DEPARTMENT OF COMPUTER SCIENCE

Practical algorithms for Set Cover beyond Greedy

Răzvan Dan David

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Bachelor of Science in the Faculty of Engineering.

13th May 2021

Declaration

This dissertation is submitted to the University of Bristol in accordance with the requirements of the degree of BSc in the Faculty of Engineering. It has not been submitted for any other degree or diploma of any examining body. Except where specifically acknowledged, it is all the work of the Author.

This project did not require ethical review, as determined by my supervisor Christian Konrad.

Contents

List of Figures

List of Tables

List of Algorithms

Executive Summary

Set Cover is the problem of selecting the minimum number of sets from an initial collection such that all elements in some universe appear in at least one of the selected sets. While this problem is well-known and has many practical applications, it is also NP-complete. There is however a widely used greedy algorithm that works by repeatedly adding the set that would cover the largest number of elements. This algorithm has an approximation factor that is very close to the theoretical maximum achievable in polynomial time. We aim to improve on the practical performance of the greedy algorithm using heuristics, and to design and implement a directed search algorithm that we can use to solve moderately sized Set Cover instances, and gain insights into the structure of the problem at hand.

- I proposed four improvements to the original greedy algorithm that make it behave better on practical instances. I designed a directed search algorithm that produces optimal solutions for the Set Cover problem in exponential time, and used it to optimally solve a number of practical instances.
- I efficiently implemented the approximation algorithm that results from applying the four improvements to the base greedy algorithm, and the new search algorithm in C++.
- I performed an in-depth empirical analysis of the two algorithms on 16 practical Set Cover instances.
- I wrote more than 3000 lines of source code, which include the final implementation of the presented algorithms, and a less efficient initial version.

This thesis describes these strands of work, their results and their implications.

Supporting technologies

I used C++20 to implement the algorithms described in this thesis, and made use of functionality and containers provided by the C++ standard library. I used Edmond's maximum cardinality matching algorithm from the Boost Graph Library in the implementation of the reduction that optimally solves Set Cover instances with sets of size 2. To detect memory corruption resulting from raw pointer use, I used the AddressSanitizer open source tool.

Chapter 1

Introduction

Given some sets that contain elements from a universe, Set Cover is the problem of determining a collection of those sets, which *covers* the universe and is also of minimum size. A collection of sets is said to cover a universe of elements if all the elements in that universe are contained in at least one of the sets in the collection. While conceptually simple, Set Cover is a fundamental problem in combinatorics, which arises in many real world scenarios, including crew scheduling [2], data mining sequenced DNA for repeats [3] and information retrieval [4].

A Dominating Set for a graph is a collection of vertices such that every vertex in the graph is either in that collection, or some vertex adjacent to it is. There are linear reductions between the Set Cover and the minimum Dominating Set problems [5], meaning that an instance of one of the problems can be transformed into an instance of the other problem in polynomial time. For example, the Domainting Set instance shown in Figure 1.1 can be transformed into the Set Cover instance shown in Figure 1.2. In both cases, a solution to the problem is marked in red: vertex 0 and 3 for the graph version, and sets $\{0, 1, 4\}$ and $\{3, 2, 4\}$ for the set based formulation.

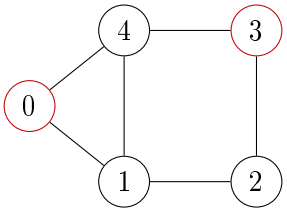

Figure 1.1: A Dominating Set instance.

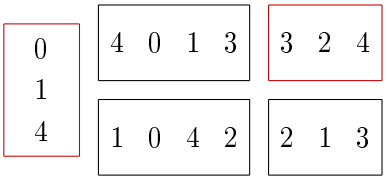

Figure 1.2: A Set Cover instance.

Given the simplicity and highly abstract nature of the Set Cover problem formulation, and also the fact that Set Cover and minimum Dominating Set are essentially equivalent, it is easy to see why many real world problems reduce to set covering. For example, in the case of information retrieval, the sets can be thought of as documents containing information on various topics (elements), and the problem consists of retrieving the minimum number of documents that cover a certain universe of topics.

Although its definition may be considered simple, optimally solving the Set Cover problem is not easy: the decision version of this problem was shown to be NP-complete [6], and its optimisation version is NP-hard. That being said, there are algorithms that yield an approximate solution in polynomial time, as is also the case for other NP-hard problems. A Greedy [7] approach to constructing an approximate solution works by repeatedly adding the set with the current maximum number of uncovered elements to the solution, until all elements in the universe are covered. This algorithm runs in polynomial time, is not complicated to implement in practice, and results in an approximation factor close to the theoretical maximum for the Set Cover problem [8, 9].

The approximation factor of the Greedy algorithm cannot be significantly improved on in theory, and this algorithm has been shown to also perform quite well in practice when compared to other approximation algorithms [10]. Additionally, Greedy has produced solutions that are, on average, only 7% larger than the optimal size when ran on 30 Set Cover instances [11]. That being said, many practical applications would greatly benefit even from small improvements over the solutions produced by the Greedy algorithm, despite the fact that obtaining these improvements may take exponential time.

This project aims (i) to improve on the performance of Greedy on practical Set Cover instances, (ii) to use exponential search algorithms to solve moderately sized Set Cover instances and (iii) to gain further insight into the structure of the solution space. We will describe changes to the Greedy algorithm that do not improve its theoretical approximation factor but make it behave better in practice, and use the main idea of Greedy to create a directed exponential search algorithm. Then, we will detail the process of efficiently implementing these algorithms in modern C++. Finally, we will run the exponential search algorithm for a limited amount of time on a number of different Set Cover instances, and analyse the results we obtain.

We approached the Set Cover problem from an algorithm engineering approach, and we were able to design an improved approximation algorithm that should produce better solutions than Greedy in practice by using four different heuristics. In addition to this approximation algorithm, we also created a directed search algorithm that we used to optimally solve a number of practical Set Cover instances with up to 7601 sets. In Chapter 2, we formally define the Set Cover problem, formally introduce the Greedy algorithm, and mention the state-of-the-art in both practical approximation algorithms and in theoretical search algorithms. We detail and motivate the improvements we made to the Greedy algorithm, and introduce our directed search algorithm in Chapter 3. In Chapter 4, we present some of the challenges we faced in creating an efficient C++ implementation of both our improved approximation algorithm, and of our exponential search algorithm. Here, we also present an elegant way of modelling the problem in software, and how we used this model to simply and efficiently implement our algorithms. Finally, Chapter 5 contains an empirical analysis of the performance of the heuristics that make up our approximation algorithm, along with an in-depth exploration of the behaviour of our search algorithm in which we present and discuss a number of different metrics recorded during execution.

Chapter 2

Preliminaries

We will now formally define the Set Cover problem, describe the Greedy algorithm, briefly introduce other practical approximation algorithms for this problem, and finally outline how state-of-the-art exponential search algorithms use reductions to obtain better run time complexities.

2.1 Set Cover definition

In order to define the Set Cover problem, we need to consider a universe of elements. Let the universe $\mathcal{U}$ be comprised of n elements, $|\mathcal{U}| = n$. For simplicity, we assume without loss of generality that this universe contains only consecutive integers, $\mathcal{U} = [0 \,..\, n-1]$. Additionally, we need to consider an initial collection of sets $\mathcal{S} = \{S_0, S_1, \ldots S_{m-1}\}$ of size m, where $S_i \subseteq \mathcal{U}$, for all $i \in [0 \,..\, m-1]$. Not only are all elements of every set included in the universe, but the union of the sets from the initial collection actually gives the universe, $\bigcup_{i=0}^{m-1} S_i = \mathcal{U}$.

A cover is defined as a collection of initial sets $\mathcal{C} \subseteq \mathcal{S}$ that also satisfies $\bigcup_{S \in \mathcal{C}} = \mathcal{U}$, meaning that the union of all sets in the cover results in the universe. In practice, it may be easier to refer to the sets in the cover by their indices: we use C to be a collection of indices $C = \{0, 1, \ldots k-1\}$ that refers to the cover $\mathcal{C} = \{S_0, S_1, \ldots S_{k-1}\}$. It is also useful to define M as the total number of elements across all sets, $M = \sum_{S \in \mathcal{S}} |S|$.

Set Cover is the problem of finding a cover of minimum size, that is finding some $\mathcal{C}$ for which $|\mathcal{C}|$ is minimised. This problem was first introduced in 1972, and appeared in Karp's list of 21 NP-complete problems [6]. It was later proved that this problem cannot be approximated within a factor of $(1 - o(1)) \ln n$ of the optimum [9], unless NP contains quasi-polynomial algorithms. The same approximation bound was proved under the weaker assumption of P $\neq$ NP in 2014 [12].

An example instance of the Set Cover problem can be seen in Figure 2.1. In this case, $\mathcal{U} = [0 \,..\, 10]$ with $n = 11$, $\mathcal{S} = \{S_0, S_1, \ldots S_7\}$, and the $m = 8$ sets are: $S_0 = \{4, 1, 5\}$, $S_1 = \{3\}$,

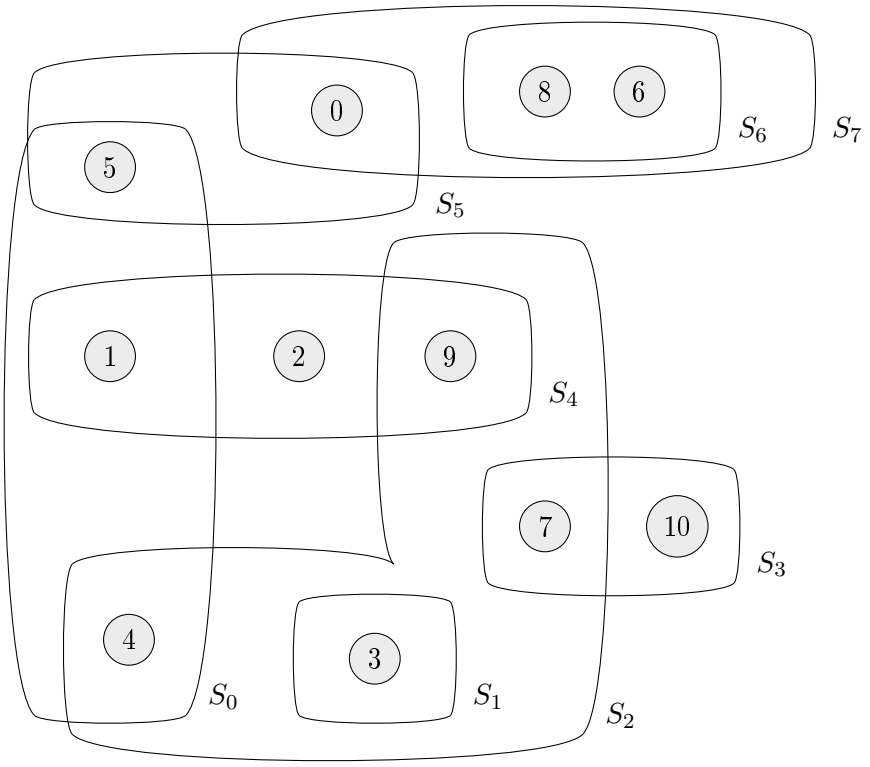

FIGURE 2.1: Example SET COVER instance, with $|\mathcal{U}| = 11$ and $|\mathcal{S}| = 8$.

$S_2 = \{4, 3, 7, 9\}$, $S_3 = \{7, 10\}$, $S_4 = \{1, 2, 9\}$, $S_5 = \{5, 0\}$, $S_6 = \{8, 6\}$, and finally $S_7 = \{0, 8, 6\}$. For any SET COVER instance, including this one, a valid cover is $\mathcal{C} = \mathcal{S}$, but it is easy to see that this cover is not optimal: we can just remove S_1 and it would still be valid. In this particular example, there are two different covers of minimum size, which are the optimal solutions to this problem instance: $\mathcal{C}_0 = \{S_2, S_3, S_4, S_0, S_7\}$ and $\mathcal{C}_1 = \{S_2, S_3, S_4, S_5, S_6\}$.

2.2 The GREEDY algorithm

The GREEDY approximation algorithm for the SET COVER problem is based on the simple idea of repeatedly adding the set that contains the most uncovered elements to the cover until all elements in the universe are covered. We give pseudocode for it in Algorithm 2.1.

The GREEDY algorithm was first introduced in 1974 [7], but a tight analysis of its approximation ratio was published only in 1996 [8]. This tight analysis showed that it had an approximation ratio of $\ln n - \ln \ln n + \Theta(1)$, making it the best known polynomial time approximation algorithm from a theoretical standpoint, and leaving very little room for improvement, as the best approximation factor achievable in polynomial time is $(1 - o(1)) \ln n$ [9].

This algorithm can be implemented to run in $O(M)$ time, and such implementation will be detailed in a later chapter of this thesis. That being said, for very large instances of the SET COVER problem, even this may be too slow. There are lazy update techniques that result in improved practical performance, but also in worse theoretical complexity [13].

Algorithm 2.1: GREEDY algorithm

input : An instance of the SET COVER problem: a collection of sets $\mathcal{S} = \{S_0, S_1, \ldots S_{m-1}\}$, such that $\mathcal{U} = \bigcup_{S \in \mathcal{S}}$.

output: A set C of indices that corresponds to an approximate solution $\mathcal{C}$ for the initial problem instance.

```
1 C ← {}
2 CE ← {}
  // CE is a set containing covered elements
3 while CE ≠ U do
4 |   i ← arg max_{k∈[0..m−1]} |S_k \ CE|
5 |   C ← C ∪ {i}
6 |   CE ← CE ∪ S_i
7 return C
```

2.3 Other practical algorithms

One practical disadvantage of the GREEDY algorithm, which is mostly evident on very large data sets, is that it accesses memory in an unpredictable, random way. Although this may also cause slowdowns when the SET COVER instance fits in main memory due to frequent cache misses, very large instances may need to be stored on disk, where randomly accessing data is even more costly.

A paper from 2010 introduces DISK FRIENDLY GREEDY [14] and experimentally shows that it performs on par with the normal GREEDY algorithm, but results in more than a 10x speedup on very large, disk residing instances. There is also a parallel and I/O efficient algorithm [15], which offers similar approximation ratios as simple GREEDY, and was also shown to perform well in practice.

The SET COVER problem was also tackled under a semi-streaming approach, in which the input sets are processed one by one, and the space available to the algorithm is limited. A one-pass semi-streaming algorithm was introduced by Emek and Rosén [16]. Konrad et al. [17] analysed this algorithm on practical instances, and found that it produces covers that are within 8% of those produced by the DISK FRIENDLY GREEDY algorithm, while using at least 10 times less memory.

2.4 Exponential search algorithms

There is a very simple naive search algorithm for the SET COVER problem that produces an exact solution: exhaustively try either including or excluding all sets from the cover, and return the smallest valid combination. This runs in $O(2^m)$, and perhaps surprisingly, no other algorithm was proved to achieve better complexity until 2004 [18].

To further improve on the naive algorithm, a common approach is to use branch and reduce conditions. The reduction conditions aim to simplify a Set Cover instance into an equivalent but computationally easier to solve instance. The branch conditions apply when no reduction can be made, and dictate how the algorithm recurses. An exact algorithm based on branch and reduce rules is given by Rooij et al. [19] in 2011, and proven to have a run time complexity of $O(1.4969^m)$, while using polynomial space. It is quite difficult to analyse the run time of exponential algorithms, but by using another analysis method, this complexity was slightly improved to $O(1.4864^m)$ [20].

We will now briefly present some notable reductions from the algorithm presented by Rooij et al. [19].

- The first reduction involves always including the sets that contain a unique element, and that by definition are always present in a valid cover. This simple first reduction results in improved run time complexity: from a naive $O(2^m)$ to $O(1.7311^m)$.
- The second reduction is based on the observation that there is no need to branch when all remaining sets have size 1. Instead, we can simply include one such set for all uncovered elements. This reduction results in further improvements to the complexity of the presented algorithm, bringing said complexity to $O(1.6411^m)$.
- The third notable reduction follows from the fact that we will never take some set A to be in the cover if another set B exists such that $A \subseteq B$. This intuitive reduction involves removing all sets for which there is a superset, and brings the complexity down to $O(1.5709^m)$.
- Finally, in the case in which all remaining sets have cardinality at most 2, we can obtain a partial cover for the elements in these sets in polynomial time. This is done by reducing the problem to maximum cardinality matching in a general graph, and results in a further improvement to the complexity of the algorithm, to $O(1.5169^m)$.

There are more reductions presented in the Rooij et al. paper than just the four we briefly mentioned here, but they result in relatively negligible complexity improvements and are significantly more conceptually involved. This drop off in complexity improvement may be a result of the fact that the Set Cover cover is NP-hard even for sets of maximum size 3 [21], so there are no other shortcuts available that are as straightforward as for example the above-mentioned reduction to maximum cardinality matching.

Chapter 3

The algorithm

We created an exponential search algorithm for the Set Cover problem that is based on an improved version of the Greedy algorithm introduced previously. In this chapter, we will first describe and motivate the improvements we propose over Greedy, and then present our search algorithm.

To allow for a better explanation of certain parts of the algorithm, we introduce the notion of a *problem state*, which we write $P = (P_{\mathcal{C}}, P_{\mathcal{S}})$. For some Set Cover instance defined by an initial set collection $\mathcal{S}$ and element universe $\mathcal{U}$, $P_{\mathcal{C}}$ is a collection of sets that represents a partial cover. In the same context, $P_{\mathcal{S}}$ is a collection of initial sets that have not been added to the partial cover, but are still being considered. It is not required that $P_{\mathcal{S}} = \mathcal{S} \setminus P_{\mathcal{C}}$, which is to say we can simply no longer consider some set (remove it from $P_{\mathcal{S}}$), without actually adding it to the partial cover $P_{\mathcal{C}}$.

We can get (transition) from some state P_0 to some state P_1 by doing some computation. This computation may involve adding sets to the partial cover, or simply removing sets from the remaining set collection. For $P_{\mathcal{C}}$ to be a valid cover for the problem, $P_{\mathcal{C}} = \mathcal{U}$ needs to hold. If some $P_{\mathcal{C}}$ is a valid cover, we call the state $P = (P_{\mathcal{C}}, P_{\mathcal{S}})$ *solved*. Not all states $P = (P_{\mathcal{C}}, P_{\mathcal{S}})$ can yield a valid cover, even if computation proceeds to some state $P_{all} = (P_{\mathcal{C}} \cup P_{\mathcal{S}}, P_{\mathcal{S}_{all}})$ by adding all remaining sets to the partial cover. This is because we can remove sets from the remaining set collection $P_{\mathcal{S}}$, without adding them to the partial cover $P_{\mathcal{C}}$. We say that a state is *valid* if it can yield a valid cover $P_{\mathcal{C}}$ by computation, and *invalid* otherwise.

When we say we *optimally solve* some valid state $P = (P_{\mathcal{C}}, P_{\mathcal{S}})$, we mean that computation continues by adding the minimum amount of sets $S \in P_{\mathcal{S}}$ to $P_{\mathcal{C}}$ in order to get to a solved state. We say that we *optimally transition* from some valid state P_0 to some other valid state P_1 if optimally solving P_0 would yield a cover of the same size as optimally solving P_1. We also define the *covered elements* of a state to refer to a set of elements $P_{covered} = \bigcup_{P_{\mathcal{C}}}$.

3.1 Reduction to matchings

As previously mentioned, optimally solving a Set Cover instance that contains sets of size greater than 2 cannot be done in polynomial time. On the other hand, if we consider an instance where all sets are of size 1, we find it to be solvable in polynomial time in a trivial manner: simply add one set for all uncovered elements. In fact, given some valid problem state $P = (P_{\mathcal{C}}, P_{\mathcal{S}})$ with $\forall S \in P_{\mathcal{S}},\ |S \setminus P_{covered}| \leq 2$, we can simply ignore the sets S with $|S \setminus P_{covered}| = 1$, optimally solve a new instance defined by the set collection $\mathcal{S}' = \{S \setminus P_{covered} \mid S \in P_{\mathcal{S}},\ |S \setminus P_{covered}| = 2\}$, and then complete the resulting cover with one previously ignored set for all the remaining uncovered elements.

3.1.1 Motivation

Optimally solving this subcase of the problem may seem as a small improvement because we intuitively and correctly expect a good majority of sets to initially contain more than just 2 elements, in the case of practical instances. We ran Greedy on a Set Cover instance with $n = 16470$ and $m = 88162$, for which only about 10% of the initial sets contained 1 or 2 elements, and we recorded how many sets contributed with 1 up to 10 new elements at the time they were added to the cover.

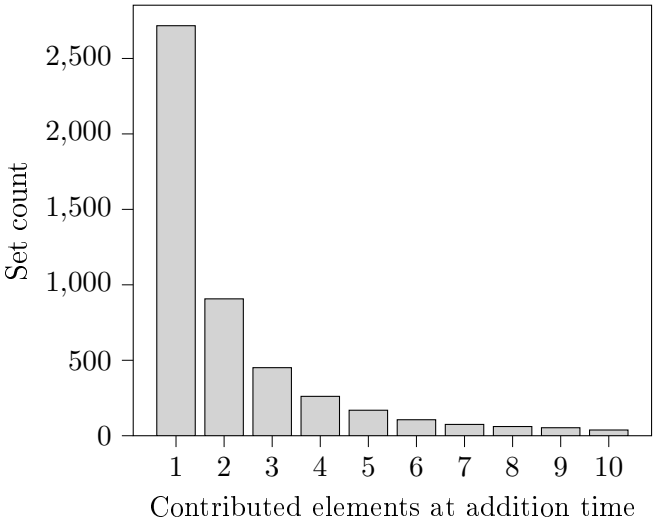

Figure 3.1: New element contributions for sets, before adding to greedy solution. Data from the *retail* instance, obtained from the Frequent Itemset Mining Dataset Repository [1]

Figure 3.1 plots the results we obtained, and it is easy to observe that the number of sets rapidly declines as the number of contributed elements increases. More precisely, we found that just over 70% of the sets that ended up in the final cover only contributed with 1 or 2 new elements when they were added. This means that for any of these sets, if $P_0 = (P_{\mathcal{C}_0}, P_{\mathcal{S}_0})$ is the state of the Greedy algorithm right before adding set S_i to the cover, and $P_1 = (P_{\mathcal{C}_0} \cup \{S_i\}, P_{\mathcal{S}_1})$ is the state of this algorithm right after, $|P_{1covered}| - |P_{0covered}| \leq 2$. As these sets represent a

very significant portion of the final cover, it should now be clear why we give such importance to solving this subcase optimally.

3.1.2 Graph matching and edge cover

A matching in an undirected graph $G = (V, E)$ is a set of edges $M \subseteq E$ such that no two edges in this set have a common vertex.

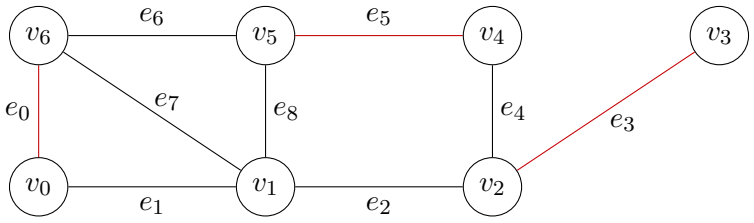

FIGURE 3.2: An undirected graph with 7 vertices and 9 edges.

The graph in Figure 3.2 is $G = (\{v_0, v_1, \dots v_6\}, \{e_0, e_1, \dots e_8\})$. In this graph, examples of valid matchings are $M_0 = \{e_0, e_2\}$ and $M_1 = \{e_7, e_5\}$. A set that is not considered a matching is $X = \{e_3, e_4, e_1\}$.

A maximum cardinality matching is a matching M that contains the maximum number of edges: there is no other matching M' with $|M'| > |M|$. There can be multiple maximum cardinality matchings for some graph. For the example graph in Figure 3.2, a maximum cardinality matching is $M_2 = \{e_0, e_3, e_5\}$, and the three edges it contains are coloured red. Edmonds' algorithm [22] computes a maximum cardinality matching for a general undirected graph in time $O(|V||E|)$.

An edge cover in a graph $G = (V, E)$ is also a set of edges $EC \subseteq E$, with the property that all vertices in V are incident to at least one edge in EC. This means that for all vertices in V, there is at least one edge in EC that connects this vertex to some other vertex. In the example graph, a valid edge cover is $EC = \{e_1, e_2, e_3, e_5, e_6\}$.

A minimum edge cover EC is an edge cover of minimum size, for which there is no other edge cover EC' with $|EC'| > |EC|$. The problem of finding an edge cover of minimum size can be solved in polynomial time by simply finding a maximum cardinality matching in the same graph, and then greedily adding edges to this matching until all vertices are covered.

3.1.3 Solving a SET COVER instance with sets of size 2

We illustrate the reduction to maximum cardinality matching with a simple example. Consider the SET COVER instance in Figure 3.3. All three sets are of size two, so the GREEDY algorithm may pick S_1 first, and thus arrive at a cover of size three, which in this case is not of minimum size. Figure 3.4 shows the graph that we construct from this SET COVER instance by simply taking a vertex for each element, and an edge for each set. The maximum cardinality matching

for this graph is $\{e_2, e_0\}$, and it corresponds to the cover of minimum size for the instance we are considering, $\{S_2, S_0\}$.

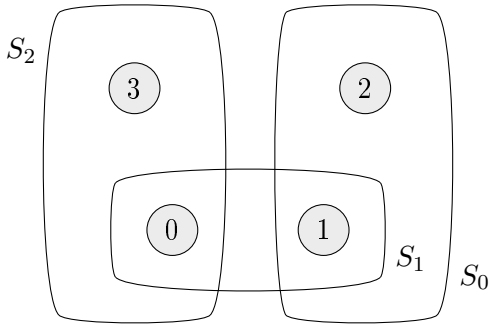

FIGURE 3.3: A simple SET COVER instance with $n = 4$ and $m = 3$.

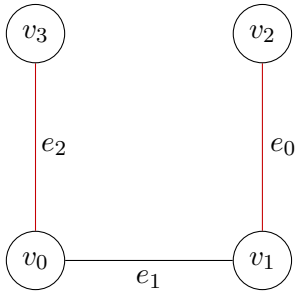

FIGURE 3.4: An undirected graph with 4 vertices and 3 edges.

Formally, we actually reduce a SET COVER problem with element universe $\mathcal{U}$ and initial sets $\mathcal{S}$ with $\forall S \in \mathcal{S},\ |S| = 2$ to a minimum edge cover in an undirected graph $G = (V, E)$, where $V = \{v \mid v \in \mathcal{U}\}$ and $E = \{\{a, b\} \mid \{a, b\} \in \mathcal{S}\}$. The minimum edge cover problem is solved by first computing a maximum cardinality matching using Edmonds' algorithm, and then adding one edge for every vertex that is not already in the edge cover, all in polynomial time. Equivalently, we could have taken only the maximum cardinality matching result, and then trivially added one set for all uncovered elements, as we exemplify below.

We now give another example, in which the maximum cardinality matching does not correspond directly to a minimum cover. Consider the SET COVER instance with $\mathcal{U} = [0\ ..\ 6]$ and $\mathcal{S} = \{S_0, S_1, \ldots S_8\}$. The 9 sets in this instance are $\{0, 6\}, \{0, 1\}, \{1, 2\}, \{2, 3\}, \{2, 4\}, \{4, 5\}, \{5, 6\}, \{1, 6\}$ and $\{1, 5\}$. Observe that the graph in Figure 3.2 is actually constructed from this instance. To generate an optimal solution, we take the three sets that correspond to the edges in the maximum matching M_2, $\{S_0, S_3, S_5\}$. Although this covers elements $\{0, 2, 3, 4, 5, 6\}$, we observe that element 1 is not covered by any of the three sets. To cover all elements in $\mathcal{U}$, and hence have a valid solution, we must add any of the 4 sets that include this element. Thus, for this particular example, a minimum set cover is $\mathcal{C} = \{S_0, S_3, S_5, S_1\}$.

We give pseudocode for optimally solving a SET COVER instance with sets of cardinality 2 in Algorithm 3.1. We construct a maximum cardinality matching, and then we complete the cover by adding one set for each uncovered element.

Algorithm 3.1: Optimal solving of Set Cover instances with sets of size 2

input : An instance of the Set Cover problem: a collection of sets $\mathcal{S} = \{S_0, S_1, \dots S_{m-1}\}$, such that $\mathcal{U} = \bigcup_{S \in \mathcal{S}}$, and $\forall S \in \mathcal{S},\ |S| = 2$.

output: A set C of indices that corresponds to an exact solution $\mathcal{C}$ for the initial problem instance.

let *indexes* be a map from sets of size 2 to integers
$indexes[S_i] \leftarrow i,\ \forall S_i \in \mathcal{S}$
$G \leftarrow (\{v \mid v \in \mathcal{U}\},\ \{\{a, b\} \mid \{a, b\} \in \mathcal{S}\})$
$MCM \leftarrow$ run Edmonds' algorithm on G to obtain a maximum cardinality matching
$C \leftarrow \{indexes[e] \mid e \in MCM\}$
$CE \leftarrow \bigcup_{i \in C} S_i$
while $CE \neq \mathcal{C}$ **do**
    let i be the index of a set, such that $|S_i \setminus CE| = 1$
    $C \leftarrow C \cup \{i\}$
    $CE \leftarrow CE \cup S_i$
return C

3.1.4 Greedy with matchings

We would now like to integrate the method we described for optimally solving Set Cover instances that only have sets of size 2 with the base Greedy algorithm. We can run the original Greedy algorithm on some instance until any set we would add to the cover would only contribute with one or two new elements to the solution, which means the algorithm would be in a state $P_0 = (P_{\mathcal{C}_0}, P_{\mathcal{S}_0})$ with $\forall S \in P_{\mathcal{S}_0},\ |S \setminus P_{0covered}| \leq 2$. At that point, we create a new Set Cover instance defined by sets $\mathcal{S}' = \{S \setminus P_{0covered} \mid S \in P_{\mathcal{S}_0}\ with\ |S \setminus P_{0covered}| = 2\}$, and optimally solve this with Algorithm 3.1 to produce some cover $\mathcal{C}_2$.

After we have $\mathcal{C}_2$, we need to add the original sets $S \in P_{\mathcal{S}_0}$ from which the sets in $\mathcal{C}_2$ were obtained to the partial cover $P_{\mathcal{C}_0}$ that we have for our main problem instance. After completing this step and thus obtaining some new partial cover $P_{\mathcal{C}_1}$ and some new collection of remaining sets $P_{\mathcal{S}_1} = \{S \mid S \in P_{\mathcal{S}_0}\ with\ |S \setminus \bigcup_{P_{\mathcal{C}_1}}| = 1\}$ representative of some state P_1, we simply add one of the sets in $P_{\mathcal{S}_1}$ for each uncovered element in $\mathcal{U} \setminus P_{1covered}$. This final step results in a solved state $P_2 = (P_{\mathcal{C}_2}, \{\})$, and we can say that we optimally transitioned from state P_0 to state P_2 using the approach we just described.

This method is further illustrated in Algorithm 3.2. Observe that once we are in state P_1, we can simply continue adding sets in a greedy way to optimally transition to P_2.

Algorithm 3.2: GREEDY with matchings

input : An instance of the SET COVER problem: a collection of sets $\mathcal{S} = \{S_0, S_1, \dots S_{m-1}\}$, such that $\mathcal{U} = \bigcup_{S \in \mathcal{S}}$.

output: A set C of indices that corresponds to an approximate solution $\mathcal{C}$ for the initial problem instance.

```
1  C ← {}
2  CE ← {}
3  while CE ≠ U do
4      i ← arg max_{k∈[0..m−1]} |S_k \ CE|
5      if |S_i \ CE| = 2 then
           // Now ∀S_i, |S_i \ CE| ≤ 2. We can construct a new instance with sets of
              cardinality 2, and optimally solve it.
6          S' ← {S_i \ EC | S_i ∈ S, |S_i \ EC| = 2}
7          C_1 ← run Algorithm 3.1 to optimally solve the instance defined by sets S'
8          C_1 ← transform returned S' set indices into indices of sets from S
9          C ← C ∪ C_1
10         CE ← CE ∪ ⋃_{i∈C_1} S_i
11         continue
12     C ← C ∪ {i}
13     CE ← CE ∪ S_i
14 return C
```

3.2 Other improvements to GREEDY

We made three other improvements to the base algorithm, beyond the reduction to maximum cardinality matching described in the previous section. These improvements come from a more practical perspective, as we observed that our algorithm behaved better in practice after applying the changes, but we do not have such a strong theoretical basis for them. We will provide an empirical analysis of the quality of all the improvements we propose in a later chapter.

3.2.1 Unique elements

The first improvement is meant to better handle the case in which some element $e \in \mathcal{U}$ occurs only once in a set $S_e \in \mathcal{S}$. Because of the way the GREEDY algorithm works, this set will always end up in the final cover, but we would instead like to add it to the cover before the main loop of GREEDY.

If there are indeed elements that appear only once in some set, this immediately reduces the problem size by the number of such unique elements, which is significant especially in the context of our exponential search algorithm. This reduction actually improves the complexity of a naive search algorithm from $O(2^m)$ to $O(1.7311^m)$, as seen in [19].

Formally, right before starting the main loop, we have a state $P_0 = \{\{\}, \mathcal{S}\}$. We first transition to $P_1 = (P_{\mathcal{C}_1}, P_{\mathcal{S}_1})$, with $P_{\mathcal{C}_1} = \{S_i \mid \exists e \in S_i \; s.t. \; \nexists S_j \neq S_i \; with \; e \in S_j\}$, and $P_{\mathcal{S}_1} = \{S_i \mid S_i \in$

$\mathcal{S}$ *with* $S_i \setminus \bigcup_{P_{\mathcal{C}_1}} \neq \emptyset\}$, and then we simply run the algorithm as before. A procedure that implements this heuristic has a run time complexity of $O(M)$.

3.2.2 Ignoring subsets

Another improvement we propose is to ignore all sets $A \in \mathcal{S}$ for which there is some $B \in \mathcal{S}$ such that $A \subset B$. We recognise that the base Greedy algorithm almost always adds some set $B \supset A$ to the cover, instead of adding A. An exception to this may sometimes occur in the case in which $B \setminus A \subseteq P_{covered}$, meaning that all elements in $B \setminus A$ were already covered by one or multiple other sets. We implemented this change but found that it resulted in no improvements over the solutions generated by the base Greedy algorithm.

That being said, we observed that applying this change did result in better solutions when combined with a variant of the previous improvement we described. Instead of first adding sets that contain unique elements to the cover, we now propose ignoring all subsets, and only then adding the sets that contain unique elements. From some initial state $P_0 = (\{\}, \mathcal{S})$, we can simply transition to $P_1 = (\{\}, \{S_i \mid S_i \in \mathcal{S},\ \nexists S_j \in \mathcal{S}\ with\ S_i \subset S_j\})$ in order to apply this subset removal reduction. Then, we apply the unique element heuristic we described above, but to this new state P_2, and not to the initial problem instance, as originally described.

The removal of subsets essentially causes the number of sets with unique elements to increase, and as such makes the unique element heuristic we described perform even better in practice. One drawback of this reduction is that there is no efficient way of performing it, and as such it may take significantly more time to execute than the actual Greedy algorithm. With the help of some sorting, we can remove all subsets from some instance in time $O(m * M)$.

3.2.3 Redundant sets

This last improvement is based on the perhaps unintuitive observation that the Greedy algorithm can in fact produce solutions that are redundant, in the sense that there can be some set $S \in \mathcal{C}$ such that $\mathcal{C} \setminus \{S\}$ is still a valid cover.

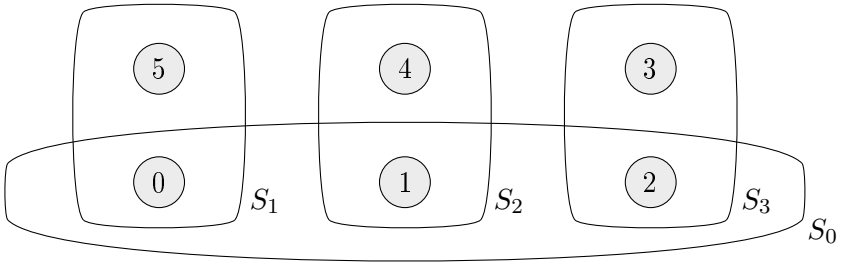

Figure 3.5: A set cover instance with $n = 6$ and $m = 4$.

We illustrate how this may happen by stepping through the execution of Greedy on the instance in Figure 3.5. In this example, the first set to be picked is S_0, because it is the largest. Then,

the algorithm will take S_1, S_2, S_3, in any order. This results in a cover $\mathcal{C} = \{S_0, S_1, S_2, S_3\}$, but it is obvious that we can remove S_3 and the resulting cover would still be valid.

Algorithm 3.3: Removing redundant sets from a cover.

input : A valid set cover $\mathcal{C}$.
output: A valid set cover $\mathcal{C}_f$ with $|\mathcal{C}_f| \leq \mathcal{C}$.

```
1  let app be a 0 initialised array of size ∑_{S∈C} |S|
2  C_f ← {}.
3  for S ∈ C do
4      for e ∈ S do
5          app[e] ← app[e] + 1
6  for S ∈ C do
7      if ∃e ∈ S s.t. app[e] = 1 then
8          for e ∈ S do
9              app[e] ← app[e] − 1
10     else
11         C_f ← C_f ∪ {S}
12 return C_f
```

Optimally removing the maximum number of sets from some cover is in fact another NP-hard combinatorial problem. If it was not, we could just add all sets to a cover and then optimally remove them to solve Set Cover in polynomial time. Instead, we choose to simply iterate over all the sets in the cover, and remove the redundant sets one by one. We give a simple algorithm that does this and runs in $O(\sum_{S \in \mathcal{C}} |S|)$ in Algorithm 3.3.

3.3 Exponential search

By combining the improvements described in the two previous sections, we obtain an algorithm that first reduces the problem by removing all subsets, then includes sets with unique elements to the cover, and afterwards behaves like the original Greedy algorithm until the problem can be optimally solved by reduction to maximum cardinality matchings. Finally, it attempts to remove any redundant sets in a greedy manner, as explained in Algorithm 3.3. We will refer to this polynomial time approximation algorithm as the Improved Greedy algorithm, and we will use it as the base of our directed search algorithm. Improved Greedy actually directly or indirectly performs all four reductions that were mentioned in Section 2.4, and that are the main reductions responsible for obtaining a theoretical search complexity better than that of a naive algorithm in the Rooji et al paper [19].

We would like to design a directed search algorithm that never selects a set that does not contribute with any element to the current partial cover. That is, for some state $P = (P_{\mathcal{C}}, P_{\mathcal{S}})$, our algorithm should never add a set $S \in P_{\mathcal{S}}$ to $P_{\mathcal{C}}$ if this would not result in $|P_{covered}|$ increasing.

Additionally, we would also like to be able to use this algorithm to relatively quickly obtain a solution that is better or at least the same as what Improved Greedy obtains.

To achieve this, our algorithm starts off by finding the exact same approximate solution Improved Greedy would find, and then begins exploring the search space around this solution. More specifically, we branch on the decision of whether or not to include the set that would cover the largest number of new elements if it were to be added to the partial cover. This means that for state $P = (P_{\mathcal{C}}, P_{\mathcal{S}})$ with $S_m = \max_{S \in P_{\mathcal{S}}} |S \setminus P_{covered}|$, we first include set S_m in the partial cover and recurse to $P_i = (P_{\mathcal{C}} \cup \{S_m\}, P_{\mathcal{S}} \setminus \{S_m\})$. Then, we ignore this set and recurse to $P_e = (P_{\mathcal{C}}, P_{\mathcal{S}} \setminus \{S_m\})$ only if P_e is a valid state. P_e may not be valid because S_m contains the last occurrence of some element out of all the remaining sets in $P_{\mathcal{S}}$, and we just removed S_m from the collection of sets we are considering. Not recursing on states that are known to be invalid is key to obtaining good performance on practical Set Cover instances.

When $S_m \setminus P_{covered}$ is of size 2, we reduce the problem to maximum cardinality matching in order to optimally complete the state we are in. This should always be possible, as we only recurse to valid states. Naturally, in the case in which $S_m \setminus P_{covered}$ is of size 1, we just add one set for all uncovered elements to optimally solve the state without recursing, and we should never have $S_m \setminus P_{covered} = \emptyset$.

Before starting to branch, we ignore all sets for which there is a superset in $\mathcal{S}$, and we also add the sets that contain a unique element to the cover. We never continue recursing if the partial cover we have in the current state is already larger than the smallest cover obtained so far by the algorithm. If the state we are in is solved, we greedily remove all redundant sets from the obtained cover $\mathcal{C} = P_{\mathcal{C}}$, as explained in Algorithm 3.3. Note that our algorithm would still arrive at an optimal solution even if it did not try to remove redundant elements, but this increases the chances that it does so sooner. The complete search algorithm is given below in Algorithm 3.4.

3.4 Chapter summary

In this chapter, we detailed how a Set Cover instance with sets of size 2 can be optimally solved by computing a maximum cardinality matching in an undirected graph constructed from this instance, in polynomial time. We explained how this reduction to maximum matching can be combined with the Greedy algorithm, and motivated its significance with empirical evidence obtained from running the base algorithm on a practical instance. We introduced three other improvements: removing subsets, adding sets with unique elements first, and removing redundant sets from the obtained cover. We referred to the algorithm that results from applying these four improvements to Greedy as Improved Greedy. Finally, we formally introduced a directed search algorithm that produces optimal solutions for Set Cover instances. This search algorithm also uses the four improvements we proposed, and starts off by exploring the search space around the solution obtained by Improved Greedy, with the aim of relatively quickly discovering better solutions.

Algorithm 3.4: Our search algorithm

input : An instance of the Set Cover problem: a collection of sets $\mathcal{S} = \{S_0, S_1, \dots S_{m-1}\}$, such that $\mathcal{U} = \bigcup_{S \in \mathcal{S}}$.

output: A set C of indices that corresponds to an exact solution $\mathcal{C}$ for the initial problem instance.

```
1  Function search(P_C, P_S, C_s):
2      if |P_C| > |C_s| then
3          return C_s
4      S_m ← max_{S∈P_S} |S \ P_covered|
5      if |S_m \ P_covered| = 2 then
6          C_2 ← obtain a collection of set indices by optimally solving the current problem
               state. First reduce the problem to maximum cardinality matching, and then
               complete the resulting partial cover with one set for all remaining uncovered
               elements, similar to Algorithm 3.2.
7          P_C ← P_C ∪ {S_i | S_i ∈ P_S, i ∈ C_2}
8          P_S ← ∅
9      if P_covered = U then
10         P_C ← run Algorithm 3.3 on P_C to remove redundant sets
11         if |P_C| < |C_s| then
12             C_s ← P_C
13         return C_s
       // Recurse on including S_m
14     C_i ← search(P_C ∪ {S_m}, P_S \ {S_m}, C_s)
15     if |C_i| < |C_s| then
16         C_s ← C_i
       // Recurse on ignoring S_m if the resulting state would still be valid
17     if (P_C, P_S \ {S_m}) is valid then
18         C_e ← search(P_C, P_S \ {S_m}, C_s)
19         if |C_e| < |C_s| then
20             C_s ← C_e
21     return C_s

   // Remove all sets for which there is a superset
22 P_S ← {S_i | S_i ∈ S, ∄S_j ∈ S with S_i ⊂ S_j}
   // Add all sets with unique elements
23 P_C ← {S_i | S_i ∈ P_S, ∃e ∈ S_i s.t. ∄S_j ∈ P_S with S_j ≠ S_i and e ∈ S_j}
24 P_S ← {S_i | S_i ∈ P_S and S_i \ ⋃P_C ≠ ∅}
   // Start the search
25 C ← search(P_C, P_S, S)
26 C ← {i | S_i ∈ C}
27 return C
```

Chapter 4

Implementation

In the previous chapter, we explained how our improvements to the GREEDY algorithm worked, and formally introduced our directed search algorithm. Now, we first tackle the problem of efficiently implementing the base GREEDY algorithm, which is more complex than it may seem at a first glance. Afterwards, we describe how we implemented and integrated the four changes we previously presented in order to obtain a working version of IMPROVED GREEDY. Finally, we detail an efficient implementation of our search algorithm. We will give snippets of C++ code to better illustrate the most important parts of our implementation.

In the formal problem state notation we introduced in the previous chapter, we used $P_{\mathcal{S}}$ to refer to a collection of remaining sets. The algorithms we described could remove some set from this collection, and optionally add that set to the partial cover, thus transitioning to some new state. Both IMPROVED GREEDY and our search algorithm require knowing what the set S_m with $S_m = \max_{S \in P_{\mathcal{S}}} |S \setminus P_{covered}|$ is. In practice, it is very inefficient to directly iterate over all remaining sets from $P_{\mathcal{S}}$ and compute the difference $S \setminus P_{covered}$ in order to find S_m.

Instead, we could still keep a collection of remaining sets, but update it immediately after adding some set to the cover. As such, when we add some set to the cover, we need to also iterate over all the uncovered elements in this set, and remove them from all other sets in our collection. This means that our collection only holds the equivalent of the difference between some original set and $P_{covered}$, not the whole original set. This solves part of the problem, but we are still interested in efficiently obtaining the set of maximum cardinality from this collection, or more specifically the index of the original set from which we removed some elements to obtain this set of maximum cardinality.

A data structure that we could use to query for the set of maximum size in $O(\log n)$ time is a priority queue. In the form this data structure is traditionally implemented, using a heap, it does not support erasing elements other than the maximum in $O(\log n)$, because the time it takes to actually find that element is $O(n)$. To get around this, we could use a self-balancing binary search tree such as a red-black tree instead of a heap.

Our first attempt made use of a `std::multiset` from the C++ Standard Library. This container is typically implemented as a red-black tree and supports all the operations we needed to create a starting implementation. We implemented Improved Greedy using this data structure, and it performed quite well, but our first implementation of the search algorithm, which also used a `std::multiset` as its primary data structure, was too slow for the use cases we intended for it.

Based on the observation that the number of elements in the sets we would like to store can only decrease, we can instead use a faster implementation of a priority queue like data structure that uses an array of lists. If this array is called `sets`, `sets[p]` is a linked list containing all sets of size p. If we are careful to properly update some index `max_postion` for which `sets[max_position]` holds the sets with the maximum number of elements every time we make changes to this data structure, we can implement the base Greedy version in $O(M)$ time.

Before doing any kind of computation on some Set Cover instance, we normalise these sets to have indexes in $[0 \,..\, m-1]$, and we also normalise the elements such that they are represented by indexes in $[0 \,..\, n-1]$. We then represent an instance of the problem by a simple class for which we give part of the signature below:

```
struct SetCoverInstance {
    int n; // elements in the universe
    int m; // number of sets
    // vector of vectors: vector i holds all elements that are in set i
    std::vector<std::vector<int>> sets;
};
```

We will now detail a data structure that uses an array of lists to store the sets in order, and how we used this data structure to implement our algorithms as efficiently as we could.

4.1 Available sets data structure

To make it easier to implement our algorithms, we decided to create a data structure in which we could insert our initial sets and then be able to perform the following operations:

- Remove some set from the data structure, and remove all elements this set contained from all other sets in the data structure.
- Ignore some set by simply removing it from the data structure but not updating the other sets in any way.
- Get the set of maximum size, and also iterate over all remaining sets.

- Query if some set contains an element of frequency 1, meaning that there is some element that only appears in that set.

Out of these operations, the first one is the most involved. We need to update all sets that contain some element by removing that element from them in the most efficient way. If we had a pointer to the elements we wanted to remove from some set, and if those elements were stored in linked lists, we could remove them in constant time.

If we do not consider the ignore operation, we observe that the sets that initially contained some element are the only ones that will ever contain that element, and that if we remove some set that contains an element e, we need to update all other sets that initially contained (and still do) this e element. This means that we can precompute an array `element_id_to_sets` for which `element_id_to_sets[e]` is a vector that contains pointers to the nodes representing element e in the linked lists of the sets that initially contained element e, and use this array to efficiently remove those nodes if need be.

There are then two kinds of linked lists, one for the elements currently in our sets, and one that stores pointers to sets with certain cardinalities. Each set can store a pointer to the first element it has (the head of their element list), and we can have an array `sets` of size n that for some position p stores the head of the list of sets currently of cardinality p.

We named the C++ class that provides the functionality described above `AvailableSets`. In this class, we defined two additional structures, `Set` and `SetElement`, which we give below as C++ code:

```
struct Set {
    Set *next, *prev;
    SetElement *first_element;
    int id, element_count;
    bool ignored;
};
struct SetElement {
    Set *parent_set;
    SetElement *next, *prev;
    int id;
};
```

These structures will be used as the nodes in our linked lists, and the `next` and `prev` variables will hold pointers to the next and previous elements in these linked lists. In the case of `Set`, the `first_element` variable holds a pointer to the first element in that set's list of elements, and the `element_count` variable is used to hold the number of elements in that set's list. Both structures have an `id` field, which uniquely identifies the set and element they respectively refer to. These

ids were obtained by normalising the input data, as mentioned earlier. The `ignored` boolean from `Set` will be used to more efficiently implement the ignore operation that we would like our data structure to have. The `parent_set` variable from `SetElement` intuitively holds a pointer to the `Set` this element belongs to.

We now give code for a part of the signature of our `AvailableSets` class:

```
class AvailableSets {
    Set *set_storage; // array of size m
    SetElement *element_storage; // array of size M
    Set **sets; // array of size m
    std::vector<SetElement *> *element_id_to_sets; // array of size n
    std::vector<int> element_frequencies; // vector of size n

    void add_element_to_set(SetElement *element);
    void remove_element_from_set(SetElement *element);
    void add_set_to_position_list(Set *set);
    void remove_set_from_position_list(Set *set, int position);
public:
    int max_position = -1, total_covered_elements = 0;

    AvailableSets(const SetCoverInstance &instance);
    void remove_set(Set *set);
    Set *ignore_set(Set *set);
    Set *get_set(int set_id) const;
    Set *get_first_set_at(int position) const;
    bool contains_freq_1_elem(Set *set) const;
};
```

The protected fields of this class include arrays `set_storage` and `element_storage`, which, as their names would suggest, are used to keep the `Set` and `SetElement` instances we need. In the case of `set_storage`, some set with id i corresponds to position i in this array. The first four functions in the snippet implement the basic linked list operations for the two kinds of lists we keep.

We use the `max_position` variable to refer to the size of the currently largest set, and `total_covered_elements` to refer to the number of elements that have been covered. When `total_covered_elements` is equal to the size of the universe, we know we have a valid cover. The `get_set(int set_id)` function simply returns the set at position `set_id` in the `set_storage` array, and `get_first_set_at(int postion)` returns `sets[position]`.

Initialisation

The constructor of our class takes a constant reference to a `SetCoverInstance` object, allocates memory for the arrays it uses, and builds the required linked lists by adding elements to each set's list and then inserting these sets in the list corresponding to their size. Here, we also initialise `max_position`, the element frequency array, and the reverse lookup array `element_id_to_sets`.

Removing a set

When removing some set S from our data structure, we need to update other sets that contain elements from S, set the frequency of those elements to 0, increment the covered elements counter, and also make sure to recompute `max_position` if needed. We give C++ code that performs these required updates below:

```
void AvailableSets::remove_set(Set *set) {
    remove_set_from_position_list(set, set->element_count);
    // iterate over the elements of this set
    SetElement *set_element = set->first_element;
    while (set_element) {
        // remove this element from all sets that contain it
        for (auto *other_sets_element : element_id_to_sets[set_element->id]) {
            auto *parent_set = other_sets_element->parent_set;
            if (parent_set == set || parent_set->ignored) continue;

            // remove element from set's list, update set position
            remove_set_from_position_list(parent_set, parent_set->element_count);
            remove_element_from_set(other_sets_element);
            if (parent_set->element_count) add_set_to_position_list(parent_set);
        }
        element_frequencies[set_element->id] = 0;
        set_element = set_element->next;
    }
    total_covered_elements += set->element_count;
    // reset set element list
    set->first_element = nullptr, set->element_count = 0;
    // update max_position
    while (!sets[max_position] && max_position != 0) --max_position;
}
```

Observe that instead of also removing the elements from `set` one by one, we simply reset the head pointer, and set the total element count of that set to 0. We also do not re-add empty sets back to the list in position 0 (line 14), and do not remove elements from ignored sets (line 9).

Ignoring a set

To mark `set` as ignored, we simply update `set->ignored` to be `true`, decrease the frequency of all elements this set contains by one, and remove it from the linked list it was in. We also update `max_position` just as we do for the remove operation. Instead of removing the pointers to the elements of this set from the reverse lookup array `element_id_to_sets`, we are careful in the `remove_set()` function to not update ignored sets. We tested an implementation that removed the pointers of ignored sets from `element_id_to_sets`, and found it to be slower than simply not updating the ignored sets in `remove_set()`.

Iteration

We can now call `get_first_set_at(max_position)` to get one of the sets that have maximum size. To simplify the implementation of our algorithms, we implemented a C++ style iterator that starts from the first set of maximum size, and goes through all the remaining sets, in descending order. We also implemented such an iterator for the elements in a set. Both these iterators support modern C++ range based iteration, meaning we can for example use `for(int element : set)` to iterate over all elements of some set.

Frequency 1 querying

We want to be able to query whether or not some set contains an element of frequency 1, because this means we cannot ignore it in our search, and thus have to add it to the cover. To accomplish this, we keep the array of element frequencies, and we simply iterate over all remaining elements of a set and return `true` if one of these elements has frequency 1.

4.2 Implementing Improved Greedy

Using the data structure we described in the previous section, we can now very easily implement the base Greedy algorithm in less than 10 lines of code. However, in order to implement our improved version of this algorithm, we also need some functions that perform the changes we described in the previous chapter. Out of the four, the reduction to maximum cardinality matching is the most complicated to implement. We will briefly detail the implementation of this reduction, and then show how we can put everything together to obtain the Improved Greedy algorithm.

4.2.1 Matching reduction

In practice, there might be duplicate sets of size 2 in our `AvailableSets` data structure, so the first step in the reduction to maximum cardinality matching is to remove all but one of the duplicates. This can be achieved by iterating over the remaining sets and only keeping the first occurrence of identical sets. We can check if we encountered an identical set before by storing the sets as integer pairs in a hash table. In fact, we use a hash map to store this information, along with a pointer to the set the integer pairs refer to, as we will require this mapping in a later step of this reduction. We are also careful to consider the sets `{e_1, e_2}` and `{e_2, e_1}` to be identical in this implementation. We provide the declaration of the hash map we use to map integer pairs to set pointers:

```
std::unordered_map<std::pair<int, int>, AvailableSets::Set *> card_2_sets;
```

We plan to use the implementation of Edmond's maximum cardinality matching algorithm from the Boost Graph Library, and for this we need to construct an undirected graph with some v vertices numbered from 0 to $v-1$. The sets we would like to transform into a graph do not contain elements from 0 to $v-1$, so the second step is to normalise the elements by attributing consecutive indexes to them, starting from 0. We also need to be able to revert this normalisation, so that we can add the required set to our partial cover.

```
std::vector<int> element_map(instance.n); // old_id -> new_id
std::vector<int> inverse_element_map; // new_id -> old_id
std::vector<std::pair<int, int>> edges;
```

We store mappings from the old element ids to the normalised element ids (`element_map`) and vice-versa (`inverse_element_map`), and we also populate a vector of integer pairs which we call `edges`. We now construct a graph with `inverse_element_map.size()` vertices from the integer pairs in the `edges` vector. We then run Edmond's algorithm on this graph, and obtain another vector of integer pairs (edges) that we call `matching`. The edges in `matching` all correspond to some cardinality 2 set, and we can use the hash map we initially created for removing duplicate sets to recover the sets corresponding to the edges in matching and add them to the cover. We complete the reduction to maximum cardinality matching by simply adding some remaining set of cardinality 1 to the cover and removing it from `AvailableSets` until all elements are covered.

4.2.2 Integrating the improvements

We created four functions to correspond to the four improvements we presented in Chapter 3. For some instance `available_sets` of our `AvailableSets` class, `add_unique()` adds all sets with a unique element to the cover and removes them from `avialable_sets`, `reduce_to_matching()`

optimally solves a state in which maximum size of some set in `available_sets` is 2, and `remove_redundant()` greedily removes sets from the cover, as explained in Algorithm 3.3.

To implement the subset removal reduction, we first sorted the initial element vectors of each set, to facilitate efficient subset checking for a pair of sets. Then, we sorted a vector that contained the initial sets by the size of those sets and compared some set with index i in this vector to all other sets with indexes $j > i$. If we found some set of index j that was a superset of the index i set, we removed the index i set from our collection. We created a function named `ignore_subsets()` that performs the steps we just described, and removes all subsets from some `AvailableSets` instance.

Now it's simply a matter of using the features of our `AvailableSets` class and the functions we just mentioned to implement Improved Greedy:

```
1  std::vector<int> run_improved_greedy(const SetCoverInstance &instance) {
2      std::vector<int> cover; // we return set ids
3      AvailableSets available_sets(instance);
4
5      ignore_subsets(instance, available_sets); // remove all subsets
6      add_unique(available_sets, cover); // add all sets with unique elements
7
8      while (available_sets.max_position) {
9          auto *set = available_sets.get_first_set_at(available_sets.max_position);
10         if (set->element_count == 2) {
11             reduce_to_matching(instance, available_sets, cover);
12             break;
13         }
14         cover.push_back(set->id);
15         available_sets.remove_set(set);
16     }
17
18     remove_redundant(instance, cover); // greedily remove redundant sets
19     return cover;
20 }
```

4.3 Efficient searching

The most difficult issue we now face in efficiently implementing our search algorithm is that we do not have a way of reverting the remove and ignore operations performed on our `AvailableSets` data structure. A straightforward alternative that avoids needing to revert operations would be to make a copy of our current instance of `AvilableSets`, remove the set of maximum size from it

and then recurse on this resulting state. When we finish exploring that branch of the search tree, we can ignore the same set in another copy, and then recurse on that new state. We implemented this approach with a copy constructor for our class, and while we did not even expect to be able to run our algorithm on larger Set Cover instances because of significant memory usage, we found that the repeated copying resulted in significant slowdowns in the case of smaller instances as well.

4.3.1 StateRecord restoration

We will now present a state restoration system we designed that helps us efficiently undo the remove and ignore operations supported by our class.

Let us first consider the simpler case in which we would like to undo one single remove operation of some `set`. We need to not only re-insert this `set` in the right position in our array of lists, but also re-add the elements we might have removed from other sets, and move those other sets to their initial positions. This can be accomplished by storing all the elements we removed (the elements `set` had at the time it was removed) in a vector, and then simply adding all these elements back to every set that initially contained them using our reverse lookup array, `element_id_to_sets`. In practice, we would like to be able to revert a sequence of ignore and remove operations, not just only one such operation. We also need to be mindful of the fact that some sets might be ignored, and as such we should not re-add elements to those sets.

As we would like to use this state restoration system to obtain a more efficient implementation of our search algorithm, we additionally need a way of specifying the location where the changes should be recorded to. This way, each recursive call of our search function could manage memory only for the changes that were made in it, meaning exactly the changes it needs to revert to end up in the same state it started in. We need to support reverting a sequence of operations because some call to our search function might optimally complete some state by reduction to maximum cardinality matching, and as such execute multiple remove and ignore operations before needing to revert to the original state.

We define a *state record* to be an object that contains all the information we need to restore some sequence of operations. If this state record is attached to some `AvailableSets` instance, that instance will keep track of all the executed remove and ignore operations by appending the required sets and elements to vectors. The C++ struct that we used to store this information is as follows:

```
1 struct StateRecord {
2     int64_t state_record_id;
3
4     struct ExtractedSet {
5         Set *set;
```

```
        SetElement *prev_first_elem;
        int prev_elem_count;
    }; std::vector<ExtractedSet> extracted_sets;

    struct ExtractedElement {
        int id, prev_frequency;
    }; std::vector<ExtractedElement> extracted_elements;
};
```

We add all sets that were modified to the `extracted_sets` vector. The two extra variables of `ExtractedSet` are used to restore the element lists of sets that were completely removed (`remove_set()` was called on them) or ignored. We need to treat these cases in a special way because the elements of those sets were not actually removed from the linked list, only the head of that list was set to `nullptr`. Only in those two cases, we store the first element in the set's list and the size of the list right before they were removed or ignored. In a sequence of operations, some set might be modified more than once, but we only want to store it in this vector once. For this, we use the `state_record_id` variable, along with an extra field we added to the `Set` struct from `AvailableSets` that tells us the id of the state record this set was last added to. If some set was already modified in the current state record, but then it was completely removed or ignored, we update the `prev_first_elem` and `prev_element_count` fields of its corresponding entry in `extracted_sets` to reflect this set's state right before it was completely removed or ignored.

For keeping track of the elements that were removed from the sets and that we need to add back we use the `extracted_elements` vector. Here, we also store the frequency of elements before they were removed, so that we can more efficiently restore it. We also added an extra variable to the `SetElement` struct in `AvailableSets`, say `is_in_parent_list`, that is `true` when this element is a node in its set's element list. If this variable is indeed `true`, we know not to add this set again. This will happen when some set was completely removed or ignored, case in which we restore (part of) that set's list using the two extra variables we saved in `ExtractedSet`. The special treatment of this case is particularly useful when the only change to our data structure is some ignored set: we just set two pointers and re-add it its corresponding list.

To support writing to and then reverting from some state record, we added some extra fields and functions to our `AvailableSets` class. First of all, we need to store the current record we will write the changes to, say in a variable `StateRecord *current_state_record`. Then, we need to store the id of some next state, and we also have a boolean that is set by the class constructor and that dictates whether or not we actually record the changes resulting from remove and ignore calls. We then added two functions that set the record our class is currently writing to, and use a record to restore the state of our class, respectively:

```
void replace_state_record(StateRecord *new_state_record);
void restore_from_state_record(StateRecord *state_record);
```

We can now use this functionality to efficiently restore some `AvailableSets` to the state it was in before some sequence of remove and ignore operations was performed. That being said, we do not need to undo the changes written to some record, we can just ignore the record in question and start writing to a new one by calling the replace function of our class.

4.3.2 Search algorithm implementation

We can now put everything together and implement our search algorithm, as specified in Algorithm 3.4. This is relatively simple now that we have all the complex state keeping functionality we need abstracted away in our `AvailableSets` class. We give a part of the search function we implemented to demonstrate how the state restore system can be used to create the final algorithm:

```
1  void search(std::vector<int> &cover, std::vector<int> &best_cover,
2              AvailableSets &av_sets) {
3      if (cover.size() >= best_cover.size()) return;
4
5      // declare a record object, start writing changes to it
6      AvailableSets::StateRecord state_record;
7      av_sets.replace_state_record(&state_record);
8
9      // reduce to maximum cardinality matching if possible.
10     // check if solution is complete, update best_cover accordingly.
11     // if we reduced to matching, restore state and return now.
12     ...
13
14     // recurse on adding the set of maximum size
15     auto *max_set = av_sets.get_first_set_at(av_sets.max_position);
16     cover.push_back(max_set->id);
17     av_sets.remove_set(max_set);
18     search(cover, best_cover, av_sets);
19
20     // restore state, reset state record
21     cover.pop_back();
22     av_sets.restore_from_state_record(&state_record);
23     av_sets.replace_state_record(&state_record);
24
25     // recurse on ignoring set of maximum size, then restore state
26     if (av_sets.contains_freq_1_elem(max_set)) return; // we must include set
27     av_sets.ignore_set(max_set);
28     search(cover, best_cover, av_sets);
```

```
29      av_sets.restore_from_state_record(&state_record);
30  }
```

Before this function is called, we ignore all sets for which there is a superset, and add all sets that contain a unique element to the cover. This function will then populate the `best_cover` vector with the indices of the sets that make up an optimal solution to some Set Cover problem.

4.4 Chapter summary

This chapter briefly presented the process of arriving at an efficient and elegant way of modelling the Set Cover problem in software. We detailed the main parts of a C++ implementation of a class that models this problem, which we designed specifically to help us easily and efficiently implement our algorithms. We explained how we implemented the reduction to maximum cardinality matching, and how we used the `AvailableSets` class we designed to implement our approximation algorithm. Finally, we presented an efficient state restoration mechanism that we can use to effectively undo any changes we made to some instance of our helper class, and demonstrated how this mechanism can be used to create a relatively efficient implementation of our directed search algorithm.

Chapter 5

Analysis

In this chapter, we will quantify the performance of the improvements we propose over the base Greedy algorithm on multiple practical Set Cover instances. We will also use our search algorithm to produce exact solutions for some of these instances, and to gain some insight into the search space it explores.

5.1 Data sets

We tested our algorithms on 16 different practical Set Cover instances. Out of the total, 10 were generated from undirected graphs of social, interaction and recommendation networks from The Network Data Repository [23]. We converted the Dominating Set instances represented by these graphs to Set Cover instances by simply taking a set to be the closed neighbourhood of a vertex. A complete list of the instances that were converted from graphs, along with various relevant data about them, such as the average set size, can be seen in Table 5.1. Observe that for all these instances, $n = m$, because there is one set for each vertex, and the sets can only contain other vertices.

Table 5.1: Instances generated from graphs.

ID	Instance	n	m	M	Avg. set size	Max. set size
1	soc-highschool-moreno	70	70	618	9	20
2	aves-songbird-social	110	110	2164	20	57
3	soc-ANU-residence	217	217	3895	18	57
4	soc-physicians	241	241	2087	9	29
5	ia-email-univ	1133	1133	12035	11	72
6	bn-fly-drosophila_medulla_1	1781	1781	19603	11	928
7	soc-hamsterster	2426	2426	35686	15	274
8	soc-wiki-elec	7118	7118	208624	29	1066
9	rec-movielens-user-movies-10m	7601	7601	118369	16	1639
10	fb-pages-company	14114	14114	118368	8	216

The other 6 instances were obtained from the Frequent Itemset Mining Dataset Repository [1], and we list them in Table 5.2. Note that some of these instances contain only sets of equal sizes, and that they come from various sources: one of them, T10I4D100K, is actually computer generated. We attributed ids to all the instances, so we can more simply refer to them from now on.

TABLE 5.2: Frequent Itemset Mining Dataset instances.

ID	Instance	n	m	M	Avg. set size	Max. set size
11	connect	129	67557	2904951	43	43
12	accidents	468	340183	11500870	34	51
13	T10I4D100K	870	100000	1010228	10	29
14	pumsb	2113	49046	3629404	74	74
15	retail	16470	88162	908576	10	76
16	kosarak	41270	990002	8018988	8	2497

5.2 Quantifying the improvements

The first improvement we presented involves reducing the problem to maximum cardinality matching when the set that contains the maximum number of uncovered elements is of size 2. We plot the percentage by which the base GREEDY solution was improved on by only applying this improvement in Figure 5.1, for all 16 instances. If we had some initial solution of size s, and obtained an improvement of p percent, the size of our improved solution size would be $s*(1-\frac{p}{100})$.

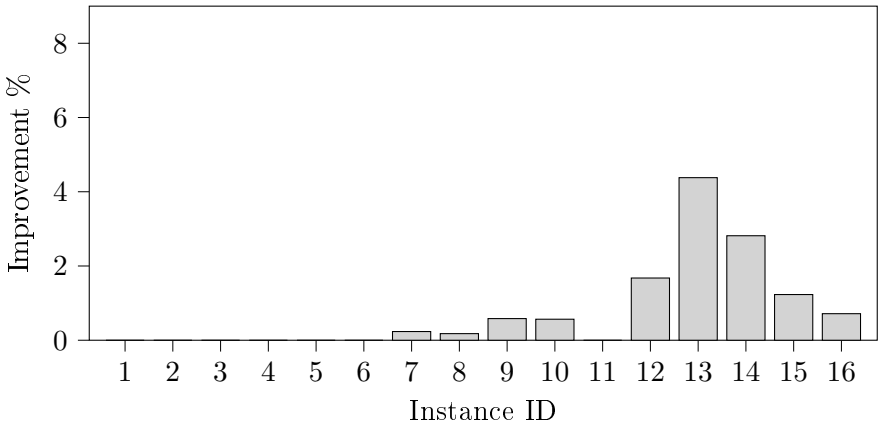

FIGURE 5.1: Improvements obtained by applying the reduction to maximum matching.

We presented an improvement that added all sets with unique elements to the cover before the main loop of the GREEDY algorithm. We then said that this change could result in even greater improvements if we first remove all sets for which there is a superset, and only then add the sets with unique elements. This is because by simply no longer considering any subset, we could now have more elements that appear in only one of the remaining sets. Figure 5.2 shows the improvements resulting from only adding sets with unique elements as the solid grey bars. The hatched bars in this figure illustrate how first removing subsets, and only then adding sets with

unique elements changes the performance of the algorithm. Note that only removing subsets before running GREEDY does not change the size of the solutions we obtain for any of the 16 instances.

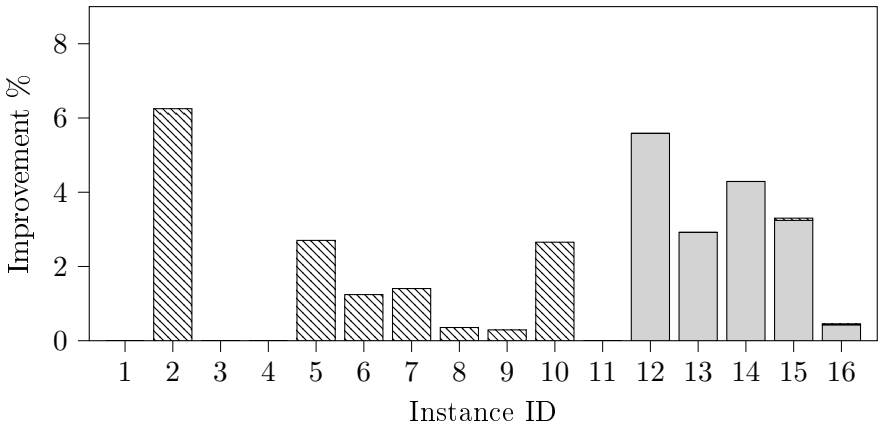

FIGURE 5.2: Improvements obtained by adding sets with unique elements first shown as solid bars. Extra improvements obtained by first removing subsets, then adding the sets with unique elements shown as hatched bars.

Greedily removing redundant sets from the cover we obtain by running the base GREEDY algorithm resulted in the largest average improvement on the 16 instances: 2.17%. The reduction to maximum matching resulted in a 0.77% mean improvement, while the combination of first removing subsets and then adding sets with unique elements resulted in a 1.96% improvement, on average. We show the improvements obtained by removing redundant sets in Figure 5.3.

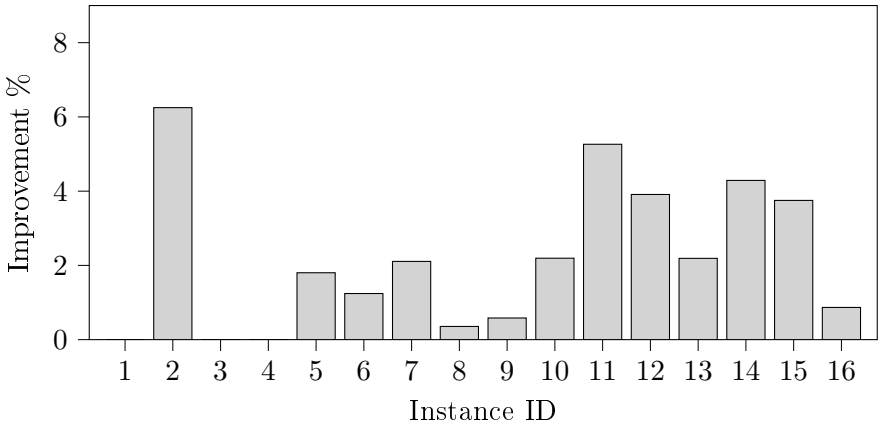

FIGURE 5.3: Improvements obtained by greedily removing redundant sets from the cover.

We now combine all these improvements into what we call IMPROVED GREEDY, and compare its performance to that of the base algorithm. Because removing subsets may take a very long time relative to the execution time of the base algorithm, we also test the performance of our algorithm without first removing subsets. Figure 5.4 shows the improvements our algorithm obtains when not removing subsets as solid grey bars, and the extra improvements it achieves when also removing subsets as the hatched bars.

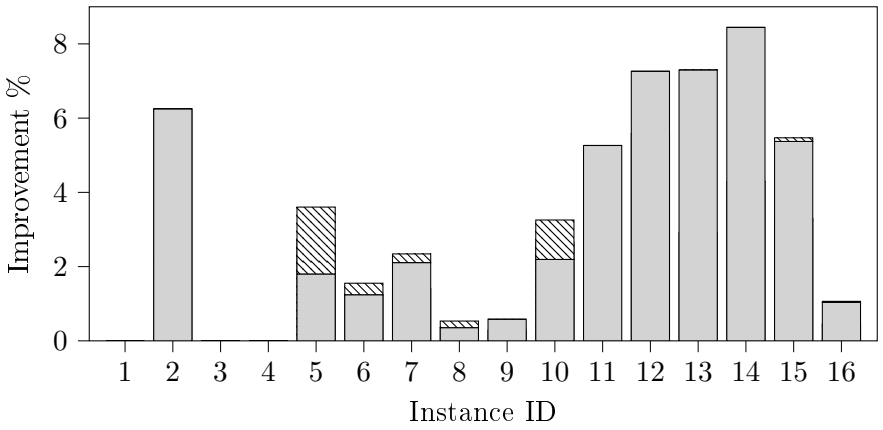

FIGURE 5.4: Improvements obtained by using all proposed changes besides the removal of subsets shown as solid bars. Extra improvements obtained when subsets are removed shown as hatched bars.

The mean improvement of our algorithm over the base version is 3.3%, and the maximum improvement of 8.44% is achieved on instance number 14. In Table 5.3, we present the cover sizes obtained by the base GREEDY algorithm, the cover sizes obtained by IMPROVED GREEDY, the percentage of initial sets that were ignored because they were a subset of some other set, the number of sets that were added to the cover because they contained an unique element, and finally the size of the cover right before we reduce the problem to maximum cardinality matching.

TABLE 5.3: Statistics obtained by running IMPROVED GREEDY on the 16 instances we are testing.

ID	Base size	Improved size	% sets ignored	Unique element sets	Cover size before matching
1	12	12	20	0	7
2	16	15	41	6	9
3	20	20	5	0	14
4	36	36	12	2	26
5	222	214	20	129	161
6	322	317	47	269	272
7	427	417	51	325	367
8	1125	1119	36	1048	1054
9	343	341	24	279	289
10	2826	2734	32	1821	2145
11	19	18	0	0	15
12	179	166	1	79	98
13	137	127	22	1	87
14	746	683	1	138	305
15	5119	4839	15	2115	2895
16	17758	17569	67	7996	8508

It is interesting to observe the difference between the cover size right before the problem is reduced to maximum matching and the number of sets with unique elements that were added to the cover. This difference is essentially the number of sets our algorithm adds to the cover in a greedy way, but is not necessarily indicative of the quality of the solution our algorithm produces.

On average, 25% of all the initial sets are ignored from the start because they are subsets of some other set. After ignoring those subsets, an average of 41% of the sets that end up in the cover of some instance must also be in an optimal cover of that instance, because they contain a unique element that is not contained in any other remaining set.

If we omit removing all subsets, the time our approximation algorithm takes to execute is on par with that of the base GREEDY algorithm. That being said, if we do remove these subsets, our algorithm becomes a lot slower, especially in the case of instances with large m and M values. In practice, perhaps the improvements we get by removing the subsets from large instances are not worth the potential multiple orders of magnitude increase in execution times.

5.3 The search algorithm in practice

We ran the solution space exploration part of our directed search algorithm for an hour. This means that we did not consider the time our algorithm took to do initial preparation such as removing all subsets from the initial instance as part of this hour, but only started measuring elapsed running time from the moment our algorithm started exploring the solution space. The measurements were taken from a single run performed on a server with a stable CPU clock frequency of 3.6 GHz.

Our search algorithm produced optimal solutions for 5 of the 16 instances we tested it on. The cover sizes obtained by the approximation algorithms are compared with the optimal cover sizes computed by our search algorithm in Table 5.4. In this table, we also show the time IMPROVED GREEDY and our search algorithm took to execute on these instances.

TABLE 5.4: Optimal solutions obtained by our search algorithm.

ID	n $(= m)$	Base size	Improved size	Optimal size	Improved time	Optimal time
1	70	12	12	11	<1ms	4.5s
2	110	16	15	14	<1ms	6ms
6	1781	322	317	317	33ms	35ms
8	7118	1125	1119	1119	734ms	783ms
9	7601	343	341	341	932ms	187s

Impressively, our search algorithm was able to optimally solve instances with thousands of sets in a very short amount of time. With the help of the improvements we proposed over the base approximation algorithm, IMPROVED GREEDY also arrived at an optimal solution for three out of the five instances.

On the other hand, there were some smaller problems, such as instance number 3, for which our algorithm was not able to produce an optimal solution in less than an hour. As expected, our algorithm works best when the instances we run it on present certain particularities that it can exploit, such as a large number of subsets, many sets with unique elements, or even a large average set size. If we look back at Table 5.3, we can see that the difference between the cover

size before we reduced the problem to matching and the number of sets with unique elements was less than 10 for each of the five instances we solved. This small difference may be some indication as to why our algorithm was able to solve these instances, but we cannot be sure.

For both the first two instances in Table 5.4, our algorithm found the greedy solution after approximately 0.02ms, and the optimal solution after approximately 0.07ms. In the case of the first instance, our algorithm found 1446 solutions of optimal size, discarded 15505 covers for being too large before the reduction to matching, and discarded 623717 more covers for being too large after the reduction to matching. It only found 17 covers of optimal size for the second instance, and it discarded only 907 covers for being too large after the reduction to matching, in this case. For the other three instances, our algorithm found 2, 5, and 1 covers of optimal size, and discarded 10, 91 and 995989 covers for being too large after the reduction to matching, respectively.

For all but one of the five instances we are discussing, our algorithm never discarded any incomplete cover before it encountered a set of cardinality 2 and reduced the problem to maximum cardinality matching. This simply means that in these cases, there was no way of constructing an incomplete cover of size greater than the best cover discovered so far in that search without the largest remaining set having size less or equal to 2.

The cover sizes our algorithm arrived at after one hour of running are compared with the sizes the base Greedy algorithm and our improved version obtained in Table 5.5. In this table, we also show the number of complete or incomplete covers our search algorithm discarded for being too large. It is immediately obvious that the speed with which our algorithm explores the solution space is highly dependent on the size and structure of the instance. In one hour, our algorithm was able to explore more than 415 million potential covers in the case of instance number 3, but less than 75 thousand in the case of the largest instance we tested it on.

Table 5.5: Solutions obtained by our search algorithm after one hour.

ID	n	m	Base size	Improved size	Search size	Covers discarded
3	217	217	20	20	17	415 196 689
4	240	240	36	36	35	209 164 149
5	1133	1133	222	214	212	62 065 044
7	2426	2426	427	417	416	97 095 655
10	14114	14114	2826	2734	2734	5 414 459
11	129	67557	19	18	16	11 893 571
12	468	340183	179	166	162	822 466
13	870	100000	137	127	123	1 970 004
14	2113	49046	746	683	679	1 471 727
15	16470	88162	5119	4839	4839	704 127
16	41270	990002	17758	17569	17569	73 475

Out of the 11 instances in Table 5.5, our search algorithm found a solution that is better than that obtained by Improved Greedy in less than an hour for 8 of them. For the other three instances (with ids 10, 15, 16), our algorithm did not find any other valid cover of the same size

as the initial one, meaning that in an hour of searching it only found one cover, and that cover is the same one Improved Greedy found, as per the design of our algorithm. That being said, the three instances are in fact the largest out of all 16 instances we tested by the number of elements in the universe (n), and also quite large if looking at the number of sets (m), so this does not necessarily indicate that the solution space for those instances is sparse, but perhaps that our algorithm was not able to explore enough of said solution space. For instance 15 and 16, the number of discarded covers is the lowest out of the 11 instances our algorithm did not terminate on, further indicating that we might have simply explored very little of the solution space.

On the other hand, the fact that we did not find any better solution in the huge search space of these instances may be evidence of the good performance of the greedy approach, and more specifically of our improved approximation algorithm. As we saw in Table 5.4, where we presented the optimal solutions our algorithm arrived at, the Improved Greedy algorithm found solutions that were the same size or only larger by one when compared with the optimal cover size. We cannot draw any conclusions regarding the performance of the greedy algorithms relative to the optimal solution size, apart from the five instances we were able to produce optimal solutions on. In those cases, the base Greedy algorithm was on average 4.7% off the optimal solution, and our Improved Greedy algorithm produced solutions that were on average 3% larger than the optimal solution. Relative to the cover size we obtained after one hour of running our search algorithm, the base Greedy solutions were 6.4% larger, and the solutions our improved variant produced were 3.2% larger, on average. This fact does not disprove previous results that indicated that Greedy was on average within 7% off the optimal solution size for practical Set Cover instances [11], but it also does not necessarily corroborate these findings, as we simply do not know the optimal solution size for many of the instances we tested.

5.3.1 Search space

We can plot histograms of the search spaces our algorithm explored in order to better understand how solutions are distributed. These plots are not representative of the whole solution space of some problem, because our search algorithm simply does not explore it completely by design.

Figure 5.5 shows the distribution of complete cover sizes our algorithm encountered when it executed on instance 9. In this case, we did not discard covers that were too large before reducing to matching, but instead completed those as well. For this instance, the optimal solution is of size 341, and while it was also found by Improved Greedy, it not clearly visible in this figure as our search algorithm encountered only one cover of this size. However, we can clearly observe the peaks of three normal distributions centred at approximately 410, 435 and 460. We observed one or more such normal distributions in almost all histograms we plotted for the 16 instances we tested.

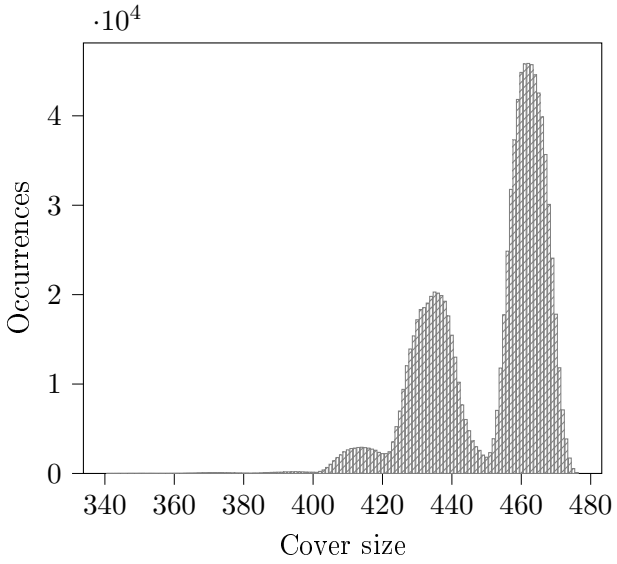

FIGURE 5.5: Distribution of solution sizes for instance 9.

As we saw in Table 5.5, our algorithm was able to improve on the greedy solution for some of the instances. In the case of instance 12, our algorithm found a solution of size 162, which is 4 elements smaller than that found by IMPROVED GREEDY. We plot the times at which our algorithm discovered incrementally better solutions in Figure 5.6, in which we use a logarithmic scale for the x-axis. Interestingly, the time between the discovery of incrementally better solutions increases exponentially, illustrated by the fact that the points in the graph are almost on a straight line. All instances on which our search algorithm managed to obtain solutions that were better by at least 2 sets than the starting solution exhibit exponential increases in discovery times, albeit not always as linear on a logarithmic scaled graph as the example we show here.

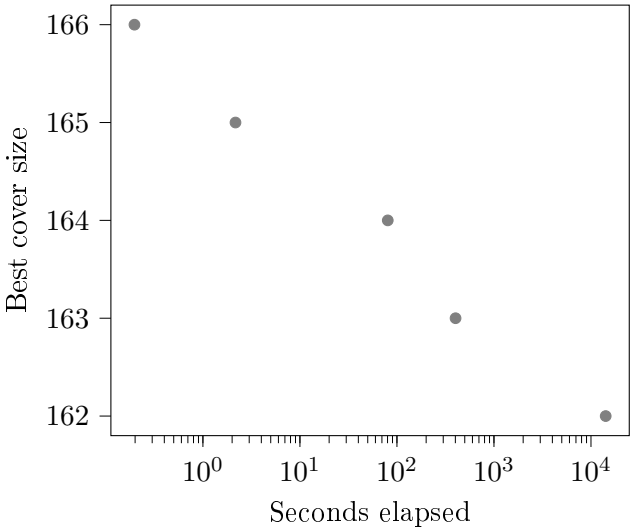

FIGURE 5.6: Better cover discovery over time for instance 12.

For the same instance, number 12, we give a histogram of the solutions our algorithm explored in Figure 5.7. No incomplete partial cover was discarded for being too large in this case, but instead all valid covers were obtained by reducing the problem to maximum matching. Out of

more than 800 thousand valid covers discovered in an hour, only 21 were of the smallest size of 162 sets. Moreover, the first cover of smallest size was discovered 24 minutes into the search, leaving more than half of the total execution time to still explore the search tree afterwards.

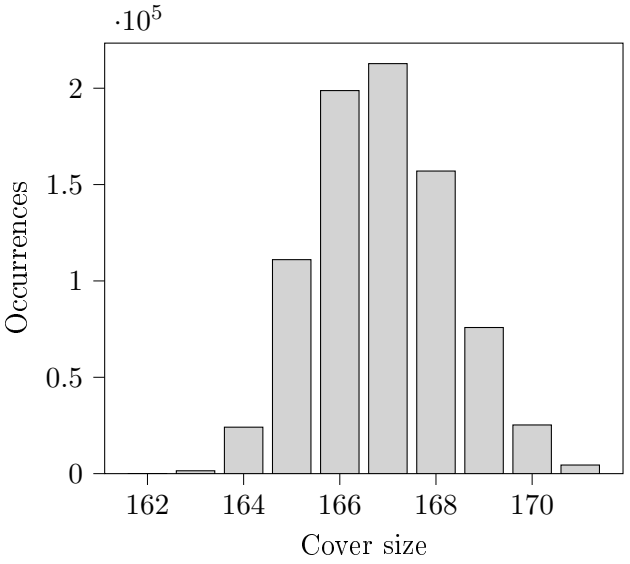

FIGURE 5.7: Distribution of solution sizes for instance 12.

Our search was centred on the IMPROVED GREEDY solution, and for large instances such as instance 12, we probably only explored a tiny fraction of the solution space, which means that the distribution of solutions we presented in Figure 5.7 may not be representative of the real solution space our algorithm would explore if we let it run for enough time.

We were interested to know how much of the search space we have explored in an hour, so that we can have an estimate of the quality of the solutions we arrived at. To achieve this, we can look at the number of calls that were made to our search function at certain depths. We consider the first call of the search function to be at depth 0, and then the recursive call made from that to be depth 1, and so on. If we were to plot the absolute number of calls at each depth, we would get a normal-like distribution with a very long and flat left tail, where the calls of small depth are. Instead, for some depth $d > 1$, we plot the ratio $calls(d)/calls(d-1)$, which indicates how many more times a recursive call of that depth was made, relative to the same count for the preceding depth. A naive search algorithm that explores the whole search space of some instance with m sets would have $calls(d)/calls(d-1) = 2$ for all $d > 1,\ d < m$.

We show a graph of these ratios for one hour of run time on instance 12 in Figure 5.8. It looks like the smallest depth at which our algorithm explored some part of the ignore branch was depth 11 out of 60. This means that our algorithm explored the whole search tree that results after we decide to always include the first 11 sets our approximation algorithm includes, and was currently exploring the search tree resulting from ignoring the 11th set when it was stopped. In the case of this instance, the 79 sets we added to cover unique elements before starting the search actually covered 306 out of $n = 468$ elements in the universe. We then have an upper bound for the depth of our search equal to 162, but we can observe that our algorithm only explored up to a depth of 64 in practice. We also see that the ratio rapidly decreases for depths greater than

20, indicating that our algorithm was able to reduce the problem to maximum matching or that the ignore branch was not explored, as it would have led to an invalid state. This instance had a total of 340183 sets initially, so we can safely say that we were able to very significantly reduce the size of the search space using our directed search approach.

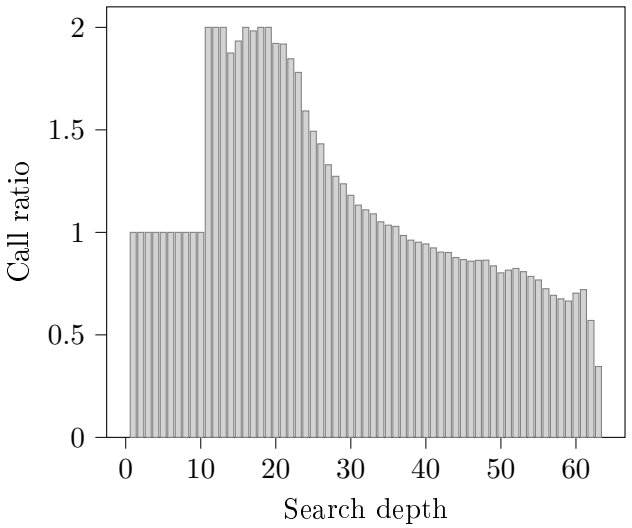

FIGURE 5.8: Call ratios for instance 12.

Another such call ratio graph for instance 1, which we were able to optimally solve, can be seen in Figure 5.9. Again, we observe a rapid decrease in the call ratio as the depth increases.

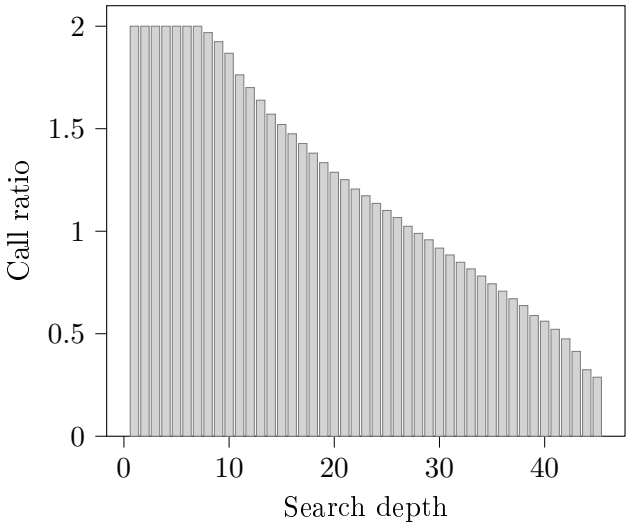

FIGURE 5.9: Call ratios for instance 1.

For instance 11, our search algorithm was able to reduce the cover size from 20 down to 17, and its call ratio graph is shown in Figure 5.10, where we can see that it was able to explore the ignore branch for depths up to 11, just like in the case of instance 12. On the other hand, we plot the call ratio graph for instance 16 in Figure 5.11. In this case, we can clearly see that a significantly smaller part of the search space was explored, and also that the search spaces of instance 11 and instance 16 are very different: the call ratio of instance 11 becomes 1 after about depth 50, but in the case of instance 16, we see no decrease at the end. This may indicate that for instance

16, adding some set to the cover does not result in reductions of the remaining search space as significant as those in the case of instance 11.

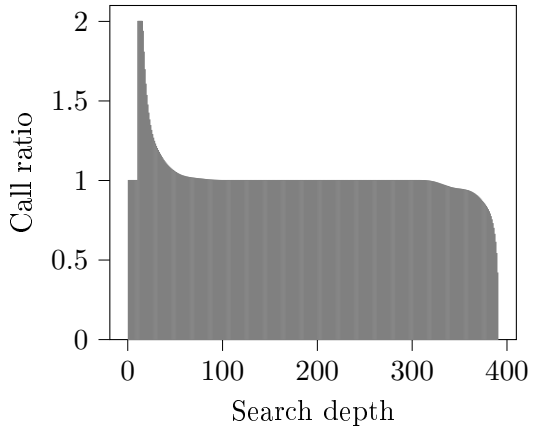

FIGURE 5.10: Call ratios for instance 11.

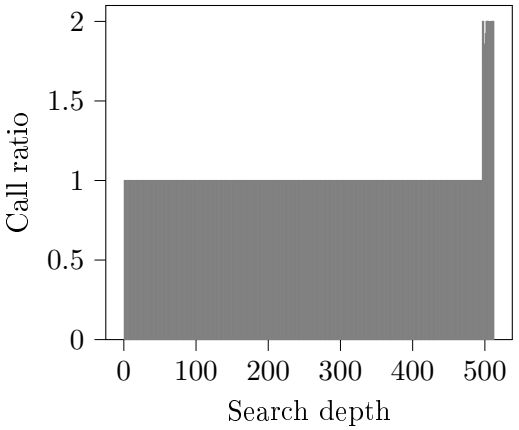

FIGURE 5.11: Call ratios for instance 16.

5.4 Chapter summary

In this chapter, we quantified on 16 practical instances the individual impact of the four improvements we proposed (reducing to maximum matching, removing subsets, adding sets with unique elements first, and removing redundant sets from the cover), and also the improvement that results from combining all these changes. We compared the performance of our approximation algorithm to that of the base GREEDY algorithm on these 16 instances, and presented several statistics of interest. We then ran our search algorithm for up to an hour, presented the improvements in cover size we were able to obtain in this time, and analysed a number of different metrics that we recorded during the execution of our algorithm.

Chapter 6

Discussion

By tackling the Set Cover problem from the more practical perspective of algorithm engineering, we were able to not only improve on the quality of the Greedy algorithm, but to also design a search algorithm that behaves impressively well on practical instances.

Our improved approximation algorithm is based on the same key principle of the Greedy algorithm: always adding the set that would cover the maximum number of new elements to the cover. Because the Greedy algorithm has an approximation factor close to the theoretical maximum, and also because it was shown to produce solutions within 7% of the optimum [11], we think that focusing our efforts on improving it, instead of trying to design a new technique or to improve on another existing algorithm, was a good approach. We proposed four improvements: reducing the problem to maximum cardinality matching when the size of the largest remaining set is 2, removing subsets from the initial instance, adding the sets with unique elements before the main part of our algorithm starts, and finally greedily removing redundant sets from the final cover. We efficiently and elegantly implemented our new Improved Greedy algorithm in C++ and ran it on 16 practical Set Cover instances. When compared to the base Greedy algorithm, we managed to obtain a mean improvement of 3.3%, and a maximum of 8.44%. Because one of the improvements we proposed significantly increases the run time of our algorithm, we also compared the performance of an algorithm without this change, and found that it still produced promising results. This faster version resulted in a 3.07% mean improvement, and the same maximum improvement of 8.44% was observed on one of the 16 instances we tested. We believe there is no compromise in using this simpler version of our approximation algorithm over the base Greedy version, as the performance is essentially the same, and we have empirical evidence to suggest it behaves measurably better on practical instances.

The contribution of this thesis is twofold: we designed and implemented a better approximation algorithm and a directed search algorithm for the Set Cover problem. Although this problem is theoretically NP-complete in general cases, we were able to create a search algorithm that can produce optimal solutions in manageable time for some moderately sized instances, by taking advantage of certain structural particularities of these instances. We wanted to create a search algorithm that could improve on the approximate Improved Greedy solutions in relatively

little time, so we decided to center our search on exactly this solution that out approximation algorithm finds, and start exploring the search space local to this solution. Intuitively, because GREEDY was shown to produce good results, and our approximation algorithm is even better in practice, searching around this solution seemed like a promising idea. Another factor that contributed to the design of our search algorithm was an observation that a lot of branches naive search algorithms explore are known to never produce optimal solutions. Therefore, our algorithm never considers adding sets that do not immediately contribute with new covered elements, and it never branches on ignoring some set that is essential in obtaining a valid cover because it is the only set that contains some element, outside of sets we already ignored on that branch. We implemented an efficient state restoration system to extend the functionality of the software model we created, and used this restoration mechanism to create a relatively efficient implementation of our directed search algorithm in C++. We used our search algorithm to optimally solve 5 out of the 16 instances we tested, with the largest instance out of the five containing 7601 sets. We then ran our algorithm for an hour on the other instances, and observed that it produced incrementally better solutions as execution progressed. We quantified the improvements our algorithm obtained in an hour of running, and provided an in-depth analysis of its execution. In this analysis, we looked at the distribution of the sizes of the complete covers our algorithm found, we illustrated the relationship between the size of the current best cover found and elapsed execution time, we gained insights into how our algorithm behaves on different instances by examining call ratio graphs, and presented other key statistics.

The SET COVER problem is a very important combinatorial problem with many real world applications, which we believe would greatly benefit from improvements in the quality of the solutions that can be obtained in reasonable time by practical algorithms. We improved on the theoretically best approximation algorithm there is for this problem when considering real world instances, and we designed a search algorithm that can be used to generate even better solutions, albeit in exponential time.

Bibliography

[1] B. Goethals. Frequent itemset mining dataset repository. URL http://fimi.cs.helsinki.fi/data.

[2] Barbara M. Smith and Anthony Wren. A bus crew scheduling system using a set covering formulation. *Transportation Research Part A: General*, 22(2):97–108, March 1988. doi: 10.1016/0191-2607(88)90022-2. URL https://doi.org/10.1016/0191-2607(88)90022-2.

[3] Ramesh V. Kantety, Mauricio La Rota, David E. Matthews, and Mark E. Sorrells. *Plant Molecular Biology*, 48(5/6):501–510, 2002. doi: 10.1023/a:1014875206165. URL https://doi.org/10.1023/a:1014875206165.

[4] Barna Saha and Lise Getoor. On maximum coverage in the streaming model & application to multi-topic blog-watch. In *Proceedings of the 2009 SIAM International Conference on Data Mining*. Society for Industrial and Applied Mathematics, April 2009. doi: 10.1137/1.9781611972795.60. URL https://doi.org/10.1137/1.9781611972795.60.

[5] Viggo Kann. *On the Approximability of NP-complete Optimization Problems*. PhD thesis, 1992.

[6] Richard M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103. Springer US, 1972. doi: 10.1007/978-1-4684-2001-2_9. URL https://doi.org/10.1007/978-1-4684-2001-2_9.

[7] David S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, 9(3):256–278, December 1974. doi: 10.1016/s0022-0000(74)80044-9. URL https://doi.org/10.1016/s0022-0000(74)80044-9.

[8] Petr Slavík. A tight analysis of the greedy algorithm for set cover. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing - STOC '96*. ACM Press, 1996. doi: 10.1145/237814.237991. URL https://doi.org/10.1145/237814.237991.

[9] Uriel Feige. A threshold of ln(n) for approximating set cover. *Journal of the ACM*, 45(4):634–652, July 1998. doi: 10.1145/285055.285059. URL https://doi.org/10.1145/285055.285059.

[10] Tal Grossman and Avishai Wool. Computational experience with approximation algorithms for the set covering problem. *European Journal of Operational Research*, 101(1):81–92,

August 1997. doi: 10.1016/s0377-2217(96)00161-0. URL https://doi.org/10.1016/s0377-2217(96)00161-0.

[11] Fernando C. Gomes, Cláudio N. Meneses, Panos M. Pardalos, and Gerardo Valdisio R. Viana. Experimental analysis of approximation algorithms for the vertex cover and set covering problems. *Computers & Operations Research*, 33(12):3520–3534, December 2006. doi: 10.1016/j.cor.2005.03.030. URL https://doi.org/10.1016/j.cor.2005.03.030.

[12] Irit Dinur and David Steurer. Analytical approach to parallel repetition. In *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*. ACM, May 2014. doi: 10.1145/2591796.2591884. URL https://doi.org/10.1145/2591796.2591884.

[13] Ching Lih Lim, Alistair Moffat, and Anthony Wirth. Lazy and eager approaches for the set cover problem. In *Proceedings of the Thirty-Seventh Australasian Computer Science Conference - Volume 147*, ACSC '14, page 19–27. Australian Computer Society, Inc., 2014. URL https://dl.acm.org/doi/abs/10.5555/2667473.2667476.

[14] Graham Cormode, Howard Karloff, and Anthony Wirth. Set cover algorithms for very large datasets. In *Proceedings of the 19th ACM international conference on Information and knowledge management - CIKM '10*. ACM Press, 2010. doi: 10.1145/1871437.1871501. URL https://doi.org/10.1145/1871437.1871501.

[15] Guy E. Blelloch, Harsha Vardhan Simhadri, and Kanat Tangwongsan. Parallel and i/o efficient set covering algorithms. In *Proceedinbgs of the 24th ACM symposium on Parallelism in algorithms and architectures - SPAA '12*. ACM Press, 2012. doi: 10.1145/2312005.2312024. URL https://doi.org/10.1145/2312005.2312024.

[16] Yuval Emek and Adi Rosén. Semi-streaming set cover. *ACM Transactions on Algorithms*, 13 (1):1–22, December 2016. doi: 10.1145/2957322. URL https://doi.org/10.1145/2957322.

[17] Michael Barlow, Christian Konrad, and Charana Nandasena. Streaming set cover in practice. In *2021 Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 181–192. Society for Industrial and Applied Mathematics, January 2021. doi: 10.1137/1.9781611976472.14. URL https://doi.org/10.1137/1.9781611976472.14.

[18] Fedor V. Fomin, Dieter Kratsch, and Gerhard J. Woeginger. Exact (exponential) algorithms for the dominating set problem. In *Graph-Theoretic Concepts in Computer Science*, pages 245–256. Springer Berlin Heidelberg, 2004. doi: 10.1007/978-3-540-30559-0_21. URL https://doi.org/10.1007/978-3-540-30559-0_21.

[19] Johan M.M. van Rooij and Hans L. Bodlaender. Exact algorithms for dominating set. *Discrete Applied Mathematics*, 159(17):2147–2164, October 2011. doi: 10.1016/j.dam.2011.07.001. URL https://doi.org/10.1016/j.dam.2011.07.001.

[20] Yoichi Iwata. A faster algorithm for dominating set analyzed by the potential method. In *Parameterized and Exact Computation*, pages 41–54. Springer Berlin Heidelberg, 2012. doi: 10.1007/978-3-642-28050-4_4. URL https://doi.org/10.1007/978-3-642-28050-4_4.

[21] Olivier Goldschmidt, Dorit S. Hochbaum, and Gang Yu. A modified greedy heuristic for the set covering problem with improved worst case bound. *Information Processing Letters*, 48 (6):305–310, December 1993. doi: 10.1016/0020-0190(93)90173-7. URL https://doi.org/10.1016/0020-0190(93)90173-7.

[22] Jack Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17:449–467, 1965. doi: 10.4153/cjm-1965-045-4. URL https://doi.org/10.4153/cjm-1965-045-4.

[23] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015. URL http://networkrepository.com.