus \{v\}) \cup \{x_2\}$. These graphs are isomorphic with the mapping from G_1 to G_2: $x_1 \mapsto w, u' \mapsto x_2$ and $a \mapsto a$ for all other vertices. □

An example for Theorem 5.6 can be found in Figure 5.1.

If u has degree greater than two, we cannot apply any reduction to it, as it cannot be simplicial because $v \in N_G(u)$ is not adjacent to any other vertex in $N_G(u)$. We have to consider that u can be removed from the graph by other reductions, changing the neighborhood of v. We split this into two cases covered in the following theorems.

Theorem 5.7. *Let v be a vertex that we can apply the vertex fold reduction on and u, w its neighbors. Let a vertex $u' \in N_G(u)$ be simplicial. Let G_1 be the graph resulting from applying the vertex fold reduction to v and G_2 the graph resulting from applying the isolated clique reduction to u', then we can reach isomorphic graphs G'_1, G'_2 by applying more reductions to G_1 and G_2, respectively.*

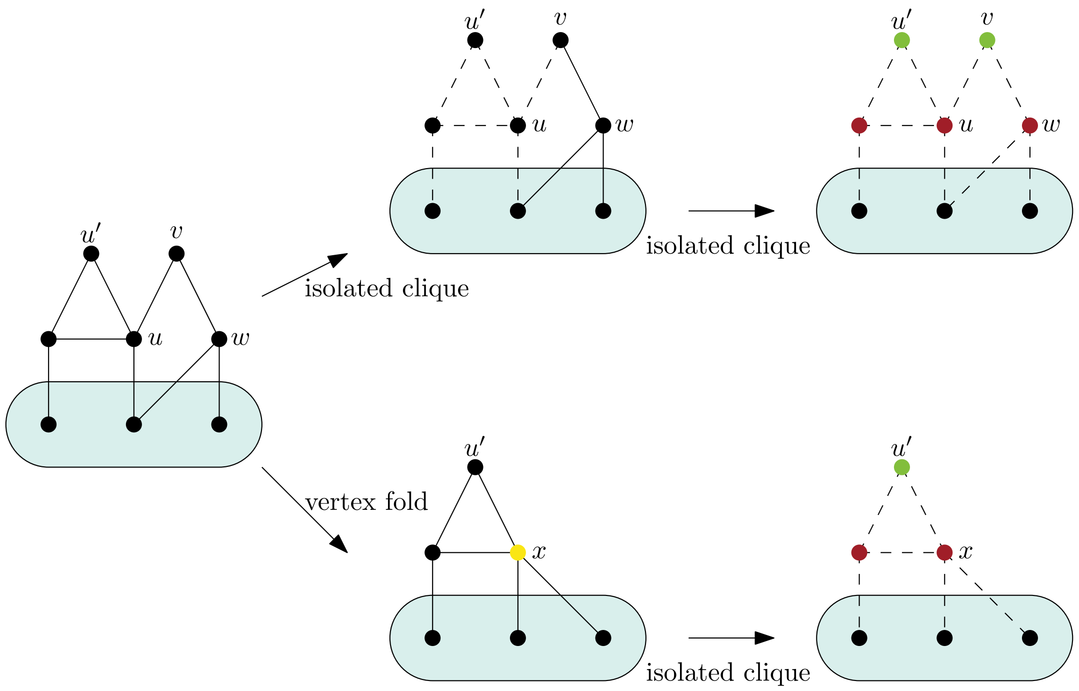


Figure 5.2: In the input graph on the left, the vertex u is part of a vertex fold and an isolated clique reduction. (Top): When applying the isolated clique reduction on vertex u' first, vertex v becomes simplicial and is reduced too. (Bottom): When applying the vertex fold reduction on vertex v first, we can still apply the isolated clique reduction on vertex u'. The resulting graphs are isomorphic. Green and red vertices as well as dashed edges are removed from the graph. Green vertices are added to, and red vertices are excluded from the independent set.

Proof. In G_1, u' is still simplicial, because $N_{G_1}(x) \cap N_{G_1}(u') = N_G(u) \cap N_G(u')$ and $x \in N_{G_1}(u')$, so $N_{G_1}[u'] \subseteq N_{G_1}[x]$ if $N_G[u'] \subseteq N_G[u]$. Let G_1' be the graph obtained by applying the isolated clique reduction on u' in G_1. The difference between G and G_1' is equivalent to removing $\{v, u, w\} \cup N_G[u'] = \{v, w\} \cup N_G[u']$ from the graph.

In G_2, v has degree one because u is removed from the graph. Let G_2' be the graph obtained from applying the isolated clique reduction to v in G_2. It is easy to see that G_2' is isomorphic to G_1' since vertices $N[u'] \cup \{v, w\}$ are removed from the graph. □

An example for Theorem 5.7 can be found in Figure 5.2.

Theorem 5.8. *Let v be a vertex that we can apply the vertex fold reduction on and u, w its neighbors. Let a vertex $u' \in N_G(u)$ be a candidate for a vertex fold reduction. Let G_1 be the graph resulting from applying the vertex fold reduction to v and G_2 the graph resulting from applying the vertex fold reduction to u', then we can reach isomorphic graphs G_1', G_2' by applying more reductions to G_1 and G_2, respectively.*

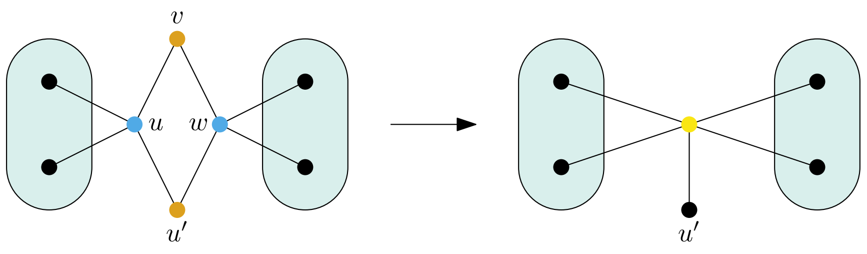


Figure 5.3: Two conflicting vertex fold reductions with same neighbors. Folding either of the vertices v or u' will result in a new vertex connected to u's and w's neighborhood and v/u' (depending on which vertex was chosen for the reduction).

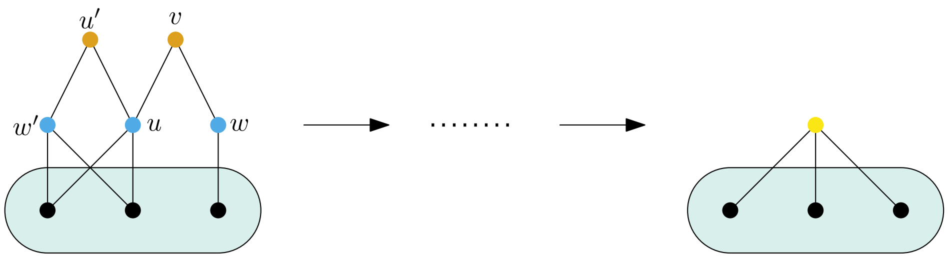


Figure 5.4: Two conflicting vertex fold reductions with nonadjacent neighbors. After applying both vertex fold reductions, the resulting graph is a new vertex connected to the neighbors of w', u and w.

Proof. Let $N(u') \setminus \{u\} = \{w'\}$. We differentiate between three cases:

(i) $w' = w$: In this case $N(v) = N(u')$. W.l.o.g. we apply the vertex fold reduction to v. The newly created vertex is connected to $(N(w) \cup N(u)) \setminus \{v\}$. See Figure 5.3 for an illustration.

(ii) w' and w are nonadjacent: Unaffected by the order, applying both reductions results in one vertex connected to the neighborhoods of w, u and w'. An example can be found in Figure 5.4.

(iii) w' and w are adjacent: Unaffected by the vertex we chose for the vertex fold reduction, the newly created vertex has neighbors u' (or v, depending on which vertex fold reduction is applied) and w' (or w) which are adjacent, so we can apply the isolated clique reduction on the new vertex. Figure 5.5 shows an example of this case.

□

Now we are going to consider a vertex v in a graph G that is simplicial. The interactions with the vertex fold reduction were already covered above. For all neighbors u of v there are two scenarios of how they could be removed from the graph by the isolated clique reduction covered in the following theorems.

Theorem 5.9. *Let v and $u \in N(v)$ be simplicial. Let G_1 and G_2 be the graphs obtained*

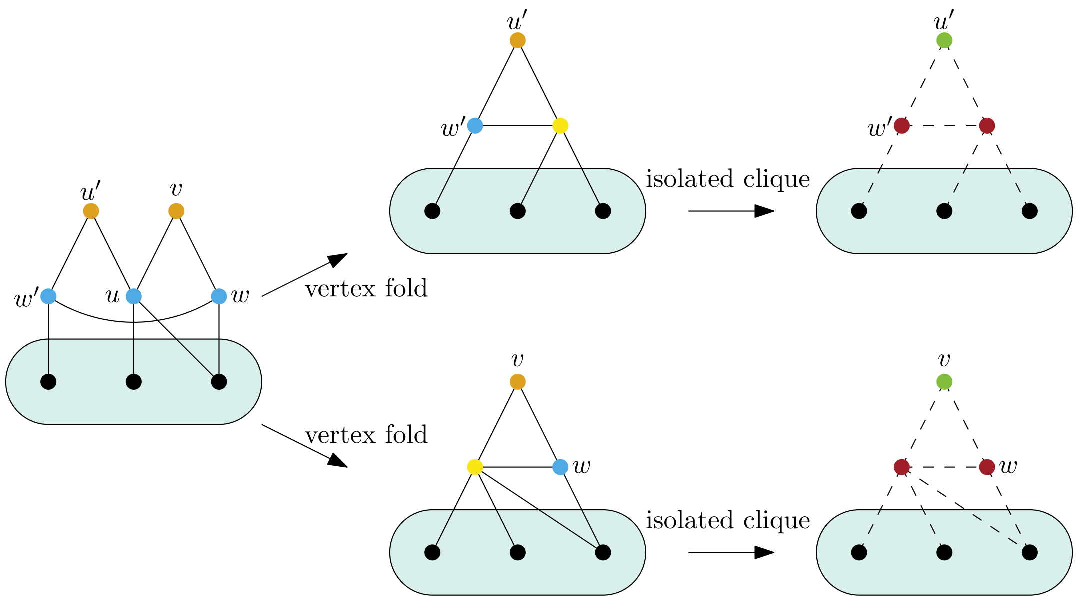


Figure 5.5: Two conflicting vertex fold reductions with adjacent neighbors. After applying the vertex fold reduction on vertex u' or v, the resulting graph is a clique with simplicial vertex v or u' which is removed from the graph by the isolated clique reduction. Green and red vertices as well as dashed edges are removed from the graph. Green vertices are added to, and red vertices are excluded from the independent set.

by applying the isolated clique reduction to v or u, respectively, then G_1 and G_2 are isomorphic.

Proof. The property that both v and u are simplicial implies that $N[u] = N[v]$ which means that the same set of vertices will be removed from the graph with u or v being added to the independent set, and all other vertices being excluded from the independent set. □

Theorem 5.10. *Let v and $w \in N(N(v)) \setminus \{v\}$ be simplicial. Let G_1 and G_2 be the graphs obtained by applying the isolated clique reduction to v or u, respectively. We can obtain isomorphic graphs G'_1, G'_2 by applying further reductions on G_1 and G_2, respectively.*

Proof. If $w \in N(v)$, see Theorem 5.9. If $w \notin N(v)$ we can apply both isolated clique reductions, resulting in $N[v] \cup N[u]$ being removed from the graph. □

We have now shown that for every combination of two "conflicting" isolated clique and vertex fold reductions, we *can* reach isomorphic graphs after applying either of them, possibly having to apply more reductions. Note that we also showed that the vertex fold and isolated clique reductions are only influenced by changes to their neighborhood in the beginning of this chapter. Some of the theorems used in this section require one more reduction r_i to be applied in order to reach isomorphic graphs. Now assume there is some other reduction r_j that can change the "neighborhood" of reduction r_i. We believe that it

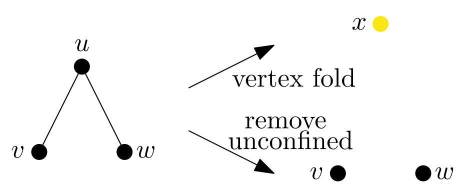


Figure 5.6: Example of the vertex fold reduction and the unconfined reduction leading to different kernel sizes. Vertex u is unconfined so it is removed. After removing u neither the unconfined reduction nor the vertex fold reduction can be applied to any of the two remaining vertices, thus the kernel size is 2. When applying the vertex fold reduction to u, the result is a single new vertex. The kernel size is 1.

can be shown that we *will* reach isomorphic graphs independent from the order we apply these in because we can inductively apply our theorems from this section.

5.1.2 Vertex Fold and Unconfined Reductions

In case we want to apply just the vertex fold and unconfined reductions, the order in which these two are applied can influence the kernel size. Figure 5.6 we can apply either the unconfined reduction or the vertex fold reduction to vertex u. After applying any of these reductions, no other vertex can be removed by either of the two reductions. Applying the unconfined reduction results in a larger kernel.

6 Parallel Framework

Our main tool for improving the performance of kernelization is parallelization. As reductions on one vertex can influence reductions on other vertices, a fine grained parallel algorithm where every vertex is processed separately seems hard to achieve. Furthermore, the number of synchronization steps for an algorithm that applies each reduction separately using a fine grained parallel algorithm would be very high when the reduced graph approaches a kernel. Which would limit the scalability of such an algorithm.

As the goal of graph partitioning is to split an input graph into blocks for parallel processing, using it seems promising. We partition the graph into p blocks and process each block on a different thread. While this means that we can apply most reductions on the inner vertices of a block in the same way as in sequential algorithms, we have to treat the vertices in the outer part of the blocks with special caution.

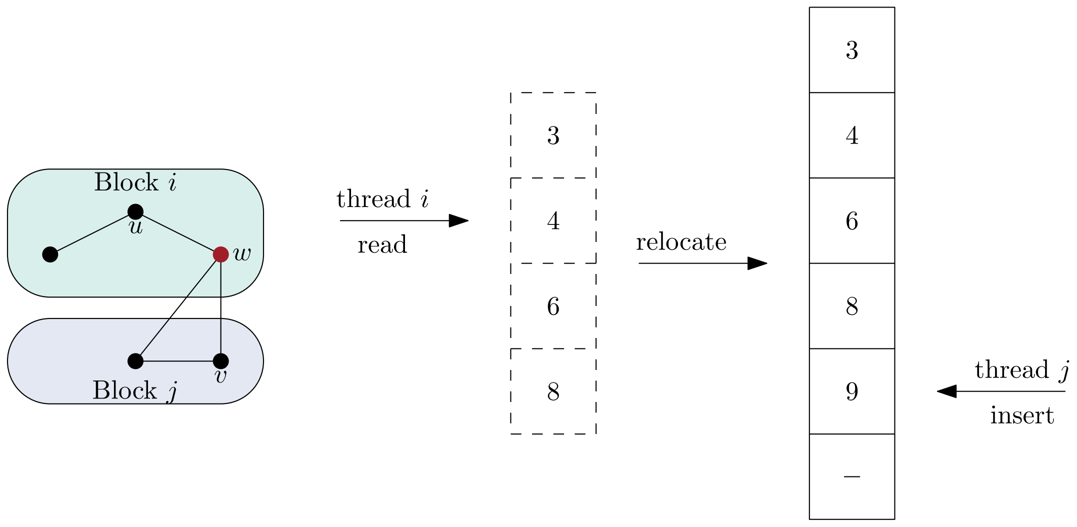
6.1 Data Structure

Our graph data structure represents vertices in a graph $G = (V, E)$ as integers $v = \{0, \ldots, |V|-1\}$ and edges using a two-dimensional array $neighbors$ such that $neighbors[v] = N(v)$. When adding a new edge $\{u, v\}$ to the graph, the size of the buffer containing $neighbors[v]$ might already be full, resulting in a reallocation of $neighbors[v]$ to a new buffer that is large enough to add u.

An array $inGraph$ contains information about which vertices are removed from the graph, so $inGraph[v] = true$ for vertices v that are still in the graph and $inGraph[v] = false$ for vertices v that have been removed from the graph. After a preliminary experimental evaluation, we decide to keep removed vertices in the $neighbors$ array of their neighbors, so when iterating over the neighborhood of a vertex, we have to check whether the neighbors are still in the graph and skip them if they are not. Removing the vertices from the $neighbors$ array of their neighbors causes a significant increase in running time.

In order to iterate over all vertices in a block of the partition $V_1, \ldots, V_k$, we hold an array $verticesInBlock$ such that $verticesInBlock[i] = V_i$. As we have to check which block contains a given vertex v, we also keep an array b such that $b[v]$ is the block that contains v.

Some of the reductions we use, make use of the degree of a vertex, which is expensive to obtain using just the array $neighbors$ because it contains vertices that are removed from the graph. In order to still have constant time access to the degree of a vertex, we keep an



(a) **(b)**

Figure 6.1: (a) The vertex fold reduction can be applied to vertex u and the isolated clique reduction can be applied to vertex v. Both would remove vertex w. We cannot apply the isolated clique reduction to vertex v without breaking Property (i). (b) Thread j inserts a new neighbor (9) to the neighborhood of some vertex. As the buffer holding the neighborhood is full, it is reallocated and the old buffer is freed. If thread i starts iterating over the array before the reallocation, it tries to access unallocated memory. This behavior is avoided by Properties (ii) and (iii).

array deg that holds the current degree for every vertex. As we have to determine whether a given vertex is a boundary vertex, we also keep an array $cutEdges$ such that for every vertex v, $cutEdges[v] = |N(v) \setminus b[v]|$. A vertex is a boundary vertex if $cutEdges[v] > 0$.

6.2 Reductions in the Parallel Framework

When applying a reduction to a vertex v in a graph $G = (V, E)$ to obtain a new graph $G' = (V', E')$, we define the following properties that must hold to avoid race conditions, allowing the application of our reductions without the use of locks:

(i) $\forall u \notin b[v] : u \in V \Leftrightarrow u \in V'$,

(ii) $\forall u \in V' : u \notin b[v] \Rightarrow N_{G'}(u) \subseteq N_G(u)$,

(iii) $\forall u \notin b[v]$: The reduction algorithm must not use the set $N(u)$.

We cannot remove vertices from different blocks in a single reduction, as this would result in race conditions when another thread tries to apply a reduction on the removed vertex, hence Property (i). An example can be found in Figure 6.1a. Property (ii) deals with a problem in the representation of the graph: Adding a new neighbor v to the $neighbors$ array of a vertex u (e.g., in the vertex fold reduction described in Sections 4.1 and 6.2.1)

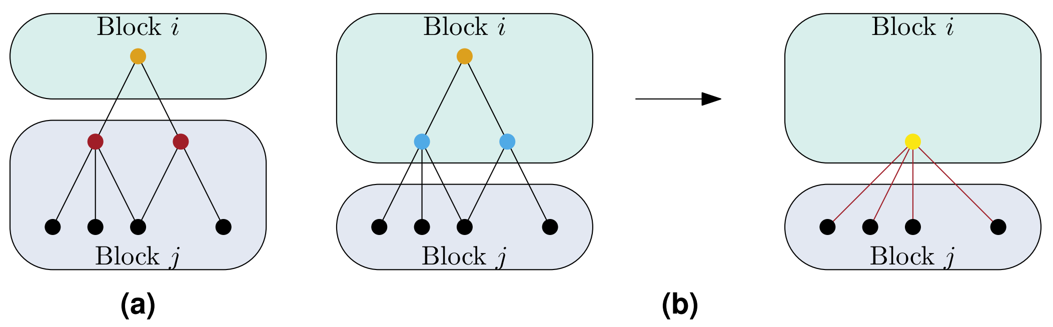


Figure 6.2: The vertex fold reduction in the parallel framework. The orange vertex is a candidate for the vertex fold reduction. (a) Its neighbors (red) are located in another block, so we cannot remove them from the graph without violating Property (i). (b) The orange vertex and its neighbors (blue) are located in the same block. However, when applying the vertex fold reduction, we have to connect the black vertices in block j to the new vertex (yellow) in block i by adding new edges (red). This would break Property (ii)

may result in a race condition when u and v are in different blocks of the partition: We have to add u to the list of v's neighbors. If another thread iterates over v's neighbors while a new neighbor is being added this can result in race conditions when there is not enough memory in v's buffer for storing neighbors and it has to be reallocated. Similarly, we cannot use the $neighbors$ array of a block j when processing a vertex in block i (see Property (iii)). The thread that processes block j can change or reallocate it while we iterate over its contents. Figure 6.1b illustrates these race conditions.

Furthermore, the algorithm must still be correct when a vertex $u \notin b[v]$ is removed from the graph. As these restrictions mean that most reductions cannot be applied to boundary vertices, it is essential for our algorithm to use a partition with a small cut. However, even with a small cut, we cannot apply all reductions that would be possible without these restrictions, so the result of our algorithm is not a kernel by definition. We call the reduced graph computed on the partitioned graph a *quasikernel*. We now describe how we apply each reduction in our parallel framework under the restrictions given above. We are going to refer to the initial graph as $G = (V, E)$ and to the reduced graph $r(G)$ and $G' = (V', E')$.

6.2.1 Vertex Fold Reduction

As connecting the newly created vertex x to v's two-neighborhood requires adding x as a neighbor to v's two-neighborhood $N(N(v))\setminus\{v\}$ we cannot apply the reduction to vertices whose two-neighborhood is not fully contained in the same block. For vertices whose two-neighborhoods are in the same block as the vertex itself, we can apply the reduction as described in Section 4.1.

Algorithm 4: ParallelVertexFoldReduction

```
Input: A vertex $v$ and a graph $G = (V, E)$
Output: A reduced graph $G'$
1  if $|N_G(v)| \neq 2$ then
2      return $G$
3  if $N_G(v) \notin b[v]$ then
4      return $G$
5  if $N_G(N_G(v)) \notin b[v]$ then
6      return $G$
7  if $\{u, w\} \notin E$ for $N_G(v) = \{u, w\}$ then
8      $V' \leftarrow (V \setminus N_G[v]) \cup \{x\}$
9      $E' \leftarrow \{\{a, b\} \in E \mid a, b \in V'\} \cup \{\{x, a\} \mid a \in N_G[N_G[v]] \setminus N_G[v]\}$
10     return $G' = (V', E')$
11 else
12     return $G$
```

Theorem 6.1. *The vertex fold reduction cannot be applied lock free to vertices v with $N[N[v]] \not\subseteq b[v]$.*

Proof. The vertex fold reduction removes $N_G[v]$ from the graph, so for v with $N_G[v] \not\subseteq b[v]$ Property (i) does not hold. It remains to show that the vertex fold reduction cannot be applied to vertices with $N_G(N_G(v)) \setminus \{v\} = N_G(u) \cup N_G(w) \not\subseteq b[v]$. As $\{\{x, a\} \mid a \in (N_G(u) \cup N_G(w))\} \subseteq E'$, Property (ii) is not fulfilled in this case. □

See Figure 6.2 for an illustration.

Theorem 6.2. *The vertex fold reduction can be applied lock free to qualified vertices v with $N[N[v]] \subseteq b[v]$.*

Proof. Similar to the proof above, we can show that Properties (i) and (ii) are fulfilled. Furthermore, Algorithm 4 does not use $N(u)$ for any vertex $u \notin b[v]$. □

6.2.2 Isolated Clique Reduction

As we cannot remove vertices from another block than v's block, we can only apply the isolated clique reduction on non-boundary vertices.

Theorem 6.3. *The isolated clique reduction cannot be applied lock free to vertices v with $N[v] \not\subseteq b[v]$.*

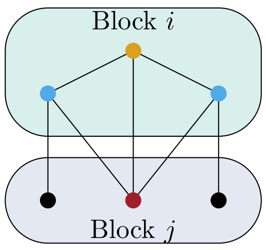


Figure 6.3: The isolated clique reduction in the parallel framework. The orange vertex is simplicial. However, one of its neighbors (red) is located in another block. We cannot apply the isolated clique reduction, which would remove the red vertex from the graph, without breaking Property (i)

Algorithm 5: ParallelIsolatedCliqueReduction

```
   Input: A vertex $v$ and a graph $G = (V, E)$
   Output: A reduced graph $G'$
1  if $N[v] \not\subseteq b[v]$ then
2      return $G$
3  for all $u \in N(v)$ do
4      if $N[v] \not\subseteq N[u]$ then
5          return $G$
6      return $G \setminus N[v]$          // Remove $N[v]$ from $V$ and all incident edges from $E$
```

Proof. The isolated clique reduction removes $N[v]$ from the graph, hence for $N[v] \not\subseteq b[v]$, violating Property (i). □

Figure 6.3 shows an example for this situation.

Theorem 6.4. *The isolated clique reduction can be applied lock free to qualified vertices v with $N[v] \subseteq b[v]$.*

Proof. As no vertex $u \notin N[v]$ is removed from the graph by the isolated clique reduction, Property (i) holds. Also, no edges are added to the graph, so $\forall u \in V' : N_{G'}(u) \subseteq N_G(u)$, so Property (ii) is fulfilled. Algorithm 5 holds Property (iii). □

6.2.3 Twin Reduction

The two cases of the twin reduction perform different actions on the graph, thus we have to consider each case separately in order to achieve a maximum number of reductions.

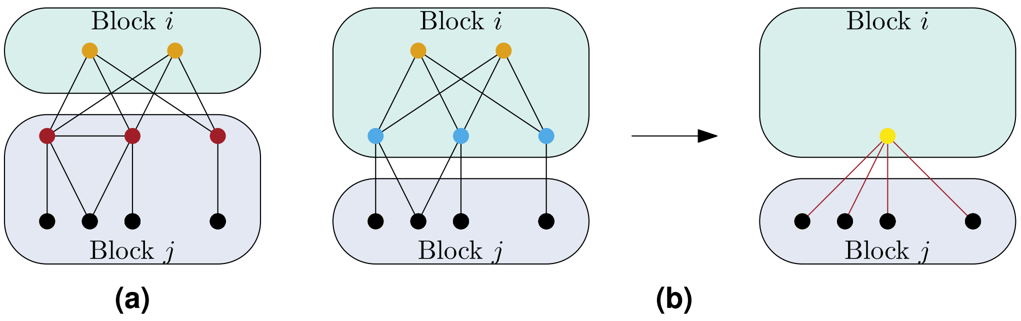


Figure 6.4: The twin reduction in the parallel framework. The orange vertices are twins. (a) The twin reduction always removes the twins and all their neighbors. In this example, the neighbors (red) of the twins are located in other blocks than the twins, hence we cannot remove them without breaking Property (i). (b) When applying case 1 of the twin reduction, we have to connect the new vertex (yellow) with its new neighborhood by adding new edges (red). However, doing so would not comply with Property (ii).

Case 1 is very similar to the vertex fold reduction. It can only be applied to vertices whose two-neighborhood $N[N[v]]$ is completely contained in the same block $b[v]$ because we can not connect the newly created vertex x to its new neighborhood if some of the neighbors are in different blocks.

Case 2 is similar to the isolated clique reduction. We can only remove vertices in a reduction that are in the same block, so we apply this case to all non boundary vertices.

Theorem 6.5. *Neither case of the twin reduction can be applied to vertices v and their twins u with $b[v] \neq b[u]$ or $N_G(v) \not\subseteq b[v]$.*

Proof. As $v, u \notin V'$ for both cases of the twin reduction, Property (i) does not hold if $b[v] \neq b[u]$. Also, both cases of the twin reduction remove $N_G[v]$ from G, so if $w \notin b[v]$ for some $w \in N_G(v)$, Property (i) does not hold. □

Figure 6.4a illustrates this situation.

Theorem 6.6. *We cannot apply case 1 of the twin reduction to twins v, u with $N_G[N_G[v]] \not\subseteq b[v]$.*

Proof. For $N_G(v) \not\subseteq b[v]$, see the theorem above. It remains to show that we cannot apply case 1 to twins v, u with $N_G(N_G(v)) \setminus N_G[v] \not\subseteq b[v]$. Case 1 of the twin reduction adds an edge (a, x) to E' for all $a \in N_G(N_G(v)) \setminus N_G[v]$, so Property (ii) is not fulfilled. □

See Figure 6.4b for an example.

Theorem 6.7. *Let v, u be twins with $b[u] = b[v]$. We can apply case 1 of the twin reductions if $N_G[N_G[v]] \subseteq b[v]$ and case 2 if $N_G[v] \subseteq b[v]$ without the use of locks.*

Algorithm 6: ParallelFindTwin

```
Input: A vertex $v$ and a graph $G = (V, E)$
Output: A twin $u$ of $v$
if $|N_G(v)| \neq 3$ then
    return None
if $N_G(v) \not\subseteq b[v]$ then
    return None
$w \leftarrow u \in N_G(v)$ such that $N_G(u)$ is minimal
for $u \in N_G(w) \cap b[v]$ do
    if $N_G(u) = N_G(v)$ then
        return $u$
return None
```

Algorithm 7: ParallelTwinReduction

```
Input: Twins $v, u$ such that $b[v] = b[u]$ and $N_G(v) \subseteq b[v]$, and a graph $G = (V, E)$
Output: A reduced graph $G'$
if $\forall a, b \in N_G(v) : \{a, b\} \notin E$ then        // Uses $N_G(w)$ for vertices $w \in N_G(v)$
    return $G \setminus (N_G[v] \cup \{u\})$ // Remove $N_G[v] \cup \{u\}$ from $V$ and all incident edges from $E$
else if $N_G(N_G(u)) \subseteq b[v]$ then
    $V' \leftarrow V \setminus (N_G[v] \cup \{u\})$
    $E' \leftarrow \{\{a, b\} \in E \mid a, b \in V'\} \cup \{\{x, a\} \mid a \in N_G(N_G(u)) \setminus \{u, v\}\}$
    return $G' = (V', E')$
```

Proof. It is easy to see that Properties (i) and (ii) hold in these cases. We can find a twin u for a vertex v (if one exists) such that $b[v] = b[u]$ and $N_G(v) \not\subseteq b[v]$ using Algorithm 6 which fulfills Property (iii). Property (iii) also holds for Algorithm 7 which applies the reduction, given twins v and u. □

6.2.4 Unconfined Reduction

The unconfined reduction can be applied to boundary vertices. However, the distance of vertices required for detecting an unconfined vertex is unbounded and iterating over the neighbors of a vertex u in another block will result in race conditions when the list of neighbors of u is modified. Algorithm 8 finds unconfined vertices without breaking Property (iii). In line 5 we only chose u if $b[u] = b[v]$ because we have to use $N(u)$ in the next steps. We also cannot return true if $N(u) \setminus N[S] = \emptyset$ for $b[u] \neq b[v]$ because u might get

Algorithm 8: ParallelIsUnconfined

```
Input: A vertex $v$ and a graph $G = (V, E)$
Output: Whether $v$ is unconfined
1  $S \leftarrow \{v\}$
2  while *True* do
3      $B \leftarrow \emptyset$                    // *Blacklisted vertices*
4      while *no vertex is added to* $S$ do
5          $u \leftarrow u \in (N(S) \cap b[v]) \setminus B$ such that $|N(u) \cap S| = 1$ and $|N(u) \setminus N[S]|$ is minimized
6          if $u =$ *None* then
7              return *False*
8          if $N(u) \setminus N[S] = \emptyset$ then
9              return *True*
10         if $N(u) \setminus N[S] = \{w\}$ then
11             if $b[w] = b[v]$ then
12                 $S \leftarrow S \cup \{w\}$
13             else
14                 $B \leftarrow B \cup \{u\}$
15         else
16             return *False*
```

removed from the graph by another thread. In line 11 we check whether w is in the same block as v because we have to use $N(s)$ for all $s \in S$, so we cannot add w to S if it is in a different block than v. Additionally, if w is in a different block than v, it could be removed from the graph by the thread responsible for $b[w]$. In the case that $b[w] \neq b[v]$, we try to find another vertex u by adding the current u to a black list. It is easy to check that all vertices for which Algorithm 8 returns true, are unconfined. However, it is possible that we falsely classify a vertex as not unconfined because we restrict the vertices u and the set S to $b[v]$. Figure 6.7 illustrates some of the situations where the parallel algorithm behaves differently from its sequential counterpart. If a vertex is unconfined, we remove it from the graph, which never breaks Properties (i) or (ii).

6.2.5 Diamond Reduction

We can apply the diamond reduction on both boundary and non-boundary vertices. However, since some vertices might not get inserted into S during the unconfined reduction, there might be u_1, u_2 such that $N(u_1)\setminus N(S) = N(u_2)\setminus N(S) = \{v_1, v_2\}$ and $\{v_1, v_2\} \not\subseteq S$ because they are located in different blocks, so we have to make sure that $v_1, v_2 \in S$. If

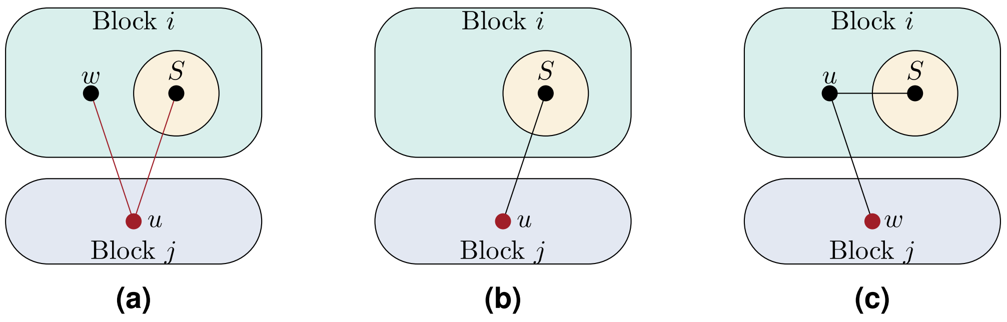


Figure 6.5: The unconfined reduction in the parallel framework. (a) We cannot add w to S because we cannot iterate over u's neighborhood to find w. (b) We cannot conclude that v is unconfined because u might be removed from the graph by a different thread. (c) We cannot add w to S because it might get removed from the graph by some other thread. We also cannot iterate over its neighborhood for future iterations.

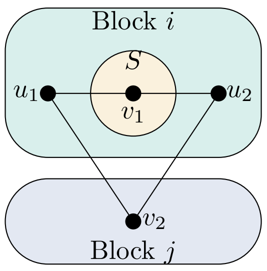


Figure 6.6: The diamond reduction in the parallel framework. In a sequential scenario, v_2 would be added to S by the algorithm for the unconfined algorithm. However, the parallel algorithm for the unconfined reduction does not add v_2 to S because it is located in a different block than v. As v_2 might get removed from the graph by another thread, we cannot apply the diamond reduction here.

they are not, they might be removed by another reduction which would lead to the diamond reduction being falsely applied. Figure 6.6 illustrates this situation.

Algorithm 9 finds diamond reductions while holding Property (iii). In Line 3, we can only consider $u_1, u_2 \in b[v]$ because we have to use their neighborhoods in the next step. In Line 4 we have to check if $v_1, v_2 \in S$ as explained above. It is easy to check that the algorithm only finds valid diamond reductions, but it might return false negatives when u_1, u_2, v_1 or v_2 are in different blocks than v. When the algorithm finds a diamond reduction on some vertex v, we simply remove it from the graph, which never violates Properties (i) or (ii).

Algorithm 9: ParallelDiamondReduction

```
Input: A vertex $v$, a set $S$ from the unconfined reduction, and a graph $G = (V, E)$
Output: Whether the diamond reduction can be applied to $v$
1 if $|S| < 2$ then
2     return *False*
3 for $u_1, u_2 \in N(S) \cap b[v]$ *such that* $u_1 \notin N(u_2)$ do
4     if $N(u_1) \setminus N(S) = N(u_2) \setminus N(S) = \{v_1, v_2\} \subseteq S$ then
5         return *True*
6 return *False*
```

6.3 Relaxed Integer Linear Program

The LP reduction is not applied on single vertices like the other reductions discussed in this thesis but uses a global view on the graph to find vertices to remove. When we apply the reduction to each block separately, the result is not guaranteed to be valid because adjacent boundary vertices can be fixed into the independent set (see Figure 6.7a). An idea that might come into mind is to use boundary vertices in the LP reductions of all adjacent blocks. This can also lead to incorrect kernels, for example vertices being excluded from the independent set that should be included, as it still lacks global knowledge of the graph. Figure 6.7b shows an example where a block incorrectly excludes a vertex from the fixed independent set. As there are scalable parallel maximum bipartite matching algorithms available, we modify Algorithm 2 to process the graph in parallel without requiring a partition:

Azad et al. [4] present a parallel augmenting path based algorithm for maximum bipartite matching. Augmenting path based algorithms usually start with a maximal matching, i.e. a matching where no edge can be added without removing some other edge from the matching. Then a path that starts and ends on an unmatched vertex is found that alternates between edges inside and outside of the matching. By removing all matched edges and adding all unmatched edges in this path to the matching, we increase the size of the matching by one. However, when these paths get long, it is hard to achieve good scalability in a parallel algorithm. Furthermore, it is desirable to start the search for augmenting paths from multiple source vertices in parallel. When starting the search at a single vertex and not finding any augmenting path in the search tree, the entire search tree can be pruned for future searches as the contained vertices and edges cannot be part of any augmenting path, even in future stages of the algorithm. When starting at multiple vertices, this does not hold. Azad et al. come around this issue by storing the disjoint search trees if they do not contain an augmenting path (so called *active trees*). When a search tree contains an augmenting path (a *renewable tree*), the matching is updated and relevant vertices connected to an active tree are added the respective active trees. Using this technique, they only have

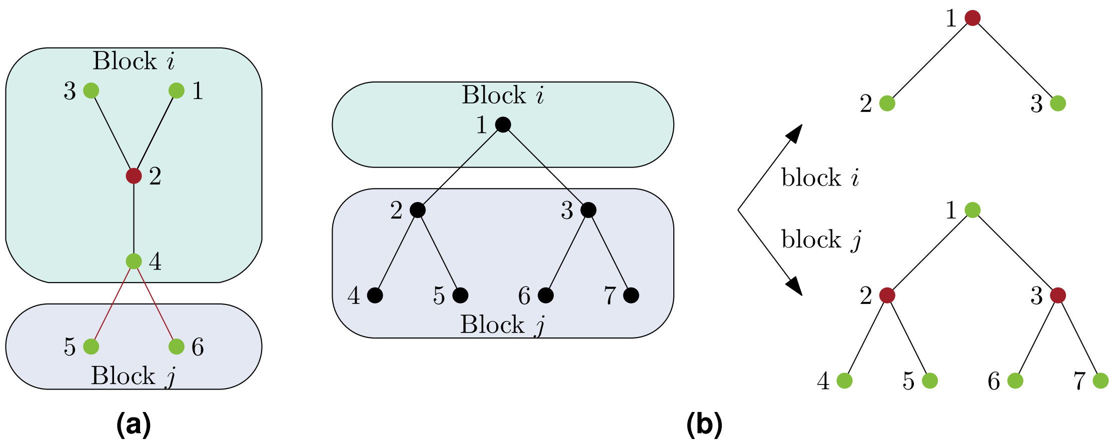


Figure 6.7: (a) We apply the LP reduction to blocks i and j independently. The result is incorrect, because vertices $4, 5$ and 6 are all fixed into the independent set. (b) We apply the LP reduction to block i and j and consider boundary vertices from other blocks. The result of block i leads to a wrong kernel because it excludes vertex 1 from the independent set, even though vertex 1 is part of the only maximum independent set of the graph.

to continue the search in the next phase from the leafs of active trees. As their algorithm uses a maximal matching as input, we start with the maximal matching algorithm by Karp and Sipser [28] which has been parallelized by Azad et al. [5].

We perform the loop in Lines 3 and 4 of Algorithm 2 in parallel for all unmatched vertices in L_V. We keep the search paths disjoint by pruning the search when we reach a vertex that has already been marked as visited by another search. The loop in Lines 7 through 12 is easy to perform in parallel and our graph data structure allows removing vertices in parallel. For better performance at repeated applications of the LP reduction, we reuse the parts of the previous matching which are still part of the graph. If the graph only changed slightly since the last application of the LP reduction, the matching from the previous application is still close to a maximum matching, which results in less work for the maximum matching algorithm.

6.4 The Final Algorithm

Our parallel algorithm using partitioning and dependency checking is described in Algorithm 10. We iterate over the blocks of the partition in parallel and keep a set of vertices that is used for dependency checking for each block separately. For each vertex in this set, we try to apply the isolated clique, vertex fold and twin reductions and add neighbors to the dependency checking set if a reduction was applied. After applying these reductions exhaustively, we apply the unconfined and diamond reductions to all vertices in the block. After applying the block-wise reductions, we synchronize the parallel loop and apply the

LP reduction. If the graph has changed, we go back to the block-wise reductions. During the unconfined, diamond and LP reductions, we add the neighbors of removed vertices to the respective set C_i. Because preliminary experiments gave faster reduction times when applying the LP reduction before the other reductions, we add an additional LP reduction before performing the block-parallel reductions.

Algorithm 10: Parallel kernelization algorithm using partitioning

```
   Input: A graph $G = (V, E)$
   Output: A reduced graph $G'$
1  $V_1, \dots V_k \leftarrow PartitionGraph(G, k)$
2  ApplyParallelLPReduction($G$)
3  for $V_i \leftarrow V_1$ to $V_k$ in parallel do
4      $C_i \leftarrow V_i$    // Set of candidates for the reductions. Used for dependency checking
5  while $G$ was changed in the last iteration do
6      for $V_i \leftarrow V_1$ to $V_k$ in parallel do
7          for $v \in C_i$ do
8              ApplyDependencyCheckingReductions($v$)    // Isolated clique, vertex fold
               and twin reductions
9          for $v \in G$ do
10             ApplyUnconfinedAndDiamondReductions($v$)
11     ApplyParallelLPReduction($G$)
12 return $G$
```

7 Experimental Evaluation

In this section we evaluate our algorithm and test it on a set of real-world instances. We show the impact of both, our dependency checking technique and parallelization by partitioning. We also compare our results to previous work to show an improvement on current implementations of kernelization.

In Table 7.1 we list the graphs we use in our evaluation. The web graphs are crawls restricted to some top level domain or a specific language. Road networks model the streets of a region. We obtain our graphs for the Laboratory for Web Algorithmics (LAW) [10, 11] and the 10th DIMACS implementation challenge [7]. Most of the web crawls from the LAW were crawled by UbiCrawler [9].

name	type	# vertices	# edges	source
arabic-2005	web	22.7 M	553.9 M	[9, 10, 11]
uk-2002	web	18.5 M	261.8 M	[7]
uk-2005	web	39.5 M	783.0 M	[9, 10, 11]
uk-2007-05	web	105.9 M	3,301.9 M	[9, 10, 11]
it-2004	web	41.3 M	1,027.5 M	[10, 11]
sk-2005	web	50.6 M	1,810.1 M	[9, 10, 11]
europe.osm	road	50.9 M	54.1 M	[7]

Table 7.1: The graphs used in our evaluation.

We implement our algorithm using C++ and compile it with gcc 4.8 using full optimizations (-O3). For shared memory parallelization we use OpenMP. We evaluate our implementation on a machine with 500 GB RAM and two Intel Xeon E5-2683 v4 processors with 16 cores each. The operating system is Ubuntu 14.04.5 LTS which runs Linux version 3.13.0-100-generic.

The structure of our experimental evaluation is as follows: In Section 7.1 we examine the impact of partitioning quality and load balancing measures on our algorithm. In Section 7.2 we evaluate the impact that the reductions used in our algorithm have on the execution time and the quasikernel size. In Section 7.3 we give results of our scaling experiments considering both the execution time and quasikernel size. Section 7.4 then closes our evaluation with a comparison with previous work. This includes the impact of our dependency checking technique as well as the speedup we obtain by running our algorithm in parallel.

7.1 Impact of Partitioning

As our algorithm uses partitioning for parallelization, we evaluate how to find a partition that helps the scaling of our algorithm without spending too much time on partitioning. We use the parallel graph partitioner by Meyerhenke et al. [36] for partitioning.

There are two factors we want to consider when evaluating partitioning for our algorithm:

(i) Kernelization time: Because we have to spend a different amount of work on every vertex, load balancing is an important factor. If many vertices requiring much work on are in the same block, the algorithm cannot scale well. We try to attack this problem by estimating the expected cost of processing a vertex and use these estimates as vertex weights for partitioning. The balance constraint should then cause a partition that is balanced in terms of computation time.

(ii) Size of the quasikernel: The size of our quasikernel is dependent on the boundary vertices because most reductions cannot be applied to these. We thereby expect a smaller quasikernel for more time consuming configurations of the partitioner.

A simple solution is to expect equal workloads for each vertex and thus use uniform vertex weights. As all reductions use at least the direct neighborhood of a vertex when processing it, we also evaluate the use of vertex degrees as vertex weights. In preliminary experiments, we also tried using the number of vertices reachable on paths of length two as weights, because most reduction algorithms also iterate over these vertices. In our experiments however, the weights of some vertices were overwhelmingly large so that the partitioner could not find a balanced partition.

In our experiments we use the three partitioner configurations *ultrafast*, *fast* and *eco* which usually compute decreasing cuts at the cost of increasing computation time. They mainly differ in the number of cycles used in their multilevel algorithm and the time used to find an initial partitioning. Additionally, we compare the results obtained from a partitioning by a sophisticated partitioner with those obtained from a much simpler sequential implementation of a label propagation algorithm (LPA) implemented by Christian Schulz. This algorithm computes a partitioning by starting with a random partition and then applying a size-constrained label propagation algorithm to improve the cut. The size-constrained label propagation algorithm has been taken from [35]. Overall, this partitioning algorithm is sequentially faster but does not provide as good solutions as the parallel partitioner.

We use all the reductions as explained in Chapter 6 but limit the isolated clique reductions to cliques with at most three vertices. In preliminary experiments we found that including larger cliques does not significantly change the quasikernel size.

We list the results of our experiments on partitioner configurations in Table 7.2. We run our full set of reductions on our benchmark graphs using 16 threads. As a preprocessing step, we partition the graphs into 16 blocks, also using 16 threads. The table shows the speedup in time and the relative change of the quasikernel size for each configuration compared to a baseline configuration. The kernelization time includes every reduction per-

		speedup for			
weight	configuration	partitioning	kernelization	total	quasikernel size
degree	eco	1.0	1.0	1.0	+0.0%
degree	fast	5.0	0.9	3.8	+1.7%
degree	ultrafast	**11.2**	1.0	**7.0**	+3.4%
degree	LPA*	5.0	0.9	4.2	+58.0%
uniform	eco	1.1	1.1	1.2	-0.3%
uniform	fast	5.9	1.0	4.4	+0.5%
uniform	ultrafast	**13.8**	1.0	**7.8**	+1.7%
uniform	LPA*	4.9	0.8	4.0	+59.9%

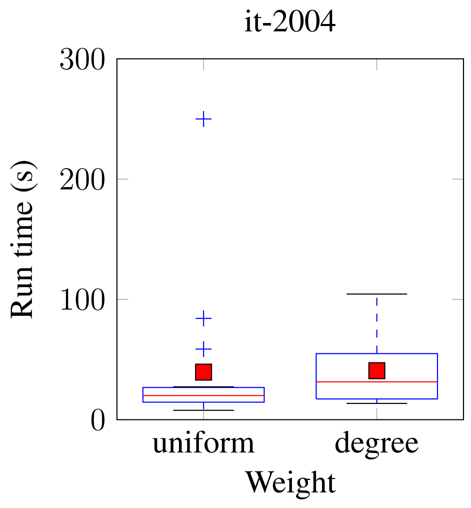
Table 7.2: Evaluation of different partitioner configuration on 16 threads. Speedups and change in quasikernel size with the eco configuration and vertex degrees as weights as baseline. Partitioning is the time for partitioning the graph into 16 blocks. Kernelization is the time spent on the block-wise parallel reductions and total is partitioning + kernelization. Results are averaged over all our benchmark graphs. Results for LPA are on a subset of our benchmark graphs.

formed in the block parallel section of our algorithm (recall that the LP reduction is not parallelized by partitioning).

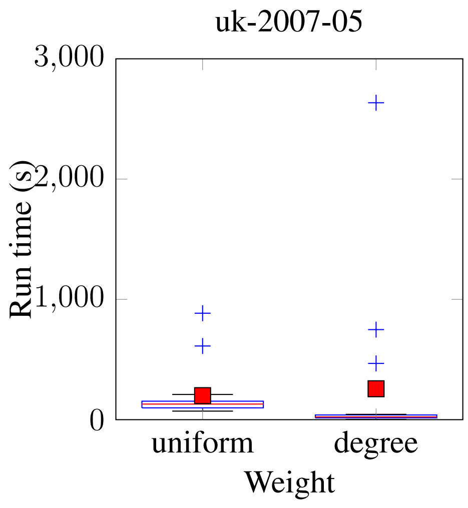
We could not run experiments using LPA for partitioning on graphs with more than 2^{31} edges due to the use of 32-bit integers in the partitioner. The quasikernel for europe.osm using a partition from LPA was more than a factor 70,000 larger than the baseline. As this largely outweighs the results from the other graphs, we decide to not include europe.osm in the results for a partitioning by LPA. The reason for this is that the structure of road networks causes a huge increase in quasikernel size with a low quality partitioning. For example, the vertex fold reduction, which is applied very often on road networks, is only applicable if a vertex does not have boundary vertices as neighbors. For the other graphs this effect is not as drastic but still significant. We conclude that a somewhat high quality partitioning is important for a small quasikernel size.

Due to the faster total times as a result of faster partitioning times, we decide to use the ultrafast configuration for further experiments. Using the fast or eco configurations results in slightly smaller kernel sizes and sometimes even faster kernelization times but this does not seem to justify the significantly higher partitioning time.

Our results also suggest that using uniform vertex weights leads to better load balancing on average, which is why we use this configuration for further experiments. However, these results are not consistent among all graphs. For some of our benchmark graphs, using vertex degrees as weights for partitioning leads to significantly shorter kernelization times. We provide sample plots for two of our benchmark graphs in Figure 7.1. The figure shows box plots of the kernelization time spent on every block for the graphs it-2004 (7.1a) and uk-2007-05 (7.1b). We can see that for it-2004, using vertex degrees as weights performs



(a)



(b)

Figure 7.1: Box plots of the running time per block using the ultrafast configuration.

significantly better than using uniform weights due to some blocks requiring much more time than the others. However, for uk-2007-05 we observe the opposite.

7.2 Impact of Reductions

We want to evaluate the impact of each reduction on our parallel kernelization. We run a set of experiments where we exclude each reduction from our algorithm and list the results in Table 7.3. As a base case, we use a configuration that is aimed to be comparable to the implementation by Akiba and Iwata [2]. As they remove degree zero and one vertices and remove isolated cliques of size three as part of their vertex fold reduction, we limit our isolated clique reduction to cliques of at most three vertices. We also list results for using isolated cliques of arbitrary sizes. As the diamond reduction needs the computation used for the unconfined reduction, we cannot exclude the unconfined reduction without excluding the diamond reduction as well. Note that the times reported here do not include the time for partitioning the graph.

We find that the LP, unconfined and vertex fold reductions largely contribute to kernelization: The quasikernel we achieve when excluding these reductions is significantly larger. Excluding the diamond or the twin reduction results in a quasikernel that is approximately 5% larger than the quasikernel computed by using all reductions and the kernelization time without these reductions does not change significantly. The difference in quasikernel size between the base case and the version without the isolated clique reduction is less than 1%. This can be explained by other reductions covering at least parts of the isolated clique reduction. For example, the LP reduction will always remove isolated vertices which are

configuration	reduction time	quasikernel size
all reductions	+0.0%	+0.0%
no LP	+52.1%	+250.0%
no diamond	+0.3%	+5.6%
no isolated clique	+7.9%	+0.2%
no twin	-4.1%	+4.1%
no unconfined, no diamond	-29.4%	+771.0%
no vertex fold	+28.5%	+1749.8%
arbitrary size isolated clique	+1.0%	-0.1%

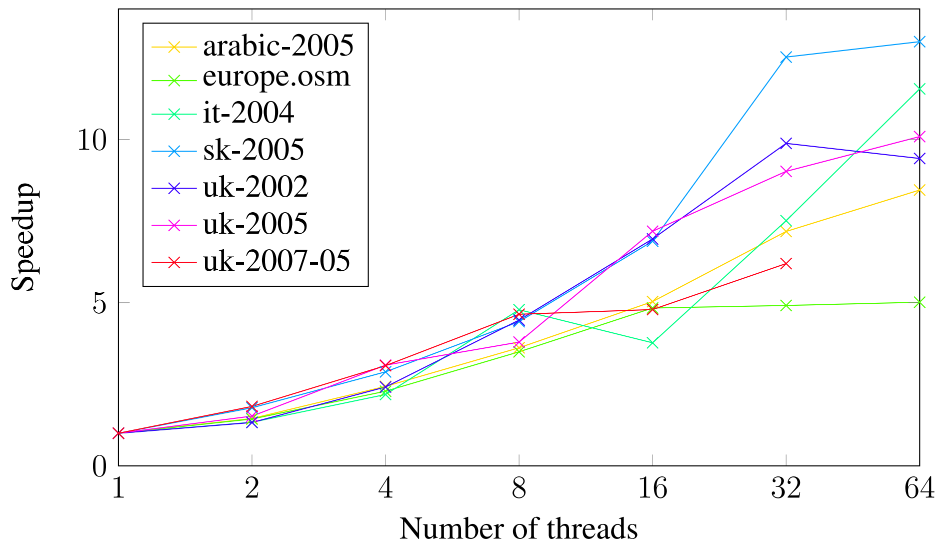
Table 7.3: We exclude each reduction from our parallel kernelization algorithm and list the change compared to the base case of using all reductions. Experiments are run on 16 threads and kernelization times do not include time for partitioning. Results are averaged over all our benchmark graphs.

always simplicial. We keep the isolated clique reduction in our algorithm because removing it increases kernelization time. Removing isolated cliques of arbitrary sizes does not significantly decrease the quasikernel size because large cliques are very rare in networks. However, because of the additional work required to check high degree vertices, the kernelization time is slightly higher than for the base case. We thus decide to limit the isolated clique reduction to cliques of size at most three for further experiments.

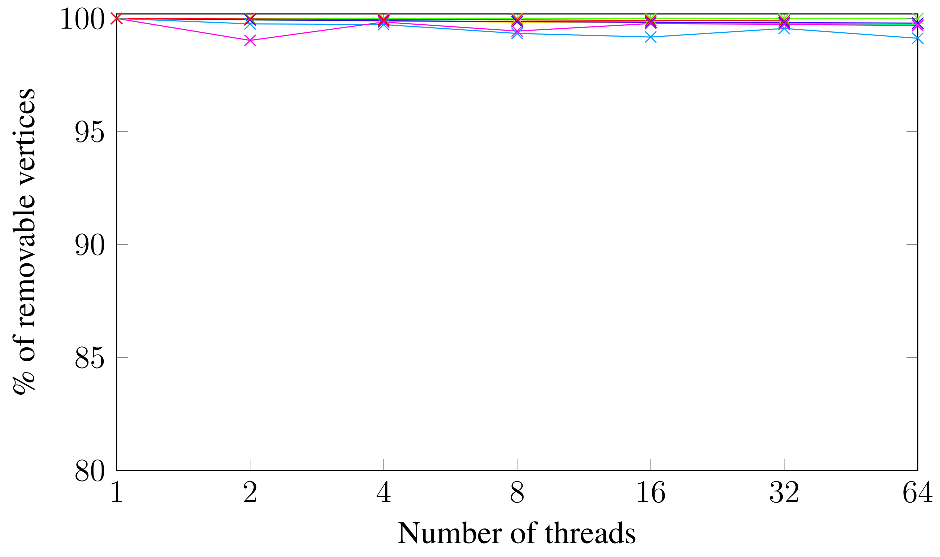
7.3 Scaling Experiments

We evaluate the scaling behavior of our algorithm on up to 64 threads on our benchmark graphs using a partitioning in p blocks for p threads. This shows the impact that parallelization has on our algorithm.

Figure 7.2a shows the speedup of our algorithms for $2^i, i \in \{0, 1, 2, 3, 4, 5, 6, 7\}$ threads (excluding the time for partitioning). The speedup for i threads is the time required by an algorithm sequentially divided by the time it takes using i threads. For our benchmark graphs, the fastest running time we achieve is typically about a factor 10 faster than the sequential running time (5 for the street graph). One reason for this is load imbalance: While some threads finish very fast, there are others that take considerably more time. As we already found imbalanced times per thread in Section 7.1, we expected similar behavior for other numbers of threads. It would be possible to bypass this by partitioning the graph into far more blocks than the number of threads used. Dynamic load balancing would then help improving the running time. However, this results in smaller quasikernel sizes due to more boundary vertices. Also, the parallel bipartite maximum matching algorithm used by the LP reduction shows a similar scaling behavior. In a micro benchmark we performed during our experimental evaluation on the instance europe.osm, we noticed that a simple program which iterates over the outgoing edges of every vertex shows a similar scaling



(a) Speedup



(b) Vertices removed

Figure 7.2: Strong scaling experiments on up to 64 threads. (a) The speedup of our algorithm compared to our algorithm on one thread. (b) The number of vertices removed by our algorithm relative to one thread.

behavior on our benchmark machine to our algorithm.

Figure 7.2b shows the number of vertices removed in the strong scaling experiments. Even though our algorithm does remove fewer vertices in parallel, this decrease is very minor. We still remove more than 99% of the removable vertices in parallel. Note that using only one block (and thus one thread), our algorithm finds an exact kernel.

7.4 Comparison with Previous Work

We compare our implementation to the one by Akiba and Iwata [2] written in Java. The code they published computes a minimum vertex cover but it can easily modified to stop after kernelization. As they implement a larger set of reduction rules, adding the desk and funnel reductions by Xiao and Nagamochi [47], we expect them to achieve smaller kernel sizes. However, as they only apply a scheme similar to our dependency checking for the removal of degree zero and one vertices, we expect faster kernelization times.

Table 7.4 shows our comparison between their and our implementation running single-threaded. For a more fair comparison, we also supply the time their algorithm takes to find a graph size equal our kernel. More specifically, we log the time and graph size after the application of every reduction and give the time when their algorithm first outputs a size smaller than (or equal to) our final size. We find that our algorithm finishes significantly faster than their implementation on all benchmark instances. Also, the time their algorithm takes to find a graph of size equal to our kernel is significantly longer than our kernelization time. We attribute this to our dependency checking technique. Their algorithm tries to apply every reduction to every vertex left in the graph until no more reductions can be applied. In contrast, our algorithm prunes a large part of the vertex set from further application of reductions because their neighborhood did not change since the last attempt of applying reductions. For the instance europe.osm, this is most notable. This can be explained by the structure of road networks. Because road networks have many degree two vertices, the vertex fold reduction can be applied very often, which is significantly sped up by dependency checking.

Table 7.5 compares the implementation of Akiba and Iwata with a fully parallel run of our algorithm. Again, for a more fair comparison we also compare our execution time to the time their implementation takes to output a smaller size than our algorithm in Table 7.6. As expected, the size of our quasikernel is higher than the exact kernel from the single-threaded experiment. For most of the runs we are able to execute our algorithm on 64 threads, we observe a faster kernelization by a factor of 10 to 30. Also, compared to the time their implementation takes to achieve a size equal to our quasikernel size, we can achieve significantly faster times using parallelization. We also observe that partitioning the graph makes up for a significant part of our algorithm for the smaller instances. For our biggest instance, partitioning does only slightly contribute to the overall running time.

As it might be important for some algorithms to find an exact kernel instead of our

	Akiba and Iwata [2]		Our algorithm			Same size comparison	
graph	size	time [s]	size	time [s]	speedup	time [s]	speedup
arabic-2005	574,878	1,033	610,408	359	2.9	655	1.8
europe.osm	8,366	302	14,044	47	6.4	218	4.6
it-2004	1,602,560	6,749	1,645,597	2,264	3.0	4,892	2.2
sk-2005	3,200,806	10,010	3,255,991	2,509	4.0	6,724	2.7
uk-2002	241,517	337	255,458	108	3.1	253	2.3
uk-2005	835,480	541	854,511	602	0.9	337	0.6
uk-2007-05	3,514,783	18,829	3,632,844	9,285	2.0	13,326	1.4

Table 7.4: Sequential comparison with the implementation by Akiba and Iwata. We give the kernel size and the kernelization time for both implementations. For better comparison we also give the time when their algorithm first found a graph smaller or equal than our kernel. Speedup columns are their time divided by our time.

	Akiba and Iwata [2]		Our algorithm				
graph	size	kern. [s]	size	part. [s]	kern. [s]	total [s]	speedup
arabic-2005	574,878	1,033	667,649	15	43	58	17.8
europe.osm	8,366	302	16,949	13	9	22	13.8
it-2004	1,602,560	6,749	1,762,705	44	198	243	27.8
sk-2005	3,200,806	10,010	3,967,500	116	235	351	28.5
uk-2002	241,517	337	293,341	23	12	35	9.7
uk-2005	835,480	541	978,740	65	64	129	4.2
uk-2007-05*	3,514,783	18,829	3,733,325	59	1,805	1,864	10.1

Table 7.5: Parallel comparison with the implementation by Akiba and Iwata with a quasikernel. We give the kernel size and kernelization time for the implementation by Akiba and Iwata. For our implementation, we give the size of the quasikernel, the time for partitioning, the kernelization time and the sum of partitioning and kernelization times. We ran our algorithm on 64 threads, using a partition into 64 blocks. For graphs marked with a star (*), we ran our algorithm on 32 threads using a partition into 32 blocks due to memory limitations.

quasikernel, we do an experiment where we find an exact kernel by an additional sequential run on the quasikernel. Because the LP reduction does find exact results, we still perform it in parallel. Tables 7.7 and 7.8 show the results for these experiments. We find that the speedup we obtain over Akiba and Iwata for finding a graph of our kernel size is actually better than for the quasikernel on most instances. This can be explained by the dependency checking that speeds up our sequential reductions, as well as the LP reduction that we perform in parallel.

		[2]	Our algorithm			
graph	size	kern. [s]	part. [s]	kern. [s]	total [s]	speedup
arabic-2005	667,649	361	15	43	58	6.2
europe.osm	16,949	202	13	9	22	9.2
it-2004	1,762,705	2,412	44	198	243	9.9
sk-2005	3,967,500	2,356	116	235	351	6.7
uk-2002	293,341	149	23	12	35	4.3
uk-2005	978,740	128	65	64	129	1.0
uk-2007-05*	3,733,325	11,828	59	1,805	1,864	6.3

Table 7.6: Comparison of our quasikernel with the time the implementation by Akiba and Iwata takes to find a graph of equal size. We give the kernel size used and times for both implementations. The experimental setup is the same as in Table 7.5.

	Akiba and Iwata [2]		Our algorithm				
graph	size	kern. [s]	size	part. [s]	kern. [s]	total [s]	speedup
arabic-2005	574,878	1,033	610,461	15	60	75	13.8
europe.osm	8,366	302	14,293	13	10	23	13.2
it-2004	1,602,560	6,749	1,651,441	44	332	376	18.0
sk-2005	3,200,806	10,010	3,260,217	116	376	492	20.3
uk-2002	241,517	337	255,514	23	15	38	8.9
uk-2005	835,480	541	854,687	65	74	138	3.9
uk-2007-05*	3,514,783	18,829	3,629,210	59	1,967	2,025	9.3

Table 7.7: Parallel comparison with the implementation by Akiba and Iwata with an exact kernel. All columns are the same as in Table 7.5. We ran our algorithm on 64 threads, using a partition into 64 blocks. After the parallel quasikernel computation, we find an exact kernel by a sequential run on the quasikernel (the LP reduction was performed in parallel). For graphs marked with a star (*), we ran our algorithm on 32 threads using a partition into 32 blocks due to memory limitations.

		[2]	Our algorithm			
graph	size	kern. [s]	part. [s]	kern. [s]	total [s]	speedup
arabic-2005	610,461	655	15	60	75	8.7
europe.osm	14,293	210	13	10	23	9.2
it-2004	1,651,441	4,376	44	332	376	11.6
sk-2005	3,260,217	5,781	116	376	492	11.7
uk-2002	255,514	250	23	15	38	6.6
uk-2005	854,687	323	65	74	138	2.3
uk-2007-05*	3,629,210	14,543	59	1,967	2,025	7.2

Table 7.8: Comparison of the exact kernel found by our algorithm with the time the implementation by Akiba and Iwata takes to find a graph of equal size. All columns are the same as in Table 7.6, the experimental setup is the same as in Table 7.7.

8 Discussion

8.1 Conclusion

We presented an algorithm that significantly speeds up kernelization for the maximum independent set problem. We believe this can be used to make large real-world instances more feasible to solve by speeding up initialization phases of algorithms that first kernelize the input graph.

Our first technique for speeding up kernelization - dependency checking - was shown to be effective in practice, achieving a speedup of up to 4.6 over previous kernelization algorithms. While dependency checking gave good results for some reductions, we currently do not know how to apply this technique to all reductions.

Our second technique - parallelization by partitioning - also sped up kernelization significantly. However, due to an overhead for partitioning the graph as a preprocessing step and load balancing issues, we found the scalability to be limited. While improving partitioning algorithms is out of the scope of this work and using lower quality partitionings has been shown to be ineffective, it remains open how load balancing can be improved.

Overall, speeding up kernelization for the maximum independent set problem is an interesting and promising step to improving the scalability of maximum independent set algorithms. While working on this thesis, we got new insights into the behavior of the reduction techniques and developed more ideas to scale up algorithms that can be an entry point for future work.

8.2 Future Work

A factor that is limiting the scalability of our implementation is load balancing. In fact, for all graphs and partitioner configurations, we find one block of the partition that has a significantly longer running time than all other blocks. It would be interesting to seek methods to better balance the work in each block. One possible solution for this can be to find a better weight to use for partitioning. This would include finding a fast-to-find estimate of how much work is spent on each vertex during the kernelization algorithm.

The problem of load balancing might also be eliminated by designing a fine grained parallel kernelization algorithm that does not require a partitioned graph. While we already parallelized the LP reduction without the need of a partitioned graph, we believe that this is also possible for at least some other reductions. We already developed ideas for the isolated clique reduction that seemed promising in preliminary experiments and for the unconfined reduction.

The dependency checking scheme used in this thesis is limited to the isolated clique, vertex fold and twin reductions. While the unconfined reduction is much more complex, it might be possible to apply some kind of dependency checking to it. Furthermore, it might be possible to use a specialized dependency checking scheme for every reduction, which needs further investigation. A simple example is that there is no need to attempt applying the vertex fold reduction to vertices with a degree other than two.

In Section5.1.1 we make first steps towards proving that the order in which we apply the isolated clique and vertex fold reductions does not change the kernel that is reached. A full proof for this seems achievable and remains for future work.

Additionally, we plan to evaluate the application of (inexact) maximum independent set algorithms as those that are mentioned in Chapter 3 on our quasikernel.

A Reduction Order

		kernel size		
graph	source	min	average	max
cnr-2000	[10, 11]	67341	67341	67341
in-2004	[10, 11]	192067	192067	192067
petster-friendships-cat-uniq	[30]	68643	68643	68643
petster-friendships-dog-uniq	[30]	139861	139861	139861
zhishi-baidu-relatedpages	[30]	14011	14011	14011
roadNet-CA	[33]	305342	305342	305342
roadNet-PA	[33]	169067	169067	169067
roadNet-TX	[33]	194358	194358	194358
web-Google	[33]	21101	21101	21101
web-NotreDame	[33]	42373	42373	42373

Table A.1: Random order of the isolated clique and vertex fold reductions over 10 runs on each graph. The kernel size is the same for all runs on every graph.

Bibliography

[1] Faisal N. Abu-Khzam, Rebecca L. Collins, Michael R. Fellows, Michael A. Langston, W. Henry Suters, and Christopher T. Symons. Kernelization algorithms for the vertex cover problem: Theory and experiments. In *Proceedings of the Sixth Workshop on Algorithm Engineering and Experiments and the First Workshop on Analytic Algorithmics and Combinatorics*, pages 62–69, 2004.

[2] Takuya Akiba and Yoichi Iwata. Branch-and-reduce exponential/fpt algorithms in practice: A case study of vertex cover. *Theoretical Computer Science*, 609:211–225, 2016.

[3] Diogo V. Andrade, Mauricio G.C. Resende, and Renato F. Werneck. Fast local search for the maximum independent set problem. In *Proceedings of the International Workshop on Experimental and Efficient Algorithms*, pages 220–234. Springer, 2008.

[4] Ariful Azad, Aydin Buluç, and Alex Pothen. Computing maximum cardinality matchings in parallel on bipartite graphs via tree-grafting. *IEEE Trans. Parallel Distrib. Syst.*, 28(1):44–59, 2017.

[5] Ariful Azad, Mahantesh Halappanavar, Sivasankaran Rajamanickam, Erik G. Boman, Arif Khan, and Alex Pothen. Multithreaded algorithms for maximum matching in bipartite graphs. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 860–872. IEEE, 2012.

[6] Thomas Back and Sami Khuri. An evolutionary heuristic for the maximum independent set problem. In *Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the First IEEE Conference on*, pages 531–535. IEEE, 1994.

[7] David A. Bader, Henning Meyerhenke, Peter Sanders, Christian Schulz, Andrea Kappes, and Dorothea Wagner. Benchmarking for graph clustering and partitioning. In *Encyclopedia of Social Network Analysis and Mining*, pages 73–82. Springer, 2014.

[8] Egon Balas and William Niehaus. Optimized crossover-based genetic algorithms for the maximum cardinality and maximum weight clique problems. *Journal of Heuristics*, 4(2):107–122, 1998.

[9] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. Ubicrawler: A scalable fully distributed web crawler. *Software: Practice & Experience*, 34(8):711–726, 2004.

[10] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th international conference on World Wide Web*, pages 587–596. ACM Press, 2011.

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

[12] Pavel A. Borisovsky and Marina S. Zavolovskaya. Experimental comparison of two evolutionary algorithms for the independent set problem. In *Workshops on Applications of Evolutionary Computation*, pages 154–164. Springer, 2003.

[13] Sergiy Butenko, Panos Pardalos, Ivan Sergienko, Vladimir Shylo, and Petro Stetsyuk. Finding maximum independent sets in graphs arising from coding theory. In *Proceedings of the 2002 ACM symposium on Applied computing*, pages 542–546. ACM, 2002.

[14] Sergiy Butenko, Panos Pardalos, Ivan Sergienko, Vladimir Shylo, and Petro Stetsyuk. Estimating the size of correcting codes using extremal graph problems. In *Optimization*, pages 227–243. Springer, 2009.

[15] Sergiy Butenko and Wilbert E. Wilhelm. Clique-detection models in computational biochemistry and genomics. *European Journal of Operational Research*, 173(1):1–17, 2006.

[16] Shaowei Cai, Kaile Su, Chuan Luo, and Abdul Sattar. Numvc: An efficient local search algorithm for minimum vertex cover. *Journal of Artificial Intelligence Research*, 46:687–716, 2013.

[17] Jianer Chen, Iyad A. Kanj, and Weijia Jia. Vertex cover: further observations and further improvements. *Journal of Algorithms*, 41(2):280–301, 2001.

[18] Jakob Dahlum, Sebastian Lamm, Peter Sanders, Christian Schulz, Darren Strash, and Renato F. Werneck. Accelerating local search for the maximum independent set problem. In *Proceedings of the International Symposium on Experimental Algorithms*, pages 118–133. Springer, 2016.

[19] Irit Dinur and Shmuel Safra. The importance of being biased. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 33–42. ACM, 2002.

[20] Fedor V. Fomin, Fabrizio Grandoni, and Dieter Kratsch. A measure & conquer approach for the analysis of exact algorithms. *Journal of the ACM*, 56(5):25, 2009.

[21] Eleanor J. Gardiner, Peter Willett, and Peter J. Artymiuk. Graph-theoretic techniques for macromolecular docking. *Journal of Chemical Information and Computer Sciences*, 40(2):273–279, 2000.

[22] Andreas Gemsa, Benjamin Niedermann, and Martin Nöllenburg. Trajectory-based dynamic map labeling. In *Proceedings of the International Symposium on Algorithms and Computation*, pages 413–423. Springer, 2013.

[23] Andreas Gemsa, Martin Nöllenburg, and Ignaz Rutter. Evaluation of labeling strategies for rotating maps. *Journal of Experimental Algorithmics*, 21(1):1–4, 2016.

[24] Andrea Grosso, Marco Locatelli, and Federico Della Croce. Combining swaps and node weights in an adaptive greedy approach for the maximum clique problem. *Journal of Heuristics*, 10(2):135–152, 2004.

[25] Jiong Guo and Rolf Niedermeier. Invitation to data reduction and problem kernelization. *SIGACT News*, 38(1):31–45, March 2007.

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

[27] David S. Johnson and Michael A. Trick. *Cliques, coloring, and satisfiability: second DIMACS implementation challenge, October 11-13, 1993*, volume 26. American Mathematical Soc., 1996.

[28] Richard M. Karp and Michael Sipser. Maximum matching in sparse random graphs. In *Foundations of Computer Science, 1981. SFCS'81. 22nd Annual Symposium on*, pages 364–375. IEEE, 1981.

[29] Tim Kieritz, Dennis Luxen, Peter Sanders, and Christian Vetter. Distributed time-dependent contraction hierarchies. In *Proceedings of the International Symposium on Experimental Algorithms*, pages 83–93. Springer, 2010.

[30] Jérôme Kunegis. Konect: the koblenz network collection. In *Proceedings of the 22nd International Conference on World Wide Web*, pages 1343–1350. ACM, 2013.

[31] Sebastian Lamm, Peter Sanders, and Christian Schulz. Graph partitioning for independent sets. In *Proceedings of the International Symposium on Experimental Algorithms*, pages 68–81. Springer, 2015.

[32] Sebastian Lamm, Peter Sanders, Christian Schulz, Darren Strash, and Renato F. Werneck. Finding near-optimal independent sets at scale. In *Proceedings of the 18th Workshop on Algorithm Engineering and Experiments (ALENEX 2016)*, pages 138–150. SIAM, 2016.

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

[34] Michael Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM journal on computing*, 15(4):1036–1053, 1986.

[35] Henning Meyerhenke, Peter Sanders, and Christian Schulz. Partitioning complex networks via size-constrained clustering. In *International Symposium on Experimental Algorithms*, pages 351–363. Springer, 2014.

[36] Henning Meyerhenke, Peter Sanders, and Christian Schulz. Parallel graph partitioning for complex networks. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 1055–1064. IEEE, 2015.

[37] George L. Nemhauser and Leslie E. Trotter Jr. Properties of vertex packing and independence system polyhedra. *Mathematical Programming*, 6(1):48–61, 1974.

[38] George L. Nemhauser and Leslie E. Trotter Jr. Vertex packings: structural properties and algorithms. *Mathematical Programming*, 8(1):232–248, 1975.

[39] Wayne Pullan. Phased local search for the maximum clique problem. *Journal of Combinatorial Optimization*, 12(3):303–323, 2006.

[40] Wayne Pullan. Optimisation of unweighted/weighted maximum independent sets and minimum vertex covers. *Discrete Optimization*, 6(2):214–219, 2009.

[41] Wayne Pullan and Holger H. Hoos. Dynamic local search for the maximum clique problem. *Journal of Artificial Intelligence Research*, 25:159–185, 2006.

[42] Pedro V Sander, Diego Nehab, Eden Chlamtac, and Hugues Hoppe. Efficient traversal of mesh edges using adjacency primitives. In *ACM Transactions on Graphics*, volume 27, page 144. ACM, 2008.

[43] Alok Singh and Ashok Kumar Gupta. A hybrid heuristic for the maximum clique problem. *Journal of Heuristics*, 12(1-2):5–22, 2006.

[44] Darren Strash. On the power of simple reductions for the maximum independent set problem. In *Proceedings of the International Computing and Combinatorics Conference*, pages 345–356. Springer, 2016.

[45] Robert Endre Tarjan and Anthony E. Trojanowski. Finding a maximum independent set. *SIAM Journal on Computing*, 6(3):537–546, 1977.

[46] Jingen Xiang, Cong Guo, and Ashraf Aboulnaga. Scalable maximum clique computation using mapreduce. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 74–85. IEEE, 2013.

[47] Mingyu Xiao and Hiroshi Nagamochi. Confining sets and avoiding bottleneck cases: A simple maximum independent set algorithm in degree-3 graphs. *Theoretical Computer Science*, 469:92–104, 2013.

[48] Mingyu Xiao and Hiroshi Nagamochi. Exact algorithms for maximum independent set. In *Proceedings of the International Symposium on Algorithms and Computation*, pages 328–338. Springer, 2013.