Algorithm Engineering for Generic Subset Optimization Problems

Masterarbeit
zur Erlangung des akademischen Grades
Master of Science (M.Sc.)
im Studiengang Informatik

Friedrich Schiller Universität Jena
Fakultät für Mathematik und Informatik

eingereicht von Henning Martin Woydt
geb. am 13.04.1998 in Mönchengladbach

Betreuer: Dr. Frank Sommer,
Prof. Dr. Christian Komusiewicz

Jena, 20. November 2023

Zusammenfassung

Die Maximierung submodularer monotoner Mengenfunktionen unter einer Größenbeschränkung taucht in vielen Problemen wie der optimalen Platzierung von Betriebsstätten oder im Clustering von Datenpunkten auf. Submodulare Funktionen zeigen einen stetig abnehmenden Ertrag, während monotone Mengenfunktionen nie an erreichtem Wert verlieren. Aufgrund der $\mathcal{NP}$-Härte des Problems fokussieren die meisten Arbeiten sich auf das Lösen durch Approximationen. Dennoch ist in vielen Bereichen eine exakte Lösung gewollt, wie zum Beispiel bei der Platzierung von Krankenhäusern und Arzt-Praxen.

Weil exakte Algorithmen üblicherweise nur für spezielle Probleme entwickelt werden, ist die Forschung für das generelle Problem noch nicht sehr weit. In dieser Arbeit entwerfen und implementieren wir einen effizienten Branch-and-Bound Algorithmus der jegliche submodulare monotone Funktion unter einer Größenbeschränkung maximieren kann. Wir nehmen dafür an, dass wir nur durch eine Black-Box Zugriff auf die Mengenfunktion f haben und diese für jede Menge S den Wert $f(S)$ zurückgeben kann. Unser Algorithmus verwendet mehrere Reduktionsregeln, die es ihm erlauben große Teile des Suchraumes zu entfernen. Zusätzlich gestalten wir die Suche des Algorithmus effizient da er immer in die Richtung des Suchraumes expandiert, in der die beste Lösung erwartet wird. Die Kosten und Genauigkeit dieser Suche kann durch Lazy Evaluation kontrolliert werden.

Um unseren Algorithmus zu evaluieren, nutzen wir die Probleme Group Closeness Centrality, Partial Dominating Set und k-Medoid Clustering. Wir werten aus welche Reduktionsregel den Suchraum am besten beschränkt. Zusätzlich vergleichen wir mehrere Konfigurationen für Lazy Evaluation um zu evaluieren welche in der Praxis am meisten Zeit spart. Unsere Experimente zeigen, dass unser Algorithmus um mehrere Größenordnungen schneller ist, als der aktuelle State-of-the-Art Algorithmus [73] für das größenbeschränkte Maximierung's Problem. Die schnellste Konfiguration unseres Algorithmus hält die nötige Arbeit an jedem Knoten gering, während gleichzeitig eine präzise Suchstrategie benutzt wird.

Abstract

The Cardinality Constrained Maximization problem for submodular monotone set functions arises in many problems like Group Closeness Centrality and k-Medoid Clustering. Submodular set functions encode a diminishing return, while monotone set functions never decrease in value. Due to the $\mathcal{NP}$-hard nature of Cardinality Constrained Maximization, most works focus on approximating good solutions. However, exactly solving the problem is required in many fields, like Healthcare Facility Location.

As exact algorithms are usually designed for one specific problem, there has not been much research for the general problem. In this work, we design and implement an efficient branch-and-bound algorithm to maximize arbitrary submodular monotone set functions under a cardinality constraint. We assume that we only have access to the set function f via a block-box oracle, which can return the value $f(S)$ for any arbitrary set S. Our algorithm utilizes multiple reduction rules that enable us to prune large parts of the search space. Additionally, we design an efficient branching strategy that traverses the search space in a gradient-like manner, always expanding into the part of the search space that most likely contains a maximizing solution. The cost and accuracy of the branching strategy can be controlled via Lazy Evaluations.

To evaluate our developed algorithm, we consider the problems Group Closeness Centrality, Partial Dominating Set, and k-Medoid Clustering. We evaluate which reduction rules prune the search space the best. Additionally, multiple configurations for Lazy Evaluation are compared to each other to determine which configuration saves the most time in practice. Our experiments show that our algorithm is multiple magnitudes faster than the current State-of-the-Art algorithm for Cardinality Constrained Maximization [73]. The overall fastest configuration of our algorithm keeps the required work at each node to a minimum, while still deploying an accurate branching-strategy.

Contents

1 Introduction

The Warehouse Problem Assume you own a company and want to distribute materials to each of the federal capitals of Germany. You have the budget to build $k = 3$ warehouses in different cities. These warehouses will supply the city they are located in but also have to supply all cities without a warehouse, which induces travel costs. The locations of the warehouses should be selected such that the summed travel cost to supply all cities without a warehouse is minimized. For simplicity, assume that traveling on a highway induces a cost of 1 unit. Figure 1 shows a graph of Germany with its $n = 16$ capitals and (approximately) the highways between them. Consider, for example, having one warehouse each in Bremen, Schwerin, and Stuttgart. Which of these warehouses should supply Erfurt? The warehouses in Bremen will only induce a cost of 2 units to supply Erfurt, while Schwerin and Stuttgart both induce a cost of 3. Therefore, the warehouse in Bremen should supply Erfurt in this example. If we repeat this process for all cities without a warehouse and sum up the costs, we know the total cost for this choice of locations. Based on this, we can rank each combination of three locations and determine the combination with the lowest travel distance. In total, $\binom{16}{3} = 560$ combinations exist, and in this case, one optimal combination of the three cities is Hamburg, Dresden, and Mainz, which achieves a cost of 11 units.

The scenario described above is an instance of the Group Closeness Centrality problem. First, let us formulate the *group farness* function and then define the problem.

Definition 1.1 (Group Farness)**.** Let $G = (V, E)$ be a connected undirected graph. Let $\operatorname{dist}(u, v)$ be the distance of a shortest path between the vertices u and v. Let $S \subseteq V$ be a subset of vertices. The *group farness* of S is

$$f_{\mathrm{GF}}(S) \coloneqq \sum_{v \in V \setminus S} \min_{u \in S} \operatorname{dist}(u, v).$$

In other words, the group farness of a set S is the sum of the minimum distances of each vertex to any vertex in S. Note that the function value of the empty set is not properly defined, so we define it in this work as $f_{\mathrm{GF}}(\emptyset) \coloneqq |V|^2$. Based on group farness, we can now properly define the problem.

Group Closeness Centrality
Input: A connected undirected graph $G = (V, E)$ and an integer $k \in \mathbb{N}$.
Task: Find a set $S \subseteq V$ with $|S| = k$ that has the lowest group farness.

This problem was introduced by Chen et al. [14] in the analysis of social networks, and they also proved that it is $\mathcal{NP}$-hard. Therefore, unless $\mathcal{P} = \mathcal{NP}$, there is no efficient algorithm that can solve the problem exactly on all instances. Chen et al. [14] showed that the function is monotone (decreasing) and supermodular (we will discuss these properties in more detail in Section 2). In the example, the monotone (decreasing) property reflects that your costs will shrink when you build an extra warehouse in a city. The supermodularity property reflects that if you build a warehouse in a city, the

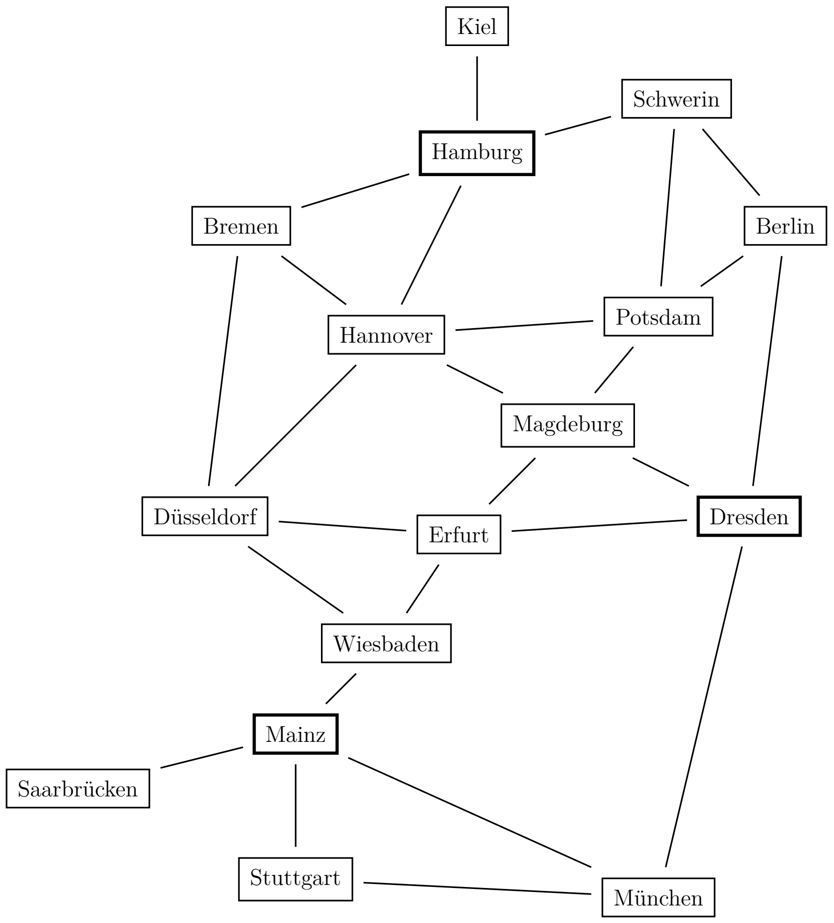


Figure 1: A graph showing the capitals of the 16 German federal states. The edges between the capitals represent highways that connect the cities. Hamburg, Dresden, and Mainz are optimal locations to place three warehouses.

benefit of building another warehouse in a different city will get smaller than before you built the first warehouse. Variants of Group Closeness Centrality have been used to measure the voltage stability in electric power grids [1] and in the analysis of protein structures [13]. Staus et al. [69] developed branch-and-bounds algorithms and ILP formulations to solve the problem exactly.

The Group Closeness Centrality problem can be considered a special version of the Facility Location problem [21, 77]. This problem asks: Given n locations, where should k facilities be built so that a utility function is minimized (or maximized)? There are many domains where the Facility Location problem arises and many more concrete utility functions that can be optimized. For example, the Healthcare Facility Location problem [4] is concerned with placing emergency and non-emergency facilities to cover most residents. Research in the Sensor Placement problem [18] asks to place k sensors such that an optimal data acquisition is guaranteed. The Hub Location problem [5, 12, 20] arises in transportation and communication and has a more economic goal. That is, costs should be minimized while gains should be maximized. Solving Hub Location problems has a long history, as the first instance was already studied in 1909 by Weber and Friedrich [75], who were concerned with placing a warehouse such that the distance to the customers is minimized. Wolf [77] give a comprehensive overview of most Facility Location problems. The Facility Location problem is known to be $\mathcal{NP}$-hard [21, 35].

With this example, we want to show two important things. First, there are set functions that have a naturally diminishing return. These functions often arise naturally and have an intuitive behavior in the context of optimization. It is often the case that the decisions we make (for example, placing facilities) will influence the state of the universe (for example, the distance to the closest facility). Other decisions we could make then lose their benefit as they would have influenced a similar part of the universe. Optimizing which decision should be made, such that a metric is optimized, is, therefore, hard, as each decision influences the other. Secondly, there are many fields where such functions arise, as we will see in the next two examples.

The Marketing Problem Let us consider an example of viral marketing [41]. You are writing the book "Algorithmische Netzwerkanalyse" and want to market it to enthusiastic computer science students. Classic marketing, like TV commercials or billboards, is not the right fit, as it does not reach your potential customers on a large scale. Therefore, you directly influence students with free book copies, hoping they advertise the book through mouth-to-mouth advertising. You assume that if you give a copy to a student, they will present it to their friends, so the student and all of their friends will know of your new book. Figure 2 shows the social network of students that we consider for this example. If you give one of your copies to, for example, Emma, then Emma, Emilia, Noah, and Mia would know of your book. How should you distribute $k = 4$ copies of the book to the $n = 20$ students, such that the maximum number of students knows of your book? In total, there are $\binom{20}{4} = 4\,845$ different combinations of four people that can be considered. In this case, the best group would be to give a copy each to Matteo, Mia, Luca, and Henry. Your advertisement would reach 18 of the 20 students, which is the maximum number of students you can reach with 4 copies.

This problem is known as Partial Dominating Set. Let us again formulate this problem in mathematical terms to analyze it. We will first define the *closed neighborhood* of a vertex to determine the *vertex domination number* of a vertex set. Then, we can properly define the problem.

Definition 1.2 (Closed Neighborhood)**.** Let $G = (V, E)$ be an undirected graph. The *closed neighborhood* of a vertex v is

$$N[v] := \{v\} \cup \{u \mid \{u, v\} \in E\}.$$

In other words, the closed neighborhood of v includes all vertices that are incident with v and v itself. We also say that a vertex u is *dominated* by a vertex v if $u \in N[v]$.

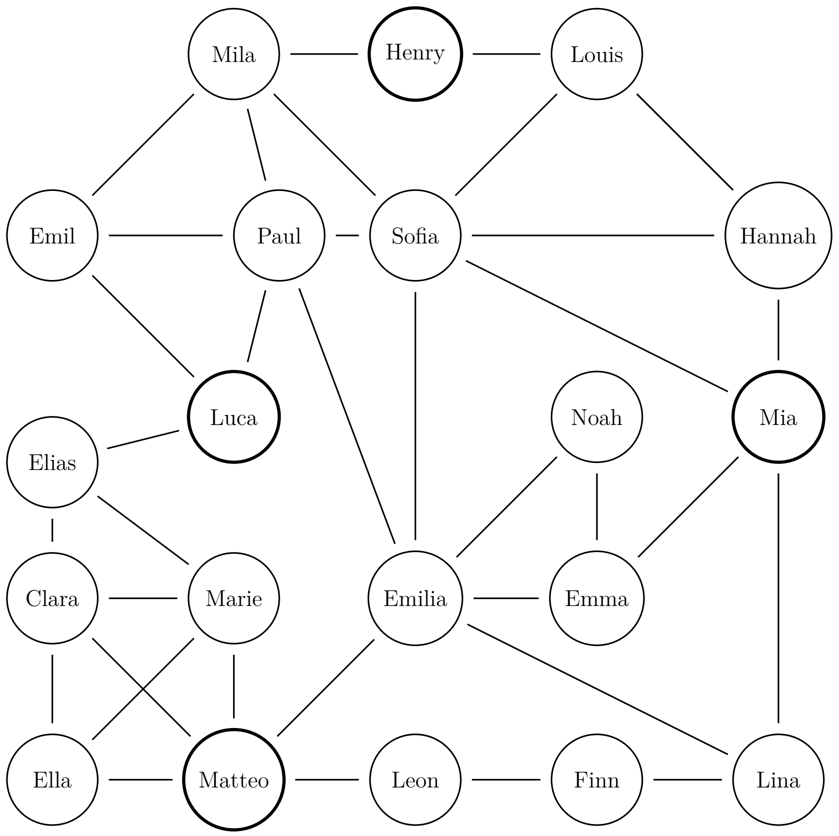


Figure 2: A graph showing 20 students and their friendships in a social network. If you give a free copy of your book to Matteo, Mia, Luca, and Henry, you will maximize the number of students that are reached by the advertisement.

Definition 1.3 (Vertex Domination Number)**.** Let $G = (V, E)$ be an undirected graph. The *vertex domination number* of $S \subseteq V$ is

$$f_{\mathrm{D}}(S) \coloneqq |\bigcup_{v \in S} N[v]|.$$

The vertex domination number is the size of the neighborhood of set S, that is, it counts how many vertices are dominated by at least one vertex $v \in S$. Note that the vertex domination number is submodular and monotone (increasing), in contrast to the group farness function, which is supermodular and monotone (decreasing). Monotonicity ensures, in this example, that if you give an additional copy to a student, you will not reach fewer students with your advertisement. Submodularity reflects that if you give a copy to a student, the effectiveness of also giving a copy to one of their friends or friends of friends will be reduced.

Partial Dominating Set
Input: An undirect graph $G = (V, E)$ and an integer $k \in \mathbb{N}$.
Task: Find a $S \subseteq V$ with $|S| = k$ that has the largest vertex domination number.

The famous Dominating Set problem is a special case of this problem. It asks: Given a graph G and an integer k, is there a set S with $|S| \leq k$ and $f_{\mathrm{D}}(S) = |V|$? Since Dominating Set is $\mathcal{NP}$-complete, so is Partial Dominating Set $\mathcal{NP}$-complete. Jiang and Zheng [37] provide an implementation to exactly solve Minimum Dominating set (determine the smallest k). Dominating sets are, for example, used in routing [70, 78], document summarization [67, 80] and sensor networks [56].

The Medication Problem As the last example of this introduction, let us consider a clustering problem as. You developed a new medication and want to test it for side effects. As tests are expensive, you only want to pick a small representative group of three people to test the medication on. Figure 3 shows 30 applicants with their normalized heights and weights. You can measure the similarity between two applicants based on their normalized heights and normalized weights. You want to select three persons so that all other persons are similar to one of these three. This is done by maximizing the sum of similarities or equivalently minimizing the sum of differences. The red dots will lead to a minimal sum of difference.

This problem is known as k-Medoid Clustering. Let us first define the *clustering cost* and then the problem.

Definition 1.4 (Clustering Cost)**.** Let $X = \{x_1, x_2, \ldots, x_n\}$ be a set of n data points and each $x_i \in \mathbb{R}^D$. Let $d(x_i, x_j)$ be a distance function, and $S \subseteq X$ be a subset of the data points. We define the *clustering cost* of S with distance functions d as

$$f_{\mathrm{CC}}(S) := \sum_{i=1}^{n} \min_{x_j \in S} d(x_i, x_j).$$

Note that the clustering cost is not defined for the empty set $\emptyset$. We, therefore, define $f_{\mathrm{CC}}(\emptyset)$ as the sum of all distances between all data points. Like group farness, the clustering cost is supermodular and monotone (decreasing). Monotonicity again reflects that the clustering cost only improves if we add more data points to S. In this specific example, it translates to that the group will be represented better. Supermodularity reflects that the benefit of selecting an additional person will only decrease if we add other applicants to the group. We may now define the k-Medoid Clustering problem.

k-Medoid Clustering
Input: A set X consisting of n datapoints, a distance function $d(x_i, x_j)$, and an integer $k \in \mathbb{N}$.
Task: Which set $S \subseteq X$ with $|S| = k$ has the lowest clustering cost?

Determining the exact solution to k-Medoid Clustering is $\mathcal{NP}$-hard [72], similar to the other proposed problems. Therefore, most researched clustering algorithms are approximate and do not guarantee an exact solution. PAM (Partitioning Around Medoids) [39] is the most well-known approximation algorithm. Clustering is used

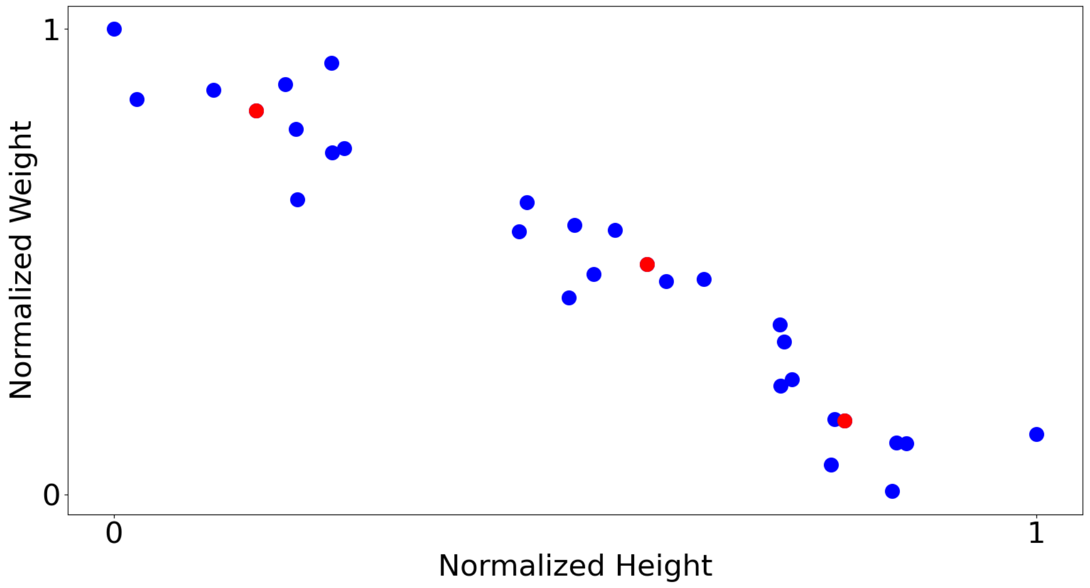


Figure 3: A plot showing the 30 applicants with their normalized heights and weights. Selecting the three applicants marked in red will best represent the group.

in many research fields, for example, in image segmentation [19, 60], information retrieval [8, 79] and data mining [7, 55]. Oyewole and Thopil [59] give a comprehensive overview of clustering and its applications.

1.1 Cardinality-Constrained Maximization

All three problems presented above have the following in common: (a) They are interested in finding a subset $S \subseteq \mathcal{U}$ of size k of some ground set $\mathcal{U}$, (b) they are interested in a solution that has the optimal value of a utility function, (c) the function to maximize is submodular monotone increasing or correspondingly, the function to minimize is supermodular monotone decreasing.

Supermodular monotone decreasing set functions are closely related to submodular monotone increasing set functions, as the negation of one results in the other and vice versa (this will be shown in Lemma 2.2). Therefore, the maximization problem for submodular monotone increasing functions is equivalent to the minimization problem for supermodular monotone decreasing functions. From now on, we will mainly talk about the maximization of submodular monotone set functions, but all results carry over to the minimization of supermodular monotone decreasing set functions. A set function f fulfills submodularity and monotonicity if

$$f(A \cup \{e\}) - f(A) \geq f(B \cup \{e\}) - f(B)$$

for $A \subseteq B \subseteq \mathcal{U}$ and $e \in \mathcal{U}$. The properties will be discussed in more detail in Section 2.

The three examples presented above and a plethora of other problems are special cases of the following problem.

Cardinality-Constrained Maximization
Input: A universe $\mathcal{U}$ of items, a submodular and monotone set function $f:\ 2^{\mathcal{U}} \rightarrow \mathbb{R}$ and an integer $k \in \mathbb{N}$.
Task: Find a set $S \subseteq \mathcal{U}$ with $|S| = k$ that has the greatest value $f(S)$?

The problem takes as input any arbitrary submodular monotone function f. For the context of this work, we will assume that f is a black-box oracle that supplies us with the value $f(S)$ for any set S. The Cardinality-Constrained Maximization problem arises in many fields, for example, in viral marketing [41], information gathering [48], image segmentation [10, 36, 45], document summarization [53] and scheduling [71]. Submodularity also often arises in the Facility Location problem [21, 77]. Additionally, many computer science problems exhibit submodularity or supermodularity and monotonicity. This list contains some of the problems that utilize a submodular or supermodular monotone function:

- Partial Dominating Set [43].
- Group Closeness Centrality [14].
- Partial Vertex Cover [44] that asks to choose k vertices such that the maximum number of edges are incident with the vertices.
- Hitting Set problem [3] (see Definition 2.9).
- Maximum Coverage [2] problem that asks to select k sets from a set family $\{S_1, S_2, \ldots, S_n\}$, such that the union of the k sets is maximized (a generalization of Partial Dominating Set).
- Weighted Coverage [42, 47] that additionally associates a weight with each element and asks to select a coverage such that the sum of weights is maximized.
- 0/1 Knapsack [51] (see Section 2.1).

The problems discussed are all $\mathcal{NP}$-hard. Consequently, Cardinality-Constrained Maximization is $\mathcal{NP}$-hard as well.

A naive algorithm to solve the problem exactly evaluates all $\binom{n}{k}$ combinations and, therefore, has a running time of $\mathcal{O}(n^k T(f))$ ($T(f)$ denotes the running time of f). Surprisingly, this running time is already close to worst-case optimal if the Exponential Time Hypothesis (ETH) [33] holds, which many researchers presume (see Theorem 2.1). This makes optimally solving these problems costly and often infeasible for large values of n.

Approximation Guarantee Due to its hardness, most work on solving the Cardinality-Constrained Maximization problem deals with approximating algorithms. Nemhauser et al. [57] were the first to investigate a greedy algorithm to approximately solve the problem. Their algorithm iterates k times across all n elements and chooses the element that maximizes the function in each iteration. They proved a $(1 - 1/e)$ approximation guarantee for this greedy algorithm and that there is no better guarantee for any polynomial-time algorithm unless $\mathcal{P} = \mathcal{NP}$ [47]. Vondrak [74] tightens the bound by considering a submodular monotone function's *total curvature* $c \in [0, 1]$. The total curvature of a monotone submodular function reflects how much the marginal gains of elements can decrease. Minoux [54] improved the speed of the greedy algorithm by using lazy evaluations. They exploited the submodularity property to determine whether an element has to be reevaluated in each iteration. The improvement results in the same set as the standard greedy algorithm but can be magnitudes faster [47].

In some fields, however, an approximation may not be desirable, and the user may be willing to invest much more time in finding an optimal solution to their problem. For example, in HEALTHCARE FACILITY LOCATION [4], non-optimal placement of facilities "are likely to be associated with increased morbidity and mortality"[4], so determining an optimal solution is highly important. In other fields, like SENSOR PLACEMENT problems [18] or FACILITY LOCATION problems, the chosen locations will likely be used for several years, if not decades. Not choosing the optimal locations can lead to years of non-optimal data acquisition or substantial economic losses.

Exact Algorithms To avoid these situations, algorithms that can solve the problem exactly are needed. Nemhauser and Wolsey [58] proposed the first algorithm in 1981 to solve the CARDINALITY-CONSTRAINED MAXIMIZATION problem optimally. They utilize a mixed integer programming (MIP) approach, which has an exponential number of constraints in its most basic form. To circumvent this, they start with fewer constraints and use their developed Constraint Generation Algorithm to add new constraints iteratively. This approach is, however, reported to be inefficient by both Kawahara et al. [40] and Uematsu et al. [73].

The next work on exactly solving the CARDINALITY-CONSTRAINED MAXIMIZATION problem was proposed by Kawahara et al. [40] in 2009. They dropped the requirement that the function has to be monotone and only focused on submodular functions. Their work uses an iterative cutting-plane method implemented by binary-integer linear programming (BILP). While their method can determine an optimal set in a finite amount of iterations, Uematsu et al. [73] report that the algorithm has to solve the BILP often as only one or two additional constraints are determined in each iteration.

Chen et al. [16] developed a framework in 2015 in which the user can tune the quality of the returned set and the time needed to compute it. This tradeoff can be made via a hyperparameter $\alpha \in [0, 1]$, with $\alpha = 0$ being a fast but inaccurate greedy approximation and $\alpha = 1$ being a slow but optimal solution. They utilize a A* search algorithm on the search space, always expanding the most prominent path.

The last publication (to our knowledge) on optimal submodular monotone function maximization with a cardinality constraint was by Uematsu et al. [73] and was published in 2020. They revisit the idea of Nemhauser and Wolsey [58] and their Constraint Generation Algorithm. They improve the Constraint Generation Algorithm by heuristically generating a set of feasible solutions that generate new constraints. Additionally, they deploy a local search to an improved branch-and-cut algorithm that improves the best-found lower bound.

As far as we are concerned, these are the only four publications that deal with exactly solving the Cardinality-Constrained Maximization problem. The works of Nemhauser and Wolsey [58], Kawahara et al. [40], Chen et al. [16] sadly do not conclude large experiments. The only work that has a larger experiment section is the work of Uematsu et al. [73]. They report that they can solve a version of the Facility Location problem with $n = 60$ locations and $k = 8$ in about 500 seconds. Our assumption is that such a small instance can be solved significantly faster.

Our Work We develop an efficient branch-and-bound algorithm based on the work of Staus et al. [69] that first explores the complete search space of all possible sets of size at most k. This algorithm is improved by using reduction rules to identify parts of the search space that do not hold a maximizing set. We also determine whether specific elements can be part of a maximizing solution, and if not, we can further reduce the search space. Additionally, we develop a method called Lazy Evaluation that controls the speed and accuracy of the branching strategy. We always assume that the set function f is only accessible via a black-box oracle.

The algorithm achieves a running time multiple magnitudes faster than the current State-of-the-Art algorithm for Cardinality Constrained Maximization. In general, the algorithm configuration that holds the required computational effort low while still managing an accurate branching strategy performs the best.

The rest of the thesis is structured as follows: Section 2 will define the properties required of the set functions. We will discuss how the modularity and monotonicity properties of the set function, influence the complexity of Cardinality Constrained Maximization. Additionally, we prove the theoretical worst-case complexity of $n^{o(k)}$ for the Cardinality-Constrained Maximization problem. Section 3 discusses the branch-and-bound algorithm proposed by Staus et al. [69]. In Section 4 we introduce new reduction rules and improvements to speed up the running time in practice. The developed algorithms are evaluated by experiments in Section 5. Section 6 gives an outlook on future work and concludes the thesis.

2 Preliminaries

In this section, we will discuss Subset Maximization and examine how different properties of the set function influence the complexity of the problem. Section 2.1 will discuss general set functions and the corresponding maximization problem. In Section 2.2, we will look more specifically at different properties of set functions and how they influence the complexity of the maximization problem. Lastly, in Section 2.3, we will see how Cardinality-Constrained Maximization differs from Subset Maximization, and we will also prove that its worst-case running time of $n^{o(k)}$ can not be improved (under standard complexity-theoretic assumptions).

2.1 Set Functions

Let $\mathcal{U}$ be a *universe* of n arbitrary items. The universe is also often called the *ground set. Set functions* are defined on collections of items of the universe.

Definition 2.1 (Set Function)**.** A function $f : 2^{\mathcal{U}} \to \mathbb{R}$ is a *set function.*

Here, the set $2^{\mathcal{U}}$ denotes the power set of U. We denote by $T(f)$ the running time of f[1]. Since this work aims to improve the running time of practical applications, we will assume that the set function is well-defined and computable on all inputs. This may not be given naturally every time, for example, in the introduction, we had to explicitly define the group farness and clustering cost on the empty set.

Although we are interested in Cardinality-Constrained Maximization, we will first discuss Subset Maximization.

Subset Maximization
Input: A set function $f : 2^{\mathcal{U}} \to \mathbb{R}$.
Task: Find a set $S \subseteq \mathcal{U}$ with the greatest value $f(S)$ of all sets.

We call a solution S to the problem a *maximizing set.* In general, the solution to the maximization problem does not have to be unique, as there can be multiple maximizing sets $S_1, S_2, \ldots, S_m$.

What is the complexity of determining a maximizing set? If no properties of f are known, we would need to evaluate all 2^n possible subsets to be confident that we have found a maximizing set. Therefore, at least a time complexity of $\Theta(2^n T(f))$ is needed to maximize the function. We transform the maximization problem into a decision problem to better identify its complexity class.

Subset Decision
Input: A set function $f : 2^{\mathcal{U}} \to \mathbb{R}$ and a value $x \in \mathbb{R}$.
Task: Is there a set $S \subseteq \mathcal{U}$ with $f(S) \geq x$?

[1]More precisely one would write $T_f(n)$ with $n = |\mathcal{U}|$, since the complexity of f depends on n

As by the definition of the $\mathcal{NP}$ class [6], a problem is in $\mathcal{NP}$ if a possible solution to the problem can be verified in polynomial time. Since we are only interested in efficiently computable set functions f, we assume that f is polynomial-time computable and the Subset Decision problem is, therefore, in $\mathcal{NP}$.

In Section 1t, we have already shown three examples of maximization problems. Let us here additionally consider the well-known 0/1 Knapsack problem [51].

Definition 2.2 (Knapsack Value and Weight)**.** Let $\mathcal{I} = \{(v_1, w_1), \ldots, (v_n, w_n)\}$ be a set of n items. Each item i has an associated value v_i and an associated weight w_i. The *knapsack value* of set S is

$$\sum_{i \in S} v_i$$

and the *knapsack weight* is

$$\sum_{i \in S} w_i.$$

The 0/1 Knapsack problem is now how to maximize the value, while not exceeding the weight constraint. Burglars are often confronted in the real-world with the 0/1 Knapsack problem. They want to optimize the sum of value they can steal, but their knapsack can carry only a limited amount of weight. Luckily[2], they know that the problem is $\mathcal{NP}$-hard [27], so they will most of the time solve the problem by a faster greedy approximation, instead of determining the exact solution. Note that the problem also arises outside of criminal activities, for example, in resource allocation tasks [11]. We can formally define the problem as follows:

0/1 Knapsack
Input: A set I of n items and a weight threshold $W \in \mathbb{R}$.
Task: Find the set $S \subseteq \mathcal{I}$ with the greatest knapsack value, while the knapsack weight of S does not exceed the weight threshold W.

Notice that the knapsack value does not behave similarly to the functions of the other problems. If we add an item to the set S, the contribution to the value of the set is guaranteed to stay unmodified for all other items. The knapsack value is, therefore, modular (see Definition 2.6). Also, notice that instead of limiting the size of our solution, we limit the potential solutions by their weight. Such a constraint is called the *Knapsack constraint* [47].

2.2 Properties of Set Functions

Now, we want to define some essential properties of set functions and discuss how they impact the maximization problem. We can best identify these properties with the *marginal gain* of an element or set. We will in this work use the notation provided by Krause and Golovin [47].

[2]at least for the burglars

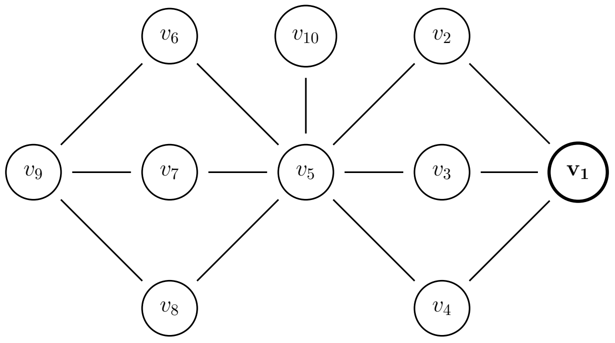


Figure 4: A graph with ten vertices. Consider that we want to optimize the vertex domination number (used in the Partial Dominating Set problem). If we enlarge the set $A = \{v_1\}$ (marked in bold) by a vertex v, then the marginal gain of all other vertices will decrease (except for the vertex pair v_9 and v_{10}).

Definition 2.3 (Marginal Gain)**.** Let $f : 2^{\mathcal{U}} \to \mathbb{R}$ be a set function, $S \subseteq \mathcal{U}$, and $e \in \mathcal{U}$. We define

$$\Delta_f(e \mid S) \coloneqq f(S \cup \{e\}) - f(S)$$

as the *marginal gain* of f at S with respect to e. Analogously, we define

$$\Delta_f(A \mid S) \coloneqq f(S \cup A) - f(S)$$

for sets $A \subseteq \mathcal{U}$ as the *marginal gain* of f at S with respect to the set A.

The marginal gain is also often called the *discrete derivative* [47] following terminology from Analysis. When f is evident from the context, we will drop the subscript and write $\Delta(e \mid S)$. The marginal gain is an important metric, allowing us to identify properties of f easily.

Submodularity The most important property for this work is *submodularity.* We will first define the property and afterward consider Example 2.1 to illustrate what submodularity entails.

Definition 2.4 (Submodularity)**.** A function $f : 2^{\mathcal{U}} \to \mathbb{R}$ is *submodular* if for every $A \subseteq B \subseteq \mathcal{U}$ and $e \in \mathcal{U} \setminus B$ it holds that

$$\Delta(e \mid A) \geq \Delta(e \mid B).$$

Equivalently[3], a function $f : 2^{\mathcal{U}} \to \mathbb{R}$ is *submodular* if for every $A, B \subseteq \mathcal{U}$

$$f(A \cap B) + f(A \cup B) \leq f(A) + f(B).$$

Example 2.1. Figure 4 shows a graph with ten vertices. Consider the submodular vertex domination number f_{D} for the Partial Dominating Set problem. The base

[3]It is well-known that these two definitions are equivalent, so we do not give a proof here.

set is $A = \{v_1\}$ with value $f_{\mathrm{D}}(A) = 4$ as vertices v_1 to v_4 are dominated by A. How does the marginal gain of the vertices v_5 to v_{10} change as we enlarge set A? For example, let us consider the vertices v_5 and v_9. The marginal gain of each vertex decreases if the other vertex is added to the set A, since

$$5 = \Delta(v_5 \mid A) \geq \Delta(v_5 \mid A \cup \{v_9\}) = 2$$

and

$$4 = \Delta(v_9 \mid A) \geq \Delta(v_9 \mid A \cup \{v_5\}) = 1.$$

In this example, the marginal gain of every vertex v_5 to v_{10} drops when A is enlarged by one of the six vertices. Only the vertices v_9 and v_{10} do not influence the marginal gain of each other because their (closed) neighborhoods are disjoint.

As another example, consider the group farness f_{GF} used in the Group Closeness Centrality problem and again the base set A in Figure 4. The base set A has a group farness value of $f_{\mathrm{GF}}(A) = 22$. The marginal gain of v_6 is $\Delta(v_6 \mid A) = 12$ and the marginal gain of v_8 is also 12. However, if we add v_6 to A, then the marginal gain of v_8 decreases to only a value of $\Delta(v_8 \mid A \cup \{v_6\}) = 2$. The same holds if we consider the marginal gain of v_6 after adding v_8 to A. In this example, the marginal gain of every vertex is decreased if we add any vertex to A.

Submodular functions encode a *diminishing return*, often naturally arising when dealing with set functions: The marginal gain of any element e will gradually decline as elements are added to the set A. Elements with a large marginal gain can quickly lose much of their gain, depending on the elements in A. This makes finding a set A with a maximum value challenging, as each time we add an element to the set, the marginal gain of all other elements can vanish. The difficulty can also be seen in the complexity of maximizing submodular functions: Although submodular set functions have more structure than arbitrary set functions, it is not easier to maximize them, and the problem remains in $\mathcal{NP}$ (instead of getting easier). You could, for example, encode the MAXSAT [30] problem as submodular set function, which makes clear that arbitrary submodular set functions are not easy to maximize. Note that *minimizing* submodular functions is achievable in a polynomial number of evaluations of f [34, 66].

In contrast to general set functions, submodularity grants an upper bound on a set's marginal gain by summing its elements' marginal gain.

Lemma 2.1 (Submodularity Upper Bound)**.** Let $A, B \subseteq \mathcal{U}$, and let $f : 2^{\mathcal{U}} \to \mathbb{R}$ be a submodular set function. Then

$$\sum_{e \in B} \Delta(e \mid A) \geq \Delta(B \mid A).$$

Proof. Let $B = \{e_1, e_2, \ldots, e_m\}$. Then the left side of the equation expands to

$$\Delta(e_1 \mid A) + \Delta(e_2 \mid A) + \ldots + \Delta(e_m \mid A).$$

Now let us consider the right side:

$$
\begin{aligned}
&\Delta(B \mid A) \\
&= \Delta(\{e_1, e_2, \ldots, e_m\} \mid A) \\
&= f(\{e_1, e_2, \ldots, e_m\} \cup A) - f(A) \\
&= f(\{e_1\} \cup \{e_2, \ldots, e_m\} \cup A) - f(\{e_2, \ldots, e_m\} \cup A) + f(\{e_2, \ldots, e_m\} \cup A) - f(A) \\
&= \Delta(e_1 \mid \{e_2, \ldots, e_m\} \cup A) + \Delta(\{e_2, \ldots, e_m\} \mid A).
\end{aligned}
$$

We can split B into $\{e_1\}$ and $\{e_2, e_3, \ldots, e_m\}$, and can iteratively split all other e_i from the remaining set. In the end, we get the equation

$$\Delta(B \mid A) = \Delta(e_1 \mid \{e_2, \ldots, e_m\} \cup A) + \Delta(e_2 \mid \{e_3, \ldots, e_m\} \cup A) + \ldots + \Delta(e_m \mid A).$$

We compare the left side of the original equation with the right side of the equation component-wise. In each comparison, we get

$$\Delta(e_i \mid A) \geq \Delta(e_i \mid \{e_{i+1}, e_{i+2}, \ldots, e_m\} \cup A)$$

due to submodularity. Therefore, $\sum_{e \in B} \Delta(e \mid A) \geq \Delta(B \mid A)$ holds. $\square$

We can give an upper bound on the marginal gain for every subset $B \subseteq \mathcal{U}$ at $A \subseteq \mathcal{U}$ if we have the marginal gain of each element $e \in \mathcal{U}$. Calculating all n marginal gains grants us the ability to give an upper bound for all 2^n possible sets. Lemma 2.1 is used extensively later in Sections 3 and 4 to prove the correctness of our numerous heuristics. However, we are not able to obtain a theoretical reduction of running time from this lemma.

Supermodularity In Section 1, we mentioned that submodular and supermodular are closely related to each other, as the negation of a submodular function results in a supermodular function and vice versa [9]. Here, we want to briefly discuss the property of *supermodularity*.

Definition 2.5 (Supermodularity)**.** A function $f : 2^{\mathcal{U}} \to \mathbb{R}$ is *supermodular* if for every $A, B \subseteq \mathcal{U}$

$$f(A \cap B) + f(A \cup B) \geq f(A) + f(B).$$

An equivalent[4] definition is

$$\Delta(e \mid A) \leq \Delta(e \mid B).$$

for $A \subseteq B \subseteq \mathcal{U}$ and $e \in \mathcal{U} \setminus B$.

The marginal gain of elements will only increase as set A grows, making it difficult to consistently find elements with a small marginal gain consistently. Contreras and Fernández [20] use supermodular functions to solve the Hub Location problem [5, 12]. Hub Location problems involve placing hubs in a transportation network to reduce costs. Goldengorin et al. [28] developed the Data-Correcting Algorithm to determine an exact and approximate minimum of supermodular set functions.

[4]The equivalency proof needed is analog to the proof for submodularity.

Lemma 2.2. A set function $f : 2^{\mathcal{U}} \to \mathbb{R}$ is a submodular function if and only if the set function $f' : 2^{\mathcal{U}} \to \mathbb{R}$ with $f'(S) \mapsto -f(S)$ is a supermodular set function.

Proof. Let $A \subseteq B \subseteq \mathcal{U}$ and $e \in \mathcal{U} \setminus B$. We need to show $\Delta_f(e \mid A) \geq \Delta_f(e \mid B)$ if and only if $\Delta_{f'}(e \mid A) \leq \Delta_{f'}(e \mid B)$.

$$\begin{aligned}\Delta_f(e \mid A) \geq \Delta_f(e \mid B) &\Leftrightarrow f(A \cup \{e\}) - f(A) \geq f(B \cup \{e\}) - f(B)\\ &\Leftrightarrow -f(A \cup \{e\}) - (-f(A)) \leq -f(B \cup \{e\}) - (-f(B))\\ &\Leftrightarrow f'(A \cup \{e\}) - f'(A) \leq f'(B \cup \{e\}) - f'(B)\\ &\Leftrightarrow \Delta_{f'}(e \mid A) \leq \Delta_{f'}(e \mid B)\end{aligned}$$

□

As previously mentioned, submodular functions can be minimized in polynomial time [34, 66], and therefore supermodular functions can be maximized in polynomial time [9]. The problem of maximizing submodular functions is also equivalent to the problem of minimizing supermodular functions, so both problems are $\mathcal{NP}$-hard. Therefore, all discussed properties and results on submodular functions can be transformed to fit supermodular functions.

Modularity Set functions that fulfill both submodularity and supermodularity are called *modular* functions.

Definition 2.6 (Modularity)**.** A set function $f : 2^{\mathcal{U}} \to \mathbb{R}$ is *modular* if it is submodular and supermodular. That is for every $A, B \subseteq \mathcal{U}$

$$f(A \cap B) + f(A \cup B) = f(A) + f(B). \tag{1}$$

What implications does modularity have? Consider the case $B = \{e\}$ and the two cases $e \in A$ and $e \notin A$. The case $B = \{e\}$ simplifies Equation 1 to

$$f(\{e\}) + f(A) = f(A) + f(\{e\}), \tag{2}$$

which does not grant any new insight into modular functions. The second case, however, simplifies Equation 1 to

$$f(\emptyset) + f(A \cup \{e\}) = f(A) + f(\{e\}). \tag{3}$$

If we assume $f(\emptyset) = 0$, then Equation 3 further simplifies to $f(A \cup \{e\}) = f(A) + f(\{e\})$. It shows that for every set $A \subseteq \mathcal{U}$, the value of $f(A)$ is simply the sum of its element function values, that is,

$$f(A) = \sum_{e \in A} f(e).$$

Modular functions are, therefore, often called *linear functions*. Note that if $f(\emptyset) \neq 0$, then

$$f(A) = \sum_{e \in A} (f(\{e\}) - f(\emptyset)) = -|A| \cdot f(\emptyset) + \sum_{e \in A} f(\{e\}) = f_0 + \sum_{e \in A} f(\{e\}).$$

We have already seen that the knapsack value is a modular function. For modular set functions, the marginal gain of $e \in \mathcal{U}$ is a constant value independent of any set. To determine a maximizing set, it is thus sufficient to identify the elements with a positive marginal gain and add them to the resulting set. Therefore, a maximizing set can be determined in $\mathcal{O}(n \cdot T(f))$ time. Since we assume that f is polynomial-time computable, the decision problem will also be polynomial-time computable and will belong to $\mathcal{P}$. Modularity truly simplifies the complexity of the maximization problem in comparison with arbitrary or submodular functions.

Monotonicity Submodular, supermodular, and modular functions specify how the marginal gain of an element (or set) with respect to a set A changes if A is enlarged. However, the properties do not specify how large the marginal gain of an element is in general. Set functions that fulfill the *monotonicity* property have a minimum marginal gain of 0 for each element with respect to each set.

Definition 2.7 (Monotonicity)**.** A function $f : 2^{\mathcal{U}} \to \mathbb{R}$ is *monotone increasing* if for every $A \subseteq \mathcal{U}$ and every $e \in \mathcal{U}$

$$\Delta(e \mid A) \geq 0.$$

Note that in the context of supermodularity, one would define *monotone decreasing* as $\Delta(e \mid A) \leq 0$ and also note that in literature, the term *non-decreasing* is also often used to highlight that the marginal gain will at least be zero. Since we will only consider the submodular case, we will use *monotone* synonymous with monotone increasing.

Monotonicity is an interesting property as the set functions closely resemble convexity, which makes it easier to maximize and analyze them (in the unconstrained maximization problem). The three examples in Section 1 are monotone functions. The Max-SAT problem [30], that asks for the maximum number of simultaneously satisfiable clauses of a CNF can also be expressed as a monotone set function (however, a constraint is needed so the same variable is not selected as a positive and negative literal). The maximization (without any additional constraint) of monotone set functions is truly trivial.

Lemma 2.3. Let $f : 2^{\mathcal{U}} \to \mathbb{R}$ be a monotone set function. The universe $\mathcal{U}$ is a maximizing set of Subset Maximization.

Proof. There is a set A that is a maximizing set. If $A = \mathcal{U}$, then the lemma holds. Otherwise $A \subseteq \mathcal{U}$, but every element $e \in \mathcal{U}$ has $\Delta(e \mid B)$ for all sets $B \subseteq \mathcal{U}$. therefore adding all elements $e \in \mathcal{U} \setminus A$ to A will not decrease the score and $\mathcal{U}$ is a maximizing set. □

Finding the maximum set $S \subseteq \mathcal{U}$ is possible in $\mathcal{O}(n)$ time since we only need to add all elements to the resulting set $S = \mathcal{U}$. The complexity is not dependent on f anymore (in the context of unconstrained maximization). Note that monotone set functions will greatly suffer under the cardinality-constrained which will be introduced in Section 2.3. Monotonicity also allows for a lower bound of the marginal gain of a set.

Lemma 2.4 (Monotonicity Lower Bound)**.** Let $A, B \subseteq \mathcal{U}$, and let f be a monotone function. Then

$$\Delta(B \mid A) \geq \max_{e \in B} \Delta(e \mid A).$$

Proof. Let $B = \{e_1, e_2, \ldots, e_m\}$ and without loss of generality we assume $\Delta(e_1 \mid A) \geq \Delta(e_2 \mid A) \geq \ldots \geq \Delta(e_m \mid A)$. We have to show that $\Delta(B \mid A) \geq (e_1 \mid A)$. Since

$$\Delta(B \mid A) = \Delta(e_1 \mid A) + \Delta(e_2 \mid A \cup \{e_1\}) + \ldots + \Delta(e_m \mid A \cup \{e_1, e_2, \ldots, e_{m-1}\})$$

and f is monotone the lower bound is correct. □

Score Functions Submodularity and monotonicity are the properties we assume throughout the rest of this work. We combine submodular monotone functions in the term *score functions*.

Definition 2.8 (Score function)**.** A function $f : 2^{\mathcal{U}} \to \mathbb{R}$ is a *score function* if f is submodular and monotone. Equivalent, f is a *score function* if for all $A \subseteq B \subseteq \mathcal{U}$ and $e \in \mathcal{U}$ the equation

$$\Delta(e \mid A) \geq \Delta(e \mid B)$$

holds.

The definition differs from the definition of submodularity (see Definition 2.4) in that we do not demand that $e \in \mathcal{U} \setminus B$. Previously, if $e \in B$ but $e \notin A$ then $\Delta(e \mid A) \geq \Delta(e \mid B)$ was not necessarily true. The marginal gain $\Delta(e \mid B)$ was 0 since e was already part of B, but $\Delta(e \mid A) \geq 0$ was not guaranteed. This case cannot occur anymore since each marginal is at least 0[5].

With both submodularity and monotonicity, we can bound the marginal gain of B with respect to A to

$$\sum_{e \in B} \Delta(e \mid A) \geq \Delta(B \mid A) \geq \max_{e \in B} \Delta(e \mid A).$$

Previously, only one bound could be applied. While we will only use the upper bound to prove the validity of heuristics (see Section 3.3), the additional lower bound can be helpful in the development of new heuristics: Normalize the range from the upper to the lower bound into the $[1, 0]$ interval, and we can determine the *degree* of diminishing return of B at A. A value close to 1 denotes that there is only a small diminishing return, while a value close to 0 denotes that there is a strong diminishing return. Such an analysis would be more challenging without monotonicity and the resulting lower bound. The time complexity of Subset Maximization for score functions is $\mathcal{O}(n)$, as every score function is monotone.

[5]The equivalncy is mentioned by Krause and Golovin [47] and we do not give a proof here.

2.3 Cardinality Constraint

Until now, we only discussed set function maximization without any constraints. However, in practical applications, one will often place a constraint on the solution space. In the 0/1 Knapsack problem, we placed the constraint that the summed weight of the items does not increase the given Knapsack threshold. In the introduction, we placed a size constraint on all discussed problems. These constraints are known as cost constraints, where we associate a cost with each item (or generally have a cost function on sets of items). Only sets that fulfill this cost constraint can be valid solutions to the maximization problem. We formally define the problem.

Cost-Constrained Maximization
Input: A set function $f : 2^{\mathcal{U}} \rightarrow \mathbb{R}$, a cost function $C : 2^{\mathcal{U}} \rightarrow \mathbb{R}$ and a cost threshold $c \in \mathbb{R}$.
Task: Find the set $S \subseteq \mathcal{U}$ with $C(S) \leq c$ that has the greatest value $f(S)$ of all sets.

The cost constraint is one of the most used constraints, as many real-world problems only have limited resources available to solve them. For example, Leskovec et al. [52] use a cost constraint when deploying sensors in a water distribution network to detect contamination. Singh et al. [68] also use a cost constraint in planning traveling routes for multiple robots.

In this thesis, we will use the *cardinality constraint*, a special case of the cost constraint, as a unit cost is associated with each element. The maximization problem is formulated as follows:

Cardinality-Constrained Maximization
Input: A set function $f : 2^{\mathcal{U}} \rightarrow \mathbb{R}$ and an integer $k \in \mathbb{N}$.
Task: Find the set $S \subseteq \mathcal{U}$ with $|S| \leq k$ that has the greatest value $f(S)$ of all sets.

As mentioned in Section 1, Dominating Set is one of the most well-known Cardinality-Constrained Maximization problems. The Hitting Set problem [3] (discussed at the end of this section) also is a Cardinality-Constrained Maximization problem. Furthermore, the Vertex Cover problem, the Set Packing problem, the Feedback Node Set problem and the Feedback Arc Set problem have a cardinality constraint. All these problems are listed in Karp's 21 $\mathcal{NP}$-complete problems [38]. Karp's list is by no means exhaustive, as many more cardinality-constrained problems exist.

Other interesting constraints exist, for example, *graph constraints* [47] and *matroid constraints* [47]. The graph constraint only allows for sets $S \subseteq V$ that, for example, form a path or a tree on a graph G. Krause et al. [50] use a graph constraint to place sensors in a wireless network. A Matroid [76] is a special set of subsets of the universe $\mathcal{U}$. Krause et al. [49] use a partition matroid constraint to maximize the location and the schedules of sensors in a wireless network.

Property of f	Subset Max.	Cardinality-Constrained Max.
arbitrary	$\mathcal{O}(2^n \cdot T(f))$	$\mathcal{O}(n^k \cdot T(f))$
submodular	$\mathcal{O}(2^n \cdot T(f))$	$\mathcal{O}(n^k \cdot T(f))$
modular	$\mathcal{O}(n \cdot T(f))$	$\mathcal{O}(n \cdot T(f) + n)$
monotone	$\mathcal{O}(n)$	$\mathcal{O}(n^k \cdot T(f))$
score function	$\mathcal{O}(n)$	$\mathcal{O}(n^k \cdot T(f))$

Table 1: Table showing the time complexity of maximizing a set function f for different properties and different maximization problems.

Time Complexities How do the complexities change with respect to the properties of f for the Cardinality-Constrained Maximization problem? Table 1 gives an overview of the complexity of the Subset Maximization and Cardinality-Constrained Maximization problem.

For arbitrary set functions, we determined a complexity of $\Theta(2^n T(f))$. With the cardinality constraint, all sets S with $|S| > k$ are not of interest anymore and therefore do not have to be evaluated. This reduces the complexity to $\mathcal{O}\left(\sum_{i=0}^{k} \binom{n}{i} \cdot T(f)\right)$. We estimate this running time roughly by $\mathcal{O}(n^k \cdot T(f))$.

We did not achieve a better complexity for submodular functions and the Subset Maximization than for arbitrary set functions. The same holds here as no algorithm with a faster running time is known and presumably does not exist (see Lemma 2.8). The time complexity is $\mathcal{O}(n^k \cdot T(f))$.

Modular set functions lead to a time complexity of $\mathcal{O}(n \cdot T(f))$ for the Subset Maximization problem. This complexity is only worsened a little bit for the Cardinality-Constrained Maximization problem, as the running time becomes $\mathcal{O}(n \cdot T(f) + n)$: After calculating all marginal gains, determine the element with rank k in the sorted list. Pass over the elements once more and collect all elements with a better rank. At the end, all collected elements with a positive marginal gain will be added to the final set. The cardinality constraint only gives an additional running time of $\mathcal{O}(n)$ to select the element of rank k and afterward collecting the correct elements.

Monotone set functions suffer the most from the introduction of the cardinality constraint. Previously, the universe $\mathcal{U}$ was guaranteed to be a maximizing set, but now, there is no clear rule on which elements form a maximizing set. There is no guaranteed behavior of how the marginal gain changes from $\Delta(e \mid A)$ to $\Delta(e \mid A \cup \{e'\})$ for any $A \subseteq \mathcal{U}$ and $e \neq e' \in \mathcal{U} \setminus A$, which makes it impossible to guarantee that a maximizing set has been found. In theory, the set S' with $|S'| = k - 1$ could have the lowest value of all sets of size $k - 1$, but the set $S = S' \cup \{e\}$ with $e \in \mathcal{U} \setminus S'$ could have the greatest value of all sets of size k. The property of monotonicity cannot exclude this case, and, therefore, we have to check all possible combinations. Like arbitrary and submodular functions, the running time is $\mathcal{O}(n^k \cdot T(f))$. However, monotonicity grants the property that a maximizing set will have a size of k (as long as $k \leq n$).

Lemma 2.5 (Maximizing set of size k)**.** Let $f : 2^{\mathcal{U}} \to \mathbb{R}$ be a monotone set function and $k \in \mathbb{N}$. Then the following equation holds:

$$f\left(\underset{S' \subseteq \mathcal{U}, |S'| \leq k}{\arg\max}\ f(S')\right) = f\left(\underset{S' \subseteq \mathcal{U}, |S'| = k}{\arg\max}\ f(S')\right).$$

Proof. Let S be a maximizing solution to the left maximization problem.

Case 1 $|S| = k$ **:** The set S is a solution for the right term, and the equation holds.

Case 2 $|S| < k$ **:** We know that for all elements $e \in \mathcal{U} \setminus S$, the marginal gain at S is 0, that is $\Delta(e \mid S) = 0$. This is true because negative marginal gains are not possible because of f's monotonicity, and marginal gains greater than 0 are also not possible because the set S would not be a maximizing set. Adding $k - |S|$ arbitrary elements to S will preserve the score and yield a solution S' of size k. Therefore, if a maximizing set with less than k elements exists, then a maximizing set with k elements exists. $\square$

To maximize monotone set functions, one only needs to evaluate sets with a size of k. While this reduces the number of function evaluations in practice compared to arbitrary set functions, which also need to evaluate all sets of size smaller than k, the worst-case time running time is still $\mathcal{O}(n^k T(f))$.

Score functions also have the same running time, as submodular monotone set functions have a complexity of $\mathcal{O}(n^k \cdot T(f))$, and the combination of both properties does not grant any theoretical reductions.

For the rest of the thesis, we focus our research on score functions instead of only submodular functions. The additional monotonicity property allows us to focus only on sets of size k, which makes a practical approach easier. However, if one wants to develop an algorithm for only submodular set functions, most of the discussed heuristics can also be applied as the heuristics do not depend on monotonicity.

Worst-Case Optimal Running Time As the last part of this section, we want to show that there (presumably) is no algorithm with a better running time than $n^{o(k)}$ (if $f \in \mathcal{P}$) that can solve the Cardinality-Constrained Maximization problem for arbitrary score functions [15]. It is enough to show that one score function exists for which we cannot derive a better running time. We use the Hitting Set problem [3], for which it is known that it cannot be solved in less than $n^{o(k)}$ time unless the Exponential Time Hypothesis (ETH) [33] fails. The Exponential Time Hypothesis has stronger implications than the hypothesis $\mathcal{P} \neq \mathcal{NP}$: The hypothesis $\mathcal{P} \neq \mathcal{NP}$ denies the existence of polynomial time algorithms for $\mathcal{NP}$-hard problems. For example, the well-known SAT problem could then not be solved in polynomial time. However, algorithms with sub-exponential running times, for example $\mathcal{O}(2^{\sqrt{n}})$, could still exist. The ETH also denies the existence of these algorithms. Certain problems, like SAT, can then not be solved faster than $\mathcal{O}(2^{o(n)})$. Another one of these problems is Hitting

SET. Let us first define the HITTING SET problem and then construct a submodular monotone function. Maximizing the function will solve the HITTING SET decision problem.

Definition 2.9 (Hitting Set)**.** Let $\mathcal{U}$ be a universe of items, $\mathcal{F}$ a collection of non-empty subsets, and $H \subseteq U$ a set. We call H a *hitting set* if $H \cap F_i \neq \emptyset$ for each $F_i \in \mathcal{F}$. We say F_i *is hit by* H if $H \cap F_i \neq \emptyset$.

HITTING SET
Input: A universe $\mathcal{U}$ of items, a collection of non-empty subsets $\mathcal{F}$, and an integer $k \leq |\mathcal{U}|$.
Question: Does a hitting set H of size at most k exist?

HITTING SET is a general case of DOMINATING SET. The non-empty subsets $\mathcal{F}$ are the closed neighborhoods of the vertices, and at most k vertices must be selected such that all neighborhoods are hit. A problem instance $(\mathcal{U}, \mathcal{F}, k)$ either has the answer `YES` if a hitting set H of size at most k exists or `NO` if no such set exists. Now, let us define a score function that models the problem.

Definition 2.10 (Hitting Set Score Function)**.** Let $\mathcal{U}$ be a universe of items, $\mathcal{F}$ a collection of non-empty subsets, and $H \subseteq \mathcal{U}$. The *hitting set score function* is defined as

$$f_{\text{HS}}(H) = \sum_{F \in \mathcal{F}} \min\{|H \cap F|, 1\}$$

The maximum achievable score is $|\mathcal{F}|$ because each term is either 0 if $H \cap F_i = \emptyset$ or 1 if $H \cap F_i \neq \emptyset$. Before showing its usefulness, let us first show that the function is submodular and monotone.

Lemma 2.6 (f_{HS} is submodular)**.** The hitting set score function is submodular.

Proof. We need to show that if $A \subseteq B \subseteq \mathcal{U}$ and $e \in \mathcal{U} \setminus B$, then

$$\Delta(e \mid A) \geq \Delta(e \mid B).$$

Let $F_A \subseteq \mathcal{F}$ be the collection of subsets hit by A, and let $F_B \subseteq \mathcal{F}$ be the collection of subsets hit by B. Let $F_e \subseteq \mathcal{F}$ be the collection of subsets hit by $\{e\}$. The marginal gain of an element a at S is defined as $\Delta(a \mid S) = f(S \cup \{a\}) - f(S)$ (see Definition 2.3). In this concrete context, it denotes the number of newly hit sets. These are the sets that $\{a\}$ hits, but S did not. We will show that adding e to A will at least hit the same number of new sets as adding e to B. There are four cases of how $F \in F_e$ might be contained in F_A and F_B.

Case 1: $F \in F_A \wedge F \in F_B$ **:** $A \cup \{e\}$ and $B \cup \{e\}$ already hit F.

Case 2: $F \in F_A \wedge F \notin F_B$ **:** Since $A \subseteq B$ and therefore $F_A \subseteq F_B$, the case $F \in F_A$ but $F \notin F_B$ can not occur.

Case 3: $F \notin F_A \wedge F \in F_B$ **:** $A \cup \{e\}$ will hit F, but $B \cup \{e\}$ will not.

Case 4: $F \notin F_A \wedge F \notin F_B$ **:** $A \cup \{e\}$ and $B \cup \{e\}$ will hit F.

In every case, if $B \cup \{e\}$ hits a new set, then $A \cup \{e\}$ will also hit the same set. Therefore, the marginal gain of e at A is at least the marginal gain of e at B. □

Lemma 2.7 (f_{HS} is monotone)**.** The hitting set score function is monotone.

Proof. We must show that if $A \subseteq \mathcal{U}$ and $e \in \mathcal{U}$, then $\Delta(e \mid A) \geq 0$. This is trivial as adding an element e to A cannot decrease the already achieved score of A. All previous sets that were hit by A will still be hit by $A \cup \{e\}$. At most, new sets hit by e can increase the score. □

We showed that the hitting set score function is submodular monotone and, therefore, it is a score function (see Definition 2.8). Now, let us show that maximizing the hitting set score function solves Hitting Set.

Lemma 2.8. Let $(\mathcal{U}, \mathcal{F}, k)$ be an instance of Hitting Set. Let

$$H = \underset{H' \subseteq \mathcal{U}, |H'| \leq k}{\arg\max} f_{\text{HS}}(H')$$

be a maximizing set of size at most k. If $f_{\text{HS}}(H) = |\mathcal{F}|$, then the instance is a `YES` instance. Otherwise, it is a `NO` instance.

Proof. Assume $(\mathcal{U}, \mathcal{F}, k)$ is a `YES`-instance and H is a corresponding hitting set of size at most k. Therefore, for each $F_i \in \mathcal{F}$, we know $H \cap F_i \neq \emptyset$. The function value $f_{\text{HS}}(H)$ will then be $|\mathcal{F}|$, as each term of the sum returns 1. The set H, therefore, is a maximizing set of the score function of size at most k.

Assume now that $(\mathcal{U}, \mathcal{F}, k)$ is a `NO`-instance. Then no hitting set exists, meaning each set H' has at least one $F' \in \mathcal{F}$ with $H' \cap F' = \emptyset$. Therefore, the function value $f_{\text{HS}}(H')$ must be smaller than $|\mathcal{F}|$ because at least the term $\min\{|H' \cap F'|, 1\}$ returns 0.

Therefore, maximizing f_{HS} and determining if a set H with $f_{\text{HS}}(H) = |\mathcal{F}|$ exists correctly decides Hitting Set. □

As previously mentioned, there is no $n^{o(k)}$ algorithm that can solve Hitting Set if the ETH holds [15]. If any algorithm could solve the Cardinality-Constrained Maximization problem for arbitrary score functions in less time, it could also maximize the hitting set score function, and we could decide Hitting Set, implying that the ETH fails. We capture this important fact in a theorem.

Theorem 2.1. Unless the Exponential Time Hypothesis fails, there is no algorithm that can solve the Cardinality-Constrained Maximization problem for arbitrary polynomial-time computabel score functions in $n^{o(k)}$ time.

This means that a brute-force algorithm that evaluates all $\binom{n}{k}$ subsets is already a worst-case optimal algorithm. However, worst-case optimal does not mean it cannot be improved in practice. The following sections discuss algorithms and reduction rules to optimize the running time of the Cardinality-Constrained Maximization problem for arbitrary score functions.

3 The Basic Search Algorithm

In this section, we want to introduce the Plain Search algorithm (Algorithm 1) and additional improvements made by Staus et al. [69], which we implement in the Basic Search algorithm (Algorithm 2). To give a quick recap, we want to solve the CARDINALITY-CONSTRAINED MAXIMIZATION problem

$$S = \underset{S' \subseteq \mathcal{U}, |S|=k}{\arg\max} f(S').$$

for arbitrary score functions f. From now on, we will assume that the universe $\mathcal{U}$ is not a set of arbitrary items, but the index set $[n] = \{0, 1, \ldots, n-1\}$.

In theory, maximizing across all k-sized sets will give a running time of $\mathcal{O}(n^k T(f))$, which (presumable) is close to the worst-case running time to solve the problem (see Theorem 2.1). However, just because the worst-case complexity is exponential, it does not mean the problem cannot be solved significantly faster for practical real-world problems. In this section, we will introduce two heuristics that guarantee that certain parts of the search space will not hold a maximizing set and, therefore, can be ignored. To apply these heuristics effectively, we need to search through the sets in a tree-like order, so we deploy a Set-Enumeration Tree (SE Tree) [63] as the basis for all algorithms.

3.1 Set-Enumeration Tree

A Set-Enumeration Tree (SE Tree) is a special tree data structure that makes it easy to iterate over all subsets of $[n]$ in an ordered fashion. To accomplish this, each node of the tree represents one unique subset of $[n]$ and additionally holds information to generate nodes with different subsets.

Definition 3.1 (Node)**.** A *node* $T = (S_T, C_T)$ holds a *working set* $S_T \subseteq [n]$ and a *candidate set* $C_T \subseteq [n]$. The sets have no common elements, so $S_T \cap C_T = \emptyset$.

The working set represents the node as it is the only node with this exact working set. We will prove this later in Lemma 3.1. The candidate set holds all candidates that still can be added to the working set to form a new working set. Note that although we define the candidate set as a set, we will use it more like a vector or a sorted set. The sets are called working set and candidate set because the search algorithm will add elements of the candidate set, called *candidates*, to the working set to generate sets of size k. By inheriting the working set and an additional node of the candidate set, we can define a *child*, the *children*, and the *descendants* of a node.

Definition 3.2 (Child)**.** Let T be a node with the working set S_T and candidate set $C_T = \{c_1, c_2, \ldots, c_m\}$. We define for each $1 \leq i \leq m$ the *child* T_i *of* T as

$$T_i = (S_T \cup \{c_i\}, C_T \setminus \{c_1, c_2, \ldots, c_i\}).$$

Definition 3.3 (Children)**.** Let T be a node with m candidates, that is, $C_T = \{c_1, c_2, \ldots, c_m\}$. We define the *children of* T as

$$\mathrm{Ch}(T) = \{T_i \mid 1 \leq i \leq m\}.$$

Definition 3.4 (Descendants)**.** Let T be a node. We define

$$\mathrm{Des}(T) = \mathrm{Ch}(T) \cup \left(\bigcup_{T_i \in \mathrm{Ch}(T)} \mathrm{Des}(T_i) \right)$$

as the *descendants* of T.

A child adopts the working set from its parent and one additional candidate c_i from the candidate set of its parent. The candidate set of the child holds all elements from the candidate set of the parent with a higher index than i. Choosing this subset will guarantee that no two nodes with the same working set exist (Lemma 3.1) and that all possible subsets are present in the tree (Lemma 3.2). A node T has exactly m different children. They all differ in the candidate added from the candidate set of T. The set $\mathrm{Des}(T)$ holds all nodes in the subtree of T but not T itself. Note that the candidate set of a child is always smaller than its parent's candidate set. At some point, the candidate set of all descendants will be empty, and no further children or descendants can be added to the set. Therefore, $\mathrm{Des}(T)$ is a finite set of nodes.

Now, we can define the *Set Enumeration Tree* and show that it is a suitable data structure for the Cardinality-Constrained Maximization problem.

Definition 3.5 (Set-Enumeration Tree)**.** A *Set-Enumeration Tree* (*SE Tree*) for $n \in \mathbb{N}$ consists of a root node R with $S_R = \emptyset$ and $C_R = \{0, 1, \ldots, n-1\}$, and all descendants of R.

The SE Tree is only helpful to the maximization problem if it holds every subset $S \subseteq [n]$ of size k. Additionally, it would be suitable for efficiency reasons that every subset is only present once, so we do not process the same set multiple times. Let us first prove that all the nodes in the SE Tree are unique and then that the SE Tree holds all possible subsets of $[n]$ and, therefore, all subsets of size k.

Lemma 3.1 (Unique Nodes)**.** Let R be the root of an SE Tree for $n \in \mathbb{N}$ elements. There are no two nodes $T_1, T_2 \in \mathrm{Des}(R) \cup \{R\}$ with $S_{T_1} = S_{T_2}$.

Proof. The root is unique because it is the only node with a working set of size 0. Therefore, we assume that $T_1, T_2 \in \mathrm{Des}(R)$. Assume that such two nodes, T_1 and T_2, exist. We will show that we will obtain a contradiction when we follow the path from these nodes to the root.

Let c_i be the element with the largest index in S_{T_1}. We know that the parent node T_1^* of T_1 has the working set $S_{T_1} \setminus \{c_i\}$. This is true due to Definition 3.2: If i were not the largest index, it would mean that another element $c_j \in S_{T_1}$ with $j > i$

exists. This element would have been added to the working set of some ancestor T' of T_1. But then c_i could not have been in the candidate set of T' since T' only holds candidates with a higher index. Therefore, c_i could also not be present in the working set of T_1. The candidate c_i is, therefore, the element with the largest index in S_{T_1}, and the parent of T_1 has the working set $S_{T_1} \setminus \{c_i\}$.

The same argument also applies to T_2 and its parent node T_2^*. The working set of T_2 also contains c_i, and the working set of its parent is $S_{T_2} \setminus \{c_i\}$. Therefore, the working set of T_1^* and T_2^* are the same, and their working set has one element less than that of their children.

We have two nodes with the same working set, so we can reapply the argument and deduce that their parents must have the same working set. After applying the argument a total of $|S_T| - 1$ times, we get to the point where the parent of T_1 and T_2 is the root $R = T_1^* = T_2^*$ because we reduce the size of the working set by one each iteration. However, we know that if T_1 and T_2 are children of the same node, they cannot have the same working set due to Definition 3.2. Therefore, two nodes with the same working set cannot exist in an SE Tree. □

Lemma 3.2 (SE Tree contains all $S \subseteq [n]$)**.** Let R be the root of an SE Tree for $n \in \mathbb{N}$ elements and let $S \subseteq [n]$. A node $T \in \mathrm{Des}(R) \cup \{R\}$ with $S_T = S$ exists.

Proof. If $S = \emptyset$, then the node T is simply the root. Therefore, let us consider the case $S \in 2^{[n]} \setminus \{\emptyset\}$ with a size of m and let $S = \{c_{i_1}, c_{i_2}, \ldots, c_{i_m}\}$. The ordering of elements in S is $i_1 < i_2 < \ldots < i_m$. We will show that there exists a path from the root to a node T with $S = S_T$.

We start at the root with the empty set. Because the candidate set of the root node is $[n]$, we can generate the child $T' = (\{c_{i_1}\}, [n] \setminus \{c_1, c_2, \ldots, c_{i_1}\})$. Because the candidate set of T' contains the elements $c_{i_2}, c_{i_3}, \ldots, c_{i_m}$, we can generate the node $T'' = (\{c_{i_1}, c_{i_2}\}, [n] \setminus \{c_1, c_2, \ldots, c_{i_2}\})$. This argument can be repeated until we get the node T^* with $S_{T^*} = S$. □

The SE Tree is a suitable data structure, as it holds all necessary subsets exactly once. However, we do not need subsets with a size larger than k. We, therefore, define a version of the SE Tree that does not hold these sets.

Definition 3.6 (k-depth-limited SE Tree)**.** Let $k \in \mathbb{N}$. We define a *k-depth-limited* Set-Enumeration Tree for $n \in \mathbb{N}$ elements as a Set-Enumeration Tree where nodes T with $|S_T| = k$ have no children ($\mathrm{Ch}(T) = \emptyset$).

In the remainder of this work, we will use the term Set-Enumeration Tree synonymously for k-depth-limited Set-Enumeration Tree since the complete Set-Enumeration Tree is not of interest to the discussed problem. Figure 5 shows a Set-Enumeration Tree for four elements.

The SE Tree is now as compact as possible and is also ordered suitably for our problem. Let T be a node in the SE Tree and let T' be any descendant of T. We know that S_T is a basis for the working set of T', that is $S_T \subseteq S_{T'}$. If we prove that S_T is not part of a maximizing set, then all nodes in the subtree of T can also not be part of a maximizing set. Identifying this property for one node automatically applies to the whole subtree.

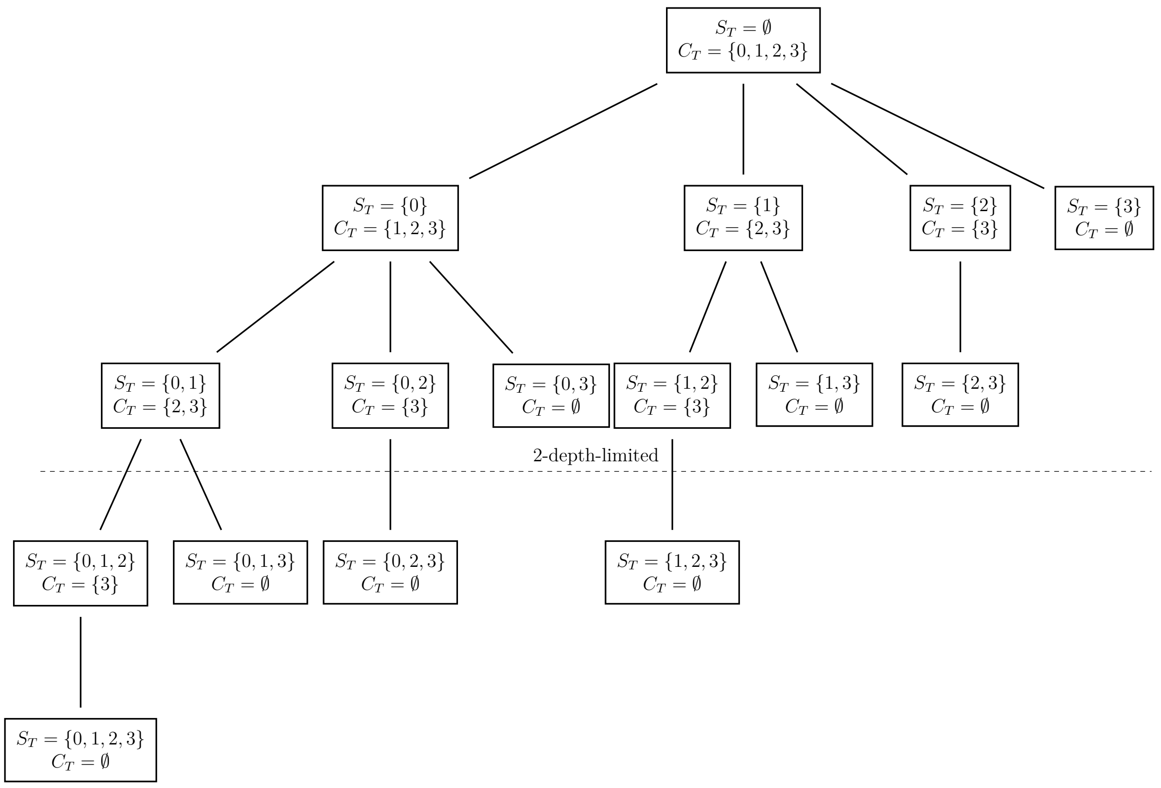

$S_T = \emptyset$ $C_T = \{0,1,2,3\}$
$S_T = \{0\}$ $C_T = \{1,2,3\}$
$S_T = \{1\}$ $C_T = \{2,3\}$
$S_T = \{2\}$ $C_T = \{3\}$
$S_T = \{3\}$ $C_T = \emptyset$
$S_T = \{0,1\}$ $C_T = \{2,3\}$
$S_T = \{0,2\}$ $C_T = \{3\}$
$S_T = \{0,3\}$ $C_T = \emptyset$
$S_T = \{1,2\}$ $C_T = \{3\}$
$S_T = \{1,3\}$ $C_T = \emptyset$
$S_T = \{2,3\}$ $C_T = \emptyset$
2-depth-limited
$S_T = \{0,1,2\}$ $C_T = \{3\}$
$S_T = \{0,1,3\}$ $C_T = \emptyset$
$S_T = \{0,2,3\}$ $C_T = \emptyset$
$S_T = \{1,2,3\}$ $C_T = \emptyset$
$S_T = \{0,1,2,3\}$ $C_T = \emptyset$

Figure 5: A Set Enumeration Tree for $n = 4$ elements. It holds $2^4 = 16$ nodes, each holding a unique subset of $\{0, 1, 2, 3\}$. A 2-depth-limited SE Tree can be derived by removing all nodes under the “2-depth-limited” line.

Now, we want to define some sets that will be important throughout this work.

Definition 3.7 (Important sets)**.** Let T be a node in an SE Tree and $k \in \mathbb{N}$. We define

$$S^*_{C_T} = \underset{S' \subseteq C_T, |S'| = k - |S_T|}{\arg\max} f(S_T \cup S')$$

as the *best remaining subset* (with respect to node T), and we define

$$S^*_T = S_T \cup S^*_{C_T}$$

as the *best set in the subtree of T*.

Definition 3.8 (Set of all nodes)**.** Let R be a root of an arbitrary SE Tree. We denote by $\mathbb{T} = \text{Des}(R) \cup \{R\}$ the set of all nodes of the SE Tree.

We formally define $\mathbb{T}$ because, we want to define functions that work on the nodes formally.

Algorithm 1 Pseudocode for the Plain Search algorithm. It recursively explores all children of a node T. Calling it on a root R of a SE Tree, will explore the SE Tree.

```
    Input: Node $T$, Integer $k$, Global variable $s_{\text{best}} \in \mathbb{R}$
    Output: Maximum Score
 1: if $|S_T| = k$ then                                ▷ Check for a new best
 2:     $s_{\text{best}} = \max(s_{\text{best}}, f(S_T))$
 3: while $|C_T| > 0$ do
 4:     $c = \text{popFront}(C_T)$
 5:     $T' = (S_T \cup \{c\}, C_T)$                   ▷ Generate child node $T'$
 6:     plainSearch($T', k, s_{\text{best}}$)
```

3.2 The Plain Search Algorithm

Based on the SE Tree, we can formulate the *Plain Search* algorithm. The pseudocode can be seen in Algorithm 1. The algorithm takes as input a node T, an integer k, and the current best score s_{best}. In Lines 1 and 2, the algorithm determines if the working set of T has a size of k and if true, it will evaluate if $f(S_T)$ is a new maximum. If T has a working set smaller than k, we explore all its children recursively. The function `popFront` returns the candidate at the front of the set (the one with the smallest index) but also removes the candidate from the set, effectively shrinking C_T by one candidate. This ensures that only children are generated that comply with Definition 3.2. We can search a SE Tree by calling `plainSearch`$(R, k, -\infty)$ with R being the root node. Note that we can additionally keep a variable for the best set that updates each time s_{best} is updated in line 2.

The correctness of the algorithm follows from the structure of the SE Tree. It holds all subsets of size k, and the algorithm only generates children that comply with Definition 3.2. A full search of the SE Tree needs to visit all $\mathcal{O}(n^k)$ leaf nodes and evaluate the score function with their working set. To reach the leaf nodes, we need to traverse the $\mathcal{O}(n^k)$ non-leaf nodes. At each non-leaf node, we need to generate the $\mathcal{O}(n)$ children, which takes $T(n)$ time. his algorithm has the advantage of only using one global candidate set: Each node only has to use a pointer to index into the global candidate set, efficiently denoting its own candidate set. However, later algorithms will not have this option, and to create a child node, we have to copy the candidate set from its parent. We, therefore, already want to denote the running time of generating children as $T(n)$, but this should not have a worse time complexity than $\mathcal{O}(n^2)$. In total, the plain search algorithm has a running time of $\mathcal{O}\left(n^k(T(f) + T(n))\right)$. This algorithm depends on the monotonicity of f because it only evaluates the k-sized solutions. However, one can easily modify this algorithm to evaluate all the sets of size at most k, which would give an algorithm for any arbitrary set function f.

While the algorithm can solve the Cardinality-Constrained Maximization problem for score functions, it has not used the submodularity property so far. To incorporate the property, we will now discuss heuristics that exploit submodularity.

3.3 Heuristics

Heuristics enable us to determine upper bounds and perform reduction rules to remove parts of the search space. Usually, there are not $\binom{n}{k}$ maximizing sets, and only a fraction of leaf nodes hold a maximizing set. Large parts of the SE Tree do not hold a solution that is even near-optimal and, in hindsight, should not have been traversed. Let us first properly define what we mean by *pruning.*

Definition 3.9 (Pruning)**.** We define the action of *pruning a set of candidates C* at node T in a SE Tree as setting $C_T = C_T \setminus C$.

Pruning the set C_T of candidates is also called *pruning node T*, as T cannot generate any more children. If $C = \{c\}$, we say that we *prune candidate c*. We use pruning as an action to remove subtrees in the SE Tree. By pruning a node, we ensure that the algorithm does not advance into the descendants of T. To ensure the exactness guarantee of the search algorithm, we may prune a node only if we can guarantee that there is no maximizing set in the descendants of T. By pruning only a single candidate or a set of candidates, we make sure that these candidates cannot be part of any working set of a descendant of T. We also have to guarantee that these candidates are not part of a maximizing set. Otherwise, we will lose the exactness guarantee of the search algorithm.

Lemma 3.3. *Let T be a node in a SE Tree and $x \in \mathbb{R}$ a score threshold that should be surpassed. Let S^*_T be the best set in the subtree of T and let $S^*_{C_T} \subseteq C_T$ be the best remaining subset. The node T can be pruned if*

$$f(S^*_T) = f(S_T \cup S^*_{C_T}) = f(S_T) + \Delta(S^*_{C_T} \mid S_T) \leq x.$$

Proof. The proof follows from the definition of $S^*_{C_T}$ (see Definition 3.7). If the score of the best set in the subtree of T is at most x, then the descendants of T will not hold a set with a greater score. □

Note that x will be the score of the best set the search algorithm has found so far. By pruning, we guarantee that no set in the subtree of T can surpass the already best-found score. In this form, the lemma is not helpful since it depends on $S^*_{C_T}$, which is not known in advance. We need ways to determine an upper bound on the marginal gain of this set.

Definition 3.10 (Valid Heuristic)**.** Let T be a node in a SE Tree, $C \subseteq [n]$, $k \in \mathbb{N}$, and $k' = k - |S_T|$. We call a function $h : \mathbb{T} \times 2^{[n]} \times \mathbb{N} \to \mathbb{R}$ a *valid heuristic* with respect to a score function f if

$$\max_{S \subseteq C, |S| = k'} f(S_T \cup S) \leq h(T, C, k).$$

We call the value $h(T, C, k)$ a *valid upper bound.*

Valid heuristics give an upper bound on the maximum score that can be achieved at node T when choosing candidates of C. The integer k specifies how large the set is that we want to give an upper bound for. The special case $C = C_T$ illustrates this more clearly. The inequality simplifies to

$$f(S_T^*) \leq h(T, C_T, k).$$

The heuristic will always return a value at least as large as the true maximum score. We define heuristics with the more general $C \subseteq [n]$, as we will later develop heuristics that only consider a subset of C_T.

Instead of pruning a node on the unknown condition $f(S_T^*) \leq x$, we will prune a node on the known condition $h(T, C, k) \leq x$. This is also why we need valid heuristics. If they are invalid, there is at least one set in the descendants of T with a score greater than $h(T, C, k)$. We risk pruning the SE Tree, although the maximizing set could be in that subtree. This destroys the guarantee that the algorithm will find a maximizing set.

When we try to identify whether a node can be pruned, there are three cases of how the unknown best score $f(S_T^*)$, the known upper bound $h(T, C, k)$, and the score threshold x stand in relation to each other:

$$\begin{array}{lccccc}
\text{Case 1:} & f(S_T^*) & \leq & h(T,C,k) & \leq & x \\
\text{Case 2:} & x & < & f(S_T^*) & \leq & h(T,C,k) \\
\text{Case 3:} & f(S_T^*) & \leq & x & < & h(T,C,k)
\end{array}$$

In the first case, the true achievable score and the upper bound of the heuristic are, at most, the score threshold x. Based on $h(T, C, k) \leq x$, the node will be pruned, which is optimal since the true achievable score is lower than the threshold x. The second case describes that there is a set in the subtree of T that surpasses the score threshold x. The node will not be pruned since $x < h(T, C, k)$. The third case is the problematic case. While the achievable score is smaller than the score threshold, and thus we could prune the node, the heuristic surpasses the threshold, and we do not. The first two cases are optimal: We prune when we can and search if we must, but the third case produces unnecessary work. The goal is to minimize the occurrence of this case. A perfect reduction of the third case would generally only occur if we use a heuristic h' such that $h'(T, C, k) = f(S_T^*)$. The only heuristic that can guarantee this would be the one that searches the subtree for the best solution. However, using such a heuristic obviously has no benefits, as we would search the subtree of T to determine if we need to search the subtree of T. Note that the search algorithm could possibly visit each leaf node and find a better solution each time. Then, the third case would also never occur, but this seems very unlikely, so this case is ignored. Sadly, we can prove that a search algorithm will always explore unnecessary subtrees. In other words, we cannot restrict the search of any algorithm to only the relevant part.

Lemma 3.4 (Unnecessary Exploration)**.** If the ETH is true, then there is no valid heuristic h with respect to any score function f and any integer k with a faster running time than $n^{o(k)}$ with $h(T, C, k) = f(S_T^*)$ for every node T in a SE Tree.

Proof. Assume that such a heuristic exists. Applying it to the root node R of the SE Tree would yield the maximum possible score. Then we could solve the Hitting Set decision problem faster than $n^{o(k)}$, but this is a contradiction as Theorem 2.1 states. □

We will discuss valid heuristics more in Section 4.5 after seeing a few examples. Now, let us look at the search algorithm developed by Staus et al. [69], which they used to solve the Group Closeness Centrality problem exactly.

3.4 The Basic Search Algorithm

In this subsection, we will introduce two heuristics that enable us to reduce the search space. So far, we have only discussed that we can prune the search tree if the heuristic h has at most the same value as a given threshold x. But what threshold should we choose? During the search, we will save the best set found so far, and each time we explore a node, we try to determine whether a better set resides in the subtree of the node. By setting x as the best score found so far, denoted by s_{best}, we make sure to only prune parts that do not hold a better solution. While the obvious approach would be to find a good heuristic h, we can also try to find a near-optimal solution quickly and thus maximize s_{best}. The greater the current best score, the more likely a node can be pruned regardless of the heuristic used.

Dynamic Candidate Ordering In its plain form, the search algorithm expands the children of node T based on the (arbitrary) ordering in C_T. In the worst case, we would first explore all children that do not contain S^*_T (the best set in T) and only at the end explore the child holding S^*_T. Exploring the child with the best solution first could improve s_{best}. This could lead to other children being pruned, which would have been explored previously. In practice, we do not know which child holds the best set or more generally, how the children are ordered based on the best set in their subtree. It would be desirable to have an ordering on the children based on how promising it is that the subtree of a child holds S^*_T. We achieve this by a suitable ordering defined as follows:

Definition 3.11 (Dynamic Candidate Ordering)**.** Let T be a node in a SE Tree with candidates $C_T = \{c_1, c_2, \dots, c_m\}$. *Dynamic Candidate Ordering* reindexes the candidates such that

$$\Delta(c_1) \geq \Delta(c_2) \geq \dots \geq \Delta(c_m).$$

Note that although we define C_T as a set, the order of candidates now becomes relevant, and thus, it resembles a list (or a vector). The Plain Search algorithm (Algorithm 1) uses the popFront function that returns the candidates at the front of the list. The algorithm would now explore the subtree of the candidate with the greatest marginal gain of all remaining candidates. The explored node will also expand the candidate with the greatest marginal gain and so on. To borrow terms from machine learning, the search algorithm always explores the path with the greatest gradient or, in this case, the greatest marginal gain. This procedure does not guarantee to find a maximizing set, as it can get stuck in local maxima, but it will often find good solutions.

Note that applying the ordering will cost $\mathcal{O}(n \cdot T(f))$ time to evaluate the score function and additionally $\mathcal{O}(n \log n)$ time to sort the candidates. This greatly increases the worst-case running time since we calculate $\mathcal{O}(n)$ marginal gains for each node of the SE Tree.

Greedy Bootstrapping If Dynamic Candidate Ordering is applied, the algorithm will always explore the best child. If the search starts, the algorithm will traverse to the first leaf and achieve the same result as the greedy algorithm by Nemhauser et al. [57]. This guarantees that a value of $(1 - 1/e)f(S_{\max})$ is already found, with $S_{\max}$ as a maximizing set. A relatively good starting solution is, therefore, already guaranteed.

We also offer an additional greedy bootstrapping solution. First, the standard greedy algorithm [57] determines a solution. Then we will deploy a local search on the solution. We will iterate over each element of the greedy solution. For each element, we determine the candidate that leads to the greatest score and swap the element with the candidate. We will cycle over the solution until either no element can be swapped to achieve a greater score, or 10% of the available time limit is spent. Note that this local search was not originally implemented by Staus et al. [69].

Simple Upper Bound Now, we will discuss the first heuristic that can prune a node. It was proposed by Staus et al. [69] in the context of solving the Group Closeness Centrality problem. They determined a lower bound by adding the top k' centrality improvements (in our context marginal gains) and removing the sum from the centrality achieved so far (in our context s_{best}). Because the group farness of the Group Closeness Centrality problem is supermodular, this represents a valid lower bound. They can prune nodes of the SE Tree if their lower bound is greater than the currently best-achieved centrality. We will define a similar heuristic for arbitrary submodular functions. At node T, we know that we have to add $k' = k - |S_T|$ candidates to get a set of size k. By adding the top k' marginal gains of candidates in C_T to the already achieved score $f(S_T)$, we will get a valid upper bound. We will first properly define the heuristic and then prove its correctness.

Definition 3.12 (Simple Upper Bound)**.** Let T be a node of an SE Tree and let $k \in \mathbb{N}$. We define the *Simple Upper Bound (SUB)* heuristic as

$$\text{SUB}(T, C, k) := f(S_T) + \left(\max_{S' \subseteq C, |S'| = k - |S_T|} \sum_{e \in S'} \Delta(e \mid S_T) \right).$$

Lemma 3.5 (SUB is valid)**.** The Simple Upper Bound heuristic is a valid heuristic[6].

[6]Here only shown for $C = C_T$, but the proof can be easily expanded (but requires more notation).

Proof. For each node T in a SE Tree and each integer k, we need to show

$$f(S_T \cup S^*_{C_T}) \leq \mathrm{SUB}(T, C_T, k).$$

Let S' be the set used by the SUB heuristic, and let $k' = k - |S_T|$. The set S' has the property that the sum of marginal gains of candidates in S' is at least that of the sum of marginal gains of any k' candidates, that is

$$\forall\ \hat{S} \subseteq C_T,\ |\hat{S}| = k' : \sum_{c \in \hat{S}} \Delta(c \mid S_T) \leq \sum_{c \in S'} \Delta(c \mid S_T).$$

This also includes the sum of marginal gains of candidates in $S^*_{C_T} \subseteq C_T$. This yields the inequality

$$\begin{aligned} f(S_T \cup S^*_{C_T}) &= f(S_T) + \Delta(S^*_{C_T} \mid S_T) \\ &\leq f(S_T) + \sum_{c \in S^*_{C_T}} \Delta(c \mid S_T) && \text{Lemma 2.1} \\ &\leq f(S_T) + \sum_{c \in S'} \Delta(c \mid S_T) && \text{Property of } S' \\ &= \mathrm{SUB}(T, C_T, k). \end{aligned}$$

□

Since the heuristic only sums the marginal gains of individual candidates, it is sufficient to consider only the set S' that consists of the k' candidates with the greatest marginal gains. If Dynamic Candidate Ordering is applied on node T, these candidates are at the front of C_T and can quickly be summed up. Otherwise, we would need to iterate over the $\mathcal{O}(n)$ marginal gains and save the k' greatest in a heap, which requires a running time of $\mathcal{O}(n \log k)$. Note that after the algorithm returns from exploring a child, we can update the upper bound of the SUB heuristic in $\mathcal{O}(1)$ time by removing the explored candidate's marginal gain and adding the next candidate's marginal gain.

Candidate Reduction Staus et al. [69] also developed another bound called *Minimal Centrality Threshold*. They noticed that a candidate needed a minimum amount of marginal gain to be part of a better solution.

For example, consider the case that the search algorithm already found a set with a score of 100. The algorithm is currently at node T, achieves the score $f(S_T) = 70$, and needs to add $k' = 4$ candidates. We assume the candidate set is $\{c_1, c_2, c_3, c_4, c_5, c_6\}$ with their respective marginal gains $10, 8, 6, 6, 4$ and 3. Can the candidate c_6 be part of a maximizing solution? Assuming that there is no diminishing return, which would grant the greatest score, the best set that includes c_6 is $\{c_1, c_2, c_3, c_6\}$ with a summed marginal gain of 27. This marginal gain is insufficient to surpass the already achieved score of 100, as at least a marginal gain of 31 is required for a better solution. Therefore, the candidate c_6 cannot be part of a maximizing solution. The same is true for candidate c_5. All sets containing c_5 have, at most, a marginal gain of 28.

The benefit of determining candidates with a marginal gain that is too small is that they can be removed from the candidate set. This will decrease the computational effort carried out at each descendant of T, and it will also decrease the number of leaf nodes of the subtree. We will call this technique *Candidate Reduction*.

Definition 3.13 (Candidate Reduction)**.** Let T be a node in a SE Tree and $k \in \mathbb{N}$. Let $s_{\text{best}} \in \mathbb{R}$ be the currently best score, and h be a valid heuristic. Let $C \subseteq C_T$ be the set of candidates c_i with

$$h(T, C_T \setminus \{c_i\}, k-1) + \Delta(c_i \mid S_T) \leq s_{\text{best}}.$$

We define the *Candidate Reduction (CR)* function as

$$\mathrm{CR}(T) \coloneqq C_T \setminus C.$$

The function returns the subset of candidates that should still be considered by the algorithm.

Lemma 3.6 (CR is Correct)**.** If $c \in C$, then c will not be part of a set with a greater score than s_{best}.

Proof. Assume that c is part of a solution S' with $|S'| = k'$ and $f(S_T \cup S') = f(S_T) + \Delta(S' \mid S_T) > s_{\text{best}}$. We can exploit submodularity and give an upper bound on the marginal gain of S' by

$$\Delta(S' \mid S_T) \leq \Delta(S' \setminus \{c\} \mid S_T) + \Delta(c \mid S_T).$$

Therefore

$$\begin{aligned} & f(S_T) + \Delta(S' \setminus \{c\} \mid S_T) + \Delta(c \mid S_T) > s_{\text{best}} \\ \Leftrightarrow\ & f(S_T) + \Delta(S' \setminus \{c\} \mid S_T) > s_{\text{best}} - \Delta(c \mid S_T) \end{aligned}$$

must hold. Since $c \in C$ it must hold that

$$\begin{aligned} & h(T, C_T \setminus \{c\}, k-1) + \Delta(c \mid S_T) \leq s_{\text{best}} \\ \Leftrightarrow\ & h(T, C_T \setminus \{c\}, k-1) \leq s_{\text{best}} - \Delta(c \mid S_T). \end{aligned}$$

Both inequalities combine to

$$\begin{aligned} & h(T, C_T \setminus \{c\}, k-1) \leq s_{\text{best}} - \Delta(c \mid S_T) < f(S_T) + \Delta(S' \setminus \{c\} \mid S_T) \\ \Leftrightarrow\ & h(T, C_T \setminus \{c\}, k-1) \leq s_{\text{best}} - \Delta(c \mid S_T) < f(S_T \cup S' \setminus \{c\}) \end{aligned}$$

but this is a contradiction because h is a valid heuristic and thus

$$f(S_T \cup S' \setminus \{c\}) \leq h(T, C_T \setminus \{c\}, k-1).$$

Therefore, c cannot be part of a maximizing set. □

To efficiently use Candidate Reduction, we will not determine the set C but update C_T each time we identify a candidate that can be removed. We can start at the back of the candidate set if Dynamic Candidate Ordering is applied and iteratively remove the elements. The candidate at the back of the set has the lowest marginal gain and is, therefore, the most likely candidate to be included in C. If we identify that the candidate belongs to C, we can immediately remove it from the candidate set, as this will not influence the result of the other calls to h. We also do not have to recalculate h for each candidate. If all candidates used previously by h are still in the candidate set, then recomputing h would give exactly the same results. A recalculation is only necessary if the last element of the candidate set was also used by h.

If we use the SUB heuristic as h, the function can further be optimized. We do not have to process each candidate $c \in C_T$ because once the last candidate of the set cannot be removed, then all remaining candidates can also not be removed. This is true, as the marginal gain of the remaining candidates is greater because the candidates are ordered. We can also reapply the Candidate Reduction each time we return from exploring a child. The candidate with the largest marginal gain was explored, and we can remove it from the candidate set. Recalculating h will likely result in a smaller upper bound and, therefore, demands a larger marginal gain for each candidate. Candidates that cannot keep up with this demand can then be removed.

Basic Search Algorithm Algorithm 2 shows the pseudocode of the *Basic Search* algorithm, which will serve as the basis for new algorithms. In contrast to the Plain Search algorithm, it uses Dynamic Candidate Ordering, the SUB heuristic, and Candidate Reduction. The algorithm takes as input a node T, an integer k, and the current best score s_{best}. Like the Plain Search algorithm, the function will update the best score if S_T has a size of k. Otherwise, it will explore the children of node T. The first step is to apply Dynamic Candidate Ordering in Line 4. In the loop, we will continuously shrink C_T by removing explored or redundant candidates. The loop will run until C_T has less than k' candidates because at this point the subtree of T will not contain a node with a working set of size k. The first step of the loop is to calculate the SUB heuristic and return if applicable. Then, we will execute Candidate Reduction that updates C_T. The function `popFront` returns the candidate at the front of the candidate set, which is the candidate with the greatest marginal gain due to Dynamic Candidate Ordering. Then, we will recursively explore the child T'. We can start the search on a SE Tree by calling basicSearch$(R, k, -\infty)$ with R as the root node. If we bootstrap the algorithm with a greedy solution, then exchange $-\infty$ with the score of the greedy solution.

The worst-case complexity of the algorithm is worse when compared to the running time of the Plain Search algorithm. Previously, we needed $\mathcal{O}(T(n))$ time to process one node of the SE Tree. The time is dominated by the time it takes to generate the node's children and should not exceed $\mathcal{O}(n^2)$. Now, we additionally have to compute the SUB heuristic, Candidate Reduction, and Dynamic Candidate Ordering. The SUB heuristic and Candidate Reduction have a running time of $\mathcal{O}(n)$ because they can be calculated once in $\mathcal{O}(k)$ and must be updated at most $\mathcal{O}(n)$ times, which costs $\mathcal{O}(1)$ per update.

Algorithm 2 Pseudocode of the basic search algorithm. In comparison to the plain search algorithm (Algorithm 1) it incorporates Dynamic Candidate Ordering, the Simple Upper Bound heuristic and the Reduction of Possible Candidates.

```
    Input: Node $T$, Integer $k$, Global variable $s_{\text{best}} \in \mathbb{R}$
    Output: Maximum Score
 1: if $|S_T| = k$ then
 2:     $s_{\text{best}} = \max(s_{\text{best}}, f(S_T))$
 3: $k' = k - |S_T|$
 4: $C_T = \text{DCO}(C_T)$                              ▷ Reorder the candidate set
 5: while $|C_T| \geq k'$ do
 6:     if $\text{SUB}(T, C_T, k) \leq s_{\text{best}}$ then            ▷ Check if we can exit early
 7:         return $s_{\text{best}}$
 8:     $C_T = \text{CR}(T)$                              ▷ Update the candidate set
 9:     $c = \text{popFront}(C_T)$          ▷ Get candidate with greatest marginal gain
10:     $T' = (S_T \cup \{c\}, C_T)$                        ▷ Generate child node
11:     basicSearch($T'$, $k$, $s_{\text{best}}$)
```

They will vanish in the $T(n)$ term. However, Dynamic Candidate Ordering is costly. It requires an additional $\mathcal{O}(n \cdot T(f) + n \log n)$ time per node. While the term $n \log n$ will also vanish in $T(n)$, the term $n \cdot T(f)$ is more significant and adds to the theoretical worst-case running time. This makes for a total running time of $\mathcal{O}\left(n^k(nT(f) + T(n))\right)$. The space complexity of the algorithm is $\mathcal{O}(nk)$. At each depth of the SE Tree we have to store the candidates and their marginal gains. Note that the space complexity of the Plain Search algorithm is only $\mathcal{O}(n)$, so we increased the needed space by a factor of k. In practice, the Basic Search algorithm outperforms the Plain Search algorithm by multiple magnitudes, as experiments in Section 5.1 will show.

4 Advanced Heuristics

Until now, we only considered marginal gains of single candidates. The SUB heuristic, for example, uses the k' greatest single marginal gains to calculate an upper bound. However, this bound can be quite large as it does not factor in the diminishing returns that many natural functions exhibit. Example 4.1 shows an example where a more detailed look at the marginal gains of sets can greatly reduce the upper bound.

Example 4.1. Let us consider the Partial Dominating Set problem and, therefore, the vertex domination number f_D as our score function. We want to find a maximizing set with $k = 4$ vertices. Figure 6 shows a graph with $n = 16$ vertices. Assume our base set is $A = \{v_9\}$. The current function value is 4 since the vertices v_6 to v_9 are dominated. We want an upper bound on the function value for the remaining $k' = 3$ vertices. The best possible solution would be adding vertices v_{10}, v_3, and v_{16} to A. Then, all vertices are dominated, granting the best possible value of 16.

Let us first consider what the SUB heuristic calculates. It adds up the three greatest marginal gains of all vertices. These are the vertex v_{10} with a marginal gain of 7 and then two vertices of v_{11} to v_{15} since they all have a marginal gain of 6. In total, the upper bound has a value of $4 + 7 + 6 + 6 = 23$. Not only is this upper bound greater than the total number of vertices and therefore not achievable, but the upper bound also dominated the vertices v_{10} to v_{15} three times. A possible maximizing set is not able to dominate a vertex multiple times. The actual value of this set is $f_\mathrm{D}(\{v_9, v_{10}, v_{12}, v_{14}\}) = 11$, which is less than half of the score that the upper bound estimated.

Now, let us consider what happens if we use the marginal gain of sets instead of only single vertices. Since we need three vertices, we can use one set of size two and one single marginal gain. Out of all possible sets with size 2, one set with the greatest marginal gain is $\{v_3, v_{10}\}$ with $\Delta(\{v_3, v_{10}\} \mid A) = 11$. Now, we have to choose one single marginal gain out of the vertex set. Note that we do not have to consider v_9, as it is already in A, and v_3 and v_{10}, as they are already in the set. One vertex with the greatest remaining single marginal gain is v_{13} with $\Delta(v_{13} \mid A) = 6$. The upper bound is therefore $\Delta(\{v_3, v_{10}\} \mid A) + \Delta(v_{13} \mid A) = 11 + 6 = 17$. While this upper bound is still greater than the number of vertices, it is substantially lower than the upper bound of the SUB heuristic, and this upper bound is almost equal to the actual best achievable score. Also, the score of the complete set $f_\mathrm{D}(\{v_9, v_3, v_{10}, v_{13}\}) = 15$ is greater than the score of the set of the SUB heuristic and almost equal to the optimal score. Using sets of size 3 in this example will simply grant the set $\{v_{10}, v_3, v_{16}\}$ as one of the best sets. This set directly solves the maximization problem for this graph and is, therefore, not an interesting example for an upper bound. However, if this graph would be a subgraph of a much larger graph, determining an upper bound for three vertices will aid in giving a sharper upper bound on the larger graph.

As the example shows, a more detailed look at the marginal gain of sets can lower the estimated upper bound, which can lead to more pruning. In general, the larger the

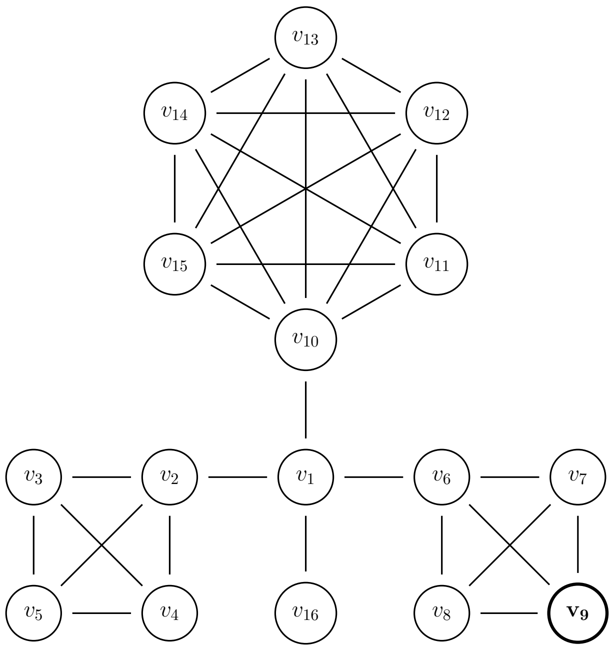


Figure 6: A graph holding 16 vertices. If we consider the PARTIAL DOMINATING SET problem on this graph we will see that marginal gains of sets can sharpen the upper bound of heuristics. A detailed description can be found in Example 4.1.

set is, the greater the diminishing return. This is a direct consequence of Lemma 2.1, since

$$\Delta(B \cup C \mid A) \leq \Delta(B \mid A) + \Delta(C \mid A)$$

for $A, B, C \subseteq \mathcal{U}$. Using fewer larger sets may result in a sharper upper bound than multiple smaller sets. However, increasing the size of sets also increases the number of available sets that could be evaluated (if $n > 2k$, which we always assume). Finding only the "relevant" sets becomes more challenging, and many sets probably must be evaluated. Hence, there is a tradeoff to be made when using heuristics that use marginal gains of larger sets: While they grant a sharper upper bound, they also require more time to compute.

4.1 Upper Bound 2D

We start with a heuristic that considers the marginal gain of sets of size 2. We will call these sets *pairs*. A node T in the SE Tree has $|C_T| = m$ candidates, and therefore $m(m-1)/2$ pairs. We need to give a valid upper bound for the best remaining set $S^*_{C_T}$ in T with size of $k' = k - |S_T|$. This means choosing $k'/2$ pairs if k' is even and $(k'-1)/2$ pairs and one single marginal gain if k' is odd. The pairs must be chosen such that the summed marginal gain is a valid upper bound for $\Delta(S^*_{C_T} \mid S_T)$. Choosing these pairs is not trivial because once we exclude $\mathcal{O}(m)$ pairs, once we choose a pair. When choosing, for example, the pair $\{a, b\}$, then the pair $\{a, c\}$ or $\{b, d\}$ is not available

	a	b	c	d	e	f
a	10	14	12	11	13	10
b	-	8	9	11	9	8
c	-	-	6	8	7	6
d	-	-	-	6	7	7
e	-	-	-	-	3	4
f	-	-	-	-	-	2

Table 2: Example of marginal gains of pairs, where a naive approach of choosing two pairs will not result in a maximized sum. The table is symmetric so only the upper right half is shown.

anymore because a or b would have been chosen twice. This (optional) requirement is called *pair partitioning.* We formally define this requirement and additional useful sets.

Definition 4.1. Let $A = \{a_1, a_2, \ldots, a_n\}$ be a set of n elements. We define the *pairs of A* as

$$P(A) \coloneqq \{\{a_i, a_j\} \mid 1 \leq i < j \leq n\}.$$

We say that a collection of pairs $\{A_1, A_2, \ldots, A_k\} \subseteq P(A)$ fulfills the *pair partitioning* requirement if

$$\forall\ 1 \leq i < j \leq k : A_i \cap A_j = \emptyset.$$

Furthermore, we define the set

$$\mathbb{P}(A, k) \coloneqq \{B \mid B \subseteq P(A), |B| = k, B \text{ fullfills pair partitioning}\}$$

as the *k-sized pair partitioning combinations of A.*

If A has n elements, then $P(A)$ has a size of $n(n-1)/2$, and $\mathbb{P}(A, k)$ has a size of $\binom{n}{2k}(2k!/2^k k!)$. First, we have to consider all possible subsets of size $2k$ of the n elements, which are $\binom{n}{2k}$ many, and secondly, we need all combinations to partition the $2k$ elements into k sets, each with size two. These are $(2k!/2^k k!)$ many combinations.

While the requirement is not necessary to generate a valid upper bound, it ensures that all candidates used in the upper bound are unique. This can be advantageous, as candidates with a large marginal gain can only be used once in the upper bound. We will present an approach that neglects this optional requirement later in Section 4.1.4. First, we will discuss three approaches that fulfill the requirement.

Let us first show that simply choosing the pairs with the greatest marginal gain one after the other is not a valid approach since it will not necessarily lead to a valid upper bound. Example 4.2 discusses such a case.

Example 4.2. Table 2 shows six elements $\mathcal{U} = \{a, b, c, d, e, f\}$ with their marginal gains and the marginal gain of their pairs. Note that the table is symmetric since the pairs are sets, and also note that the diagonal contains the marginal gain for $x \in \mathcal{U}$. The algorithm may only choose pairs from the upper-right triangle as the lower-left triangle is redundant, and the diagonal does not hold any pair. Consider that $k' = 4$, so we need to choose two pairs. A naive algorithm first chooses the pair $\{a, b\}$ since it has the greatest marginal gain of 14. Now, it may only choose a pair from the remaining elements $\{c, d, e, f\}$. The second pair is $\{c, d\}$, with a marginal gain of 8, which sums up to a total score of 22.

However, the sum $\Delta(\{a, e\}) + \Delta(\{b, d\}) = 13 + 11 = 24$ is greater than the upper bound from the naive approach. If $S^*_{C_T} = \{a, e, b, d\}$ and $\Delta(\{a, e, b, d\}) = 24$, then this naive approach violates Definition 3.10 and would not grant a valid upper bound. Using it does, therefore, not guarantee that a maximizing solution will be found, and the algorithm loses its correctness guarantee.

In the following sections, we discuss different approaches on how to give a valid upper bound using the marginal gain of pairs. Section 4.1.1 describes a brute-force approach, Section 4.1.2 describes an approach using a maximum-weight matching, and Section 4.1.3 describes an approach utilizing dynamic programming. All these approaches fulfill the pair partitioning requirement. Section 4.1.4 describes a greedy algorithm that neglects this requirement. While discussing these heuristics, we will always assume that k' is even to simplify the analysis. In Section 4.1.6 we will discuss how to handle the case that k' is odd. Note that all approaches initially suffer from a large unpractical running time. Section 4.1.7 discusses how to avoid these running times, by introducing an additional hyperparameter.

4.1.1 Brute-Force

A brute-force approach is the most straightforward method. It will iterate over all $k'/2$-sized combinations of pairs. If the combination fulfills the pair partitioning requirement, it is determined whether its summed marginal gain is a new maximum, and the result is updated accordingly. This guarantees that we will find the combination maximizing the sum and that the upper bound is valid. This is true because we will also maximize over the combinations of $S^*_{C_T}$. Since the sum of marginal gains of the pairs is at least $\Delta(S^*_{C_T} \mid S_T)$ (due to Lemma 2.1), the resulting upper bound is valid. We formally define this approach.

Definition 4.2. Let T be a node in a SE Tree, let $k \in \mathbb{N}$ and let $k' = k - |S_T|$ be even. The *Upper Bound 2D Brute-Force* heuristic is

$$\mathrm{UB2D}_B(T, C_T, k) := f(S_T) + \max_{P \in \mathbb{P}\left(C_T, \frac{k'}{2}\right)} \sum_{p \in P} \Delta(p \mid S_T).$$

While the approach is very simple, it needs to evaluate many combinations. A total of $\mathcal{O}(n^2)$ pairs exists, and we have to choose $k'/2$ out of them. This makes for a total of $\mathcal{O}(n^{k'})$ combinations, including combinations that do not fulfill the pair partitioning requirement. This approach is, in general, not feasible in practice. Instead of having a running time of $\mathcal{O}(n^k T(f))$ to solve the original maximization problem, we have a running time of $\mathcal{O}(n^{k'})$ to solve a problem for one of the $\mathcal{O}(n^k)$ nodes in the SE Tree. As previously mentioned, Section 4.1.7 discusses how to reduce this running time.

4.1.2 Maximum-Weight Matching

The brute-force approach suffers from a large running time because it evaluates all combinations to find a maximum. Using a maximum-weight matching for graphs can more efficiently determine the same maximum. We interpret the candidate set C_T with $|C_T| = m$ as vertices and the pairs as edges. The resulting graph will be a clique of size m. With this approach, we want to achieve two things. For once, we do not want to iterate across the invalid combinations, and secondly, we quickly want to find the maximizing combination instead of searching all valid combinations. Both requirements can be achieved with a maximum-weight matching of size $k'/2$ on the clique. We will first describe a maximum-weight matching and then prove that it generates a valid upper bound. Let us start with the definition of a matching.

Definition 4.3 (Matching)**.** Let $G = (V, E)$ be an undirected graph with vertex set V and edge set E. We say that $E' \subseteq E$ is a *matching on* G if every vertex $v \in V$ has at most one edge $e \in E'$ with $v \in e$. We say that E' is a *matching on* G *of size* k if $|E'| = k$.

In other words, a matching is a subset of edges, such that no two edges in the matching are incident. Matchings of size $k'/2$ are precisely the matchings we are interested in. Translating a matching of size $k'/2$ into our problem domain means that we have found a subset of size $k'/2$ of the pairs. Now, we need to identify the matching with the greatest sum out of all these matchings. These subsets are defined as follows.

Definition 4.4 (Maximum-Weight Matching)**.** Let $G = (V, E, W)$ be an undirected graph with weight function $W : E \to \mathbb{R}$, which maps a weight to each edge, and let E' be a matching on G with size k. We say that E' is a *maximum-weight matching on* G *of size* k if

$$\sum_{e \in E'} W(e) \geq \sum_{e \in E^*} W(e).$$

holds for all other matchings E^* of size k.

How can we compute a maximum-weight matching of size k? Most literature focuses either on maximum-weight matchings [23, 29], without the size constraint k, or on perfect matchings [24, 46]. Developing a new algorithm for such a problem is by no means simple. However, we can exploit a maximum-weight perfect matching to help in finding a solution. Let us first define what a perfect matching is.

Definition 4.5 (Perfect Matching)**.** Let E' be a matching on a graph $G = (V, E)$. We say that E' is a *perfect matching* on G if $|E'| = |V|/2$.

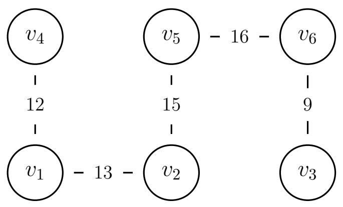


Figure 7: A clique of size 6. All edges that are not drawn have a small weight that does not influence the result. The edge set $\{\{v_1, v_4\}, \{v_2, v_5\}, \{v_3, v_6\}\}$ is the maximum-weight perfect matching with a sum of weights of 36. Removing the smallest contributing edge will lead to the matching $\{\{v_1, v_4\}, \{v_2, v_5\}\}$ with a sum of weights of 27. However, the matching $\{\{v_1, v_2\}, \{v_5, v_6\}\}$ has a sum of weights of 29 and therefore is the maximum-weight matching of size 2.

A perfect matching does not have to be unique. That is, in general, multiple perfect matching can exist on a graph. For example, a clique of size $2n$ has $(2n)!/2^n n!$ perfect matchings. Maximum-weight perfect matchings are defined analogously to maximum-weight matchings.

Definition 4.6 (Max.-Weight Perfect Matching)**.** Let $G = (V, E, W)$ be an undirected graph with weight function $W : E \to \mathbb{R}$. We say that $E' \subseteq E$ is a *maximum-weight perfect matching* if E' is a maximum-weight matching on G of size $|V|/2$.

Finding maximum-weight matchings in graphs is a well-studied problem. Edmonds Blossom algorithm [24] was the first to find a maximum-weight matching on general graphs efficiently. The algorithm has a running time of $\mathcal{O}(n^2 m)$ [46] with n as the number of vertices and m as the number of edges. While the development of the theoretical running time has been improving steadily to a $\mathcal{O}(n(m + n \log n))$ [26] and even further, we will use the Blossom V algorithm proposed by Kolmogorov [46]. The Blossom V′s worst-case complexity is $\mathcal{O}(n^3 m)$, and therefore worse than the Blossom algorithm. However, Blossom V is the most efficient algorithm to solve the Minimum-Cost Perfect Matching problem [17]. A Minimum-Cost Perfect Matching asks to compute a perfect matching with the smallest sum of weights. It is equivalent to the Maximum-Weight Perfect Matching problem. If one needs to compute a maximum-weight perfect matching on some graph $G = (V, E, W)$ one can also compute a minimum-cost perfect matching on $G' = (V, E, W')$ with $W'(e) = -W(e)$. The edges of a minimum-cost perfect matching on G' will be the edges of a maximum-weight perfect matching on G.

How can an algorithm for finding a maximum-weight perfect matching help to find a maximum-weight matching of size $k'/2$? Simply calculating a maximum-weight perfect matching on the clique and then removing the edges with the smallest contribution is not a valid option since it can lead to wrong results, as Figure 7 shows. To get a maximum-weight matching of size $k'/2$ on the clique, we have to modify the graph.

Matching Graph We build a new graph, which we call a *matching graph*. The graph contains the previously constructed clique and additional dummy vertices fully

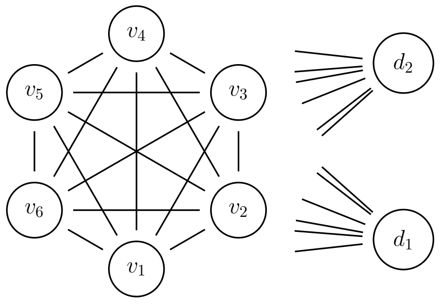


Figure 8: A matching graph for $k = 4$ vertices. The graph holds a clique of size six and has $6 - 4 = 2$ additional dummy vertices, each fully connected to the clique but not with each other. Edge weights are not shown in this example.

connected to the clique but not between each other. These types of graphs are also called fully connected split graphs.

Definition 4.7 (Matching Graph)**.** Let $G_K = (V_K, E_K, W_K)$ be a clique on n vertices and let $k \leq n$. We define the *matching graph* $G = (V, E, W)$ *for* k *vertices* with vertex set V, edge set E, and weight function $W : E \to \mathbb{R}$. The vertex set V contains all vertices of the clique and exactly $n - k$ dummy vertices.

$$V = V_K \cup V_D = V_K \cup \{u_i \mid 1 \leq i \leq n - k\}.$$

The edges are the original edges of the clique and additional edges that connect every vertex of the clique to every dummy vertex.

$$E = E_K \cup E_D = E_K \cup \{\{v_i, u_j\} \mid v_i \in V_K, u_j \in V_D\}.$$

The weight functions is

$$W(e) = \begin{cases} W_K(e) & \text{if } e \in E_K, \\ 0 & \text{otherwise.} \end{cases}$$

The dummy vertices V_D are fully connected to each vertex of the clique with a weight of 0. The weights of E_D do not differ so that the edges can be swapped with each other in the matching without altering the sum. We call G a matching graph for k vertices because the dummy vertices match all but k vertices of the clique. This will be shown in Lemma 4.1. Figure 8 shows a matching graph for two vertices with a clique of size 6.

Now, we will show that a maximum-weight perfect matching on a matching graph for k' vertices leads to the desired properties. The result will hold a maximum-weight matching of size $k'/2$ on the clique G_K. We will first prove that the maximum-weight perfect matching holds exactly $k'/2$ edges from the clique and then that this matching of the $k'/2$ clique edges is a maximum-weight matching of size $k'/2$ on the clique.

Lemma 4.1. Let $G = (V, E, W)$ be a matching graph for k' vertices, and let n be the number of vertices in the clique. Each maximum-weight perfect matching $E' \subseteq E$ of G contains $k'/2$ edges from E_K.

Proof. Graph G contains n vertices from the clique and $n - k'$ dummy vertices. Since every vertex is matched by exactly one edge to another vertex, we know that the $n - k'$ dummy vertices must be connected to $n - k'$ different vertices in the clique since the dummy vertices cannot be matched to each other. The edges to match these vertices are exclusively from E_D. Now, only $(2n - k') - 2(n - k') = k'$ vertices remain from the clique. We need exactly $k'/2$ edges from E_K to match these vertices. □

Now let us prove that a maximum-weight perfect matching on the matching graph also gives a maximum-weight matching of size $k'/2$ on the clique.

Lemma 4.2. Let $G = (V, E, W)$ be a matching graph for k' vertices, and let n be the number of vertices in the clique. Let E' be a maximum-weight perfect matching on G and let $E'_K \subseteq E'$ be a matching of size $k'/2$ on the clique. Such a matching exists due to Lemma 4.1. Then, for all other matching $\hat{E_K}$ of size $k'/2$ on G_K, we have

$$\sum_{e \in E'_K} W(e) \geq \sum_{e \in \hat{E_K}} W(e).$$

Proof. Assume there exists a matching $E^*_K \subseteq E_K$ of size $k'/2$ on the clique with

$$\sum_{e \in E^*_K} W(e) > \sum_{e \in E'_K} W(e).$$

We will construct a new maximum-weight perfect matching E^* on G using E^*_K and show that it has a greater sum of weights than E'. This will contradict that E' is a maximum-weight perfect matching.

Since E^*_K has $k'/2$ edges from the clique, k' vertices from the clique are already matched. We use an arbitrary matching of size $n - k'$ for the remaining $n - k'$ clique and $n - k'$ dummy vertices. These new edges all have the same weight of 0. This means we have a matching E^* with

$$\sum_{e \in E^*} W(e) = \sum_{e \in E^*_K} W_K(e) > \sum_{e \in E'_K} W_K(e) = \sum_{e \in E'} W(e).$$

The edge set E^* has a greater sum of weights than E', which is absurd since E' is a maximum-weight perfect matching. Therefore, the matching E^*_K cannot exist, and E'_K is a maximum-weight matching of size $k'/2$ on the clique. □

Maximum-Weight Matching Heuristic A maximum-weight perfect matching on a matching graph G will guarantee that $k'/2$ edges from the clique with the greatest sum of weights are chosen. Now we just have to show that using the marginal gains of these edges grants a valid heuristic for the Cardinality-Constrained Maximization problem.

Definition 4.8. Let T be a node in a SE Tree, $k \in \mathbb{N}$ and $k' = k - |S_T|$ be even. Let G_K be the clique formed by the candidates of C_T. Let E'_K be a maximum-weight matching of size $k'/2$ on G_K. The *Upper Bound 2D Maximum-Weight Matching* heuristic is

$$\mathrm{UB2D}_M(T, C_T, k) \coloneqq f(S_T) + \sum_{e \in E'_K} \Delta(e \mid S_T).$$

Lemma 4.3. Let T be a node in a SE Tree, $k \in \mathbb{N}$ and $k' = k - |S_T|$ be even. The Upper Bound 2D Maximum-Weight Matching heuristic is valid.

Proof. We need to show

$$\Delta(S^*_{C_T} \mid S_T) \leq \sum_{e \in E'_K} \Delta(e \mid S_T).$$

The set $S^*_{C_T}$ has a size of k' and is a subset of C_T. We can arbitrarily split $S^*_{C_T}$ into $k'/2$ pairs. All the pairs contain candidates from C_T, and therefore, all these pairs were edges in the clique G_K. These pairs form a matching E^*_K of size $k'/2$ on G_K. Lemma 4.2 proved that E'_K is a maximum-weight matching of size $k'/2$ on G_K and therefore, its sum of weights is at least the sum of weights of E^*_K. □

What is the worst-case running time of the heuristic? As mentioned above, we use the Blossom V algorithm [46] that has a worst case complexity of $\mathcal{O}(n^3 m)$ with n and m being the number of vertices and edges respectively. In our case, the number of vertices and edges of the matching graph G is

$$|C_T| + |C_T| - k' \in \mathcal{O}(n) \quad \text{and} \quad \frac{|C_T|(|C_T| - 1)}{2} + |C_T| \cdot (|C_T| - k') \in \mathcal{O}(n^2)$$

respectively. This results in a total running time of $\mathcal{O}(n^5)$.

4.1.3 Dynamic Programming

Now, we present a heuristic that uses dynamic programming to calculate an upper bound. But first, let us show why a different approach than the maximum-weight matching is needed.

In the correctness proof for Upper Bound 2D Maximum-Weight Matching we split $S^*_{C_T}$ arbitrarily into pairs. That is, we arbitrarily chose a set $P \in \mathbb{P}(S^*_{C_T}, k'/2)$. Since we did not need to specify which P is used in the proof, we can assume that it is the set with the greatest sum of marginal gains. The upper bound is, therefore, at least as great as the worst set in $\mathbb{P}(S^*_{C_T}, k'/2)$. Since $S^*_{C_T}$ is not known in practice, we have to consider all sets $A \subseteq C_T$ of size k'. The maximum-weight matching approach, therefore, computes

$$f(S_T) + \max_{A \subseteq C_T, |A| = k'} \max_{P \in \mathbb{P}(A, k'/2)} \sum_{p \in P} \Delta(p \mid S_T).$$

as the upper bound. While the upper bound is valid, it can happen that the maximum-weight matching is not very sharp. Example 4.3 shows an example where the maximum-weight matching has a large upper bound.

Example 4.3. Assume we have a clique of size n and two "Star"-graphs with m vertices each and let $m < n/2$. We denote the internal vertices of the star graphs by v_1 and v_2. Both the star graphs contain one leaf vertex, u_1 and u_2, that is part of the clique, and these vertices differ from one another. Let us consider the vertex domination number f_{D} (from the Partial Dominating Set problem) and $k' = 4$. The best set of size 4 is to choose the set containing the two internal vertices of the star graph, one vertex of the clique, and one arbitrary vertex. The last vertex is not of interest since the first three already dominate all other vertices. Therefore, a maximizing set is $\{v_1, v_2, u_1, u_2\}$ with a marginal gain of $n + 2m$. The maximum-weight matching approach chooses two vertex pairs with the greatest sum of marginal gains that fulfill the pair partitioning requirement. One vertex pair could be $\{v_1, u_1\}$, and the other would be $\{v_2, u_2\}$. Both pairs have a marginal gain of $n + m$,and the upper bound is $2n + 2m$. The upper bound overestimates the optimal score by a value of n. These vertex pairs are the worst way to divide $\{v_1, v_2, u_1, u_2\}$ into two pairs, and another way greatly reduces the upper bound. The pairs $\{v_1, v_2\}$ and $\{u_1, u_2\}$ have an upper bound of $2m+n+2$, which only overestimates by 2. Using a different way to divide $\{v_1, v_2, u_1, u_2\}$ into pairs thus reduces the overestimate from $\Theta(n)$ to a constant $\mathcal{O}(1)$.

A more precise upper bound is the sum of marginal gains of $P_{\min} \in \mathbb{P}(S^*_{C_T}, k'/2)$, with $P_{\min}$ as the set with the lowest sum of marginal gains. The upper bound would then be computed as

$$f(S_T) + \max_{A \subseteq C_T, |A| = k'} \min_{P \in \mathbb{P}(A, k'/2)} \sum_{p \in P} \Delta(p \mid S_T).$$

Managing to find $P_{\min}$ can substantially lower the estimated upper bound leading to more pruning. Let us first prove that the formula computes a valid upper bound before describing an approach that calculates it.

Lemma 4.4. Let T be a node in a SE Tree, $k \in \mathbb{N}$ and $k' = k - |S_T|$ be even. The upper bound

$$f(S_T) + \max_{A \subseteq C_T, |A| = k'} \min_{P \in \mathbb{P}(A, k'/2)} \sum_{p \in P} \Delta(p \mid S_T).$$

is valid.

Proof. We need to show

$$\Delta(S^*_{C_T} \mid S_T) \leq \max_{A \subseteq C_T, |A| = k'} \min_{P \in \mathbb{P}(A, k'/2)} \sum_{p \in P} \Delta(p \mid S_T).$$

Because the upper bound uses all sets $A \subseteq C_T$ with $|A| = k'$, it also uses $S^*_{C_T}$. Let $P_{\min} \in \mathbb{P}(S^*_{C_T}, k'/2)$ be the set with the lowest sum of marginal gains. Lemma 2.1 guarantees

$$\Delta(S^*_{C_T} \mid S_T) \leq \sum_{p \in P_{\min}} \Delta(p \mid S_T). \qquad \square$$

Modeling this approach using graph matchings is not possible as maximum-weight matchings do not distinguish between different sets. They globally look for a maximum value. A minimum-cost matching could compute

$$f(S_T) + \min_{A \subseteq C_T, |A| = k'} \min_{P \in \mathbb{P}(A, k'/2)} \sum_{p \in P} \Delta(p \mid S_T)$$

to find a global minimum, but this is not helpful, as it is not a valid heuristic.

It is also not a valid approach to first compute a maximum-weight matching E'_K with candidates $\mathbb{E} = \bigcup_{e \in E'_K}$, and then minimize only over $\mathbb{P}(\mathbb{E}, k'/2)$. In general, $\mathbb{E} \neq S^*_{C_T}$ and we can not make any assumptions about the smallest marginal gain of $S^*_{C_T}$ by $\mathbb{E}$. The smallest marginal gain of any pair partition of $\mathbb{E}$ may be smaller than that of any pair partition of $S^*_{C_T}$. We need a different approach to maximize over the sets and minimize over the $k'/2$-sized pair partitioning combinations of A.

A brute-force approach would iterate over all $\mathcal{O}(n^{k'})$ sets of size k', and for each set, minimize over the $k'/2$-sized pair partitioning combinations of A. This would lead to a running time of $\mathcal{O}(n^{k'} \cdot |\mathbb{P}(C_T, k'/2)|)$. Note that while the running time looks higher than that of the brute-force approach in Section 4.1.1, which has a running time of $\mathcal{O}(n^{k'})$, they are, in fact, the same: Both approaches iterate over all $k'/2$-sized combinations of pairs, the brute-force approach only does it unordered, while this approach deploys an ordering on the sets.

Dynamic Programming To reduce the running time of the brute-force approach, we use dynamic programming. We will incrementally build a solution that minimizes the split of the sets with size k'. Note that we still assume that k' is even. We define a *dynamic update* function to find the set $P_{\min}$, and with it, we can define the *Upper Bound 2D Dynamic Programming* heuristic.

Definition 4.9. Let T be a node in a SE Tree. We define the *dynamic update* function $d : 2^{\mathcal{U}} \to \mathbb{R}$

$$d(A) := \begin{cases} \min_{c_i \neq c_j \in C} d(A \setminus \{c_i, c_j\}) + d(\{c_i, c_j\}) & \text{if } |\text{A}| > 2 \\ \Delta(A \mid S_T) & \text{if } |\text{A}| = 2. \end{cases}$$

Note that the function is not defined for sets of odd size since we only determine upper bound for sets of pairs and thus an event amount of candidates.

Definition 4.10. Let T be a node in a SE Tree, $k \in \mathbb{N}$ and $k' = k - |S_T|$ be even. Let d be the dynamic update function. The *Upper Bound 2D Dynamic Programming* heuristic is

$$\text{UB2D}_D(T, C_T, k) := f(S_T) + \max_{A \subseteq C_T, |A| = k - |S_T|} d(A).$$

Let us prove that the heuristic is valid before discussing the complexity of the heuristic.

Lemma 4.5. The Upper Bound 2D Dynamic Programming heuristic is valid.

Proof. We only have to prove

$$d(A) = \min_{P \in \mathbb{P}(A,k'/2)} \sum_{p \in P} \Delta(p \mid S_T)$$

because, in Lemma 4.4, we proved that such a heuristic will be valid.

If $|A| = 2$, the equation holds, so consider the case $|A| > 2$ and even. Let $\{c_i^*, c_j^*\}$ be the pair that achieved the lowest sum, that is

$$d(A \setminus \{c_i^*, c_j^*\}) + d(\{c_i^*, c_j^*\}) \leq \min_{c_i \neq c_j \in A} d(A \setminus \{c_i, c_j\}) + d(\{c_i, c_j\}).$$

Assume the sum is not equal to $\min_{P \in \mathbb{P}(A,k'/2)} \sum_{p \in P} \Delta(p \mid S_T)$. Then, there would be a set $P' = \{p_1, p_2, \ldots, p_j\} \in \mathbb{P}(A, k'/2)$ with a lower sum. We can partition P' into $P' \setminus \{p_j\}$ and $\{p_j\}$. But then, we would have chosen $\{c_i^*, c_j^*\} = p_j$ and $d(P' \setminus \{p_j\}) + d(p_j)$ would be the lowest sum. Therefore, $d(A)$ computes the minimal sum of marginal gains across all $k'/2$-sized pair partitioning combinations of A. □

What is the time complexity of this approach? For every $A \subseteq C_T$ of size k', we need to compute $d(A)$. The value $d(A)$ depends on all subsets of A with size $k' - 2$ and all subsets of size 2. If we calculate all $d(A)$ naively, then we would not gain any improvement in contrast to the corresponding brute-force algorithm, which also iterates over all $k'/2$-sized pair partitioning combinations of A. However, the dynamic approach has redundant calculations that can be cached to improve the running time. For example, consider the sets $A = \{a, b, c, d, e, f\}$ and $B = \{c, d, e, f, g, h\}$. Both evaluations of $d(A)$ and $d(B)$ use $d(\{c, d, e, f\})$ as an intermediate result. When iterating over all sets $A \subseteq C_T$ with a size of k', most of the calls to d will be executed multiple times. In fact, only the calls $d(A)$ with a size of k' will be called once. All other calls with smaller sets will be called at least twice. Instead of calculating these results over and over again, we can save them in a table and look them up if required. To efficiently look the results up, we implement the function by associating a bit vector with each set. If candidate c_i is in the set, the bit vector will encode this with 1 at position i and with 0 else. The bit vector will be used as an index into an array, and d can therefore be looked up in $\mathcal{O}(1)$.

We can calculate the result for all $d(A)$ from the bottom up. The values for sets of size 2 are trivial. The values for sets A of size $2i$ are determined by the values for sets of $2(i-1)$ and sets of size two for any $i > 1$. In total, we need to determine the value of

$$\sum_{i=1}^{k'/2} \binom{|C_T|}{2i} \in \mathcal{O}(n^{k'})$$

sets. To determine the value of set A, we need to iterate over all combinations of $c_i \neq c_j \in A$, which are $\mathcal{O}(k'^2)$. In total, a running time of $\mathcal{O}(n^{k'} k'^2)$ is required, which is better than the $\mathcal{O}(n^{k'} \cdot |\mathbb{P}(C_T, k'/2)|)$ complexity of the corresponding brute-force algorithm. However, due to the extensive caching, we will need additional memory to store the result for all sets. Since there are a total of $\mathcal{O}(n^{k'})$ sets, this approach needs additional $\mathcal{O}(n^{k'})$ memory. This is a large amount, even for relatively small values for n and k', which makes the approach infeasible for practical applications. In Section 4.1.7, we will discuss how to circumvent the time and memory requirement.

4.1.4 Greedy Approach

Until now, we have only considered heuristics, which fulfill the pair partitioning requirement. This requirement is not necessary to compute a valid upper bound, but it will generally result in sharper upper bounds, as we will see in Section 4.5. However, the requirement made determining the pairs for the upper bound more complicated. The greedy approach now presented will determine a valid upper bound but neglect the pair partitioning requirement. The greedy variant simply chooses the $k'/2$ pairs with the greatest marginal gain regardless of whether candidates are used multiple times.

Definition 4.11. Let T be a node in a SE Tree, $k \in \mathbb{N}$ and $k' = k - |S_T|$ be even. The *Upper Bound 2D Greedy* heuristic is

$$\mathrm{UB2D}_G(T, C_T, k) \coloneqq f(S_T) + \max_{P \subseteq P(A), |P| = k'/2} \sum_{p \in P} \Delta(p \mid S_T).$$

Lemma 4.6. The Upper Bound 2D Greedy heuristic is valid.

Proof. The Upper Bound 2D Greedy heuristic chooses at least the same set of pairs as the Upper Bound 2D Maximum-Weight Matching heuristic. Therefore, it has at least the same upper bound as the matching approach and is valid. □

The running time of the greedy approach is only $\mathcal{O}(n^2 \log k')$ since we need to store the top $k'/2$ pairs in a heap when iterating across the $\mathcal{O}(n^2)$ pairs.

In general, any approach that fulfills the pair partitioning requirement can give a much more accurate upper bound since the greedy variant can overestimate by a lot. Assume, for example, that we have one candidate with a marginal gain of 100 and all other candidates have a very small marginal gain of, for example, 2 to 5. All the pairs that contain the first candidate also have at least a marginal gain of 100 due to the monotonicity of the score function. When the greedy variant chooses the pairs, it will exclusively choose pairs containing the first candidate, resulting in a large upper bound. An approach that fulfills the pair partitioning requirement would have only considered one pair that includes the first candidate, and all other pairs would have a small marginal gain.

However, there are also cases where the greedy variant can be useful. These are when the variance of the $\mathcal{O}(n^2)$ marginal gains is relatively small. Then, the upper bound of the other approaches and the greedy variant will not differ much. It is more likely that the search algorithm prunes the same nodes, and the reduced running time of the greedy approach could have a noticeable effect.

4.1.5 Comparison of the Approaches

In this section, we want to briefly compare the complexities of the four approaches. Each approach has at least a time complexity of $\mathcal{O}(n^2T(f))$ to compute the marginal gains of all pairs. To reduce the number of calls of f, we will cache the marginal gains in a table which needs $\mathcal{O}(n^2)$ space. If we were not caching them, then each time the algorithm returns from a child, we would have to recompute all $\mathcal{O}(n^2)$ marginal gains, which is wasteful.

The *Brute-Force* approach described in Section 4.1.1 is only one of three possible brute-force approaches. The approach maximizes over all possible sets of $\mathbb{P}(A, k'/2)$ for each $A \subseteq C_T$ with $|A| = k'$ in an unordered way and returns the maximum sum of marginal gains. We called this approach *Upper Bound 2D Brute-Force.* In Section 4.1.3, we also described that we could maximize across all subsets A and minimize across all $k'/2$-sized pair partitioning combinations of A. This approach has the same theoretical complexity and determines the lower bound provided by the dynamic approach. We called this approach *Upper Bound 2D Minimum Brute-Force.*

There is also a third option that maximizes across all sets A and only evaluates one set of the $k'/2$-sized pair partitioning combinations of A. This would also grant a valid heuristic because we would have maximized across one pair partition of $S^*_{C_T}$, and because of submodularity, it would grant a valid upper bound. The approach reduces the complexity by a factor of $\mathcal{O}(|\mathbb{P}(A, k'/2)|)$, as we do not have to evaluate all sets in $\mathbb{P}(A, k'/2)$. The sharpness of the bound is somewhere between the worst and the best split. We call this approach *Upper Bound 2D Random Brute-Force* because the used split is chosen arbitrarily.

The last two brute-force approaches were only thought of later in the development of this work and are, therefore, not implemented. Theoretically, they could provide useful upper bounds or fast computable upper bounds, but the chance of them succeeding any other heuristics by a large margin is slim. Later in Section 5.6, we will discuss how all the heuristics performed in our benchmarks and also look at the running time of the search algorithm when using an oracle. The running time of the oracle replaces the time complexity of the approaches with a constant $\mathcal{O}(1)$, and we get the results for free. Using the oracle, however, only marginally improves the time, meaning that using these heuristics will not have a large impact. This suggests that these approaches are not as powerful as one might hope to expect.

We determine the sharpness of the four approaches by comparing their upper bounds to

$$h_{\max}(T, C_T, k) := f(S_T) + \max_{A \subseteq C_T, |A| = k'} \max_{P \in \mathbb{P}(A, k'/2)} \sum_{p \in P} \Delta(p \mid S_T),$$

and

$$h_{\min}(T, C_T, k) := f(S_T) + \max_{A \subseteq C_T, |A| = k'} \min_{P \in \mathbb{P}(A, k'/2)} \sum_{p \in P} \Delta(p \mid S_T).$$

These are the upper bounds that the maximum-weight matching and the dynamic

Heuristic	Time Comp.	Space Comp.	Sharpness
Max. BF	$\mathcal{O}(n^{k'} \cdot \lvert\mathbb{P}(C_T, k'/2)\rvert)$	$\mathcal{O}(k')$	$\mathrm{UB2D}_B = h_{\max}$
Min. BF*	$\mathcal{O}(n^{k'} \cdot \lvert\mathbb{P}(C_T, k'/2)\rvert)$	$\mathcal{O}(k')$	$h = h_{\min}$
Rnd. BF*	$\mathcal{O}(n^{k'})$	$\mathcal{O}(k')$	$h_{\min} \leq h \leq h_{\max}$
Matching	$\mathcal{O}(n^5)$	$\mathcal{O}(n^2)$	$\mathrm{UB2D}_M = h_{\max}$
Dynamic	$\mathcal{O}(n^{k'} k'^2)$	$\mathcal{O}(n^{k'})$	$\mathrm{UB2D}_D = h_{\min}$
Greedy	$\mathcal{O}(n^2 \log k')$	$\mathcal{O}(k')$	$\mathrm{UB2D}_G \geq h_{\max}$

Table 3: Table showing the complexity and sharpness of the four different approaches that utilize marginal gains of pairs. Heuristics marked with a star are not implemented within this thesis. Each heuristic has an additional time complexity of $\mathcal{O}(n^2 T(f))$ and space complexity of $\mathcal{O}(n^2)$ to calculate and store the marginal gains.

approach will compute.

Table 3 shows the time and space complexities of the approaches. The heuristics with the sharpest bound are the Upper Bound 2D Dynamic Programming and the Upper Bound 2D Minimum Brute-Force heuristic. While the dynamic approach is faster in theory, it also utilizes more memory and is only applicable if enough memory is available. The next best heuristic in terms of sharpness is the Upper Bound 2D Random Brute-Force heuristic. Compared to Upper Bound 2D Minimum Brute-Force, it drops the term $|\mathbb{P}(C_T, k'/2)|$, but there is no guarantee for its sharpness. Note that

$$|\mathbb{P}(C_T, k'/2)| = \binom{m}{k'} \frac{k'!}{2^{k'/2}(k'/2)!}$$

for $|C_T| = m$. Theoretically, it could deteriorate to the sharpness of the Upper Bound 2D Maximum-Weight Matching heuristic. While the matching approach has the worst sharpness of all approaches that fulfill the pair partitioning requirement, it has a fixed polynomial running time of $\mathcal{O}(n^5)$. It is, therefore, faster than all other approaches (if $k' > 5$). The greedy variant is the only heuristic that neglects the pair partitioning requirement. While it is considerably faster than all approaches, it also pays for the speed by its reduced sharpness.

A clear trend can be seen if we compare the running time and the sharpness of the approaches. In general, we want sharp bounds to prune as many nodes as possible, but to determine these bounds, much time, and in the case of the dynamic approach, much memory is needed. If these approaches are too expensive, we can resort to faster methods, but the upper bounds will be less precise, and we lose the opportunity to prune more nodes. There is no obvious answer as to which method should be preferred over any other method.

4.1.6 Odd case

Until now, we have assumed that k' is divisible by two, but this is only sometimes true, as each depth in the SE Tree has a different k'. When k' is odd, we can only select $(k'-1)/2$ pairs and have to select one candidate additionally. The question is which candidate should be selected and how do we include it. In general, one can first choose k'_1 pairs, then a single candidate with its marginal gain and then choose the remaining k'_2 pairs such that $k'_1 + k'_2 = (k'-1)/2$. We formulate the *Odd-Option for k'_1 and k'_2 Pairs* heuristic and then prove that it is a valid heuristic.

Definition 4.12. Let T be a node in a SE Tree, $k \in \mathbb{N}$ and $k' = k - |S_T|$ is odd. Let $0 \leq k'_1, k'_2$ with $k'_1 + k'_2 = (k'-1)/2$. Let h_1 and h_2 be two heuristics chosen from UB2D_B, UB2D_M, UB2D_D and UB2D_G. Let P be the set of pairs used in $h_1(T, C_T, k-2k'_2)$. Let c be the candidate with the greatest marginal gain in $C_T \setminus (\bigcup_{p\in P} p)$. Let Q be the set of pairs used in $h_2\left(T, C_T \setminus \left(\{c\} \cup (\bigcup_{p\in P} p)\right), k - 2k'_1\right)$. The *Odd-Option for k'_1 and k'_2 Pairs* heuristic is

$$\text{UB2D}_O(T, C_T, k) \coloneqq f(S_T) + \sum_{p\in P} \Delta(p \mid S_T) + \Delta(c \mid S_T) + \sum_{q\in Q} \Delta(q \mid S_T).$$

The heuristic works in three steps. First, we calculate an upper bound with $k-2k'_2$ on C_T with heuristic h_1. This will return a set of pairs P with $|P| = k'_1$. Secondly, the single candidate c is selected from $C_T \setminus \bigcup_{p\in P} p$. These are all remaining candidates that are not contained in P. Note that we select the candidate c with the greatest marginal gain. At last, we will call heuristic h_2 on $C_T \setminus \{c\} \cup \bigcup_{p\in P} p$ with $k - 2k'_1$. The sum of pairs in Q, with $|Q| = k'_2$ is an upper bound For all candidates not used by P or c. We will get a valid upper bound by summing the marginal gains of P, Q, and the marginal gain of c

We will first prove that this is a valid heuristic for two special cases and then for the general case. The first special case is $k'_1 = 0$ and $k'_2 = (k'-1)/2$, when we select the single candidate upfront. The second special case is to choose the single candidate afterward by using $k'_1 = (k'-1)/2$ and $k'_2 = 0$.

Lemma 4.7. The Odd-Option for 0 and $(k'-1)/2$ Pairs heuristic is valid.

Proof. We have to show

$$\Delta(S^*_{C_T} \mid S_T) \leq \Delta(c \mid S_T) + \sum_{q\in Q} \Delta(q \mid S_T).$$

As a quick reminder: The set $S^*_{C_T}$ is the best remaining subset in the subtree of T. If the heuristic gives an upper bound for the marginal gain of $S^*_{C_T}$, then the heuristic is valid.

Since k'_1 equals 0, we know that c is the candidate with the greatest marginal gain in C_T, and Q is the set of pairs used on the candidate set $C_T \setminus \{c\}$. We divide $S^*_{C_T}$ into $j = (k'-1)/2$ pairs $R = \{r_1, r_2, \ldots, r_j\}$ and one single element r^* that has the greatest single marginal gain of all elements in $S^*_{C_T}$. We can give an upper bound for $\Delta(S^*_{C_T} \mid S_T)$ by

$$\Delta(S^*_{C_T} \mid S_T) \leq \Delta(r^* \mid S_T) + \sum_{r \in R} \Delta(r \mid S_T).$$

We will show that

$$\Delta(r^* \mid S_T) + \sum_{r \in R} \Delta(r \mid S_T) \leq \Delta(c \mid S_T) + \sum_{q \in Q} \Delta(q \mid S_T)$$

holds in the two cases $c = r^*$ and $c \neq r^*$. Note that $\Delta(r^* \mid S_T) \leq \Delta(c \mid S_T)$ always holds because we select c to be the candidate with the greatest marginal gain. Only the inequality between the sum of marginal gains of R and Q must be shown.

Case 1: $c = r^*$ **:** The set Q was chosen by the valid heuristic h_2 on the set $C_T \setminus \{c\}$. Therefore, the sum of marginal gains of Q is a valid upper bound for all sets $A \subseteq (C_T \setminus \{c\})$ with $|A| = k' - 1$. The set $S^*_{C_T} \setminus \{r^*\}$ is such a set, and the inequality holds.

Case 2: $c \neq r^*$ **:** Since r^* is the candidate with the greatest single marginal gain in $S^*_{C_T}$ and is not equal to c, we know that c is not part of $S^*_{C_T}$. This means that $S^*_{C_T}$ is a subset of $C_T \setminus \{c\}$ and also that $S^*_{C_T} \setminus \{r^*\}$ with size $k' - 1$ is a subset of $C_T \setminus \{c\}$. As in Case 1, the sum of marginal gains of Q is a valid upper bound for all subsets of size $k' - 1$ on $C_T \setminus \{c\}$, and the inequality holds. □

Lemma 4.8. The Odd-Option for $(k'-1)/2$ and 0 Pairs heuristic is valid.

Proof. We have to show

$$\Delta(S^*_{C_T} \mid S_T) \leq \sum_{p \in P} \Delta(p \mid S_T) + \Delta(c \mid S_T).$$

We divide the best remaining subset $S^*_{C_T}$ into $j = (k'-1)/2$ pairs $R = \{r_1, r_2, \ldots, r_j\}$ and one single element r^*, that has the lowest single marginal gain of all elements in $S^*_{C_T}$. We can give an upper bound for $\Delta(S^*_{C_T} \mid S_T)$ by

$$\Delta(S^*_{C_T} \mid S_T) \leq \Delta(r^* \mid S_T) + \sum_{r \in R} \Delta(r \mid S_T).$$

The set P consists of $(k'-1)/2$ pairs and was chosen by a valid heuristic h_1 on the set C_T. The sum of marginal gains of P is, therefore, a valid upper bound for all sets $A \subseteq C_T$ with $|A| = k' - 1$. The set $S^*_{C_T} \setminus \{c^*\}$ is such a set. Therefore, the sum of marginal gains of P is a valid upper bound for the sum of marginal gains of $S^*_{C_T} \setminus \{r^*\}$.

It remains to be shown that $\Delta(r^* \mid S_T) \leq \Delta(r \mid S_T)$ also holds. The candidate c has the greatest marginal gain in $C_T \setminus \bigcup_{p \in P}$. If we sort C_T by the marginal gains of each candidate, then c has at least rank k' in the sorted set. That is because P already uses $k' - 1$ candidates, and only if all these candidates have a higher than c will c have a rank of k'. Else, c will have an even higher rank.

The candidate r^* has the lowest marginal gain of all candidates in $S^*_{C_T}$. Therefore, r^* has at most rank k' in C_T: Only if all other candidates in $S^*_{C_T}$ have a higher rank, then r^* will have rank k', otherwise it will have an even lower rank. Because c will never have a worse rank than r^* the inequality $\Delta(r^* \mid S_T) \leq \Delta(c \mid S_T)$ will always be true. □

We will now prove the validity of the general approach. We will focus on the cases that $k'_1, k'_2 \geq 1$ since the cases with $k'_1 = 0$ and $k'_2 = 0$ were just proven in Lemma 4.7 and Lemma 4.8.

Lemma 4.9. The Odd-Option for k'_1 and k'_2 Pairs heuristic with $1 \leq k'_1, k'_2$ and $k'_1 + k'_2 = (k'-1)/2$ is valid.

Proof. We have to show

$$\Delta(S^*_{C_T} \mid S_T) \leq \sum_{p \in P} \Delta(p \mid S_T) + \Delta(c \mid S_T) + \sum_{q \in Q} \Delta(q \mid S_T).$$

Let r^* be the candidate with the lowest marginal gain of all candidates in $S^*_{C_T}$. Similar to the proof in Lemma 4.8, we know that c has a better rank than r^* and, therefore, $\Delta(c \mid S_T) \geq \Delta(r^* \mid S_T)$. Due to submodularity, it remains to be shown

$$\Delta(S^*_{C_T} \setminus \{r^*\} \mid S_T) \leq \sum_{p \in P} \Delta(p \mid S_T) + \sum_{q \in Q} \Delta(q \mid S_T).$$

If there is a pair $p_i = \{p_{i_1}, p_{i_2}\} \in P$ with $p_{i_1} \in S^*_{C_T}$ and $p_{i_2} \in S^*_{C_T}$, then we can partition $S^*_{C_T}$ into $S^*_{C_T} \setminus \{p_{i_1}, p_{i_2}\}$ and $\{p_{i_1}, p_{i_2}\}$. The proof simplifies to

$$\Delta(S^*_{C_T} \setminus (\{r^*\} \cup \{p_{i_1}, p_{i_2}\}) \mid S_T) + \Delta(\{p_{i_1}, p_{i_2}\} \mid S_T) \leq \sum_{p \in P} \Delta(p \mid S_T) + \sum_{q \in Q} \Delta(q \mid S_T).$$

Because $\Delta(\{p_{i_1}, p_{i_2}\} \mid S_T)$ is part of the sum of P, we can remove it from both sides. Let $P' \subseteq P$ hold all the pairs whose candidates are part of $S^*_{C_T}$, and since the same argument holds for pairs in Q, let Q' be defined analogous. Therefore, let $\hat{P} = P \setminus P'$ and $\hat{Q} = Q \setminus Q'$ be the remaining pairs. Let $\hat{S}$ be the remaining candidates of $S^*_{C_T}$, that is, $\hat{S} = S^*_{C_T} \setminus \left(\{c^*\} \cup \bigcup_{p \in P'} p \cup \bigcup_{q \in Q'} q\right)$. It remains to be shown

$$\Delta\left(\hat{S} \mid S_T\right) \leq \sum_{p \in \hat{P}} \Delta(p \mid S_T) + \sum_{q \in \hat{Q}} \Delta(q \mid S_T).$$

We will further partition $\hat{P}$ and $\hat{Q}$. Let $\hat{P}_1$ be all pairs of P that have exactly one element in $S^*_{C_T}$ and let $\hat{P}_2$ be the pairs that have no element in $S^*_{C_T}$. The pairs of $\hat{Q}$ are partitioned the same way into $\hat{Q}_1$ and $\hat{Q}_2$. We will also partition $\hat{S}$ into the set $\hat{S}_1$, candidates that are present in a pair of $\hat{P}_1$, the set $\hat{S}_2$, candidates that are present in a pair of $\hat{Q}_1$, and the set $\hat{S}_3$ that holds candidates that are not present in any pair. Note that

$$2|\hat{P}_1| + 2|\hat{P}_2| + 2|\hat{Q}_1| + 2|\hat{Q}_2| = |\hat{S}_1| + |\hat{S}_2| + |\hat{S}_3|$$

must be true because we started with $|S^*_{C_T} \setminus \{c^*\}| = |P| + |Q|$,and since the start, we always removed the same amount of pairs from each side. We will now process each pair of $\hat{P}_1$, then each pair of $\hat{P}_2$, then each pair of $\hat{Q}_1$, and then each pair of $\hat{Q}_2$ and show for each pair, that a pair in $\hat{S}$ with at most the same marginal gain exists.

Step 1: Let $p = \{p_1, p_2\} \in \hat{P}_1$, with $p_1 \in \hat{S}_1$. If $|\hat{S}_2| > 0$, we will construct a pair with p_1 and a candidate $c \in \hat{S}_2$. If $|\hat{S}_2| = 0$, we construct a pair using p_1 and a candidate $c \in \hat{S}_3$. Both times, it will hold that $\Delta(p \mid S_T) \geq \Delta(\{p_1, c\} \mid S_T)$. Assume that the marginal gain of p would be smaller. Then P would not have been a valid upper bound because swapping p with $\{p_1, c\}$ would have granted a larger upper bound on a different set of candidates. We can remove p from $\hat{P}_1$, p_1 from $\hat{S}_1$, and c either from $\hat{S}_2$ or $\hat{S}_3$. This will be repeated until $|\hat{P}_1| = 0$. Note that $\hat{S}_1$ will then also be empty.

Step 2: Let $p = \{p_1, p_2\} \in \hat{P}_2$. Both p_1 and p_2 are in $\hat{S}_3$. If $|\hat{S}_2| \geq 2$, we will use candidates $c_1, c_2 \in \hat{S}_2$. If $|\hat{S}_2| = 1$, we will use candidate $c_1 \in \hat{S}_2$ and $c_2 \in \hat{S}_3$. If $|\hat{S}_2| = 0$, we will use candidate $c_1, c_2 \in \hat{S}_3$. The pair p is guaranteed to have a larger marginal gain than $\{c_1, c_2\}$. Otherwise, the upper bound of P would not be correct, and h_1 should have chosen $\{c_1, c_2\}$ instead of p (it had the chance to do it since P was calculated on C_T). We will remove c_1 and c_2 from their respective sets, $\hat{S}_2$ or $\hat{S}_3$, and we remove p from $\hat{P}_2$. This will be repeated until $|\hat{P}_2| = 0$.

Step 3: Let $q = \{q_1, q_2\} \in \hat{Q}_1$. If $q_1 \in \hat{S}_2$, we will use $c_1 = q_1$ and a candidate $c_2 \in \hat{S}_3$ to construct a pair. Otherwise, we will use two candidates $c_1, c_2 \in \hat{S}_3$. Since Q did not choose pair $\{c_1, c_2\}$, it can only be because q has a greater marginal gain. We can remove c_1 and c_2 from their respective set, $\hat{S}_2$ or $\hat{S}_3$. We also remove q from $\hat{Q}_1$. We repeat this process until $|\hat{Q}_1| = 0$.

Step 4: Let $q = \{q_1, q_2\} \in \hat{Q}_2$. Only candidates $c_1, c_2 \in \hat{S}_3$ remain. Since Q uses q instead of $\{c_1, c_2\}$, it must have a larger marginal gain. Otherwise, the upper bound can be invalid. We can remove c_1 and c_2 from $\hat{S}_3$ and q from $\hat{Q}_2$. This is repeated until $|\hat{Q}_2| = 0$.

For each pair in the sets $\hat{P}_1$, $\hat{P}_2$, $\hat{Q}_1$ and $\hat{Q}_2$ we have constructed a pair using the candidates of $S^*_{C_T} \setminus \{c^*\}$ that has at most the same marginal gain. Therefore, the sum of marginal gains of the pairs of the upper bound is greater than that of the constructed pairs. □

We described the two special cases separately as they are the simplest to implement, as only one of the heuristics, h_1 and h_2, is needed. All other cases need to compute an upper bound using pairs two times. In the case of the maximum-weight matching approach, we would have to build a graph once for h_1 and once for h_2 each time with different sets. While these operations are not expensive from a theoretical view, they add a factor of 2 of unnecessary overhead in practical applications since we would have to build a graph two times. Alternatively, one could alter the structure of the first graph, but this is not supported by all implementations.

Generally, there is no guarantee that one specific Odd-Option is better than any other. Table 4 shows this in two examples. In one example, choosing the single candidate first grants a lower bound than any other other Odd-Option. In the second example, Odd-Option-0-2 results in the sharpest upper bound. Since there is no guarantee that any variant results in a sharper upper bound and the special cases do not add extra overhead, only the two special cases are implemented. We do not expect that a slightly sharper marginal gain will justify the extra overhead of any option with $1 \leq k'_1, k'_2$.

	a	b	c	d	e	f
a	10	17	18	16	15	10
b	-	9	11	12	10	10
c	-	-	9	10	11	10
d	-	-	-	8	9	9
e	-	-	-	-	5	6
f	-	-	-	-	-	2

(a) Using the Odd-Option-0-2 heuristic will result in the upper bound $10 + 12 + 11 = 33$ with sets $\{\{a\}, \{b, d\}, \{c, e\}\}$. Odd-Option-1-1 will result in the sets $\{\{a, c\}, \{b\}, \{d, e\}\}$ with the sum of marginal gains of $18 + 9 + 9 = 36$. Odd-Option-0-2 results in the set $\{\{a, c\}, \{b, d\}, \{e\}\}$ with an upper bound of $18 + 12 + 5 = 35$.

	a	b	c	d	e	f
a	11	13	12	13	11	11
b	-	9	14	12	11	10
c	-	-	9	12	10	9
d	-	-	-	8	10	9
e	-	-	-	-	5	6
f	-	-	-	-	-	2

(b) Odd-Option-0-2 uses the set $\{\{a\}, \{b, c\}, \{d, e\}\}$ with a sum of marginal gains of $11 + 14 + 10 = 35$. The Odd-Option-1-1 heuristic computes the upper bound $14 + 11 + 10 = 35$ with set $\{\{b, c\}, \{a\}, \{d, e\}\}$. Odd-Option-2-0 will choose set $\{\{b, c\}, \{a, d\}, e\}$ with an upper bound of $14 + 13 + 5 = 32$.

Table 4: Two examples with candidates $\{a, b, c, d, e, f\}$. Comparing the upper bound of Odd-Option-2-0, Odd-Option-1-1, and Odd-Option-0-2 when using the maximum-weight matching as both h_1 and h_2 will show that no option is guaranteed to be better than any other.

4.1.7 Avoiding Practical Limitations

Until now, we have always assumed that we use the pairs between all $\mathcal{O}(n)$ candidates, resulting in $\mathcal{O}(n^2)$ pairs. Any approach that uses the pairs must calculate their marginal gains, which requires at least an $\mathcal{O}(n^2 \cdot T(f))$ running time. Additionally, the running time of any algorithm is needed to generate an upper bound, which ranges from $\mathcal{O}(n^5)$ to $\mathcal{O}(n^{k'})$ (see Table 3).

Such an approach is not feasible in practice since the running time of both the calculation of the marginal gains and generating the upper bound needs too much time. This is especially true if we factor in that these heuristics can be executed at each of the $\mathcal{O}(n^k)$ nodes of the SE Tree. To reduce the complexity of the approach, we introduce a hyperparameter ℓ. This hyperparameter will partition C_T into the sets P_1 and P_2. The set P_1 will hold the ℓ candidates with the greatest marginal gain in C_T, and P_2 will hold all other candidates. Note that if Dynamic Candidate Ordering is applied, the candidates will be ordered, and the partition can be applied trivially.

We apply this partition for two reasons. Since all heuristics maximize over all possible k'-sized sets, there is no use to maximize across sets with low scores. We expect that these low-scoring sets are formed by candidates at the end of the sorted candidate set. Elements at the back of the candidate set already have low scores, and pairs containing them will probably also have lower scores than pairs, which would have replaced these "bad" candidates with candidates from the front of the sorted set. The other reason is that we believe that candidates with a large score will often have a strongly diminishing return between each other. Calculating the pairs between them will substantially reduce the upper bound. For example, if we consider the vertex domination number f_{D} of the Partial Dominating Set problem and have a huge clique of size n, any pair of two vertices will reduce the upper bound from an additional $2n$ to only an additional n. Figure 6 has a clique of size 6, and any pair between the candidates v_{10} to v_{15} will have a diminishing return of 6.

From now on, we only compute the marginal gains of pairs for candidates in P_1. Therefore, we can only compute The Upper Bound 2D heuristics on P_1. This reduces the running time from $\mathcal{O}(n^2T(f))$ to $\mathcal{O}(\ell^2T(f))$ to calculate the marginal gains, and it reduces the space requirement from $\mathcal{O}(n^2)$ to $\mathcal{O}(\ell^2)$ to store them. The running time of the heuristics is also reduced and now is between $\mathcal{O}(\ell^5)$ for the maximum-weight matching and $\mathcal{O}(\ell^{k'} \cdot |\mathbb{P}(P_1, k'/2)|)$ for the brute-force approach.

However, this partitioning does come with additional costs. Previously, we knew that $S^*_{C_T} \subseteq C_T$ with $|S^*_{C_T}| = k'$. Now, we do not know how $S^*_{C_T}$ is distributed across P_1 and P_2, and theoretically, all distributions are possible. Determining any upper bound only on P_1 will not grant a valid upper bound, as Example 4.4 shows.

Example 4.4. Consider that all candidates in P_1 and P_2 have a marginal gain of 9. The candidates in P_1 have the largest diminishing return possible, that is, $\Delta(\{a, b\} \mid S_T) = \max\{\Delta(a \mid S_T), \Delta(b \mid S_T)\} = 9$ for $a, b \in P_1$, while the elements $c, d \in P_2$ have no diminishing return, that is, $\Delta(\{c, d\} \mid S_T) = \Delta(c \mid S_T) + \Delta(d \mid S_T) = 18$. The best-remaining subset $S^*_{C_T}$ is exclusively a subset of P_2. If we consider $k' = 4$, it will have a marginal gain of 36.

An upper bound exclusively on P_1 will return 18 since it uses two pairs with a marginal gain of 9. An upper bound with three candidates from P_1 and one from P_2 will result in a value of $9+9+9 = 27$, as the pair and one single candidate from P_1 and the single candidate from P_2 all have a marginal gain of 9. The upper bound using one candidate from P_1 and three from P_2 will result in a valid upper bound of 36. Selecting all candidates from P_2 will also give a valid upper bound of 36.

Partitioning Heuristic How do we solve this problem? In the proof of Lemma 4.7 and Lemma 4.9, we divided $S^*_{C_T}$ into smaller sets and gave an upper bound for each of them. We can use the same approach here. The best remaining subset $S^*_{C_T}$ can be split into two disjoint sets, $R_1 \subseteq P_1$ and $R_2 \subseteq P_2$. The sum of their marginal gains is a valid upper bound for the marginal gain of $S^*_{C_T}$ due to submodularity. The sum of an upper bound for the marginal gain of R_1 and an upper bound for the marginal gains of R_2 will then be a valid upper bound for the marginal gain of $S^*_{C_T}$.

In practice, we do not know $S^*_{C_T}$, and therefore, we also do not know R_1 and R_2 or their size. In total, there are $k' + 1$ possible combinations, which values $|R_1|$ and $|R_2|$ can take so that $|R_1| + |R_2| = k'$ is true. Since $S^*_{C_T}$ must have exactly one partition into $R_1 \subseteq P_1$ and $R_2 \subseteq P_2$, one of the combinations will have the correct sizes for R_1 and R_2. As stated above, only an upper bound with this combination will grant a guaranteed valid upper bound. No other combinations are guaranteed to produce a valid upper bound. If they overestimate the upper bound of the "true" split, then it will deteriorate the running time as it might lead to less pruning. However, underestimating the upper bound will no longer guarantee the correctness of the algorithm. In general, there is no guarantee that any combination will always overestimate or underestimate. One could construct examples similar to Example 4.4, where one combination grants an overestimating upper bound, but in another example, it will grant an underestimating invalid upper bound. To ensure we use at least a correct upper bound, we need to maximize across all combinations for $|R_1| + |R_2| = k'$. Let us properly define this method as a heuristic.

Definition 4.13. Let T be a node in a SE Tree, $k \in \mathbb{N}$ and $k' = k - |S_T|$. Let P_1, P_2 be a partition of C_T with $|P_1| = \ell$ and $\forall a \in P_1, b \in P_2 : \Delta(a \mid S_T) \geq \Delta(b \mid S_T)$. Let h be a valid Upper Bound 2D heuristic. The *Upper Bound 2D Partitioning* heuristic is

$$\mathrm{UB2D}_P(T, C_T, k) := \max_{0 \leq i \leq k'} \left(h(T, P_1, k - k' + i) + \mathrm{SUB}(T, P_2, k - i) - f(S_T) \right).$$

The upper bound on P_1 is computed by any Upper Bound 2D heuristics. Note that $k - i$ will alternate between even and odd in each iteration. The heuristic h should, therefore, act more like an interface and choose the appropriate heuristic to call. If $k - i$ is even, it will call one of the heuristics defined on even inputs, and else it should call a version of the Odd-Option heuristic. The upper bound on P_2 will be exclusively calculated by the Simple Upper Bound heuristic, as only single marginal gains are available on P_2. In general, we could replace this heuristic with any other heuristic and achieve a sharper bound, but the sharpness and running time tradeoff is probably not worth it. The term $-f(S_T)$ comes from $f(S_T) - 2f(S_T)$, and we need it since we only want the marginal gain that h and SUB provide. Let us now prove that the heuristic is correct.

Lemma 4.10. The Upper Bound 2D Partitioning heuristic is valid.

Proof. We need to show

$$\Delta(S^*_{C_T} \mid S_T) \leq \max_{0 \leq i \leq k'} h(T, P_1, k - k' + i) + \mathrm{SUB}(T, P_2, k - i) - 2f(S_T).$$

Heuristic h	Previous Time Comp.	New Time Comp.
Max Brute-Force	$\mathcal{O}(n^{k'} \cdot \lvert\mathbb{P}(P_1, k'/2)\rvert)$	$\mathcal{O}(k'\ell^{k'} \cdot \lvert\mathbb{P}(P_1, k'/2)\rvert)$
Min. Brute-Force*	$\mathcal{O}(n^{k'} \cdot \lvert\mathbb{P}(P_1, k'/2)\rvert)$	$\mathcal{O}(k'\ell^{k'} \cdot \lvert\mathbb{P}(P_1, k'/2)\rvert)$
Rnd. Brute-Force*	$\mathcal{O}(n^{k'})$	$\mathcal{O}(k'\ell^{k'})$
Max. W. Matching	$\mathcal{O}(n^5)$	$\mathcal{O}(k'\ell^5)$
Dynamic	$\mathcal{O}(n^{k'}k'^2)$	$\mathcal{O}(\ell^{k'}k'^3)$
Greedy	$\mathcal{O}(n^2 \log k')$	$\mathcal{O}(k'\ell^2 \log k')$

Table 5: Table showing the complexity of the Upper Bound 2D Partitioning heuristic, dependent on heuristic h. Each heuristic has an additional time complexity of $\mathcal{O}(\ell^2 T(f))$ and space complexity of $\mathcal{O}(\ell^2)$ to calculate and store the marginal gains. Heuristics marked with a star are not implemented within this thesis.

The best remaining subset $S^*_{C_T}$ has exactly one partition into the disjoint set $R_1 \subseteq P_1$ and $R_2 \subseteq P_2$. This is true because $S^*_{C_T} \subseteq P_1 \cup P_2 = C_T$. Assume $|R_1| = j$ and analog $|R_2| = k' - j$. Since $j \leq k'$, we will maximize with $i = j$ at one point.

The first heuristic computes an upper bound for $k - (k' - j)$ elements exclusively on P_1. Note that the value of heuristic h is built by summing marginal gains of pairs and $f(S_T)$. It, therefore, already uses $|S_T|$ candidates and only has to give an upper bound on the marginal gain for the remaining $k - (k' - j) - |S_T| = j$ elements. We subtract $f(S_T)$ from the result of the heuristic to only get the upper bound on the j elements. The same holds for the Simple Upper Bound heuristic. By subtracting $f(S_T)$, we only get an upper bound on the $k - j - |S_T| = k' - j$ candidates of P_2.

The first heuristic computes an upper bound for j elements from P_1 and is, therefore, a valid upper bound for the marginal gain of R_1. The second heuristic analog computes a valid upper bound for the marginal gain of R_2. Summing both will grant a valid upper bound for $S^*_{C_T}$. □

Note that we assumed that both $|P_1| \geq k'$ and $|P_2| \geq k'$. If this requirement is not fulfilled, one can remove all iterations with $|P_1| \leq i$ and $|P_2| \leq k' - i$. This is correct because the corresponding split into R_1 and R_2 of $S^*_{C_T}$ will not exist. Such an approach does not only hold for Upper Bound 2D heuristics but for all valid heuristics, and we will also use it later in Sections 4.2 and 4.3.

Now, let us reevaluate the complexity of the approaches in Table 3. If Dynamic Candidate Ordering is applied to a node, then the Simple Upper Bound heuristic can be implemented in $\mathcal{O}(1)$ and does not increase any complexity. While we were able to replace the parameter n by the hyperparameter ℓ, we also introduced an additional factor of k' into the time complexities. Table 5 shows how the complexities have changed.

Until now, we have not discussed how to choose the hyperparameter ℓ. An intuitive value would be $\ell = \sqrt{n}$. Then, the time complexity of evaluating the $\mathcal{O}(\ell^2)$ score function evaluations would be $\mathcal{O}(n \cdot T(f))$, and the space complexity would be reduced to $\mathcal{O}(n)$. This does not worsen the already needed complexity to apply Dynamic Candidate

Ordering on the node. Another possible value could be $\ell = k'$. The heuristics then would only construct pairs on the top k' elements. This value is, in the most cases, pretty small, so the heuristics are fast to execute.

There is a tradeoff for the hyperparameter ℓ that has to be considered. On the one hand, using a smaller value for ℓ will speed up the execution time, as both the time it takes to evaluate the pairs and calculate the upper bound. On the other hand, a smaller value will also result in a less sharp upper bound, which leads to less pruning, and more parts of the search space have to be considered. An optimal value for ℓ will, presumably, decide whether this approach is helpful in practice. In Section 5, we will experimentally evaluate which value for ℓ works best.

A Small Practical Tip Before moving to the next type of heuristic, let us briefly mention that there is a practical improvement that can be made to the Upper Bound 2D Partitioning heuristic. While maximizing over all cases $0 \leq i \leq k'$, the used Upper Bound 2D heuristic h will alternate between an even and odd number of used candidates. As previously discussed, the odd case needs special handling to choose the single candidate, but it also allows for an improvement. If a result for an odd number j of candidates is required, the result of the previous call with $j - 1$ candidates can be reused in two cases.

- Let c be the element with the greatest marginal gain of C_T, and let j be odd. If the algorithm uses the Odd-Option-0-k'_2 heuristic, it will compute an upper bound only using pairs constructed from $C_T \setminus \{c\}$. This upper bound consists of $j - 1$ candidates. If the call in the previous iteration returned an upper bound that does not use the candidate c, then the algorithm can reuse this upper bound instead of recalculating it. This is true because the upper bound also uses $j - 1$ elements, and it does not use c. It is, therefore, an upper bound for $C_T \setminus \{c\}$.

- If the algorithm uses the Odd-Option-k'_1-0 heuristic, it can always reuse the result from the previous iteration. In the even and the odd iteration, an upper bound of size $j - 1$ would have been calculated on C_T, which is redundant.

While not improving the theoretical running time, it can half the number of calls to the Upper Bound 2D heuristic, which can be quite expensive, as seen in Table 5.

4.2 Partial Brute-Force

The last section discussed multiple heuristics that use the marginal gain of pairs. Because of Lemma 2.1, these sets can have lower marginal gains than the sum of marginal gains of the single candidates. This can, of course, be advanced to sets of larger size, which can further increase the diminishing return property and grant sharper upper bounds. Before discussing newer heuristics, let us first show that the previous maximum-weight matching approach (see Section 4.1.2) is much more difficult to utilize.

The maximum-weight matching approach would need to use hyperedges that can be incident to more than two vertices. Finding a maximum-cardinality matching on 3-uniform hypergraphs, which are graphs where each edge is incident to exactly three vertices, is $\mathcal{NP}$-complete and one of Karp's 21 $\mathcal{NP}$-complete problems [38]. Finding a maximum-weight matching on hypergraphs is a generalization of the maximum-cardinality matching and, therefore, at least as hard. Exact maximum-weight matchings on hypergraphs are, therefore, not feasible. Note that approximate methods [31] exist that can approximate a maximum-weight matching on hypergraphs. To use the approximations in a valid heuristic, the approximations must always overestimate the real upper bound. Otherwise, the search algorithm will lose its exactness guarantee. Usually, there is no guarantee that an approximation will always overestimate. An approximation usually denotes an interval around the true value, so underestimations are possible. If no guarantee is known, then approximations should not be used, which is why we do not explore the approach.

The approach that we explore is a dynamic programming approach that can handle sets of multiple sizes. A hyperparameter λ is introduced that limits the size of the considered sets. The hyperparameter is called the *block size*. The candidate set C_T with $|C_T| = m$ will be partitioned into $\lceil m/\lambda \rceil$ equal parts called *blocks*. For simplicity, assume that m is divisible by λ. We denote $\eta \coloneqq m/\lambda$ as the *number of blocks*.

Each block determines for each $0 \leq i \leq \lambda$ the subset of size i with the greatest marginal gain. This is done in a brute-force manner as λ will typically be small. This is why the approach is called *Partial Brute-Force*, as we brute-force small parts of the search space. To determine a valid upper bound on C_T, one must consider all possible distributions of $S^*_{C_T}$ across the blocks. Maximizing over the sum of each possible combination will then grant a valid upper bound. Figure 9 shows one concrete example.

Let us formally define the approach as a heuristic and prove its validity. Note that we still assume that $|C_T| = m$ is divisible by λ to simplify the analysis. Assuming this does not alter the time or space complexity, and we will later partition C_T again into P_1 and P_2 to reduce the complexity of the approach. The size of P_1 will then be divisible by λ by construction.

Definition 4.14. Let T be a node in a SE Tree, $k \in \mathbb{N}$ and $k' = k - |S_T|$. Let C_T be partitioned into blocks $B_1, B_2, \dots, B_\eta$, each of size λ. Let T_i be a lookup table for the block B_i and let $T_i[j]$ denote the greatest marginal gain of the j-sized subset of B_i. The *Naive Partial Brute-Force* heuristic is

$$\mathrm{PBF}_N(T, C_T, k) \coloneqq f(S_T) + \max_{i_1+i_2+\ldots+i_\eta=k'} T_1[i_1] + T_2[i_2] + \ldots + T_\eta[i_\eta].$$

Note that all i_j cannot be negative as this would translate to sets with a negative size.

Lemma 4.11. The Naive Partial Brute-Force heuristic is valid.

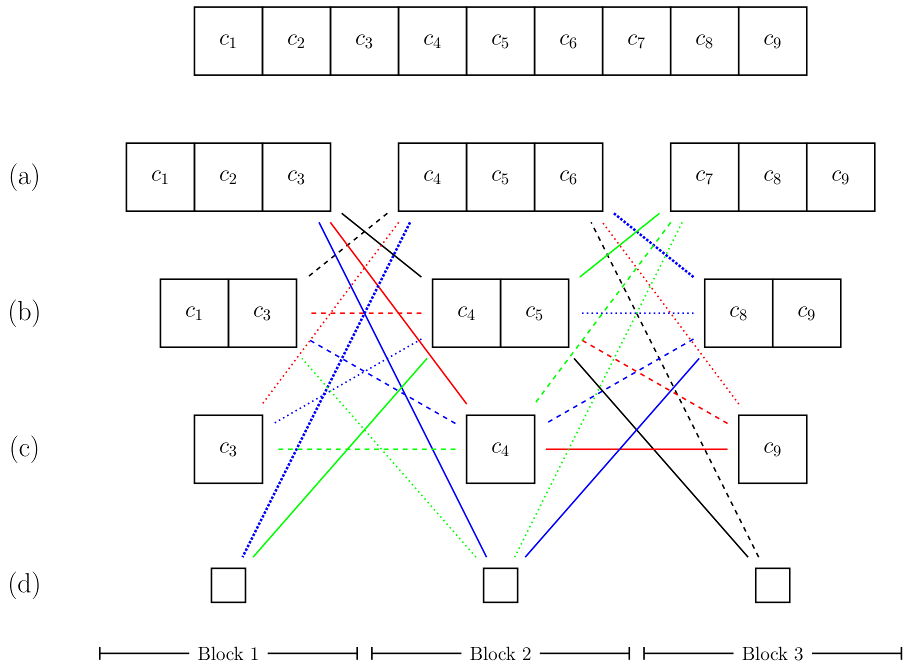


Figure 9: Example showing how the Partial Brute-Force heuristic determines a bound for $k' = 5$ candidates. The nine candidates $c_1, \ldots, c_9$ are split into three blocks of size 3. In this case, these are the blocks $\{\{c_1, c_2, c_3\}, \{c_4, c_5, c_6\}, \{c_7, c_8, c_9\}\}$. For each block, we determine the sets with the greatest marginal gain for the sizes 0, 1, 2, and 3. The sets of size 0 are at the height of the (d)-marker, the sets of size one at the height of (c), the sets of size two at the height of (b), and the sets of size three at the height of (a). The algorithm will now maximize across all combinations with five candidates in total and use one set of each block. For example, one combination will be to select $\{c_3\}$, then $\{c_4, c_5, c_6\}$ and then $\{c_9\}$. Each line distinguished by color and style represents such a combination. The empty squares at the bottom represent that no element from this block is part of the combination.

Proof. We have to show

$$\Delta(S^*_{C_T} \mid S_T) \leq \max_{i_1+i_2+\ldots+i_\eta=k'} T_1[i_1] + T_2[i_2] + \ldots + T_\eta[i_\eta].$$

We can partition $S^*_{C_T}$ into $R_1 \subseteq B_1, R_2 \subseteq B_2, \ldots, R_\eta \subseteq B_\eta$. Due to Lemma 2.1, an upper bound for $\Delta(S^*_{C_T} \mid S_T)$ is

$$\Delta(R_1 \mid S_T) + \Delta(R_2 \mid S_T) + \ldots + \Delta(R_\eta \mid S_T).$$

We will show that at least this upper bound is reached by maximizing across all combinations. During the maximization, the case $i_j = |R_j|$ for all $1 \leq j \leq \eta$ will occur because the summed sizes of the R_j is k', and each R_j is a subset of block B_j. Each entry $T_j[i_j]$ holds the greatest marginal gain of all subsets of B_j of size i_j. Therefore,

$T_j[i_j]$ is an upper bound for $\Delta(R_j \mid S_T)$ because R_j has size i_j and is a subset of B_j. Therefore

$$\sum_{j=1}^{\eta} T_j[i_j] \geq \sum_{j=1}^{\eta} \Delta(R_j \mid S_T),$$

and the heuristic estimates an upper bound for $S^*_{C_T}$. □

To initialize the tables, the approach must calculate $\mathcal{O}(\eta 2^\lambda)$ marginal gains. A time complexity of $\mathcal{O}(\eta^\lambda)$ is required to maximize across all combinations, as we have $\lambda + 1$ options to choose a subset from a block, and we have η blocks in total. Therefore, a total time complexity of $\mathcal{O}(\eta 2^\lambda T(f) + \eta^\lambda)$ is required. A space requirement of $\mathcal{O}(\eta\lambda)$ is needed to store all tables T_j.

4.2.1 Dynamic Approach

How can we reduce the running time of the previous approach? We can either reduce the running time for calculating the marginal gains or reduce the running time to compute the upper bound. Let us first discuss the calculation of marginal gains before presenting an approach to reduce the running time of the maximization process.

The term $\eta 2^\lambda T(f)$ is needed to compute the marginal gains, and dominates the running time in practice, as preliminary experiments have shown. The time-consuming factor is the evaluation of the score function f. To reduce the number of evaluations, one must either shrink the number of blocks η or the block size λ. Shrinking λ has a more significant effect and is therefore preferable. Note that values $\lambda > k - |S_T|$ are not of interest since they give upper bounds for sets greater than $k - |S_T|$. Later, we will discuss how to shrink η by partitioning C_T into P_1 and P_2.

Note that a helpful approach in practice would be not to evaluate all 2^λ subsets of a block. One could either give an upper bound for some sets and therefore save time but worsen the achieved upper bound or preferably not evaluate all sets. From the 2^λ evaluated sets, only $\lambda + 1$ remain in the table for each block. While a theoretical reduction to only $\mathcal{O}(\lambda)$ evaluations is highly unlikely (then we could solve the Cardinality-Constrained Maximization problem in $\mathcal{O}(n)$ evaluations), a reduction of the running time in practice is conceivable. We did not look for a practically faster approach to fill the lookup table, as we always assume λ to have a relatively small value. We do not expect a more sophisticated approach to save much running time if λ is small. However, if larger values for λ lead to drastically more pruning, one should consider a more efficient approach.

In total, we do not modify the running time of calculating the marginal gains but only keep η and λ relatively small so the running time does not explode. Note that smaller values for η and λ will result in less sharp upper bounds. There is a crucial tradeoff between the sharpness of the bound and the required time to compute it. Using "bad" values for the hyperparameters can and will (as preliminary experiments have shown) worsen the running time of the search algorithm.

However, we can modify the running time of the maximization step and reduce its complexity from $\mathcal{O}(\eta^\lambda)$ to only $\mathcal{O}(\eta\lambda k')$ by an approach utilizing dynamic programming. During maximization of the Partial Brute-Force heuristic, many sub-combinations are evaluated multiple times, and caching them will reduce the running time. For example, consider the case that during maximization, $i_1 = 1$, $i_2 = 2$, and all i_j should sum up to a value k'. In total, all other i_j have to sum to $k' - 3$ as i_1 and i_2 already use three candidates. However, the algorithm will also maximize over $i_1 = 0$ and $i_2 = 3$, $i_1 = 3$ and $i_2 = 0$ and $i_1 = 2$ and $i_2 = 1$. All these combinations have in common that the sum of the remaining i_j must add up to $k' - 3$. The Partial Brute-Force heuristic will naively iterate over all combinations, but this is unnecessary and can be better computed via dynamic lookup tables.

Definition 4.15. Let T be a node in a SE Tree, $k \in \mathbb{N}$ and $k' = k - |S_T|$. Let C_T be partitioned into blocks $B_1, B_2, \ldots, B_\eta$, each of size λ. Let T_i be a lookup table for the block B_i and let $T_i[j]$ denote the greatest marginal gain of the j-sized subset of B_i. We define the *PBF Dynamic Lookup Table* as

$$T[i, j] \coloneqq \begin{cases} \max_{0 \le l \le j} T_i[l] + T[i-1, j-l] & \text{if i} > 1 \\ T_1[j] & \text{if i} = 1. \end{cases}$$

With the lookup table, we can define a new heuristic. The table is only defined for values $1 \le i \le \eta$ and $0 \le j \le k'$. Other values either have no corresponding block B_i or determine an upper bound for more than k' candidates.

Definition 4.16. Let T be a node in a SE Tree, $k \in \mathbb{N}$ and $k' = k - |S_T|$. The *Dynamic Partial Brute-Force* heuristic is

$$\text{PBF}_D(T, k) \coloneqq f(S_T) + T[\eta, k'].$$

Lemma 4.12. The Dynamic Partial Brute-Force heuristic is valid.

Proof. We have to show

$$\Delta(S^*_{C_T} \mid S_T) \le T[\eta, k'].$$

We will show via induction on i that $T[i, j]$ is an upper bound for the marginal gain of all sets of size j that only use candidates from blocks B_1 to B_j. Therefore, $T[\eta, k']$ will be an upper bound for the marginal gain of all sets of size k' that use candidates from B_1 to B_η. Since $S^*_{C_T}$ is such a set, the upper bound will be valid.

The induction start is $i = 1$. The entry $T[1, j]$ is the greatest marginal gain of the j-sized subset of B_1. The induction start, therefore, holds. The induction claim is that if $T[i-1, j']$ is an upper bound for all sets of size j_1 that only use candidates from blocks B_1 to B_{i-1}, then $T[i, j]$ will be an upper bound for all sets of size j that only use candidates from blocks B_1 to B_i. The entry $T[i, j]$ computes by $\max_{0 \le l \le j} T_i[l] + T[i-1, j-l]$. Table T_i exclusively was built by subsets of B_i, while $T[i-1, j-l]$ exclusively used candidates from blocks B_1 to B_{i-1}. Due to submodularity, we know that the sum of an upper bound on the first part of a partition and an upper bound on the second part of the partition will grant an upper bound on the whole set. Because $T_i[j]$ and $T[i-1, j-l]$ are upper bounds on different parts of a partition, we know that $T[i, j]$ is an upper bound for all sets of size j using candidates from the blocks B_1 to B_i. The induction claim holds, and therefore the lemma is true. □

The approach still needs to calculate and store all marginal gains used in the tables T_i, so at least a running time of $\mathcal{O}(\eta 2^\lambda T(f))$ and space of $\mathcal{O}(\eta\lambda)$ is needed. The PBF Dynamic Lookup Table has the same space requirement, so the complexity is not worsened. To calculate one entry in the table, we need to maximize across $\mathcal{O}(k')$ additions. This makes for a total time complexity of $\mathcal{O}(\eta\lambda k')$ to calculate each entry in the table. In total, the heuristic has a time complexity of $\mathcal{O}(\eta 2^\lambda T(f) + \eta\lambda k')$ and a space complexity of $\mathcal{O}(\eta\lambda)$.

The Dynamic Partial Brute-Force heuristic should be preferred over the Naive Partial Brute-Force heuristic, as it has a faster running time while not worsening the theoretical required space. In practice, the required space is doubled for the Dynamic Partial Brute-Force heuristic, but since both η and λ are relatively small, it has little impact.

Lastly, we also want to introduce the *Partial Brute-Force Partitioning* heuristic to reduce the number of blocks η.

Definition 4.17. Let T be a node in a SE Tree, $k \in \mathbb{N}$ and $k' = k - |S_T|$. Let P_1, P_2 be a partition of C_T with $|P_1| = \eta\lambda$ and $\forall a \in P_1, b \in P_2 : \Delta(a \mid S_T) \geq \Delta(b \mid S_T)$. Let h be a valid Partial Brute-Force heuristic. The *Partial Brute-Force Partitioning* heuristic is

$$\mathrm{PBF}_P(T, C_T, k) \coloneqq \max_{0 \leq i \leq k'} h(T, P_1, k - k' + i) + \mathrm{SUB}(T, P_2, k - i) - f(S_T).$$

The partitioning heuristic is also valid. The proof is analogous to the correctness proof of the Upper Bound 2D Partitioning heuristic from Lemma 4.10. We do not repeat the proof here. Both η and λ are hyperparameters that need to be correctly tuned. Note that if $\eta\lambda > |C_T|$, we will reduce η by one until $\eta\lambda \leq |C_T|$ to apply the heuristic. The partitioning heuristic was developed for the same reason the Upper Bound 2D Partitioning heuristic was developed. We expect that the diminishing returns of candidates in P_1 will be much greater than those in P_2 and that P_1 will hold more candidates that are part of a maximizing set. Computing a sharper upper bound on P_1 will, therefore, have more impact than a sharper upper bound on P_2.

4.3 Divide-And-Conquer Heuristic

The Upper Bound 2D and the Partial Brute-Force heuristics both have a partitioned version that uses the sets P_1 and P_2 instead of C_T. We introduced the partition because the heuristics are generally too expensive if calculated on the full candidate set C_T, and we presume that candidates at the back of C_T do not substantially improve the upper bound. In other words, only the candidates at the front are essential as they have large diminishing returns and lead to substantially sharper upper bounds. The Upper Bound 2D only uses the diminishing return of two candidates, and the Partial Brute-Force uses at most λ candidates. However, if we believe that these candidates will have strongly diminishing returns, why not also calculate the most accurate upper bound for them? While this will take more time, it will, presumably, lead to substantially sharper upper bounds and, therefore, to more pruning. The Divide-and-Conquer heuristic presented now applies a recursive approach to achieve this. The idea is to search on P_1 for the exact upper bound and use the SUB heuristic on P_2. Let us formally define the heuristic and prove its validity.

Definition 4.18. Let T be a node in a SE Tree, $k \in \mathbb{N}$ and $k' = k - |S_T|$. Let P_1 and P_2 be a partition of C_T. Let $S(P, j, C)$ be a search algorithm that returns the greatest marginal gain of the j-sized subset of P with base set C. Let h be a valid heuristic. The *Divide-And-Conquer* heuristic is

$$\mathrm{DAC}(T, C_T, k) \coloneqq f(S_T) + \max_{0 \leq i \leq k'} S(P_1, i, S_T) + h(T, P_2, k - i) - f(S_T).$$

Lemma 4.13. The *Divide-And-Conquer* heuristic is valid.

Proof. We have to show

$$\Delta(S^*_{C_T} \mid S_T) \leq \max_{0 \leq i \leq k'} S(P_1, i, S_T) + h(T, P_2, k - i) - f(S_T).$$

We can partition $S^*_{C_T}$ into $R_1 \subseteq P_1$ and $R_2 \subseteq P_2$ and give an upper bound for $S^*_{C_T}$ by $\Delta(R_1 \mid S_T) + \Delta(R_2 \mid S_T)$. At some point, the heuristic will maximize with $i = |R_1|$ and $k' - i = |R_2|$. Since R_1 is a subset of P_1, the search algorithm S will at least return the marginal gain of the set. Since R_2 is a subset of P_2, the valid heuristic will return at least an upper bound for R_2. Therefore, it will at least calculate an upper bound of $\Delta(R_1 \mid S_T) + \Delta(R_2 \mid S_T)$, and the heuristic is valid. $\square$

To compute the value of $S(P_1, i, C)$ we will use the here developed search algorithm. It has a time complexity of $\mathcal{O}(n^k T(f))$ with n being the number of elements and k being the cardinality constraint. Let $|P_1| = n_1$ and $|P_2| = n_2$, then the Divide-And-Conquer heuristic has a time complexity of $\mathcal{O}(k'(n_1^{k'} + T(h)))$, with $T(h)$ denoting the running tme of heuristic h. Note that the SUB heuristic has a running time of $\mathcal{O}(1)$, so using it does not influence the complexity.

The heuristic has two drawbacks in practice. The first is algorithmic and can be solved with a specialized algorithm, but the second might make the heuristic not as powerful as one might hope.

The first problem arises when maximizing across all $0 \leq i \leq k'$ and searching on P_1 for the exact upper bound each time. In the first and second iteration, we will search for the best subset with zero candidates and one candidate, which is trivial. However, the algorithm will search for the best subset with two candidates, then three candidates, and so on. In each iteration, the algorithm will expand the search tree from the root of P_1 to the desired size. This means that when the algorithm looks for a subset of size i, it will also expand across the subsets of $i-1$, $i-2$, and so on. The algorithm has to recalculate nodes of the SE Tree multiple times, which is inefficient. However, storing the expanded SE Tree is also not an option since it can have an exponential size. A solution to the problem would be a search algorithm that can determine the best set of every size instead of only one best set. However, such a solution is currently not implemented and would need some engineering.

The second drawback arises from the fact that we use Candidate Reduction. Assume we are at node T and partition C_T into P_1 and P_2. If we do not use the Divide-And-Conquer heuristic, we will move further down the tree, and because of the diminishing returns, the marginal gain of candidates in P_2 (and P_1) will get smaller. This results in Candidate Reduction, removing these candidates. Depending on how much diminishing return there is, we could end up at an ancestor T' of T with $C_{T'} \approx P_1$. In other words, further down the tree, we will only consider the candidates in P_1, and the algorithm will naturally search for an exact upper bound. Additionally, the algorithm has to search $C_{T'}$ only once while the Divide-And-Conquer heuristic will search it k' times.

The Divide-And-Conquer heuristic could be an expensive but powerful heuristic that prunes many nodes of the SE Tree. However, it suffers from the lack of a specialized algorithm, and its effectiveness may be decreased in the presence of Candidate Reduction.

4.4 Additional Methods

In this section, we will give a brief overview of five additional methods that can be used with all presented heuristics. We say that a heuristic call is *successful* if it leads to pruning the node. Otherwise, it is *unsuccessful*. The first method applies Candidate Reduction and potentially has the greatest running time improvement, as we can prune the SE Tree, although the heuristic call was unsuccessful. The other four methods are concerned with improving the ratio of successful heuristic calls to unsuccessful heuristic calls.

Candidate Reduction In Lemma 3.6, we proved that we can use any valid heuristic h to derive an upper bound $h(T, C_T \setminus \{c\}, k-1) + \Delta(c \mid S_T)$ to determine if c can be part of a maximizing set. Since the newly developed heuristics result in sharper upper bounds than the result of the SUB heuristic (discussed in Section 4.5), they can likely remove more candidates. Applying the method can positively affect the running time, as fewer candidates result in smaller subtrees and fewer leaf nodes.

While $\Delta(c \mid S_T)$ is already given by Dynamic Candidate Ordering, we additionally need the value $h(T, C_T \setminus \{c\}, k-1)$ to decide if c can be removed. Acquiring this value can be trivial but can also have a negative side effect: We specifically developed the *partitioning* versions of our heuristics to decrease the computational effort on the partition P_1, and to only considere the Simple Upper Bound heuristic on P_2. This forced us to maximize across all possible partitions of $S^*_{C_T}$ across P_1 and P_2, e.g., we had to consider $k' + 1$ cases.

If we compare $h(T, C_T \setminus \{c\}, k-1)$ and $h(T, C_T, k)$ to each other, we will notice that they maximize almost across the same cases. That is, $h(T, C_T, k)$ calculates the case with $k'_1 = i, k'_2 = k' - i$, and $h(T, C_T \setminus \{c\}, k-1)$ calculates the case $k'_1 = i, k'_2 = k' - i - 1$ at some point. Since we only consider the SUB heuristic on P_2, we can remove the last candidate of SUB from the result of $h(T, C_T, k)$, and we get a solution for $h(T, C_T \setminus \{c\}, k-1)$. We can, therefore, transfer the result to $h(T, C_T \setminus \{c\}, k-1)$ for little cost if we compute $h(T, C_T, k)$.

However, this is only half the truth. When we calculate the value $h(T, C_T, k)$, we are not interested in the exact value, but only if $h(T, C_T, k) > s_{\text{best}}$ to know whether we can prune the node. Since $h(T, C_T, k)$ maximizes across distinct cases, we can preemptively return if we know that $h(T, C_T, k) > s_{\text{best}}$ will be true. However, if we preemptively exit, there is no guarantee that $h(T, C_T \setminus \{c\}, k-1)$ was correctly calculated. This is because the case that maximizes $h(T, C_T, k)$ does not necessarily maximize $h(T, C_T \setminus \{c\}, k-1)$. If we want to use Candidate Reduction with a partitioning heuristic, we always have to calculate all $k+1$ cases of the partitioning heuristic. Therefore, Candidate Reduction should only be used if the gain from removing some candidates outweighs the cost of computing all $k+1$ cases for each heuristic call.

Safe Skip The *Safe Skip* method concerns itself with removing unsuccessful heuristic calls. Assume we have calculated a heuristic h, and h used the collection of sets $\{C_1, C_2, \ldots, C_j\}$ to calculate its upper bound. If the heuristic cannot prune the node, then the search algorithm will explore the candidate c at the front of the candidate set. The algorithm will return from the recursive call at some point, and we can remove c from the candidate set. Do we have to recalculate h?

The answer is that if c was not used by any C_i then there is also no need to recompute h. The heuristic will at least return the same result $\{C_1, C_2, \ldots, C_j\}$ as the collection of candidates is still available. A not-so-obvious answer is that even if $c \in C_i$ for some i, the upper bound of $\{C_1, C_2, \ldots, C_j\} \setminus \{C_i\}$ may still be larger than s_{best}, so we will still not prune the node. Recalculating h will at least return the same upper bound, so the node will not be pruned.

Therefore, if $c \in C_i$ and the upper bound of $\{C_1, C_2, \ldots, C_j\} \setminus \{C_i\}$ is at most s_{best}, then we should recompute h (it would even be more precise to consider $\{C_1, C_2, \ldots, C_i \setminus \{c\}, \ldots C_j\}$, but this can result in a score function evaluation). Only then is it possible that a recalculation of h will lead to pruning.

Note that the method introduces some overhead if enabled, as we have to keep track of the C_i. This mainly involves much copying during the heuristic calls. However, in the case of the Partial Brute-Force heuristic, it also requires that we keep track of the candidates while calculating the PBF Dynamic Lookup Table. This adds about a factor of 2 in practice for both the time and memory complexity.

Lazy Skip In contrast to the Safe Skip method that precisely indicates when a computation of the heuristic is needed, we can also use a simpler approach. However, the approach cannot guarantee whether the recalculation is helpful. We will use three values, $a_0, a, b \in [0, 1]$, to give a simple rule for when to recalculate. We call a_0 the *start value*, a the *decision value*, and b the *increment*. Each time we reach a node for the first time, we set $a = a_0 + b$. If $a \geq 1$, we update a by $a = a - 1$ and compute the heuristic. Now, each time the search algorithm returns from exploring a candidate, we will update a by $a = a + b$, and if $a \geq 1$, we will again reduce it by one and recalculate the heuristic.

This method is a more general version of recalculating every ith-heuristic call. For example, we will always recompute the heuristic by setting $a_0 = 0$ and $b = 1$. If we set $b = 0.5$ instead, we will skip the first call and only recalculate every second call. Setting $b = 1/3$ would only recalculate every third call, and so on.

How well this method performs depends on how accurate the user can make assumptions about the heuristics. Note that this is generally very hard to tune correctly, and we do not expect some "magic" configuration of a_0 and b to exist that recalculates the heuristic only if truly needed. However, one can control how often the heuristics should be executed through the hyperparameters. For example, by setting $b = \epsilon$ for some small constant ϵ and $a_0 = 1 - \epsilon$, we will only calculate the heuristic when reaching the node for the first time and never else. This is a good way to limit the number of score function evaluations if the score function is costly.

Depth Constraint In general, pruning nodes at a low depth (nodes close to the root) has a much greater effect than pruning nodes at the bottom of the SE Tree. This is because the nodes closer to the root hold a substantially greater subtree than those at the bottom. Therefore, one might be willing to invest more time in pruning nodes at the top than at the bottom. Additionally, there are far more nodes at greater depths of the tree than there are at the top of the tree. The heuristics will thus be called much more often at these depths.

Because of this, we introduce two additional hyperparameters, $l, h \in [0, 1]$. The value $l \cdot k$ denotes the lowest depth at which the heuristic should be calculated, and the value $h \cdot k$ the greatest depth.

One could give a much more general version by introducing a bit vector of size k that holds 1 at index i if the heuristic should be executed at depth i and 0 otherwise. In general, tuning the hyperparameters l and h (or the bit vector) is likely very specific for each score function and probably also specific for each problem instance for the score function. However, this approach is an easy way to control the heuristics at each depth.

Simple Upper Bound Constraint A way to increase the success chance of a heuristic h is to adaptively start it based on the result of SUB. If the result of the SUB heuristic is very large, will h be able to prune the node? Probably not, and we should not calculate it. On the other hand, if the upper bound is only marginally greater than s_{best}, then calculating h will (likely) prune the node, and we should do it.

Let s_{SUB} be the upper bound of the SUB heuristic, and let s_h be the result of heuristic h. Usually, we will prune the node based on h if

$$s_h - f(S_T) \leq s_{\text{best}} - f(S_T) < s_{\text{SUB}} - f(S_T),$$

but this is only known after calculating h. This method assumes that we can estimate $s_h - f(S_T)$ by $p \cdot (s_{\text{SUB}} - f(S_T))$ with some value $p \in [0, 1]$. To determine if a node will be pruned after calculating h, we will first determine whether

$$p \cdot (s_{\text{SUB}} - f(S_T)) \leq s_{\text{best}} - f(S_T)$$

is true and only then calculate h. If our estimation were correct every time, we would always prune the node after executing h, which would be the best case. However, there (presumably) is no value p that accurately determines $s_h - f(S_T) = p \cdot (s_{\text{SUB}} - f(S_T))$. We therefore require that the user sets p with some prior knowledge. The parameter p informally indicates how much sharper h will be (or at least what the user thinks) compared to the SUB heuristic. If $p = 1$, then we do not think that the heuristic returns a sharper upper bound, so we will never calculate h. The further we move p to 0, the more confident we are in the heuristic, and we will calculate it more often. If $p = 0$, then we expect that the heuristic returns $f(S_T)$ and can prune every node. The heuristic will be calculated every time in that case.

The advantage of the heuristic is that it only requires the computation of s_{SUB}, which is very cheap. Well-chosen values for p can increase the chance that a heuristic call will result in pruning the node.

4.5 Comparison of all Heuristics

All developed heuristics aim to determine a sharp upper bound to prune many nodes. Which of the heuristics should we choose? For example, if one heuristic, always generates a sharper upper bound and is faster to compute than a second heuristic, we do not have to implement the second heuristic as it offers no benefits. So far, we can only make the decision based on the running time because we do not know which heuristic generates the sharpest upper bound. In this section, we will compare the heuristics based on their upper bound and see which heuristic is guaranteed to prune more nodes than another.

We measure the effectiveness of heuristics by determining which heuristic *dominates* another heuristic.

Definition 4.19 (Dominating heuristics)**.** Let h and g be two valid heuristics. If

$$h(T, C_T, k) \leq g(T, C_T, k)$$

for all nodes T and $k \in \mathbb{N}$, then h *dominates* g.

If a heuristic dominates another one, we know it will at least prune the same nodes, if not even more. Note that the relation is only based on the ability to prune nodes and does not factor in time or space complexities. From a practical perspective, however, there is little use for a heuristic that dominates all other heuristics but has a large running time. For example, we could combine all heuristics into one function and choose the minimum of each heuristic. It would dominate the other heuristics but has a large running time in practice.

SUB and the Advanced Heuristics Let us consider the SUB heuristic and any newly developed heuristics to get an intuition for dominating heuristics. The newly developed heuristics exploit the diminishing returns between candidates to determine sharper upper bounds. In the worst case, there is no diminishing return, and the heuristics are forced to choose the top k' marginal gains, like the Simple Upper Bound heuristic. The advanced heuristics determine at most the same upper bound as SUB and, therefore, dominate SUB.

The only exception is the Upper Bound 2D Greedy heuristic. It can choose a candidate multiple times, and if the candidate has a large marginal gain, it can appear in multiple pairs. These pairs will, therefore, have a large marginal gain. The sum of pairs will result in a larger upper bound than the result of SUB. It can, therefore, not dominate SUB. However, SUB also does not dominate the greedy approach. If there are strongly diminishing returns, the sum of pairs can result in a sharper upper bound than SUB, which will, therefore, prune more nodes. We will ignore the greedy approach in the analysis of the other cases.

UB2D and PBF Both heuristics do not dominate each other. The Partial Brute-Force heuristic divides the candidates into blocks and only determines the marginal gains of sets inside each block. In the worst case, these sets have no diminishing return. The Upper Bound 2D heuristic can choose candidates of different blocks, and if these have a diminishing return, then the Upper Bound 2D will have a sharper upper bound. However, if there are diminishing returns, the Partial Brute-Force heuristic can achieve a sharper upper bound since it uses larger sets.

Divide-And-Conquer and UB2D By P_1, we denote the partition for the Divide-And-Conquer heuristic in this paragraph. There are two cases we can consider: $\ell \leq |P_1|$, and $\ell > |P_1|$. The Divide-And-Conquer heuristic calculates the exact upper bound on P_1. If $|P_1| \geq \ell$, then the Divide-and-Conquer heuristic dominates the Upper Bound 2D heuristics since it is exact. Otherwise, none of the heuristics dominate each other. Assume that $k' = \ell = |P_1| + 2$ and that all candidates in P_1 have no diminishing return, while the two candidates, used by a UB2D heuristic, have a strongly diminishing return with all candidates in P_1. The Upper Bound 2D heuristic can then use the two candidates to construct a sharper upper bound, while the Divide-And-Conquer can only use candidates in P_1 and the marginal gains of the last two elements. In such a case, the Upper Bound 2D heuristic can return a sharper upper bound. While it is a very specific case, it still prevents the Divide-And-Conquer heuristic from dominating the Upper Bound 2D heuristic.

Divide-And-Conquer and PBF The same argument we made with the Upper Bound 2D heuristic can be made here. Therefore, only if $\lambda \cdot \eta \leq |P_1|$ will the Divide-and-Conquer heuristic dominate the Partial Brute-Force heuristic. If PBF can use more candidates, we can construct cases where it will have a sharper upper bound than the Divide-and-Conquer heuristic.

Conclusion The developed heuristics all dominate SUB (except the Upper Bound 2D Greedy heuristic). This should be expected of any new heuristic since SUB has only a little computational cost. Any heuristic that puts more effort into determining an upper bound should also return a sharper upper bound.

There is, however, no domination relation between the newly developed heuristics. The only new insight that we got is that the recursive heuristic, which is very expensive, can be outperformed by the Upper Bound 2D and the Partial Brute-Force heuristic. However, we presume that this only rarely happens as there usually is a diminishing return between candidates.

Note that we can expand the idea of dominating heuristics further to include Candidate Reduction. Instead of measuring the upper bounds, we measure which candidates get removed after executing the heuristic or Candidate Reduction. If one of the methods always removes the same candidates as the other, then the method would dominate the other. For example, Candidate Reduction with SUB will dominate SUB because when SUB prunes the node, Candidate Reduction would also remove all candidates. However, if SUB does not prune the node, Candidate Reduction can still remove candidates.

None of the heuristics here presented dominates Candidate Reduction, but Candidate Reduction (with SUB as the heuristic) also does not dominate them. If we assume that there are no diminishing returns, all heuristics will calculate the result of SUB and Candidate Reduction dominates SUB. However, the heuristics may be able to prune the node with their sharper bounds, which Candidate Reduction cannot achieve.

4.6 Lazy Evaluations

This section will not discuss a new heuristic to prune the search tree but how to update the marginal gains for Dynamic Candidate Ordering. This adresses the bottleneck of the Basic Search algorithm (Algorithm 2), which is the time it takes to evaluate the score function: At each of the $\mathcal{O}(n^k)$ nodes of the SE Tree, the marginal gain of each candidate is needed, which requires a running time of $\mathcal{O}(n \cdot T(f))$ per node. This is very time-consuming, especially for expensive score functions. Is such an exhaustive calculation necessary?

In the previous sections, we assumed that candidates at the end of the sorted candidate set are unimportant. They have a low score and will probably not be part of a maximizing set. If we make the same assumption here, it would save time not to evaluate f for these candidates. They already have a low marginal gain at parent node T^*, and updating them at node T will not improve the marginal gain because of submodularity. Keeping these low marginal gains up to date is this not as important. This can also hold for candidates with a large marginal gain. If their marginal gain only changed a little, it would have been advantagous not to recalculate it, as their new score will likely not influence the search algorithm.

Another reason not to update certain candidates is that they are not used by any heuristic. For example, we developed the partitioning heuristics that utilize only a limited number of candidates. These candidates are at the front of the candidate set, and candidates at the back will not be used. If they are not used, then why should we update them?

The Lazy Evaluation approach addresses both of these ideas. It wants to update only the marginal gains of elements that are actually considered by the search algorithm. In this context, let us remark that Minoux [54] also used a Lazy Evaluation approach to improve the running time of the greedy algorithm proposed by Nemhauser et al. [57].

We will broadly categorize the Lazy Evaluation approaches into two categories: Search-Altering Lazy Evaluation and Search-Retaining Lazy Evaluation. The first category uses Lazy Evaluations in a more simple manner. However, as a side effect, it can enlarge the search space and change the order in which nodes are traversed, compared to the search algorithm that does not utilize Lazy Evaluations. We presume that such an approach can have great benefits, but also large drawbacks, so the user will be required to tune the correct hyperparameters. The method uses *Update Schemes* and is discussed in Section 4.6.1.

The second category uses Lazy Evaluation and retains the same search space and traversal order as previously. Currently, no hyperparameters are needed for the approach and in general it relies heavily on an efficient computation to be used. The approach is discussed in Section 4.6.2 under the name *Just as Many as We Need*.

4.6.1 Search-Altering Lazy Evaluation

To formalize the Search-Altering Lazy Evaluation approach, we define *update schemes*.

Definition 4.20. Let T be a node in an SE Tree, and let T^* be its parent. The function $U_T : \mathcal{U} \to \mathbb{R}$ subject to some predicate π is defined via

$$U_T(c) := \begin{cases} \Delta(c \mid S_T) & \text{if } \pi \text{ is true,} \\ U_{T^*}(c) & \text{otherwise} \end{cases}$$

is called an *update scheme* with respect to node T. We call the value $U_T(c)$ the *lazy marginal gain*.

A lazy marginal gain is either the exact marginal gain $\Delta(c \mid S_T)$ that requires a function evaluation, or it is the value assigned by the update scheme of the parent. The parent's update scheme can either return $\Delta(c \mid S_{T^*})$ or use its parent update scheme to return a value for c. Note that this definition requires the existence of a parent. The root node would not have a properly defined update scheme. To circumvent this, we assign the *default update scheme* to the root node.

Definition 4.21. Let T be a node in a SE Tree. The function $\mathrm{DU} : \mathcal{U} \to \mathbb{R}$ with

$$\mathrm{DU}(c) \coloneqq \Delta(c \mid S_T)$$

is called the *default update scheme.*

The default update scheme arises if the condition of an update scheme is always true. Until now, all nodes in the SE Tree used the default update scheme. Previously, Definition 3.11 used the marginal gain of the candidates to apply Dynamic Candidate Ordering on node T. That is, the candidates were reindexed such that

$$\Delta(c_1 \mid S_T) \geq \Delta(c_2 \mid S_T) \geq \ldots \geq \Delta(c_m \mid S_T).$$

We reformulate this approach to incorporate update schemes. Dynamic Candidate Ordering for an update scheme U_T will then be applied by reindexing the candidates such that

$$U_T(c_1) \geq U_T(c_2) \geq \ldots \geq U_T(c_m).$$

Using the default update scheme will result in the same ordering and requires the same time.

Any update scheme must fulfill a necessary requirement so that the search algorithm does not lose its exactness guarantee. Each used scheme must be *valid* because only these update schemes will guarantee the correctness of the search algorithm. This is proved in Lemma 4.14. Note that all of our developed heuristics can be expressed via $h(T, C, k) = g(T, C, k) + \sum_{e \in E} \Delta(e \mid S_T)$: The function $g(T, C, k)$ represent all marginal gains of sets, while the sum represents all marginal gains of single candidates. The validity proof will only hold for these types of heuristics.

Definition 4.22 (Valid Update Scheme)**.** Let T be a node in a SE Tree. We define update scheme U_T as *valid* if $U_T(c) \geq \Delta(c \mid S_T)$ holds for each candidate $c \in C_T$.

Lemma 4.14. Let T be a node in a SE Tree, and let U_T be a valid update scheme. Each valid heuristic h that uses $U_T(e)$ instead of $\Delta(e \mid S_T)$ will stay valid.

Proof. We divide the value $h(T, C, k)$ into two parts, that is

$$h(T, C, k) = g(T, C, k) + \sum_{e \in E} \Delta(e \mid S_T).$$

The function g does not depend on the marginal gain of single candidates, and the second part exclusively relies on the marginal gains of single candidates. Let us now consider the heuristic

$$h'(T, C, k) = g(T, C, k) + \sum_{e \in E} U_T(e)$$

that uses $U_T(e)$ instead of $\Delta(e \mid S_T)$. The function value of g is not altered, as it does not include single marginal gains. It is, therefore, sufficient to only consider the second part. Since $U_T(e) \geq \Delta(e \mid S_T)$ for each $e \in C_T$ it follows that $\sum_{e \in E} U(e) \geq \sum_{e \in E} \Delta(e \mid S_T)$ since $E \subseteq C_T$ and therefore $h'(S_T, C, k) \geq h(S_T, C, k)$. The heuristic h' is an upper bound for h, and because h is valid, h' is also valid. □

In this work, we will only consider update schemes defined for single candidates and not for sets of candidates, which is why we assume that $g(T, C, k)$ does not change its value. However, there is no obvious reason why update schemes should not also be used on sets. We limited the number of needed sets by various hyperparameters, so there was no need for update schemes for sets. However, if heuristics need to calculate marginal gains for many sets, it may also be beneficial to evaluate some of them lazily.

The proof of Lemma 4.14 shows why update schemes belong to the Search-Altering Lazy Evaluation category. The heuristics become less sharp when we use update schemes and, therefore, will be less likely to prune nodes. Additional nodes will be explored that were not explored previously. Also, the candidates are sorted differently by Dynamic Candidate Ordering. We may thus explore subtrees previously ruled out by the search algorithm.

If we, for example, never update any candidate and only use the marginal gains from the root node, then we generally explore too many parts of the search tree. While we have only a minimal cost for score function evaluations, we have to search large parts of the SE Tree, and in total, we will probably need more time. On the other hand, if we continually update the marginal gains, we have a considerable cost to evaluate the score functions, but we will consider a smaller part of the search tree.

The question is: Is there some way to reduce the number of score function evaluations while not massively expanding the search space? We assume the most important candidates will be at the front of the sorted candidate set. Therefore, we want to update them and only lazily evaluate the candidates at the back of the candidate set. Here, we will present two specific update schemes: The *average update scheme* and the *rank update scheme*.

Average Update Scheme The *average update scheme* only calculates the marginal gain of a candidate at node T if the lazy marginal gain of the candidate at parent node T^* was greater than the average required marginal gain.

Definition 4.23. Let T be a node in a SE Tree, T^* be the parent of T and $k \in \mathbb{N}$. Let s_{best} be the currently best-found score of the search algorithm and let $r_{\text{avg}} = (s_{\text{best}} - f(S_T))/(k - |S_T|)$ be the average marginal gain required per candidate, to surpass the best score. We define the *average update scheme* $U_T^{\text{avg}} : \mathcal{U} \to \mathbb{R}$ as

$$U_T^{\text{avg}}(c) := \begin{cases} \Delta(c \mid S_T) & \text{if } U_{T^*}(c) \geq r_{\text{avg}}, \\ U_{T^*}(c) & \text{otherwise} \end{cases}$$

The hope is that all candidates c with $U_{T^*}(c) \geq r_{\text{avg}}$ recieve a marginal gain that is below the average marginal gain required. If this aim is achieved for every candidate $c \in C_T$, then SUB will prune the node T, and we have saved the evaluations for all candidates with $U_{T^*}(c) < r_{\text{avg}}$. Only reducing a few candidates with $U_{T^*}(c) \geq r_{\text{avg}}$ to a smaller marginal gain can also result in pruning the tree, but no guarantee can be given. Only one candidate with a large marginal gain after the update may be sufficient to prevent the Simple Upper Bound heuristic from pruning the node.

The threshold of r_{avg} is specifically set for SUB, but any threshold is applicable, and there is no guarantee that one threshold is better than the other. As a hyperparameter, one can also give an additional parameter $y \in \mathbb{R}$ to scale r_{avg}, that is the update scheme compares $U_{T^*}(c) \geq y \cdot r_{\text{avg}}$. If $y = 0$, then the average update scheme will behave like the default update scheme, and if $y = \infty$ no candidate will ever be updated.

Rank Update Scheme The *rank update scheme* only updates a candidate's marginal gain if the candidate has a sufficiently high rank in the candidate set of the parent.

Definition 4.24. Let T be a node in an SE Tree and T^* be the parent of T. Let $l : \mathbb{T} \to \mathbb{N}$ be a function that maps a ranking threshold to each node. We define the *rank update scheme* $U_T^l : \mathcal{U} \to \mathbb{R}$ as

$$U_T^l(c) \coloneqq \begin{cases} \Delta(c \mid S_T) & \text{if } \operatorname{rank}(c, C_{T^*}) \leq l(T), \\ U_{T^*}(c) & \text{otherwise} \end{cases}$$

The function $\operatorname{rank}(c, C_{T^*})$ will return the rank of c in the set C_{T^*}. The candidate with the greatest marginal gain will have a rank of 1, the second-greatest marginal gain will have a rank of 2, and so on. If Dynamic Candidate Ordering was applied to the parent node T^*, then determining each candidate's rank is straightforward.

This scheme is only as powerful as the ranking function l. It should only update the relevant candidates, which likely have a high rank, and ignore the uninteresting ones. Example functions for l could be

$$\begin{aligned} l(T) &= l_0 && l_0 \in \mathbb{N} \\ l(T) &= \lfloor n \cdot y \rfloor && y \in [0,1] \\ l(T) &= \lfloor k \cdot y \rfloor && y \in \mathbb{R} \\ l(T) &= \lfloor |C_T| \cdot y \rfloor && y \in [0,1] \\ l(T) &= \lfloor (k - |S_T|) \cdot y \rfloor && y \in \mathbb{R}. \end{aligned}$$

Note that the function $l(T) = |C_T|$ results in the default update scheme, and $l(T) = 0$ will never update any candidate. This update scheme is, in general, also search-altering.

Combination of Update Schemes The average update scheme U_T^{avg} and the rank update scheme U_T^l can be combined to form a new update scheme. We define the *average and rank update scheme* as

$$U_T^{\wedge}(c) \coloneqq \begin{cases} \Delta(c \mid S_T) & \text{if } U_{T^*}(c) \geq r_{\text{avg}} \text{ and } \operatorname{rank}(c, C_{T^*}) \leq l(T), \\ U_{T^*}(c) & \text{otherwise} \end{cases}$$

and the *average or rank update scheme* as

$$U_T^{\vee}(c) \coloneqq \begin{cases} \Delta(c \mid S_T) & \text{if } U_{T^*}(c) \geq r_{\text{avg}} \text{ or } \operatorname{rank}(c, C_{T^*}) \leq l(T), \\ U_{T^*}(c) & \text{otherwise.} \end{cases}$$

Combining the update schemes might be more powerful as one scheme can compensate for the shortcomings of the other and vice versa. However, this can also backfire if the combination results in the worst of both schemes. Only an experimental evaluation will show which update scheme works best in practice.

Fast-Pruning An improvement that we deploy during update schemes is to prune the node as early as possible, which we call *fast-pruning.* Let T be a node in an SE Tree and s_{best} be the best-found score. Let $s_{\text{rem}} = s_{\text{best}} - f(S_T)$ be the marginal gain needed to surpass the best score. Assume that we have applied any update scheme, reordered the candidates, and the Simple Upper Bound heuristic now prunes node T because the som of the $k - |S_T|$ greatest marginal gains is at most s_{rem}. Maybe it was already evident earlier that the Simple Upper Bound heuristic will prune the node. Identifying this point can be advantageous as we decrease the number of score function evaluations without changing the result. If we want to fast-prune after the update of the ith marginal gain, the following property must hold: The sum of the $k - |S_T|$ greatest lazy marginal gains (updated or not updated) is smaller than s_{rem}. How can we identify this time point? A trivial way to determine if the property is fulfilled after the ith update would be to sort all lazy marginal gains after each update and to sum the $k - |S_T|$ greatest. While this approach is quite simple, it will need to sort the marginal gains $\Omega(n)$ times, which adds to a time complexity of $\Omega(n^2 \log n)$. This is infeasible for large values of n if we deploy it at every node.

We deploy a method that utilizes an additional Min-Heap to store the $k - |S_T|$ greatest marginal gains. Algorithm 3 shows how to include fast pruning. The first $k - |S_T|$ loop cycles will be spent filling the heap. In each loop cycle afterward, we can determine if we can fast-prune the node. We can fast-prune if the sum of the heap, denoted by $\sum H$, is smaller than s_{rem} and if the smallest lazy marginal gain is greater than the lazy marginal gain of c at parent node T^*. We will verify the correctness of the algorithm in Lemma 4.15. Note that the functions $\sum H$ and $\min H$ require only a constant-time lookup. Each time we modify the heap, we keep the sum up to date, and access to the minimum element is guaranteed in a Min-Heap. If we cannot fast-prune in this cycle, we will determine how to process candidate c. If its lazy marginal gain is greater than the smallest lazy marginal gain in the heap, then we will overwrite the minimum of the heap by $U_T(c)$ and call `heapify` to make sure that heap is in a

Algorithm 3 Pseudocode to determine if node T can be fast-pruned. This pseudocode is contained in the Dynamic Candidate Ordering function, but only the parts that are needed to determine if the node can be fast-pruned are shown.

```
    Input: Node $T$, Integer $k$, Remaining score $s_{\text{rem}} \in \mathbb{R}$, Update scheme $U_T$
    Output: YES if we can fast-prune, NO otherwise.
 1: $H = \text{InitEmptyHeap}()$
 2: $i = 0$
 3: while $i < |C_{T^*}|$ do
 4:     $c = C_{T^*}[i]$                                          ▷ Get $i$th candidate
 5:     if $|H| = k - |S_T|$ and $\sum H \leq s_{\text{rem}}$ and $\min(H) \geq U_{T^*}(c)$ then
 6:         return YES
 7:     if $|H| < k - |S_T|$ then
 8:         $\text{addToHeap}(H, U_T(c))$                                 ▷ Fill the heap
 9:     else if $U_T(c) > \min(H)$ then
10:         $\text{setMinAndHeapify}(H, U_T(c))$          ▷ Save the greatest marginal gains
11:     $i = i + 1$
12: return NO
```

valid state. This procedure continues either until all candidates are processed and the algorithm returns `NO` or if we can fast-prune and the algorithm returns `YES`.

Note that a practical improvement can be made, which is not shown in the pseudocode for readability reasons. The first $k - |S_T|$ loop cycles are spent filling the heap with lazy marginal gains. We miss the opportunity to fast-prune in these first cycles as we have to ensure that enough elements are in the heap before we can make a decision correctly. However, if we fill the heap with the lazy marginal gains $U_{T^*}(c)$ of the parent and each time update the elements in the heap if we calculate $U_T(c)$, we can also determine for the first $k - |S_T|$ cycles if we can fast-prune.

Let us prove the correctness of the algorithm.

Lemma 4.15. Let T be a node in a SE Tree, T^* be the parent of T and let T^* be sorted by Dynamic Candidate Ordering. Let $k \in \mathbb{N}$, s_{best} the best-found score and U_T an update scheme. If Algorithm 3 returns `YES`, then the Simple Upper Bound heuristic will prune the node T.

Proof. Assume the algorithm returns `YES` and let c_j be the candidate when it returns. The following properties currently hold:

- The sum of the $k - |S_T|$ lazy marginal gains in the heap is at most s_{rem}.
- $U_{T^*}(c_j) \leq \min(H)$.

The first property tells us that the Simple Upper Bound heuristic can prune with the lazy marginal gains in the heap. The second property tells us that the lazy marginal gains in the heap are the $k - |S_T|$ largest lazy marginal gains. We need to show that no other lazy marginal gain will be placed in the heap.

The parent node T^* is sorted by Dynamic Candidate Ordering, so we know that $U_{T^*}(c_j) \geq U_{T^*}(c_{j+1}) \geq \ldots \geq U_{T^*}(c_m)$. The lazy marginal gain $U_{T^*}(c_j)$ is an upper bound for all lazy marginal gains of $c_{j+1}, \ldots, c_m$ at parent node T^*. It is also an upper bound for all lazy marginal gains of $c_{j+1}, \ldots, c_m$ at node T. That is because $U_T(c_i)$ either computes $U_{T^*}(c_i)$, the value stays the same, or it computes $\Delta(c_i \mid S_T)$, which has a smaller value. Therefore, if $U_{T^*}(c_j)$ will not be placed in the Min-Heap, then no value of $U_T(c_j), \ldots, U_T(c_m)$ will be placed in the heap. The heap stays unmodified. □

What is the running time of this approach? If we assume the worst case that we can never fast-prune, we must process each of the $\mathcal{O}(n)$ candidates. The check in Line 5 can be made in $\mathcal{O}(1)$ time, and the `addToHeap` operation in Line 8 can be made in $\mathcal{O}(\log k)$, just like the `heapify` operation in Line 12. We, therefore, have a worst-case running time of $\mathcal{O}(n \log k)$. Note that we do not factor in the time it takes to compute the update scheme $U_T(c)$, as this time is already allocated in the Dynamic Candidate Ordering. The worst-case only represents the extra operations we must execute to determine if we can fast prune. Also note that in the worst case, if we cannot abort early, we will have to sort the candidates via Dynamic Candidate Ordering. This will take an additional $\mathcal{O}(n \log n)$ time and dominate the running time. The fast pruning improvement hence does not degrade the running time if Dynamic Candidate Ordering is applied.

4.6.2 Just as Many as We Need

The second category of Lazy Evaluation methods is Search-Retaining Lazy Evaluation methods. These methods aim to maintain the search path the algorithm would have taken previously. As discussed, the Search-Altering Lazy Evaluation methods can lead to the exploration of nodes and subtrees that the search algorithm would have originally not explored. If the update schemes are poorly tuned, it can thus increase the running time. The now presented Search-Retaining Lazy Evaluation method will traverse the search space on the same path and additionally save score function evaluations. However, it requires much effort to organize.

The idea of the *Just as Many as We Need* method is to only update the candidates until the first ℓ candidates in the sorted candidate set are accurate. Since Partitioning heuristics use at most $k - |S_T|$ candidates from partition P_2, we know that at most the first $|P_1| + k - |S_T| = \ell$ candidates will be used by the heuristic. Therefore, if we manage that the first ℓ marginal gains are accurate in the sorted candidate set, then the heuristics will always return the same result as if we did not use any Lazy Evaluation method.

The idea can be implemented as follows. Let T be a node with parent node T^*. The candidates of the parent are sorted by Dynamic Candidate Ordering. This makes it straightforward to build a Max-Heap out of them, as a sorted list is automatically a Max-Heap in a valid state. Now, we will continually update the candidate at the top of the Max-Heap and each time call `heapify`, so that the heap is in a valid state again. If the element at the top of the heap is accurate, then we can remove it and store it in the sorted candidate list. This is correct because the element at the top of the Max-Heap has the greatest marginal gain of all elements, otherwise it would not be on the top. Additionally, all marginal gains not updated in the Max-Heap will only decrease their score if we update them because of submodularity. They will thus not be placed at the top of the heap. We can repeat this process until the sorted candidate list holds ℓ candidates. The additional fast-pruning technique can also be incorporated.

Although the method looks straightforward on paper, it did not perform well in practice. Assume we just applied the method, and the heuristic could not to prune the node. Because we demand that the parent T^* is sorted by Dynamic Candidate Ordering, the child of T will demand that T is sorted by Dynamic Candidate Ordering. We, therefore, have to sort the remaining elements in the heap. We will explore the first candidate and, at some point, return and remove the first candidate from the candidate set. The heuristic will again use the first ℓ candidates, but since we just removed one candidate, we do not know if the last of the ℓ candidates is accurate. In the best case, the candidate is accurate, and we can continue, but in the worst case, it is not, and we again have to deploy the heap. After that, we again have to sort the remaining candidates. The method potentially has to sort the candidates very often, greatly limiting its use. Remember that there are potentially $\Theta(n^k)$ nodes in the SE Tree, so a simple inefficiency can substantially increase the running time. Sadly, the method performed worse than not using the method at all during development. The method will, therefore, not be included in our experiments. However, we believe that there is a way to implement this approach efficiently.

5 Experiments

To measure the effectiveness of our developed algorithm, we perform experiments on the three problems introduced at the beginning of this work. These are:

- Group Closeness Centrality on undirected graphs,
- Partial Dominating Set on undirected graphs,
- k-Medoid Clustering with an euclidian distance on a set of datapoints.

Graphs We use 13 graphs from the Network Repository [62]. The graphs range from 370 to 50515 vertices. Table 6 shows the graphs with their number of vertices, number of edges, diameter, and density. The diameter may be an interesting property for Group Closeness Centrality as it denotes the maximum distance two vertices can have. Intuitively, it may be easier to find a set of vertices that minimizes the distances for graphs with a smaller diameter, as the distances are already relatively small. On the other hand, the density could be an interesting property for Partial Dominating Set, as it measure how well-connected the graph is. Note that we only consider the largest connected component of the graph as Group Closeness Centrality is not defined on graphs with multiple connected components. The graphs used are (mostly) a subset of those used by Staus et al. [69], who developed exact algorithms for Group Closeness Centrality.

Clustering Datasets For k-Medoid Clustering, we consider six synthetic datasets[7] proposed by Fränti and Sieranoja [25] and four proposed by Rezaei and Fränti [61]. The datasets range from 1000 to 7500 data points and have different numbers of clusters and different cluster structures. Preliminary experiments have shown that exactly solving k-Medoid Clustering is harder than the other problems. Therefore, we also introduce a sampled version (denoted by the ending "-sampled") of the datasets, where 30% of points were randomly selected. Table 7 shows the original ten datasets used.

Experiment Setup Our algorithm is implemented in C++[8] using the C++ 20 Standard. We use `-O3` as a compilation flag to generate the most efficient machine code. To parse command line arguments we use the Boost library [65] version 1.78.0. The code is compiled using the GCC compiler version 10.2.0. As the build tool, we use CMake version 3.22.2.

We conduct the experiments on an Intel® Xeon® Gold 6140 Processor with 2.0 GHz. The system has 192GB RAM and runs CentOS Linux release 7.6[9]. Note that although the processor has 18 cores and 36 threads, we do not use them but conduct our

[7]The data was retrieved from `https://cs.uef.fi/sipu/datasets/`, accessed on 2023-11-02.
[8]The code can be found at github.com/HenningAthor/SSM
[9]The resources are provided by Friedrich Schiller University Jena, supported in part by DFG grants INST 275/334-1 FUGG and INST 275/363-1 FUGG

Name of the Graph	# Vertices	# Edges	diameter	density
ca-netscience	379	913	17	0.013
soc-wiki-Vote	889	2 914	13	0.007
bio-yeast	1 458	1 948	19	0.002
econ-orani678	2 529	86 768	5	0.027
soc-advogato	5 054	39 374	9	0.003
bio-dmela	7 393	25 569	11	0.001
ia-escorts-dynamic	10 106	39 016	10	0.001
soc-anybeat	12 645	49 132	10	0.001
ca-AstroPh	17 903	196 972	14	0.001
fb-pages-media	27 917	205 964	15	0.001
soc-gemsec-RO	41 773	125 826	19	< 0.001
soc-gemsec-HU	47 538	222 887	14	< 0.001
fb-pages-artist	50 515	819 090	11	0.001

Table 6: The 13 graphs used in the experiments for Group Closeness Centrality and Partial Dominating Set.

experiments serially: Our algorithm is already memory-bound on a single core, meaning the processor fetches more data from RAM than the memory bus can support. On a private system with an Intel® Core™ i7-8700K with 3.7 GHz and 64 GB RAM, the VTune™ Profiler [22] showed that about 55% of all micro-ops were memory-bound when using one core. Using more cores will put even more strain on the memory bus, resulting in increased running times. Preliminary experiments confirmed this as it took the algorithm 4 to 6 times longer to solve a problem instance when we tested multiple instances in parallel. Therefore, a serial execution is the only means to get valid results.

We limit the experiments by only allowing $k \in [1, 20]$ and setting the time limit to 15 minutes (900 seconds). In other words, the algorithm has 15 minutes to solve a problem instance for one value of k. Should the algorithm solve the instance, it starts with $k + 1$, and the time limit is reset to 15 minutes.

Note that preprocessing is not included in the time measurements. The vertex domination number of the Partial Dominating Set problem is implemented via an adjacency matrix, while the group farness and the clustering cost are implemented via a distance matrix. The calculation of these matrices is done rather naively. The additional time required to compute the matrices first is not measured since we only want to compare the running time of the search algorithm. For the largest graph, fb-pages-artist, with about 50 000 vertices, it took about 5 minutes to calculate the distance matrix.

Result Interpretability It may might not look like much initially when an algorithm only improves by one value of k. Keep in mind that solving the problem for $k+1$ instead of k brings a tremendous amount of additional work. The number of potential solutions

Data Name	# Entries	Dimension	# Clusters	Description
skewed	1 000	2	6	stretched shape
asymmetric	1 000	2	5	small overlap, different shapes
overlap	1 000	2	6	strong overlap
dim032	1 024	32	16	no overlap
a1	3 000	2	20	no overlap
s1	5 000	2	15	almost no overlap
s2	5 000	2	15	small degree of overlap
a2	5 250	2	35	no overlap
unbalance2	6 500	2	8	no overlap, different sizes
a3	7 500	2	50	no overlap

Table 7: The ten cluster datasets used in the k-Medoid experiments. Note that the given description is only an ad-hoc description. For a more detailed description we refer to Fränti and Sieranoja [25] and Rezaei and Fränti [61].

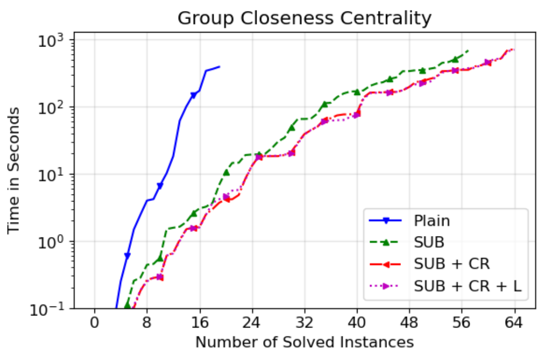
and, therefore, leaf nodes that can be visited grows in $\mathcal{O}(n)$ Roughly speaking, solving the problem for $k+1$ is thus equivalent to solving the problem for k a total number of n times. The heuristics, of course, try to mitigate this growth.

Experiment Structure First, we will compare the Plain Search algorithm and variants of the Basic Search algorithm in Section 5.1 to understand how difficult it is to solve instances in practice. We will also compare our algorithm for Group Closeness Centrality with three exact algorithms implemented and developed by Staus et al. [69]. After that, we will evaluate Lazy Evaluation with different update schemes in Section 5.2. The Upper Bound 2D heuristic, the Partial Brute-Force heuristic, and the Divide-and-Conquer heuristic are evaluated in Section 5.3, Section 5.4, and Section 5.5, respectively. Section 5.7 compares our algorithm to the state-of-the-art algorithm developed by Uematsu et al. [73], which also solves Cardinality Constrained Maximization for arbitrary score functions. Section 5.8 gives concluding remarks on the experiments.

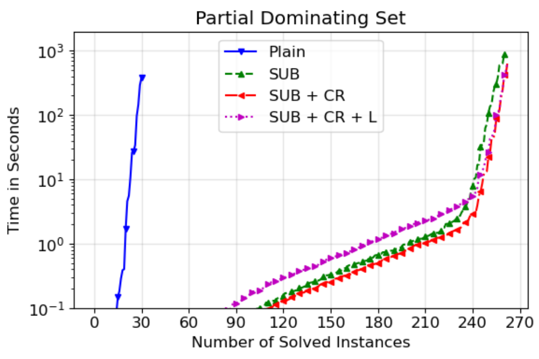
5.1 Basic Search Algorithm

Let us first evaluate the performance of the Basic Search algorithm (Algorithm 2). Compared to the Plain Search algorithm (Algorithm 1), it deploys Dynamic Candidate Ordering, the Simple Upper Bound heuristic, and Candidate Reduction. We will compare the following four algorithms:

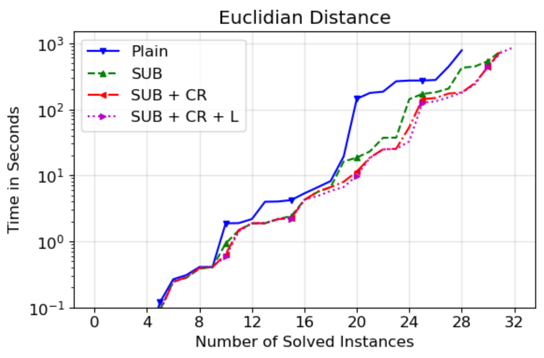
- `Plain`: Plain Search algorithm.
- `SUB`: Basic Search with Dynamic Candidate Ordering and the SUB heuristic.
- `SUB + CR`: The `SUB` algorithm and Candidate Reduction.



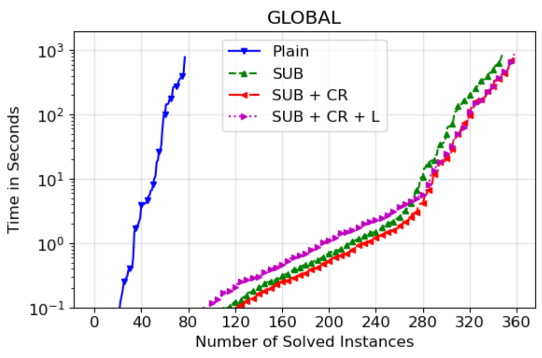
(a) Group Closeness Centrality



(b) Partial Dominating Set



(c) k-Medoid



(d) Global

Figure 10: The number of instances solved by the `Plain`, `SUB`, `SUB + CR`, and `SUB + CR + L` algorithm. Point (x, y) on a line denotes that the algorithm solved x problem instances each in most y seconds. The solved instances of one algorithm must not necessarily be the solved instances of another algorithm, but there is a large overlap. Solved instances for $k = 1$ are not included.

- `SUB + CR + L`: The `SUB + CR` algorithm and greedy local search.

We mentioned in Section 3 a more sophisticated local search greedy starting solution, which we denote with the letter `L`. First, the approach calculates the standard greedy solution and then improves it by iteratively swapping elements which lead to a better solution. This is done until either 10% of the time limit is reached (in this case, 90 seconds) or until no swap results in a better solution. We will discuss the experiments for the three problems, one after the other. Figure 10 shows the result for the three problems and one plot for all results globally accumulated.

Group Closeness Centrality The `SUB` algorithm gives a tremendous speedup over the `Plain` algorithm, as Figure 10a shows. Additionally, `SUB + CR` can further reduce the running time. Note that the time difference does not seem large in the plot because of the log scale, but `SUB + CR` outperforms `SUB` by almost a factor of two. Bootstrapping the search algorithm with the more advanced greedy approach does not improve the running time. Table 8 shows the largest solved k and the required time to solve each instance. `SUB + CR` consistently solves one largest k than `SUB`, except for soc-anybeat and the four largest instances, which have more than 27 000 vertices.

Partial Dominating Set The difference between `Plain` and the three other approaches is especially large for PARTIAL DOMINATING SET (see Figure 10b). While `Plain` only solves about 30 instances out of the 260, the other three algorithms solve about 255 instances. For this problem, `SUB` and `SUB + CR` do not differ too much in the number of solved instances, but `SUB + CR` still solves the instances faster. The `SUB + CR + L` algorithm, again, does not improve the running time. Figure 10b shows that `SUB + CR + L` needs a bit more time for the first 270 instances, which is probably the time spent in the greedy local search. Table 9 shows that `SUB` and `SUB + CR` almost solve all instances, except for the graph econ-orani678 and fb-pages-artist. For econ-orani678, the `SUB + CR` algorithm saves around 25% of running time, and for fb-pages-artist, around 33% of running time compared to `SUB`. Note that econ-orani678 has by far the greatest density compared to the other graphs. This could explain why the instance is harder to solve, as dense graphs have more vertices with large neighborhoods. A more comprehensive evaluation of graphs with a wider diversity of densities could give more insight into which instances are hard and which are easy to solve. The other not fully solved instance is fb-pages-artist, which has the largest number of vertices in these experiments. While this could be an explanation, note that soc-gemsec-HU only has about 3 000 vertices less and is easily solvable. The instance could simply be harder to solve than the other instances for our algorithm.

***k*-Medoid Clustering** The experiment shows the same results as for the first two problems but not as strong, as shown in Figure 10c. `SUB`, `SUB + CR`, and `SUB + CR + L` reduce the running time over `Plain`, but all algorithms do not solve many instances. At most, about 50 out of 400 instances were solved. The difference between `SUB` and `SUB + CR` is not as large in these experiments. The only three instances that `SUB + CR` solved faster than `SUB` are skewed-sampled, overlap-sampled, and dim032-sampled, as Table 10 shows.

`SUB` and `SUB + CR` did not perform much better than `Plain` when comparing the largest solved k. Only for four instances did `SUB` and `SUB + CR` solve the instance with a larger k. The other instances (which were solved for $k \geq 4$) were solved substantially faster by the three Basic Search algorithms compared to `Plain`. The `SUB + CR + L` algorithm again shows no substantial improvement, except for dim032-sampled, as it solves it for $k = 7$. Why dim032-sampled was solved faster can only be speculated. The instance almost timed out, so maybe the better starting solution was able to decrease the running time by the crucial seconds to solve it.

Dataset Name	`Plain`		`SUB`		`SUB + CR`		`SUB + CR + L`	
	k	seconds	k	seconds	k	seconds	k	seconds
ca-netscience	5	391	12	575	**13**	700	**13**	655
soc-wiki-Vote	4	147	10	462	**11**	522	**11**	510
bio-yeast	3	2	9	508	**10**	712	**10**	708
econ-orani678	3	10	10	448	**11**	483	**11**	463
soc-advogato	3	100	7	271	**8**	338	**8**	340
bio-dmela	3	363	5	231	**6**	256	**6**	214
ia-escorts-dynamic	2	172	5	350	**6**	412	**6**	518
soc-anybeat	2	341	**5**	687	**5**	454	**5**	416
ca-AstroPh	1	< 1	1	< 1	**3**	879	2	876
fb-pages-media	**1**	< 1	**1**	< 1	**1**	< 1	**1**	< 1
soc-gemsec-RO	**1**	1	**1**	1	**1**	1	**1**	1
soc-gemsec-HU	**2**	4	**2**	3	**2**	3	**2**	4
fb-pages-artist	**2**	5	**2**	3	**2**	3	**2**	5

Table 8: Table holding the results for Group Closeness Centrality. It shows the largest k and the time needed to solve this k.

Dataset Name	`Plain`		`SUB`		`SUB + CR`		`SUB + CR + L`	
	k	seconds	k	seconds	k	seconds	k	seconds
ca-netscience	5	34	**20**	< 1	**20**	< 1	**20**	< 1
soc-wiki-Vote	4	5	**20**	< 1	**20**	< 1	**20**	< 1
bio-yeast	4	26	**20**	< 1	**20**	< 1	**20**	< 1
econ-orani678	4	293	**17**	876	**17**	632	**17**	633
soc-advogato	3	1	**20**	< 1	**20**	< 1	**20**	< 1
bio-dmela	3	4	**20**	1	**20**	< 1	**20**	< 1
ia-escorts-dynamic	3	10	**20**	< 1	**20**	< 1	**20**	< 1
soc-anybeat	3	27	**20**	1	**20**	1	**20**	3
ca-AstroPh	3	95	**20**	626	**20**	101	**20**	103
fb-pages-media	3	382	**20**	4	**20**	2	**20**	5
soc-gemsec-RO	2	< 1	**20**	1	**20**	1	**20**	4
soc-gemsec-HU	2	< 1	**20**	2	**20**	1	**20**	3
fb-pages-artist	2	< 1	17	663	**19**	421	**19**	475

Table 9: Table holding the results for Partial Dominating Set. It shows the largest k and the time needed to solve this k.

Dataset Name	`Plain`		`SUB`		`SUB + CR`		`SUB + CR + L`	
	k	seconds	k	seconds	k	seconds	k	seconds
skewed	3	34	3	34	3	34	3	33
asymmetric	**4**	613	**4**	252	**4**	258	**4**	254
overlap	3	28	**4**	683	**4**	649	**4**	626
dim032	**4**	468	**4**	194	**4**	163	**4**	162
a1	**2**	16	**2**	16	**2**	16	**2**	16
s1	**2**	76	**2**	77	**2**	75	**2**	77
s2	**2**	80	**2**	76	**2**	77	**2**	77
a2	**2**	88	**2**	86	**2**	92	**2**	87
unbalance2	**2**	169	**2**	170	**2**	168	**2**	163
a3	**2**	262	**2**	259	**2**	252	**2**	262
skewed-sampled	5	265	**6**	536	**6**	248	**6**	238
asymmetric-sampled	**5**	272	**5**	36	**5**	24	**5**	24
overlap-sampled	5	272	**6**	425	**6**	148	**6**	124
dim032-sampled	5	277	6	205	6	53	**7**	876
a1-sampled	**3**	19	**3**	18	**3**	18	**3**	18
s1-sampled	**3**	173	**3**	174	**3**	171	**3**	151
s2-sampled	**3**	146	**3**	141	**3**	142	**3**	132
a2-sampled	**3**	185	**3**	180	**3**	179	**3**	177
unbalance2-sampled	**3**	443	**3**	443	**3**	438	**3**	443
a3-sampled	**3**	781	**3**	782	**3**	768	**3**	720

Table 10: Table holding the results for k-Medoid Clustering. It shows the largest k and the time needed to solve this k.

Conclusion from the First Experiments There are three key points that we can observe for the algorithms. (a) The `Plain` algorithm performs the worst as expected. (b) The additional usage of Candidate Reduction has a great effect over only using SUB. This is because the method is very cheap, as removing one candidate is possible in $\mathcal{O}(1)$ time, so basically for free. (c) Bootstrapping the algorithm with the better greedy solution has, in most cases, no effect. We hoped that the better solution would enable the algorithm to prune more nodes and save time. This intuition, however, was not confirmed. Either the greedy solution does not improve by much, or the Basic Search algorithm quickly finds an equally good solution, so pruning is possible in both cases. It would be interesting to see if bootstrapping the algorithm with the optimal solution will greatly reduce the running time. If yes, other greedy heuristics could substantially reduce the running time by providing better initial solutions. If not, then the algorithm spends most of its time to validate that the solution is optimal, and a more sophisticated greedy approach would not be necessary.

We can also compare the problems. Partial Dominating Set is by far the easiest problem to solve: `SUB`, `SUB + CR`, and `SUB + CR + L` solve 11 out of the 14 instances with $k = 20$ in under 6 seconds. The other three instances also get solved for large k.

Group Closeness Centrality is harder to solve. No algorithm manages to solve any instance for $k = 20$. We can see a clear trend that the instances get harder to solve the larger they are, which is reasonable. We do not notice any correlation between diameter and instance difficulty.

Exactly solving k-Medoid Clustering is hard, even for smaller n. The largest k that any algorithm solves is 7 for the sampled version of dim032, which has 307 data points. One might think this problem is hard to solve if k is not equal to the number of clusters. Then, the algorithm has to look for data points that do not represent the cluster centers, which makes the problem more difficult. However, separate experiments showed that the instances with k as the number of clusters will still not be solved in 15 minutes. Another explanation could be that many data points are very similar in their score, making eliminating them very difficult. Data points that lie close to each other have very similar distances to all other data points, so if the algorithm considers one, the other ones will also be considered. If this happens often, which is likely since these are clusters, the algorithm can rarely prune nodes and eliminate candidates. Another drawback, in this case, is that the search algorithm explores elements with the greatest marginal gain. From the root, the search algorithm will explore all data points in the center of the dataset and only slowly move outwards. If there is no cluster at the center of all datapoints, then the search algorithm will not find the best solution for a long time and cannot prune often.

Comparison to SOTA Group Closeness Centrality Staus et al. [69] developed branch-and-bound algorithms and integer linear programming methods to solve Group Closeness Centrality exactly. We compare the `SUB + CR` algorithm to three of their developed algorithms, namely `CI`, `DVind`, and `ILPind`. Their `CI` algorithm is a branch-and-bound algorithm that deploys the same improvements as our `SUB + CR` algorithm. It also uses Dynamic Candidate Ordering, SUB, and Candidate Reduction. They recommend the `DVind` algorithm as their fastest branch-and-bound algorithm. It additionally uses domain-specific knowledge to identify vertex pairs u and v with $N[v] \subseteq N[u]$. Then, v can be removed from the graph without altering the result. Additionally, they deploy an approach that starts with a heavily constrained solution space and iteratively relaxes the constraint only if truly needed. Their `ILPind` algorithm is their fastest integer linear programming (ILP) algorithm. It also deploys the iterative approach.

Table 11 shows the results of our algorithm and the three mentioned algorithms. Our `SUB + CR` algorithm is faster than the `CI` algorithm. We solved the smallest four instances with larger ks, and the next five instances were solved faster. The last four instances were barely solved by `SUB + CR` and their `CI` algorithm. Their `DVind` algorithm surpasses our algorithm in almost all instances. The only two outliers are ia-escorts-dynamic and ca-AstroPh, for which `DVind` only solved $k = 1$.

The `ILPind` algorithm is either very powerful, as it can solve large ks very fast, or

Dataset Name	`SUB + CR`		`CI`		`DVind`		`ILPind`	
	k	seconds	k	seconds	k	seconds	k	seconds
ca-netscience	13	700	10	619	**20**	< 1	17	1
soc-wiki-Vote	11	522	10	648	**20**	319	**20**	< 1.0
bio-yeast	10	712	9	484	**20**	551	**20**	3
econ-orani678	11	483	9	709	**12**	603	**20**	1
soc-advogato	8	338	8	699	**14**	823	5	34
bio-dmela	**6**	256	**6**	417	**6**	480	N/A	N/A
ia-escorts-dynamic	**6**	412	**6**	728	1	9	N/A	N/A
soc-anybeat	5	454	5	626	**20**	374	2	99
ca-AstroPh	**3**	879	1	< 1	1	33	N/A	N/A
fb-pages-media	**1**	< 1	**1**	< 1	**1**	83	N/A	N/A
soc-gemsec-RO	**1**	1	**1**	1	**1**	252	N/A	N/A
soc-gemsec-HU	**2**	3	1	2	1	409	N/A	N/A
fb-pages-artist	**2**	3	N/A	N/A	1	347	N/A	N/A

Table 11: Table holding the running times for our SUB + CR algorithm and three algorithms of Staus et al. [69] for Group Closeness Centrality. The algorithm `CI` is comparable to our `SUB + CR`. `DVind` enhances `CI` by including domain-specific knowledge and an iterative approach, which our algorithm does not support. The algorithm `ILPind` uses integer linear programming (ILP) to solve the instances. The table shows the largest k and the time needed to solve this k.

it does not work at all. However, these results do not resemble those obtained by Staus et al. [69]. In their work, `ILPind` solves all instances except ca-AstroPh with either $k = 19$ or $k = 20$ (note that they did not evaluate the four largest graphs present here). In their experiments, the time limit was set to 60 minutes, which could explain why the algorithms solved more instances. However, ca-netscience only has 379 vertices, so it seems remarkable that `ILPind` can solve it for $k = 17$ in 1 second but not for $k = 18$ in 900 seconds. The same holds for soc-advogato and soc-anybeat. The graphs bio-dmela and ia-escorts-dynamic did not get solved at all by `ILPind`, although they have fewer vertices than soc-anybeat and are, therefore, intuitively easier to solve. Note that their algorithm is implemented in Kotlin, and they were running Java OpenJDK 17.04. Our system only supports Java 1.8, and we had to make small changes to their code. This could also be a source of errors, but would not explain why our `CI` results are consistent with their results.

5.2 Lazy Evaluation

In this section, we want to evaluate the Lazy Evaluation approach. It is used to control which candidates at node T should have their marginal gain updated and which should be approximated by T's parent. We will use the `SUB + CR` algorithm as a basis, as the additional local search at the beginning does not improve the running time (except for one instance). We rename `SUB + CR` to `B` as the algorithm represents the Basic Search algorithm (Algorithm 2). We evaluate the following six Lazy Evaluation configurations:

- `B + FP`: Default Update Scheme with Fast Pruning.
- `B + Avg`: Average Update Scheme.
- `B +` k': Rank Update Scheme with $l(T) = k - |S_T|$.
- `B +` $0.5|C|$: Rank Update Scheme with $l(T) = 0.5|C_T|$.
- `B + Avg or` $3k$: Or Update Scheme with $l(T) = 3k$.
- `B + Avg or` $0.1n$: Or Update Scheme with $l(T) = 0.1n$.

The first configuration only includes Fast Pruning. The technique allows us to identify earlier if a node will be pruned, saving score function evaluations. The other five algorithms also use Fast Pruning, although this is not added to their name. The `B + Avg` algorithm uses the average update scheme, while `B +` k' and `B +` $0.5|C|$ use the rank update scheme, as defined in Section 4.6.1. Recall that the average update scheme only updates a candidate if its marginal gain at parent node T^* exceeds the average required score per candidate. The rank update scheme only updates candidates with a better rank than $l(T)$ in the sorted candidate list of parent T^*. The two last algorithms combine the average and rank update schemes and will update a marginal gain if at least one of both update schemes would update the candidate.

Figure 11 shows the number of solved instances for each of the three problems. The blue line denotes the `B` algorithm. The configurations that are generally slower will be to the left and above the blue line, while faster configurations will be to the right and below the blue line.

Fast Pruning The first observation is that `B + FP` is faster than the standard `B` algorithm. In Section 4.6.1, we described that Fast Pruning does not worsen the worst-case upper bound and is generally very cheap to implement. The saved time can also be seen in Table 12. `B + FP` either solves the instances for a larger k or in a faster time if the same k is solved.

Rank Update Scheme The `B + k'` and `B +` $0.5|C|$ configurations perform the worst out of all the presented configurations. The rank update scheme updates only the first $k' = k - |S_T|$ candidates. The further we go down in the tree, the fewer candidates will be updated, resulting in a less precise branching strategy. The negative consequence has, as it seems, a much greater effect than the saved time by not updating most of the candidates. The `B +` $0.5|C|$ configuration performs better than `B + k'` but is still worse than not using Lazy Evaluation. It updates half of the candidates at each node, much more than `B + k'`. However, the tradeoff between more precise branching and saved score function evaluations is not beneficial compared to always evaluating the score function. While we think it is sufficient to update only half of the candidates close to the root, it could be problematic to only update half of them close to the leaves: Nodes close to the leaves only have a limited number of candidates, so updating them is not costly, and the more precise branching could have a large effect.

Average Update Scheme The `B + Avg` configuration uses the average update scheme. This has a positive effect on the running time, as we can see in Figure 11 and Table 12 (except for k-Medoid Clustering). The configuration outperforms `B` and `B + FP` consistently by either solving the instances for larger k or solving them faster.

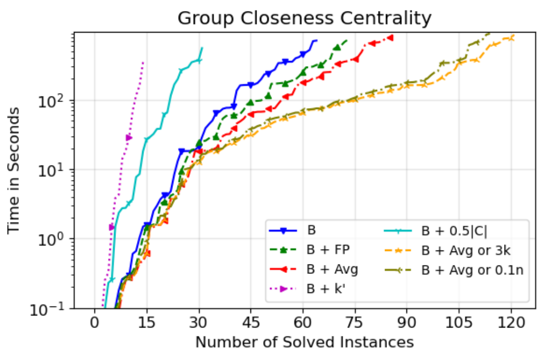
We, therefore, extended the average update scheme by additionally using the rank update scheme with either $3k$ or $0.1n$. These configurations are indicated by the yellow and olive lines, respectively. The configuration `B + Avg or` $3k$ generally outperforms `B + Avg or` $0.1n$. Note that, in these experiments, $3k$ is at most 60 for all instances, while $0.1n$ varies for each instance. If the instance has more than 10 000 vertices, then at least 100 candidates are updated in each node, while `B + Avg or` $3k$ may not update all these candidates. The gained accuracy from `B + Avg or` $0.1n$ may not impact the branching strategy much, so saving the score function evaluations has more impact.

If we observe the running times achieved in Table 12, we will notice that `B + Avg` mostly performs marginally better than `B + Avg or` $3k$. Only two outliers exist where the combined update scheme is better than the average update scheme: The graphs soc-gemsec-HU and fp-pages-artist are solved for $k = 20$ by the combined update scheme, while the average update scheme only manages $k = 2$ for each instance. The reason for this is currently not known and this behavior is not consistent with the other instances. It could be that solving these instances for a small k is very hard but gets easier once we have crossed a certain threshold for k: Suppose `B + Avg` barely fails to solve the instances for $k = 3$. In that case, it will be terminated, not even attempting any larger k, as we assume that a larger k gets harder to solve. This assumption may not be true every time. We can intuitively construct examples where solving a problem with $k + 1$ seems easier than k. Increasing the time limit or evaluating all possible $k \in [1, 20]$ could give more insight.

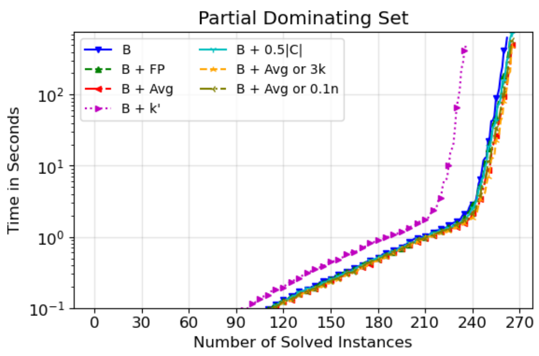
Conclusion for Lazy Evaluation The Lazy Evaluation approach can generally improve the running time of our algorithm. The average update scheme performed best, either combined with an additional rank update scheme or by itself. The rank update scheme did not prove useful, as it only worsened the running time. However, we only evaluated a few configurations, so we do not want to give a final statement on their usability. It very well can be that there is a globally better configuration and that there are better configurations for each separate problem.

5.3 Upper Bound 2D

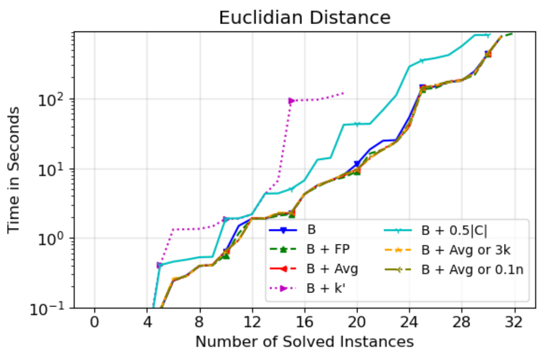
Let us now evaluate the first newly discussed heuristics. Since `B + Avg or` $3k$ performed as the best Lazy Evaluation configuration, we chose it as the basis for all algorithms. We will call the algorithm `LE` for Lazy Evaluation algorithm for better readability. We will test the following configurations:



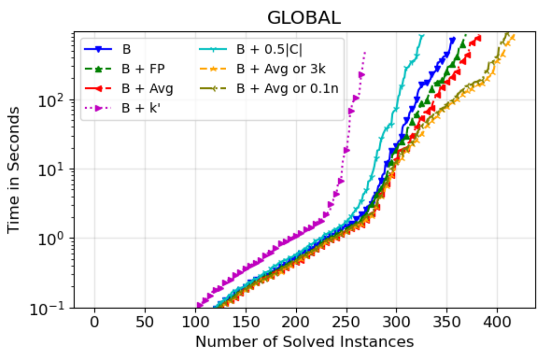
(a) Group Closeness Centrality



(b) Partial Dominating Set



(c) k-Medoid



(d) Global

Figure 11: The number of instances solved by the `B` algorithm and the different Lazy Evaluation configurations. Point (x, y) on a line denotes that the algorithm solved x problem instances each in most y seconds. The solved instances of one algorithm must not necessarily be the solved instances of another algorithm, but there is a large overlap. Solved instances for $k = 1$ are not included.

- `LE + M Sqrt(n)`: Maximum Weight Matching with $\ell = \sqrt{n}$
- `LE + M k'`: Maximum Weight Matching with $\ell = k - |S_T|$
- `LE + D 15`: Dynamic with $\ell = 15$
- `LE + G 10`: Greedy with $\ell = 10$

We exclusively use the UB2D Partitioning heuristics, with $|P_1| = \ell$, and SUB on P_2. Note that in the appendix in Tables 18 and 19, further tested configurations are shown.

Results Figure 12 and Table 13 show how each configuration performed. The configurations do not differ much from each other, and not one of them generally improves the running time of the `LE` algorithm. The only configuration that solves an instance for a larger k is `LE + G 10` for the graph soc-gemsec-RO for Group Closeness Centrality. It solves the instance for $k = 5$, while all other algorithms only manage to

	Dataset Name	`B`		`B + FP`		`B + Avg`		`B + `k'		B + $0.5\|C\|$		`B + Avg or `$3k$		`B + Avg or `$0.1n$	
		k	seconds	k	seconds	k	seconds	k	seconds	k	seconds	k	seconds	k	seconds
Group Close. Centr.	ca-netscience	13	700	13	389	**14**	783	4	28	9	555	**14**	867	**14**	869
	soc-wiki-Vote	11	522	12	539	**13**	673	3	14	8	369	**13**	761	**13**	899
	bio-yeast	10	712	10	250	**11**	417	3	115	7	360	**11**	460	**11**	748
	econ-orani678	11	483	13	597	**16**	860	2	1	5	293	15	529	14	637
	soc-advogato	8	338	11	582	**13**	704	2	18	2	18	**13**	767	11	402
	bio-dmela	6	256	7	291	**9**	648	2	68	2	61	**9**	637	8	614
	ia-escorts-dynamic	6	412	7	732	**8**	666	2	160	2	198	**8**	628	7	689
	soc-anybeat	5	454	6	675	**8**	765	2	350	2	347	**8**	779	6	677
	ca-AstroPh	**1**	< 1	**1**	< 1	**1**	< 1	**1**	< 1	**1**	< 1	**1**	< 1	**1**	< 1
	fb-pages-media	**1**	< 1	**1**	< 1	**1**	< 1	**1**	< 1	**1**	< 1	**1**	< 1	**1**	< 1
	soc-gemsec-RO	**1**	1	**1**	1	**1**	1	**1**	1	**1**	1	**1**	1	**1**	1
	soc-gemsec-HU	2	2	2	3	2	6	2	4	2	2	**20**	161	**20**	161
	fb-pages-artist	2	4	2	4	2	3	2	3	2	3	**20**	190	**20**	190

Table 12: Table holding the running times for the Basic Search algorithm `B` and the Lazy Evaluation configurations for Group Closeness Centrality. We refer to Table 17 in the appendix for the table that also holds the data for Partial Dominating Set and k-Medoid Clustering. Numbers marked in bold denote the largest k for which the instance could be solved. The reported time is the seconds it takes for the specific instance with the specific k.

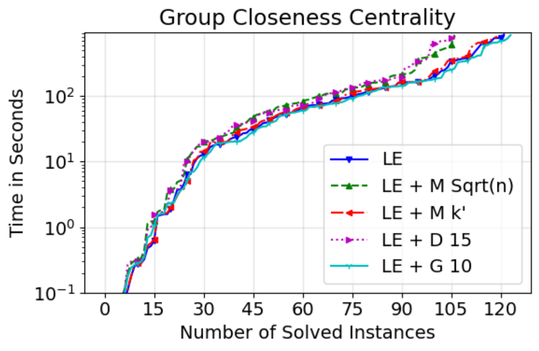
solve it for $k = 1$.

The slowest configuration overall is `LE + M Sqrt(n)`, as Figure 12d shows. For Group Closeness Centrality, it solves much fewer instances, and for Partial Dominating Set and k-Medoid Clustering, the algorithm solves the same k, but is generally slower. While `LE + D 15` performs better than `LE + M Sqrt(n)` for Partial Dominating Set, it is slower for Group Closeness Centrality. The sharper upper bound returned by the dynamic heuristic does not have the desired effect. The greedy configuration performs similarly to `LE + M k'`, sometimes faster and sometimes slower. When we compare the greedy and dynamic configuration, we notice that the faster speed is preferrable over the better sharpness.

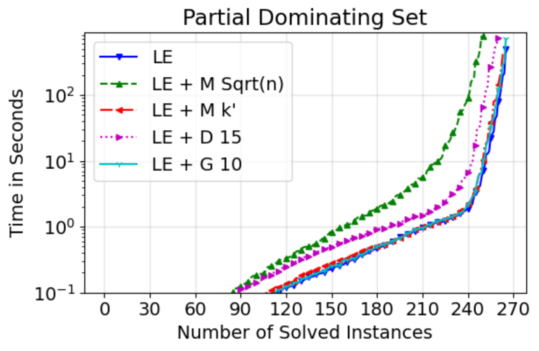
Conclusion for UB2D Sadly, the Upper Bound 2D heuristic does not prove useful in practice. The only case where a heuristic solved a larger instance was for Group Closeness Centrality and the graph soc-gemsec-RO, where `LE + G 10` manages to solve the instance for $k = 5$. However, this is the only outlier, and all other instances for all other problems were solved slower and did not reach the same value of k as `LE`.

	Dataset Name		`LE`		`LE + M Sqrt(n)`		`LE + M k'`		`LE + D 15`		`LE + G 10`
		k	seconds	k	seconds	k	seconds	k	seconds	k	seconds
Group Close. Centr.	ca-netscience	**14**	867	12	857	13	707	9	624	13	694
	soc-wiki-Vote	**13**	761	11	505	12	385	10	692	12	389
	bio-yeast	**11**	460	10	544	**11**	553	9	189	**11**	598
	econ-orani678	**15**	529	12	406	**15**	713	12	691	**15**	652
	soc-advogato	**13**	767	11	499	**13**	804	12	880	**13**	849
	bio-dmela	**9**	637	8	603	**9**	682	**9**	803	**9**	696
	ia-escorts-dynamic	**8**	628	7	529	**8**	655	**8**	717	**8**	655
	soc-anybeat	**8**	779	5	441	**8**	770	**8**	779	**8**	753
	ca-AstroPh	**1**	< 1	**1**	< 1	**1**	< 1	**1**	< 1	**1**	< 1
	fb-pages-media	**1**	< 1	**1**	< 1	**1**	< 1	**1**	< 1	**1**	< 1
	soc-gemsec-RO	1	1	1	1	1	1	1	1	**5**	16
	soc-gemsec-HU	**20**	161	**20**	162	**20**	164	**20**	160	**20**	162
	fb-pages-artist	**20**	190	**20**	190	**20**	188	**20**	193	**20**	189
Part. Dom. Set	ca-netscience	**20**	< 1	**20**	< 1	**20**	< 1	20	1	**20**	< 1
	soc-wiki-Vote	**20**	< 1	**20**	1	**20**	< 1	**20**	4	**20**	< 1
	bio-yeast	**20**	< 1	**20**	< 1	**20**	< 1	**20**	< 1	**20**	< 1
	econ-orani678	**19**	491	16	839	18	433	15	722	**19**	701
	soc-advogato	**20**	< 1	**20**	8	**20**	< 1	**20**	2	**20**	< 1
	bio-dmela	**20**	< 1	**20**	17	**20**	< 1	20	6	**20**	< 1
	ia-escorts-dynamic	**20**	< 1	**20**	10	**20**	< 1	**20**	2	**20**	< 1
	soc-anybeat	**20**	1	**20**	2	**20**	1	**20**	1	**20**	1
	ca-AstroPh	**20**	27	16	773	**20**	87	**20**	531	**20**	44
	fb-pages-media	**20**	2	**20**	107	**20**	2	**20**	6	**20**	2
	soc-gemsec-RO	**20**	1	**20**	5	**20**	1	**20**	1	**20**	1
	soc-gemsec-HU	**20**	1	**20**	19	**20**	1	**20**	2	**20**	1
	fb-pages-artist	**20**	196	13	772	**20**	416	19	598	**20**	289
k-Medoid	skewed-sampled	**6**	235	**6**	429	**6**	254	**6**	673	**6**	358
	asymmetric-sampled	**5**	23	**5**	44	**5**	25	**5**	70	**5**	36
	overlap-sampled	**6**	154	**6**	288	**6**	165	**6**	446	**6**	236
	dim032-sampled	**6**	41	**6**	68	**6**	44	**6**	98	**6**	58
	a1-sampled	**3**	18	**3**	24	**3**	18	**3**	23	**3**	20
	s1-sampled	**3**	170	**3**	208	**3**	173	**3**	185	**3**	176
	s2-sampled	**3**	140	**3**	167	**3**	143	**3**	159	**3**	147
	a2-sampled	**3**	179	**3**	218	**3**	182	**3**	196	**3**	186
	unbalance2-sampled	**3**	446	**3**	531	**3**	440	**3**	490	**3**	454
	a3-sampled	**3**	781	**3**	888	**3**	770	**3**	809	**3**	800

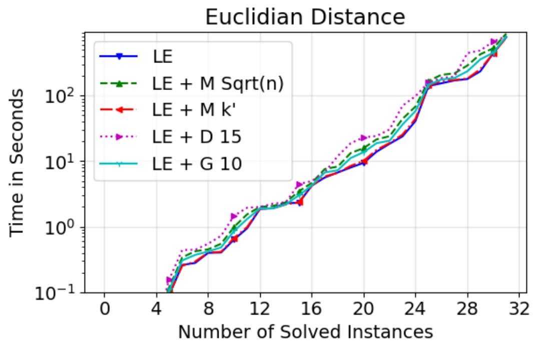
Table 13: Table holding the running times for the Lazy Evaluation algorithm `LE` and some UB2D heuristics configurations. Numbers marked in bold denote the largest k for which the instance could be solved. The reported time is the seconds it takes for the specific instance with the specific k.



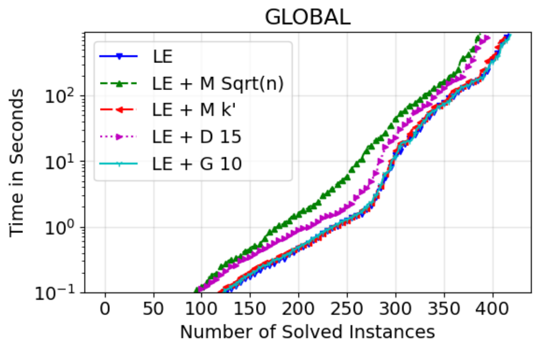
(a) Group Closeness Centrality



(b) Partial Dominating Set



(c) k-Medoid



(d) Global

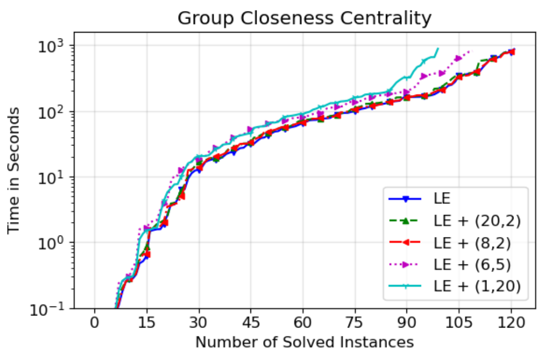
Figure 12: The number of instances solved by the `LE` algorithm and the different UB2D configurations. Point (x, y) on a line denotes that the algorithm solved x problem instances each in most y seconds. The solved instances of one algorithm must not necessarily be the solved instances of another algorithm, but there is a large overlap. Solved instances for $k = 1$ are not included.

5.4 Partial Brute-Force

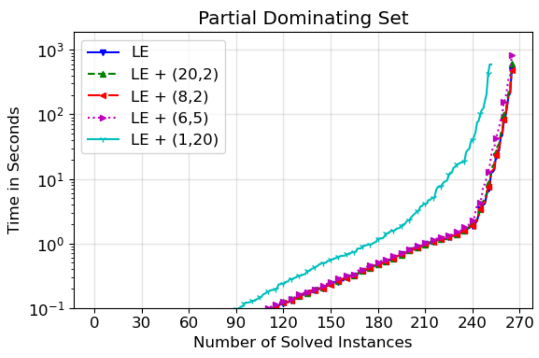
Since we concluded that the Upper Bound 2D heuristics are not useful, let us determine if the Partial Brute-Force heuristics can be helpful. Remember that PBF can take the marginal gain of greater sets and determine a sharper upper bound. However, it will also need more time to compute the upper bound. We will test the following configurations:

- `LE + (20,2)`: $\eta = 20$ blocks each of size $\lambda = 2$.
- `LE + (8,2)`: $\eta = 8$ blocks each of size $\lambda = 2$.
- `LE + (6,5)`: $\eta = 6$ blocks each of size $\lambda = 5$.
- `LE + (1,20)`: $\eta = 1$ block with size $\lambda = 20$.

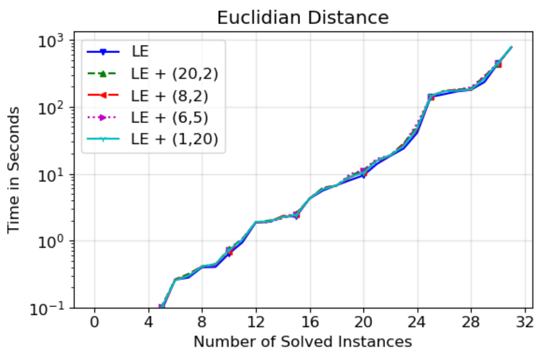
Note that all configurations use a constant number of blocks η and a constant block length λ. During development, we noticed that dynamic η and λ often result in long



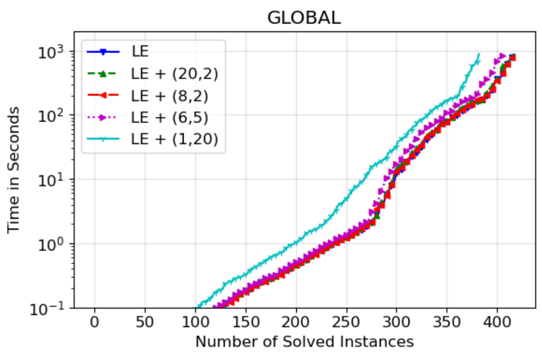
(a) Group Closeness Centrality



(b) Partial Dominating Set



(c) k-Medoid



(d) Global

Figure 13: The number of instances solved by the `LE` algorithm and the four (η, λ) PBF configurations. Point (x, y) on a line denotes that the algorithm solved x problem instances each in most y seconds. The solved instances of one algorithm must not necessarily be the solved instances of another algorithm, but there is a large overlap. Solved instances for $k = 1$ are not included.

running times for either small or large instances. Therefore, we only evaluate constant configurations here but there is no guarantee that they perform better than dynamic values for η or λ. In the appendix, Table 20 and Table 21 show more `LE + ` (η, λ) configurations for constant η and λ.

Results Figure 13 shows the number of solved instances of each configuration for each of the three problems. Table 14 shows the corresponding largest solved k for each instance. The blue line in the figures denotes the `LE` algorithm that any PBF configurations should surpass. Sadly, no configuration manages to improve the running time. We can clearly see that `LE + (1,20)` has the worst running time of the tested configurations. The gained sharpness of the larger sets does not compensate for its costly computation time. Note that $2^{k-|S_T|}$ sets must be evaluated at each node T, which

	Dataset Name	LE		LE + (20,2)		LE + (8,2)		LE + (6,5)		LE + (1,20)	
		k	seconds	k	seconds	k	seconds	k	seconds	k	seconds
Group Close. Centr.	ca-netscience	**14**	867	13	609	13	409	11	519	10	621
	soc-wiki-Vote	**13**	761	12	367	**13**	793	11	604	10	585
	bio-yeast	**11**	460	**11**	575	**11**	483	10	644	9	267
	econ-orani678	**15**	529	**15**	582	**15**	525	13	728	10	653
	soc-advogato	**13**	767	**13**	798	**13**	758	12	809	9	487
	bio-dmela	**9**	637	**9**	654	**9**	644	8	455	7	230
	ia-escorts-dynamic	**8**	628	**8**	692	**8**	648	7	379	7	462
	soc-anybeat	**8**	779	**8**	754	**8**	748	7	791	7	878
	ca-AstroPh	**1**	< 1	**1**	< 1	**1**	< 1	**1**	< 1	**1**	< 1
	fb-pages-media	**1**	< 1	**1**	< 1	**1**	< 1	**1**	< 1	**1**	< 1
	soc-gemsec-RO	**1**	1	**1**	1	**1**	1	**1**	1	**1**	1
	soc-gemsec-HU	**20**	161	**20**	162	**20**	161	**20**	162	**20**	161
	fb-pages-artist	**20**	190	**20**	216	**20**	192	**20**	193	**20**	192
Part. Dom. Set	ca-netscience	**20**	< 1	**20**	< 1	**20**	< 1	**20**	< 1	**20**	15
	soc-wiki-Vote	**20**	< 1	**20**	< 1	**20**	< 1	**20**	< 1	**20**	79
	bio-yeast	**20**	< 1	**20**	< 1	**20**	< 1	**20**	< 1	**20**	1
	econ-orani678	**19**	491	**19**	603	**19**	490	**19**	840	13	590
	soc-advogato	**20**	< 1	**20**	< 1	**20**	< 1	**20**	< 1	**20**	205
	bio-dmela	**20**	< 1	**20**	< 1	**20**	< 1	**20**	< 1	**20**	39
	ia-escorts-dynamic	**20**	< 1	**20**	< 1	**20**	< 1	**20**	< 1	**20**	29
	soc-anybeat	**20**	1	**20**	1	**20**	1	**20**	1	**20**	16
	ca-AstroPh	**20**	27	**20**	34	**20**	29	**20**	63	17	422
	fb-pages-media	**20**	2	**20**	2	**20**	2	**20**	2	**20**	117
	soc-gemsec-RO	**20**	1	**20**	1	**20**	1	**20**	1	**20**	12
	soc-gemsec-HU	**20**	1	**20**	1	**20**	1	**20**	1	**20**	18
	fb-pages-artist	**20**	196	**20**	222	**20**	200	**20**	362	16	593
k-Medoid	skewed-sampled	**6**	235	**6**	287	**6**	271	**6**	290	**6**	264
	asymmetric-sampled	**5**	23	**5**	28	**5**	27	**5**	27	**5**	26
	overlap-sampled	**6**	154	**6**	189	**6**	177	**6**	196	**6**	174
	dim032-sampled	**6**	41	**6**	50	**6**	45	**6**	56	**6**	48
	a1-sampled	**3**	18	**3**	18	**3**	18	**3**	18	**3**	18
	s1-sampled	**3**	170	**3**	172	**3**	174	**3**	171	**3**	171
	s2-sampled	**3**	140	**3**	142	**3**	144	**3**	141	**3**	144
	a2-sampled	**3**	179	**3**	181	**3**	180	**3**	182	**3**	183
	unbalance2-sampled	**3**	446	**3**	442	**3**	441	**3**	453	**3**	454
	a3-sampled	**3**	781	**3**	786	**3**	784	**3**	771	**3**	785

Table 14: Table holding the running times for the `LE` algorithm and four (η, λ) PBF configurations. Numbers marked in bold denote the largest k for which the instance could be solved. The reported time is the seconds it takes for the specific instance with the specific k.

is too much. This can be seen well for Partial Dominating Set for ca-AstroPh. While all other configurations solve the instance for $k = 20$ in a relatively short time, `LE + (1,20)` only solves the instance for $k = 17$.

The second worst configuration is `LE + (6,5)`, which uses 30 candidates at each node. The partitioning into multiple blocks seems beneficial compared to `LE + (1,20)`. The loss in accuracy is compensated by the less costly calculation of the upper bound. However, the configuration still performs worse than `LE + (20,2)` and `LE + (8,2)`. While both of these configurations are relatively equal, `LE + (8,2)` manages to solve soc-wiki-Vote for one larger k than `LE + (20,2)`. In general, it seems that the less costly calculation of `LE + (8,2)` is preferable over the accuracy of `LE + (20,2)`. Note that both of these heuristics are relatively cheap, as each block has only four subsets in total. Surprisingly, even for two relatively cheap heuristics, the cheaper one is preferred over the more accurate one.

Note that the results for k-Medoid Clustering are not influenced much by the configurations. While the configurations need more time to solve the largest k, as shown in Table 14, it is not as bad as for Group Closeness Centrality or Partial Dominating Set, for which less instances get solved by the heuristic. This is unusual, as the `LE + (1,20)` configuration needs much time at each node, so we would expect the running time to suffer more.

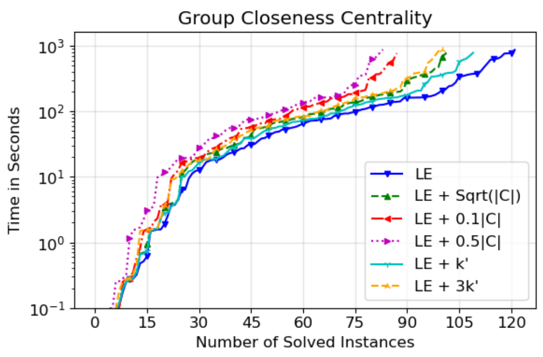
Conclusion for PBF Sadly, Partial Brute-Force also could not decrease the running times either. In general, it seems that small but many blocks should be preferred over large but few blocks. The additional cost of calculating up to all 2^λ marginal gains for each block outweighs the gained sharpness.

5.5 Divide-and-Conquer

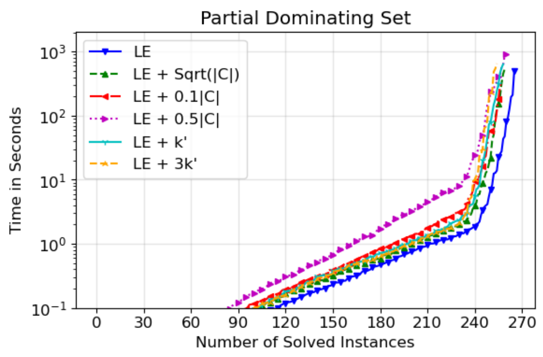
As the last heuristic, we developed the Divide-and-Conquer heuristic, which partitions C_T into P_1 and P_2 and searches on P_1 for the exact solution while using SUB on P_2. Here, we will evaluate the following configurations:

- `LE + Sqrt(|C|)`: $|P_1| = \sqrt{|C_T|}$.
- `LE + 0.1|C|`: $|P_1| = 0.1|C_T|$.
- `LE + 0.5|C|`: $|P_1| = 0.5|C_T|$.
- `LE + k'`: $|P_1| = k'$ with $k' = k - |S_T|$.
- `LE + 3k'`: $|P_1| = 3k'$ with $k' = k - |S_T|$.

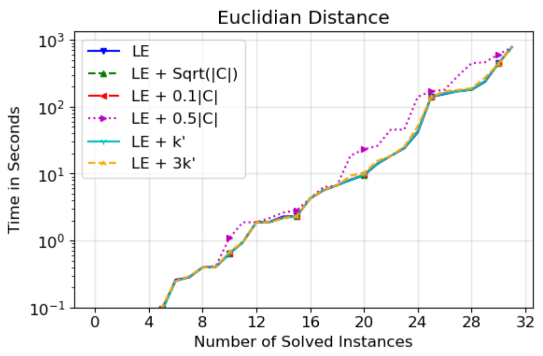
The first three configurations scale with the size of the candidate set, while the other two scale with the depth of the node. In both cases, the cost to compute the heuristic will decrease the further we go down the tree. The `LE + 0.5|C|` configuration will be the most costly, while `LE + k'` is the cheapest. In Figure 14 and Table 15, the results are shown.



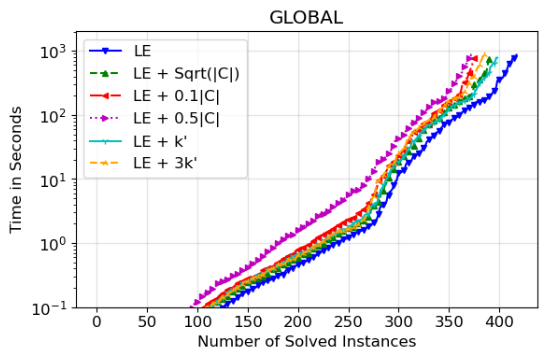
(a) Group Closeness Centrality



(b) Partial Dominating Set



(c) k-Medoid



(d) Global

Figure 14: The number of instances solved by the `LE` algorithm and the different Divide-and-Conquer configuartions. Point (x, y) on a line denotes that the algorithm solved x problem instances each in most y seconds. The solved instances of one algorithm must not necessarily be the solved instances of another algorithm, but there is a large overlap. Solved instances for $k = 1$ are not included.

Results The Divide-and-Conquer heuristics show an even worse behavior than the UB2D and PBF heuristics. `LE` stays by far the fastest configuration. It solves multiple instances more for Group Closeness Centrality and is generally faster for Partial Dominating Set and k-Medoid Clustering. The second fastest configuration is `LE + k'`, followed by `LE + Sqrt(|C|)` and `LE + 3k'`. `LE + 0.5|C|` is the slowest configuration. It solves fewer instances for Group Closeness Centrality and Partial Dominating Set, and for k-Medoid Clustering, it solves the instances much slower.

Note that the running time increases if a configuration uses a larger set P_1. The cheapest heuristic, `LE + k'`, has the fastest running time of all heuristics, while `LE + 0.5|C|` has the worst running time. Similar to Upper Bound 2D and Partial Brute-Force, the running times increase if the heuristic is more expensive to calculate.

	Dataset Name	`LE`		`LE + Sqrt(|C|)`		`LE + 0.1|C|`		`LE + 0.5|C|`		`LE + k'`		`LE + 3k'`	
		k	seconds	k	seconds	k	seconds	k	seconds	k	seconds	k	seconds
Group Close. Centr.	ca-netscience	**14**	867	11	751	10	771	9	768	11	386	9	378
	soc-wiki-Vote	**13**	761	10	381	9	605	9	878	11	571	9	206
	bio-yeast	**11**	460	9	197	8	330	7	641	10	379	9	386
	econ-orani678	**15**	529	11	449	8	545	7	668	12	414	11	550
	soc-advogato	**13**	767	10	838	7	501	6	42	11	389	10	887
	bio-dmela	**9**	637	8	470	5	75	5	169	8	282	8	827
	ia-escorts-dynamic	**8**	628	7	315	5	197	5	232	**8**	701	7	397
	soc-anybeat	**8**	779	6	685	5	367	5	534	**8**	782	7	707
	ca-AstroPh	1	< 1	1	< 1	1	< 1	1	< 1	1	< 1	1	< 1
	fb-pages-media	1	< 1	1	< 1	1	< 1	1	< 1	1	< 1	1	< 1
	soc-gemsec-RO	1	1	1	1	1	1	1	1	1	1	1	1
	soc-gemsec-HU	**20**	161	**20**	162	**20**	160	**20**	162	**20**	162	**20**	162
	fb-pages-artist	**20**	190	**20**	195	**20**	191	**20**	194	**20**	188	**20**	189
Part. Dom. Set	ca-netscience	**20**	< 1	**20**	< 1	**20**	< 1	**20**	< 1	**20**	< 1	**20**	< 1
	soc-wiki-Vote	**20**	< 1	**20**	2	**20**	1	**20**	< 1	**20**	< 1	**20**	2
	bio-yeast	**20**	< 1	**20**	< 1	**20**	< 1	**20**	< 1	**20**	< 1	**20**	< 1
	econ-orani678	19	491	14	450	11	287	14	887	16	638	14	483
	soc-advogato	**20**	< 1	**20**	8	**20**	1	**20**	1	**20**	4	**20**	12
	bio-dmela	**20**	< 1	**20**	< 1	**20**	< 1	**20**	< 1	**20**	2	**20**	< 1
	ia-escorts-dynamic	**20**	< 1	**20**	< 1	**20**	< 1	**20**	1	**20**	< 1	**20**	< 1
	soc-anybeat	**20**	1	**20**	2	**20**	2	**20**	7	**20**	1	**20**	1
	ca-AstroPh	**20**	27	19	541	**20**	21	**20**	40	19	561	18	651
	fb-pages-media	**20**	2	**20**	1	**20**	3	**20**	9	**20**	5	**20**	2
	soc-gemsec-RO	**20**	1	**20**	2	**20**	7	**20**	29	**20**	2	**20**	1
	soc-gemsec-HU	**20**	1	**20**	1	**20**	3	**20**	10	**20**	2	**20**	1
	fb-pages-artist	**20**	196	**20**	321	**20**	327	**20**	897	17	457	16	541
k-Medoid	skewed-sampled	**6**	235	**6**	242	**6**	240	**6**	606	**6**	243	**6**	273
	asymmetric-sampled	**5**	23	**5**	24	**5**	24	**5**	45	**5**	24	**5**	25
	overlap-sampled	**6**	154	**6**	159	**6**	157	**6**	464	**6**	158	**6**	190
	dim032-sampled	**6**	41	**6**	43	**6**	42	**6**	291	**6**	42	**6**	51
	a1-sampled	**3**	18	**3**	18	**3**	18	**3**	18	**3**	18	**3**	18
	s1-sampled	**3**	170	**3**	170	**3**	173	**3**	170	**3**	170	**3**	171
	s2-sampled	**3**	140	**3**	142	**3**	142	**3**	141	**3**	140	**3**	140
	a2-sampled	**3**	179	**3**	179	**3**	181	**3**	179	**3**	181	**3**	178
	unbalance2-sampled	**3**	446	**3**	446	**3**	445	**3**	447	**3**	447	**3**	439
	a3-sampled	**3**	781	**3**	772	**3**	770	**3**	771	**3**	782	**3**	770

Table 15: Table holding the running times for the `LE` algorithm and five DAC configurations. Numbers marked in bold denote the largest k for which the instance could be solved. The reported time is the seconds it takes for the specific instance with the specific k.

Conclusion for DAC The Divide-and-Conquer heuristic can give the most sharp upper bounds of all presented heuristics. However, it heavily pays for the sharpness by a massively increased running time, which overall worsens the heuristic when compared to `LE`. However, since we only evaluated a small fraction of all possible configurations and problems, there is no definitive way to say that the heuristic is never beneficial.

5.6 Using Oracles

The presented heuristics did not have the desired effect. This could have multiple explanations. Maybe the time required to compute them is simply too high, or their returned upper bound is not substantially sharper, so no pruning is possible. Another possibility would be that we do not gain much by using the heuristics because SUB would have pruned the node shortly after. For example, assume that any heuristic h prunes the node. How large is the subtree if we would not prune the node and simply explored the subtree and use SUB on all nodes? It could be pretty small, so any heuristic with a great cost will lose time. Assume, for example, that any UB2D heuristic uses $2p$ candidates in p pairs. If we have strongly diminishing returns, we will sum the marginal gain of $p/2$ candidates, as the smaller marginal gain will vanish. Compare this now with the marginal gains that are present at the node's child. At the child, we will update the marginal gain of all $2p-1$ candidates, and because of submodularity, the marginal gains will be smaller since we just added a candidate to the working set. Depending on how strong the diminishing return is at the child, we can probably also prune right there. Note that one can construct examples where the subtree is presumably pretty large, but these special cases may not necessarily occur in practice. We presume that if the UB2D heuristic or any other heuristic can prune the node, then the explored subtree would have been pretty small anyways, so no massive gain can be acquired.

Oracles To determine if this is the case, we introduce an *oracle*. The oracle can give us the solution of a heuristic for free. By comparing the algorithm that uses the oracle, we can measure the theoretically fastest time we could achieve with the heuristic. This will give us a lower bound on the best achievable time of the algorithm. We will compare two variants of the oracle. Once we will use a *full oracle* that always returns the result for free, and we will use a *partial oracle* that needs $\mathrm{T}(h)$ time[10] to return the result if it leads to pruning, and for free if it does not lead to pruning. In this section, we will use the oracle for the Upper Bound 2D heuristic and the Divide-and-Conquer heuristic. In the appendix in Figure 17 the results for the Partial Brute-Force heuristic are also shown. The results do not differ too much to those of GROUP CLOSENESS CENTRALITY.

Experiment Setup We will use the `LE + D 15` algorithm for the Upper Bound 2D heuristic since it has the sharpest upper bound of all UB2D heuristics and can prune

[10]Denoting the running time of heuristic h.

the most often. Since we get the result of the oracle for free, we will see the greatest difference using this algorithm when using an oracle. For the Divide-And-Conquer heuristic, we will use the `LE + 0.5|C|` algorithm since it has the sharpest upper bound. We will also compare the times to the `LE` algorithm. Note that we can only use instances that were solved by `LE` and the other algorithm. To determine the running times of the oracle, we measured each heuristic call and, at the end, subtracted it from the running time. For Group Closeness Centrality, we will not include the solved instances for soc-gemsec-HU and fb-pages-artist. They solved many instances and would dominate the plots. Because they show no interesting behaviour, we remove them such that a clearer figure for all other instances is shown.

Expected Behaviour It should be clear that the full oracle version is faster than the partial oracle version and the standard algorithm. The question is how fast the `LE` algorithm is in comparison. We know from the previous experiments that it will be faster than `LE + D 15` and `LE + 0.5|C|`. If it is slower than the partial oracle, then we could improve the ratio of successful to unsuccessful heuristic calls to achieve a better running time for our algorithm. Should `LE` be faster than the partial oracle, we would need to improve the running time of the heuristics, either with a more efficient implementation or faster methods. However, suppose `LE` and the full oracle are roughly the same[11] In that case, there is not much hope that the heuristic can be improved to greatly reduce the running time: `LE` traverses the SE Tree in the same order as the other algorithms but has to explore the subtrees that the other algorithms can prune for free. The difference between the `LE` algorithm and the other algorithms tells us how costly the search of the subtree was. If the full oracle and the `LE` algorithm have roughly the same running time, then we know that exploring the subtrees was pretty cheap. The call to the heuristic could in that case be more expensive than pruning the node.

Results Figure 15 shows the results for the Upper Bound 2D heuristic. The blue line represents the `LE` algorithm and is often very close to the pink line, which represents the full oracle for the `LE + D 15` configuration. Therefore, the gain of pruning is not as large as we might have hoped for. The green line represents the `LE + D 15` configuration, while the red line represents the partial oracle for the configuration. The `LE + D 15` configuration has marginally greater running times than the partial oracle. It indicates that the failed heuristic calls were relatively inexpensive. The difference to the pink line is much greater, indicating that the successful heuristic calls are much more expensive. This could be because the heuristic can preemptively return if it determined that no pruning is possible, while it has to check all cases of the partitioning heuristic if it can not prune.

Figure 16 shows the results for the Divide-and-Conquer heuristic. The blue and

[11]Theoretically, the full oracle has to be faster since it can prune nodes for free, but this can vary in practice due to imprecise measurements and code side effects.

pink lines represent the `LE` algorithm and the full oracle again. This time, the difference between both of them is much greater at the beginning. This indicates that the algorithm must spend a substantial portion of time exploring the subtrees that `LE + 0.5|C|` can prune. However, remember that `LE + 0.5|C|` is a very expensive heuristic, so the result should be expected. In other words, it would have been more surprising if such an expensive heuristic only prunes small subtrees. As the instances get harder to solve and need more time, the difference between the `LE + 0.5|C|` configuration and the full oracle starts to vanish for Group Closeness Centrality. It indicates that harder-to-solve instances will not profit from the heuristic. A possible explanation could be that the subtrees, which the full oracle can prune, no longer have a substantial size when compared to the rest of the tree. For small instances, pruning has a much greater effect, while for larger instances, the number of nodes is already very large, and pruning a few does not make a large difference.

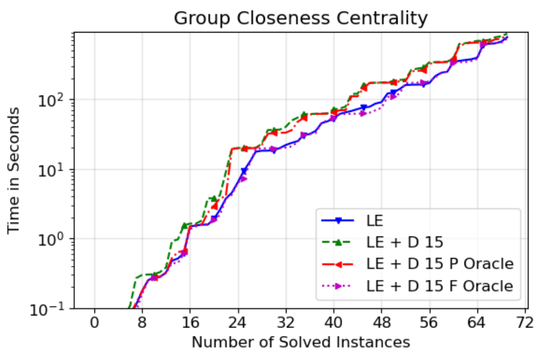
The red and green lines are, this time, even more closely next to each other. This, again, could be an effect of preemptively exiting. Note that the Divide-and-Conquer heuristic can additionally preemptively exit, even though no maximizing set has been found in the recursive search. The heuristic is only concerned with finding a set that exceeds the threshold to pruning, so we can even preemptively cancel the search. This is not possible if the node will be pruned, since the SE Tree of P_1 has to be fully searched to validate that pruning is valid.

Oracle Conclusion Sadly, the oracles show us that our developed heuristics are not helpful in practice. While the heuristics will return sharper upper bounds than SUB, the effect is relatively small. We suspect that exploring a child and thereby getting the updated marginal gains has a similar effect and is relatively cheap, especially when factoring in Lazy Evaluation that decreases the computational effort carried out at each node. While the heuristics can theoretically have a great effect, as the full oracle of `LE + 0.5|C|` shows, it will require a much more sophisticated approach. Note, however, that we only evaluated a small number of configurations and mostly compared the configurations across all instances and all solved k for one specific problem. It is generally hard to find the best hyperparameters for one instance, and finding some for all instances is even harder. A more exhaustive search for hyperparameters for only one instance may be fruitful.

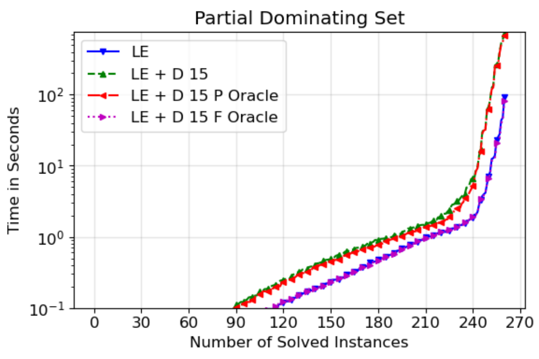
5.7 Comparison to SOTA Algorithm

In the last section of the experiments, we want to compare our algorithms to the State-of-the-Art algorithm of Uematsu et al. [73] published in 2020. Their work also focuses on maximizing submodular monotone functions, using a branch-and-cut algorithm on the variables of a BIP. Their algorithm is an improved version of the constraint generation algorithm [58].

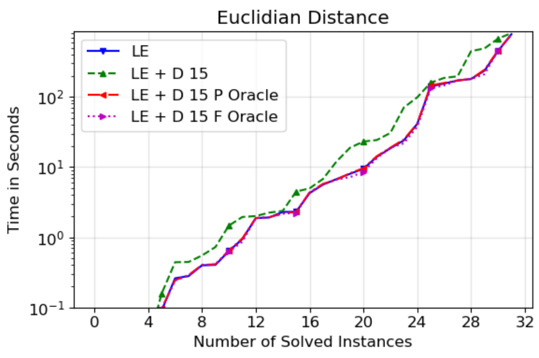
They evaluated their algorithm on three different problems. The first was a version of a Facility Location problem, the second a Weighted Coverage problem,



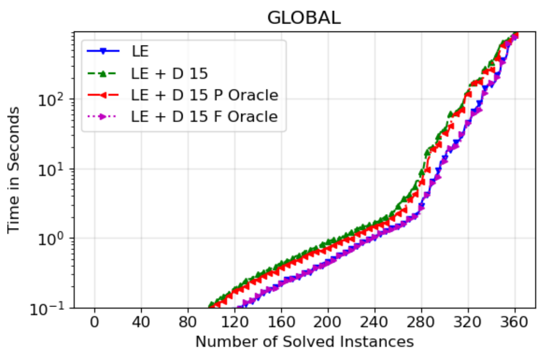
(a) Group Closeness Centrality



(b) Partial Dominating Set



(c) k-Medoid

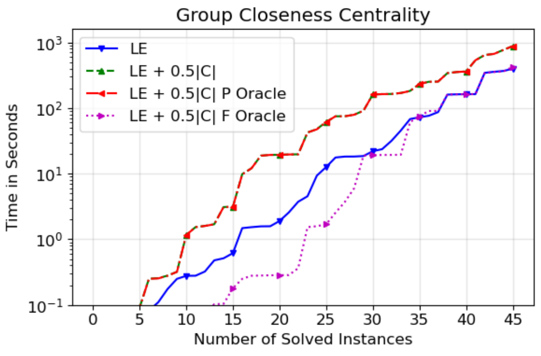


(d) Global

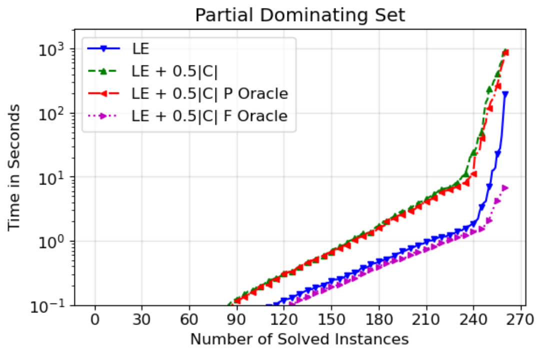
Figure 15: The number of instances solved by `LE` and `LE + D 15`. Additionally, the partial and full oracle are shown for `LE + D 15`. Point (x, y) on a line denotes that the algorithm solved x problem instances each in most y seconds. The solved instances are, in this plot, the same for a fair comparison between the oracle algorithms. Also, all instances of soc-gemsec-HU and fb-pages-artist are removed since they have the same required time. Solved instances for $k = 1$ are not included.

and the third a Bipartite Influence problem. Note that the achieved execution times of their algorithms differ for all the problems. In general, their version of the Facility Location problem seems to be the hardest as the solved instances are smaller (measured by n) but take longer to solve. Therefore, we will only compare our algorithm to their algorithm on this problem. However, comparing the algorithms on all problems will produce more meaningful results and a fairer comparison.

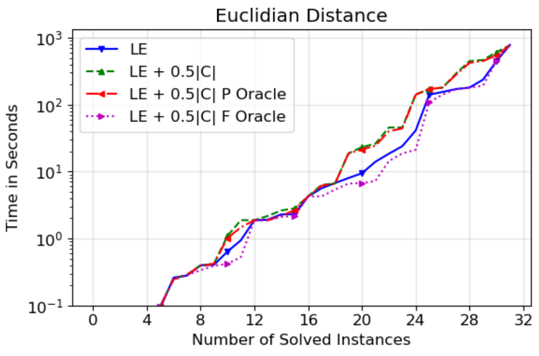
Facility Location Let $N = \{1, \ldots, n\}$ be a set of locations and $M = \{1, \ldots, n+1\}$ a set of customers. By $g_{ij} \geq 0$, we denote the benefit for customer $j \in M$ when building a facility at location $i \in N$. The task is to choose a set $S \subseteq N$ of k locations to maximize the sum of benefits. Each customer will only consider the location that benefits them



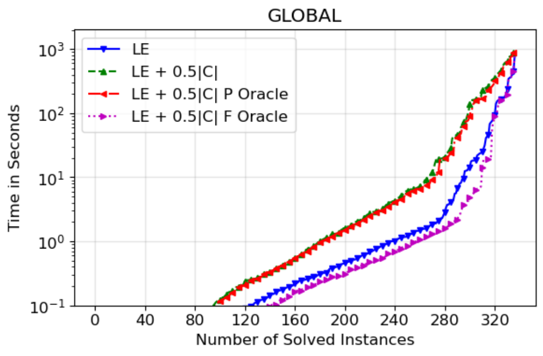
(a) Group Closeness Centrality



(b) Partial Dominating Set



(c) k-Medoid



(d) Global

Figure 16: The number of instances solved by `LE` and the DAC `LE + 0.5|C|` configuration. Additionally, the partial and full oracle are shown for `LE + 0.5|C|` Point (x, y) on a line denotes that the algorithm solved x problem instances each in most y seconds. The solved instances are, in this plot, the same for a fair comparison between the oracle algorithms. Also, all instances of soc-gemsec-HU and fb-pages-artist are removed since they have the same required time. Solved instances for $k = 1$ are not included.

the most. The problem can be formulated by

$$f(S) = \sum_{j \in M} \max_{i \in S} g_{ij}.$$

Without proof, this function is submodular and monotone. Uematsu et al. [73] evaluated instances with parameters $n = 20, 40, 60$, and $k = 5, 8$. The benefits g_{ij} are uniformly sampled from $[0, 1]$. Their used data was not freely available, so we randomly generated three datasets for each n on our own.

Sadly, we could not test their implementation ourselves (as it is also not freely available), so we will here take the results report by Uematsu et al. [73]. They performed their experiments on a 4.0 GHz Intel® Core™ i7 processor and 32 GB of main memory. We, on the other hand, perform this experiment on an Intel® Core™ i7-8700K with 3.7

n	k	`Plain`	`SUB + RPC`	`BC-ICG+`
20	5	0.001	0.000	0.46
40	5	0.051	0.002	40.52
60	5	0.642	0.020	508.97
20	8	0.007	0.000	0.30
40	8	6.856	0.018	79.86
60	8	333.3	0.336	4758.70

Table 16: The results of our developed `Plain` and `SUB + RPC` algorithm compared to the `BC-IGG+` algorithm of Uematsu et al. [73]. The tested instances are facility location problems with n locations and $n + 1$ customers.

GHz and 64 GB RAM. While our system has a slightly lower clock speed, the systems are still roughly comparable to each other. We will compare `Plain` and `SUB + CR` against their `BC-ICG+` algorithms. Note that they also compared their algorithm to the work of Nemhauser and Wolsey [58], Chen et al. [16], and Sakaue and Ishihata [64]. Their `BC-ICG+` algorithm is, in general, the fastest algorithm to solve the Facility Location problem.

Table 16 shows the results. We averaged the execution time across the three generated instances. Surprisingly, even the Plain Search algorithm is magnitudes faster than their `BC-ICG+` algorithm. It thus seems that their BIP approach is not very efficient. Why that is, we can only speculate. They mention in their that the standard constraint generation algorithm often has to solve many BIP problems. Their improved algorithm mitigates the number of BIP problems solved, but it could be that this number is still very large.

5.8 Summary of the Results

We tested our algorithm on three problems: Group Closeness Centrality, Partial Dominating Set, and k-Medoid Clustering. The experiments showed that the Basic Search algorithm (Algorithm 2) performs well compared to the specialized branch-and-bound algorithm for Group Closeness Centrality by Staus et al. [69]. The Basic Search algorithm also performs magnitudes faster than the SOTA algorithm for Cardinality Constrained Maximization by Uematsu et al. [73]. The additionally deployed Lazy Evaluation can even further decrease the running time by multiple factors. The newly developed heuristics, however, had no positive effect. Using them will mostly result in slower running times. Only once did the `LE + G 10` configuration of the Greedy Upper Bound 2D heuristic surpass the `LE` algorithm. We noticed that all partitioning heuristics decrease in efficiency the larger we set the partition P_1. It indicates that keeping the computational effort low at each node of the SE Tree is preferable. An additional indication that the heuristics can not perform well is given by the oracle version of the algorithms. They show that the additional pruning gained by the heuristics only has a marginal effect that can quickly be compensated by the Lazy Evaluation algorithm.

Note that due to the great computational cost of evaluating an algorithm configuration, we could only test a few configurations for each heuristic. It can very well be that we missed crucial configurations that would have performed better.

6 Conclusion

6.1 Summary

In this work, we developed an efficient branch-and-bound algorithm to solve Cardinality Constrained Maximization for arbitrary submodular monotone set functions. Section 3.4 described the Basic Search algorithm, primarly based on the work of Staus et al. [69]. The algorithm improves the previously achieved running times for Group Closeness Centrality. The Basic Search algorithm also achieves magnitudes faster running times for Cardinality Constrained Maximization when compared to the State-of-the-Art algorithm by Uematsu et al. [73]. We further added new methods and heuristics to the algorithm to improve its running time. The best improvement is to utilize Lazy Evaluation, which updates only a fraction of all candidates at a node. The average update scheme and its variations can decrease the running time by multiple factors for some instances, and they are generally faster than all other update schemes.

We also developed new heuristics that determine sharper upper bounds, so the algorithm can more often prune a node. These are Upper Bound 2D, which considers the marginal gain of pairs, Partial Brute Force, which considers marginal gains of larger sets and Divide-and-Conquer, which determine the exact upper bound on a fraction of the candidates. While we could show that the heuristics will return sharper upper bounds than SUB, they sadly could not deliver the hoped for impact in practice. We suspect that the size of the pruned subtrees is not that large and that SUB can also easily prune the nodes, once explored. Although it is a disappointing result, it shows that we should prefer to keep the required work at a node to the minimum to achieve the most efficient search.

Overall, we improved the current State-of-the-Art to solve Cardinality Constrained Maximization for arbitrary submodular monotone set function.

6.2 Future Work

We have many ideas that can still be incorporated to improve the practical performance and usability of the algorithm. Here, we want to name a few.

A* Search Algorithm The here-developed search algorithm performs a depth-first search. It will explore the tree from the root down as far as possible. Only when arriving at a leaf node or having exhausted all children of a node will cause the algorithm to backtrack to an earlier node.

The A* Search algorithm [32] introduces a more sophisticated approach. For each node explored in the SE Tree, the algorithm considers the achieved value $g(S_T) = f(S_T)$ and a heuristic $h(S_T)$ that denotes how much score can still be achieved at the node. By considering the sum $g(S_T) + h(S_T)$, each node is ranked by the algorithm, and the one node with the greatest value is expanded first, as it seems the most promising.

The algorithm is one of the most used ones and has seen huge success in pathfinding and other fields. Chen et al. [16] use an A* Search algorithm in their framework to exactly solve submodular functions. Such an approach should be explored. However, keep in mind that the algorithm has an $\mathcal{O}(n^k)$ space complexity, so it cannot be used trivially. A tradeoff would be to use an A* Search algorithm close to the root of the SE Tree and the depth-first search algorithm once a memory threshold is exceeded. Another approach would be to specify a depth as a hyperparameter, after which the algorithm will change from the A* search to a depth-first search.

Another Ordering than DCO So far, we have only considered Dynamic Candidate Ordering to reorder the candidates at a node. However, who says this ordering is perfect or no other suitable ordering exists? In general, we would only like to expand the subtrees that hold a better solution, as it increases the chances that we can prune all other subtrees. Dynamic Candidate Ordering tries to achieve this by sorting the candidates by their marginal gain, but this only estimates where the best set may reside. How well does it estimate? An experimental evaluation could indicate which child holds the set with the greatest score. Based on the results, we could analyze if a different ordering results in a faster search. Note that all of our developed heuristics often assume the candidate set of the parent to be sorted by Dynamic Candidate Ordering. Using a different ordering could worsen their running time.

More Sophisticated Greedy Algorithm The literature on greedy algorithms for the maximization of submodular is huge. We only deployed two very basic greedy algorithms to bootstrap our search algorithm. However, better greedy algorithms at the start can lead to more pruning and decrease the running time. Before such an approach is explored, however, one should first determine if bootstrapping the search algorithm with the perfect solution substantially improves the running time. If it does not improve the running time, then the algorithm will spend most of the time by verifying that the current solution is indeed the best. A more sophisticated greedy approach would not help in such a case.

Parallelization Currently, the algorithm only uses one core to solve the problem. Nowadays, multicore systems are the standard in the industry. By not using them, we lose the opportunity to solve larger problem instances with greater k's.

The parallelization could be utilized in two places. Either we parallelize the search, that is, multiple threads search different parts of the SE Tree. At the root, we could assign each thread a set of children, and then each thread could work independently. Note that the subtrees of the children will have different sizes, so a good distribution is needed. Should one thread finish before the others, it should help the other threads solve their part of the SE Tree. The only value that the threads should share is the best score found. Then, each thread could prune based on the global best score and not only on the best score it found by itself.

The other place where we could utilize parallelization would be when calculating the marginal gains for a node. In general, we have to evaluate the score function a total of $\mathcal{O}(n)$ times at each node. The function calls are all independent of each other, which makes them suitable to parallelize.

A hybrid approach of both methods can, of course, also be considered. Theoretically, a parallelization with p processes should also almost result in p times the speedup if the work is distributed equally and communication is kept to a minimum. However, the algorithm is currently memory-bound, so that more processes will put further strain on the memory bus. A speedup of p will most likely not be achieved.

GPU Support An improvement that could help with the memory-bound problem would be to utilize the Graphics Processing Unit (GPU). In contrast to the CPU, the GPU is designed for simple repetitive tasks that can be broken down into smaller components and executed in parallel. The GPU, therefore, offers itself up to evaluate the $\mathcal{O}(n)$ score function evaluations at each node. This, however, is only efficient if the way to evaluate the score function suits the GPU architecture. Luckily, the problems evaluated in the experiments (see Section 5) are very well computable on a GPU. Under the hood, all problems can be expressed via a "Matrix Row Maximization" problem. That is, we are interested in choosing k rows of the matrix, such that when the rows are maximized component-wise into a single vector, and we take the sum of this vector, this sum should be maximal. The process involves component-wise vector maximization and the sum of a vector, which can both be implemented efficiently on GPUs.

If we utilize a GPU, it would also help with the memory-bound problem. The corresponding matrix would not be stored in the main memory but in the VRAM (Video RAM). Parts of the matrix do not have to be transferred across the memory bus as the VRAM is directly connected to the cores on the GPU.

Include Domain Specific Knowledge Such an improvement generally has to be performed for each individual score function. Staus et al. [69] used Dominating Vertices to initially reduce the number of vertices for the Group Closeness Centrality problem. For the k-Medoid Clustering problem, we can probably identify outliers that will definitely not be part of a solution. They can be removed as a preprocessing or intermediate step to reduce the search space. Additionally, we could expose some functionality such that the user can write its own problem-specific pruning rules that submodularity cannot exploited.

Extending the Usability One way to increase the usability of our algorithm is to allow other constraints. Cardinality constraints are not the only constraints that arise in practice, but Knapsack and matroid constraints, as mentioned in Section 2, also arise in various problems. Knapsack constraints associate a cost with each candidate, and matroid constraints are a special subset of $2^{\mathcal{U}}$. All these constraints are similar, and we think they can be incorporated without much effort in the algorithm. Note that the cardinality constraint can be considered as a special case of the Knapsack constraint. Chen et al. [16] already incorporated them in their framework, which also solves the Cardinality Constrained Maximization problem for submodular functions.

Another change could be to drop the monotonicity requirement, as in the work of Kawahara et al. [40]. This would allow us to solve even more problems. However, this improvement requires more work to be incorporated, as all of our heuristics assume that the optimal solution will have exactly k elements. Submodular functions without the monotonicity property can also have maximizing solutions with fewer elements.

Test more Problems Our experiments showed the most interesting results for Group Closeness Centrality, while Partial Dominating Set was too easy (at least for $k \in [1, 20]$) and k-`Medoid Clustering` was too hard. We mentioned various problems like Facility Location, Hitting Set, or Weighted Coverage that exhibit submodularity and monotonicity. Evaluating them could grant more insight into the performance of our algorithm and the developed methods. Maybe for those problems, our developed heuristics will show more beneficial behavior.

References

[1] Isaiah G. Adebayo and Yanxia Sun. A novel approach of closeness centrality measure for voltage stability analysis in an electric power grid. *International Journal of Emerging Electric Power Systems*, 21(3):2020–0013.

[2] Alexander A. Ageev and Maxim Sviridenko. Approximation algorithms for maximum coverage and max cut with given sizes of parts. In *Integer Programming and Combinatorial Optimization, 7th International IPCO Conference (IPCO '99)*, volume 1610 of *Lecture Notes in Computer Science*, pages 17–30. Springer, 1999.

[3] Akanksha Agrawal, Pratibha Choudhary, Pallavi Jain, Lawqueen Kanesh, Vibha Sahlot, and Saket Saurabh. Hitting and covering partially. In *Computing and Combinatorics - 24th International Conference (COCOON '18)*, volume 10976 of *Lecture Notes in Computer Science*, pages 751–763. Springer, 2018.

[4] Amir Ahmadi-Javid, Pardis Seyedi, and Siddhartha S. Syam. A survey of healthcare facility location. *Computers & Operations Research*, 79:223–263, 2017.

[5] Sibel A. Alumur and Bahar Yetis Kara. Network hub location problems: The state of the art. *European Journal of Operational Research*, 190(1):1–21, 2008.

[6] Sanjeev Arora and Boaz Barak. *Computational Complexity - A Modern Approach*. Cambridge University Press, 2009.

[7] Pavel Berkhin. A survey of clustering data mining techniques. In *Grouping Multidimensional Data - Recent Advances in Clustering (ICGSP '19)*, pages 25–71. Springer, 2006.

[8] Sanjiv K. Bhatia and Jitender S. Deogun. Conceptual clustering in information retrieval. *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, 28(3): 427–436, 1998.

[9] Endre Boros and Peter L. Hammer. Pseudo-boolean optimization. *Discrete Applied Mathametics*, 123(1-3):155–225, 2002.

[10] Yuri Boykov and Marie-Pierre Jolly. Interactive graph cuts for optimal boundary and region segmentation of objects in N-D images. In *Proceedings of the Eighth International Conference On Computer Vision (ICCV-01) - Volume 1 (IVVV '01.*

[11] Kurt M. Bretthauer and Bala Shetty. The nonlinear knapsack problem - algorithms and applications. *European Journal of Operational Research*, 138(3):459–472, 2002.

[12] James F. Campbell and Morton E. O'Kelly. Twenty-five years of hub location research. *Transportation Science*, 46(2):153–169, 2012.

[13] Eric Chea and Dennis R. Livesay. How accurate and statistically robust are catalytic site predictions based on closeness centrality? *BMC Bioinformatics*, 8, 2007.

[14] Chen Chen, Wei Wang, and Xiaoyang Wang. Efficient maximum closeness centrality group identification. In *Databases Theory and Applications - 27th Australasian Database Conference (ADC '16)*, volume 9877 of *Lecture Notes in Computer Science*, pages 43–55. Springer, 2016.

[15] Jianer Chen, Benny Chor, Mike Fellows, Xiuzhen Huang, David W. Juedes, Iyad A. Kanj, and Ge Xia. Tight lower bounds for certain parameterized np-hard problems. *Information and Computation (CCC '04)*, 201(2):216–231, 2005.

[16] Wenlin Chen, Yixin Chen, and Kilian Q. Weinberger. Filtered search for submodular maximization with controllable approximation bounds. In *Proceedings of the Eighteenth International Conference on Artificial Intelligence and Statistics, AISTATS '15*, volume 38 of *JMLR Workshop and Conference Proceedings*. JMLR.org, 2015.

[17] Kevin K. H. Cheung and Kyel Governor. A detailed introduction to a minimum-cost perfect matching algorithm based on linear programming. *Journal of Combinatorial Optimization*, 45(3):85, 2023.

[18] Donald J. Chmielewski, Tasha Palmer, and Vasilios Manousiouthakis. On the theory of optimal sensor placement. *AIChE Journal*, 48(5):1001–1012, 2002.

[19] G.B. Coleman and H.C. Andrews. Image segmentation by clustering. *Proceedings of the IEEE*, 67(5):773–785, 1979.

[20] Iván A. Contreras and Elena Fernández. Hub location as the minimization of a supermodular set function. *Operations Research*, 62(3):557–570, 2014.

[21] Gérard Cornuéjols, George Nemhauser, and Laurence Wolsey. The uncapicitated facility location problem. Technical report, Cornell University Operations Research and Industrial Engineering, 1983.

[22] © Intel Corporation. Intel® Vtune™ Profiler. URL `https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html#gs.7qxdmf`. Accessed: 2023-11-02.

[23] Ran Duan and Seth Pettie. Linear-time approximation for maximum weight matching. *Journal of the ACM*, 61(1):1:1–1:23, 2014.

[24] Jack Edmonds. Maximum matching and a polyhedron with 0,1-vertices. *Journal of Research of the National Bureau of Standards Section B Mathematics and Mathematical Physics*, page 125, 1965.

[25] Pasi Fränti and Sami Sieranoja. K-means properties on six clustering benchmark datasets. *Applied Intelligence*, 48(12):4743–4759, 2018.

[26] Harold N. Gabow. Data structures for weighted matching and nearest common ancestors with linking. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms (ACM-SIAM '90)*, pages 434–443. SIAM, 1990.

[27] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.

[28] Boris Goldengorin, Gerard Sierksma, Gert A. Tijssen, and Michael Tso. The data-correcting algorithm for the minimization of supermodular functions. *Management Science*, 45(11):1539–1551, 1999.

[29] Marco Gori, Marco Maggini, and Lorenzo Sarti. Exact and approximate graph matching using random walks. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, 27(7):1100–1111, 2005.

[30] Pierre Hansen and Brigitte Jaumard. Algorithms for the maximum satisfiability problem. *Computing*, 44(4):279–303, 1990.

[31] David G. Harris. Distributed local approximation algorithms for maximum matching in graphs and hypergraphs. *SIAM Journal on Coputing*, 49(4):711–746, 2020.

[32] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[33] Russell Impagliazzo and Ramamohan Paturi. On the complexity of k-sat. *Journal of Computer and System Sciences*, 62(2):367–375, 2001.

[34] Satoru Iwata, Lisa Fleischer, and Satoru Fujishige. A combinatorial, strongly polynomial-time algorithm for minimizing submodular functions. In *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing (STOC '00)*, pages 97–106. ACM, 2000.

[35] KRAR Jakob and Peter Mark Pruzan. The simple plant location problem: Survey and synthesis. *European Journal of Operational Research*, 12(36-81):41, 1983.

[36] Stefanie Jegelka and Jeff A. Bilmes. Submodularity beyond submodular energies: Coupling edges in graph cuts. In *The 24th IEEE Conference on Computer Vision and Pattern Recognition (CVPR '11)*, pages 1897–1904. IEEE Computer Society, 2011.

[37] Hua Jiang and Zhifei Zheng. An exact algorithm for the minimum dominating set problem. In *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence (IJCAI '23)*, pages 5604–5612. ijcai.org, 2023.

[38] Richard M. Karp. Reducibility among combinatorial problems. In *Proceedings of a symposium on the Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972.

[39] Leonard Kaufmann and Peter Rousseeuw. Clustering by means of medoids. *Data Analysis based on the L1-Norm and Related Methods*, pages 405–416, 1987.

[40] Yoshinobu Kawahara, Kiyohito Nagano, Koji Tsuda, and Jeff A. Bilmes. Submodularity cuts and applications. In *Proceedings of Advances in Neural Information Processing Systems 22: 23rd Annual Conference on Neural Information Processing Systems 2009 (NIPS '09).*, pages 916–924. Curran Associates, Inc., 2009.

[41] David Kempe, Jon M. Kleinberg, and Éva Tardos. Maximizing the spread of influence through a social network. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '03)*, pages 137–146. ACM, 2003.

[42] Samir Khuller, Anna Moss, and Joseph Naor. The budgeted maximum coverage problem. *Information Processing Letters*, 70(1):39–45, 1999.

[43] Joachim Kneis, Daniel Mölle, and Peter Rossmanith. Partial vs. complete domination: t-dominating set. In *SOFSEM 2007: Theory and Practice of Computer Science, 33rd Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM '07)*, volume 4362 of *Lecture Notes in Computer Science*, pages 367–376. Springer, 2007.

[44] Joachim Kneis, Alexander Langer, and Peter Rossmanith. Improved upper bounds for partial vertex cover. In *Graph-Theoretic Concepts in Computer Science, 34th International Workshop, WG 2008. Revised Papers (WG '08)*, volume 5344 of *Lecture Notes in Computer Science*, pages 240–251, 2008.

[45] Pushmeet Kohli, M. Pawan Kumar, and Philip H. S. Torr. Px00b3 & beyond: Move making algorithms for solving higher order functions. *IEEE Transaction on Pattern Analysis and Machine Intelligence*, 31(9):1645–1656, 2009.

[46] Vladimir Kolmogorov. Blossom V: a new implementation of a minimum cost perfect matching algorithm. *Mathatmetical Programming Computation*, 1(1):43–67, 2009.

[47] Andreas Krause and Daniel Golovin. Submodular function maximization. In *Tractability: Practical Approaches to Hard Problems*, pages 71–104. Cambridge University Press, 2014.

[48] Andreas Krause and Carlos Guestrin. Near-optimal observation selection using submodular functions. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence (AAAI '07)*, pages 1650–1654. AAAI Press, 2007.

[49] Andreas Krause, Ram Rajagopal, Anupam Gupta, and Carlos Guestrin. Simultaneous placement and scheduling of sensors. In *Proceedings of the 8th International Conference on Information Processing in Sensor Networks, IPSN 2009*, pages 181–192. IEEE Computer Society, 2009.

[50] Andreas Krause, Carlos Guestrin, Anupam Gupta, and Jon M. Kleinberg. Robust sensor placements at informative and communication-efficient locations. *ACM Transactions on Sensor Networks (TOSN '10)*, 7(4):31:1–31:33, 2011.

[51] Marion Lauriere. An algorithm for the 0/1 knapsack problem. *Mathematical Programming*, 14(1):1–10, 1978.

[52] Jure Leskovec, Andreas Krause, Carlos Guestrin, Christos Faloutsos, Jeanne M. VanBriesen, and Natalie S. Glance. Cost-effective outbreak detection in networks. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '07)*, pages 420–429. ACM, 2007.

[53] Hui Lin and Jeff A. Bilmes. A class of submodular functions for document summarization. In *The 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies, Proceedings of the Conference (ACL '11)*, pages 510–520. The Association for Computer Linguistics, 2011.

[54] Michel Minoux. Accelerated greedy algorithms for maximizing submodular set functions. In *Optimization Techniques*, pages 234–243. Springer Berlin Heidelberg, 1978.

[55] Boris G. Mirkin. *Clustering for data mining - a data recovery approach.* Computer science and data analysis series. Chapman & Hall, 2005.

[56] Thomas Moscibroda and Roger Wattenhofer. Maximizing the lifetime of dominating sets. In *19th International Parallel and Distributed Processing Symposium (IPDPS 2005), CD-ROM / Abstracts Proceedings (IPDPS '05)*. IEEE Computer Society, 2005.

[57] George L. Nemhauser, Laurence A. Wolsey, and Marshall L. Fisher. An analysis of approximations for maximizing submodular set functions - I. *Mathatmetical Programming*, 14(1):265–294, 1978.

[58] G.L. Nemhauser and L.A. Wolsey. Maximizing submodular set functions: Formulations and analysis of algorithms. In *Annals of Discrete Mathematics (11)*, volume 59 of *North-Holland Mathematics Studies*, pages 279–301. North-Holland, 1981.

[59] Gbeminiyi John Oyewole and George Alex Thopil. Data clustering: application and trends. *Artificial Intelligence Review*, 56(7):6439–6475, 2023.

[60] Thrasyvoulos N. Pappas and Nikil S. Jayant. An adaptive clustering algorithm for image segmentation. In *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP '89)*, pages 1667–1670. IEEE, 1989.

[61] Mohammad Rezaei and Pasi Fränti. Can the number of clusters be determined by external indices? *IEEE Access*, 8:89239–89257, 2020.

[62] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence (AAAI '15)*, pages 4292–4293. AAAI Press, 2015.

[63] Ron Rymon. Search through systematic set enumeration. In *Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning (KR'92)*, pages 539–550. Morgan Kaufmann, 1992.

[64] Shinsaku Sakaue and Masakazu Ishihata. Accelerated best-first search with upper-bound computation for submodular function maximization. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18)*, pages 1413–1421. AAAI Press, 2018.

[65] Boris Schling. *The Boost C++ Libraries*. XML Press, 2011.

[66] Alexander Schrijver. A combinatorial algorithm minimizing submodular functions in strongly polynomial time. *Journal of Combinational Theory, Series. B*, 80(2): 346–355, 2000.

[67] Chao Shen and Tao Li. Multi-document summarization via the minimum dominating set. In *COLING 2010, 23rd International Conference on Computational Linguistics, Proceedings of the Conference (COLING '10)*, pages 984–992. Tsinghua University Press, 2010.

[68] Amarjeet Singh, Andreas Krause, Carlos Guestrin, William J. Kaiser, and Maxim A. Batalin. Efficient planning of informative paths for multiple robots. In *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI '07)*, pages 2204–2211, 2007.

[69] Luca Pascal Staus, Christian Komusiewicz, Nils Morawietz, and Frank Sommer. Exact algorithms for group closeness centrality. In *SIAM Conference on Applied and Computational Discrete Algorithms (ACDA '23)*, pages 1–12. SIAM, 2023.

[70] Ivan Stojmenovic, Mahtab Seddigh, and Jovisa D. Zunic. Dominating sets and neighbor elimination-based broadcasting algorithms in wireless networks. *IEEE Transactions on Parallel Distributed Systems*, 13(1):14–25, 2002.

[71] Matthew J. Streeter and Daniel Golovin. An online algorithm for maximizing submodular functions. In *Advances in Neural Information Processing Systems 21, Proceedings of the Twenty-Second Annual Conference on Neural Information Processing Systems (NIPS '08)*, pages 1577–1584. Curran Associates, Inc., 2008.

[72] Sergios Theodoridis and Konstantinos Koutroumbas. Pattern recognition. *IEEE Transactions on Neural Networks*, 19(2):376, 2008.

[73] Naoya Uematsu, Shunji Umetani, and Yoshinobu Kawahara. An efficient branch-and-cut algorithm for submodular function maximization. *Journal of the Operations Research Society of Japan*, 63(2):41–59, 2020.

[74] Jan Vondrak. Submodularity and curvature: the optimal algorithm. *RIMS Kôkyûroku Bessatsu*, pages 253–266, 2010.

[75] Alfred Weber and Carl J. Friedrich. *Alfred Weber's theory of the location of industries.* Materials for the study of business. The University of Chicago Press Chicago, Ill., 1929.

[76] Hassler Whitney. On the abstract properties of linear dependence. *American Journal of Mathematics*, 57(3):509–533, 1935.

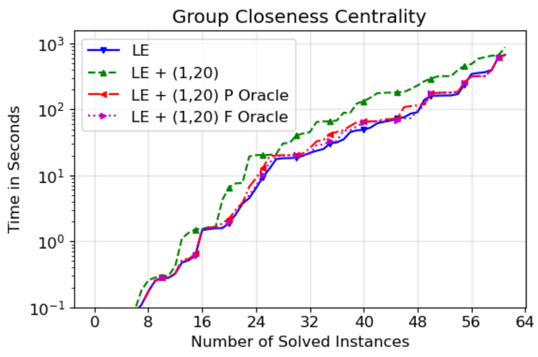
[77] Gert W. Wolf. Facility location: concepts, models, algorithms and case studies. series: Contributions to management science. *International Journal of Geographical Information Science*, 25(2):331–333, 2011.

[78] Jie Wu and Hailan Li. A dominating-set-based routing scheme in ad hoc wireless networks. *Telecommunications Systems*, 18(1-3):13–36, 2001.

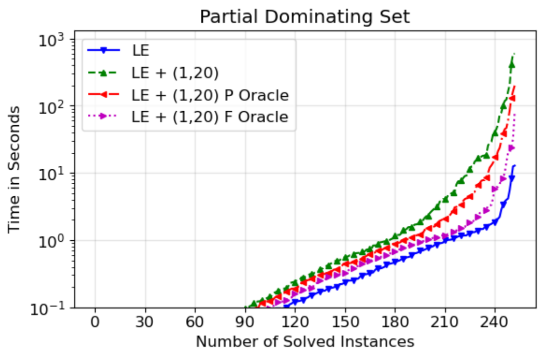
[79] Weili Wu, Hui Xiong, and Shashi Shekhar. *Clustering and Information Retrieval.* Kluwer, 2003.

[80] Yi-Zhi Xu and Hai-Jun Zhou. Generalized minimum dominating set and application in automatic text summarization. *CoRR*, abs/1602.04930, 2016.

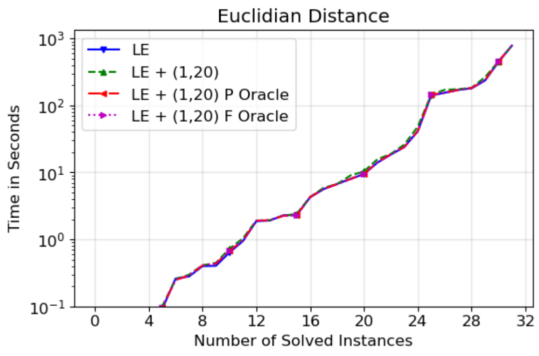
A Appendix



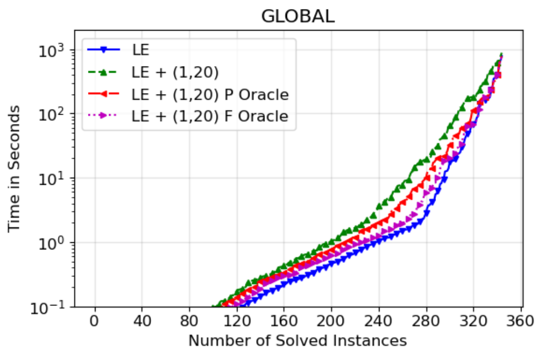
(a) Group Closeness Centrality



(b) Partial Dominating Set



(c) k-Medoid



(d) Global

Figure 17: The number of instances solved by `LE` and `LE + (1,20)`. The partial and full oracle are also shown for `LE + (1,20)`. Note that the blue line can theoretically not be faster than the pink line. However, this could be caused by inaccurate measurements and other side effects, as the timings are taken by two different runs of the algorithm. Point (x, y) on a line denotes that the algorithm solved x problem instances each in most y seconds. The solved instances are in this plot the same, for a fair comparison between the oracle algorithms. Solved instances for $k = 1$ are not included.

	Dataset Name	`B`		`B + FP`		`B + Avg`		`B +` k'		`B +` $0.5\|C\|$		`B + Avg or` $3k$		`B + Avg or` $0.1n$	
		k	seconds	k	seconds	k	seconds	k	seconds	k	seconds	k	seconds	k	seconds
Group Close. Centr.	ca-netscience	13	700	13	389	**14**	783	4	28	9	555	**14**	867	**14**	869
	soc-wiki-Vote	11	522	12	539	**13**	673	3	14	8	369	**13**	761	**13**	899
	bio-yeast	10	712	10	250	**11**	417	3	115	7	360	**11**	460	**11**	748
	econ-orani678	11	483	13	597	**16**	860	2	1	5	293	15	529	14	637
	soc-advogato	8	338	11	582	**13**	704	2	18	2	18	**13**	767	11	402
	bio-dmela	6	256	7	291	**9**	648	2	68	2	61	**9**	637	8	614
	ia-escorts-dynamic	6	412	7	732	**8**	666	2	160	2	198	**8**	628	7	689
	soc-anybeat	5	454	6	675	**8**	765	2	350	2	347	**8**	779	6	677
	ca-AstroPh	**1**	< 1	**1**	< 1	**1**	< 1	**1**	< 1	**1**	< 1	**1**	< 1	**1**	< 1
	fb-pages-media	**1**	< 1	**1**	< 1	**1**	< 1	**1**	< 1	**1**	< 1	**1**	< 1	**1**	< 1
	soc-gemsec-RO	**1**	1	**1**	1	**1**	1	**1**	1	**1**	1	**1**	1	**1**	1
	soc-gemsec-HU	2	2	2	3	2	6	2	4	2	2	**20**	161	**20**	161
	fb-pages-artist	2	4	2	4	2	3	2	3	2	3	**20**	190	**20**	190
Part. Dom. Set	ca-netscience	**20**	< 1	**20**	< 1	**20**	< 1	**20**	< 1	**20**	< 1	**20**	< 1	**20**	< 1
	soc-wiki-Vote	**20**	< 1	**20**	< 1	**20**	< 1	**20**	15	**20**	< 1	**20**	< 1	**20**	< 1
	bio-yeast	**20**	< 1	**20**	< 1	**20**	< 1	**20**	< 1	**20**	< 1	**20**	< 1	**20**	< 1
	econ-orani678	17	632	**19**	614	**19**	504	6	231	**19**	622	**19**	491	**19**	592
	soc-advogato	**20**	< 1	**20**	< 1	**20**	< 1	15	424	**20**	< 1	**20**	< 1	**20**	< 1
	bio-dmela	**20**	< 1	**20**	< 1	**20**	< 1	**20**	< 1	**20**	< 1	**20**	< 1	**20**	< 1
	ia-escorts-dynamic	**20**	< 1	**20**	< 1	**20**	< 1	**20**	< 1	**20**	< 1	**20**	< 1	**20**	< 1
	soc-anybeat	**20**	1	**20**	1	**20**	1	**20**	4	**20**	1	**20**	1	**20**	1
	ca-AstroPh	**20**	98	**20**	93	**20**	40	15	497	**20**	73	**20**	27	**20**	47
	fb-pages-media	**20**	3	**20**	3	**20**	2	**20**	2	**20**	3	**20**	2	**20**	2
	soc-gemsec-RO	**20**	1	**20**	1	**20**	1	**20**	1	**20**	1	**20**	1	**20**	1
	soc-gemsec-HU	**20**	1	**20**	1	**20**	1	**20**	1	**20**	1	**20**	1	**20**	1
	fb-pages-artist	19	417	19	414	**20**	257	14	271	**20**	722	**20**	196	**20**	408
k-Medoid	skewed-sampled	**6**	248	**6**	219	**6**	231	4	95	**6**	815	**6**	235	**6**	239
	asymmetric-sampled	**5**	24	**5**	23	**5**	23	4	92	**5**	41	**5**	23	**5**	23
	overlap-sampled	**6**	148	**6**	135	**6**	149	4	94	**6**	563	**6**	154	**6**	155
	dim032-sampled	6	53	**7**	867	6	40	4	105	6	286	6	41	6	44
	a1-sampled	**3**	18	**3**	18	**3**	18	**3**	119	**3**	43	**3**	18	**3**	18
	s1-sampled	**3**	172	**3**	170	**3**	176	2	1	**3**	380	**3**	170	**3**	174
	s2-sampled	**3**	142	**3**	140	**3**	145	2	1	**3**	353	**3**	140	**3**	140
	a2-sampled	**3**	179	**3**	181	**3**	183	2	2	**3**	420	**3**	179	**3**	183
	unbalance2-sampled	**3**	439	**3**	438	**3**	439	2	4	**3**	813	**3**	446	**3**	439
	a3-sampled	**3**	772	**3**	771	**3**	770	2	6	2	6	**3**	781	**3**	769

Table 17: Table holding the running times for the Basic Search algorithm `B` and the Lazy Evaluation configurations. Numbers marked in bold denote the greatest k for which the instance could be solved. The reported time is the seconds it takes for the specific instance with the specific k.

	Dataset Name	LE		LE + M Sqrt(n)		LE + M k'		LE + M 3k'	
		k	seconds	k	seconds	k	seconds	k	seconds
Group Close. Centr.	ca-netscience	**14**	867	12	857	13	707	11	540
	soc-wiki-Vote	**13**	761	11	505	12	385	11	369
	bio-yeast	**11**	460	10	544	**11**	553	10	222
	econ-orani678	**15**	529	12	406	**15**	713	13	482
	soc-advogato	**13**	767	11	499	**13**	804	12	491
	bio-dmela	**9**	637	8	603	**9**	682	**9**	760
	ia-escorts-dynamic	**8**	628	7	529	**8**	655	**8**	682
	soc-anybeat	**8**	779	5	441	**8**	770	**8**	834
	ca-AstroPh	**1**	< 1	**1**	< 1	**1**	< 1	**1**	< 1
	fb-pages-media	**1**	< 1	**1**	< 1	**1**	< 1	**1**	< 1
	soc-gemsec-RO	**1**	1	**1**	1	**1**	1.55	**1**	1
	soc-gemsec-HU	**20**	161	**20**	162	**20**	164.30	**20**	166
	fb-pages-artist	**20**	190	**20**	190	**20**	188.90	**20**	193
Part. Dom. Set	ca-netscience	**20**	< 1	**20**	< 1	**20**	< 1	**20**	2
	soc-wiki-Vote	**20**	< 1	**20**	1	**20**	< 1	**20**	4
	bio-yeast	**20**	< 1	**20**	< 1	**20**	< 1	**20**	< 1
	econ-orani678	**19**	491	16	839	18	433.25	16	463
	soc-advogato	**20**	< 1	**20**	8	**20**	< 1	**20**	3
	bio-dmela	**20**	< 1	**20**	17	**20**	< 1	**20**	3
	ia-escorts-dynamic	**20**	< 1	**20**	10	**20**	< 1	**20**	1
	soc-anybeat	**20**	1	**20**	2	**20**	1	**20**	1
	ca-AstroPh	**20**	27	16	773	**20**	87	**20**	599
	fb-pages-media	**20**	2	**20**	107	**20**	2	**20**	5
	soc-gemsec-RO	**20**	1	**20**	5	**20**	1	**20**	1
	soc-gemsec-HU	**20**	1	**20**	19	**20**	1	**20**	2
	fb-pages-artist	**20**	196	13	772	**20**	416	19	888
k-Medoid	skewed-sampled	**6**	235	**6**	429	**6**	254	**6**	308
	asymmetric-sampled	**5**	23	**5**	44	**5**	25	**5**	31
	overlap-sampled	**6**	154	**6**	288	**6**	165	**6**	202
	dim032-sampled	**6**	41	**6**	68	**6**	44	**6**	53
	a1-sampled	**3**	18	**3**	24	**3**	18	**3**	19
	s1-sampled	**3**	170	**3**	208	**3**	173	**3**	174
	s2-sampled	**3**	140	**3**	167	**3**	143	**3**	143
	a2-sampled	**3**	179	**3**	218	**3**	182	**3**	187
	unbalance2-sampled	**3**	446	**3**	531	**3**	440	**3**	447
	a3-sampled	**3**	781	**3**	888	**3**	770	**3**	808

Table 18: Table holding the running times for the LE algorithm and three of the UB2D heuristics. The used configurations are always the Maximum Weight mathcing with $\ell = \sqrt{n}, k', 3k'$ with $k' = k - |S_T|$. Numbers marked in bold denote the greatest k for which the instance could be solved. The reported time is the seconds it takes for the specific instance with the specific k. The LE algorithm is the fastest one for all instances.

	Dataset Name	LE		LE + M 2		LE + M 6		LE + M 10		LE + M 20	
		k	seconds	k	seconds	k	seconds	k	seconds	k	seconds
Group Close. Centr.	ca-netscience	**14**	867	**14**	869	13	411	13	782	11	381
	soc-wiki-Vote	**13**	761	**13**	755	**13**	766	12	418	12	818
	bio-yeast	**11**	460	**11**	466	**11**	504	**11**	600	10	213
	econ-orani678	**15**	529	**15**	535	**15**	561	**15**	644	14	545
	soc-advogato	**13**	767	**13**	751	**13**	759	**13**	799	**13**	871
	bio-dmela	**9**	637	**9**	644	**9**	653	**9**	655	**9**	722
	ia-escorts-dynamic	**8**	628	**8**	685	**8**	656	**8**	686	**8**	701
	soc-anybeat	**8**	779	**8**	782	**8**	749	**8**	732	**8**	771
	ca-AstroPh	**1**	< 1	**1**	< 1	**1**	< 1	**1**	< 1	**1**	< 1
	fb-pages-media	**1**	< 1	**1**	< 1	**1**	< 1	**1**	< 1	**1**	< 1
	soc-gemsec-RO	**1**	1	**1**	1	**1**	1	**1**	1	**1**	1
	soc-gemsec-HU	**20**	161	**20**	163	**20**	163	**20**	161	**20**	159
	fb-pages-artist	**20**	190	**20**	194	**20**	189	**20**	190	**20**	188
Part. Dom. Set	ca-netscience	**20**	< 1	**20**	< 1	**20**	< 1	**20**	< 1	**20**	< 1
	soc-wiki-Vote	**20**	< 1	**20**	< 1	**20**	< 1	**20**	< 1	**20**	< 1
	bio-yeast	**20**	< 1	**20**	< 1	**20**	< 1	**20**	< 1	**20**	< 1
	econ-orani678	**19**	491	**19**	505	**19**	540	**19**	679	18	584
	soc-advogato	**20**	< 1	**20**	< 1	**20**	< 1	**20**	< 1	**20**	< 1
	bio-dmela	**20**	< 1	**20**	< 1	**20**	< 1	**20**	< 1	**20**	< 1
	ia-escorts-dynamic	**20**	< 1	**20**	< 1	**20**	< 1	**20**	< 1	**20**	< 1
	soc-anybeat	**20**	1	**20**	1	**20**	1	**20**	1	**20**	1
	ca-AstroPh	**20**	27	**20**	27	**20**	33	**20**	51	**20**	147
	fb-pages-media	**20**	2	**20**	2	**20**	2	**20**	2	**20**	2
	soc-gemsec-RO	**20**	1	**20**	1	**20**	1	**20**	1	**20**	1
	soc-gemsec-HU	**20**	1	**20**	1	**20**	1	**20**	1	**20**	1
	fb-pages-artist	**20**	196	**20**	196	**20**	225	**20**	303	**20**	668
k-Medoid	skewed-sampled	**6**	235	**6**	251	**6**	299	**6**	349	**6**	472
	asymmetric-sampled	**5**	23	**5**	25	**5**	30	**5**	35	**5**	47
	overlap-sampled	**6**	154	**6**	162	**6**	193	**6**	227	**6**	318
	dim032-sampled	**6**	41	**6**	43	**6**	49	**6**	55	**6**	74
	a1-sampled	**3**	18	**3**	18	**3**	19	**3**	20	**3**	21
	s1-sampled	**3**	170	**3**	171	**3**	174	**3**	181	**3**	185
	s2-sampled	**3**	140	**3**	145	**3**	147	**3**	146	**3**	152
	a2-sampled	**3**	179	**3**	180	**3**	185	**3**	188	**3**	197
	unbalance2-sampled	**3**	446	**3**	448	**3**	453	**3**	454	**3**	472
	a3-sampled	**3**	781	**3**	773	**3**	781	**3**	788	**3**	809

Table 19: Table holding the running times for the `LE` algorithm and four of the UB2D heuristics. The used configurations are always the Maximum Weight mathcing with $\ell = 2, 6, 10, 20$. Numbers marked in bold denote the greatest k for which the instance could be solved. The reported time is the seconds it takes for the specific instance with the specific k. We observe that is we increase ℓ the algorithm will get slower, and that no of the heuristics is substantially better than `LE`.

	Dataset Name		`LE`		`LE + (15,2)`		`LE + (10,2)`		`LE + (6,2)`		`LE + (4,2)`
		k	seconds	k	seconds	k	seconds	k	seconds	k	seconds
Group Close. Centr.	ca-netscience	**14**	867	13	523	13	443	**14**	839	**14**	798
	soc-wiki-Vote	**13**	761	12	336	**13**	810	**13**	748	**13**	739
	bio-yeast	**11**	460	**11**	528	**11**	497	**11**	473	**11**	467
	econ-orani678	15	529	15	551	15	530	**16**	899	15	524
	soc-advogato	**13**	767	**13**	763	**13**	757	**13**	742	**13**	749
	bio-dmela	**9**	637	**9**	700	**9**	691	**9**	665	**9**	667
	ia-escorts-dynamic	**8**	628	**8**	673	**8**	655	**8**	665	**8**	669
	soc-anybeat	**8**	779	**8**	769	**8**	757	**8**	782	**8**	779
	ca-AstroPh	**1**	< 1	**1**	< 1	**1**	< 1	**1**	< 1	**1**	< 1
	fb-pages-media	**1**	< 1	**1**	< 1	**1**	< 1	**1**	< 1	**1**	< 1
	soc-gemsec-RO	**1**	1	**1**	1	**1**	1	**1**	1	**1**	1
	soc-gemsec-HU	**20**	161	**20**	159	**20**	161	**20**	161	**20**	165
	fb-pages-artist	**20**	190	**20**	203	**20**	192	**20**	194	**20**	202
Part. Dom. Set	ca-netscience	**20**	< 1	**20**	< 1	**20**	< 1	**20**	< 1	**20**	< 1
	soc-wiki-Vote	**20**	< 1	**20**	< 1	**20**	< 1	**20**	< 1	**20**	< 1
	bio-yeast	**20**	< 1	**20**	< 1	**20**	< 1	**20**	< 1	**20**	< 1
	econ-orani678	**19**	491	**19**	555	**19**	501	**19**	483	**19**	480
	soc-advogato	**20**	< 1	**20**	< 1	**20**	< 1	**20**	< 1	**20**	< 1
	bio-dmela	**20**	< 1	**20**	< 1	**20**	< 1	**20**	< 1	**20**	< 1
	ia-escorts-dynamic	**20**	< 1	**20**	< 1	**20**	< 1	**20**	< 1	**20**	< 1
	soc-anybeat	**20**	1	**20**	1	**20**	1	**20**	1	**20**	1
	ca-AstroPh	**20**	27	**20**	32	**20**	30	**20**	28	**20**	28
	fb-pages-media	**20**	2	**20**	2	**20**	2	**20**	2	**20**	2
	soc-gemsec-RO	**20**	1	**20**	1	**20**	1	**20**	1	**20**	1
	soc-gemsec-HU	**20**	1	**20**	1	**20**	1	**20**	1	**20**	1
	fb-pages-artist	**20**	196	**20**	213	**20**	204	**20**	197	**20**	195
k-Medoid	skewed-sampled	**6**	235	**6**	282	**6**	273	**6**	265	**6**	259
	asymmetric-sampled	**5**	23	**5**	28	**5**	27	**5**	26	**5**	26
	overlap-sampled	**6**	154	**6**	186	**6**	179	**6**	175	**6**	167
	dim032-sampled	**6**	41	**6**	48	**6**	46	**6**	45	**6**	44
	a1-sampled	**3**	18	**3**	19	**3**	18	**3**	18	**3**	19
	s1-sampled	**3**	170	**3**	171	**3**	176	**3**	174	**3**	173
	s2-sampled	**3**	140	**3**	141	**3**	143	**3**	141	**3**	143
	a2-sampled	**3**	179	**3**	182	**3**	180	**3**	182	**3**	180
	unbalance2-sampled	**3**	446	**3**	450	**3**	452	**3**	441	**3**	446
	a3-sampled	**3**	781	**3**	772	**3**	774	**3**	796	**3**	777

Table 20: Table holding the running times for the `LE` algorithm and four (η, λ) PBF configurations. Each block has size two, only the number of blocks changes. Numbers marked in bold denote the greatest k for which the instance could be solved. The reported time is the seconds it takes for the specific instance with the specific k. We observe that is we increase ℓ the algorithm will get slower, and that no of the heuristics is substantially better than `LE`.

	Dataset Name	LE		LE + (6,5)		LE + (5,5)		LE + (1,10)		LE + (1,20)	
		k	seconds	k	seconds	k	seconds	k	seconds	k	seconds
Group Close. Centr.	ca-netscience	**14**	867	11	519	12	844	10	620	10	621
	soc-wiki-Vote	**13**	761	11	604	11	540	10	586	10	585
	bio-yeast	**11**	460	10	644	10	578	9	276	9	267
	econ-orani678	**15**	529	13	728	13	663	10	660	10	653
	soc-advogato	**13**	767	12	809	12	801	9	494	9	487
	bio-dmela	**9**	637	8	455	8	430	7	233	7	230
	ia-escorts-dynamic	**8**	628	7	379	7	351	7	457	7	462
	soc-anybeat	**8**	779	7	791	7	735	7	874	7	878
	ca-AstroPh	**1**	< 1	**1**	< 1	**1**	< 1	**1**	< 1	**1**	< 1
	fb-pages-media	**1**	< 1	**1**	< 1	**1**	< 1	**1**	< 1	**1**	< 1
	soc-gemsec-RO	**1**	1	**1**	1	**1**	1	**1**	1	**1**	1
	soc-gemsec-HU	**20**	161	**20**	162	**20**	163	**20**	162	**20**	161
	fb-pages-artist	**20**	190	**20**	193	**20**	191	**20**	188	**20**	192
Part. Dom. Set	ca-netscience	**20**	< 1	**20**	< 1	**20**	< 1	**20**	< 1	**20**	15
	soc-wiki-Vote	**20**	< 1	**20**	< 1	**20**	< 1	**20**	< 1	**20**	79
	bio-yeast	**20**	< 1	**20**	< 1	**20**	< 1	**20**	< 1	**20**	1
	econ-orani678	**19**	491	**19**	840	**19**	811	14	891	13	590
	soc-advogato	**20**	< 1	**20**	< 1	**20**	< 1	**20**	1	**20**	205
	bio-dmela	**20**	< 1	**20**	< 1	**20**	< 1	**20**	< 1	**20**	39
	ia-escorts-dynamic	**20**	< 1	**20**	< 1	**20**	< 1	**20**	< 1	**20**	29
	soc-anybeat	**20**	1	**20**	1	**20**	1	**20**	1	**20**	16
	ca-AstroPh	**20**	27	**20**	63	**20**	57	**20**	359	17	422
	fb-pages-media	**20**	2	**20**	2	**20**	2	**20**	3	**20**	117
	soc-gemsec-RO	**20**	1	**20**	1	**20**	1	**20**	1	**20**	12
	soc-gemsec-HU	**20**	1	**20**	1	**20**	1	**20**	1	**20**	18
	fb-pages-artist	**20**	196	**20**	362	**20**	334	18	493	16	593
k-Medoid	skewed-sampled	**6**	235	**6**	290	**6**	287	**6**	266	**6**	264
	asymmetric-sampled	**5**	23	**5**	27	**5**	27	**5**	26	**5**	26
	overlap-sampled	**6**	154	**6**	196	**6**	193	**6**	175	**6**	174
	dim032-sampled	**6**	41	**6**	56	**6**	54	**6**	48	**6**	48
	a1-sampled	**3**	18	**3**	18	**3**	19	**3**	18	**3**	18
	s1-sampled	**3**	170	**3**	171	**3**	171	**3**	171	**3**	171
	s2-sampled	**3**	140	**3**	141	**3**	141	**3**	143	**3**	144
	a2-sampled	**3**	179	**3**	182	**3**	183	**3**	180	**3**	183
	unbalance2-sampled	**3**	446	**3**	453	**3**	449	**3**	441	**3**	454
	a3-sampled	**3**	781	**3**	771	**3**	772	**3**	787	**3**	785

Table 21: Table holding the running times for the LE algorithm and four (η, λ) PBF configurations. Numbers marked in bold denote the greatest k for which the instance could be solved. The reported time is the seconds it takes for the specific instance with the specific k. As we increase λ the algorithm gets slower.

Selbständigkeitserklärung

Ich erkläre, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe. Seitens des Verfassers bestehen keine Einwände die vorliegende Masterarbeit für die öffentliche Benutzung im Universitätsarchiv zur Verfügung zu stellen.

Jena, 20. November 2023, Henning Martin Woydt