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

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

Acknowledgments **v**

Abstract **vii**

1. Introduction **1**
1.1. Motivations . . . 1
1.2. Contributions . . . 1
1.3. Outline . . . 2

2. Preliminaries **3**
2.1. Isabelle and Dependencies . . . 3
2.2. NP-Hardness and polynomial-time reductions . . . 4
2.2.1. Asymptotic Notation . . . 4
2.2.2. Decision problems . . . 5
2.2.3. Polynomial-time reductions . . . 6
2.2.4. NP-Hardness and Satisfiability . . . 6
2.3. Application of NREST and paradigm . . . 7

3. Set Covering Problems **9**
3.1. Exact Cover . . . 9
3.1.1. Choice of reduction . . . 9
3.1.2. Reduction Details . . . 10
3.1.3. Implementation Details . . . 12
3.2. Exact Hitting Set . . . 15
3.2.1. Reduction Details . . . 15
3.2.2. Implementation Details . . . 16

4. Weighted Sum Problems **19**
4.1. Subset sum . . . 19
4.1.1. Reduction Details . . . 19
4.1.2. Implemantation Deatails . . . 21
4.2. Subset sum in list and number partition . . . 23
4.2.1. Implemantation Details of Subset Sum in Sequence . . . 24
4.2.2. Implemantation Details of Partition . . . 26
4.3. Knapsack and Zero-One Integer Programming . . . 27
4.3.1. Knapsack . . . 27
4.3.2. Zero-one Integer Programming . . . 28

5. Conclusion **29**

A. Examples for reductions **31**
A.1. Example for polynomial reduction from Satisfiability To Exact Cover . . . 31
A.2. Example for polynomial reduction from Exact Cover To Subset Sum . . . 31
A.3. Example for polynomial reduction from Subset Sum To Partition 32

List of Figures **33**

List of Tables **35**

Bibliography **37**

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 board games like Go and Chess. These problems are formally defined as decision problems. One of the most famous decision problem classes is the NP-hard problems.

NP-hard problems are known be hard to be computed by a polynomial-time algorithm. They have 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 $\mathbf{P} = \mathbf{NP}$ and to develop algorithms that efficiently compute NP-hard problems. 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. 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, to formalise NP-hard problems in Karp's paper in 1972. 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 set that contains all elements in another set. In weighted sum problems, we look for a set of instances s.t. their weighted sum reaches another constant bound. While they are the basis of further reductions to problems like 3-dimensional matching, neither of the categories was formalised and verified in the existing project. Thus, we chose these problems to complete the project and prepare for future work.

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

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.

In Chapter 3, we discuss the polynomial-time reduction of the set cover problems, while Chapter 4 consists of the weighted sum problems. A list of examples of reductions is also available in the appendix for better understanding.

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 is a library of projects developed by Isabelle. Our work is implemented by Isabelle and used many entries from the 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, especially for sufficiently large inputs.

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

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.

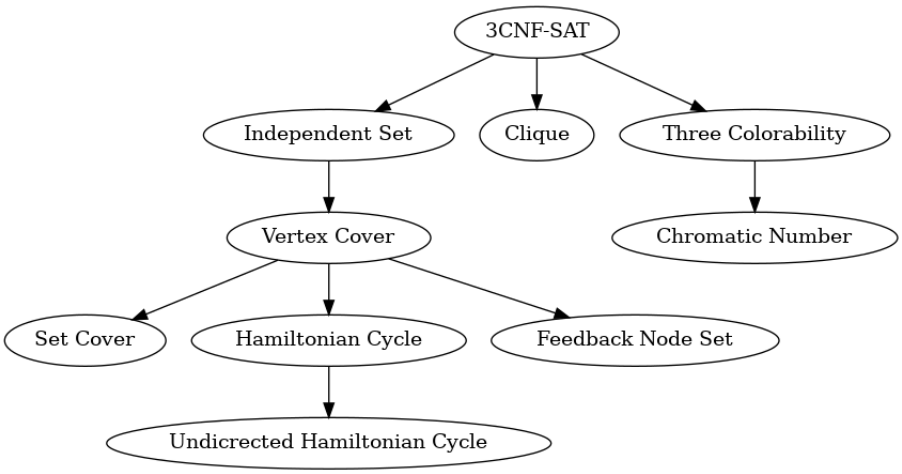

Figure 2.1.: The reduction graph of the Karp21 project

DigitsInBase[Sta23]

This entry of Archive of Formal Proofs 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. NP-Hardness and polynomial-time reductions

2.2.1. 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, which contains most classes listed above except for the factorial 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**.

2.2.2. 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. 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. 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 define them formally as NP.

Definition 3. *If there is a non-deterministic algorithm that decides the solution to the problem in polynomial time, it 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.

2.2.3. 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.

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

2.2.4. 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 s.t. $A \leq_p B$. The first proven NP-hard problem is Satisfiability, which was independently proven by Cook in 1971 and Levin in 1973. 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 time unit.
- *SPECT* **[cond** $\rightarrow$ **cost]**. A command used for checking a condition. Checking the condition **cond** take **cost** time units.
- *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 $Q\ x$ time units.

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 only time unit.
2. During the iteration, it costs 1 time unit each for access, modification and insertion.
3. All other operations should cost one time unit, 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 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. 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, 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. In Exact Cover, it not only looks for a cover, but also requires the existence of cover 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'?*

$$\textbf{Exact Cover} := \{(X, S) | \bigcup S \subseteq X \wedge \exists S' \subseteq S. \forall x \in X. \exists s \in S'. x \in s\}$$

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 Exact Cover is a fundamental NP-hard problem, there are many different reductions available. 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 **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. 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 a container type. More details follow in the section 3.1.3.

3.1.2. Reduction Details

Given a propositional logical formula F, 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 and which contains all 3 different kinds of objects.

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

Furthermore, we construct S, a collection of subsets of X. 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 contains all of the four types of subsets.

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

The pair of (X, S) is the input for Exact Cover.

Lemma 1 (Soundess)**.** *Let F be satisfiable. The pair (X, S) is then an instance of the exact cover.*

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

1. For each $x_i \in vars\ F$, $pos(x_i)$ is included in S' when $\sigma(pos(x_i)) \equiv \bot$. Otherwise we insert $neg(x_i)$ into S'.
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'
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.

Obviously, each position in $pos(x_i)$ and $neg(x_i)$ 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, each clause and variable is unique in the construction. Hence S' is disjoint.

From this fact, we can also conclude that clauses and variables are covered in S'. Now we only have to show that all the positions are covered. If a position p_{ij} is false under σ, it is covered in the first step. Otherwise it is either covered in the second step or the third step. □

Lemma 2 (Completeness)**.** *Let (X, S) be reduced from F. If (X, S) is an instance of the exact cover, F has to be satisfiable.*

Proof. Given an exact cover pair (X, S) reduced from F, it is easy to reconstruct the model σ with the same approach as in the proof of the soundness, showing that F is satisfiable. Thus, the completeness of the reduction is proven. □

Lemma 3 (Polynomial Complexity)**.** *The construction of (X, S) from F is polynomial.*

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, indicating a linear complexity.

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. $|c_i| \leq c$. Apparently, 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 costs the polynomial complexity of $k + cm + 2nk \in \mathcal{O}(nk + m)$. With the linear complexity of the construction of X, we conclude that the reduction has the quadratic complexity. □

3.1.3. Implementation Details

Choice of Definitions

A most primary definition of Exact Cover, as shown in Figure 3.1, states the disjointness with nested quantifiers. Three different conditions have be satisfied for an exact cover

```
definition "exact_cover ≡
   {(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

instance. 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 it gives us a direct view of the exact cover, 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

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

Figure 3.2.: A second definition of Exact Cover

predicate **disjoint**, we managed to simplify the definition. Moreover, we also benefitted from reusing the existing lemmas in the relating dependency.

A Container Type

Intuitively, the variables, positions and clause can be represented by tuple of indices. However, it is not easy to unify the representation s.t. 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, a container type is exempted from the tedious handling of tuples. Thus, we implement this container type **xc_element** in Figure 3.3 The constructor **V, C** and **L** stands for the variables, 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. Then, it is possible to define the reduction function.

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

Figure 3.3.: Definition of the container type

```
abbreviation "comp_X F ≡
  vars_of_sat F ∪ clauses_of_sat F ∪ literals_of_sat F"
abbreviation "comp_S F ≡
    literal_sets F ∪ clauses_with_literals F
  ∪ var_true_literals F ∪ var_false_literals F"

definition sat_xc :: "'a three_sat ⇒ 'a xc_element set * 'a
xc_element set set" where
"sat_xc F = (comp_X F, comp_S F)"
```

Figure 3.4.: Definition of the reduction

A Non-deterministic Construction

When constructing a cover S', we have included $\{c_i, p_{ij}\}$, where j is the smallest number s.t. 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

```
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 3.5.: Non-deterministic Construction using SOME

$\{c_i, p_{ij}\}$ non-deterministically, as defined in **constr_cover_clause**. 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 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 is less interesting. 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 logical formula F, we will iterate all variables, clauses and positions. Hence all of them are need 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 exact cover instance (X, S). Then we can define the NREST algorithm.

```
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)
    }
)"

definition "sat_to_xc_time_aux l m n =
  2 * m + 3 * n + 1 + 1
+ 3 * l + 3 * m + m + 3 * n
+ (3 + 1) * n + (3 + 1 + 1) * n + m
+ (3 + 1) * l + (3 + 1 + 1) * n + (1 + 1) * m + 1
+ (3 + 1) * l + (3 + 1 + 1) * n + (1 + 1) * m + 1
+ 2 + 3"

definition "sat_to_xc_space_aux l m n =
l + m + n
+ n + n
+ l + l"
```

The proof of the polynomial bound is, however, hardly automated. Part of the reason is

[1] A few previous reductions from Satisfiability use the number of clauses as a metric. It was also correct, for those reductions iterated only the clauses.

that Isabelle fails to find and apply the monotonicity of the space function, time function and the cardinality. For this reason, it is necessary to show such relationships. For example, one of them is about the cardinality

```
lemma card_Un_le_4:
"card (a ∪ b ∪ c ∪ d) ≤ card a + card b + card c + card d"
```

Finally, we show the *refines* and *size* lemma as well as that the reduction is polynomial.

```
lemma sat_to_xc_size:
"size_XC (sat_xc F) ≤ sat_to_xc_space (size_SAT_max F)"

lemma sat_to_xc_refines:
"sat_to_xc_alg F ≤
  SPEC (λy. y = sat_xc F) (λ_. sat_to_xc_time (size_SAT_max F))"

theorem is_reduction_sat_xc:
"is_reduction sat_xc cnf_sat exact_cover"
```

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 the set covering, the exact hitting set is a variant of the exact cover.

Definition 7. *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?*

$$\textbf{Exact Hitting Set} := \{S \mid \exists W.\ \forall s \in S.\ |W \cap s| = 1\}$$

3.2.1. Reduction Details

Given an exact cover pair (X, S), the hitting set input C is constructed by

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

Thus, C is the set of sub-collections denoted by c_u. All elements in c_u share the same element u.

[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.

Lemma 4 (Soundess). *Let (X, S) be an exact cover instance. A collection C reduced from (X, S) is then an instance of the exact hitting set.*

For the soundness of the reduction, it suffices to show

$$\exists W.\ \forall c_u \in C.\ |W \cap c_u| = 1$$

Proof. Let $W = S'$ and $c_u = \{s | u \in s, s \in S\} \in C$ be an arbitrary element. Since S' covers X exactly, there is exactly one $s \in S'$ that contains u. Moreover, this s is also included in the c_u, for $s \in S'$ and $S' \subseteq S$. Thus, it is the only element in the set $W \cap c_u$. Consequently, it holds that $|W \cap c_u| = 1$. □

Lemma 5 (Completeness). *Let the collection C 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 the 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 fulfills the same property as W. Let $S' = W'$, the completeness is then proven analogously as the soundness. □

Lemma 6 (Polynomial Complexity). *The construction of C from (X, S) is polynomial.*

Proof. Finally, we show that the reduction is polynomial. 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 $\mathcal{O}(|X||S|)$. □

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 3.6

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

Figure 3.6.: Definition of the reduction, exact cover to exact hitting set

Polynomial Complexity

We determine the size of the exact hitting set entry C as $|C|$. According to the paradigm, we define the NREST algorithm and show the *refines* and *size* lemma as follows

```
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 {{}}
    }
  }
"

definition "xc_to_ehs_space n ≡ 1 + n*n"
definition "xc_to_ehs_time n ≡ 1 + 3 * n * n"

lemma xc_to_ehs_size:
"size_ehs (xc_to_ehs xc) ≤ xc_to_ehs_space (size_XC xc)"
lemma xc_to_ehs_refines:
"xc_to_ehs_alg xc ≤ SPEC (λy. y = xc_to_ehs xc) (λ_. xc_to_ehs_time (size_XC
xc))"
```

The proof of 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 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 about the cardinality of the exact hitting set.

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

Finally, we show that the reduciton is correct and polynomial.

```
theorem xc_to_ehs_is_polyred:
  "ispolyred xc_to_ehs_alg exact_cover exact_hitting_set size_XC size_ehs"
```

4. Weighted Sum Problems

In this chapter, we discuss the NP-Hardness of a few weighted problems. In weighted sum problems, there is a weighting function w over a set S. The problems ask about whether the weight of this elements in S satisfies a 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 Exact Cover, 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 the subset sum problem with a set and a weighting function. There are also a few alternative definitions using a multiset or a list without a weighting function, which is useful for our other reductions. More details follow in the number partition section.

Definition 8. *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.*

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

4.1.1. Reduction Details

We reduce the exact cover problem to the subset sum. We start with the exact cover problem over the sets of natural numbers. Given (X, S) an exact cover instance over natural numbers, let S be the set in the entry of the subset sum problem. Then we define the weighting function w and the sum B by

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

Where p is a natural that is no less than $|S|$ The triple (S, w, B) is then an input for the subset sum problem.

Lemma 7 (Soundness)**.** *If (X, S) is an instance of the exact cover. The reduced (S, w, B) is then an instance of the subset sum problem.*

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

Apparently, the reduction is sound. Let the $S' \subseteq S$ be an exact cover of X. It then holds that

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

Thus, (S, w, B) is an instance of the subset sum problem.

Lemma 8 (Completeness)**.** *Let (S, w, B) be reduced from (X, S). If (S, w, B) is a subset sum instance, (X, S) has to be an exact cover instance.*

Obtain $S' \subseteq S$ for which **(1)** holds. We show the disjointness of S' with 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 $x \in s_1 \cap s_2$ be arbitrary. Then there are two cases for the coefficient c_x of p^x in the polynomial $\sum_{x \in S'} w(x)$ of p.

1. $c_x \geq 2$. The corresponding coefficient c'_x in the polynomial $w(X)$ is one. From **(1)** it is obvious that this case is not valid.

2. $c_x = 1$ or $c_x = 0$. Though **(1)** is satisfied, there are still at least two p^x in the polynomial $\sum_{x \in S'} w(x)$. Hence the number of p^x in this polynomial is at least p. However, there are utmost $|S|$ elements in S', meaning that there are utmost $|S|$ such p^x. From the fact the $p > |S|$, it is not possible that the number of p^x is greater equal p. As a result, this case is also not valid.

In conclusion, the assumption is false and S' is thus disjoint. Then, it follows directly from the disjointness that S' covers X, 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 to any arbitrary type. For this reason, we need to construct a mapping.

Lemma 9. *Let S be an arbitrary finite S. Then there exists a bijective function $f, S \to N$ s.t. the image of f is empty when S is empty, and is $\{1, 2, ..., |S| - 1\}$ otherwise.*

The proof for this lemma is trivial. With this approach, we can covert each exact cover problem to an exact cover problem over the natural numbers and then reduce it to the subset sum problem.

Lemma 10 (Polynomial Complexity)**.** *The construction of (S, w, B) from (X, S) is polynomial.*

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 over S to construct w and B, resulting in the complexity of $2|S|$. In total, it only costs $|X| + 2|S| \in \mathcal{O}(|X| + |S|)$, i.e. the linear complexity.

4.1.2. Implemantation Deatails

Reduction and Proof

In implementation, we started with the construction as presented in Lemma 9. The function **map_to_nat** returns a function that maps X to a corresponding set of natural numbers. Following this, we define the weighting function and reduction function. Similar to what happened with the 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.

```
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))}"
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)))
      )
    )
  )"
```

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 the lemma **weight_eq_poly**.

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

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 in the form of the polynomials with the base p is unique. To

achieve this, we imported the Archive of Formal Proofs entry DigitInBase and applied the theorem **seq_uniqueness**.

```
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"
```

Then, we can show that the reduction is correct.

```
theorem is_reduction_xc_to_ss:
  "is_reduction xc_to_ss exact_cover subset_sum"
```

Polynomial Complexity

The implementation of the proof for polynomial complexity is identical to what is introduced in reductio details. Fortunately, there is not additional step to be stated, for all of the proof is automated.

```
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)
    }
  }
)"

definition "xc_to_ss_space n = 1 + n + 1 + n + 1"
definition "xc_to_ss_time n = 1 + (3 * n + 1) + (n + 3) + 1 + (3 * n + 1)"
lemma xc_to_ss_size:
"size_SS (xc_to_ss (X, S)) ≤ xc_to_ss_space (size_XC (X, S))"


lemma xc_to_ss_refines:
"xc_to_ss_alg (X, S) ≤ SPEC (λy. y = xc_to_ss (X, S)) (λ_. xc_to_ss_time
(size_XC (X, S)))"

theorem xc_to_ss_ispolyred:
  "ispolyred xc_to_ss_alg exact_cover subset_sum size_XC size_SS"
```

4.2. Subset sum in list and number partition

The next problem that we want to reduce to is partition.

Definition 9. *Partition*
***Input**: A finite sequence as of natural numbers.*
***Output**: Is there a sub-sequence $as' \subset s$ s.t.*

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

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 the subset sum problem, 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 definition are valid and reducible under Isabelle.
2. Staying consistent with the available Archive of Formal Proof instance *Hardness of Lattice Problems*, which is also the purpose of this work, i.e. providing theoretical bases for a few other verification projects.

Thus, we need to perform the reduction from the list version of the subset sum problem. We give the definition of the subset sum problem using a sequence.

Definition 10. *Subset Sum in Sequence*
***Input**: A finite sequence as of natural numbers, a natural number s*
***Output**: Is there a sub-sequence $as' \subset as$ s.t.*

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

Given the subset sum instance (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 a partition instance bs.

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

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

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

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.

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)*.*

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.

Lemma 13 (Polynomial Complexity)**.** *The reduction from the subset sum to partition is polynomial.*

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

4.2.1. Implemantation Details of Subset Sum in Sequence

Reduction and Proof

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 is a reduction, which we have to verify. The intermediate step is then defined by

```
definition "subset_sum_indices ≡ {(S, w, B). finite S ∧ S = (if S = {} then
{} else {1..card S}) ∧ (∃S' ⊆ S. sum w S' = B)}"
```

Apparently, to map S to an set of integers from 1 to $|S|$, we need to apply **Lemma 9** again. Additionally, we have to check if S is finite, for finiteness is a requirement of the reduction. Thus, the reduction and the proof is given by

```
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_indices ≡ (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))"

theorem is_reduction_ss_to_ss_indices:
"is_reduction ss_to_ss_indices subset_sum subset_sum_indices"
```

Then, it suffices to convert the set into a list and perform the map, 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 "subset_sum_indices ≡ {(S, w, B). finite S ∧ S = (if S = {} then
{} else {1..card S}) ∧ (∃S' ⊆ S. sum w S' = B)}"

definition ss_indices_to_ss_list :: "nat set × (nat ⇒ nat) × nat ⇒ nat
list × nat" where
"ss_indices_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))"

theorem is_reduction_ss_indices_to_ss_list:
"is_reduction ss_indices_to_ss_list subset_sum_indices subset_sum_list"
```

Polynomial Complexity

Similar to the reduction from exact cover to subset sum, the proof for the polynomial complexity for two reductions is rather trivial.

```
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_to_ss_indices_space n = 1 + n + n + 1"
definition "ss_to_ss_indices_time n = 1 + (3 * n + 3 * n + 1) + (3 * n +
3 * n + 1) + 1"

theorem ss_to_ss_indices_is_polyred:
  "ispolyred ss_to_ss_indices_alg subset_sum subset_sum_indices size_SS
size_ss_indices"

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)
    }
  }"

definition "ss_indices_to_ss_list_space n = 1 + 2 * n"
definition "ss_indices_to_ss_list_time n = 1 + 3 * n + 3 * n + 1"

theorem ss_indices_to_ss_list_is_polyred:
  "ispolyred ss_indices_to_ss_list_alg subset_sum_indices subset_sum_list
size_ss_indices size_ss_list"
```

4.2.2. Implemantation Details of Partition

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 twice of the sum of a sub-sequence is equal to to the sum of the sequence. We have also shown that this definition is equivalent to the original definition, i.e. **part_alter** in implementation.

```
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"

theorem is_reduction_ss_list_to_part:
"is_reduction ss_list_to_part subset_sum_list part"
```

The reason was initially for the convenience of the proof. In the original definition, it is necessary to consider the sum of the sub-sequence $(as - as')$ when showing the soundness lemma. This is, unfortunately, 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 show the lemma **sum__part**. If we use the new definition, this is avoidable.

```
lemma sum_binary_part:
```

```
assumes "(∀i < length xs. xs!i = (0::nat) ∨ xs!i = 1)" "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)"
```

However, when showing the completeness lemma, we found out that we have to show the same statement for the new definition, too. Thus, it is not a absolutely better definition.

The proof of the polynomial complexity is also trivial.

> Should I include the codes without much description, or leave them unincluded? Possible description: choice of the complexity and what is done in each step, but would be too lengthy.

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 exists a trivial reduction from subset sum to both of the problems. Thus, we include them in this chapter and present a reduction for them each. Since the reduction is short and trivial, the implementation does not have many meaningful details and is thus omitted.

4.3.1. Knapsack

Definition 11. *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.*

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

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

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

where $b = w$ and $W = B$. Apparently, the reduction is constant, for no iteration is necessary.

4.3.2. Zero-one Integer Programming

Definition 12. *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.*

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

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

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

(X, c, B) is then an instance of the zero-one integer programming problem, for there exists an xs s.t. $xs^T \cdot as = s$. Since there is no iteration, the reduction is contant, too.

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 cpntribute 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 reductions and to 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. Traveling salesman problem, for example, can be reduced from Hamilton's circuit, while the bin packing problem is also reducible from the partition.

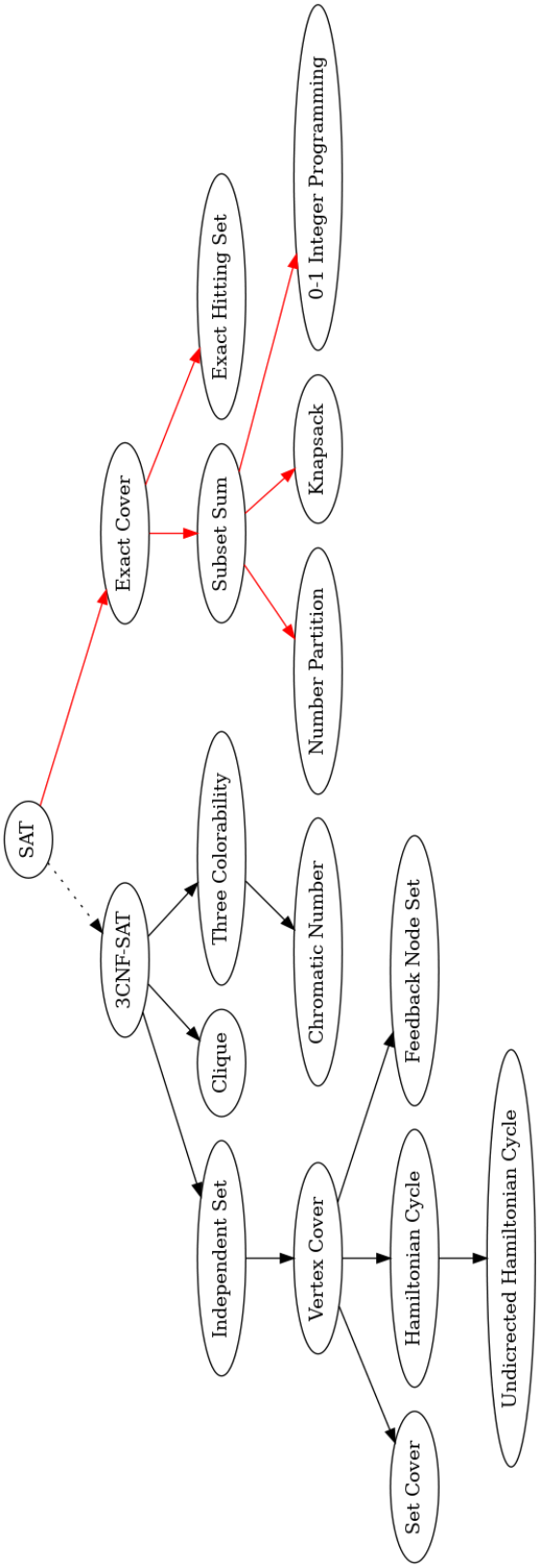

Figure 5.1.: The updated reduction graph of the Karp21 project.

A. Examples for reductions

In this part we present examples for polynomial reductions to assist with the understanding.

A.1. Example for polynomial reduction from Satisfiability To Exact Cover

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

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

Wir construct an exact cover S' by

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

A.2. Example for polynomial reduction from Exact Cover To Subset Sum

Input: The instance of exact cover is given by

$$\begin{aligned} X &:= \{1, 2, 3, 4\} \\ S &:= \{\{1\}, \{2\}, \{2, 3\}, \{2, 4\}, \{3, 4\}, \{1, 2, 3\}\} \end{aligned}$$

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

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

A.3. Example for polynomial reduction from Subset Sum To Partition

Input: We use the same subset sum entry as in the previous example. (S, w, B) be then converted to

$$as := [4, 16, 80, 272, 320, 72]$$
$$s := 340$$

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

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

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.

List of Figures

2.1. The reduction graph of the Karp21 project 4
2.2. Definition of Satisfiability 7

3.1. A first definition of Exact Cover 12
3.2. A second definition of Exact Cover 12
3.3. Definition of the container type 13
3.4. Definition of the reduction 13
3.5. Non-deterministic Construction using SOME 13
3.6. Definition of the reduction, exact cover to exact hitting set 16

5.1. The updated reduction graph of the Karp21 project. 30

List of Tables

2.1. List of commonly encountered complexity classes. 5

Bibliography

[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.

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

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

[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.