Université libre de Bruxelles

Master Thesis

Reconfiguration Problems

Author:
Prateeba Ruggoo

Supervisor:
Dr. Jean Cardinal

Department of Computer Science

September 6, 2020

"*Understand well as I may, my comprehension can only be an infinitesimal fraction of all I want to understand.*"

Ada Lovelace

UNIVERSITÉ LIBRE DE BRUXELLES

Abstract

Department of Computer Science

Reconfiguration Problems

by Prateeba RUGGOO

Reconfiguration problems arise when we wish to find a step-by-step transformation between two feasible solutions of a problem such that all intermediate results are also feasible [20]. Recently, study about the solution space of reconfiguration problems defined as *reconfiguration graph* sparked great interest. In this work, we focus on the structural questions (Is the reconfiguration graph connected?, Is there a path between a node s and a node t in the reconfiguration graph?). In the first half this thesis we analyse various aspects of the Constraint Logic framework, from the book "Games, Puzzles and Computation" by Hearn and Demaine which provides several problems that are often a convenient starting point for reductions in order to prove PSPACE-hardness and present an in-depth study of the alternative formulation of NCL, the SLIDING TOKENS problem. In the second half of this thesis, we analyse the complexity results around the Boolean Satisfiability reconfiguration problems and focus on complementing some recent PSPACE-hardness proofs given in [7] concerning the k-move SUBSET SUM RECONFIGURATION problem.

Acknowledgements

First and foremost, my utmost gratitude to Prof. Jean Cardinal, whose patience and encouragement I will never forget. And whose door was always open whenever I had questions concerning my research or whenever I was lost, I could always count on him for guidance. He consistently allowed me to be creative but steered me in the right direction whenever he thought I needed it.

I am also deeply indebted to my parents who have given me the opportunity of an education and support throughout my life. Finally, I must express my very profound gratitude to my partner for providing me with continuous encouragement and a sounding board when required. This accomplishment would not have been possible without them. Thank you.

Contents

Every challenging work needs self efforts as well as the support of those who are very close to our heart. My humble effort I dedicate to my sweet and loving

Family & Partner

Whose love, affection and encouragement made me able to complete this challenge,

Along with all hard working and respected
Teachers

Chapter 1

Introduction

Reconfiguration problems are computational problems in which we wish to find a step-by-step transformation between two feasible solutions of a problem such that all intermediate results are also feasible. A solution can be the arrangement of puzzle pieces, the ordering of symbols to form a string or the location of a robot with respect to obstacles in space. Computational reconfiguration problems ask the reachability between the two given satisfying solutions. The area of reconfiguration considers both structural and algorithmic problems on the space of solutions, under various definitions of feasibility and adjacency.

The *reconfiguration framework* is defined in terms of a source problem, an instance of the source problem, a definition of a feasible solution and a definition of adjacency of feasible solutions. Viewing reconfiguration problems from a graph-theoretic perspective, the notion of a *reconfiguration graph* naturally arises. Let G be a reconfiguration graph where the vertex set consists of all possible configurations and two nodes are connected by an edge if the corresponding configurations can each be obtained from the other by the application of a single transformation rule, *a reconfiguration step*. Any path or walk in the reconfiguration graph corresponds to a sequence of reconfiguration steps called a *reconfiguration sequence*.

Interest in computational reconfiguration begun with the Sliding blocks puzzles and steadily increased during the last decade. The reconfiguration framework has recently been applied in a number of settings, including vertex colouring [2, 3, 8], list-edge colouring [21], clique, set cover, integer programming, matching, spanning tree, matroid bases [20], block puzzles [15], shortest path [23], independent set [15, 20, 24], and satisfiability [14]. Many problems in P have their reconfiguration problems in P as well, such as spanning tree, matching, and matroid problems in general. On the other hand, the reconfigurability of independent set, set cover, and integer programming are PSPACE-complete [20]. In general however, knowing the complexity of a decision problem does not allow us to directly infer the complexity status of its reconfiguration problem(s). Several NP-complete problems have their reconfiguration analogues that are in P, for example the 3-colourability problem [22]. Alternatively, some problems in P have their reconfiguration versions that are PSPACE-complete, such as shortest paths [5] or the problem of deciding whether two 4-colourings of a given bipartite or planar graph are reconfigurable [2].

This thesis does not attempt to catalogue all research results that can be categorized as reconfiguration, but instead focuses on demonstrating the main themes in the area and complement some recent PSPACE−hardness proofs given in [7]. More precisely, we go over and detail the PSPACE−completeness of the following decision problems :

- Given two subsets of a set S of integers with the same sum, can one subset be transformed into the other by adding or removing at most 3 elements of S at a time, such that the intermediate subsets also have the same sum ?
- Given two independent sets I_b and I_r of a graph $G = (V, E)$ and suppose that a labelled token is placed on each vertex in I_b where each label is unique, is it possible to transform I_b into I_r such that all intermediate solutions are also independent sets ?

In the first half of this thesis, we study the Nondeterministic Constraint Logic of Computation introduced by Hearn and Demaine. The NCL framework consists of different graph games (*i.e.* problems) specific for every major complexity class, more specifically the class PSPACE. These graph games are created in order to facilitate the reduction to other games. Reviewing NCL as part of this thesis is fundamental since it has been and still is the trigger for many PSPACE− hardness results.

The second part of this thesis focuses on Satisfiability reconfiguration problems and the Subset sum reconfiguration problem and provides a geometric interpretation for the latter problem with the intent on helping to have a better understanding of the reconfiguration graph and its connectivity properties. Finally, in chapter 6 our conclusions are presented and some proposals for future work are made.

Chapter 2

Preliminaries

This chapter serves as a general introduction to some mathematical concepts that are of interest to us.

2.1 Graph theory

2.1.1 Basic Notation

Let G be a simple, undirected graph with vertex set $V(G)$ and edge set $E(G)$. The *neighbourhood* of a vertex v is denoted by $N_G(v) = \{u|uv \in E(G)\}$. The *degree* of a vertex $v \in G$, denoted by $deg_G(v)$, is $|N_G(v)|$. Then $\Delta(G) = max_{v\in V(G)}deg_G(v)$ and $\delta(G) = min_{v\in V(G)}deg_G(v)$ is the maximum and minimum degree of G, respectively. A subgraph of G is a graph $G^{'}$ such that $V(G^{'}) \subseteq V(G)$ and $E(G^{'}) \subseteq E(G)$ (see Fig. 5.5).

A *walk* of length l from v_0 to v_l in G is a vertex sequence $v_0, \dots, v_l$, such that for all $i \in \{0, \dots, l-1\}$, $v_iv_{i+1} \in E(G)$. It is a path if all vertices are distinct. A path from a vertex u to a vertex v is also called a *$uv-$path.* A graph G is *connected* if there is a path between every pair of vertices. The *k-th power* of a graph $G = (V, E)$ is the graph G^k whose vertex set is V and two distinct vertices u, v are adjacent in G^k if and only if the shortest path distance between u and v in G is at most k. A *hypergraph* H is a pair $H = (X, E)$ where X is a set of elements, called nodes or vertices, and E is a set of non-empty subsets of X called *hyperedges* or *links.*

An *independent set* of a graph G is a vextex-subset of G in which no two vertices are adjacent. Given a set of elements $\{1, 2, \dots, n\}$ (called the universe, denoted $\mathcal{U}$) and a collection $\mathcal{S}$ of m sets whose union equals the universe, an *exact cover* is a sub-collection $\mathcal{S}^*$ of $\mathcal{S}$ such that each element in $\mathcal{U}$ is contained in exactly one subset in $\mathcal{S}^*$. The set $\mathcal{S}$ of subsets of U can be considered as a hypergraph $H = (U, \mathcal{S})$, where each element of U is a vertex and each element of $\mathcal{S}$ is a hyperedge.

For an in-depth review of general graph theoretic definitions, the reader can refer to Diestel's textbook [11].

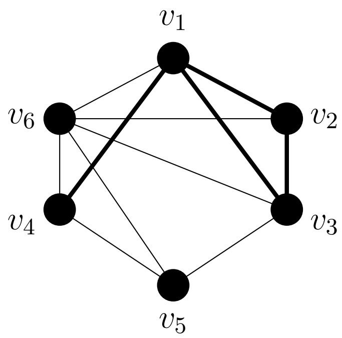


FIGURE 2.1: Graph G' (shown darker) is a subgraph of G.

2.1.2 Graph colouring

In graph theory, a *graph colouring* is a special case of *graph labelling* which is an assignment of labels traditionally called "colours" vertices of a graph subject to certain constraints. In this work, the type of colourig that is of interest is *vertex colouring*. A *proper vertex colouring* is a labelling of the graph's vertices with colours such that no two adjacent vertices have the same colour.

A colouring using at most k colours is called a *(proper) k-colouring*. A (proper) *k-vertex-colouring* of a graph G is a mapping from $V(G)$ to $\{1, 2, \ldots, k\}$ (whose elements are called *colours*) such that no two adjacent vertices receive the same colour. The *chromatic number* of a graph G, denoted $\chi(G)$, is the least number of distinct colours with which G can be properly coloured. A graph that can be assigned a *(proper) k-colouring* is *k-colourable*, and it is *k-chromatic* if its chromatic number is exactly k. A subset of vertices assigned to the same colour is called a *colour class*, every such class forms an *independent set*. Thus, a k-colouring is the same as a partition of the vertex set into k independent sets.

2.2 Computational Complexity Theory

Computational problems come in different varieties; some are easy, and some are hard. For example the sorting problem is an easy one compared to the scheduling problem where say we have to find a schedule for the entire university to satisfy some reasonable constraints, such as that no two classes take place in the same room at the same time. The scheduling problem seems to be much harder than the sorting problem.

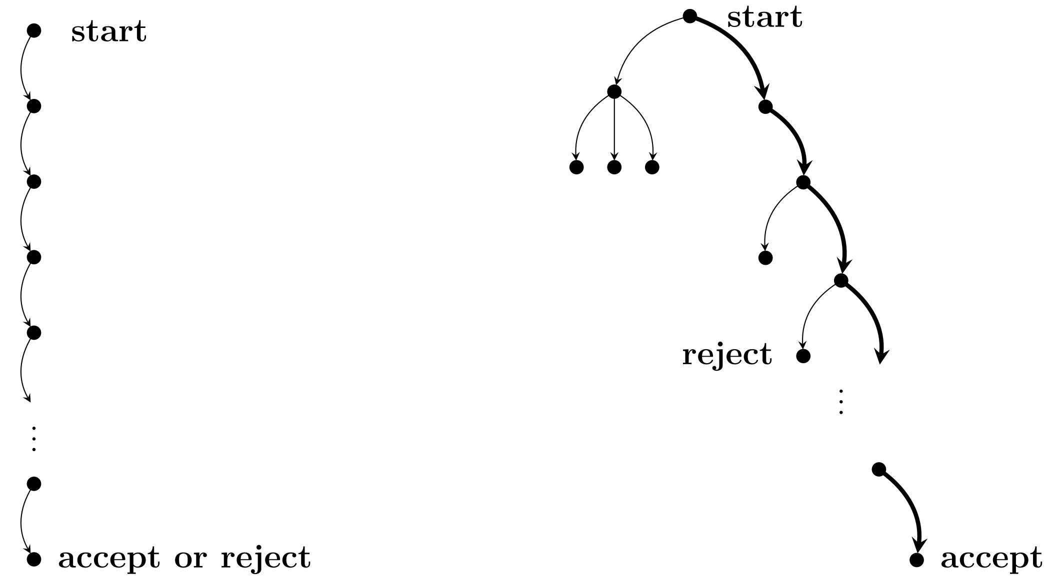
In theoretical computer science, the theory of computation studies how efficiently a problem can be solved on a model of computation, using an algorithm. A *problem* or a *language* is a set L of strings of length at most n over a finite alphabet Σ. A *decision problem* is a problem that can be posed as a YES/NO question. A string $s \in L$ is a yes instance of L and a string $s \notin L$ is a no-instance of L. An *algorithm* is an unambiguous procedure of how to solve a class of problems.

The model of computation focused on in standard complexity theory is the *Turing Machine*. 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. A Turing Machine can be viewed as an automaton, following simple rules to change states, with an aim to end in an accepting or a rejecting state. Two critical ressources for the Turing Machine are *time* which is the number of steps it requires to reach an accepting or a rejecting state and *space* being the amount of information that needs to be remembered throughout the computation.

A *Nondeterministic Turing Machine* can at any computation step proceed with various possibilities in order to reach an accepting state contrary to a *Deterministic Turing Machine* (see Fig. 2.2).

2.2.1 Computational Complexity Classes

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 as mentioned earlier. In this work, we give an informal description of the classical complexity classes generally encountered. The interested reader can find full details and formal definitions in the excellent textbook of Sipser [36]. The



(A) Deterministic computation. (B) Nondeterministic computation.

FIGURE 2.2: Deterministic and Nondeterministic computations with an accepting branch.

following section describes the different computational complexity classes in which different decision problems fits.

In general, these complexity classes study how the critical resources (*time* and *space*) grows in terms if the input size. For decision problems, the input is the description of the instance and its size is the number of bits required to encode it.

2.2.1.1 Time complexity classes

We start by characterizing in terms of time. The class P consists of all problems solvable in polynomial time, that is, all problems solved by some algorithm in time that is at most linear, quadratic, cubic, or similar in the input size. If n represents the input size, a general polynomial might look like $5n^4 + 3n^2 + 10n - 1$. Similarly, the class EXP consists of all problems solvable in exponential time: $2^n, 5^n, 2^{n^2}$ or in general $2^{p(n)}$ where $p(n)$ is some polynomial. Note that EXP contains easier problems too, in particular, all of P.

2.2.1.2 Space complexity classes

On the other hand, the class PSPACE consists of all problems solvable in polynomial space. This class is the analog of P but measuring space instead of time. Similarly, EXPSPACE consists of all problems solvable in exponential space.

Time and space complexity classes are related in the following way: An optimal algorithm never uses more space than time. Thus, every problem in P is also in PSPACE. Also, any (deterministic) algorithm that uses s space can never use more than exponential-in-s time without repeating a position. Thus, every problem in PSPACE is also in EXP.

2.2.1.3 Nondeterminism

Next, we consider allowing nondeterminism in order to solve a problem. A nondeterministic algorithm can at any computation step, proceed with various possibilities (see Fig. 2.2b). A nondeterministic algorithm can be thought of as an extremely lucky: whenever it needs to make a decision, it by definition makes the correct choice. The class NP consists of all problems that can be solved in polynomial time by such a nondeterministic algorithm. Similarly, we can define NPSPACE for the nondeterministic analog of PSPACE, and NEXP for EXP.

2.2.1.4 Completeness

For each complexity class X, we call a problem X-hard if it is about as hard as every problem in X. (Here, we ignore polynomial factors in the difficulty.) We call a problem X-complete if it is both X-hard and in X. Thus, for example, NP-complete problems are among the hardest problems in NP, so they must not be in any strictly easier complexity class. Whether $\mathsf{P} = \mathsf{NP}$ is of course a major open problem, but assuming they are even slightly different, NP-complete problems are not in P. Thus, when classifying a problem into a particular complexity class, showing that the problem is amongst the hardest problems in a certain complexity class eliminates any doubt of whether the latter belongs in a lower complexity class. One technique of doing so is by *reduction*.

Reducibility By taking a known X-complete problem (B) and showing that solving the problem in which we are interested (A) is at least as hard as solving B, we can conclude that A is X-hard. We usually do this by showing a way to transform problem B into problem A.

Definition 2.2.1. 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.

Definition 2.2.2. Given two languages A and B, B is reducible to A written $B \leq_p A$ if there is a computable function $f : \Sigma^* \to \Sigma^*$, where for every $w, w \in B \iff f(w) \in B$. The function f is called the reduction of B to A.

2.2.1.5 Relationship of Complexity Classes

[16] So far we have that $\mathsf{P} \subseteq \mathsf{PSPACE} \subseteq \mathsf{EXP}$. $\mathsf{PSPACE} = \mathsf{NPSPACE}$ follows from the celebrated result of Savitch [33]. Concerning nondeterminism, a nondeterministic computation is at least as powerful as regular deterministic computation, so, for example, every problem in P is also in NP. On the other hand, nondeterministic computation can be simulated by trying both choices of each decision in turn, which takes exponentially more time, but about the same amount of space. Thus, for example, every problem in NP is also in PSPACE. Summing all together, we can conclude that :

$$\mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE} = \mathsf{NPSPACE} \subseteq \mathsf{EXP} \subseteq \mathsf{NEXP} \subseteq \mathsf{EXPSPACE}$$

All of the containments are believed to be strict, but beyond the above relations, the only strict containment known among those classes is $\mathsf{P} \subsetneq \mathsf{EXP}$. Whether $\mathsf{P} = \mathsf{NP}$ is the most famous unresolved question in Computational Complexity Theory.

2.3 Reconfiguration Graph

Viewing Reconfiguration problems from a graph-theoretic perspective, the notion of a *reconfiguration graph* naturally arises. Let $G = (V, E)$ be a reconfiguration graph where $V(G)$ is the vertex set consisting of all possible configurations and two nodes are connected by an edge if the corresponding configurations can each be obtained from the other by the application of a single transformation rule, *a reconfiguration step*. Any path or walk in the reconfiguration graph corresponds to a sequence of reconfiguration steps called a *reconfiguration sequence*. Although the terminology concerning reconfiguration problems has not yet stabilized in the litterature those are the terms that will be used throughout this work.

Chapter 3

The Nondeterministic Constraint Logic (NCL)

In this chapter, the Nondeterministic Constraint Logic model of computation is presented. This framework developed by Demaine and Hearn is motivated by the Sliding-block puzzles [17]. The main result of [15] introduces the new nondeterministic model of computation based on reversing edge directions in weighted directed graphs with minimum in-flow constraints on vertices. This model, referred to as Nondeterministic Constraint Logic, or NCL, is shown to have the same computational power as a space-bounded Turing machine.

Several decision problems in the NCL framework are proved to be PSPACE-complete [15]. These decision problems are then used to prove the PSPACE-completeness of well-known Sliding-block puzzles such as Rush Hour and Sokoban [16]. Demaine and Hearn argue that NCL can be considered as a model of computation in its own right instead of just a set of decision problems. Thus, proving a problem to be PSPACE-hard in the NCL framework simply requires the construction of a couple of gadgets that can be connected together. In the last section of [15] gives an interesting equivalent formulation of NCL in terms of sliding tokens along graph edges. This latter formulation will be the focus of sections 3.4 and 3.5 to prove that the Sliding token problem and labelled variant of the sliding token problem are PSPACE-complete (theorems 3.4.2.3 and 3.5 respectively).

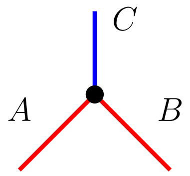
Roadmap. Section 3.1 describes the constraint logic using a graph formulation. Section 3.2 gives an overview of AND/OR constraint graphs which is the primary formulation used in the NCL framework. Section 3.3 present some complexity results of decision problems stemming from the constraint logic framework. In section 3.4 we detail the PSPACE-completeness proof of the sliding token problem using the alternative formulation of NCL (theorem 3.4.2.3), followed by the hardness proof of the labelled variant of the sliding token problem in section 3.5 (theorem 3.5). Lastly, in section 3.6 the variant of the SLIDING TOKEN problem studied by Bosnma is presented.

3.1 Graph Formulation

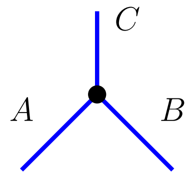
The simplest description of NCL is a graph formulation. An NCL machine consists of a *constraint graph*, $G = (V, E)$ that we can think of as our computation model. Let G be an undirected graph to which each edge is assigned a nonnegative integer and each vertex has a non negative *minimum inflow constraint*. A *configuration* of this machine is an orientation (direction) of the edges such that the sum of incoming edge weights at each vertex is at least the minimum in-flow constraint of that vertex. A *move* from one configuration to another configuration is simply the reversal of a single edge such that the minimum in-flow constraints remain satisfied.

3.2 AND/OR Constraint Graphs

As part of the constraint logic framework, Hearn and Demaine provided a restricted variant of Nondeterministic Constraint Logic (restricted NCL), in which the constraint graph G is planar, 3-regular, uses only weights $\in \{1, 2\}$ (referred to as red and blue edges respectively), the minimum in-flow = 2 and the graph is constructed from only two specific vertex types (AND and OR vertices). A vertex v of G is an *AND vertex* if exactly one incident edge has weight 2 (Figure 3.1a) and a vertex v of G is an *OR vertex* if all the incident edges have weight 2 (Figure 3.1b). Thus, a graph G is an *AND/OR constraint graph* if it consists of only AND and OR vertices.



(A) And vertex. Edge C may be directed outward if and only if edges A and B are both directed inward.



(B) Or vertex. Edge C may be directed outward if and only if either edge A or edge B is directed inward.

FIGURE 3.1: And and Or vertices. Red edges have weight 1, blue edges have weight 2, and all vertices have a minimum in-flow constraint of 2.

3.3 NCL Results

This section compiles the important complexity results linked to NCL. A first fundamental decision problem that arises in the NCL framework is about the satisfiability of a given constraint graph G. It is defined as follows :

CONSTRAINT GRAPH SATISFIABILITY
Instance: A constraint graph G.
Question: Does G have a legal configuration ?

In [16] Demaine and Hearn proved that the CONSTRAINT GRAPH SATISFIABILITY problem is NP−complete.

Another important problem regarding constraint graphs is about their reconfigurability. The CONFIGURATION-TO-CONFIGURATION (C2C) problem asks if given two configurations of a constraint graph G, whether they can be reconfigured into each other.

CONFIGURATION-TO-CONFIGURATION (C2C)
Instance: A constraint graph G and two legal configurations C_1, C_2 for G.
Question: Is there a sequence of legal configurations from $C_0, C_1, \ldots, C_t$ such that C_i is obtained from C_{i-1} by a legal move for each i with $1 \leq i \leq t$ and $C_0 = C_1, C_t = C_2$?

Hearn and Demaine established that the C2C problem is PSPACE-complete[16].

Similar to the C2C problem, the CONFIGURATION-TO-EDGE (C2E) problem asks whether a target edge e can be reversed given a constraint graph G.

CONFIGURATION-TO-EDGE (C2E)
Instance: A constraint graph G, a target edge e from G and an initial legal configuration C for G .
Question: Is there a sequence of legal configurations, starting with C, where every configuration is obtained from the previous by changing the orientation of one edge, so that e is eventually reversed?

Hearn and Demaine proved that the C2E problem is also PSPACE-complete[16].

More interestingly, the hardness result for C2C and C2E still holds when the vertices of G are restricted to be AND and OR vertices defined in section 3.2 ans referred as restricted NCL. C2C and C2E hardness proof involves a reduction from quantified Boolean formulas, based on the logical interpretation of AND/OR constraint graphs. Additional gadgets are required for simulating quantifiers and for converting red edges into blue edges (and vice versa), which can all be accomplished by combinations of AND and OR vertices [16].

Demaine and Hearn in fact strengthen this result even further and show that C2C and C2E both remain PSPACE-complete when the constraint graph G is planar [16]. This proof involves the construction of crossover gadgets that allow two edges to cross each other.

It is also possible to impose an additional restriction, while preserving the hardness of these problems: each vertex with three blue edges can be required to be part of a triangle with a red edge. Such a vertex is called a protected or, and it has the property that (in any valid orientation of the whole graph) it is not possible for both of the blue edges in the triangle to be directed inwards. This restriction makes it easier to simulate these vertices in hardness reductions for other problems[16]. Additionally, the constraint graphs can be required to have bounded bandwidth, and the problems on them will still remain PSPACE-complete[38].

3.4 Alternative formulation : SLIDING TOKENS

The SLIDING TOKEN problem was introduced by Hearn and Demaine in [15] as a variant of SLIDING-BLOCK puzzle with 1×1 blocks on a graph that requires no adjacent tokens, which can be seen as a reconfiguration problem for Independent Set. Suppose that we are given two independent sets I_b and I_r of a graph $G = (V, E)$ such that $|I_b| = |I_r|$ and imagine that a *token* is placed on each vertex in I_b. Then, the SLIDING TOKEN problem is to determine whether there exists a sequence $S = \langle I_1, I_2, \ldots, I_l \rangle$ of independent sets of G such that :

1. $I_1 = I_b, I_l = I_r$, and $|I_i| = |I_b| = |I_r|$ for all $i, 1 \leq i \leq l$; and
2. For each $i, 2 \leq i \leq l$ there is an edge xy in G such that $I_{i-1} \setminus I_i = \{x\}$ and $I_i \setminus I_{i-1} = \{y\}$.

That is, I_i can be obtained from I_{i-1} by sliding exactly one token on a vertex $x \in I_{i-1}$ to its adjacent vertex $y \in I_i$ along an edge $xy \in E(G)$. Such a sequence S, if exists, is called a *TS-sequence* in G between I_b and I_r. We denote by a 3-tuple (G, I_b, I_r) an instance of SLIDING TOKEN problem. If a TS-sequence S in G between I_b and I_r exists, we say that I_b is *reconfigurable* to I_r (and vice versa), and write $I_b \overset{G}{\leftrightsquigarrow} I_r$. The sets I_b and I_r are the *initial* and *target* independent sets, respectively. For a TS-sequence S, the *length* $\text{len}(S)$of S is defined as the number of independent sets in S minus one. In other words, $\text{len}(S)$ is the number of *token-slides* described in S. Figure 3.2 illustrates a TS-sequence of length 4 between two independent sets $I_b = I_1$ and $I_r = I_5$.

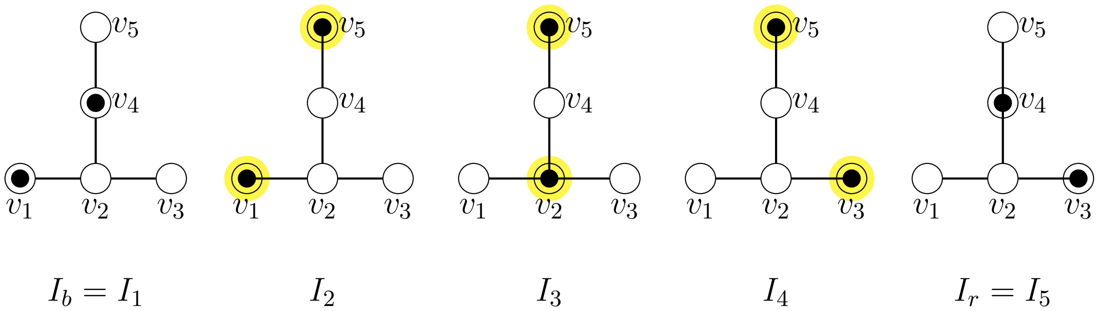


FIGURE 3.2: TS sequence $\langle I_1, I_2, \ldots, I_5 \rangle$ of independent sets which transforms $I_b = I_1$ into $I_r = I_5$ where the vertices in independent sets are depicted by small black circles (tokens).

3.4.1 Known results for the SLIDING TOKENS problem.

Analogous to the Independent Set problem being the key problem among thousands of NP-complete problems to prove NP-hardness, the SLIDING

TOKENS problem plays an important role since several PSPACE-hardness results have been proved using reductions from it.

For the SLIDING TOKENS problem, some polynomial time algorithms have been investigated as follows: Linear time algorithms have been shown for cographs (also known as P4-free graphs) [24] and trees [10]. Polynomial time algorithms are shown for bipartite permutation graphs [12], and claw-free graphs [4]. On the other hand, PSPACE-completeness is shown for graphs of bounded treewidth [31], and planar graphs [15].

3.4.2 PSPACE-completeness.

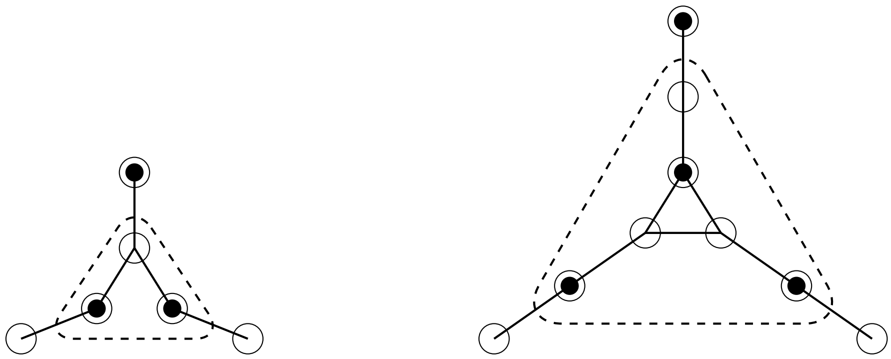
In this section we go over the PSPACE-completeness result of the SLIDING TOKENS problem, proved by a reduction from NCL. As seen in section 3.3, there are slightly different versions of decision problems for NCL and all of them are PSPACE$-$complete. For our purpose, we just need the version for the configuration-to-configuration for planar NCL. Recall that an instance of the C2E planar NCL problem is defined on a $3-$regular, planar, directed graph where each edge has a weight $\in \{1, 2\}$ and each vertex is either and AND or an OR vertex. The proof of theorem 3.4.2.3 is organised in sections 3.4.2.1 to 3.4.2.3 which explains the reduction structure, gadgets used and how they are connected together.

3.4.2.1 Reduction structure.

To show that the SLIDING TOKENS problem is PSPACE-complete we provide a reduction from C2E for AND/OR graphs such that the NCL instance is solvable if and only if the corresponding SLIDING TOKENS is solvable. The SLIDING TOKENS instance is constructed by piecing together gadgets which emulates the directed edges, the AND vertices and the OR vertices of the given NCL instance. We construct the corresponding NCL AND and OR vertex gadgets out of SLIDING-TOKENS subgraphs illustrated in figures 3.3a and 3.3b respectively.

3.4.2.2 The OR gadget and the AND gadget.

The construction of Fig.3.3a satisfies the same constraints as an NCL AND vertex, with the upper token corresponding to the blue edge and both lower tokens corresponding to the red edges. The upper token can slide in only when both lower tokens are slid out thus maintaining the flow constraint of an NCL AND vertex. Likewise, the construction of Fig.3.3b satisfies the same constraints as an NCL OR vertex with the upper and two lower tokens corresponding the the OR blue edges. The upper token in the OR gadget can slide in when either lower token is slid out and the internal token can then slide to one side or the other to make room. Here it is the internal token that ensures the NCL flow constraint is satisfied by sliding on a appropriate vertex among the three internal nodes to force one among the outer tokens are slid in.



(A) AND gadget (B) OR gadget

FIGURE 3.3: SLIDING TOKENS vertex gadgets.

3.4.2.3 AND/OR Graphs

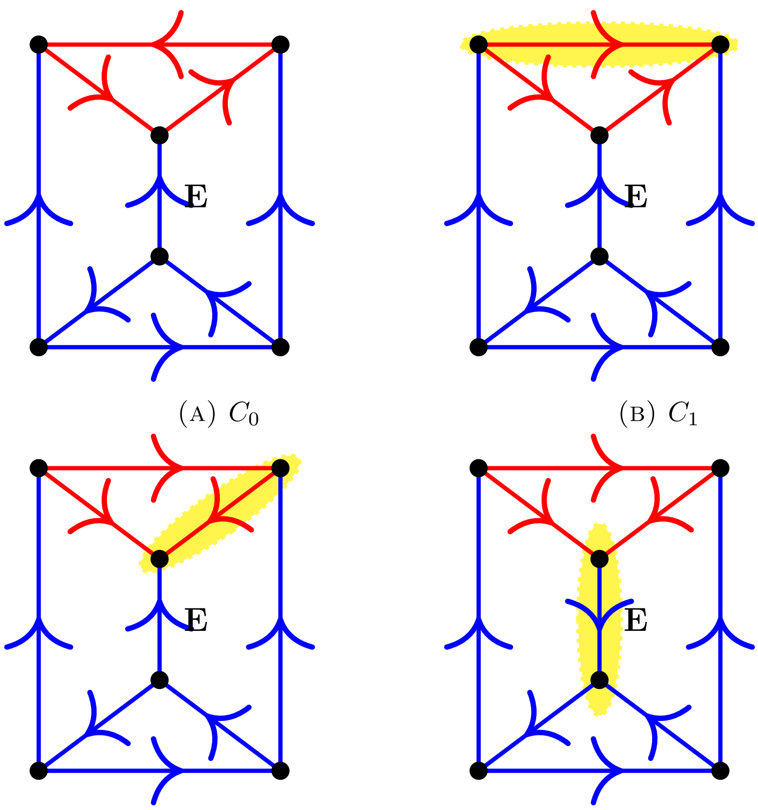
We showed how to construct AND and OR vertices. We now show how to connect the vertices into an arbitrary planar constraint graph. First, the edges that cross the dotted-line gadget borders are called "port" edges. A token on an outer port-edge vertex represents an inward-directed NCL edge, and vice-versa. Second, observe that no port token may ever leave its port edge. Choosing a particular port edge E, if we inductively assume that this condition holds for all other port edges, then there is never a legal move outside E for its token – another port token would have to leave its own edge first. Given an AND/OR graph G and two legal configurations C_1, C_2 for G, we construct a corresponding SLIDING-TOKENS graph by joining together AND and OR vertex gadgets at their shared port edges, placing the port tokens appropriately.

Theorem 3.4.1. *SLIDING TOKENS problem is* PSPACE*-complete.*

Proof. First, we show that SLIDING TOKENS problem is in PSPACE. The SLIDING TOKENS problem is in PSPACE since the state of the input graph can be described in a linear number of bits, specifying the position of each token and the list of possible moves from any state can be computed in polynomial time. Thus we can nondeterministically traverse the state space, at each step nondeterministically choosing a move to make, and maintaining the current state but not the previously visited states showing that SLIDING TOKENS is in NPSPACE. By Savitch's celebrated theorem, we have that NPSPACE = PSPACE [33], implying that SLIDING TOKENS is in PSPACE.

The SLIDING TOKENS problem is PSPACE-hard by a reduction from planar Nondeterministic Constraint Logic using the reduction structure provided above. The NCL instance is solvable if and only if the corresponding SLIDING TOKENS is solvable. □

Example 3.4.2. C2E to SLIDING TOKENS problem reduction.
Input instance : C2E for restricted NCL.

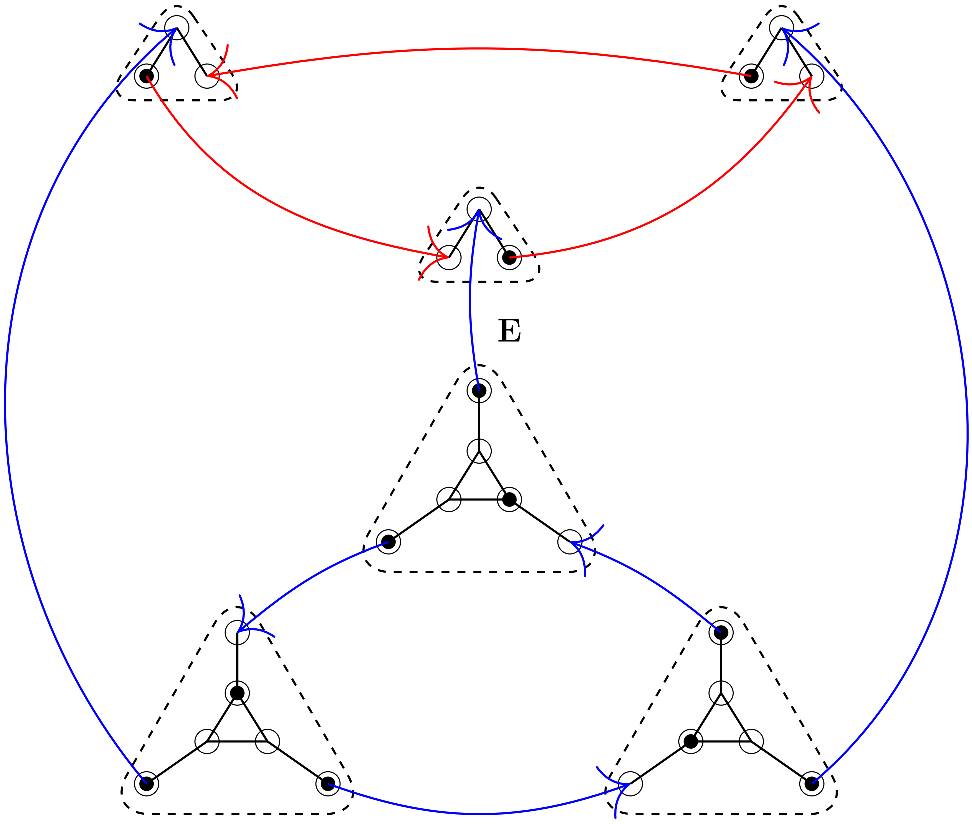


(A) C_0 (B) C_1

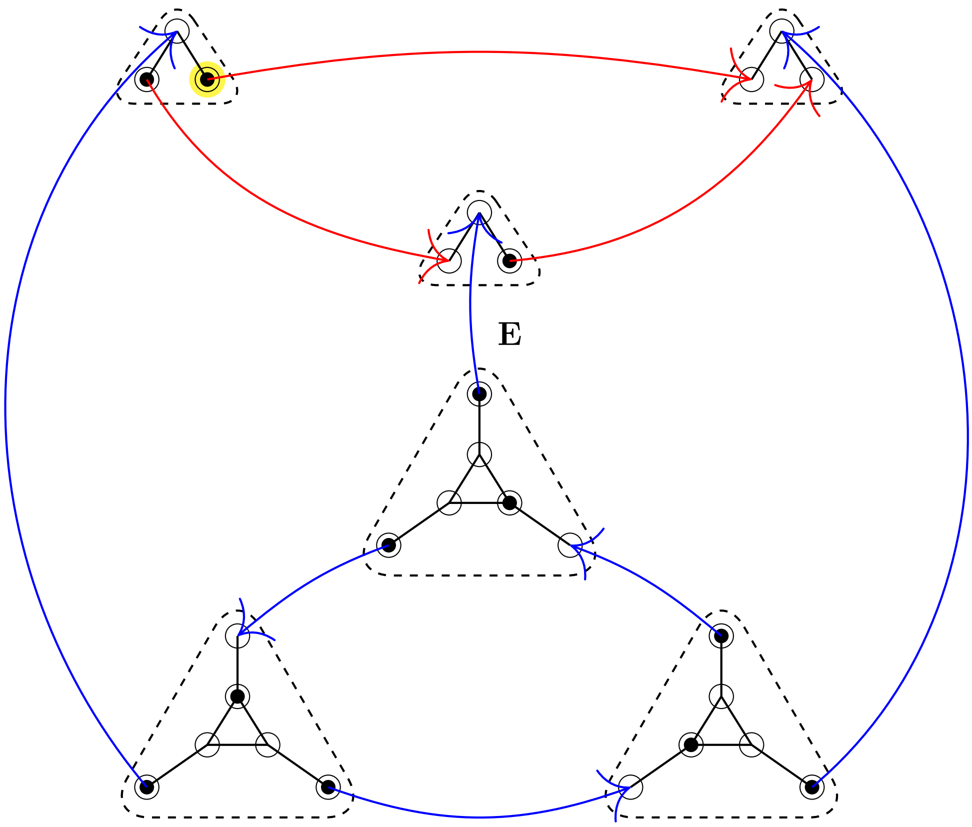
(C) C_2 (D) C_3

FIGURE 3.4: Reconfiguration sequence which transforms C_0 which is the initial configuration into C_3 = which is the target configuration.

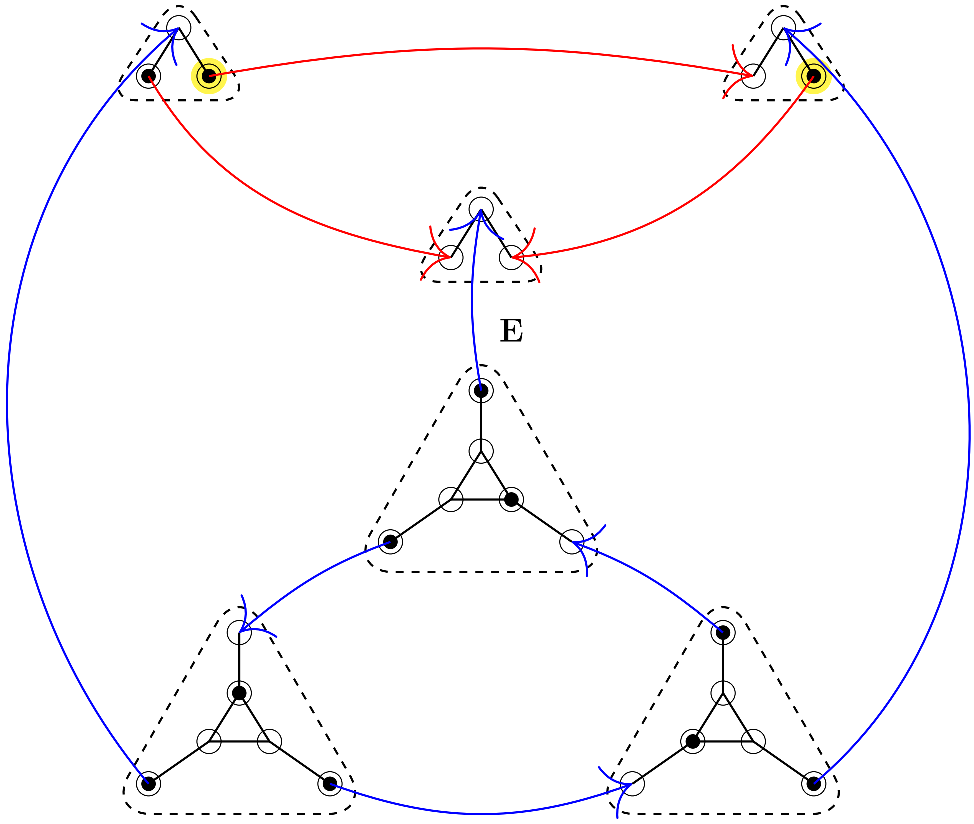
Output instance : SLIDING TOKENS instance.



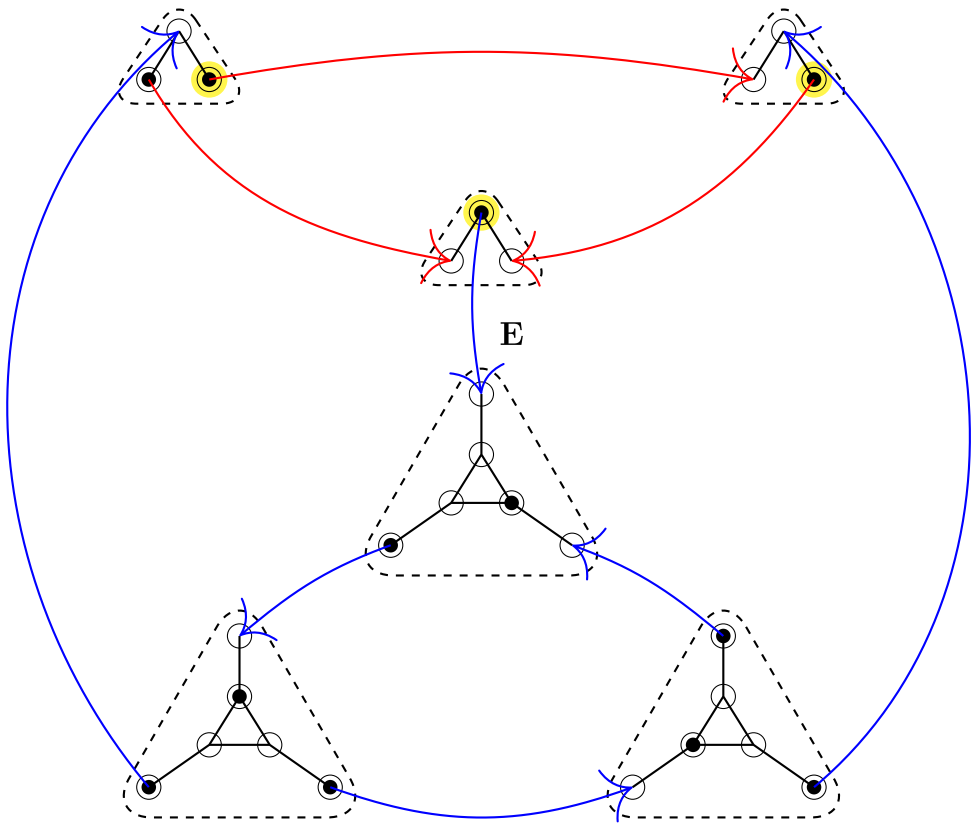
(A) S_0.



(B) S_1.



(c) S_2.



(d) S_3.

Figure 3.5: Reconfiguration sequence which transforms the initial configuration S_0 of the corresponding output SLIDING TOKENS instance to the target configuration S_3.

3.5 LABELLED variant of SLIDING-TOKENS Problem

The LABELLED SLIDING TOKENS problem is a variant of the SLIDING TOKENS problem where each token has a unique label. The purpose of this section is to prove that the SLIDING TOKENS problem remains PSPACE-hard even with labelled tokens. We will first start by giving a formal definition of the LABELLED SLIDING TOKENS problem.

Definition 3.5.1. (LABELLED SLIDING TOKENS problem) Suppose that we are given two independent sets I_b and I_r of a graph $G = (V, E)$ such that $|I_b| = |I_r|$ and imagine that a *labelled token* is placed on each vertex in I_b where each label is unique. Then, the LABELLED SLIDING TOKEN problem is to determine whether there exists a sequence $S = \langle I_1, I_2, \ldots, I_l \rangle$ of independent sets of G such that :

1. $I_1 = I_b, I_l = I_r$, and $|I_i| = |I_b| = |I_r|$ for all $i, 1 \leq i \leq l$; and
2. For each $i, 2 \leq i \leq l$ there is an edge xy in G such that $I_{i-1} \setminus I_i = \{x\}$ and $I_i \setminus I_{i-1} = \{y\}$.

That is, I_i can be obtained from I_{i-1} by sliding exactly one labelled token on a vertex $x \in I_{i-1}$ to its adjacent vertex $y \in I_i$ along an edge $xy \in E(G)$. We denote by a 4-tuple (G, I_b, I_r, T) an instance of LABELLED SLIDING TOKEN problem where T is a set of labelled tokens.

Theorem 3.5.2. *The LABELLED SLIDING TOKENS problem is* PSPACE-*complete.*

Proof. Theorem 3.5 is proved by a reduction from the C2E problem for AND/OR graph such that the NCL instance is solvable if and only if the corresponding LABELLED SLIDING TOKENS instance is solvable. The AND and OR gadgets used in the reduction are exactly the same gadgets shown in figure 3.3 except for the fact that the tokens are labelled. Each token of each gadget is attributed a unique label. The same reduction given by Demaine and Hearn can be applied even when the input NCL instance is labelled due this following observation.

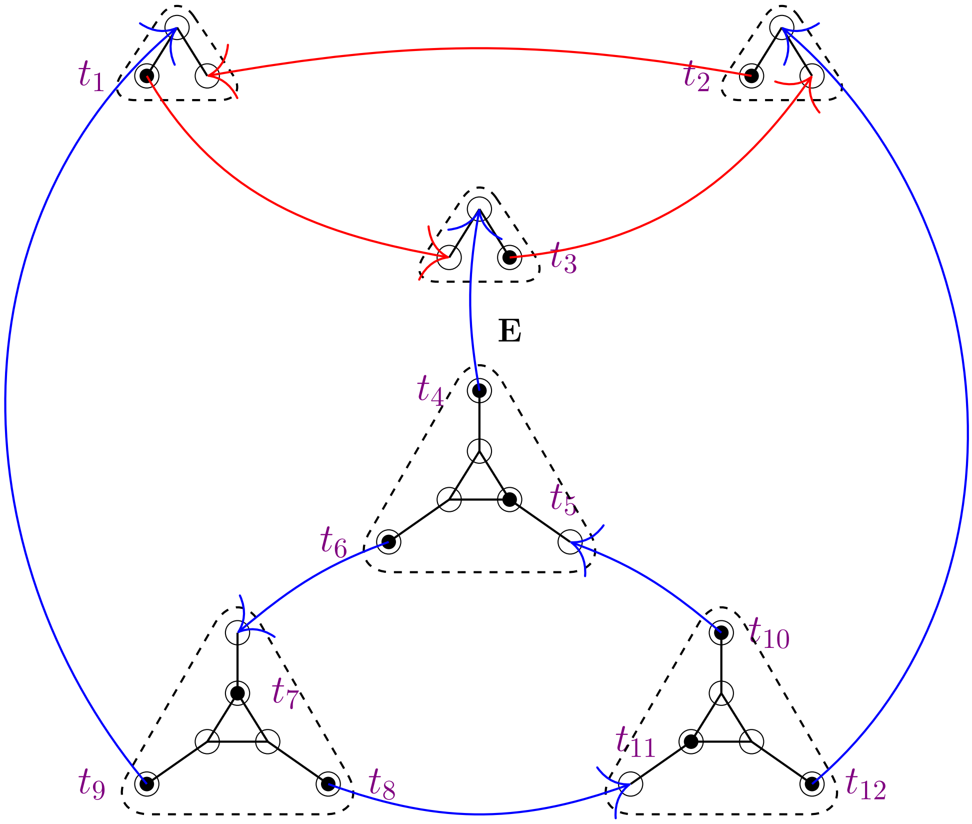
Observation 3.5.3. *No port token may ever leave its port edge.*

This condition is inductively assumed for all other port edges, choosing a particular port edge E there is never a legal move outside E for its token – another port token would have to leave its own edge first. Thus, there is no possibility of having solutions that are only feasible solutions for the SLIDING TOKENS problem and not for the LABELLED SLIDING TOKENS problem since the tokens of each gadget never slides further than their port edge. □

Example 3.5.4. C2E to LABELLED SLIDING TOKENS problem reduction.

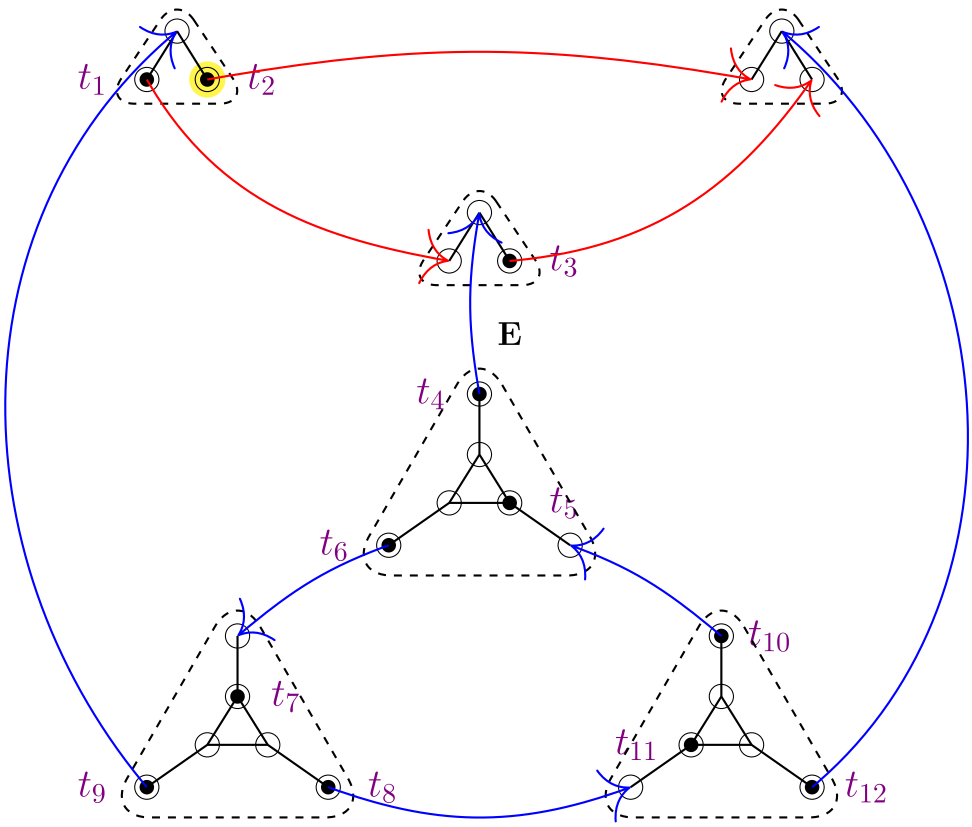
Input instance : C2E for restricted NCL same as figure 3.4.

Output instance : LABELLED SLIDING TOKENS instance.



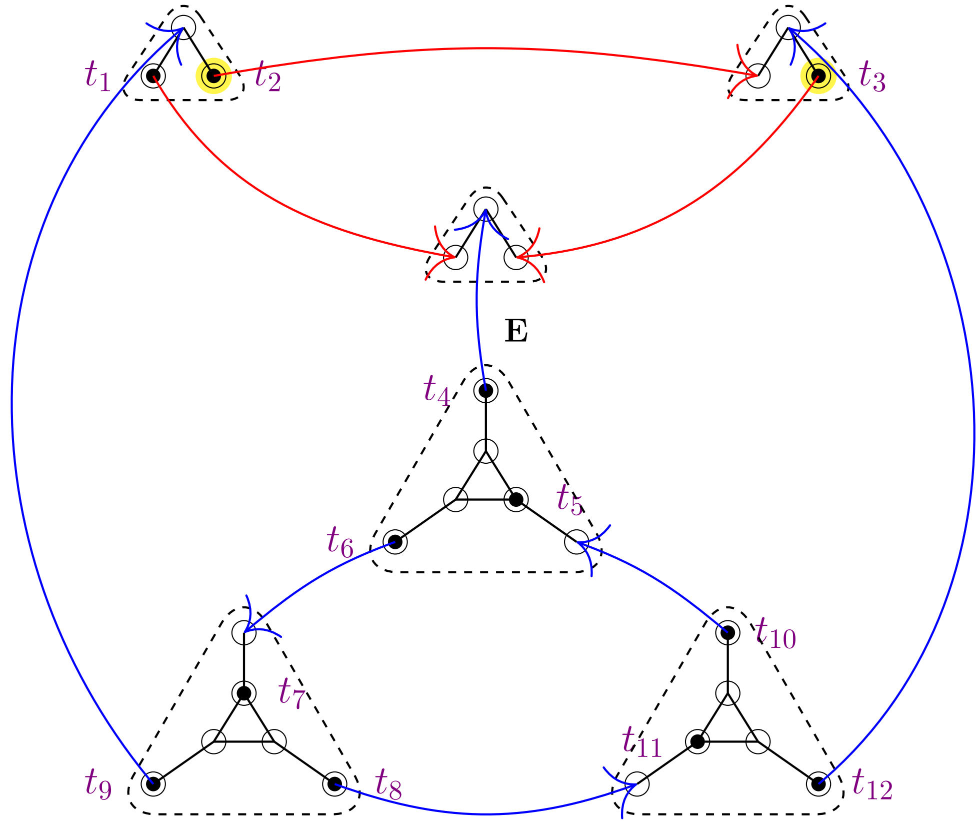
t_1
t_2
t_3
E
t_4
t_5
t_6
t_7
t_8
t_9
t_{10}
t_{11}
t_{12}

(A) S_0.

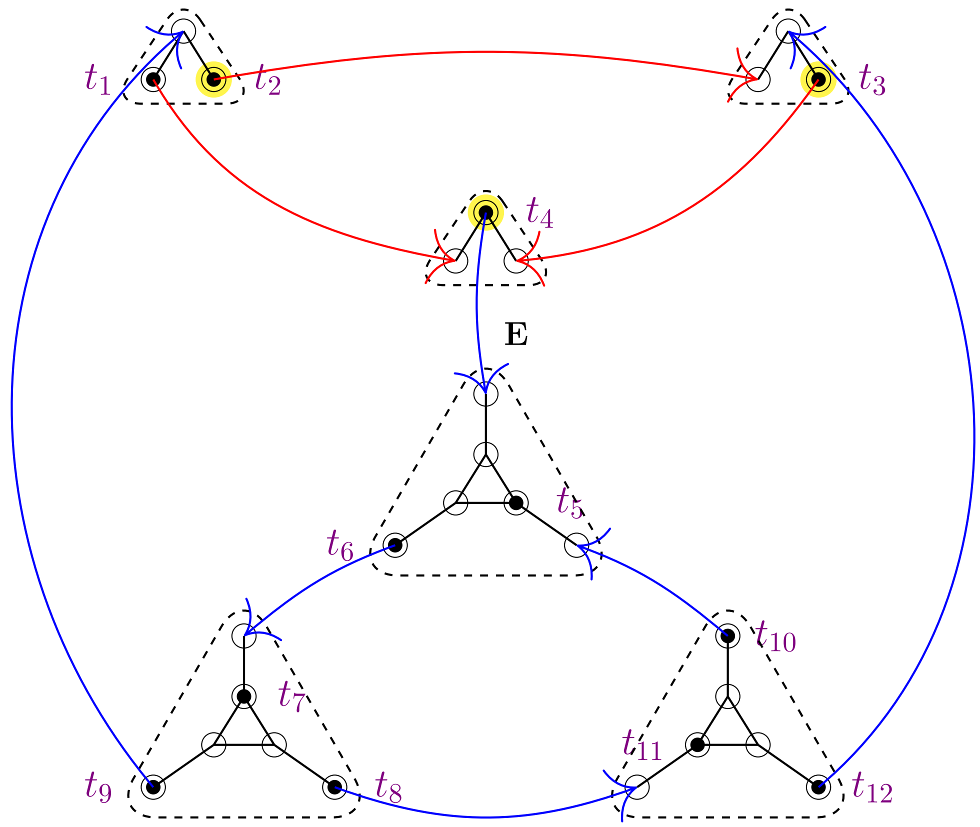


t_1
t_2
t_3
E
t_4
t_5
t_6
t_7
t_8
t_9
t_{10}
t_{11}
t_{12}

(B) S_1.



(c) S_2.



(d) S_3.

Figure 3.6: Reconfiguration sequence which transforms the initial configuration S_0 of the corresponding output LABELLED SLIDING TOKENS instance to the target configuration S_3.

3.6 Standard Sliding-Token Problem

Another variant of SLIDING TOKEN problem was studied by Bosnma. In [2], Bonsma showed that a slightly different version of the SLIDING TOKEN problem is also PSPACE-hard. This latter version called the Standard sliding token problem. Bonsma and Cereceda showed in [2] that the sliding tokens problem remains PSPACE-complete even for very restricted graphs and token configurations, defined as follows : The graph G_s is composed of *token triangles* (i.e., copies of K_3), *token edges* (i.e., copies of K_2) and link edges. Every vertex of G_s is part of exactly one token triangle or one token edge. Token triangles and token edges are all mutually disjoint, and joined together by link edges. Moreover, each vertex in a token triangle is of degree exactly 3, and G_s has a planar embedding such that every token triangle forms a face. Thus, The maximum degree of G_s is 3 and minimum degree is 2. An instance of the Standard Sliding token problem is shown in figure 3.7.

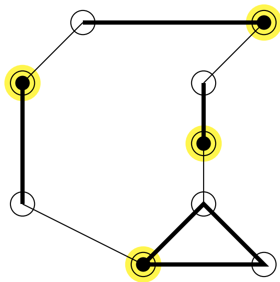


FIGURE 3.7: An example of a restricted instance graph G together with a standard token configuration.

We say that a token configuration T of G_s is *standard* if each token triangle and token edge of G_s contains exactly one token in T. Then, any move from a standard token configuration results in another standard token configuration since any token will never leave its token triangle or token edge, and will never slide along a link edge because the first time any token would slide to another triangle or edge, it would become adjacent to the token belonging to this triangle or edge. So tokens may never slide along a link edge. It is this latter observation that will allow us to prove the PSPACE-hardness of the labelled variant.

The SLIDING TOKENS problem remains PSPACE-complete even if G_s is such a restricted graph and both T_0 and T_t are standard token configurations [2]. This restricted problem is called the Standard SLIDING TOKENS problem.

Chapter 4

Reconfiguration of satisfiability problems

In this chapter, Boolean satisifiability reconfiguration problems are presented. For decades the Boolean satisfiability problem also known as SAT has fascinated scientific world. In 1978 Schaefer proposed a framework for expressing variants of the satisfiability problem, and showed a dichotomy theorem: the satisfiability problem for certain classes of Boolean formulas is in P while it is NP-complete for all other classes in the framework [34]. In a single stroke, this result pinpoints the computational complexity of all well-known variants of SAT, such as 3-SAT, HORN 3-SAT, NOT-ALL-EQUAL 3-SAT, and 1-IN-3 SAT [14]. Since then, dichotomies or trichotomies have been established for several aspects of the satisfiability problem such as optimization [9, 27], inverse satisfiability [26], minimal satisfiability [28], and 3-valued satisfiability [6].

Continuing Schaefer's work, Gopalan et al. were interested by the connectivity properties of the solution graph motivated mainly by research on satisfiability algorithms and the satisfiability threshold. The structure of the solution space of Boolean formulas can be characterized by a *solution graph* where the vertices are the solutions (satisfying assignments) and two solutions are connection iff they differ in exactly one variable. The connectivity problem (Conn) is to decide whether the solutions of a given Boolean formula φ on n variables induce a connected subgraph of the n-dimensional hypercube, while the st-connectivity problem (st-Conn) is to decide whether two specific solutions s and t of φ are connected. Concerning the st-connectivity question, they proved dichtomies for the diameter of connected components and the complexity. For the connectivity question, they conjectured a trichotomy. Building on this work, recently the trichotomy was established by Schwerdtfeger in [35] and showed that Conn is either in P, coNP-complete, or PSPACE-complete.

Recently, Gopalan et al.'s results have been applied to reconfiguration problems, mostly the st-connectivity and connectivity questions have been applied to reconfiguration problems such as INDEPENDENT-SET RECONFIGURATION, GRAPH-k-COLORING RECONFIGURATION, and many complexity results were obtained.

Roadmap. In section 4.1 we introduce Schaefer's framework and remarkable results, followed by section 4.2 presenting Gopalan et al.'s results. Finally in section 4.3 we present Schwerdtfeger's results which establishes the trichotomy conjectured by Gopalan et al.'s for the connectivity problem.

4.1 Schaefer's framework

Schaefer identified the complexity of every satisfiability problem SAT($\mathcal{S}$), where $\mathcal{S}$ ranges over all finite sets of logical relations [35]. To state Schaefer's main result, we need to define some basic concepts. We will use the standard notions and definitions defined in [34] and [14].

4.1.1 Preliminaries

A CNF-formula is a Boolean formula of the form $C_1 \wedge \cdots \wedge C_m$, where each C_i is a finite disjunction of literals where C_i is referred to as a clause. A k-CNF formula ($k \geq 1$) is a CNF-formula where each C_i has at most k literals.

A *logical relation* R is a non-empty subset of $\{0,1\}^n$, for some $n \geq 1$ where n is the *arity* of R. A logical relation is a function that takes as input a Boolean vector and returns a Boolean. For a set $\mathcal{S}$ of logical relations, a $\mathcal{S}$-formula is a conjuntion of logical relations from $\mathcal{S}$, where the arguments of each relation are freely chosen among a set of variables.

For a finite set of relations $\mathcal{S}$, a $CNF(\mathcal{S})$-formula over a set of variables $V = \{x_1, \ldots, x_n\}$ is a finite conjunction $C_1 \wedge \cdots \wedge C_m$, where each C_i is an expression of the form $R(\xi_1, \ldots, \xi_k)$, with a k-ary relation $R \in S$, and each ξ_j is a variable from V or one of the constants $0, 1$. A *solution* if a CNF($\mathcal{S}$)-formula φ is an assignment $s = (a_1, \ldots, a_n)$ of Boolean values to the variables that makes every clause of φ true. A CNF($\mathcal{S}$)-formula is *satisfiable* if it has at least one solution.

The satisfiability problem SAT($\mathcal{S}$) associated with a finite set $\mathcal{S}$ of logical relations asks: given a CNF($\mathcal{S}$)-formula φ, is it satisfiable? All well known restrictions of Boolean satisfiability, such as 3-SAT, NOT-ALL-EQUAL 3-SAT, and POSITIVE 1-IN-3 SAT, can be cast as SAT($\mathcal{S}$) problems, for a suitable choice of $\mathcal{S}$.

Schaefer identified the complexity of every satisfiability problem SAT($\mathcal{S}$), where $\mathcal{S}$ ranges over all finite sets of logical relations. To state Schaefer's main result, we need to define some properties and relations for R and $\mathcal{S}$.

Definition 4.1.1. Let R be a logical relation.

1. R is *bijunctive* if it is the set of solutions of a 2-CNF formula.
2. R is *Horn* if it is the set of solutions of a Horn formula, where a Horn formula is a CNF-formula where each C_i has at most one positive literal.
3. R is *dual Horn* if it is the set of solutions of a dual Horn formula, where a dual Horn formula is a CNF-formula where each C_i has at most one negative literal.
4. R is *affine* if it is the set of solutions of a system of linear equations over $\mathcal{Z}_2$.

Definition 4.1.2. A set S of logical relations is Schaefer if at least one of the following conditions holds:

1. Every relation in $\mathcal{S}$ is bijunctive.
2. Every relation in $\mathcal{S}$ is Horn.
3. Every relation in $\mathcal{S}$ is dual Horn.
4. Every relation in $\mathcal{S}$ is affine.

We are now ready to present Schaefer's theorem :

Theorem 4.1.3. *(Schaefer's Dichotomy Theorem [34]) Let S be a finite set of logical relations. If $\mathcal{S}$ is Schaefer, then SAT($\mathcal{S}$) is in* P*; otherwise, SAT($\mathcal{S}$) is* NP*-complete.*

Theorem 4.1.1 is called a dichotomy theorem because Ladner [29] has shown that if $\mathsf{P} \neq \mathsf{NP}$, then there are problems in NP that are neither in P, nor NP-complete. Thus, Theorem 4.1.1 asserts that no SAT($\mathcal{S}$) problem is a problem of the kind discovered by Ladner.

4.2 Gopalan et al.'s results

Gopalan et al. continued expanding Schaefer's dichotomie and focused on the connectivity of Boolean satisfiability. They believed that the connectivity properties of Boolean satisfiability merit some study in their own right, as they shed light on the structure of the solution space of Boolean formulas thus specifically addressed CNF($\mathcal{S}$)-formulas and studied the complexity of the following two decision problems,

- the connectivity problem $Conn(S)$, that asks for a given $CNF(S)$-formula φ whether $G(\varphi)$ is connected,
- the st-connectivity problem $st{-}Conn(S)$, that asks for a given $CNF(S)$-formula φ and two solutions s and t whether there a path from s to t in $G(\varphi)$.

4.2.1 Complexity-theoric dichotomies

Schaefer showed that the satisfiability problem is solvable in polynomial time precisely for formulas built from Boolean relations all of which are bijunctive, or all of which are Horn, or all of which are dual Horn, or all of which are affine. Gopalan et al. identified new classes of Boolean relations, called *tight relations*, that properly contain the classes of bijunctive, Horn, dual Horn, and affine relations. They then showed that st-connectivity is solvable in linear time for formulas built from tight relations, and PSPACE-complete in all other cases. Their second main result is a dichotomy theorem for the connectivity problem: it is in coNP for formulas built from tight relations, and PSPACE-complete in all other cases.

4.2.2 Structural dichotomy theorem

In addition to these two complexity-theoretic dichotomies, they established a structural dichotomy theorem for the diameter of the connected components of the solution space of Boolean formulas. This result asserts that, in the

PSPACE-complete cases, the diameter of the connected components can be exponential, but in all other cases it is linear. Thus, small diameter and tractability of the st-connectivity problem are remarkably aligned.

Their results are summarized in comparison to the satisfiability problem SAT($\mathcal{S}$) in table 4.1.

$\mathcal{S}$	SAT($\mathcal{S}$)	ST-CONN($\mathcal{S}$)	CONN($\mathcal{S}$)	Diameter
Schaefer	P	P	coNP	$O(n)$
Tight, not Schaefer	NP-compl.	P	coNP-compl.	$O(n)$
Not tight	NP-compl.	PSPACE-compl.	PSPACE-compl.	$2^{\Omega(\sqrt{n})}$

TABLE 4.1: Gopalan et al.'s results [13]

4.3 A computational Trichotomy for Conn(S)

Schwerdtfeger investigated issues in Gopalan et al.'s work and argued that repeated occurrences of variables in constraint applications can make the problems harder and the diameter exponential in some cases. This lead to a slight shift of the boundaries established by Gopalan et al.'s in the hard direction and an introduction to the new classes called *safely tight* and relations *CPSS* fitted to the correct boundaries.

The following table summarizes their results.

$\mathcal{S}$	SAT($\mathcal{S}$)	ST-CONN($\mathcal{S}$)	CONN($\mathcal{S}$)	Diameter
CPSS	P	P	P	$O(n)$
Schaefer, not CPSS	P	P	coNP-compl.	$O(n)$
Safely tight, not Schaefer	NP-compl.	P	coNP-compl.	$2^{\Omega(\sqrt{n})}$
Not safely tight	NP-compl.	PSPACE-compl.	PSPACE-compl.	$2^{\Omega(\sqrt{n})}$

TABLE 4.2: Complete classification of the connectivity problems and the diameter for CNF($\mathcal{S}$)-formulas with constants, in comparison to SAT [35]

Chapter 5

SUBSET SUM RECONFIGURATION

The subset sum problem is a well-known NP-complete problem [36] in which given an integer x and a set of integers $S = \{a_1, a_2, \ldots, a_n\}$, we wish to find a subset $A \subseteq [n]$ such that $\sum_{i \in A} a_i = x$.

In [19], Ito and Demaine considered the following version of the subset sum problem which they defined as follows : Given a knapsack with a non negative integer capacity c, and a set A of n items $a_1, a_2, \ldots, a_n$, each of which has a non negative integer size $s(a_i), 1 \leq i \leq n$. We call a subset $A^{'}$ of A a *packing* if the total size of A does not exceed the capacity c, that is, $\sum_{a \in A'} s(a) \leq c$. Given an integer threshold k, the subset sum problem is to find a packing A whose total size is at least k, that is, $k \leq \sum_{a \in A'} s(a) \leq c$.

They then defined the SUBSET SUM RECONFIGURATION problem as follows : Given two *packings* A_1 and A_2, both of total size at least k, can we transform A_1 into A_2 via packings by moving (namely, either adding or removing) a single item to/from the previous one without ever going through a packing of total size less than k. Ito and Demain proved that the SUBSET SUM RECONFIGURATION is strongly NP-hard, and PSPACE-complete for the variant with conflict graph [19].

In this chapter, we explore another reconfiguration version of the subset sum problem referred to as the k-move SUBSET SUM RECONFIGURATION problem presented in [7]. We say that a set of integers A_1 can be *k-move reconfigured* into a second set of integers A_2 whenever the symmetric difference of A_1 and A_2 has cardinality at most k. It turns out that the

k-move Subset Sum reconfiguration problem is PSPACE-complete [7]. This chapter is dedicated to this result.

Roadmap. This chapter is organised in the following way : Sections 5.1 to 5.3 introduces the k-move sSUBSET SUM RECONFIGURATION problem, the LABELLED SLIDING TOKENS problem and the EXACT COVER RECONFIGURATION problem respectively along with proofs of their complexity results added with illustrative examples to help visualize the concepts. Finally, section 5.5 is dedicated to the geometric interpretation of the SUBSET SUM RECONFIFURATION problem and the k-move SUBSET SUM RECONFIGURATION problem.

5.1 k-move SUBSET SUM RECONFIGURATION

To begin our journey to the hardness proof of the 3-move SUBSET SUM RECONFIGURATION problem, we first start by defining the decision problem of the k-move SUBSET SUM RECONFIGURATION.

Definition 5.1.1. (k-move SUBSET SUM RECONFIGURATION Problem). Given two solutions A_1 and A_2 to an instance of the subset sum problem, can A_2 be obtained by repeated k-move reconfiguration, beginning with A_1, so that all intermediate subsets are also solutions?

Notice that the reconfiguration problem is trivial if the reconfiguration steps are restricted to involve only the removal or addition of a single element of S, as no single such move can maintain the same sum. The problem remains trivial for $k = 2$, since any removed element must be replaced by itself. For $k = 3$, the following theorem is proved:

Theorem 5.1.2. *The* 3*-move SUBSET SUM RECONFIGURATION problem is strongly* PSPACE*-complete.*

The proof of theorem 5.1.2 is organised in the following way : We first start by proving the membership of the k-move SUBSET SUM RECONFIGURATION problem in PSPACE (lemma 5.1.3). Proving the hardness is done in two steps. The first step consists of reducing the LABELLED SLIDING TOKENS RECONFIGURATION problem to the EXACT COVER RECONFIGURATION problem (lemma 5.3.4). However before diving into the proof of lemma 5.3.4, we first take a detour to sections 5.2 and 5.3 where the LABELLED SLIDING TOKENS RECONFIGURATION problem and EXACT COVER RECONFIGURATION problem are introduced respectively.

The second step involves reducing the EXACT COVER RECONFIGURATION problem problem to the 3-move SUBSET SUM RECONFIGURATION problem (theorem 5.4.1).

Lemma 5.1.3. *For every $k \in \mathbb{N}$, the k-move SUBSET SUM RECONFIGURATION problem is in* PSPACE.

Proof. For an instance with $|S| = n$, there are $O(n^k)$ other subsets reachable by a $k-$move reconfiguration, since each such move can be specified by the set of items in the symmetric difference of the two subsets. So all adjacent subsets in the reconfiguration graph can be enumerated in polynomial time. Then the k-move SUBSET SUM RECONFIGURATION problem is in NPSPACE by the following algorithm : in the reconfiguration graph, repeatedly move between subsets by non-deterministically selecting a neighbour in polynomial time and space. Since NPSPACE= PSPACE[33], the k-move subset sum is also in PSPACE. □

5.2 LABELLED SLIDING TOKENS problem.

The SLIDING TOKENS problem is introduced section 3.4 of chapter 3. Hearn and Demaine proved that the SLIDING TOKENS PROBLEM is PSPACE-complete for planar graphs [15], as an example of the application of the nondeterministic constraint logic model and they implicitly proved that the SLIDING TOKENS PROBLEM is PSPACE-hard on 3-regular graphs since the reduction is done from a restricted NCL machine in which the underlying graph is planar and all vertices have degree three. The labelled variant of the SLIDING TOKENS problem, where each token has a unique label, is also PSPACE-hard and is proved in section 3.5 of chapter 3.

5.3 EXACT COVER RECONFIGURATION problem

The EXACT COVER RECONFIGURATION problem is the second problem introduced along this reduction. It will be referred to as the ECR problem throughout the rest of this thesis. The variant of the ECR problem considered here is the Exact cover split and merge reconfiguration defined as follows :

Definition 5.3.1. (Exact Cover Split and Merge Reconfiguration). Given a set $\mathcal{S}$ of subsets of a set $\mathcal{U}$, and two exact covers C_1 and $C_2 \subseteq \mathcal{S}, C_1$ can be reconfigured into C_2 via a split (and C_2 can be reconfigured into C_1 via a merge) provided that there exist $S_1, S_2, S_3 \subseteq \mathcal{S}$ with $C_1 - C_2 = S_1$ and $C_2 - C_1 = \{s_2, s_3\}$.

Since C_1, C_2 are exact covers it is mandatory that $S_1 = S_2 \cup S_3$ and $S_2 \cap S_3 = \emptyset$.

Thus, the ECR decision problem can be reformulated as follows :

Definition 5.3.2. (EXACT COVER RECONFIGURATION problem). Given a set $\mathcal{S}$ of subsets of a set $\mathcal{U}$, and two configuration C_1 and C_2, can C_1 be reconfigured into C_2 via repeated splits and merges ?

Example 5.3.3. Let $\mathcal{U} = \{1, 2, 3\}$ and $\mathcal{S} = \{\{1\}, \{2\}, \{3\}, \{1, 2\}, \{2, 3\}\}$ and the two given configurations be the following : $C_1 = \{\{1\}, \{2, 3\}\}$ and $C_2 = \{\{1, 2\}, \{3\}\}$. A solution would be the following :

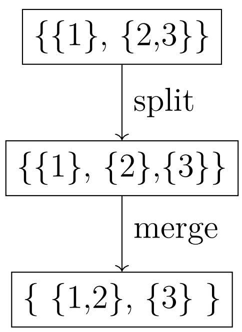


FIGURE 5.1: Reconfiguration sequence which transforms C_1 into C_2 via splits and merges.

5.3.1 k-colourability

Recall that a set $\mathcal{S}$ of subsets of a set U can be considered as a hypergraph $H = (U, \mathcal{S})$, where each element of U is a vertex and each element of $\mathcal{S}$ is a hyperedge. We say that a hypergraph is k-colourable whenever we can assign one of k colours to each vertex such that no two vertices in a hyperedge have the same colour. The colourability of the ECR problem is introduced for further use.

5.3.2 PSPACE-hardness result of the ECR problem.

In this section we prove the hardness result of the ECR problem as the first step in order to prove theorem 5.1.2. This is done by reducing the labelled variant of the SLIDING TOKENS PROBLEM to the ECR problem as mentioned earlier.

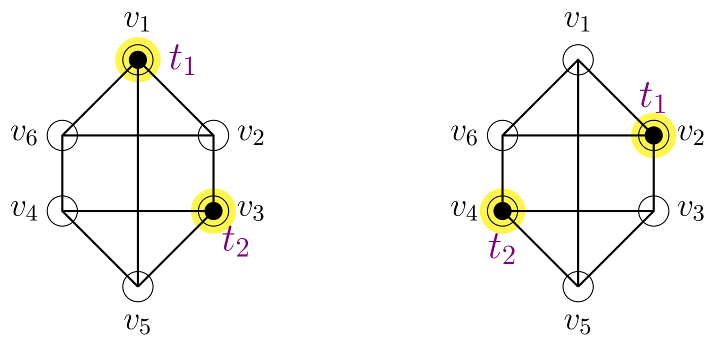
Lemma 5.3.4. *The Exact Cover Reconfiguration problem is* PSPACE*-hard for instances that are* 23*-colourable hypergraphs.*

Proof. The proof is structured in the following way : First, the input instance of the LABELLED SLIDING TOKENS reconfiguration problem is presented in section 5.3.2.1, followed by the description of the output instance in section 5.3.2.2. The definitions of some terms used in the reduction in given in section 5.3.2.3. Finally, sections 5.3.2.5 to 5.3.2.6 are devoted to the proof.

5.3.2.1 Input instance of the LABELLED SLIDING TOKENS PROBLEM

- $G = (V, E)$, a 3-regular graph.
- T, a set of labelled tokens.
- $p_1 : T \to V$, a function mapping each labelled token to a vertex placement in the starting configuration of the output instance.
- $p_2 : T \to V$, a function mapping each labelled token to a vertex placement in the ending configuration of the output instance.
- $I_1 = \{p_1(t) : t \in T\}$ and $I_2 = \{p_2(t) : t \in T\}$ are independent sets of size $|T| \leq |V|$.

Example 5.3.5. Input instance of the LABELLED SLIDING TOKENS PROBLEM



(A) Initial configuration S_1. (B) Target configuration S_2.

FIGURE 5.2: A LABELLED SLIDING TOKENS instance with initial and target configurations S_1 and S_2 respectively where $T = \{t_1, t_2\}$, $p_1(t_1) = v_1, p_1(t_2) = v_3$ and $p_2(t_1) = v_2, p_2(t_2) = v_4$.

5.3.2.2 Output $\mathcal{U}$ and $\mathcal{S}$

The output instance of the ECR problem contains a set $\mathcal{S}$ of subsets of a set $\mathcal{U}$ defined in 5.3 where :

- $\mathcal{U} = \{v_1, v_2, \ldots, v_{|V|}\} \cup \{t_1, t_2, \ldots, t_{|T|}\}$.
- The set $\mathcal{S}$ is defined as follows, for every pair of adjacent vertices v_i, v_j and token t_k
  - All subsets of $S_{i,j} - \{v_i\}$ and $S_{i,j} - \{v_j\}$.
  - $\{v_i, t_k\}$ and $\{v_j, t_k\}$.
  - $S_{i,j} \cup \{t_k\}$.

Example 5.3.6. Continuing with the running example, the output instance of the ECR problem is the following :

1. $\mathcal{U} = \{v_1, v_2, v_3, v_4, v_5, v_6\} \cup \{t_1, t_2\}$
2. And $\mathcal{S} = \{\{v_2, v_3, v_5, v_6\}, \{v_1, v_3, v_5, v_6\}, \{t_1, v_1\}, \{t_1, v_2\}, \{t_1, v_1, v_2, v_3, v_5, v_6\},$
   $\{v_1, t_2\}, \{v_2, t_2\}, \{v_1, v_2, v_3, v_5, v_6, t_2\}$ $(1, 2)$
   $\{v_2, v_4, v_5, v_6\}, \{v_2, v_4, v_5, v_6\}, \{t_1, v_1\}, \{t_1, v_3\}, \{t_1, v_2, v_4, v_5, v_6\},$
   $\{v_1, t_2\}, \{v_3, t_2\}, \{v_2, v_4, v_5, v_6, t_2\}$ $(1, 3)$
   $\{v_2, v_3, v_4, v_5, v_6\}, \{v_1, v_2, v_3, v_4, v_6\}, \{t_1, v_1\}, \{t_1, v_5\}, \{t_1, v_1, v_2, v_3, v_4, v_5, v_6\},$

$\{v_1, t_2\}, \{v_5, t_2\}, \{v_1, v_2, v_3, v_4, v_5, v_6, t_2\} \qquad (1,5)$
$\{v_2, v_3, v_5, v_6\}, \{v_2, v_3, v_5, v_6\}, \{t_1, v_1\}, \{t_1, v_4\}, \{t_1, v_2, v_3, v_5, v_6\},$
$\{v_1, t_2\}, \{v_4, t_2\}, \{v_2, v_3, v_5, v_6, t_2\} \qquad (1,4)$
$\{v_2, v_4, v_5, v_6\}, \{v_1, v_2, v_4, v_5\}, \{t_1, v_1\}, \{t_1, v_6\}, \{t_1, v_1, v_2, v_4, v_5, v_6\},$
$\{v_1, t_2\}, \{t_2, v_6\}, \{v_1, v_2, v_4, v_5, v_6, t_2\} \qquad (1,6)$
$\{v_1, v_3, v_4, v_5, v_6\}, \{v_1, v_2, v_4, v_5, v_6\}, \{t_1, v_2\}, \{t_1, v_3\}, \{t_1, v_1, v_2, v_3, v_4, v_5, v_6\},$
$\{v_2, t_2\}, \{v_3, t_2\}, \{v_1, v_2, v_3, v_4, v_5, v_6, t_2\} \qquad (2,3)$
$\{v_1, v_3, v_4, v_6\}, \{v_1, v_3, v_4, v_6\}, \{t_1, v_2\}, \{t_1, v_5\}, \{t_1, v_1, v_3, v_4, v_6\},$
$\{v_2, t_2\}, \{v_5, t_2\}, \{v_1, v_3, v_4, v_6, t_2\} \qquad (2,5)$
$\{v_1, v_3, v_5, v_6\}, \{v_1, v_3, v_5, v_6\}, \{t_1, v_2\}, \{t_1, v_4\}, \{t_1, v_1, v_3, v_5, v_6\},$
$\{v_2, t_2\}, \{v_4, t_2\}, \{v_1, v_3, v_5, v_6, t_2\} \qquad (2,4)$
$\{v_1, v_3, v_4, v_6\}, \{v_1, v_2, v_3, v_4\}, \{t_1, v_2\}, \{t_1, v_6\}, \{t_1, v_1, v_2, v_3, v_4, v_6\},$
$\{v_2, t_2\}, \{t_2, v_6\}, \{v_1, v_2, v_3, v_4, v_6, t_2\} \qquad (2,6)$
$\{v_1, v_2, v_4, v_5\}, \{v_1, v_2, v_3, v_4\}, \{t_1, v_3\}, \{t_1, v_5\}, \{t_1, v_1, v_2, v_3, v_4, v_5\},$
$\{v_3, t_2\}, \{v_5, t_2\}, \{v_1, v_2, v_3, v_4, v_5, t_2\} \qquad (3,5)$
$\{v_2, v_4, v_5, v_6\}, \{v_2, v_3, v_5, v_6\}, \{t_1, v_3\}, \{t_1, v_4\}, \{t_1, v_2, v_3, v_4, v_5, 6\},$
$\{v_3, t_2\}, \{v_4, t_2\}, \{v_2, v_3, v_4, v_5, v_6, t_2\} \qquad (3,4)$
$\{v_1, v_2, v_4, v_5\}, \{v_1, v_2, v_4, v_5\}, \{t_1, v_3\}, \{t_1, v_6\}, \{t_1, v_1, v_2, v_4, v_5\},$
$\{v_3, t_2\}, \{t_2, v_6\}, \{v_1, v_2, v_4, v_5, t_2\} \qquad (3,6)$
$\{v_1, v_3, v_4, v_6\}, \{v_1, v_3, v_5, v_6\}, \{t_1, v_5\}, \{t_1, v_4\}, \{t_1, v_1, v_3, v_4, v_5, v_6\},$
$\{v_5, t_2\}, \{v_4, t_2\}, \{v_1, v_3, v_4, v_5, v_6, t_2\} \qquad (5,4)$
$\{v_1, v_2, v_3, v_4\}, \{v_1, v_2, v_3, v_4\}, \{t_1, v_5\}, \{t_1, v_6\}, \{t_1, v_1, v_2, v_3, v_4\},$
$\{v_5, t_2\}, \{t_2, v_6\}, \{v_1, v_2, v_3, v_4, t_2\} \qquad (5,6)$
$\{v_1, v_2, v_3, v_5, v_6\}, \{v_1, v_2, v_3, v_4, 5\}, \{t_1, v_4\}, \{t_1, v_6\}, \{t_1, v_1, v_2, v_3, v_4, v_5, v_6\},$
$\{v_4, t_2\}, \{t_2, v_6\}, \{v_1, v_2, v_3, v_4, v_5, v_6, t_2\} \qquad (4,6)\}$

5.3.2.3 Preliminaries

Before diving in the proof, a few notions used in the reduction are presented hereunder :

Slide Set. For each pair of adjacent vertices $v_i, v_j \in V$ of the input instance of the SLIDING TOKENS problem, the set consisting of these two vertices and their neighbours is called a *slideset*, denoted $S_{i,j}$.

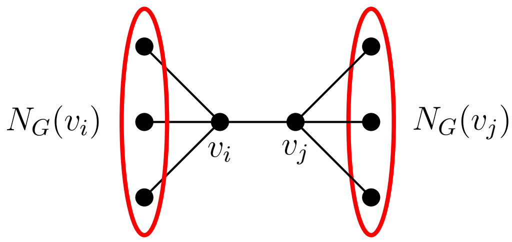


FIGURE 5.3: Slide set denoted S_{ij} of vertices v_i and v_j.

Maximally split configuration C. A configuration C of the output Exact Cover instance is called *maximally split* if every $c \in C$ contains exactly one vertex and up to one token.

Example 5.3.7. A maximally split configuration C

$$\boxed{C_1 = \{\{v_2\}, \{v_4\}, \{v_5\}, \{v_6\}, \{v_1, t_1\}, \{v_3, t_2\}\}.}$$

FIGURE 5.4: A maximally split configuration C_1.

Sploot set of a configuration C. For each EXACT COVER configuration C in the output instance, let sploot(C) be the set of all maximally split configurations reachable from C.

5.3.2.4 Output EXACT COVER starting and ending configurations, C_1 and C_2

The starting configuration C_1 is the union of $\{\{v_i\} : v_i \in V - I_1\}$ and, for every $v_i \in I_1$, a set $\{v_i, t_k\}$ with a distinct t_k. The ending configuration C_2 is then the union of $\{\{v_i\} : v_i \in V - I_2\}$ and, for every $v_i \in I_2$, a set $\{v_i, t_k\}$ with a distinct t_k.

Example 5.3.8. Continuing the running example, C_1 and C_2 would be :

- $C_1 = \{\{v_2, v_4, v_5, v_6\}, \{v_1, t_1\}, \{v_3, t_2\}\}$.
- $C_2 = \{\{v_1, v_3, v_5, v_6\}, \{v_2, t_1\}, \{v_4, t_2\}\}$.

5.3.2.5 23-colourability of the output instance $H = (U, \mathcal{S})$

The goal here is to make sure that no two vertices of distance at most 3 (i.e. in a common slide set see figure 5.3) have the same colour. More precisely, we want to prove that given 23-colours, no two vertices having a common slide set would have the same colour. This constraint will enforce the absence of tokens on neighbors of v_i and v_j, and the presence of a token on v_i or v_j, but not both making sure that any merge or split will result into a feasible configuration.

Given that the *kth* power G^k of an undirected graph G is another graph that has the same set of vertices, but in which two vertices are adjacent when their distance in G is at most k, to find the colourability of the output instance $H = (U, \mathcal{S})$, it suffices to compute $\Delta(G^3)$. Since G is 3−regular, G^3 has degree at most 21 proven by the following figure.

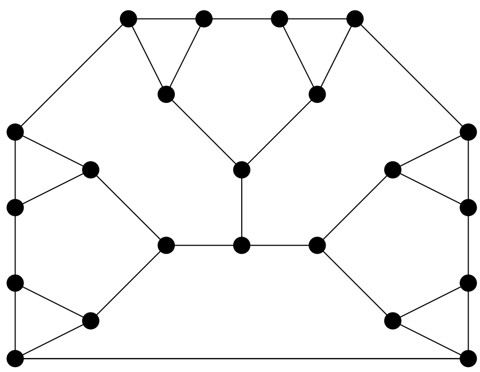


FIGURE 5.5: The maximum degree of a vertex in G^3

In graph theory, Brooks' theorem states that for any connected undirected graph G with maximum degree Δ, the chromatic number of G is at most Δ unless G is a complete graph or an odd cycle, in which case the chromatic number is $\Delta + 1$. So G can be 22−coloured such that no two vertices of distance at most 3 have the same colour. Colouring the tokens in $\mathcal{T}$ a distinct (23rd) colour then gives a colouring of $\mathcal{U}$ such that no pair of elements of a common set share the same colour.

5.3.2.6 High level idea

We first note that the subsets containing exactly one vertex and a token (e.g., $\{v_i, t_k\}$) represent the presence of the token t_k on vertex v_i and the subsets consisting of a slide set and a token (e.g., $S_{i,j} \cup t_k$) represent the presence of a "mid-slide" token between v_i and v_j. A "mid-slide" token can be interpreted as the token t_k not being properly on v_i or v_j but is "on it's way" from v_i to v_j. Therefore, sliding a token t_k from v_i to v_j is simulated by first merging $\{v_i, t_k\}$ and $S_{i,j} - \{v_i\}$ into $S_{i,j} \cup \{t_k\}$, and then splitting this set into this set into $S_{i,j} - \{v_j\}$ and v_j, t_k. The first step of this sequence enforces the absence of tokens on neighbors of v_i and v_j since by definition, the slide set S_{ij} of v_i and v_j contains all the of v_i and v_j and having the token t_k merged in S_{ij} means that t_k is not on any vertex. The second step then ensures the presence of a token on v_i or v_j, but not both. Before a merge-split sequence, additional splits and merges of token-less sets may be needed to obtain $S_{i,j} - \{v_i\}$.

5.3.2.7 Bijection between configurations.

The following defines a function f_{red} from token arrangements to maximally split covers in the following way :

1. Each token-less vertex corresponds to a set $\{v_i\}$ in the cover.
2. Each token t_k placed at v_i corresponds to a set $\{v_j, t_k\}$ in the cover.

Notice that f_{red} is a bijection since each cover configuration is an exact cover (no duplicates) and each token arrangement contains no duplicates either. Notice also that $f_{red}(p_1) = C_1$ and $f_{red}(p_2) = C_2$.

5.3.2.8 Reduction structure.

The remainder of the proof is devoted to proving the following claim:

Claim 5.3.9. *A token arrangement $p^{'}$ is reachable from a token arrangement p if and only if $f_{red}(p^{'})$ is reachable from $f_{red}(p)$ via split and merges.*

Both directions are proved inductively. That is, we consider only "adjacent" configurations. We also assume that the starting token arrangement $p : T \to V$ has $\{p(t) : t \in T\}$ independent.

5.3.2.9 SLIDING TOKENS reachability $\to$ EXACT COVER reachability

Let p be a token arrangement that can be reconfigured into $p^{'}$ via a token slide from v_i to v_j. Then $f_{red}(p^{'})$ can be reached from $f_{red}(p)$ via the following sequence of merges and splits.

1. Repeatedly merge token-less vertex sets to form $S_{i,j} - \{v_i\}$.

2. Merge $S_{i,j} - \{v_i\}$ and $\{v_i, t_k\}$ into $S_{i,j} \cup \{t_k\}$.

3. Split $S_{i,j} \cup \{t_k\}$ into $S_{i,j} - \{v_j\}$ and $\{v_j, t_k\}$.

4. Repeatedly split the token-less vertex set $S_{i,j} - \{v_j\}$ into single vertex sets.

Example 5.3.10. A token slide from v_1 to v_6 is simulated by first merging $\{v_1, t_1\}$ and $S_{1,6} - \{v_1\}$ into $S_{1,6} \cup \{t_1\}$, and then splitting this set into this set into $S_{1,6} - \{v_6\}$ and $\{v_6, t_1\}$.

5.3.2.10 EXACT COVER reachability → SLIDING TOKENS reachability

For each exact cover configuration C in the output instance, let *sploot(C)* be the set of all maximally split configurations reachable from C. Let C and $C^{'}$ be two maximally split configurations such that C can be reconfigured into $C^{'}$ and C_{inter} be the first configuration encountered in the reconfiguration sequence such that *sploot(C_{inter})* $\neq \{C\}$.

Observation 5.3.11. *Since C and $C^{'}$ are both maximally split configurations, the only way of obtaining C_{inter} is by merging two sets, one of which contains a token. Thus, C_{inter} is obtained by merging $\{v_i, t_k\}$ and $S_{i,j} - \{v_i, t_k\}$ to form $S_{i,j} \cup \{t_k\}$ for some v_i, v_j and t_k.*

Remark 5.3.12. It may be the case for other pairs $i^{'}, j^{'}$ that $S_{i,j} = S_{i^{'},j^{'}}$.

Once C_{inter} is reached two moves can be considered to move forward :

1. Either split $S_{i,j} \cup t_k$ back to $S_{i,j} - \{v_i, t_k\}$ to obtain the previous configuration.

2. Or split $S_{i,j} \cup t_k$ into $S_{i^{'},j^{'}} - \{v_j^{'}, t_k\}$ where $S_{i,j} = S_{i^{'},j^{'}}$.

By definition of the exact cover configuration C, since $S_{i,j} - \{v_i\}, \{v_i, t_k\} \in C$, the token arrangement p with $f_{red}(p) = C$ has no tokens on vertices in $S_{i,j}$ except for token t_k on v_i. Added the fact that $S_{i,j} = S_{i^{'},j^{'}}$, it contains all the neighbors of $v_i, v_i^{'}, v_j, v_j^{'}$. Thus the token arrangement obtained by moving the location of t_k in p from v_i to v_j, $v_i^{'}$, or $v_j^{'}$ results in an independent set because the constraint to statisfy in order to split from C_{inter} to $C^{'}$ was to split s.t $S_{i,j} = S_{i',j'}$.

So all that remains is to prove that there are a sequence of slides moving t_k from v_i to $v_j^{'}$ via vertices in $\{v_i, v_j, v_i^{'}, v_j^{'}\}$. Since $S_{i,j} = S_{i',j'}$ it means that $v_i^{'}, v_j^{'} \in S_{i,j}$ too. So either $v_i \in v_i^{'}, v_j^{'}$, or there is an edge $\{v_i, v_i^{'}\}$ or $\{v_i, v_j^{'}\} \in E$. So t_k can slide from v_i to either $v_i^{'}$ or $v_j^{'}$ (via 0 or 1 slide), and then from $v_i^{'}$ or $v_j^{'}$ to $v_j^{'}$ (via 0 or 1 slide). □

5.4 3-move SUBSET SUM RECONFIGURATION problem

Having established the complexity of the ECR problem, we can finally prove the main theorem of this chapter (theorem 5.4.1).

Theorem 5.4.1. *The* 3*-move SUBSET SUM RECONFIGURATION problem is strongly* PSPACE*-complete.*

Proof. The reduction is from the ECR problem for instances that are 23−colourable induced hypergraphs, proved PSPACE-hard. Recall that the ECR instance contains a set $\mathcal{S}$ of subsets of a set $\mathcal{U}$ and two exact covers C_1 and $C_2 \subseteq \mathcal{S}$.

5.4.0.1 High level idea

The goal is to find a correspondence between the addition and removal of elements in the 3−move SUBSET SUM RECONFIGURATION instance and the splits and merges of the ECR instance. The correspondence is the following : Every 3−move SUBSET SUM RECONFIGURATION step is either a *merge* where elements a_i and a_j are replaced by $a_i + a_j$ or a *split* where $a_i + a_j$ is replaced by a_i and a_j.

5.4.0.2 Reduction Structure

Given an instance of the ECR problem, each element a of U is given an arbitrary label i where $i \in \{1, \ldots, |U|\}$ and is partitioned according to its colour j where $j \in \{1, \ldots, 23\}$. A function $f : \mathcal{U} \to \mathbb{N}$ maps each element a of the universe $\mathcal{U}$ of the input Exact Cover Reconfiguration problem to a positive integer. The positive integer is computed using the encoding of the label i and colour j of an element a. The function f maps a colour-j

element a_i to $i.2^{100j\lceil log_2(|U|)\rceil}$. In binary, this mapping consists of the binary encoding of i followed by $100j\lceil log_2(|U|)\rceil$ zeros. The idea here is to use j to create an interval for elements of U have the same colour or are part of the same colour class and i as an offset to differentiate elements of the same colour. The reduction structure is explained with an illustrative example in example 5.4.2.

5.4.0.3 Output $\mathcal{S}$ and x

The numbers in the output 3−move Subset Sum Reconfiguration instances are $\{\sum_{a \in S} f(a) : S \in \mathcal{S}\}$ and the output target sum is $\sum_{a \in \mathcal{U}} f(a)$.

5.4.0.4 Correctness

A reconfiguration in both the ECR problem and 3-move SUBSET SUM RECONFIGURATION problem involves splitting or merging elements. Thus it suffices to prove that the function f yields a one-to-one mapping $g : \mathcal{S} \to N$ given by $g(\mathcal{S}) = \sum_{a \in \mathcal{S}} f(a)$.

Recall that the function f maps each element $a_i \in U$ to a value based upon the colour of a_i. The sums of the outputs of f for all elements of all colours 1 to $j-1$ is at most $2^{100(j-1)log_2(|U|)}.|U|^2 \leq 2^{(100j-98)log_2(|U|)}$ while the output of f for any element of any colour j or larger is at least $2^{100(j)log_2(|U|)} \geq 2^{(98)}.2^{(100j-98)log_2(|U|)}$.

Thus if a pair of sets $\mathcal{S}_1, \mathcal{S}_2 \subseteq S$ have $\mathcal{S}_1 \neq \mathcal{S}_2$, then their colour-$j$ elements differ, this difference cannot be made up by adding or removing elements of colours 1 to $j-1$ (values too small) or colours $j+1$ to 23 (values too large). Thus if $\mathcal{S}_1 \neq \mathcal{S}_2$, then $g(\mathcal{S}_1) \neq g(\mathcal{S}_2)$. □

Example 5.4.2. Concerning this proof, in order to illustrate the encoding used a new smaller example is used. Just to demonstrate our purpose we will consider that we have only 5 colours and only $|\mathcal{S}| = 4$.

The set of subsets of $\mathcal{U}$

$$\mathcal{S} = \{S_1, S_2, S_3, S_4\} \text{ where :}$$
$$S_1 = \{a_1, a_3, a_4\}$$
$$S_2 = \{a_2, a_5, a_6, a_7\}$$
$$S_3 = \{a_1, a_3, a_2, a_6\}$$
$$S_4 = \{a_1, a_2, a_3, a_4, a_5, a_6, a_7\}$$

Partitioning of each element s_i according to their colour j.

colours, $j =$:	4	3	2	1	0
	a_1	a_3	a_4		
		a_5	a_2	a_6	a_7

Mapping of each element s_i to a positive integer according to their label and colour.

$$f(a_1) = 10000$$
$$f(a_2) = 200$$
$$f(a_3) = 3000$$
$$f(a_4) = 400$$
$$f(a_5) = 5000$$
$$f(a_6) = 60$$
$$f(a_7) = 7$$

Output $\mathcal{S}$ and x.

$$S_1 = 10000 + 3000 + 400$$
$$S_2 = 200 + 5000 + 60 + 70$$
$$S_3 = 10000 + 3000 + 200 + 60$$
$$S_4 = 10000 + 200 + 3000 + 400 + 5000 + 60 + 7$$

The numbers are $\mathcal{S} = \{13400, 05267, 13260, 18667\}$
The output target sum $x = \sum_{a \in U} f(a) = 18667$

An element T in $\mathcal{S}$ can be split in T_1 and T_2 of $\mathcal{S}$ if and only if $g(T) = g(T_1) + g(T_2)$.

$$\begin{array}{ccc} \boxed{\{S_4\}} & & \boxed{g(S_4)} \\ \Big\downarrow \text{split} & \overset{\text{iff}}{\Longleftrightarrow} & \Big\downarrow \overset{?}{=} \\ \boxed{\{S_1\}, \{S_2\}} & & \boxed{g(S_1) + g(S_2)} \end{array}$$

(A) (B)

FIGURE 5.6: Reduction structure.

5.5 Geometric interpretations

In this section, we will view the k-move SUBSET SUM RECONFIGURATION problem from a geometric point of view. We will first start by defining the notion of a n-hypercube, Constrained Hypercube Path problem and the Knapsack reconfiguration problem below :

The n-hypercube is the graph with vertex set $\{0, 1\}^n$ such that two vertices are adjacent whenever their coordinates differ by exactly one component. In this section, we consider the following abstraction of reconfiguration problems involving subsets.

Definition 5.5.1. (Constrained Hypercube Path). Given two vertices s, t of the n-hypercube, both contained in a polytope $P := \{x \in \mathbb{R}^n : Ax \leq b\}$ for some $A = (a_{ij}) \in \mathbb{Z}^{d \times n}$ and $b \in \mathbb{Z}^d$, does there exist a path from s to t in the hypercube, all vertices of which lie in P?

Definition 5.5.2. (Knapsack (decision) Problem). Given integers l and u and two sets of integers $S = \{a_1, a_2, \ldots, a_n\}$ and $W = \{w_1, w_2, \ldots, w_n\}$, does there exist a subset $A \subseteq [n]$ such that $\sum_{i \in A} a_i \geq l$ and $\sum_{i \in A} w_i \leq u$?

Definition 5.5.3. (Knapsack Reconfiguration Problem). Given two solutions A_1 and A_2 to an instance of the knapsack problem, can A_2 be obtained by repeated $1-$move reconfiguration, beginning with A_i, so that all intermediate subsets are also solutions ?

Notice that the problem defined by Ito and Demaine is the knapsack reconfiguration problem where $S = W$, the capacity c of the knapsack is the integer u, the given threshold k is the integer l and the size of each item a_i of

A is the weight w_i in W. Given the definition of the Constraint Hypercube Path problem above, we can argue that the knapsack reconfiguration where $S = W$ is equivalent to the Constrained Hypercube path problem where $d = 2$ since it involves exactly two linear constraints. The equivalence and geometric interpretation is shown in section 5.5.1.

5.5.1 Knapsack reconfiguration where $S = W \to$ Constrained Hypercube Path problem

Let u be the capacity of the knapsack, l be the given integer threshold, $W = \{w_1, w_2, \ldots, w_n\}$ where w_i is the weight of the item i of the knapsack and $x \subseteq \{0,1\}^n$ be a Boolean variable that indicates whether an item is chosen. The solution space of this given instance of the subset sum reconfiguration problem is characterized by an n-hypercube since in an n-hypercube the vertices are adjacent whenever their coordinates differ by exactly one component corresponding to the addition or removal of an item. The solutions to this given instance is the points of the n-hypercube that lies in the polytope $P := \{l \leq \sum_{i=1}^{n} x_i w_i \leq u\}$.

Example 5.5.4. The following example shows the geometric interpretation of an instance denoted I of the Knapsack reconfiguration problem where $\mathcal{W} = \{1, 3, 6\}$, $l = 1$ and $u = 7$. The n-hypercube induced by all possible configurations of I is shown in blue in figure 5.7, the two linear upper and lower bound constraints are represented by the yellow and orange hyperplanes respectively. It is very interesting to notice that the feasible configurations to the given instance are the points lying between the yellow and orange hyperplanes and the red path represent the reconfiguration sequence between the feasible solutions. Notice that this path is also included in the polytope P delimited by the hyperplanes.

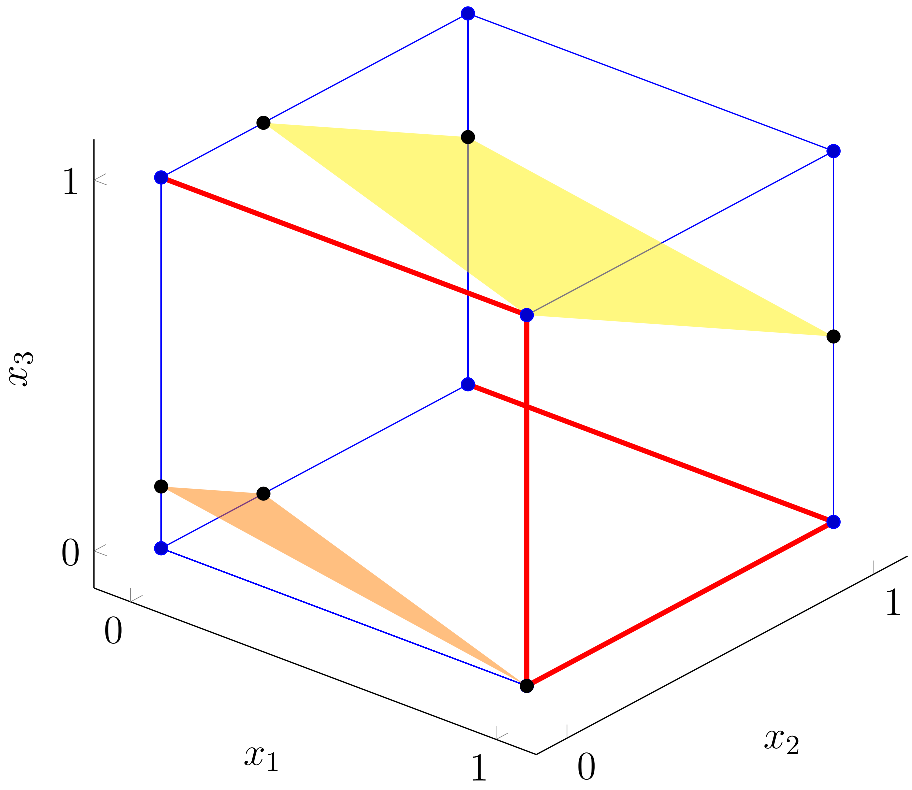


FIGURE 5.7: Geometric visualisation of a given knapsack reconfiguration problem.

5.5.2 3-move Subset sum reconfiguration → Constrained Hypercube Path problem

Let $A = \{a_1, a_2, \ldots, a_n\}$ be the set of integers, x be the target sum and $x \subseteq \{0,1\}^n$ be a Boolean variable that indicates whether an integer a_i is chosen. The solution space of this given instance is characterized by an n-hypercube where the vertices are adjacent whenever the symmetric difference of their coordinates equals k corresponding to replacing elements a_i and a_j by $a_i + a_j$ or $a_i + a_j$ by a_i and a_j. The solutions to this given instance are the points that lies in the polytope $P := \{\sum_{i=1}^{n} x_i a_i = x\}$.

Example 5.5.5. The following example shows the geometric interpretation of an instance denoted I of the 3-move subset sum reconfiguration problem where $\mathcal{S} = \{2, 3, 5\}$ and $x = 5$. The n-hypercube induced by all possible configurations of I denoted Q is shown in dotted blue in figure 5.7. The hyperplane representing the target sum constraint is represented in yellow.

The solutions to this given instance is the points of Q that lies in the hyperplane. The red path represents the reconfiguration sequence between the feasible solutions.

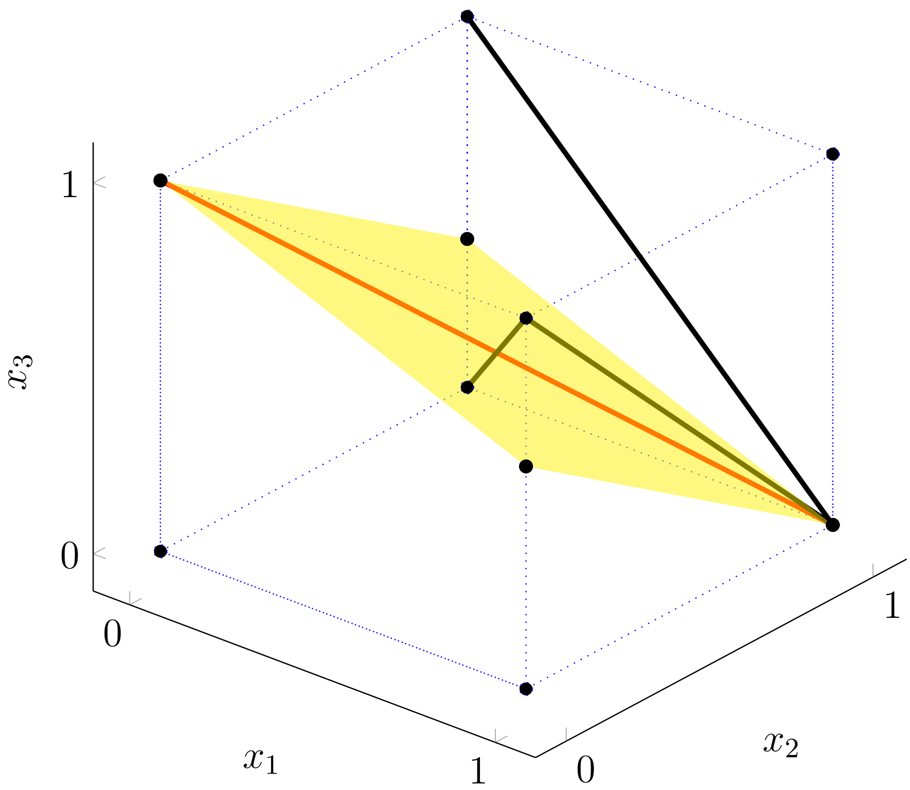


FIGURE 5.8: Geometric visualisation of a given instance of the 3-move SUBSET RECONFIGURATION problem.

Chapter 6

Conclusions and Future Works

Throughout this thesis, we have researched and analysed reconfiguration problems by starting with the Nondeterministic Constraint Logic which seems to be the go-to reduction in order to prove PSPACE-hardness results. More specifically, we focused on the alternative formulation of NCL and detailed the PSPACE-completeness result of the sliding token problem. We then showed that the labelled variant of the SLIDING TOKEN problem is also PSPACE-complete. This latter result was then used to establish the complexity result of the Exact Cover merge and split reconfiguration problem which was then used to prove the hardness of the k-move Subset sum reconfiguration problem for $k = 3$. As a contribution to this part, we provided detailed explanatory examples and clarifications to complement the proof. Every reconfiguration problems we encountered during our journey to the completion of this thesis is summed in fig 6.1.

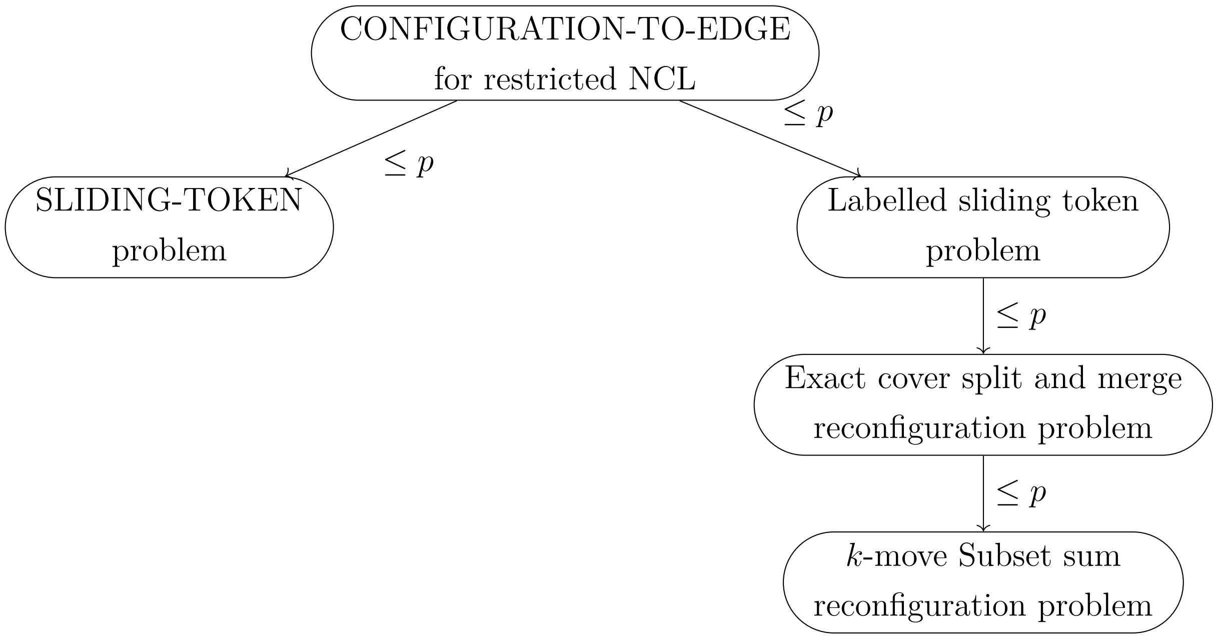
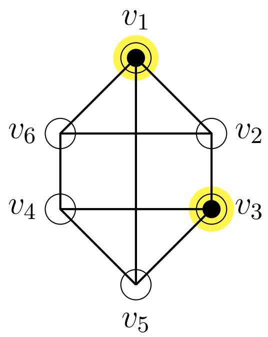


FIGURE 6.1: PSPACE-complete problems encountered and their relationship.

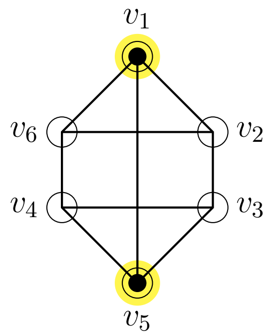
6.1 Future works

6.1.1 SLIDING TOKENS problem

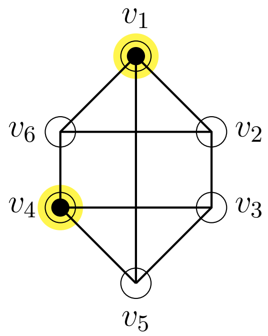
Given a SLIDING TOKENS instance and two configurations C_1 and C_2, there may be several reconfiguration sequences to transform the given initial configuration C_1 into C_2 (see figures 6.2 and 6.3). Thus, the question of finding the Shortest reconfiguration path in the SLIDING TOKENS problem naturally arised.



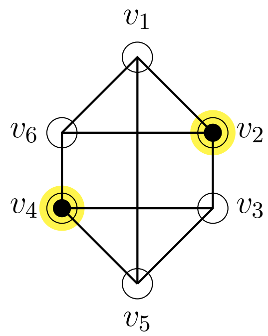
(A) Initial Independent Set $I_1 = \{v_1, v_3\}$



(B) Intermediate Independent Set

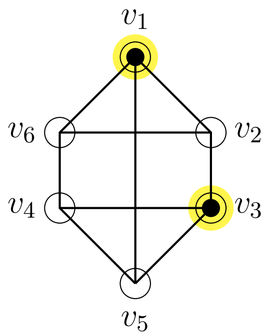


(C) Target Independent Set $I_2 = \{v_2, v_4\}$

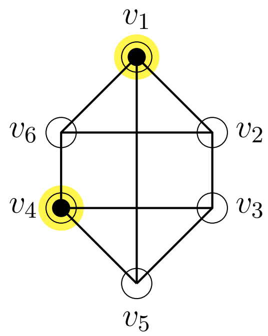


(D) A 3-regular graph $G = (V, E)$ and initial independent set $I_1 = \{v_1, v_3\}$

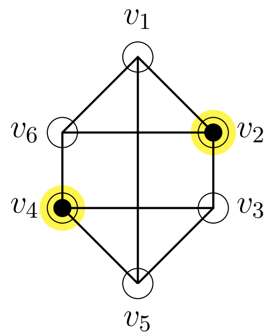
FIGURE 6.2: Configuration-to-edge input instance



(A) Initial Independent Set $I_1 = \{v_1, v_3\}$



(B) Intermediate Independent Set



(C) Target Independent Set $I_2 = \{v_2, v_4\}$

FIGURE 6.3: Configuration-to-edge input instance

However this question has already been asked by Yamada and Uehara in 2016, defined below and remains open. The difficulty lies in computing the reconfiguration graph.

Shortest Sliding Token [Yamada and Uehara 2016]

Instance: A yes-instance (G, I, J) of Sliding Token, where I, J are independent sets of a graph G.
Question: Find a shortest TS-sequence that transforms I into J (and vice versa)

On the bright side, in the past years positive results have been established for cographs, interval graphs, caterpillars, trees, prefect graphs and spider graphs. Those interesting results are summed up below.

Theorem 6.1.1. *(Kaminski et al. 2012) It is* NP*-complete to decide if there is a TS-sequence having at most l token-slides between two independent sets I, J of a perfect graph G even when l is polynomial in $|V(G)|$.*

Theorem 6.1.2. *(Kaminski et al. 2012) Shortest Sliding Token can be solved in linear time for cographs (P4-free graphs).*

Theorem 6.1.3. *(Yamada and Uehara 2016) Shortest Sliding Token can be solved in polynomial time for proper interval graphs, trivially perfect graphs, and caterpillars.*

Theorem 6.1.4. *(Sugimori, AAAC 2018) Shortest Sliding Token can be solved in $O(poly(n))$ time when the input graph is a tree on n vertices.*

Theorem 6.1.5. *(Ryuhei Uehara, CIAC 2019) Shortest Sliding Token can be solved in $O(n^2)$ time when the input graph is a spider (i.e., a tree having exactly one vertex of degree at least 3) on n vertices.*

6.1.2 3-move SUBSET SUM RECONFIGURATION problem

Recall that we studied the following problem in chapter 5 :

Definition 6.1.6. (k-move SUBSET SUM RECONFIGURATION Problem). Given two solutions A_1 and A_2 to an instance of the subset sum problem, can A_2 be obtained by repeated k-move reconfiguration, beginning with A_1, so that all intermediate subsets are also solutions?

It would be interesting to study the connectivity question of the k-move SUBSET SUM RECONFIGURATION problem defined below :

Definition 6.1.7. (k-move SUBSET SUM RECONFIGURATION Problem conn). Given an integer x and a set of integers $S = \{a_1, a_2, \ldots, a_n\}$, let $G(S)$ be the subgraph graph induced by the feasible solutions of S where there is an edge between two vertices of $G(S)$ iff the symmetric difference is at most k. Is $G(S)$ connected ?

Another question concerns finding an optimal coloring with k colors that is smaller than 23 for the ECR problem.

Bibliography

[1] Eric Allender, Michael Bauland, Neil Immerman, Henning Schnoor, and Heribert Vollmer. The Complexity of Satisfiability Problems: Refining Schaefer's Theorem. page 12.

[2] Paul Bonsma and Luis Cereceda. Finding paths between graph colourings: Pspace-completeness and superpolynomial distances. *Theor. Comput. Sci.*, 410:5215–5226, 01 2009.

[3] Paul Bonsma, Luis Cereceda, Jan van den Heuvel, and Matthew Johnson. Finding paths between graph colourings: Computational complexity and possible distances. *Electronic Notes in Discrete Mathematics*, 29:463–469, 08 2007.

[4] Paul Bonsma, Marcin Kamiński, and Marcin Wrochna. Reconfiguring independent sets in claw-free graphs.

[5] Paul S. Bonsma. Shortest path reconfiguration is pspace-hard. *CoRR*, abs/1009.3217, 2010.

[6] Andrei A. Bulatov. A dichotomy theorem for constraint satisfaction problems on a 3-element set. *J. ACM*, 53(1):66–120, January 2006.

[7] 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.

[8] Luis Cereceda, Jan van den Heuvel, and Matthew Johnson. Connectedness of the graph of vertex-colourings. *Discrete Mathematics*, 308:913–919, 03 2008.

[9] N. Creignou. A dichotomy theorem for maximum generalized satisfiability problems. *Journal of Computer and System Sciences*, 51(3):511 – 522, 1995.

[10] Erik D. Demaine, Martin L. Demaine, Eli Fox-Epstein, Duc A. Hoang, Takehiro Ito, Hirotaka Ono, Yota Otachi, Ryuhei Uehara, and Takeshi Yamada. Linear-Time Algorithm for Sliding Tokens on Trees. *arXiv e-prints*, page arXiv:1406.6576, June 2014.

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

[12] Eli Fox-Epstein, Duc A. Hoang, Yota Otachi, and Ryuhei Uehara. Sliding token on bipartite permutation graphs. In Khaled Elbassioni and Kazuhisa Makino, editors, *Algorithms and Computation*, pages 237–247. Springer Berlin Heidelberg.

[13] 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.

[14] Parikshit Gopalan, Phokion G. Kolaitis, Elitza N. Maneva, and Christos H. Papadimitriou. The connectivity of boolean satisfiability: Computational and structural dichotomies. *SIAM J. Comput.*, 38(6):2330–2355, 2009.

[15] Robert A. Hearn and Erik D. Demaine. PSPACE-completeness of sliding-block puzzles and other problems through the nondeterministic constraint logic model of computation.

[16] Robert Aubrey Hearn. *Games, Puzzles, and Computation.* PhD thesis, USA, 2006. AAI0810083.

[17] Edward Hordern. *Sliding piece puzzles.* Oxford University Press. OCLC: 14100017.

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

[19] Takehiro Ito and Erik D. Demaine. Approximability of the subset sum reconfiguration problem, 2011.

[20] 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. *Theor. Comput. Sci.*, 412(12-14):1054–1065, 2011.

[21] Takehiro Ito, Marcin Kamiński, and Erik D. Demaine. Reconfiguration of list edge-colorings in a graph. In Frank Dehne, Marina Gavrilova, Jörg-Rüdiger Sack, and Csaba D. Tóth, editors, *Algorithms and Data Structures*, pages 375–386. Springer Berlin Heidelberg.

[22] Matthew Johnson, Luis Cereceda, and Jan van den Heuvel. Finding paths between 3-colourings. In Mirka Miller and Koichi Wada, editors, *Proceedings of the 19th International Workshop on Combinatorial Algorithms, IWOCA 2008, September 13-15, 2008, Nagoya, Japan*, pages 182–196. College Publications, 2008.

[23] Marcin Kaminski, Paul Medvedev, and Martin Milanic. Shortest paths between shortest paths. *Theor. Comput. Sci.*, 412:5205–5210, 09 2011.

[24] Marcin Kamiński, Paul Medvedev, and Martin Milanič. Complexity of independent set reconfigurability problems. 439:9–15.

[25] 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.

[26] Dimitris Kavvadias and Martha Sideri. THE INVERSE SATISFIABILITY PROBLEM. *SIAM Journal on Computing*, 28:12.

[27] Sanjeev Khanna, Madhu Sudan, Luca Trevisan, and David P. Williamson. The Approximability of Constraint Satisfaction Problems. *SIAM Journal on Computing*, 30(6):1863–1920, January 2001.

[28] Lefteris M. Kirousis and Phokion G. Kolaitis. The complexity of minimal satisfiability problems. *Information and Computation*, 187(1):20 – 39, 2003.

[29] Richard E. Ladner. On the structure of polynomial time reducibility. *J. ACM*, 22(1):155–171, January 1975.

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

[31] Amer E. Mouawad, Naomi Nishimura, Venkatesh Raman, and Marcin Wrochna. Reconfiguration over tree decompositions.

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

[33] Walter J. Savitch. Relationships between nondeterministic and deterministic tape complexities. 4(2):177–192.

[34] 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.

[35] Konrad W. Schwerdtfeger. A Computational Trichotomy for Connectivity of Boolean Satisfiability. *arXiv:1312.4524 [cs]*, October 2015. arXiv: 1312.4524.

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

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

[38] Tom C. van der Zanden. Parameterized complexity of graph constraint logic.