Local Search with Optimal Neighborhood Exploration for the Maximum Weight Independent Set Problem

Jannick Borowitz

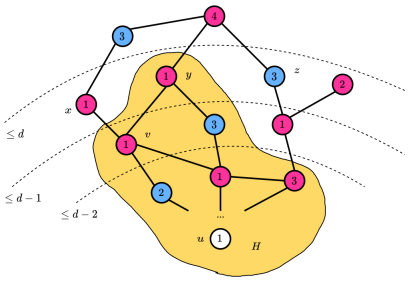
August 23, 2022

4130176

Bachelor Thesis

at

Algorithm Engineering Group Heidelberg
Heidelberg University



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

Co-Supervisor:
Ernestine Großmann

Acknowledgements

Zunächst gilt mein Dank Christian Schulz und Ernestine Großmann, die diese Arbeit betreut haben. In den wöchentlichen Meetings hatten sie stets spannende Ideen und Anregungen, die mir geholfen haben, meine Bachelorarbeit mit einem tollen Ergebnis abschließen zu können und immer ermöglicht haben, von ihnen zu lernen.
Ich möchte mich an dieser Stelle bei Alexander Noe bedanken, ohne den wir METAMIS nicht hätten reimplementieren können. Des Weiteren hatte ich anregende Diskussionen und Unterhaltungen in dieser Zeit mit Marcelo Fonseca Faraj, Joseph Holten, Marco Schröder und Patrick Steil zum Thema der zugrunde liegenden Arbeit und C++.
Starken Rückhalt hatte ich in dieser Zeit von meinen Freunden, die immer ein offenes Ohr hatten. Dabei gilt insbesondere mein Dank meinem Mitbewohner und guten Freund, Nils Witt.
Diese Arbeit wäre aber zu guter Letzt nie ohne meine Familie möglich gewesen. Meine Familie stand mir immer zur Seite und hat mir so überhaupt erst ermöglicht, ein Studium der Informatik aufnehmen zu können.

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Heidelberg, den August 23, 2022

Jannick Borowitz

Abstract

This work considers the NP-complete *maximum weight independent set problem* and introduces a new local-search technique that we call *optimal neighborhood exploration* (ONE). The maximum weight independent set problem applies to many fields of interest like map-labeling or vehicle routing problems. In these fields of interest, the problem-modeling graphs are typically extremely large, with up to millions of vertices and edges. That makes it very difficult to find (near-)optimal solutions in a short time. On the one hand, there exist good exact solvers who can find optimal solutions. However, they still have an exponential run-time in the worst-case. On the other hand, heuristic algorithms improve solutions using a local-search technique by deciding a vertex locally optimally. We generalize the idea and combine both. ONE explores the neighborhood of the seed vertex up to a certain distance and builds a local induced subgraph covering these vertices. Every improvement of a solution for the local induced subgraph improves the solution of our input graph. To improve the solution quality, we solve these induced subgraphs (near-)optimally using the state-of-the-art branch-and-reduce solver KaMIS (KaMIS BaR). Furthermore, we devise a new iterated local-search algorithm called Exhaustive Optimal Neighborhood Exploration (EONE) using ONE. In numerous experiments we will investigate the effectiveness of this approach and compare it to other state-of-the-art solvers. These experiments underline that our solver often finds optimal solutions or at least near-optimal solutions. Furthermore, we outperform the state-of-the-art branch-and-reduce solver KaMIS BaR on hard instances. Finally, we discuss ONE and EONE and give an overview of future work.

Contents

CHAPTER 1

Introduction

1.1 Motivation

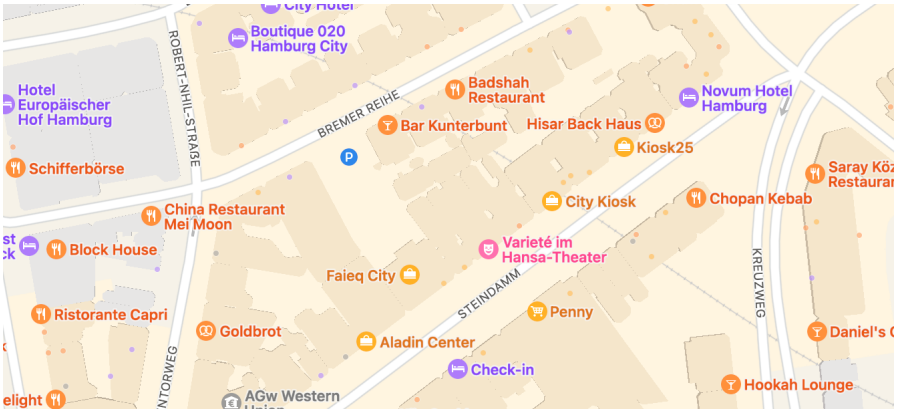


Figure 1.1: Screenshot of Apple Maps showing small part of Hamburg city centre with many places (labels). Labels which were not rendered are marked as dots, because they would overlap in this perspective.

Imagine a map of a city centre on the screen of a device as in Figure 1.1. We have labels of special places like restaurants, cafes, markets and many more placed in the map which are tied to specific coordinates. Further, the labels are weighted due to their relevance for a certain user, e.g. stars one to five. The user wants to see as many important labels on the screen as possible. However, since the screen size is limited and labels have a minimum size and must not overlap, the map provider has to regard the demand for a clear display and cannot simply present all labels in one view. A perfect solution would be a subset of labels whose accumulated weights form

the maximum among all subsets while there is a clean representation of labels on the screen of the user's end device.
This is just one typical example where the *maximum weight independent set problem* applies. Situations as such are modeled by undirected, node-weighted graphs. For this application the nodes of a graph represent the labels with assigned weights while those conflicts in the presentation on the screen form a binary relation which can be represented by undirected edges. When applying the terminology to the *maximum weight independent set problem*, one is interested in a subset of vertices of maximum weight while any two vertices in this set are not adjacent.
The *maximum weight independent set problem* applies in many fields of interest: the *vehicle routing problem* [18], the *map-labeling problem* [11, 22], and many more [10]. In the case of real-world instances, approaches are challenged with extremely large graphs with up to millions of vertices and edges. Solving the *maximum weight independent set problem* is non-trivial and to be more precise: NP-complete. This makes the task for exact and in-exact solvers even more challenging. They have to consider real-world modeling graphs, should compute (near-)optimal solutions while maintaining good run times.
Heuristic algorithms so far often search for improvements operating on neighborhood structures to find high-quality solutions in good run times. They exchange solution vertices with non-solution vertices within these neighborhoods in the solution if the difference in the accumulated weights of the independent set yields an improvement. Conceiving these neighborhoods as induced subgraphs, one can observe that these techniques try to optimize the solution in this subgraph, i. e., try to solve the *maximum weight independent set problem* locally. For large instances their found solution might not be good enough. On the other hand, exact algorithms still have an exponential run time in the worst-case besides all kernelization and search-space pruning techniques. Therefore, they can be infeasible for too-large instances. Hence, the question arises whether we can both combine and find (near-)optimal solutions for the input graph by solving subgraphs induced by a set of explored vertices, which are then solved optimally.

1.2 Our Contribution

We devise a local-search technique, called *optimal neighborhood exploration* (ONE) that explores the neighborhood of a vertex locally up to a certain depth $d = \lfloor \varepsilon^{-1} \rfloor$ in order to obtain induced subgraphs (with at most $\beta > 0$ vertices) such that local solutions to the *maximum weight independent set problem* improve the global solution. The first algorithm we present finds such a set with at most $\mathcal{O}(\Delta(G)^d)$ vertices inducing the subgraphs. The second and more sophisticated algorithm incorporates a bound $\beta > 0$ on the number of vertices in the induced subgraph. This allows us

to solve instances whereby the graph size is strictly bounded independently of G. As a consequence, given β, solving the local instance is *fixed-parameter tractable*. These local instances can be solved with a solver of our choice which makes this approach highly adaptive and hence, is able to interface with other solvers. However, in this work, we restrict ourselves to solving these instances using the state-of-the-art branch-and-reduce solver KaMIS by Lamm et al. [28]. Finally, we engineer an *iterated local-search* solver which exhaustively applies ONE and has a worst-case complexity bounded by the run-time of $n(G)$ ONE-applications per scan over the vertices.

1.3 Structure

The remainder of this thesis is organized as follows. In Chapter 2 basic concepts are introduced ending with a formal definition of the *maximum weight independent set problem*. Chapter 3 gives an overview of related work on which our contribution builds on. We present our contribution and introduce *optimal neighborhood exploration* in Chapter 4 ending with a new iterated local-search solver. This is followed by an experimental evaluation in Chapter 5. We close this work with a discussion and questions intended for future work in Chapter 6.

CHAPTER 2

Fundamentals

First, we introduce some important terms and notations used in the following and then define the *maximum weight independent set problem*.

2.1 General Definitions

Given a weighted, undirected graph $G = (V, E, \omega)$, where V is the set of vertices, $E = \{\{u, v\} : u, v \in V\}$ the set of edges, and $\omega : V \to \mathbb{N}$ is a weight function that assigns every vertex $v \in V$ a weight $\omega(v) \in \mathbb{N}$. We generalize the weight function to a set of vertices by defining $\omega(A) = \sum_{u \in A} \omega(u)$ for some $A \subset V$. In general, the weight function could also map into the real numbers. However, in this work and especially in the experimental section, we will only consider graphs whose nodes are assigned natural numbers. We use $n(G)$ to denote the number of vertices in V and $m(G)$ denotes the number of edges in E of G. In the remaining work, we assume any graph to be weighted and undirected. Further, '$\subset$' denotes a subset, but in general not a proper subset.
The distance $\text{dist}(u, v)$ between $u \in V$ and $v \in V$ is the length of a shortest path connecting u and v in G or ∞ if there is no path between u and v.
The neighborhood of some vertex $u \in V$ is $\text{N}(u) \coloneqq \{v \in V : \{u, v\} \in E\}$. We generalize the neighborhood of u to the neighborhood of a subset $A \subset V$ by setting $\text{N}(A) = \bigcup_{u \in A} \text{N}(u) \setminus A$. The neighborhood of distance $d \in \mathbb{N}$ is defined as $\text{N}^d(u) \coloneqq \{v \in V : \text{dist}(u, v) = d\}$. An equivalent inductive definition for $d \geq 2$ is: $\text{N}^d(u) = \text{N}(\text{N}^{d-1}(u)) \setminus \text{N}^{d-2}(u)$ where $\text{N}^0(u) = \{u\}$. Moreover, $\text{N}^{\leq d}(u)$ is the disjoint union of neighborhoods of u from level 0 to d. Further, $d_G(u) = |\text{N}(u)|$ for $u \in V$ denotes the degree of u and $\Delta(G)$ the maximum degree in G.
A set $\mathcal{I} \subset V$ is called an *independent set* (IS) if it contains no adjacent vertices, i. e., for every vertices $u, v \in \mathcal{I}$ it holds that $\{u, v\} \notin E$. We denote with $\text{IS}(G)$ the set of

all independent sets of G. $\mathcal{I}$ is *maximal* if there is no vertex $u \in V \setminus \mathcal{I}$ which can be added to $\mathcal{I}$ such that $\mathcal{I} \cup \{u\}$ is an independent set.
Let $u \in V \setminus \mathcal{I}$ and $A \subset \mathcal{I}$. We say u is *tight* to A if $A = \mathrm{N}(u) \cap \mathcal{I}$. The definition is motivated by the fact that u can always join $\mathcal{I}$ if we remove A from it. The *tightness* $\tau(u)$ of u is defined as $|\mathrm{N}(u) \cap \mathcal{I}|$. We also say that u is *k-tight* if $k = \tau(u)$. In case we remove some $v \in \mathcal{I}$, we can add any 1-tight neighbor of v to $\mathcal{I} \setminus \{v\}$. This terminology goes back to Andrade et al. [6] and is used along the analysis of (x,y)-swaps [18]. Further, let $t(A) \subset V \setminus \mathcal{I}$ be the *set of tight neighbors of* A, i. e., vertices which are tight to some $A' \subset A \cap \mathcal{I}$.
A *maximum independent set* (MIS) $\mathcal{I} \in IS(G)$ is an independent set of a graph G if it maximizes $|\mathcal{I}|$. The *maximum independent set problem* asks for a maximum independent set given an unweighted, undirected graph. It is a well-known NP-complete problem [20]. As before, a solution to the problem is not unique and must always be maximal.
Given an independent set $\mathcal{I} \in \mathrm{IS}(G)$, if one is interested in $\omega(\mathcal{I})$, then we refer to it as a *weight independent set* (WIS). A *maximum weight independent set* (MWIS) $\mathcal{I} \in \mathrm{IS}(G)$ is an independent set that maximizes $\omega(\mathcal{I})$. The *maximum weight independent set problem* asks for a maximum weight independent set given a weighted, undirected graph. The problem is NP-complete since the *maximum independent set problem* can be reduced to it if one defines $\omega(v) \coloneqq 1$ for every vertex $v \in V$. In general, a solution is not unique and must always be a maximal independent set . Figure 2.1 gives an example of a WIS, a maximal WIS and a MWIS of G.
The MWIS problem is strongly related to other well-known NP-complete problems. An MWIS $\mathcal{I}$ of G is a *maximum weight clique* in in the complement graph of G. Moreover, $V \setminus \mathcal{I}$ a *minimum weight vertex cover* of G.
A *weight clique cover* of G is a collection of cliques $C_1, \ldots, C_k \subset V$, with associated weights $W_1, \ldots, W_k$ such that the union of all cliques covers V, and for every vertex $v \in V$ holds that $\sum_{i:\ v \in C_i} W_i \geq \omega(v)$. The *weight of a clique cover* is the sum of all W_i and provides an upper bound for a MWIS of G [28].

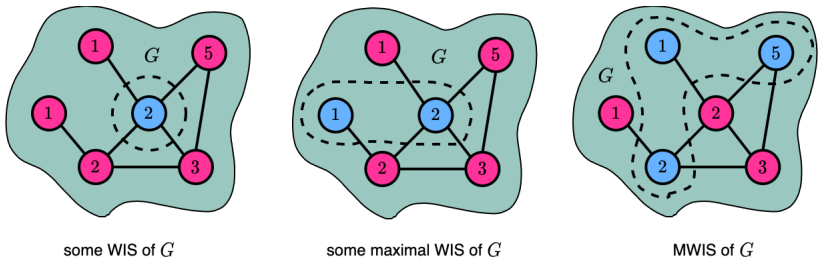


Figure 2.1: An example for an IS (left), a maximal WIS (middle) and a MWIS of G (right). The blue vertices in the dashed areas form the independent sets. The pink vertices are not in the independent set.

CHAPTER 3

Related Work

This chapter discusses related work in solving the MWIS problem optimally, as well as how to determine a high-quality maximal weight independent set. As there has been a lot of research already, we will focus on more recent work and particularly those on which our contribution build on.

3.1 Exact Algorithms

A lot of research has been devoted to exact algorithms. These are algorithms which find an optimal solution and can prove optimality.
First, one can state and solve the MWIS problem as an *integer linear program* (ILP) [18].

$$\begin{aligned} &\text{maximize} && \sum_{u \in V(G)} \omega(u) x_u \\ &\text{subject to} && x_u + x_v \leq 1, \quad \forall e_{\{u,v\}} \in E(G) \\ & && x_u \in \{0,1\}, \quad \forall u \in V(G) \end{aligned}$$

Padberg [31] introduces other ILPs using stronger formulations by utilizing *clique inequalities*. For an k-clique $\{v_1, \ldots, v_k\} \subset V(G)$ one adds the inequality $x_{v_1} + \ldots + x_{v_k} \leq 1$ as additional constraint to the ILP. This is based on the observations that in one clique at most one vertex can be part of an independent set. However, the efficiency of solving an ILP depends heavily on the chosen solver framework in the end. This motivates to think of solvers that are specifically engineered to find optimal solutions for the MWIS problem.
A *branch-and-bound* scheme is common a approach towards fast exact solvers [7]. Roughly speaking, *branching* on a vertex v means to split the solution space in two

parts. One which includes the vertex and one which does not. At least one solution space contains an optimal solution. The vertex is picked using a branching rule. The branching rule is a heuristic which determines the order in which the algorithm branches on vertices. Although this rule depends on the problem, the intuition is to order the vertices such that the number of overall branching steps is small. This is to prevent an exhaustive search for the best solution. *Bounding* is done using *upper* and *lower bounds* on the solution space. For the upper bound a weighted clique cover of G is typically computed [33, 28]. For the lower bound one can compute a solution using a heuristic algorithm such that we have an current best solution. When the computed upper bound for the current sub-problem falls below the best solution, we know that branching on the sub-problem cannot improve the best solution. Lamm et al. [28] compute a lower bound using an iterated local-search solver, a weighted variant of ARW by Andrade et al. [6], after applying reductions initially and dividing the problem into sub-problems, i. e., G is divided into its connected components. Nevertheless, a branch-and-bound algorithm still has an exponential run-time in the worst-case because of the branch-operation.
To further prune the exponent a common technique is *kernelization* [4]. It uses *reduction rules* to reduce the graph to an irreducible *kernel* by applying them incrementally, removing subgraphs and modifying node-weights in a sound manner [28]. These reduction rules allow us to efficiently decide vertices optimally, i. e., we can decide whether a vertex is in the solution or not (given an optimal solution for the *kernel*). Many reductions were found for the MWIS problem by transferring them from the unweighted problem [12, 28]. If no reductions are applicable anymore then the branching takes place on the highest degree vertex. Thus, many neighborhoods become smaller since the decided vertices, including their neighborhoods, can be removed from the graph, and it remains a simpler subgraphs which needs to be solved. This results in a branch-and-reduce framework. The branch-and-reduce solver KaMIS by Lamm et al. [28] is state-of-the-art.
Lately, Gellner et al. [21] developed new optimal transformation rules called *structions* which allow to increase the size of a graph. Although it sounds counterintuitive to increase the size of a graph, experiments show that applying reductions after increasing the instance yield even smaller kernels when not already solved to optimality. The intuition is to apply reductions after increasing the size of the graph to obtain a smaller kernel in the end.
Improving and finding new kernelization techniques is an open topic. Figiel et al. [19] introduced rules to undo reductions for the unweighted vertex cover problem which they name *backward rules*. They apply existing reductions backwards to obtain slightly larger equivalent instances. Considering these increased instances yields overall smaller kernels because they not only undo an applied reduction. Instead they search for opportunities to apply reductions backward in the current graph without considering the last applied reduction.

3.2 Heuristic Algorithms

Heuristic algorithms aim to find (near)-optimal solutions. However, it is unclear whether the final solution is optimal. As heuristics they often use *local-search* techniques to incrementally improve a single or a fixed set of current best solutions. They are denoted as *local-search solvers*, or *iterated local-search solvers* if they apply a local-search for multiple iterations.

3.2.1 Local-Search

Common local-search techniques are so-called (x,y)-swaps which move $x \in \mathbb{N} \cup \{*\}$ vertices out in order to move $y \in \mathbb{N} \cup \{*\}$ into the solution operating on neighborhoods. If '$*$' is used for x or y, it means that the swap applies to arbitrary x and y, respectively. Such a swap is improving in the weighted case if the accumulated weight of the moved-in vertices is larger than the one of the moved-out vertices. We discuss $(1,*)$- and $(*,1)$-swaps in more detail in Chapter 4.
This technique was first used in the unweighted case by Andrade et al. [6]. They use (1,2)-swaps in their famous iterated local-search solver (ARW). Their algorithm processes the swaps in linear-time and in addition, it can decide in linear-time whether an improving (1,2)-swap still exists. Nogueira et al. [30] developed the Hybrid Iterated Local-Search (HILS) based on ARW but for the MWIS problem. They use (1,2)-swaps combined with $(*,1)$-swaps.
Lately, Dong et al. [18] developed an *iterated local-search* algorithm (METAMIS) for the MWIS problem designed for large instances and particularly optimized it for the long-haul vehicle routing (VR) instances [16, 17]. METAMIS is based on the *greedy randomized adaptive search procedure* (GRASP) by Resende et al. [32] combined with *path relinking*. The idea of GRASP as used in their algorithm is to repetitively generate diverse greedy solutions to which one applies a local-search procedure to improve them afterwards. Instead of evaluating whether this improved solution should be added to the set of best solutions, they use this solution as guiding solution for a truncated path-relinking. Truncated path-relinking takes a best solution (typically a local optimum of the local-search) and a guiding solution to find a new solution which is far away enough from the best one in order to prevent converging to the same local optimum again. Because the guiding solution is typically worse than the best one, the path-relinking is truncated at some point in order to obtain a solution which is not much worse than the best one. For path-relinking $(1,*)$- and $(*,1)$-swaps (not necessarily improving) to transform the best solution step-by-step to the guiding solution.
In the local-search procedure they search for improving $(*,1)$-, $(1,*)$- and $(2,*)$-swaps. Furthermore, they introduce and use *alternating augmenting path moves* (AAP-moves). An AAP is defined as a path such that solution vertices and non-

solution vertices along the path alternate in the order they appear and *flipping* the vertices, i. e., exchanging the vertices between the solution set $\mathcal{I}(G)$ and the complement $V \setminus \mathcal{I}(G)$, must maintain an independent set. The move is accepted if the solution weight is improved by flipping the path.

3.2.2 Greedy Algorithms

It is common to determine initial solutions for heuristic algorithms (if not given) using *greedy algorithms*. They process an ordered sequence of the vertices in the graph and choose them into the solution if not contradicting an independent set. The order is determined using a simple heuristic like the weight $\omega(u)$ or the weight-per-degree rating $\omega(u)/\deg(u)$. For METAMIS Dong et al. [18] use a randomized strategy in the local-search and an adaptive greedy variant if no initial solution is provided. The greedy algorithm can be randomized by choosing greedily uniform at random from a fixed-sized set of best feasible candidates. The adaptive variant uses the weight-per-degree rating and removes the added and infeasible vertices from the graph while maintain the degree and therefore can update the rating after each greedy addition.

3.3 Hybrid Algorithms

Besides swaps, another local-search technique uses kernelization techniques [14, 13, 24]. They combine exact reductions with heuristics. In the case of the MIS problem, an observation is that high-degree vertices are unlikely to be in a MIS. Dahlum et al. [14] compute a kernel using the reductions by Akiba et al. [5]. The kernel is then solved by ARW. To make ARW more effective, they remove higher degree vertices from the kernel.This results in an algorithm called KerMIS. Moreover, they devise an online-fashion algorithm, called OnlineMIS. The algorithm does not compute a kernel in advance, but instead forces *isolated vertices*, i. e., vertices that form a clique (here of degree 1, 2, 3), into the solution using the isolated-vertex-removal reduction and marks them and their neighbors as removed. It removes a few high-degree vertices and runs ARW. A vertex joins the solution if it is isolated. The local-search continues until every vertex is marked as removed.

The benefits of kernelization in an inexact setting are also used for the weighted problem. Gu et al. [24] present a reduce-and-peeling framework that does not branch on a vertex, like in a branch-and-reduce framework, but instead decides a vertex using heuristics to break the tie when reductions are applied exhaustively.

3.4 Evolutionary Algorithms

Another family of algorithms are evolutionary algorithms for the MIS problem as EvoMIS by Lamm et al. [26] and ReduMIS by Lamm et al. [27]. In this context, an evolutionary algorithm maintains a population of individuals (independent sets). By selecting the fittest parents (independent sets) as input for a combine-operation, we obtain improved new children (independent sets) which are added to the population [27, 23]. ReduMIS incorporates kernelization with the evolutionary algorithm EvoMIS. This way, the evolutionary algorithm operates over exact kernels instead of the input graph.

CHAPTER 4

Optimal Neighborhood Exploration

This chapter presents our new local-search technique *optimal neighborhood exploration* (ONE). We start with a small example to motivate ONE. Afterwards, we give the details of ONE and answer how local-search can be done for the MWIS problem. The main result is the contribution of two algorithms for the *exploration*. Finally, we close this chapter with a new iterated local-search solver for the MWIS problem.

4.1 Motivation For A New Local-Search

We motivate this chapter with two examples. For this, we look on the well-known (1,∗)- and (∗,1)-swaps. In particular, we look at two situations where we try to apply one of the swaps and think about what an optimal decision could be by considering only involved vertices and neighbors which are tight to them.
In Figure 4.1, we consider a (1,∗)-swap for some $u \in \mathcal{I}$ where $\mathcal{I} \in \mathrm{IS}(G)$ is some maximal WIS. A (1,∗)-swap applied to u removes u from the solution $\mathcal{I}$, and adds a subset of 1-tight neighbors to the solution such that the obtained solution is again a maximal WIS of G, denoted $\mathcal{I}'$. To that end, we must choose a weight independent set of tight neighbors $t(\{u\})$. The swap improves if

$$\omega(\mathcal{I}') > \omega(\mathcal{I}) \iff \omega(\mathcal{I}' \cap V(H)) > \omega(\mathcal{I} \cap V(H))$$

where $H \subset G$ is the induced subgraph of considered vertices. In our case, we obtain $2 + 1 = 3 \geq 2 = \omega(u)$. Hence, the swap is improving. However, we note that the swap is only improving because we inserted the vertex with weight two instead of its adjacent vertex with weight one. A MWIS of $G[t(\{u\})]$ was substantial to make this specific swap succeed. Instead of $G[t(\{u\})]$, we could consider H and search for a MWIS of H because the weight of a MWIS of H is always larger than the weight of a MWIS of $G[t(\{u\})]$. The intuition is that H does not force u to be in the MWIS.

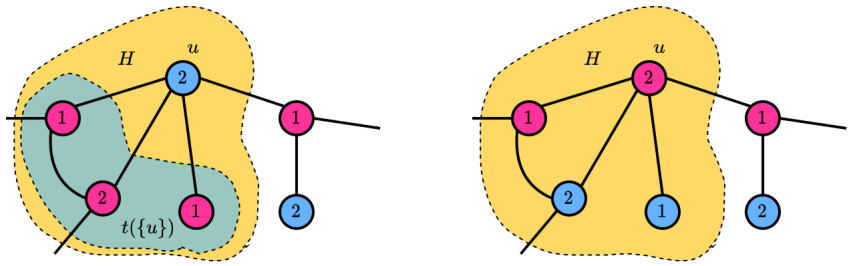


Figure 4.1: Example for an improving (1,∗)-swap for $u \in V$ (blue vertices are part of the solution and pink vertices are out; numbers represent weights), left: before swap, right: after swapping.

If a MWIS of H is not of larger weight than $\omega(\mathcal{I} \cap V(H))$, then the swap cannot be improving because of the latter observation. On the other hand, if the weight is larger, it immediately improves our solution if we embed the solution found for H. That means a vertex in $V(H)$ is part of the solution for G if and only if it is part of the solution found for H.
Next, we discuss the (∗,1)-swap for $u \in V \setminus \mathcal{I}$ in Figure 4.2. It removes $\mathrm{N}(u) \cap \mathcal{I} = \{v, w\}$ from $\mathcal{I}$ and adds u if the *weight difference*

$$\lambda(u) := \omega(u) - \sum_{x \in \mathrm{N}(u) \cap \mathcal{I}} \omega(x) > 0.$$

In our example the swap fails, because $\lambda(u) = 5 - (2+3) = 0$. However, if we take the tight neighbors $t(\mathrm{N}(u))$ into account, from which we can add vertices to the solution after performing the swap nonetheless, then we obtain an overall improvement. We used that if the swap is applied, the solution is no longer maximal, and we can add tight neighbors of $\mathrm{N}(u)$. Figure 4.2 illustrates an MWIS of H on the right-hand side. H is the induced subgraph of vertices that are either swapped by a (∗,1)-swap applied to u or tight to the neighborhood of u. Again, we observe that solving the MWIS problem for H gives us the best improvement possible $\mathcal{I}$ subject to H.
We generalize the example and call these induced subgraphs *local induced subgraphs*. They cover some seed vertex, which is the single vertex u these examples, and its neighbors up to a certain depth (here one and two). Note that every maximal WIS $\mathcal{I}_H$ for the subgraph H does not contradict a maximal WIS of G when we remove the vertices in $V(H)$ from $\mathcal{I}$ and add the vertices in $\mathcal{I}_H$. We call this exchange of solution vertices *embedding* a maximal WIS of H into $\mathcal{I}$. Because of this property of the local induced subgraph, it is possible that a MWIS of H corresponds to the best improvement for $\mathcal{I}$. Moreover, a subgraph as in the example must be chosen in such a way that every maximal WIS $\mathcal{I}_H$ for the subgraph H does not contradict a maximal WIS for G when we remove the vertices in $V(H)$ from $\mathcal{I}$ and add the vertices in $\mathcal{I}_H$.

As a side note, in this fairly simple example a (2,∗)-swap applied to v and w probably finds the improvement in Figure 4.2. But also in this case it depends on the found WIS of tight neighbors of v and w whether the move is accepted.
We conclude these two examples with two questions:

1. The considered swaps often perform very good in iterated local-search solvers. As we saw, this is very similar to solving the associated local induced subgraph H. The question arises, if we can find even better solutions by (optimally) solving local induced subgraphs which cover the neighborhood of u to a larger distance?

2. This leads to another question: how can we efficiently find larger local induced subgraphs? After embedding a better solution for H we do not want to create conflicts contradicting an independent set in G. Furthermore, this new independent should be maximal.

4.2 High-Level Overview

Roughly speaking, our new local-search technique for the MWIS problem, called *optimal neighborhood exploration* (ONE), explores the neighborhood of some seed vertex $u \in V$ as shown in Figure 4.3. We cover it neighbors up to a certain distance in an induced subgraph H given some maximal WIS of G. Afterwards, we search for a MWIS or at least a better near-optimal solution for H and embed it into the solution of G, i. e., we remove vertices covered by H which are not in the locally better solution and add new ones.
We avoid conflicts in the way we choose the induced subgraph. For this induced subgraph we ensure that every vertex in $V(H)$ is not adjacent to a solution vertex not covered by H. Under those circumstances, every potentially new solution vertex

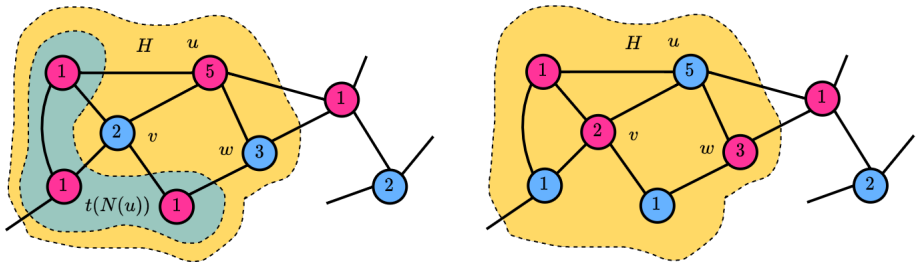


Figure 4.2: Example for a none-improving (∗,1)-swap for $u \in V$ (blue vertices are part of the solution and pink vertices are out; numbers represent weights), left: before swap, right: after solving H optimally.

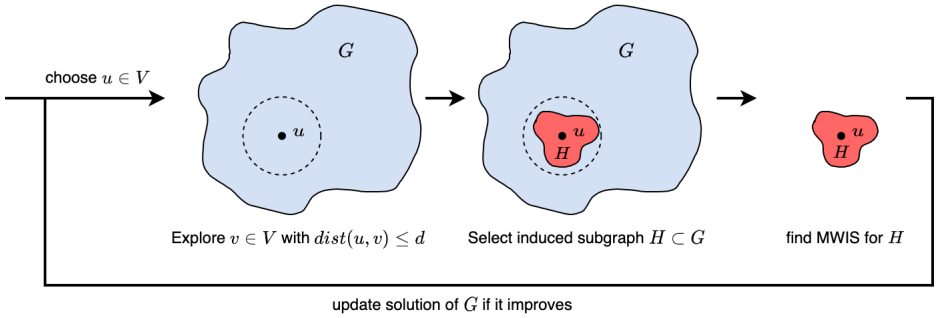


Figure 4.3: High-level overview of *optimal neighborhood exploration*.

in $V(H)$ cannot be adjacent to a solution vertex that is not covered by H. Consequently, we have no conflict, i. e., an edge whose incident vertices are both in the solution, when we embed a maximal WIS for H. Furthermore, we ensure that every vertex which is not part of the solution is only uncovered if it has at least one neighbor in the solution which is not covered by H. As a result, all vertices which are tight to a subset of vertices in $V(H)$ are covered by H. This means that the obtained solution for G is maximal, because every non-solution vertex will have a neighbor in this solution.

We devise a *border policy* which carefully excludes considered vertices from H at distance d and $d-1$ to u if they could yield such a conflict as described. We call a vertex $v \in V$ *border vertex* if v is explored, but we do not intend to explore or cover some neighbors of v. Besides the natural constraint $\varepsilon > 0$ limiting the explored vertices, we later introduce a second constraint $\beta > 0$ which limits the number of vertices in H.

All in all, we present a local-search technique which explores local induced subgraphs and solves the MWIS problem on these subgraphs. In the following we present two algorithms to find local induced subgraphs and explain, how we solve them in order to improve a given maximal WIS.

4.3 Local Induced Subgraphs

First, we specify how we *explore* the neighborhood and obtain a *local induced subgraph* H and then explain how we manage to *solve* the local instance of the MWIS problem. Let $G = (V, E)$ be the given graph and assume that a maximal weight independent set $\mathcal{I} \in \mathrm{IS}(G)$ was already determined. Further, let $u \in V$ be the *initial vertex* to which ONE is applied.

4.3.1 Exploration

The most challenging part of optimal neighborhood exploration is to select the set of vertices that induces the subgraph. The local induced subgraph should cover the initial vertex u and every maximal WIS for the subgraph must not be in conflict with the given maximal WIS for G. Further, $\varepsilon \in (0,1]$ limits the vertices which we can consider for our local induced subgraph: they have at most distance $d = \lfloor \varepsilon^{-1} \rfloor$ to u. Since a graph can have many high-degree vertices, our local induced subgraph can have without any further constraints many vertices for fairly large ε. To that end, we introduced a fourth constraint with $\beta \in \mathbb{N} \cup \{\infty\}$ limiting the number of vertices in the subgraph. We incorporate this bound later on in this section.
This section is structured into two parts: in the first part we present the first approach that selects such a set of vertices but ignores β. From this first approach we lead to a second approach which additionally considers β.

NAIVE-BFS

We consider every vertex $v \in V$ with a distance of at most d to u for the local induced subgraph H. The *border policy* decides whether a vertex will be covered by H or not. The goal is to reject considered vertices such that there is no conflict when a maximal WIS for H is embedded in $\mathcal{I}$.
Border Policy. The *border policy rejects* a vertex $v \in \mathrm{N}^d(u)$ if $v \notin \mathcal{I}$ is not tight to a subset of solution vertices with a distance less than d, or $v \in \mathcal{I}$. In the latter case, vertices in $\mathrm{N}(v)$ at distance $d-1$ and d must also be rejected, because if they join a WIS after solving H they are adjacent to the solution vertex v. In summary, the rejected vertices are either solution vertices at distance d, vertices at distance $d-1$ adjacent to solution vertices at distance d and vertices at distance d adjacent to solution vertices at distance $d+1$ (the not-tight vertices).
A considered vertex $v \in \mathrm{N}^{\leq d}(u)$ is *accepted* if it is not rejected. We cover a vertex in H if it is accepted. We denote this set of accepted vertices $A(d,u)$ which induce H. The set of all accepted vertices $A(d,u)$ contains all vertices with distance less or equals $d-2$. Further, accepted solution vertices appear at most at distance $d-1$. All neighbors tight to a subset of them are accepted too and appear at latest at distance d.
Covering all the tight neighbors ensures that the solution is still maximal after embedding a maximal WIS for H. Since there is no accepted vertex, adjacent to an uncovered solution vertex, $\mathcal{I}$ remains an independent set. Figure 4.4 illustrates the special cases for border vertices in an example.
NAIVE-BFS. In short, our first algorithm NAIVE-BFS (NBFS) finds H given $\varepsilon \in (0, 0.5]$ and an initial vertex $u \in V$ by visiting every vertex of feasible distance to u and accepts them if the border policy agrees. Algorithm 1 gives the pseudo-code.

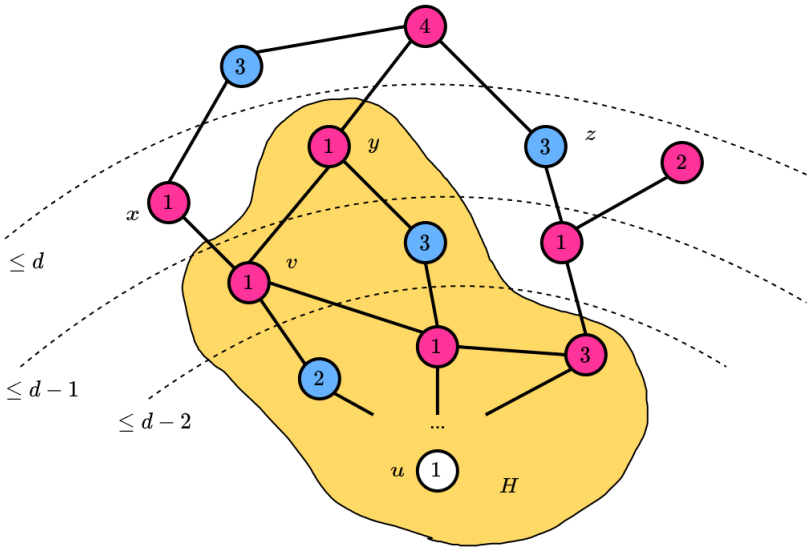


Figure 4.4: Example for the border policy: u is some initial vertex (which is white because solution status does not matter for the example); blue vertices are solution vertices and pink vertices are not in the solution; v at distance $d-1$ is in the solution because it is tight to vertices with smaller distance to u, same holds for y; x is not tight to vertices distance $\leq d$ and therefore rejected, z at distance d is a solution vertex and therefore we reject it with its neighbors.

Every vertex of feasible distance to u is visited by performing a breadth-first-search (BFS) in G starting from u. Thereby, it builds a set of vertices V_H, starting with $V_H = \emptyset$, such that $V_H = A(d, u)$ when the algorithm terminates. The BFS is an appropriate choice because it allows us to process the vertices in levels following the *first-in-first-out* principle implemented through a queue. At level $i \leq d$, we process vertices with distance i to u, and if possible we can enqueue unexplored neighbors. Whenever we process a vertex, we know the level, i. e., the distance to u. Remember that the decision of the border policy for a considered vertex $v \in V$ depends on the distance to u. When a vertex is popped from the queue, we decide whether this vertex is accepted or not. This decision is possible because all vertices are processed in levels, and thus, we know all vertices at smaller levels were decided and at the current level at least enqueued.
If the level is small enough and the vertex is accepted, we enqueue its unexplored neighbors, i. e., vertices which were not in the queue yet. The border policy accepts every vertex which we process at level $\leq d-2$. For this reason, we required $\varepsilon \leq 0.5$ such that $d \geq 2$ and at least u is covered.
Vertices at distance $d-1$ can be accepted if they are in $\mathcal{I}$ or are tight to solution vertices with distance $\leq d-1$. Let v be a vertex which appears at distance $d-1$ Whether v is in $\mathcal{I}$ can be simply checked by a look-up. If $v \notin \mathcal{I}$ a little more effort

is necessary. We need to check whether $\mathrm{N}(v) \cap \mathcal{I} \subset \mathrm{N}^{\leq d-1}(u)$, i. e., *is v tight to the later H?* However, at level $d-1$ it is possible that not all solution vertices are accepted yet. That is to say, we cannot guarantee $V_H \cap \mathcal{I} = \mathrm{N}^{\leq d-1}(u) \cap \mathcal{I}$. But we know all solution vertices at distance $\leq d-1$ are at least marked as explored. Hence, we can scan over the neighborhood of v and check whether a solution vertex exists which was not explored yet. If at least one vertex exists, the border policy rejects v. Otherwise, we accept u because it will be tight to a subset of solution vertices in $V(H)$.
Note that at level $d-1$ only non-solution vertices are enqueued. Thus, at level d we only consider non-solution vertices. At distance d the border policy accepts v only if it is tight to the accepted solution vertices. For this, we check if every solution vertex is in $\mathrm{N}(v)$ is in V_H, i. e., was accepted.
When the queue becomes empty, we obtain $V_H = A(d, u)$ and can build the local induced subgraph.
Run-time. In the worst-case we enqueue and accept $\mathcal{O}(\Delta(G)^d)$ vertices. Checking whether we can accept some vertex takes us at most $\mathcal{O}(\Delta(G))$, because in the worst-case we scan over the neighborhood of some vertex to check for tightness. Building the local induced subgraph can be done in $\mathcal{O}(\Delta(G)^{d+1})$ where we insert for each vertex in $A(d, u)$ its incident edges. Hence, finding $A(d, u)$ and building H has an overall worst-case complexity of $\mathcal{O}(\Delta(G)^{d+1})$.
Conclusion. From a theoretical point of view, it would be desirable to do the `isTight(`v`,` $A(d, u)$`)` in Algorithm 1 at distance d in $\mathcal{O}(1)$, because then the worst-case complexity for finding $A(d, u)$ reduces to $\mathcal{O}(\Delta(G)^d)$.
This variant explores the neighborhood of tight vertices at distance $d-1$ which is not necessary. To make this more vivid, consider v in Figure 4.4. The vertex v is tight to vertices with distance $\leq d-2$ and consequently, it is covered. However, it is not necessary to explore its neighbors. The reason is that its neighbors at distance d are only covered if they are tight to vertices with neighbors that are part of the solution at distance $d-1$. In Figure 4.4 the neighbor y is such a vertex at distance d which is tight to vertices at level $d-1$. It will be explored via its solution neighbor at distance $d-1$ even if v does not explore vertices at distance d.
Moreover, we mentioned introducing a second constraint, namely $\beta > 0$, such that $A(d, u) \leq \beta$. Note that the current policy does not yet support the second constraint β. To obtain local induced subgraphs whose solutions yield maximal WISs in G, we had to ensure that $t(V(H) \cap \mathcal{I}) \subset V(H)$. We know coverable solution vertices will appear at most at distance $d-1$ and all tight vertices will be added when distance d is processed. For example, if we want to add a solution vertex v, we need to ensure that new tight vertices (tight to the induced subgraph which covers v) are added too without exceeding β. To that end, we need to add vertices to V_H before we process them at the respective level.

TIGHT-BFS

We start by adapting the *border policy* such that it incorporates β. In case of $\beta = \infty$, we want that it should yield the same local induced subgraph as the border policy for the same ε. After specifying the requirements, we make an observation which is the basis for the new algorithm. We close this section with an algorithm for exploration which sticks to β.

Dynamic Border Policy. Remember that every local induced subgraph H should yield a maximal WIS in G when we embed a maximal WIS for H in the given solution $\mathcal{I}$. The border policy ensures this by rejecting specific vertices when they have distance $> d-2$ to the seed vertex. Hence, NAIVE-BFS has to take action when vertices with distance $> d-2$ are processed. At smaller levels, it just accepts every vertex. Breaking things down, it computes a sequence of growing induced subgraphs, where the last subgraph is feasible for further computations.

The *dynamic border policy* is a little more restrictive: for such a growing sequence of induced subgraphs, we require that every induced subgraph should be feasible for further computations. That is to say, they all must have vertices less or equal to β and in particular, every induced subgraph should yield a maximal WIS in G when we embed a maximal WIS for H in $\mathcal{I}$. It still holds that only vertices with a distance to u of at most d are considered, and if β is large enough the final induced subgraph must be the same induced subgraph as the border policy proposes, i. e., $H_{max} \coloneqq G[A(d,u)]$. The objective is to maximize the number of vertices in the final induced subgraph subject to the upper bound β and the other constraints when exploring the neighborhood.

All in all, the dynamic border policy specifies the requirements for our new approach. In the following, we give a rough idea of how we accomplish to implement the dynamic border policy using Observation 1.

Observation 1 (Marriage Induced Subgraphs)**.** *Let H_1,$H_2 \subset G$ be induced subgraphs which both yield for maximal WIS of H_i, $i \in \{1,2\}$ a maximal WIS in* $\mathrm{IS}(G)$ *if we embed them in the given maximal WIS $\mathcal{I} \in \mathrm{IS}(G)$. We obtain an induced subgraph M with the same property if it covers $V(H_1)$, $V(H_2)$ and all tight vertices $t(V(H_1) \cup V(H_2))$, i. e., vertices which are tight to a subset of the solution vertices in $V(H_1)$ and $V((H_2)$. We write for this obtained induced subgraph $M = H_1 \cup^{\mathcal{I}} H_2$.*

Figure 4.5 illustrates a marriage of two induced subgraphs as in Observation 1. The new induced subgraph M must cover, besides vertices of H_1 and H_2, two non-solution vertices in the example. These two non-solution vertices are tight to vertices at the border of H_1 and H_2, i. e., their neighbors in $\mathcal{I}$ are covered by H_1 and H_2.

High-Level Overview of TIGHT-BFS. The whole idea of our new algorithm, called TIGHT-BFS (TBFS), is exploring the neighborhood of some initial vertex $u \in V$ *level-by-level* and marry the current induced subgraph H with an induced

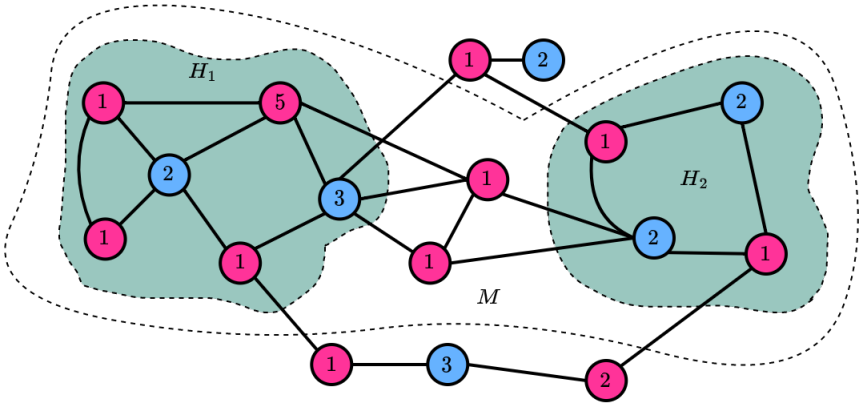


Figure 4.5: Marriage of two induced subgraphs H_1 and H_2. M covers H_1, H_2 and vertices which are tight to solution vertices (blue) in $V(H_1)$ and $V(H_2)$.

subgraph which covers the considered vertex v as described in *Observation* 1. If the induced subgraph obtained by the marriage is too large, i. e., by adding the new vertices we exceed β, the marriage fails.
The induced subgraph which covers v should be of minimum vertex size as we want to maintain the *locality* property, i. e., we want to prioritize vertices which are nearer located to u, and since we process the neighborhood of u in levels vertices of larger distance can still be covered later on if β and ε allow us to do so.
To this end, we investigate how an induced subgraph of minimum vertex size covering v looks like. We remember the subgraphs which we considered for (1,$*$)- and ($*$,1)-swaps in Figures 4.1 and 4.2. These are already the induced subgraphs of minimum vertex size covering v. They are the local induced subgraph we obtain with the border

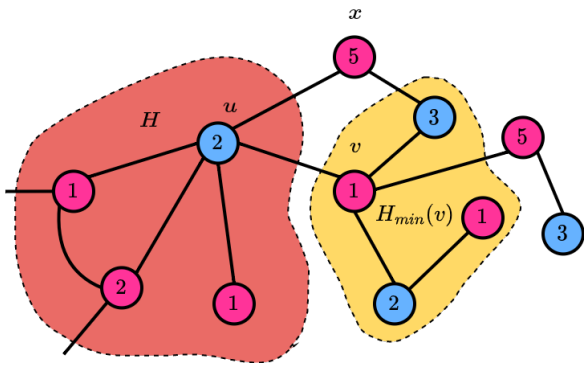


Figure 4.6: Marrying the current local induced subgraph H and $H_{min}(v,G,\mathcal{I})$ to cover v by H. Observe that x must also be covered because its neighbors in the solution (blue) are in H and $H_{min}(v,G,\mathcal{I})$.

policy for $\varepsilon = 0.5$ where v is the initial vertex. If $v \in \mathcal{I}$, this is simply $G[A(2,v)] = G[\{v\} \cup t(\{v\})]$. For $v \notin \mathcal{I}$ it is $G[A(2,v)] = G[\{v\} \cup (\mathrm{N}(v) \cap \mathcal{I}) \cup t(\mathrm{N}(v))]$. In the following we denote these induced subgraphs $H_{min}(v, G, \mathcal{I})$.
In Figure 4.6 we give an example of how TBFS tries to cover v in the local induced subgraph H. We marry H with $H_{min}(v, G, \mathcal{I})$. Note that x in Figure 4.6 must also join the local induced subgraph when marrying them, because it is tight to solution vertices in $V(H)$ and $H_{min}(v, G, \mathcal{I})$.
TIGHT-BFS. Roughly speaking, TIGHT-BFS (TBFS), computes a finite sequence of vertex sets $(V_i)_{i=1}^{k}$ with $k \in \mathbb{N}$ such that associating induced subgraphs $H_i = G[V_i]$ gives us local induced subgraphs as described in the dynamic border policy. The algorithm does so by starting to compute a sequence which respects the dynamic border policy and if every next considered H_{i+1} would contradict $n(H_{i+1}) \leq \beta$, we terminate. In general, it is unclear whether we obtain the largest possible local induced subgraph for u. But to cover a vertex v adjacent to a vertex which is in our instance, we add only the minimum number of necessary vertices, i. e., we marry with with the minimum local induced subgraph for v which we denoted $H_{min}(v, G, \mathcal{I})$.
TBFS is an *online* algorithm, i. e., for computing H_{i+1} we only remember H_i but not the whole sequence. Therefore, we now abuse notation and denote with H the current local induced subgraph and with H' the one after marrying with H and some

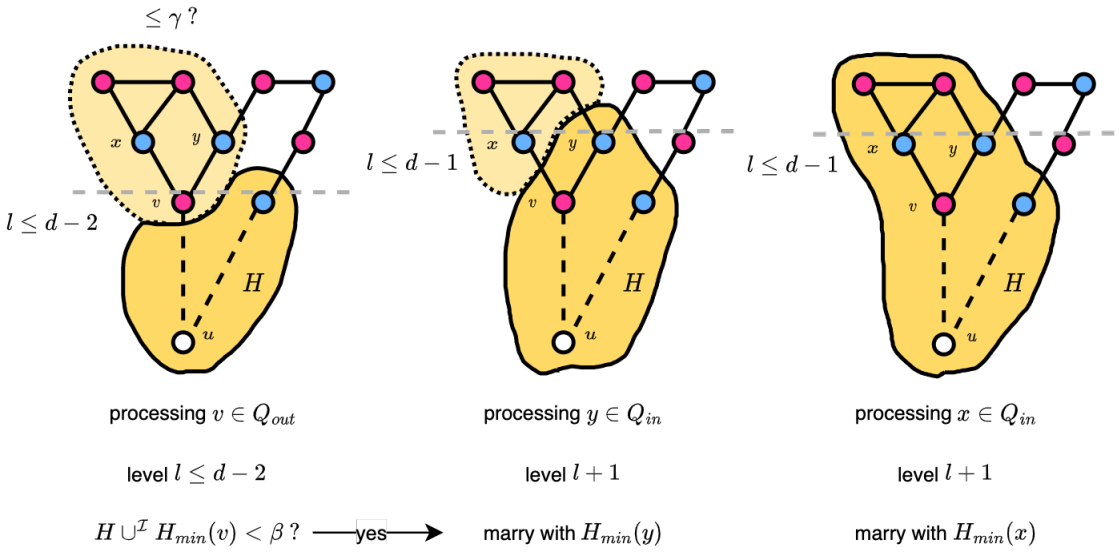


Figure 4.7: Example illustrating the processing of vertices in Q_{in} and Q_{out}. To cover v we precompute whether we can cover the new vertices of marriage of H and $H_{min}(v, G, \mathcal{I})$. Assume for this example β is large enough and the marriage will be successful. At the next level we enlarge H by marrying with the local induced subgraphs of u and v of minimum vertex size.

minimum local induced subgraph.
Again we perform a BFS starting with u. However, this time we process all vertices in $\mathcal{I}$ at the current level at first and then the vertices not in $\mathcal{I}$. We do so by using two queues Q_{in} and Q_{out} in which we add explored solution and non-solution vertices, respectively. Let γ denote the number of vertices which we can still add. In the beginning, γ is initialized with β.
Let $v \in \mathcal{I}$ be the current vertex popped from Q_{in}, then we just marriage H with $H_{min}(u,G,\mathcal{I})$ and obtain $H' = H \cup^{\mathcal{I}} H_{min}(u,G,\mathcal{I})$. We assume that v was only added to Q_{in} if $\beta - n(H') \geq 0$ holds after the marriage. If v is visited at distance less than d and has unexplored neighbors, then we enqueue them in Q_{out} (every neighbor is not part of the solution).
Now we process Q_{out}. We do not change H while processing non solution vertices, but precompute some $\tilde{H}'$ which we will obtain, namely H', when we processed Q_{in} at the next level. We give an example in Figure 4.7. Let $v \in V \setminus \mathcal{I}$ be the popped vertex by Q_{out} and $\tilde{H}$ be the current precomputed local induced subgraph. To decide whether we can add v if it is not covered yet, we need to check whether we can add new vertices when marrying with $H_{min}(v,G,\mathcal{I})$. Let $\tilde{H}' = \tilde{H} \cup^{\mathcal{I}} H_{min}(v,G,\mathcal{I})$ is the precomputed graph by marrying with $H_{min}(v,G,\mathcal{I})$. If $k \coloneqq \left|V(\tilde{H}') \setminus V(\tilde{H})\right| \geq \gamma$, we can cover v and the k necessary vertices. In this case we enqueue all the new neighbors and decrease γ by k. The necessary vertices are v itself, uncovered neighbors in $\mathcal{I}$ and all vertices which would becomes to tight the local induced subgraph. They all appear in $\mathrm{N}^{\leq 2}(v)$. Observe, that one can either add the minimum vertex size local induced subgraph of v or the ones of the solution vertices in the neighborhood of v:

$$\bigcup_{x \in \mathcal{I} \cap \mathrm{N}(v)}^{\mathcal{I}} H_{min}(x,G,\mathcal{I}) = H_{min}(v,G,\mathcal{I}).$$

Because we enqueued all new neighbors of v, it is guaranteed that $H_{min}(v,G,\mathcal{I})$ will be married with H when processing Q_{in} at the next level. Thus, we cover $H_{min}(v,G,I)$ by marrying H with all $H_{min}(x,G,\mathcal{I})$ for all $x \in \mathrm{N}(v) \in Q_{in}$.
This cover check for v is done, if v is appears at distance $< d-1$, because otherwise $H_{min}(v,G,\mathcal{I})$ might contain vertices beyond H_{max}. Note that we still still cover acceptable vertices at distance $d-1$ or $d-2$ since we do the cover check at most at distance $d-2$. Thereby, we visit neighbors of v with distance ≤ 2 to v. We initialize the BFS by enqueuing the initial vertex u in the respective queue. If $u \in \mathcal{I}$, we need to decrease γ by $n(H_{min}(u,G,\mathcal{I}))$ in advance. The algorithm terminates if $\gamma \leq 0$ after processing Q_{in}. Note that the algorithm can return the empty induced subgraph if β is chosen too small.
Run-time. For the run-time analysis we assume that we can do *tight queries* of the scheme *is* $v \in V \setminus \mathcal{I}$ *in* $t(V(H))$ *for* $V(H) \subset V$ in $\mathcal{O}(1)$.
Processing a solution vertex runs in $\mathcal{O}(\Delta(G))$, because we add at most the solution

vertex and some neighbors which become tight to the induced subgraph. Because solution vertices appear at most at level $d-1$, we have at most $\mathcal{O}(\Delta(G)^{d-1})$ solution vertices. Thus, processing all solution vertices can be done in $\mathcal{O}(\Delta(G)^d)$.
For non-solution vertices v we need to compute whether $H_{min}(v, G, \mathcal{I})$ will be covered. This forces us to do tight queries for vertices in $\mathrm{N}^{\leq 2}(v)$, i. e., for vertices with distance of at most two to v. Since the tight queries are done in $\mathcal{O}(1)$ in the worst-case, processing v has worst-case complexity $\mathcal{O}(\Delta(G)^2)$. Note that non-solution vertices are popped from the queue at most at level $d-2$. Consequently, all non-solution vertices can be processed in $\mathcal{O}(\Delta(G)^d)$ too.
As before, we can build an induced subgraph in $\mathcal{O}(\Delta(G)^{d+1})$. This gives us an worst-case complexity of $\mathcal{O}(\Delta(G)^{d+1})$ for selecting the vertices and building the induced subgraph.
If $\beta < \infty$, then the number of vertices in H is bounded by β. From a theoretic point of view, building the induced subgraphs with β vertices can be done in $\mathcal{O}(\beta\Delta)$. Thus, if $\beta < \infty$, we have a worst-case complexity of $\mathcal{O}(\Delta(G)^d + \beta\Delta(G))$. We maintain the tightness of a vertex to $V \setminus V(H)$, and $V \setminus V(\tilde{H})$ in order to make $O(1)$ queries. Maintaining this information for H and $\tilde{H}$ can be done when we perform and evaluate marriages and adds only a constant in the run-time per vertex. We maintain $\tau(v)$ for all vertices $v \in V$ over all calls of ONE. Thus, we can initialize the tightness to $V \setminus V(H)$ and $V \setminus V(\tilde{H})$ for some v in $\mathcal{O}(1)$ when TIGHT-BFS is called. The idea of maintaining $\tau(v)$ is already used by Dong et al. [18]. They implemented it to efficiently update the internal data structure of METAMIS.

4.3.2 Solving

In the following, we explain how the instance H of the MWIS problem is solved where H a local induced subgraph in G computed by one of the algorithms in the previous section. In principle, we could use any solver which aims to solve the MWIS problem optimally or at least optimizes $\mathcal{I}_H \coloneqq V(H) \cap \mathcal{I}$. We choose the branch-and-reduce solver from KaMIS (KaMIS BaR) by Lamm et al. [28] since it is state-of-the-art among the exact algorithms. It is capable to solve many (large) instances optimal while outperforming other approaches. Even compared to iterated local-search approaches like HILS it can often compete in run-time, e. g., for OSM instances (street networks), as the experiments by Lamm et al. [28] show.
In case that KaMIS BaR cannot find an optimal solution in a certain time limit when branching on the initially computed irreducible kernel recursively, the best maximal WIS $\mathcal{I}_H$ for H is returned. Since every maximal WIS for H results in maximal WIS for G when embedded in $\mathcal{I}$, we can simply overwrite the old solution $\mathcal{I}$ in G by removing the vertices which are not in the returned solution for H and adding the vertices $\mathcal{I}_H \setminus \mathcal{I}$ if $\mathcal{I}_H$ improves the old one. Therefore, we only need to

evaluate $\omega(\mathcal{I}_H) > \omega(\mathcal{I} \cap \mathrm{V}(H))$, and if it is true, we can embed the new solution.
Run-time. KaMIS BaR implements the branch-and-reduce framework. Recursively branching on vertices has in the worst-case an exponential run-time in the input graph. In our case the input graph is the explored local induced subgraph H. NBFS and TBFS produce instances with $\mathcal{O}(\Delta(G)^d)$ vertices. If $\beta < \infty$ is given, TBFS finds instances with at most β vertices. In the latter case the worst-case complexity becomes independent of the local induced subgraph H since the number of vertices is bounded by β and the number of edges bounded by $\beta(\beta - 1)$. As a result, we can control the worst-case run-time of KaMIS BaR only by $\beta < \infty$, and consequently solving the local induced subgraph is *fixed-parameter tractable*.

4.4 Iterated Local-Search

Finally, we propose an *iterated local-search* algorithm which uses OPTIMAL NEIGHBORHOOD EXPLORATION. It scans over a shuffled sequence of V and applies ONE to every $u \in V$ as sketched in Figure 4.8. This is done in rounds, i. e., when the end of the sequence is reached, it starts again with the first element. Whenever ONE finds a better solution for H, we update $\mathcal{I}$ of G.

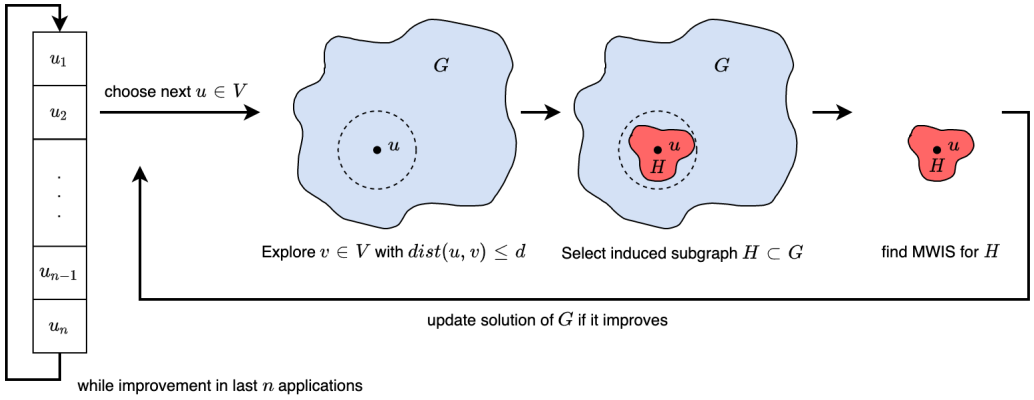


Figure 4.8: High-level view of *iterated local-search* with *optimal neighborhood exploration*.

For the initial solution we use a *greedy algorithm* described in Chapter 3 to find an initial maximal weight independent set or an initial solution is given in advance.
The algorithm terminates when there was no successful local-search in the last $n(G)$ applications. In this case, it is stuck in a local optimum and another round would consider the same instances without yielding any improvement. Therefore, we simply introduce a counter which starts counting applications when in some application no improvement was found. However, if one was found, we reset the counter to zero. If it reaches $n(G)$, it terminates. The algorithm terminates always, because the

solution only changes when a solution with a greater solution weight is found, and is bounded by the optimal solution weight. We call this algorithm Exhaustive Optimal Neighborhood Exploration (EONE). Depending on the algorithm used for the exploration, we call it EONE NBFS or EONE TBFS. We give the pseudo-code for EONE in Algorithm 2. It includes also a taboo-mechanism which we will discuss in the next section.

Run-time. In one round, i. e., in one scan over the vertices, we perform at most $n(G)$ ONE-applications. Therefore, the run-time of EONE per round is in worst-case bounded by the run-time of $n(G)$ ONE-applications. Since the solution cannot decrease in one round and improves it at least by one in the weight except for the last round, we perform at most $\omega(\mathcal{I}^*)+1-\omega(\mathcal{I}')$ rounds where $\mathcal{I}^*$ is a MWIS of G and $\mathcal{I}'$ is the initial maximal WIS.

4.4.1 Taboo-Mechanism

A bottleneck of the presented iterated local-search so far is that we solve an local induced subgraph for some vertex $u \in V$ again if it was already considered in the previous round and yielded no improvement. We cannot improve the global solution if did not improved it last time and consider it in the next round again. Especially the closer the algorithm comes to its local-optimum, it seems to be likely that we consider many instances do not change anymore and are considered for a potential improvement over and over again. In this case we do not want to solve non-improving local induced subgraphs again since it costs unnecessary run-time. Therefore, we propose a taboo-mechanism and implement it into EONE. The goal is to prevent as many unnecessary applications of the BaR solver while we only skip instances if we can be sure that we solved exactly the same instance before (in the last round) and found no improvement.

Algorithm 3 introduces the necessary functions to maintain an *active flag* which indicates a vertex changed its solution status (was removed or added) in the last round. We use a counter `ONECounter` which counts the ONE applications. Hence, we can store the application using `setActive` when the solution state for some vertex changed and finally, we can evaluate whether we observed in the last $n(G)$ applications (one round) a change using `isActive`. It remains to clarify when to use these methods and when we can skip an instance.

We skip an instance whenever there was no *active* vertex considered in any condition by EONE when determining H. Thus, we have to check for each seen vertex one more condition in the algorithm. Clearly, it does not change the overall worst-case complexity. We call `isActive` to evaluate the condition for a single vertex. For NBFS it is sufficient to call `setActive` for some $v \in V$ whenever its solution status changes. For TBFS we set its whole neighborhood active in addition, because we do not scan over the (changed) neighborhood of a vertex $v \in V$ to determine the

tightness, but use our maintained statistic. If a neighbor of v changes its solution status it must notify v as it can affect the tightness of v and therefore the decision whether v is covered in the local induced subgraph.

4.4.2 Kernelization

As we saw in Chapter 3 KerMIS by Dahlum et al. [14] combines kernelization with local-search very well. They use reductions to compute a kernel and operate with a heuristic approach on a much smaller graph. This allows the solver to consider an equivalent graph with less vertices.
Lamm et al. [28] presented new reductions for the weighted problem. Moreover, the authors devise a strategy for their state-of-the-art branch-and-reduce solver KaMIS BaR to compute an irreducible kernel. They incrementally apply the reductions until no reduction can be applied anymore. Therefore, it is checked for each vertex whether a reduction is applicable. If a reduction rule applies, the graph is modified, and when all checks finished for the current reduction rule, one goes back and starts with the first reduction rule again. Otherwise, the next reduction rule is chosen from the set of reduction rules. Most reduction rules act locally which allows to efficiently maintain a queue for each local reduction rule with vertices changed during graph modifications. If no reduction rule left which can be applied, we obtain an irreducible kernel.
Important to note is that a MWIS of kernel has a *weight offset* to an MWIS $\mathcal{I}$ of the actual graph G. Reductions as *neighborhood removal* [28] remove a dedicated vertex v from the graph and thereby modify the overall weight of the graph. If v has a weight $\omega(v) \geq \omega(\mathrm{N}(v))$, this reduction states that v is in some MWIS of G. Hence, v can be added to the solution and v and its neighbors can be removed from the graph. That increases the weight offset to the actual graph since v contributes $\omega(v)$ to $\mathcal{I}$ but is not covered in the reduced graph.
Remember that the run-time per round of EONE depends among the run-time of ONE on the number of vertices. It determines local induced subgraphs for every vertex for at least two rounds and solves them for at least one round. Moreover, ONE depends on the input graph even if $\beta < \infty$ because of the exploration. Due to the fact that EONE should operate on relatively large graphs and might consider many local induced subgraphs which do not yield an improvement, operating on kernels could close this potential bottleneck.
Moreover, thinking about solution quality, for vertices which can be efficiently decided optimally it is not worth it to risk to classify them wrong using an inexact algorithm. As Figure 4.9 sketches, we can compute a kernel using KaMIS BaR and then apply EONE. We investigate the effects on solution quality and efficiency in Section 5.4.

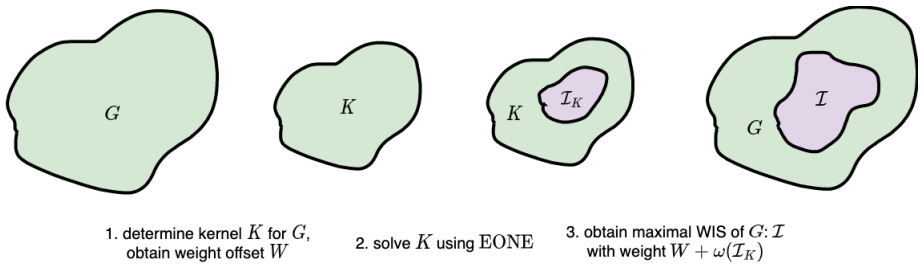


Figure 4.9: Example illustrating kernelization combined with EONE.

Algorithm 1: Exploration (NAIVE-BFS)

```
input  : G = (V,E,ω), maximal I ∈ IS(G), u ∈ V, 0 < ε ≤ 0.5
output: H ⊂ G[V_H]
V_H ← ∅ Q ← {u};
nodesCurrentDepth ← 1;
l ← 0;
queued[v] ← false ∀v ∈ V;
queued[v] ← true;
while Q ≠ ∅ do
    nodesNextDepth ← 0;
    while nodesCurrentDepth > 0 do
        v ← pop(Q);
        /* assert:  queued[v] = true                                   */
        if l < ⌊ε^-1⌋ then
            accept ← true;
            if l+1 = ⌊ε^-1⌋ ∧ v ∉ I then
                foreach w ∈ N(v) do
                    if !queued[w] ∧ w ∈ I then
                        /* assert:  w ∈ N^{d+1}(u)                     */
                        /* assert:  v cannot be tight to N^{≤d}(u)     */
                        accept ← false;
                        break;
            if accept then
                V_H ← V_H ∪ {v} for w ∈ N(v) do
                    if !queued[w] then
                        push(w, Q);
                        queued[w] ← true; ++nodesNextDepth;
        else
            /* assert:  l = ⌊ε^-1⌋                                     */
            /* assert:  v ∉ I                                          */
            if isTight(v,V_H) then
                V_H ← V_H ∪ {v}
    nodesCurrentDepth ← nodesNextDepth;
    ++l;
```

Algorithm 2: Exhaustive Optimal Neighborhood Exploration (EONE)

input : $G = (V, E, \omega)$, maximal $\mathcal{I} \in \mathrm{IS}(G)$, $0 < \varepsilon \leq 0.5$, $\beta > 0$
output: maximal WIS $\mathcal{I}'$
$\mathcal{I}' \leftarrow \mathcal{I}$ $\mathcal{V} \leftarrow$ shuffle(V);
noImprovementStartPos $\leftarrow \infty$;
currentPos $\leftarrow 0$;
`/* for taboo mechanism: */`
lastSolChange$[v] \leftarrow 0\ \forall v \in V$;
ONECounter $\leftarrow 0$;
while *noImprovementStartPos* $\neq$ *currentPos* **do**
  `/* Run ONE for` u`: */`
  $u \leftarrow \mathcal{V}$[currentPos];
  `/* 'explore' uses isActive */`
  H, taboo $\leftarrow$ explore$(u, G, \varepsilon, \beta$,lastSolChange, ONECounter);
  ++ONECounter;
  if *not taboo* **then**
    $\mathcal{I}_H \leftarrow solve(H)$
    if $\omega(\mathcal{I}_H) > \omega(\mathcal{I} \cap V(H))$ **then**
      `/* set swapped vertices (+ neighborhoods) active */`
      setActiveNodes($\mathcal{I}_H$, $\mathcal{I}'$, V, ONECounter, lastSolChange);
      $\mathcal{I}' \leftarrow (\mathcal{I} \setminus V(H)) \cup \mathcal{I}_H$;
      noImprovementStartPos $\leftarrow \infty$;
    else if *noImprovementStartPos* $= \infty$ **then**
      noImprovementStartPos $\leftarrow$ currentPos;
  ++currentPos;
  if *currentPos* $= n(V)$ **then**
    currentPos $\leftarrow 0$;

Algorithm 3: Components for taboo-mechanism

`/* ONECounter is incremented in every ONE application after exploring the` H `*/`
ONECounter $\leftarrow 0$;
lastChange$[v] \leftarrow 0\ \forall v \in V$;
Function `setActive(`$v \in V$`):`
  lastChange$[v] \leftarrow$ ONECounter;
Function `isActive(`$v \in V$`):`
  `/* was a vertex set active in the last` n `applications? */`
  return ONECounter $-$ lastChange$[v] < n(G)$;

CHAPTER 5

Experimental Evaluation

In this chapter we present experimental results considering our new iterated local-search solver. We start with giving an overview over the set of instances for our experiments in Section 5.1, followed by the methodology in Section 5.2. Section 5.3 gives an overview of the different experimental parameters of the solver we compare in the experiments and the metrics which we obtain and present. Finally, we move on to the experiments in Section 5.4.
We start by comparing our two approaches, NBFS and TBFS, in run-time in Section 5.4.1. Afterwards we investigate the solution quality and compare the found solutions to optimal solutions by KaMIS BaR by Lamm et al. [28] in Section 5.4.2. On a set of hard instances, we compare EONE with the state-of-the-art exact solver KaMIS BaR in Section 5.4.3. Finally, we compare EONE with an own implementation of the state-of-the-art solver METAMIS by Dong et al. [18] for the vehicle routing (VR) instances [16, 17] in Section 5.4.4. We reimplemented METAMIS as it was not publicly available. Thanks to the help of Alexander Noe, a co-author of [18], our implementation finds solutions very close to the ones presented in their experiments with METAMIS in [18]. The small gap in solution quality is probably related to the implementation itself or to fine-tuned optimizations we are not aware of. However, to make this experiment meaningful, we will mention the best solutions found by METAMIS in the respective experiments. In order to distinguish our implementation from their algorithm, we will denote our implementation METAMIS*.
At the end of the experiments, we will close with a conclusion in Section 5.5.

5.1 Instances

This section gives an overview over the set of instances which we consider.

5.1.1 Weighted DIMACS Instances

The first set of instances is a subset of DIMACS instances [9, 8]. Our subset consists of four street networks and five citation (social) networks. The street networks are very sparse. The average degree in these graphs is two. The social networks are a little denser. In average they have an average degree of 18. The original graphs are unweighted. We assigned them weights from 1 to 30 using the formula

$$\omega(v) = 1 + (v.id \mod 30)$$

where $v.id \in \{0, \ldots, n(G)-1\}$ denotes the node id of $v \in V$. Note that in our internal graph representation node ids are shifted by one. Therefore, node ids start to count from zero.

Instance	$n(G)$	$m(G)$	$d(G)$
citation networks			
citationCiteseer	268 495	1 156 647	9
coAuthorsCiteseer	227 320	814 134	7
coAuthorsDBLP	299 067	977 676	7
coPapersCiteseer	434 102	16 036 720	74
coPapersDBLP	540 486	15 245 729	56
GEOMEAN	336 003	2 954 412	18
OSM street networks			
belgium.osm	1 441 295	1 549 970	2
germany.osm	11 548 845	12 369 181	2
great-britain.osm	7 733 822	8 156 517	2
italy.osm	6 686 493	7 013 978	2
GEOMEAN	5 416 528	5 754 841	2

Table 5.1: Weighted DIMACS Instances. $d(\cdot)$ is the average degree of the instance.

5.1.2 Hard Instances

The second set contains twelve instances. All these graphs in Table 5.2 have in common that KaMIS BaR is not able to prove optimality or is not capable to find an optimal solution in a certain amount of time. The first instance, *fe_sphere-uniform*, is a 3d mesh obtained from simulations using the finite element method (FE) [21, 3]. KaMIS BaR was not able to solve the instance within 1000 s [21]. Further, we consider three *Open Street Map* (OSM) networks [2, 28], All these three graphs were not solved within 1000 s in experiments by Gellner et al. [21]. Same holds for the four

Instance	$n(G)$	$m(G)$	$d(G)$	$n(K)$	$m(K)$	$n(K)/n(G)$ [%]	$m(K)/m(G)$ [%]	$d(K)$
FE								
fe_sphere-uniform	16386	49152	6	15269	44641	93.18	90.82	6
OSM								
district-of-columbia-AM2	13597	1609795	237	6360	592457	46.78	36.80	186
greenland-AM3	4986	3652361	1465	3942	2348539	79.06	64.30	1192
rhode-island-AM2	2866	295488	206	1103	81688	38.49	27.65	148
SNAP								
as-skitter	1696415	11095298	13	9633	45665	0.57	0.41	9
loc-gowalla_edges	196591	950327	10	1102	5513	0.56	0.58	10
soc-LiveJournal1-uniform	4847571	42851237	18	29508	200915	0.61	0.47	14
wiki-topcats	1791489	25444207	28	187301	793723	10.46	3.12	8
SSMC								
ca2010	710145	1744683	5	167299	357449	23.56	20.49	4
il2010	451554	1082232	5	151815	316669	33.62	29.26	4
nh2010	48837	117275	5	12123	26384	24.82	22.50	4
ri2010	25181	62875	5	9707	22060	38.55	35.09	5

Table 5.2: Hard Instances: G is the instance itself and K is the irreducible kernel computed with KaMIS BaR, $d(\cdot)$ is the average degree of the instance.

SNAP instances from the *Stanford Large Network Dataset Collection* (SNAP) [29, 21] in the experiments by Gellner et al. [21] and by Gu et al. [24]. Finally, we added to the set four SSMC instances from *The Suite Sparse Matrix Collection* [25, 15]. KaMIS BaR was not able to find an optimal solution within 1000 s [24].

5.1.3 Vehicle Routing Instances

The last set are the vehicle routing instances by Dong et al. [16, 17]. We consider a subset containing 19 instances. These are the instances with smaller weights. They have around 10^5 vertices and 10^8 edges. They are conflict graphs where a solution to the MWIS problem corresponds to an optimal, conflict-free planning of routes for vehicles assigned to drivers and the load which should be transported [16, 17].

5.2 Methodology

EONE and METAMIS* are implemented in C++17. KaMIS BaR is implemented in C++11. We use KaMIS BaR with git commit `3d08a14` for comparisons and for solving local induced subgraphs produced by ONE. We computed kernels as input for EONE with KaMIS BaR using git commit `254fd16`. They were compiled using g++ (gcc) 9.4 with full optimizations turned on (-O3 flag). All experiments were executed on a machine with an Intel Xeon Silver 4216 16-Core CPU, 100W, 2.10GHz, 22.00MB L3 Cache, DDR4-2400, Turbo Core max. 3.20GHz, bis AVX-512 RAM 96GB (6x 16GB) DDR4-2933 DIMM, REG, ECC and 2R. The machine runs

Ubuntu 20.04.1 LTS and a Linux kernel 5.4.0-65-generic x86_64.

5.3 Setup

In the following, we introduce the experimental parameters of the compared solvers and metrics which we use to analyze them.

5.3.1 Experimental Parameters

In this section we give an overview over the experimental parameters which are used for all solvers and those specific to certain solvers. First of all, all algorithms support a time-limit tl which is used in the experiments. If the execution-time of an algorithm exceeds tl, it must return the best solution found so far. In the METAMIS*-experiments we will observe execution times larger than tl. The reason for this is that an initial solution is computed and the time-limit for EONE and METAMIS* bounds only the time spent in local-search improving this initial solution. Since both algorithms use the same greedy algorithm to find an initial solution, both results are biased to the same degree in run-time.

EONE

The exploration by ONE in EONE can be controlled by $\varepsilon \in (0, 0.5]$. A smaller ε leads to larger local induced subgraphs because vertices with larger distance to the seed vertex are explored. Besides ε, EONE TBFS supports a bound on the vertices $\beta \in \mathbb{N} \cup \{\infty\}$ covered by a local induced subgraph. In addition, we introduce a third parameter which is a time-limit tl_H for KaMIS BaR. This way, we ensure that the exact solver spends not to much time on a single local induced subgraph.

KaMIS BaR

Across all experiments KaMIS BaR is used in one way or another as a solver for local induced subgraphs, for precomputations or as competitor. There are two variants which differ in the *reduction style*. It can be either set to *normal/sparse* or to *dense/osm*. The difference is that the dense/osm mode omits costly reductions. Across the experiment, we will use all reductions. The reason is that we use KaMIS BaR for the local induced subgraphs which are typically relatively small. When we use KaMIS BaR as competitor we use a relatively large time limit such that it should be able to find (near)-optimal solutions.

METAMIS*

Thanks to the help of Alexander Noe, we know most of the values of the parameters for METAMIS used for the VR instances which we consider. Since we use only these particular values, we mention the best configuration for them in the following. METAMIS* allows to maintain a set of best solutions, so-called elite solutions. For the VR instances, METAMIS maintains only a single best solution as this seems to perform best. The authors apply no local-search on the random greedily selected solution used as guiding solution for path-relinking for one round. In the iterated local-search applied on the relinked solution, they limit the number of non-improving iterations by two. If no improvement was found at the end of one round of the iterated local-search, they perform a perturbation that adds a limited number (here 2) of vertices to the solution and removes their neighbors. As mentioned in Chapter 3, they incrementally apply (∗,1)-swaps, AAP-moves, (1,∗)-swaps, and if no improvement was found yet: one improving (2,∗)-swap in one iteration the local-search. We limit the number of AAP-move searches to 200 per round. The number of vertices in the path is limited to 20, and when the path is constructed but its gain in the solution weight exceeds a lower bound of −300, an improving prefix of the path, maximizing the gain, is used.

5.3.2 Metrics

EONE, METAMIS* and KaMIS are randomized, e. g., processing vertices in a shuffled order for random tie-breaking. The random mechanism is determined using an initial seed. In order to present metrics obtained from an execution independent of the random generator, we use three of such seeds. For plots showing the solution over time we average the best solutions at each certain point in time using the geometric mean. Let $w_1, \ldots, w_3$ be the bests solutions weights computed by some algorithm $\mathcal{A}$ using three seeds within the time limit tl. The *average best solution* w_{best} is the geometric mean of all w_i. Further, let $t_r(w_i)$ be the time $\mathcal{A}$ used to compute w_i. By abuse of notation, we will denote w_{best} as the best solution found by $\mathcal{A}$. The *average best solution report time* t_r is the geometric mean over all $t_r(w_i)$ as estimate for the run-time of the *best solution*. Again by abuse of notation, we will denote it as the *report time* of the best solution. Sometimes we compare a solution to a (near-)optimal solution to get an impression of how close we are to an (near-)optimal solution. This is done using the *solution quality* which is the ratio of the considered solution and the (near-)optimal solution.

5.4 Experiments

Because we presented two approaches, NAIVE-BFS and TIGHT-BFS, to explore and find an induced subgraph, we want to compare them. The main difference between them (disregarding β) is that TIGHT-BFS maintains the tightness globally and the tightness to the unaccepted vertices during exploration. As a result we do not need to perform tightness queries in $O(\Delta(G))$ for border vertices.
We start by comparing our algorithms for explorations: NBFS and TBFS. Therefore, we consider the set of weighted instances taken from the DIMACS challenge [9, 8].
The main question we want to investigate is whether we can find optimal solutions or at least near optimal solutions using these approaches of considering local-induced subgraphs. Therefore we compare our solution for the weighted DIMACS instances with (near-)optimal solutions computed by KaMIS BaR.
In a second experiment we want to compare EONE with the state-of-the-art exact solver KAMIS BaR by Lamm et al. [28]. Therefore, we consider instances which KaMIS BaR struggles to solve optimally or prove optimality for the computed solutions. Moreover, we computed irreducible *kernels* of these hard instances using KaMIS BaR. Solving the kernel optimally is equivalent to solving the actual instance optimally. Since the kernel is typically much smaller (in vertices and edges), we suppose that EONE is even more effective. We study the effects on solution quality and run-time comparing it with applying EONE directly on the instances.
Finally, we compare EONE with the state-of-the-art solver METAMIS* by Dong et al. [18] on a subset of the VR instances [16, 17].

5.4.1 EONE NBFS versus EONE TBFS

We start the experiments with a benchmark which compares EONE NBFS and EONE TBFS in run-time. To this end, we compare the time both algorithms need to converge to the same local optimum. In the experiment, we set $\varepsilon = 0.5$ and give KaMIS BaR a time limit of 1 s. The initial solutions are computed using a greedy algorithm which processes vertices in the descending order of their weights and adds them to the solution if feasible.
In Table 5.3 we report the best final solution weight ω_{best}. Both find the same best solution, because they consider the same local induced subgraphs for the same configuration if $\beta = \infty$. Moreover, it shows the report time of both variants.
Further, Table 5.3 reports the differences in the run-time between EONE NBFS and EONE TBFS. For almost all instances, there are just small differences in the run-time, except for *coPapersCiteseer* and *coPapersDBLP*. For these instances EONE NBFS is more than twice the time than EONE TBFS. These two graphs have an average degree of 74 and 56, respectively, which is significantly larger than the

average degree of the other graphs in this set. Comparing both, this first benchmark indicates that EONE TBFS performs better when it comes to denser graphs.

Instance	EONE NBFS		EONE TBFS		
	$\omega(\mathcal{I}_{\text{best}})$	t_r^{NBFS}	$\omega(\mathcal{I}_{\text{best}})$	t_r^{TBFS}	$t_r^{\text{NBFS}}/t_r^{\text{TBFS}}$ [%]
citation networks					
citationCiteseer	2503500	7.00	2503500	6.78	103.36
coAuthorsCiteseer	1651875	3.00	1651875	3.01	99.85
coAuthorsDBLP	2385565	3.77	2385565	3.90	96.59
coPapersCiteseer	943481	160.03	943481	65.28	245.13
coPapersDBLP	1320894	73.60	1320894	33.82	217.60
OSM street networks					
belgium.osm	11616209	12.46	11616209	12.86	96.82
germany.osm	93816891	111.58	93816891	118.35	94.29
great-britain.osm	63494427	72.99	63494427	75.57	96.59
italy.osm	53756998	62.43	53756998	66.61	93.72
GEOMEAN		26.25		22.27	117.88

Table 5.3: EONE NAIVE-BFS and EONE TIGHT-BFS applied on weighted DIMACS instances with $\varepsilon = 0.5$ and $t_H = 1\,\text{s}$.

We do this experiment again, but now setting $\varepsilon = 0.25$, and tl to 7200 s (two hours). The local induced subgraphs become larger because we allow to consider vertices of larger distance to the seed vertex. Remember that NBFS has to run (costly) tightness checks for border vertices. Because there are more border vertices, we should expect more tightness queries, and thus, observe a larger gap to TBFS in run-time. We give the best solution weight over time in Figure 5.1 for a street network, *germany.osm*, and for a social network, *coPapersCiteseer*. First, we observe that a local optimum for the street network is found within a few seconds. After 170 s both algorithms converged and EONE NBFS is only 15 s faster. Focusing on *coPapersCiteseer*, we can observe that the last improvement of EONE NBFS is found 1500 s earlier. Although this set of instances is fairly small, EONE TBFS seems to be faster than EONE NBFS if the underlying graph is not extremely sparse. In further experiments we will use EONE NBFS only for ε set to 0.5.

5.4.2 Performance With Weighted DIMACS Instances

In the following experiment, we want to investigate the solution quality of the maximal WIS found by EONE. The time limit is set for all algorithm for the DIMACS instances to two hours again. We determine optimal solutions using KaMIS BaR. In Table 5.4 we list the solution qualities for different $\varepsilon \in \{0.5, 0.25, 0.2\}$ for

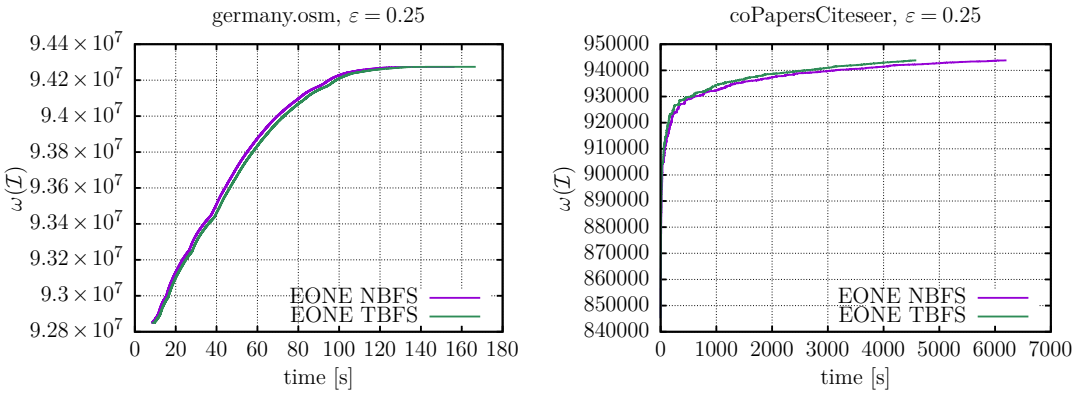


Figure 5.1: Improvements of solution weight over time: EONE NBFS and EONE TBFS for $\varepsilon = 0.25$ applied on *germany.osm* and *coPapersCiteseer*.

Instance	EONE TBFS						KaMIS BaR	
	$\omega_{\text{best}}^{\varepsilon=0.2}/\omega_{\text{opt}}$ [%]	$t_r^{\varepsilon=0.2}$ [s]	$\omega_{\text{best}}^{\varepsilon=0.25}/\omega_{\text{opt}}$ [%]	$t_r^{\varepsilon=0.25}$ [s]	$\omega_{\text{best}}^{\varepsilon=0.5}/\omega_{\text{opt}}$ [%]	$t_r^{\varepsilon=0.5}$ [s]	$\omega(\mathcal{I}_{\text{opt}})$	t_r^{BaR} [s]
citation networks								
citationCiteseer	100.00	4952.68	100.00	1071.89	99.85	6.78	2507336	0.42
coAuthorsCiteseer	**100.00**	430.19	**100.00**	61.55	99.99	3.01	1652034	0.18
coAuthorsDBLP	**100.00**	1635.47	**100.00**	135.34	99.98	3.90	2385956	0.26
coPapersCiteseer	99.66	7105.27	**100.00**	4584.43	99.96	65.28	943830	1.19
coPapersDBLP	99.82	7195.35	99.94	7178.56	99.97	33.82	1321337	1.24
OSM street networks								
belgium.osm	99.82	22.21	99.70	18.66	99.22	12.86	11707292	2.28
germany.osm	99.85	193.22	99.75	162.58	99.26	118.35	94515182	28.22
great-britain.osm	99.86	126.02	99.77	107.15	99.31	75.57	63934818	13.48
italy.osm	99.89	91.64	99.83	76.96	99.55	66.61	54000201	40.18
GEOMEAN	99.88	591.24	99.89	268.90	99.68	22.27		2.17

Table 5.4: Solution quality: EONE TBFS applied on weighted DIMACS instances; solution quality is bold if for at least one seed an optimal solution was found, the solution quality is rounded to the second digit.

EONE TBFS together with the optimal solutions found by KaMIS BaR. Moreover, we tested $\varepsilon = 0.1$. However, we leave $\varepsilon = 0.1$ out of consideration because it was not improving the solution quality while report times got worse. We discuss this behaviour in the following on the remaining choices for ε. Except for *italy.osm*, KaMIS BaR was able to prove optimality of the found solutions within seconds. In the case of *italy.osm*, it was not possible to prove optimality within two hours. Probably it is either optimal already and KAMIS was not able to prove optimality in the given time or it is a near-optimal solution. For optimal solutions we list the report time of the respective best solution.

We observe for EONE TBFS with $\varepsilon = 0.5$ that the solution weight already converges to over 99.9 % of the respective optimal solution for every instance. Typically, after a few seconds no improvement is found anymore and unable to find an optimal solution.

For $\varepsilon = 0.25$ EONE TBFS is able to report optimal solutions for three of five social networks. In case of the street networks, we are able to find better solutions in average if we use smaller ε, e. g.,. $\varepsilon = 0.25$. However, we are not able to find optimal solutions for the street networks for our choices of $\varepsilon \in \{0.5, 0.25, 0.2\}$.

We want to draw the attention to the report times of the best solutions. For $\varepsilon = 0.5$ the report times are fairly small compared to those for smaller ε. We apply ONE to very small local induced subgraphs if we choose $\varepsilon = 0.5$ and therefore they can be solved very fast. A local optimum is typically reached after a few seconds. For smaller ε this behaviour for the last improvement remains the same at least for street networks. The last improvement within two hours is found in less than 200 s. Considering social networks, we observe rapidly growing report times for the last improvement, while the it has no positive effect on the solution quality considering $\varepsilon = 0.2$ and $\varepsilon = 0.25$ These graphs are denser. Hence, for a fixed ε we consider significantly larger local induced subgraphs than compared to the street networks. These local induced subgraphs can cover large parts of the actual graph. Note that KaMIS BaR is able to solve these social networks very quickly. Since we use this solver for the local induced subgraphs too, it should have no problem to solve them also very fast. The reason for the larger report time for smaller ε is probably the accumulated run-time of more expensive exploration which selects large part of the graph and the subsequent solving if the instance is active (taboo mechanism). The effect on solution quality within these two hours becomes visible for *coPapersCiteseer* and $\varepsilon = 0.2$: EONE TBFS is unable to find an optimal solution within two hours.

In this context, another observation should not go unmentioned. Table 5.5 lists the solution quality of the greedy initial solutions used in this experiment. For the street networks we obtain very good solutions with the greedy solutions. Their solution weight is 98 % of the optimal solution weight. Thus, the improvement by EONE TBFS can only be relatively small. The solution quality for the social networks is 90 % of the optimal solutions on average. With $\varepsilon = 0.25$ we improve it to almost 100 %.

5.4.3 EONE Compared To KaMIS BaR

The results in Table 5.4 indicate that EONE is capable to find (near-)optimal solutions. However, KaMIS BaR reports optimal solutions in magnitudes faster than EONE does. The question arises whether EONE is an efficient approach and whether EONE is able to perform better than KaMIS BaR. The next experiment answers the question.

Instance	init. sol. for EONE $\omega_{\text{init}} / \omega_{\text{best}}$ [%]
citation networks	
citationCiteseer	90.57
coAuthorsCiteseer	91.23
coAuthorsDBLP	91.39
coPapersCiteseer	89.55
coPapersDBLP	89.61
GEOMEAN	**90.47**
OSM street networks	
belgium.osm	98.31
germany.osm	98.24
great-britain.osm	98.29
italy.osm	98.98
GEOMEAN	**98.45**

Table 5.5: Solution quality of greedy algorithm used by EONE TIGHT-BFS applied on weighted DIMACS instances.

Performance With Hard Instances

We consider the set of hard instances which EONE TBFS, EONE NBFS, and KaMIS BaR are going to solve with a time limit set to 7200 s. In order to find good configurations we tried $\varepsilon \in \{0.5, 0.25\}$. We ran EONE NBFS with $\varepsilon = 0.5$, but it was not faster than EONE TBFS. To shrink the space of configurations, we did not try other configurations with EONE NBFS. For $\varepsilon = 0.25$ we use different $\beta \in \{100, 200, 400, \infty\}$. Since for larger local induced subgraph it might take more time to find (near-)optimal solutions, we choose for $\beta \geq 200$ a time limit $tl_H = 3\,\text{s}$ besides the default time limit of 1 s. In Table 5.6 we report, next to the results for KaMIS BaR, the results received with the best configurations for EONE. These are configurations which found (for at least one instance) the best solution with the fastest report time.

First, we note that KaMIS BaR still finds improvements for half of the instances after two hours. For the remaining instances, KaMIS BaR typically reports the last improvement very fast, but is unable to prove optimality or the last found solution is not yet optimal. The best EONE TBFS configurations all use $tl_H = 1\,\text{s}$ instead of $tl_H = 3\,\text{s}$. For except two instances they find solutions of larger solution weights compared to KaMIS BaR. In particular, the configurations using $\varepsilon = 0.25$ with $\beta \in \{400, \infty\}$ perform very good for most of the instances. Especially noticeable is the mean report time over all instances when comparing these two configurations. Limiting the number of vertices in the local induced subgraphs for this set of instances seems to decrease the overall average report time from 268.41 s to 151.71 s while maintaining the solution quality. Given the fact that configurations with a smaller

tl_H perform better, we suspect that they can explore more local induced subgraphs within the overall time limit than the configurations with a larger tl_H. Therefore we can conclude using more local induced subgraphs compared to increasing the time spent on less induced subgraphs results in better solutions.

Instance	EONE TBFS								KaMIS BaR	
	$\varepsilon = 0.25$		$\varepsilon = 0.25,\ \beta = 400$		$\varepsilon = 0.25,\ \beta = 100$		$\varepsilon = 0.5$			
	ω_{best}	t_r [s]	ω_{best}	t_r [s]	ω_{best}	t_r [s]	ω_{best}	t_r [s]	ω_{best}	t_r [s]
FE										
fe_sphere-uniform	**616393**	1.56	**616393**	**1.53**	**616393**	**1.53**	606969	0.24	600017	1.00
OSM										
district-of-columbia-AM2	198339	5876.07	**209113**	**1677.11**	207091	10.89	207589	275.43	196515	7200.02
greenland-AM3	12646	138.61	13521	373.63	12828	77.69	**13959**	**5700.51**	13828	93.56
rhode-island-AM2	**184596**	1828.78	184084	106.04	182517	1.56	**184596**	**12.49**	**184596**	19.13
SNAP										
as-skitter	123763358	7169.79	123669674	5425.79	123524336	1651.09	123940661	117.45	**123982623**	**7200.59**
loc-gowalla_edges	**12276822**	**4903.80**	12275842	185.04	12274127	101.07	12261793	3.59	12276794	0.37
soc-LiveJournal1-uniform	283488236	6257.46	283598126	6075.98	283885744	7128.30	283646716	788.21	**284009063**	**179.26**
wiki-topcats	106572372	7072.11	**106620081**	**7195.83**	106420771	3927.79	106447505	133.44	106323717	225.37
SSMC										
ca2010	**16854514**	69.63	**16854514**	**66.55**	16853409	62.15	16661165	10.80	16571662	7200.22
il2010	**5992685**	38.94	**5992685**	**37.98**	5992604	38.03	5925427	6.00	5852319	7200.13
nh2010	**588837**	5.06	**588837**	4.82	**588837**	**4.61**	578444	0.71	581615	7200.02
ri2010	**459146**	2.82	**459146**	**2.75**	459097	2.65	452617	0.35	446952	7200.01
GEOMEAN		268.41		151.71		50.48		19.72		352.66

Table 5.6: Results for hard instances received with best configurations of EONE TBFS versus KaMIS BaR, $tl = 7200$s, every EONE TBFS configuration uses $tl_H = 1$s, best solutions weights globally are bold, the report time is bold if it is the smallest for the global best solution weight.

Instance	EONE TBFS applied on kernels									
	$\varepsilon = 0.25,\ \beta = 400,\ *$		$\varepsilon = 0.25\ \beta = 400$		$\varepsilon = 0.25\ \beta = 200$		$\varepsilon = 0.25\ \beta = 100$		$\varepsilon = 0.5$	
	ω_{best}	t_r [s]	ω_{best}	t_r [s]	ω_{best}	t_r [s]	ω_{best}	t_r [s]	ω_{best}	t_r [s]
FE										
fe_sphere-uniform	**616628**	2.02	**616628**	**2.01**	**616628**	**2.01**	**616628**	2.03	607206	0.86
OSM										
district-of-columbia-AM2	**209132**	5882.25	208027	5689.86	**209132**	**419.51**	207930	34.25	205886	51.70
greenland-AM3	13521	621.26	13521	657.80	13294	114.15	13125	65.69	**14011**	**1901.23**
rhode-island-AM2	**184596**	202.73	**184596**	145.24	183653	5.62	183653	1.27	184470	4.01
SNAP										
as-skitter	123991215	6916.78	123992049	7109.46	123992437	3912.09	**123993262**	**301.52**	123988977	62.48
loc-gowalla_edges	**12276929**	17.06	**12276929**	**16.14**	12276862	0.62	12276854	0.33	12276444	0.24
soc-LiveJournal1-uniform	284009584	5408.66	284008442	5291.34	284030469	7001.43	**84035821**	**396.01**	284026010	430.85
wiki-topcats	**106659115**	**6639.12**	106659091	3104.65	106658256	504.82	106652105	267.76	106570377	197.53
SSMC										
ca2010	**16856943**	46.62	**16856943**	45.82	**16856943**	**45.74**	**16856943**	45.93	16748979	29.63
il2010	**5993472**	34.75	**5993472**	34.73	**5993472**	34.70	**5993472**	**34.66**	5949127	23.95
nh2010	**588909**	2.08	**588909**	2.07	**588909**	2.07	**588909**	**2.04**	584397	0.81
ri2010	**459158**	1.85	**459158**	1.78	**459158**	1.79	**459158**	**1.77**	455893	0.55
GEOMEAN		135.84		123.04		41.68		16.32		15.04

Table 5.7: Results for hard instances received with best configurations of EONE TBFS on kernels, $tl = 7200$s, $tl_H = 1$s or $tl_H = 3$s when marked with a '$*$', global best solution weights including Table 5.6 are bold, the report time is bold if it is the smallest for the best solution weight including Table 5.6.

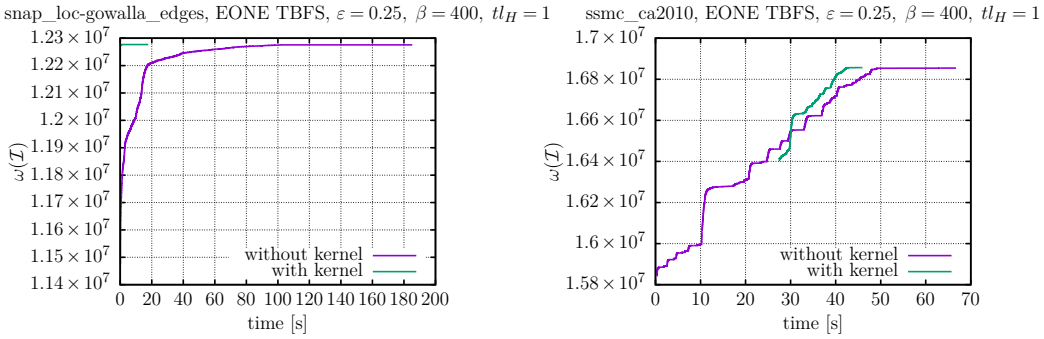


Figure 5.2: Improvements of solution weight over time with and without kernelization.

Local-Search Using Kernelization

This first benchmark shows the effectiveness of EONE TBFS comparing to KaMIS BaR. We suppose that EONE becomes even more effective overall as discussed in Section 4.4.2 if it is applied to the irreducible graphs with the previous experimental configurations. We computed kernels for EONE using KaMIS BaR with all available reductions (normal/sparse). This way, many vertices are decided already optimal comparing with the kernel sizes in Table 5.2 and EONE 'focuses' on the hard-to-decide problem kernel.
In Table 5.7 we report the best configurations (which slightly changed) for the kernels of the instances. The best report time takes the reduction time used to compute the kernel into account and for the best solution weights we added the weight offset between solution of the actual graph and the computed kernel such that we obtain metrics for the hard instances.
First, we note that for almost every instance (except for *osm_rhode_island-AM2*) the average solution weight was increased by some configuration compared to the best solution from Table 5.6. For *osm_rhode_island-AM2* no further improvement was possible because in the first experiment, we already found an optimal solution [21].
Furthermore, we note that these best solution weights are now obtained with slightly changed configurations. The best configurations seem to be the ones with $\varepsilon = 0.25$, $\beta = 400$ and $tl_H \in \{1,3\}$. As one can see, the configuration with $tl_H = 3$ found the best solution weight for nine of twelve instances. The other three instances, *osm_greenland-AM3*, *snap_as_skitter*, and *snap_soc-LiveJournal1-uniform* were improved with $\varepsilon = 0.5$ and $\varepsilon = 0.25$ using $\beta = 100$.
Moreover, the experiment indicates that the mean report time for the same configurations decreases when we apply local-search to the kernel while maintaining solution quality. Comparing the configuration with $\varepsilon = 0.25$, $\beta = 400$ and $tl_H = 1$ used for the kernel and for the input graph, we observe the mean report time of the best solution decreased from 151.71 s to 123 s. Despite the fact this is a small decrease, note that, applied to the kernel, it finds for eleven instances a better solution weight. These

solution weights are at least as good when not taking the kernel as input.
We compare the development of the solution weight over time with and without kernel in Figure 5.2 for two instances. The first instance, *snap_loc-gowalla_edges*, is reduced very quickly to a kernel which has only 0.6 % of the original size. EONE TBFS finds small improvements and is able to find better solutions than without kernelization in seconds. Without the kernelization it takes more than 100 s to come close to the solution quality with kernel in Figure 5.2. Secondly, we consider *ssmc_ca2010*. It takes a littler longer to find the first solution because the reduction time is longer (28 s) as shown in Figure 5.2. Furthermore, the computed kernel is not as small as for the last instance. It still has 24 % of the number of vertices. Nonetheless, we find out that EONE TBFS is able to find many good improvements in a short time for *ssmc_ca2010*. And thus, overtakes the solution of the execution without kernelization.

5.4.4 Solving The VR Instances

In our last experiment we consider the VR instances and compare our approach with state-of-the-art METAMIS for these instances. Furthermore, we investigate the question whether EONE TBFS has a potential to improve the solution quality further if METAMIS has already found a near-optimal solution. Therefore, we combine both by applying them sequentially: first METAMIS* and then EONE TBFS.

With Greedy Solutions

In the first benchmark in Table 5.8 we present initial solutions using the adaptive greedy algorithm devised by Dong et al. [18]. All algorithm have a time-limit set to $tl = 7200$ s to improve the initial solution.
We tried a configuration with $\varepsilon = 0.5$ for both exploration algorithms applied on the instance *CR-S-L-4* but EONE TBFS was the better choice because it finds a solution weight of 5 644 909 after 4480 s, whereas EONE NBFS uses the full two hours to compute a solution with a weight of 5 580 128. This solution is still over 1 % behind the solution of EONE TBFS. Therefore, we use only EONE TBFS.
TBFS uses in addition $\varepsilon = 0.25$ with $\beta \in \{100, 250\}$. For all configurations tl_H is set to 1 s. The reason that we did not try $\varepsilon = 0.25$ with $\beta = \infty$ is that the local induced subgraphs become very larger and thus, make improvements hard to find even if we increase tl_H to several minutes. The combined algorithm, METAMIS* + EONE TBFS, executes METAMIS* for one hour, then takes the best solution and uses it as input for EONE TBFS to find further improvements. Table 5.8 shows results of the best configurations. First, we note in Table 5.8 that our implementation of METAMIS reaches a mean solution quality of 99.55 %, compared to the solution by Dong et al. [18], after 3600 s. For the instance *CW-*

T-D-6 METAMIS* was able to find a better solution. Our best configuration for EONE TBFS ($\varepsilon = 0.25$ and $\beta = 100$) is slightly behind METAMIS* with respect to the solution weights. Most interesting is the combination of both using $\varepsilon = 0.25$ and $\beta = 100$. It finds better maximal WIS than the standalone variants. Only for *CW-T-D-6* a better one than in the work by Dong et al. [18] was found.
Figure 5.3 shows the solution improvement of *CR-S-L-4* over time among the best configurations and in addition: EONE TBFS for $\varepsilon = 0.5$. Using $\varepsilon = 0.5$ a local optimum is reached after 3000 s. Moreover, we can observe a boost in the solution weight after one hour, when METAMIS* stops and EONE TBFS takes over. Without METAMIS*, EONE TBFS performs noticeably worse.

With Warm-Start

We do this experiment again but with good initial solutions instead of a greedy solution. This is what Dong et al. [18] call a *warm-start*. The initial solutions are significantly better than the ones computed by METAMIS after one hour without warm-start [18]. It should be more challenging for a local-search solver to find further improvements because the solution is better.
Table 5.9 shows the results when performing a warm-start. The mean solution quality of our implementation of METAMIS is now slightly better than in the first benchmark. However, no variant produces better maximal WISs than the solutions presented by Dong [18]. Comparing our implementations, we can observe that except for three instances every instance was improved by a combination of both. Again a boost in the solution weight is noticeable in Figure 5.4 for *CR-S-L-4* after one hour.

Instance	METAMIS* + EONE TBFS $\varepsilon = 0.25,\ \beta = 250$		METAMIS* + EONE TBFS $\varepsilon = 0.25,\ \beta = 100$		EONE TBFS $\varepsilon = 0.25,\ \beta = 100$		METAMIS*		METAMIS best w.o. warm-start, $tl = 1\,\mathrm{h}$		Greedy Init.	
	ω_{best}	t_r [s]	ω_{best}	t_r [s]	ω_{best}	t_r [s]	ω^*_{best}	t_r [s]	ω_{best}	$\omega^*_{\text{best}}/\omega_{\text{best}}$ [%]	ω_{best}	t_r [s]
CR-S-L-1	5567502	7240	**5570283**	7244	5545419	7253	5561556	7263	5588489	99.52	5096089	56
CR-S-L-2	**5663772**	7256	5662829	7256	5637803	7232	5656089	7267	5691892	99.37	5186006	60
CR-S-L-4	5652518	7258	**5657505**	7240	5628028	7251	5646210	7268	5681336	99.38	5135960	60
CR-S-L-6	3844410	7219	**3846531**	7215	3831797	7199	3839980	7239	3859513	99.49	3498940	33
CR-S-L-7	1982361	7099	**1984630**	7147	1979908	7033	1981704	7212	1989879	99.59	1792103	10
CR-T-C-1	4631382	7224	**4636165**	7226	4626022	7224	4626933	7239	4654419	99.41	4244943	32
CR-T-C-2	4847671	7229	**4851259**	7201	4839717	7214	4839881	7242	4874346	99.29	4436701	38
CR-T-D-4	4794542	7234	**4797668**	7223	4782798	7214	4785953	7243	4817281	99.35	4382829	38
CR-T-D-6	2956501	7178	**2958802**	7124	2948265	7139	2953663	7218	2970011	99.45	2701284	17
CR-T-D-7	1435476	7175	**1437980**	7016	1433223	6452	1434836	7204	1440281	99.62	1306503	4
CW-S-L-1	1629500	7174	**1630123**	7191	1621688	7197	1629588	7234	1634950	99.67	1497216	29
CW-S-L-2	1703917	7163	**1704070**	7156	1691248	7210	1703496	7237	1708820	99.69	1556743	30
CW-S-L-4	**1720063**	7233	1719985	7162	1708295	7063	1718673	7237	1725591	99.60	1566333	32
CW-S-L-6	1153217	7136	**1154227**	7082	1149341	7199	1153907	7217	1158925	99.57	1050778	15
CW-S-L-7	584452	6986	**584518**	6852	583653	6812	584276	7205	587288	99.49	528774	4
CW-T-C-1	1312312	7136	**1313025**	7064	1309662	7027	1312990	7216	1317775	99.64	1200391	13
CW-T-C-2	927594	7139	**928227**	6983	923822	7035	927457	7210	931802	99.53	846578	8
CW-T-D-4	456101	7029	**456517**	6811	456390	5091	456079	7203	457185	99.76	416064	1
CW-T-D-6	457756	7064	**457947**	6227	456909	6207	457869	7202	457790	100.02	418485	1
GEOMEAN	1996613	7167	1997855	7071	1990503	6928	1995464	7229	2004501	99.55	1821765	16

Table 5.8: Results for VR instances without warm-start: METAMIS* + EONE TBFS, EONE TBFS and METAMIS* for $tl = 7200\,\mathrm{s}$ compared with best results with METAMIS by Dong et al. [18] without warm-start after 3600 s. The best solution in our experiments is bold (ignoring METAMIS).

Instance	METAMIS* + EONE TBFS $\varepsilon = 0.25,\ \beta = 250$		METAMIS* + EONE TBFS $\varepsilon = 0.25,\ \beta = 100$		EONE TBFS $\varepsilon = 0.25,\ \beta = 100$		METAMIS*		METAMIS best w.o. warm-start, $tl = 1\,\mathrm{h}$		WARM-START
	ω_{best}	t_r [s]	ω_{best}	t_r [s]	ω_{best}	t_r [s]	ω^*_{best}	t_r [%]	ω_{best}	$\omega^*_{\text{best}}/\omega_{\text{best}}$ [s]	ω_{init}
CR-S-L-1	**5 672 275**	7 188	5 672 098	7 127	5 663 661	7 177	5 666 856	7 206	5 690 515	99.58	5 547 904
CR-S-L-2	**5 762 319**	7 145	5 760 921	7 178	5 752 253	7 157	5 753 945	7 208	5 780 449	99.54	5 617 351
CR-S-L-4	**5 755 718**	7 198	5 754 740	7 121	5 748 562	7 152	5 748 560	7 205	5 775 704	99.53	5 629 351
CR-S-L-6	3 923 765	7 123	**3 924 214**	7 132	3 921 539	7 155	3 921 281	7 204	3 935 089	99.65	3 841 538
CR-S-L-7	2 013 152	7 032	**2 013 929**	7 027	2 013 129	6 753	2 012 436	7 201	2 017 836	99.73	1 969 254
CR-T-C-1	4 728 055	7 195	**4 730 271**	7 195	4 725 304	7 073	4 723 760	7 204	4 738 289	99.69	4 605 156
CR-T-C-2	4 948 226	7 171	**4 950 076**	7 165	4 948 782	7 179	4 942 716	7 206	4 964 446	99.56	4 844 852
CR-T-D-4	4 896 615	7 133	**4 897 357**	7 189	4 891 424	7 154	4 889 946	7 206	4 909 999	99.59	4 789 561
CR-T-D-6	3 016 850	7 159	**3 017 250**	7 017	3 014 993	7 061	3 013 730	7 203	3 023 349	99.68	2 953 177
CR-T-D-7	1 458 071	6 403	**1 458 261**	6 243	1 457 166	3 793	1 457 669	7 201	1 459 948	99.84	1 451 562
CW-S-L-1	1 654 881	7 010	**1 655 170**	6 835	1 654 402	7 020	1 654 100	7 203	1 660 815	99.60	1 622 723
CW-S-L-2	1 731 673	7 159	**1 731 964**	6 975	1 727 378	7 185	1 731 326	7 203	1 736 245	99.72	1 692 255
CW-S-L-4	1 748 091	7 139	**1 748 342**	7 070	1 743 700	7 023	1 747 388	7 203	1 751 988	99.74	1 709 043
CW-S-L-6	1 173 756	7 113	1 173 835	6 974	1 172 263	7 130	**1 173 888**	7 203	1 176 233	99.80	1 159 946
CW-S-L-7	**593 153**	6 595	593 004	5 976	592 611	5 694	593 008	7 201	593 947	99.84	589 723
CW-T-C-1	1 333 524	7 142	**1 333 675**	6 978	1 331 266	6 810	1 333 080	7 203	1 336 563	99.74	1 298 968
CW-T-C-2	943 330	6 997	943 577	7 115	942 943	6 532	**943 591**	7 202	945 565	99.79	933 792
CW-T-D-4	460 611	5 146	460 659	5 589	460 291	4 086	**460 734**	7 201	461 025	99.94	457 715
CW-T-D-6	460 813	5 369	**460 869**	5 215	460 424	5 595	460 809	7 201	461 174	99.92	457 605
GEOMEAN	2 030 254	6 835	2 030 491	6 769	2 028 227	6 466	2 029 149	7 203	2 035 051	99.71	1 994 139

Table 5.9: Results for VR instances with warm-start (good initial solution given): METAMIS* + EONE TBFS, EONE TBFS and METAMIS* for $tl = 7200\,\mathrm{s}$ compared with best results with METAMIS by Dong et al. [18] with warm-start after 3600 s. The best solution in our experiments is bold (ignoring METAMIS).

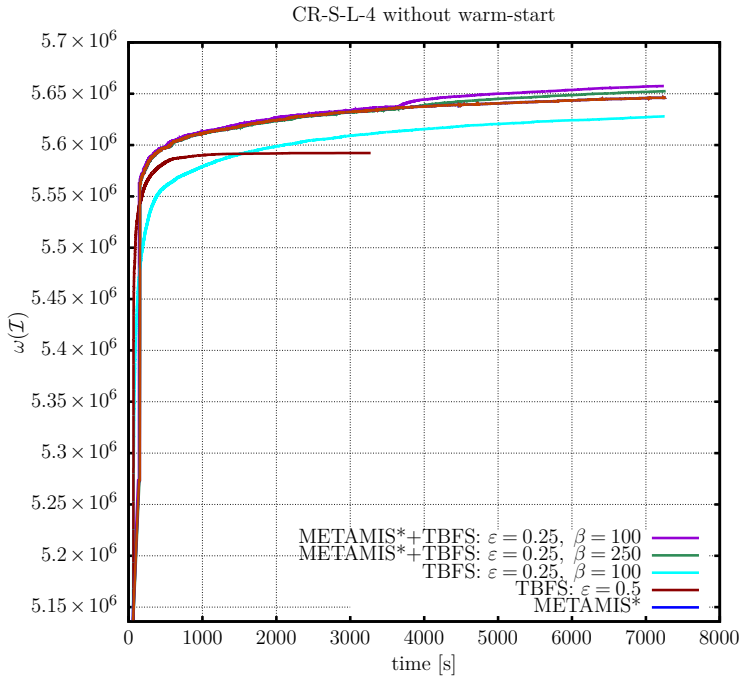


Figure 5.3: Solution improvement over time for *CR-S-L-4* **without** a warm-start; besides the best configuration is shows EBNF with $\varepsilon = 0.5$ which stucks in a local optimum after 3300 s.

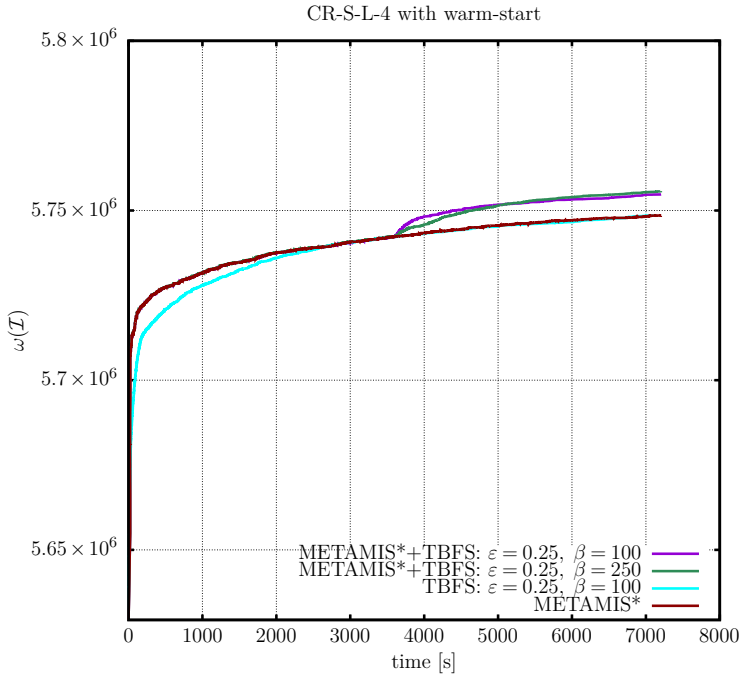


Figure 5.4: Solution improvement over time for *CR-S-L-4* **with** a warm-start with the best configurations.

5.5 Conclusion

To summarize the experiments: we have shown that EONE and in particular our new local-search technique ONE is a working approach to find (near-)optimal solutions by considering a subset of the DIMACS instances. As we supposed, exploring neighborhoods to a larger distance given a seed vertex is a decisive factor in finding optimal solutions. However, making ε too small yields larger instances, and makes them harder to solve with KaMIS BaR. For eight of twelve hard instances the experiments have shown that EONE TBFS finds better solutions than the state-of-the-art exact solver KaMIS BaR given a time limit of two hours. On average our best solutions are 1.52 % better than the ones received with KaMIS BaR. Applying EONE on kernels of these hard instances allows us to further improve the solutions to 1.57 % compared to the solutions received with KaMIS BaR. Not only the improvement of solution quality underlines the benefits of kernelization combined with our approach, but also the report times get better because many vertices are often decided optimally in advance. Thus, EONE operates on a smaller problem. It should not go unmentioned that β is a crucial factor, to shrink the sizes of subgraphs when graphs get denser because the number of explored vertices rapidly gets larger. During this work, the development of EONE TBFS which incorporates β was essential to scale to large instances,e. g., the VR instances because they not only have hundred of thousand of vertices but also millions of edges. Without β considering larger instances as $\varepsilon = 0.5$ was not feasible. The last experiment has shown that EONE TBFS not only finds high-quality solutions but also improves the solution of METAMIS* after improving the initial solution with METAMIS* for one hour. Overall we received slightly better solutions than the standalone variant.

CHAPTER 6

Discussion

The last chapter is dedicated to the discussion of our new local-search technique. We conclude this thesis and give an outlook on future work.

6.1 Conclusion

In this thesis, we have developed and implemented a new local-search technique, called *optimal neighborhood exploration* (ONE), for the *maximum weight independent set problem*. Particular about this technique is that ONE generalizes known local-search techniques by not only considering the immediate neighborhood of a seed vertex but a large-scaleable neighborhood to find improvements. These explored neighborhoods can be considered as induced subgraphs and then solved optimally by any MWIS solver. Improving the local solution of the subgraphs always corresponds to an improvement to the same extent globally. To ensure that a better maximal WIS for the local induced subgraph results globally in an improvement, we introduced two boundary rules: the *border policy* and the *dynamic border policy*. These boundary rules were the building blocks for the two *exploration* algorithms, NBFS and the more sophisticated algorithm: TBFS. NBFS has a worst-case complexity of $\mathcal{O}(\Delta^{d+1})$ for selecting the set of vertices for the local induced subgraph, whereas TBFS has worst-case complexity $\mathcal{O}(\Delta^{d})$ where Δ is the maximum degree of G. We solve the local induced subgraphs using KaMIS BaR. By bounding the number of vertices in the local induced subgraph by β, the worst-case complexity for solving the subgraph is independent of G and only depends on β. As a result, solving the local induced subgraphs is fixed-parameter tractable.
Starting from ONE, we have built an iterated local-search solver called Exhaustive Optimal Neighborhood Exploration (EONE). To avoid solving local instances whose vertices did not change the solution status in the last iteration, we devised a *taboo mechanism* that avoids solving instances twice.

Through the evaluation of our extensive experiments, we have shown that EONE can find high-quality maximal WISs. Using TBFS for exploration seems to be the way-to-go option compared to NBFS. Especially for ε smaller 0.5 we often found optimal solutions, which underlines the effectiveness of considering neighborhoods of larger distance. Using a bound on the number of vertices often results in solutions better than those found by KaMIS BAR within a certain time limit. Using kernelization to find a kernel in advance and solving it with EONE improves the solution quality while the report time of the best solution decrease on average. Considering the VR instances, we realized that we do not outperform the state-of-the-art solver METAMIS yet. However, we have shown that EONE TBFS finds high-quality maximal WISs on its own and can boost the solution quality when it is applied after running METAMIS* for one hour. We consider this a success because it shows METAMIS* misses potential improvements that EONE can find.
In conclusion, this work has produced an effective and competitive iterated local-search solver that can be used for large graphs modeling real-world applications by implementing our new local-search technique. The basis of this success is the *optimal neighborhood exploration* of a graph.

6.2 Future Work

The experiments have shown, that while EONE can find (near-)optimal solutions, there is still unused potential for further improvements. To this end, we discuss them and explain how we plan to improve EONE in the future.
The most significant drawback so far is that EONE processes the ONE-applications sequentially. The closer it comes to its local optimum, the longer sequences of non-improving ONE-applications are. Although the taboo-mechanism helps skipping instances which are not improvable, we still have to figure out for that an instance is not improvable by solving it as least once.
Parallelization probably is a solution to this problem. It is of great potential because we can consider relatively small subgraphs and solve them independently of each as long as there is no edge between them and the vertex sets are disjoint. Figure 6.1 illustrates how multiple local induced subgraphs are solved in parallel. To do so, we need to select a set of vertices that are used as seed vertices for ONE in one iteration. One approach could be to select this set uniformly at random. In the case instances cannot be solved independently, we could solve them nonetheless. We accept the change with the most largest improvement among the instances which are in conflict. Another approach for resolving conflicts could be to marry the graphs with a conflict and solve a larger local induced subgraph. However, there might be a risk that the local-induced subgraphs become too big.
A second approach to selecting the seed vertices could be to compute a vertex-coloring

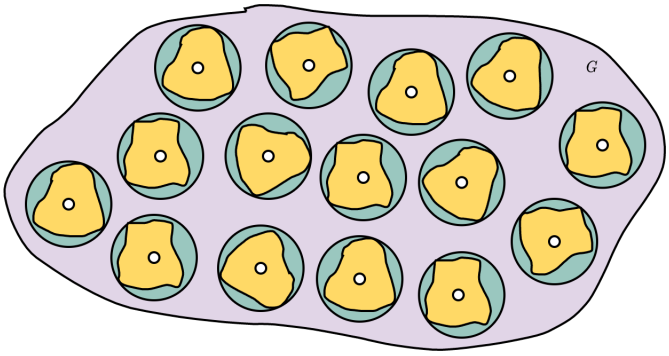


Figure 6.1: Example illustrating how EONE can be accelerated: solving local induced subgraphs in parallel.

in advance as in Figure 6.2. The vertex-coloring is feasible if vertices have the same color only if their distance is at least $2 \cdot \lfloor \varepsilon^{-1} \rfloor + 1$. Vertices of the same color can be used as seed vertices for parallel execution because ONE will consider for each seed vertex at most vertices with a distance of at most $\lfloor \varepsilon^{-1} \rfloor$. That is to say, the vertices are located far enough from each other located.

In this work, we have restricted ourselves to the state-of-the-art exact solver KaMIS BaR. For the experiments, we considered, among others, instances which are hard to reduce for KaMIS BaR comparing with Table 5.2. Large kernels imply more branching. Thus, it takes more time to solve them optimal. As mentioned in Chapter 3, Lamm et al. [21] recently advanced KaMIS BaR with *increasing transformations*, called *structions*, which are exact reductions. The idea is to increase the

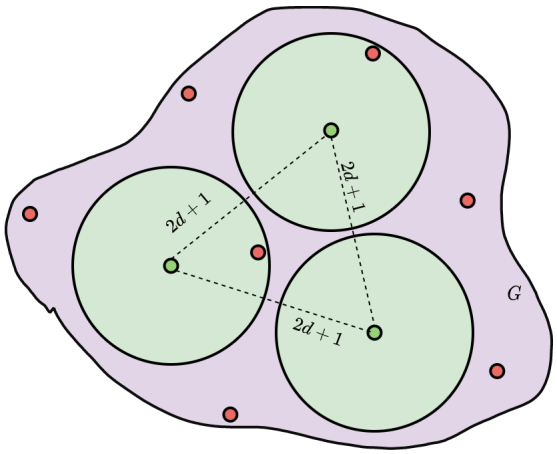


Figure 6.2: Example shows two colors (red and green) of a vertex-coloring where vertices with distance $\geq 2d+1$ have the same color, green vertices are currently used as seed vertices for local induced subgraphs.

size of the current kernel to a certain threshold to find a smaller kernel when *decreasing* reductions are applied again. They successfully solve most of the instances within seconds. For the few left, it would be interesting to use the advanced *KaMIS BaR* solver. This could help to solve instances obtained with a smaller ε in a faster time. After all, it would be interesting to find out how EONE TBFS performs on the VR instances, comparing it with the current best solutions computed by METAMIS by Dong et al. [18].

Implementation Details

This chapter should give an overview over the implementation details. In particular we make notes on the data structures and operation which were used and sketch how EONE can be implemented efficiently.

Breadth-First-Search (BFS)

Roughly speaking, NBFS and TBFS are modified breath-first-search implementation. We give the pseudo-code for NBFS in Algorithm 1. This difference in the pseudo-code from our implementation is that this pseudo-code might not scale very well to many applications of ONE as in EONE.
EONE applies ONE exhaustively until a local optimum is (ideally) reached, i. e.,. we apply the BFS very often. In the pseudo-code, we remember whether a vertex was enqueued or not by setting a boolean flag to true in a vector of size $n(G)$. Allocating and initializing this vector has a worst-case complexity of $\mathcal{O}(n(G))$. This is infeasible for many applications. Therefore, we maintain a counter of the ONE-application together with a vector of integers of size $n(G)$ in a dedicated data-structure. Whenever we perform a BFS, we increment this counter. When a vertex is enqueued we store the value of the counter in the vector in order to remember that this vertex was explored.
Since we know $n(G)$ vertices can be enqueued at most in one BFS, we allocate the storage for the queue in advance using a vector. Instead of clearing the vector for a second BFS, we just overwrite the entries of the vector.
We maintain this queue in the dedicated data-structure.

Maintaining Tightness

For TBFS we use look-ups for the tightness instead of computing it on-demand. First of all, we maintain for each vertex $v \in \mathrm{V}(G)$ the tightness $\tau(v)$ which is the number of neighbors of v which are part of the solution. Before the first ONE application, we initialize $\tau(v)$ to zero for each $v \in V$, and scan once over the set of the vertices.

If v is part of the solution, we visit its neighbors and increment their tightness. If the solution now changes we can increment or decrement the tightness of the respective vertices.
When we perform a BFS and check for marriages, we want to do look-ups as *is v tight to H* where H is the current local induced subgraph. That is to say, we want to know how many neighbors of v, that are part of the solution, are covered by H. Therefore, we maintain the number of solution vertices in the neighborhood of H which are **not** covered. We denote it $\tau_{\neg H}(v)$ and initialize $\tau_{\neg H}(v)$ to $\tau(v)$ when a solution neighbor is covered for the first time. Note, we do not change τ during exploration.
Whenever we cover a solution neighbor of a non-solution vertex u, we decrement $\tau_{\neg H}(v)$. If now $\tau_{\neg H}(v)$ becomes zero, we know it is tight to the current H and we must cover it (when processing solution vertices).
That should give an intuition of maintaining tightness efficiently. However, in the actual implementation we go one step further and maintain in addition the tightness to the precomputed subgraph $\tilde{H}$. Since the idea is the same as the idea of maintaining $\tau_{\neg H}(v)$ we do not elaborate on this.

Command-Line Arguments

The following is an excerpt of the Command-Line Interface of the application implementing EONE and METAMIS*. In the later case, they are executed sequentially. To keep this interface clearly arranged, we will just list the options and an example from our experiments which demonstrate how to use this application. For the exact grammar of the commands we refer to the repository of *DynWMIS*.

Command-Line Interface

In the following we list the arguments for the binary of *DynWMIS* which can be used together with `solve` and `batch-solve`. To see a full help message use `-h`. Roughly speaking, it allows one to apply EONE and METAMIS* as standalone solvers, as well as arbitrarily configured batches of solvers. One can read in given a solution from a file for a warm-start, or use a greedy algorithm which were described in the experiments Section 5.4 and in the Chapter 3. Note, default values cannot be set if `bath-solve` is used.

- `--seed=<int>` set random seed [default: 1]
- `--time_limit=<double>` time_limit [default: 0]
- `--solver=<string>` EONE_SOLVER, METAMIS_SOLVER [default: EONE_SOLVER]
- `--rating=<string>` determine greedy solution using a heuristic: WEIGHT_RATING, WEIGHT_DEG_RATING [default: WEIGHT_RATING]
- `--rel_rand_size=<double>` relative (to number of nodes) size of random set with best nodes with respect to rating [default: 0.1]
- `--adaptive_greedy` determine adaptive greedy solution using heuristic WEIGHT_DEG

- `--eone` EONE (iterated local-search) config
- `--max_ONE=<unsigned>` set upper bound for ONE applications [default: 0]
- `--max_none_improving_ONE_seq=<unsigned>` max length of allowed non-improving ONE seq [default: 0]
- `--eps=<eps>` eps determines the radius for the BFS [default: 0.5]
- `--eps2=<eps2>` eps2<=eps determines the radius for the second BFS (only used in FLOW_SEPARATOR) [default: 0.5]
- `--max_nodes=<unsigned>` bound number of nodes in local instance (currently only used in NAIVE_BFS, TIGHT_BFS) [default: 0]
- `--kamis_time_limit=<double>` time limit for static solver in seconds [default: 1000]
- `--kamis_ils_iterations=<unsigned>` maximum iterations used in WEIGHTED_ILS [default: 100]
- `--algorithm=<string>` NAIVE_BFS, TIGHT_BFS, FLOW_SEPARATOR, TRIVIAL, FLOW_SEPARATOR_UNWEIGHTED, FLOW_SEPARATOR_ALLWEIGHTED, FLOW_SEPARATOR_ALLWEIGHTED2, FLOW_SEPARATOR_ALLWEIGHTED2DEG, FLOW_SEPARATOR_NEIGHBOURHOOD [default: NAIVE_BFS]
- `--local_algorithm=<string>` local instance solver: BRANCH_AND_REDUCE, WEIGHTED_ILS [default: BRANCH_AND_REDUCE]
- `--metamis` METAMIS config (own Implementation of METAMIS by Dong et al.)
- `--mm_time_limit=<double>` time limit for METAMIS [default: 100]
- `--mm_ls_num_iterations=<unsigned>` number of iterations in local-search [default: 2]
- `--mm_ls_before_relinking=<on:off>` apply local-search on random greedy solution before path relinking [default: off]
- `--mm_ls_perturbation_amount=<unsigned>` number of (x,1)-swaps if stuck in local optimum [default: 2]

- `--mm_aap_searches=<unsigned>` number of AAP searches [default: 100]
- `--mm_aap_neg_threshold=<signed>` max negative gain threshold in one AAP search [default: -300]
- `--mm_aap_max_length=<unsigned>` max length of one AAP [default: 20]
- `--mm_max_elite_sol=<unsigned>` max maintained elite solutions [default: 1]
- `--mm_use_eone=<on:off>` use EONE after MetaMIS' local-search (EONE within METAMIS) [default: off]
- `--mm_eone_eps=<double>` eps of in EONE [default: 0.5]
- `--mm_eone_max_nodes=<unsigned>` max nodes in local instance in EONE [default: 0]
- `--mm_eone_kamis_time_limit=<limit>` time limit for BRANCH_AND_REDUCE [default: 1000]

Examples

DynWMIS has two commands: `solve` for using exactly one local-search solver and `batch-solve` for executing an arbitrary sequence of local-search solvers.
This is an example how to execute a batch of solvers. It determines a greedy solution which processes the vertices in the descending order of their weights, applies EONE for 3600 s and finally applies METAMIS for 3600 s.

```
batch-solve
graph
../examples/germany.osm.weighted.graph
germany.osm.out
--greedy_init
--rating=WEIGHT_RATING
--solver=EONE_SOLVER
--eone
--time_limit=3600
--algorithm=TIGHT_BFS
--local_algorithm=BRANCH_AND_REDUCE
--eps=0.25
--eps2=0.25
--kamis_time_limit=100
--kamis_ils_iterations=100
--max_nodes=0
--max_none_improving_ONE_seq=0
--max_ONE=0
```

```
19 --solver=METAMIS_SOLVER
20 --metamis
21 --mm_time_limit=3600
22 --mm_ls_num_iterations=2
23 --mm_ls_before_relinking=off
24 --mm_ls_perturbation_amount=2
25 --mm_aap_searches=100
26 --mm_aap_neg_threshold=-300
27 --mm_aap_max_length=20
28 --mm_max_elite_sol=1
29 --mm_use_eone=off
```

File Format

DynWMIS reads the graphs from a file. We stick to the file format used for the solvers in repository *KaMIS* [1]. The first line lists the number of vertices and edges. The remaining lines describe vertices with their assigned weights and neighbors in the graph. The i-th line (counting from one) lists the weight of the vertex with node id $i-1$, followed by its adjacency list, i. e.,. the node ids.
An initial solution can be read in from a file. This file must list the solution node ids ascending. Each line contains one node id.

Zusammenfassung

In dieser Arbeit betrachten wir das NP-vollständige Maximum Weight Indepndent Set Problem und stellen eine neue Technik, namens *Optimal Neighborhood Exploration* (ONE) (zu deutsch: Optimal Nachbarschaftserkundung), zur lokalen Suche vor. Das Maximum Weight Independent Set Problem findet in vielen wichtigen Bereichen Anwendung wie dem Map-Labeling Problem (zu deutsch: Kartenbeschriftungsproblem) oder Vehicle Routing Problem (zu deutsch: Routenplanungsproblem). In diesen Anwendungen sind die Graphen, die diese Probleme modellieren, oft extrem groß, mit Millionen von Knoten und Kanten. Dies macht es sehr schwer (fast-)optimale Lösungen in einer kurzen Zeit zu finden. Auf der einen Seite existieren gute exakte Lösungsverfahren, die optimale Lösungen finden können. Auf der anderen Seite verbessern heuristische Algorithmen durch das Benutzen von lokalen Suchtechniken gegebene Weight Indepndent Sets. Sie finden Lösungen mit größerem Gewicht, indem sie oft Knoten lokal optimal entscheiden. Wir verallgemeinern diese Idee und kombinieren beide Ansätze. ONE erkundet the Nachbarschaft des Startknotens bis zu einer bestimmten Distanz und baut einen lokalen induzierten Teilgraph, der diese Knoten enthält. Dabei verbessert jede Verbesserung der Lösung in dem lokalen induzierten Teilgraphen die Lösung des gesamten Graphens. Um die Lösungsqualität zu verbessern, lösen wir diese induzierten Teilgraphen (fast-)optimal mit dem auf dem Stand der Technik entsprechenden branch-and-reduce (zu deutsch: verzweige-und-reduziere) Löser KaMIS (KaMIS BaR).
Darüber hinaus präsentieren wir eine neue iterierte lokale Suche namens Exhaustive Optimal Neighborhood Exploration (EONE), die unsere lokale Suche ONE verwendet. In vielen und umfangreichen Experimente untersuchen wir die Effektivität dieses Ansatzes und vergleichen EONE mit anderen Verfahren, die dem Stand der Technik entsprechen. Diese Experimente unterstreichen, dass unser Verfahren oft optimale Lösungen oder zumindest fast-optimale Lösungen findet. Des Weiteren, übertreffen wir den Stand der Technik entsprechenden branch-and-reduce Löser KaMIS für schwere Instanzen. Zu guter Letzt diskutieren wir ONE und EONE und geben einen Ausblick auf Arbeit, die für die Zukunft bestimmt ist.

Bibliography

[1] KaMIS. URL https://github.com/KarlsruheMIS/KaMIS.

[2] Open Street Map. URL https://www.openstreetmap.org/.

[3] Chris Walshaw's Graph Partitioning Archive. URL https://chriswalshaw.co.uk/partition/.

[4] Faisal N. Abu-Khzam, Sebastian Lamm, Matthias Mnich, Alexander Noe, Christian Schulz, and Darren Strash. Recent advances in practical data reduction. *CoRR*, abs/2012.12594, 2020. URL https://arxiv.org/abs/2012.12594.

[5] Takuya Akiba and Yoichi Iwata. Branch-and-reduce exponential/fpt algorithms in practice: A case study of vertex cover. *Theor. Comput. Sci.*, 609:211–225, 2016. doi: 10.1016/j.tcs.2015.09.023. URL https://doi.org/10.1016/j.tcs.2015.09.023.

[6] Diogo Vieira Andrade, Mauricio G. C. Resende, and Renato Fonseca F. Werneck. Fast local search for the maximum independent set problem. *J. Heuristics*, 18 (4):525–547, 2012. doi: 10.1007/s10732-012-9196-4. URL https://doi.org/10.1007/s10732-012-9196-4.

[7] Luitpold Babel. A fast algorithm for the maximum weight clique problem. *Computing*, 52(1):31–38, 1994. doi: 10.1007/BF02243394. URL https://doi.org/10.1007/BF02243394.

[8] David A. Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner, editors. *Graph Partitioning and Graph Clustering, 10th DIMACS Implementation Challenge Workshop, Georgia Institute of Technology, Atlanta, GA, USA, February 13-14, 2012. Proceedings*, volume 588 of *Contemporary Mathematics*, 2013. American Mathematical Society. ISBN 978-0-8218-9038-7. doi: 10.1090/conm/588. URL https://doi.org/10.1090/conm/588.

[9] 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. 2014. doi: 10.1007/978-1-4614-6170-8_23. URL https://doi.org/10.1007/978-1-4614-6170-8_23.

[10] Egon Balas and Chang Sung Yu. Finding a maximum clique in an arbitrary graph. *SIAM J. Comput.*, 15(4):1054–1068, 1986. doi: 10.1137/0215075. URL https://doi.org/10.1137/0215075.

[11] Lukas Barth, Benjamin Niedermann, Martin Nöllenburg, and Darren Strash. Temporal map labeling: a new unified framework with experiments. In Siva Ravada, Mohammed Eunus Ali, Shawn D. Newsam, Matthias Renz, and Goce Trajcevski, editors, *Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, GIS 2016, Burlingame, California, USA, October 31 - November 3, 2016*, pages 23:1–23:10. ACM, 2016. doi: 10.1145/2996913.2996957. URL https://doi.org/10.1145/2996913.2996957.

[12] Sergiy Butenko and Svyatoslav Trukhanov. Using critical sets to solve the maximum independent set problem. *Operations Research Letters*, 35(4):519–524, 2007.

[13] Lijun Chang, Wei Li, and Wenjie Zhang. Computing A near-maximum independent set in linear time by reducing-peeling. In Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu, editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 1181–1196. ACM, 2017. doi: 10.1145/3035918.3035939. URL https://doi.org/10.1145/3035918.3035939.

[14] Jakob Dahlum, Sebastian Lamm, Peter Sanders, Christian Schulz, Darren Strash, and Renato F. Werneck. Accelerating local search for the maximum independent set problem. In Andrew V. Goldberg and Alexander S. Kulikov, editors, *Experimental Algorithms - 15th International Symposium, SEA 2016, St. Petersburg, Russia, June 5-8, 2016, Proceedings*, volume 9685 of *Lecture Notes in Computer Science*, pages 118–133. Springer, 2016. doi: 10.1007/978-3-319-38851-9_9. URL https://doi.org/10.1007/978-3-319-38851-9_9.

[15] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, 2011. doi: 10.1145/2049662.2049663. URL https://doi.org/10.1145/2049662.2049663.

[16] Yuanyuan Dong, Andrew V. Goldberg, Alexander Noe, Nikos Parotsidis, Mauricio G. C. Resende, and Quico Spaen. New instances for maximum weight independent set from a vehicle routing application. *CoRR*, abs/2105.12623, 2021. URL `https://arxiv.org/abs/2105.12623`.

[17] Yuanyuan Dong, Andrew V Goldberg, Alexander Noe, Nikos Parotsidis, Mauricio GC Resende, and Quico Spaen. New instances for maximum weight independent set from a vehicle routing application. In *Operations Research Forum*, volume 2, pages 1–6. Springer, 2021.

[18] Yuanyuan Dong, Andrew V. Goldberg, Alexander Noe, Nikos Parotsidis, Mauricio G. C. Resende, and Quico Spaen. A metaheuristic algorithm for large maximum weight independent set problems. *CoRR*, abs/2203.15805, 2022. doi: 10.48550/arXiv.2203.15805. URL `https://doi.org/10.48550/arXiv.2203.15805`.

[19] Aleksander Figiel, Vincent Froese, André Nichterlein, and Rolf Niedermeier. There and back again: On applying data reduction rules by undoing others. *CoRR*, abs/2206.14698, 2022. doi: 10.48550/arXiv.2206.14698. URL `https://doi.org/10.48550/arXiv.2206.14698`.

[20] M. R. Garey, David S. Johnson, and Larry J. Stockmeyer. Some simplified np-complete graph problems. *Theor. Comput. Sci.*, 1(3):237–267, 1976. doi: 10.1016/0304-3975(76)90059-1. URL `https://doi.org/10.1016/0304-3975(76)90059-1`.

[21] Alexander Gellner, Sebastian Lamm, Christian Schulz, Darren Strash, and Bogdán Zaválnij. Boosting data reduction for the maximum weight independent set problem using increasing transformations. In Martin Farach-Colton and Sabine Storandt, editors, *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2021, Virtual Conference, January 10-11, 2021*, pages 128–142. SIAM, 2021. doi: 10.1137/1.9781611976472.10. URL `https://doi.org/10.1137/1.9781611976472.10`.

[22] Andreas Gemsa, Martin Nöllenburg, and Ignaz Rutter. Evaluation of labeling strategies for rotating maps. In Joachim Gudmundsson and Jyrki Katajainen, editors, *Experimental Algorithms - 13th International Symposium, SEA 2014, Copenhagen, Denmark, June 29 - July 1, 2014. Proceedings*, volume 8504 of *Lecture Notes in Computer Science*, pages 235–246. Springer, 2014. doi: 10.1007/978-3-319-07959-2_20. URL `https://doi.org/10.1007/978-3-319-07959-2_20`.

[23] David E. Goldberg. *Genetic Algorithms in Search Optimization and Machine Learning*. Addison-Wesley, 1989. ISBN 0-201-15767-5.

[24] Jiewei Gu, Weiguo Zheng, Yuzheng Cai, and Peng Peng. Towards computing a near-maximum weighted independent set on massive graphs. In Feida Zhu, Beng Chin Ooi, and Chunyan Miao, editors, *KDD '21: The 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, Singapore, August 14-18, 2021*, pages 467–477. ACM, 2021. doi: 10.1145/3447548.3467232. URL https://doi.org/10.1145/3447548.3467232.

[25] Scott P. Kolodziej, Mohsen Aznaveh, Matthew Bullock, Jarrett David, Timothy A. Davis, Matthew Henderson, Yifan Hu, and Read Sandström. The suitesparse matrix collection website interface. *J. Open Source Softw.*, 4(35):1244, 2019. doi: 10.21105/joss.01244. URL https://doi.org/10.21105/joss.01244.

[26] Sebastian Lamm, Peter Sanders, and Christian Schulz. Graph partitioning for independent sets. In Evripidis Bampis, editor, *Experimental Algorithms - 14th International Symposium, SEA 2015, Paris, France, June 29 - July 1, 2015, Proceedings*, volume 9125 of *Lecture Notes in Computer Science*, pages 68–81. Springer, 2015. doi: 10.1007/978-3-319-20086-6_6. URL https://doi.org/10.1007/978-3-319-20086-6_6.

[27] Sebastian Lamm, Peter Sanders, Christian Schulz, Darren Strash, and Renato F. Werneck. Finding near-optimal independent sets at scale. *J. Heuristics*, 23(4): 207–229, 2017. doi: 10.1007/s10732-017-9337-x. URL https://doi.org/10.1007/s10732-017-9337-x.

[28] Sebastian Lamm, Christian Schulz, Darren Strash, Robert Williger, and Huashuo Zhang. Exactly solving the maximum weight independent set problem on large real-world graphs. In Stephen G. Kobourov and Henning Meyerhenke, editors, *Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments, ALENEX 2019, San Diego, CA, USA, January 7-8, 2019*, pages 144–158. SIAM, 2019. doi: 10.1137/1.9781611975499.12. URL https://doi.org/10.1137/1.9781611975499.12.

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

[30] Bruno C. S. Nogueira, Rian G. S. Pinheiro, and Anand Subramanian. A hybrid iterated local search heuristic for the maximum weight independent set problem. *Optim. Lett.*, 12(3):567–583, 2018. doi: 10.1007/s11590-017-1128-7. URL https://doi.org/10.1007/s11590-017-1128-7.

[31] Manfred W Padberg. On the facial structure of set packing polyhedra. *Mathematical programming*, 5(1):199–215, 1973.

[32] Mauricio GC Resende and Celso C Ribeiro. *Optimization by GRASP*. Springer, 2016.

[33] Jeffrey S Warren and Illya V Hicks. Combinatorial branch-and-bound for the maximum weight independent set problem. *Relatório Técnico, Texas A&M University, Citeseer*, 9:17, 2006.