A Practical Evaluation of Kernelisation Algorithms for the MULTICUT IN TREES Problem

Computing Science Master Thesis

Steven Heinen
ICA-5911486

Utrecht University

Department of Information and Computing Sciences
Utrecht University
The Netherlands
September 27th, 2021

Supervisors
Dr. J.M.M. van Rooij
Dr. E.J. van Leeuwen
S. Pandey, MSc

Abstract

In this thesis, we consider the parameterised version of MULTICUT IN TREES. The problem consists of a tree $T = (V, E)$, a set P of pairs of distinct nodes in V called demand pairs, and a non-negative integer k. We want to find a set $E' \subseteq E$, with $|E'| \leq k$, such that there is no path between the two nodes of every demand pair when we remove every edge $e \in E'$ from T.

The problem is closely related to VERTEX COVER. Finding a new or improved algorithm for either problem could yield the same improvement for the other.

In this research, we evaluated two kernelisation algorithms for MULTICUT IN TREES. The first is by Guo and Niedermeier [23] and computes a kernel of size $O(k^{3k})$. The second algorithm is by Bousquet *et al.* [4] and results in a kernel of size $O(k^6)$.

We tested these algorithms on different instances, using various methods to generate the input tree and the set of demand pairs. We introduced some of these methods in this research.

This research partially fills a gap in the area of kernelisation algorithms. It is one of the first studies to practically evaluates these algorithms.

We aimed to answer five questions. We wondered what instances are easy or difficult to compute a small kernel on for the algorithms, how the kernel size and the running time scale with the size of the instance, and how usable the newly introduced ways to generate demand pairs are. Then, we wanted to know which reduction rules in the algorithms take the most time. Finally, we answered how the two algorithms compare in terms of both kernel size and running time. To measure the running time, we counted the number of operations that the algorithms perform.

Instances with a caterpillar tree generally resulted in smaller kernels than instances with a tree based on a Prüfer sequence. The newly introduced Degree3 trees were present in the most challenging instances. The algorithms could easily compute a small kernel on instances with long paths between every demand pair's two nodes or a tiny solution. When the instance contained demand pairs such that the path between the nodes of most pairs is short, the algorithms computed a large kernel on these instances.

In most cases, the size of the kernel depended on the size of the instance, and the running time did as well.

The first newly introduced demand pair generation method generates demand pairs such that the length of the path between the two nodes in the pair can be influenced. This method resulted in excellent instances and is quite usable. The other newly introduced method generates demand pairs such that the solution of the instance is known beforehand. This method, however, is not very usable in its current state.

Each algorithm had one reduction rule that took the majority of the time. This rule was not the same between the two algorithms. The rules were the Dominated Edge, and Disjoint Paths rule, respectively.

We also saw that the sizes of the kernels obtained by the two algorithms only differed in a few cases. Both algorithms had instances on which they computed a smaller kernel than the other algorithm. The algorithm by Guo and Niedermeier was in our implementation faster than the algorithm by Bousquet *et al.*

Acknowledgements

I would like to thank my supervisors for their incredible enthusiasm, support, valuable discussions, willingness to answer questions, and above all for the random jokes and chit-chat during our meetings that made me feel very much at ease. I would also like to thank my parents for other helpful discussions, comments and suggestions, and for lending me a computer to use for more experiments. Finally, I want to thank my rabbits Mozzie and Hibana, for being alive equivalents of rubber duckies, for being soft and for keeping me company during the pandemic lockdown.

Contents

1 Introduction

In this thesis, we study Multicut in Trees. This problem exists in the area of graph theory and knows its applications in networking [11]. For instance, one can use the problem to evaluate robustness in telecommunication and transportation networks. The problem is exciting because of its close connection to the widely known Vertex Cover problem. We explain this connection in more detail later in this thesis.

Minimum s-t Cut. Multicut in Trees is closely related to the better known Minimum s-t Cut problem. In this problem, we are given a graph $G = (V, E)$, with $n = |V|$ and $m = |E|$, and two distinct nodes $s, t \in V$. Our task is to remove edges from G, such that no path between s and t remains. It relates to the Min-Cut Max-Flow theorem, introduced by Ford and Fulkerson in 1962 [19].

Minimum s-t Cut can be solved using the Edmonds-Karp algorithm [16] in $O(nm^2)$ time. This algorithm is based on the Ford-Fulkerson method [18].

Faster algorithms exist, like the algorithm that combines the algorithms by King *et al.* [33] and Orlin [47]. This combination algorithm has a time complexity of $O(nm)$.

Multicut. Multicut is a generalisation of Minimum s-t Cut. It has multiple pairs of nodes that need to be separated. The input of this problem consists of a graph $G = (V, E)$, with $n = |V|$ and $m = |E|$, and a set P of p pairs of distinct nodes in V. A single node can appear in multiple pairs in P, but it cannot appear twice in a single pair. To solve the problem, we need to find a set $E' \subseteq E$ of minimum size, such that there exists no path in G between the two nodes of every pair in P after we remove every edge $e \in E'$ from G.

Hu introduced Multicut in 1969 [27]. Obviously, for $p = 1$, Multicut is equivalent to Minimum s-t Cut. We can then solve it in polynomial time. For $p = 2$, Multicut can still be solved in polynomial time, for instance using an algorithm by Itai [29] in $O(n^3)$ time. For any fixed $p \geq 3$, Multicut is $\mathcal{NP}$-complete, as shown by Dahlhaus *et al.* [13].

Multicut in Trees. This research considers a particular version of Multicut, namely Multicut in Trees. Here, our input graph G must be a tree. We give a more formal definition of the exact version we study in Section 2. Garg *et al.* [21] showed that Multicut in Trees is $\mathcal{NP}$-complete as well, using a reduction from Vertex Cover.

In particular, this thesis considers the parameterised version of Multicut in Trees. In this version, the input contains an extra parameter $k \in \mathbb{N}_0$. Instead of finding a set $E' \subseteq E$ of minimum size of edges that we need to remove, we want to find a set $E'' \subseteq E$, such that $|E''| \leq k$. We also give a proper definition of this problem in Section 2.

Let us now show a small example of the parameterised version of Multicut in Trees. After that, we explain the goal of this research and the outline of this thesis.

1.1 Example

To help understand the parameterised version of Multicut in Trees, we give a small example here. The example is visualised in Figure 1. Consider an instance with a tree consisting of 10 nodes, labelled $1, \ldots, 10$, parameter $k = 2$ and $P = \{(1, 6), (4, 5), (8, 10)\}$. See also Subfigure 1a, in which the two nodes of each pair in P have the same colour.

To separate a pair in P, we have to include an edge on the path between the two nodes of that pair in the solution. We need to find a solution that includes such an edge for every pair in P to separate all pairs. The paths between the two nodes of every pair are visible in Subfigure 1b, with the same colours as the nodes of that pair.

Subfigure 1c shows a possible solution, visualised by the red edges, that includes an edge on each of the coloured paths. However, this solution includes three edges, but k, the maximum size the solution is allowed to have, equals two. It is thus not a valid solution.

Finally, in Subfigure 1d, the red edges form another possible solution. This solution consists of only two edges. Indeed, the solution includes an edge on each of the coloured paths. This solution is valid for this instance.

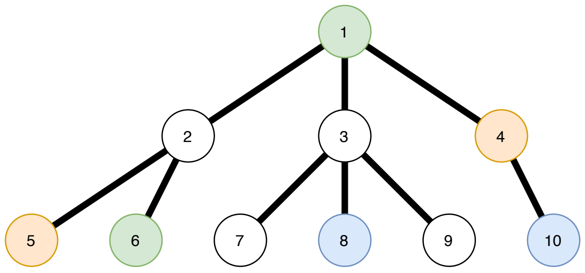
1.2 Research Goals

In this research, we evaluate two kernelisation algorithms for Multicut in Trees. We explain kernelisation in Section 2.

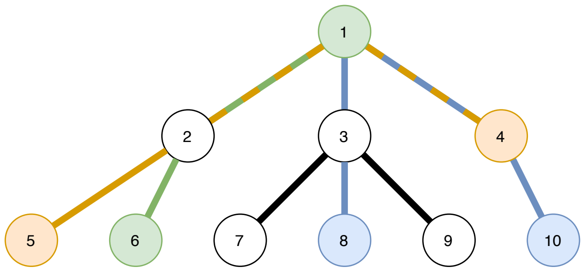
Many different algorithms for different variations of Multicut in Trees exist, but we were not able to find any practical evaluations of them. Of course, proofs for the performance of different algorithms do exist. These proofs give some insight into how well these algorithms work, but factors like the tightness of the used analyses could significantly influence the theoretical performance. A practical evaluation of such algorithms can help better understand how the algorithms perform compared to each other. This understanding can be helpful for those who wish to use these algorithms in their own applications.

The two algorithms we study are an algorithm by Guo and Niedermeier from 2005 [23] and an algorithm from 2009 by Bousquet *et al.* [4]. We explain these algorithms in Section 4. For this research, we implement these algorithms and see how they perform on different instances. We try to answer five research questions for these two algorithms.

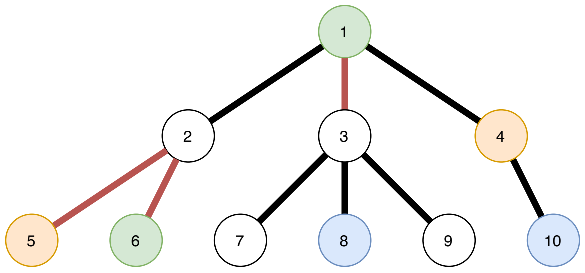
To understand the research questions, we introduce a few definitions here. We give the formal definitions in Section 2. First, a *kernel* is a generally smaller version of an instance for



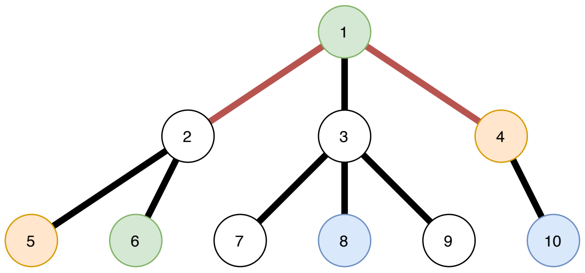
(a) The input tree with 10 nodes and three pairs to be separated. The pairs in P are each coloured in a different colour.



(b) The paths between the pairs in P.



(c) Edges in the solution are coloured red. Removing one edge from the path of each pair separates all pairs. However, there are three edges in this solution, and parameter k equals two, so this solution is not valid.



(d) Another possible solution. Edges in the solution are again coloured red. The solution contains only two edges: one for each path between the two nodes of every pair in P. This solution is valid.

Figure 1: An example of an instance for the parameterised version of Multicut in Trees. Given a tree with 10 nodes, parameter $k = 2$, and a set $P = \{(1,6),\ (4,5),\ (8,10)\}$.

a problem, such that the problem can be solved on the kernel if and only if the problem can be solved on the original instance. The kernelisation algorithms compute such a kernel from an instance using *reduction rules*, which determine how to reduce the size of the instance. Finally, the pairs of nodes in P that need to be separated are called *demand pairs*, and the path between the two nodes of a demand pair is called a *demand path*.

Easy and difficult instances. Between different graphs that are all trees, there can still be many differences. The depth of a tree and the maximum degree of a node in a tree could potentially influence the performance of the algorithms we study.

The pairs that need to be separated can also influence the performance. For example, there could be differences when the two nodes of each pair are very close together or very far apart. The amount of overlap between the demand paths can also influence the performance.

The first research question is about identifying which instances are easy for the tested algorithms and which are complex.

Research Question 1. *What types of instances are easy or difficult for the kernelisation algorithms?*

To answer this question, we consider the following subquestions.

Research Question 1a. *For which types of instances does the kernelisation process result in a small kernel in little time?*

This question looks for easy instances. The following looks for difficult instances.

Research Question 1b. *For which types of instances does the kernelisation process result in a large kernel in much time?*

Finally, the last subquestion is about instances that we generated using methods used in $\mathcal{NP}$-completeness proofs for Multicut in Trees. We explain these methods and from which proofs they originate in Section 6.

Research Question 1c. *How difficult are instances based on $\mathcal{NP}$-completeness proofs?*

Scaling of the kernel size. The second question is about the scaling of a kernel with the size of an instance. With more demand pairs, the expected amount of overlap between their demand paths is higher than with fewer demand pairs. This overlap could mean that the size of the kernel is also larger. Maybe, however, this overlap allows the algorithms to compute kernels more efficiently.

Similarly, with more nodes in the tree, the overlap between the demand paths can decrease. Does this then mean that the size of the kernel decreases?

We also look at the computation time to compute these kernels. With a larger instance, it can be the case that an algorithm needs to perform more modifications to compute a small kernel.

The question we want to answer about this topic is as follows.

Research Question 2. *Do kernels scale with the size of the instance?*

We answer the following four subquestions.

Research Question 2a. *Does increasing the number of nodes in an instance mean that the kernel becomes larger?*

Research Question 2b. *Does increasing the number of demand pairs in an instance mean that the kernel becomes larger?*

Research Question 2c. *Does increasing the number of nodes in an instance mean that the computation of the kernel takes longer?*

Research Question 2d. *Does increasing the number of demand pairs in an instance mean that the computation of the kernel takes longer?*

Different ways to generate demand pairs. Generating demand pairs is possible by uniformly at random picking two distinct nodes in the tree for each demand pair. We use this method in our experiments, but we also use two other methods. One generates demand pairs, such that the length of their respective demand paths is in a particular range, and the other generates demand pairs in such a way that the solution has a specific size. We explain these methods in more detail in Section 6.

In particular, we look at the difficulty of the instances with different parameters for these extra demand pair generation methods. The main research question we want to answer is as follows.

Research Question 3. *How usable are the demand pair generation methods we introduce?*

For demand paths with a certain length, we compare instances with mostly short demand paths to instances with mostly long demand paths. We explain which lengths we used in Section 7. For this method, we answer two subquestions.

Research Question 3a. *Do instances with mostly short demand paths result in a smaller or larger kernel than instances with mostly long demand paths?*

Research Question 3b. *Does computing a kernel on an instance with mostly short demand paths take shorter or longer than computing a kernel on an instance with mostly long demand paths?*

For the other generation method, we vary the size of the solution. We also explain how we varied this size in Section 7. For this method, we answer two subquestions as well.

Research Question 3c. *Do instances with demand pairs generated using a small solution result in a smaller or larger kernel than instances with demand pairs generated using a large solution?*

Research Question 3d. *Does computing a kernel on an instance with demand pairs generated using a small solution take shorter or longer than computing a kernel on an instance with demand pairs generated using a large solution?*

Dominating reduction rules. We can divide the time spent by an algorithm into roughly two parts. One is performing the modifications to reduce the size of the instance, and the other one is determining which modifications the algorithm can perform on the instance. The reduction rules do this second part.

If one of these rules is a lot slower than the other rules in the same algorithm, that rule influences the time performance of that algorithm the most. The fourth research question is about identifying these dominant reduction rules.

Research Question 4. *Which reduction rules take the most time during kernelisation?*

The dominating rule can differ between various difficulties of instances. In two subquestions, we look at both the easy and difficult instances as identified in Research Question 1.

Research Question 4a. *Which reduction rules take the most time on easy instances?*

Research Question 4b. *Which reduction rules take the most time on difficult instances?*

Comparing both algorithms. Our final research question is about the difference between the two algorithms. We talk more about the algorithms in Sections 3 and 4, but in the literature, the newer algorithm by Bousquet *et al.* [4] results in a smaller kernel than the algorithm by Guo and Niedermeier [23]. Both algorithms are proven to run in polynomial time, but tight analyses of their running times are not present in the papers.

With the following research question, we want to see how these two algorithms compare in practice.

Research Question 5. *Is the kernelisation algorithm by Bousquet et al. [4] better than the kernelisation algorithm by Guo and Niedermeier [23]?*

In the two subquestions, we look separately at the kernel size and the running time of both algorithms.

Research Question 5a. *Does the algorithm by Bousquet et al. result in smaller kernels than the algorithm by Guo and Niedermeier?*

Research Question 5b. *Does the algorithm by Bousquet et al. take less time to compute kernels than the algorithm by Guo and Niedermeier?*

1.3 Outline

Section 2 explains some preliminary knowledge and definitions needed to understand this research. Section 3 provides context about the problem and the literature that exists about Multicut in Trees. Then, Section 4 explains the reduction rules used in the three kernelisation algorithms for Multicut in Trees, and Section 5 how we implemented the two algorithms for this research. After that, we discuss how we generated instances in Section 6. Section 7 explains our experimental setup, and in Section 8 we show and discuss our results and answer our research questions. Finally, Section 9 summarises research and provides ideas for future work.

2 Preliminaries

This section explains most of the definitions necessary to understand this research. We assume the reader to be proficient with basic graph theory. For more information about this topic, see the book by West *et al.* [53]. Furthermore, we assume knowledge about computational complexity theory. The reader can find more information about this topic in the book by Arora and Barak [3].

2.1 Problem Definition

In Section 1, we used Multicut and Multicut in Trees to define the problem we study in this research. However, we can find three different variations of this problem throughout the literature. These are Edge Multicut (EMC), Restricted Vertex Multicut (RVMC), and Unrestricted Vertex Multicut (UVMC). The difference between these three problems is what we remove from the graph to achieve the goal. In EMC, we remove edges from the graph. In both RVMC and UVMC, we remove nodes. RVMC restricts which nodes we can remove. In this version, we cannot remove nodes that belong to one or more pairs that need to be separated. In UVMC, we can remove any node.

All these problems exist in general graphs. However, in this research, we are more interested in the problems in trees. Călinescu *et al.* [8] showed UVMC in Trees is solvable in polynomial time. In the same article, they showed EMC in Trees and RVMC in Trees are $\mathcal{NP}$-hard for bounded-degree trees.

Throughout this research, we only consider EDGE MULTICUT IN TREES. We refer to this problem by simply MULTICUT IN TREES, unless it is clear from the context that we mean another version. Let us give a formal definition of the problem.

Definition 1 (MULTICUT IN TREES).
Input: *A tree $T = (V, E)$, $n = |V|$, and a set P of pairs of nodes in V, called* demand pairs*,*
$P = \{(s_i, t_i) \mid s_i, t_i \in V, s_i \neq t_i, 1 \leq i \leq p)\}$.
Task: *Find a set $E' \subseteq E$ of minimum size, such that the removal of each $e \in E$ from T separates each demand pair in P.*

We say a demand pair between nodes u and v is *separated* when there is no path between u and v in T. Note that this is never possible in a single tree. We can only separate demand pairs when we remove edges from T, so we separate them in the forest $\mathcal{F}$ that is left after we have removed the edges. The unique path between u and v in T is called its *demand path*. We say a demand path is *destroyed* when its demand pair is separated.

2.2 Fixed-Parameter Tractability

In this research, we study the fixed-parameter tractable (FPT) version of MULTICUT IN TREES. We give a basic definition of FPT in this subsection. The reader can find more information about this topic in, for instance, the book by Downey and Fellows [14].

We say a problem is *fixed-parameter tractable* when, given the input I for the problem and a parameter $k \in \mathbb{N}_0$, the problem can be solved in $O(f(k) \cdot g(|I|))$ time, for any computable function f, and a polynomial function g.

Kernelisation. A way to show that a problem is FPT is by giving a kernel for the problem through *kernelisation*. Let $\mathcal{P}$ be a problem, with input I and parameter k. We want to map (I, k) to (I', k'), such that:

- (I, k) has a solution for $\mathcal{P}$ if and only if (I', k') has a solution for $\mathcal{P}$,
- $|I'| \leq h(k)$, for some computable function h,
- $k' \leq l(k)$, for some computable function l, and
- the conversion from (I, k) to (I', k') takes time polynomial in $|I|$ and k.

We call I' the *reduced* input or instance. We can also call it a *kernel* for $\mathcal{P}$. The conversion from (I, k) to (I', k') is done using a set of *reduction rules*. With a kernel for $\mathcal{P}$, it is possible to solve the current instance I in the required running time of $O(f(k) \cdot g(|I|))$.

The parameterised version of MULTICUT IN TREES is defined as follows.

Definition 2 (PARAMETERISED MULTICUT IN TREES).
Input: *A tree $T = (V, E)$, $n = |V|$, a set P of pairs of nodes in V, called* demand pairs*,*
$P = \{(s_i, t_i) \mid s_i, t_i \in V, s_i \neq t_i, 1 \leq i \leq p)\}$, and a parameter $k \in \mathbb{N}_0$.
Task: *Find a set $E' \subseteq E$, $|E'| \leq k$, such that the removal of each $e \in E$ from T separates each demand pair in P.*

From now on, we refer to PARAMETERISED MULTICUT IN TREES simply by MULTICUT IN TREES, unless it is clear from the context we mean another version.

2.3 Problem-related Definitions

This final subsection contains more definitions that are necessary to understand the literature and algorithms studied. The definitions explained here can also be found in Appendix A.

Modifications to the instance during kernelisation. The algorithms studied in this research preprocess an instance for Multicut in Trees to a kernel for the problem. They do this using a set of reduction rules. Each of these rules can trigger one of four possible modifications to the instance.

First, we can *contract* an edge. Let e be an edge between nodes u and v. When we contract e, we create a new node w and connect it to all neighbours of u and all neighbours of v, except for u and v themselves. Then, we replace u and v in T by w. All demand pairs in the instance that started at either u or v now start at w.

The second modification is removing a demand pair. This modification should speak for itself.

The third modification is changing the endpoint of a demand pair. For instance, we can change a demand pair between nodes u and v to be between v and w. In the studied algorithms, this only happens when w is on the path between u and v. The new demand path is thus a strict subset of the old path.

Finally, we can *cut* an edge in T. When we cut an edge e, we add e to the solution and decrease k by one. Then, we remove all demand paths that go over e, and finally, we contract e.

Types of nodes. In the input tree T, nodes are separated into *leaves* and *internal nodes*. A *leaf* is a node with precisely one neighbour, and an *internal node* is a node with at least two neighbours.

The internal nodes in T are further divided into three types: I_1*-nodes*, I_2*-nodes*, and I_3*-nodes*. From these, I_1-nodes have at most one neighbour that is an internal node. I_2-nodes have precisely two neighbours that are internal nodes. Finally, I_3-nodes have at least three neighbours that are internal nodes.

In a similar way, we can divide the leaves of T into three types: L_1*-leaves*, L_2*-leaves*, and L_3*-leaves*. L_1-leaves are leaves connected to I_1-nodes, L_2-leaves are connected to I_2-nodes, and L_3-leaves to I_3-nodes.

Furthermore, an *inner* node is an I_2-node without leaves.

Leaves can either be *good* or *bad*. Let u be a leaf, and let v be the neighbour of u. We say u is a *good leaf*, if there exists a demand pair between u and a node w, such that w is a leaf connected to v. If u is not a good leaf, it is a *bad leaf*.

Caterpillars. A *caterpillar* is a tree consisting of a path of I_2-nodes with an I_1-node at each end, and the leaves connected to all these nodes. There are no I_3-nodes in a caterpillar. The path of I_2-nodes is called the *backbone* of a caterpillar.

In a tree T, we can define a *caterpillar component* as a maximal induced subgraph of T consisting only of I_2-nodes and L_2-leaves. A *caterpillar* in T is a maximal induced subgraph of T consisting only of I_1-nodes, I_2-nodes, L_1-leaves and L_2-leaves.

The *extremities* of a caterpillar C in T are the two internal nodes in C that would become leaves in C if we would remove all leaves of C. In other words, they are the two nodes at the end of the path of internal nodes in C. Should C only consist of a single internal node v, v is both extremities of C.

Demand paths in the same direction. Let us define demand paths going in the *same direction*. If zero or one demand paths start at a node v, we say its demand paths go in the same direction.

On the other hand, if two or more demand paths start at a node v, we need to consider two cases.

In the first case, v is an internal node. Its demand paths go in the same direction if the first edge on their paths from v to their other endpoint is the same for all these demand paths.

In the other case, v is a leaf. Its demand paths go in the same direction if the second edge on their paths from v to their other endpoint is the same for all these paths.

Minimal demand paths. The *internal path* of a demand path is the subset of the demand path consisting only of edges between two internal nodes. Consequently, the *internal length* of a demand path is the length of its internal path.

We say a demand path starting at a node v is a *minimal demand path* if its internal path is not a superset of the internal path of any other demand path starting at v.

Let u be the other endpoint of a minimal demand path starting at v. We call u the *closest P-neighbour* of v, and v the *closest P-neighbour* of u. Note that a node can have multiple closest P-neighbours.

Wingspans. Let v be a bad L_2-leaf, and let w be its neighbour. Call the two edges between two internal nodes connected to w the "left" and "right" edge without loss of generality. Determine the two minimal demand paths starting at v, such that one of these paths goes over the left edge and the other over the right edge. Call them P_1 and P_2.

We define the *wingspan* W of v to be the union of the internal nodes of P_1 and P_2.

The *subcaterpillar* of W consists of all nodes in W and all leaves connected to these nodes.

Finally, W *dominates* a demand pair between nodes y and z if both y and z belong to the subcaterpillar of W.

Covering a caterpillar. Let v be a bad L_2-leaf of a caterpillar (in T) C. If we remove the neighbour of v and all its leaves from T, we are left with two connected components. We call these two components $A(v)$ and $B(v)$. We define $a(v)$ as the extremity of C in $A(v)$. If this extremity does not exist, define $a(v)$ as the neighbour of v. Likewise, define $b(v)$ to be the extremity of C in $B(v)$. Finally, define P_1 and P_2 as the minimal demand paths going left and right from the neighbour of v, like before. We say v *covers* C if P_1 and P_2 together go over the entire backbone of C.

3 Related Literature

There exist more algorithms for Multicut in Trees than the kernelisation algorithms we focus on in this study. This section discusses some related problems to provide more context and mention some algorithms for Multicut in Trees.

3.1 Related Problems

This subsection discusses some problems that are related to Multicut in Trees.

Minimum s-t Cut. We have already seen Minimum s-t Cut in Section 1. Multicut with a single demand pair is equal to Minimum s-t Cut. As mentioned in Section 1, Minimum s-t Cut is solvable in polynomial time. We have also seen Multicut with two demand pairs is solvable in polynomial time, using a two-commodity flow algorithm, like the one by Itai [29]. Dahlhaus *et al.* showed that Multicut is $\mathcal{NP}$-hard for any fixed $p \geq 3$ [13].

Multiway Cut. Another related problem is Multiway Cut. Instead of having a set P of p demand pairs, we have a set H of h terminal nodes. The goal is to separate each pair of terminal nodes.

Multiway Cut is known to be $\mathcal{NP}$-complete for any fixed $h \geq 3$, as shown by Dahlhaus *et al.* in 1992 [12]. This is not entirely surprising, as Multiway Cut with $h = 3$ has three pairs of nodes to be separated. It can thus be compared to MULTICUT with $p = 3$, which is an $\mathcal{NP}$-complete problem.

MULTIWAY CUT has three variants, just like MULTICUT. For MULTICUT, these variants were explained in Subsection 2.1. From these, Marx [34] showed NODE MULTIWAY CUT is FPT. For this problem, the best-known approximation algorithm, by Sharma *et al.* [51], has an approximation ratio of 1.2965.

GENERALISED MULTIWAY CUT. The final related problem we mention is GENERALISED MULTIWAY CUT. In contrast to MULTIWAY CUT, we have a set $Q = \{S_1, \ldots, S_q\}$ of sets of nodes, instead of a single set of nodes H. The goal is to separate each pair of nodes within each set S_i from each other.

If all sets in Q contain precisely two nodes, GENERALISED MULTIWAY CUT is equivalent to MULTICUT. With this knowledge, it is possible to reduce GENERALISED MULTIWAY CUT to MULTICUT. We thus know GENERALISED MULTIWAY CUT is also $\mathcal{NP}$-complete and FPT.

3.2 FPT Algorithms for MULTICUT IN TREES

Kernelisation, as explained in Section 2, is only one way to show that a problem is fixed-parameter tractable. Remember the only requirement for FPT is that the problem with an instance I and parameter $k \in \mathbb{N}_0$ can be solved in $O(f(k) \cdot g(|I|))$ time, for any computable function f and a polynomial function g. We can also achieve this running time using a bounded search-tree algorithm through branching.

This subsection explains a branching algorithm to solve MULTICUT IN TREES, shows the results the three kernelisation algorithms achieve, and highlights two of the fastest algorithms to solve MULTICUT IN TREES.

A branching algorithm for MULTICUT IN TREES. The first algorithm that showed MULTICUT IN TREES is FPT is a branching algorithm from 2005 by Guo and Niedermeier [23]. This algorithm runs in $O(2^k \cdot pn)$ time, where n is the number of nodes in the tree, p is the number of demand pairs, and k is the maximum size of the solution.

The algorithm uses the *least common ancestor* of each demand pair. The least common ancestor of two nodes u and v is the node in the rooted tree with the greatest distance from the root r on both the path from u to r and from v to r. For a demand pair, its least common ancestor is the least common ancestor of the endpoints of this demand pair.

For each demand pair, the algorithm branches on at most two cases. It either includes in the solution the edge from the least common ancestor of the demand pair towards the first endpoint of the demand pair, or the edge from the least common ancestor towards the second endpoint of the demand pair. If one of the endpoints of the demand pair itself is the least common ancestor, it does not branch and includes the edge from this least common ancestor to the other endpoint of the demand pair in the solution.

Kernelisation algorithms for MULTICUT IN TREES. Three kernelisation algorithms for MULTICUT IN TREES exist. In this research, we study two of them. We explain all three algorithms in more detail in Section 4.

A first algorithm. Guo and Niedermeier presented the first kernelisation algorithm for MULTICUT IN TREES in the same paper as their branching algorithm [23]. Their algorithm uses eight reduction rules. With these reduction rules, Guo and Niedermeier proved the existence of a kernel on caterpillars of size $O(k^{2k+1})$. With the same algorithm on general trees, they could reduce an instance to one of size $O(k^{3k})$. This kernel has exponential size.

They referenced an interleaving technique introduced by Niedermeier and Rossmanith in 2000 [46], with which they could solve the problem on their kernel in $O(k^{3k} + p^3 n + n^3 p)$ time.

One of the questions Guo and Niedermeier raised was whether a polynomial kernel would exist for MULTICUT IN TREES.

The second algorithm. This question was answered in 2009 by Bousquet *et al.* [4]. Their algorithm results in a kernel of size $O(k^5)$ on caterpillars and one of size $O(k^6)$ on general trees.

To compute this kernel, they used a set of seven reduction rules. From these, three were entirely new, and three were also present in the algorithm by Guo and Niedermeier. The last rule seems to be an improved version of a combination of two other rules that Guo and Niedermeier used.

Bousquet *et al.* said they focused on proving a polynomial kernel and mentioned that their analysis of the kernel size might not be very tight. They expressed the hope that an even smaller kernel could be proven using the same set of reduction rules.

The third and final algorithm. Chen *et al.* presented the smallest known kernel for MULTICUT IN TREES in 2012 [10]. Their kernel has a size of $O(k^3)$ for general trees. In contrast to the other two articles, no separate proof for the size of the kernel on caterpillars was present in this paper.

Chen *et al.* explicitly looked at the connection that exists between MULTICUT IN TREES and VERTEX COVER. This connection is visible in their algorithm, as they used two reduction rules that use (parts of) kernelisation algorithms for VERTEX COVER.

Chen *et al.* used a slightly different problem definition for MULTICUT IN TREES. Instead of restricting the input graph to a tree, they restricted it to a forest. During their algorithm, it is also possible that reduction rules split a tree into multiple smaller trees.

The algorithm uses 12 reduction rules. Guo and Niedermeier already used four of these rules, and Chen *et al.* used one rule Bousquet *et al.* presented in their algorithm. One rule is specific for the forest variant of the problem. That leaves six new rules, of which two are related to VERTEX COVER. The other four are very detailed and specific rules with much overlap.

To conclude, there exists quite some overlap between the three kernelisation algorithms. The most general rules are present in each of them.

Algorithms to solve MULTICUT IN TREES. Besides presenting their kernel of $O(k^3)$ size, Chen *et al.* gave an algorithm to solve MULTICUT IN TREES in $O^*(\rho^k)$ time, where $\rho = \frac{\sqrt{5}+1}{2} \approx 1.618$ [10]. Strikingly, they used four more reduction rules in this algorithm, which they did not use to compute the kernel.

In the conclusion of their article, they mentioned that an $O^*(\rho'^k)$ time algorithm to solve MULTICUT IN TREES is possible, where $\rho' = \sqrt{\sqrt{2}+1} \approx 1.554$. They would have had to handle more cases in branching rules and give a more detailed case-by-case analysis to prove this running time. They deemed this analysis unworthy of being presented in their paper.

A faster algorithm to solve MULTICUT IN TREES. Three years later, Kanj *et al.* gave the analysis for this improved algorithm [31]. By further using the connection between MULTICUT IN TREES and VERTEX COVER, they proved their running time of $O^*(\rho^k)$, where $\rho = \sqrt{\sqrt{2}+1} \approx 1.554$.

This result also means that GENERALISED MULTIWAY CUT IN TREES, a problem they also studied in the same article, is solvable with the same time complexity.

They wondered if an even better algorithm for MULTICUT IN TREES is possible, by exploiting the connection to VERTEX COVER further, as an $O^*(\sigma^k)$ time algorithm, where $\sigma \approx 1.2738$, exists for VERTEX COVER [9]. As far as we know, this algorithm by Kanj *et al.* is the fastest algorithm to solve MULTICUT IN TREES.

3.3 Approximation Algorithms for Multicut in Trees

Garg *et al.* [21] gave the best-known approximation algorithm for Multicut in Trees. They presented a primal-dual approximation algorithm with an approximation ratio of two.

Garg *et al.* showed in the same paper that Multicut in Trees is at least as hard to approximate as Vertex Cover. Khot and Regev [32] showed that Vertex Cover is hard to approximate with a factor $2 - \epsilon$ for any $\epsilon > 0$, assuming the unique games conjecture. This means that the 2-approximation algorithm by Garg *et al.* could very well be the best possible approximation algorithm for Multicut in Trees.

Details about the approximation algorithm. The algorithm by Garg *et al.* computes the multi-commodity flow in the tree, where the commodities are the demand pairs in the instance. They proved that the size of the minimum solution for a Multicut in Trees instance is between the maximum integral multi-commodity flow in that instance and twice that flow value. Reduction Rule 4.1.5 partially reuses this result.

3.4 Multicut on General Graphs

Multicut on general graphs is also proven to be FPT. This proof was given independently by Marx *et al.* [36] and Bousquet *et al.* [6] within weeks of each other. The preprints of their respective papers are both from 2010 [35, 5].

Marx *et al.* presented an $O^*(2^{O(k^3)})$ time algorithm for the problem, and Bousquet *et al.* an algorithm that runs in $O^*(k^{O(k^6)})$ time. Both groups of authors noted that their focus was not to give an efficient algorithm.

An approximation algorithm. Garg *et al.* [20] presented the best-known approximation algorithm for Multicut in general graphs. This algorithm has an approximation ratio of $O(\log p)$, where p is the number of demand pairs to be separated.

Another parameter than k. All results mentioned above are achieved using k, the maximum size of the solution, as a parameter. Gottlob and Lee [22] showed Multicut on general graphs is FPT for a parameter ω^*, instead of k. This ω^* represents the treewidth of the "input structure". The input structure contains an extra edge between the two nodes of every demand pair.

A practical algorithm based on dynamic programming was given later by Pichler *et al.* [48]. Their algorithm runs in $O(2^{2\omega^* \log \omega^*} \cdot |I|)$ time, where $|I|$ is the size of this input structure.

4 Algorithms

This section explains the reduction rules used in the algorithms by Guo and Niedermeier [23], Bousquet *et al.* [4] and Chen *et al.* [10]. Even though we do not use the algorithm by Chen *et al.* in this research, we explain their reduction rules in Subsection 4.3. It is safe to skip the subsection about their algorithm.

We gave all definitions necessary to understand the first two algorithms in Subsection 2.3. Appendix A contains these definitions as well. We explain the definitions for the third algorithm between the reduction rules where needed.

We note that the notation and definitions used in this section sometimes differ from the original notation. We refer the reader to the original articles for proofs regarding the correctness of the reduction rules and the kernel size.

4.1 Guo and Niedermeier: An $O(k^{3k})$ Sized Kernel

Guo and Niedermeier [23] were the first to prove that a kernel for Multicut in Trees exists. Their resulting kernel is one of size $O(k^{3k})$, and they used eight reduction rules to achieve this kernel.

Reduction Rule 4.1.1 (Idle Edge).
If there is an edge in the instance with no demand path passing through it, contract this edge.

This straightforward rule gets rid of all unnecessary edges in the instance.

Reduction Rule 4.1.2 (Unit Path).
If a demand path has length one, cut its corresponding edge.

This edge is the only candidate to destroy the demand path of length one. We have to include it in the solution.

Reduction Rule 4.1.3 (Dominated Edge).
If all demand paths that pass through an edge e_1 also pass through another edge e_2, contract e_1.

It is never better to include e_1 in the solution than to include e_2. Since cutting e_2 would destroy at least the same demand paths as cutting e_1, we do not have to consider e_1 by itself.

Reduction Rule 4.1.4 (Dominated Path).
If $P_i \subseteq P_j$ for two distinct demand paths $P_i, P_j \in P$, remove P_j.

Cutting any edge on P_i also destroys P_j. Since we have to destroy P_i, we do not need to consider P_j separately.

Reduction Rule 4.1.5 (Disjoint Paths).
If the instance has more than k pairwise edge-disjoint demand paths, there is no solution with parameter value k.

To destroy two pairwise edge-disjoint demand paths, we need to cut at least two edges. In order to cut $k+1$ pairwise edge-disjoint demand paths, we need to cut at least $k+1$ edges, which we are not allowed to do.

Reduction Rule 4.1.6 (Overloaded Edge).
If more than k length-two demand paths pass through an edge e, cut e.

If we were not to cut e, we would have to cut the other edge for each of these more than k demand paths. Given that we applied Reduction Rule 4.1.4 exhaustively, this would mean we have to cut more than k edges, which we are not allowed to do.

Reduction Rule 4.1.7 (Overloaded Caterpillar).
If there are $k+1$ demand pairs (v, u_1), (v, u_2), …, (v, u_{k+1}) such that nodes u_1, u_2, …, u_{k+1} all belong to the same caterpillar component that does not contain v, then (one of) the longest of these demand paths can be removed.

Let C be the caterpillar component u_1, u_2, …, u_{k+1} belong to. Because all paths in a tree are unique, all demand paths going from the nodes u_1, u_2, …, u_{k+1} to v have to leave C over the same edge. In order to separate these $k+1$ demand pairs, we need to cut an edge that is on at least two demand paths. Given that we applied Reduction Rule 4.1.4 already, we will always destroy one of the longest of these paths.

Reduction Rule 4.1.8 (Overloaded L_3-leaves).
If there are $k+1$ demand pairs (v, u_1), (v, u_2), …, (v, u_{k+1}) such that nodes u_1, u_2, …, u_{k+1} are all L_3-leaves of the same I_3-node u, remove all these demand pairs and add a new demand pair (u, v).

To separate these $k+1$ demand pairs, we have to cut an edge e that is on at least two of the corresponding demand paths. Given that we have applied Reduction Rule 4.1.4 exhaustively already, we know that e is on the path from u to v. Cutting e thus separates all of the $k+1$ demand pairs. Thus, we can replace these demand pairs with the single demand pair (u, v).

4.2 Bousquet *et al.*: An $O(k^6)$ Sized Kernel

Bousquet *et al.* [4] were the first to prove that a kernel with polynomial size exists. They used seven reduction rules in their algorithm.

Reduction Rule 4.2.1 (Unit Path)**.**
If a demand path has length one, cut its corresponding edge.

This is the same rule as Reduction Rule 4.1.2.

Reduction Rule 4.2.2 (Disjoint Paths)**.**
If the instance has more than k pairwise edge-disjoint demand paths, there is no solution with parameter value k.

This is the same rule as Reduction Rule 4.1.5.

Reduction Rule 4.2.3 (Unique Direction)**.**
If all demand paths starting at a leaf u go in the same direction, contract the edge incident to u. If all demand paths starting at an inner node v go in the same direction, contract the edge incident to v that is not on any demand path that starts at v.

This rule takes care of most edges without demand paths passing through them and dominated edges. This rule seems to be an optimised combined version of Reduction Rules 4.1.1 and 4.1.3 that we mentioned in Subsection 3.2.

Reduction Rule 4.2.4 (Dominated Path)**.**
If $P_i \subseteq P_j$ for two distinct demand paths $P_i, P_j \in P$, remove P_j.

This rule is the same as Reduction Rule 4.1.4.

Reduction Rule 4.2.5 (Common Factor)**.**
Let P_0 be a demand path. If there are $k+1$ other distinct demand paths P_1, P_2, …, P_{k+1} that each intersect P_0, such that for each $i, j \in \{1, 2, \dots, k+1\}$, $i \neq j$, the intersection of P_i and P_j is a subset of P_0, remove P_0 from P.

This rule only works if we have already applied Reduction Rule 4.2.4 exhaustively. Any solution must include an edge e that lies on at least two of P_1, P_2, …, P_{k+1}. Since the intersection of any combination of these demand paths is a subset of P_0, cutting e destroys P_0.

Reduction Rule 4.2.6 (Bidimensional Dominating Wingspan)**.**
If u is a bad L_2-leaf of a caterpillar C with a wingspan W such that $W \cap C$ dominates at least $k+1$ endpoint-disjoint demand paths, contract the edge incident to u.

Reduction Rule 4.2.7 (Generalised Dominating Wingspan)**.**
Assume u is a bad L_2-leaf of a caterpillar C, and that u covers C. If, for every closest P-neighbour z of u in $A(u)$, there exist $k+1$ endpoint-disjoint demand paths between nodes lying on the path from $b(u)$ to u and nodes on the path from z to $a(u)$, contract the edge incident to u.

We can implement both these reduction rules using a matching algorithm on an auxiliary graph.

4.3 Chen *et al.*: An $O(k^3)$ Sized Kernel

Chen *et al.* [10] showed an algorithm that results in a kernel with size $O(k^3)$. We only explain this algorithm for the sake of completeness. It is safe to skip this subsection.

In their paper, the authors use an input forest $\mathcal{F}$ instead of an input tree T. The most considerable difference between the modifications the reduction rules can make is that an edge removal can replace an edge contraction. When cutting an edge e, the algorithm also removes e from $\mathcal{F}$ instead of contracting e. This does not lead to any problems, as we remove all demand paths over e when we cut e. We can easily modify the rules to contract edges instead of removing them to make the algorithm work on a single tree. As we do not use this algorithm in our experiments, we have chosen to present them here the way they work on a forest.

Reduction Rule 4.3.1 (Idle Edge)**.**
If there is an edge e in the instance with no demand path passing through it, remove e from $\mathcal{F}$.

This rule is mostly the same as Reduction Rule 4.1.1.

Reduction Rule 4.3.2 (Useless Demand Pair)**.**
If there is a demand pair P_i between nodes u and v, where u and v are in different trees of $\mathcal{F}$, remove P_i from the set of demand pairs.

Obviously, u and v are already separated if they are in different trees of $\mathcal{F}$. We can never apply this rule when we restrict the instance to a single tree.

Reduction Rule 4.3.3 (Unit Path)**.**
If a demand path has length one, cut its corresponding edge.

This rule is identical to Reduction Rule 4.1.2.

Reduction Rule 4.3.4 (Disjoint Paths)**.**
If the instance has more than k pairwise edge-disjoint demand paths, there is no solution with parameter value k.

This rule is the same as Reduction Rule 4.1.5.

Reduction Rule 4.3.5 (Unique Direction)**.**
If all demand paths starting at a leaf u go in the same direction, contract the edge incident to u. If all demand paths starting at an inner node v go in the same direction, contract the edge incident to v that is not on any demand paths that starts at v.

This rule is the same as Reduction Rule 4.2.3. However, the definition of an *inner node* differs between the two reduction rules. Remember the inner node explained in Subsection 2.3 as an I_2-node without leaves.

The definition of an inner node given by Chen *et al.* for this reduction rule does not contain the restriction that the neighbours of the internal node cannot be leaves. Here, an *inner node* is an internal node with precisely two neighbours, which can be leaves.

Reduction Rule 4.3.6 (Dominated Path)**.**
If $P_i \subseteq P_j$ for two distinct demand paths $P_i, P_j \in P$, remove P_j.

This is the same rule as Reduction Rule 4.1.4.

Introducing an auxiliary graph. For some of the following rules, we need to define an auxiliary graph G. We create a node in G for each good leaf in $\mathcal{F}$. The edges in G exist between two nodes u and v if a demand pair between the corresponding leaves in $\mathcal{F}$ exists, such that these corresponding leaves are connected to the same neighbour.

Reduction Rule 4.3.7 (Bound on Good Leaves).
Apply Buss's kernelisation algorithm for Vertex Cover *[7] on G with parameter k: For each node u in G with degree at least $k+1$, cut the edge in $\mathcal{F}$ incident to the leaf corresponding to u. If the number of good leaves in $\mathcal{F}$ after the application of Buss's kernelisation algorithm is more than $2k^2$, there is no solution with parameter value k.*

All edges in G correspond to demand paths of length two, so this rule partially overlaps with Reduction Rule 4.1.6. As with that rule, we need to apply Reduction Rule 4.3.6 before this rule.

For the following rules, the authors assume Reduction Rules 4.3.1 to 4.3.7 are not applicable on the instance.

Reduction Rule 4.3.8 (Crown Reduction).
Apply the Crown Reduction algorithm [1] to G with parameter k to partition the nodes of G into three sets H, I, and O. If $|O| > 3k$ or $|H| > k$, there is no solution with parameter value k.

We named the sets H, I, and O after the original paper by Abu-Khzam *et al.* [1]. They have the following properties: First, I is an independent set of G, and there exists no edge between the nodes in I and those in O. Furthermore, there exists a minimum Vertex Cover in G that contains all nodes in H. There also exists a matching M that matches every node in H to a node in I, and finally $|O| \leq 3k$ if a solution for Vertex Cover on G with parameter k exists.

Groups. For the last four rules, we need to divide the nodes in $\mathcal{F}$ into three different types of groups: Type-I, Type-II and Type-III groups. A *Type-I group* consists of an internal node u that has at least one good leaf and all u's leaves. This group is *formed* by u. A *Type-II group* consists of an I_3-node v that does not have any good leaves, together with all v's leaves. Node v *forms* this group. Finally, we need to remove all Type-I and Type-II groups from T to define Type-III groups. The resulting connected components are caterpillars in which no leaves are good leaves. These caterpillars are then divided greedily into subcaterpillars such that there is no demand pair between two nodes in the same subcaterpillar. These subcaterpillars are called *Type-III groups.*

We refer to a Type-I, Type-II, or Type-III group simply by *group*, unless a specific type of group is required.

For any group γ_i, the number of edges with one endpoint in γ_i and one endpoint outside of γ_i is denoted by d_i.

Finally, these rules use $\pi(u)$ to denote the neighbour of a leaf u.

Reduction Rule 4.3.9 (Bound on the Number of Bad Leaves in a Group That Form a Demand Pair with a Certain Node).
Let u be a node, and let γ_i be a group. If there exist $\ell \geq d_i$ bad leaves in γ_i that each form a demand pair with u, replace the longest $\ell - d_i + 1$ of these demand pairs between a node v and u with the demand pair $(\pi(v), u)$.

Offsets. Consider two distinct groups γ_i and γ_j. Let u be a bad leaf in γ_i. If u forms a demand pair with an internal node w in γ_j, we call w the *node-offset of u with respect to γ_j*, or, in short, *n-offset$_j(u)$*. Similarly, if u forms a demand pair with any bad leaves in γ_j, consider w to be any such leaf with the minimum distance to u. Then we call w the *leaf-offset of u with respect to γ_j*, or *l-offset$_j(u)$*.

An ordering based on offsets. Consider two distinct groups γ_i and γ_j again. Let u be the node in γ_j that has the minimum distance to the nodes in γ_i. Then, for any two bad leaves w and z in γ_i we say $w \preccurlyeq_j^n z$ if the distance from the n-offset$_j(w)$ to u is not larger than the distance from the n-offset$_j(z)$ to u. Similarly, we say that $w \preccurlyeq_j^l z$ if the distance from the l-offset$_j(w)$ to u is not larger than the distance from the l-offset$_j(z)$ to u.

Reduction Rule 4.3.10 (Bound on the Number of Bad Leaves in a Group That Form a Demand Pair with Internal Nodes of Another Group).
Let γ_i and γ_j be two distinct groups. If there are $\ell \geq d_i$ bad leaves in γ_i that each form a demand pair with internal nodes of γ_j, replace the largest $\ell - d_i + 1$ with respect to the order $\preccurlyeq_j^n$ of these demand pairs between such a bad leaf $u \in \gamma_i$ and an internal node $v \in \gamma_j$ with the demand pair $(\pi(u), v)$.

Reduction Rule 4.3.11 (Bound on the Number of Bad Leaves in a Group That Form a Demand Pair with Bad Leaves in Another Group).
Assume Reduction Rule 4.3.9 does not apply. Let γ_i and γ_j be two distinct groups. If there are $\ell \geq d_j(d_i - 1) + 1$ bad leaves in γ_i that each form a demand pair with bad leaves of γ_j, replace the largest $\ell - d_j(d_i - 1)$ with respect to the order $\preccurlyeq_j^l$ of these demand pairs between such a bad leaf $u \in \gamma_i$ and a bad leaf $v \in \gamma_j$ with the demand pair $(\pi(u), v)$.

Obviously, this rule requires Reduction Rule 4.3.9 to be applied exhaustively before this rule.

The set OUT_u for a node u. Finally, the last reduction rule requires the notation of OUT_u for a node u that forms a Type-I group. Remember the auxiliary graph G consisting of good leaves and their demand pairs defined above. Let G_u be the subgraph of G induced by the good leaves attached to u. Reduction Rule 4.3.8 split G into the sets H, I and O. Consider the sets H_u, I_u and O_u to be the intersection of the nodes of G_u and H, I and O respectively. The matching M in G that matches H into I induces a matching M_u in G_u that matches H_u into I_u. Finally, define OUT_u to be the set of nodes in I_u that are not matched by M_u.

Reduction Rule 4.3.12 (Bound on the Number of Bad Leaves in a Group That Form a Demand Pair with Good Leaves in OUT_u for a Type-I Group γ_i Formed by a Node u).
Assume Reduction Rule 4.3.9 does not apply. Let u be a node that forms a Type-I group γ_i, and let γ_j be another group. If there are $\ell \geq d_j(d_i - 1) + 1$ bad leaves in γ_i that each form a demand pair with nodes in OUT_u, replace the longest $\ell - d_j(d_i - 1)$ of these demand pairs between a bad leaf v in γ_i and a node w in OUT_u with the demand pair $(\pi(v), w)$.

Again, it is clear Reduction Rule 4.3.9 must not be applicable on the instance when executing this rule. Because we need OUT_u, we also need to execute Reduction Rule 4.3.8 before this rule.

5 Implementation Details

This section explains the implementation details for the algorithms and their reduction rules. We implemented the algorithms in C#. The code and scripts used to run all experiments are available on GitHub [1]. There will be a footnote to Microsoft's documentation site when we mention a built-in data structure for the first time.

5.1 Data Structures

Many data structures, like the set of demand pairs or the edges on a demand path, are modified often. Ideally, we want to add, remove, and check whether an element is present in this data structure in $O(1)$ time. We have created our own data structure, as existing data structures do not achieve these time complexities. For instance, a `HashSet` [2] [41] would allow us to add and remove elements in $O(1)$ time, as well as checking whether it contains a specific element, but

[1] `https://github.com/stevenheinen/Thesis---Multicut-in-Trees`

[2] `https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.hashset-1?view=net-5.0`

its enumeration is slower than that of a `List` [3] [43]. On the other hand, a `List` with faster enumeration and a preserved order of elements cannot remove an element in $O(1)$ time.

Our own data structure consists of a `Dictionary` [4] [40] and a `LinkedList` [5] [42]. It might be best compared to an `OrderedDictionary` [6] [44], except that it is explicitly typed. We refer to this data structure throughout this thesis as "`OrderedDictionary`" [7] . We use the `Dictionary` to check the presence of an element in the data structure in expected $O(1)$ time, as well as to go to the correct node in the `LinkedList` given an element. For enumeration, we use the `LinkedList`. Other operations, like adding and removing elements, can be done in $O(1)$ time, making this data structure perfect for how we want to use it.

5.2 Algorithm Data Structures

An algorithm contains multiple data structures with the information required to compute a kernel. For instance, many reduction rules need information about which demand paths go over a certain edge. This is saved as a `Dictionary` with per edge an `OrderedDictionary` with demand pairs. We used a similar data structure to store, for each node, the demand pairs that start at that node. We compute these data structures when the algorithm starts and update them each time the instance gets modified. The data structures do not contain edges with no demand path passing through them or nodes that are not the starting point of a demand pair. When a reduction rule needs information from one of these data structures, it can just grab it.

Data structures in reduction rules. Reduction rules themselves have access to three more data structures. These data structures contain information about parts of the instance that the algorithm modified. A reduction rule can use this data to restrict the part of the instance it needs to check. We talk more about this in Subsection 5.4. First, we explain how the different input modifications work.

5.3 Input Modifications

Performing a modification like an edge contraction might be more complicated than expected because we need to update multiple data structures. This subsection explains the four modifications that the reduction rules can apply to the instance.

5.3.1 Contracting Edges

The first input modification we explain is the edge contraction. When contracting an edge between nodes u and v, we replace u and v by a new node w that is connected to all neighbours of u and v, except for u and v themselves.

Updating the caterpillar components. Some reduction rules depend on the information about which caterpillar component a node is in. This information can change when the algorithm contracts an edge and thus needs to be updated. For instance, contracting an edge between an

[3]`https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1?view=net-5.0`

[4]`https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.dictionary-2?view=net-5.0`

[5]`https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.linkedlist-1?view=net-5.0`

[6]`https://docs.microsoft.com/en-us/dotnet/api/system.collections.specialized.ordereddictionary?view=net-5.0`

[7]In the code, the data structure is called a `CountedCollection`. We explain the reason for the prefix "`Counted`" in Subsection 7.1. It is not to be confused with the existing `Collection` [8] [39] data structure.

[8]`https://docs.microsoft.com/en-us/dotnet/api/system.collections.objectmodel.collection-1?view=net-5.0`

I_1-node and an I_2-node means that the I_2-node becomes an I_1-node. It will then not be a part of a caterpillar component anymore. Most of the time, we can easily update this information. When an I_3-node becomes an I_2-node, we must merge two caterpillar components. In such a case, all caterpillar components are recomputed instead of updated. We recompute them later.

Contracting the edge. In this step, we contract the edge in the tree. We start by updating the type (I_1/L_2/etc.) of the edge's endpoints, and potentially their neighbours and the leaves of those neighbours. After that, we choose one of the edge's endpoints as the node that results from the edge contraction. Let us call it v and let u be the other endpoint of the edge. We change the endpoint of the edges connected to u from u to v. We can then remove u from the graph.

At this moment, we recompute the caterpillar components if necessary.

Updating demand pairs and paths. We also need to update the demand paths that went over the contracted edge e. We remove e from each demand path over e. If e is either the first or last edge on the demand path, we update the endpoint of its corresponding demand pair. Finally, we remove e from the `Dictionary` with information about demand pairs per edge.

We need to update more demand pairs. The edge contraction can also mean that an endpoint of a demand path that does not use e changes. All demand paths that start at u now begin at v. So, for each demand pair that starts at u, we change its endpoint from u to v. This change does not change its demand path since we already changed the edge previously connected to u. We then remove u from the `Dictionary` with demand pairs per node.

Provide the reduction rules with information. The final thing we have to do is update the information available to the reduction rules. We provide each reduction rule with the following information: the contracted edge, the node resulting from this edge contraction, and the demand paths that use this edge.

Time complexity. Let us briefly analyse the time required for an edge contraction. Updating caterpillar components is done using a depth-first search and takes $O(n)$ time. The same goes for updating the endpoint of the edges connected to the contracted edge and updating the nodes' types. We loop over $O(p)$ demand pairs twice to update their endpoints and path. These updates are constant time operations, so this part takes $O(p)$ time in total. Providing the reduction rules with information takes $O(1)$ time, as the number of reduction rules is constant. These operations sum up to $O(n + p)$ time.

5.3.2 Removing Demand Pairs

Removing a demand pair is simpler than contracting an edge. The process of removing a demand pair P_i is as follows.

Modification details. We start by providing the reduction rules with the information that we removed P_i. We then remove P_i from the demand pairs that start at its endpoints in the data structure in the algorithm. The other data structure that needs to be updated is the one containing demand pairs per edge. By looping over the edges on its demand path, we can remove P_i from these edges in this data structure. Finally, we remove P_i from the set of demand pairs.

Time complexity. Updating the demand pairs per edge is the only operation that we cannot do in constant time. Instead, this part requires $O(n)$ time. Since all other operations require constant time, the time complexity of removing a demand pair is $O(n)$.

5.3.3 Changing Demand Pairs

The third operation is changing the endpoint of a demand pair. Remember that we can only change a demand pair by removing edges from one of the ends of its corresponding demand path. Let us consider changing the endpoint of a demand pair P_i from u to v.

Modification details. First, we gather the edges that we will remove from the demand path by walking over the path from u until we encounter v, and we remove the encountered edges from P_i. We then remove P_i from the demand pairs starting at u and add P_i to the demand pairs starting at v. We provide the reduction rules with information about this instance modification: the edges removed from P_i and P_i itself. Finally, for each edge removed from P_i, we remove P_i from the demand paths over this edge.

Time complexity. Looping over the demand path to gather the edges that we must remove and looping over these edges again to update the data structure in the algorithm takes $O(n)$ time in total. We can perform all other operations in constant time, so this modification takes $O(n)$ time.

5.3.4 Cutting Edges

The final instance modification is cutting an edge. This modification mostly overlaps with contracting an edge and removing demand pairs. Consider cutting an edge e.

Modification details. We start by adding e to the partial solution the algorithm found and decrease k by one. Then, we remove all demand pairs that go over e using the method described in Subsection 5.3.2. We contract e, using the method described in Subsection 5.3.1. These two methods already provide the reduction rules with the information they need, so we do not need to do anything else.

Time complexity. Adding e to the solution takes $O(1)$ time. Removing the $O(p)$ demand pairs takes $O(n)$ time each, for a total of $O(np)$ time. Contracting e requires $O(n + p)$ time. In total, this instance modification thus takes $O(np)$ time.

5.4 Reduction Rules Implementations

This subsection explains how we implemented each reduction rule. We start by providing some general details for reduction rules. Then, we describe the implementation details for each unique reduction rule. When we analyse the time of a reduction rule, this time does not include the time needed for the modifications themselves.

Applying a set of reduction rules. In the original algorithms, the authors present the reduction rules in a specific order. In our implementation, we apply the reduction rules in the same order. We only apply a rule if all rules before it are not applicable on the instance. After the algorithm applies a rule successfully, it tries to apply the first rule again. If it applies a rule unsuccessfully, it moves on to the next rule. Some rules can determine that an instance is not solvable. In such a case, the algorithm stops. Otherwise, the algorithm keeps applying reduction rules until it no rule is applicable on the instance.

Two ways to apply a reduction rule. There are two possible ways to apply a reduction rule. Each time we execute a reduction rule, we can either apply it on a single edge or demand pair or scan the entire instance to find all possible edges or demand pairs the rule can modify. In the second case, we find all places where we can modify the instance before applying any of these modifications. Both options should lead to the same result. In this research, we used the second option where possible.

Restricting the part of the instance to check. The first time a reduction rule is applied, we check the entire instance to see if it is applicable. For some rules, it is possible to restrict the part of the instance it needs to check when we apply it after its first application. We use the three data structures mentioned earlier with information about which parts of the instance were modified to achieve this. For instance, Reduction Rule 4.1.2 looks for demand paths of length one. When we apply it for the i^{th} time, we do not need to consider demand paths that did not change since the $i - 1^{\text{st}}$ application of this rule since it would give the same results on these demand paths as it did then.

5.4.1 Reduction Rule 4.1.1 (Idle Edge)

Remember that the Idle Edge rule contracts edges that are not used by any demand path. To implement this rule, we are using the information about which demand paths go over which edge.

First application. During the first application of this rule, we loop over all edges in the tree. We gather all edges that are not used by any demand path in a set. Once we have looped over all edges, we contract all edges in the set. In total, gathering the edges to be contracted takes $O(n)$ time.

Later applications. In later applications of this rule, we only need to check edges that were used by a demand path and are no longer used by that demand path. This situation exists when the algorithm deleted a demand pair from the instance or changed a demand path. We check every edge on all removed demand paths and all edges that are no longer part of shortened demand paths. These are first gathered in a `HashSet` to ensure we check each edge at most once. Since we check $O(n)$ edges during a later application, this rule takes $O(n)$ time in later applications [9].

5.4.2 Reduction Rules 4.1.2 and 4.2.1 (Unit Path)

The Unit Path rule finds demand paths of length one and cuts their edge.

First application. We check the length of each demand path in the first application of this rule. This operation can be done in $O(1)$ time for each of the $O(p)$ demand paths, leading to a running time of $O(p)$.

Later applications. In later applications, we check demand paths for which their length decreased. This happens when the algorithm changes a demand path but can also happen when it contracts an edge. As with the previous rule, we first gather the set of demand pairs to check in a `HashSet`. For a contracted edge, we add all demand paths that used that edge to this set. We add every modified demand pair as well. Since we check $O(p)$ demand paths, this rule takes $O(p)$ time in a later application [10].

[9]Technically, gathering the edges to be checked takes $O(np)$ time additionally in the current implementation. It is possible to eliminate this time by gathering these edges when we perform the modifications on the instance.

[10]Gathering the set of demand paths to be checked takes $O(np)$ time in the current implementation. Again, it is possible to eliminate this time by gathering the set when we apply the modifications

5.4.3 Reduction Rule 4.1.3 (Dominated Edge)

The Dominated Edge reduction rule is slightly more complicated than the two rules before. It checks for two edges e_1 and e_2 whether the set of demand paths that go through e_1 is a subset of the demand paths that go through e_2. If that is the case, we contract e_1.

A more efficient way to check applicability. A naive implementation of this rule would check every combination of two edges in the tree. However, most of the time, checking just a few of these combinations is sufficient.

Observation 3.
When applying the Dominated Edge rule on an edge e, we only need to compare its demand paths to the demand paths of all edges (except for e) on one of the demand paths that go through e.

Proof. Let e be an edge in the tree, let P_e be the set of demand paths through e, and let P_0 be any demand path in P_e. We have two cases. Either P_0 has a length of one, or P_0 is longer.

In the first case, e will never be contracted by the Dominated Edge reduction rule, as P_e will never be a subset of the demand paths through any edge other than e. We thus do not need to compare P_e to the set of demand paths on any other edge.

Let us now consider the other case, when P_0 has a length of at least two. Consider an arbitrary edge e', $e' \neq e$, in the tree. Denote by $P_{e'}$ the set of demand paths through e'. If $P_e \subseteq P_{e'}$, we need to have $P_0 \in P_{e'}$ as $P_0 \in P_e$. This is only the case when e' is on P_0.

To conclude, we only need to compare P_e to the set of demand paths that use e'', for every edge $e'' \neq e$ on P_0. □

Applying the rule. To apply this reduction rule, we compare each edge e to all edges $e' \neq e$ on the shortest demand path through e. We find this shortest demand path by looping over all demand paths through e. This takes $O(p)$ time, and result in $O(n)$ candidate edges e'. Checking whether the set of demand paths through e is a subset of the set of demand paths through e' can be done in $O(p)$ time using `HashSets`. Once we determine the demand paths through e are a subset of the demand paths through an edge e', we mark e and move on to the next edge. Per edge, this reduction rule takes $O(np)$ time. After we have checked every edge, we contract the marked edges.

Edges dominating each other. With the description above, edges can dominate each other. Consider as an example a small tree with three nodes: u, v and w. Let u and v be leaves and w an internal node. Denote by e_1 the edge between u and w, and by e_2 the edge between v and w. Let there be a single demand pair in this instance, between u and v. We give a visualisation of this example in Figure 2. The set of demand paths that use e_1 is a subset of the set of demand paths that use e_2 and vice versa. This means that this rule would contract both e_1 and e_2. However, this should not be allowed as we then remove a demand pair from the instance without separating it.

To resolve this, we add another restriction to the edges we compare each edge e to. Besides only comparing e to all edges $e' \neq e$ on the shortest demand path through e, we require every edge e' not to be marked. This way, at least one edge per demand path will be left in the instance after applying this reduction rule.

First application. In its first application, we apply this rule on every edge. For each of the $O(n)$ edges, the rule needs $O(np)$ time to check whether we can contract this edge. The first application of this reduction rule thus takes $O(n^2p)$ time.

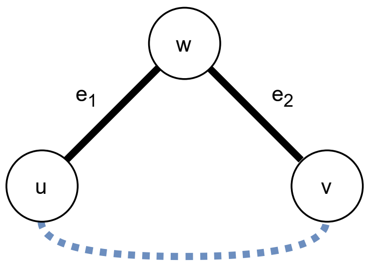


Figure 2: A small tree with an internal node w, leaves u and v, two edges: e_1 between u and w and e_2 between v and w, and a single demand pair (displayed as a dotted blue line) between u and v.

Later applications. In later applications, we only check edges that had their set of demand paths change. These edges are all edges on every removed demand path and all edges that are no longer part of shortened demand paths. The running time of a later application of this rule is also $O(n^2p)$, as we still check $O(n)$ edges.

5.4.4 Reduction Rules 4.1.4 and 4.2.4 (Dominated Path)

The Dominated Path reduction rule removes demand paths from the instance that are a superset of another demand path in the instance.

Avoiding subset checks. We seem to have to check whether a set is a subset of another set, the sets here being the respective edges on two demand paths. However, we can use knowledge about our instance to make this check easier.

Observation 4.
When applying the Dominated Path rule on demand paths P_i and P_j, we only need to check whether the first and last edge of P_i are on P_j or the other way around.

Proof. Consider two distinct demand paths P_i and P_j. Let e_1 be the first edge of P_i, and let e_2 be the last edge of P_i.

Assume e_1 and e_2 are part of P_j. Path P_j thus consists of a (potentially empty) set E' of edges from the start of P_j to e_1, a set E'' from e_2 to the end of P_j, and a set E''' with the path from e_1 to e_2. We know that paths in a tree are unique. In other words, there is only one possible path E''' from e_1 to e_2. All edges in E''' are part of P_i, since P_i goes from e_1 to e_2. Because of the definition of E''', all edges in E''' are also part of P_j.

Thus, we can conclude that $P_i \subseteq P_j$ if the first and last edge of P_i are part of P_j. □

Applying the rule. We now explain how exactly we apply this reduction rule for a demand path P_i. We find all demand paths that are a superset of P_i and remove these. To gather the set of demand paths to compare P_i to, we use similar reasoning as in the implementation of Reduction Rule 4.1.3. All demand paths that are a superset of P_i must go over the first edge of P_i. We can directly query for which demand paths this is the case.

We check whether the last edge of P_i is part of any of these demand paths. If so, we mark this other demand path.

We can check whether an edge is part of a path in $O(1)$ time using our own data structure. Because we loop over $O(p)$ demand paths that go over the first edge of P_i, this reduction rule requires $O(p)$ time per demand path.

Once we checked every demand path, we removed those that were marked.

Demand paths dominating each other. We run into a similar problem as we saw with the previous reduction rule. Two demand paths can dominate each other.

Consider a simple tree with three nodes and two edges. Let u and v be the tree's leaves and w its internal node. Let there be two demand pairs: (u, v) and (v, u). See Figure 3 for a figure of this example. Both demand paths use both edges in the tree, and they are thus a subset of each other. With the implementation described above, the algorithm would remove both demand paths from the instance.

However, we want to remove at most one demand pair. We only consider as possible subset demand paths the demand paths that are not already marked. By this, we mean that we do not try to find demand paths that are a superset of a demand path that the rule will already remove.

In our small example, we mark one of the paths. Then, the algorithm cannot remove the other path since there is no candidate demand path in the tree that is a subset of this other path. Because of the transitivity of subsets, this does not break the reduction rule.

First application. During the first application, we apply this rule to all demand paths. Since the rule required $O(p)$ time per demand path, its first application requires $O(p^2)$ time.

Later applications. We can restrict the set of demand paths this rule checks in later applications. We check all demand paths that went over contracted edges and demand paths that changed. Additionally, we make sure not to check any removed demand paths. Since we check $O(p)$ demand paths, the rule still requires $O(p^2)$ time [11].

5.4.5 Reduction Rules 4.1.5 and 4.2.2 (Disjoint Paths)

This reduction rule checks whether more than k edge-disjoint demand paths exist in the instance. We used the maximum multi-commodity flow in trees algorithm by Garg *et al.* [21] to implement this rule. We refer the reader to Theorem 4.1 in the paper by Garg *et al.* for details about this algorithm.

Applying the reduction rule. We directly used the multi-commodity flow algorithm for this reduction rule. When providing the algorithm with the endpoints of the demand pairs as commodities, the resulting flow is equal to the number of edge-disjoint demand paths.

We apply this algorithm in each application of the reduction rule and do not use any information about modified parts of the graph to speed it up. Trying to do so would have required complicated changes to the original algorithm, and we chose not to spend time on that.

[11] Gathering the set of demand paths to be checked takes $O(np)$ time in the current implementation. As before, it is possible to eliminate this time by creating this set when we perform the modifications

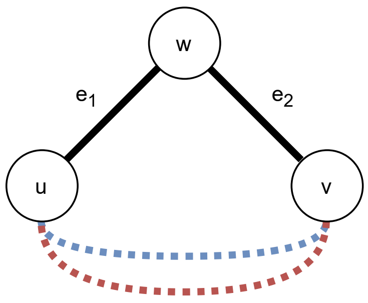


Figure 3: A small tree with an internal node w, leaves u and v, two edges: e_1 between u and w and e_2 between v and w, and two demand pairs: one between u and v (displayed as a dotted blue line) and one between v and u (displayed as a dotted red line).

The paper by Garg *et al.* [21] tells us that the running time of this reduction rule is $O(n^2p)$.

5.4.6 Reduction Rule 4.1.6 (Overloaded Edge)

The Overloaded Edge rule tries to cut edges that are used by more than k demand paths of length two. We discuss how we apply the rule and why we cannot cut multiple edges with it in one application. Then, we look at the rule during its first application and discuss why later applications are the same as the first application.

Applying the rule. We apply the reduction rule by looping over all demand paths. The rule only considers paths of length two. We keep track of how many times each edge occurred on these paths. Once we encounter an edge for the $k + 1^{\text{st}}$ time, we cut it and stop executing this reduction rule.

Applying the rule on multiple edges in one application. Trying to cut multiple edges in one application of this rule is not easy. Since the rule depends on k, and k decreases when we cut an edge, we must consider this when checking edges. We can encounter an edge exactly k times, and later another edge $k+1$ times. When we cut the second edge, we would have to go back to all edges we already checked and check whether they occurred k times and cut those. However, the occurrences of those edges can decrease. When we cut an edge, we namely also remove demand pairs from the instance. We decided not to modify the rule to make this work and instead chose to apply it on at most one edge in each application.

First application. During the first application, we check every demand path. We do a constant amount of work per demand path, so the running time of the first application is $O(p)$.

No smarter way in later applications. This rule cannot use information about modified parts of the instance to restrict what it checks in later applications. This is again because it depends on k. Consider a case where an edge e occurs s times on demand paths of length two, and let $s = k$. Now, we cut an edge on the other side of the graph. Assume e still occurs s times. Now, $s = k+1$, so we would have to cut e. However, since there were no modifications in the instance where e is, we would not look at it. Thus, we decided to check the entire instance every time this reduction rule is applied. The rule thus has a running time of $O(p)$ in later applications as well.

5.4.7 Reduction Rule 4.1.7 (Overloaded Caterpillar)

The Overloaded Caterpillar reduction rule tries to find $k + 1$ demand pairs from a node v in caterpillar component C to nodes in a caterpillar component C', $C \neq C'$.

Applying the rule. The rule loops over all nodes, skipping nodes that are not I_2-nodes or L_2-leaves, as these are not part of a caterpillar component. It also skips nodes where at most k demand pairs start. For each node v, we check the caterpillar component C' of the other endpoint of the demand pairs that start at v. If C' is the same as the caterpillar component of v, we skip this demand pair. We store the demand pairs per destination caterpillar component. After checking all demand pairs, we check if at least $k+1$ go to the same destination caterpillar component. If so, we remove the longest of the demand paths corresponding to these pairs and stop executing this rule.

As before, we do not apply the rule on multiple demand pairs in one application, as it depends on k.

An application of the rule. This rule is the first in the algorithm where caterpillar components are required. We compute these components in the first application of this rule. The rule does not recompute the caterpillar components in later applications, as the algorithm updates them during modifications. Computing the caterpillar components takes $O(n)$ time.

Per node, we need to loop over all $O(p)$ demand pairs starting at that node. All other operations require constant time. We check each demand pair at most twice, so in total, we also check $O(p)$ demand pairs. The running time of this rule then sums to $O(n+p)$.

As with the previous rule, we did not restrict the part of the graph the rule checks in later applications because it depends on k.

5.4.8 Reduction Rule 4.1.8 (Overloaded L_3-leaves)

The Overloaded L_3-leaves reduction rule tries to find $k+1$ demand pairs between a node v and L_3-leaves connected to the same I_3-node u.

Applying the rule. There are many similarities between the implementations of this rule and the previous rule. We loop over each node v where at least $k+1$ demand pairs start. For each of these demand pairs, we check its other endpoint w. We track how many times each I_3-node occurred as the second-last node on the corresponding demand paths. If w is an L_3-leaf, we grab its neighbour u and store the demand pair. After checking all demand pairs, we check whether an I_3-node occurred at least $k+1$ times. If so, we remove all but one of the demand pairs between v and leaves connected to this node. We then change the endpoint of the final demand pair from the L_3-leaf to its neighbour.

Again, we only apply this rule at most once during each application since it depends on k.

Running time. Using a similar argument as for the previous rule, the running time of this rule is $O(n+p)$.

There are no differences between the different applications of this rule as it depends on k.

5.4.9 Reduction Rule 4.2.3 (Unique Direction)

The Unique Direction reduction rule checks whether all demand paths starting at a node go in the same direction. If so, it contracts an edge.

Demand paths going in the same direction. The rule works differently on leaves and inner nodes. Remember, an inner node is not the same as an internal node. For each node v, the reduction rule checks whether v is a leaf, inner node, or neither.

First, we handle the case when v is a leaf. If zero or one demand paths start at this node, we can apply the rule here. Otherwise, we grab the second edge (from v) of the first demand path that starts at v. Call this edge e. For each other demand path that starts at v, we check whether e is part of this demand path. If not, not all demand paths from v go in the same direction. If we determine that all demand paths from v go in the same direction, we mark the edge connected to v and move on to the next node.

If v is an inner node, the process is similar. Instead of looking at the second edge of the demand paths, we look at the first edge. Call this edge e again. If all demand paths from v go in the same direction, we mark the edge connected to v that is not e.

Once we have processed every node, we contract all marked edges.

Contracting too many edges. When the process described above is applied, we risk removing demand pairs without separating them. We have seen this complication before, in the Dominated Edge rule. We resolve it similarly.

First, let us give an example of how this can happen. Consider a small graph with three nodes: an internal node w, and two leaves u and v. Let P_0 be a demand pair between u and v, and let us apply the Unique Direction reduction rule to this instance. Figure 2 shows this small example. All demand paths starting at u go in the same direction, and all demand paths starting at v as well. This means we would mark and contract both edges in the graph. The demand pair P_0 would disappear without being separated.

We modify the process described above slightly to make sure at least one edge per demand path remains. Before we mark a certain edge, we look at all demand paths that use that edge. If there are none, we can safely contract the edge. Otherwise, we check for each of these paths whether all other edges on that path are already marked. If so, we cannot contract this edge, and we skip it.

In the small example, assume we marked one of the edges. Then, we cannot contract the other one, as that would mean that there would be no edges left on P_0.

First application. We try to apply the reduction rule on every node in the first application. Checking whether all demand paths go in the same direction takes $O(p)$ time per node. However, the additional check to ensure we can contract a certain edge takes $O(np)$ time per candidate edge.

We check all demand pairs at most twice. We can sum the running time to check whether all demand pairs go in the same direction to $O(p)$ time. However, we still need to perform the additional check for $O(n)$ edges. The total running time thus is $O(n^2p)$.

Later applications. In later applications of this rule, we can apply this rule only on modified parts of the graph. We check the endpoints of each removed demand pair and the endpoints of the edges that are no longer part of a shortened demand path. Additionally, we check each node that resulted from an edge contraction, as contracting an edge can also lead to a different endpoint of a demand pair. Since we still check $O(n)$ nodes, the running time of this reduction rule is $O(n^2p)$ during later applications as well.

5.4.10 Reduction Rule 4.2.5 (Common Factor)

This reduction rule wants to find, for a demand path P_0, $k+1$ other (distinct) demand paths that each intersect P_0, and whose pairwise intersections are all a subset of P_0.

Applying the rule. Bousquet *et al.* [4] explained how this rule could be applied using a matching algorithm. We compute the matching in an auxiliary graph. Consider a demand path P_0 on which we want to apply this rule.

To create the auxiliary graph, we create a set Z of edges that do not lie on P_0 but share a node with P_0. We also gather all demand paths that start at nodes on P_0. Because of Reduction Rule 4.2.4, we know that these demand paths leave P_0. We gather the edges through which these demand paths leave P_0 in a set Y. We loop over all edges in Z and check whether a demand path on this edge starts at P_0 to find Y. It is possible to identify $|Y|$ demand paths that intersect P_0 and do not pairwise intersect each other outside of P_0.

We still need to consider demand paths that leave P_0 over two edges. For this, we use the auxiliary graph. We create a node for each edge in $Z \setminus Y$. Then, we create an edge between two nodes if a demand path uses both edges that correspond to these nodes.

Finally, we compute a matching in the auxiliary graph. If a matching of size $k+1-|Y|$ exists, we can remove P_0.

Running time. Gathering Z can be done in $O(n)$ time and gathering all demand paths that start at a node on P_0 in $O(n+p)$ time. Computing Y requires $O(n+p)$ time since, for each edge in Z, we need to consider all demand paths that start in P_0. The total number of checked demand paths sums to $O(p)$.

The auxiliary graph has $O(n)$ nodes. Finding the edges is done in $O(n^2 + p)$ time. For each combination of two nodes in the auxiliary graph, we check $O(p)$ demand paths. This way, we check each demand path at most twice, so this factor sums to $O(p)$.

To compute the matching in the graph, we use the blossom algorithm by Edmonds [15]. The auxiliary graph has $O(n)$ nodes and $O(p)$ edges. Computing the matching thus takes $O(n^2p)$ time.

The total time complexity of this reduction rule is $O(n^2p)$ per demand path.

A faster matching algorithm, like the algorithm by Micali and Vazirani [38], would run in $O(\sqrt{n}p)$ time on the auxiliary graph. With this algorithm, the time complexity of this reduction rule would be $O(\sqrt{n}p + n^2)$ per demand path. We have chosen not to use this algorithm, as we could not find an easily usable implementation, and it was too complicated to implement in the available time.

An application of the rule. This reduction rule depends on k as well. As before, we only apply it on at most one demand path per application. Additionally, we do not reduce the part of the instance that the rule checks in later applications. The rule thus runs in $O(n^2p^2)$ time in all its applications.

5.4.11 Reduction Rule 4.2.6 (Bidimensional Dominating Wingspan)

The Bidimensional Dominating Wingspan rule tries to find a bad L_2-leaf u, such that the intersection of the caterpillar C and the wingspan W of u dominates at least $k + 1$ endpoint-disjoint demand pairs. As in the proof in the paper by Bousquet *et al.* [4], we used an auxiliary graph per bad L_2-leaf and computed a matching in this graph.

Consider we want to apply the reduction rule on a L_2-leaf u. Let C be the caterpillar u is part of.

Finding the wingspan. To find the wingspan W of u, we need to know the two minimal demand paths that start at u and go in different directions from the neighbour of u. Call the two edges between internal nodes from the neighbour of u the “left” and “right” edge without loss of generality. For each demand pair that starts at u, check whether its path goes over this left or right edge. The rule keeps track of the minimal demand path for both of these edges. After checking every demand pair that starts at u, the other endpoints of the two minimal demand paths are saved as the endpoints of the wingspan of u. If such an endpoint is a leaf, the neighbour of that leaf is the endpoint of W instead. Finally, if there is no minimal demand pair going left or right, we say the neighbour of u is the endpoint of W for that side.

In this step, we also identify if u is a good leaf. If we encounter a demand path that does not use the left or the right edge, u is a good leaf, and we move on to the next leaf.

Finding the intersection between W and C. To find the intersection between W and C, we first find the path between the endpoints of W. We loop over each node on this path and check if it is in C. Denote by c the caterpillar component of u. A node is in C when it is in c, or when it is an I_1-node connected to an I_2-node in c. All leaves of the nodes on this path in the intersection are part of the intersection as well.

Creating the auxiliary graph. The nodes in the intersection between W and C form the nodes in the auxiliary graph. We add an edge to the auxiliary graph between two nodes if a demand pair exists between the two corresponding nodes in the original instance. We create these edges by looping over all demand pairs and checking their endpoints.

We compute a matching using Edmonds's blossom algorithm [15]. Once the first matching of size $k+1$ is found, we stop the rule and contract the edge connected to u.

We only contract at most one edge per application, as this rule also depends on k.

Running time. For a brief analysis of the running time of this reduction rule, we start by finding the endpoints of a wingspan. This process takes $O(p)$ time. It takes $O(n)$ time to find the path between the wingspan's endpoints and the intersection between the caterpillar and the wingspan. Creating the edges in the auxiliary graph takes $O(p)$ time. The matching can then be computed in $O(n^2 p)$ time, leading to a total running time of $O(n^2 p)$ per leaf [12].

An application of the rule. For this reduction rule, we need to know the caterpillar component for each node again. Just like Reduction Rule 4.1.7, this is the first rule in the algorithm that needs this data. We start the first application by computing this information. Later applications will not compute the caterpillar components again.

Then, we perform the process described above for each leaf. This leads to a total running time of $O(n^3 p)$. Since the rule depends on k, there is no efficient way to run the rule in a later application.

5.4.12 Reduction Rule 4.2.7 (Generalised Dominating Wingspan)

The Generalised Dominating Wingspan can apply on a bad L_2-leaf u that covers its caterpillar C. It also tries to find a set of $k+1$ endpoint-disjoint demand paths, but the area where these demand paths start slightly differs from the previous rule.

Finding whether u covers C. We find the minimal demand paths going left and right from u using the method described in the previous reduction rule. The rule then finds the extremities of C, using the available information about caterpillar components. These extremities should each lie on one of the minimal demand paths. If so, u covers C.

Computing the closest P-neighbours. The next step is computing the closest P-neighbours of u. Using the length of the minimal demand paths, we can find these P-neighbours by looping over all demand paths that start at u. If a demand path goes to the left and its internal length is at most the internal length of the minimal demand pair going left, the other endpoint of this path is (one of) the closest P-neighbour(s) of u. If this other endpoint is a leaf, this leaf's neighbour is (one of) the closest P-neighbour(s) of u instead of the leaf itself. We do the same for demand paths to the right.

The auxiliary graph. We create an auxiliary graph for each of the found P-neighbours. Without loss of generality, assume that the current closest P-neighbour z of u lies to the left of u. We find the path between z and the left extremity of C. The nodes on this path, and their leaves, will be nodes in the auxiliary graph. We call these nodes in the auxiliary graph the "left" nodes for now.

We then find the path between u and the right extremity of C. The nodes on this path and their leaves are nodes in the auxiliary graph as well. We call these the "right" nodes.

We add an edge between a node on the left and a node on the right if a demand pair exists between the corresponding nodes in the original graph. We do this by looping over all demand pairs.

We use Edmonds's algorithm [15] again to compute a matching in this auxiliary graph. Once we find a matching of sufficient size, we contract the edge connected to u and stop the reduction rule.

[12] This running time could be improved using the matching algorithm by Micali and Vazirani [38].

Running time. The analysis of the running time of this rule is similar to the analysis used for the previous rule. Finding whether u covers C can be done in $O(n + p)$ time. Finding the $O(\min(n, p))$ closest P-neighbours requires $O(p)$ time. Creating the auxiliary graph takes $O(n + p)$ time, and computing the matching $O(n^2p)$ time [13]. For a single L_2-leaf, the reduction rule requires $O(\min(n, p) \cdot n^2p)$ time.

An application of the rule. This reduction rule also needs to know the caterpillar component for each node. As the previous rule already computed this information, we do not have to calculate it here.

We apply the rule on every L_2-leaf in the instance. This results in a total running time of $O(\min(n, p) \cdot n^3p)$. This rule does not run differently in later applications, as it depends on k.

5.4.13 Reflection

In hindsight, we would have refrained from trying to implement the reduction rules smartly. The smarter implementations include getting rules to perform multiple modifications per application, restricting the part of the instance looked at during later applications, and restricting the number of checked combinations of instance elements like Reduction Rule 4.1.3 does. We fear that most rules that do not depend on the value of k have an unfair advantage over rules that do depend on k, as we could not improve those in any way.

6 Instance Generation

This research uses both instances generated from scratch with different methods and instances generated from instances for other problems. The following subsections discuss ways to generate trees, demand pairs, and complete instances from other instances. We explain which existing instances and which parameters we used during instance generation in Subsection 7.2.

6.1 Trees Generation Methods

The three types of generated trees are random trees, caterpillars, and trees where many nodes have a degree equal to three. Each tree used in this research is a labelled tree. This allowed us to use Prüfer sequences to generate random trees.

6.1.1 Prüfer Trees

We wanted to use trees that are as random as possible, to use as a baseline for our experiments. One way to create these trees is using a Prüfer sequence. These sequences were, for instance, used by Prüfer in a proof of Cayley's formula [49].

Theorem 5 (Cayley's Formula)**.**
For every positive integer n, the number of trees on n labelled nodes is n^{n-2}.

We refer from now on to the trees created using a Prüfer sequence as Prüfer trees. Each tree can be associated with a unique Prüfer sequence, and we can directly convert them to and from each other. A Prüfer sequence for a tree with n nodes contains $n - 2$ integers, each having n possible values. Thus, there are n^{n-2} unique Prüfer sequences, and each can be associated with one of the n^{n-2} labelled trees that are possible on n nodes. When generating a Prüfer sequence uniformly at random, each of these n^{n-2} trees has an equal probability of appearing.

[13]We can again improve the running time of the matching algorithm to $O(\sqrt{n}p)$ using the algorithm by Micali and Vazirani [38]. Since the auxiliary graph is bipartite, we can achieve the same faster running time using the algorithm by Hopcroft and Karp [26].

6.1.2 Caterpillars

Another type of tree used in our experiments is a caterpillar.

Different kernel sizes. Both Guo and Niedermeier [23] and Bousquet *et al.* [4] proved in their respective articles a kernel for Multicut on caterpillars. Only after discussing this kernel, they moved on to the kernel for Multicut on general trees. The bound on the kernel size on caterpillars differed in both papers from the bound on the kernel size on general trees. Guo and Niedermeier showed a kernel of size $O(k^{2k+1})$ for caterpillars, and one of size $O(k^{3k})$ for general trees. Bousquet *et al.* proved a kernel of size $O(k^5)$ for caterpillars and one of size $O(k^6)$ on general trees. This difference between these bounds makes caterpillars an exciting choice as a tree type.

Restriction on applicable reduction rules. Another argument for using caterpillars is that not all reduction rules can apply to caterpillars. For instance, Reduction Rule 4.1.7 (Overloaded Caterpillar) only works when there are multiple caterpillar components in an instance. This is not the case in a caterpillar. Another example is Reduction Rule 4.1.8 (Overloaded L_3-leaves). This rule requires the presence of L_3-leaves in the instance, but caterpillars do not have any.

Observation 6.
The number of I_3-nodes in a Multicut in Trees *instance cannot increase when applying one of the two studied kernelisation algorithms.*

Proof. The modifications that remove demand pairs or modify them do not change the type of any nodes. It suffices to handle edge contractions.

When contracting an edge between an internal node and a leaf, the number of internal nodes amongst the neighbours of the internal node does not increase. We only consider an edge contraction between two internal nodes.

Let us try to create an I_3-node w during an edge contraction between nodes u and v. After the edge contraction, we want w to have at least three neighbours that are internal nodes. Let d_u be the number of neighbours of u that are internal nodes, let d_v be the number of neighbours of v that are internal nodes, and let d_w be the number of neighbours of w that are internal nodes. We get $d_w = d_u + d_v - 2$. The minus two is because we counted u as an internal neighbour of v, and the other way around.

To get $d_w \geq 3$, we must have $d_u + d_v - 2 \geq 3$. This can be rewritten to $d_u + d_v \geq 5$. This can only be the case when either $d_u \geq 3$ or $d_v \geq 3$. So, at least one of u and v should have been an I_3-node.

In conclusion, we cannot create an I_3-node during an edge contraction where both the endpoints of the contracted edge are not I_3-nodes. The number of I_3-nodes in an instance thus cannot increase. □

There must be at least one I_3-node in an instance for L_3-leaves to exist. Caterpillars, by definition, have zero I_3-nodes, and according to Observation 6, we do not create any I_3-nodes during the execution of a kernelisation algorithm. To conclude, Reduction Rules 4.1.7 and 4.1.8 will never be applicable on caterpillars.

Creating caterpillars. To create a caterpillar, we randomly pick how many internal nodes we will use for the backbone. The remaining nodes will be leaves.

We do not want the caterpillars to consist mainly of internal nodes, as such a tree would greatly resemble a path. In a path, a Multicut in Trees instance would probably not be challenging to solve.

To ensure the number of internal nodes is not too large, we do not pick this number uniformly at random. Instead, we use a beta distribution with parameters $\alpha = 2$ and $\beta = 8$. Figure 4

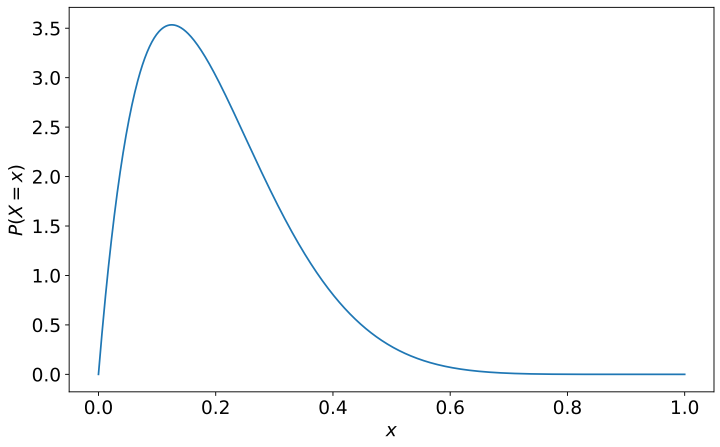


Figure 4: A beta distribution with $\alpha = 2$ and $\beta = 8$.

shows this distribution. We chose to use a beta distribution because it gives us plenty of influence over its shape and thus the caterpillar backbone length. It is perfect for mainly getting small backbone sizes while still sometimes encountering larger backbone sizes.

To create a caterpillar with n nodes, we pick a random number i from the beta distribution described above, scale i to the interval $[0, n-4]$, and round it to the nearest integer. We scale i to at most $n-4$ because we create four nodes manually. We create two I_1-nodes manually and give them each a leaf. We then create the path between the two I_1-nodes going through i I_2-nodes. Finally, we create $n-i-4$ leaves and connect them each to any of the internal nodes uniformly at random.

6.1.3 Degree3 Trees

The last generated type of instance is the Degree3 tree.

Restriction on applicable reduction rules. One of the reasons to use caterpillars was that we cannot apply all reduction rules on caterpillars. We used a similar argument as motivation for Degree3 trees. For instance, Reduction Rule 4.1.7 (Overloaded Caterpillar) only works when there are multiple caterpillar components. For these to exist, there must be multiple I_2-nodes, and they should not be all connected to each other. Some reduction rules, like Reduction Rule 4.2.6 (Bidimensional Dominating Wingspan) and Reduction Rule 4.2.7 (Generalised Dominating Wingspan) require an instance to have L_2-leaves. This means there need to be I_2-nodes in the instance. We designed the Degree3 trees to have as few I_2-nodes as possible.

Description of Degree3 trees. We can compare Degree3 trees best to complete binary trees. A binary tree has one obvious I_2-node: the root node. Degree3 trees differ from binary trees because the root note has three neighbours instead of two.

Figure 5 shows an example of a Degree3 tree with 17 nodes. In this example, we have five I_1-nodes: 4, 5, 6, 7 and 8, zero I_2-nodes, and three I_3-nodes: 1, 2 and 3. All leaves: 9, 10, 11, 12, 13, 14, 15, 16 and 17 are L_1-leaves. Note that Degree3 trees can still have an I_2-node. In our example, if we remove node 17, node 3 would become an I_2-node. However, there can not be more than one I_2-node.

Creating a Degree3 tree. We now show how to create a Degree3 tree with n nodes. We use a `Queue` [14] [45] to make sure nodes get neighbours in an order such that the levels of the tree

[14]`https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.queue-1?view=net-5.0`

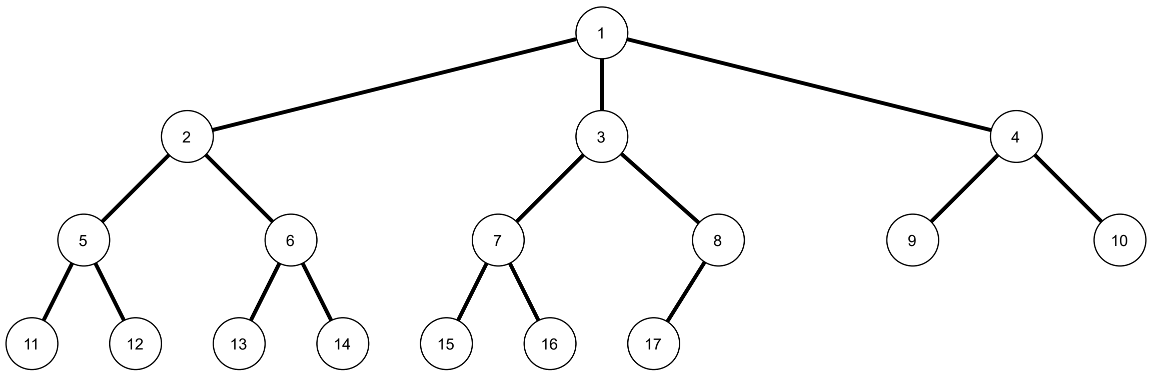


Figure 5: A Degree3 tree with 17 nodes.

get filled from left to right. We start by creating the root node and add it to the `Queue`. We also manually create the first neighbour of the root node and add it to the `Queue`. Then, we repeat the following process until we have reached the required number of n nodes. Grab a node v from the `Queue`. Create a neighbour for v and add it to the `Queue`. If the current number of nodes in the tree is smaller than n, create another neighbour for v and add it to the `Queue`.

Note that this generation method, in contrast to the other two, does not include any randomness.

6.2 Demand Pairs Generation Methods

We also have three possibilities to generate demand pairs: uniformly at random, using a length distribution and generation through a solution.

6.2.1 Uniformly at Random Demand Pairs

The most obvious way to generate a demand pair given a tree is by picking two distinct nodes in the tree uniformly at random and mark them as the endpoints of this demand pair. That is exactly how this method works.

Creating demand pairs uniformly at random. Consider we want to create demand pairs in a tree with n nodes. For each demand pair, we pick a number $i \in \mathbb{Z}$ in the interval $[1, n]$ uniformly at random. Then, we pick another number $j \in \mathbb{Z}$ in the interval $[1, n-1]$ uniformly at random. If $j \geq i$, we increment j. After this step, j lies in the interval $[1, i-1] \cup [i+1, n]$. We then create a demand pair between nodes i and j and repeat this process until we have created enough demand pairs.

6.2.2 Demand Pairs from a Length Distribution

The second way to generate demand pairs is using a distribution on the preferred demand path length. We created this method because we were interested in how the different algorithms work on instances with mostly short or mostly long demand paths. A reduction rule like Reduction Rule 4.1.6 works best when there are many demand paths of length two. On the other hand, rules like Reduction Rules 4.2.6 and 4.2.7 can benefit from longer paths as the wingspan of a node can then be larger.

Explanation of the method. The demand path range distribution consists of tuples of a range of numbers and the chance for that range to occur. To generate a demand pair, pick one of these ranges from the distribution according to their respective probabilities. The result from this distribution is a range for the length of the current demand path. Say we want the length of this path to be in the interval $[s, t]$. The method then picks a pair of nodes, such that the path between those two nodes is in this interval.

Implementation of the method. The method needs to know the path between every pair of two nodes. This knowledge allows it to immediately pick all possible node pairs, given a certain path length. Then, for each demand pair, it picks a path length range. It grabs all possible node pairs in this range. If there are not enough demand pairs (by default: ≤ 4), it expands the search range by one on both sides until it has found enough possible node pairs. Then, it grabs one of these pairs uniformly at random and denotes these nodes as the endpoints of the current demand pair.

6.2.3 Demand Pairs Through a Solution

In contrast to the two methods described above, this method gives us direct influence on the value of parameter k. It works by creating demand paths through a solution of size k. We determine the solution by picking k edges in the tree uniformly at random. Then, we create k edge-disjoint demand paths, each going through precisely one of these edges. Finally, we create the remaining $p - k$ demand pairs.

Creating the first k edge-disjoint demand paths. We create edge-disjoint demand paths rather trivially. Consider an edge e in the solution between nodes u and v. Let P' be the set of demand paths that we have already generated. We say that all edges on the paths in P' are forbidden edges. The edges in the solution are forbidden as well. Create a set U with nodes, such that, for every node $u' \in U$, the path between u and u' does not go over any of the forbidden edges. Similarly, create a set V with nodes reachable from v without going over forbidden edges. Pick a node u' from U and a node v' from V uniformly at random. Then, create a demand pair between u' and v'. Note that the corresponding demand path goes over e, no other solution edge, and does not intersect any other demand path.

An example of creating k edge-disjoint demand paths. We give an example of creating three demand pairs through a solution of size two in Figure 6. The edges in the solution are coloured red in Subfigure 6a.

The first demand path we want to create goes over the edge between nodes 3 and 5. Let u be node 3, and v node 5. We find $U = \{1, 2, 3, 4\}$ and $V = \{5, 6, 7\}$. Assume we pick node $u' = 1$ and $v' = 6$. The first demand pair goes from node 1 to node 6; see Subfigure 6b.

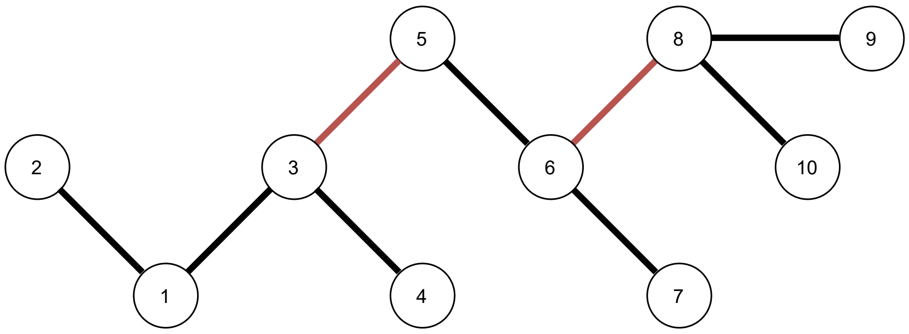
For the second demand pair, say u is node 6, and v is node 8. We then find $U = \{6, 7\}$ and $V = \{8, 9, 10\}$. Assume we pick nodes 7 and 8 as endpoints for the second demand pair; see Subfigure 6c. We have now generated two edge-disjoint demand paths through a solution.

We first explain how to generate the remaining demand paths before we finish this example.

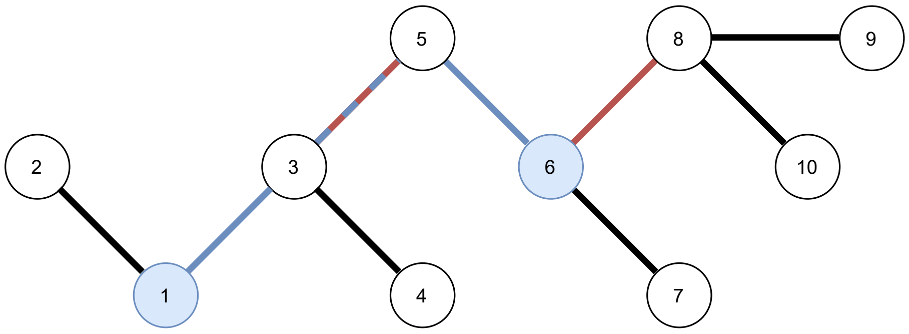
Creating the remaining demand paths. Now we show how to create the remaining $p - k$ demand paths. The only requirement is that these go through at least one of the edges in the solution.

For each edge e in the solution between nodes u and v, we create a set U again, with all nodes that we can reach from u without going through e. Similarly, we create a set V with nodes reachable from v.

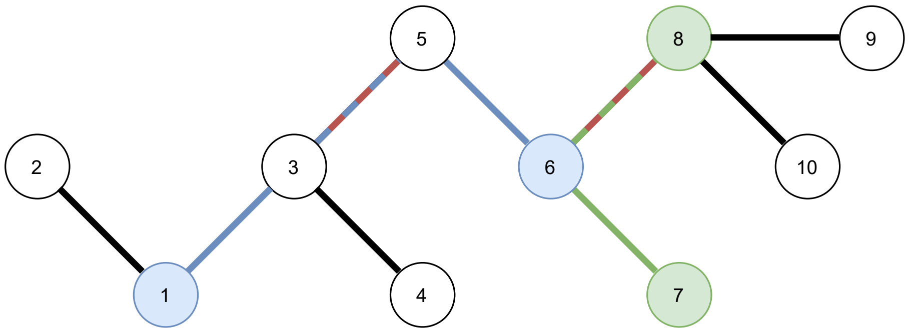
Then, for each demand pair, pick an edge in the solution uniformly at random. Pick a node u' from the corresponding set U and a node v' from the corresponding set V, uniformly at random. Finally, create a demand pair between u' and v', and repeat this process for the remaining demand pairs.



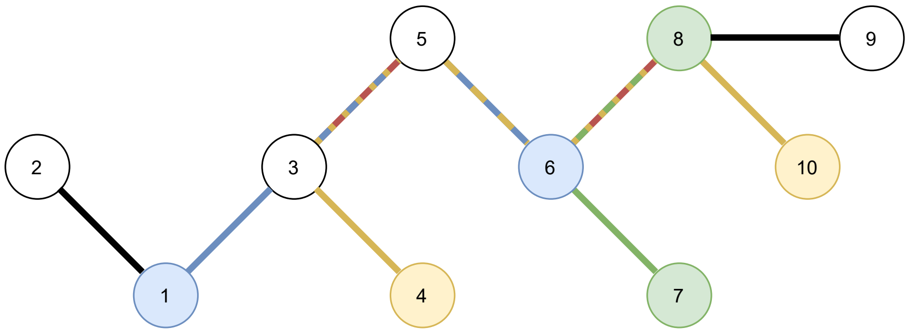
(a) A tree for the example. The edges in the solution are coloured red.



(b) We have generated the demand pair for the left solution edge and coloured its endpoints and path in blue.



(c) We have generated demand pair for the right solution edge and coloured its endpoints and path in green.



(d) We have generated the demand paths through the solution. We randomly selected the left solution edge for the current demand path to pass through, generated it, and coloured its endpoints and path in yellow.

Figure 6: An example of creating demand pairs through a solution of size 2.

Example of creating the remaining $p - k$ demand paths. We continue on the example above. We have already generated two of the three demand pairs, which Subfigure 6c shows. We need to create one additional demand pair. Assume we pick the left solution edge as the edge the final demand path must go through. Let u be node 3, and v node 5. We find $U = \{1, 2, 3, 4\}$ and $V = \{5, 6, 7, 8, 9, 10\}$. Assume we pick node 4 from U and node 10 from V. We create a demand path through these two nodes; see Subfigure 6d. This concludes the generation of the three demand pairs through a solution of size two.

A limitation of this method. The way we generate edge-disjoint demand paths in this method has a significant limitation. A generated demand path can restrict the other demand paths that still must be generated by a lot.

Take a look at Figure 7. We have a solution of size four, displayed in red. We already generated the first demand path and gave it a blue colour. The three other solution edges are between nodes 6 and 9, between nodes 7 and 9, and between nodes 8 and 9. The demand paths we generate through these edges will only consist of the solution edges themselves. They cannot go anywhere else as the first demand path blocks node 9's incident edges. These demand paths will never use the whole part of the graph to the right of the blue demand path.

An alternative would be to allow overlap between the demand paths. We tried this method, but it resulted in an optimal solution that was, in some cases, way smaller than we expected. Sometimes, the size of the solution was less than half of the expected size. Retrying the generation method until we found a solution almost as large as the expected solution was also unsuccessful. Sometimes, we needed over 10,000 trials to find an acceptable solution size. We opted to use the generation method described above, and we hope that its limitation does not result in weird demand paths very often.

6.3 Instance Generation from Instances of Other Problems

Besides generating Multicut in Trees instances from scratch, we also used two ways to generate them from instances for other problems: Vertex Cover and CNF-SAT. Both these ways generate the tree and demand pairs, so we do not combine them with the generation methods described above.

6.3.1 Instances from Vertex Cover Instances

The way to generate Multicut in Trees instances from Vertex Cover instances is based on the $\mathcal{NP}$-hardness proof for Multicut in Trees by Garg *et al.* [21]. In Section 3 of their article, they showed the equivalence of Vertex Cover on general graphs and Multicut on trees of height one and unit capacities.

Constructing the instance. Let $G = (V, E)$ be a general graph. Construct a star graph H with an internal node u_0 and a leaf u_i for each $v_i \in V$. For each edge $e \in E$, create a demand pair in H. Assume e has v_i and v_j as endpoints. We create a demand pair in H between u_i and u_j. This concludes the creation of the Multicut in Trees instance. The optimal k for this Multicut in Trees instance is equal to the size of the minimum Vertex Cover in G.

6.3.2 Instances from CNF-SAT Instances

Generating a Multicut in Trees instance from a CNF-SAT instance is slightly more complicated than the previous generation method. This method is based on another $\mathcal{NP}$-hardness proof for Multicut in Trees, this time by Călinescu *et al.* [8]. In Theorem 11 of their article, they proved Multicut on binary trees is $\mathcal{NP}$-hard using a reduction from 3-SAT. We have taken a more general approach by creating the instances from CNF-SAT instances, where

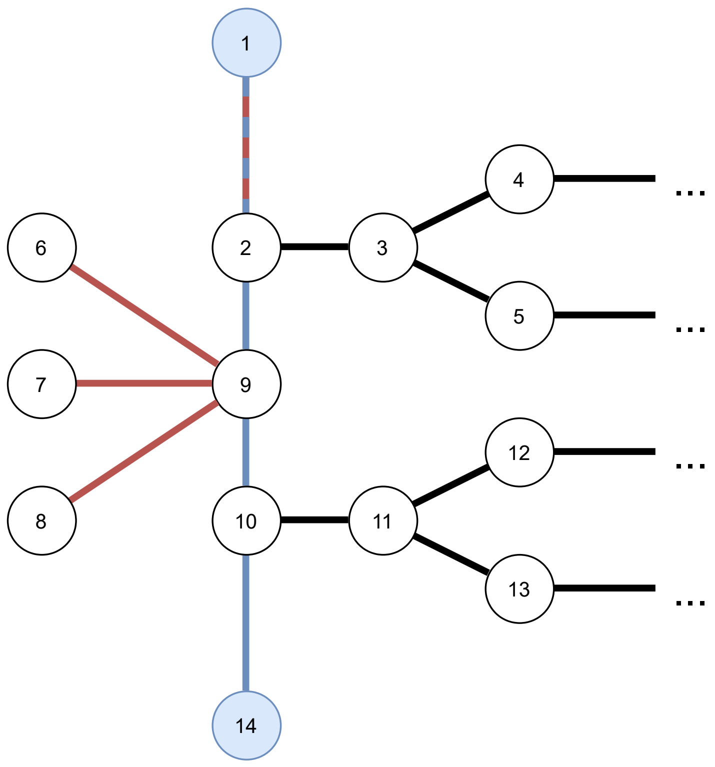


Figure 7: An example of the limiting factor of generating the edge-disjoint demand paths used in the demand pair generation method that generated demand pairs through a solution. A part of the tree is visible here. The tree continues where we placed triple dots. The solution has three edges, coloured red: between 1 and 2, between 6 and 9, between 7 and 9, and between 8 and 9. We generated the first demand path through the edge between 1 and 2. Assume it goes from 1 to 14. The figure displays this demand path in blue. The three demand paths that we need to generate through the other three solution edges will only have one edge, as they cannot go anywhere from node 9.

clauses can all have any size. The generated tree is still a binary tree. We first explain how we create a Multicut in Trees instance from a 3-SAT instance. After that, we explain how the generation method for CNF-SAT differs.

Creating the instance from 3-SAT. Given a set of n variables $x_1, \dots, x_n$ and m clauses $c_1, \dots, c_m$. We create a structure for each variable, a different structure for each clause, and connect them to form a tree. We took the following examples from the article by Călinescu *et al.* [8].

The variable structure. Subfigure 8a shows the structure for a variable x_i. We have a node representing x_i and one representing $\neg x_i$. We connect them to a parent node and create a demand pair between x_i and $\neg x_i$.

The clause structure. For each clause, we create a structure like the one in Subfigure 8b. The clause shown there is the clause $c_i = (x_j, \neg x_k, \neg x_l)$. There is a demand pair between x_j and $\neg x_k$, and one between $\neg x_l$ and the neighbour of x_j and $\neg x_k$.

Creating the tree. We create the total tree by connecting all clause structures and all variable structures. We connect all variable structures to a backbone. Likewise, we connect all clause structures to a different backbone. Finally, we connect these two backbones through a root node. They are all connected in such a way that the result is a binary tree.

Adding more demand pairs. Finally, we add demand pairs between the nodes in the variable structures and the clause structures. Let $I = c_1 \wedge c_2$, where $c_1 = (x_1, \neg x_2, x_3)$ and $c_2 = (\neg x_1, x_2, x_3)$ be a 3-SAT instance. We create a demand pair between node x_1 in the variable structure of x_1, and node x_1 in the clause structure of c_1. We then create a demand pair between node $\neg x_2$ in the variable structure of x_2, and node $\neg x_2$ in the clause structure of c_1. Similarly, we create a demand pair for x_3 in c_1, and for all variables in c_2. Subfigure 8c shows the Multicut in Trees instance generated from the 3-SAT instance I.

From the proof by Călinescu *et al.*, we know that the optimal solution size for this Multicut in Trees instance is $n + 2m$, where n is the number of variables, and m the number of clauses.

Creating the instance from CNF-SAT. The generation of a Multicut in Trees instance from a CNF-SAT instance follows roughly the same lines. The only difference is in the generation of the clause structures.

Since clauses can be smaller or bigger than three, the structures for the clauses themselves also need to be smaller or bigger. The generation method for these structures is similar to the method for structures for clauses of length three. Figure 9 contains an example of a structure for a clause of length five.

As the structures can be smaller or bigger, the number of demand pairs in each structure can also be smaller or bigger. The proof by Călinescu *et al.* can easily be adapted to work on these differently sized structures if we add a demand pair between every two nodes on the same level of the structure. For the structure of a clause c_i, with length l_i, we add $l_i - 1$ demand pairs.

This also influences the size of the optimal solution for the Multicut in Trees instance. Remember that for instances created from 3-SAT instances, the optimal solution for the Multicut instance is $n + 2m$, where n is the number of variables, and m the number of clauses. Let l_i be the length of clause c_i, for $i = 1, \ldots, m$. Note that $n + 2m = n + \sum_{i=1}^{m}(l_i - 1)$ for a 3-SAT instance. We can use the right part of this equation to compute the optimal solution size for a Multicut in Trees instance from a CNF-SAT instance.

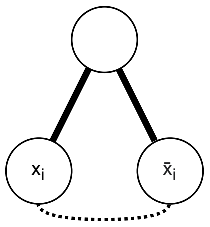
Adding randomness. We added one more step during the conversion process. When creating a clause structure, we do not necessarily attach the nodes corresponding to the variables in a clause to the backbone of the clause in the order they are present in the clause itself. Instead, we shuffle the order of the nodes and attach them to the backbone in this random order. We do the same when we attach variable structures and clause structures to their respective backbones. The shuffling does not influence the working of any algorithm, and the proof by Călinescu *et al.* still holds, but it still gave us slightly more variation in the instance trees.

7 Experimental Setup

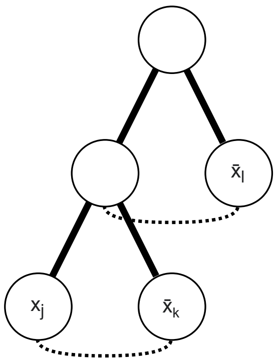
To measure the performance of the kernelisation algorithms, we consider both the size of the resulting kernel and the time spent to compute that kernel. The following subsection describes how we measured the time spent by each algorithm and presents the exact instances we used.

7.1 Counting Operations

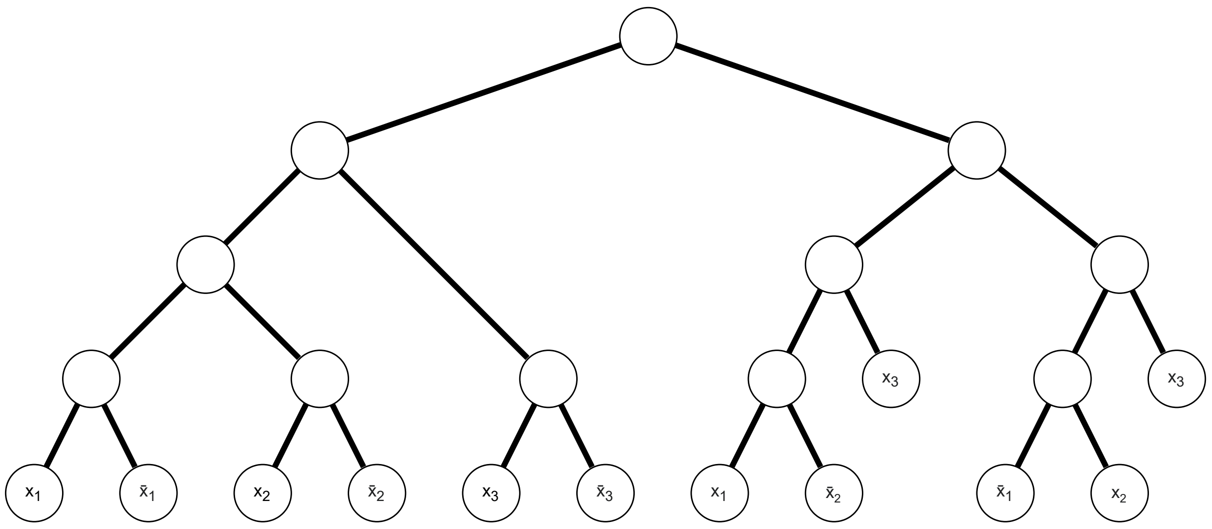
One of the obvious performance measurements for an algorithm is its running time. In theoretical research, we can analyse the running time using the Big O notation. For more practical studies, like this one, that does not always suffice.



(a) The structure for a variable x_i. The bar on variables in a node represents the logical not ($\neg$). The dotted line represents a demand pair between x_i and $\neg x_i$.



(b) The structure for a clause $c_i = (x_j, \neg x_k, \neg x_l)$. The bar on variables in a node represents the logical not ($\neg$). The dotted lines represent demand pairs.



(c) An example of the binary tree for MULTICUT IN TREES generated for the 3-SAT instance $c_1 \wedge c_2$, where $c_1 = (x_1, \neg x_2, x_3)$ and $c_2 = (\neg x_1, x_2, x_3)$. The bar on variables in a node represents the logical not ($\neg$). We do not explicitly show the demand pairs in this example. The ones in the structures for each variable and each clause should be obvious. There are seven such demand pairs. Then, there are six more demand pairs: between x_1 in the variable structure of x_1 and x_1 in the structure of the first clause, between $\neg x_2$ in the variable structure of x_2 and $\neg x_2$ in the structure of the first clause, between x_3 in the variable structure of x_3 and x_3 in the structure of the first clause, and likewise three for the second clause.

Figure 8: Structures for variables and clauses for the generation of a MULTICUT IN TREES instance from a 3-SAT instance, and an example of the result of such a conversion.

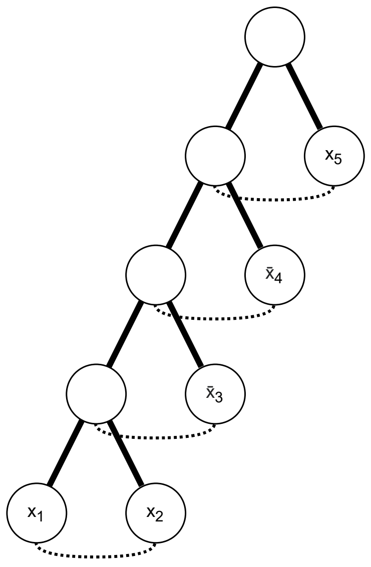


Figure 9: The structure for a clause $c_i = (x_1, x_2, \neg x_3, \neg x_4, x_5)$, as used during the generation of a Multicut in Trees instance from a CNF-SAT instance. The bar on variables in a node represents the logical not ($\neg$). The dotted lines represent demand pairs.

The stopwatch approach. An easy alternative is using a stopwatch to measure the time spent by an algorithm. This method has its limitations, like external factors. These include, for instance, background tasks in the operating system and the hardware the algorithm runs on.

System-independent metrics. Angriman *et al.* [2] wrote a paper about experimental research in network analysis. In Section 4.6, they wrote about performance metrics. A category of these is system-independent metrics. These metrics should give the same result, regardless of the system they run on. Other external factors should also not influence the measurements.

Examples of system-independent metrics. McGeoch [37] gave an excellent example of a system-independent metric. They dedicated Chapter 3 of their book to what one should measure during practical algorithm analysis. More specifically, Section 3.1.1 is about adding counters to the source code to measure dominant operations. For instance, when measuring the performance of a sorting algorithm, the number of comparisons it makes can be a good measure. Sadly, for the kernelisation algorithms studied in this research, no such single dominant operation exists.

We still use counters in this research to measure the time spent by each algorithm as an alternative to the stopwatch approach. These counters count the number of operations performed on custom data structures. We explain these data structures in the following subsection.

Besides being reproducible on other machines, another advantage of using counters is that the measured performance is entirely deterministic. We do not need to do multiple repetitions of the same experiment.

Hardware. Just for the sake of completeness: all experiments were conducted on the same machine running Windows 10 with an Intel Core i7-7700K running at 4.7GHz and 32GB of RAM. We implemented each algorithm without parallelism.

7.1.1 Data Structures

We created wrappers around existing data structures to make counting the number of operations more straightforward. Additionally, we created an object called a `Counter` to keep track of how many operations the algorithm performed.

The `CountedList`. For instance, we implemented a `CountedList`. This data structure uses a `List` to store elements, and it exposes most methods that a `List` itself also exposes. For example, an element can be added to a `CountedList` using the `Add` method. However, this method requires an extra parameter: a `Counter` object. The `CountedList` updates the `Counter` and adds the new element to its internal `List`. A table with how many operations are counted for each method of the `CountedList` can be found in Table 4 in Appendix B.

We also used a counted version of a `Dictionary` and the `OrderedDictionary` explained in Subsection 5.1. These are called the `CountedDictionary` and the `CountedCollection` respectively. Tables with the number of operations that get counted for the methods of these data structures are also present in Appendix B.

Almost every data structure we used in our implementation is one of these `Counted` data structures.

Enumerating over the custom data structures. Enumerating over the elements in the `Counted` data structures is possible using a `foreach`-loop. We can ask the data structures for a `CountedEnumerable` to enumerate over. This enumerable automatically updates the counter by one for each accessed element.

7.1.2 Correctness

We are reasonably confident the counters work in most cases. One exception we know of is when a reduction rule computes many matchings. In that case, we do not count enough operations.

Consider, for instance, Figure 10, where we tested the counters on instances with caterpillars with 384 nodes and demand pairs through a solution of size 50, with the algorithm by Guo and Niedermeier [23]. In Subfigure 10a, we see the number of operations, and in Subfigure 10b the number of ticks. One tick is 100 nanoseconds. There is no one-to-one correspondence between the counters and the ticks, but the shapes of both graphs are very similar.

The results on the same instances, but using the algorithm by Bousquet *et al.* [4] are present in Figure 11. Here, the shape is still similar, but it becomes clear that reduction rules with matchings do not count enough operations. Even so, we think the operation counters are good enough to use in our experiments.

Finally, we show one result that is frankly a mess. On the instances generated from the Vertex Cover instances, especially when there are 385 nodes, there are significant differences between the measured number of operations and ticks. The differences can be seen in Figure 39 in Appendix C. We strongly believe that the computation of matchings causes the difference between the ticks and operations, so we do not worry too much about the differences.

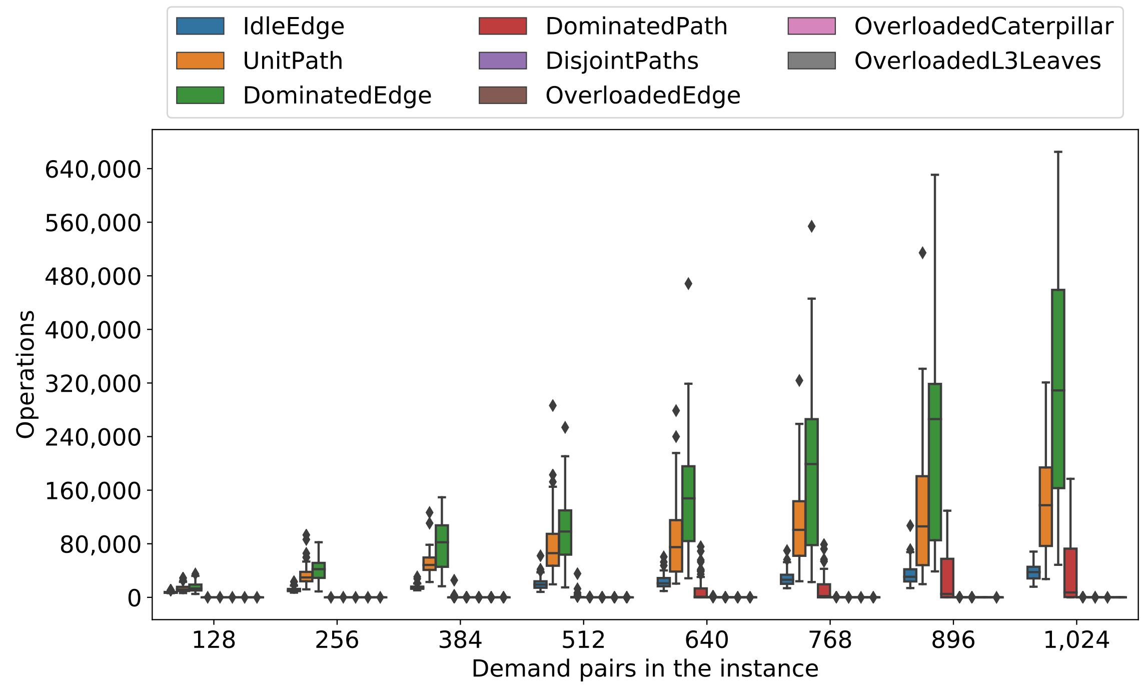
7.2 Used Instances

We explained how we generate instances in Section 6. This subsection discusses the parameters we used for each of these instances and which existing instances we used.

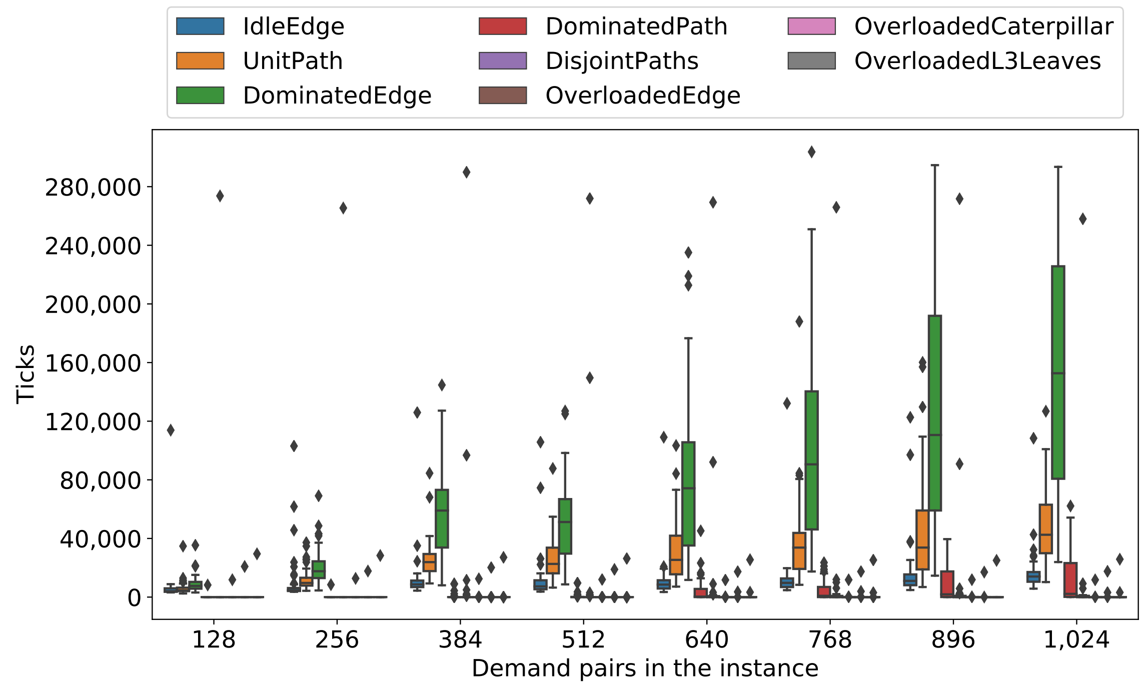
7.2.1 Instances from Scratch

The instances we generated ourselves have a number of nodes and demand pairs ranging between 128 and 1,024 each, in steps of 128. So, we have 64 different combinations of the number of nodes and the number of demand pairs. We also used every combination of the three tree generation methods and the three demand pair generation methods.

For demand pairs through a solution, we used five different values for k: 5, 10, 20, 50 and 100. How we used demand pairs generated using the length distribution requires a bit more explanation.

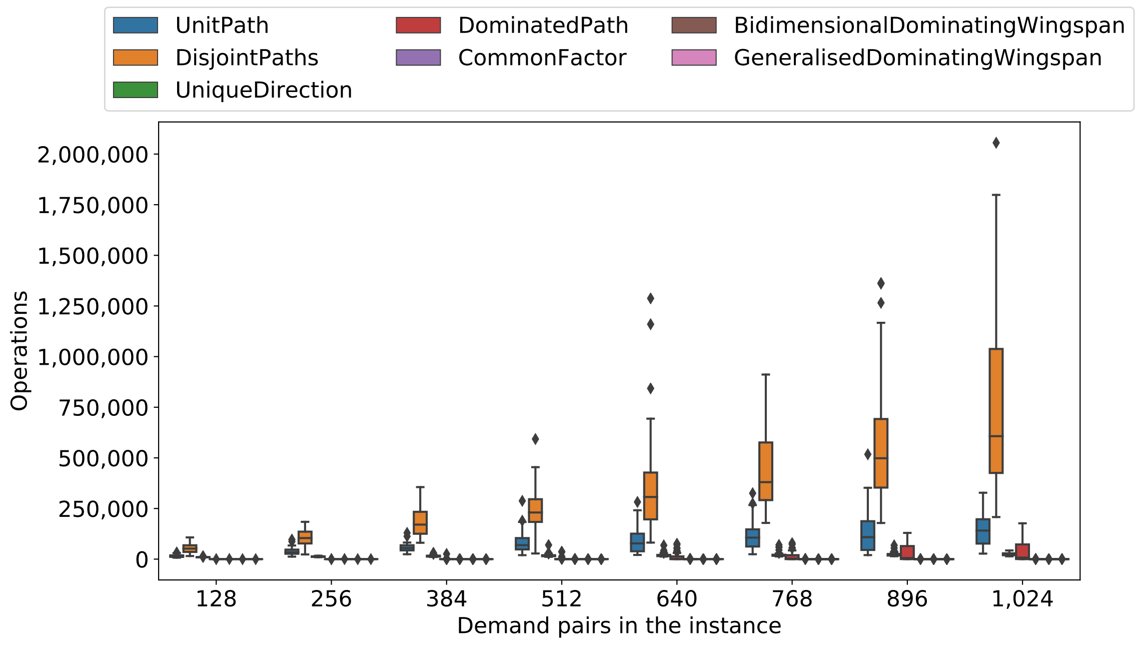


(a) The number of operations.

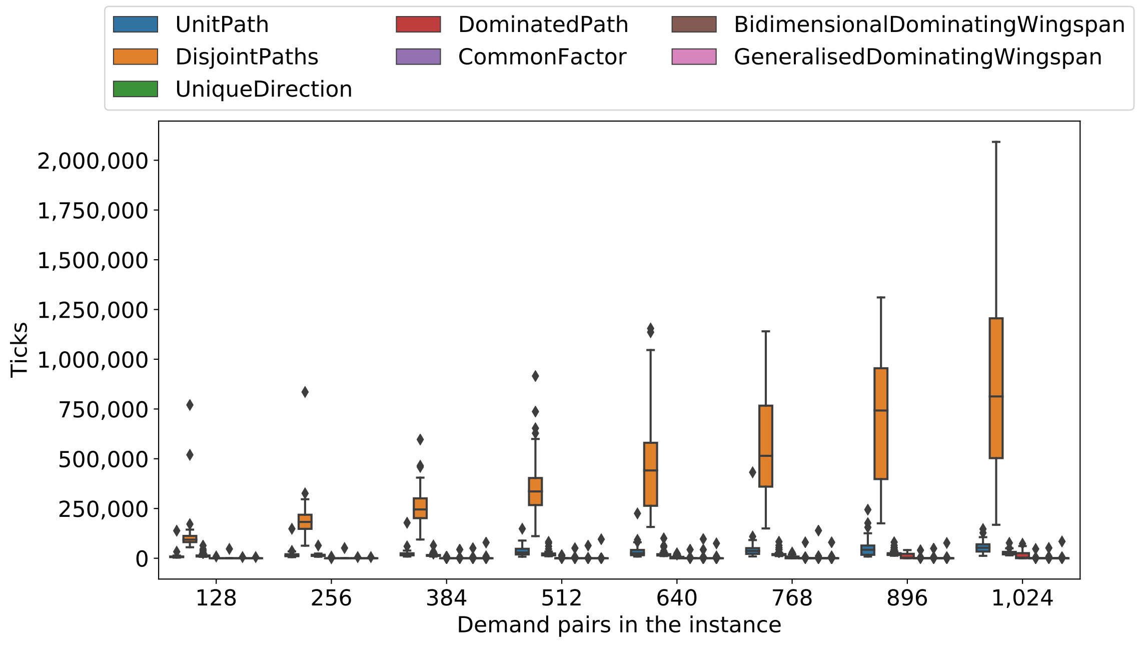


(b) The number of ticks.

Figure 10: Box plots of the number of operations and ticks per reduction rule in the algorithm by Guo and Niedermeier [23] to compute kernels on caterpillars with 384 nodes and demand pairs generated through a solution of size 50. For each number of demand pairs, there were 50 experiments.



(a) The measured number of operations.



(b) The measured number of ticks.

Figure 11: Box plots of the number of operations and ticks per reduction rule in the algorithm by Bousquet *et al.* [4] to compute kernels on caterpillars with 384 nodes and demand pairs generated through a solution of size 50. For each number of demand pairs, there were 50 experiments.

Demand pairs using the length distribution. We created instances with mostly short demand paths and ones with mostly long demand paths. Short demand paths are paths with a length in the interval $[2, s]$. The length of long demand paths lies in the interval $[l, m]$, where m is the longest path possible in the instance. The values of s, l and m differ per instance type and size.

Approach. To determine s, l and m per instance type and size, we first computed the trees for all instances. In these trees, we computed the length of the path for every pair of two nodes. We used this data to create a histogram per tree type and size. We then used these histograms to determine s, l and m. We tried to create a logical formula for each type of tree to compute s and l. The value of m for each tree type and size equals the highest value in the histogram.

Prüfer trees. We use at the expected diameter of Prüfer trees. Since these are random trees, their expected diameter is $3.342171\sqrt{n}$, where n is the number of nodes in the tree. Szekeres [52] gave this value in a paper from 1983. For l, we settled on approximately three-quarters of this value. For s, we used slightly over 10% of this value. Ultimately, for Prüfer trees, we use $s = 0.35\sqrt{n}$ and $l = 2.5\sqrt{n}$, rounded to the nearest integer.

Caterpillars. We generate the size of the backbone of our caterpillars using a beta distribution. Since we only add leaves to this backbone, we use the distribution to develop sensible values for s and l. For this, we use the inverse regularised beta function $I_x(\alpha, \beta)$. We wanted the short paths to be at most approximately 5% of the backbone length and long paths to be bounded by about 75% of the backbone length. We computed x_s in $I_{x_s}(2, 8) = 0.05$ and x_l in $I_{x_l}(2, 8) = 0.75$. This resulted in $x_s \approx 0.0410232$ and $x_l \approx 0.2722695$. For caterpillars, we used $s = x_s n$ and $l = x_l n$, rounded to the nearest integer.

Degree3 trees. Since we generate Degree3 trees without randomness, it should be easy to determine the formulas for them. We can view a Degree3 tree as three binary trees attached by their respective roots to a general root node. We say that the expected number in each of these three binary subtrees is $\frac{n-1}{3}$. Because we do not know how many nodes the lowest level in each subtree contains, we compensate for this by dividing the number of nodes in this subtree by two. This division ensures that we do not consider the nodes in the lowest layer. We can use the base-2 logarithm to compute the height of this subtree. The expected height of the tree is one longer: the edge between the root of the binary subtree and the general root of the tree. A long path can be twice as long as this number by existing between two leaves in two different binary subtrees. This results in the following calculation for a long path:

$$\begin{aligned} 2(\log_2\left(\frac{\left(\frac{n-1}{3}\right)}{2}\right) + 1) &= 2\log_2\left(\frac{n-1}{3}\right) - 2\log_2(2) + 2 \\ &= 2\log_2\left(\frac{n-1}{3}\right). \end{aligned}$$

For s, we used 35% of this value, and for l 85%. We gathered the percentages using the histograms. So, for Degree3 trees, we use $s = 0.7\log_2\left(\frac{n-1}{3}\right)$ and $l = 1.7\log_2\left(\frac{n-1}{3}\right)$.

Finishing the instances. Once we computed s and l for each tree type and size, we recomputed the instances. In these instances, the trees are the same as those used to compute the path length histograms.

For half of the instances, we preferred short demand paths. We told the algorithm to use the distribution for short paths for 90% of the demand pairs and the distribution for long paths for

10% of the demand pairs. For the other half of the instances, we used the distribution for short paths for 10% of the demand pairs and the distribution for long paths for 90% of the demand pairs.

Randomness and seeding. We seeded all experiments for repeatability. The tree and demand pair generation methods used their own random number generator with a different seed. No two seeds used should be the same.

We generated the initial random seeds using the integer set generator from random.org [15] [50]. We generated a single set with 10,000 unique random integers in the range of 1 to 10,000,000.

Multiple experiments per instance type and size. As mentioned before, we did not run multiple repetitions of the same experiment. However, we did use multiple instances with the same generation methods and sizes. For instance, for Prüfer trees with 256 nodes and 896 demand pairs generated through a solution of size 20, we performed 50 experiments. Let the initial seed for tree generation be t, the initial seed for demand pair generation d, and let E be the number of experiments to run. The experiments then used $t + i$ and $d + i$ as seed, for $0 \leq i < E$. We give an overview of how many experiments we ran for each combination of tree generation method and demand pair generation method in Table 1.

7.2.2 Instances from Other Problems

The other types of instances we used are the ones generated from instances for Vertex Cover, 3-SAT and CNF-SAT.

Vertex Cover. We used the simple $G(n, m)$ Erdős-Rényi model [17] to create our own graphs to use as Vertex Cover instances. This model creates a general graph with n nodes and m edges. For these graphs, we used nodes and edges in the range between 128 and 1,024 in steps of 128, just like the generated instances explained in the previous subsection. For each combination of number of nodes and edges, we created 100 graphs that were each converted to a Multicut in Trees instance. We also used seeds from the set of random integers explained above to generate these graphs.

3-SAT and CNF-SAT. For the 3-SAT and CNF-SAT instances we only used existing instances. All used instances are from the SATLIB library [16], by Hoos and Stützle [25]. In

[15]`https://www.random.org/integer-sets/`

[16]The original website as used in the article is not used anymore. The library can still be found at `https://www.cs.ubc.ca/~hoos/SATLIB/index-ubc.html`

Tree generation method / Demand pair generation method	**Prüfer trees**	**Caterpillars**	**Degree3 trees**
Uniformly at random	100	100	100
Length distribution	100	100	100
Through a solution	50	50	50

Table 1: The number of experiments run for each combination of tree generation method and demand pair generation method.

particular, we used the uniform random 3-SAT instances [17]. For CNF-SAT we used instances created by Hooker [18] that were part of the DIMACS benchmark suite for SAT problems [19] [30].

We divided the instances into categories based on their number of nodes. We present these categories for the 3-SAT instances in Table 2 and for the CNF-SAT instances in Table 3.

7.2.3 Determining the Optimal Solution

We do not know the size of the optimal solution for every used instance. Since some reduction rules use the value for k, we thought it fair to let k equal the optimal solution.

If we provided the reduction rules with a value for k that is smaller than the optimal value, all algorithms would probably stop because of Reduction Rule 4.1.5 (Disjoint Paths) before reaching the smallest kernel. On the other hand, if the value of k is too big, some reduction rules might not decide to perform a modification that they would have performed with a smaller value of k.

Finding the optimal solution. To find the optimal solution, we used the Gurobi MIP solver [20] [24]. Let I be a Multicut in Trees instance, consisting of a tree $T = (V, E)$ and a set P of demand paths. We create a variable e_i for each edge in E. This variable e_i will be 1 if we include its corresponding edge in the solution for I and 0 if we do not include it. We also create a linear expression P_j for each demand path in P. Each P_j is the sum of all variables e_i that correspond to the edges on the demand path that P_j represents.

[17]`https://www.cs.ubc.ca/~hoos/SATLIB/Benchmarks/SAT/RND3SAT/descr.html`

[18]`https://www.cs.ubc.ca/~hoos/SATLIB/Benchmarks/SAT/DIMACS/JNH/descr.html`

[19]The original site is no longer available. An archive can be found at: `http://archive.dimacs.rutgers.edu/pub/challenge/satisfiability/benchmarks/cnf/`

[20]`https://www.gurobi.com/`

Category	Number of nodes	Number of instances	Original instance name(s)
(u)uf50-218	1,507	1,000 + 1,000	uf50-0x.cnf, for $x = 1, \dots, 1,000$ uuf50-0y.cnf, for $y = 1, \dots, 1,000$
(u)uf75-325	2,249	100 + 100	uf75-0x.cnf, for $x = 1, \dots, 100$ uuf75-0y.cnf, for $y = 1, \dots, 100$
(u)uf100-430	2,979	1,000 + 1,000	uf100-0x.cnf, for $x = 1, \dots, 1,000$ uuf100-0y.cnf, for $y = 1, \dots, 1,000$
(u)uf125-538	3,727	100 + 100	uf125-0x.cnf, for $x = 1, \dots, 100$ uuf125-0y.cnf, for $y = 1, \dots, 100$
(u)uf150-645	4,469	100 + 100	uf150-0x.cnf, for $x = 1, \dots, 100$ uuf150-0y.cnf, for $y = 1, \dots, 100$
(u)uf175-753	5,217	100 + 100	uf175-0x.cnf, for $x = 1, \dots, 100$ uuf175-0y.cnf, for $y = 1, \dots, 100$
(u)uf200-860	5,959	100 + 99	uf200-0x.cnf, for $x = 1, \dots, 100$ uuf200-0y.cnf, for $y = 1, \dots, 99$
(u)uf225-960	6,659	100 + 100	uf225-0x.cnf, for $x = 1, \dots, 100$ uuf225-0y.cnf, for $y = 1, \dots, 100$
(u)uf250-1,065	7,389	100 + 100	uf250-0x.cnf, for $x = 1, \dots, 100$ uuf250-0y.cnf, for $y = 1, \dots, 100$

Table 2: Categories used for the 3-SAT instances, their size, the number of instances in each category, and the original instance names.

Category	**Number of nodes**	**Number of instances**	**Original instance name(s)**
8,175-8,323 nodes	8,175-8,323	19	jnh20x.cnf, for $x = 2, \ldots, 9$ jnh2y.cnf, for $y = 10, \ldots, 20$
8,651-8,783 nodes	8,651-8,783	20	jnh$x$.cnf, for $x = 2, \ldots, 20$ jnh201.cnf
9,129-9,707 nodes	9,129-9,707	11	jnh1.cnf jnh30x.cnf, for $x = 1, \ldots, 9$ jnh310.cnf

Table 3: Categories used for the CNF-SAT instances, their size, the number of instances in each category, and the original instance names.

The problem then becomes

$$\text{minimise} \sum_{e_i \in E} e_i$$

s. t.

$$\sum_{e \in P_j} e \geq 1 \qquad \forall\, P_j \in P$$

$$e_i \in \{0, 1\} \qquad \forall\, e_i \in E.$$

We used the resulting objective value directly as k.

8 Results

This section presents the main results obtained in our experiments and answers the questions we posed in Subsection 1.2. Appendix C contains additional figures for some results. We refer to the corresponding figures if this is the case.

Throughout this section, we refer to the algorithm by Guo and Niedermeier [23] as the `GN-algorithm`. We refer to the algorithm by Bousquet *et al.* [4] as the `BQ-algorithm`.

We also abbreviate the description of the instances. To talk about a set of instances, we use `Prüfer`, `Caterpillar` and `Degree3` to indicate the tree type in these instances. After a dash symbol (`-`), we use `Random` to indicate uniformly at random demand pairs, `Short` for mostly short demand paths, `Long` for mostly long demand paths, `TS` for demand pairs generated through a solution of undetermined size, and `TS`x for `TS` instances with a solution of size x. For instance, we abbreviate instances with a caterpillar and demand pairs generated through a solution of size 10 as `Caterpillar-TS`10. We can also refer to all instances with a Prüfer tree simply by `Prüfer` and similarly for the other tree types and demand pair generation types.

We start by explaining the figures we use to visualise the results. Then, we show the required results to answer each research question and give a final answer for each question.

8.1 Explanation of Figures

We explain what the figures we use to visualise our results represent and how we generated them. We used two types of figures to show the size of a kernel. One is a box plot, and the other a contour plot. We also used a contour plot to show the total running time of an algorithm over a set of instances. To show the dominant reduction rule, we used a modified heat map. We can also show a box plot for the dominant reduction rules.

8.1.1 Box Plot for Kernel Size

The first figure we use to show the kernel size is a box plot.

The figure. In a single figure, the tree generation method, demand pair generation method, algorithm and number of nodes in the original instance are constant. A single figure shows two box plots: one for the number of nodes in the kernel (in blue) and one for the number of demand pairs in the kernel (in orange). The x-axis shows the different numbers of demand pairs in the original instance, and the y-axis the size of the kernel.

The computation of the box and whiskers. The box goes from the first quartile (Q_1) to the third quartile (Q_3). The size of the box, $Q_3 - Q_1$, shows the *interquartile range* (IQR). It thus contains half of the data points. Inside the box, a horizontal line represents the median (Q_2) of the data. There are two whiskers. One whisker starts at Q_3 and goes to the largest point in the data that is at most 1.5 times the IQR away from Q_3. The other whisker starts at Q_1 and goes to the smallest point in the data that is at most 1.5 times the IQR away from Q_1. Diamonds mark any data points outside of the box and whiskers.

An example box plot computed using the `GN-algorithm` on `Prüfer-Random` can be found in Figure 12.

A note on outliers. External factors do not cause any outliers that occur in our experiments. The kernelisation algorithms cannot stop before they have applied all reduction rules exhaustively. Another reason for the existence of outliers is related to the running time of an algorithm. Since the counters we use to measure the running time of an algorithm are entirely deterministic and unaffected by external factors, the outliers in running time figures are also legitimate.

Any outlier that occurred during our experiments can occur in the real world. We cannot simply ignore them.

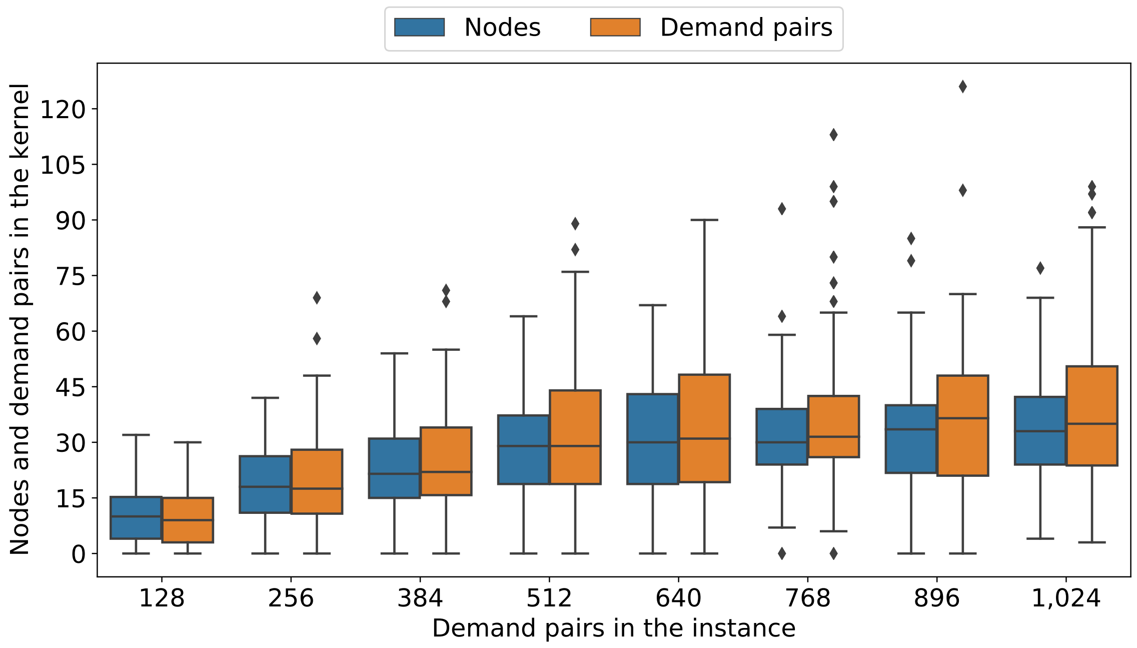


Figure 12: A box plot showing the remaining number of nodes and demand pairs in kernels computed by the algorithm by Guo and Niedermeier [23] on instances with a Prüfer tree with 768 nodes and uniformly at random demand pairs. For each number of demand pairs, there were 100 experiments.

8.1.2 Contour Plot for Kernel Size

The box plots described above only show the results for a constant number of nodes. We created a contour plot to show more results in a single plot and get more insight into the results on all instances of a particular type.

The figure. In a single contour plot, the tree generation method, demand pair generation method and algorithm are still constant. We also cannot show both the number of nodes and demand pairs in a kernel. To resolve this, we made one contour plot for the nodes and one for the demand pairs in the kernels.

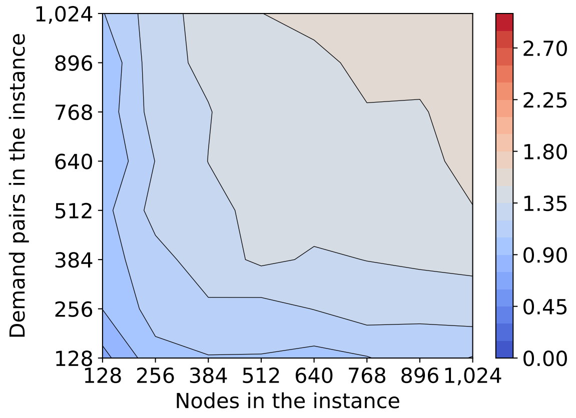
The x-axis of a contour plot shows the different numbers of nodes we tested, and the y-axis the number of demand pairs. Per grid point, the contour plot shows the base-10 logarithm of one plus the median of the instances corresponding to that grid point. These graphs are created using the `countour` and `contourf` functions in the matplotlib library for Python [21] [28].

All contour plots that show the kernel size use the same logarithmic scale between 0 and 3, in 20 steps of 0.15 each. The colours go from a dark blue at zero to a dark red at three. Using the same scale for all plots makes it easier to see the results on different instances without first interpreting the scale. We use a logarithmic scale to still show differences between “small” and “tiny” kernels, and we do not mind putting both “large” and “huge” kernels in one category.

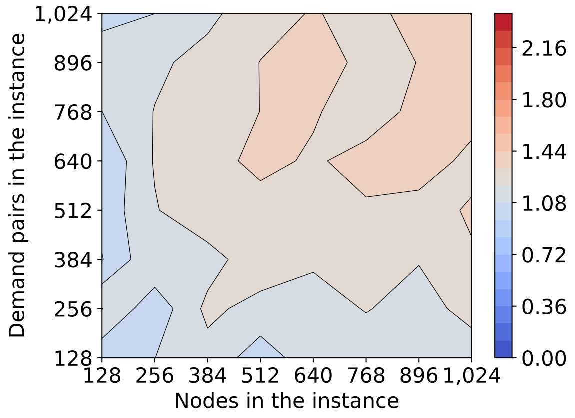
Subfigure 13a shows an example contour plot of the number of nodes in kernels computed by the `GN-algorithm` on `Prüfer-Random`.

A note on the spread of the data. An obvious advantage of using the box plots over the contour plots is that they show all data points instead of one summary point. This single summary point in the contour plot tells us nothing about the spread of the data. To battle this, we show a different contour plot with the base-10 logarithm of the IQR for each grid point every time we use a contour plot to talk about kernel size. These plots are also logarithmic, but they use a slightly different scale than the kernel size contour plots. The scale for the IQR contour plots goes from 0 to 2.4, in 20 steps of 0.12 each. The colour of this different contour plot indicates how certain we can be about the value displayed in the kernel size contour plot. Subfigure 13b shows the IQR contour plot corresponding to Subfigure 13a.

[21] `https://www.matplotlib.org/`



(a) Base-10 logarithm of the median number of nodes in kernels.



(b) Base-10 logarithm of the IQR of the number of nodes in kernels.

Figure 13: Contour plots of the number of nodes in the kernels computed by the algorithm by Guo and Niedermeier [23] on Prüfer trees with uniformly at random demand pairs. Each grid point is computed using 100 experiments.

8.1.3 Contour Plot for Running Time

To show the running time of an algorithm, we used a contour plot as well. This plot is created the same way as the plots for the kernel size, except that a grid point represents the base-10 logarithm of the median of the sums of all operations of all reduction rules. Since it contains different data, the scale is different as well. It goes from 3.8 to 9 in 20 steps of 0.26 each. Note that the scale does not start at zero. Subfigure 14a shows an example of such a contour plot for the number of operations by the `GN-algorithm` on `Prüfer-Random`.

For each of these contour plots, we also used a different contour plot showing the IQR. These plots use contour levels from 1.5 to 8.3, in 20 steps of 0.34 each. Subfigure 14b shows the contour plot with the spread corresponding to Subfigure 14a.

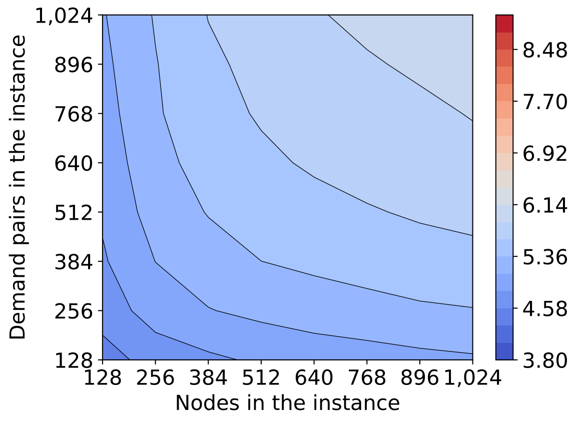
8.1.4 Heatmap for Dominant Reduction Rule

To show which reduction rule takes the most time, we used a graph comparable to a heatmap. We created a graph per combination of algorithm, tree generation method, and demand pair generation method. The x-axis shows the number of nodes in the instance, and the y-axis the number of demand pairs.

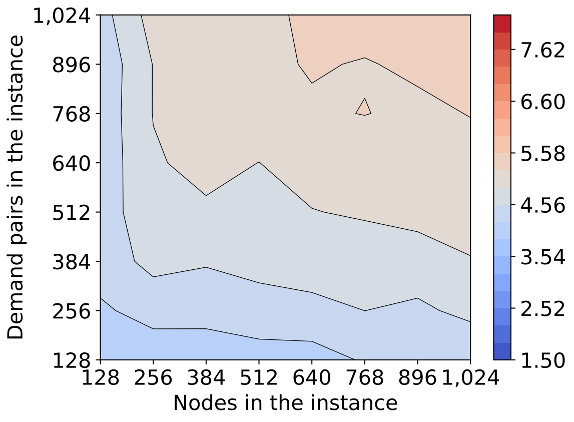
Each cell represents a grid point, with the colour corresponding to a particular reduction rule. On the right, the legend indicates which colour corresponds to which reduction rule. Each cell contains the fraction of the total time spent by the dominant reduction rule. We computed this fraction as follows. For each reduction rule, we grabbed the median number of operations it took to run. We then summed these medians to get the total "median" time. Finally, we computed the fraction of time spent by each reduction rule by dividing its median time by the total summed medians.

Figure 15 shows an example of such heatmap for the dominant reduction rules in the `GN-algorithm` on `Prüfer-Random`.

Note that these graphs only show one reduction rule. If two rules take approximately equal time, this is not clear from the heatmap. Such a situation could exist around 128 nodes and 384 demand pairs, or 128 nodes and 512 demand pairs in Figure 15, as between these two grid points, the dominant reduction rule changes from Dominated Edge to Unit Path. If such a situation exists, we can also show a box plot, which we explain in the following subsection.



(a) Base-10 logarithm of the median total number of operations.



(b) Base-10 logarithm of the IQR of the total number of operations.

Figure 14: Contour plots of the number of operations to compute kernels using the algorithm by Guo and Niedermeier [23] on Prüfer trees with uniformly at random demand pairs. Each grid point is computed using 100 experiments.

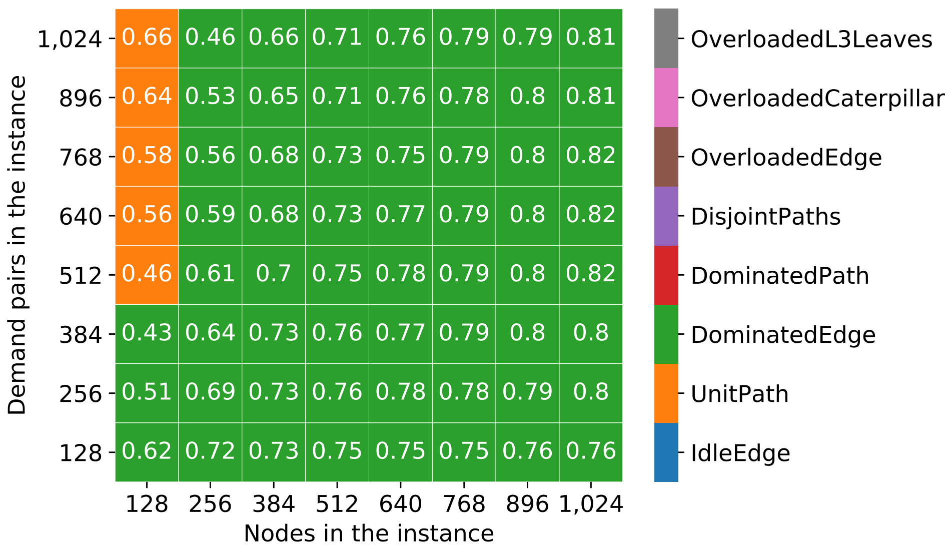


Figure 15: Dominant reduction rule in the algorithm by Guo and Niedermeier [23] on Prüfer trees with uniformly at random demand pairs. The text in each cell is the fraction of the sum of the median times of all reduction rules spent by the dominant reduction rule. These results were obtained using 100 experiments per cell.

8.1.5 Box Plot for Dominant Reduction Rule

We also used box plots to show the number of operations per reduction rule. These box plots are very similar to those that show the kernel size, as explained above. Instead of showing two boxes per number of demand pairs, these plots show r boxes, where r is the number of reduction rules in the algorithm that computed the results shown in that box plot.

Figure 16 shows an example box plot showing operations of the reduction rules in the `GN-algorithm` on `Prüfer-Random` with 128 nodes.

Insight in k. A weakness of all our figures is that they provide no insight into the parameter k. Exceptions are, of course, `TS` and instances from 3-SAT and CNF-SAT instances.

Since every experiment used the minimum solution size as k, which differs per instance, we could not easily give a single value for k for a set of instances. Knowing how large k is can influence the expectations for the size of a kernel.

We did not find an acceptable way to give insight into the value of k for our experiments without making the figures unclear.

8.2 Easy and Difficult Instances

This subsection focuses on both time and kernel size results to identify easy and difficult instances. It also discusses the instances generated from instances for Vertex Cover and CNF-SAT, and whether these could be identified as easy, difficult, or somewhere in between.

Research Question 1. *What types of instances are easy or difficult for the kernelisation algorithms?*

We briefly answer each subquestion before moving on to showing the results that led to these answers.

Research Question 1a. *For which types of instances does the kernelisation process result in a small kernel in little time?*

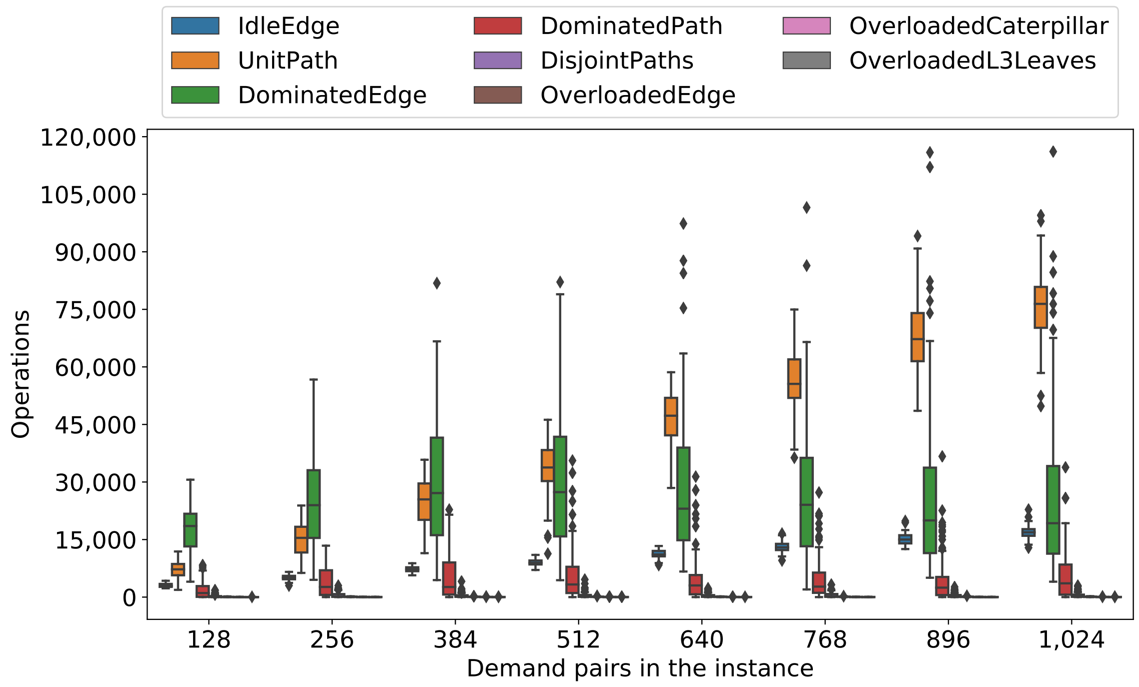


Figure 16: Box plot of the number of operations per reduction rule in the algorithm by Guo and Niedermeier [23] to compute kernels on Prüfer trees with 128 nodes and uniformly at random demand pairs. For each number of demand pairs, there were 100 experiments.

`TS5`, `TS10` and `TS100` seem easy. Furthermore, there are more easy instances in `Caterpillar` than in `Prüfer`. `Prüfer` contains, in turn, more easy instances than `Degree3`. Instances in `Caterpillar` also take less time than instances in `Prüfer` and `Degree3`.

Research Question 1b. *For which types of instances does the kernelisation process result in a large kernel in much time?*

`Short` seem the most difficult. We again see that `Degree3` are more complex than `Prüfer`. `Caterpillar` are not difficult. Furthermore, instances generated from CNF-SAT and 3-SAT instances are also difficult.

Research Question 1c. *How difficult are instances based on $\mathcal{NP}$-completeness proofs?*

From the previous subquestion, we know that instances generated from CNF-SAT and 3-SAT instances are challenging. Instances generated from Vertex Cover instances can be considered easy and difficult, depending on the number of nodes and demand pairs in the instance.

A final answer. Generally, `Caterpillar` are easy, as are instances with demand pairs through a very small or too large solution. `Degree3` is the most difficult. Additionally, `Short` and instances from 3-SAT and CNF-SAT instances are difficult as well. Finally, instances from Vertex Cover instances can be easy or difficult.

8.2.1 Easy Instances

We look for dark blue contour plots with a small IQR and no high running time to determine easy instances. There were plenty of these plots.

For both tested algorithms, `Caterpillar-TS5`, `Caterpillar-TS10` and `Caterpillar-TS100` were all easy. `Caterpillar-Long` were pretty easy as well.

However, the IQR on these instances was slightly larger on a small tree with many demand pairs than on `Caterpillar-TS`. `Prüfer-TS`5, `Prüfer-TS`10 and `Prüfer`100 are also easy to compute a kernel on, though on `Prüfer-TS`10, the IQR is slightly more prominent again. Finally, `Degree3-TS`5 were also easy for the algorithms.

In short, it is easy to compute a kernel on instances with demand pairs through a tiny solution. The same is true for instances with demand pairs through a solution that is too large for the size of the tree. These cases are both intuitive.

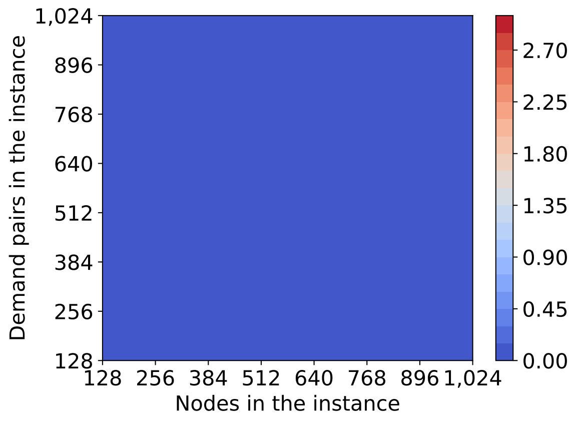
The contour plots for these instances look almost identical, so we only show the one for `Caterpillar-TS`10 computed by the `BQ-algorithm`. Figure 17 shows these plots. In particular, Subfigure 17a shows the number of nodes in the kernels, and Subfigure 17b the IQR.

Figure 18 shows the IQR of the number of nodes in the kernel computed by the `GN-algorithm` on `Prüfer-TS`10. The contour plot with the number of nodes in the kernels on these instances can be seen in Figure 40, and a box plot at 896 nodes in Figure 42, both in Appendix C.

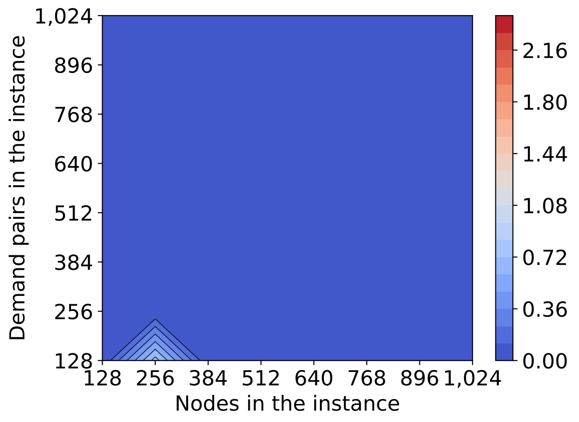
Figure 19 shows the IQR of the number of nodes in the kernel on `Caterpillar-Long`. The contour plot of the number of nodes in the kernel on these instances is displayed in Figure 41, and a box plot at 128 nodes in Figure 43, both in Appendix C again.

The `BQ-algorithm` also has not much trouble with `Degree3-TS`10; see Subfigure 20b. For these instances, the `GN-algorithm` results in somewhat larger kernels on larger trees with few demand pairs; see Subfigure 20a. The IQRs for these instances can be found in Figure 44 in Appendix C.

Running time on the easy instances. Though the instance types we discussed all result in small kernels, there are some differences in the time required to compute these kernels. In short, the algorithm can compute kernels on easy instances in `Caterpillar` faster than on other easy instances. Subfigure 21a shows contour plots of the running times of the `BQ-algorithm` on `Caterpillar-TS`100, and Subfigure 21b for `Prüfer-TS`5. The IQR plots for these figures can be found in Figure 45 in Appendix C.



(a) Base-10 logarithm of the median number of nodes in the kernels.



(b) Base-10 logarithm of the IQR of the number of nodes in the kernels.

Figure 17: Contour plots of the number of nodes in kernels computed by the algorithm by Bousquet *et al.* [4] on caterpillars with demand pairs through a solution of size 10. Each grid point is determined using 50 experiments.

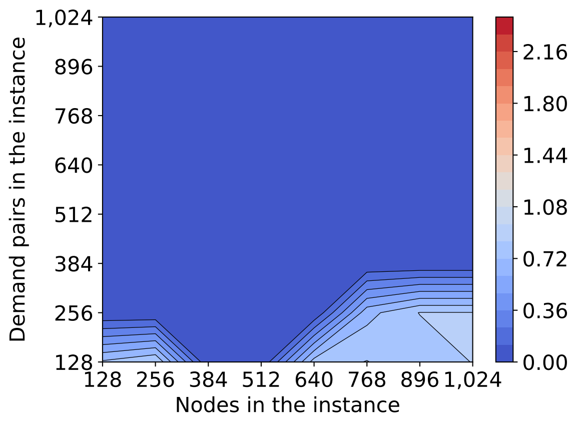


Figure 18: Contour plot of the base-10 logarithm of the IQR of the number of nodes in kernels computed by the algorithm by Guo and Niedermeier [23] on Prüfer trees with demand pairs through a solution of size 10. Each grid point is determined using 50 experiments.

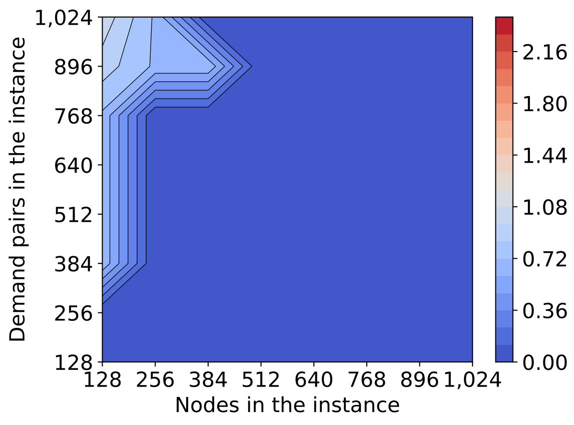
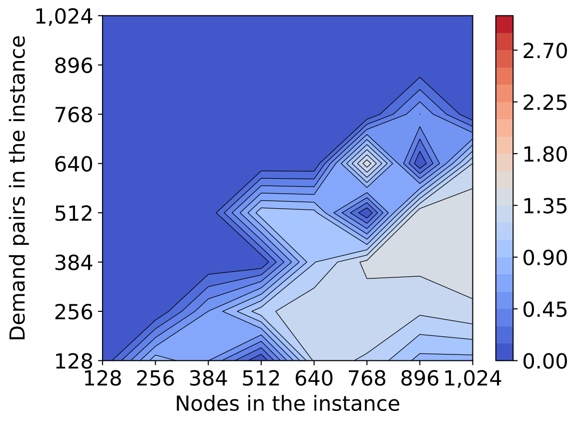
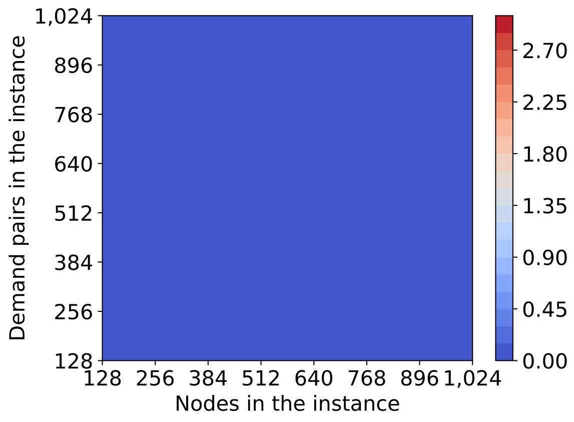


Figure 19: Contour plot of the base-10 logarithm of the IQR of the number of nodes in kernels computed by the algorithm by Bousquet *et al.* [4] on caterpillars with mainly long demand paths. Each grid point is determined using 100 experiments.

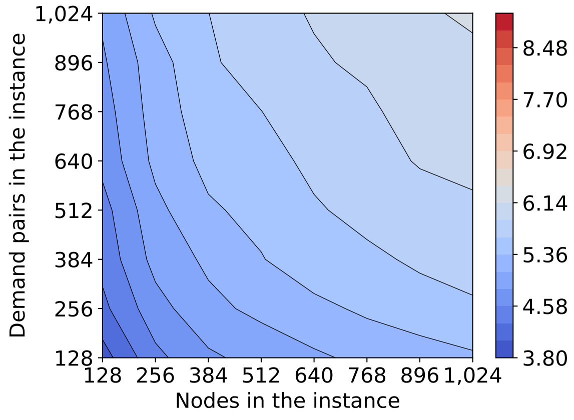


(a) Guo and Niedermeier [23].

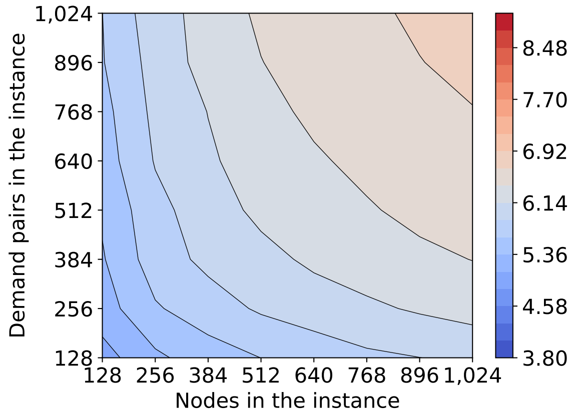


(b) Bousquet *et al.* [4].

Figure 20: Contour plots of the base-10 logarithm of the median number of nodes in kernels computed on Degree3 trees with demand pairs through a solution of size 10, using both algorithms. Each grid point is determined using 50 experiments.



(a) Instances with caterpillars and demand pairs generated through a solution of size 100.



(b) Instances with Prüfer trees and demand pairs generated through a solution of size 5.

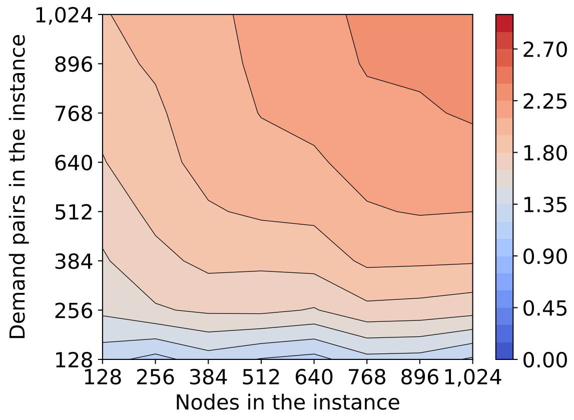
Figure 21: Contour plots of the base-10 logarithm of the median total number of operations needed to compute kernels using the algorithm by Bousquet *et al.* [4] on two different types of instances. Each grid point is determined using 50 experiments.

8.2.2 Difficult Instances

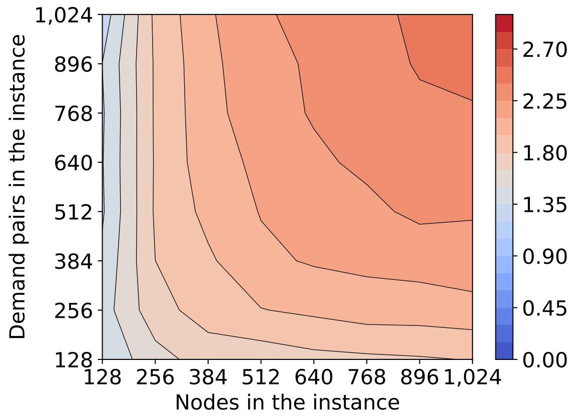
At the other end of the spectrum, there are difficult instances. The contour plots show that the two most difficult instances are `Degree3-Short` and `Degree3-Random`. The first has both a large number of nodes and a large number of demand pairs in its kernels. The latter has fewer nodes but still many demand pairs. Figure 22 shows the contour plots for the number of demand pairs in the kernels as computed by the `BQ-algorithm` for these two types of instances. The IQR of these two plots can be found in Figure 46 in Appendix C.

`Prüfer-Short` were also pretty difficult.

It seems that `Short` result in large kernels. With many short demand paths, we expect the number of edges that an algorithm needs to cut to separate each demand pair to be significant, as there is just a small chance for each combination of two demand paths to overlap. With a large value for k, the expected size of the kernel is larger than for a smaller value for k. This could be a reason why `Short` are difficult.



(a) Mostly short demand paths.



(b) Uniformly at random demand pairs.

Figure 22: Contour plots of the base-10 logarithm of the median number of demand pairs in kernels computed by the algorithm by Bousquet *et al.* [4] on Degree3 trees, for two different types of demand pairs. Each grid point is determined using 100 experiments.

Interesting instances were `Degree3-TS50`. If there are either few nodes or few demand pairs in the instance, the kernel becomes very small. If both these numbers increase, the size of the kernel also quickly grows, especially considering the demand pairs. Subfigures 23a and 23b show the contour plots with the number of nodes and demand pairs in kernels computed using the `BQ-algorithm`. The IQR for both of these plots can be found in Figure 47 in Appendix C. For some instance sizes, these instances can be considered difficult, and for some, easy.

Running time on the difficult instances. The contour plots of the instance types we talked about are all very similar, and we do not think they show anything meaningful, so we opted not to include them.

8.2.3 Instances Generated from Vertex Cover Instances

Now, we discuss the results on the instances generated from Vertex Cover instances.

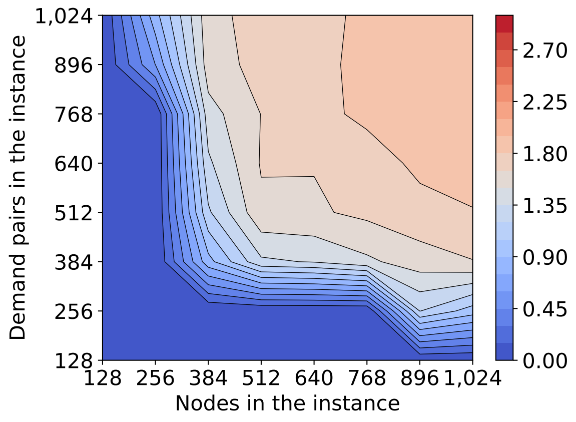
Kernel size. The size of the kernel of instances from Vertex Cover instances is highly dependent on the size of the instance. The kernel becomes tiny with more nodes than demand pairs, but it becomes huge with more demand pairs than nodes. This result is quite intuitive, considering that the graph used is a star graph. The amount of overlap between demand paths and how many edges are used by demand paths are factors that influence the size of the kernel.

The number of nodes and demand pairs in the kernels computed using the `BQ-algorithm` can be seen in Figure 24, with the IQR being present in Figure 48 in Appendix C.

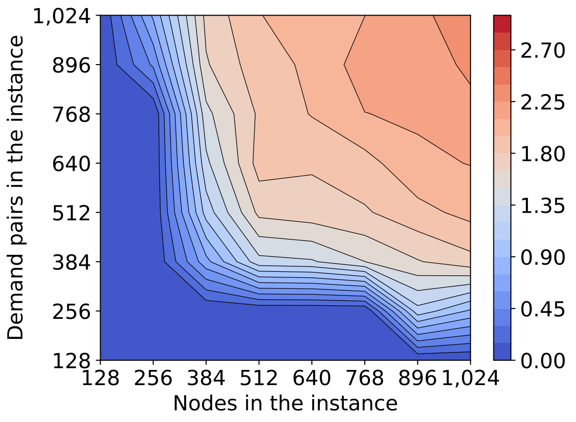
The IQR shows a large spread in the data around the border between the easy and difficult instances. On the parts where the kernel is large or small, the spread is pretty tiny again.

Running time. For the running time on these instances, we again have interesting results. These instances took the least time out of all generated instances for the `GN-algorithm`. On the other hand, they took the most time out of all generated instances for the `BQ-algorithm`. Contour plots for the operations of both algorithms can be seen in Figure 25, with the spread in Figure 49 in Appendix C.

We note that only the instances resulting in a large kernel required much time for the `BQ-algorithm`. The instances with a small kernel did not take that much time.

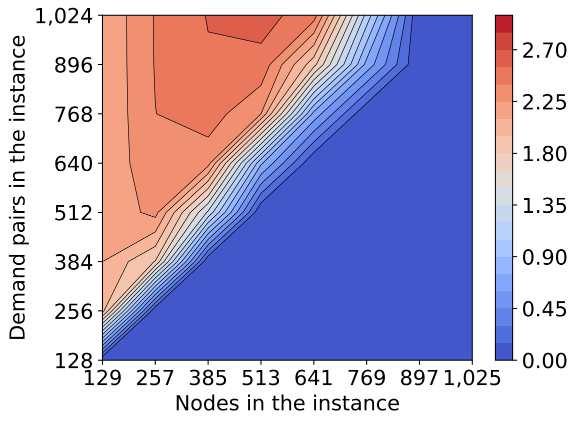


(a) Base-10 logarithm of the median number of nodes in the kernels.

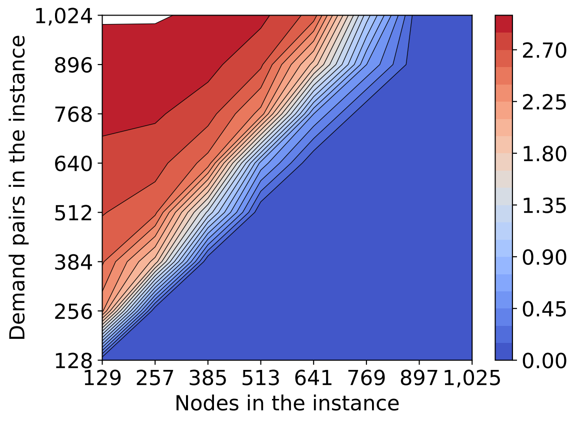


(b) Base-10 logarithm of the median number of demand pairs in the kernels.

Figure 23: Contour plots of the median number of nodes and demand pairs in kernels computed by the algorithm by Bousquet *et al.* [4] on Degree3 trees with demand pairs through a solution of size 50. Each grid point is determined using 50 experiments.

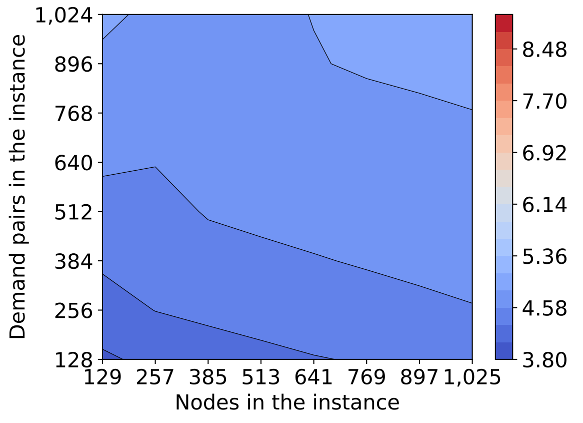


(a) Base-10 logarithm of the median number of nodes in the kernel.

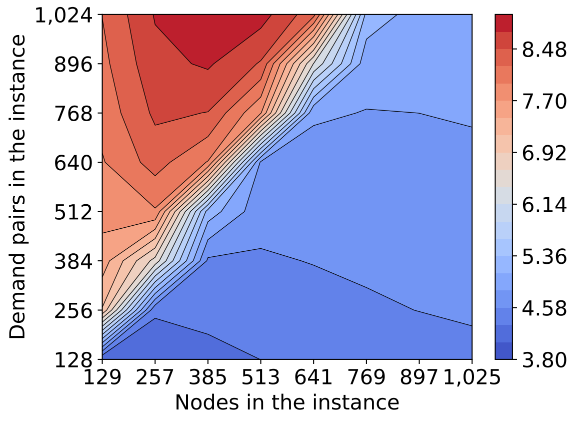


(b) Base-10 logarithm of the median number of demand pairs in the kernel. The white part in the top left of the plot means more than 1,000 demand pairs.

Figure 24: Contour plots of the median number of nodes and demand pairs in kernels computed by the algorithm by Bousquet *et al.* [4] on instances generated from Vertex Cover instances. Each grid point is determined using 100 experiments.



(a) Guo and Niedermeier [23].



(b) Bousquet *et al.* [4].

Figure 25: Contour plots of the base-10 logarithm of the median total number of operations needed to compute kernels on instances generated from Vertex Cover instances, using both algorithms. Each grid point is determined using 100 experiments.

8.2.4 Instances Generated from CNF-SAT Instances

Both the 3-SAT and CNF-SAT instances proved to be difficult for both algorithms. Over all 3-SAT instances, both algorithms removed approximately 0.5% of the demand pairs. The `GN-algorithm` removed about one-third of the nodes, but the `BQ-algorithm` could only remove about 0.5% of the nodes. Both algorithms removed no demand pairs in the instances from CNF-SAT instances. Here, the percentage of nodes removed was about 22% for the `GN-algorithm`, and the `BQ-algorithm` only came as far as about 0.01%.

Box plots with the kernel size results on the CNF-SAT instances for both algorithms can be found in Figure 26. For the 3-SAT instances, these plots can be found in Figure 50 in Appendix C.

We can explain this difference by looking at the reduction rules used. Reduction Rule 4.1.3 (Dominated Edge) can remove edges on these instances. The rule that seems to be its replacement in the second algorithm, Reduction Rule 4.2.3 (Unique Direction), does not work on all nodes. Since there are only a few inner nodes, and the demand paths from leaves often do not go in the same direction, the conditions for this rule are not optimal.

The instances created from CNF-SAT and 3-SAT instances can be compared quite well to `Degree3-Short`, which were also very difficult. Differences between the instances include that the root node in instances from CNF-SAT and 3-SAT instances only has two children and that the location of the short demand paths is fixed. Nevertheless, the similarities can explain that both types of instances are complex for the algorithms.

8.3 Scaling of Kernels with the Instance Size

The contour plots give us an excellent insight into the scaling of kernels with the size of the instance. We can identify specific patterns that occur, like the kernel size increasing closer to a corner of the contour plot. These patterns help us to answer Research Question 2.

Research Question 2. *Do kernels scale with the size of the instance?*

Again, we start by answering the research question before moving on to the relevant results.

Research Question 2a. *Does increasing the number of nodes in an instance mean that the kernel becomes larger?*

We cannot answer this subquestion for instances that were solved entirely, as there is no kernel to analyse.

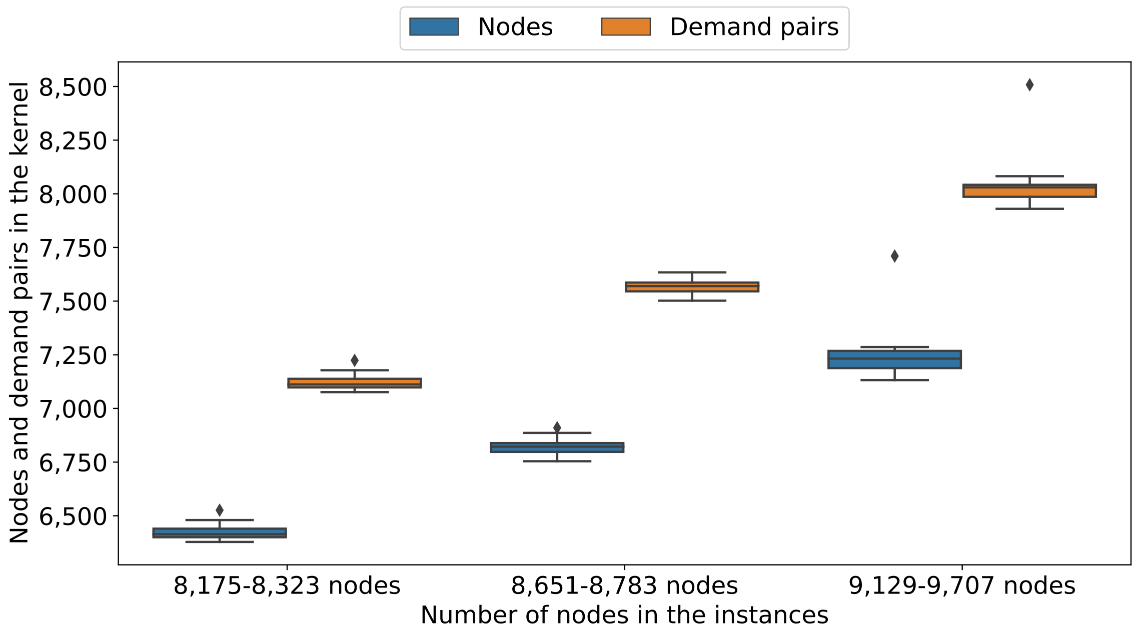
For almost all instances in `Degree3` and `Prüfer`, increasing the number of nodes in the instance means that the kernel size increases. An exception is `Prüfer-Long`, where the number of nodes in an instance does not seem to influence the size of the kernel. On instances created from 3-SAT or CNF-SAT instances, the size of the kernel increases as well as the number of nodes in the instance increases.

For `Caterpillar-Short` and `Caterpillar-Random`, the size of the kernel decreases when we add more nodes to the instance. The same goes for the instances created from Vertex Cover instances.

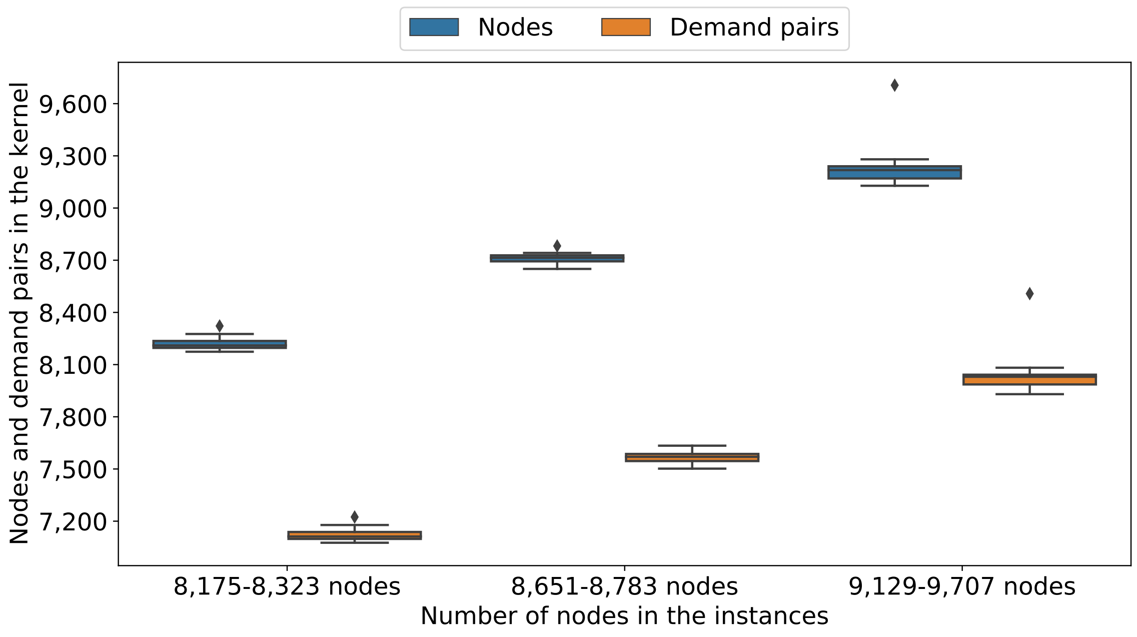
Research Question 2b. *Does increasing the number of demand pairs in an instance mean that the kernel becomes larger?*

Again, we cannot answer this subquestion for instances the algorithms solved completely.

Except for one instance type per algorithm, all other instances resulted in a larger kernel when the instance contained more demand pairs. For the `GN-algorithm`, the exception was `Degree3-TS10`, where the kernel size decreases if we add more demand pairs to the instance. The `BQ-algorithm` differed on `Degree3-TS20`. At first, the kernel size increased when we added more demand pairs to the instance, but after a while, it decreased again.



(a) Guo and Niedermeier [23].



(b) Bousquet *et al.* [4].

Figure 26: Box plots of the number of nodes and demand pairs left in kernels computed on the CNF-SAT instances, using both algorithms. We explained the instance categories displayed here in Table 3.

Research Question 2c. *Does increasing the number of nodes in an instance mean that the computation of the kernel takes longer?*

On almost all tested instances, the time to compute a kernel increases when the number of nodes in the instance increases. Only the instances from VERTEX COVER instances, in combination with the `BQ-algorithm`, resulted in a running time that decreases the more nodes we add to the instance. Adding more nodes to these instances means that edges are used by fewer demand paths, which apparently results in a quicker kernelisation process.

Research Question 2d. *Does increasing the number of demand pairs in an instance mean that the computation of the kernel takes longer?*

Both algorithms needed more time to compute a kernel for all tested instances when the number of demand pairs in the instance increased.

A final answer. We only answer this research question for instances that were not easy. That is, only for instances that resulted in a kernel and were not immediately solved. In many cases, the kernel size and computation time increase as the instance size grows. In a few cases, the size of the kernel decreases when the instance grows.

Approach. We look separately at patterns we can identify in contour plots that show the kernel size and contour plots that show the IQR of the kernel size. We do not explicitly look at the patterns in the contour plots with the number of operations, as they all show the same pattern. Subfigure 21b shows an example of this pattern. This figure shows a contour plot of the operations required by the `BQ-algorithm` on `Prüfer-TS5`. The only exception on this is the plot with the running time for the `BQ-algorithm` on instances created from the VERTEX COVER instances. We have seen this plot before; see Subfigure 25b.

8.3.1 Patterns Identified in the Kernel Size

The patterns we identified in contour plots showing the number of nodes are identical to those observed in the contour plots with the number of demand pairs. There are six main patterns: nothing, top right, top left, bottom right, top and right.

No pattern. We have already seen contour plots that show no scaling of the kernel size, for instance, in Subfigure 17a. We observed this pattern in 11 of the 25 instance types for which we created contour plots. This pattern happened on `Caterpillar-Long`, `Caterpillar-TS5`, `Caterpillar-TS10`, `Caterpillar-TS20`, `Caterpillar-TS50`, `Caterpillar-TS100`, `Degree3-TS5`, `Prüfer-TS5`, `Prüfer-TS10`, `Prüfer-TS20`, and `Prüfer-TS100`.

Top right. The next most occurring pattern was top right. This pattern was present in 8 of the 25 contour plots. These plots occurred for `Degree3-Short`, `Degree3-Long`, `Degree3-Random`, `Degree3-TS50`, `Degree3-TS100`, `Prüfer-Short`, `Prüfer-Random`, and `Prüfer-TS50`. Figure 22 shows an example of this pattern.

Top left. The top left pattern occurred three times: on `Caterpillar-Short` and `Caterpillar-Long`, and on instances generated from VERTEX COVER instances. This last one does not entirely go to the top left corner, but we felt it belonged in this category. We already showed the contour plots for these instances; see Figure 24. The graph for `Caterpillar-Short` is still interesting to show. It can be found in Figure 27 for the `GN-algorithm`, and the IQR for that plot in Figure 51 in Appendix C.

Top. One pattern remains that occurred for both algorithms. This pattern means that the number of nodes does not seem to impact the size of the kernel and occurred on `Prüfer-Long`. Figure 28 shows the contour plot for this scenario as computed using the `GN-algorithm`. The IQR for Figure 28 can be found in Figure 52 in Appendix C.

Bottom right. We observed the bottom right case for `Degree3-TS`10. The contour plot with the result of the `BQ-algorithm` shows no scaling here at all; see Subfigure 20b. The `GN-algorithm` shows a more interesting pattern, which Subfigure 20a shows. The IQR for this plot is present in Appendix C in Figure 44.

Right. The last case refers to `Degree3-TS`20. Using the results obtained using the `GN-algorithm`, we would classify this type as going to the top right; see Subfigure 29a. However, the `BQ-algorithm` shows a different pattern. This pattern can be seen in Subfigure 29b. The IQR of these plots is shown in Figure 53 in Appendix C. This pattern implies that having either a few or many demand pairs results in a small kernel and that a number of demand pairs somewhere in the middle results in a larger kernel.

8.3.2 Patterns Identified in the Spread of the Kernel Size

The contour plots with the IQR of the kernel size show essentially the same patterns as the normal contour plots. Though, they are not all as clear. As an example, consider Subfigure 53a in Appendix C, which shows the IQR of the number of nodes in kernels computed by the `GN-algorithm` on `Degree3-TS`20. In this subfigure, we see a pattern going to the right, but plenty of height differences make it difficult to say how the IQR scales on these instances.

Different patterns compared to kernel size plots. We briefly mention the types of instances where the pattern observed in the contour plot with the IQR differs from the pattern seen in the contour plots with the kernel size.

`Caterpillar-TS`50 go from having no pattern to having a pattern towards the top. This change can be seen in Figure 54 in Appendix C. Note that this figure only shows the results of the `BQ-algorithm`. For the `GN-algorithm`, the pattern is the same.

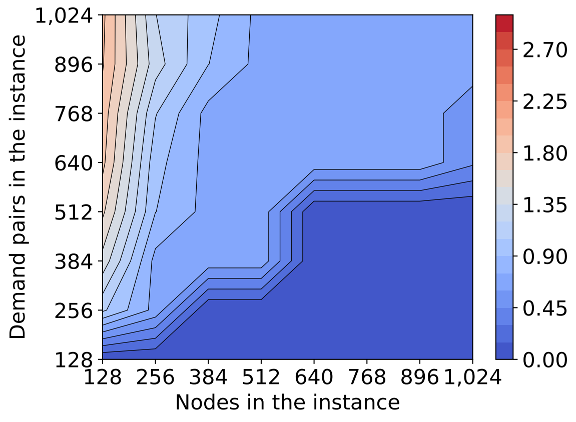


Figure 27: Contour plot of the base-10 logarithm of the median number of nodes in kernels computed by the algorithm by Guo and Niedermeier [23] on instances with a caterpillar and mostly short demand paths. Each grid point is determined using 100 experiments.

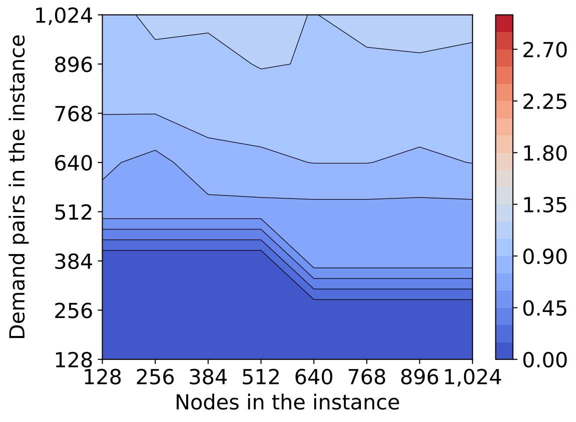
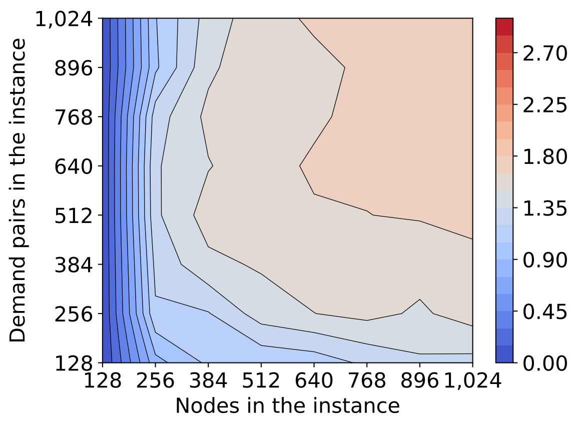
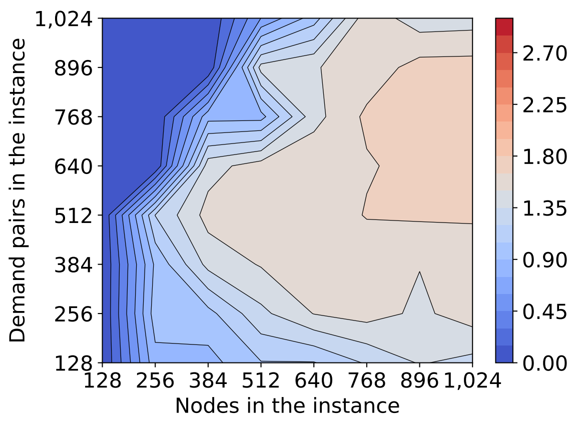


Figure 28: Contour plot of the base-10 logarithm of the median number of nodes in kernels computed by the algorithm by Guo and Niedermeier [23] on instances with a Prüfer tree and mostly long demand paths. Each grid point is determined using 100 experiments.



(a) Guo and Niedermeier [23].



(b) Bousquet *et al.* [4].

Figure 29: Contour plots of the base-10 logarithm of the median number of nodes in kernels computed on instances with a Degree3 tree and demand pairs going through a solution of size 20 for both algorithms. Each grid point is determined using 50 experiments.

Specifically for the results with the `GN-algorithm`, the pattern on `Degree3-Long` changes from top right to top. A visualisation can be found in Figure 55 in Appendix C.

`Degree3-TS`20 swap patterns between the two algorithms. The pattern in the plot of the `GN-algorithm` goes from top right to right. The pattern in the plot of the `BQ-algorithm` goes from right to top right. Figure 29 shows the contour plots with the kernel size, and Figure 53 in Appendix C the contour plot with the IQR.

Then, the pattern on `Prüfer-TS`20 for the `GN-algorithm` goes from nothing to right, as can be seen in Figure 56 in Appendix C.

Finally, the pattern on instances generated from the VERTEX COVER instances goes from somewhat top left to something we cannot easily classify. The pattern achieved by the `BQ-algorithm` can be seen in Figure 48 in Appendix C.

8.4 Demand Pair Generation Methods

In this subsection, we discuss the results of the two slightly more advanced ways to generate demand pairs. With the results presented in this subsection, we answer Research Question 3.

Research Question 3. *How usable are the demand pair generation methods we introduce?*

As before, we answer the subquestions and then show the results that led to these answers. Finally, we briefly discuss the limitations of both methods and which approach we favour.

Research Question 3a. *Do instances with mostly short demand paths result in a smaller or larger kernel than instances with mostly long demand paths?*

`Short` result in a larger kernel than `Long` with the same tree type.

Research Question 3b. *Does computing a kernel on an instance with mostly short demand paths take shorter or longer than computing a kernel on an instance with mostly long demand paths?*

The algorithms required less time to compute a kernel on `Short` than `Long` with the same tree type.

Research Question 3c. *Do instances with demand pairs generated using a small solution result in a smaller or larger kernel than instances with demand pairs generated using a large solution?*

We cannot answer this subquestion for `Caterpillar-TS`, as these were all solved.

`TS`50 resulted in the largest kernels, followed by `TS`20. The other three possible solution sizes all resulted in smaller kernels.

The answer for this subquestion is consequently yes, until the solution becomes too large.

Research Question 3d. *Does computing a kernel on an instance with demand pairs generated using a small solution take shorter or longer than computing a kernel on an instance with demand pairs generated using a large solution?*

For this subquestion, we ignore `TS`100, as the solution size used for these was too large for the size of the instances. We briefly explain this later.

Both algorithms needed the least time to compute kernels on `TS`50. `Caterpillar-TS`5 and `Caterpillar-TS`10 were the subsequent fastest of `Caterpillar-TS`. `Prüfer-TS`5 and `Prüfer-TS`10 were also the following fastest of `Prüfer-TS`. The slowest instances were `TS`20. `Degree3-TS`20 took less time than `Degree3-TS`5 and `Degree3-TS`10.

If we consider 20 to be neither a small nor a large solution, we can conclude that the algorithms needed more time to compute kernels on instances with demand pairs through a small solution than on instances with demand pairs through a large solution. If 20 is a large solution, we cannot answer this question for all tree types.

A final answer. How usable the demand pair generation methods are, depends on the context.

We assume the methods will be used to generate instances that are difficult to compute a kernel on. `Short` were the most challenging. The method that generated demand paths according to a length distribution would get our recommendation over the others. However, both methods are not perfect.

8.4.1 Demand Pairs From a Length Distribution

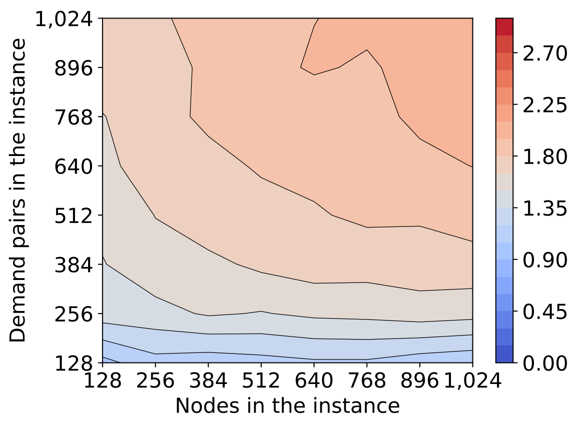
We used either mostly long or mostly short demand paths for instances that use this method.

Mostly short demand paths. Again, the results for nodes and demand pairs in the kernel are pretty similar, as are the results of the two algorithms. We show a mixture here. We have mentioned the results for nodes in kernels on `Caterpillar-Short` before. The `GN-algorithm` computed these results, which Figure 27 shows. In Appendix C, Figure 51 shows the IQR. We have also used the results showing the number of demand pairs in kernels on `Degree3-Short` computed by the `BQ-algorithm` before. They can be found in Subfigure 22a, and the IQR in Subfigure 46a in Appendix C. Finally, Figure 30 shows the number of demand pairs in kernels on `Prüfer-Short` as computed by the `GN-algorithm`.

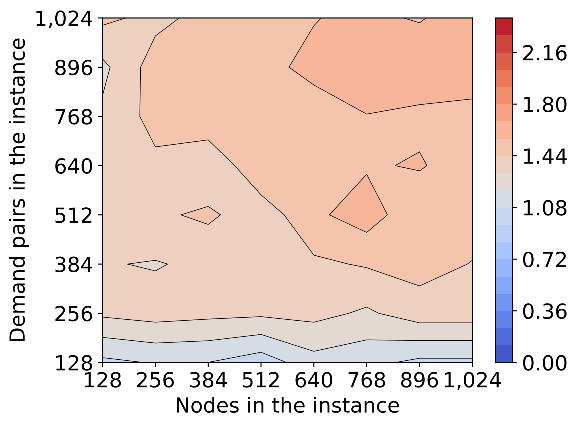
`Caterpillar-Short` were the easiest for the algorithms to compute a kernel on, and `Prüfer-Short` and `Degree3-Short` quite tricky. `Degree3-Short` were slightly more difficult than `Prüfer-Short`. Finally, there were no significant differences in computation time for all of these instances.

Mostly long demand paths. We show a mixture of results for `Long`. Figure 41 in Appendix C shows the number of nodes in kernels computed using the `BQ-algorithm` on `Caterpillar-Long`. The IQR for these results is present in Figure 19. For `Degree3-Long`, we show the number of nodes in the kernel again, computed using the `GN-algorithm`. The results can be seen in Figure 55 in Appendix C. Finally, for the number of nodes in kernels computed on `Prüfer-Long` using the `GN-algorithm`, see Figure 28 and its IQR in Figure 52 in Appendix C.

`Long` all seem to result in smaller kernels than `Short`. They also require less time than `Short`. For instance, consider `Prüfer-Long` and `Prüfer-Short`; see Figure 31. The IQR



(a) Base-10 logarithm of the median number of demand pairs in the kernels.



(b) Base-10 logarithm of the IQR of the number of demand pairs in the kernels.

Figure 30: Contour plots of the number of demand pairs in kernels computed on instances with Prüfer trees and mostly short demand paths, using the algorithm by Guo and Niedermeier [23]. Each grid point is determined using 100 experiments.

for these situations can be found in Figure 57 in Appendix C. From this last figure, it is also clear that there is more spread in the time required to compute the kernels on `Long`. The larger spread occurs in all cases, except for `Caterpillar-Long` and the `BQ-algorithm`, where it is the other way around.

Possible explanations. We saw that `Long` resulted in a smaller kernel than `Short`. One reason for this could be that longer demand paths have a larger chance of being removed from the instance when the algorithm cuts an edge, making computing the kernel on the remainder of the instance easier. On the other hand, with short demand paths, edges are generally used by fewer demand paths. Apparently, that does not make computing a kernel easier.

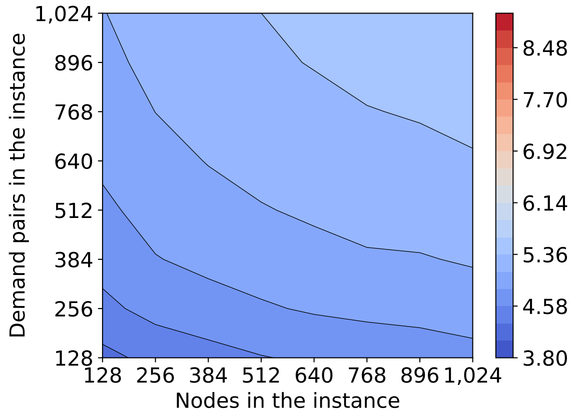
We also saw that the computation of kernels on `Long` took less time than on `Short`. An apparent reason is that the kernels on `Short` are larger than on `Long`. The algorithms need to perform more operations to compute the smaller kernel on `Long`, so they need more time. Another argument could again be about how many demand paths use a certain edge. For instance, reduction rules like Reduction Rules 4.1.3 and 4.2.3 benefit from edges being used by fewer demand paths.

8.4.2 Demand Pairs Through a Solution

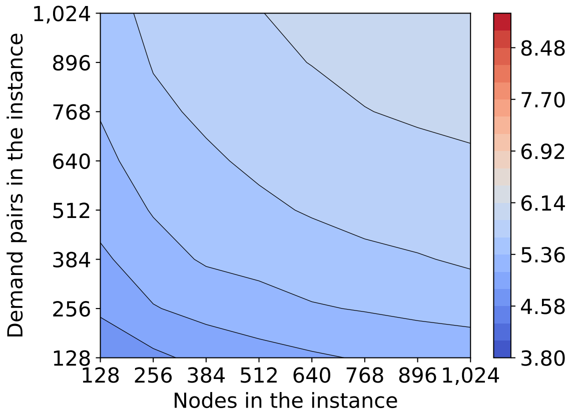
The sizes of the solutions we generated demand pairs through are 5, 10, 20, 50 and 100. We do not look at all of these sizes but rather at three categories: small, large, and in between.

A small solution. `TS5` and `TS10` contain hardly any useful results. There are no kernels, or they are tiny. `TS10` has slightly larger kernels than `TS5`.

Some results can be seen in figures we have shown before. In particular, Subfigure 17a shows the number of nodes in kernels computed by the `BQ-algorithm` on `Caterpillar-TS10`, and Subfigure 17b the corresponding IQR. Figure 20 shows the number of nodes in kernels computed on `Degree3-TS10` by both algorithms, and Figure 44 in Appendix C the corresponding plot with the IQR. Finally, in Appendix C, Figure 40 shows the number of nodes in kernels computed on `Prüfer-TS10` using the `GN-algorithm`. The IQR plot can be found in Figure 18.



(a) Mostly short demand paths.



(b) Mostly long demand paths.

Figure 31: Contour plots of the base-10 logarithm of the median total number of operations required to compute kernels on instances with Prüfer trees for two demand pair generation methods, using the algorithm by Guo and Niedermeier [23]. Each grid point is determined using 100 experiments.

A large solution. In `TS`100, we again see either no or tiny kernels. Only `Degree3-TS`100 with many nodes and many demand pairs resulted in a large kernel. This can be seen in Figure 32. `Caterpillar-TS`100 and `Prüfer-TS`100 resulted in mostly no kernels, so they are not interesting to show.

A solution of size 100 is too large for the size of instances we used. Let us consider the extreme case of instances with 128 nodes. There are 127 edges, from which we choose 100 to put in the solution. The first 100 demand pairs we generate have a path that goes through exactly one of these edges, and they have to be pairwise edge-disjoint. This restriction means that there are at least 73 demand paths of length one. The algorithm will, of course, quickly cut the corresponding edges, and the remaining instance will be a lot smaller.

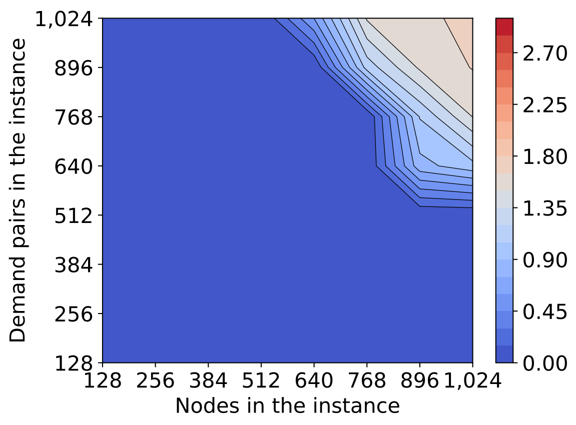
A medium solution. That leaves us with `TS`20 and `TS`50. These generally resulted in larger kernels than instances with demand pairs generated through a smaller or larger solution.

For `Caterpillar-TS`, however, only a few small kernels were computed. The algorithms could solve the other instances. Figure 54 in Appendix C shows the number of nodes in kernels on `Caterpillar-TS`50 achieved by the `BQ-algorithm`.

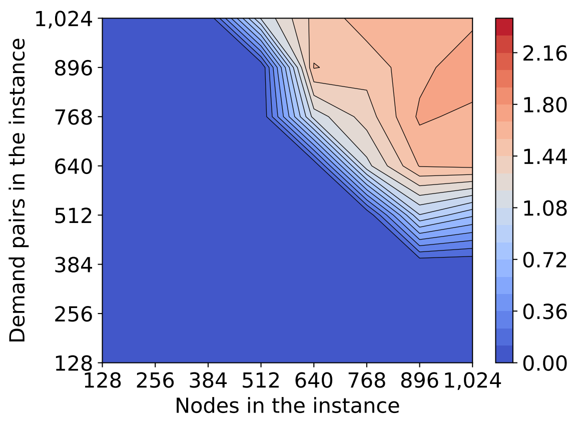
We obtained similar results on `Prüfer-TS`20 and `Prüfer-TS`50. There are only some kernels for instances with many nodes and demand pairs. For instance, the number of nodes in the kernel computed by the `GN-algorithm` on `Prüfer-TS`20 can be seen in Figure 56 in Appendix C.

Finally, let us consider `Degree3-TS`. These are the most interesting of the tree types. Depending on the algorithm, `Degree3-TS`20 can result in larger kernels than `Degree3-TS`50. Consider Figure 29, where the number of nodes in the kernels computed by both algorithms is displayed on `Degree3-TS`20. The corresponding plot with IQR can be found in Figure 53 in Appendix C. The number of nodes in kernels on `Degree3-TS`50 is similar for the two algorithms. For the `BQ-algorithm`, these results are displayed in Figure 23, with the IQR in Figure 47 in Appendix C.

Suppose we do not take `TS`100 into account. In that case, we can confidently say that instances with demand pairs through a small solution result in a smaller kernel than instances with demand pairs through a large solution. Of course, this result is not entirely unexpected. We used the size of the solution for our parameter k. As both algorithms result in a kernel that becomes larger as k grows, we do not see any weird behaviour.



(a) Base-10 logarithm of the median number of nodes in the kernels.



(b) Base-10 logarithm of the IQR of the number of nodes in the kernels.

Figure 32: Contour plots of the number of nodes in kernels on instances with Degree3 trees and demand pairs through a solution of size 100, using the algorithm by Guo and Niedermeier [23]. Each grid point is determined using 50 experiments.

Running time. The running time on `Caterpillar-TS` is less for both algorithms than on instances with the other tree types. The algorithms require fewer operations for instances with a small or huge solution. They require the most time for `TS`20 and `TS`50.

One could expect that instances with a small solution would take less time than instances with a large solution. With our generation method, cutting an edge will, on average, remove more demand pairs from the instance when the solution size is small than when the solution size is large. However, the edges in the solution are each used by more demand paths, which could mean that some reduction rules take more time. Our experiments seem to show that this second argument has more impact than the first argument.

8.4.3 A Small Evaluation of Both Methods

We briefly evaluate both demand pair generation methods and explain why we think generating mostly short demand paths is the best option from all of these. As mentioned earlier, both methods have their limitations.

First, to generate demand paths of a certain length, we need information about the length of all paths in the tree. Computing this can take much time if an instance is large. On the other hand, to generate short demand paths, we could take a more greedy approach by picking a node uniformly at random and using a `BFS` or `DFS` algorithm to find all nodes within a certain distance of the first node. For long demand paths, we fear that method would not work.

Secondly, while it can be advantageous to be able to influence the size of the solution directly, our method of generating demand pairs through a solution is not optimal. It is easy to choose a solution size that is way too big. We expect this to be because of the limitation we talked about in Subsection 6.2. We cannot recommend this method with this limitation. It may be possible to improve this method by generating the first k pairwise edge-disjoint demand paths iteratively, so they do not limit each other. Additionally, it can be wise to ensure that there are no demand paths of length one, as the algorithm would almost immediately remove these and all demand paths through the corresponding solution edges.

So, when mainly generating short demand paths, the limitation of having to compute the length of the path between each pair of nodes can somewhat be avoided. Plus, these instances seem to be the most difficult, making them more usable in future experiments to stretch the limits of kernelisation algorithms.

8.5 Dominant Reduction Rules

Research Question 4. *Which reduction rules take the most time during kernelisation?*

We answer the subquestions and then show the results that we used to find the answers.

Research Question 4a. *Which reduction rules take the most time on easy instances?*

There was mainly one dominant reduction rule in each algorithm. For the `GN-algorithm`, this was Reduction Rule 4.1.3 (Dominated Edge). On some of the easy instances, with demand pairs through a (too) large solution, Reduction Rule 4.1.2 (Unit Path) took most of the time.

In the `BQ-algorithm`, Reduction Rule 4.1.5 (Disjoint Paths) was the dominant rule. In a very few cases, again with demand pairs through a (too) large solution, Reduction Rule 4.2.1 (Unit Path) dominated the running time.

Research Question 4b. *Which reduction rules take the most time on difficult instances?*

On the difficult instances, the dominating reduction rule in the `GN-algorithm` was again Reduction Rule 4.1.3 (Dominated Edge). In arguably the most difficult instances (`Degree3-Short`), Reduction Rule 4.1.5 (Disjoint Paths) would sometimes dominate the running time.

The `BQ-algorithm` had one clear dominating reduction rule: Reduction Rule 4.2.2 (Disjoint Paths). This rule was only overtaken by Reduction Rule 4.2.5 (Common Factor) in some instances from VERTEX COVER instances and most instances from 3-SAT and CNF-SAT instances.

A final answer. The dominating rule in the `GN-algorithm` was in most cases Reduction Rule 4.1.3 (Dominated Edge). For the `BQ-algorithm`, Reduction Rule 4.2.2 (Disjoint Paths) dominated the running time.

After comparing the times spent by these two rules, we concluded that the Disjoint Paths rule was the overall dominating reduction rule.

The results. Both algorithms have one reduction rule that took the most time in many cases. It did not seem to matter whether an instance was easy or difficult.

8.5.1 Dominant Reduction Rules in the `GN-algorithm`

Reduction Rule 4.1.3 (Dominated Edge) dominated the running time of the `GN-algorithm`. In 9 of the 25 types of generated instances, all grid points show that this rule took the most time. On `Prüfer-Long`, this looks like Figure 33.

The second rule that appears most in the heatmaps is Reduction Rule 4.1.2 (Unit Path). This one dominated the running time when there were many demand pairs in an instance with few nodes; see, for instance, the results on `Prüfer-Random` in Figure 15. Specifically, this happened on `TS50` and `TS100`, with the rule dominating more instances in the latter case. Another example on `Degree3-TS50` is present in Figure 34.

There are three more rules that dominated the running time in a minimal number of cases. First, Reduction Rule 4.1.4 (Dominated Path) sometimes dominated the running time when there were few nodes and many demand pairs in `Caterpillar-Short`. Then, Reduction Rule 4.1.5 (Disjoint Paths) dominated the running time on `Degree3-Short` with many nodes and many demand pairs. This rule also dominated the running time on the instances from 3-SAT and CNF-SAT instances. Finally, Reduction Rule 4.1.2 (Unit Path) dominated the running time on `Degree3-TS100` with many nodes and few demand pairs. The heatmaps for these cases can be found in Figure 58, Figure 59, Figure 60 (only 3-SAT) and Figure 61 in Appendix C respectively.

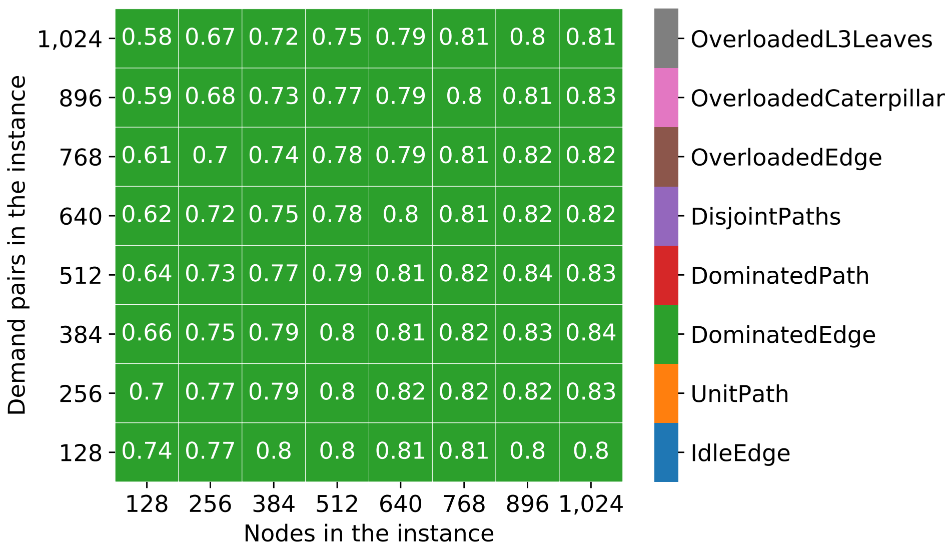


Figure 33: Dominant reduction rule in the algorithm by Guo and Niedermeier [23] on Prüfer trees with mostly long demand paths. Each cell is determined using 100 experiments. The text in each cell is the fraction of the total time spent by the dominant reduction rule.

There is one final interesting heatmap: the one on the instances generated from the Vertex Cover instances. The heatmap can be found in Appendix C in Figure 62. Four different reduction rules dominated the running time on these instances, depending on the size of the instance. However, we do not consider this heatmap, as we have shown in Subsection 7.1.2 that in this case, there are quite some differences between the actual time and the measured number of operations.

8.5.2 Dominant Reduction Rules in the `BQ-algorithm`

The results achieved by the `BQ-algorithm` tell a more straightforward story. The rule that dominated the running time in most of the experiments is Reduction Rule 4.2.2 (Disjoint Paths). For instance, for `Prüfer-Long`, Figure 35 shows the heatmap. Reduction Rule 4.2.2 (Disjoint Paths) dominated the running time entirely in 17 of the 25 types of instances and in at least three-quarters of all other types of instances. The only two exceptions are the instances generated from the Vertex Cover instances and those from 3-SAT and CNF-SAT instances.

For the instances from Vertex Cover instances, we again got a mixture of dominating rules, as can be seen in Figure 63 in Appendix C.

For instances from CNF-SAT and 3-SAT instances, Reduction Rule 4.2.5 (Common Factor) was the dominating reduction rule. This can be seen in Figure 64 for 3-SAT in Appendix C. For CNF-SAT, the results are very similar.

The only other rule that dominated the running time in some cases was Reduction Rule 4.2.1 (Unit Path). This rule dominated the running time for some instance types with few nodes and many demand pairs. Again, this happened mainly on `TS`; see for instance Figure 65 in Appendix C, in which we show the dominant reduction rule on `Caterpillar-TS`100.

Final notes. We concluded that the Disjoint Paths reduction rule was the overall dominating reduction rule. This rule is present in both algorithms: Reduction Rules 4.1.5 and 4.2.2. We concluded this because the rule dominated the running time in the `BQ-algorithm`, and this algorithm was the slowest of the two.

The reduction rule was also present in the `GN-algorithm`. Here, it occurred way later in the ordering of reduction rules: fifth instead of second. The other ordering means that the algorithm

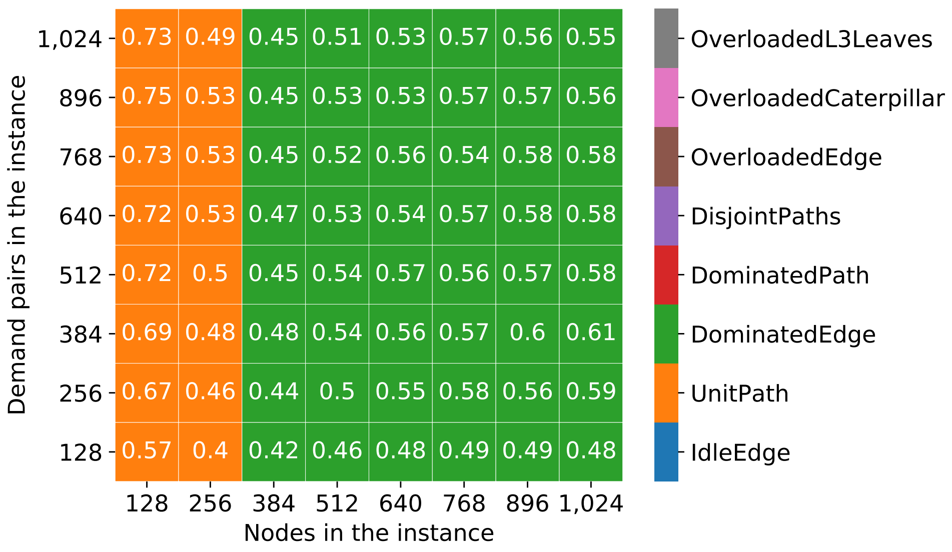


Figure 34: Dominant reduction rule in the algorithm by Guo and Niedermeier [23] on Degree3 trees with demand pairs going through a solution of size 50. Each cell is determined using 50 experiments. The text in each cell is the fraction of the total time spent by the dominant reduction rule.

checked the rule less often, but also that the earlier four rules might have decreased the instance size by a lot already. The Disjoint Paths rule generally needed less time on smaller instances. In the `BQ-algorithm`, the algorithm applied the rule after almost every single modification. It was thus checked more often and on larger instances. We would be very interested to see what would happen if this rule were the last in both algorithms. Sadly, we did not have enough time to test this.

In Section 4, we mentioned that Reduction Rule 4.2.3 (Unique Direction) seems to take care of the same edges as Reduction Rule 4.1.1 (Idle Edge) and Reduction Rule 4.1.3 (Dominated Edge). The time complexity of the first rule in the current implementation is $O(n^2p)$. The other two rules together have the same time complexity. We would also be interested to see how these rules behave when we directly compare them to each other. Assume Disjoint Paths is the last rule in both algorithms. Reduction Rule 4.1.3 (Dominated Edge) was the dominating rule in the `GN-algorithm`. Would that mean that Reduction Rule 4.2.3 (Unique Direction) would be the dominating rule in the `BQ-algorithm`? Again, we did not have enough time to test this.

8.6 Comparing the Two Kernelisation Algorithms

Finally, to answer Research Question 5, we compare the two kernelisation algorithms we studied.

Research Question 5. *Is the kernelisation algorithm by Bousquet et al. [4] better than the kernelisation algorithm by Guo and Niedermeier [23]?*

We start by answering the subquestions and then show the results.

Research Question 5a. *Does the algorithm by Bousquet et al. result in smaller kernels than the algorithm by Guo and Niedermeier?*

In most cases, both algorithms resulted in the same or approximately the same kernel size. In two of the tested cases, the `BQ-algorithm` resulted in a marginally smaller kernel. The `GN-algorithm` could compute a smaller kernel on instances from 3-SAT and CNF-SAT instances.

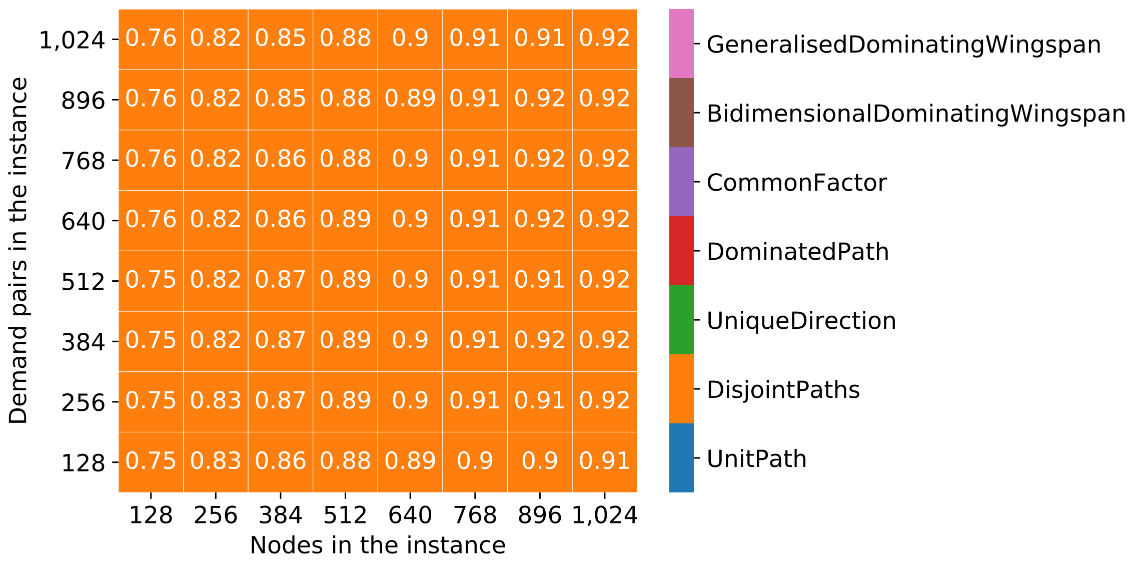


Figure 35: Dominant reduction rule in the algorithm by Bousquet *et al.* [4] on Prüfer trees with mostly long demand paths. Each cell is determined using 100 experiments. The text in each cell is the fraction of the total time spent by the dominant reduction rule.

Research Question 5b. *Does the algorithm by Bousquet et al. take less time to compute kernels than the algorithm by Guo and Niedermeier?*

The `BQ-algorithm` was (much) slower than the `GN-algorithm` in all but two of the tested cases. These two cases were `Caterpillar-Long` and `Caterpillar-Random`. The IQR of the running time of the `BQ-algorithm` was also higher across the board than that of the `GN-algorithm`.

A final answer. The `BQ-algorithm` did not always result in the smallest kernel. It did in a few cases but resulted in larger kernels in a few other cases. For most tested cases, there were no significant differences in the size of the resulting kernels.

The `BQ-algorithm` required a lot more time than the `GN-algorithm`.

With these results, we cannot say the `BQ-algorithm` is better than the `GN-algorithm`. For most cases, we would personally prefer the `GN-algorithm`, as its kernels are still small, it takes less time, and it is much easier to understand and implement.

8.6.1 Kernel Size

Let us first consider the size of the kernels that both algorithms computed. Remember the `GN-algorithm` has a claimed kernel size of $O(k^{3k})$. The `BQ-algorithm` should result in a kernel of size $O(k^6)$. It seems like the second algorithm should result in smaller kernels, but in most of our experiments, the kernels were very similar. Look, for instance, at the number of demand pairs in kernels computed on `Degree3-Random`. Figure 37 shows these results for the `GN-algorithm`, and Subfigure 22b the results for the `BQ-algorithm`. The IQR for these plots can be found in Figure 67 and Subfigure 46b respectively in Appendix C. In just one or two grid points, the colour of the contour differs.

Small differences. In almost all cases where the kernel size did differ on more points, the `BQ-algorithm` resulted in a slightly smaller kernel. Figure 20 shows the number of nodes in kernels on `Degree3-TS10` as an example. In Appendix C, we show the IQR of these plots in Figure 44.

Only in the instances created from 3-SAT and CNF-SAT instances, we see that the BQ-algorithm computed larger kernels than the GN-algorithm. Figures for these results can be found in Figure 26 and Figure 50, the latter being located in Appendix C.

Differences in the IQR. The IQR of the size of the kernels computed by both algorithms was also similar in most cases. Consider, for instance, Figure 36, where we show the IQR of the number of nodes in kernels computed on Caterpillar-Random. The IQRs of these results can be found in Appendix C in Figure 66.

In a few cases, the IQR of the results by the BQ-algorithm was better than the results obtained by the GN-algorithm. One such case, the IQR of the number of nodes in kernels on Degree3-TS10, can be seen in Figure 44 in Appendix C.

Sometimes, however, the results obtained by the GN-algorithm show a slightly lower spread. For instance, look at the spread in the number of nodes in kernels computed on Degree3-TS20 in Figure 53 in Appendix C. Figure 29 contains the contour plot with the number of nodes in the kernel for these instances.

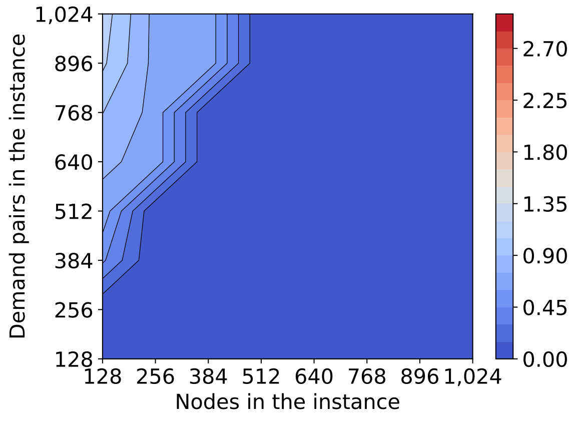
Expectation versus reality. We saw that there were not many significant differences in kernel size between the two algorithms. When the GN-algorithm resulted in a smaller kernel, the kernel was quite a bit smaller. When the BQ-algorithm resulted in a smaller kernel, the kernel was only a bit smaller.

These results are fascinating, as the presented bounds on the kernel sizes ($O(k^{3k})$ for the GN-algorithm, $O(k^6)$ for the BQ-algorithm) seem to tell us differently. The most logical reason is that the difference is not caused by the set of used reduction rules but rather by the tightness of the analyses of the kernel size.

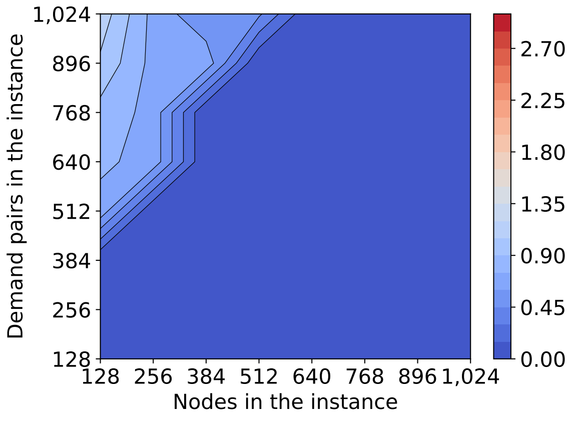
8.6.2 Running Time

We can observe more significant differences in the running time.

Consider the instances from the VERTEX COVER instances. We have already shown the running time for both algorithms in Figure 25. The IQR of these contours can be found in Figure 49 in Appendix C. We see that the GN-algorithm is way faster than the BQ-algorithm in many grid points.



(a) Guo and Niedermeier [23].



(b) Bousquet *et al.* [4].

Figure 36: Contour plots of the base-10 logarithm of the median number of nodes in kernels on instances with caterpillars and uniformly at random demand pairs, using both algorithms. Each grid point is determined using 100 experiments.

A more typical example. The instances from Vertex Cover instances are an extreme example. Most of the time, the `GN-algorithm` was faster than the `BQ-algorithm`. Consider the results obtained on `Prüfer-Random`. Subfigure 14a shows the number of operations required by the `GN-algorithm`. The results for the `BQ-algorithm` are displayed in Figure 38. The IQRs for both of these plots can be found in Subfigure 14b and Figure 68 respectively, with the latter being located in Appendix C. From the colours, we can see that the `BQ-algorithm` required more time than the `GN-algorithm`. We note that both algorithms took approximately equal time in very few cases, like on `Caterpillar-Long`.

IQR. The contour plots with the IQR show similar results. The IQR of the time required by the `GN-algorithm` was generally smaller than the IQR for the `BQ-algorithm`. This can be seen in Subfigure 14b and in Figure 68 in Appendix C again. These results were obtained on `Prüfer-Random`.

Theoretical running time. Let us take a look at the theoretical running time of both algorithms. We do this by summing the theoretical times of the reduction rules.

- `GN-algorithm`: $O(n)+O(p)+O(n^2p)+O(p^2)+O(n^2p)+O(p)+O(n+p)+O(n+p) = O(n^2p+p^2)$.
- `BQ-algorithm`: $O(p)+O(n^2p)+O(n^2p)+O(p^2)+O(n^2p^2)+O(n^3p)+O(\min(n,p)\cdot n^3p) = O(n^3p\cdot\min(n,p)+p^2)$ [22].

The sum of the theoretical running times of the reduction rules in the `BQ-algorithm` is higher than the sum for the `GN-algorithm`. This seems like an easy explanation for the results we obtained. However, if the earlier reduction rules already remove 90% of the instance, the later ones work on a much smaller instance, which influences the real world time they need. For instance, in our experiments, we found that a rule that takes $O(n^2p)$ time took way more time

[22]Assuming $p \leq n^2$ after Reduction Rule 4.2.4. If we assume the input does not contain duplicate demand pairs, the factor p^2 disappears.

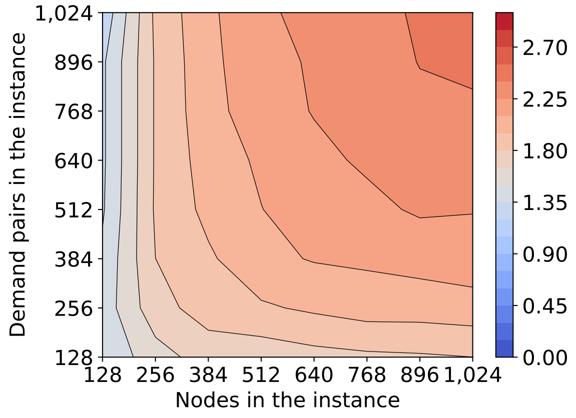


Figure 37: Contour plot of the base-10 logarithm of the median number of demand pairs in kernels computed by the algorithm by Guo and Niedermeier [23] on instances with a Degree3 tree and uniformly at random demand pairs. Each grid point is determined using 100 experiments.

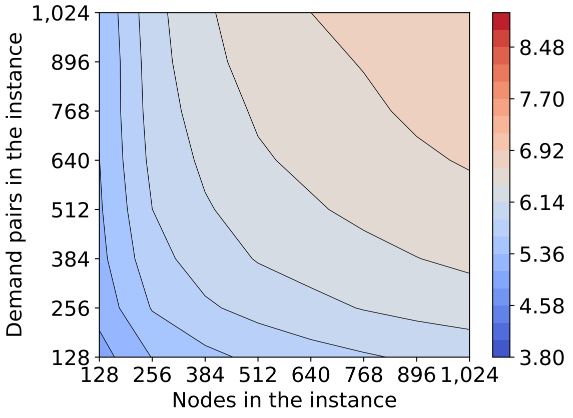


Figure 38: Contour plot of the base-10 logarithm of the median total number of operations required to compute kernels on instances with a Prüfer tree and uniformly at random demand pairs using the algorithm by Bousquet *et al.* [4]. Each grid point is determined using 100 experiments.

than the rule that takes $O(\min(n,p)\cdot n^3 p)$ time. Asymptotically, this should not hinder our analysis.

The smart ways to run rules during their later applications can also help reduce the actual required time. For the first algorithm, we used such a later smart application for the first four rules. The second algorithm's first, third, and fourth reduction rules also benefited from later smart applications. However, the second rule (Reduction Rule 4.2.2 (Disjoint Paths)) did not. This could also be a reason why there are such significant differences in the running time.

Again, we would be interested to see what happens to the running times if the Disjoint Paths reduction rule would be placed last in both algorithms.

9 Conclusion

This section summarises this research and provides concise answers to the research questions. We also mention some ideas for future research.

9.1 Summary and Concise Answers

We have experimented with two kernelisation algorithms for Multicut in Trees. The literature tells us that the first algorithm, by Guo and Niedermeier [23], computes a kernel of size $O(k^{3k})$. The second algorithm, by Bousquet *et al.* [4], results in a kernel of size $O(k^6)$.

We listed each of the reduction rules these algorithms used and explained how we implemented them. We also explained three tree types and three demand pair generation types, some of which are our own creations. Other instances we used were created from Vertex Cover,3-SAT and CNF-SAT instances.

We analysed these algorithms on different sets of instances and compared the size of the kernel they computed and the time they needed to compute this kernel. We used counters throughout the code to measure the time to ensure external factors did not influence the results.

With the results from the experiments, we answered our five research questions.

We concluded that instances with a caterpillar are easy to compute a kernel on and that instances with a Degree3 tree are difficult. Also, instances with mostly short demand paths result in the largest kernels. Finally, instances from Vertex Cover, 3-SAT and CNF-SAT instances were quite difficult, though one should think about the number of nodes and demand pairs in these instances.

We observed that in most cases, the size of the kernel grows when the number of nodes or demand pairs in the instance increases, but there were some exceptions. Increasing the size of the kernel does mean that the time required to compute the kernel increases.

From the two demand pair generation methods we introduced, we found that the technique that creates demand pairs through a solution can be improved by a lot. The other method, which creates demand paths with a certain length, is a lot more functional. This method created the most difficult instances.

We looked at which reduction rule dominated the running time of an algorithm. In almost all cases, for the algorithm by Guo and Niedermeier, this was Reduction Rule 4.1.3 (Dominated Edge). For the algorithm by Bousquet *et al.*, Reduction Rule 4.2.2 (Disjoint Paths) was the dominating rule. Looking at the total time spent by the algorithms, we concluded that the Disjoint Paths reduction rule was the overall dominating reduction rule.

Finally, we looked at the differences between the performance of both algorithms. We experienced that the algorithms computed kernels of approximately equal size in almost all tested cases. In two cases, the algorithm by Bousquet *et al.* resulted in a marginally smaller kernel. On instances from 3-SAT or CNF-SAT instances, the algorithm by Guo and Niedermeier resulted in a kernel that was quite a bit smaller. The algorithm by Bousquet *et al.* was slower than the algorithm by Guo and Niedermeier in all cases, except for two. In these two cases, the running time was approximately the same. To conclude, we cannot say that one algorithm is always better than the other.

9.2 Future Research

We explain a few ideas for future research.

The third kernelisation algorithm. There exist three kernelisation algorithms for Multicut in Trees. We have only used two of these in this research. According to the literature, the last one, by Chen *et al.* [10], computes a kernel of size $O(k^3)$.

It would be fascinating to see what this algorithm can do in practice. The reduction rules that are unique in the two tested algorithms did not contribute much to the size of the kernel. We wonder if this would also be the case in the third algorithm. If so, we do not think that the kernels it computes would be smaller than those of the first two algorithms.

Difficult instances. We have tried to design instances that would be difficult to compute a kernel on for the tested algorithms. Instances with Degree3 trees seem to be the most difficult from the ones we tested. However, the kernels on these instances are still smaller than the claimed bound.

Another interesting direction for future research is creating even more difficult instances. For instance, one could try to find out what makes an instance difficult and, with that information, create a set of instances on which the algorithms reach their claimed kernel bound.

Different values for parameter k. We have only used the optimal value for parameter k in each of our experiments. With a smaller value, the algorithms could stop the computation of the kernel earlier, and with a larger value, some reduction rules would fire less often. Still, it could be interesting to see what happens when we combine the instances we used with a value for k that is slightly smaller or bigger than the size of the optimal solution. Testing with a k that is way smaller or larger than the optimal solution size could also yield interesting results.

Defining the size of a kernel. In most literature we studied, the size of the kernel is defined only by the number of nodes in that kernel. This is technically sufficient, because we can assume that the number of demand pairs in the kernel is at most the number of nodes in the kernel squared. In our experiments, the number of demand pairs was almost always approximately equal to the number of nodes. Only when the instance contained a Degree3 tree could the number of demand pairs be noticeably higher than the number of nodes. This result could indicate that we also need to consider the number of demand pairs when determining the kernel size.

For instance, consider two kernels. Kernel A has 13 nodes and 13 demand pairs, and kernel B has 13 nodes and 19 demand pairs. Both these kernels could occur after applying the algorithm by Guo and Niedermeier [23] exhaustively. The difference in the number of demand pairs is caused by the shape of the remaining tree. If we only consider the number of nodes in these kernels, we would conclude that the kernels are equally large. However, we would say that kernel A is smaller than kernel B if we consider both the number of nodes and the number of demand pairs in the kernels.

For more theoretical research, it could be interesting to see a good measure for kernel size that encapsulates both the number of nodes and the number of demand pairs.

Improving the demand pair generation methods. The demand pair generation method that generates demand paths from a length distribution helped create difficult instances. We have also talked about its most significant limitation: we need to know the length of the path between every pair of two nodes. This information is not always readily available and can take quite some time to compute on larger instances.

Additionally, to define what a "long" path would be in a tree, we have to compute all possible paths in that tree before being able to generate demand paths of that length. In the current implementation, we have to compute the path between every pair of two nodes twice.

It would be handy if the implementation of this generation method could be improved so that it works consistently on as many graphs as possible. Ideally, this should work without having to compute the path between every pair of nodes.

How to choose realistic lengths for each path remains an open question.

The other demand pair generation method we introduced generates demand pairs through a solution of a given size. This method was not as usable as we had hoped. We talked about a giant flaw where our method of generating the first edge-disjoint demand paths can immediately lead to a very easy instance. Plus, we saw that the size of the solution should depend on the size of the instance. Improving the generation method for the first edge-disjoint demand paths, maybe iteratively, and finding some formula for the perfect solution size are both exciting topics to look at in the future.

Modifying the algorithms. Finally, one more interesting topic for future research is modifying the existing algorithms. Think, for instance, about changing the order in which an algorithm applies the reduction rules or only using a subset of the rules.

Changing the order of reduction rules could lead to differences in running time. Consider the algorithm by Guo and Niedermeier [23]. Reduction Rule 4.1.3 runs in $O(n^2p)$ time and can reduce n. The next rule, Reduction Rule 4.1.4, runs in $O(p^2)$ and can reduce p. Executing Reduction Rule 4.1.3 before Reduction Rule 4.1.4 does not mean that Reduction Rule 4.1.4 runs faster. But, executing Reduction Rule 4.1.4 before Reduction Rule 4.1.3 does mean that Reduction Rule 4.1.3 can run faster, as p can be smaller.

The dominant reduction rule can also change, which could lead to different insights into the performance of the algorithms. Consider, for instance, Reduction Rule 4.2.2, which was the dominant rule in the algorithm by Bousquet *et al.* [4] for almost all experiments. Moving this rule to later in the ordering might improve the overall running time a bit.

Using only a subset of the rules might also improve the running time by a lot, though it could come at the cost of a slightly larger kernel. Imagine if a rule performed almost no modifications to the input but still took about half of the time. Then, we could consider the algorithm to be better if we do not use that rule. Another example is leaving out a rule that performs many modifications. Maybe the other rules will perform the same modifications, or perhaps the size of the kernel would increase by a lot.

Using a combination of different rules from different algorithms could also be interesting. With this, it might be possible to find an algorithm that results in an even smaller kernel than the existing algorithms do.

9.3 Closing Words

With this research, we hope to have partially filled the gap of the missing practical research in the area of Multicut in Trees. In contrast to pure theoretical results, we hope this research provides insight into what is possible in practice. We also hope to have given ideas for other things that can be done in this area.

References

[1] F. N. Abu-Khzam, R. L. Collins, M. R. Fellows, M. A. Langston, W. H. Suters, and C. T. Symons. Kernelization algorithms for the vertex cover problem: Theory and experiments. In *Proceedings of the Sixth Workshop on Algorithm Engineering and Experiments and the First Workshop on Analytic Algorithmics and Combinatorics*, pages 62–69, New Orleans, LA, USA, 2004. SIAM. ISBN 9780898715644.

[2] E. Angriman, A. van der Grinten, M. von Looz, H. Meyerhenke, M. Nöllenburg, M. Predari, and C. Tzovas. Guidelines for experimental algorithmics in network analysis. *arXiv preprint arXiv:1904.04690*, Mar. 2019.

[3] S. Arora and B. Barak. *Computational complexity: a modern approach.* Cambridge University Press, 2009.

[4] N. Bousquet, J. Daligault, S. Thomassé, and A. Yeo. A polynomial kernel for multicut in trees. *arXiv preprint arXiv:0902.1047*, 2009.

[5] N. Bousquet, J. Daligault, and S. Thomassé. Multicut is fpt. *arXiv preprint arXiv:1010.5197*, 2010.

[6] N. Bousquet, J. Daligault, and S. Thomassé. Multicut is fpt. *SIAM Journal on Computing*, 47(1):166–207, 2018. ISSN 0097-5397.

[7] J. F. Buss and J. Goldsmith. Nondeterminism within p^. *SIAM Journal on Computing*, 22 (3):560–572, June 1993. ISSN 0097-5397.

[8] G. Călinescu, C. G. Fernandes, and B. Reed. Multicuts in unweighted graphs and digraphs with bounded degree and bounded tree-width. *Journal of Algorithms*, 48(2):333–359, Sept. 2003. ISSN 0196-6774.

[9] J. Chen, I. A. Kanj, and W. Jia. Vertex cover: further observations and further improvements. *Journal of Algorithms*, 41(2):280–301, 2001.

[10] J. Chen, J.-H. Fan, I. Kanj, Y. Liu, and F. Zhang. Multicut in trees viewed through the eyes of vertex cover. *Journal of Computer and System Sciences*, 78(5):1637–1650, Sept. 2012. ISSN 0022-0000.

[11] M.-C. Costa, L. Létocart, and F. Roupin. Minimal multicut and maximal integer multiflow: a survey. *European Journal of Operational Research*, 162(1):55–69, 2005.

[12] E. Dahlhaus, D. S. Johnson, C. H. Papadimitriou, P. D. Seymour, and M. Yannakakis. The complexity of multiway cuts. In *Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, pages 241–251, New York, NY, USA, 1992. Association for Computing Machinery. ISBN 0897915119.

[13] E. Dahlhaus, D. S. Johnson, C. H. Papadimitriou, P. D. Seymour, and M. Yannakakis. The complexity of multiterminal cuts. *SIAM Journal on Computing*, 23(4):864–894, 1994. ISSN 0097-5397.

[14] R. G. Downey and M. R. Fellows. *Parameterized complexity.* Springer Science & Business Media, 2012.

[15] J. Edmonds. Paths, trees, and flowers. *Canadian Journal of mathematics*, 17:449–467, 1965.

[16] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19(2):248–264, Apr. 1972. ISSN 0004-5411.

[17] P. Erdős and A. Rényi. On random graphs I. *Publicationes Mathematicae*, 6:290–297, 1959.

[18] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

[19] L. R. Ford Jr and D. R. Fulkerson. *Flows in networks*, volume 54. Princeton university press, 2015.

[20] N. Garg, V. V. Vazirani, and M. Yannakakis. Approximate max-flow min-(multi) cut theorems and their applications. *SIAM Journal on Computing*, 25(2):235–251, May 1996. ISSN 0097-5397.

[21] N. Garg, V. V. Vazirani, and M. Yannakakis. Primal-dual approximation algorithms for integral flow and multicut in trees. *Algorithmica*, 18(1):3–20, May 1997.

[22] G. Gottlob and S. T. Lee. A logical approach to multicut problems. *Information Processing Letters*, 103(4):136–141, Aug. 2007. ISSN 0020-0190.

[23] J. Guo and R. Niedermeier. Fixed-parameter tractability and data reduction for multicut in trees. *Networks: An International Journal*, 46(3):124–135, Oct. 2005.

[24] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2021. URL `https://www.gurobi.com`.

[25] H. H. Hoos and T. Stützle. Satlib: An online resource for research on sat. *Sat*, 2000: 283–292, 2000.

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

[27] T. C. Hu. Integer programming and network flows. Technical report, University of Wisconsin-Madison, department of computer sciences, 1969.

[28] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007. doi: 10.1109/MCSE.2007.55.

[29] A. Itai. Two-commodity flow. *Journal of the ACM (JACM)*, 25(4):596–611, Oct. 1978. ISSN 0004-5411.

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

[31] I. Kanj, G. Lin, T. Liu, W. Tong, G. Xia, J. Xu, B. Yang, F. Zhang, P. Zhang, and B. Zhu. Improved parameterized and exact algorithms for cut problems on trees. *Theoretical Computer Science*, 607:455–470, Nov. 2015. ISSN 0304-3975.

[32] S. Khot and O. Regev. Vertex cover might be hard to approximate to within 2- ε. *Journal of Computer and System Sciences*, 74(3):335–349, May 2008. ISSN 0022-0000.

[33] V. King, S. Rao, and R. Tarjan. A faster deterministic maximum flow algorithm. *Journal of Algorithms*, 17(3):447 – 474, Nov. 1994. ISSN 0196-6774.

[34] D. Marx. Parameterized graph separation problems. *Theoretical Computer Science*, 351(3): 394–406, Feb. 2006. ISSN 0304-3975.

[35] D. Marx and I. Razgon. Fixed-parameter tractability of multicut parameterized by the size of the cutset. *arXiv*, pages arXiv–1010, 2010.

[36] D. Marx and I. Razgon. Fixed-parameter tractability of multicut parameterized by the size of the cutset. *SIAM Journal on Computing*, 43(2):355–388, 2014. ISSN 0097-5397.

[37] C. C. McGeoch. *A guide to experimental algorithmics*. Cambridge University Press, 2012.

[38] S. Micali and V. Vazirani. An O(sqrt(|V|) |E|) algorithm for finding maximum matching in general graphs. In *21st Annual Symposium on Foundations of Computer Science (sfcs 1980)*, pages 17–27, Oct. 1980. doi: 10.1109/SFCS.1980.12.

[39] Microsoft. Collection<t> class, n.d.. URL `https://docs.microsoft.com/en-us/dotnet/api/system.collections.objectmodel.collection-1?view=net-5.0`.

[40] Microsoft. Dictionary<tkey,tvalue> class, n.d.. URL `https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.dictionary-2?view=net-5.0`.

[41] Microsoft. Hashset<t> class, n.d.. URL `https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.hashset-1?view=net-5.0`.

[42] Microsoft. Linkedlist<t> class, n.d.. URL `https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.linkedlist-1?view=net-5.0`.

[43] Microsoft. List<t> class, n.d.. URL `https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1?view=net-5.0`.

[44] Microsoft. Ordereddictionary class, n.d.. URL `https://docs.microsoft.com/en-us/dotnet/api/system.collections.specialized.ordereddictionary?view=net-5.0`.

[45] Microsoft. Queue<t> class, n.d.. URL `https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.queue-1?view=net-5.0`.

[46] R. Niedermeier and P. Rossmanith. A general method to speed up fixed-parameter-tractable algorithms. *Information Processing Letters*, 73(3-4):125–129, 2000.

[47] J. B. Orlin. Max flows in O(nm) time, or better. In *Proceedings of the Forty-Fifth Annual ACM Symposium on Theory of Computing*, STOC '13, page 765–774, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450320290.

[48] R. Pichler, S. Rümmele, and S. Woltran. Multicut algorithms via tree decompositions. In *International Conference on Algorithms and Complexity*, pages 167–179. Springer, 2010. ISBN 9783642130724.

[49] H. Prüfer. Neuer beweis eines satzes über permutationen. *Arch. Math. Phys*, 27(1918): 742–744, 1918.

[50] Randomness and Integrity Services Ltd. Random integer set generator, n.d. URL `https://www.random.org/integer-sets/`.

[51] A. Sharma and J. Vondrák. Multiway cut, pairwise realizable distributions, and descending thresholds. In *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*, pages 724–733, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450327107.

[52] G. Szekeres. Distribution of labelled trees by diameter. In *Combinatorial Mathematics X*, pages 392–397. Springer, 1983.

[53] D. B. West et al. *Introduction to graph theory*, volume 2. Prentice hall Upper Saddle River, NJ, 1996.

A List of Definitions and Abbreviations

This part of the appendix contains definitions and abbreviations used throughout this thesis.

Definition	**Explanation**	**Page**
$A(v)$	Let v be a bad L_2-leaf of a caterpillar (in a tree) C. Remove the neighbour of v and all its leaves from the graph. The two connected components that are left are called $A(v)$ and $B(v)$.	13
$a(v)$	Let v be a bad L_2-leaf of a caterpillar (in a tree) C. Create $A(v)$ and $B(v)$. The extremity of C in $A(v)$ is called $a(v)$. If this extremity does not exist, the neighbour of v is $a(v)$.	13
$B(v)$	Let v be a bad L_2-leaf of a caterpillar (in a tree) C. Remove the neighbour of v and all its leaves from the graph. The two connected components that are left are called $A(v)$ and $B(v)$.	13
$b(v)$	Let v be a bad L_2-leaf of a caterpillar (in a tree) C. Create $A(v)$ and $B(v)$. The extremity of C in $B(v)$ is called $b(v)$. If this extremity does not exist, the neighbour of v is $b(v)$.	13
Backbone	The path of I_2-nodes in a caterpillar.	12
Bad leaf	A leaf v, such that there exists no demand pair between v and another leaf u, where u is connected to the neighbour of v.	12
`BQ-algorithm`	Abbreviation for the algorithm by Bousquet *et al.* [4].	51
Caterpillar (1)	By itself: A tree consisting of a path of I_2-nodes with an I_1-node at each end, and all leaves connected to all these nodes.	12
Caterpillar (2)	In a tree: A maximal induced subgraph of the tree consisting only of I_1-nodes, I_2-nodes, L_1-leaves and L_2-leaves.	12
`Caterpillar`	Abbreviation for instances with a caterpillar.	51
Caterpillar component	A maximal induced subgraph of a tree consisting only of I_2-nodes and L_2-leaves.	12
`Caterpillar-Long`	Abbreviation for instances with a caterpillar and mostly long demand paths.	51
`Caterpillar-Random`	Abbreviation for instances with a caterpillar and uniformly at random demand pairs.	51
`Caterpillar-Short`	Abbreviation for instances with a caterpillar and mostly short demand paths.	51
`Caterpillar-TS`	Abbreviation for instances with a caterpillar and demand pairs through a solution.	51

Definition	**Explanation**	**Page**
`Caterpillar-TS`x	Abbreviation for instances with a caterpillar and demand pairs through a solution of size x.	51
Closest P-neighbour	A closest P-neighbour of a node v is the other endpoint of a minimal demand path starting at v.	13
Contract	If we contract an edge between nodes u and v, we replace u and v in the tree by a new node w that is connected to all neighbours of u and v, except for u and v themselves.	12
Cover	Let v be a bad L_2-leaf of a caterpillar (in a tree) C. Let P_1 and P_2 be two minimal demand paths that start at v, such that P_1 and P_2 use a different edge between two internal nodes connected to the neighbour of v. Leaf v covers C if P_1 and P_2 together go over the entire backbone of C.	13
Cut	When we cut an edge e, we remove all demand paths that go over e, decrease k by one, and contract e.	12
`Degree3`	Abbreviation for instances with a Degree3 tree.	51
Degree3 tree	A tree that looks like a binary tree, except that the root node has three neighbours.	36
`Degree3-Long`	Abbreviation for instances with a Degree3 tree and mostly long demand paths.	51
`Degree3-Random`	Abbreviation for instances with a Degree3 tree and uniformly at random demand pairs.	51
`Degree3-Short`	Abbreviation for instances with a Degree3 tree and mostly short demand paths.	51
`Degree3-TS`	Abbreviation for instances with a Degree3 tree and demand pairs through a solution.	51
`Degree3-TS`x	Abbreviation for instances with a Degree3 tree and demand pairs through a solution of size x.	51
Demand pair	A pair of distinct nodes in T that needs to be separated.	11
Demand path	The path between the two nodes of a demand pair.	11
Destroyed	A demand path is destroyed when its corresponding demand pair is separated.	11
Dominate	A wingspan W dominates a demand pair between nodes u and v if both u and v belong to the subcaterpillar of W.	13
E	The set of edges in a MULTICUT IN TREES instance.	11
EMC	Abbreviation of EDGE MULTICUT.	10
Extremity	The extremities of a caterpillar C in a tree are the two internal nodes in C that would become leaves if all leaves in the tree would be removed.	12

Definition	**Explanation**	**Page**
Fixed-parameter tractable	We say a problem is fixed-parameter tractable when, given the input I for the problem and a parameter $k \in \mathbb{N}_0$, the problem can be solved in $O(f(k) \cdot g(\lvert I \rvert))$ time, for any computable function f, and a polynomial function g.	11
FPT	Abbreviation of fixed-parameter tractable.	11
`GN-algorithm`	Abbreviation for the algorithm by Guo and Niedermeier [23].	51
Good leaf	A leaf v, such that there exists a demand pair between v and another leaf u, where u is connected to the neighbour of v.	12
I_1-node	Internal node with at most one neighbour that is an internal node.	12
I_2-node	Internal node with at precisely two neighbours that are internal nodes.	12
I_3-node	Internal node with at least three neighbours that are internal nodes.	12
Inner node	I_2-node without leaves.	12
Internal length	The internal length of a demand path is the length of its internal path.	13
Internal node	A node in a tree with more than one neighbour.	12
Internal path	The internal path of a demand path is the subset of the demand path consisting only of edges between two internal nodes.	13
Interquartile range	For a set of data: $Q_3 - Q_1$.	52
IQR	Abbreviation for interquartile range.	52
k	The maximum number of edges we are allowed to remove from the tree to separate all demand pairs.	11
Kernel	A generally smaller input I' for a problem computed from an input I such that the problem can be solved on I' if and only if it can be solved on I.	11
Kernelisation	The process of computing a kernel.	11
L_1-leaf	Leaf connected to an I_1-node.	12
L_2-leaf	Leaf connected to an I_2-node.	12
L_3-leaf	Leaf connected to an I_3-node.	12
Leaf	A node in a tree with precisely one neighbour.	12
`Long`	Abbreviation for instances with mostly long demand paths.	51

Definition	**Explanation**	**Page**
Minimal demand path	A demand path starting at a node v is a minimal demand path if its internal path is not a superset of the internal path of any other demand path starting at v.	13
n	The number of nodes in a MULTICUT IN TREES instance.	11
P	The set with demand pairs in a MULTICUT IN TREES instance.	11
p	The number of demand pairs in a MULTICUT IN TREES instance.	11
`Prüfer`	Abbreviation for instances with a Prüfer tree.	51
Prüfer tree	A tree generated using a Prüfer sequence.	34
`Prüfer-Long`	Abbreviation for instances with a Prüfer tree and mostly long demand paths.	51
`Prüfer-Random`	Abbreviation for instances with a Prüfer tree and uniformly at random demand pairs.	51
`Prüfer-Short`	Abbreviation for instances with a Prüfer tree and mostly short demand paths.	51
`Prüfer-TS`	Abbreviation for instances with a Prüfer tree and demand pairs through a solution.	51
`Prüfer-TS`x	Abbreviation for instances with a Prüfer tree and demand pairs through a solution of size x.	51
Q_1	The first quartile of a set of data.	52
Q_2	The second quartile or median of a set of data.	52
Q_3	The third quartile of a set of data.	52
`Random`	Abbreviation for instances with uniformly at random demand pairs.	51
Reduced	A reduced input for a problem is also called a kernel.	11
Reduction rules	A set instructions that run in polynomial time used to compute a kernel.	11
RVMC	Abbreviation of RESTRICTED VERTEX MULTICUT.	10
Same direction	If zero or one demand paths start at a node v, its demand paths go in the same direction. If two or more demand paths start at a leaf u, and the second edge (from u) of all demand paths starting at u is the same, u's demand paths go in the same direction. If two or more demand paths start at an internal node w, and the first edge (from w) of all demand paths starting at w is the same, w's demand paths go in the same direction.	12
Separated	A demand pair between nodes u and v is separated if there exists no path between u and v.	11

Definition	**Explanation**	**Page**
`Short`	Abbreviation for instances with mostly short demand paths.	51
Subcaterpillar	The subcaterpillar of a wingspan W consists of all nodes in W and all leaves connected to those nodes.	13
T	The tree in a MULTICUT IN TREES instance.	11
`TS`	Abbreviation for instances with demand pairs generated through a solution.	51
`TS`x	Abbreviation for instances with demand pairs generated through a solution of size x.	51
UVMC	Abbreviation of UNRESTRICTED VERTEX MULTICUT.	10
V	The set of nodes in a MULTICUT IN TREES instance.	11
Wingspan	The wingspan of a bad L_2-leaf v is the union of internal nodes of the minimal demand paths P_1 and P_2 that start at v, such that P_1 and P_2 use a different edge between two internal nodes connected to the neighbour of v.	13

B Number of Operations Counted for Methods in the `Counted` Data Structures

This section of the appendix contains tables with the number of operations that are counted in the `Counted` data structures explained in Subsection 7.1.

`CountedList`	
Method	**Operations**
`[ ]`	1.
`Add`	1.
`AddRange`	j: The number of elements to be added.
`Clear`	n: The number of elements in the `CountedList`.
`Contains`	i: The index of the requested element, or n if it is not part of the `CountedList`.
`Count`	1.
`Insert`	$n - i$: The number of elements in the `CountedList` minus the index where an element is inserted.
`Remove`	n: The number of elements in the `CountedList`.
`RemoveAt`	$n - i$: The number of elements in the `CountedList` minus the index where an element is removed.

Table 4: The number of operations counted for the methods in a `CountedList`.

`CountedDictionary`	
Method	**Operations**
`[ ]`	1.
`Add`	1.
`Clear`	n: the number of elements in the `CountedDictionary`.
`ContainsKey`	1.
`Count`	1.
`GetKeys`	1. Resulting `CountedEnumerable` is also counted.
`GetValues`	1. Resulting `CountedEnumerable` is also counted.
`Remove`	1.
`TryGetValue`	1.

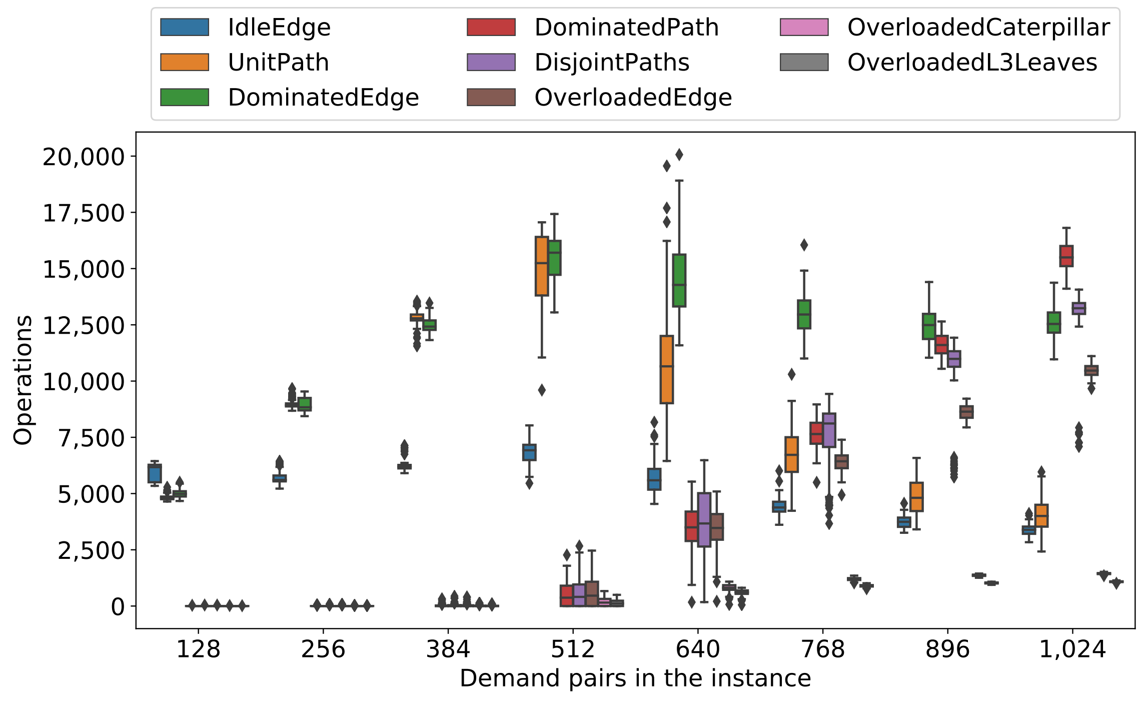
Table 5: The number of operations counted for the methods in a `CountedDictionary`.

CountedCollection	
Method	**Operations**
`Add`	1.
`ChangeElement`	1.
`Clear`	n: the number of elements in the `CountedCollection`.
`Contains`	1.
`Count`	1, or n: 1 if the number of elements in the `CountedCollection` is queried, n if a predicate is used to count the number of elements that fit a certain condition.
`ElementBeforeAndAfter`	2.
`First`	1, i, or n: 1 if the first element is queried, i if we want the first element that fits a predicate, where i is the index of that element, or n when we use a predicate but no fitting element is present.
`FirstOrDefault`	1, i, or n: 1 if the first element is queried, i if we want the first element that fits a predicate, where i is the index of that element, or n when we use a predicate but no fitting element is present.
`Last`	1. This menthod has no predicate options.
`Remove`	1.
`RemoveFromEnd`	i: the number of elements to be removed.
`RemoveFromEndWhile`	$2i + 1$: $i + 1$ elements for which we check a condition, and i elements that will be removed. Removing the elements does not update the counter again.
`RemoveFromStart`	i: the number of elements to be removed.
`RemoveFromStartWhile`	$2i + 1$: $i + 1$ elements for which we check a condition, and i elements that will be removed. Removing the elements does not update the counter again.

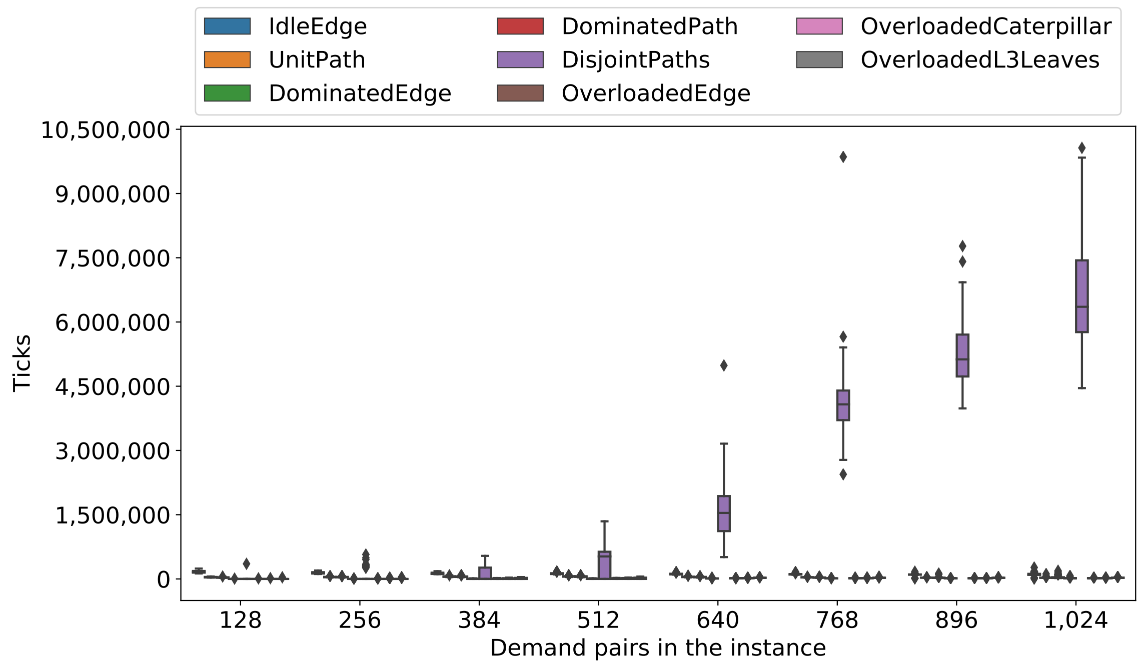
Table 6: The number of operations counted for the methods in a `CountedCollection`.

C Additional Results

This part of the appendix contains additional figures.



(a) The number of operations.



(b) The number of ticks.

Figure 39: Box plots of the number of operations and ticks per reduction rule in the algorithm by Guo and Niedermeier [23] to compute kernels on instances with 385 nodes generated from Vertex Cover instances. For each number of demand pairs, there were 100 experiments.

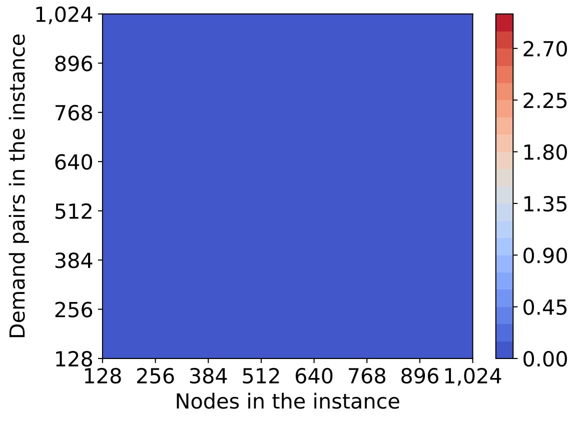


Figure 40: Contour plot of the base-10 logarithm of the median number of nodes in kernels computed by the algorithm by Guo and Niedermeier [23] on Prüfer trees with demand pairs through a solution of size 10. Each grid point is determined using 50 experiments.

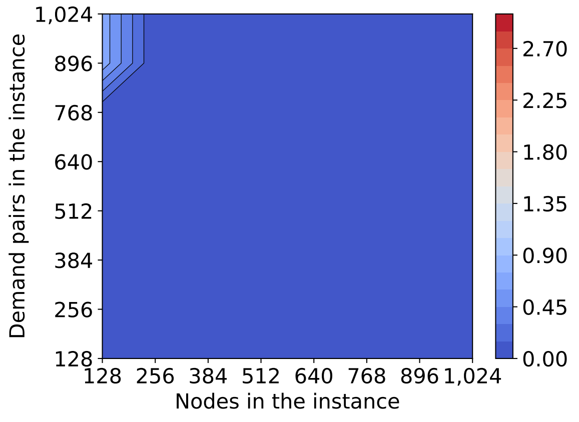


Figure 41: Contour plot of the base-10 logarithm of the median number of nodes in kernels computed by the algorithm by Bousquet *et al.* [4] on caterpillars with mostly long demand paths. Each grid point is determined using 100 experiments.

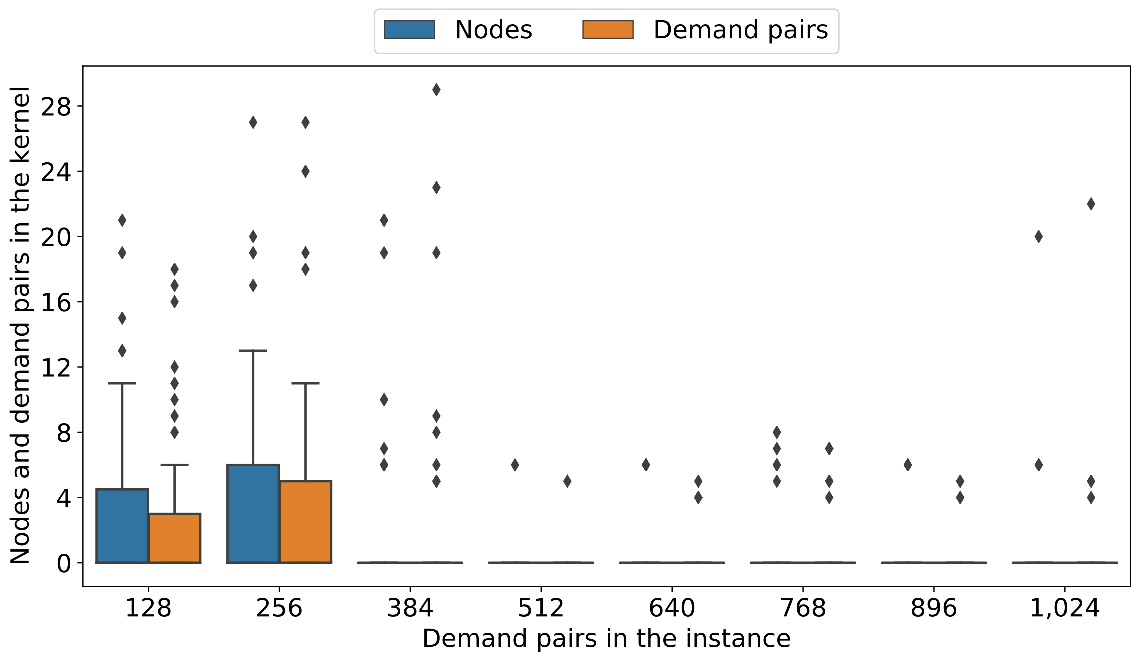


Figure 42: Box plot showing the remaining number of nodes and demand pairs in kernels computed by the algorithm by Guo and Niedermeier [23] on Prüfer trees with 896 nodes and demand pairs through a solution of size 10. For each number of demand pairs, there were 50 experiments.

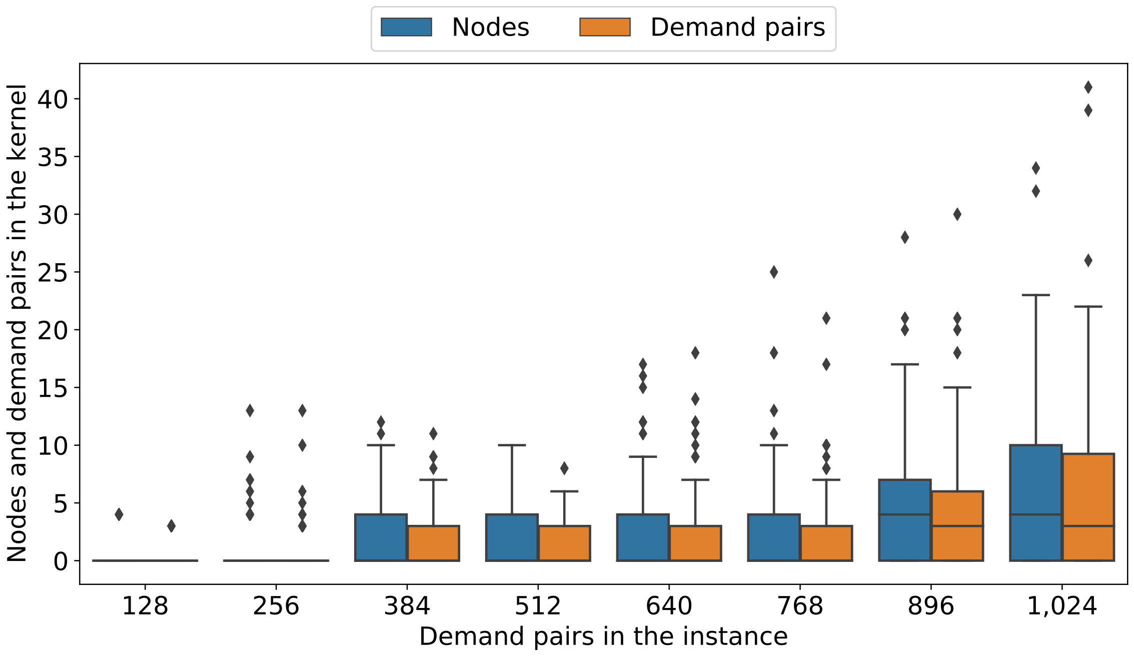
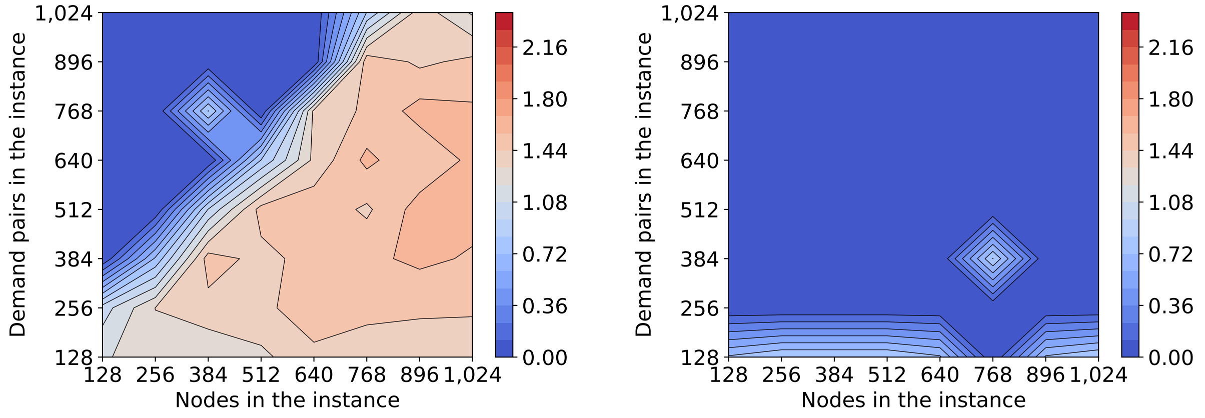


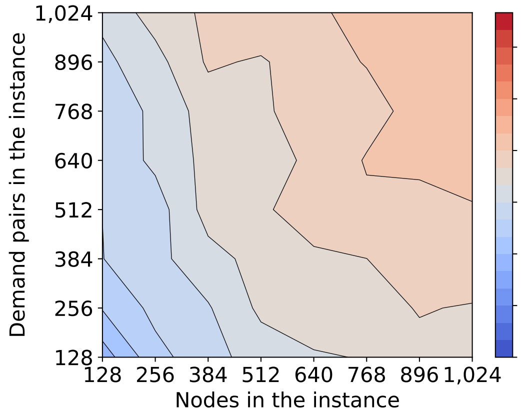
Figure 43: Box plot showing the remaining number of nodes and demand pairs in kernels computed by the algorithm by Bousquet *et al.* [4] on caterpillars with 128 nodes and demand pairs through a solution of size 10. For each number of demand pairs, there were 100 experiments.



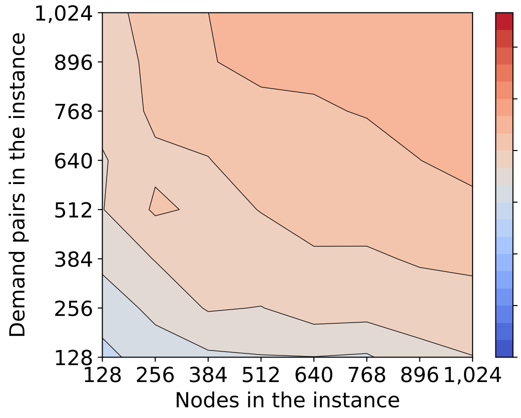
(a) Guo and Niedermeier [23].

(b) Bousquet *et al.* [4].

Figure 44: Contour plots of the base-10 logarithm of the IQR of the number of nodes in kernels computed on Degree3 trees with demand pairs through a solution of size 10, using both algorithms. Each grid point is determined using 50 experiments.

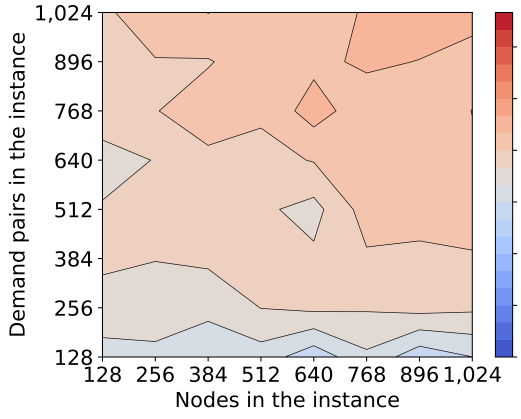


(a) Instances with caterpillars and demand pairs generated through a solution of size 100.

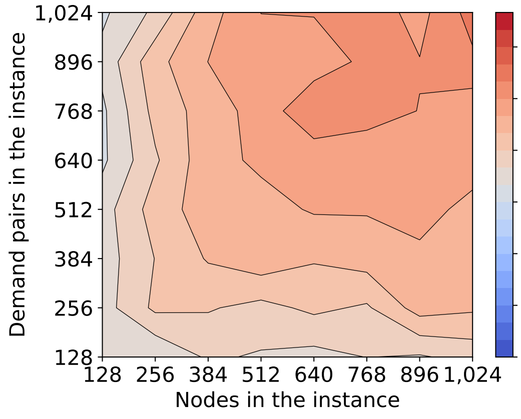


(b) Instances with Prüfer trees and demand pairs generated through a solution of size 5.

Figure 45: Contour plots of the base-10 logarithm of the IQR of the total number of operations needed to compute kernels using the algorithm by Bousquet *et al.* [4] on two different types of instances. Each grid point is determined using 50 experiments.

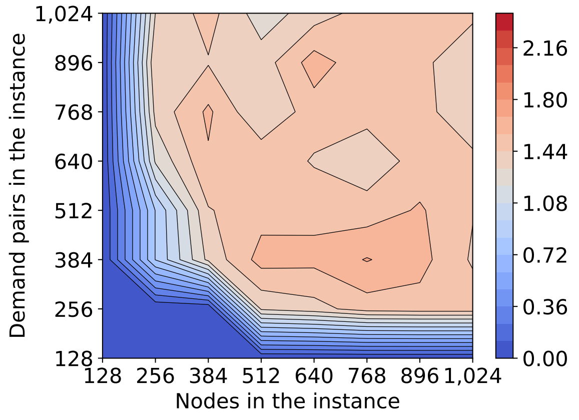


(a) Mostly short demand paths.

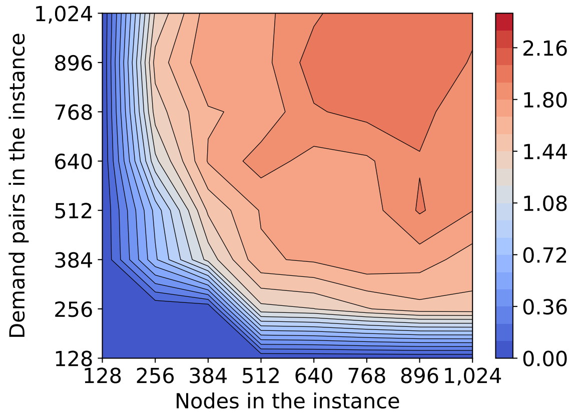


(b) Uniformly at random demand pairs.

Figure 46: Contour plots of the base-10 logarithm of the IQR of the number of demand pairs in kernels computed by the algorithm by Bousquet *et al.* [4] on Degree3 trees, for two different types of demand pairs. Each grid point is determined using 100 experiments.

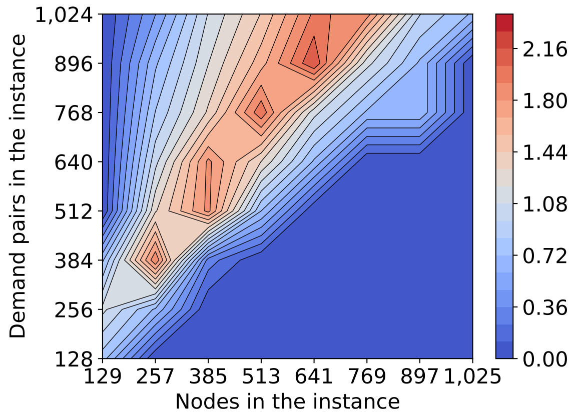


(a) IQR of the number of nodes in the kernels.

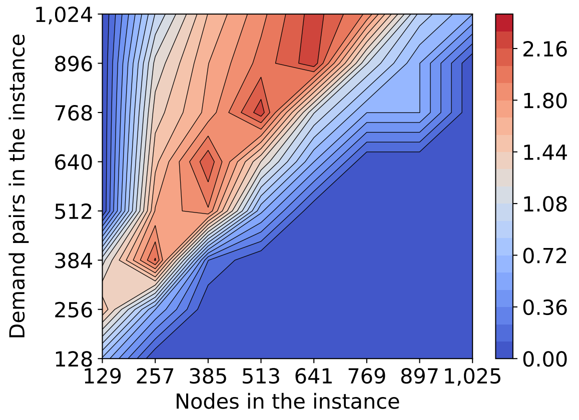


(b) IQR of the number of demand pairs in the kernels.

Figure 47: Contour plots of the base-10 logarithm of the IQR of the number of nodes and demand pairs in kernels computed by the algorithm by Bousquet *et al.* [4] on Degree3 trees with demand pairs through a solution of size 50. Each grid point is determined using 50 experiments.

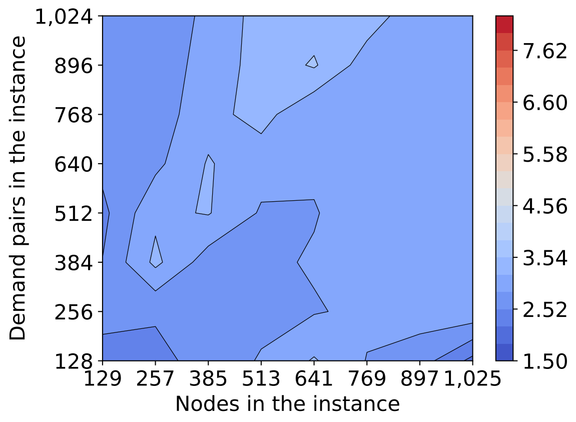


(a) IQR of the number of nodes in the kernels.

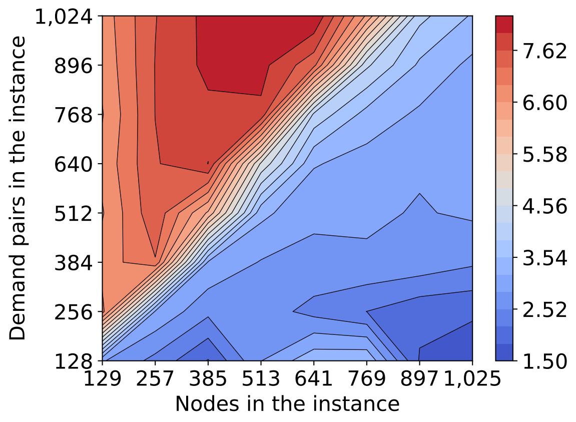


(b) IQR of the number of demand pairs in the kernels.

Figure 48: Contour plots of the base-10 logarithm of the IQR of the number of nodes and demand pairs in kernels computed by the algorithm by Bousquet *et al.* [4] on instances generated from Vertex Cover instances. Each grid point is determined using 100 experiments.

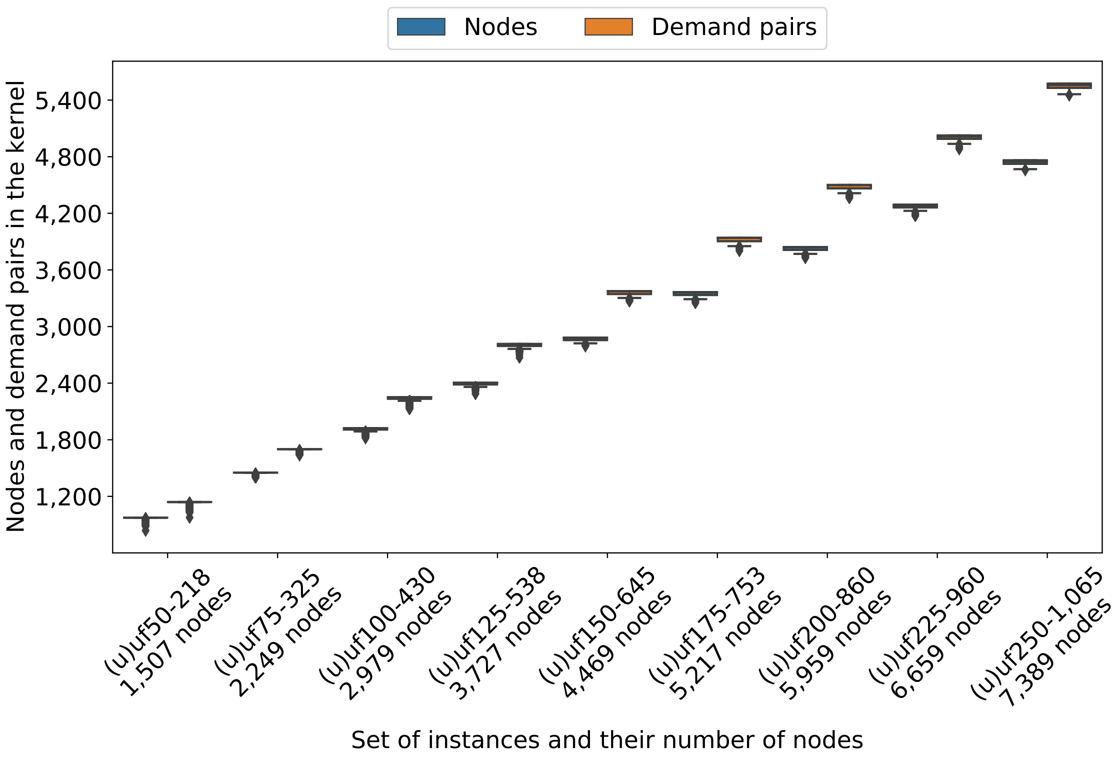


(a) Guo and Niedermeier [23].

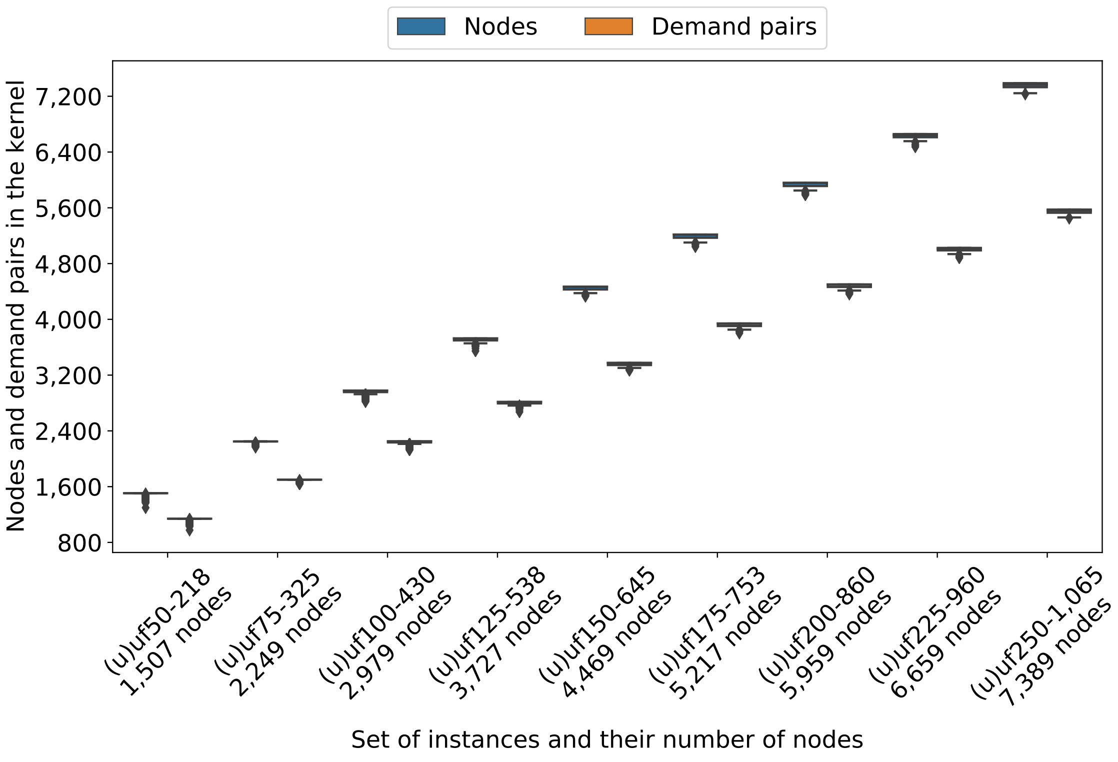


(b) Bousquet *et al.* [4].

Figure 49: Contour plots of the base-10 logarithm of the IQR of the number of operations needed to compute kernels on instances generated from Vertex Cover instances, using both algorithms. Each grid point is determined using 100 experiments.



(a) Guo and Niedermeier [23].



(b) Bousquet *et al.* [4].

Figure 50: Box plots of the number of nodes and demand pairs left in kernels computed on the 3-SAT instances, using both algorithms. We explained the instance categories displayed here in Table 2.

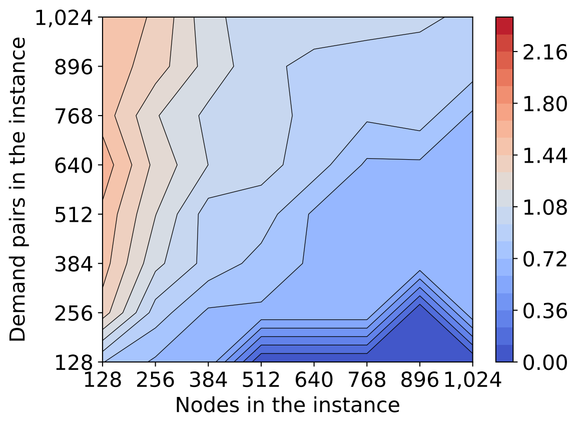


Figure 51: Contour plots of the base-10 logarithm of the IQR of the number of nodes in kernels computed by the algorithm by Guo and Niedermeier [23] on instances with a caterpillar and mostly short demand paths. Each grid point is determined using 100 experiments.

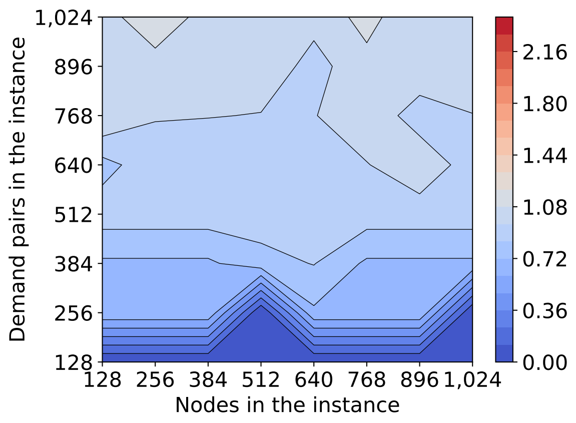
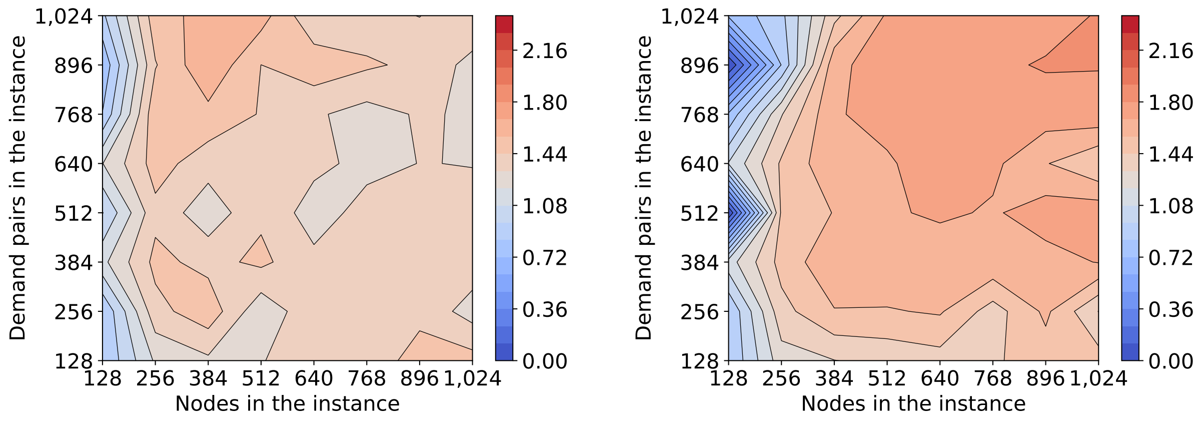


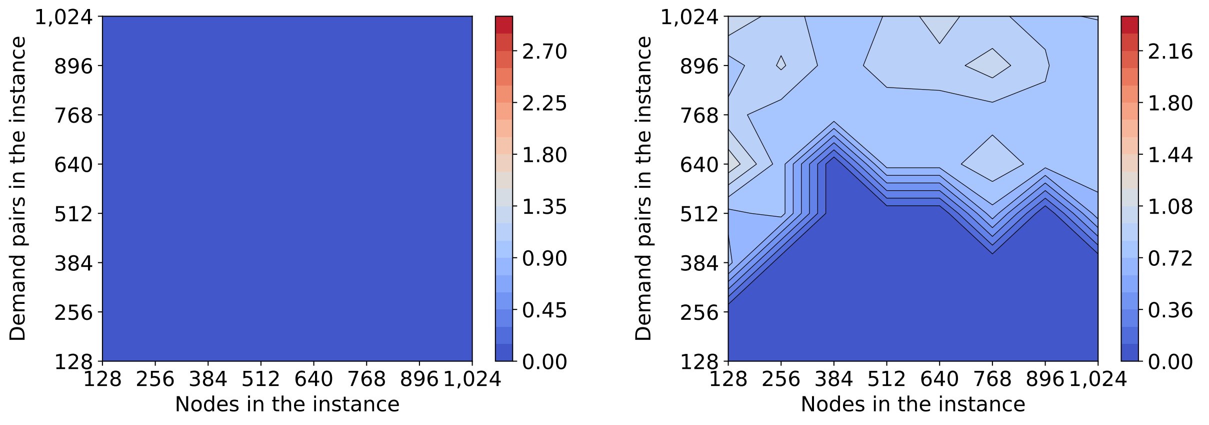
Figure 52: Contour plots of the base-10 logarithm of the IQR of the number of nodes in kernels computed by the algorithm by Guo and Niedermeier [23] on instances with a Prüfer tree and mostly long demand paths. Each grid point is determined using 100 experiments.



(a) Guo and Niedermeier [23].

(b) Bousquet *et al.* [4].

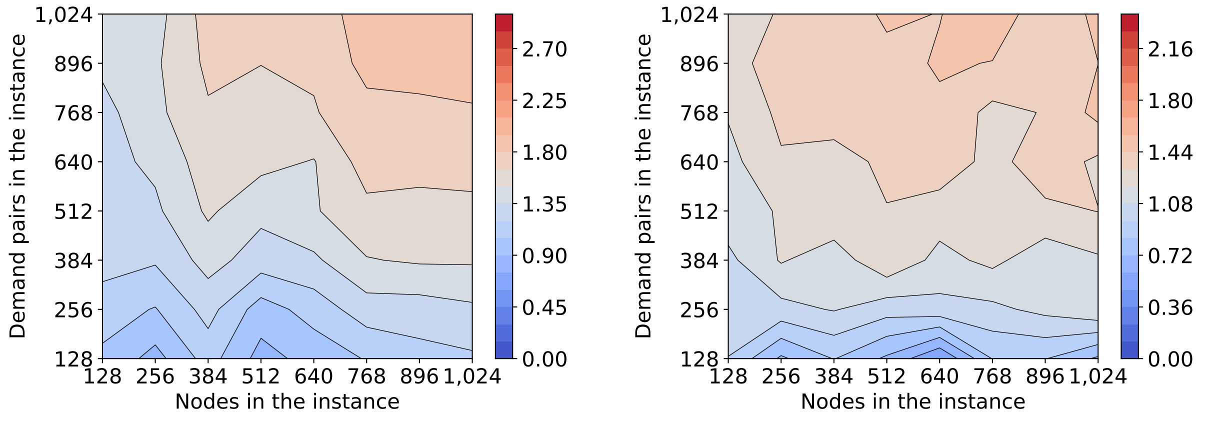
Figure 53: Contour plots of the base-10 logarithm of the IQR of the number of nodes in kernels computed on instances with a Degree3 tree and demand pairs going through a solution of size 20, using both algorithms. Each grid point is determined using 50 experiments.



(a) Base-10 logarithm of the number of nodes in the kernels.

(b) Base-10 logarithm of the IQR of the number of nodes in the kernels.

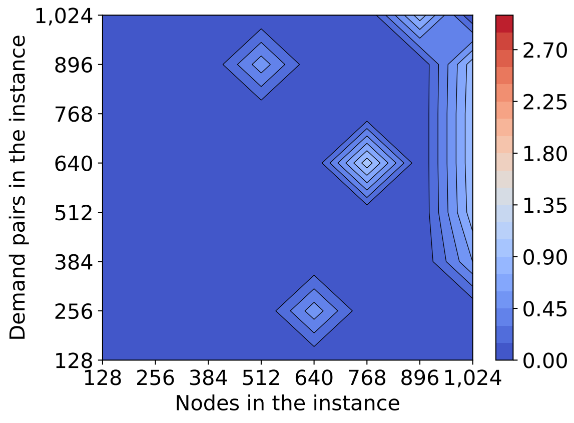
Figure 54: Contour plots of the number of nodes in kernels computed by the algorithm by Bousquet *et al.* [4] on caterpillars with demand pairs through a solution of size 50. Each grid point is determined using 50 experiments.



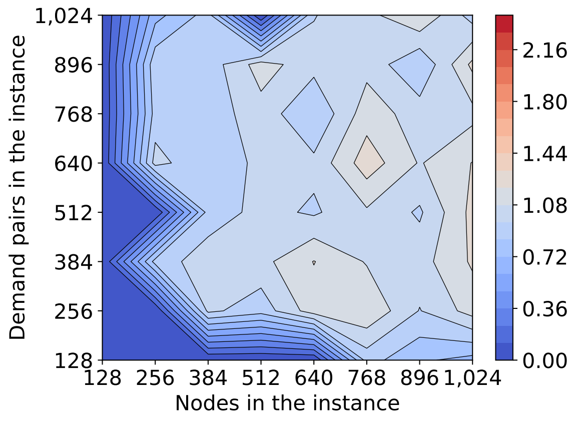
(a) Base-10 logarithm of the median number of nodes in the kernels.

(b) Base-10 logarithm of the IQR of the number of nodes in the kernels.

Figure 55: Contour plots of the number of nodes in kernels computed by the algorithm by Guo and Niedermeier [23] on Degree3 trees with mostly long demand paths. Each grid point is determined using 100 experiments.

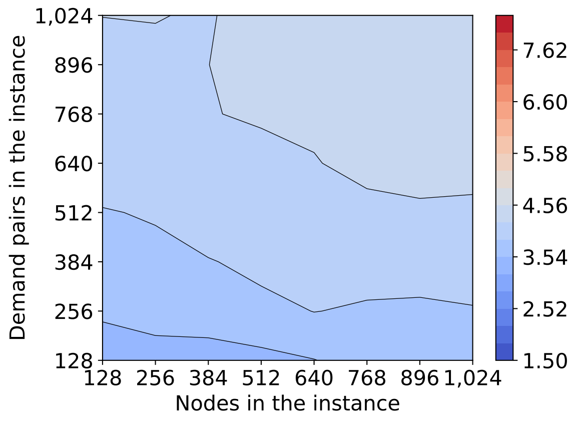


(a) Base-10 logarithm of the median number of nodes in the kernels.

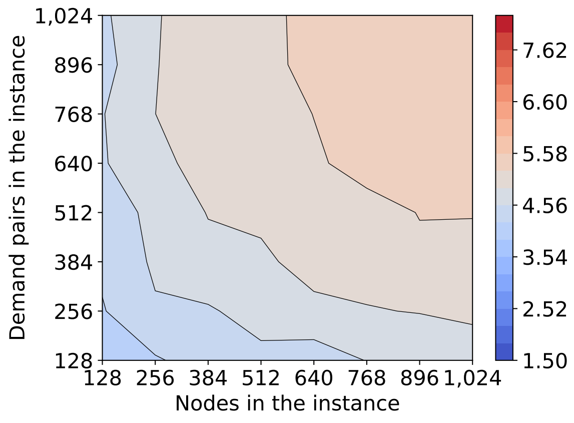


(b) Base-10 logarithm of the IQR of the number of nodes in the kernels.

Figure 56: Contour plots of the number of nodes in kernels computed by the algorithm by Guo and Niedermeier [23] on Prüfer trees with demand pairs going through a solution of size 20. Each grid point is determined using 50 experiments.



(a) Mostly short demand paths.



(b) Mostly long demand paths.

Figure 57: Contour plots of the base-10 logarithm of the IQR of the total number of operations required to compute kernels on instances with Prüfer trees for two demand pair generation methods, using the algorithm by Guo and Niedermeier [23]. Each grid point is determined using 100 experiments.

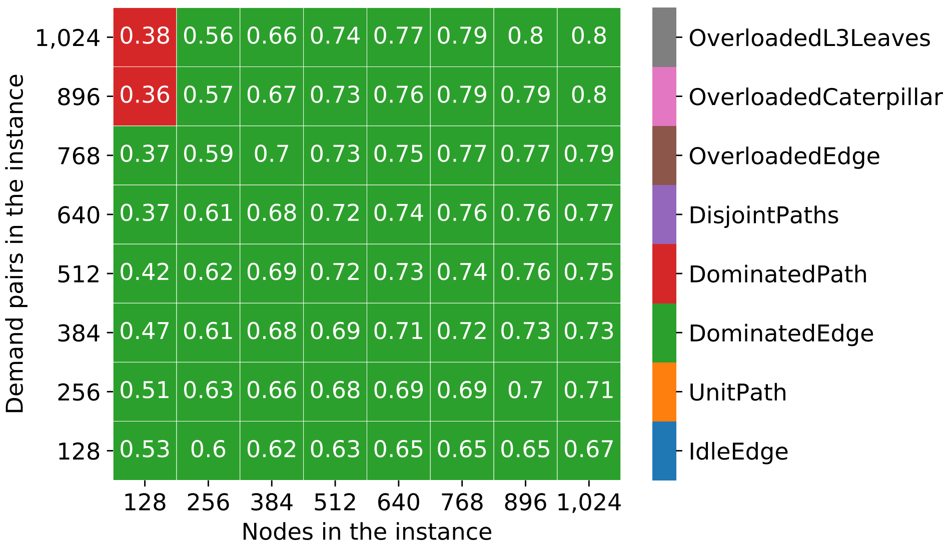


Figure 58: Dominant reduction rule in the algorithm by Guo and Niedermeier [23] on caterpillars with mostly short demand paths. Each cell is determined using 100 experiments. The text in each cell is the fraction of the total time spent by the dominant reduction rule.

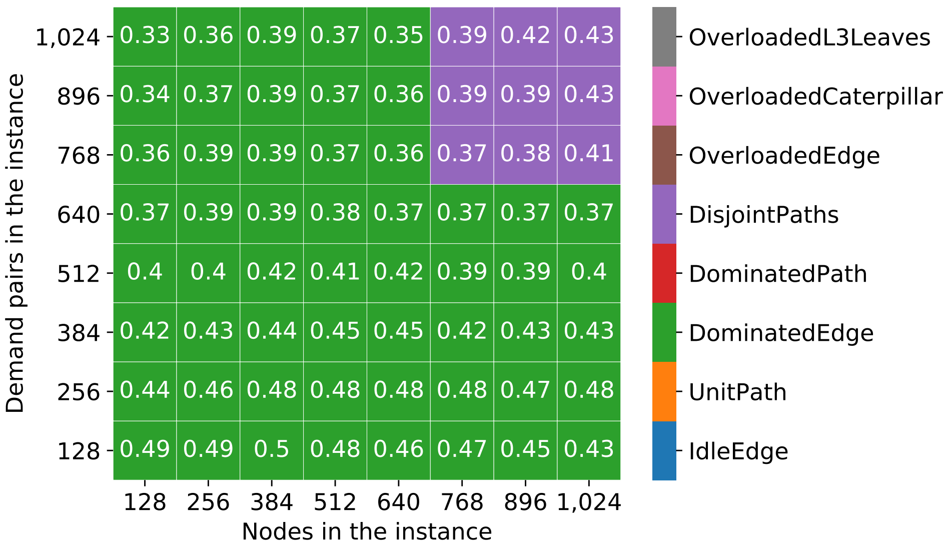


Figure 59: Dominant reduction rule in the algorithm by Guo and Niedermeier [23] on Degree3 trees with mostly short demand paths. Each cell is determined using 100 experiments. The text in each cell is the fraction of the total time spent by the dominant reduction rule.

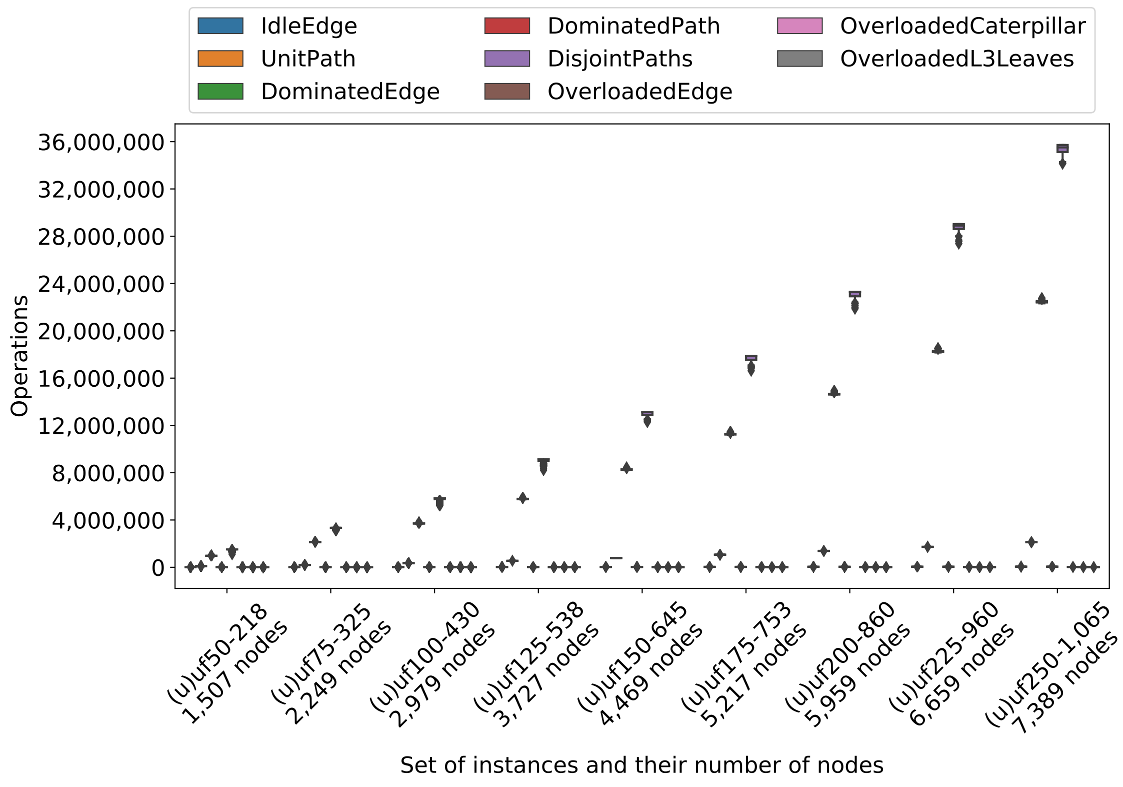
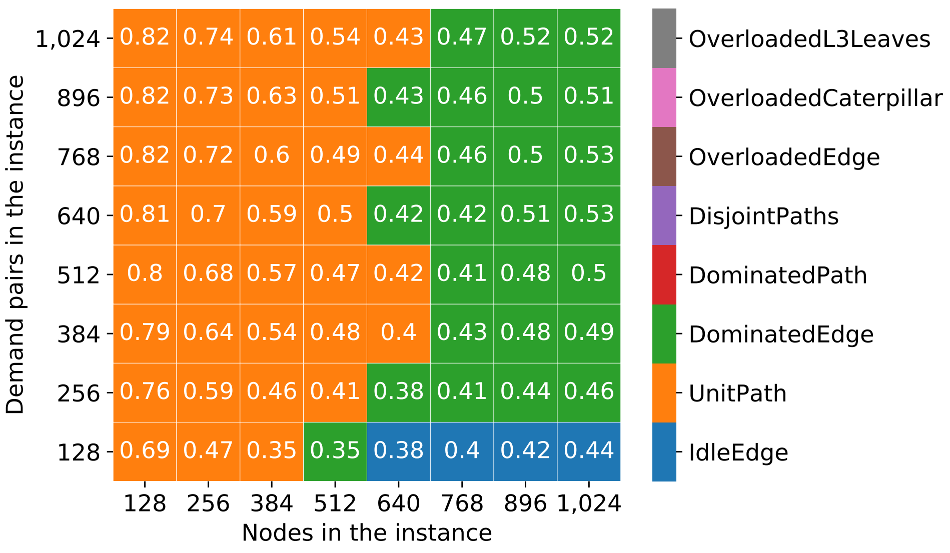


Figure 60: Box plots of the number of operations required to compute kernels on the 3-SAT instances, using the algorithm by Guo and Niedermeier [23]. We explained The instance categories displayed here in Table 2.



	128	256	384	512	640	768	896	1,024
1,024	0.82	0.74	0.61	0.54	0.43	0.47	0.52	0.52
896	0.82	0.73	0.63	0.51	0.43	0.46	0.5	0.51
768	0.82	0.72	0.6	0.49	0.44	0.46	0.5	0.53
640	0.81	0.7	0.59	0.5	0.42	0.42	0.51	0.53
512	0.8	0.68	0.57	0.47	0.42	0.41	0.48	0.5
384	0.79	0.64	0.54	0.48	0.4	0.43	0.48	0.49
256	0.76	0.59	0.46	0.41	0.38	0.41	0.44	0.46
128	0.69	0.47	0.35	0.35	0.38	0.4	0.42	0.44

Figure 61: Dominant reduction rule in the algorithm by Guo and Niedermeier [23] on Degree3 trees with demand pairs generated through a solution of size 100. Each cell is determined using 50 experiments. The text in each cell is the fraction of the total time spent by the dominant reduction rule.

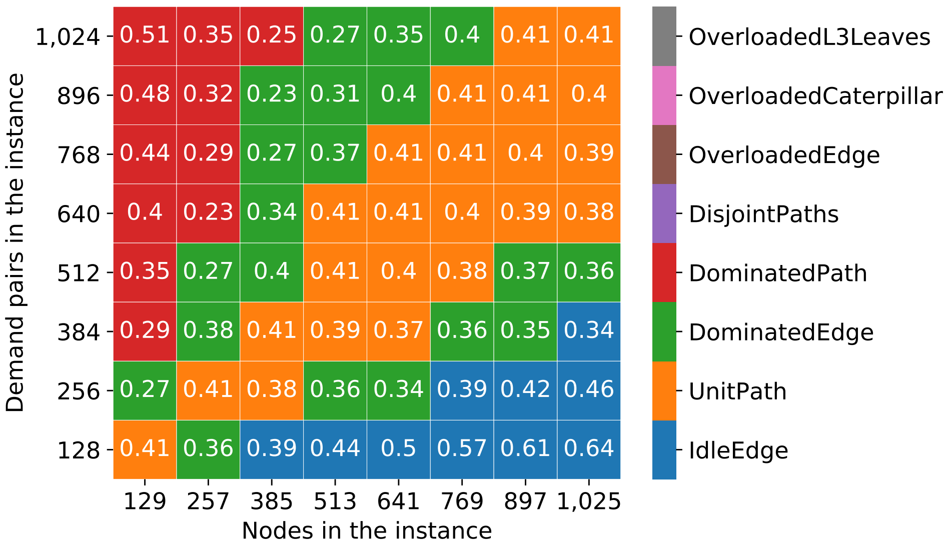


Figure 62: Dominant reduction rule in the algorithm by Guo and Niedermeier [23] on instances generated from Vertex Cover instances. Each cell is determined using 100 experiments. The text in each cell is the fraction of the total time spent by the dominant reduction rule.

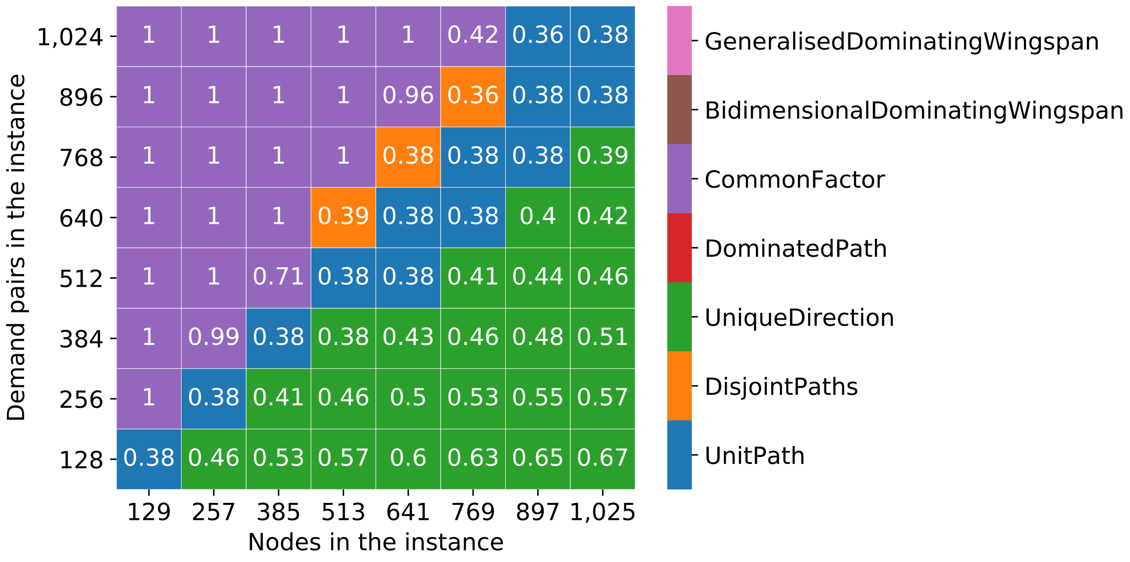


Figure 63: Dominant reduction rule in the algorithm by Bousquet *et al.* [4] on instances generated from Vertex Cover instances. Each cell is determined using 100 experiments. The text in each cell is the fraction of the total time spent by the dominant reduction rule.

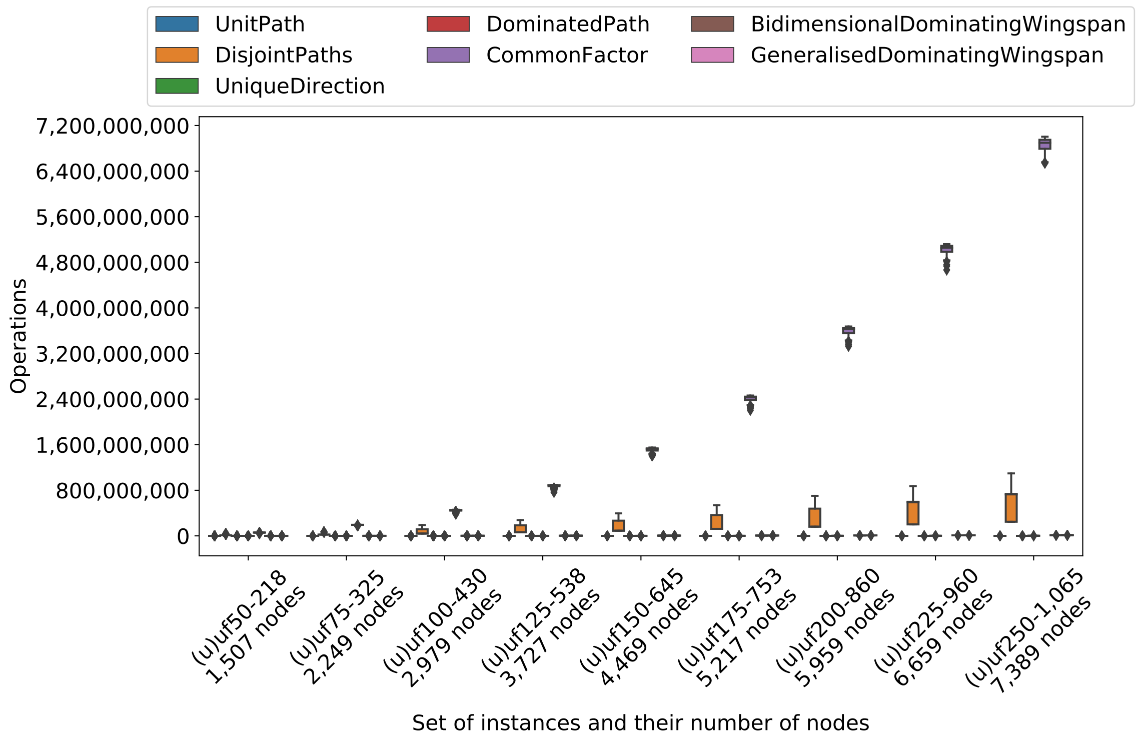


Figure 64: Box plots of the number of operations required to compute kernels on the 3-SAT instances, using the algorithm by Bousquet*et al.* [4]. We explained the instance categories displayed here in Table 2.

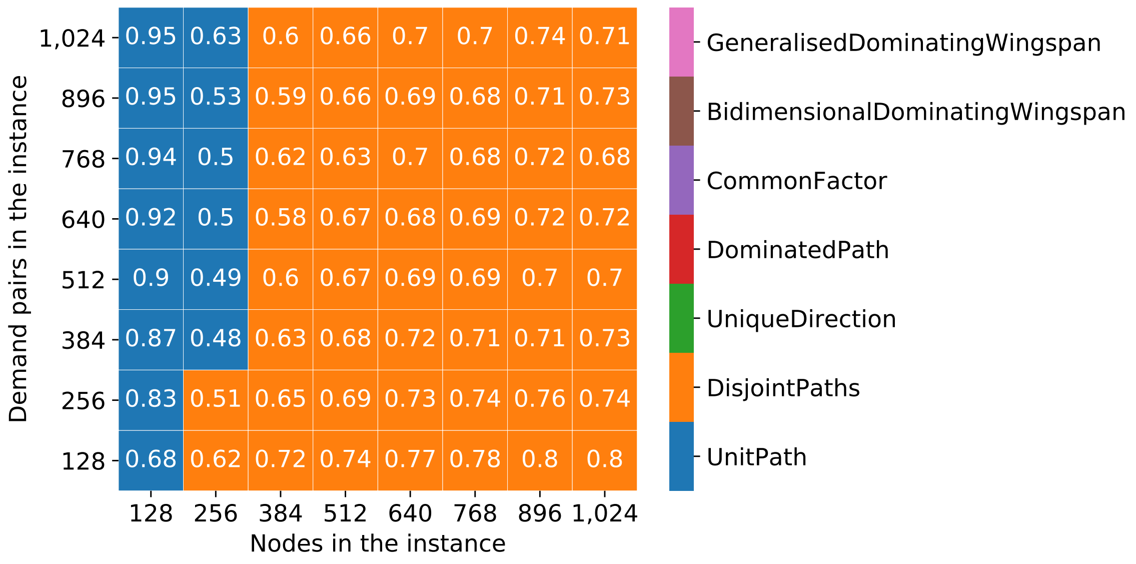
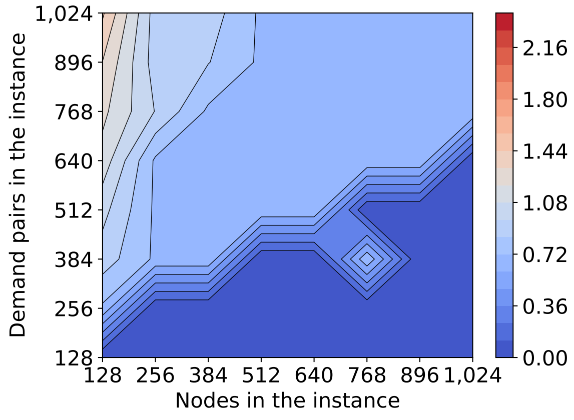
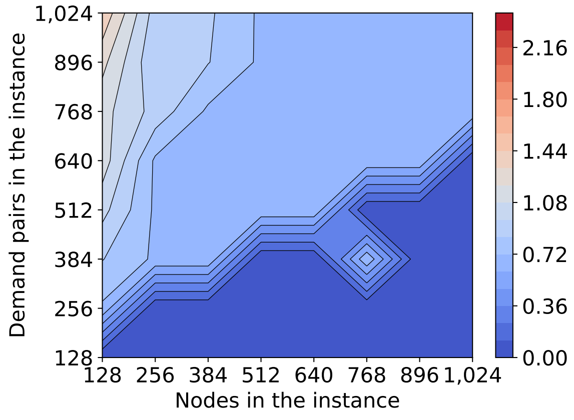


Figure 65: Dominant reduction rule in the algorithm by Bousquet *et al.* [4] on caterpillars with demand pairs generated through a solution of size 100. Each cell is determined using 50 experiments. The text in each cell is the fraction of the total time spent by the dominant reduction rule.



(a) Guo and Niedermeier [23].



(b) Bousquet *et al.* [4].

Figure 66: Contour plots of the base-10 logarithm of the IQR of the number of nodes in kernels on instances with caterpillars and uniformly at random demand pairs, using both algorithms. Each grid point is determined using 100 experiments.

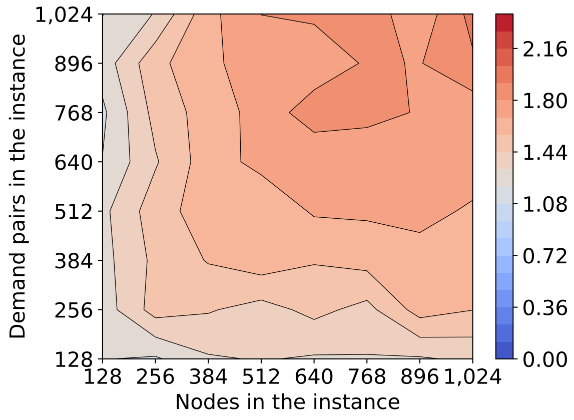


Figure 67: Contour plot of the base-10 logarithm of the IQR of the number of demand pairs in kernels computed by the algorithm by Guo and Niedermeier [23] on instances with a Degree3 tree and uniformly at random demand pairs. Each grid point is determined using 100 experiments.

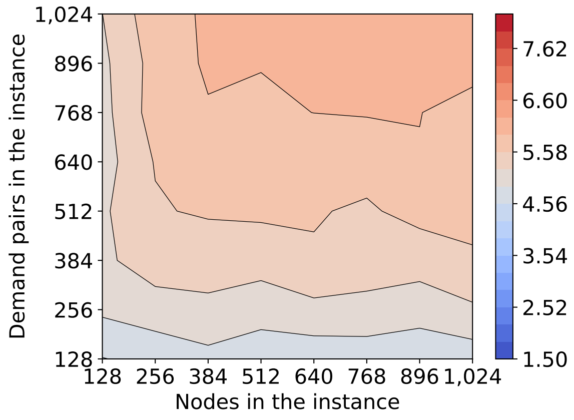


Figure 68: Contour plot of the base-10 logarithm of the IQR of the total number of operations required to compute kernels on instances with a Prüfer tree and uniformly at random demand pairs using the algorithm by Bousquet *et al.* [4]. Each grid point is determined using 100 experiments.