rijksuniversiteit groningen

Faculty of Mathematics and Natural Sciences

Giving a step-by-step reduction from SAT to TSP and giving some remarks on Neil Tennant's *Changes of Mind*

Bachelor Thesis Mathematics

July 2014

Student: M.M. Bronts

Supervisors: Prof. dr. J. Top and Prof. dr. B.P. Kooi

Abstract

Karp wrote an article about 21 decision problems. He gave instances for polynomial reductions between these problems, but he did not prove that these instances actually worked. In this thesis we will first prove that some of these instances are indeed correct. Second we will give some remarks on a belief contraction problem given by Neil Tennant in his book *Changes of Mind*. He describes what happens in our mind when we change our mind and argues that this is an NP-complete decision problem.

Contents

1 Introduction **4**

2 Historical background **5**

3 Introductory definitions and decision problems **7**
- 3.1 The Turing machine 7
- 3.2 Some important definitions 7
- 3.3 Decision problems 8

4 The Boolean satisfiability problem **10**
- 4.1 Propositional logic 10
- 4.2 Polynomial functions over $\mathbb{F}_2$ 11
- 4.3 SAT 13

5 Complexity theory and reduction **15**
- 5.1 Complexity theory 15
- 5.2 Polynomial time reduction 17
- 5.3 Example: 3-SAT is NP-complete 19

6 Reduction of SAT to TSP **23**
- 6.1 SAT $\preceq$ CLIQUE 23
- 6.2 CLIQUE $\preceq$ VC 24
- 6.3 VC $\preceq$ HC 25
- 6.4 HC $\preceq$ TSP 29

7 Remarks on *Changes of Mind* by Neil Tennant **31**
- 7.1 Tennant's belief system 31
- 7.2 The contraction problem 34
- 7.3 How realistic is Tennant's belief system? 35
- 7.4 Can you say something about the complexity of a problem about the mind? 36

8 Conclusion **39**

1 Introduction

This thesis is about problems that can be answered with 'yes' or 'no', so-called decision problems. We focus especially on the Boolean satisfiability problem (SAT). This is a problem that has to do with propositional logic. You are given a propositional formula and the question is whether there is a satisfying truth assignment. That is, whether you can give an assignment of truth values to the variables in the propositional formula that makes the formula true. If there is such an assignment, the answer to the decision problem is 'yes'. For example, $(p \vee \neg q) \to (p \wedge q)$ is satisfiable, since making both p and q true makes the formula true. The Boolean satisfiability problem is the first problem that was shown to be NP-complete. NP-complete is, next to P and NP, an important complexity class in complexity theory. When a problem is NP-complete, the problem is in the complexity class NP and every other problem which is in the class NP can be reduced quickly (in polynomial time) to it. Informally, the class P contains all decision problems that can be solved in polynomial time and the class NP contains all decision problems of which it can be verified in polynomial time whether there is a solution.

In the first part of this thesis the focus lies on polynomial time reductions used to show that a problem is NP-complete. A given decision problem in the class NP can be shown to be NP-complete by reducing another problem, of which we already know it to be NP-complete, to it. This process of reducing problems produces a scheme of reductions. The Boolean satisfiability problem is at the center of this scheme and we will show that the problem can be reduced step by step in polynomial time to the Traveling Salesman Problem (TSP), another problem in this scheme, see Figure 3. TSP is about a man who has to visit a certain number of cities. It costs money to drive from one city to another and the salesman has to find the cheapest way to visit each city exactly once. The decision version is then to find out whether such a tour exists without exceeding a given amount of money. TSP is not directly connected to SAT in this scheme, there are some reductions in between. In this thesis we will look at the reduction process from SAT to TSP. We know that both problems are NP-complete, so in theory there exists a direct reduction. Finding a direct reduction proved to be difficult, so we will reduce SAT to TSP via CLIQUE, Vertex Cover and Hamiltonian Circuit. In an article by Karp [1972, p. 97-98] instances of these reductions are given but Karp did not provide detailed proofs. Some of these proves will be given in this thesis.

We will start with some historical background on the mathematics which led to the development of complexity theory. We will mention George Boole, the founder of Boolean Algebra, Kurt Gödel, who questioned the provability of statements, Alan Turing, the inventor of the Turing machine and Stephen Arthur Cook, a mathematician who is specialized in computational complexity theory. In Section 3 we explain briefly how a Turing machine works and give some important definitions which we need to understand the decision problems that are mentioned. The fourth section starts with a short introduction to propositional logic and an introduction to polynomial functions over $\mathbb{F}_2$ before stating the problem SAT. In Section 5 complexity theory will be explained and definitions of the three most important complexity classes as mentioned above are given. We will also define what a reduction is and give as example the reduction from SAT to 3-SAT. In Section 6 we will reduce SAT to TSP via CLIQUE, Vertex Cover and Hamiltonian Circuit. In the last section we will use the knowledge about complexity to say something about the book *Changes of mind* by Neil Tennant. Tennant describes a contraction problem, which is about changing beliefs. It is a decision problem, and Tennant proves that his problem is NP-complete. We discuss whether the system of beliefs, as represented by Tennant, is a realistic representation of the mind and we discuss whether we can say something about the complexity of a problem about the mind.

2 Historical background

In the middle of the nineteenth century mathematics became more abstract. Mathematicians started thinking that mathematics was more than just calculation techniques. They discovered that mathematics was based on formal structures, axioms and philosophical ideas. George Boole (1815-1864) and Augustus De Morgan (1806-1871), two British logicians, came up with important systems for deductive reasoning, which formally captures the idea that a conclusion follows logically from some set of premises. Boole is known for his book *An Investigation of the Laws of Thought* in which he, as the title suggests, wants

> [...] to investigate the fundamental laws of those operations of the mind by which reasoning is performed; to give expression to them in the symbolical language of a Calculus, and upon this foundation to establish the science of Logic and construct its method; [...] [Boole, 1854, p. 1]

Boole also discovered Boolean Algebra in which he reduces logic to algebra. He thought of mathematical propositions as being true or not true, 1 or 0 respectively. This will be of importance in the Boolean satisfiability problem, a decision problem which plays a central role in this thesis and will be treated in Section 4.

In the beginning of the twentieth century formalism was upcoming. Formalism says that mathematics is completely built up of (strings of) symbols which are manipulated by formal rules. Starting with a string of symbols to which we can apply the formal rules, we generate new strings (see the tq-system in **??**). The influential mathematician David Hilbert (1862-1943) made it possible to develop the formalist school. He believed that logic was the foundation of mathematics. Hilbert thought that for every true statement in mathematics there could be found a proof (completeness) and that it could be proven that there was no contradiction in the formal system (consistency). Kurt Gödel (1906-1978) criticized the completeness of formal systems and proved his *Incompleteness Theorems.* These theorems mark a crucial point in the history of mathematics. It was because of these theorems that mathematicians became interested in the question what can be proved and what cannot. Hofstadter (1945-) wrote the book *Gödel, Escher, Bach: An eternal golden braid* in which he combines mathematics with art and music in a philosophical way. He popularizes the work of Gödel, Escher and Bach and has no direct influence on mathematical research. In this book he mentions one of the Incompleteness theorems of Gödel briefly, but concludes that it is too hard to understand it immediately: "The Theorem can be likened to a pearl, and the method of proof to an oyster. The pearl is prized for its luster and simplicity; the oyster is a complex living beast whose innards give rise to this mysteriously simple gem" [Hofstadter, 1979, p. 17]. Hofstadter therefore gives "a paraphrase in more normal English: All consistent axiomatic formulations of number theory include undecidable propositions" [Hofstadter, 1979, p. 17]. Although we will not elaborate on the theorems themselves, these theorems gave rise to the central questions in complexity theory: 'Where lies the boundary between computable and not computable?' and 'When is a computational problem easy or hard to solve?'

Around 1936 mathematicians became more and more interested in questions about complexity. Because of this, it was necessary to know how a computational process works. Alan Turing (1912-1954) therefore, stated exactly what a computation is. He invented the Turing machine in 1937, which is an abstract machine that can imitate any formal system. In the next section we will explain briefly how this machine works. The invention of the Turing machine was one of the most important inventions for further research in complexity theory. The machine was used to show whether something was computable and it could also tell how long it took for a certain algorithm to find an answer. The invention of the Turing machine was important because people could now look at the behavior of these algorithms to find out whether they are easy or hard to solve. This was done by looking at the time and space (memory) the Turing machine, and later the computer, needed to solve the problem or to conclude that

there was no solution. These problems were placed in different complexity classes according to their computational difficulty. Later on in this thesis we will introduce the most important classes: P, NP and NP-complete.

Stephen Arthur Cook (1939-) is an American mathematician who is specialized in computational complexity theory. In 1971 he wrote a paper in which he introduced some new terms to study this subject. He also came up with a proof that there is at least one NP-complete problem, namely the Boolean satisfiability problem (SAT). Independently of him, Leonid Levin (1948-) came up with a proof of the same theorem. SAT was the first decision problem which was proven to be NP-complete. In short, this means that any problem in NP to which SAT could be reduced, would also be NP-complete. In 1971 Cook stated one of the *Millennium Prize Problems*[1], namely the 'P versus NP' problem. It is a problem about computationality which asks whether it is true that $P = NP$. In the introduction we mentioned briefly what the complexity classes P and NP are. Namely, the class P contains all decision problems that can be solved in polynomial time and the class NP contains all decision problems of which it can be verified in polynomial time whether there is a solution. When a problem is NP-complete, the problem is in the complexity class NP and every other problem which is in the class NP can be reduced quickly (in polynomial time) to it. From this informal definition of NP-completeness it follows that if we can show that a problem in NP-complete can be solved in polynomial time (or, is in P), then every NP problem can be solved in polynomial time too. If someone is able to show this, it must be true that $P = NP$. Up until now, this problem has still remained unsolved. In Section 5.2 we will explain this Millennium Prize Problem briefly after we have given the definitions of the complexity classes P, NP and NP-complete.

[1] Millennium Prize Problems are a couple of problems in mathematics which are very hard to solve. Therefore, the person who comes up with a correct solution of one of these problems will receive one million dollars being awarded by the Clay Mathematics Institute. This institute stated seven problems and until now only one has been solved.

3 Introductory definitions and decision problems

Decision problems play a central role in this thesis. They are a group of problems or questions which can be answered with 'yes' or 'no'. For example, the Boolean satisfiability problem and the Traveling Salesman Problem are decision problems. TSP will be introduced here and we will also briefly explain the Hamiltonian Circuit problem, the Vertex Cover Problem and the problem CLIQUE. They will be useful for finding a good reduction in Section 6. The decision problem SAT needs a bit more explanation and will therefore be treated in the next section. We will start this section by introducing Alan Turing's invention, the Turing machine. After this we will introduce some important definitions that will be used often in the rest of this thesis.

3.1 The Turing machine

We want to find out whether problems can be solved within reasonable time or whether it can be verified that there is a solution. This can be done by a Turing machine. In this thesis we will only explain informally what a Turing machine is. A formal definition can be found in [Papadimitriou, 1994, p. 19]. A *Turing machine* is an abstract automatic machine. A problem can be written as an algorithm and will be implemented in a Turing machine to find out whether it has a solution. It is a model of computation which forms a basic structure for every computer.

You can think of a Turing machine as a machine that can read and write on an infinitely long tape with a finite alphabet and a finite set of states, starting in an initial state. The alphabet often consists of elements from the set $\{0, 1, B, \triangleright\}$, where B is the blank symbol and $\triangleright$ is the first symbol. There are rules, called *transition functions*, which tell the machine what has to be done in which state. Some examples of these functions are: replacing the symbol by another one, move left or right, change the current state or stop and give output 'yes' or 'no'. When a finite input is given on the tape, the machine reads the first symbol in the current state. The new symbol the machine reads together with the state the machine is in, determine which transition function will be applied.

As an example we will explain what happens if we want to add two numbers with a Turing machine. Let $11 + 111$ be our finite input on the tape, where $11 = 2$ and $111 = 3$. We want to add these numbers such that we get $11111 = 5$. Note that a Turing machine does not interpret the symbol $+$, it is just to separate the two numbers. The Turing machine starts at the first 1 and goes to the right until he reaches the $+$. He replaces the $+$ by a 1 and goes to the right. The machine continues moving to the right until he reaches a blank symbol B which means that he reached the end of the string. We now have the string 111111 which equals 6. The machine goes one place to the left and replaces the last 1 by a B. The machine then goes back to the start and is finished. He has reached $11111 = 5$.

3.2 Some important definitions

To understand decision problems like TSP, Hamiltonian Circuit, Vertex Cover and CLIQUE we will give some important definitions that we will use often in this thesis. For example, we need to know what a *(complete) graph* and how a graph can be represented as an *adjacency matrix*.

Definition 1 (Decision problem)**.** *A decision problem is a problem or question that can be answered with 'yes' or 'no'.*

Graphs are crucial for this thesis, so it helps to understand what a graph is. For example, they are used in all the decision problems mentioned above.

[2] http://www.cis.upenn.edu/~dietzd/CIT596/turingMachine.gif

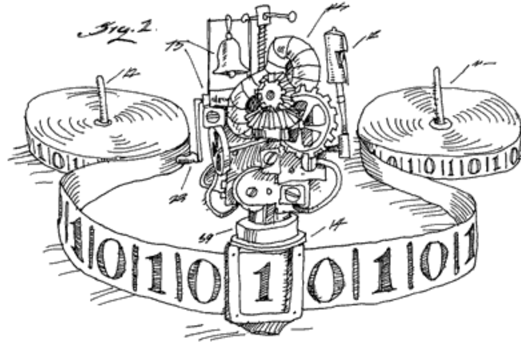

Figure 1: A Turing machine[2]

Definition 2 (Graph)**.** *A graph G is an ordered pair $G = (V, E)$ where $V = \{1, ..., n\}$ is a set of nodes and $E = \{(u, v) \mid u, v \in V\}$ is a set of edges.*

Definition 3 (Complete graph)**.** *A complete graph is a graph $G = (V, E)$ where for every two nodes $u, v \in V$ with $u \neq v$, we have $(u, v) \in E$.*

We can also represent a graph as a binary $n \times n$ matrix A. Thus all elements in A are either 0 or 1. We call such a matrix an adjacency matrix.

Definition 4 (Adjacency matrix)**.** *Given a graph G, the adjacency matrix $A = (A_{ij})$ of G is the $n \times n$ matrix defined by*

$$A_{ij} = \begin{cases} 0, & (i, j) \notin E \\ 1, & (i, j) \in E \end{cases}.$$

We only work with undirected graphs, which means that we have a symmetric matrix, that is, $A_{ij} = A_{ji}$. We therefore only need to look at the upper half of the matrix. We set $A_{ii} = 0$, because we do not allow edges that go from a node to itself. This representation of a graph we will use later for the understanding of the complexity of reductions.

3.3 Decision problems

We will now give some examples of decision problems. A well-known decision problem is the Traveling Salesman Problem. Informally is it about a traveling salesman who has to visit a number of cities. It costs money to drive from one city to another and the salesman has to find the cheapest way to visit each city exactly once. The decision version of TSP is then to find out whether such a tour exists without exceeding a given amount of money.

Formally we are looking at n cities $1, ..., n$, that represent the nodes of a complete graph G. The costs to travel from city v_i to v_j are denoted by $c(v_i, v_j) \in \mathbb{N}$. These costs are weights attached to the edges of G. Assume that for all i and j $c(v_i, v_j) = c(v_j, v_i)$, that is, it takes the same amount of money to travel from city v_i to v_j as from city v_j to v_i. We now want to find the cheapest tour to visit all the cities exactly once. Thus, find a rearrangement π of the the cities $1, ..., n$ such that

$$\left(\sum_{i=1}^{n-1} c(v_{\pi(i)}, v_{\pi(i+1)}) \right) + c(v_{\pi(n)}, v_{\pi(1)}) \tag{1}$$

is minimized. In this expression the total cost of the tour is given, starting at $v_{\pi(1)}$. When every city is visited once in the order given by π, the salesman returns to his hometown $v_{\pi(1)}$.

The problem TSP as described above is not yet a decision problem, because it is not a question which can be answered with yes or no. The decision version of TSP is to answer the question whether there exists a tour, where every city is visited exactly once and the amount of money does not exceed M. Here M is a given integer bound. The graph G existing of n cities, the costs c and the bound M form an instance $I = (G, c, M)$ for the problem. If an instance I has a solution, Schäfer [2012, p. 67] calls I "a "yes-instance"; otherwise I is a "no-instance"".

Another decision problem used in this thesis is the **Hamiltonian Circuit** problem, which is closely related to the Traveling Salesman Problem. We are given a graph G and we want to know whether a given rearrangement of the nodes of G gives us a Hamiltonian circuit. A Hamiltonian circuit is a cycle in a graph where every node is visited exactly once and where we start and finish in the same node. We can thus reformulate TSP as follows: 'Does the graph contain a Hamiltonian circuit?'

Another decision problem is **CLIQUE**. For this problem we are given a graph $G = (V, E)$ and an integer $K \leq |V|$. A *clique* is a complete subgraph in G. The question is whether G contains a clique Q of at least K nodes.

The last decision problem we will explain here is the **Vertex Cover Problem** (VC). For the Vertex Cover Problem we are given a graph $G = (V, E)$ and an integer $L \leq |V|$. A vertex cover W is a subset of V, such that the elements of W 'cover' the edges of G. With 'cover' we mean that if we take an arbitrary edge $(u, v) \in E$, then either $u \in W$ or $v \in W$, or both. The question is whether G contains a vertex cover of at most L nodes.

In the next section we will introduce the problem SAT by explaining the basic ideas of propositional logic and we will give a definition of SAT. The section ends with some examples.

4 The Boolean satisfiability problem

The Boolean satisfiability problem is, next to TSP a well-known decision problem. SAT plays an important role in this thesis. We will use it in Section 6 to show that the Traveling Salesman Problem is NP-complete. SAT is the first known NP-complete problem and it is a problem of propositional logic. To have a better understanding of the problem we give an introduction to propositional logic first. Another way to work with SAT is by using polynomial functions over $\mathbb{F}_2$. We will also explain this method. We end this section by stating SAT and giving some examples.

4.1 Propositional logic

Propositional logic is a formal system that contains *propositional formulas*. Propositional formulas are built up of Boolean variables and so called *connectives*. The connectives used in propositional logic are $\neg$, $\vee$, $\wedge$, $\rightarrow$ and $\leftrightarrow$. $\neg$ means 'negation' or 'not', $\vee$ stands for 'disjunction' or 'or', $\wedge$ is the notation for 'conjunction' or 'and' and $\rightarrow$ and $\leftrightarrow$ stand for 'implication' and 'bi-implication' respectively. Boolean variables (we will call them variables from now on) represent atomic sentences and can be true or false, 1 or 0 respectively. We will state what a propositional formula is.

Definition 5 (Propositional formula)**.** *A propositional formula is a well formed syntactic formula which satisfies the following:*

- *Every variable is a propositional formula.*
- *If p is a propositional formula, then $\neg p$ is one too.*
- *If $(p_1, p_2, ..., p_n)$ are propositional formulas, then $(p_1 \vee p_2 \vee ... \vee p_n)$ also is a propositional formula.*
- *If $(p_1, p_2, ..., p_n)$ are propositional formulas, then $(p_1 \wedge p_2 \wedge ... \wedge p_n)$ is one too.*
- *If p and q are propositional formulas, then $(p \rightarrow q)$ also is a formula.*
- *If p and q are propositional formulas, then $(p \leftrightarrow q)$ also is a formula.*

We then get a collection of propositional formulas which we will denote as F. Propositional formulas have, next to variables, a truth value. Truth values are given by a valuation function v, defined as follows:

Definition 6. *A valuation is a function $v : F \rightarrow \{0, 1\}$ that satisfies the following properties. Let p and q be propositional formulas, then*

1. $v(\neg p) = 1 - v(p)$;
2. $v(p \vee q) = v(p) + v(q) - v(p)v(q)$;
3. $v(p \wedge q) = v(p)v(q)$;
4. $v(p \rightarrow q) = 1 - v(p) + v(p)v(q)$;
5. $v(p \leftrightarrow q) = 1 - v(p) - v(q) + 2 \cdot v(p)v(q)$.

A valuation of a propositional formula is completely determined by the valuation of its distinct variables.

Let us now look at an example, let p_1 and p_2 be two atomic sentences.

$$p_1 = \textit{Strawberries are red} \quad \text{and} \quad p_2 = \textit{Bananas are blue.}$$

From this we can see that the propositional formula $(p_1 \vee p_2)$ is true, because $v(p_1) = 1$. The formula $(p_1 \wedge p_2)$ is not true, since $v(p_2) = 0$. This can also be done for the other formulas containing p_1 and p_2.

In propositional logic we use the term *literals* for variables and negated variables. Thus, p is a literal, but $\neg p$ is one too. With these literals we form clauses. A *clause* C is a disjunction of literals. The length of a clause is given by the number of literals used to form the clause. For example,

$$C = (\neg p_1 \vee p_2 \vee p_4 \vee \neg p_6 \vee \neg p_7)$$

has length five. This formula has, according to the connective $\vee$, truth value 1 if at least one of these five literals has truth value 1. In general this holds for each clause. With these clauses we form propositional formulas which are in *conjunctive normal form* (CNF). A CNF-formula is a conjunction of clauses C_i and is of the form

$$\Phi = C_1 \wedge C_2 \wedge ... \wedge C_m = \bigwedge_{i=1}^{m} C_i.$$

This Boolean formula is, according to the connective $\wedge$, true if and only if each clause has the value 1. Different clauses may contain the same literals, they don't have to be disjoint. For the Boolean satisfiability problem we use these formulas in conjunctive normal form as instances. With this we still cover all propositional formulas, because every propositional formula can be written in conjunctive normal form. An inductive proof of this can be found in [Pohlers and Glaß, 1992, p. 16].

4.2 Polynomial functions over $\mathbb{F}_2$

Another way to look at propositional logic is by using polynomial functions with coefficients in $\mathbb{F}_2$. $\mathbb{F}_2$ equals the set $\{0, 1\}$ with the standard multiplication on it and addition modulo 2. For example, we have that $1 + 1 = 0$. Let $x_1, x_2, x_3, ...$ be variables and let $\mathbb{F}_2[x_1, x_2, x_3, ...]$ be the set of all polynomials with coefficients in $\mathbb{F}_2$ in the variables $x_1, x_2, x_3, ...$ Such a polynomial gives us a *polynomial function*

$$g : \mathbb{F}_2^{\infty} \to \mathbb{F}_2.$$

This means that when given a polynomial f and a sequence $(a_1, a_2, a_3, ...)$ with every $a_j \in \mathbb{F}_2$, we fill in a_1 for x_1, a_2 for x_2 and so on. Since $0^2 = 0$ and $1^2 = 1$ we get that x_j, x_j^2, x_j^3, ... give the same function g. If we have a polynomial in which there is a power of a variable, we can easily make the exponent 1 without changing the function. In this way we can give a different definition of a *"propositional formula"* then Definition 5.

Definition 7 ("Propositional formula"). *A propositional formula satisfies the following properties:*

- *Every variable x_j is a propositional formula.*
- *If f is a propositional formula, then $1 + f$ is one too.*
- *If $(f_1, f_2, ..., f_n)$ are propositional formulas, then $1 + (1 + f_1)(1 + f_2)...(1 + f_n)$ also is a propositional formula.*
- *If $(f_1, f_2, ..., f_n)$ are propositional formulas, then $f_1 f_2 ... f_n$ is one too.*
- *If f_1 and f_2 are propositional formulas, then $1 + f_1 + f_1 f_2$ also is a formula.*
- *If f_1 and f_2 are propositional formulas, then $1 + f_1 + f_2$ also is a formula.*

The ordering in this definition is the same as in Definition 5. We can see that a polynomial function and a propositional formula are the same. Both work with coefficients in the set $\{0, 1\}$ and both are functions that map a formula in the variables $x_1, x_2, x_3, ...$ to the set $\{0, 1\}$. Every propositional formula can be represented as a polynomial function and vice versa and is still mapped to the same element in $\{0, 1\}$.

By using truth tables (Table 1, ..., 5) we show that a polynomial function is the same as a propositional formula. If two columns in a truth table are equal we say that the two formulas are *equivalent*. Here $p_1, p_2, ..., p_n$ are propositional formulas and $f_1, f_2, ..., f_n$ are polynomial functions.

p_1	f_1	$\neg p_1$	$1 + f_1$
0	0	1	1
1	1	0	0

Table 1: $\neg p$ is equivalent with $1 + f$.

p_1	...	p_n	f_1	...	f_n	$p_1 \vee ... \vee p_n$	$1 + (1 + f_1)...(1 + f_n)$
0	...	0	0	...	0	0	0
0	...	1	0	...	1	1	1
$\vdots$		$\vdots$	$\vdots$	...	$\vdots$	$\vdots$	$\vdots$
1	...	0	1	...	0	1	1
1	...	1	1	...	1	1	1

Table 2: $p_1 \vee ... \vee p_n$ is equivalent with $1 + (1 + f_1)...(1 + f_n)$.

p_1	...	p_n	f_1	...	f_n	$p_1 \wedge ... \wedge p_n$	$f_1...f_n$
0	...	0	0	...	0	0	0
0	...	1	0	...	1	0	0
$\vdots$		$\vdots$	$\vdots$	...	$\vdots$	$\vdots$	$\vdots$
1	...	0	1	...	0	0	0
1	...	1	1	...	1	1	1

Table 3: $p_1 \wedge ... \wedge p_n$ is equivalent with $f_1...f_n$.

p_1	p_2	f_1	f_2	$p_1 \to p_2$	$1 + f_1 + f_1 f_2$
0	0	0	0	1	1
0	1	0	1	1	1
1	0	1	0	0	0
1	1	1	1	1	1

Table 4: $p_1 \to p_2$ is equivalent with $1 + f_1 + f_1 f_2$.

p_1	p_2	f_1	f_2	$p_1 \leftrightarrow p_2$	$1 + f_1 + f_2$
0	0	0	0	1	1
0	1	0	1	0	0
1	0	1	0	0	0
1	1	1	1	1	1

Table 5: $p_1 \leftrightarrow p_2$ is equivalent with $1 + f_1 + f_2$.

Having seen these truth tables, we can conclude that propositional formulas are indeed the same as polynomial functions. Let us for example look at the propositional formula $p_1 \wedge ... \wedge p_n$. We know that this formula is true when every conjunct is true. It is easy to see that the polynomial $f_1...f_n$ is mapped to 1 only when all f_j's are mapped to 1.

We can also give another definition of a valuation.

Definition 8 (Valuation)**.** *Let $(a_1, a_2, ...)$ be a point in $\mathbb{F}_2$. A valuation is a function $v : \{polynomials\} \to \mathbb{F}_2$, defined by $f \mapsto f(a_1, a_2, ...)$.*

We can now state the decision problem SAT in two different ways.

4.3 SAT

We explain the satisfiability problem in two different ways. First we state the problem SAT by using polynomial functions and second we state it by using propositional logic. We are given the variables $x_1, ..., x_n$ and a polynomial function $f = f(x_1, ..., x_n)$. SAT can be defined as follows.

Definition 9 (SAT)**.** *Let $x_1, ..., x_n$ be variables and $f = f(x_1, ..., x_n)$ a polynomial function. f is satisfiable if there exists a point $(a_1, a_2, ..., a_n) \in \mathbb{F}_2^\infty$, such that $f(a_1, a_2, ...) = 1$.*

We can also define SAT by using propositional logic. This is the method we will use in the rest of this thesis. We are given a finite set of propositional variables $\{p_1, ..., p_n\}$ and a propositional formula $\Phi = \Phi(p_1, ..., p_n)$ in conjunctive normal form. We want to find a valuation v such that $v(\Phi) = 1$.

Definition 10 (SAT)**.** *A propositional CNF-formula Φ is satisfiable if there exists $\{a_1, a_2, ..., a_n\}$, where $a_n \in \{0, 1\}$, such that any valuation v with $v(p_j) = a_j$ for $1 \leq j \leq n$ satisfies $v(\Phi) = 1$.*

The decision version of SAT will be: 'Is the given propositional formula satisfiable?' This question can be answered with 'yes' or 'no'. A related decision problem is 3-SAT, which is almost the same as SAT. The difference is that an instance of 3-SAT only contains clauses of length three.

Remark 1. *Note that we work with CNF-SAT instead of SAT in this thesis (in what follows we still use the abbreviation SAT when talking about CNF-SAT). This doesn't give any problems, because SAT can be reduced to CNF-SAT in polynomial time. In short this means that a function from instances in SAT to instances in CNF-SAT can be found within reasonable time. A proof can be found in [Papadimitriou, 1994, p. 75/76]. It is useful to use instances which are of only one form, because otherwise we had to work out a lot of different cases in the proofs.*

We will now give some examples of SAT and 3-SAT.

1. Let $\Phi = (p_1 \vee p_2) \wedge (p_3 \vee p_4)$ be a propositional formula. We can immediately see that this formula is satisfiable. If we let $T = \{1, 1, 0, 1\}$ be a truth assignment, it satisfies Φ.

2. Let us now look at $\Phi = (p_1 \vee p_2) \wedge (\neg p_1 \vee p_2) \wedge (p_1 \vee \neg p_2) \wedge (\neg p_1 \vee \neg p_2)$. We cannot immediately see if this formula is satisfiable, but trying some valuations for p_1 and p_2 leaves us with the idea that Φ is not satisfiable. We will give a short proof that shows that Φ is indeed not satisfiable. Suppose that Φ is satisfiable. We start finding a truth assignment by trying to make p_1 true. This implies that the first and the third clause are true. All clauses have to be true, so, let us now look at the second and fourth clause. We see that it is impossible for both to be true, because we cannot have $v(p_2) = v(\neg p_2) = 1$. Thus, from this we can conclude that $v(p_1) = 0$. Continuing with $v(p_1) = 0$ implies that the second and fourth clause are true. It also means that $v(p_2) = v(\neg p_2) = 1$, which is impossible. We have now reached a contradiction. p_1 has to be either true or false, but both possibilities lead to a contradiction. Therefore Φ is not satisfiable. We cannot find a satisfying truth assignment without reaching a contradiction.

3. Let p_1 and p_2 be literals and $\Phi = (p_1 \vee p_2)$ be a satisfiable propositional formula. We will show how we can transform Φ into a 3-CNF-formula in 3-SAT. We can do this by adding a new variable p_3 and its negation. This gives us the following result: $(p_1 \vee p_2 \vee p_3) \wedge (p_1 \vee p_2 \vee \neg p_3)$. This formula is equivalent to Φ. We show this by giving a truth table, see Table 3.

p_1	p_2	p_3	$p_1 \vee p_2$	$\Leftrightarrow$	$((p_1 \vee p_2 \vee p_3)$	$\wedge$	$(p_1 \vee p_2 \vee \neg p_3))$
0	0	0	0	**1**	0	0	1
0	0	1	0	**1**	1	0	0
0	1	0	1	**1**	1	1	1
0	1	1	1	**1**	1	1	1
1	0	0	1	**1**	1	1	1
1	0	1	1	**1**	1	1	1
1	1	0	1	**1**	1	1	1
1	1	1	1	**1**	1	1	1

These examples give us an idea of how instances of SAT and 3-SAT work. In the next section we will introduce complexity theory and give an example of a reduction from SAT to 3-SAT.

5 Complexity theory and reduction

In this section we will explain some important concepts of complexity theory. The complexity classes P, NP and NP-complete are defined and we will explain what it means for a computer or Turing machine to find a solution in polynomial time. Next we will introduce the concept of reduction which is followed by an example. In this example we show that the decision problem 3-SAT, which contains only instances with clauses of length three, is NP-complete, by reducing SAT to it.

5.1 Complexity theory

The theory of complexity can be seen as subdividing decision problems into different classes according to their computational difficulty. A *complexity class* is a set of decision problems which can be decided by an automatic machine M (e.g. Turing machine). Every class operates in a computation mode. This mode is used to see when a machine will accept its input. The most important ones are the deterministic mode and the nondeterministic mode. Deterministic means that a machine generates an output, based only on the input and the steps that follow logically from it. In the nondeterministic mode there can be external influences of the user, a place to store information and extra tapes in the machine to perform multiple tasks at once. This is much more efficient and uses less space (not necessarily time). In the deterministic mode you have to remember everything you did before which uses a great amount of space.

With an example we will have a closer look at the difference between deterministic and nondeterministic. We will use the Traveling Salesman Problem to do this. We were given n cities and costs c_{v_i,v_j} to travel between two cities v_i and v_j and we wanted to find out whether there is a tour (starting and finishing in the same city), visiting every city exactly once, costing at most M money. If we try to find out in a deterministic way if such a tour exists, we are not doing it very efficiently. We then have to consider $(n-1)!/2$ tours. Our starting point is randomly chosen, which means that there are $(n-1)!$ possible routes to follow. But, we know that $c_{v_i,v_j} = c_{v_j,v_i}$ which implies that there are $(n-1)!/2$ tours left. This takes too much time to compute. A better way to look at this problem is in a nondeterministic way where we can store our previously gained information and are able to recall it later on. We then only need space which is proportional to n. It becomes clear that working in the nondeterministic mode uses less space if n gets larger, but you have to be capable of a lot more things (such as storing information and performing multiple tasks at once). It is not always the case that it will be faster to compute something nondeterministically. We are therefore pleased if we can solve the problem deterministically in reasonable time, because this means that the problem is not very hard.

Next to operating in a certain computation mode, a complexity class has another property. Namely, we want to bound the time or space a Turing machine needs to accept its input. For every input x of size $n \in \mathbb{N}$, machine M uses at most $f(n)$ units of time or space. The bound is defined as follows:

Definition 11 (Bound)**.** *A bound is a function $f : \mathbb{N} \to \mathbb{N}$ which maps nonnegative integers to nonnegative integers.*

We don't want $f(n)$ to be too large. For example, we don't want it to be exponential, say 2^n, because the number of time steps will then grow too fast and this is not efficient. $n!$ will also grow too fast. If the number of steps is too large, it means we cannot find a solution or even verify if there will be a solution. The following quote is a beautiful example of exponential growth.

> According to the legend, the game of chess was invented by the Brahmin Sissa to amuse and teach his king. Asked by the grateful monarch what he wanted in return, the wise man requested that the king place one grain

> of rice in the first square of the chessboard, two in the second, four in the third, and so on, doubling the amount of rice up to the 64th square. The king agreed on the spot, and as a result he was the first person to learn the valuable - albeit humbling - lesson of *exponential growth*. Sissa's request amounted to $2^{64} - 1 = 18,446,744,073,709,551,615$ grains of rice, enough rice to pave all of India several times over! [Dasgupta et al., 2006, p. 248].

In complexity theory we want everything to be computed in *polynomial time*, which is more efficient. Polynomial time means that the time that it takes a machine to run through the number of steps, is bounded above by a polynomial function in the input size. Let $f(n) = c_k n^k + c_{k-1} n^{k-1} + ... + c_1 n + c_0$ be a polynomial, where each c_k is constant. $f(n)$ has degree k, which means that the highest power of n is k. Then $f(n) = O(n^k)$ which means that the first term of the polynomial captures the rate of growth. We will now define this O-function.

Definition 12 (O-function)**.** *Let f and g be two functions from $\mathbb{N}$ to $\mathbb{N}$. $f(n) = O(g(n))$ means that there are c and m, both elements of $\mathbb{N}$ such that $\forall n \geq m$ it holds that $f(n) \leq c \cdot g(n)$.*

Informally this means that f grows as fast as g or slower, f is of the order of g. We will give an example of this O-function to make it more clear. Let $\mathcal{T}$ be a truth table with entries zeros and ones. If we want to check, for example, whether $\mathcal{T}$contains only ones, a machine M has to visit every box in the table to check whether there is a 1 or not. Suppose that $\mathcal{T}$ consists of n rows and n columns. Then the input size is n^2, because M has to visit exactly $n \cdot n = n^2$ boxes. Therefore, this example can be computed in linear time in the size of the input.

Another example of the O-function is $O(nlog(n))$. This is known as the order of a sorting algorithm. Think of a set of graded exams that have to be sorted alphabetically. We randomly pick one exam x_1 to start forming a new set. For every next exam we check whether it is alphabetically before or after x_1 and place it there. This does not give us an ordered sequence, but two subsets (one on the left and one on the right of x_1). Next, we will randomly pick one exam x_2 on the left of x_1 and an exam x_3 on the right of x_1. For both subsets we do the same process of sorting exams alphabetically again, but now around x_2 and x_3 instead of x_1. Repeating this process until we are finished takes $O(nlog(n))$ time.

We will now introduce the two complexity classes P and NP, but first we define what an algorithm is.

Definition 13 (Algorithm)**.** *An algorithm for a decision problem is a finite sequence of operations that specifies a calculation that solves every instance of the problem in a finite amount of time.*

An algorithm is mostly seen as something deterministic. That is, with every step the algorithm has one possible option to choose from. For an algorithm to be nondeterministic, there are two phases, a guessing phase and a verification phase. In the guessing phase a string of symbols is produced, which might correspond to a solution of the instance. In the verification phase, there is a deterministic algorithm that checks the length of the guessed string and verifies whether it is a solution of the instance. In this thesis we work with algorithms that are implemented by a Turing machine.

Definition 14 (Complexity class P)**.** *For a decision problem Π to belong to the complexity class P there has to be an algorithm that determines in polynomial time whether each instance I of Π is a 'yes-instance' or not.*

The complexity class P can thus be seen as the class of decision problems which can be solved by a deterministic Turing machine in polynomial time. A problem in P is for example the question what the greatest common divisor (gcd) is of two or more natural numbers. The class NP is a larger class of decision problems of which it can

be verified by a deterministic Turing machine in polynomial time if there is a solution when an instance is given. A definition is given below.

Definition 15 (Complexity class NP)**.** *For a decision problem Π to belong to the complexity class NP, there has to be a deterministic algorithm that can verify in polynomial time whether a given instance is a 'yes-instance' or not.*

Note that the definition of the complexity class NP is equivalent to saying that there is a nondeterministic algorithm that accepts the problem in polynomial time. With 'accept' we mean that there is a guess that is answered 'yes' in the verification phase of a nondeterministic algorithm.

Remark 2. *By saying that a problem is P, NP or NP-complete, we mean that the problem belongs to one of these complexity classes respectively.*

For example let us look at TSP. This problem lies in the complexity class NP. This means that there have to be some difficulties in the problem which cannot be deterministically computed in polynomial time, otherwise the problem would have been in P. But what makes this problem hard? The problem is whether there is a tour with costs less than M. To see if there is such a tour, up until now, the only way to find it is by trial and error. Just try every permutation of the cities and see if the costs are less than M. But as we saw in 5.1 there are $(n-1)!/2$ possible tours to visit n cities, which cannot be done in polynomial time. We therefore cannot find an algorithm that determines in polynomial time whether each instance is a 'yes-instance' or not. Hence TSP is not in P according to the definition. But it is in NP. If we are given a tour visiting all cities, we are able to check in polynomial time whether this tour has costs less than M.

An important difference between the complexity classes P and NP lies in the words 'solve' and 'verify'. Johnson and Garey [1979, p. 28] say the following about this:

> Notice that polynomial time verifiability does not imply polynomial time solvability. In saying that one can verify a "yes" answer for a TRAVELING SALESMAN instance in polynomial time, we are not counting the time one might have to spend in searching among the exponentially many possible tours for one of the desired form. We merely assert that, given any tour for an instance I, we can verify in polynomial time whether or not that tour "proves" that the answer for I is "yes."

It now becomes clear that the class P is a subclass of the class NP. When you can solve a problem you can also verify that there is a solution, but not the other way around. Figure 2 shows how the complexity classes are related to each other. We then see that the complexity class NP-complete is also a subclass of NP and is disjoint with the class P. Informally, when a problem is NP-complete, the problem is in NP and every other problem which is in NP can be reduced in polynomial time to this problem. We will now have a closer look at these reductions.

5.2 Polynomial time reduction

To understand the definition of NP-completeness we will show what it means for a problem to reduce to another problem. Reduction is used to show that a particular problem is as difficult as another. We will start by giving the definition of a polynomial time reduction.

Definition 16 (Polynomial time reduction)**.** *A polynomial time reduction is a transformation from a decision problem Π_1 to a decision problem Π_2. This transformation function $f : I_1 \mapsto I_2$ maps every instance of Π_1 to an instance of Π_2 such that:*

[3] http://upload.wikimedia.org/wikipedia/commons/4/4a/Complexity_classes.png

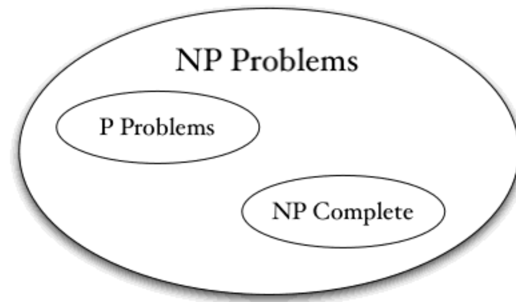

Figure 2: The complexity classes P, NP, NP-complete[3]

1. *the transformation can be done by a deterministic algorithm in polynomial time, proportional to the size of* I_1*;*

2. I_1 *is a 'yes-instance' of the problem* $\Pi_1 \Leftrightarrow I_2$ *is a 'yes-instance' of the problem* Π_2*.*

We write this relation between two decision problems as '$\Pi_1 \preceq \Pi_2$' and say that 'Π_1 reduces to Π_2' or 'Π_1 transforms to Π_2'. A consequence of this definition is that polynomial time reductions are transitive.

Lemma 1. *If* $\Pi_1 \preceq \Pi_2$ *and* $\Pi_2 \preceq \Pi_3$*, then* $\Pi_1 \preceq \Pi_3$*.*

Proof. Let I_1, I_2 and I_3 be three instances of the decision problems Π_1, Π_2 and Π_3 respectively. Let $f : I_1 \rightarrow I_2$ and $g : I_2 \rightarrow I_3$ be two polynomial time reductions. $h = g \circ f$ and then $h(I_1) = g(f(I_1))$. We must check both parts of the definition of a polynomial time reduction.

- Because f is a polynomial time reduction between Π_1 and Π_2 we know that I_1 is a 'yes-instance' of $\Pi_1 \Leftrightarrow I_2 = f(I_1)$ is a 'yes-instance' of Π_2. Similarly for g we know that $f(I_1)$ is a 'yes-instance' of $\Pi_2 \Leftrightarrow I_3 = g(f(I_1))$ is a 'yes-instance' of Π_3. Therefore it follows that I_1 is a 'yes-instance' of $\Pi_1 \Leftrightarrow I_3 = g(f(I_1)) = h(I_1)$ is a 'yes-instance' of Π_3.

- Let $p(n)$ and $q(n)$ be two polynomials. Suppose that f takes time $\leq p(n)$ and is bounded by the size of I_1, which is n. Thus $f(I_1)$ can be computed in time polynomial in n. Similarly suppose that g takes time $\leq q(n)$ and is bounded by the size of $f(I_1) = I_2$, which is polynomial in n. Therefore $h(I_1) = g(f(I_1))$ can be computed in time polynomial in n.

We have found a polynomial transformation function $h : I_1 \rightarrow I_3$ that maps every instance of Π_1 to an instance of Π_3 such that both conditions hold. □

Recall that, informally, the definition of the complexity class NP-complete was that a decision problem in this class had to be in NP and that every other problem in NP could be reduced to this problem in polynomial time. But how can we manage to show that *every* problem in NP can be reduced to this problem? To simplify things we state the following lemma.

Lemma 2. *If* Π_1 *and* Π_2 *are in NP,* Π_1 *is NP-complete and* $\Pi_1 \preceq \Pi_2$*, then* Π_2 *is NP-complete.*

Proof. To show that Π_2 is NP-complete we have to show that Π_2 is in NP (but this is already given) and that for every other problem $\Pi' \in$ NP holds that $\Pi' \preceq \Pi_2$. Let $\Pi' \in$ NP be arbitrary. Because Π_1 is NP-complete we know that $\Pi' \preceq \Pi_1$. It was given that $\Pi_1 \preceq \Pi_2$. Using lemma 1 we can conclude that $\Pi' \preceq \Pi_2$. Therefore Π_2 is NP-complete. □

Thus, we only have to show that one problem, of which we know it is NP-complete, can be reduced to the original problem. We can now state the definition of the complexity class NP-complete.

Definition 17 (Complexity class NP-complete)**.** *A decision problem is NP-complete if the following two conditions hold.*

1. $\Pi \in NP$;
2. *a decision problem* Π' *of which we know it is NP-complete can be reduced in polynomial time to* Π.

From the definition of NP-completeness it follows that if we can show that a problem in NP-complete can be solved in polynomial time (or, more formally, is in P), then every NP problem can be solved in polynomial time too. If someone is able to show this, it must be true that P=NP. Until now, nobody managed to show that P=NP and therefore it is commonly believed that P≠NP.

When having the definition of NP-completeness, another thing to mention here is the theorem, proven by Cook and Levin, which says that (CNF-)SAT is NP-complete.

Theorem 1. *SAT is NP-complete.*

SAT was the first known NP-complete problem. Before this it was not possible to use the standard reduction process as explained above. We will not give a proof of this theorem, because it is not relevant for the purpose of this thesis, but a proof can be found in [Johnson and Garey, 1979, p. 39-44].

Ruben Gamboa and John Cowles, associate professor and professor of the University of Wyoming respectively, have written an article in which they prove Cook's theorem. Before proving it formally they give an informal version of the theorem.

> **Theorem 1 (Cook, Levin).** Let M be a Turing Machine that is guaranteed to halt on an arbitrary input x after $p(n)$ steps, where p is a (fixed) polynomial and n is the length of x. $L(M)$, the set of strings x accepted by M, is polynomially reducible to satisfiability [Gamboa and Cowles, 2004, p. 100].

The idea of the proof is to show that any NP problem can be reduced to SAT (instead of the usual NP-complete problems used for reductions.) Of problems in NP we know that there is a nondeterministic algorithm that accepts the problem in polynomial time. The theorem shows the connection between satisfiability and Turing machines.

5.3 Example: 3-SAT is NP-complete

In this section we will give an example of the process of reduction. We will show that the decision problem 3-SAT is NP-complete by reducing SAT, of which we already know it is NP-complete, to it. Before proving this, we state the following lemma.

Lemma 3. *Let* C_1 *and* C_2 *be arbitrary clauses in SAT. Then it holds that* C_1 *is equivalent with* $(C_1 \vee C_2) \wedge (C_1 \vee \neg C_2)$.

Proof. We prove this lemma by using a truth table. We see that $(C_1 \vee C_2) \wedge (C_1 \vee \neg C_2)$ and C_1 have the same truth values for every value for C_1 and C_2.

C_1	C_2	C_1	$\Leftrightarrow$	$((C_1 \vee C_2)$	$\wedge$	$(C_1 \vee \neg C_2))$
0	0	0	**1**	0	0	1
0	1	0	**1**	1	0	0
1	0	1	**1**	1	1	1
1	1	1	**1**	1	1	1

□

Theorem 2. *3-SAT is NP-complete.*

Proof. The proof consists of four steps.

1. Show that 3-SAT is in NP.

2. Find a transformation from SAT to 3-SAT. Let F and F' be sets of Boolean formulas in SAT and 3-SAT respectively and let $\Phi \in F$ and $\Psi \in F'$. Then, a transformation is a function $f : \Phi \mapsto \Psi$.

3. Show that f indeed is a transformation, by showing that Φ is satisfiable $\Leftrightarrow$ Ψ is satisfiable.

4. Show that the transformation can be done in polynomial time. That is, in time bounded by a polynomial in the size of Φ.

We will start this proof by showing that 3-SAT is in NP. Let $\Psi = \Psi(p_1, ...p_l)$ be an instance of 3-SAT of length n with $p_1, ..., p_l$ some literals. If we are given $\{a_1, ..., a_l\}$ with $a_l \in \{0, 1\}$ and $v(p_j) = a_j$ with $1 \leq j \leq l$, we see that a computer only has to read $\Psi(a_1, ..., a_l)$ and check whether $v(\Psi) = 1$ or 0. This can be done in time at most $O(n)$ which is linear in n.

In the second and third part of the proof we want to find a reduction from SAT to 3-SAT and show that this is indeed a transformation. Let Φ be a given instance of SAT. Φ is built up of literals $p_1, ..., p_t$ and clauses $C_1, ..., C_m$ and every clause C_j is built up of some of these p_i. Φ has length $n = l_1 + ... + l_m$ where l_j is the length of C_j, $1 \leq j \leq m$. We want to construct an equivalent formula Ψ in 3-SAT containing only clauses of length three, by adding some new variables.

The clauses in SAT are of different length l_j and we can divide these lengths over four groups: $l_j = 1$, $l_j = 2$, $l_j = 3$ and $l_j > 3$. Notice that we don't have to look at clauses of length 3, because they already have the right length. For clauses in SAT with $l_j = 1$ and $l_j = 2$ we use Lemma 3.

- $l_j = 1$: Let $C_1 = (p_1)$ and $C_2 = (q_1)$ then according to Lemma 3, (p_1) is equivalent to $(p_1 \vee q_1) \wedge (p_1 \vee \neg q_1)$. Using the lemma again for both clauses and $C_2 = (q_2)$ gives us an equivalence between (p_1) and

$$((p_1 \vee q_1 \vee q_2) \wedge (p_1 \vee q_1 \vee \neg q_2) \wedge (p_1 \vee \neg q_1 \vee q_2) \wedge (p_1 \vee \neg q_1 \vee \neg q_2)).$$

  We now have constructed an equivalent formula for (p_1) in 3-SAT.

- $l_j = 2$: Let $C_1 = (p_1 \vee p_2)$ and $C_2 = (q_1)$, then according to Lemma 3 $(p_1 \vee p_2)$ is equivalent to $((p_1 \vee p_2 \vee q_1) \wedge (p_1 \vee p_2 \vee \neg q_1))$. We have thus constructed an equivalent formula for $(p_1 \vee p_2)$ in 3-SAT.

- $l_j > 3$: Let $C_j = (p_1 \vee p_2 \vee ... \vee p_{l_j})$ be a clause of length l_j. C_j is then equivalent to

$$((p_1 \vee p_2 \vee q_1) \wedge (\neg q_1 \vee p_3 \vee q_2) \wedge (\neg q_2 \vee p_4 \vee q_3) \wedge ... \wedge (\neg q_{l_j - 3} \vee p_{l_j - 1} \vee p_{l_j}). \quad (2)$$

To show that this equivalence for $l_j > 3$ holds we must check several cases for the truth values of the p_i's and q_k's. We know that $v(C_j) = 1$ whenever at least one of $p_1, ..., p_{l_j}$ is true. It then follows that all the clauses of (2) need to be true if only one of the p_{l_j}'s is true. We now have to assign specific truth values to each q_k, depending on the truth values of p_t. We will look at four different cases.

1. Let all p_t be false. This means that C_j is false and therefore there needs to be only one false clause in equation (2). Let q_1 be false. Then the first clause is false and therefore is equation (2) false.

2. Let $(p_1 \vee p_2)$ be true. Then C_j is true, which implies that all the clauses of equation (2) must be true. The first clause is true, because of our assumption. For the same reason, the last clause can only be true if $v(\neg q_{l_j-3}) = 1$ and thus $v(q_{l_j-3}) = 0$. But then q_{l_j-3} and p_{l_j-2} are false in the clause before the last one, which implies that $v(\neg q_{l_j-2}) = 1$ for the clause to be true. It follows that, when we continue this backwards, all q_k's must have the truth value 0 for all the clauses to be true. This means that there is a satisfying truth assignment.

3. Let $p_{l_j-1} \vee p_{l_j}$ be true. Then C_j is again true, so all the clauses of equation (2) must be true. This case works in the opposite direction as the previous case. So, we now have to choose all q_k's to be true to find a satisfying truth assignment.

4. Let p_t be true if $2 < t < l_j - 1$. It means that C_j is true and again all the clauses of equation (2) must be true. The first clause can only be true if $v(q_1) = 1$ and the last clause can only be true if $v(\neg q_{l_j-3}) = 1$. For the clauses in between the following holds:

$$q_k = \begin{cases} 1, & 1 \leq k \leq t-2 \\ 0, & t-1 \leq k \leq l_j - 3 \end{cases}.$$

   q_k is true until you reach the clause which contains p_n. For this clause and the clauses which come next we choose $\neg q_k$ to be true.

The last part of this proof is to show that the transformation can be done in polynomial time in the size of Φ in SAT. Let Φ be an instance of SAT of length $n = l_1 + ... + l_m$, where l_j is the length of C_j, $1 \leq j \leq m$. For clauses of length ≤ 3 we can do the transformation in time at most $O(1)$ which can be neglected. We only have to look at clauses of length > 3. So if we are given a clause C_j of length $l_j > 3$ we see in equation 2 that we must create a new clause for every literal except the last two and the first two. This means that we need l_j steps to transform clause C_j. Repeating this for every clause means that we need at most $O(l_1 + ... + l_m) = O(n)$ time to transform a formula Φ in SAT to a formula Ψ in 3-SAT.

□

In a similar way, it can be shown of other decision problems that they are NP-complete. Figure 3 shows the reductions between some common problems. It is a rough sketch, but it can help to understand how problems are related to each other.

In this section we gave an introduction to complexity theory and the process of reduction. We defined the complexity classes P, NP and NP-complete and we explained what a polynomial time reduction is. We ended this section with an example of the process of reduction. Namely, we proved that 3-SAT is NP-complete by reducing SAT to it.

In the next section we will work out some more reduction proofs with the goal to reduce SAT to TSP. These proofs are based on transformations of the instances of different NP-complete problems given in an article written by Karp [1972, p.97]. In his article he did not prove that his transformations really worked, so we will work these reductions out in more detail.

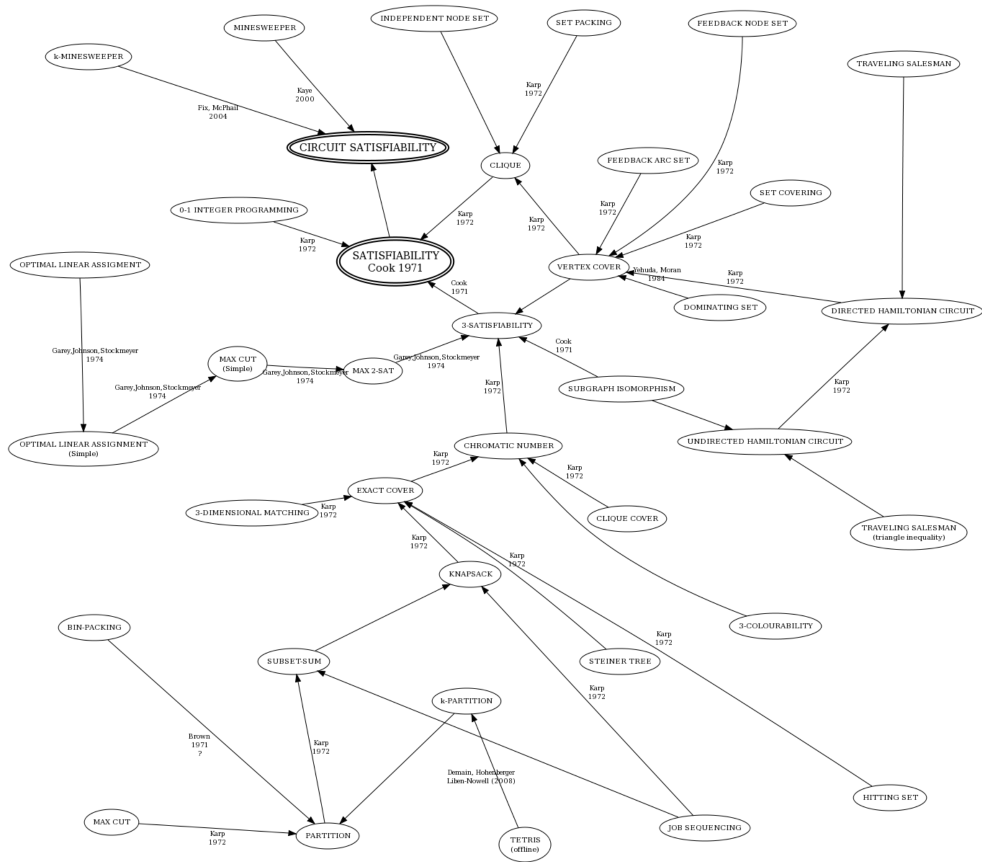

Figure 3: Scheme of reductions[4]

[4] http://www.edwardtufte.com/bboard/images/0003Nw-8838.png

6 Reduction of SAT to TSP

It is known that SAT and TSP are NP-complete. So, the goal of this thesis was not to show that TSP is NP-complete, but to find out whether a direct polynomial transformation function from SAT to TSP could be found. Unfortunately we only found the composition of the reductions in between SAT and TSP. We will show these reductions step by step based on an article written by Karp [1972, p. 97-98]. In this article Karp gave transformations of instances for different NP-complete problems, but he did not provide detailed proofs. The idea is to work out some of these intermediate proofs and conclude with some remarks about the direct reduction from SAT to TSP which is possible in theory.

SAT was the first known NP-complete problem, but it is not very often used for reductions. The reason for this is that the problem has many different instances and thus a lot of different cases have to be worked out in the proofs. More often, mathematicians use 3-SAT for reductions. For example, Schäfer [2012, p. 74] proved that the Vertex Cover problem (VC) is NP-complete by using a reduction from 3-SAT. In most literature TSP is shown to be NP-complete via the undirected Hamiltonian Circuit Problem, because HC is similar to TSP. The reductions we will show are SAT $\preceq$ CLIQUE, CLIQUE $\preceq$ VC, VC $\preceq$ HC and HC $\preceq$ TSP.

6.1 SAT $\preceq$ CLIQUE

We will start with the proof of the theorem that SAT can be reduced to CLIQUE in polynomial time. The proof of this theorem is based on the reduction of the instances given by Karp [1972, p. 97]. We know of both problems that they are in NP and of SAT we also know that it is NP-complete. Therefore we just need to show the process of reduction and that the reduction can be done in polynomial time.

Theorem 3. *SAT $\preceq$ CLIQUE*

Proof. For this reduction we are given a Boolean formula $\Phi = C_1 \wedge C_2 \wedge ... \wedge C_m$ with n literals $x_1, ..., x_n$ and m clauses. Within these n literals there may be some double ones if they appear in more than one clause. We want to construct a graph $G = (V, E)$ in polynomial time by representing it as a matrix, such that G has a clique Q of size at least $K = m$ if and only if Φ is satisfiable.

First we construct the graph $G = (V, E)$ according to the given Boolean formula Φ. Let V and E be stated as follows:

$$\begin{aligned} V &= \{\langle x, i\rangle \mid x \text{ is a literal in clause } C_i\}, \\ E &= \big\{\{\langle x, i\rangle, \langle y, j\rangle\} \mid \text{where } i \neq j \text{ and } x \neq \neg y\big\}. \end{aligned}$$

Thus, for every literal in each clause in Φ we add a node to the set V. For each clause C_i there is a subset of V, namely the group of nodes that correspond to the literals in this clause. We call such a group of nodes a "clause-gadget". To explain the set E, it is easier to say which edges do **not** belong to E. Edges that are not in E are edges that connect nodes within a clause-gadget. So, the nodes in a clause-gadget are all separate nodes without any connection. Another group of edges which is not in E are the edges that connect a literal to its negation. We now have given a transformation of the Boolean instance to a graph G. This transformation can be done in polynomial time bounded by the size of Φ, depending on n and m. Indeed, $\#V = |\Phi|$ and $\#E \leq |\Phi|^2$.

We will now prove the following lemma.

Lemma 4. *Φ is satisfiable if and only if G has a clique Q of size at least $K = m$.*

Proof. ($\Rightarrow$) Let T be a truth assignment that satisfies Φ. We know that our formula Φ is true, which means that each clause has at least one true literal. Because we do not connect the nodes within the clause-gadgets with each other, we must have exactly

m nodes in our clique. That is, exactly one literal in each clause must be true, and therefore we let these m nodes correspond to a truth assignment for Φ. But how do we connect these m nodes such that we obtain a clique? Because we have drawn the edges according to the set E, we can form a clique from these m nodes if and only if we do not have contradicting literals within these m nodes. Therefore, G has a clique of size $K = m$ whenever Φ is satisfiable.

($\Leftarrow$) To prove the implication to the left we show that if G has a clique Q with size $K = m$, then the Boolean formula Φ is satisfiable. Let Q be a clique of size $K = m$. Each of these m nodes belongs to a different clause-gadget. This can easily be seen by a contradiction. Suppose two of the m nodes belong to the same clause-gadget. A clique is a complete subgraph, but between two nodes within a clause-gadget there are no edges. Therefore, we cannot have a clique if not every node in the clique belongs to a different clause-gadget. We now let all nodes in Q correspond to true literals in Φ. Because each node in Q belongs to a different clause-gadget, we have that each clause becomes true, which implies that we have found a satisfying truth assignment for Φ. Therefore, Φ is satisfiable if G has a clique of size $K = m$. □

With this, we have shown that a graph G can be constructed in polynomial time, and that Φ is satisfiable if and only if the graph G contains a clique of at least $K = m$ nodes. □

Now, having proved this reduction, it will lead us one step further in our reduction process towards TSP. We have three more reductions to go: CLIQUE $\preceq$ VC, VC $\preceq$ HC and HC $\preceq$ TSP.

6.2 CLIQUE $\preceq$ VC

We will now prove the reduction from CLIQUE to Vertex Cover found by Karp [1972, p. 98].

Theorem 4. *CLIQUE $\preceq$ VC*

Proof. For this reduction we are given a graph $G = (V, E)$ and an integer $L = |V| - K$ where K is the size of the clique in G. We want to construct a new graph $G_1 = (V_1, E_1)$ in polynomial time by representing it as a matrix, such that G_1 has a vertex cover $W \subseteq V_1$ with $|W| \leq L$ if and only if G has a clique Q with $|Q| \geq K$. Again we first show how to construct a graph G_1 in polynomial time and second we prove the bi-implication stated above.

Let G_1 be the *complement* of the graph G. That is, the nodes remain the same, but G_1 consists of the edges opposite to those of G. Thus, G_1 contains the edges in a complete graph minus the edges in G. We thus have

$$\begin{aligned} |V_1| &= |V|, \\ |E_1| &= |E|^C = |V| \cdot (|V| - 1)/2 - |E|. \end{aligned}$$

This gives us a transformation of CLIQUE to Vertex Cover. The transformation can be done in polynomial time bounded by the size of G. To see this, we represent our given graph G as an $n \times n$ adjacency matrix A, where $n = |V|$, and walk through it to form a new binary $n \times n$ matrix B belonging to the complementary graph G_1. That is, if $a_{ij} = 1$ we write $b_{ij} = 0$ and if $a_{ij} = 0$ we write $b_{ij} = 1$, where $b_{ij} \in B$ and $i, j \in \{1, ..., |V|\}$, $i \neq j$. We also set $a_{ii} = b_{ii} = 0$. This process can be done in polynomial time of order $O(|V|^2)$.

The next thing we have to show is the following lemma.

Lemma 5. *G has a clique Q with $|Q| \geq K$ if and only if G_1 has a vertex cover W with $|W| \leq L$.*

Proof. If we prove that G has a clique Q with $|Q| = K$ if and only if G_1 has a vertex cover W with $|W| = L$, we are finished, because then we satisfy both parts of the lemma.

($\Rightarrow$) Let Q be a clique in G of size K. This means we have $|V| - K$ nodes left in $G \backslash Q$. We will show that these nodes form a vertex cover for G_1. Let (u, v) be an edge in G_1. This means that not both u and v can be elements of Q, because otherwise (u, v) would have been an edge in G. Thus at least one of the two nodes of (u, v) is contained in $G \backslash Q$. Therefore, (u, v) is covered by $G \backslash Q$. Since this is true for every edge in G_1 it holds that $G \backslash Q$ is a vertex cover for G_1. Thus, if G has a clique Q of size K, G_1 has a vertex cover of size $|V| - K$.

($\Leftarrow$) Let W be a vertex cover in G_1 with size $|V| - K$. For every two nodes $u, v \in V$ it holds that, if (u, v) is an edge in G_1, then either $u \in W$ or $v \in W$ or both. It also holds that whenever neither u nor v is in W, then (u, v) is an edge in G. This means that all nodes that are not in W are connected by an edge, which implies that $G \backslash W$ is a clique. The size of this clique is then $|V| - L = K$. Thus, G_1 has a vertex cover W with $|W| = L$. ☐

We now have shown that we can construct a graph G_1 from G in polynomial time and that G has a clique Q with $|Q| \geq K$ if and only if G_1 has a vertex cover W with $|W| \leq L$. ☐

Is is now possible to link these two theorems to each other. If we take $|Q| = K = m$ in the second proof, where m is the number of clauses, we can immediately see that if we have a Boolean formula Φ and we construct the graph $G_1 = (V_1, E_1)$, the following lemma is true.

Lemma 6. *Φ is satisfiable if and only if G_1 has a vertex cover W with $|W| \leq |V_1| - m$.*

We will illustrate this lemma with an example.

Example 1. *Let $\Phi = (p)$ be a Boolean formula with only one clause and one literal. First, we reduce (p) to a graph G. We construct G such that we get $V = \{\langle p, 1 \rangle\}$ and $E = \emptyset$. This is the graph with only one node and no edges. We want to find a clique of at least $K = 1$ nodes. This clique exists, so we can conclude that (p) is satisfiable. We then continue with reducing this graph G to another graph G_1 to check whether G_1 contains a vertex cover of at most $L = 1 - 1 = 0$ nodes. To construct G_1 we only have to look at edges, because the nodes remain the same. In this case we thus have that $G = G_1$. Therefore we can conclude that G_1 indeed has a vertex cover of at most 0 nodes, because there are no edges that need to be covered.*

As you have noticed, this example was not very hard. After the next reduction we will give another example which is a bit more interesting.

6.3 VC $\preceq$ HC

We will now continue with the next reduction towards TSP, which is the reduction from Vertex Cover to Hamiltonian Circuit. The proof is based on a proof given by Johnson and Garey [1979, p. 56-60] and they have based their proof on the article by Karp [1972, p. 98].

Theorem 5. *VC $\preceq$ HC*

Proof. For this reduction we are given a graph $G_1 = (V_1, E_1)$ and an integer $L \leq |V_1|$. We want to construct a graph $G_2 = (V_2, E_2)$ in polynomial time by representing it as a matrix, such that G_1 has a vertex cover $W \subseteq V_1$ of size at most L if and only if G_2 has a Hamiltonian circuit. We start with constructing the graph G_2 and show that this can be done in polynomial time. Second, we prove the bi-implication stated above.

G_2 is made up of several components. First, G_2 has L "selector vertices", as they are called in [Johnson and Garey, 1979, p. 56], $s_1, ..., s_L$. They are used to select L nodes from V_1.

Second, we add "cover testing components" to G_2 for every edge in E_1. They are used to test whether at least one of the endpoints of each edge is part of the selector vertices. A cover testing component for edge $e = \{u, v\} \in E_1$ can be made as follows. It consists of 12 nodes and 14 edges:

$$\begin{aligned} V_2(e) &= \{\langle u,e,i\rangle, \langle v,e,i\rangle \mid 1 \leq i \leq 6\}, \\ E_2(e) &= \big\{\{\langle u,e,i\rangle\, \langle u,e,i+1\rangle\}, \{\langle v,e,i\rangle, \langle v,e,i+1\rangle\} \mid 1 \leq i \leq 5\big\} \\ &\cup \big\{\{\langle u,e,3\rangle, \langle v,e,1\rangle\}, \{\langle v,e,3\rangle, \langle u,e,1\rangle\}\big\} \\ &\cup \big\{\{\langle u,e,6\rangle, \langle v,e,4\rangle\}, \{\langle v,e,6\rangle, \langle u,e,4\rangle\}\big\}. \end{aligned}$$

An illustration can be found in [Johnson and Garey, 1979, p. 57], but Figure 4 is also a good illustration. Having constructed this component, we only connect the endpoints $\langle u,e,1\rangle$, $\langle v,e,1\rangle$, $\langle u,e,6\rangle$ and $\langle v,e,6\rangle$ to other parts of the graph to ensure that we visit each node of this component exactly once. It is always the case that if we arrive at a point $\langle u,e,1\rangle$, we will leave the component at $\langle u,e,6\rangle$. There are three ways to walk through this component. One of them is illustrated in Figure 4.

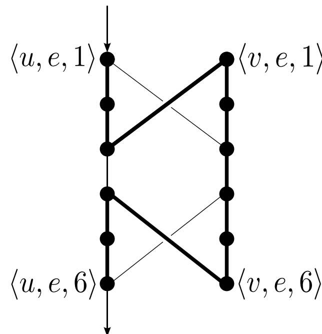

Figure 4: Possible route through cover testing component.

Walking through the component in this way means that only u belongs to the vertex cover. If only v belongs to the vertex cover we walk through the component in the opposite way. If both nodes are in the vertex cover, we enter the component at two points $\langle u,e,1\rangle$ and $\langle v,e,1\rangle$. Then there are two paths visiting the nodes, namely, via $\{\langle u,e,i\rangle \mid 1 \leq i \leq 6\}$ and via $\{\langle v,e,i\rangle \mid 1 \leq i \leq 6\}$.

The next step is to connect these cover testing components to each other and to selector vertices. Therefore, for any $v \in V_1$ we order the edges incident to v arbitrarily as $e_{v(1)}, e_{v(2)}, ..., e_{v(deg(v))}$, where $deg(v)$ is the degree of v in G_1. We will connect all cover testing components that belong to the same node v as follows:

$$E_2(v) = \big\{\{\langle v, e_{v(i)}, 6\rangle, \langle v, e_{v(i+1)}, 1\rangle\} \mid 1 \leq i < deg(v)\big\}.$$

Thus, for each v we get a chain of cover testing components that belong to edges incident to v.

The last thing we have to add to G_2 are edges that connect every one of the selector vertices to the first and last nodes of these chains. They are described as follows:

$$E_2' = \big\{\{a_i, \langle v, e_{v(1)}, 1\rangle\}, \{a_i, \langle v, e_{v(deg(v))}, 6\rangle\} \mid 1 \leq i \leq L, v \in V_1\big\}.$$

This completes the graph $G_2 = (V_2, E_2)$ and we have found a good transformation

from Vertex Cover to Hamiltonian Circuit.

$$
\begin{aligned}
V_2 &= \{a_i \mid 1 \leq i \leq L\} \cup \big(\bigcup_{e \in E_1} V_2(e)\big), \\
E_2 &= \big(\bigcup_{e \in E_1} E_2(e)\big) \cup \big(\bigcup_{v \in V_1} E_2(v)\big) \cup E_2'.
\end{aligned}
$$

This transformation can be done in polynomial time bounded by the size of G_1 and L. To see this, we represent G_1 as an $n \times n$ adjacency matrix A where $n = |V_1| = |V|$. The adjacency matrix B for G_2 gets a lot bigger, because for each edge in G_1 we have a cover testing component consisting of twelve nodes, plus, we have the L selector vertices. The number of edges in G_1 is of order $O(n^2)$, which implies that B is a $t \times t$ matrix where $t = L + 12 \cdot$ number of edges in G_1. This means that it takes $O(n^4)$ time to make the matrix B, which is polynomial, bounded by the size of G_1 and L.

We will now prove the following lemma.

Lemma 7. *G_1 has a vertex cover W with $|W| \leq L$ if and only if G_2 has a Hamiltonian circuit.*

Proof. If we prove that G_1 has a vertex cover W with $|W| = L$ if and only if G_2 has a Hamiltonian circuit we are finished, because we can always add nodes to the vertex cover for it to remain a vertex cover.

($\Rightarrow$) Let $W \subseteq V_1$ be a vertex cover in G_1 of size L. The elements in W are $w_1, ..., w_L$, which correspond to the selector vertices in G_2. We check for every edge $e = \{u, v\}$ in G_1 whether u, v or both are in W to obtain the right route through each cover testing component. Exactly one of the three cases for walking through a cover testing component holds, because W is a vertex cover. Second, we choose all edges in $E_2(v_i)$ where we have that $1 \leq i \leq L$. We end by choosing the edges $\{a_i, \langle v_i, e_{v_i(1)}, 1\rangle\}$, $\{a_{i+1}, \langle v_i, e_{v_i(deg(v_i))}, 6\rangle\}$ and $\{a_1, \langle v_L, e_{v_L(deg(v_L))}, 6\rangle\}$ where $1 \leq i \leq L$. This gives us indeed a Hamiltonian circuit. We go from a selector node to the first node of a chain of cover testing components. We then walk through the chain according to the path determined by which nodes are in the vertex cover. After finishing a chain, we go to a second selector node and repeat the same process again until we have come to our last chain. Then, there are no selector vertices left, which forces us to go to the node where we started. This give us a cycle in which we have visited every node, a Hamiltonian circuit.

($\Leftarrow$) We are given a graph G_2 that contains a Hamiltonian circuit. Let $(v_1 \to v_2 \to ... \to v_r)$ be a Hamiltonian circuit, where $r = |V_2|$ and $v_1, ..., v_r \in V_2$. We always start in one of the selector vertices. Then pass through the cover testing component of an edge in G_1 and, with that, all the cover testing components of edges incident to the same node. We walk through every cover testing component in G_2 in one of the three ways mentioned before, such that every node in the component is visited exactly once. We end in another selector node and the process starts again. This means that the L selector vertices divide the circuit into L paths, where the paths correspond to L distinct nodes in G_1. Every node in the Hamiltonian circuit must be visited exactly once, thus every node of every cover testing component is visited. Also, a cover testing component for $e = \{u, v\} \in E_1$ can only be traversed by a path which belongs to either u, v or both u and v. This implies that either u, v or both are among the L selector vertices. Since we had L paths, we have L nodes that form a vertex cover for G_1. □

We now have shown that we can construct a graph G_2 from G_1 in polynomial time and that G_1 has a vertex cover W with $|W| \leq L$ if and only if G_2 has a Hamiltonian circuit. □

The following lemma now holds.

Lemma 8. *Φ is satisfiable if and only if G_2 has a Hamiltonian circuit.*

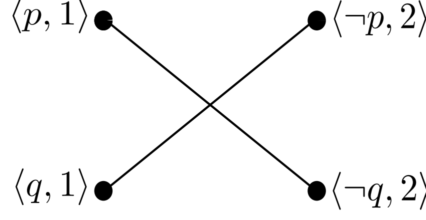

Figure 5: Graph G containing a clique of size 2.

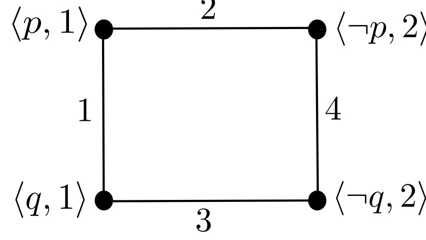

Figure 6: Graph G_1 containing a vertex cover of size 2.

We will give an example to illustrate this lemma.

Example 2. *Let $\Phi = (p \vee q) \wedge (\neg p \vee \neg q)$ be a Boolean formula of which we want to show that it is satisfiable if and only if the graph G_2, which we will construct according to the proof above, has a Hamiltonian circuit. We will prove Lemma 8 according to the reduction steps we showed above. First, we will reduce Φ to a graph $G = (V, E)$ to show that Φ is satisfiable if and only if G has a clique Q of size at least $K = 2$. G can be constructed such that we get*

$$\begin{aligned} V &= \{\langle p, 1\rangle, \langle q, 1\rangle, \langle \neg p, 2\rangle, \langle \neg q, 2\rangle\}, \\ E &= \{\{\langle p, 1\rangle, \langle \neg q, 2\rangle\}, \{\langle q, 1\rangle, \langle \neg p, 2\rangle\}\}. \end{aligned}$$

The graph is illustrated in Figure 5. We can immediately see that G indeed contains a clique of size two and therefore can we conclude that Φ is satisfiable.

Continuing with our reduction process gives us a second graph G_1 which can be constructed out of G. We will show that G has a clique Q of size $K = 2$ if and only if G_1 has a vertex cover W of size $L = |V| - m = 4 - 2 = 2$. We know that G_1 is the complement of G. The graph G_1 is illustrated in Figure 6. The edges are labeled from 1 up to 4, but we do not need these labels yet in this reduction step. The nodes opposite to the ones used for the clique are now forming a vertex cover which are indeed two nodes. We can easily check that all edges are covered by these two nodes. Thus G_1 has a vertex cover of size $L = 2$.

The last step in the reduction process is reducing Vertex Cover to Hamiltonian Circuit. We want to construct a graph G_2 such that G_2 contains a Hamiltonian circuit if and only if G_1 has a vertex cover of size two. Because our vertex cover contains two nodes, we add two selector vertices to G_2, s_1 and s_2. For every edge in G_1 we construct a cover testing component labeled from 1 to 4 according to the labeled edges in G_1. This is illustrated in Figure 7. We will now use the labels 1, 2, 3 and 4 which we added to the edges in the previous graph G_1. We put the nodes belonging to a node in G_1 together to form a chain in G_2. We then get the combinations $(1, 2)$, $(1, 3)$, $(2, 4)$ and $(3, 4)$ belonging to each of the four nodes in G_1. We see that by using only two of these combinations (because we have a vertex cover of size 2 and thus two selector vertices), we can visit each cover testing component and thus every node in G_2. This gives us a Hamiltonian circuit in G_2. We now have shown how to reduce SAT to HC in several steps which we have proved before.

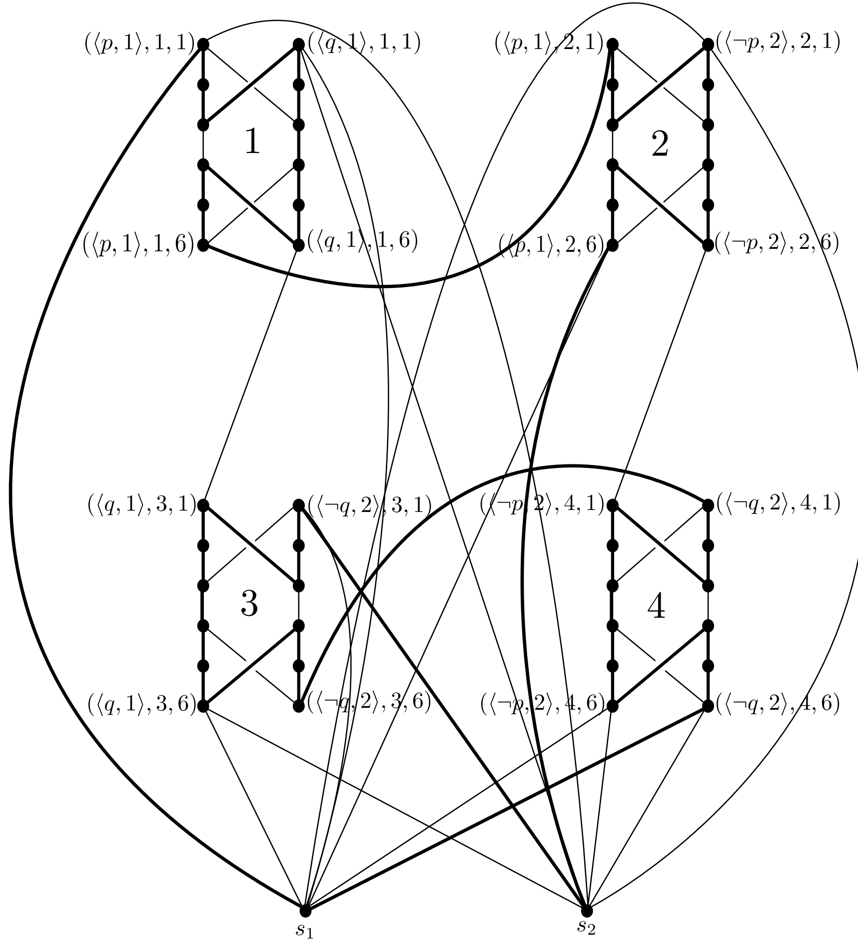

Figure 7: The bold lines form a possible Hamiltonian circuit in the graph G_2 starting in s_1.

6.4 HC $\preceq$ TSP

The last step in our reduction process is from Hamiltonian Circuit to, finally, the Traveling Salesman Problem. The proof is based on an argument for the directed version of TSP by Papadimitriou [1994, p. 198]. A detailed proof can be found below.

Theorem 6. *HC $\preceq$ TSP*

Proof. For this reduction we are given a graph $G_2 = (V_2, E_2)$. We want to construct a graph $G_3 = (V_3, E_3)$ in polynomial time by representing it as a matrix, such that G_2 has a Hamiltonian circuit if and only if G_3 has a Hamiltonian circuit which does not exceed a given amount of money, $M = |V_2| = n$. We start with constructing the graph G_3 and show that this can be done in polynomial time. Second, we prove the bi-implication stated above.

G_3 can easily be constructed from G_2. We just add all the edges which were not in G_2 to make the graph G_3 complete. For the Traveling Salesman Problem we need a weighted graph with costs on the edges. We want some edges to be more expensive than others and therefore we attach a high weight, $n + 1$, to all the edges we just added. The edges that were already in G_2 we give weight 1. This completes the graph G_3 and we have found a good transformation from Hamiltonian Circuit to the Traveling Salesman Problem.

This transformation can be done in polynomial time, bounded by the size of G_2. We can see this by representing the graphs as matrices. Represent G_2 as an $n \times n$ adjacency matrix A. To construct a new matrix B for G_3 from A we replace every 0 by $n+1$ and every 1 remains the same. This takes at most $O(n^2)$ time, which is indeed polynomial, bounded by the size of G_2.

The next step is to prove the following lemma.

Lemma 9. *G_2 has a Hamiltonian circuit if and only if G_3 has a Hamiltonian circuit which does not exceed a given amount of money, $M = |V_2| = n$.*

Proof. ($\Rightarrow$) Let $(v_1 \to v_2 \to ... \to v_n)$ be a Hamiltonian circuit in G_2, where $n = |V_2|$ and $v_1, ..., v_n \in V_2$. If we just walk through these nodes n in G_2 in this order, we only come across edges that have weight 1 in the graph G_3. Thus $(v_1 \to v_2 \to ... \to v_n)$ will also be a Hamiltonian circuit in G_3. This Hamiltonian circuit costs exactly $n = M$ money and not $n-1$, because we have $|V_2| = n$ nodes which form a circuit, and we have to go back to our starting point. Because we attached weight $n+1$ to the edges we added to G_3, we will not walk over these edges. So, we will not exceed our given amount of money $M = n$ which gives us a yes-instance for TSP.

($\Leftarrow$) Let $(v_1 \to v_2 \to ... \to v_n)$ be a Hamiltonian circuit in G_3, where $n = |V_3|$ and $v_1, ..., v_n \in V_3$ and let $M = |V_3| = |V_2| = n$ be a given amount of money which we are not allowed to exceed. We cannot walk over the edges labeled by $n+1$, because then we would exceed our given amount of money. Therefore we can only walk over the edges which are labeled 1. But those are exactly the edges of G_2, which immediately implies that we have a Hamiltonian circuit in G_2. □

We have shown that we can construct the graph G_3 from G_2 in polynomial time and that G_2 has a Hamiltonian circuit if and only if G_3 has a Hamiltonian circuit which does not exceed a given amount of money, $M = |V_2| = n$. □

This completes our reduction process from SAT to TSP. Unfortunately we were not able to find a direct reduction from SAT to TSP which was not the composition of reductions we used in between. A lot of questions came up while thinking about this direct reduction problem. For example, how do we show the difference between true and false literals and clauses in our graph? We cannot only visit the literals which are true, because this violates the requirement that all nodes must be visited. What to do with edges between a variable and its negation? It is intuitively clear that these edges should be very expensive. And if we make an edge between a variable and its negation expensive, how can we ensure that there is no other indirect low-priced route? These questions made us think that it was more difficult to find a direct reduction than we thought at first. As you can see, for example in Figure 7, the graph becomes really complicated for only a simple Boolean formula. This is mainly, because of two things. In the reduction from CLIQUE to Vertex Cover we take the complement and when we reduce Vertex Cover to Hamiltonian Circuit, we look at edges instead of nodes. These two things make it more difficult to see what the original Boolean formula was. Therefore we were not able to find a direct reduction from SAT to TSP (without using the composition of the reductions in between).

7 Remarks on *Changes of Mind* by Neil Tennant

In the previous sections of this thesis we have learned much about complexity theory. In this section we will evaluate and criticize some aspects of the book *Changes of Mind*, written by Neil Tennant. Neil Tennant is a professor of philosophy, at the Ohio State University. He is interested in different areas of philosophy, namely mathematical and computational logic, but also philosophy of mind and cognitive science. These topics are also found in the book *Changes of mind*. In short, the book is about how a rational agent changes his beliefs. Tennant describes a contraction problem, which is about belief contraction: the act of giving up a belief. According to Neil Tennant, the contraction problem is a decision problem, namely: is there a revised system of beliefs such that, when one gives up a belief p, we can "ensure that what is left will not give one p back again?" [Tennant, 2012, p. 96]. Tennant shows in his book that the contraction problem is NP-complete. We will look at some aspects of the book, which are mainly discussed in the first four chapters. We start by describing how Tennant's belief system works and we explain the contraction problem. Then we discuss whether the belief system is a realistic representation of the mind and we discuss whether we can say something about the complexity of a problem about the mind.

7.1 Tennant's belief system

We start by explaining the notions of a belief system as it is described by Tennant. A belief system of Tennant represents the mind of a *logical paragon*. He distinguishes a logical paragon from a *logical saint*. A logical saint is capable of deriving everything that follows logically from any starting set of beliefs instantly. This infinite logical saint never makes a mistake and is therefore unrealistic. Instead, Tennant uses a logical paragon for his theory. A logical paragon is an idealized rational agent with, in contrast to the logical saint, finite memory. He can only deduce finitely many propositions in a finite amount of time. This is more realistic, because we only have finite lives. A logical paragon never makes a mistake in constructing a logical proof out of premises and implications as soon as the premises and the conclusion are known. Nevertheless, it can happen that the logical paragon does not know for example that in the real world not all his premises are necessary for his conclusion to be true.

Thus, Neil Tennant uses the logical paragon, which we will call a (rational) agent from now on, for his belief system. A belief system can formally be thought of as a finite dependency network $S = (N, A, I)$. Informally, this is a directed graph containing finitely many nodes N, representing beliefs, and finitely many arrows A, representing relations between these beliefs, mediated by inference strokes I. Beliefs are thoughts and opinions about things in the world. In the belief system they are represented as atomic propositional sentences like 'I think that my room is clean' or 'I believe that this sweater looks good on you'. A rational agent can believe the first proposition, because he just vacuumed the floor and did the dishes. The second proposition is more an opinion, because somebody else might think that the sweater doesn't look good on that same person. In a dependency network propositional sentences are represented as nodes. Nodes are black if the agent believes them and white if the agent does not believe them or has not yet formed an opinion about them. The nodes and inference strokes in the belief system are connected by arrows. These arrows only represent the structure of the diagram, that is, the relation between the nodes and the inference strokes.

Tennant gives a formal definition of a dependency network, but before stating this definition we have to know what is meant by a *step*.

Definition 18 (Step)**.** *Let U be a non-empty finite set of nodes and x a node. A pair (U, x) is a step, if it holds that $x \in U$ if and only if $U = \{x\}$.*

A step is also denoted as $\{U\}|x$. We talk about U as its premise set and x as its conclusion. In a dependency network we have two types of steps, initial steps and

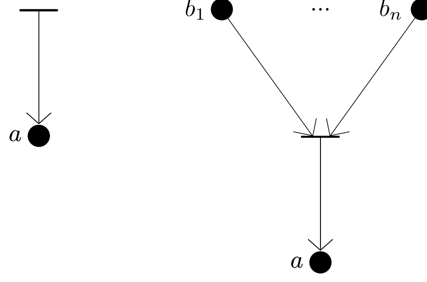

Figure 8: Left: initial step, right: transitional step

transitional steps, see Figure 8. An *initial step* is used when “a sentence or proposition a is believed without need, as far as the agent is concerned, for any justification” [Tennant, 2012, p. 39]. For an initial step we have an arrow coming from a starting inference stroke pointing at the node a. Tennant denotes this step as $\{a\}|a$, (see Figure 8, left). The belief a can thus be seen as self-justifying (from the agent’s point of view). A *transitional step* is a step from a set $\{b_1, ..., b_n\}$ to a node a. We can understand this step as follows: “*if* one is justified in believing each of $b_1, ..., b_n$, *then* one is justified in believing a” [Tennant, 2012, p. 40]. This step is denoted by $\{b_1, ..., b_n\}|a$.

We can now give a formal definition of a dependency network. This definition is based on the definition given by Tennant [2012, p. 117].

Definition 19 (Dependency network)**.** *Let S be a finite set of steps whose conclusions are not $\bot$. S is a dependency network if and only if*

1. $\forall(U, x) \in S\ \forall y \in U\ \exists(V, y) \in S$;
2. *if $(U, x) \in S$ and $(W, x) \in S$, and neither U nor W is equal to $\{x\}$, then W is not a proper subset of U.*

The first condition of Definition 19 says that every node in the dependency network is justified by some set. The second condition makes sure that we do not have the step (U, x) if we also have the step (W, x), where W is a proper subset of U. To make clear how we should represent nodes, inference strokes and arrows graphically, Tennant describes a set of ‘Axioms of Configuration’. “[A]xioms determine the possible arrangements of nodes and strokes, connected by arrows, into a dependency network” [Tennant, 2012, p. 46]. All axioms follow directly from Definition 19. If a system S violates one of the axioms, then S isn’t a well-defined dependency network. Inference strokes are only used in the graphical representation and have no further meaning. They just make the graph more clear. In the enumeration below we will have that Nx means that ‘x is a node’, Sx means that ‘x is an inference stroke’ and Axy means that ‘x points at y’. Here x and y are either nodes or inference strokes.

Axioms of configuration:

1. $\forall x(Nx \vee Sx)$
2. $\neg\exists x(Nx \wedge Sx)$
3. $\forall x(Sx \to \forall y(Axy \to Ny))$
4. $\forall x(Nx \to \forall y(Axy \to Sy))$
5. $\forall x(Sx \to \exists y\forall z((y = z) \leftrightarrow Axz))$
6. $\forall x\forall y(Axy \to \neg Ayx)$

7. $\forall x \forall y((Sx \wedge Sy) \rightarrow \forall z((Axz \wedge Ayz) \rightarrow (\forall w(Awx \rightarrow Awy) \rightarrow (x = y))))$

8. $\forall x(Nx \rightarrow \exists y Ayx)$

Let us now look at the dependency network in a more informal way. Let us have a finite set of nodes $N = \{b_1, ..., b_n\}$, representing all the beliefs in your belief system before having deduced anything. Let $M = \{b_1, ..., b_m\}$, $m < n$, be a subset of N that forms a set of beliefs which are used as premises for some conclusion a. The corresponding nodes can be white or black. From all the nodes in this subset there is an arrow pointing at an inference stroke which is also white or black. The inference stroke is black if all the nodes pointing at it are black and the inference stroke is white if at least one node corresponding to an element of the set M is white. From the inference stroke there is an arrow pointing at the conclusion a. The node corresponding to a is white if the inference stroke is white and black if the inference stroke is black. Thus, as soon as a rational agent does not believe one of his premises anymore, he is obliged to stop believing the conclusion. This means that believing the premises is necessary for believing the conclusion.

Note that it doesn't matter whether beliefs are true or not. We only talk about whether something is believed by a rational agent. Every rational agent has his own beliefs and draws his own conclusions from it. An agent can believe that $b_1, ..., b_k$, $k < n$, are necessary to believe a. But, in the real world it can be possible that b_1 is not necessary at all for a to be true. The rational agent is just not aware of the existence of this step. Here, read the words 'true' and 'believe' carefully. Everything goes well, as long as the agent does not ignore the fact that if he believes all premises of a set M and he is aware of the step from these premises to a, he must believe a as well. If he ignores this fact, he is not a rational agent.

According to Tennant, the actual system of beliefs are only the black nodes and the black inference strokes. They form a *subsystem* (or *subnetwork*) R in the total system S, that is, R is also a dependency network and has to satisfy all the axioms of configuration. This system R is a *closed* system, which means that R is equal to its *closure*.

Definition 20 (Closure). *The closure $\overline{R}$ of a subsystem $R \subseteq S$ is the smallest subsystem in S such that the following two properties are satisfied:*

1. *R is a subsystem of $\overline{R}$;*

2. *if for every step in S it holds that if the premise set $\{b_1, ..., b_m\}$ is in $\overline{R}$, then the step $\{b_1, ..., b_m\}|a$ is in $\overline{R}$.*

Tennant gives another set of axioms, called 'Axioms of Coloration'. These axioms describe how the nodes and inference strokes should be colored. They follow directly from Definition 20. If one of these axioms of coloration is violated we would have that R is not closed, that is, $R \neq \overline{R}$. In the following enumeration we have that Bx means 'x is black' and Wx means 'x is white'.

Axioms of coloration:

1. $\forall x((Bx \wedge Nx) \rightarrow \exists y(By \wedge Sy \wedge Ayx))$

2. $\forall x((Wx \wedge Nx) \rightarrow \forall y(Ayx \rightarrow (Wy \wedge Sy)))$

3. $\forall x((Bx \wedge Sx) \rightarrow \forall y(Ayx \rightarrow (By \wedge Ny)))$

4. $\forall x((Wx \wedge Sx \wedge \exists z Azx) \rightarrow \exists y(Wy \wedge Ny \wedge Ayx))$

A more detailed explanation of these axioms can be found in [Tennant, 2012, p. 47-50]. In Figure 9 violations of the four axioms of coloration are given. The numbers in the figure correspond to the numbers of the axioms of coloration. The first and

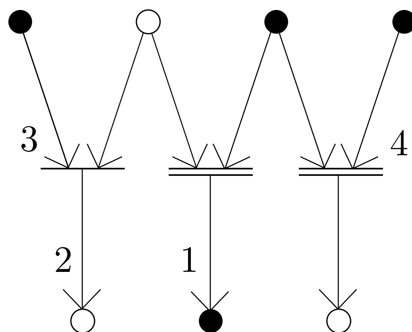

Figure 9: Violations of the four axioms of coloration.

second axiom are of the same type and are violated, because the conclusion node and the inference stroke pointing at it do not have the same color. The third axiom is violated, because an inference stroke can only be black if all premise nodes pointing at it are black. The fourth axiom is violated in a similar way as the third.
Thus, if we would have a violation of an axiom of configuration, then we cannot speak of a well-defined dependency network and if one of the axioms of coloration is violated we would have that R is not closed.

Finally, according to Tennant, the geometry of the system is static. This means that the arrows connecting nodes with inference strokes cannot change and there can not be new nodes, inference strokes or arrows added to the network. The only things that change are the colors of nodes and inference strokes. This is what the book *Changes of Mind* is about, changing your beliefs. This will be explained next.

7.2 The contraction problem

Having explained Tennant's belief system, we can show what he means with the contraction problem. It is a problem about changing your mind. *Contraction* means that an agent gives up one or more beliefs (make a black node white), whereas *expansion* means that the system is expanded with a new belief (color a white node black). Contraction and expansion together form belief revision, but the step of belief expansion is relatively simple. Therefore, Tennant talks about the problem of contraction.

Contracting a system with respect to a node a means that a rational agent removes a from his actual belief system, that is, the agent doesn't believe a anymore. But you cannot just remove a from the belief system, because then the system isn't closed anymore. The premise set $\{b_1, ..., b_m\}$ is still in $\overline{R}$ which implies that the step $\{b_1, ..., b_m\}|a$ must be in $\overline{R}$. But, since a is not in R anymore, a is not in $\overline{R}$ and therefore the step $\{b_1, ..., b_m\}|a$ is not in $\overline{R}$ anymore. This means that the agent is obliged to stop believing at least one of the premises $b_1, ..., b_m$, say for example b_1. Then the agent has to check what the consequences are of not believing b_1 anymore. He should check whether the remaining system is closed or not until he reaches an initial step. This is called the contraction problem. Tennant wants this problem to be "as economical as possible, in the sense that one gives up as little as possible" [Tennant, 2012, p. 130] when contracting the system from S to $S - a$. He calls this the *'minimal mutilation'* condition. The contraction problem can be stated as the following decision problem.

> Given a finite belief system S containing the belief a and an integer $K < |S|$. Does there exist a contraction of the system with respect to a of at least size K?

In other words, does there exist a subsystem $R \subseteq S$, not containing a, of size at least K? Tennant also gives a more formal definition of the problem.

> Given a finite belief system S containing the belief a and an integer $K < |S|$. Does there exist a closed subsystem R of at least size K that does not contain a?

According to Tennant, this decision problem is NP-complete. Indeed, there are difficulties in the contraction problem which cannot be deterministically computed in polynomial time. We can only find out by trial and error whether there is a contraction of at least size K that does not contain a. Just try all possible subsystems of at least size K to check whether these subsystems are closed and do not contain a. This will give us a problem of at least order $O(2^{|S|})$, because the system is of size $|S|$ and for each step in S we have two possibilities: either the agent believes the step or he doesn't. Thus, the number of possibilities is exponential. Hence, Tennant's contraction problem is not in P. But it is in NP. If we are given a system S and $R \subseteq S$ a subsystem of a least size K, then we can check in polynomial time whether R does not contain a and whether R is closed in S. Checking whether R does not contain a is easy, because we only have to walk through the system and check whether a is contained or not. This can be done in linear time, bounded by the size of R. According to Tennant we can check in polynomial time whether R is closed in S. Remember that R is a set of steps and we must check that $R = \overline{R}$. Thus, for every step in R we must check whether all premises are believed, and if so, then the conclusion has to be believed too. This means we have to run $c \cdot |R|$ times through the steps in R, where c is a positive constant. Thus, we can check in polynomial time of order $O(|R|^2)$ whether R is closed in S. Therefore, the contraction problem is in NP. In his book, Tennant proves the NP-completeness of the contraction problem by giving a reduction from Vertex Cover to it [Tennant, 2012, p. 133-134].

Having explained how Tennant's belief system and contraction problem work we will look at two aspects of the book. We discuss whether the belief system is a realistic representation of the mind. This is done according to the fact that the geometry of the system is static. We also discuss whether we can say something about the complexity of a problem about the mind.

7.3 How realistic is Tennant's belief system?

We ended 7.1 by saying that, according to Tennant, the geometry of the system was static. With this he meant that the only things that can change in the dependency network are the colors of the nodes and inference strokes. There is no way to add anything new to the network. If Tennant meant to represent human beings with the logical paragon, then the belief system is not very realistic. If there is no way to add new steps to the network, how is it possible to learn new things? And when in life will the geometry of the system be fixed? Is the geometry fixed as soon as we are born or can it be fixed at any moment in life? Or do we assume that the system contains every belief there is to know? All options sound strange given the fact that the system is static. Fixing a belief system at birth, suggests that the geometry of a system is the same for everyone, because the system does not contain any beliefs yet, the dependency network is empty. According to Tennant we cannot add things to the network, which leaves us with a system without beliefs. This is not a fine prospect for mankind.

Then there is the option of fixing the geometry of a belief system at any moment in life. Every individual agent has a different system built up from his own beliefs until that moment. Within the system, changing a belief can be done by making a black node white. But if we observe and learn new things, what will happen to the system? For example, you are shopping with your friend and she shows you a sweater which she likes very much. You see the sweater for the first time and you believe that it looks good on her. This is a new belief, a black node. According to Tennant we needed a white node first to transform it into a black node. But, in a static system there is no possibility to add a new node to the system, even if the belief that you liked the sweater was really in your mind. How do we learn new things? Instead of

fixing the geometry of the system at birth, fixing the geometry at the end of your life could be an option, so that the system is complete. But what is the point of changing your mind if you are about to die? Thus, the option of fixing the geometry of the belief system at any moment in life is not realistic.

The third option to learn new things is to accept the assumption that the system contains every belief there is to know. Every rational agent can color it according to his beliefs. But, it is impossible for humans to have a belief system containing every possible belief, because nobody has complete knowledge about the world. We know that Tennant describes a belief system as a set of steps which are all known to the rational agent, thus, even the white nodes in the dependency network have to be known to the agent. In this way an agent can check for example whether he believes p or $\neg p$. If it would be the case that the system contains every belief in the world, then the system is probably not finite anymore. If the system would become infinitely large, we could only work with it if we were logical saints. Tennant wants to avoid this by introducing the logical paragon. Therefore, the option that we could only learn new things if the belief system contains every belief, is not the option Tennant has in mind.

The most realistic option could be that the system is fixed every time a belief is changed. If an agent changes a belief p into $\neg p$, then the system is refreshed before changing of belief again. If the agent then makes up his mind once more, the system is fixed again and so on. This still doesn't explain how to learn new things. New things simply appear in the system and are related immediately to the rest of the system. This is something a logical paragon can, but humans can't. The idea of learning new things might be an NP-complete (or even harder) problem itself and Tennant separates these two problems. He doesn't mention that these problems are very closely related. The geometry of Tennant's belief system is indeed static, which is useful for showing that the contraction problem is NP-complete, but it is not realistic. You cannot separate belief change from learning new things. Humans often change their mind because they learn a new belief which, for example, contradicts an old belief. The system should be a more dynamic system, but then the problem becomes too hard to solve.

Next, we will discuss whether we can speak of the computational complexity of a problem about belief change.

7.4 Can you say something about the complexity of a problem about the mind?

We have seen that the contraction problem of Tennant is NP-complete. Does this mean that Tennant says that we are solving NP-complete problems in our mind? And why does Tennant want his problem to be NP-complete?

The contraction problem of Tennant is NP-complete. This means that a computer cannot solve the problem deterministically in polynomial time, but there is an algorithm that can guess a solution and can verify in polynomial time if this is indeed a solution. Tennant has placed a lot of restrictions that make the problem easier to solve, but also less realistic. For example, the fact that the geometry of his dependency network is static is not very realistic, but makes the problem look very similar to other decision problems with graphs (e.g. TSP, CLIQUE, VC). Therefore, Tennant can reduce Vertex Cover to the contraction problem. Another restriction is that every object and structure Tennant works with is finite. It can be the case that there are infinitely many beliefs, but according to Tennant a rational agent never believes more than finitely many of them. If the belief system would contain infinitely many steps there is no possibility that the contraction problem is NP-complete or even computable. Thirdly, Tennant has restricted the problem to just looking whether a belief is justified by the agent. He doesn't look at the logical structure within a belief. If he looked at the logical structure, the problem would become more complicated. Probably because of all the restrictions, the contraction problem has become

NP-complete.

Tennant thinks that it is useful that the contraction problem is NP-complete. If Tennant means to represent humans with the logical paragon, then in a way he says that we are solving NP-complete problems in our mind. He formally represents in a belief system what is happening in the mind of an agent (our mind) when changing a belief. By solving the contraction problem exactly Tennant proves that this is an NP-complete problem. Are we continuously solving NP-complete problems when changing our mind? No we aren't. But if something that happens in our mind is NP-complete for a computer, how can it be that we have no problems when we are changing our beliefs? Suppose you are given an arbitrary instance of size n of the Traveling Salesman Problem and you are asked to check whether there exists a tour with costs less than K. You cannot easily solve this decision problem and it is neither something we do all day long. If the instance is small, say $n = 4$, we can solve the problem. As soon as the instances become larger, we aren't capable of solving it exactly, but we can sometimes make a good guess of what the answer should be. Guessing is something humans are good at, in contrast to computers. Our guessing is based on rational thinking, whereas computers guess randomly, based on an algorithm together with a whole set of data.

The same idea works for the contraction problem where we change a belief (assuming that Tennant still wants to represent humans with his logical paragon). When we contract the belief system with respect to a belief p, we remove p from the system. The color of p changes from black to white. A logical paragon can check in polynomial time whether the remaining system is still a closed subnetwork of at least size K. But we, humans, do not check directly whether everything we believe still holds after a contraction. We don't check our memory step by step on every belief. We just stop believing p, because we observed the opposite of p or we stopped believing one of the premises necessary for believing p. There are many reasons why we could stop believing p and we don't always think about the consequences that a contraction might have. It could be that after ten years we still believe something for which it was necessary that we believed p. Tennant assumes with the contraction problem that a rational agent knows directly what he can and cannot believe after contracting with respect to p. If this assumption is correct, an agent had to check all of his memory every time there was a contraction and he would be busy with doing this for ever. There would be no time to think about other things, because his memory is constantly busy with computing. Changing our beliefs is something we are doing multiple times a day, which means that if we were represented by the logical paragon we would be busy with changing our beliefs for the rest of our life. To refer to Tennant's dependency network: humans would probably only check the nearby beliefs in the network which can be thought of as a small subnetwork. This smaller subnetwork can be seen as a small instance which we can solve. We are capable of solving small instances of NP-complete problems as long as we can visualize them. As soon as the instances become larger or the problem is generalized for arbitrary n, we can only guess an answer. We saw that humans are good at guessing in contrast to computers.

Thus, there is a possibility that we can solve NP-complete problems for small instances. But we don't seem to have problems with doing this. We sometimes even find a solution instead of only the yes/no-answer that a computer gives. A computer can only verify in polynomial time whether a given instance is a solution to the problem, but he cannot find a solution to a problem in polynomial time. For small instances we often produce an answer faster than a computer. This is because a computer needs an algorithm together with all data to 'understand' the problem and the mind only needs frames of the most relevant beliefs and fills in the details. We don't run through our whole memory to check whether there is a contradiction in our beliefs. For example, if you are changing your mind about whether it rains or not, there is no reason to look at the beliefs about whether you like the color of a sweater. We only search through the beliefs of which we intuitively think that they are relevant and a computer needs to search the whole set of beliefs. Maybe the mind

uses a faster algorithm than a computer or maybe we don't use an algorithm at all and are we just doing what seems best to do at that moment.

Another fact is that we are even solving problems with our mind which a computer can't solve. For example, checking the validity of an argument in first order logic. An argument is valid if its conclusion follows logically from is premises. First order logic differs from propositional logic, because of the use of the quantifiers '$\forall$' and '$\exists$'. The problem of checking whether a formula is valid can be undecidable for a computer (in fact it is semi-decidable, but we will not elaborate on this). We are able to solve this problem for small instances. This means that we are capable of a lot more than we can imagine. If we ever discover how our mind solves such problems, we might be able to solve them for arbitrary n. But for this to happen we have to wait a long time, maybe even infinitely long.

Tennant writes about the complexity of a problem about the mind. He thinks it is useful and important that his contraction problem is NP-complete. Tennant believes that we are one step further in the search if our mind can be seen as a computational object. By using a computer Tennant tries to show what, according to him, happens in the mind of a rational agent when this agent changes his beliefs. But, we saw before that Tennant's belief system was not very realistic, because not even a logical paragon has knowledge about all beliefs, and is therefore not capable of learning new things. This means that it is neither the mind of a logical paragon that is represented by a computer nor the mind of a human. Changing beliefs is much more complicated then Tennant says. We probably need more to represent the mind. Humans don't always behave like rational agents. Sometimes humans just ignore the fact that they believe contradictory things. Here we enter the discussion about whether humans have 'free will', but this lies outside the scope of this thesis. It is impossible to separate the part where we change our beliefs from the rest of our mind. To link the two themes 'mind' and 'complexity' to each other in the way Tennant does, is not helpful in searching whether our mind can be seen as a computational object.

8 Conclusion

In this thesis we focused on several decision problems and we proved polynomial reductions between them. The decision problems we looked at were SAT, 3-SAT, CLIQUE, Vertex Cover, Hamiltonian Circuit and TSP. These were introduced in Sections 3 and 4. Before proving these reductions we defined in Section 5 the complexity classes P, NP and NP-complete. We also needed a definition of a polynomial reduction and we proved in Lemma 1 the transitivity of a polynomial reduction. We ended Section 5 with showing that the decision problem 3-SAT is NP-complete by reducing SAT to it. In Section 6 we proved step by step that SAT can be reduced to TSP. We started this project with the goal to find out whether a direct polynomial transformation function from SAT to TSP could be found. It is possible in theory, because we know of both problems that they are NP-complete. Unfortunately we only found the composition of the reductions in between SAT and TSP. We have shown these reductions step by step based on an article written by Karp [1972, p. 97-98]. In his article Karp gave transformations of instances for different NP-complete problems, but he did not prove them. The polynomial reductions we proved were SAT $\preceq$ CLIQUE, CLIQUE $\preceq$ VC, VC $\preceq$ HC and HC $\preceq$ TSP. Having proved this, we could immediately see why we weren't able to find a direct reduction from SAT to TSP. The resulting graph becomes really complicated for only a simple Boolean formula. It is difficult to discover what the original Boolean formula was in the resulting graph. This is mainly because of two things. When reducing CLIQUE to Vertex Cover we took the complement and by reducing Vertex Cover to Hamiltonian Circuit, we looked at edges instead of nodes. These things lead to some confusion when trying to reduce SAT directly to TSP. It is a nice puzzle to try and find a direct reduction from SAT to TSP, but for this thesis it took too much time.

The second part of this thesis, Section 7, was about a more philosophical decision problem. We gave some remarks on a contraction problem proven to be NP-complete by Neil Tennant in *Changes of Mind*. The book is about changing your beliefs in a belief system. We described Tennant's belief system and contraction problem. Then we have discussed how realistic this belief system is and concluded that it is not very realistic that the geometry of his system is static. This meant that nothing could be added to the system, which immediately lead to the question how we learn new things. Tennant probably sees this as a different problem, but he doesn't mention that they are closely related.

We discussed whether we can say something about the complexity of a problem about the mind. We saw that the contraction problem of Tennant was NP-complete, but does this mean that he says that we are solving NP-complete problems in our mind? A logical paragon can check in polynomial time whether the remaining system is still a closed subnetwork of at least size K. But we, humans, do not check directly whether everything we believe still holds after a contraction. We don't check our memory step by step on every belief. Humans would probably only check the nearby beliefs in the network which can be thought of as a small subnetwork. This smaller subnetwork can be seen as a small instance which we can solve. We are capable of solving small instances of NP-complete problems as long as we can visualize them. As soon as the instances become larger or the problem is generalized for arbitrary n, we can only guess an answer. Therefore, to link the two themes 'mind' and 'complexity' to each other in the way Tennant does, is not helpful in searching whether our mind can be seen as a computational object.

References

M. H. Alsuwaiyel. *Algorithms: Design techniques and analysis*, volume 7 of *Lecture notes series on computing*. World Scientific Publishers, 1999.

G. Boole. *An investigation of the laws of thought: On which are founded the mathematical theories of logic and probabilities*. Walton and Maberly, London, 1854.

S. Dasgupta, C. H. Papadimitriou, and U. V. Vazirani. *Algorithms*. McGraw-Hill, 2006.

R. Gamboa and J. Cowles. A mechanical proof of the Cook-Levin theorem. In K. Slind, A. Bunker, and G. Gopalakrishnan, editors, *Theorem proving in higher order logics*, pages 99–116. Springer, 2004.

Jiahong Guo and Lei Zhang. Book review of Changes of mind: An essay on rational belief revision by Neil Tennant. *Notre Dame philosophical reviews*, 2013.

D. R. Hofstadter. *Gödel, Escher, Bach: An eternal golden braid*. Basic Books, United States of America, 1979.

D. S. Johnson and M. R. Garey. *Computers and intractability*. W. H. Freeman and Company, New York, 1979.

R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of computer computations*, pages 85–103. Springer, New York, 1972.

C. H. Papadimitriou. *Computational complexity: A guide to the theory of NP-completeness*. Addison-Wesley Publishing Company, Massachussetts, 1994.

W. Pohlers and T. Glaß. Lecture notes of the course an introduction to mathematical logic. Wilhelms-Universität Münster, 1992.

G. Schäfer. Lecture notes of the master course discrete optimization, Utrecht University, 2012.

N. Tennant. Theory-contraction is NP-complete. *Logic Journal of the IGPL*, 11: 675–693, 2003.

N. Tennant. *Changes of mind: An essay on rational belief revision*. Oxford University Press, Oxford, 2012.

J. P. Warners. *Nonlinear approaches to satisfiability problems*. PhD thesis, Eindhoven University of Technology, 1999.