Université libre de Bruxelles

Master Thesis

Reconfiguration Problems

Author:
Prateeba Ruggoo

Supervisor:
Dr. Jean Cardinal

Department of Computer Science

August 14, 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. The solution space of reconfiguration problems can be represented as a reconfiguration graph, where two vertices representing solutions are adjacent if one can be formed from the other in a single step. Work in the area encompasses both structural questions (Is the reconfiguration graph connected?, Is there a path between a node s and a node t in the reconfiguration graph?) and algorithmic ones (How can one find the shortest sequence of steps between two solutions?) 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 and present an 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 boolean Satisfiability reconfiguration problems and investigate dichotomies/tricotomies that have been established and their applications. We also focus on complementing some recent PSPACE−hardness proofs given in [6] concerning Subset sum reconfiguration problem.

Acknowledgements

First and foremost, my utmost gratitude to prof. Jean cardinal, whose patience and encouragement I will never forget. I am indebted to my parents who have given me the opportunity of an education and support throughout my life. Thanks also to my partner for providing guidance and a sounding board when required.

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 combinatorial problems in which we are given a collection of configurations, together with some transformation rule(s) that allows us to change one configuration to another such that each intermediate solution remains satisfiable at all times. A configuration can be the arrangement of puzzle pieces, the location of a robot with respect to obstacles in space, or the ordering of symbols to form a string. Combinatorial 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 combinatorial 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 coloring [2], [3], [7], list-edge coloring [19], clique, set cover, integer programming, matching, spanning tree, matroid bases [18], block puzzles [13], shortest path [21], independent set [13],[18], [22], and satisfiability

[12]. Many problems in P have their reconfigurability 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 [18]. In general however, knowing the complexity of a decision problem does not allow us to directly infer the complexity status of its reconfigurability problem(s). Several NP-complete problems have reconfigurability analogues that are in P, for example the 3-colorability problem [20]. Alternatively, some problems in P have reconfigurability versions that are PSPACE-complete, such as shortest paths [5] or the problem of deciding whether two 4-colorings 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 [6]. More precisely, we go over and detail the PSPACE−completeness of the following decision problems :

- Given two subsets of a set S of integers 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 ?
- In the process of complementing the 3-move subset sum reconfiguration problem, we also give a simple hardness proof the labelled variant of the sliding token problem, described in section 3.5.

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 in recreational mathematics.

The second part of this thesis focuses on Satisfiability reconfiguration problems and the Subset sum reconfiguration problem and provides a visual support for the latter problem with the intent on helping on having a better idea of the reconfiguration graph and it's connectivity properties. Finally, in

Chapter 7 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 *neighborhood* 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 $v_0, \ldots, v_l$, such that for all $i \in \{0, \ldots, l-1\}$, $v_i v_{i+1} \in E(G)$. 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.

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 $\{1, 2, \ldots, n\}$ (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 $\mathcal{S}^*$. For an in-depth review of general graph theoretic definitions, the reader can refer to Diestel's textbook [9].

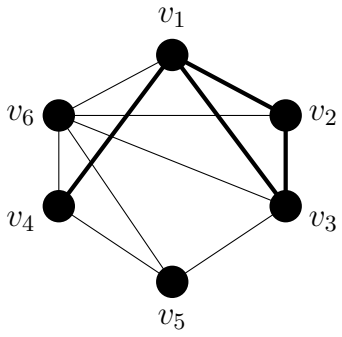


FIGURE 2.1: Graph $G^{'}$ (shown darker) is a subgraph of G.

2.1.2 Graph colouring

In graph theory, a *graph colouring* is a special case of *graph labeling* as it is an assignment of labels traditionally called "colours" to elements 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 labeling of the graph's vertices with colours such that no two adjacent vertices have the same color.

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 ϕ from $V(G)$ to $\{1, 2, \ldots, k\}$ (whose elements are called *colours*)such that no two adjacent vertices receive the same colour, that is, $\phi(v) \neq \phi(v^{'})$ for all $v, v^{'} \in V(G)$ where $v \neq v^{'}$. The *chromatic number* of a graph G, denoted χG, 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-colorable*, 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-coloring is the same as a partition of the vertex set into k independent sets.

2.2 Computational Complexity Theory

Computer 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.

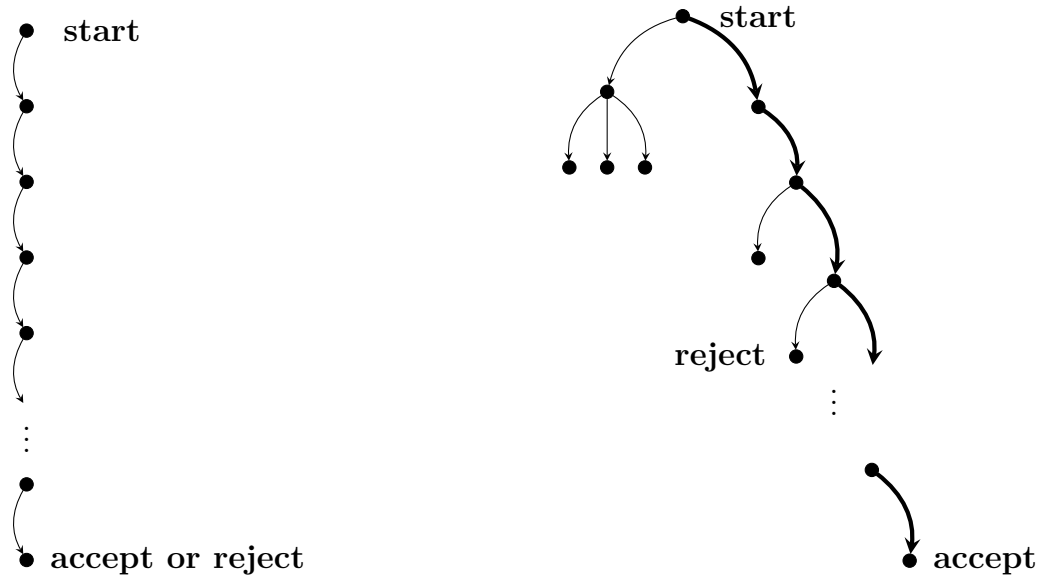
The model of computation focused on in standard complexity theory is the *Turing Machine*. It uses an unlimited tape as its unlimited memory and has a tape head that can write and read symbols and move along the tape. A Turing Machine can be viewed as an automaton, following simple rules to change states, 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 Turing Machine that can choose which moves to take in order to reach an accepting state is known as a *Nondeterministic Turing Machine* 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 [29]. The following section describes the different computational complexity classes in which different decision problems fits.

In general, these complexity classes study how the critical ressources (*time* and *space*) grows in terms if the input size. For decision problems, the input is the description of the problem and its size measured as bits of information.



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

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

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.

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.

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 extremly lucky: whenever it needs to make a decision, it by defintion 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.

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.

Relationship of Complexity Classes

[14] So far we have that $\mathsf{P} \subseteq \mathsf{PSPACE} \subseteq \mathsf{EXP}$. $\mathsf{PSPACE} = \mathsf{NPSPACE}$ follows from the celebrated result of Savitch [27]. 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 $\mathsf{P} = NP$ 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.

2.4 Set theory

The *symmetric difference* of two sets A and B are elements in A or B, but not in both A and B and written as $A \triangle B$.

Chapter 3

The Nondeterministic Constraint Logic (NCL)

In this Chapter, the Nondeterministic Constraint Logic model of computation is presented. This framework developped by Demaine and Hearn is motivated by the Sliding-block puzzles [15]. The main result of [13] 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 surrounding the NCL framework are proved to be PSPACE-complete [13]. These decision problems are then used to prove the PSPACE-completeness of well-known Sliding-block puzzles such as Rush Hour and Sokoban [14]. 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 [13] 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 and 3.5.3 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

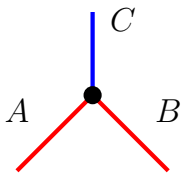
some complexity results of decision problems stemming from the constraint logic framework. In section 3.4 we detail the PSPACE-completeness proof of the sliding token problem using the alternative formulation of NCL (theorem 3.4.2). Lastly, in section 3.5 the hardness proof of the labeled variant of the sliding token problem is given.

3.1 Graph Formulation

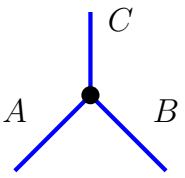
An NCL machine consists of a *constraint graph*, $G = (V, E)$ that we can think of as our computation model. Let G be a 3-regular graph with edge weights $\in \{1, 2\}$. An edge is then called *red* or *blue*, respectively. Each vertex has a nonnegative *minimum inflow* which is the sum of the weights on inward-directed edges. A *legal configuration* is an assignment of an *orientation(direction)* to each edge such that for every vertex v of G, the sum of weights of inward-directed edges of v is at least 2. A *legal move* is the reversal of a single edge that results in another legal configuration.

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 $\in \{1, 2\}$ 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 [14] 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[14].

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[14].

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. 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 [14].

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 [14]. 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[14]. Additionally, the constraint graphs can be required to have bounded bandwidth, and the problems on them will still remain PSPACE-complete[31].

3.4 Alternative formulation : Sliding tokens

The SLIDING TOKEN problem was introduced by Hearn and Demaine in [13] as a variant of SLIDING-BLOCK puzzle with 1×1 blocks on a graph but require 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 $I_b \overset{G}{\rightsquigarrow} I_r$. The sets I_b and I_r are the *initial* and *target* independent sets, respectively. For a TS-sequence S, the *length* len(S)of S is defined as the number of independent sets in S minus one. In other words, 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 $I_r = I_5$.

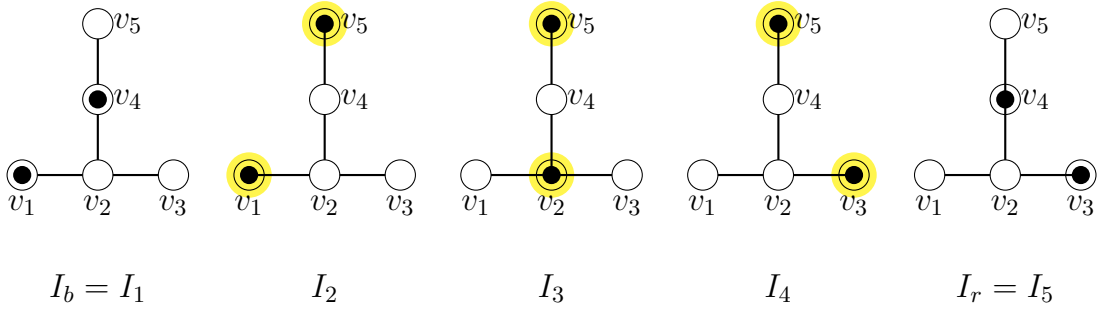


FIGURE 3.2: TS sequence $\langle I_1, I_2, \ldots, I_5 \rangle$ of independent sets which transforms $I_b = I_1$ into $I_r = I_5$ where the vertices in independent sets are depicted by small black circles (tokens).

3.4.1 Known results for the SLIDING TOKEN problem.

Analogous to the Independent Set problem being the key problem among thousands of NP-complete problems to prove NP-hardness, the SLIDING TOKEN problem plays an important role since several PSPACE-hardness results have been proved using reductions from it.

For the sliding token problem, some polynomial time algorithms have been investigated as follows: Linear time algorithms have been shown for cographs (also known as P4-free graphs) [22] and trees [8]. Polynomial time algorithms are shown for bipartite permutation graphs [10], and claw-free graphs [4]. On the other hand, PSPACE-completeness is shown for graphs of bounded treewidth [25], and planar graphs [13].

3.4.2 PSPACE-completeness.

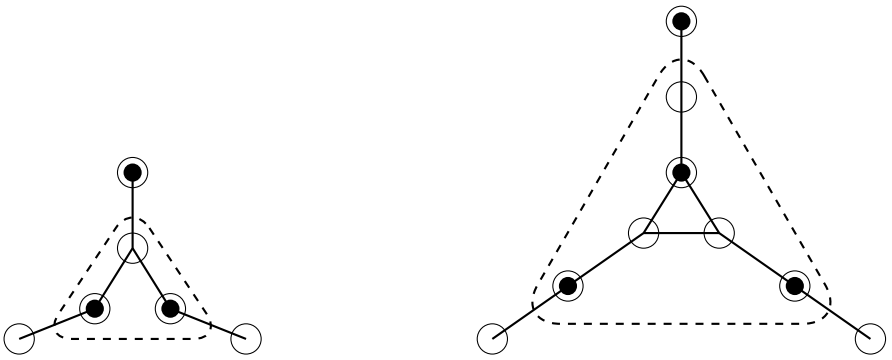
In this section we go over the PSPACE-completeness result of the SLIDING TOKEN problem, proved by a reduction from NCL. As seen in section 3.3, there are slightly different versions of decision problems for NCL and all of them are PSPACE−complete. For our purpose, we just need the version for the configuration-to-configuration for planar NCL. Recall that an instance of the C2C 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 is organised in sections 3.4.2 to 3.4.2 which explains the reduction structure, gadets used and how they are connected together.

Reduction structure.

To show that the SLIDING TOKEN problem is PSPACE-complete we provide a reduction from configuration-to-configuration for AND/OR graphs such that the NCL instance is solvable if and only if the corresponding SLIDING TOKEN is solvable. The sliding token 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-token subgraphs illustrated in figures 3.3a and 3.3b respectively.

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.

AND/OR Graphs

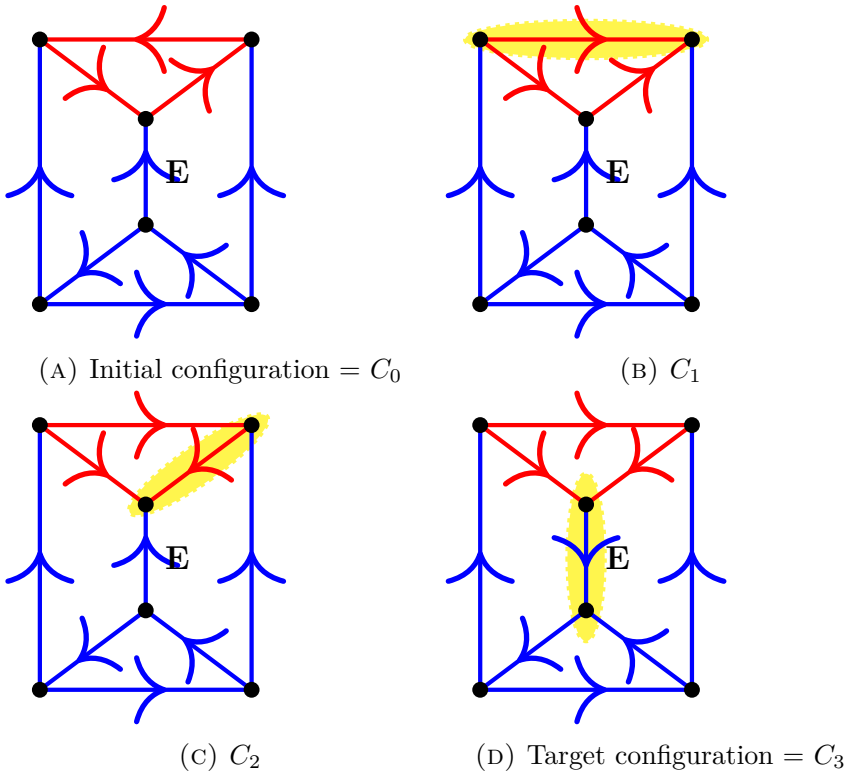
We showed how to construct AND and OR vertices. We now show how to connect the vertices into an arbitrary planar constraint graph. First, the edges that cross the dotted-line gadget borders are called "port" edges. A token on an outer port-edge vertex represents an inward-directed NCL edge, and vice-versa. 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-token graph by joining together AND and OR vertex gadgets at their shared port edges, placing the port tokens appropriately.

Theorem 3.4.1. *Sliding Token problem is* PSPACE*-complete.*

Proof. First, we show that SLIDING TOKEN problem is in PSPACE. The SLIDING TOKEN 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 TOKEN is in NPSPACE. By Savitch's celebrated theorem, we have that NPSPACE = PSPACE [27], implying that SLIDING TOKEN is in PSPACE.

The SLIDING TOKEN 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 TOKEN is solvable. □

Example 3.4.2. Find a name.
Input restricted NCL C2E instance



(A) Initial configuration = C_0 (B) C_1

(C) C_2 (D) Target configuration = C_3

FIGURE 3.4: C2E input instance

Output Sliding-Token instance

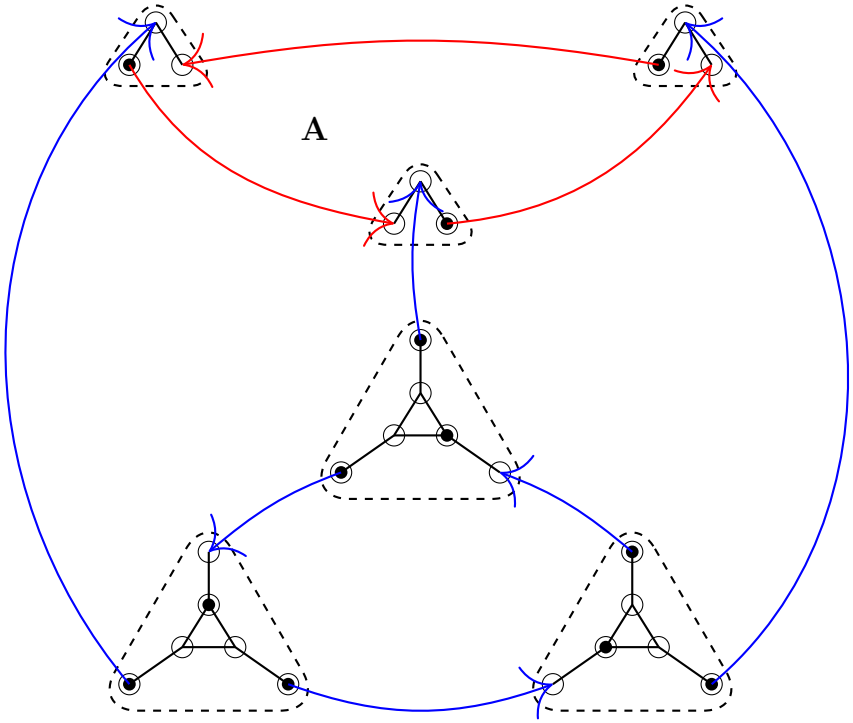


FIGURE 3.5: Sliding Tokens vertex gadgets.

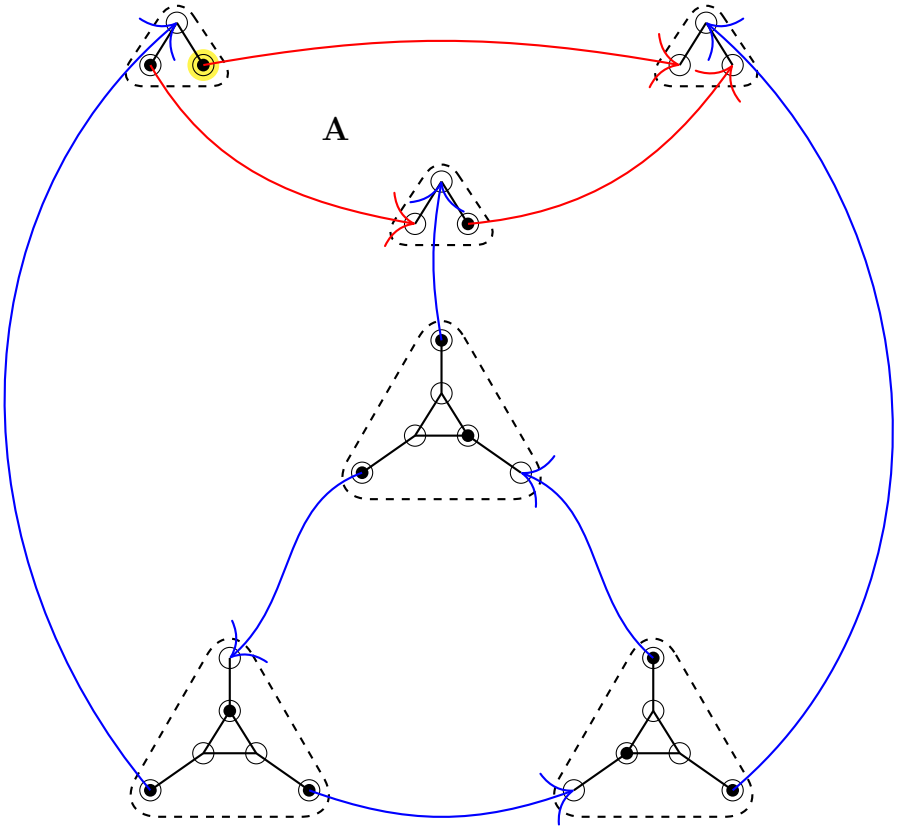


FIGURE 3.6: Sliding Tokens vertex gadgets.

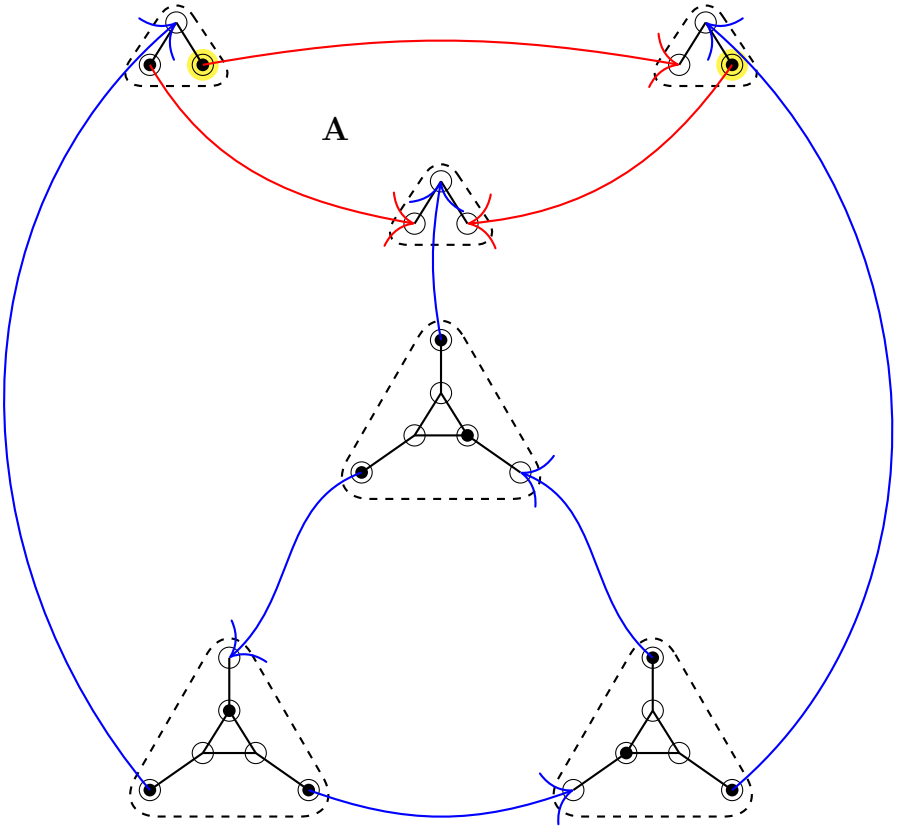


FIGURE 3.7: Sliding Tokens vertex gadgets.

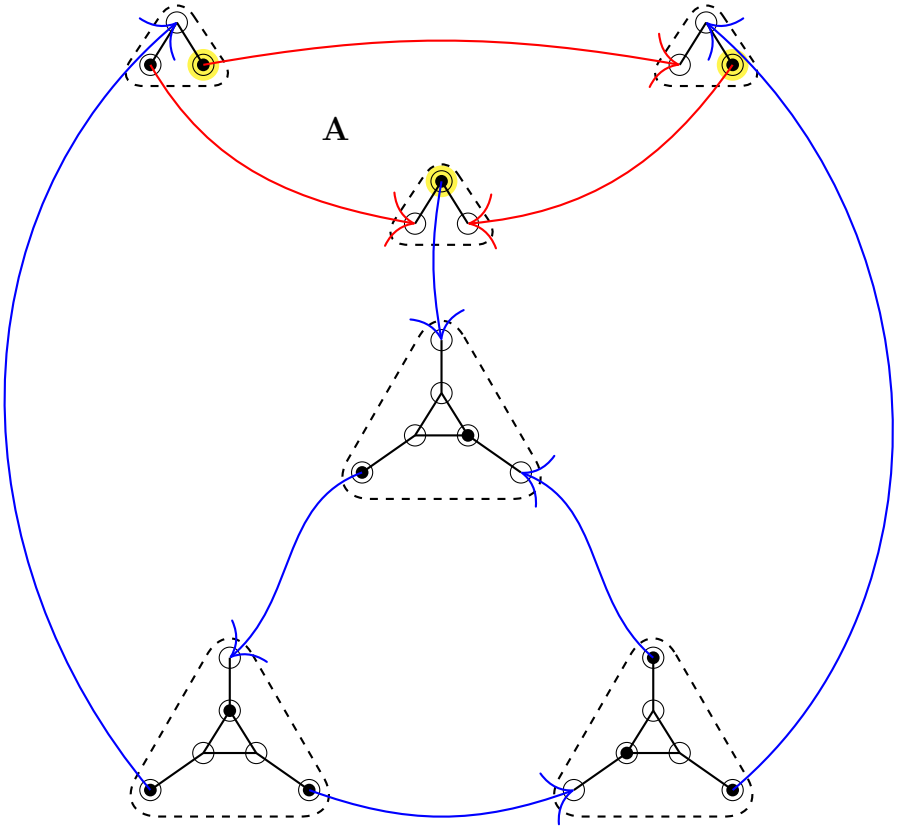


FIGURE 3.8: Sliding Tokens vertex gadgets.

3.5 Labelled variant of Sliding-Token Problem

The labelled sliding token problem is a variant of the Sliding token problem where each token has a unique label. The purpose of this section is to prove that the sliding token problem remains PSPACE-hard even with labeled tokens. 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 (described in section 3.5.1) is then used to establish the hardness of the k-COLOUR PATH problem. To achieve our goal only a slight modification of that proof is needed. For the sake of completeness we will go through the original proof and add the modification needed. On our journey we will encouter some interesting graph colouring problems described in sections 3.5.2.

3.5.1 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 (i.e., copies of K_2). 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 3.9.

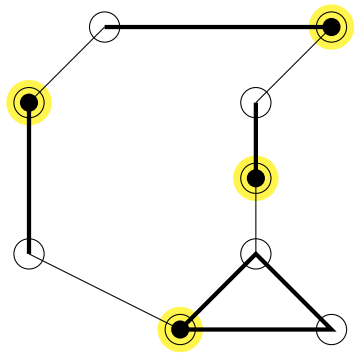


FIGURE 3.9: An example of a restricted instance graph G together with a standard token configuration.

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 labeled variant while doing only a little modification in the original proof.

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.

redo this part

3.5.2 Labelled tokens

In graph theory, graph coloring is a special case of graph labeling; it is an assignment of labels traditionally called "colors" to elements of a graph subject to certain constraints. In its simplest form, it is a way of coloring the vertices of a graph such that no two adjacent vertices are of the same color; this is called a vertex coloring. Two prominent vertex coulouring problems that will help achieve our goal are defined hereunder :

k-colouring.

For a positive integer k and a graph G, we define the k-colour graph of G, denoted $\mathcal{C}_k(G)$, as the graph that has the k-colourings of G as its node set, with two k-colourings joined by an edge in $\mathcal{C}_k(G)$ if they differ in colour on just one vertex of G. The k-COLOUR PATH is then defined as follows :

k-COLOUR PATH
Instance: Graph G, two k-colourings of G, α and β.
Question: Is there a path between α and β in $\mathcal{C}_k(G)$?

List colouring

Suppose that we are given a graph $G = (V, E)$, and for each vertex of G, a list of colours permitted at that particular vertex. In this context, the LIST-COLOUR PATH is defined as follows :

LIST-COLOUR PATH
Instance: Graph G, colour lists $L(v) \subseteq \{1, 2, 3, 4\} \forall v \in V(G)$, two list-colourings α and β.
Question: Is there a path between α and β in $\mathcal{C}(G, L)$?

The goal now is to prove that if the Standard Sliding token problem is k colourabable *i.e.*, labeled with k-colours, then having labeled token should not be a bother. In [2] Bonsma established that k-Colour Path is PSPACE-complete for several graph classes and values of $k \geq 4$. This result was obtained through a reduction from the Standard Sliding Tokens. In order to facilitate the reduction, a first reduction was done to the List-Colour path problem.

In [2] Bonsma introduced a lemma that makes going back to the k-colour path easier.

3.5.3 Reduction Structure

Given a restricted instance G, T_A, T_B, T of the Standard Sliding Token problem where T is a set of labelled token, we construct an instance $G^{'}, L, \alpha, \beta, T$ of LIST-COLOUR Path such that standard token configurations of G correspond to list-colourings of $G^{'}$, and sliding a token in G corresponds to a sequence of vertex recolourings in $G^{'}$.

We first start by labelling the vertices of G:

1. The token triangles are labelled $1, \ldots, n_t$, and the vertices of each triangle i are labelled t_{i1}, t_{i2} and t_{i3}.

2. The token edges are labelled $1, \ldots, n_e$, and the vertices of each token edge i are labelled e_{i1} and e_{i2}.

The construction of $G^{'}$ is as follows: For every token triangle i in G we introduce a vertex t_i, with colour list $L(t_i) = \{1, 2, 3\}$ in $G^{'}$. For every token edge i in G we introduce a vertex e_i in $G^{'}$, with colour list $L(e_i) = \{1, 2\}$. Whenever a link edge of G joins a vertex t_{ia} with a vertex e_{jb}, we add an (a, b)-forbidding path of even length between t_i and e_j in G . We do the same for pairs t_{ia} and t_{jb}, and pairs e_{ia} and e_{jb}. Linking the vertices in $G^{'}$ through (a, b)-forbidding paths will make sure that no tokens are adjacent to each other. Note that this is a polynomial-time transformation.

Standard token configurations of G now correspond to colourings of $G^{'}$ as follows:

1. For each token edge i of the token configuration, the token being on $e_{ij} (j = 1, 2)$ corresponds to colourings of G where e_i has colour j.

2. For each token triangle i of the token configuration, the token being on $t_{ij} (j = 1, 2, 3)$, corresponds to colourings where t_i has colour j.

Since tokens are not adjacent, it is possible to choose colours for the internal vertices of the (a, b)-forbidding paths so as to obtain a proper colouring of $G^{'}$. Two colourings α and β corresponding to T_A and T_B respectively are constructed this way. Note that to a given standard token configuration of G there can correspond multiple colourings of G because of the freedom in choice of colours for the internal vertices of the (a, b)-forbidding paths.

For the labels, let W be the set of token edges and token triangles of G i.e., $W = \{e_1, \ldots, e_{|E|}\} \cup \{t_1, \ldots, t_{|T|}\}$ where $|E|$ is the total number of edge tokens in G and $|T|$ is the total number of triangle tokens in G and $f : T \to W$, a function mapping the set of given labels to the tokens on token edges and token triangles of G.

Claim 3.5.1. *Let G, T_A, T_B be a restricted instance of Sliding Tokens where each token is given a unique label as described in section 3.5, and let G, L, α, β be the corresponding instance of List-Colour Path as constructed above. Then G, T_A, T_B is a Yes-instance if and only if G, L, α, β is a YES-instance.*

Proof. [2] Recall that a token configuration in which the token of token edge i (token triangle i) is on e_{ij} (on t_{ij}) corresponds to multiple colourings of G where ei (t_i) has colour j. Because of this multiplicity of colourings, we define colour classes of colourings: if two colourings κ and λ of G have $\kappa(t_i) = \lambda(t_i)$ and $\kappa(e_i) = \lambda(e_i)$ for every i, then κ and λ are said to be in the same colour class.

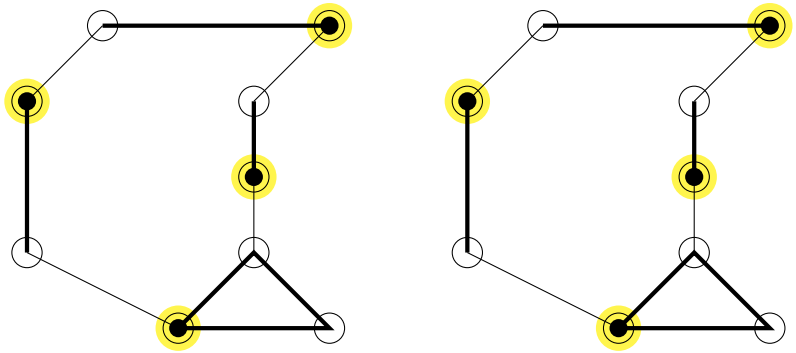
Hence the correspondence between standard token configurations and colourings defines a mapping between standard token configurations and colour classes. This mapping is in fact a bijection: (a, b)-forbidding paths restrict their end vertices from having colours a and b respectively, but they pose no other restriction on the possible colours of their end vertices. So t_{ia} and e_{jb} cannot both be occupied by a token in a token configuration if and only if no colouring κ has $\kappa(t_i) = a$ and $\kappa(e_j) = b$. (Similar statements hold for pairs t_i and t_j, and pairs e_i and e_j.)

The function f also yields a bijective mapping : Since no token can slide along a link edge in the Standard sliding token, each element $w \in W$ can be attributed a label such that the colour of w in the output LIST-COULOUR path instance represents the vertex placement of the label $t \in T$ s.t $f(t) = w$.

Now we claim that if there exists a sequence of moves that transforms T_A into T_B, then there exists a sequence of recolourings that transforms α into β. We mentioned earlier that any token configuration obtainable from T_A is a standard token configuration [2]. Hence every token move corresponds to recolouring a vertex t_i or a vertex e_i. Note that before recolouring t_i (or e_i), it may be necessary to first recolour some internal vertices of (a, b)-forbidding paths incident with t_i (or e_i), but by the definition of (a, b)-forbidding paths,

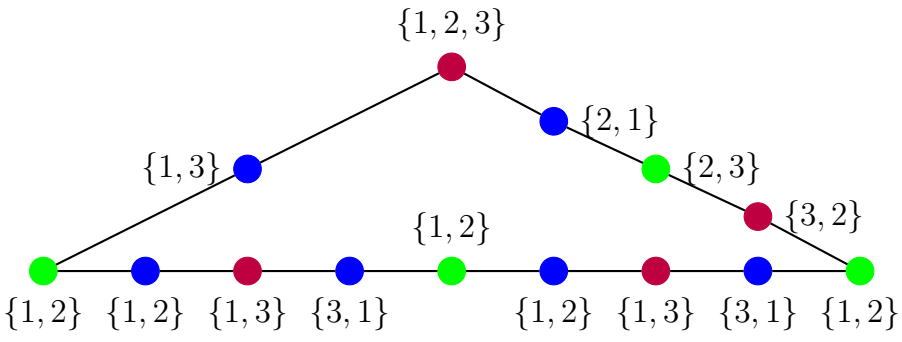
we know this is always possible. It can also be seen that when we finally arrive in the colour class that contains β in this way, the internal vertices of all (a, b)-forbidding paths can be recoloured so that exactly the colouring β is obtained.

Similarly, for every sequence of recolourings from α to β we can construct a sequence of token moves from T_A to T_B: whenever a vertex t_i (e_i) is recoloured from colour a to colour b, we move the corresponding token from t_{ia} to t_{ib} (from e_{ia} to e_{ib}). This completes the proof. □

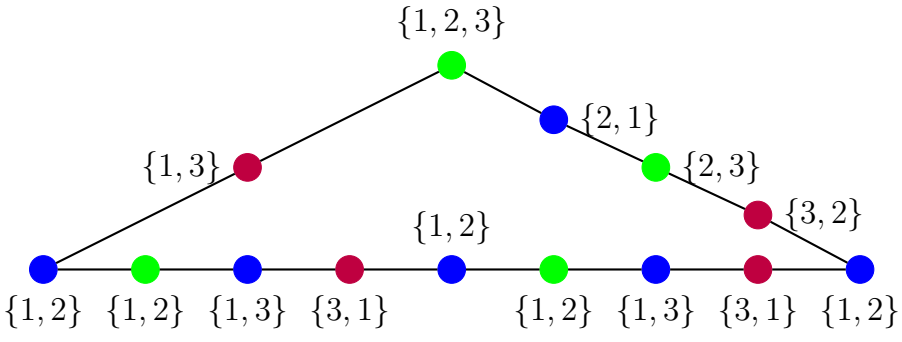


(A) Initial configuration = C_0 (B) C_1

FIGURE 3.10: Configuration-to-edge input instance



(A) Initial configuration = C_0



(B) Initial configuration = C_0

FIGURE 3.11: Configuration-to-edge input instance

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. The craze behind it led to the celebrated theorem of Cook Levin, that SAT is NP-complete. Schaefer proposed in [28] 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. 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.

Since then, dichotomies or trichotomies have been established for several aspects of the satisfiability problem such as optimization [6,8,24], counting [7], inverse satisfiability [23], minimal satisfiability [28], unique satisfiability [19], 3-valued satisfiability [3] and propositional abduction [9]. Very recently, Gopalan et al. studied in [17,18] connectivity properties of the solution-space of Boolean formulas, and investigated complexity issues on connectivity problems in Schaefer's framework [31].

For Boolean satisfiability problems, the structure of the solution space is characterized by the solution graph, where the vertices are the solutions, and two solutions are connected iff they differ in exactly one variable. In 2006, Gopalan et al. studied connectivity properties of the solution graph motivated mainly by research on satisfiability algorithms and the satisfiability threshold. 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. They proved dichotomies for the diameter of connected components and for the complexity of the st-connectivity question, and conjectured a trichotomy for the connectivity question. Recently, the trichotomy was established in [**?**].

A direct application of st-connectivity in solution graphs are reconfiguration problems. Gopalan et al.'s results have also been applied directly to reconfiguration problems, that arise when a step-by-step transformation between two feasible solutions of a problem is searched, such that all intermediate results are feasible. The solutions (satisfying assignments) of a formula Φ over n variables induce a subgraph $G(\Phi)$ of the n-dimensional hypercube graph, that is, the vertices are the solutions of Φ, and two solutions are connected iff they differ in exactly one variable.

In the reconfiguration context the following general set-up is considered : Given a Boolean formula φ with n Boolean variable and two satisfying assignments s_1 and s_2, is it possible to transform s_1 to s_2 such that at each step, only one variable x_i can be flipped and each intermediate assignment remains feasible.

eds rework

- Roadmap.

4.1 Schaefer's framework

4.1.1 Basic concepts

A CNF-formula is a Boolean formula of the form $C_1 \wedge \cdots \wedge C_n$, where each C_i is a clause, that is, a finite disjunction of literals. 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 $\{0,1\}^k$, for some $k \geq 1$; k 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.

Let $\mathcal{S}$ be a finite set of logical relations. A CNF($\mathcal{S}$)-*formula* over a set of variables $V = \{x_1, \ldots, x_n\}$ is a finite conjunction $C_1 \wedge \cdots \wedge C_n$ of clauses built using relations from $\mathcal{S}$, variables from V, and the constants 0 and 1; this means that each C_i is an expression of the form $R()$, where $R \in \mathcal{S}$ is a relation of arity k, and each C_i is a variable in 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. To clean up

4.1.2 Statement of Results

The *satisfiability problems* SAT($\mathcal{S}$) associated with a finite set $\mathcal{S}$ of logical relation asks : Given a CNF($\mathcal{S}$) φ, is it satisfiable ?

Theorem 4.1.1. *Let $\mathcal{S}$ 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.*

Dichotomie Results

4.2 Connectivity of CNF-Formulas / Gopalan et al's.

Research has focused on the structure of the solution space only quite recently: One of the earliest studies on solution-space connectivity was done for CNF($\mathcal{S}$)-formulas. In this paper, we are interested in the connectivity properties of the space of solutions of CNF($\mathcal{S}$)-formulas. If φ is a CNF($\mathcal{S}$)-formula with n variables, then the solution graph G(φ) of φ denotes the subgraph of the n-dimensional hypercube induced by the solutions of φ. This means that the vertices of G(φ) are the solutions of φ, and there is an edge between two solutions of G(φ) precisely when they differ in exactly one variable.

The following decision problems were considered in this context : Gopalan et al. studied the following two decision problems for CNF($\mathcal{S}$)-formulas:

The *st-Connectivity Problem* ST-CONN($\mathcal{S}$)
Instance: A CNF($\mathcal{S}$)-formula φ and two satisfying assignments s and t of Φ.

Question: Is there a path from s to t in $G(\varphi)$?

The *Connectivity Problem* CONN($\mathcal{S}$)
Instance: A CNF($\mathcal{S}$)-formula Φ .
Question: Is $G(\Phi)$ connected ?

4.3 Planar NAE 3-SAT Reconfiguration / Cardinal

Chapter 5

Subset sum Reconfiguration

The subset sum problem is a well-known NP-complete problem 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 [17], Ito and Demaine considered the following problem : Given two *packings*[1] 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. This problem is referred to as the SUBSET SUM RECONFIGURATION problem and is proved to be strongly NP-hard, and PSPACE-complete for the variant with conflict graph [17].

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 [6]. 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 [6]. This chapter is dedicated to this result.

Roadmap. particular proof in an attempt to detail the hardness proof. To finish at the en

[1] Please refer to [17] for the definition of packings

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 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). Proving the hardness is done in two steps. The first step consists of reducing the labelled Sliding Token Reconfiguration problem to the Exact Cover Reconfiguration problem (lemma 5.3.2). However before diving into the proof of lemma 5.3.2, we first take a detour to sections 5.2 and 5.3 where the labeled sliding token reconfiguration and exact cover reconfiguration problems are introduced respectively.

The second step involves reducing the Exact Cover Reconfiguration 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[27], the k-move subset sum is also in PSPACE. □

5.2 Labelled SLIDING TOKEN problem.

The Sliding token problem is introduced in chapter 3 and detailed in section 3.4. Hearn and Demaine proved that the SLIDING TOKEN PROBLEM is PSPACE-complete for planar graphs [13], as an example of the application of the nondeterministic constraint logic model and they implicitly proved that the SLIDING TOKEN PROBLEM is PSPACE-hard on 3-regular graphs since the reduction is done from a restricted NCL machine (NCL-CONFIGURATION-TO-EDGE) in which the underlying graph is planar and all vertices have degree three. The labelled variant of the sliding token 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 :

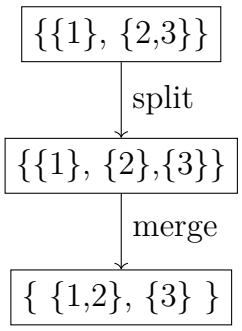


FIGURE 5.1: Reconfiguration sequence which transforms C_1 into C_2 via splits and merges.

5.3.1 k-colorability

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-colorable whenever we can assign one of k colors to each vertex such that no two vertices in a hyperedge have the same color. 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. This is done by reducing the labelled variant of the SLIDING TOKEN PROBLEM to the Exact Cover Reconfiguration problem as mentioned earlier.

The proof is structured in the following way : First the input instance of the labelled sliding token reconfiguration problem is presented in section 5.3.2, followed by the definition of some terms used in the reduction proof in section 5.3.2. The output instance is described in section 5.3.2. Sections 5.3.2 to 5.3.2 are devoted to the proof.

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

Proof. Labeled variant of the SLIDING TOKEN PROBLEM $\leq_p$ Exact Cover Reconfiguration problem.

Input instance of the Labeled SLIDING TOKEN PROBLEM

- $G = (V, E)$, a 3-regular graph.
- T, a set of labeled tokens.
- $p_1 : T \to V$, a function mapping each labeled token to a vertex placement in the starting configuration of the output instance.
- $p_2 : T \to V$, a function mapping each labeled 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 Labeled SLIDING TOKEN PROBLEM

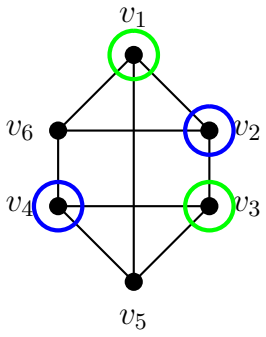


FIGURE 5.2: A 3-regular graph $G = (V, E)$ and initial independent set $I_1 = \{v_1, v_3\}$

Preliminaries

Before diving in the proof, a few concepts that will be used are presented hereunder :

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

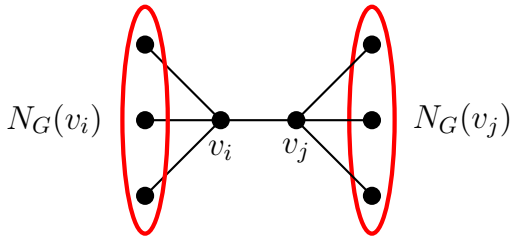


FIGURE 5.3: Slide set denoted S_{ij} of vertices v_i and v_j.

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 5.3.6. A maximally split configuration C

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

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.

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

The output instance of the Exact Cover Reconfiguration 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 5.3.7. According to the above example, 5.2, the output instance of the Exact Cover Reconfiguration problem is the following :

Do the output instance in tikz

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 :

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

23-colorability 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 color. 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. .

reformulate

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 colorability 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 where the worst case scenario (*i.e.*, a tree is attached to each root so as to force the root to meet new nodes and add new edges).

To reformulate thi correctly

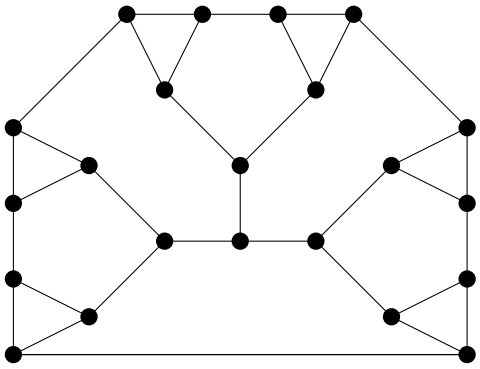


FIGURE 5.5: Graph $G^{'}$ (shown darker) is a subgraph of G.

So G can be 22−colored such that no two vertices of distance at most 3 have the same color. Coloring the tokens in $\mathcal{T}$ a distinct (23rd) color then gives a coloring of $\mathcal{U}$ such that no pair of elements of a common set share the same color.

High level idea

We first note that the subsets containing exactly one vertex and a token (e.g., $\{v_i, t_k\}$) represents 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 $S_{i,j} - \{v_i\}$.

Bijection between configurations.

Given the definition of maximally split configurations in **??**. 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 (pas de doublons) and each token arrangement contains no doublons no plus. Notice also that $f_{red}(p_1) = C_1$ and $f_{red}(p_2) = C_2$.

I do not agree wit f(p1) = c1 etc

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.

Sliding tokens reachability → 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 $\{v_6, t_1\}$.

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 slides), and then from $v_i^{'}$ or $v_j^{'}$ to $v_j^{'}$ (via 0 or 1 slides).

□

d simple example
e to demonstrate
last part

Example 5.3.13.

5.4 3-move Subset Sum reconfiguration problem

5.4.1 PSPACE-hardness result

Theorem 5.4.1. *The 3-move Subset Sum Reconfiguration problem is strongly* PSPACE*-complete.*

Proof. The reduction is from the Exact Cover Reconfiguration problem for instances that are 23−colorable induced hypergraphs, proved PSPACE-hard.

Input Instance of the Exact Cover Reconfiguration problem

Recall that the Exact Cover Reconfiguration instance contains a set $\mathcal{S}$ of subsets of a set $\mathcal{U}$ and two exact covers C_1 and $C_2 \subseteq \mathcal{S}$.

High level idea

Every 3−move Subset Sum Reconfiguration step is either a *merge* where 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.

Reduction Structure

Given an instance of the exact cover problem, each element a of U is given an arbitrary label i where $i \in \{1, \ldots, |U|\}$ and is partitioned according to its color j where $j \in \{1, \ldots, 23\}$. A function $f : \mathcal{U} \to \mathbb{N}$ maps each element 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 and color of an element a. The function f maps a color-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.

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 $\sum_{a \in \mathcal{U}} f(a)$.

Output size

Correctness

A reconfiguration in both the exact cover and 3-move subset sum problems 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 color of a_i. The sums of the outputs of f for all elements of all colors 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 color j or larger is at least $2^{100(j)} log_2(|U|) \geq 2^{(98)} log_2(|U|)$.

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

mythyfy !!!!!!!!! + Cor- ion needed

□

Chapter 6

Future works

While working on this thesis a few questions popped up in my brain :

1. Shortest reconfiguration path in the sliding token reconfiguration problem.
2. St-Conn(S) problem in NCL
3. Can schaefer's dichotomy be applied to the 3-move subset sum reconfiguration problem i.e can we have a tight result for $k \leq 3$ and $k \geq 3$?
4. Finding an optimal $k < 23$ for the colour classes of the exact cover problem. More en rapport avec le path in hypercube.

Chapter 7

Conclusion

We have analysed various aspect of Bounded NCL and have shown that despite some ambiguities, Planar Bounded NCL is indeed NP-complete. We have reduced Bounded NCL to both Klondike and Mahjong and

verified the prior results that these puzzles are NP-complete. We have reduced Planar Bounded NCL to Nonograms, and verified the prior result that Nonograms are NP-complete. We consider our contribution on these results amongst the verification, also the simplicity of the established proof. We have reduced Bounded 2CL to Dou Shou Qi, and proven it PSPACE-hard.

By doing so we have added weight to the claim of [6], stating that Con- straint Logic is indeed a decent framework to analyse and proof the complex- ity of puzzles and games.

As a suggestion for future work it would be interesting to see whether Dou Shou Qi is also in PSPACE, so we can classify it as PSPACE-complete. If it turns out not to be in PSPACE, a mayor contribution would be to prove it EXPTIME-hard (for Constraint Logic can provide 2CL) or maybe even EXPSPACE-hard.

Besides the games and puzzles that are currently classified, there are games which seem to yield to a reduction from Constraint Logic, for exam- ple the Binary Puzzle and Rummikub. Reducing the appropriate Constraint Logic problem to one of these games would be an interesting result.

We have studied the parameterized complexity of constraint logic problems with regards to (combinations of) solution length, maximum degree and treewidth as parameters. As a main result, we showed that Nondeterministic Constraint Logic is PSPACE- complete on planar, 3-regular graphs of

bounded bandwidth constructed using only AND and (protected) OR vertices. We gave several applications of this result, showing reconfiguration versions of several classical graph problems PSPACE-hard on planar graphs of bounded bandwidth, as well as showing Rush Hour PSPACE-complete when played on a rectangular board where one of the lengths of the sides is bounded by a constant.

We showed that when parameterized by solution length, C2E, C2C and bounded C2E variants of Nondeterministic Constraint Logic are W[1]-hard, but bounded C2C NCL becomes fixed parameter tractable. Essentially, the hardness of bounded C2E NCL when parameterized by solution length is due to the complexity of finding a suitable target configuration, rather than from that of determining a reconfiguration sequence. However, when parametrizing by maximum degree in addition to solution length, all cases become fixed parameter tractable.

When parameterized by treewidth, Constraint Graph Satisfiability becomes weakly NP-complete (rather than strongly NP-complete). When considering maximum degree in addition to treewidth, CGS becomes fixed parameter tractable. Note that this is in stark contrast to the complexity of NCL, which remains PSPACE-complete even for fixed values of this parameter. This is an interesting example of how reconfiguration problems can be much harder than their decision variants.

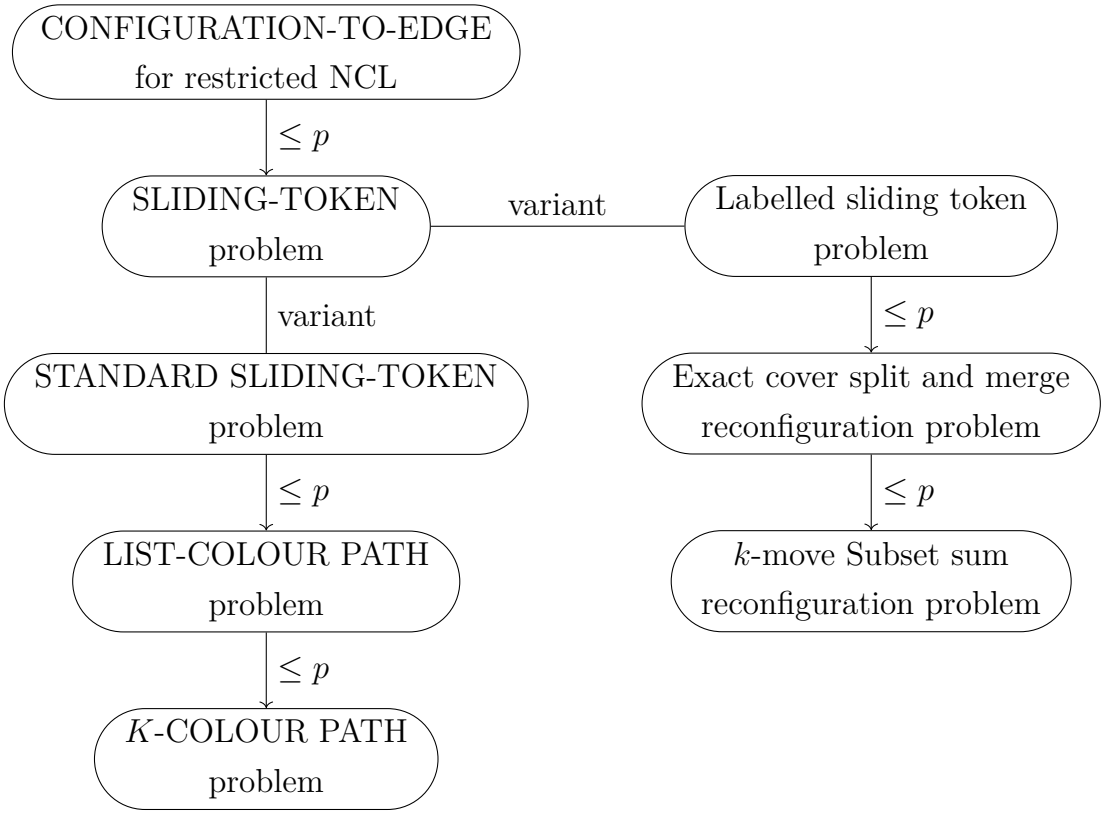


FIGURE 7.1: PSPACE-complete problems encoutered and their relationship.

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] 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.

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

[8] 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.

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

[10] 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.

[11] 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.

[12] 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.

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

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

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

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

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

[18] 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.

[19] 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.

[20] 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.

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

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

[23] 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.

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

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

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

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

[28] 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.

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

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

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