Algorithm Engineering for the Partial Dominating Set Problem

Marie Kastning

Department of Computer Science
Philipps-University of Marburg

A thesis submitted for the degree of
Bachelor of Science

Matriculation number: 3237388
Supervisors:
Prof. Dr. Christian Komusiewicz
M. Sc. Nils Morawietz
M. Sc. Frank Sommer

April 2022

Summary

Partial Dominating Set is a well known NP-complete problem, with applications in social network analysis. The task is to determine a subset of at most size k that covers as much vertices as possible. More precisely, given a graph and the maximum numbers of vertices k, a subset of at most size k that covers most possible vertices in their closed neighborhood is requested. In this paper, two algorithms are presented that compute an exact solution to this problem. The first algorithm is a search tree algorithm based on the brute-force-algorithm, which computes a trivial solution by forming all possible k-sized subsets and returning the one that covers the most vertices. This is then improved by various algorithm engineering techniques, all of which are based on reducing the number of computed subsets by discarding subtrees prematurely. The second algorithm is based on an integer linear programming (ILP) formulation. This is then solved using the Gurobi ILP solver. After these algortihms are individually tested on established benchmark graphs for k=$\{1,\ldots,15\}$, they are compared with regards to their running time. These tests show that the running time of the brute force algorithm, is substantially decreased by the presented improvements. However, the ILP algorithm outperforms the best version of the search tree algorithm. This is especially true for larger k, since the ILP formulation is mostly independent of k.

Contents

1 Introduction

With time the importance of social networks is constantly growing. And as their importance increases, so does their size. This has been demonstrated by the authors M. A. Jayaram, Gayitri Jayatheertha and Ritu Rajpurohit, who observed the size of the social networks Twitter, Facebook and LinkedIn over several years. They discovered that all of these networks, but especially Facebook, have grown significantly in size over the years [JJR20]. Consequently, the algorithms, which process these networks must be designed very efficiently.

Computing a MINIMUM DOMINATING SET of a network is a common way to analyze a social network. In order to compute this, a subset of the vertex set of a network is requested. Given a graph $G = (V, E)$, a subset of V is a dominating set if the vertices of that subset cover the complete vertex set of G. A vertex v covers a vertex u if and only if $v = u$ or the edge $\{u, v\} \in E$. The MINIMUM DOMINATING SET is a dominating set of minimum size. The NP $\neq$ P hypothesis implies that all problems that are NP-complete are not solvable in polynomial time. And Karp has proven in 1972 that MINIMUM DOMINATING SET is NP-complete [Kar72]. Dominating sets are often computed, in order to localize influential vertices in social networks. This brings us to a dilemma, as MINIMUM DOMINATING SET is a central problem of combinatorial optimization. Moreover, there are (as of 1998) more than 1200 publications on MINIMUM DOMINATING SET, or on one of the 75 known variations of the problem [HHS98].

In practice, there are numerous fields of application. One field is described in "New metrics for dominating set based energy efficient activity scheduling in ad hoc networks". The authors emphasize the importance of a dominating set in multi-hop wireless networks. Here, the dominating set is used to keep the network active by ensuring that at all times each vertex is either active itself or one of its neighbors is. The authors published this paper to present new metrics for this purpose [Sha+03]. Moreover, there are various other applications ranging from applications in algorithmic biology [NA16] and game theory [OTA21] to the analysis of social networks. An example of this is provided by a paper on the positive influence of social networks on today's society. Where a dominating set algorithm is presented to measure this positive influence [WCX09]. Furthermore, from a theoretical point of view, MINIMUM DOMINATING SET proves to be extremely robust against various algorithmic approaches such as approximation theory. For example, it has been shown that the problem cannot be approximated with factor $(1 - \epsilon) \ln |V|$ for any $\epsilon > 0$ unless NP has $n^{\mathcal{O}((\log \log n))}$-time algorithms, where n denotes the input size [Fei98].

In some situations, however, it is more interesting to cover only a part of the vertex set. For this purpose numerous problems are defined. One example is PARTIAL DOMINATING SET, where the set has maximum size k and as many vertices as possible should be covered. Another example is the MINIMUM PARTIAL DOMINATING SET, where p percent of the vertex set must be covered, such that the cardinality of the set is minimal. These problems are directly related, since both can be reduced to each other, meaning they can be solved by polynomially many executions of the other problem. Furthermore, these problems are also NP-complete, since they are a generalization of the MINIMUM DOMINATING SET problem. In practice, these situations often occur in social networks, for example, where companies advertise their products through influencers. The PARTIAL DOMINATING SET can be used here to achieve maximum reach by investing in only k influencers. Regarding this, the authors of the paper "Partial vs. Complete Domination: t-Dominating Set" presented an algorithm that solves the T-PARTIAL DOMINATING SET in $\mathcal{O}((4 + \epsilon)^t poly(|V|))$ time. In this case the T-PARTIAL DOMINATING SET is defined as the problem of finding a set of at most k vertices that dominate at least t vertices. The running time is a so called fixed-parameter tractable (FPT) running time. The corresponding FPT algorithms are polynomial as a function of $|V|$, but not as a function of t. Therefore, these algorithms are particularly efficient for instances with small values of t. [KMR07].

An approach to the MINIMUM PARTIAL DOMINATING SET can be found in the paper "Efficient Approximation Algorithms to Determine Minimum Partial Dominating Sets in Social Networks". This paper presents a common approach to NP-hard problems, which is to provide an approximation of the exact solution. For this purpose numerous approximation-algorithms for the minimum partial dominating set problem are presented and tested. Furthermore, it is discussed which algorithm is suitable for which instances and coverage parameters [CTB15].

Since an FPT algorithm and several approximation algorithms for the PARTIAL DOMINATING SET problem have already been presented, the purpose of this work is to develop a third approach to the introduced partial dominating set problems. It is to develop an algorithm that provides an exact solution for PARTIAL DOMINATING SET and is relatively efficient. Since all the above introduced partial dominating set problems can be decreased to each other, this approach can be used to compute any of these definitions of the partial dominating set. For this purpose, two algorithms are presented, tested and compared in the following. Firstly, in Section 4 a brute-force-algorithm is presented which finds the exact solution in a trivial way. For this purpose, all possible k sized subsets are generated. Then, the subset that covers the most vertices is extracted. This subset then represents the partial dominating set. In the further course of the work, this algorithm is then improved by algorithm engineering techniques in Section 5. These are based on decreasing the number of subsets generated by the algorithm. The second algorithm is presented in Section 6. It is a formulation of PARTIAL DOMINATING SET as an integer linear programming (ILP) problem. This formulation is solved using the Gurobi ILP solver and compared to the first algorithm in terms of running time in Section 7. The tests are performed on established benchmark datasets, which are presented in Section 3.

2 Preliminaries

This section presents the notation and more precise definitions that will be used in the following thesis. An *undirected graph* $G = (V, E)$ consists of a set V of vertices and a set E of edges. An edge is given by an unordered pair of vertices, the terminal vertices of the edge. The vertices are often also called nodes [Tur09]. In this thesis a graph $G = (V, E)$ is defined by specifying V and E.
In an undirected graph $G = (V, E)$, the *open neighborhood* $N(v)$ of a vertex $v \in V$ is defined as the set of vertices that are adjacent to v. Formally, $N(v) := w \in V : \{v, w\} \in E$. Observe that $N(v)$ does not contain v itself. The open neighborhood of a set S is defined as: $N(S) := \cup_{v \in S} N(v)$ [SS10]. In contrast, the *closed neighborhood* $N[v]$ of a vertex $v \in V$ in an undirected graph $G = (V, E)$, is defined as the set of vertices that are adjacent to v and v itself. Formally, $N[v] = N(v) \cup \{v\}$. Observe that $N[v]$ contains v as well. The closed neighborhood of a set S is defined as $N[S] := \cup_{v \in S} N[v]$ [SS10].

The *degree deg(v)* of a vertex $v \in V$ is defined as the number of edges $e \in E$ incident with v. Thus $deg(v)$ is equivalent to $|N(v)|$. A vertex $v \in V$ is considered *isolated* if $deg(v) = 0$. The maximum degree of a graph is defined as $\Delta(G) := \max\{deg(v) : v \in V\}$, whereas the minimum degree is defined as $\delta(G) := \min\{deg(v) : v \in V\}$ [Bra13].

The *edge density* $d(G)$ of an undirected graph is an indicator of the ratio of the edges of G to the maximum number of edges that might exist in G. Formally, $d(G) = |E|/\binom{|V|}{2}$ [Die17].

In a graph $G = (V, E)$, there exists a *path* between two vertices u and v if there exists a sequence of vertices $(u, \ldots, v) \in V$ such that these vertices are all pairwise distinct and each two consecutive vertices are adjacent. The length of this path is defined as the decremented size of that sequence. A Graph $G = (V, E)$ is a *tree* if and only if $G = (V, E)$ contains exactly one path from u to v for each pair $\{u, v\}$ with $v \neq u$. A *rooted tree* is a tree that has a node assigned as its root. Moreover, the *depth of a node* in a rooted tree is defined as the length of the path from that node to the root. Correspondingly, the *depth of the tree* is defined as the depth of the deepest node of that tree [Gou12]. Note that in this work, the elements of V are called vertices unless G is a tree, then they are called nodes.

Important definitions for the problem are now introduced.

Definition 2.1: (Dominating Set)
A vertex subset D of an undirected graph G is a *dominating set* if the following holds for all vertices $v \in V$: $v \in D$ or $\exists\{u, v\} \in E : u \in D$. Thus $D \subseteq V : N[D] = V$.

Dominating Set
Input: An undirected graph $G = (V, E)$.
Task: Find a dominating set.

Minimum Dominating Set
Input: An undirected graph $G = (V, E)$.
Task: Find a dominating set of minimum size.

Definition 2.2: (Partial Dominating Set)
The partial dominating set problem requests a set, limited to the size k. The goal is then to find a subset D that covers the most vertices through its closed neighborhood.
The formal problem definition is:

Partial Dominating Set
Input: An undirected graph $G = (V, E)$ and an integer k.
Task: Find a partial dominating set $D \subseteq V$: $|D| \leq k$, such that $|N[D]|$ is maximized.

In the improvements that follow in Section 5, it is important to know for each vertex, which progress the vertex brings at each state when added to the current subset. For this purpose, the undominated neighborhood of a vertex, depending on the current subset is defined in the following. This neighborhood is the set of the vertices that would be newly covered (by the subset) if v were added to the current subset.

Definition 2.3: (Undominated Neighborhood $N_S^+[v]$)
$N_S^+[v]$ denotes the *undominated neighborhood* of the vertex v. This neighborhood is defined as the relative complement of the current subset in the closed neighborhood of v. Formally, $N_S^+[v] = N[v] \setminus N[subset]$.

3 Experimental Setup

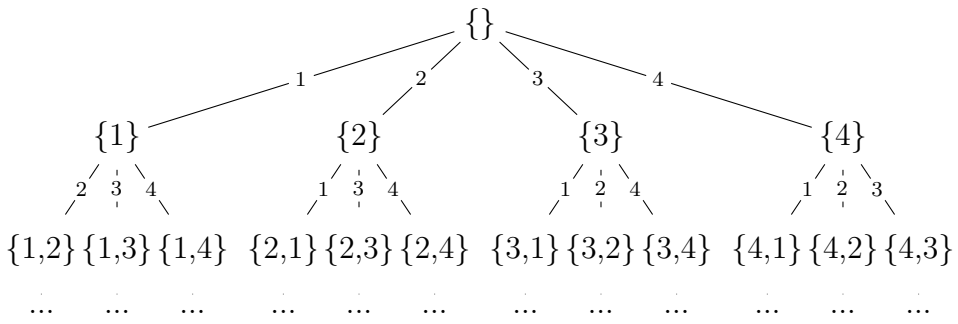
In the subsequent sections, the basic brute-force-algorithm and certain extensions, as well as an ILP formulation to solve the PARTIAL DOMINATING SET problem, are tested and compared. All experiments are performed on a single thread on an Intel(R) Xeon(R) Silver 4116 CPU with 2.1 GHz, 24 CPUs and 128 GB RAM. The algorithms are written in Java. The Gurobi Optimizer version 9.1.2 is used to solve the ILP from Section 6, using the python interface. (`https://www.gurobi.com`). The source code of all algorithms used in this thesis can be found on GitHub (`https://github.com/MarieKastning/Bachelor-Thesis-source-code`). Twenty social and technical networks were used for the experiments. They come from the Konect dataset [Kun13] and the Network Repository [RA15] and are listed in Table 1. The graphs are sorted in ascending order by number of vertices. The selected graphs all have between 25 and 1800 vertices. The algorithms are tested for all $k \in \{1, \ldots, 15\}$ and the execution is stopped after 20 minutes. Note that the improvements are not tested on their own. With each step an Improvement is added. Meaning BF is just the basic brute-force-algorithm. IR is the Brute-Force-Algorithm with an initial candidate set reduction, UB is the Brute-Force-Algorithm with an initial candidate set reduction and the use of an upper bound, and so on. In Table 1 the following abbreviations are used for the corresponding algorithms:

- BF is the brute-force-algorithm.
- IR is the Initial reduction in addition to BF.
- UB is the sum upper bound in addition to IR.
- UB+ is the sum upper bound and the union upper bound in addition to UB.
- LB is the lower bound in addition to UB+.
- IR+ is the Internal Reduction in addition to LB.
- GE is the greedy Extension in addition to IR+.
- TE is the trivially-optimal Extension in addition to GE.

Name of G	|V|	|E|	d(G)	BF	IR	UB	UB+	LB	IR+	GE	TE	ILP
moreno_zebra	27	111	0.316	**15**	**15**	**15**	**15**	**15**	**15**	**15**	**15**	**15**
contiguous-usa	49	107	0.091	6	6	14	14	14	**15**	**15**	**15**	**15**
ca-sandi-auths	86	124	0.034	4	5	**15**	**15**	**15**	**15**	**15**	**15**	**15**
adjnoun_adjacency	112	425	0.068	4	4	**15**	**15**	**15**	**15**	**15**	**15**	**15**
ca-netscience	379	914	0.013	2	2	**15**	**15**	**15**	**15**	**15**	**15**	**15**
econ-beause	507	39428	0.307	2	2	4	4	4	5	**15**	**15**	**15**
bio-diseasome	516	1188	0.009	2	2	**15**	**15**	**15**	**15**	**15**	**15**	**15**
soc-wiki-Vote	889	2914	0.007	2	2	**15**	**15**	**15**	**15**	**15**	**15**	**15**
bio-yeast	1458	1948	0.002	2	2	**15**	**15**	**15**	**15**	**15**	**15**	**15**
comsol	1500	48119	0.043	1	1	5	5	5	6	9	9	**15**
ca-CSphd	1882	1740	0.001	1	2	**15**	**15**	**15**	**15**	**15**	**15**	**15**
heart2	2339	340229	0.124	1	1	2	2	2	3	3	3	**15**
inf-openflights	2939	15677	0.004	1	1	7	7	7	9	9	9	**15**
psmigr_1	3140	410781	0.083	1	1	3	3	3	3	6	6	**15**
ca-GrQc	4158	13422	0.002	1	1	9	9	9	12	**15**	**15**	**15**
inf-power	4941	6594	0.001	1	1	**15**	**15**	**15**	**15**	**15**	**15**	**15**
soc-advogato	6541	39432	0.002	1	1	11	11	10	10	13	13	**15**
bio-dmela	7393	25569	0.001	1	1	10	10	8	11	11	11	**15**
ca-HepPh	11204	117619	0.002	1	1	2	2	2	2	2	2	**15**
ca-AstroPh	17903	196972	0.001	1	1	1	1	1	1	1	1	**15**

Table 1: Biggest values of k for which the algorithms solved the different graphs.

(a) With duplicates



(b) Duplicates eliminated

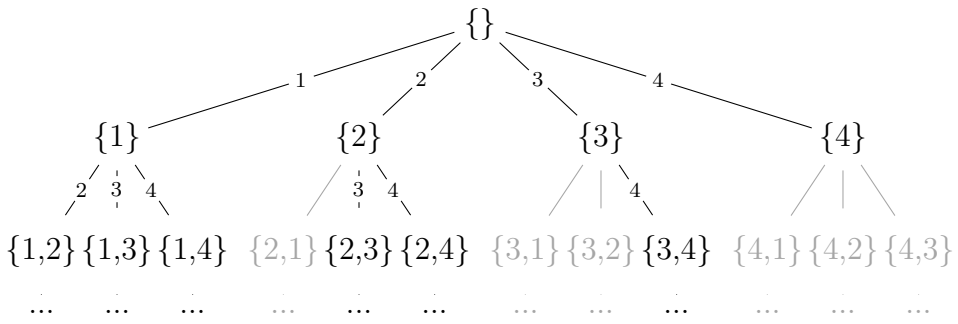


Figure 1: Subset tree for G=(V,E) where V={1,2,3,4}

4 Brute-Force-Algorithm

The PARTIAL DOMINATING SET problem is NP-complete. Therefore any algorithm that computes exact solutions for this problem runs in exponential time, assuming NP $\neq$ P. In the following, a brute-force-algorithm to the stated problem is introduced. This algorithm will initially compute all possible subsets of size k of the vertices in the given graph. Afterwards it will return the set that covers the most vertices in the graph. The brute-force-algorithm will be tested. Then to assure a better running time this algorithm will be modified in the Section 5. For this purpose algorithm engineering techniques will be applied to make the algorithm more efficient.

4.1 Subset Tree

The brute-force-algorithm is based on the computation of all possible subsets of size k of the vertex set V, which has size n. The procedure for iterating over all subsets of size k is based on a rooted tree. In this tree, each node represents a subset of the node set V. The set of all nodes of the tree is denoted by T in the following, and the following notations are defined for a tree node $t \in T$:

- $S(t)$ is the subset of vertices represented by t.
- $C(t)$ represents the list of vertices from $(V \setminus S(t))$ that can be added to $S(t)$; $C(t)$ may be indexed from 1 to n.

Consider Figure 1. The root r of the tree represents the empty set. Exactly one child is added to r for each graph node $v \in V$, representing the set $\{v\}$. Now at a node $t \neq r$ the children are constructed as follows. For each graph node $v \in C(t)$, exactly one child t_{child} of t is added, with $S(t_{child})$ representing the union of $S(t)$ and $\{v\}$. vertices that are already contained in $S(t)$ are ignored. This process is repeated for all children, as long as $|S(t)| < k$ is valid. In this

construction $|S(t)|$ is equal to the depth of t in T. Therefore the leaves of the tree represent all subsets of V with size k. However, this construction corresponds to the tree in Figure 1a. Looking at the tree in Figure 1b, it can be seen that it is not necessary to compute all possible subtrees at every stage: Since the order in which the objects are arranged in the subsets is irrelevant, the amount of subsets to be built can be decreased from $n \cdot (n-1) \cdot \ldots \cdot (n-k)$ to $\mathcal{O}(\binom{n}{k})$. To this end the index i is introduced:

- $i(t)$ denotes the index that is used to iterate through $C(t)$, avoiding elements that would create duplicates.

To avoid calculating duplicates, the index i is now added. With this, the construction of the children of a node t can be adjusted as follows to eliminate the duplicates. For each graph node $v \in C(t)$, starting at index $i(t)$, exactly one child t_{child} of t is added, with $S(t_{child})$ representing the union of $S(t)$ and $\{v\}$. vertices that are already contained in $S(t)$ or the set of the previous subtrees. Consider Figure 1b as an example. Assuming $k = 2$, then the tree nodes with depth 2 are the leaf nodes. Since the set {2,1} is equal to the set {1,2}, it is not necessary to compute {2,1} or even the following sets, since they are already completely computed using the subtree of the set {1,2}. Therefore, to avoid that sets are computed multiple times, the recursive brute-force algorithm operates with the index i. This index is used to iterate over $C(t)$. Evidently, at the root, $C(t)$ is equal to V and $i(t)$ is initially 1. Thus, according to the example in Figure 1, the child {1} is created from the root. Then the recursion is started to go deeper into this tree. For this, at first {1} is removed from $C(t)$. So at this point there are only three elements left in $C(t)$. Since $i(t)$ has not been increased yet, $i(t) = 1$, all elements from $C(t)$ are represented as children of t in the next level. This will not be the case for the sibling nodes of {1}. This is because the sibling nodes are not added until the next iterations of the loop. It follows that $i(t) > 1$. This will cause elements to be skipped in $C(t)$ in the recursion where the tree gets deeper and t gets child nodes. Namely the union of these elements which are already present in previous subtrees and $S(t)$. In the example of Figure 1, after the subtree {1} has been formed, the subtree {2} would be started. In this case, t= {2}, $i(t) = 2$, and $C(t)$= {1,3,4}. Observe that now only two children of t are created. One is the union of {2} and $C(t)[2]$ and the other is one is the union of {2} and $C(t)[3]$. This skips the subtree {2,1} as in Figure 1b and starts the subtree {2,4} and the subtree {2,3}. Note that if $|C(t)| - i(t) + |S(t)| \leq k$, branches will not reach the size of k. Thus these nodes t can be discarded.

4.2 Pseudocode

Algorithm 1 and Algorithm 2 provide the pseudocode for the brute-force-algorithm. Algorithm 1 covers some edge cases and starts the first recursion of Algorithm 2, the recursive searchtree. Algorithm 2 computes a partial dominating set, which is then returned in Line 7 of Algorithm 1. Algorithm 2 computes all subsets of size k. Since the order in which the objects are arranged in the subsets is irrelevant, the amount of subsets to be built can be decreased to $\binom{n}{k}$, as stated in Section 4.1. Algorithm 2 is an implementation of the duplicate free algorithm presented in Section 4.1. S in the pseudocode corresponds to $S(t)$, C corresponds to $C(t)$ and i corresponds to $i(t)$. Additionally, k, the maximum size of the searched partial dominating set is given as the input. As stated in Section 4.1 i is used to ensure that the same subset is not built more than once. In addition the global variables l_b and D are used in Algorithm 2. Where l_b manages the current lower bound, a value that describes the quality of the current solution for D. The lower bound is the size of the closed neighborhood of the best partial dominating set at that time. D stores the corresponding set. If a more detailed look is taken at Algorithm 2, it can be seen that in Line 1 the case where the current subset S has reached size k is intercepted. In this case the algorithm checks whether S is a better solution for the partial dominating set

Algorithm 1: Brute Force Algorithm

```
   Data: Graph G = (V, E), integer k
   Global Variables: current solution D
   Result: partial dominating set D of size k
1  S ← {};
2  C ← V;
3  if |C| ≤ k then
4  |  return C;
5  end
6  Build Subsets(S, C, k, 0);
7  return D;
```

than the current solution D (Line 2). If this is true, l_b and D are updated in Lines 3 and 4. If $|S| = k$ is true, then the current subtree is not built any deeper since the node of S has reached the depth of k. Otherwise, the subtree is extended by another level in Lines 8 to 14. Line 8 starts the for-loop, which iterates over all vertices in C that can be added to the current set without creating duplicates. As described in Section 4.1, this is ensured by the index i which ensures that certain elements that already exist in such a subtree are skipped. Then the first element w, which comes after the index i in C, is removed from C in Line 10 and added to the current subset S in Line 11, to start another recursion with these sets in Line 12. After w is added to C again in Line 13 and removed from S again in Line 14 to restore the size of C. Reinserting w into C is important for the index-based iteration over C in Line 8. After the index i is incremented, the sibling nodes are created in the next iteration of the for loop.

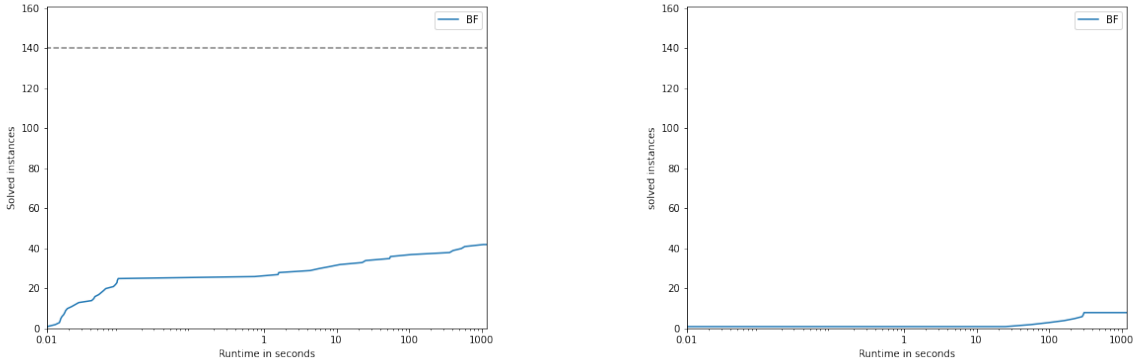
Algorithm 2: Build Subsets

```
   Data: Set S, List C, integer k, integer i
   Global Variables: lower bound lB, current solution D
1  if |S| = k then
2  |  if |N[S]| > lB then
3  |  |  lB ← |N[S]|;
4  |  |  D ← S;
5  |  end
6  end
7  else
8  |  for j ← i to |C| do
9  |  |  w ← C[j];
10 |  |  remove w from C;
11 |  |  S ← S ∪ {w};
12 |  |  Build Subsets(S, C, k, j);
13 |  |  reinsert w at position j into C;
14 |  |  S ← S \ {w};
15 |  end
16 end
```

4.3 Implementation Details

In order to implement the Java version of the brute-force-algorithm, numerous data structures were used. First, a graph class *MyGraph* is implemented. In this class the vertices are represented as integers, such that $G = (V, E) : V = \{1, 2, \ldots, n\}$, and the edges are stored for each vertex by adjacency lists, for this a *HashMap* is used, which maps each vertex to a list of vertices. To compute the partial dominating set an instance of this graph class and an integer k must be passed. Furthermore, the set S is implemented as a *HashSet*, while C is implemented as an *ArrayList* to be able to address its contents by index. The l_b is a simple int value declared to 0. The partial dominating set D is represented by a *HashSet*, initially declared to be null. To compute the closed neighborhood of a set a separate method was implemented. This method iterates through the elements of the given set and combines all neighborhoods, as well as the elements themselves.

4.4 Evaluation



(a) $k \in \{1, \ldots, 7\}$ (b) $k \in \{8, \ldots, 15\}$

Figure 2: Brute-Force-Algorithm - solved instances depending on running time per instance.

Figures 2a and b show how many instances the brute-force-algorithm could solve with $k \in \{1, \ldots, 7\}$ and how many with $k \in \{8, \ldots, 15\}$ in 20 minutes per instance. Since the algorithm is tested on 20 graphs, at most 140 instances could possibly be solved in Figure 2a and at most 160 instances could possibly be solved in Figure 2b. The evaluation of the solved instances for $k \in \{1, \ldots, 7\}$ by the brute-force-algorithm shows that the algorithm solves most instances within 1 second per instance. BF solved 26 instances within 1 second, 34 within 30 seconds, 36 within 1 minute and 42 within 20 minutes. Thus, the algorithm solves only one third of the inputs. For the inputs with $k \in \{8, \ldots, 15\}$ the algorithm solves only 8 instances, none of them within 1 second and only two within 1 minute. Table 1 shows similarly poor values for the brute-force-algorithm. Although it solves the smallest instance with $|V| = 27$ for $k = 15$, only instances $k \leq 2$ are solved from $|V| > 379$. These poor results are not surprising, because due to the computation of all subsets, the running time grows exponentially as k increases. Thus, in Section 5, improvements are presented in order to decrease the running time.

5 Improvements

In the following section modifications that improve the brute-force-algorithm by decreasing the number of subtrees formed, are presented. For this purpose, the set S, the list C, and the global variables D, and l_b are used as in Section 4. In the following, the improvements, candidate set reductions, bounds, and the greedy as well as the trivially-optimal extension are presented. Additionally the vertices in C are sorted descendingly by size of undominated neighborhood. This will ensure that the best solution is found quicker, which will allow additional subsets to be eliminated through the upper bound. The upper bound is a measure for the maximum coverage of a subset. The algorithms that prepare, combine, and handle the following improvements can be viewed in the appendix. These also handle the sorting of C.

5.1 Candidate Set reduction

The purpose of the following candidate set reductions is to decrease the number of vertices that will be considered in the subset trees, since this part is decisive for the exponential running time. The list C will represent the decreased vertex set that contains the set of vertices from V, which for the current subset S, may still be in an optimal solution for the Partial Dominating Set. In other words, the vertices in $V \setminus C$ are not relevant for the current subset-tree.

5.1.1 Initial Reduction

To decrease the number of vertices for which the algorithm builds separate subtrees and ultimately the effort, an initial candidate set reduction is carried out. This is done by deleting every vertex v, for which the closed neighborhood is completely covered by the closed neighborhood of another vertex u. Formally, if $N[v] \subseteq N[u] \rightarrow$, then v is deleted from C.
In order to assure that not both vertices u and v are deleted in the case of $N[u] = N[v]$, the constraint $v > u$ is added.

Algorithm 3: Initial Reduction

```
Data: Graph G = (V, E).
Result: A reduced vertex set C.
C ← V;
foreach v ∈ V do
    foreach u ∈ N[v] do
        if v > u ∧ N[v] ⊆ N[u] then
            C ← C\{v} ;
        end
    end
end
C ← C sorted descendingly by degree;
return C;
```

This candidate set reduction provides a correct solution despite the decreased number of vertices, because only the vertices that would not be present in the optimal solution are deleted from C. This is clear in the initial candidate set reduction, since only those vertices are deleted from C that cover exactly the same as another vertex or less. This is due to the following implication. For this purpose, a set S represents a subset of size k and contains $\{v\}$, is considered. Moreover, the condition that $N[u] \supseteq N[v]$ is true. Then Fact 1 follows directly from

this: $N[S \setminus \{v\} \cup \{u\}] \supseteq N[S]$. Furthermore, Fact 2 is clear: $N[S \setminus \{v\}] \supseteq N[s] \setminus N[v]$, because it may be that other elements from S cover a subset of $N[v]$. It then follows that $N[S] \setminus N[v] \cup N[u] \supseteq N[S]$. Thus v can be deleted from C, since it is always better for S to contain u. Moreover, $N[u] \supseteq N[v]$ can only be true, if v and u are adjacent, therefore in Lines 2 and 3 all vertices are compared only with their neighbors. Since $N[u]$ cannot be a subset of $N[v]$, if u and v are not adjacent, because $u \in N[u]$, but $u \notin N[v]$.

5.1.2 Internal Reduction

In contrast to the initial reduction, the following reduction rule operates with the undominated neighborhood of elements. This neighborhood changes only when a new element w is added to the set S. Since w was added to S further vertices are covered by S. This influences the undominated neighborhood of the vertices in C. Furthermore, when w is added to S, only the undominated neighborhood of the vertices of the first and second neighborhood of w may have changed, due to the addition of w to S. This fact is exploited to make further candidate set reduction on C more efficient. The number of vertices to be compared is decreased, which saves a lot of time. Since the undominated neighborhood changes each time a new element is added to S, this reduction is performed every time a new vertex has been added to S.

Algorithm 4: Internal Reduction

Data: Set S, Integer k, List C, Integer w.
Global Variables: lower bound l_b
Result: A reduced vertex set C.

```
/* Note that C is sorted descendingly by undominated neighborhood */
/* DR1 */
1  ℓ ← k − |S|;
2  if |C| < ℓ then
3  |  return C;
4  end
5  acc ← |N[S]|;
6  for i ← 1 to ℓ − 1 do
7  |  acc ← acc + |N_S^+[C[i]]|;
8  end
9  δ ← l_b − acc;
10 foreach v ∈ C do
11 |  if |N_S^+[C[i]]| ≤ δ then
12 |  |  C ← C\ {v};
13 |  end
14 end
/* DR2 */
15 foreach v ∈ N_1[w] ∪ N_2[w] do
16 |  foreach u ∈ N[v] do
17 |  |  if v > u ∧ N_S^+[v] ⊆ N_S^+[u] then
18 |  |  |  C ← C\ {v};
19 |  |  end
20 |  end
21 end
22 return C;
```

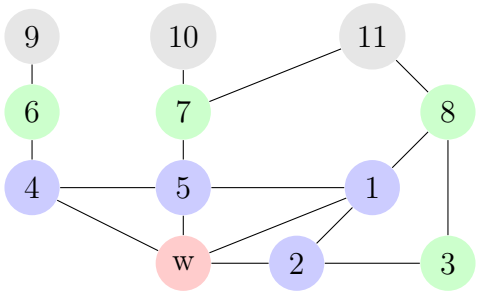


Figure 3: Example Graph. An element w has just been added to S, in this figure, the corresponding vertices are highlighted. It shows how adding an element w (highlighted in red) to the subset S affects the undominated neighborhood of the vertices in the first and second neighborhood of w (highlighted in blue and green), while other nodes are not influenced just like the ones from the third neighborhood of w (highlighted in gray).

In Algorithm 4 two functions for the candidate set reduction are implemented. The first one, DR1 accumulates the size of the undominated neighborhood of the $\ell - 1$ first vertices in C (Lines 6 to 9), this value is assigned to *acc*. Note that ℓ equals the number of vertices that can still be added to S, without violating the condition that $|S|$ is smaller than k. Consider that at this point C has already been sorted in descending order of undominated neighborhood size (Line 1). Afterwards, the value δ is computed. This value indicates how large the undominated neighborhood of a vertex has to be, to be contained in any better solution for D extending S. More precisely, if the size of the undominated neighborhood of a vertex is at most $\delta = l_b - acc$, then any set extending S and containing any of these vertices will never cover more than the current partial dominating set D, which coverage is stored in l_b. This is true because the size of the undominated neighborhoods of the $\ell - 1$ vertices whose undominated neighborhood is largest have been accumulated. Thus *acc* is indicates what can be covered at most with $\ell - 1$ vertices. Hence all vertices whose undominated neighborhood is not greater than δ are deleted in Lines 11 to 15.

The second one, DR2, performs a similar reduction that is performed in the initial reduction. Recall that the vertices whose neighborhoods are completely dominated by another vertex are deleted in Section 5.1.1. Except for two differences. The main difference being, that instead of the closed neighborhood of a vertex v ($N[v]$) now the dominated neighborhood of v ($N_S^+[v]$) is used to compare vertices. Also, the above mentioned fact is used that only the first and second neighborhood of the newly added element must be checked, and Fig. 2 shows the newly added element w and its first, second and third neighborhoods of an example graph. The vertices that are within the first neighborhood of w are highlighted in blue. Those that are within the second neighborhood of w are highlighted in green and the ones that are within the third neighborhood are highlighted in gray. It is highlighted how the first and second neighborhoods are affected by the addition of w. In fact, the complete first neighborhood and w is now covered by S. This may effect every vertex in the first neighborhood since at least the vertex itself is now covered by S and every vertex u in the second neighborhood, since at least one vertex $v \in N[u] \cap N[w]$ is now covered by S and the undominated neighborhood must be updated if it contained that vertex. Thus, in Lines 16 to 22, only the vertices v that are in the first and second neighborhood of w are checked and the ones that now cover a subset of the undominated neighborhood of another vertex are deleted. Formally expressed, If $N_S^+[v] \subseteq N_S^+[u]$, then v is deleted from C. Here again the condition $v > u$ covers the case $N_S^+[u] = N_S^+[v]$, so that not both v and u are deleted. Finally the decreased and sorted list C is returned in Line 23.

5.2 Upper and Lower Bounds

Since the basic algorithm computes all different subsets, a good way to make this algorithm more efficient is to limit the number of subtrees that are pursued. To this end, it is common to introduce an upper and a lower bound. The lower bound is a global variable which is updated every time a better solution is found. It indicate how good the current solution is. So the subtrees that in no way are going to contain a solution that is better than the current solution D, which is evaluated by the lower bound, may be discarded. To identify such subtrees, an upper bound which indicates the maximum value of any subset of size k, that contains the current subset needs to be computed. If this value is lower than the current lower bound, then the current subtree does not have to be pursued any further, reducing the running time.

5.2.1 Upper Bound

There are numerous ways to compute an upper bound. The lower the upper bound, the more subtrees may be discarded. The following algorithm, Algorithm 5, computes two upper bounds: The *sum upper bound* and the *union upper bound*. For both of these upper bounds, the set C is to be sorted descendingly by undominated neighborhood. Doing so makes computing the sum upper bound, almost free of charge, since it simply sums up $|N(s)|$ and the size of the undominated neighborhoods of the ℓ first vertices from C. If this first upper bound is not bigger than the current lower bound, then it is returned, so that the second, more expensive upper bound does not have to be computed. The second upper bound is the union upper bound which is based on computing two values: Firstly the size of the union of the neighborhood of the ℓ first vertices from C and the current subset, which is the expensive part, and secondly the sum of the size of the undominated neighborhood of the $\ell - 1$ first vertices from C plus the size of the undominated neighborhood of the $(\ell + 1)^{\text{th}}$ vertex from C. Note that ℓ equals the number of vertices that can still be added to S, without violating the condition $|S| \leq k$. The upper bound then is the maximum of those two values.

5.2.2 Lower Bound

A universal way to approximate a solution to an NP-hard problem is to solve it with a greedy heuristic. This is so popular, because it computes a valid solution in a very efficient way. However, the computed result is not always optimal.
So how can a greedy heuristic be used to our advantage when computing an optimal solution? The answer is simple: It can be used as an initial lower bound. The bigger and more accurate the computed lower bound is, the more subtrees may be discarded due to their upper bound, and therefore less subtrees are computed, which makes the algorithm a lot more efficient.

On that account, the next step will be to determine an initial lower bound by computing a greedy solution. The lower bound is then updated when a better solution was found. The greedy strategy, that is used in the following algorithm, Algorithm 6, is to always add the vertex that, at the current state, produces the optimal outcome for the subset. More precisely the element v, with the biggest undominated neighborhood is added. Formally, v is chosen such that $|N_S^+[v]|$ is maximized.

Algorithm 5: Upper Bound

Data: Set S, List C, Integer k.
Global Variables: lower bound l_b
Result: upper Bound of current subset

```
1  C ← C sorted descendingly by undominated neighborhood;
2  sumUB ← |N[S]|;
3  ℓ ← k − |S|;
4  for i ← 1 to ℓ do
5  |  sumUB ← sumUB + |N_S^+[C[i]]|;
6  end
7  if sumUB ≤ l_b then
8  |  return sumUB
9  end
10 acc ← |N[S]|;
11 U ← N[S] ;
12 for i ← 1 to ℓ do
13 |  U ← U ∪ N[C[i]] ;
14 |  if i < ℓ then
15 |  |  acc ← acc + |N_S^+[C[i]]|;
16 |  end
17 end
18 acc ← acc + |N_S^+[C[ℓ + 1]]| ;
19 return max{acc, |U| };
```

Algorithm 6: Lower Bound

Data: Graph $G = (V, E)$, Integer k.
Result: Coverage of a greedily computed partial dominating set

```
1  lB ← 0;
2  foreach v ∈ V do
3  |  S ← {v};
4  |  while |S| ≤ k do
5  |  |  w ← vertex u that maximizes |N_S^+[v]|;
6  |  |  S ← S ∪ {w};
7  |  end
8  |  if |N[subset]| > lB then
9  |  |  lB ← |N[subset]|;
10 |  end
11 end
12 return lB;
```

5.3 Greedy and Trivially-Optimal Extensions

In some situations, one can read a solution, that is better than the current solution D or even the best solution for the current subtree, directly from the set C. To achieve this Algorithm 7 computes a few values and sets. Firstly, in Line 7 the set U is computed. This corresponds to the union of the undominated neighborhood of the ℓ vertices with the largest undominated neighborhood. Note that ℓ equals the number of vertices that can still be added to S, without violating the condition that $|S|$ is at most k. Furthermore, the sum s of the cardinalities of these neighborhoods is computed in Line 8. In the following, we use these computations to

decide whether these ℓ vertices with the largest undominated neighborhood combined with S provide an update of the lower bound l_b and D, or even whether the current subtree can be discarded.

Algorithm 7: Extension preparation

Data: Set S, List C, Integer k.
Result: true if subtree is discarded. And false if it is pursued further.

```
1  ℓ ← k − |S|;
2  U ← {} ;
3  L ← {} ;
4  s ← 0 ;
5  for i ← 1 to ℓ do
6  |  U ← U ∪ N_S^+[C[i]];
7  |  s ← s + |N_S^+[C[i]]|;
8  |  L ← L ∪ C[i];
9  end
10 Greedy Extension(S, U, L);
11 return Trivially-Optimal Extension(U, s);
```

5.3.1 Greedy Extension

The following algorithm, Algorithm 8, called "Greedy Extension" is called by Algorithm 7. "Greedy Extension", uses the in Algorithm 7 computed values and iterables to check whether $|U|$ plus $|N[s]|$ exceeds the current lower bound (Line 1). Because if it does, then the union of the sets S and U dominates more than the current solution for the partial dominating set D. This would result in updating the current solution (Lines 2 - 3). However, the tree is not discarded since even better solutions may still exist in the subtree.

Algorithm 8: Greedy Extension

Data: Set S, Set U, Set L .
Global Variables: lower bound l_b, current solution D

```
1  if |U| + |N[S]| > l_b then
2  |  D ← S ∪ L;
3  |  l_b ← |N[D]| ;
4  end
```

5.3.2 Trivially-Optimal Extension

Algorithm 9, called "Trivially-Optimal Extension" is called by Algorithm 7, after "Greedy Extension" was called. In "Trivially-Optimal Extension", the computed value s and set U is used to check whether the potential partial dominating set that may have been built in Algorithm 8 is optimal for the current subtree node. Because if the sum s is equal to U (Line 1), then the undominated neighborhoods of the ℓ vertices with the largest undominated neighborhoods are disjoint. It follows that the union of S and U provides the optimal partial dominating set for this complete subtree. Thus this subtree can be discarded.

Algorithm 9: Trivially-Optimal Extension

Data: Set U, Integer s.
Result: true if subtree is discarded. And false if it is pursued further.

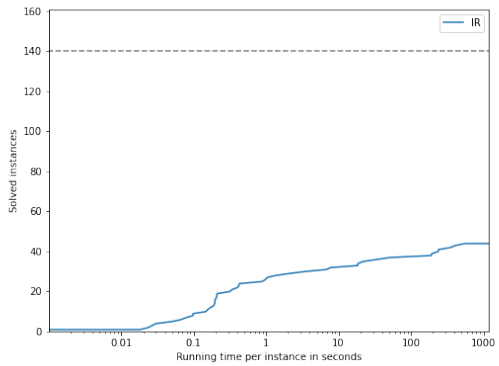
```
1 if s = |U| then
2 |   return true;
3 end
4 return false;
```

5.4 Implementation Details

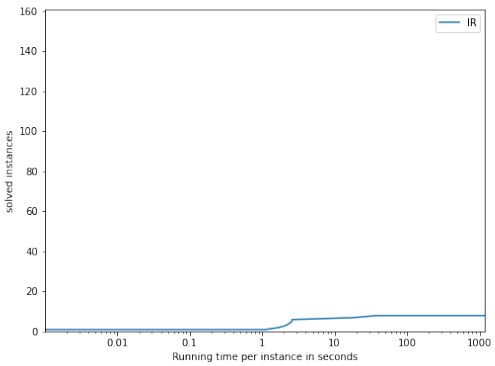
In order to implement the improved algorithm, additional data structures were used. A significant change to the implementation in the brute-force-algorithm is that the undominated neighborhood of the vertices was managed dynamically. For this purpose, a *HashMap* is used that maps each vertex to its undominated neighborhood. It is declared before the first vertex is even added to S. For this purpose, each vertex is first mapped to its closed neighborhood. This is then updated after an element w is added. This update, is performed only on the first and second neighborhoods of w, as explained in Section 5.1.2.

5.5 Evaluation

The following Figures 4-11 a and b show how many instances each improvement could solve with $k \in \{1, \ldots, 7\}$ and how many with $k \in \{8, \ldots, 15\}$ in 20 minutes per instance. Recall that, a maximum of 140 instances can be solved in Figure a and a maximum of 160 instances can be solved in Figure b. Table 1, in Section 3, is also used to evaluate the tests.



a) $k \in \{1, \ldots, 7\}$

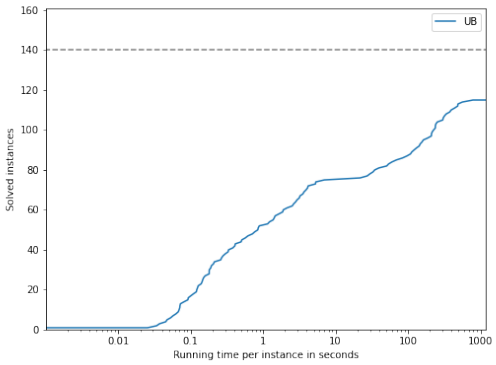


b) $k \in \{8, \ldots, 15\}$

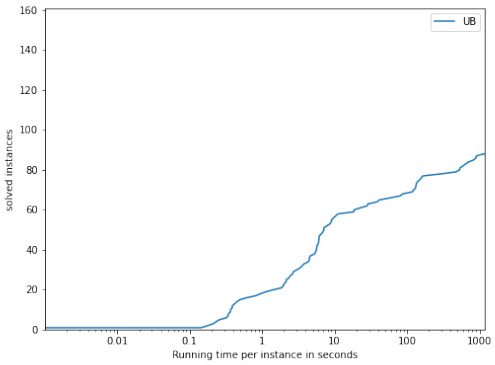
Figure 4: Initial Candidate Set Reduction - solved instances depending on running time per instance.

The evaluation of the solved instances for $k \in \{1, \ldots, 7\}$ (Figure 4a) by the initial candidate reduction improvement (IR) shows that the algorithm solves most instances within 1 second per instance. More precisely, IR solves 26 instances within 1 second, 35 within 30 seconds, 37 within 1 minute and 44 within 20 minutes. Thus, the algorithm solves only approximately one third of the inputs as well. It is noticeable that this improvement has not greatly improved the running time compared to the brute-force-algorithm. Also the number of solved instances remains mostly the same. For $k \in \{1, \ldots, 7\}$ a few more instances are solved, but for $k \in \{8, \ldots, 15\}$ (Figure 4b) IR still solves only 8 instances. However, it is noticeable that the running time has decreased for larger k, since all 8 instances could now be solved within 1 minute. The Table 1, in Section 3 underlines that IR alone does not bring a great running time improvement. This is

predictable, because IR only decreases the candidate set once before the subtrees are generated. Thus, almost all subsets are generated.



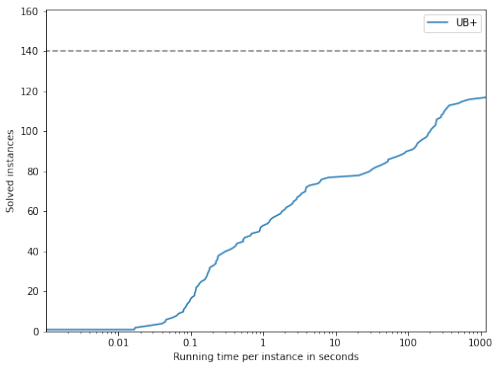
a) $k \in \{1, \ldots, 7\}$



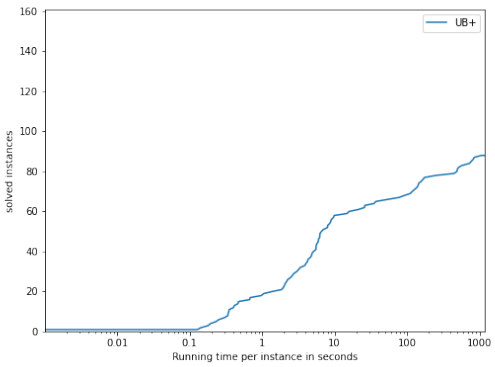
b) $k \in \{8, \ldots, 15\}$

Figure 5: Sum Upper Bound - solved instances depending on running time per instance.

The improvement that is tested next is the sum upper bound, in Section 3 abbreviated as UB, provides the first major improvement in running time. This improvement can be seen when looking at Figures 5 a and b. These figures show the running time per instance of UB. The evaluation shows that UB solves 52 instances within 1 second, 78 within 30 seconds, 84 within 1 minute and 115 within 20 minutes, for $k \in \{1, \ldots, 7\}$. This means that UB already computes 82% of the instances, with $k \in \{1, \ldots, 7\}$. For $k \in \{8, \ldots, 15\}$ the algorithm solves 18 instances within 1 second, 63 within 30 seconds, 66 within 1 minute and 88 within 20 minutes. Thus, UB solves for $k \in \{8, \ldots, 15\}$ 55% of all instances. Table 1 also indicates a major running time improvement, as many more instances can now be solved in the same time. In addition, UB reaches the limit of $k = 15$ with 9 instances. A large improvement of the running time by adding an upper bound was to be expected, since thereby a lot of subsets may be discarded.



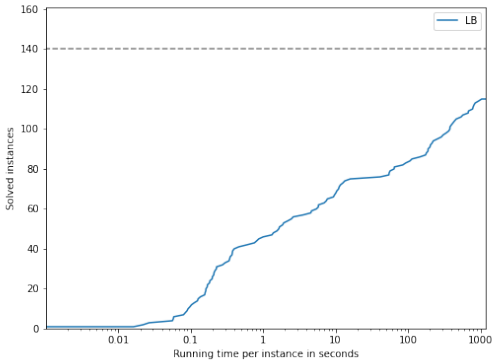
a) $k \in \{1, \ldots, 7\}$



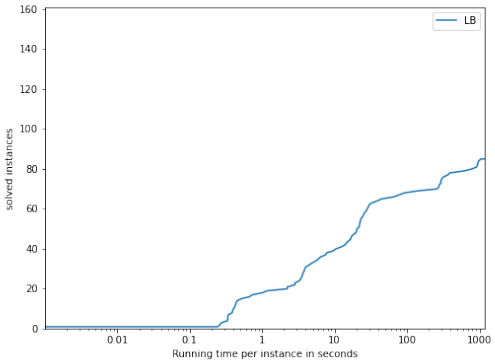
b) $k \in \{8, \ldots, 15\}$

Figure 6: Union Upper Bound - solved instances depending on running time per instance.

The test of the union upper bound improvement, in Section 3 abbreviated as UB+, are evaluated in Figure 6 and Table 1. At first glance, on the given data in Figure 6, one does not see any significant running time improvement due to the union upper bound. If we take a look at Table 1, we also see that UB+ solves just as many instances as UB, which only uses the simple sum upper bound. A closer look at the values does not seem to show any significant improvement either. For $k \in \{1, \ldots, 7\}$, UB+ solves 53 instances within 1 second, 80 instances within 30 seconds, and 86 instances within 1 minute. And for $k \in \{8, \ldots, 15\}$, UB+ solves 18 instances within 1 second, 63 instances within 30 seconds, and 66 instances within 1 minute.



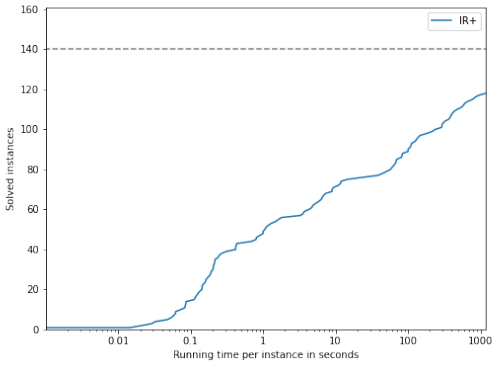
a) $k \in \{1, \ldots, 7\}$



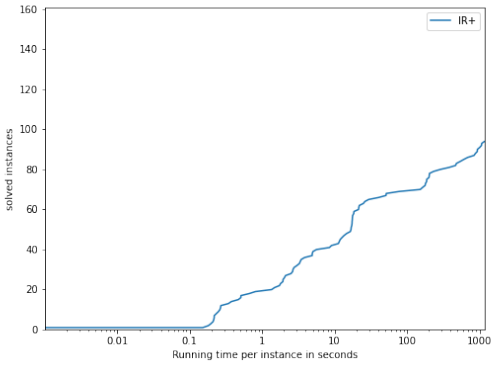
b) $k \in \{8, \ldots, 15\}$

Figure 7: Lower Bound - solved instances depending on running time per instance.

The situation is similar, when adding the lower bound. In Table 1 it is evident that three fewer instances can be solved now that the lower bound has been added. The increased running time is due to the computation of the greedy solution, which is then used as the lower Bound. However it seems like the lower bound does not increase the running time. This may be due to the algorithm quickly finding a big lower bound by itself due to C being sorted descendingly by undominated neighborhood.



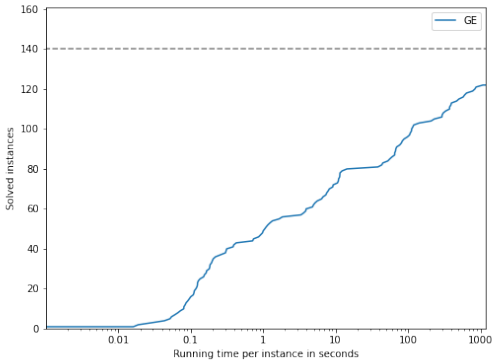
a) $k \in \{1, \ldots, 7\}$



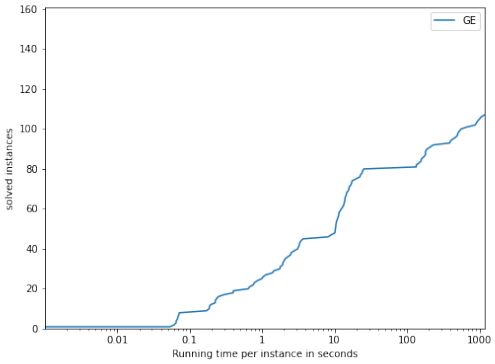
b) $k \in \{8, \ldots, 15\}$

Figure 8: Internal Candidate Set Reduction - solved instances depending on running time per instance.

With the internal candidate set reduction (Figure 8), however, one can again see a difference. This is especially noticeable if a look at Table 1 is taken. There, new solvable k-values are now reached for some instances. Furthermore, a deeper look into the data reveals that for $k \in \{1, \ldots, 7\}$ with 118 solved instances already more than 84% of the instances can be solved in less than 20 minutes. For $k \in \{8, \ldots, 15\}$, however, the algorithm solves 94 instances (52%) in 20 minutes.



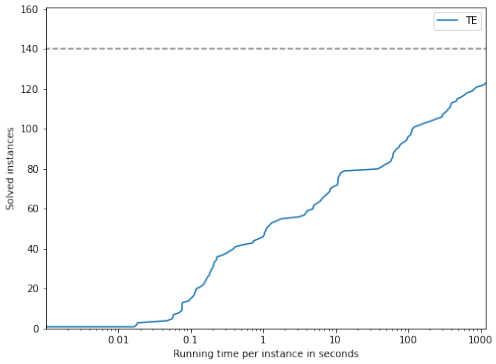
a) $k \in \{1, \ldots, 7\}$



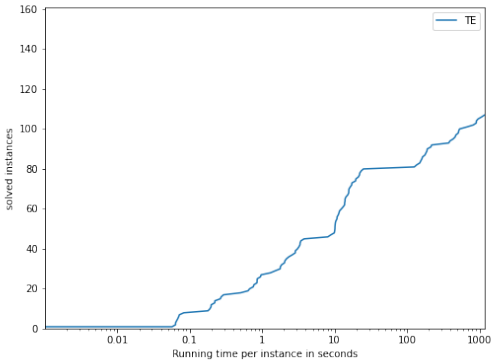
b) $k \in \{8, \ldots, 15\}$

Figure 9: Greedy Extension - solved instances depending on running time per instance.

The improvement that is added next is the greedy Extension, in Section 3 abbreviated as GE. A look at Figure 9 reveals that GE was able to solve new instances especially for $k \in \{8, \ldots, 15\}$. But also for $k \in \{1, \ldots, 7\}$ GE finds new solutions, which a look at Table 1 reveals. This improvement is due to an early updated solution being found often by the greedy extension. By this a better lower bound was obtained and consequently further search tree nodes were excluded by the upper bound. The algorithm now solves 88% of the instances for $k \in \{1, \ldots, 7\}$ and 67% of the instances for $k \in \{8, \ldots, 15\}$.



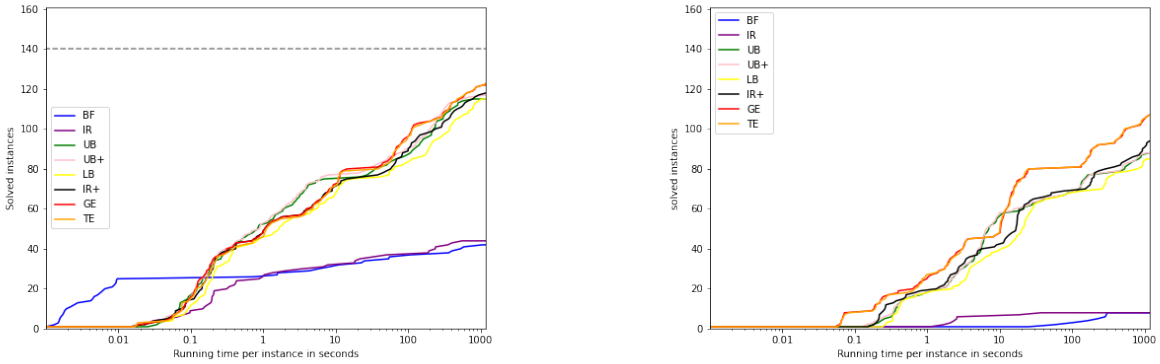
a) $k \in \{1, \ldots, 7\}$



b) $k \in \{8, \ldots, 15\}$

Figure 10: Trivially-Optimal Extension - solved instances depending on running time per instance.

Both, Table1 and Figure 10 do not seem to indicate any significant improvement by adding the trivially-optimal extension, which is abbreviated as TE in Section 3. The TE Algorithm also solves 88% of the instances for $k \in \{1, \ldots, 7\}$ and 67% of the instances for $k \in \{8, \ldots, 15\}$. Thus adding the trivially-optimal extension TE does not substantially improve the running time for the given instances. However, this outcome was to be expected, since lucky solution covers a special case that would only be applied on a few graphs. Nevertheless it remains reasonable to use this improvement, since this check is free due to the previous computation of greedy extension.



a) $k \in \{1, \ldots, 7\}$ b) $k \in \{8, \ldots, 15\}$

Figure 11: All Improvement-Algorithms - solved instances depending on running time per instance.

Figure 11 shows the running time performance of the individual improvements. The remarks made above are underlined once again in this graph. It is now more obvious to see that the initial candidate set reduction promotes an improvement of the running time, especially for larger ks. As expected, the Upper Bound shows the greatest improvement, which could be surpassed by the UB+ for some instances. The slight deterioration of the running time due to the introduction of the lower bound, which is then somewhat improved again by the internal candidate set reduction, is also clearly visible on this graph. Finally, the same running time of GE and TE exceeds all other improvements. Thus TE provides the best algorithm and is referred to as PDS algorithm (the pseudo code is presented in the appendix) in the further course of the work.

6 Integer Linear Programming as an Alternative

In order to compare the previously presented algorithm to a competitor, an integer linear programming formulation is introduced below. Integer linear programming (ILP) is a technique optimized by numerous heuristics to solve NP-hard algorithms. It is based on formulating the problem as a mathematical optimization problem in which some or all variables are restricted to be integers. An ILP always consists of a set of variables, an objective function over these variables that is either minimized or maximized, and linear constraints that restrict the solution space. The special case of 0-1 integer linear programming, in which all variables may only take the values 0 or 1 is particularly suitable for graph problems, since the decision whether a certain vertex is part of the result set or not can easily be modelled as a binary decision variable [Mar10]. Integer programming is NP-complete. In particular the special case of 0-1 integer linear programming, in which unknowns are binary, and only the existence of a solution that satisfies all constraints must be checked, is one of Karp's 21 NP-complete problems [Kar71].

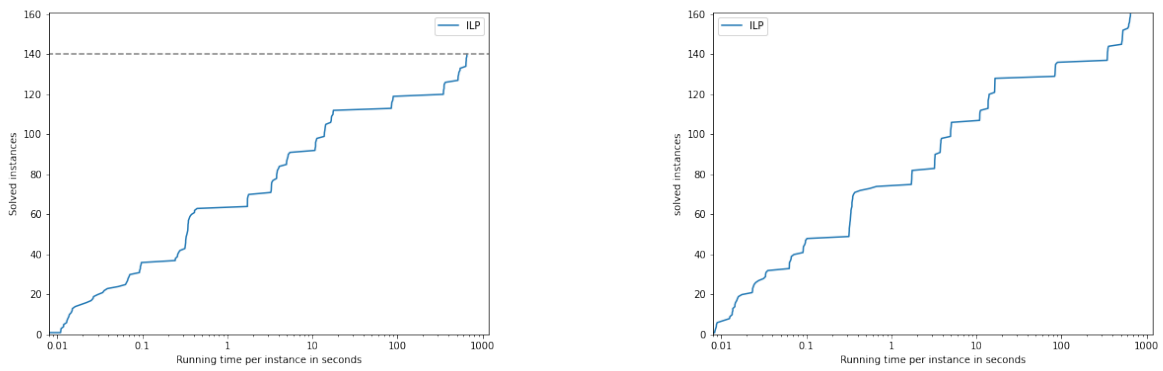
6.1 ILP Formulation

To solve an ILP, it must first be formulated. For this purpose, a binary variable x_v is introduced for each vertex $v \in V$. The variable x_v is assigned 1 if and only if $v \in D$, provided that D is the sought set for the partial dominating set problem. The problem can then be solved by maximizing the objective function z under the following constraints:

$$\begin{array}{lll} \text{Maximize} & z = \sum_{v \in V} c_v & \\ \text{s.t.} & c_v \leq \sum_{w \in N[v]} x_w & \forall v \in V \\ & k \geq \sum_{v \in V} x_v & \forall v \in V \\ & x_v, c_v \in \{0, 1\} & \forall v \in V \end{array}$$

This formulation solves the PARTIAL DOMINATING SET as follows. The sum to be maximized computes the size of the vertices covered by the partial dominating set. For this, a variable c_v is introduced, which may be assigned 1 if the vertex v is covered by the dominating set. This is ensured by the first constraint, since this sets c_v to 1 exactly when one of its neighbors or itself is in the dominating set. Since the PARTIAL DOMINATING SET requests a set of size at most k, the second constraint ensures that the solution set consists of at most k vertices. And finally, the third constraint restricts the variables used to be binary. Therefore the solution set D consists of the vertices $v \in V$ for which $x_v = 1$.

6.2 Evaluation

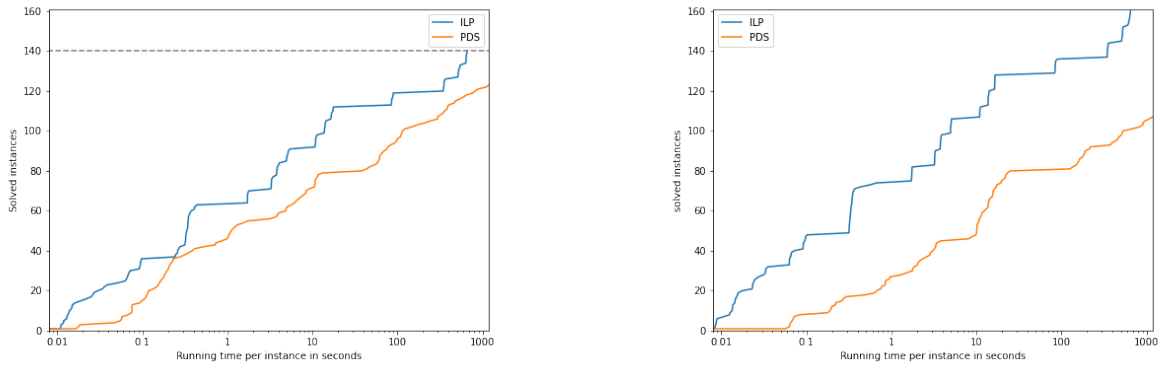


a) $k \in \{1, \ldots, 7\}$ b) $k \in \{8, \ldots, 15\}$

Figure 12: ILP - solved instances depending on running time per instance.

The ILP algorithm solves for $k \in \{1, \ldots, 7\}$ and for $k \in \{8, \ldots, 15\}$ all of the input instances in less than 20 minutes per instance. This can also be seen in Table 1 in Section 3. It is noticeable that the graph in Figure 12 grows in a step-like manner. It should be noted that the running time of ILP solutions strongly depends on the number of decision variables and constraints. In the ILP formulation from Section 6.1, the number of decision variables is in $\mathcal{O}(n)$ and the number of constraints is in $\mathcal{O}(n)$. It is evident that these numbers are independent of k. Thus it may be expected that the running time of this algorithm is not strongly dependent on k. This is confirmed by the experiments. Consequently, the graph in Figure 12 is so step-shaped, because the running time is mainly dependent on the instance and not on k. Thus, the 15 different instances with the same graph but different k are often solved at very similar times. This statement is further emphasized by a closer look at the data. Since ILP solves for $k \in \{1, \ldots, 7\}$ 63 instances (45%) within 1 second, 112 instances (80%) within 30s , and 112 instances (80%) within 1 minute. For $k \in \{8, \ldots, 15\}$ the algorithm solves 74 instances (46%) within 1 second , 128 instances (80%) within 30s , and 128 instances within (80%) 1 minute. Notably, the ratios are approximately the same for $k \in \{1, \ldots, 7\}$ and $k \in \{8, \ldots, 15\}$.

7 Comparison of ILP and PDS



a) $k \in \{1, \ldots, 7\}$ b) $k \in \{8, \ldots, 15\}$

Figure 13: ILP vs PDS - solved instances depending on running time per instance.

In this section the two algorithms presented in Section 5 and Section 6 are compared in regards to their running time. For this purpose, we consider Table 1 of Section 3, which indicates for each instance which $k \in \{1, \ldots, 15\}$ was the highest k that the respective algorithms could solve in 20 minutes. We also look at the plots in Figure 13. In these plots the ratio of solved instances in 20 minutes to running time per solved instance is shown for each algorithm. Figure 13 shows that the ILP algorithm produces better results for $k \in \{1, \ldots, 7\}$ (Figure 13a) and for $k \in \{8, \ldots, 15\}$ (Figure 13b) than the PDS algorithm, which contains all presented improvements. It can be seen, both in the figure and in the table, that for $k \in \{1, \ldots, 7\}$ and $k \in \{8, \ldots, 15\}$ ILP solves 100% of the tested instances, while for $k \in \{1, \ldots, 7\}$ PDS could only solve 88% and for $k \in \{8, \ldots, 15\}$ only 66% of the instances in 20 minutes. This correlation can also be read from Figure 12, as PDS comes much closer to ILP in a) than in b). In conclusion, ILP provides a faster solution for the PARTIAL DOMINATING SET than PDS, especially for larger values of k, since this algorithm, as described in Section 6.2, does not exhibit a strong dependence on k.

8 Conclusion

In this work a basic searchtree algorithm is presented in Section 4 to compute an exact solution to the PARTIAL DOMINATING SET problem. In Section 4 several improvements were introduced to decrease the running time of the searchtree algorithm by decreasing the number of subtrees that are pursued. In the further course of the work, an ILP formulation was then introduced in order to compare it to the searchtree algorithm.

In conclusion, the basic searchtree algorithm, called brute-force-algorithm, has been substantially improved by the improvements presented in Section 5. Considering Table 1, it can be seen that the brute-force-algorithm could solve $k = 15$ only for 1 graph (5%), failed to compute a solution for $k < 9$ for 19 graphs (95%), and for 11 graphs (55%) it could only compute solutions for $k = 1$. These values are exceeded highly in all categories by the PDS algorithm, which contains all the improvements from Section 5. For example, PDS finds a partial dominating set with $k = 15$ for 12 graphs (60%) and fails to find a solution for $k > 10$ in only 5 graphs (25%). This performance boost is especially due to the Upper Bound from Section 5.2.1 and the greedy extension from Section 5.3.1, reducing the running time substantially.

The Lower Bound seems to have slightly degraded the running time due to the time needed for its computation, although it was often equal to the number of covered vertices in the solution. Furthermore, it does not seem to have a big impact on the number of discarded subtrees. Looking at Table 1, the effect can be seen directly. The UB+ algorithm solves the graph "soc-advogato" for $k = 11$. While the LB algorithm, in which the only change is the addition of the lower bound, fails to solve this instance. A similar behaviour can be observed for the graph "bio-dmela".

Moreover, it should be noted that the graphs with lower density could be solved by the improvement sum upper bound (UB). While UB could only solve graphs G with density $d(G) > 0.1$, if $k > 10$ or the graph is very small ($|V| < 50$). Whereas the PDS algorithm could solve more instances for graphs G with density $d(G) > 0.1$. These instances could be solved once the improvement greedy extension has been added.

In Section 6, an ILP formulation for the PARTIAL DOMINATING SET is presented. This formulation is then implemented as a Gurobi script in Python in order to be tested by the Gurobi solver. The tests reveal that the ILP algorithm appears to be mostly independent of k.

In Section 7 it has been established that the ILP algorithm provides a better implementation than the PDS algorithm especially for larger k. As an outlook for further improvements of the PDS-Algorithm in addition to those from Section 5, a dynamic sorting of the list C can be considered. This could be done, similarly to the internal candidate reduction from Section 5.1.2, only on the first and second neighborhood of the newly added element, since the other elements do not change with respect to their undominated neighborhood. Thereupon, the newly sorted elements would have to be merged with the rest of C again. In addition, the algorithm can still be further improved in terms of memory requirements and a more accurate upper bound. In addition to this the ILP-Formulation could be enhanced, for example by providing the solver with a good heuristic solution.

References

[Bra13] Andreas Brandstädt. *Graphen und Algorithmen*. Springer, 2013.

[CTB15] Alina Campan, Traían Marius Truta, and Matthew Beckerich. "Efficient approximation algorithms to determine minimum partial dominating sets in social networks". In: *2015 IEEE International Conference on Big Data (Big Data)*. 2015, pp. 2390–2397.

[Die17] Reinhard Diestel. "Extremal Graph Theory". In: *Graph Theory*. Springer, 2017, pp. 173–207.

[Fei98] Uriel Feige. "A threshold of ln n for approximating set cover". In: *Journal of the ACM* 45.4 (1998), pp. 634–652.

[Gou12] Ronald Gould. *Graph theory*. Courier Corporation, 2012.

[HHS98] Teresa W. Haynes, Stephen T. Hedetniemi, and Peter J. Slater. *Fundamentals of domination in graphs*. Vol. 208. Pure and applied mathematics. Dekker, 1998.

[JJR20] MA Jayaram, Gayitri Jayatheertha, and Ritu Rajpurohit. "Time Series Predictive Models for Social Networking Media Usage Data: The Pragmatics and Projections". In: *Asian Journal of Research in Computer Science* (2020).

[Kar71] Richard M Karp. "A simple derivation of Edmonds' algorithm for optimum branchings". In: *Networks* 1.3 (1971), pp. 265–272.

[Kar72] Richard M. Karp. "Reducibility among Combinatorial Problems". In: *Complexity of Computer Computations: Proceedings of a symposium on the Complexity of Computer Computations, held March 20–22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, and sponsored by the Office of Naval Research, Mathematics Program, IBM World Trade Corporation, and the IBM Research Mathematical Sciences Department*. Springer US, 1972, pp. 85–103.

[KMR07] Joachim Kneis, Daniel Mölle, and Peter Rossmanith. "Partial vs. Complete Domination: t-Dominating Set". In: *SOFSEM 2007: Theory and Practice of Computer Science*. Ed. by Jan van Leeuwen et al. Springer Berlin Heidelberg, 2007, pp. 367–376.

[Kun13] Jérôme Kunegis. "Konect: the koblenz network collection". In: *Proceedings of the 22nd international conference on world wide web*. 2013, pp. 1343–1350.

[Mar10] François Margot. "Symmetry in Integer Linear Programming". In: *50 Years of Integer Programming 1958-2008: From the Early Years to the State-of-the-Art*. Springer Berlin Heidelberg, 2010, pp. 647–686.

[NA16] Jose C. Nacher and Tatsuya Akutsu. "Minimum dominating set-based methods for analyzing biological networks". In: *Methods* 102 (2016), pp. 57–63.

[OTA21] Simon T Obenofunde, Olivier Togni, and Wahabou Abdou. "Optimised disjoint virtual backbone algorithms for wireless sensor networks". In: *IET Wireless Sensor Systems* 11.5 (2021), pp. 219–232.

[RA15] Ryan A. Rossi and Nesreen K. Ahmed. "The Network Data Repository with Interactive Graph Analytics and Visualization". In: *AAAI*. 2015.

[Sha+03] J.A. Shaikh et al. "New metrics for dominating set based energy efficient activity scheduling in ad hoc networks". In: *28th Annual IEEE International Conference on Local Computer Networks, 2003. LCN '03. Proceedings*. 2003, pp. 726–735.

[SS10] Suk Jai Seo and Peter J Slater. “Open neighborhood locating dominating sets.” In: *The Australasian Journal of Combinatorics* 46 (2010), pp. 109–120.

[Tur09] Volker Turau. *Algorithmische Graphentheorie (2. Aufl.)* Oldenbourg, 2009, p. 17.

[WCX09] Feng Wang, Erika Camacho, and Kuai Xu. “Positive influence dominating set in online social networks”. In: *International Conference on Combinatorial Optimization and Applications*. Springer, 2009, pp. 313–321.

Appendix

The following Algorithms represent the PDS algorithm mentioned in Section 5 and following. It calls the individual improvements and processes them to find the optimal partial dominating set $\leq k$.

Algorithm 10: PDS

Data: Graph $G = (V, E)$, Integer k.
Global Variables: lower bound l_b, current solution D.
Result: Partial Dominating Set D of size at most k.

```
S ← {};
C ← Initial Reduction(G,k);
l_b ← Lower Bound(G, k) - 1;
if |C| ≤ k then
    return C;
end
Improved Build Subsets(S, C, k);
return D;
```

Algorithm 11: Improved Build Subsets

Data: Set S, List C, Integer k.
Global Variables: lower bound l_b, current solution D.

```
for i ← 0 to |C| do
    w ← C[i] ;
    C ← C \ {w};
    S ← S ∪ {w};
    if |S| = k then
        if N[S] > l_b then
            D ← S;
            l_b ← |D| ;
        end
        S ← S \ {w};
        continue;
    end
    else
        C ← C sorted descendingly by undominated neighborhood;
        C ← Internal Reduction(S, k, C, w);
        if |S| + |C| ≤ k then
            if N[S ∪ C] > l_b then
                D ← S ∪ C;
                l_b ← |D| ;
            end
            S ← S \ {w};
            continue;
        end
        if Extension Preperation(k, S, C) then
            S ← S \ {w};
            continue;
        end
        if Upper Bound(k, C, S) > l_b then
            Improved Build Subsets(S, C, k);
        end
    end
end
S ← S \ {w};
```

Statutory Declaration

I herewith formally declare that I have written the submitted dissertation independently.
I did not use any outside support except for the quoted literature and other sources mentioned in the paper. I clearly marked and separately listed all of the literature and all of the other sources which I employed when producing this academic work, either literally or in content.
I am aware that the violation of this regulation will lead to failure of the thesis.

Marburg, 08.04.2022

Place, Date

Signature