# Reconfiguration problems

MEMO-F403 – Preparatory work for the master thesis

Prateeba Ruggoo[1]
Department of Computer Science
Universite Libre de Bruxelles
pruggoo@ulb.ac.be[1]

Advisor: Jean Cardinal

May 27, 2019

*ABSTRACT*

Reconfiguration problems are generally defined as follows, given two feasible solutions of a problem is there a way to transform one feasible solution to another such that all intermediate solutions obtained are also feasible? In this survey a state of art about older and more recent work on this type of problem is given. The emphasis will be on the computational complexity of the different types of problems seen.

## 1 Introduction

The goal of this work is to study the different classifications established among different types of reconfiguration problems. This study is to help find a general rule or pattern to explain the complexity of reconfiguration problems. In order to do so, the framework for proving the PSPACE completeness of reconfiguration problems is studied in section 5. The different categories of reconfiguration problems considered are the following :

1. Satisfiability reconfiguration problems.
2. Binary integer programming reconfiguration problems.

The properties and relevant complexity results concerning each category specified above is given in section 6, section 7 respectively. Open questions about this subject are proposed at the end of the article.

# 2 Preliminaries

## 2.1 Mathematical Notions and terminologies

### 2.1.1 Functions and relations

**Definition 2.1.1** *A function $f$ is a process that associates to each element of a set $X$ at most one element of a set $Y$. The set $X$ is called the domain of $f$ and the set $Y$ is its range.* $f : \mathcal{D} \to \mathcal{R}$

### 2.1.2 Graphs

[Die00]

**Definition 2.1.2** *A graph is a mathematical structure used to model pairwise relations between objects. More formally, a graph is a pair $G = (V, E)$ where $V$ is the set of vertices or nodes of the graph $G$ and $E$ its set of edges. An edge $x, y$ will be written as $xy$.*

**Definition 2.1.3** *A graph $G'$ is a subgraph of $G$ if $V' \subseteq V$ and $E' \subseteq E$. $G$ is then the supergraph of $G'$.*

**Definition 2.1.4** *If $G'$ is the subgraph of $G$ and $G'$ contains all the edges $xy \in E$ with $x, y \in V'$ then $G'$ is said to be the induced subgraph of $G$.*

**Definition 2.1.5** *A path $P$ in a graph is a sequence of edges which connects a sequence of distinct vertices. More formally, a path is a non-empty graph $P = (V, E)$ where $V = \{x_0, x_1, \ldots, x_k\}$ and $E = \{x_0x_1, x_1x_2, \ldots, x_{k-1}x_k\}$ and all the $x_i$ are distinct.*

# 3 Theory of Computation

## 3.1 Basic notions

In theoretical computer science, the theory of computation studies how efficiently problems can be solved on a model of computation, using an algorithm. It is divided in three main branches : Automata Theory, Computability Theory and Computational Complexity Theory. This research work will mostly focus on the third branch.

**Definition 3.1.1** *An algorithm is an unambiguous procedure of how to solve a class of problems.*

**Definition 3.1.2** *A decision problem is a problem that can be posed as a YES/NO question. Given a decision problem $A$ and an input $n$, it verifies whether or not $n$ satisfies a certain property. Another convenient way of defining a decision problem is to give the set $L \subseteq 0, 1^*$ of inputs for which the answer is YES. $L$ is also called the language of the problem.*

## 3.2 Model of computation

As any other field, computer science is no stranger to using models in order to conceptualize certain concepts.

### 3.2.1 Turing Machine

Turing Machine is the computation model used in theoretical computer science to represent a general purpose computer. It uses an unlimited tape as its unlimited memory and has a tape head that can write and read symbols and move along the tape. More formally :

**Definition 3.2.1** *A Turing machine is a* $7-tuple$, $(\mathcal{Q}, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ *where* $\mathcal{Q}, \Sigma, \Gamma$ *are all finite sets and*

1. $\mathcal{Q}$ *is the set of states,*
2. $\Sigma$ *is the input alphabet not containing the blank symbol* $\sqcup$,
3. $\Gamma$ *is the tape alphabet, where* $\sqcup \in \Gamma$ *and* $\Sigma \subseteq \Gamma$,
4. $\delta : \mathcal{Q} \times \Gamma \to \mathcal{Q} \times \Gamma \times \{L, R\}$ *is the transition function,*
5. $q_0 \in \mathcal{Q}$ *is the start state,*
6. $q_{accept} \in \mathcal{Q}$ *is the accept state, and*
7. $q_{reject} \in \mathcal{Q}$ *is the reject state, where* $q_{reject} \neq q_{accept}$

**3.2.1.1 Semantics :** *The transition function* $\delta$ *explains how the Turing Machine operates, i.e how it goes from one state to another.*
*If* $\delta\{q, a\} = (r, b, L)$, *and the machine is in state* $q$ *with its head on the tape cell reading the symbol* $a$, *it will replace the* $a$ *with a* $b$, *transition to state* $r$ *and move the head left.*

### 3.2.2 Non deterministic Turing Machine

The non deterministic Turing Machine is defined the same way as a deterministic one except for the transition function. A non deterministic machine can at any computation step, proceed with various possibilities. Its transition is defined as follows : $\delta : \mathcal{Q} \times \Gamma \to \mathcal{P}\{\mathcal{Q} \times \Gamma \times \{L, R\}\}$ where $\mathcal{P}$ is the power set of $\mathcal{Q}$.

## 3.3 Computability Theory

Computability theory focuses on whether a problem is solvable. Solvable problems are called decidable problems or tractable problems.

**Definition 3.3.1** *Decidable problems are problems that can be solved by a conventional Turing Machine in a number of steps which is proportional to a polynomial function of the size of its input. The class of problems with this property is known as $\mathcal{P}$ or polynomial time .*

In contrast, unsolvable problems are called undecidable problems or untractable problems.

## 3.4 Computational Complexity Theory

Computational complexity theory contemplates not solely the solvability of a problem but also the resources required to solve computational problems. It is divided in two branches: Time complexity and Space complexity.

One important tool used to prove the complexity of a problem is the reduction.

**Definition 3.4.1** *[Sip] A reduction is the art of converting one problem into another such that a solution to the second problem can be used to solve the first problem.*
*More formally : Given two languages $A$ and $B$, $A$ is reducible to $B$ written $A \leq_m B$ if there is a computable function $f : \Sigma^* \to \Sigma^*$, where for every $w, w \in A \iff f(w) \in B$. The function $f$ is called the reduction of $A$ to $B$.*

**Definition 3.4.2** *[Sip] A function $f : \Sigma^* \to \Sigma^*$ is a computable function if on every input $w$, some Turing machine $M$ halts with just $f(w)$ on its tape.*

### 3.4.1 Time complexity

Time complexity focuses on providing a way to measure the time used to solve a problem and to classify each problem according to its time complexity. To classify each problem, time complexity theory attempts to establish lower bounds on how efficient an algorithm can be for a given problem.

**Definition 3.4.3** *Let $M$ be a Turing machine that halt on all inputs. The time complexity of $M$ is the function $f : \mathcal{N} \to \mathcal{N}$, where $f(n)$ is the maximum number of steps $M$ uses on any input of length $n$.*

**Definition 3.4.4** ***Polynomial time :*** *$\mathcal{P}$ is the class of languages that are decidable by a deterministic single tape turing machine.*

**Definition 3.4.5** ***Non deterministic polynomial time :*** *$\mathcal{NP}$ is the class of languages that are decidable by a Non deterministic polynomial time Turing machine.*

**3.4.1.1** $\mathcal{P}$ **vs** $\mathcal{NP}$ Since the introduction to these complexity classes, a natural question arises. Is $\mathcal{P} = \mathcal{NP}$ or $\mathcal{P} \neq \mathcal{N}P$.

**Definition 3.4.6** *$\mathcal{NP}$-**completeness** is the class of languages to which any language $A$ of $\mathcal{NP}$ can be reduced to. More formally a problem $A$ is NP-complete if :*

1. *$A$ is in $\mathcal{NP}$*
2. *Any problem $B$ in $\mathcal{NP}$ is reducible to $A$ in polynomial time.*

### 3.4.2 Space complexity

**Definition 3.4.7** *Let $M$ be a Turing machine that halts on all inputs. The space complexity of $M$ is the function $f : \mathcal{N} \to \mathcal{N}$, where $f(n)$ is the maximum number of tape cells $M$ scans on any input of length $n$.*

Analogous to the complexity classes defined in the earlier section, the different classes of space complexity will be defined here.

**Definition 3.4.8** *PSPACE is the class of languages that are decidable in polynomial space on a deterministic Turing Machine.*

**Definition 3.4.9** *NPSPACE is the class of languages that are decidable in polynomial space on a non deterministic Turing Machine.*

**Theorem 3.4.10** *(Savitch's theorem) For any function $f \in \Omega(\log(n))$, NSPACE($f(n)$) $\subseteq$ PSPACE($(f(n))^2$)*

**Corollary 3.4.11** *PSPACE = NPSPACE*

**Definition 3.4.12** *A language $A$ is PSPACE-complete if it satisfies two conditions :*

1. *$A$ is in PSPACE, and*
2. *every $B$ in PSPACE is polynomial time reducible to $A$.*

# 4 Reconfiguration problems

## 4.1 Problem definition

**Definition 4.1.1** *The general form of reconfiguration problems considered here is the following : Given two combinatorial configurations satisfying a problem while satisfying some constraints, is it possible to transform one configuration to another by modifying only one element at a time and that the intermediate solution remains satisfiable at all times. Combinatorial reconfiguration problems ask the reachability between the two given satisfying solutions.*

## 4.2 Theory Model

### 4.2.1 Configuration Graph

**4.2.1.1** In computation complexity theory, a Turing Machine was used to capture the idea of an algorithm. For configuration problems, a powerful tool is the Configuration Graph. The latter is used to represent the solution space of the reconfiguration problem and to introduce an adjacency relation on the set of satisfying solutions.

**Definition 4.2.1** *Let $G = (V, E)$ be the configuration graph where $V = \{$collection of all configurations$\}$ (ie all possible solutions) and an edge $xy \in E$ if one configuration can be transformed to another by changing only one variable. (i.e in a single reconfiguration step).*

**Definition 4.2.2** *A reconfiguration sequence is a path between two solutions in the configuration graph. It can be seen as a sequence of reconfiguration steps transforming one solution into the other.*

**Definition 4.2.3** *Using the language of configuration graphs, two connectivity questions arises:*

1. *Given a configuration graph $G = (V, E)$ and two vertices $s, t \in V$, is there a path from $s$ to $t$ in $G$ such that each intermediate solution remains satisfying?*

2. *Given the set of all satisfying configurations, is the configuration graph connected?*

*The first connectivity problem will be further referred to as the st-connectivity problem. The second problem will be referred to as the connectivity problem.*

## 4.3 Tools for proving the complexity of reachability problems of reconfiguration problems

In general the reconfiguration problem of an $\mathcal{NP}-complete$ problem is PSPACE-complete. And the reconfiguration problem of a polynomial-time solvable problem is PSPACE. However there are exceptions to this general rule :

1. The $3-$coloring problem is $\mathcal{NP}-hard$ and its corresponding reconfiguration problem is solvable in polynomial time.

2. The Shortest path problem is solvable in polynomial time whereas it's corresponding reconfiguration problem is PSPACE-complete.

Many of the reductions used to prove the PSPACE-hardness of various reconfiguration problems mimics the $\mathcal{NP}-hardness$ reductions used to prove that the host problem is $\mathcal{NP}-hard$.

Just as there was a need for the first $\mathcal{NP}-$complete problem to prove the $\mathcal{NP}-$completeness of other problems in $\mathcal{NP}$, a first PSPACE complete reconfiguration problem was needed to prove the PSPACE completeness of other reconfiguration problems.

The basis problem used to prove the PSPACE-completeness of the reconfiguration problems is the Non deterministic Constraint Logic Machine reconfiguration problem $NCL$.

# 5 The Non deterministic Constraint Logic Model of Computation

The NCL machine can be viewed as a framework for proving PSPACE hardness. It can be seen as a model of computation that captures the class PSPACE. The machine notion will be formulated using a graph notation as described below.

## 5.1 Graph formulation

The NCL machine can be interpreted as an undirected graph $G = (V, E)$ where to each edge is assigned a weight $w : E \to \mathbb{Z}^+$ and to each vertex is assigned an integer to represent its minimum inflow.

**Definition 5.1.1** *A configuration using the graph formulation given above is an orientation of its edges such that it satisfies the following constraint :*

1. *The sum of all in-degree edges at each vertex must be at least the minimum inflow of the vertex. This constraint is called the minimum inflow constraint.*

**Definition 5.1.2** *A reconfiguration step from one configuration to the other is simply the reversal of an edge at a time such that the minimum inflow constraint remains satisfiable.*

**Definition 5.1.3** *A constraint graph is a directed graph satisfying the minimum inflow constraint.*

**Definition 5.1.4** *A constraint graph is in normal form if it is a cubic graph and all edges weights* $\in \{1, 2\}$. *The weights may also be represented graphically by drawing edges of weight one as red and edges of weight two as blue. The minimum inflow constraint for each vertex is set to* 2. *Additionally each vertex should be incident to an even number of red edges.*

*This type of constraint graph is also called and/or constraint graph and will be referred to as such in the following sections.*

## 5.2 Circuit formulation

And/or constraint graphs are called as such because the two possible type of vertices behave like an AND gate and an OR gate in Boolean logic.


Figure 1: left image: AND vertex right image : OR vertex

A vertex with two red edges and one blue edge behaves like an AND gate because it requires both red edges to point inwards before the blue edge can be made to point outwards.

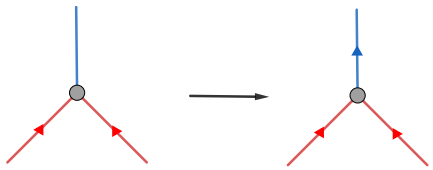

Figure 2: Activation of the output edge when the two input edges are activated

A vertex with three blue edges behaves like an OR gate, since it requires at least one input edge to point inwards before the output edge can be activated.

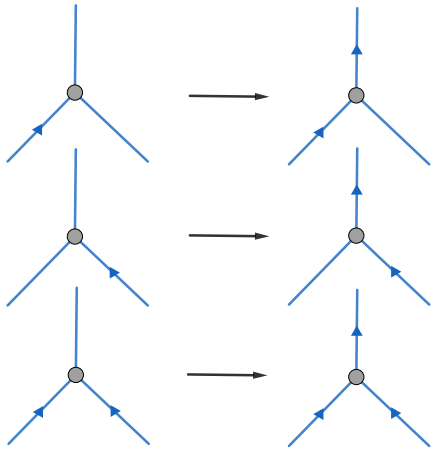

Figure 3: Different behaviours of an OR vertex

**Definition 5.2.1** *Activation of an edge :*

1. *An input edge is active when it is an incoming edge.*
2. *An output edge is active when it is an outgoing edge.*

**Definition 5.2.2** *Two decision problems arise regarding the graph formulation of the NCL machine.*

1. *Given the and/or constraint graph, can a specified edge be eventually reversed by a sequence of reconfiguration steps ?*
2. *Given two satisfying configurations, is one configuration reachable to the other by a single edge reversal at a time?*

*The answer to both questions happens to be PSPACE-complete. In this work a proof sketch is given for the second question.*

## 5.3 PSPACE-completeness

### 5.3.1 NCL is PSPACE-hard

**Proof.** The PSPACE-hardness of NCL is proved by doing a reduction from QBF problem, a well known PSPACE-complete problem. The goal of this reduction is to translate a given Quantified Boolean Formula $\phi$ into an instance of NCL so that the result gate in the resulting circuit may be activated if and only if $\phi$ is true.

**Definition 5.3.1** *QBF is the generalized satisfiability problem that includes universal ($\forall$) and existential ($\exists$) quantifiers. A QBF formula is a Boolean formula containing quantifiers.*

#### 5.3.1.1 QBF $\leq_p$ NCL

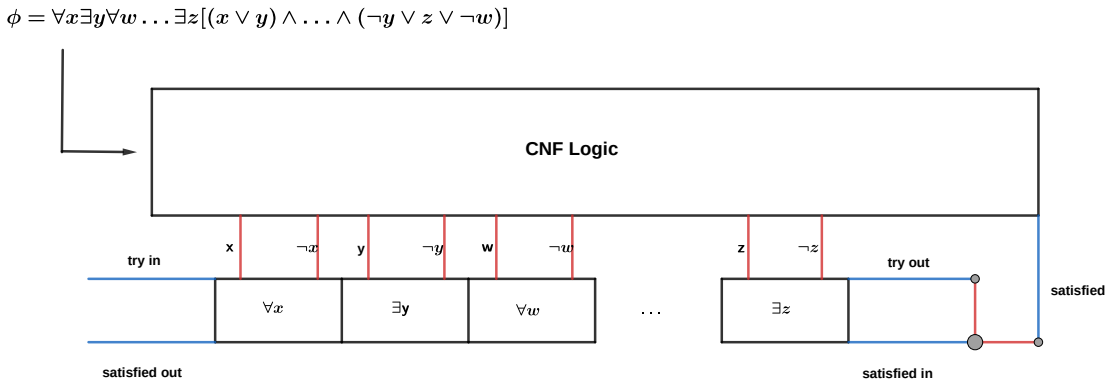


Figure 4: Schematic of the reduction from Quantified Boolean Formulas to NCL

The reduction from QBF to NCL to prove the PSPACE-hardness is given in the following steps :

1. Compute the and/or constraint graph of the unquantified formula ($[(x \vee y) \wedge \cdots \wedge (\neg y \vee z \vee \neg w)]$) which will referred to as the CNF netork. This problem is $\mathcal{NP}-complete$[HD02].
2. Associate a quantifier gadget to each quantifier variable in the formula($\forall x \exists y \forall w \ldots \exists z$). Each quantifier gadget is connected together.
3. Associate a variable gadget to represent each variable in the formula.
4. Connect each quantifier gadget to its corresponding variable gadget. The orientation of each edge designates which variable is assigned to true or false.
5. The variable gadgets are then fed into the CNF network.
6. The output of the CNF network is connected to the rightmost quantifier gadget through an AND vertex.
7. The output of the overall circuit comes through the satisfied out port from the leftmost quantifier gadget.

#### 5.3.1.2 The gadgets in more detail

1. Existential quantifier gadget.

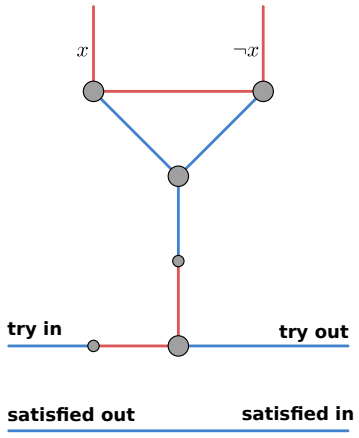


Figure 5: Existential gadget

(a) The try in edge is considered as the input edge.

(b) The try out edge is considered as the output edge.

(c) The top part of the existential gadget is called a latch.
The latch is in locked state when its designated output edge is activated. It is called locked state because when the output edge is activated no other edge in the latch can change orientation.

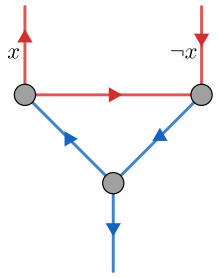


Figure 6: Latch

**Example 5.3.2** *If the designated output port is the edge* (1) *and it points downwards, it prevents other edges from changing orientation. For instance edge* (2) *cannot flip, which then prevents edges* (3) *and* (4) *from reversing. Thus edge* (5) *cannot point inwards either. Only edge* (6) *can reverse.*

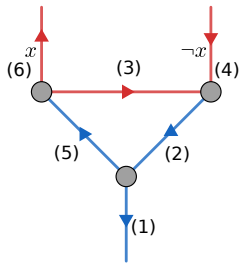


Figure 7: Locked state

(d) The latch is then connected to the bottom part of the existential gadget through an edge converter which converts edge (1) from a red edge to a blue edge. (the details about the edge converter gadget is omitted here. Refer to [HD02] for more details).

In summary the existential gadget works as follows : It receives a signal from the previous quantifier gadget to assign a value to its variable. Once the variable value is locked, edge (6) in figure 7 points out, activating the red edge.Thus allowing

the try out edge to activate and signal the next quantifier gadget to do the same process. If the later reports back with a positive answer, the current gadget returns the answer back to its caller.

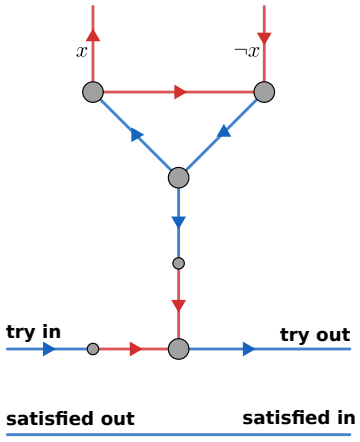


Figure 8: Universal quantifier gadget signaled to assign a value to its variable and to signal the next quantifier to do the same

2. Universal quantifier gadget.
   The universal quantifier has a slightly more complex structure than the existential quantifier. There are two latches in the universal gadget. Both whose edges are connected to an AND gate. As earlier the latches play the role of a one bit memory since their locking mechanism allows the assignment of the different variables to true or false. The satisfied out edge is activated only when both latches are in locked in state i.e when the assignments are successful.

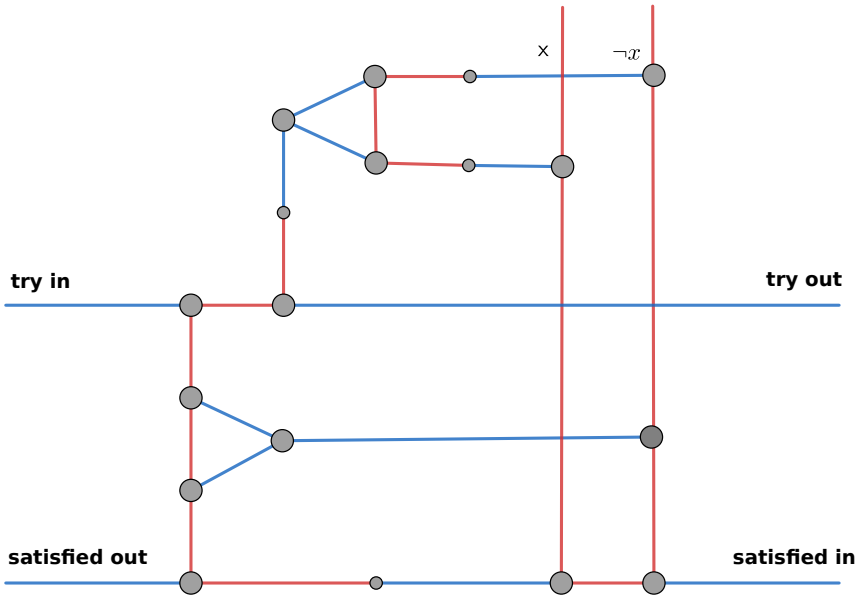


Figure 9: Universal quantifier gadget

3. The variable gadget
   The variable gadget is none other than the latches encountered in the universal and existential gadgets.

#### 5.3.1.3 The general workflow

Once the CNF network is computed, the result comes through the satisfied edge. The satisfied edge is activated if and only if the CNF formula is satisfied. Once all the variables are set, the try out edge is activated as well which then allows edges (1) and (2) to activate leading to the activation of the satisfied in edge.

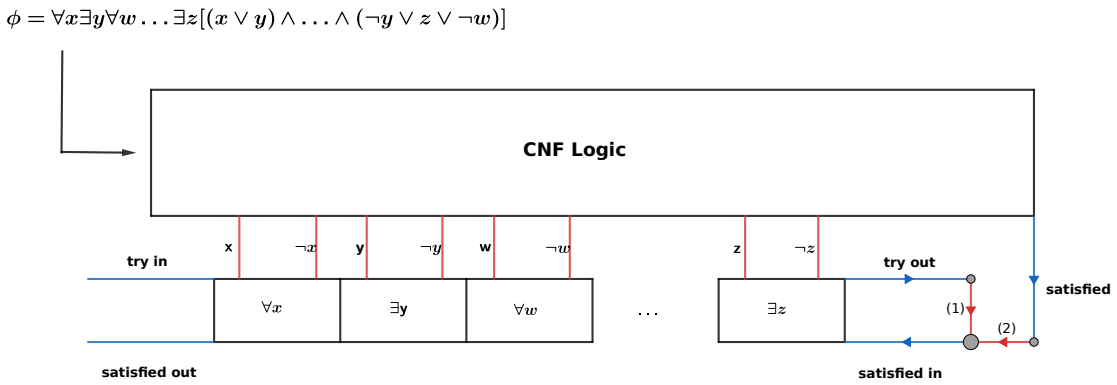


Figure 10: Schematic of the reduction from Quantified Boolean Formulas to NCL

This, then activates the quantifier gadgets leading to the activation of the satisfied out edge if and only if $\phi$ is satisfied.

**Lemma 5.3.3** *[HD02] A quantifier gadget's satisfied in port may not activate unless its try out port is active.*

**Lemma 5.3.4** *[HD02] An existential quantifier gadget may activate its satisfied out port if and only if its satisfied in port is active with its variable locked in some state.*

**Lemma 5.3.5** *[HD02] A universal quantifier gadget may activate its satisfied out port if and only if its satisfied in port is at one time active with its variable locked in the false (x) state, and at a later time is again active with its variable locked in the true (x) state, with try in remaining active throughout.*

**Lemma 5.3.6** *[HD02] A quantifier gadget may activate its satisfied out port if and only if its try in port is active, and the formula read from the corresponding quantifier to the right is true given the variable assignments that are fixed by the quantifier gadgets to the left.*

**Theorem 5.3.7** *NCL is PSPACE-complete [HD02]*

# 6 Reconfiguration of satisfiability problems

The focal point of this section is the connectivity-related properties of the solution space of Boolean satisfiability problems with a focus on the complexity of the connectivity and st-connectivity questions defined earlier in section 4.

## 6.1 SAT

**Definition 6.1.1** *The satisfiability problem is to test whether a Boolean formula is satisfiable.* $SAT = \{\langle\phi\rangle | \phi$ *is a satisfiable Boolean formula*$\}$. *i.e to decide if an assignment of* $0_s$ *and* $1_s$ *to its variables, makes the formula evaluate to true.*

**Definition 6.1.2** *A CNF formula is a Boolean formula of the form* $C_1 \wedge C_2 \wedge, \ldots, \wedge C_m$ *where each* $C_i$ *is a clause. A clause is a disjunction of multiple literals. A literal may be a variable or a negation of a variable.*

**Theorem 6.1.3 (Cook-Levin)** *SAT is* $\mathcal{NP}$*-complete.*

## 6.2 Satisfiability reconfiguration problems

**Definition 6.2.1** *Given a boolean formula* $\phi$ *with* $n$ *boolean variables* $\{x_i, x_{i+1}, \ldots, x_n\}$ *with* $i = \{0, \ldots, n\}$ *and two configurations* $s_0$, $s_t$ *from* $\{T, F\}^n$ *that satisfy* $\phi$, *is there a way to transform one configuration to the other with the following constraints:*

1. *At each step, only one variable* $x_i$ *can be flipped.*
2. *Each intermediate configuration* $x_k$ *must be feasible i.e satisfy* $\phi$.

### 6.2.1 Configuration graph

**Definition 6.2.2** *The graph* $G = (V, E)$ *considered here is an* $n$ *dimensional hypercube where* $V = \{$*the collection of all possible assignments*$\}$ *and* $E = \{$*collection of edges that links all pairs of vertices that differ exactly in one variable*$\}$.

**Example 6.2.3** $\phi = (x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_2 \vee \neg x_3 \vee \neg x_1)$ *and the two satisfying assignments are :* $s_0 = (true, false, true)$ *and* $s_t = (false, true, true)$ *i.e* $s_0 = (1, 0, 1)$ *and* $s_t = (0, 1, 1)$.

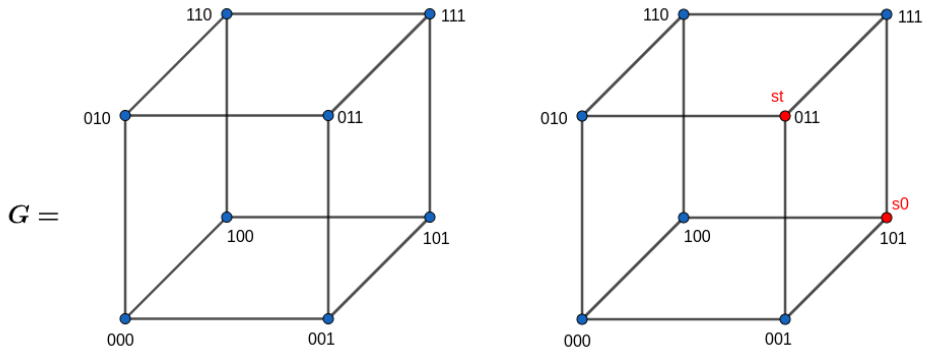


Figure 11: left image:3-dimensional hypercube containing all possible assignments. Right image : two assignments satisfying $\phi$.

### 6.2.2 Basic concepts and statements of results

The first to analyze the connectivity properties of the configuration graphs of satisfiability problems were Gopalan et al. [GKMP06]. Their work is the continuity of Schaefer's work [Sch78]. The latter introduced a framework for expressing variants of Boolean satisfiability and a very well celebrated theorem called the Dichotomy theorem.

#### 6.2.2.1 Schaefer's framework

**Definition 6.2.4** *A logical relation $\mathcal{R}$ is a non-empty subset of $\{T,F\}^k$ where $k$ is the arity of $\mathcal{R}$ for some $k \geq 1$.*

***Example 6.2.5*** *If $\mathcal{R}_{1/3} = \{TFF, FTF, FFT\}$, then $\mathcal{R}_{1/3}(x_1,x_2,x_3)$ is TRUE if and only if exactly one of $x_1$,$x_2$,$x_3$ is assigned to TRUE.*

**Definition 6.2.6** *Let $\mathcal{S}$ be a finite set of logical relations. A $CNF(\mathcal{S}) - formula$ over a set of variables $V = \{x_1, x_2, \ldots, x_n\}$ is a finite conjunction $c_1 \wedge c_2 \wedge \cdots \wedge c_m$ of clauses built using relations from $\mathcal{S}$.*
*Hence each $c_i$ is an expression of the form $\mathcal{R}(\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_k)$ where $\mathcal{R}$ is a logical relation and each $\varepsilon_j \in Vor\{T,F\}$. A solution of a $CNF(\mathcal{S})-formula$ $\phi$ is an assignment $s = (s_1, \ldots, a_n)$ of Boolean values to the variables that makes every clause of $\phi$ true.*

**Definition 6.2.7** *The $SAT(\mathcal{S})$ problem associated with a finite set of logical relations $\mathcal{S}$ asks : Given a CNF($\mathcal{S}$)-formula $\phi$, is it satisfiable ?*

**Remark 6.2.8** *All well known restrictions of Boolean satisfiability problems, such as 3-SAT, NOT-ALL-EQUAL 3-SAT and POSITIVE 1-IN-3SAT can be cast as SAT($\mathcal{S}$) problems, for a suitable choice of $\mathcal{S}$.*

**Example 6.2.9** *Let* $R_0 = \{0,1\}^3 \backslash \{000\}$, $R_1 = \{0,1\}^3 \backslash \{100\}$, $R_2 = \{0,1\}^3 \backslash \{110\}$, $R_3 = \{0,1\}^3 \backslash \{111\}$. *Then* $3-SAT$ *is the* $SAT(\mathcal{S})$ *problem where* $\mathcal{S} = (R_0, R_1, R_2, R_3)$. *Similarly, POSITIVE 1- IN -3S AT is* $SAT(R_{1/3})$, *where* $R_{1/3} = \{100, 010, 001\}$.

**6.2.2.2 Dichotomy theorem** Schaefer classified the Boolean constraint satisfaction problem and showed that, for certain sets $\mathcal{S}$, $SAT(\mathcal{S})$ is solvable in polynomial time while for all other sets $\mathcal{S}$ the problem is $\mathcal{NP} - complete$.
Schaefer gave 6 classes for which $SAT(\mathcal{S})$ is solvable in polynomial and proved that all other sets of relations generate an $\mathcal{NP}$-complete problem.
A set $\mathcal{S}$ of logical relations is Schaefer if all relations in $\mathcal{S}$ are either bijunctive, Horn, dual Horn, or affine. [Sch78]

**6.2.2.3 Tight relations class[GKMP06]** The class of tight relations can be seen as a superset of Schaefer relations. Gopalan et al. were mostly interested about the reachability problem and thus created a dichotomy theorem analogous to Schaefer's.

**Definition 6.2.10** *st-Conn($\mathcal{S}$) : Given a CNF($\mathcal{S}$)-formula $\phi$, and two satisfying assignments $s$ and $t$ of $\phi$,is there a path between $s$ and $t$ in the configuration graph of solutions of $\phi$?*

**Definition 6.2.11** *Conn($\mathcal{S}$)Given a CNF($\mathcal{S}$)-formula $\phi$, is the configuration graph of solutions of $\phi$ connected?*

**Theorem 6.2.12** *[vdH13] Let $\mathcal{S}$ be a finite set of logical relations.*

1. *If $\mathcal{S}$ is tight, then st-Conn($\mathcal{S}$) is in $\mathcal{P}$;*
   *otherwise, st-Conn($\mathcal{S}$) is PSPACE-complete.*

2. *If $\mathcal{S}$ is tight, then Conn($\mathcal{S}$) is in coNP,*
   *if it is tight but not Schaefer, then it is coNP-complete;*
   *otherwise, it is PSPACE-complete.*

3. *If every relation R in S is the set of solutions of a 2-CNF-formula, then Conn(S) is in P.*

Please refer to [GKMP06] for the formal definition of tight sets

# 7 Binary Integer programming Reconfiguration problems

## 7.1 Integer programming

**Definition 7.1.1** *An integer programming problem is defined as a mathematical optimization problem where the objective function and the constraints must be integers. More formally it is defined as follows :*

$$\underset{x}{\text{maximize}} \quad cx : Ax \leq b, x \in \mathbb{Z}^n \tag{1a}$$

*where $A$ is an $m$ integer matrix, $c$ an $n-$dimensional row vector, $b$ an $m-$dimensional column vector and $x$ an $n-$dimensional column vector of variables or unknowns.*

**Definition 7.1.2** *A binary integer programming problem also known as $0-1$ integer programming problem is an integer programming problem where $x \in \{0,1\}^n$.*

**Definition 7.1.3** *The set of solution to the BIP is the set of vectors $V$ that satisfy all the constraints. The set $V$ is also called the set of feasible solutions.*

### 7.1.1 Geometry of integer programming

**Definition 7.1.4** *Each constraint in an integer programming problem can be transformed into an equation which then represents a hyperplane in $\mathbb{R}^n$, a plane in $\mathbb{R}^3$ and a line in $\mathbb{R}^2$ and a point in $\mathbb{R}^1$ .*

**Definition 7.1.5** *The inequation represents one of two half spaces, which is the solution set of the inequations. A halfspace in $\mathbb{R}^n$ is a set of the form $\{x \in \mathbb{R}^n : a^T x \leq b\}$ for some vector $a \in \mathbb{R}^n$ and $b \in R$.*

**Definition 7.1.6** *A polyhedron is the intersection of finitely many halfspaces: $P = \{x \in \mathbb{R}^n : Ax \leq b\}$ with edges as lines, planes, hyperplanes as the case may be.*

**Definition 7.1.7** *A polytope is a bounded polyhedron.*

**Example 7.1.8**

$$\text{maximize} \quad Z = 8x_1 - 7x_2 \tag{2a}$$

*subject to :*

$$x_1 + x_2 \leq 5 \tag{3}$$
$$4x_1 + 7x_2 \leq 28 \tag{4}$$
$$2x_1 - 3x_2 \leq 6 \tag{5}$$
$$-3x_1 + 4x_2 \leq 12 \tag{6}$$
$$x_1 \geq 0, x_2 \geq 0 \tag{7}$$

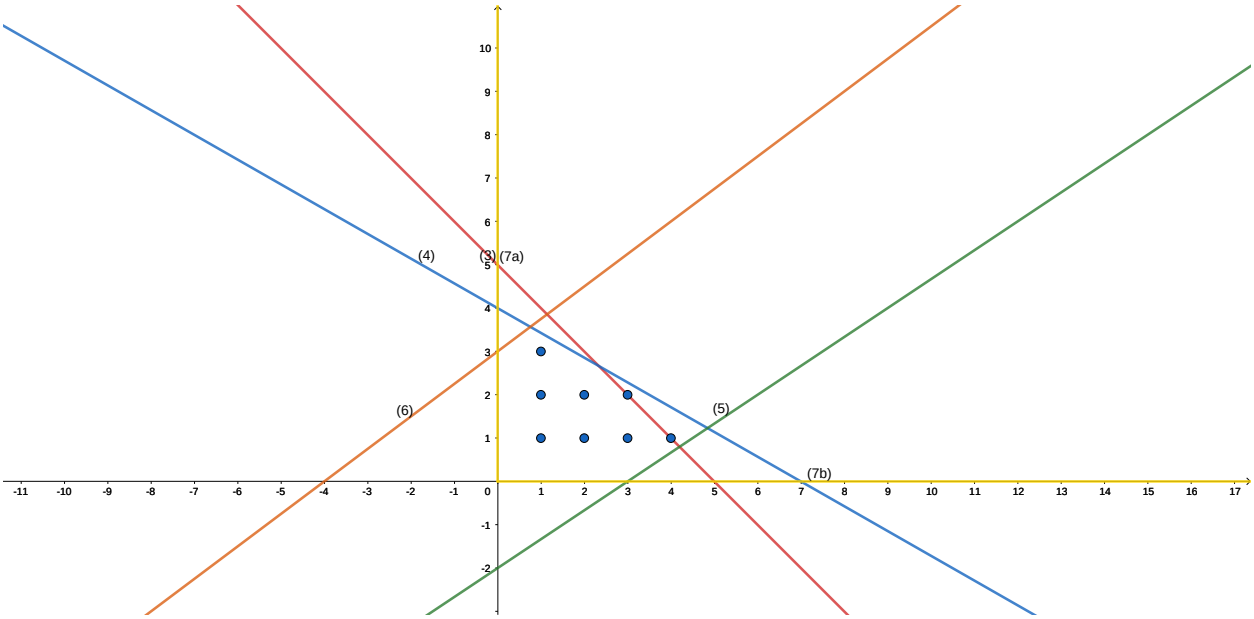


Figure 12: Polytope yielded by the constraints of the optimization problem where the solution points correspond to the lattice points.

### 7.1.2 Complexity

**Theorem 7.1.9** *The* $0-1$ *integer programming problem is* $\mathcal{NP}-$*complete.*

**Proof.** To prove that $0-1$ integer programming problem is $\mathcal{NP}-$complete. The two following conditions have to be satisfied :

1. $0-1$ integer programming problem is in $\mathcal{NP}$.
2. Any problem $B$ in $\mathcal{NP} \leq_p 0-1$ integer programming problem.
   The second condition can be proved by the reduction from a known $\mathcal{NP}-$complete problem.

#### 7.1.2.1 The $0-1$ integer programming problem is in $\mathcal{NP}$ .

To prove that $0-1$ integer programming is in $\mathcal{NP}$ a straightforward polynomial time algorithm deciding it, is the following :

$M =$"On input $(A, b, x)$
1. multiply matrix $A$ with vector $x$
2. for $i \in (1, \ldots, m)$
3. if $(Ax)_i > b_i$ reject
4. accept

1. Multiplying each component of a row of $A$ with each component of $x$ and doing this for $m$ rows of $A$ takes $m \cdot O(n) = O(m \cdot n)$ time.

2. Comparing component wise each element of $(Ax)$ with each element of $b$ makes in total $m$ comparisons is $O(m)$.

Final time complexity is $O(m \cdot n) + O(m) = O(m \cdot n)$ which means that $0-1$ integer programming $\in \mathcal{NP}$.

**Definition 7.1.10** *A cnf-formula is a 3cnf-formula if all its clauses has 3 litterals. For example* $\phi = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3)$. $3SAT = \{\langle\phi\rangle | \phi$ *is a satisfiable Boolean formula*$\}$

**Theorem 7.1.11** $3SAT$ *is* $\mathcal{NP}$*-complete.*

#### 7.1.2.2 3SAT $\leq_p$ $0-1$ integer programming

Let $\phi$ be a $3SAT$ formula with $m$ clauses and $n$ variables. The binary integer programming instance will be constructed in the following way :

For each clause in the $3SAT$ instance, we create the constraint that the sum of literals, using $z_i$ to represent $x_i$ and $(1 - z_i)$ to represent $\neg x_i$, is at least 1. For example if $\phi = (x_1 \vee \neg x_2 \vee x_3)$, the BIP instance will be constructed as following : $z_1 + (1 - z_2) + z_3 \geq 1$.

To satisfy this inequality either $z_1$ should be set to 1 or $z_2 = 0$ or $z_3 = 1$, which means we either set $x_1 = true$ or $x_2 = false$ or $x_3 = true$ in the corresponding truth assignment.

We can easily check that this construction works.

1. Assuming that we have a satisfying assignment $S$ for $\phi$. After the construction, since $S$ is a satisfying assignment, at least one literal in each clause must be satisfied. Therefore the associated sum corresponding to constraints in the BIP instance is $\geq 1$.

2. In the other direction, we assume we have a feasible solution to the BIP. Hence it means that each inequality must be satisfied i.e be $\geq 1$. This implies that at least one of the corresponding literal is set to 1, since each inequality is the sum of three variables set to 1 or 0.

## 7.2 Constrained Hypercube Path

The constrained hypercube path can be seen as a reconfiguration analogue to the $0-1$ Integer programming problem defined above. It is defined as follows :

**Definition 7.2.1** *[CDE+18] Given two vertices $s, t$ of the $n-$hypercube both be contained in a polytope $P := \{x \in \mathbb{R}^n : Ax \leq b\}$ for $A \in \mathbb{Z}^{d \times n}$ and $b \in \mathbb{Z}^d$, is there a path from $s$ to $t$ in the hypercube such that each vertices of the intermediate solution lies in $\mathcal{P}$*

**Example 7.2.2**

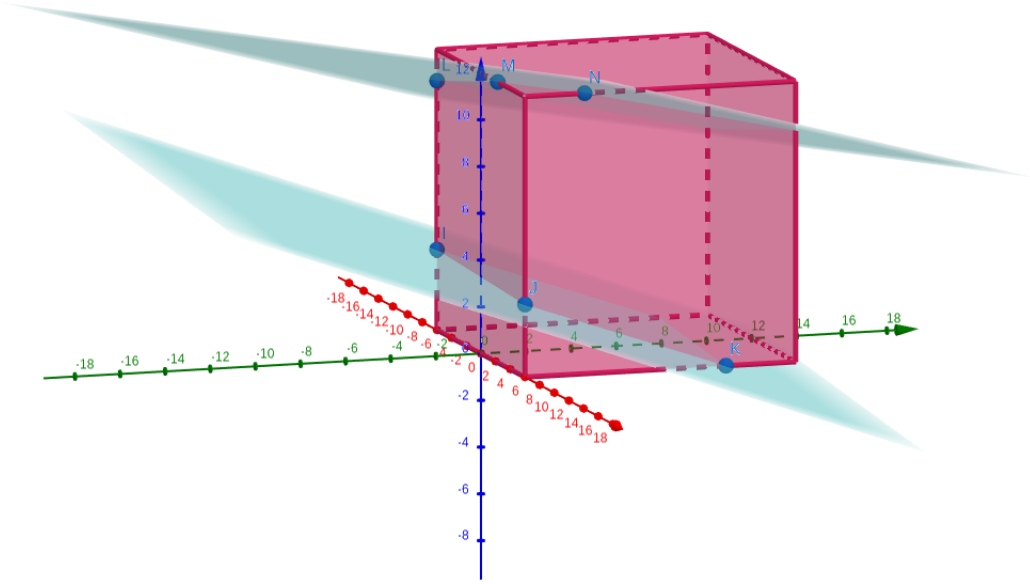


Figure 13: 3-hypercube solution space where the constraints are planes

**Theorem 7.2.3** *The constrained hypercube path problem is PSAPCE-complete, even when* $d = O(1)$. *[CDE+ 18]*

**Proof.** The above theorem is proved by a reduction from a variant of the exact cover reconfiguration problem. It is the exact cover reconfiguration problem where more-than-2-way merges and splits are permitted.

**Definition 7.2.4** *Exact cover problem : Given a collection* $S$ *of subsets of set* $X$*, an exact cover is the subset* $S^*$ *of* $S$ *such that each element of* $X$ *is contained is exactly one subset of* $S^*$*. It should satisfy the following two conditions :*

1. *The intersection of any two subsets in* $S^*$ *should be empty. That is, each element of* $X$ *should be contained in at most one subset of* $S^*$*.*
2. *Union of all subsets in* $S^*$ *is* $X$*. That means union should contain all the elements in set* $X$*. So we can say that* $S^*$ *covers* $X$*.*

*The Exact cover problem is a decision problem to determine if exact cover exists or not.*

**Example 7.2.5** *Let* $S = \{A, B, C, D, E, F\}$ *and* $X = \{1, 2, 3, 4, 5, 6, 7\}$ *such that :*
$A = \{1, 4, 7\}$
$B = \{1, 4\}$
$C = \{4, 5, 7\}$
$D = \{3, 5, 6\}$
$E = \{2, 3, 6, 7\}$
$F = \{2, 7\}$
*Then* $S^* = \{B, D, F\}$ *is an exact cover, because each element in* $X$ *is contained exactly once in subsets* $\{B, D, F\}$ *. If we union subsets then we will get all the elements of* $X$ *âĂŞ* $B \cup D \cup F = \{1, 2, 3, 4, 5, 6, 7\}$

**Definition 7.2.6** *Exact cover reconfiguration problem : Given a collection* $S$ *of subsets of a set* $X$ *and two exact covers* $C_1$ *and* $C_2$*, can* $C_1$ *be reconfigured into* $C_2$ *via repeated splits and merges?*

**Definition 7.2.7** *We could also transform an instance of the exact cover problem as a hypergraph* $G = (X, S)$ *where* $X$ *is a set and* $S$ *is a collection of subsets of* $X$*. The vertices of* $G$ *would be each element of* $X$ *and each element of* $S$ *is a hyperedge. We say that a hypergraph is* $k$*-colorable whenever we can assign one of* $k$ *colors to each vertex such that no two vertices in a hyperedge have the same color.*

**Lemma 7.2.8** *The exact cover reconfiguration problem is PSPACE-hard for instances that are* 23*-colorable hypergraphs. [CDE+ 18]*

# 8 Open questions

1. What is the connection between the complexity of reconfiguration problems and the complexity of the decision problem on the existence of configurations of a particular kind ?

2. What properties of problems result in the pattern holding or not? Related to the exceptions discussed in section 4

3. It is possible to refine Gopalan et al.'s dichotomy even more ?

4. (Related to the constrained hypercube path problems) Is the traveling salesman reconfiguration problem (where two tours are adjacent if they differ in two edges) PSPACE-complete?

# References


[ABI+] Eric Allender, Michael Bauland, Neil Immerman, Henning Schnoor, and Heribert Vollmer. The Complexity of SatisïňĄability Problems: ReïňĄning SchaeferâĂŹs Theorem. page 12.

[CDE+18] Jean Cardinal, Erik D. Demaine, David Eppstein, Robert A. Hearn, and Andrew Winslow. Reconfiguration of Satisfying Assignments and Subset Sums: Easy to Find, Hard to Connect. *arXiv:1805.04055 [cs]*, May 2018. arXiv: 1805.04055.

[Die00] Reinhard Diestel. *Graph theory*. Number 173 in Graduate texts in mathematics. Springer, New York, 2nd ed edition, 2000.

[GKMP06] Parikshit Gopalan, Phokion G. Kolaitis, Elitza Maneva, and Christos H. Papadimitriou. The Connectivity of Boolean Satisfiability: Computational and Structural Dichotomies. *arXiv:cs/0609072*, September 2006. arXiv: cs/0609072.

[HD02] Robert A. Hearn and Erik D. Demaine. The Nondeterministic Constraint Logic Model of Computation: Reductions and Applications. In Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, Peter Widmayer, Stephan Eidenbenz, Francisco Triguero, Rafael Morales, Ricardo Conejo, and Matthew Hennessy, editors, *Automata, Languages and Programming*, volume 2380, pages 401–413. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

[IDH+11] Takehiro Ito, Erik D. Demaine, Nicholas J.A. Harvey, Christos H. Papadimitriou, Martha Sideri, Ryuhei Uehara, and Yushi Uno. On the complexity of reconfiguration problems. *Theoretical Computer Science*, 412(12-14):1054–1065, March 2011.

[Ito] Takehiro Ito. Invitation to Combinatorial Reconfiguration. page 47.

[Kar] Richard M. Karp. Reducibility among Combinatorial Problems. In Raymond E. Miller, James W. Thatcher, and Jean D. Bohlinger, editors, *Complexity of Computer Computations: Proceedings of a Symposium on the Complexity of Computer Computations, Held March 20–22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, and Sponsored by the Office of Naval Research, Mathematics Program, IBM World Trade Corporation, and the IBM Research Mathematical Sciences Department*, pages 85–103. Springer US.

[MSOS] Vasco M Manquinho, JoÃčo P Marques Silva, Arlindo L Oliveira, and Karem A Sakallah. SatisïňĄability-Based Algorithms for 0-1 Integer Programming. page 10.

[Nis17] Naomi Nishimura. Introduction to Reconfiguration. page 24, 2017.

[Sch78] Thomas J. Schaefer. The complexity of satisfiability problems. In *Proceedings of the tenth annual ACM symposium on Theory of computing - STOC '78*, pages 216–226, San Diego, California, United States, 1978. ACM Press.

[Sip] Michael Sipser. *Introduction to the Theory of Computation*. Course Technology, second edition.

[vdH13] Jan van den Heuvel. The complexity of change. *CoRR*, abs/1312.2816, 2013.