SCHOOL OF COMPUTATION,
INFORMATION AND TECHNOLOGY

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Verfication of selected NP-hard Problems

Zixuan Fan

SCHOOL OF COMPUTATION,
INFORMATION AND TECHNOLOGY

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Verfication of selected NP-hard Problems

Verifikation der ausgewählten NP-schweren Probleme

Author:	Zixuan Fan
Supervisor:	Prof. Tobias Nipkow
Advisor:	Katharina Kreuzer
Submission Date:	Submission date

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, Submission date

Zixuan Fan

Acknowledgments

Firstly, I would like to thank Prof. Tobias Nipkow, who gave me the opportunity to work on this work and introduced Isabelle to me. Then, I want to thank my advisor, Katharina Kreuzer, for all the meaningful instructions and dedicated guidance she offered. It has been a great pleasure to work with her. I also want to thank Simon Rosskopf and Kevin Kappelmann, who more or less helped in finding this topic and gave some helpful advices about the ongoing project.

Additionally, I want to thank my parents for their support of my studies in Germany. I also want to thank my brother Zichun for sample reading this thesis and providing some insights from the perspective of a mathematician.

Abstract

NP-hardness is a fundamental concept in the complexity theory. It represents a class of problems that are hard to be computed by a polynomial algorithm. Polynomial-time reductions are used to classify the NP-hardness of such problems. While proofs of the correctness of the polynomial-time reductions were limited to pen-and-paper proofs, it is now possible to reproduce and verify these proofs with the aid of computers. In order to demonstrate the capability of interactive theorem provers in verifying NP-hardness, we formalized and verified the NP-hardness of six selected problems using the well-known interactive theorem prover, Isabelle.

Contents

1. Introduction

1.1. Motivations

We may encounter many real-life problems that require a decision process to find a solution. For example, we always want to choose the shortest queue when shopping at a supermarket. Another example is Seven Bridges of Königsberg—is there a way to go through all seven bridges without visiting a bridge twice? These problems are formally defined as decision problems.

NP-hard problems is one of the most famous decision problem classes. NP-hard problems are hard to be computed by a polynomial-time algorithm. Choosing the shortest queue, known as scheduling, is a well-known example for NP-hard problems. There also many other real-life examples such as map colouring and sudoko. NP-hardness has been a fundamental research topic in theoretical computer science since the 1970s when the first few results in NP-hardness were given by Cook [Coo23], Levin [Lev73] and Karp [Kar10]. In the next few decades, many attempts were made to show whether NP-hard problems can be computed by a polynomial-time algorithm and to develop approximation algorithms that compute NP-hard problems optimally. Among many fields related to NP-hardness, we focus on the polynomial-time reductions, which show the NP-hardness of decision problems.

All the existing proofs of the correctness of polynomial-time reductions were pen-and-paper proofs, which lack automated verification by a computer. While human researchers may make mistakes in a proof, the computers are accurate once the system is correctly defined. In addition, it is also interesting to show that the computers are able to verify the first few theories that are highly related modern computers today. With the help of interactive theorem provers, it is possible to formalise and verify the classical results of NP-hardness on a computer. In this manner, we contribute to the theoretical basis of many existing formalisation results, e.g. cryptography and approximation algorithms.

There has been an attempt, known as the Karp21 project [Has23], to formalise NP-hard problems in Karp's paper in 1972 [Kar10]. Our work benefits from this attempt and continues to formalise some of the remaining problems in Karp's paper with the interactive theorem prover Isabelle.

1.2. Contributions

Our work contains two categories of problems.

1. Set covering problems: Exact Cover, Exact Hitting Set
2. Weighted sum problems: Subset sum, Partition, Knapsack, Zero-One Integer Programming

Set covering problems are problems in which we search for a cover of a given set. In weighted sum problems, we look for a set of instances such that their weighted sum reaches another constant bound. While they are the basis of further reductions to problems like scheduling, neither of the categories was formalised and verified in the existing project. Thus, we chose these problems to complete the project and prepare for potential work in the future.

For each listed problem, we present a polynomial-time reduction either from Satisfiability or another problem listed above. Thus, a trace of reductions from Satisfiability can be witnessed. Furthermore, proofs of the soundness, completeness, and the polynomial-time complexity of each polynomial-time reduction is also presented.

On the basis of our contribution, it is possible to construct and formalise polynomial-time reduction to other NP-hard problems. Additionally, it also provides the theoretical background for other formalisation works related to complexity theory, e.g. approximation algorithms, combinatorical optimization etc.

1.3. Outline

In Chapter 2, we introduce Isabelle dependencies and mathematical backgrounds.

Chapter 3 and Chapter 4 follow with the formalisation and verification of the listed problems. For each decision problem, we define the problem and the reduction. Then, we sketch the proof of the correctness of the reduction and the polynomial-time complexity. Finally, we present a few concrete implementation details. Examples are also offered for reductions that are rather complicated. In Chapter 3, we discuss the polynomial-time reduction of the set cover problems, while Chapter 4 consists of the weighted sum problems.

To finish, we conclude the current status of the Karp21 project and present a few possibilities for verifying the rest of the problems in Chapter 5.

2. Preliminaries

2.1. Isabelle and Dependencies

Isabelle/HOL and Archive of Formal Proofs

Isabelle [Wen+04] is a generic interactive theorem prover. HOL is the Isabelle's formalization of Higher-Order Logic, a logical system with inductive sets, types , well-founded recursion etc. Archive of Formal Proofs [al23] is a library of verification projects developed by Isabelle. Our work is implemented with Isabelle and used many entries from Archive of Formal Proofs.

HOL-Real_Asymp and Laudau_Symbols

HOL-Real_Asymp is a component of the Isabelle/HOL, in which the asymptotic classes are defined. Moreover, the Archive of Formal Proof entry, Landau_Symbols [Ebe15], defines the Landau's symbols and provides tool for reasoning about the asympototic growth of the functions.

NREST

NREST is part of the verification frame work Isabelle-LLVM with time [Has21]. It enables the systematic verification of asympototic complexity of imperative algorithms in Isabelle [ZH18]. We apply this framework to verify the polynomial-time complexity of iteration of unordered data structures, such as sets.

The Karp21 Project [Has23]

The project aims to formalise all of the twenty-one NP-hard problems in Karp's paper in 1972 [Kar10]. Up till now, there are eight problems of them finished, with a few related NP-hard problems that are not in Karp's list. Our work also contributes to this project, formalising six of the remaining problems. Though dependent on this project, our work only reuses a few existing definitions, while the most formalisation and verification is original. An overview of the project is given Figure 2.1. The reusable part stems from the reduction from 3CNF-Satisfiabitliy to other problems, for our work starts from Satisfiabitliy.

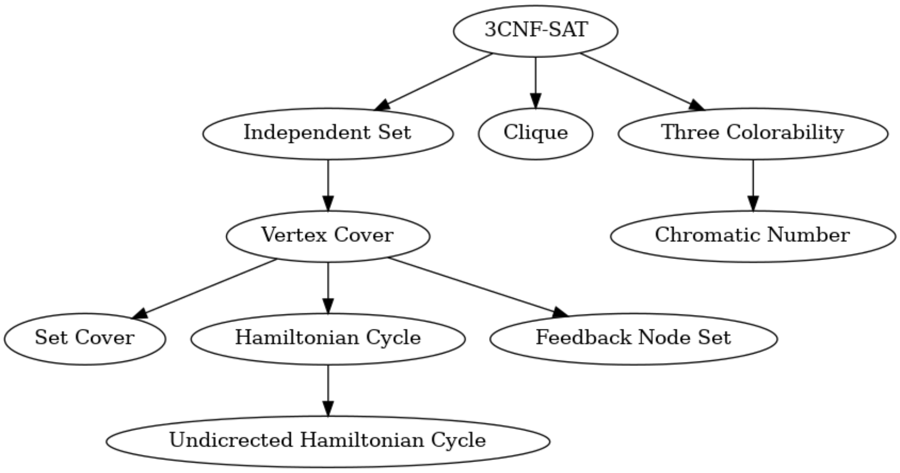

Figure 2.1.: The reduction graph of the Karp21 project

Positional Notation for Natural Numbers in an Arbitrary Base [Sta23]

This entry of Archive of Formal Proofs, short for DigitInBase, shows the uniqueness of representation of natural numbers with an arbitrary base. In other words, it proves the well-definedness of the n-ary counting systems. Our implementation benefits from this repository in showing the correctness of the polynomial-time reduction from Exact Cover to Subset Sum.

2.2. Mathematical Backgrounds

Asymptotic Notation

Conventionally, the asymptotic notation is used to define complexity classes and perform the algorithm analysis. We follow this convention and choose the big $\mathcal{O}$ notation for algorithm analysis. To begin with, we present a brief introduction to the asymptotic notation.

Definition 1. *Let $f : \mathbb{R} \to \mathbb{R}$, $g : \mathbb{R} \to \mathbb{R}$ be two real valued functions. $f(x)$ is big $\mathcal{O}$ of $g(x)$, which writes*

$$f(x) \in \mathcal{O}(g(x))$$

if there exists a real $M \geq 0$ and a real x_0 s.t.

$$|f(x)| \leq M|g(x)|, \forall x \geq x_0$$

Thus, f is above bounded by g. In other words, f is utmost as complex as g. Following this definition, we can derive many complexity classes with different g. A short list of commonly encountered complexity classes is given in Table 1.

Name	Big $\mathcal{O}$ Notation	Algorithmic Examples
Constant	$\mathcal{O}(1)$	Parity check
Logarithmic	$\mathcal{O}(\log n)$	Binary search in a sorted array
Linear	$\mathcal{O}(n)$	Addition of integers
Quasilinear	$\mathcal{O}(n \log n)$	Merge-sort and heap-sort
Polynomial	$\mathcal{O}(n^c), c \in \mathbb{N}$	Matrix multiplication
Factorial	$\mathcal{O}(n!)$	Enumeration of partitions of a set

Table 2.1.: List of commonly encountered complexity classes.

In this work, we only consider the polynomial class. For simplicity, we did not formalise the theory of asymptotic classes and big $\mathcal{O}$ notation but used the available Isabelle dependencies of **HOL-Real_Asymp** and **Landau_Symbols**.

Decision Problems

Definition 2. *A decision problem is a yes-no question on an infinite set of fixed type of inputs.*

Generally, if we refer to a decision problem A, we refer to the set of inputs whose answer to the yes-no question is yes. We follow this convention throughout this work. The handling of a decision problem usually involves two questions:

1. Is there a terminating algorithm which computes the solution to this problem?
2. If the answer to the first question is yes, is this algorithm efficient?

If the answer to the first question is yes for a problem, it is a decidable problem, otherwise it is non-decidable. We do not expect a yes-no answer for the second question, but would like to find the optimal complexity for the algorithm, e.g. logarithmic complexity, polynomial complexity, etc. While some problems are possible to be computed in an optimal compleixty by a deterministic algorithm, there are also a few problems, for which no deterministic polynomial-time algorithm is found. We limit these problems and given them a formal definition.

Definition 3. *If there is a deterministic algorithm that decides the solution to the problem in polynomial time, the problem is in the complexity class **P**. If this polynomial algorithm is non-deterministic, the problem is in the complexity class **NP**.*

Definition 4. *If a problem is at least as complex as the most complex problems in **NP**. It is in the complexity class NP-hard.*

Instead of trying to prove or reject the existence of a non-deterministic algorithm for the NP problems, we focus on the NP-hardness. We would like to formally prove that many classical decision problems are NP-hard. For this reason, we have to introduce polynomial-time reductions.

Polynomial-time Reductions

Given two decision problems A and B, a reduction is a function $f : A \to B$, which maps the instances of A to those of B. A reduction is a polynomial-time reduction if and only if the reduction function has a polynomial-time complexity. For a polynomial-time reduction from A to B, we write $A \leq_p B$.

Let M and N denote the universe of A and B. A function $g : M \to N$ is a polynomial-time reduction if and only if the following conditions are satisfied.

$$x \in A \iff g(x) \in B$$
$$\exists k \in \mathbb{N}. g \in \mathcal{O}(n^k)$$

For convenience, we separate (2.1) into the soundness and completeness of the reduction.

$$\begin{aligned} \textit{soundness}: &\quad x \in A \Longrightarrow g(x) \in B \\ \textit{completeness}: &\quad g(x) \in B \Longrightarrow x \in A \end{aligned}$$

NP-Hardness and Satisfiability

To show a decision problem B is NP-hard, we have to find a NP-hard problem A and polynomial-time reduction such that $A \leq_p B$. The first proven NP-hard problem is Satisfiability, which was independently proven by Cook in 1971 [Coo23] and Levin in 1973 [Lev73]. The Satisfiability problem is defined by

Definition 5. *Satisfiability*
Input*: A propositional logical formula in conjunctive normal form.*
Output*: Is there a valid assignment for this formula?*

In the previous implementation of the project, Satisfiability is defined with a list of clauses, with the clauses as sets of variables. Our first reduction also stems from Satisfiability, while all the other reductions are constructed upon novel introduced problems. More details on the reduction and implementation are given in Chapter 3 and Chapter 4. A glimpse of the available definition is given by Figure 2.2.

In this definition, a literal is defined as either a positive or negative existence of the variables. Then the type **three_sat** is defined. It is technically not a 3-Satisfiability instance, for each clause can contain more than three literals. This naming mistake is a historical legacy, which should be renamed after negotiation with the original contributors. After this, the definitions of **lifting**, **models**, and **sat** are given. They describe how an assignment is checked over the propositional logical formula. Finally, we add the definition of **cnf_sat**, upon which our first reduction is defined.

```
datatype 'a lit = Pos 'a | Neg 'a

type_synonym 'a three_sat = "'a lit set list"

definition lift :: "('a ⇒ bool) ⇒ 'a lit ⇒ bool" ("_↑" 60)
where
  "lift σ ≡ λl. case l of Pos x ⇒ σ x | Neg x ⇒ ¬ σ x"

definition models :: "('a ⇒ bool) ⇒ 'a three_sat ⇒ bool" (infixl
"⊨" 55) where
  "σ ⊨ F ≡ ∀cls ∈ set F. ∃l ∈ cls. (σ↑) l"

definition sat :: "'a three_sat ⇒ bool" where
  "sat F ≡ ∃σ. σ ⊨ F"

definition"cnf_sat ≡ {F. sat F ∧ (∀cls ∈ set F. finite cls)}"
```

Figure 2.2.: Definition of Satisfiability

2.3. Application of NREST and Paradigm

The NREST package offers an approach for approximating the complexity of non-deterministic processes. This is especially useful when iterating a set, a collection or any other unordered data structures. Thus, we use this package throughout this work. In our complexity analysis, the following commands are used.

- *RETURNT* **res**. A command that returns the result **res**. It costs exactly one unit of time.
- *SPECT* **[cond → cost]**. A command used for checking a condition. Checking the condition **cond** take **cost** units of time.
- *SPEC P Q*. A command used for assignment. Should $P\ x$ hold for an object x, it is a valid object after the assignment The assignment then takes $Q\ x$ units of time.

To apply the NREST approach in the complexity analysis, it is necessary to rewrite the algorithm with the NREST commands. In our implementation, it means to convert the reduction function into an algorithm implemented with NREST commands. During the conversion, we follow the following principles for complexity analysis.

1. Checking the condition always costs exactly one unit of time.
2. During the iteration, it costs one unit of time each for access, modification and insertion of the set elements.
3. All other operations should cost one unit of time, if not stated explicitly.

Then, it is possible show the property of the reduction with this approach. In addition to the polynomial-time complexity, the existing definition of polynomial-time reduction in the Karp21 project requires a polynomial-space complexity, too. We are consistent with this definition, and hence verified the polynomial-space complexity in each reduction as well. Let f denote a polynomial-time reduction from A to B, while f_{alg} denotes the NREST version of the reduction. Furthermore, we define sizing functions s_A and s_B as metrics for the asymptotic classes. To show that the reduction is polynomial, we show that the reduction is polynomially bounded in terms of time and space, which are respectively the *refines* and the *size* lemma.

$$\begin{aligned} refines: \quad & f_{alg}(A) \leq \mathcal{O}((s_A(A))^k) \\ size: \quad & s_B(f(A)) \leq \mathcal{O}((s_A(A))^k) \end{aligned}$$

In the end, we can conclude the following implementation paradigm to show that a reduction is correct and polynomial.

1. Implement the reduction in Isabelle/HOL.
2. Prove that the reduction is correct.
3. Implement the reduction in NREST commands.
4. Prove that the reduction costs polynomial time.
5. Prove that the algorithm costs polynomial space.

3. Set Covering Problems

In this chapter, we discuss about the NP-hardness of a few set covering problems. Set covering problems ask whether a certain combinatorical structure A covers another structure B. Alternatively, it may also ask for the minimal size of A to cover B. We focus on a subclass of covering problems, the exact covering problem. In this subclass, A covers B exactly, i.e. no element in B is covered twice in A. In Karp's paper in 1972 [Kar10], the following covering problems were included: Exact Cover, Exact Hitting Set, 3-Dimensional Matching, and Steiner Tree. In our implementation, we reduced Satisfiability to Exact Cover, and then reduced Exact Cover to Exact Hitting Set.

3.1. Exact Cover

The Exact Cover problem is a special case of the Set Cover. Exact Cover problems not only look for a cover, but also require the existence of covered elements to be unique.

Definition 6. *Exact Cover*
Input: *A set X and a collection S of subsets of X.*
Output: *Is there a disjoint subset S' of S s.t. each element in X is contained in one of the elements of S'?*

$$\begin{aligned}\textbf{Exact Cover} := \{(X, S) \mid \bigcup S \subseteq X \wedge \exists S' \subseteq S.\\ \forall x \in X. \exists s \in S'. x \in s\\ \wedge \forall s\ t \in S'. s \neq t \longrightarrow s \cap t = \emptyset\}\end{aligned}$$

We call (X, S) an instance of Exact Cover and S' an exact cover of X. In other words, S' covers X exactly.

3.1.1. Choice of Reduction

Since the Exact Cover problem is a fundamental NP-hard problem, there are many different reductions found by different researchers. Karp's reduction[Kar10] is based on the Chromatic Number problem. Although the Chromatic Number problem was formalised in Karp's 21 project, we did not choose this reduction because of the complexity of the graph traversal and the differences between Karp's definition and the available Isabelle's definition. In the available definition of Chromatic Number, the Chromatic Number k is limited to be at least three. Since the Chromatic Number is in $\boldsymbol{P}$ in the case $k = 2$, the alternative definition is correct. However, this raises the difficulty of performing a reduction, for it is necessary to consider a special case for $k = 2$.

On the contrary, there is an easy reduction from Satisfiability to Exact Cover [19]. This reduction does not involve graph traversal. The only technical barrier is the typeless set. While Isabelle only supports typed sets, we resolve this problem by creating an encapsulation type. More details follow in the Section 3.1.3.

3.1.2. Reduction Details

Now we present the reduction from Satisfiability to Exact Cover. Before defining the reduction function, we first state the notations that we use for this part. Given a propositional logical formula F, $vars\ F$ denotes the set of variables in F. An assignment σ is a function that maps a variable to a true-false value. With $\top$ as true and $\bot$ as false, $\sigma(x_i) = \top$ reads x_1 is true under the assignment σ.

The assignment is also defined over the formula F. F is satisfiable under σ if there is at least one variable that is true under σ in each clause of F. This writes $\sigma \models F$.

Additionally, we index the variables and the clauses and use the following notations.

1. x_i denotes the i-th variable in the formula with $x_i \in vars\ F$
2. c_i denotes the i-th clause in the formula with $c_i \in F$
3. p_{ij} denotes the j-th position/literal in the i-th clause with $p_{ij} \in c_i$

Then we construct a set X_F and which contains all three different kinds of objects.

$$X_F = vars\ F \cup F \cup \bigcup_{c_i \in F} c_i$$

Furthermore, we construct S_F, a collection of subsets of X_F. We determine the following subsets

1. $\{p_{ij}\}$. The unary set of positions
2. $\{c_i, p_{ij}\}$. The binary set of a clause and a position inside the clause.
3. $pos(x_i) := \{x_i\} \cup \{p_{ab} | p_{ab} = x_i\}$. The set of a variable with its positive occurrences as positions.
4. $neg(x_i) := \{x_i\} \cup \{p_{ab} | p_{ab} = \neg x_i\}$. The set of a variable with its negative occurrences as positions.

S_F contains all of the four types of subsets.

$$\begin{aligned} S_F = &\{p_{ij} | p_{ij} \in c_i, c_i \in F\} \cup \{\{c_i, p_{ij}\} | p_{ij} \in c_i, c_i \in F\} \\ &\cup \{\{x_i\} \cup \{p_{ij} | p_{ij} \in c_i, c_i \in F\} | x_i \in vars\ F, x_i = p_{ij}\} \\ &\cup \{\{x_i\} \cup \{p_{ij} | p_{ij} \in c_i, c_i \in F\} | x_i \in vars\ F, \neg x_i = p_{ij}\} \end{aligned}$$

Definition 7 (Reduction SAT to XC)**.** *Given an instance of Satisfiability F, it is reduced to an instance of Exact Cover (X_F, S_F) as presented above.*

After definiting the reduction function, we also show that the reduction is correct and in polynomial-time.

Lemma 1 (Soundess)**.** *Let F be a satisfiable propositional logical formula. The pair (X_F, S_F) is then an instance of the exact cover.*

Proof. Let $\sigma \models F$ be a valid assignment. We construct an exact cover $S' \subseteq S_F$ of X_F in the following steps.

1. For each $x_i \in \mathit{vars}\ F$, $\mathit{pos}(x_i)$ is included in S' when $\sigma(\mathit{pos}(x_i)) \equiv \bot$. Otherwise we insert $\mathit{neg}(x_i)$ into S'. This steps covers all variables and all positions that are false under σ.

2. For each $c_i \in F$, we choose the minimal j with $\sigma(p_{ij}) \equiv \top$, and insert $\{c_i, p_{ij}\}$ into S'. This steps covers all clauses and one position that is true under σ in each clause.

3. For each $p_{ij} \in c_i$, if $\sigma(p_{ij}) \equiv \top$ and $\{c_i, p_{ij}\}$ is not in S', the unary set $\{p_{ij}\}$ is included. This steps covers all positions that are true under σ and are not covered in the previous step.

Now it suffices to show that S' is disjoint in our construction. Obviously, each position in $\mathit{pos}(x_i)$ and $\mathit{neg}(x_i)$ is false under the assignment σ, while the positions in the other sets are all true. By design, the positions in the second and the third steps never duplicate. Thus, no same position occurs in two different sets in the collection S'. Furthermore, since each clause and variable is unique in the construction, no two sets contain the same clause or the same variable. Hence S' is disjoint. ☐

For the correctness, we have to show the completeness, too.

Lemma 2 (Completeness)**.** *Let (X_F, S_F) be reduced from F. If (X_F, S_F) is an instance of the Exact Cover, F has to be satisfiable.*

Proof. It is easy to reconstruct the model σ with the same approach as in the proof of the soundness. We iterate over the variables. For each variable x_i, if $\mathit{pos}(x_i) \in S'$, we set $\sigma(x_i) = \bot$. Otherwise, $\mathit{neg}(x_i)$ has to be in S', for x_i is covered exactly. In this case, we set $\sigma(x_i) = \top$.

Now, we show that this assignment is valid. For each clause c_i, we obtain the unique set $\{c_i, p_{ij}\} \in S'$. There are two cases for p_{ij}.

1. $p_{ij} = x_k$. $\mathit{neg}(x_k)$ is then covered in S', resulting in $\sigma(p_{ij}) = \sigma(x_k) = \top$.

2. $p_{ij} = \neg x_k$. $\mathit{pos}(x_k)$ is then covered in S. resulting in $\sigma(p_{ij}) = \sigma(\neg x_k) = \neg\sigma(x_k) = \top$

Thus, for each clause, there is at least one position that is true under σ. Consequently, σ is a valid assignment of F. $\square$

In the end, we present a complexity analysis.

Lemma 3 (Polynomial Complexity)**.** *The construction of (X_F, S_F) from F can be computed within polynomial time.*

Proof. In the reduction, we have to iterate all of the variables, the clauses and the positions. Thus, we have to find a polynomial bound with regards to all of the three metrics. Let n, m and k denote the number of variables, clauses and postitions. To obtain the three metrics, we have to iterate all of the clauses, resulting the complexity of $\mathcal{O}(m)$. With these iterations, we can construct the set X_F, indicating a linear complexity for the construction of X_F.

Now the interesting part is the collection S. For each type of subsets, we present a polynomial bound

1. $\{p_{ij}\}$. It is sufficient to iterate the positions, requiring the complexity of k.
2. $\{c_i, p_{ij}\}$. Each clause is iterated for $|c_i|$ times. Since $|c_i|$ is a constant, there is a $c \in \mathbb{N}$ s.t. $|c_i| \leq c$. Then, the complexity is above bounded by $c \cdot m$.
3. $pos(x)$ and $neg(x)$. For each variable x, it is required to iterate all of the positions, which produces the complexity of $2 \cdot nk$ in total.

Thus, the construction of S_F costs the polynomial complexity of $k + cm + 2nk \in \mathcal{O}(nk + m)$. With the linear complexity of the construction of X_F, we conclude that the reduction has the quadratic complexity. $\square$

In the end, we summarize the lemmas to obtain the central theorem.

Theorem 1. *The presented reduction is a polynomial-time reduction from Satisfiability to Exact Cover.*

3.1.3. Implementation Details

Choice of Definitions

A definition of Exact Cover, as shown in Figure 3.1, states the disjointness with nested quantifiers. It is an implementation of the mathematical definition in Definition 6. Three different conditions have be satisfied for an instance of exact cover. The first condition $finite\ X$ checks if X is a finite set, while the second condition $\bigcup S \subseteq X$ limits S to be a collection of X. The third condition states about the existence of the exact cover. Although this definition gives us a direct view of the problem, it is lengthy for the further implementation. Hence we choose to use an alternative definition in Figure 3.2 with the help of the dependency **HOL-Library.Disjoint_Sets**. With the pre-defined predicate **disjoint**, we managed to simplify the definition. Moreover, we also benefitted from reusing the existing lemmas in the relating dependency.

```
definition "exact_cover_orig ≡  {(X, S). finite X ∧ ⋃S ⊆ X ∧
    (∃S' ⊆ S. ∀x ∈ X. ∃s ∈ S'. x ∈ s
    ∧ (∀t ∈ S'. s ≠ t ⟶ x ∉ t) )}"
```

Figure 3.1.: A first definition of Exact Cover

```
definition "exact_cover ≡ {(X, S).
  finite X ∧ ⋃S ⊆ X ∧ (∃S' ⊆ S. ⋃S' = X ∧ disjoint S')}"
```

Figure 3.2.: A second definition of Exact Cover

Encapsulation Type for the Construction

Intuitively, the variables, positions and clauses can be represented by tuple of indices. However, it is not easy to unify the representation such that they are of the same type. For example, a unary set of position is representable with one single index, whereas a binary set of a clause and a position needs at least two indices. On the contrary, an encapsulation type is exempted from the tedious handling of tuples. Thus, we implement this encapsulation type **xc_element** with the help of **datatype** keyword from Isabelle, as shown in Figure 3.3 The constructor **V, C** and **L** stands for the variables,

```
datatype 'a xc_element = V 'a | C "'a lit set" | L "'a lit" "'a
lit set"
```

Figure 3.3.: Definition of the encapsulation type

clauses and literals. **V** encapsules the variable, while **C** encapsules the clauses. The definition of **L** is a bit different. To locate the literals exactly, we have to encapsule both the literal and the clause. The reason is that literals are also located by clauses. If we discard the clauses, the same literal may exist in many clauses, whereas there is only one literal covered in our construction.

Non-deterministic Construction of a Sub-collection

When constructing a cover S', we have included $\{c_i, p_{ij}\}$, where j is the smallest number such that p_{ij} is true under the assignment σ. However, in our implementation, we did not use an integer to index the positions. For this reason, we cannot deterministically choose a suitable p_{ij}. The predicate **SOME** resolves this problem—**SOME** x. $P\ x$ returns an arbitrary x that satisfies the predicate P. Using this predicate, we are able to choose $\{c_i, p_{ij}\}$ non-deterministically, as defined in **constr_cover_clause** in Figure 3.4. Another benefit of this approach is that we can also include $\{p_{ij}\}$ simultaneously. Thus, we are exempted from introducing another function to compute this sub-collection. Apart

```
definition constr_cover_clause
  :: "'a lit set ⇒ ('a ⇒ bool) ⇒ 'a xc_element set set" where
"constr_cover_clause c σ =
  (SOME s. ∃p ∈ c. (σ↑) p ∧ s = {{C c, L p c}} ∪ {{L q c} | q. q
∈ c ∧ q ≠ p ∧ (σ↑) q})"

lemma constr_cover_clause_unfold:
assumes "σ ⊨ F" "c ∈ set F"
shows "∃p∈c. (σ↑) p ∧ constr_cover_clause c σ = {{C c, L p c}}
∪ {{L q c} | q. q ∈ c ∧ q ≠ p ∧ (σ↑) q}"
```

Figure 3.4.: Non-deterministic Construction using SOME

from the definition, the lemma **constr_cover_clause_unfold** is also proven to remove the **SOME** predicate and obtain the property of the encapsuled sub-collections.

The rest of the proof of the correctness follows the mathematical proof precisely. It involves showing the disjointness and covering of each part of the construction. Though it is bit lengthy, the general idea is clear and straightforward.

Polynomial-time Complexity

In the complexity analysis, it is necessary to determine the metrics on which the complexity is dependent. For a propositional logical formula F, we will iterate all variables, clauses and positions. Hence all of them are needed as metrics. Nevertheless, the NREST implementation does not support a complexity bound with different metrics. As a result, we choose the maximum of all metrics and use it as our sole metrics[1]. Similarly, we define $max|X||S|$ as the metrics for the instance of exact cover (X, S). Then we can define the NREST algorithm in Figure 3.5.

Let n be the metric for the Satisfiability problem. A comprehensive list of the complexity of each operation is given in Table 3.1. Since we follow the principles as stated in Chapter 2, all complexity are easy to follow except for **mop_literals_of_sat**, **mop_var_true_literals** and **mop_var_false_literals**. In all exceptional cases, we have to iterate the set of clauses, even though it is not necessary in the pen-and-paper reduction, because we defined the literals with the help of the clauses instead of the indices. Consequently, the resulting complexity is cubic instead of quadratic as stated in the reduction details. However, it is still a polynomial class, which is an acceptable result.

The main part of the proof of the polynomial-time complexity is automated following

[1] A few previous reductions in Karp21 project use the number of clauses as a metric. Since those reductions iterated only the clauses, this choice of metric is not an issue.

```
definition "sat_to_xc_alg ≡ (λF.
 do
    {
      VS ← mop_vars_of_sat F;
      CS ← mop_clauses_of_sat F;
      LS ← mop_literals_of_sat F;
      s1 ← mop_literal_sets F;
      s2 ← mop_clauses_with_literals F;
      s3 ← mop_var_true_literals F;
      s4 ← mop_var_false_literals F;
      X ← mop_union_x CS VS LS;
      S ← mop_union_s s1 s2 s3 s4;
      RETURNT (X, S)
    }
)"
```

Figure 3.5.: The NREST version of the reduction, SAT to XC

Operation	Functionality	Complexity
mop_vars_of_sat	collecting variables	$3n$
mop_clauses_of_sat	collecting clauses	$3n$
mop_literals_of_sat	collecting literals	$4n$
mop_literal_sets	encapsuling literals	$3n$
mop_clauses_with_literals	non-deterministic construction	$6n$
mop_var_true_literals	obtaining variables with negative existences	$3n^3$
mop_var_false_literals	obtaining variables with positive existences	$3n^3$

Table 3.1.: Complexity of operations in reduction SAT to XC.

the style of previous works from the Karp21 project. Nevertheless, automation failed when it came to show the upperbound of the cardinality of the constructed sets, such as **card_comp_S** in Figure 3.6. The function **comp_S** constructs the collection S in the instance of exact cover. This lemma shows that the cardinality of the collection S is above bounded by the quardruple of our metric.

We analyzed the proof and found out the reason to be cardinality function. Whenever we want to show the (in)equality of cardinality between two sets, there are a few assumptions for the usable lemmas. In most cases, these premises need to be manually passed to the lemmas. One frequently used lemma with such property is **card_image** in Figure 3.6. It requires to find an injective function between the preimage and the image. Without passing the premises to this lemma, we cannot obtain an inequality relationship.

```
lemma card_comp_S:
  "card (comp_S F) ≤ 4 * (size_SAT_max F)"

lemma card_image: "inj_on f A ⟹ card (f ‘ A) = card A"
```

Figure 3.6.: Details in the proof of the polynomial-time complexity, SAT to XC

3.1.4. Example of the reduction

In this final part, we present an example with a bried explanation how the reduction is correct for this certain example.

Input: A logical formula in conjunctive normal form

$$F := (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2) \wedge (x_1 \vee x_3)$$

Output: The constructed set is

$$X := \{x_1, x_2, x_3\} \cup \{c_1, c_2, c_3\} \cup \{p_{11}, p_{12}, p_{13}, p_{21}, p_{22}, p_{31}, p_{32}\}$$

The constructed collection is

$$\begin{aligned} S := &\{\{p_{11}\}, \{p_{12}\}, \{p_{13}\}, \{p_{21}\}, \{p_{22}\}, \{p_{31}\}, \{p_{32}\}\} \\ &\cup \{\{c_1, p_{11}\}, \{c_1, p_{12}\}, \{c_1, p_{13}\}, \{c_2, p_{21}\}, \{c_3, p_{31}\}, \{c_3, p_{32}\}\} \\ &\cup \{\{x_1, p_{11} p_{31}\}, \{x_1, p_{21}\}, \{x_2, p_{12}, p_{22}\}, \{x_2\}, \{x_3, p_{32}\}, \{x_3, p_{13}\}\} \end{aligned}$$

Validity: Apparently, the only valid assignment σ of F is given by

$$\sigma = \{x_1 = \bot, x_2 = \top, x_3 = \top\}$$

We construct an exact cover S' by

$$S' = \{\{c_1, p_{22}\}, \{c_2, p_{21}\}, \{c_3, p_{32}\}, \{x_1, p_{11}, p_{31}\}, \{x_2\}, \{x_3, p_{31}\}, \{p_{22}\}\}$$

3.2. Exact Hitting Set

The hitting set problems are variants of the set covering problems. Essentially, they are two different ways of viewing the same problem. Just as the Hitting Set[2] is a variant of Set Cover, the Exact Hitting Set is a variant of Exact Cover.

Definition 8. *Exact Hitting Set*
Input: *A collection of sets S*
Output: *Is there a finite set W s.t. the intersection of W and each element $s \in S$ contains exactly one element?*

[2]Note that the exact hitting set problem was referred to as the hitting set problem in Karp's work, whereas it is generalized to be another problem nowadays.

Exact Hitting Set $:= \{C \mid \exists W.\ \forall c \in C.\ |W \cap c| = 1\}$

We call W to hit C exactly.

3.2.1. Reduction Details

We use the reduction from Karp's work. Given an instance of Exact Cover (X, S), the instance of Exact Hitting Set C_{XS} is constructed by

$$C_{XS} = \{\{s|u \in s, s \in S\}|u \in X\}$$

Thus, C_{XS} is the set of sub-collections denoted by $c_u = \{s|u \in s, s \in S\}$. All sets in c_u share the same element u.

Definition 9 (Reduction XC to XHS)**.** *Given an instance of Exact Cover F, it is reduced to an instance of Exact Hitting Set C_{XS} as presented above.*

It is possible to show the correctness of the reduction with this defintiion.

Lemma 4 (Soundess)**.** *Let (X, S) be an instance of Exact Cover. A collection C_{XS} reduced from (X, S) is then an instance of the exact hitting set.*

Proof. For the soundness of the reduction, it suffices to show

$$\exists W.\ \forall c_u \in C_{XS}.\ |W \cap c_u| = 1$$

Let $W = S'$, where S' covers X exactly. Moreover, let c_u be an arbitrary element of C_{XS}. Since S' covers X exactly, there is exactly one $s \in S'$ that contains u. c_u is a collection of sets in S that contain u. Thus, s is the only element in the set $S' \cap c_u$. Consequently, S' hits C_{XS} exactly. □

Then, we show the completeness.

Lemma 5 (Completeness)**.** *Let the collection C_{XS} be a collection reduced from a pair (X, S). If C is an instance of the exact hitting set, (X, S) has to be an instance of Exact Cover.*

Proof. The proof of the completeness shares a similar construction. The only difference is that W is not necessarily a subset of S. Nevertheless, there exists a subset $W' \subseteq W$ s.t. it is not only a subset of S, but it also satisfies the property $\forall c_u \in C_{XS}.|W' \cap c_u| = 1$.

Let $S' = W'$. For each $u \in X$, there is exactly one set $s \in W'$ s.t. $u \in s$. This ensures that X is fully covered and each element in X is existent in only one set. Moreover, W' is a subset of S, indicating that all sets in W only contain elements of X. Hence W' is also disjoint. □

With the correctness proven, we show that the reduction is in polynomial time.

Lemma 6 (Polynomial Complexity). *The construction of C_{XS} from (X, S) can be computed within polynomial time.*

Proof. In our reduction, it is necessary to iterate the set X and the collection S in a nested loop. With the cardinality $|X|$ and $|S|$ as the metrics, it is obvious that the reduction costs the complexity of $\mathcal{O}(|X||S|)$. $\square$

In the end, we summarize the lemmas to obtain the main theorem.

Theorem 2. *The presented reduction is a polynomial-time reduction from Exact Cover to Exact Hitting Set.*

```
definition "exact_hitting_set ≡
  {S. ∃W. finite W ∧ (∀s ∈ S. card (W ∩ s) = 1)}"

definition "xc_to_ehs ≡ λ(X, S). if finite X ∧ ⋃S ⊆ X
  then {{s. u ∈ s ∧ s ∈ S} | u. u ∈ X}
  else {{}}"
```

Figure 3.7.: Definition of the reduction, XC to XHS

3.2.2. Implementation Details

Definition of the reduction

Since the Exact Cover problem is defined over a finite set X and a finite collection S, we have to check if the X and S are finite and if S is a collection of X. Thus, a condition statement which checks this requirement is added to the implementation in Figure 3.7

Polynomial-time Complexity

Operation	Functionality	Complexity
mop_check_finiteness_and_is_collection	checking the requirement	1
mop_construct_sets	constructing the new instance	$3n^2$

Table 3.2.: Complexity of operations in reduction XC to XHS.

We determine the size of the instance of Exact Hitting Set C as $|C|$. According to the paradigm, we define the NREST algorithm in Figure 3.9. In the same way as the implementation for the reduction from Satisfiability to Exact Cover, we can only use one single metric. Let $n = max|X||S|$ be the metric for Exact cover. The complexity of each operation is given in Table 3.2. Just as concluded in the pen-and-paper proof, the

```
definition "xc_to_ehs_alg ≡ λ(X, S).
  do {
    b ← mop_check_finiteness_and_is_collection (X, S);
    if b
    then do {
      S' ← mop_construct_sets (X, S);
      RETURNT S' }
    else do {
      RETURNT {{}} }}"
```

Figure 3.8.: The NREST version of the reduction, XC to XHS

```
lemma card_ehs_le:
assumes "finite X" "card X ≤ Y"
shows "card {{s. u ∈ s ∧ s ∈ S} |u. u ∈ X} ≤ Suc (Y * Y)"
```

Figure 3.9.: Details in the proof of the polynomial-time complexity, XC to XHS

reduction costs quadratic time.

The proof of the polynomial complexity is mostly automated after unfolding the necessary definitions. However, an additional step is required for indicating the relationship between the sizing functions. While it holds $|C| = |X|$, the size of the Exact Cover problem is defined by $max|X||S|$ instead of $|X|$. Hence we can only conclude that the size of the Exact Hitting Set is less equal than the size of the Exact Cover. The proof automation will then fails in showing $|C| \leq |S| \cdot |S| + 1$ when $|S| \geq |X|$. For this reason, we have to prove one additional lemma showing this property in Figure 3.9.

4. Weighted Sum Problems

In this chapter, we discuss the NP-Hardness of a few weighted sum problems. In weighted sum problems, there is a weighting function w over a set S. The problems ask about whether the weight of elements in S satisfies an equality or inequality relationship. The weighted sum problems are also mathematical programming problems, which is another large discipline of NP-hard problems. In Karp's paper, there were Partition, Subset Sum [1], and Knapsack introduced. We present reductions from Exact Cover to Subset Sum, from Subset Sum to Partition, Knapsack and Zero-one Integer Programming.

4.1. Subset sum

We define Subset Sum with a set and a weighting function. There is also an alternative definition using a multiset or a sequence without a weighting function, which is useful for other reductions in this work. More details follow in the Partition section.

Definition 10. *Subset Sum*
Input: *A finite set S, a weighting function w, and an integer B*
Output: *Is there a subset $S' \subseteq S$ s.t.*

$$\sum_{x \in S'} w(x) = B \tag{1}$$

$$\textbf{Subset Sum} := \{(S, w, B) | \exists S' \subseteq S. \sum_{x \in S'} w(x) = B\}$$

4.1.1. Reduction Details

We reduce Exact Cover to Subset Sum. We start with Exact Cover problems over natural numbers. Given (X, S) an instance of Exact Cover over natural numbers, let S be the set in the entry of Subset Sum. Then we define the weighting function w_p and the sum B_X by

$$w_p(s) = \sum_{x \in s} p^x, B_X = w(X)$$

where p is a natural number no less than $|S|$

[1]What Karp presented was referred to as Knapsack, although the definition is closer to the Subset Sum nowadays

Definition 11 (Reduction XC to SS)**.** *Given an instance of Exact Cover over natural numebers (X, S), it is reduced to an instance of Subset Sum (S, w_p, B_X) as presented above.*

With this definition, we show the correctness of the reduction.

Lemma 7 (Soundness)**.** *If (X, S) is an instance of Exact Cover. The reduced (S, w_p, B_X) is then an instance of Subset Sum.*

Proof. Let the $S' \subseteq S$ be an exact cover of X. It then holds that

$$\sum_{s \in S'} w_p(s) = \sum_{s \in S'} (\sum_{x \in s} p^x) = \sum_{x \in \bigcup S'} p^x = \sum_{x \in X} p^x = B_X$$

Thus, (S, w_p, B_X) is an instance of Subset Sum. □

Lemma 8 (Completeness)**.** *Let (S, w_p, B_X) be reduced from (X, S). If (S, w_p, B_X) is an instance of Subset Sum, (X, S) has to be an instance of Exact Cover.*

Proof. Obtain $S' \subseteq S$ for which **(1)** holds. We show the disjointness of S' by contradiction. Assume that S' is not disjoint. Then there exist $s_1, s_2 \in S'$ s.t. $s_1 \cap s_2 \neq \emptyset$. Let $a \in s_1 \cap s_2$ be arbitrary. Then there are two cases for the coefficient c_a of p^a in the polynomial $\sum_{x \in S'} w_p(x)$ of p.

1. $c_a \neq 1$. This case is invalid. It is obvious that the p-nary representation of a natural number is unique. Since **(1)** holds and $B_X = \sum_{x \in X}^{p^X}$, the coefficient c_a has to be exactly one.
2. $c_a = 1$. Though **(1)** is satisfied, there are still at least two p^a in the polynomial $\sum_{x \in S'} w_p(x)$. One of them stems from s_1, the other from s_2. Hence the number of p^a in this polynomial is at least p. However, there are utmost $|S|$ elements in S', meaning that there are utmost $|S|$ such p^a. From the fact the $p > |S|$, it is not possible that the number of p^a is greater equal p. As a result, this case is also invalid.

In conclusion, the assumption is false, and S' is disjoint. Then, it follows directly from the disjointness that S' covers X exactly, otherwise **(1)** is not satisfied. □

This reduction is, however, limited to the Exact Cover over natural numbers. It is still necessary to generalize the reduction. For this reason, we need to construct a mapping.

Lemma 9. *Let S be an arbitrary finite set. Then there exists a bijective function $f, S \to N$ s.t.*

$$f(S) = \begin{cases} \emptyset & S = \emptyset \\ \{0, 1, ..., |S| - 1\} & \textit{otherwise} \end{cases}$$

Proof. Trivial. □

With this approach, we are able to generalize Exact Cover and covert each instance to an equivalent instance over natural numbers.

Definition 12 (Reduction XC to SS, updated)**.** *Given an instance of Exact Cover* (X, S), *we first map it into natural numbers using the bijective function* f. *The resulting instance of Exact Cover* (X_f, S_f) *is reduced to an instance of Subset Sum* (S_f, w_p, B_X) *as presented above.*

Then, we perform a complexity analysis over the whole construction.

Lemma 10 (Polynomial Complexity)**.** *The construction of* (S_f, w, B) *from* (X, S) *can be computed within polynomial time.*

Proof. When we map an arbitrary set into a natural number set as presented in Lemma 9, we have to iterate over the set X, which costs the complexity of $|X|$. Furthremore, we have to iterate S to construct w and B, resulting in the complexity of $2|S|$. In total, it costs $|X| + 2|S| \in \mathcal{O}(|X| + |S|)$, i.e. the linear complexity. □

Finally, we summarize to obtain the main theorem.

Theorem 3. *The presented reduction is a polynomial-time reduction from Exact Cover to Subset Sum.*

4.1.2. Implemantation Deatails

The Isabelle definition of Subset Sum is given in Figure 4.1. It is easy to notice that the definition is identical to the pen-and-paper definition.

```
definition "is_subset_sum SS ≡
  (case SS of (S, w, B) => (sum w S = B))"
definition "subset_sum ≡
  {(S, w, B) | S w B. finite S ∧ (∃S' ⊆ S. is_subset_sum (S', w,
B))}"
```

Figure 4.1.: Definition of Subset Sum

Definition of the Reduction

In implementation shown in Figure 4.2, we started with the construction as presented in Lemma 9. The function **map_to_nat** returns a mapping function with the property in Lemma 9, in whose definition the predicate **SOME** is used again. Following this, we define the weighting function and reduction function. Similar to what happened to Exact Hitting Set, we also check if X is finite and if S is a collection before we perform a reduction, for these conditions are a requirement under our definition. Apart from the definition, we need a lemma that converts the sum to a polynomial. To guarantee that p^k is unique for an arbitrary k, we require that $p \geq 2$ as in Figure 4.3.

```
definition "map_to_nat X ≡ (SOME f. (if X = {} then bij_betw f X {}
else bij_betw f X {1..card X}))"
definition "weight p X ≡ (sum (λx. p ^ x)  X)"

definition
"xc_to_ss XC ≡
  case XC of (X, S) ⇒
    (if infinite X ∨ (¬ ⋃S ⊆ X) then ({}, card, 1::nat)
     else
      (let f = map_to_nat X; p = max 2 (card S + 1)
        in
      (S, λA.  (weight p (f ‘ A)), weight p (f ‘ X)) ))"
```

Figure 4.2.: Definition of the reduction, XC to SS

```
lemma weight_eq_poly:
fixes X:: "nat set" and p::nat
assumes "p ≥ 2"
shows  "weight p X = ∑{p ^ x | x . x ∈ X}"
```

Figure 4.3.: Definition of the reduction, XC to SS

Uniqueness of Polynomials

The main part of the proof is implemented as discussed in the reduction details. A problem occurred when showing the completeness of the reduction. Besides the proof presented in the implementation details, it is necessary to show that the representation of a natural number as polynomial with the base p is unique. To achieve this, we imported the Archive of Formal Proofs entry DigitInBase and applied the theorem in Figure 4.4.

m is computed by $\sum \mathbf{D\ j} \cdot b^i$, hence $\mathbf{D\ j}$ should be exactly the i-th digit of m with base b, i.e. $\mathbf{D\ j} = \mathbf{ith\_digit\ m\ j}$. In our proof by contradiction as in Lemma 8, we found an $m = \sum_{x \in S'} w(x) = B$ that can be represented with two coefficient functions $\mathbf{D\ j}$ and $\mathbf{E\ j}$. $\mathbf{D\ j}$ is a representation resulted from $\sum_{x \in S'} w(x)$, while $\mathbf{E\ j}$ is a representation resulted from B. Hence it holds that

$$\mathbf{D\ j} = \mathbf{ith\_digit\ m\ j} = \mathbf{E\ j}$$

However, due to the existence of $a \in s_1 \cap s_2$, it holds that $\mathbf{D\ a} \geq 2$ and $\mathbf{E\ a} = 1$ simultaneously. By this contradiction, we are able to show the completeness.

```
theorem seq_uniqueness:
  fixes m j :: nat and D :: "nat ⇒ nat"
  assumes "eventually_zero D"
and "m = (∑i. D i * b^i)" and "⋀i. D i < b"
  shows "D j = ith_digit m j"
```

Figure 4.4.: Snippets from DigitInBase

Polynomial-time Complexity

The implementation of the proof for polynomial complexity is identical to what is introduced in reduction details. Fortunately, the proof is automated, hence we only present and discuss about complexity in Figure 4.5 and Table 4.1. Different from the previous reductions, this reduction returns not only a set but also two constants, resulting a few constant operations during the reduction. Overall, the reduction has linear complexity, which is the same as the pen-and-paper proof.

```
definition "xc_to_ss_alg ≡ λ(X, S).
  do {
    b ← mop_check_finite_collection (X, S);
    if b
    then do {
        RETURNT ({},  card, 1)   }
    else do {
        f ← mop_constr_bij_mapping (X, S);
        p ← mop_constr_base (X, S);
        w ← mop_constr_weight p f;
        B ← mop_constr_B p f X;
        RETURNT (S, w, B)   } }
```

Figure 4.5.: The NREST version of the reduction, XC to SS

Operation	Functionality	Complexity
mop_check_finite_collection	checking the requirements	1
mop_constr_bij_mapping	constructing the mapping	$3n+1$
mop_constr_base	computing the base of polynomials	$n+3$
mop_constr_weight	constructing the weighting function	1
mop_constr_B	computing the constant sum	$3n+1$

Table 4.1.: Complexity of operations in reduction XC to SS.

4.1.3. Example

Again, we present an example for the reduction from Exact Cover to Subset Sum for better understanding.

Input: The instance of exact cover is given by

$$X := \{1,2,3,4\}$$
$$S := \{\{1\},\{2\},\{2,3\},\{2,4\},\{3,4\},\{1,2,3\}\}$$

Output: While S is not changed, the weighting function w and the sum B is given by

$$w(s) = \sum_{x \in s} 4^x, B = w(X) = 4 + 4^2 + 4^3 + 4^4 = 340$$

Validity: An exact cover S' is given by

$$S' = \{\{1\},\{2\},\{3,4\}\}$$

Hence it holds that

$$w(\{1\}) + w(\{2\}) + w(\{3,4\}) = 4 + 4^2 + (4^3 + 4^4) = 340 = B$$

4.2. Subset Sum in Sequence and Partition

The next problem that we want to reduce to is Partition. For convenience, we use $as - as'$ to demonstrate the difference between two sequences. Since all of the valid inputs require the sequences to be finite, we model the sequences using lists in all implementations. In addition # is used to append elements at the front of the lists.

Definition 13. *Partition*
Input: *A finite sequence as of natural numbers.*
Output: *Is there a sub-sequence $as' \subset s$ s.t.*

$$\sum_{x \in as'} x = \sum_{x \in as - as'} x \tag{2}$$

Partition $:= \{(a_1, a_2, ..., a_n) | \exists S' \subseteq \{1, ..., n\}. \sum_{j \in S'} a_j = \sum_{j \notin S'} a_j\}$

4.2.1. Reduction Details

For the modeling of the sequence, we use the list. Although it is possible to define the partition problem using the set and weighting function as in Subset Sum. We choose the presented definition for two reasons.

1. Showing that the definition of the problem is not an important factor in reduction, i.e. both definitions are valid and reducible under Isabelle.

2. Being consistent with the available Archive of Formal Proof instance *Hardness of Lattice Problems* [Kre23], which is also the purpose of this work, i.e. providing theoretical basis for other verification projects.

Thus, we need to perform the reduction from the sequence version of Subset Sum. We give the definition of Subset Sum using a sequence.

Definition 14. *Subset Sum in Sequence*
Input: *A finite natural number sequence as, a natural number s*
Output: *Is there a sub-sequence $as' \subset as$ s.t.*

$$\sum_{x \in as'} x = B \tag{3}$$

$$\textbf{Subset Sum Seq} := \{(a_1, a_2, ..., a_n, s) | \exists S' \subseteq \{1, ..., n\}. \sum_{j \in S'} a_j = s\}$$

Given the instance of Subst Sum (S, w, B), it is obvious that we can obtain a sequence as by converting S into a sequence and map the sequence with the weighting function w. Let $s = B$. The resulting pair (as, s) is then an instance of the Subset Sum problem in sequence representation. Then we reduce (as, s) to an instance of Partition bs.

$$bs = (1 - s + \sum_{x \in as} x) \# (s + 1) \# as$$

With this definition, we start to show the soundness of the reduction.

Lemma 11 (Soundness)**.** *If there exists an as' s.t. the equation* **(3)** *holds for (as, B),* **(2)** *should hold for the reduced bs.*

We construct a bs' from as' by

Proof.

$$bs' = (1 - s + \sum_{x \in as} x) \# as'$$
$$bs - bs' = (s + 1) \# (as - as')$$

Where the sums of the sequences satisfy the equation

$$\sum_{x \in bs'} x = (1 - s + \sum_{x \in as} x) + \sum_{x \in as'} x = (1 - s + \sum_{x \in as} x) + s = (s + 1) + (\sum_{x \in as} x - s) = \sum_{x \in bs - bs'}$$

Thus, the reduction is sound. $\square$

Similarly, we can show the completeness of the reduction.

Lemma 12 (Completeness)**.** *Let bs be reduced from* (as, B)*. If there exists a* bs' *s.t.* **(2)** *holds for bs,* **(3)** *should then hold for* (as, B)*.*

Proof. It holds that

$$(1 - s + \sum_{x \in as} x) + (s + 1) = \sum_{x \in as} x + 2 > \sum_{x \in as} x$$

As a result, $(1 - s + \sum_{x \in as} x)$ and $s + 1$ are not supposed be simultaneously existent in bs'. After separating the first two elements of bs into different subsequences, the as' is constructed by obtaining the tail of bs', with which the completeness is proven. □

Finally, we analyse the complexity of the reduction.

Lemma 13 (Polynomial Complexity)**.** *The reduction from Subset Sum to Partition can be computed within polynomial time.*

Proof. The conversion between the definitions of the Subset Sum problem costs the complexity of $|S| + 1$, for it it necessary to iterate the set S and map the sequence with the weighting function. Furthermore, we iterate the sequence similary when reducing the subset sum to partition, costing the complexity of $|as| + 2$. In total, the complexity is $(|S| + 1) + (|as| + 2) \in \mathcal{O}(|S|)$ because of $|as| = |S|$. Thus, the reduction has linear complexity. □

Then, we conclude the the main theorem with all the lemmas proven.

Theorem 4. *The presented reduction is a polynomial-time reduction from Subset Sum to Partition.*

4.2.2. Implemantation Details of Subset Sum in Sequence

Intermediate Step

Although the reduction is more straightforward compared to the previously introduced ones, the implementation is even lengthier. The reason is that conversion of the set to a list also needs an additional reduction for indexing. For this reason, we introduce a intermediate step, **subset_sum_indices** in Figure 4.5. Apparently, to map S to an set of integers from 1 to $|S|$, we need to apply **Lemma 9** again. As always, we have to check if S is finite, because finiteness is a requirement of the reduction. Then, it suffices to convert the set into a list and perform the mapping, in which we used the function **sorted_list_of_set**, converting a set of ordered type to a sorted list. Similarly, we check if S is finite and if S is of form $\{1, 2, .., |S|\}$ as a requirement. The proof for the correctness is then mostly straightforward after unfolding the necessary definitions and using a few available lemmas in the list library, such as **nth_equalityI** etc.

```
definition "generate_func S ≡ (SOME f.
  (if S = {} then bij_betw f S {} else bij_betw f S {1..card S}))"

definition "ss_to_ss_indeces ≡ λ(S, w, B). if finite S then
((generate_func S) ‘ S, λx. w (inv_into S (generate_func S) x), B)
else ({}, id, 1)"

definition "ss_indeces_to_ss_list ≡ λ(S, w, B).
   if (finite S ∧ S = {1..card S})
     then (map w (sorted_list_of_set S), B) else ([], 1)"
```

Figure 4.6.: Intermediate step of the reduction I, SS to Part

Polynomial-time Complexity

Similar to the reduction from Exact Cover to Subset Sum, the proof of the polynomial-time complexity for two reductions is rather trivial. An NREST version of the reduction is given in Figure 4.7. As shown in Table 4.2 and Table 4.3, the reduction costs linear-time complexity. Similar to the reduction from Exact Cover to Subset Sum, there are a few constant steps taken because of the structure of the instances.

Operation	Functionality	Complexity
mop_check_finiteness	checking the requirements	1
mop_mapping_of_set	constructing the mapping	$6n + 1$
mop_updating_the_weighting	updating the weighting function	$6n + 1$

Table 4.2.: Complexity of operations in reduction SS to SS Indices.

Operation	Functionality	Complexity
mop_check_finiteness_set	checking the requirements	1
mop_mapping_to_list	mapping the set to the list	$6n$
mop_nat_to_int	computing the constant	1

Table 4.3.: Complexity of operations in reduction SS Indices to SS List.

4.2.3. Implemantation Details of Partition

Choice of Definitions

Instead of the original definition, where the sum a sub-sequence is equal to another, we use a different definition in the implementation, where the double of the sum of a sub-sequence is equal to to the sum of the whole sequence. We have also shown that

```
definition "ss_to_ss_indices_alg ≡ λ(S, w, B).
  do {
    b ← mop_check_finiteness (S, w, B);
    if b
    then do {
      S' ← mop_mapping_of_set (S, w, B);
      w' ← mop_updating_the_weighting (S, w, B);
      RETURNT (S', w', B)  }
    else do {
      RETURNT ({}, id, 1)   } }"

definition "ss_indices_to_ss_list_alg ≡ λ(S, w, B).
  do {
    b ← mop_check_finiteness_set (S, w, B);
    if b
    then do {
      as ← mop_mapping_to_list (S, w, B);
      s ← mop_nat_to_int B;
      RETURNT (as, s) }
    else do {
      RETURNT ([], 1)   } }"
```

Figure 4.7.: The NREST version of the reductions, SS to SS list

```
definition "part ≡ {as::nat list. ∃xs. (∀i < length xs. xs!i ∈ {0,
1}) ∧ length as = length xs
  ∧ 2 * (∑i < length as. as ! i * xs ! i) =( ∑i < length as. as
! i)}"

definition "part_alter ≡ {as::nat list. ∃xs. (∀i < length xs. xs!i
∈ {0, 1}) ∧ length as = length xs
  ∧ (∑i < length as. as ! i * xs ! i) =(∑i < length as. as ! i *
(1 - xs ! i))}"

theorem part_eq_part_alter: "part = part_alter"
```

Figure 4.8.: Definitions for Partition

this definition is equivalent to the original definition, i.e. **part_alter** in Figure 4.8.

The reason was initially the convenience of the proof. In **part_alter**, it is necessary to consider the sum of the sub-sequence $(as - as')$ when showing the soundness lemma. Unfortunately, this is rather complex under our definition, for we have to flip the list *xs*, the zero-one list that is used for multiplication. For this flipping operation, we have shown the lemma **sum_binary_part** in Figure 4.9. If we use the new definition, this is avoidable. However, when showing the completeness lemma, we found out that we

```
lemma sum_binary_part:
assumes  "(∀i < length xs. xs!i = (0::nat) ∨ xs!i = 1)"
  and "length as = length xs"
shows
  "(∑i < length as. as ! i * xs ! i) + (∑i < length as. as ! i *
(1 - xs ! i))
    = (∑i < length as. as ! i)"
```

Figure 4.9.: Details of the reduction, SS List to Partition

have to show the same statement for the new definition, too. Thus, it is not an absolutely better definition.

Polynomial-time Complexity

The proof of the polynomial-time complexity is also trivial. An NREST version is given in Figure 4.10. As shown in Table 4.5, the reduction costs linear complexity. It is slightly different from the estimation of the pen-and-paper proof, because a few more constance steps are taken.

```
definition "mop_check_not_greater_eq ≡ λ(as, s). SPECT [s ≤ sum
((!) as) {..<length as} ↦ 1]"
definition "mop_cons_new_sum ≡ λ(as, s). SPEC (λas'. as' = (sum
((!) as) {..<length as} + 1 - s) # (s + 1) # as) (λ_. 2 * length
as + 3 + 2)"

definition "ss_list_to_part_alg ≡ λ(as, s).
  do {
    b ← mop_check_not_greater_eq (as, s);
    if b
    then do {
      as' ← mop_cons_new_sum (as, s);
      RETURNT as' }
    else do {
      RETURNT [1] }}"
```

Figure 4.10.: The NREST version of the reductions, SS List to Partition

Operation	Functionality	Complexity
mop_check_not_greater_eq	checking the requirements	1
mop_cons_new_sum	constructing the new sequence	$2n + 5$

Table 4.4.: Complexity of operations in reduction SS List to Partition.

4.2.4. Example

As a last part of Partition, we present an example indicating how an instance of Subset Sum is reduce to an instance of Partition.
Input: We use the same instance of subset sum as in the previous example. (S, w, B) be then converted to

$$\begin{aligned} as &:= [4, 16, 80, 272, 320, 72] \\ s &:= 340 \end{aligned}$$

Output: The reduced bs is then

$$bs := [425, 341, 4, 16, 80, 272, 320, 72]$$

Validity: With $as' = [4, 16, 320]$, the corresponding bs' is

$$\begin{aligned} bs' &= [425, 4, 16, 320] \\ bs - bs' &= [341, 80, 272, 72] \end{aligned}$$

with the equality

$$425 + 4 + 16 + 320 = 765 = 341 + 80 + 272 + 72$$

The other reductions are easier to understand, hence no example is provided here.

4.3. Knapsack and Zero-One Integer Programming

Knapsack and zero-one integer programming are another two classical weighted sum problems. While subset sum was referred to as knapsack in Karp's paper, its definition is nowadays different. Additionally, zero-one integer programming was originally reduced from Satisfiability. Nevertheless, there exist trivial reductions from subset sum to both of the problems. Thus, we include them in this chapter and present a reduction for them each. Since the reductions and definitions are nicely chosen, the implementation does not have many interesting details and is thus omitted.

4.3.1. Knapsack

Definition 15. *Knapsack*
Input: *A finite set S, a weighting function w, a limiting function b, a upperbound W, a*

lowerbound B
***Output**: Is there a subset $S' \subseteq S$ s.t.*

$$\sum_{x \in S'} w(x) \leq W$$
$$\sum_{x \in S'} b(x) \geq B \tag{4}$$

We reduce Subset Sum to Knapsack. With (S, w, B) as an instance of Subset Sum, (S, w, w, B, B) is then an instance of knapsack.

Theorem 5 (Polynomial-time Reduction)**.** *The presented reduction from Subset Sum to Knapsack is correct and polynomial-time bounded.*

Proof. Trivially, it holds that

$$\sum_{x \in S'} w(x) = W$$
$$\sum_{x \in S'} b(x) = B$$

where $b = w$ and $W = B$. Thus, equations in **(4)** are satisfied. Apparently, the reduction is constant, for all operations are constant. ☐

4.3.2. Zero-one Integer Programming

Definition 16. *Zero-one Integer Programming*
***Input**: A finite set X of pairs (x, b), where x is an m-tuple of integers and b is an integer, an m-tuple x and an integer B*
***Output**: Is there an m-tuple y of integers s.t.*

$$x^T \cdot y \leq b$$
$$c^T \cdot y \geq B, \forall (x, b) \in X \tag{5}$$

Although most researchers tend to use matrix for the definition of the zero-one integer programming, we follow the definition from *Computers and Intractability* [GJ79], because it is convenient for our definition and consequently requires less effort. Given an instance of subset sum problem in sequence, (as, s), let

$$X = \{(as, s)\}, c = as, B = s$$

Theorem 6 (Polynomial-time Reduction)**.** *The presented reduction from Subset Sum to Zero-One Integer Programming is correct and polynomial-time bounded.*

Proof. (X, c, B) is then an instance of the zero-one integer programming problem, for there exists an xs s.t. $xs^T \cdot as = s$. Hence all equations in **(5)** hold. Since all the operations are constant, the resulting complexity is also constant. ☐

It is not hard to notice that this definition can be converted to an alternative definition using matrices. The function **sorted_list_of_set** from the list library may be useful here. We did not apply and use this feature, but present it as a possibility, in case other reductions in the future are based on the alternative definition with matrices.

5. Conclusion

In this work, we have successfully formalised the NP-hardness of a few selected classical decision problems. The whole work consists of more than 3800 lines of codes, with which we contribute six new problems into the Karp21 project. A general overview of the progress of the list is given in Figure 5.1.

The new reductions from this work are linked with red arrows. It is also noticed that a reduction from Satisfiability to 3CNF-Satisfiability is not yet formalised, and should be formalised in the future. With this work, we manage to show the possibility of formalising and verifying the polynomial-time reductions and provide a theorectical basis for other works related to the complexity theory, especially the NP-hardness.

Future Work

As a future work, it is necessary to add a few more reductions and complete Karp's list of twenty-one NP-hard problems. We summarize the remainng problems as follows,

1. 3cnf-satisfiability, feedback arc set, clique cover. Reductions to these problems are not related to this work. Reductions are strongly dependent on the previous works.
2. 3-dimensional match, steiner tree, job sequencing, max cut. The reductions presented by Karp are dependent on this work.

Furthermore, a few other classical NP-Hard problems that are not in Karp's list can also be added to Karp21 project to make the result more convincing. Traveling salesman problem, for example, can be reduced from Hamilton's circuit, while the bin packing problem is also reducible from partition.

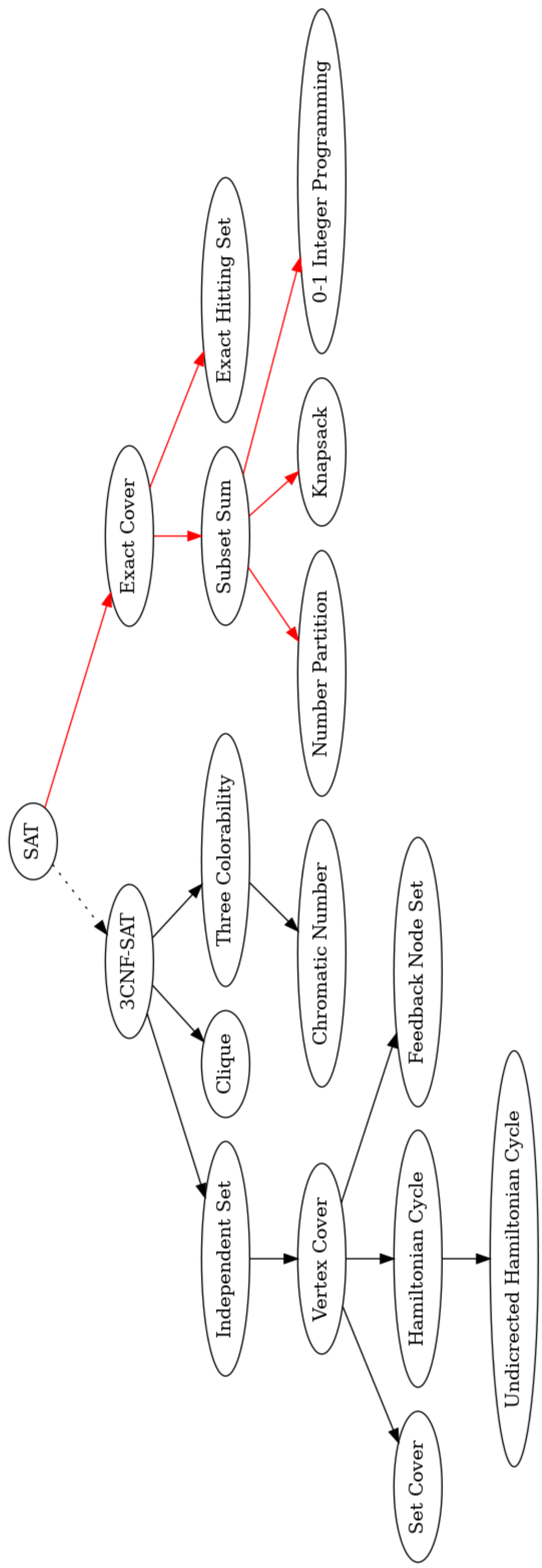

Figure 5.1.: The updated reduction graph of the Karp21 project.

A. Karp's Definition of NP-hard problems mentioned

In this part, we present Karp's original definition of the NP-hard problems included in this work. Readers are encouraged to compare them with the definitions in the previous chapters.

Definition 17. *Satisfiability*
Input: *Clauses $C_1, C_2, ..., C_p$*
Output: *Is there a set $S \subset \{x_1, x_2, ..., x_n, \overline{x_1}, \overline{x_2}, ..., \overline{x_n}\}$ s.t.*

1. *S does not contain a complementary pair of literals.*
2. *$S \cap C_k \neq \emptyset$, for all k from 1 to p.*

Definition 18. *Exact Cover*
Input: *Family $\{S_j\}$ of subsets of a set $\{u_i | i = 1, 2, ..., t\}$*
Output: *Is there a subfamily $\{T_h\} \subseteq \{S_j\}$ such that the sets T_h are disjoint and $\bigcup T_h = \bigcup S_j = \{u_i | i = 1, 2, ..., t\}$?*

Definition 19. *(Exact) Hitting Set*
Input: *Family $\{U_j\}$ of subsets of $\{s_j | j = 1, 2, ..., r\}$*
Output: *Is there a set W s.t. for each i, $|W \cap U_i| = 1$?*

Definition 20. *Subset Sum, Knapsack in Karp's definition*
Input: *$(a_1, a_2, ..., a_n, b) \in Z^{n+1}$*
Output: *Does $\sum a_j x_j = b$ have a zero-one solution?*

Definition 21. *Partition*
Input: *$(c_1, ..., c_s) \in Z^s$*
Output: *Is there a subset $I \subseteq \{1, 2, ..., s\}$ s.t. $\sum_{h \in I} c_h = \sum_{h \notin I} c_h$?*

Definition 22. *Zero-One Integer Programming*
Input: *Integer Matrix C and integer vector d*
Output: *Does there exist a zero-one vector x s.t. $Cx = d$?*

List of Figures

List of Tables

Bibliography

[19] *How to prove that Exact Cover is NP-complete using SAT?* 2019. URL: https://cs.stackexchange.com/questions/118054/how-to-prove-that-exact-cover-is-np-complete-using-sat.

[al23] M. E. et al., ed. *Archive of Formal Proofs*. 2023. URL: https://www.isa-afp.org/.

[Coo23] S. A. Cook. "The complexity of theorem-proving procedures". In: *Logic, Automata, and Computational Complexity: The Works of Stephen A. Cook*. 2023, pp. 143–152.

[Ebe15] M. Eberl. "Landau Symbols". In: *Archive of Formal Proofs* (July 2015). https://isa-afp.org/entries/Landau_Symbols.html, Formal proof development. ISSN: 2150-914x.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and intractability*. Vol. 174. freeman San Francisco, 1979.

[Has21] M. P. L. Haslbeck. "Verified Quantitative Analysis of Imperative Algorithms". PhD thesis. Technische Universität München, 2021.

[Has23] M. P. e. a. Haslbeck. *poly-reductions*. 2023. URL: https://github.com/rosskopfs/poly-reductions.

[Kar10] R. M. Karp. *Reducibility among combinatorial problems*. Springer, 2010.

[Kre23] K. Kreuzer. "Hardness of Lattice Problems". In: *Archive of Formal Proofs* (Feb. 2023). https://isa-afp.org/entries/CVP_Hardness.html, Formal proof development. ISSN: 2150-914x.

[Lev73] L. A. Levin. "Universal sequential search problems". In: *Problemy peredachi informatsii* 9.3 (1973), pp. 115–116.

[Sta23] C. Staats. "Positional Notation for Natural Numbers in an Arbitrary Base". In: *Archive of Formal Proofs* (Apr. 2023). https://isa-afp.org/entries/DigitsInBase.html, Formal proof development. ISSN: 2150-914x.

[Wen+04] M. Wenzel et al. *The isabelle/isar reference manual*. 2004.

[ZH18] B. Zhan and M. P. Haslbeck. "Verifying asymptotic time complexity of imperative programs in Isabelle". In: *Automated Reasoning: 9th International Joint Conference, IJCAR 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings 9*. Springer. 2018, pp. 532–548.