UNIVERSIDADE FEDERAL DE MINAS GERAIS
Instituto de Ciências Exatas
Programa de Pós-Graduação em Ciência da Computação

Caio Henrique Segawa Tonetti

The Token Swap Problem: Polynomial Time Algorithms for Graph Classes and Integer Linear Programming Formulations

Belo Horizonte
2022

Caio Henrique Segawa Tonetti

The Token Swap Problem: Polynomial Time Algorithms for Graph Classes and Integer Linear Programming Formulations

Final Version

Thesis presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Vinicius Fernandes dos Santos
Co-Advisor: Sebastián Urrutia

Belo Horizonte
2022

Tonetti, Caio Henrique Segawa.

T664t The token swap problem:[recurso eletrônico] polynomial time algorithms for graph classes and integer linear programming formulations / Caio Henrique Segawa Tonetti – 2022.

1 recurso online (39 f. il., color.) : pdf.

Orientador: Vinicius Fernandes dos Santos
Coorientador: Sebastián Urrutia

Dissertação (mestrado) - Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação.
Referências: f. 36-39.

1. Computação – Teses. 2. Arquitetura de computador – Teses. 3. Teoria dos Grafos – Teses. 4. Programação linear – Teses. I. Santos, Vinicius Fernandes dos. II. Urrutia, Sebastián. III. Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da computação. IV. Título.

CDU 519.6*61(043)

Ficha catalográfica elaborada pela bibliotecária Irenquer Vismeg Lucas Cruz
CRB 6/819 - Universidade Federal de Minas Gerais – ICEx

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

The Token Swap Problem: Polynomial Time Algorithms for Graph Classes and Integer Linear Programming Formulations

CAIO HENRIQUE SEGAWA TONETTI

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

Prof. Vinícius Fernandes dos Santos - Orientador
Departamento de Ciência da Computação - UFMG

Prof. Sebastián Alberto Urrutia - Coorientador
Departamento de Ciência da Computação - UFMG

Profa. Celina Miraglia Herrera de Figueiredo
Programa de Engenharia de Sistemas e Computação - UFRJ

Prof. Fernando Magno Quintão Pereira
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 21 de Fevereiro de 2022.

This dissertation would not been possible unless for the support of all my peers.

Acknowledgments

I would like to first thank my mother, Liria Yuriko, who always supported my decisions. She was always my teacher, protector, and showed me all I needed to live my life. She always accepted me for who I am and continues to do so until today. I own everything I am to her. Secondly, I thank my brothers, who helped me and my mother when needed.

I also have to thank all that helped and supported me during the last two years of my masters degree and showed me clarity of mind even when things were not so good, and Fernanda Martins and his mother, Marcia, for everything they did for me during my entire undergraduate years.

In my years as a undergraduate in Universidade Estadual de Maringá, my ex-advisor, Anderson Faustino, taught me a lot about research and computer science. In my masters, I have to thank my advisor, Vinicius Fernandes dos Santos, and co-advisor, Sebastián Urrutia, for the encouragement, patience and vast knowledge in their respective fields.

Lastly, I have to thank all the professors that were part of my life in some way and gave me the necessary knowledge to finish this dissertation. And, of course, to all the friends and my psychologist, Luis Mello, that helped me overcome these last difficult years. I am what I am not because of my accomplishments, but only because of those that raised me and helped me learn how to tread my own path in life.

"*For nothing is self-sufficient, neither in us ourselves nor in things; and if our soul has trembled with happiness and sounded like a harp string just once, all eternity was needed to produce this one event—and in this single moment of affirmation all eternity was called good, redeemed, justified, and affirmed.*"

(Friedrich Nietzsche)

Resumo

O framework de reconfiguração introduz o conceito de transformação em problemas computacionais, mostrando novas preocupações como resultado da necessidade de compreender estas mudanças sob uma variedade de operações e restrições.

Esta dissertação estuda o problema de Token Swap, um tipo de tarefa de reconfiguração em que começamos com fichas distribuídas nos vértices de um grafo e buscamos movê-las até que cada ficha alcance o vértice alvo que lhe corresponde. O objetivo é realizar essa transformação usando o menor número possível de operações de troca.

O principal resultado dessa dissertação é a construção das ferramentas matemáticas necessárias e da prova de existência de um algoritmo ótimo para grafos da classe *threshold* e, subsequentemente, cografos. Então, alguns trabalhos preliminares sobre modelos de programação linear inteira para os problemas de *Token Swap* e *Parallel Token Swap* também são apresentados, juntamente com o raciocínio por trás de cada restrição.

Palavras-chave: teoria dos grafos; reconfiguração; programação inteira.

Abstract

The reconfiguration framework introduces the concept of transformation into computational issues, posing new concerns as a result of the need to comprehend these changes under a variety of operations and constraints.

This dissertation studies the Token Swap problem, a type of reconfiguration problem in which we start with tokens placed on the vertices of a graph and aim to move them so that each token ends up on its correct target vertex. The objective is to achieve this using as few swap operations as possible.

The main result of this dissertation is the construction of the necessary mathematical tools and the proof of existence of an optimal algorithm for the class of threshold graphs and subsequently cographs. Then, some preliminary work on integer linear programming models for the problems of Token Swap and Parallel Token Swap will also be presented, together with a simple reasoning behind each constraint.

Keywords: graph theory; reconfiguration; integer programming.

List of Figures

List of Algorithms

Contents

Chapter 1

Introduction

The reconfiguration framework [19] formalizes the notion of *transformation* of abstract objects and questions arises from the necessity of understanding these changes under various operations and constraints.

A simple problem in this class is sorting a list of numbers by adjacent swaps between two elements. A swap sequence can be calculated in polynomial time by a modified bubble sort algorithm [24]. The minimum number of swaps is called the *swap number* and in this case, is exactly the number of pairs of elements that are out of order.

Under other conditions, in a related problem, the list is to be sorted by *prefix-reversals*, an elementary operation that flips a prefix of the list. In this case, finding the minimum number of flips is **NP-Hard** [9]. This problem is called the pancake sorting problem.

Another well-known reconfiguration problem is the n-puzzle, a sliding puzzle on a grid of n numbered square tiles with exactly one tile missing. In this problem, each step slides a tile to an adjacent empty tile space to achieve a final sorted configuration state. Research of the 15-puzzle dates back to the late 19^{th} century [21] and is commonly used as an introductory problem for modelling heuristics. Figure 1.1 is an example of a 15-puzzle instance.

Considering the reconfiguration framework, the interest usually lies in one of the three following problems: connectivity, diameter or distance. These problems translate, respectively, into the following questions: (a) can any configuration be reconfigured into another?; (b) what is the maximum number of required steps between any pair of configurations?; and (c) given two configurations, what is the minimum number of operations to move from one to the other? In the examples aforementioned, the distance between two configurations (the initial and the target) was used. More information on reconfiguration problems can be found in the surveys avaliable in the literature [35, 27, 28].

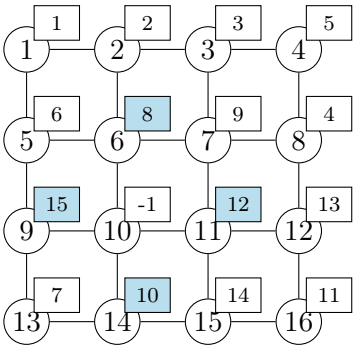


Figure 1.1: Instance of the 15-puzzle problem in a grid graph. Each vertex is represented by a circle and the rectangle is the tile currently positioned in the vertex. The -1 rectangle represents the empty tile. Each light blue colored rectangle is a possible swap operation with the empty tile in the current configuration.

This dissertation focuses on a reconfiguration problem called the Token Swap problem (TS), defined in Definition 1.0.1.

Definition 1.0.1 (Token Swap problem)**.** Let $G := (V, E)$ be a graph with $n = |V|$ vertices and $|E|$ edges. Let $f_0 : V \mapsto V$ be an initial bijective token placement. The objective is to reconfigure this initial token placement into the identity token placement f_ι that maps every node to itself with minimum distance. The reconfiguration must consist of a token swapping sequence S, in which each element of S is a pair of adjacent graph vertices, meaning that the elementary operation is restricted to a swap between two tokens placed on vertices that share an edge in the graph.

The decision version of this problem aims to determine if it is possible to have a swap sequence S that transforms f_0 to f_ι in k or less swaps, for a given $k \in \mathbb{N}$. In Figure 1.2, a complete example of a TS instance is shown. The formal mathematical definitions needed to fully understand this work will be given in Section 1.2.

As we will review in this section, TS is know to be hard, in many senses, but solvable in polynomial-time for a few special cases. In this work we generalize some results in the literature for a much larger class of graphs, namely, cographs. Our main contribution is the following theorem: The Token Swapping Problem can be solved in polynomial time in cographs.

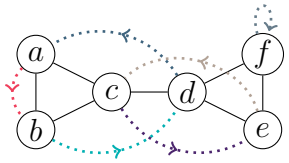


Figure 1.2: Example of a instance of the Token Swap problem with $V = \{a, b, c, d, e, f\}$ and tokens represented by colored dotted arrows to differentiate. The arrows point from the vertice that is originally positioned in mapping f_0 to the target vertice in the identity map f_ι.

Every reconfiguration problem may be defined using a reconfiguration graph where each node is a possible state of the combinatorial or geometric object and an edge exists between two vertices if and only if each of the vertices can be reached in exactly one reconfiguration step from the other [28]. In the case of the TS, the vertices of the reconfiguration graph correspond to the possible token placements and the reconfiguration graph will be connected as long as the original graph is also connected. The connectivity guarantees that any configuration can be reconfigured into any other, as it is possible to take any spanning tree of the original graph and greedly move any token corresponding to a leaf to its correct position. Then, the leaf can be disregarded and the procedure continues inductively until the final configuration is achieved. This process give us an upper-bound of $O(n^2)$ swaps for any instance of the TS problem [38].

Although the reconfiguration graph of a given graph G could be theoretically built from G, in practice this is not viable, since, in general, the number of configurations is factorial in $|V|$.

If the TS problem is brought to the realm of permutation group theory, it is possible to model this problem by a group $(F, \circ)$, in which every element of F is a bijective function representing a possible configuration. The binary operation is function composition and every element of F can be represented by the product of finitely many elements of a subset C of F and their inverses. The elements of C are called the generators of the group $(F, \circ)$ and they represent each possible transpositions corresponding to the edges of the original graph. Thereupon, given a TS instance, the Cayley Graph $\Gamma(F, C)$ of the symmetric group will correspond (under isomorphism) exactly to the reconfiguration graph of the original problem, the distance between two configurations will be the shortest path between those two vertices and the worst case will match the diameter of this graph. The shortest path in a Cayley Graph, also known as the `Minimum Length Generator Sequence` problem, is a generalization of the TS problem, and a **PSPACE-complete** problem [20]. The diameter of the Cayley Graphs has been researched in the context of transposition trees [3, 12, 6, 15, 16, 11, 25, 10]—i.e., transpositions generators from the edges of a tree. It resulted in many heuristic algorithms to calculate upper-bounds that do not depend on the vertex number of the Cayley Graph[1].

Applications of the TS problem encompass a wide range of fields and some examples are: computing efficient interconnection network structures [5], computational biology [6, 18], model Wireless Sensor Networks (WSS) [37], protection routing [29] and qubit allocation for quantum computers [32, 31]. More applications can be found in [2].

Although variants and particular cases of TS have been studied of decades the reconfiguration version of TS considered was first formalized in 2015 [38], and further generalizations and variations were studied in the same year [39]. The TS problem was proved **NP-Complete**, even when restricted to bipartite graphs with degree bounded

[1] In the case of the TS, the vertex number of a Cayley Graph is $O(n!)$.

by 3. It is also **APX-complete** and **W[1]-hard** parameterized by number of swaps, but fixed parameter tractable (**FPT**) for the class of nowhere dense graphs [22, 26]. Subsequently, it was proved that the problem remains hard even when both the treewidth and the diameter of the input graph are constant [8]. These are some special classes of graphs in which the problem can be solved through exact polynomial time algorithms: cliques, paths, cycles, stars, brooms, lollipop [22], complete bipartite graphs and complete split graphs [7], where the last two are subclasses of cographs. We remark that all these graph classes are extremely restricted, in the sense that, for each n, there is a constant number of graphs on n vertices in any of them.

Approximation algorithms have also been studied for the problem. In particular, there are 2-approximation algorithms for square of paths [18] and for trees [38, 26], the latter being known to be tight [7]. It is established that the known techniques for approximating TS on trees cannot achieve better approximation factors [2]. Both approximation algorithms can be adapted to general graphs, providing α- and 4-approximation algorithms, respectively, where α is a value for which the input graph admits an α-spanner tree. In [8] it was conjectured that TS remains **NP-Complete** even in trees and [7] reinforced this conjecture by showing a counterexample of the *Happy Leaf Conjecture* [36]. And finally, it was shown that TS and other two variations (Weighted Token Swapping and Parallel Token Swapping) are **NP-Complete** on trees [2].

1.1 Organization of the Work

This chapter focuses on introducing the mathematical tools and other already researched graph classes necessary for understanding the rest of this dissertation. Chapter 2 explains the main subject of this dissertation, introducing the class of cographs. The first section, Section 2.1, presents the method for finding an optimal swap sequence for threshold graphs and the respective proof of correctness, while Section 2.2 improves on the past proof to generalize the method for cographs.

The following chapter, Chapter 3, presents two initial integer linear programming models for the Token Swap problem and Parallel Token Swap problem, being Section 3.1 and Section 3.2, respectively. Then, the subsequent sections and subsections render a discussion about each of the constraints and their design. The dissertation is then concluded with a simple revision and exploration of what can be done in the future.

1.2 Preliminaries

For an integer k, the notation $[k] := \{1, 2, \ldots, k\}$ is used. For a set V, a mapping function $f : V \mapsto V$ is a bijective function that internally maps elements of the set. An identity map is a special mapping function f_ι that maps every element to itself.

For a set V, an ordering of the elements of the set is a bijective function $M : V \mapsto [|V|]$ and the shorthand $u <_M v$ for the comparison $M(u) < M(v)$ of the order of two elements $u, v \in V$ is adopted.

A graph $G := (V, E)$ is a pair of a vertex set $V = \{v_1, v_2, \ldots, v_n\}$ and edge set E, and the elementos of E are pairs of elements of V. The graph is called *undirected* when edges E are unordered tuples, while in *directed* graphs edges are ordered tuples.

The shorthand 'uv' is used to describe a pair $\{u, v\}$ (or (u, v) in case directed graphs) and the functions $V(G)$ and $E(G)$ are used to respectively retrieve the sets V and E when they are omitted in the graph definition.

A *subgraph* $\dot{G} := (\dot{V}, \dot{E})$ of the graph G, denoted as $\dot{G} \subseteq G$, is a graph such that $\dot{V} \subseteq V(G)$ and $\dot{E} \subseteq E(G) \cap \dot{V}^2$ and is called an *induced* subgraph when $\dot{E} = E(G) \cap \dot{V}^2$.

The outdegree $\delta_{out}(v)$ and indegree $\delta_{in}(v)$ of a vertex $v \in V(G)$ is the number of edges (v, w), $\forall w \in V(G)$ and (w, v), $\forall w \in V(G)$ that belong to $E(G)$, respectively. The degree $\delta(v)$ of a vertex is defined as the sum of $\delta_{in}(v)$ and $\delta_{out}(v)$ for directed graphs and as the number of edges having v as endpoint for undirected graphs.

The open neighborhood $N_G(u)$ of a vertex is defined as a set of all vertices v such that $uv \in E(G)$ and u itself is included. If u is not included, the set is called a closed neighborhood as is denoted as $N_G[u]$. The complement of a graph G is a graph $\dot{G}$ with the same vertex set $V(G)$ such that an edge exists between two distinct vertices if and only if it does not exist on G.

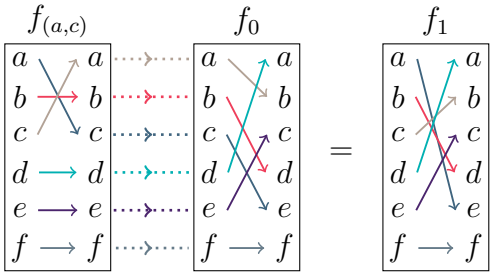


Figure 1.3: Let f_0 be the TS instance represented in Figure 1.2, $f_{(a,c)}$ be the mapping function representation of swap (a, c) and f_1 be the resulting mapping of the operation $f_0 \circ f_{(a,c)}$. Each rectangle represents a function with domain on the left, codomain on the right and color coded arrow mappings. It is possible to observe the resulting operation by following each arrow from left to right between $f_{(a,c)}$ and f_0.

An instance of the TS problem is composed of an undirected graph $G := (V, E)$

and a mapping function f_0 that denotes an initial token placement on vertices of G. A *swap* is defined by a vertex pair $s := (u, v)$ and it can be *applied* to exchange two tokens in the current mapping of an instance provided that $uv \in E$. More formally, any swap $s = (u, v)$ can be represented as a mapping function f_s, such that $f_s(u) = v$, $f_s(v) = u$ and $f_s(w) = w$, $\forall w \in V \setminus \{u, v\}$. f_s can be applied to any token placement f by the composition $f \circ f_s = f_1$. An example on how the composition operation can swap function values is shown in Figure 1.3.

Let $G := (V, E)$ be a graph. Two vertices $u, v \in V$ are *connected* if there is at least one *path of vertices* $(v_1, v_2, \ldots, v_k)$ in G such that $v_i v_{i+1} \in E, \forall_{i \in [k-1]}$, $v_1 = u$ and $v_k = v$. A graph G is connected if all vertex pairs of V are connected and disconnected otherwise. The function $dist_G(u, v)$ denotes the minimum number of edges in any path of vertices between nodes u and v for a graph G. Note that this distance is considered infinite in the case there is no path between the two vertices. A token placement f is considered *valid* if all vertex pairs $(f(u), u)$ are connected. As all token placements in this paper are assumed to be valid, every disconnected subgraph of a TS instance can be separated into smaller connected instances with restrictions to the domain of the initial placement. Hence, all graphs can be assumed connected without loss of generality.

Let $S := (s_1, s_2, \ldots, s_k)$ be a sequence of swaps. A swap sequence S *solves* an instance of the TS problem if and only if the identity function is resulted by applying each swap iteratively, as shown in Equation 1.1. Every intermediate mapping function created is represented by a f_p, where p is the number of swaps applied from the initial configuration. Moreover, two placement mappings are said to be *adjacent* if there is exactly one swap to transform one placement into another. The Token Swap problem asks for a sequence of swaps that solves a given instance with the minimal number of swaps k. The function $OPT_G(f)$ is a special function to symbolize an optimal swap sequence size for a token placement f.

$$\underbrace{(((\underbrace{(f_0 \circ f_{s_1})}_{f_1} \circ f_{s_2}) \circ \ldots) \circ f_{s_k})}_{f_k} = f_\iota \tag{1.1}$$

This version of the problem is equivalent to the problem of finding a swap sequence between two arbitrary mapping functions f_0, f_e by the use of the following process of *token renaming* from f_0 to $\dot{f}_0$: if a token i has $f_0(i) = u$, the i is renamed to the token $f_e(u)$, generating an instance of the TS problem with the same graph and initial mapping $\dot{f}_0 = f_e \circ f_0$. Given a S that solves the TS instance, the vertices can just be renamed back to get a swap sequence of the initial problem. Any swap sequence S is also *reversible*, meaning that if S transforms f_i to f_j, then it is possible to transform f_j to f_i by applying the swap sequence of S in the reverse order. This property is specially useful in some applications

like quantum computing, where all quantum logic operations must be reversible.

Another important definition needed throughout this work is the notion of the *Lowest Common Ancestor* (LCA), sometimes called nearest common ancestor, for trees. A *tree* G is a undirected and connected graph that has no cycles and is called *rooted*, denoted as G^r, if there exists a special node $r \in V(G)$ called *root* that functions as a reference node for heights in the graph. Any vertex u of a tree with $\delta(u) = 1$ is called a leaf, with the possible exception of the root of a rooted tree. A *subtree* is a subgraph of a tree that is still a tree. A subtree $\dot{G}$ of a rooted tree G^r can also be rooted in relation to the original root, as the nearest vertex from the subtree to r in G^r will be the subtree's root. For a given rooted tree G^r, $u, v \in V(G^r)$, the lowest common ancestor between u and v, denoted as $LCA_{G^r}(u, v)$, is the lowest node such that both nodes are descendants of (possibly one of them). In other words, it is the nearest shared ancestor of both u and v.

This notion is extended to calculate the LCA for any vertex subset of the tree. Let G^r be a rooted tree and $\dot{V} \subseteq V(G^r)$ a vertex subset. The lowest common ancestor $LCA_{G^r}(\dot{V})$ is the nearest node from the root in the set of nodes built from the lowest common ancestors between every pair of nodes in $\dot{V}$.

Given the set of every pairwise LCA of the subset, the LCA of the entire subset $\dot{V}$ is the nearest node from the root in the set.

The problem of calculating $LCA_{G^r}(u, v)$ is called *offline* when the graph and queries are being given as input and has been first proposed in [1] with an optimally efficient algorithm. Other versions were studied later [17], with many different algorithms and general improvements over the years [4]. Some of these algorithms present a linear time pre-process stage that creates a data structure that can be dynamically queried in asymptotically constant time and others offers efficient algorithms that queries on trees that can be changed dynamically. The exact improvements and methods used to calculate the lowest common ancestor go out of the scope of this work. From the above, the calculation of the $LCA_{G^r}(\dot{V})$ can also be derived in optimally efficient time, as the number of pairs of vertex is bounded by $O(|\dot{V}|^2)$.

A Conflict Graph $CG_f := (V(G), E_{CG})$ is a directed graph that, for a token placement f of a graph G, an edge $(u, v) \in E_{CG}$ if and only if $f(u) = v$. Note that each node has outdegre 1, as there can be only one token per vertex, and indegree 1, as there is exactly one vertex for each token. Then, the graph is composed by a set of vertex disjoint directed cycles. Observe that the directed graph may contain self-loops whenever a token is already in the correct vertex. The number of cycles in the Conflict Graph is bounded by the number of vertices in the graph and it matches this bound when the current token configuration matches the identity map f_ι.

In Figure 1.2 the conflict graph is shown with dotted lines. The *joint representation* is an easy way to visually represent a Conflict Graph and the original graph as one, as

seen in Figure 1.2, and will be used throughout this dissertation.

Let f be a token configuration and CG_{f_i} the related conflict graph with permutation cycle set $CS(CG_{f_i})$. Every swap that can applied to f_i transform CS in some way. These transformation can be classified into two types: *Merge* and *Split* as they are described in the following paragraph.

Take two distinct permutation cycles $C_i, C_j \in CS(CG_{f_i})$ and assume, without loss of generality, that $C_i = (u, u_1, \ldots, u_k, u)$ and $C_j = (v, v_1, \ldots, v_p, v)$, for $k, p \in \mathbb{N}_0$. A Merge is a swap (u, v) that is applied between the two cycles C_i and C_j resulting in a configuration $\dot{f}_i$ such that $CS(CG_{\dot{f}_i}) = (CS(CG_{f_i}) \setminus \{C_i, C_j\}) \cup \{C_{ij}\}$, where $C_{ij} = (v, u_1, \ldots, u_k, u, v_1, \ldots, v_p, v)$. Now, take a permutation cycle $C \in CS(CG_{f_i})$ and assume, without loss of generality, that $C = (u, u_1, \ldots, u_k, v, v_{k+1}, \ldots, v_{k+p}, u)$, for $k, p \in \mathbb{N}_0$. A Split is a swap (u, v) that is applied on the cycle C resulting in a configuration $\dot{f}_i$ such that $CS(CG_{\dot{f}_i}) = (CS(CG_{f_i}) \setminus \{C\}) \cup \{C_i, C_j\}$, where $C_i = (v, u_1, \ldots, u_k, v)$ and $C_j = (u, v_{k+1}, \ldots,$
$v_{k+p}, u)$.

Lemma 1.2.1. *Let G be a graph and f be a configuration of a TS problem instance. Any possible swap $(u, v) \in E(G)$ is either a merge or a split transformation in $CS(CG_f)$.*

Proof. By the conflict graph definition, every vertex must have outdegree one and indegree one and they are in cycles and self-loops. As every vertex is on a cycle and vertices between cycles are disjoint, every swap must be applied either internally on a cycle or between two cycles. A swap applied internally on a cycle is called a split swap and a swap applied between two cycles is called a merge swap by the above definition. □

From these transformations, there are two special cases that are worth mentioning: Swap between a cycle of size one and any other cycle and swap on an edge of the cycle. Figure 1.4 is an example to help visualize split and merge swaps. These two cases can be used as tools to add or remove, respectively, a node from a cycle and will be useful in this work. For any cycle $C \in CS$ and vertex $v \in V(G)$, it is said that v *dominates* the cycle C if and only if the vertices $V(C)$ are a subset of the open neighborhood of v, $V(C) \subseteq N_G(v)$.

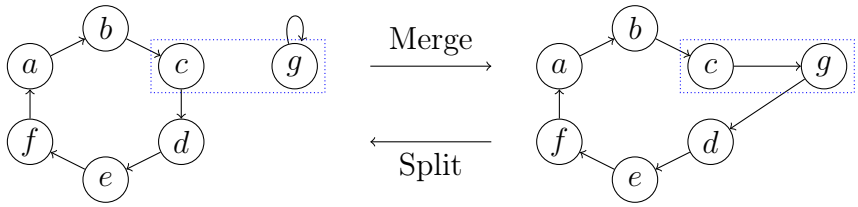


Figure 1.4: Let G be a connected graph with seven vertices where edge (c, g) must exist in $E(G)$. The left image denotes the current configuration as directed cycles and the blue rectangle shows which swap will be applied to achieve the configuration on the right image. This operation *merges* the two cycles and reapplying the swap returns the configuration to its original configuration, effectively *splitting* the two cycles.

Let f be a token placement map of a TS instance. The sum of the distances $\delta(f(u), u)$ is the sum of the distances between each token to its target vertex. With the sum, one could test if a swap sequence S solves the instance by checking if is 0 for the resulting placement $f_{|S|}$. For trees, every swap can be classified in one of three categories related to the sum of distances of an instance: (a) The swap decreases the sum by two through moving two tokens closer to their target vertices, also called a *happy swap*; (b) the swap does not change the total sum by moving one token closer and one token further from their target vertices and (c) the swap increases the sum by two, as it moves two tokens further from its target vertices.

Intuitively, one could think that any swap sequence that solves a TS instance with swaps restricted to categories (a) and (b) will have less swaps than a swap sequence that solves the same instance and uses swaps of category (c) —as [33] tried to prove, but subsequently found an error [34]. Then, [36] conjectured that any optimal swap sequence would not swap already correct tokens on leafs and called them *happy leafs*, resulting in the so-called *Happy Leaf Conjecture* This conjecture was disproved by [7], as they shown that there is a class of infinite trees that need (c) swaps to achieve an optimal solution and that any algorithm that do not consider swaps on happy leafs has an approximation factor of *at least* $\frac{4}{3}$ for general graphs and trees. An example of this graph family and the process used can be seen at Figure 1.5.

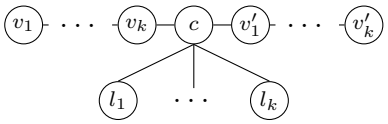


Figure 1.5: Let $T_k := (V, E)$ be a graph with a path with $2k + 1$ vertices built by using the vertex order $v_1, \ldots, v_k, c, v'_1, \ldots, v'_k$. Additionally, the center vertex c will have k leafs $l_1, \ldots, l_k$. Let tokens be positioned at c and let its leaves be correct, while the path nodes are inverted, ie. the token on vertex v_i have the final position at v'_i and vise-versa. Any algorithm that anchors happy leaves will be bounded by the same swap number in a path P_{k+1} with inverted tokens, which in this case is $2k^2 + k$. But, by using the leaf nodes as a *staging area* for tokens, it is possible to expertly switch tokens from $v_1, \ldots, v_k$ to $l_1, \ldots, l_k$, then swap $l_1, \ldots, l_k$ to $v'_1, \ldots, v'_k$, correcting the first half, then performing the same process to correct the second half. After this process, only the original happy leaves remain out of place, and they can be corrected by using the center node as a pivot. This procedure, described in more detail in [7], yields a tighter upper bound on the number of swaps with $\frac{3}{2}k^2 + 9k$ total swaps.

In the realm of Group Theory, it is possible to represent each permutation cycle C as a mapping function f_C, as seen before in Figure 1.3, by a decomposition of a original configuration f. A *partition* $\mathcal{P}(A) = \{P_1, \ldots, P_k\}$ of a set A is a grouping of its elements into non-empty subsets $P_i \in [k]$, such that every element of the set A belong to exactly one of these subsets. Let G be a graph, f be a token configuration and $\mathcal{P}(V(G))$ be a vertex partition such that every configuration f_P, $P \in \mathcal{P}(V(G))$ is a valid token configuration on G. This partitioning is called a *valid* partition.

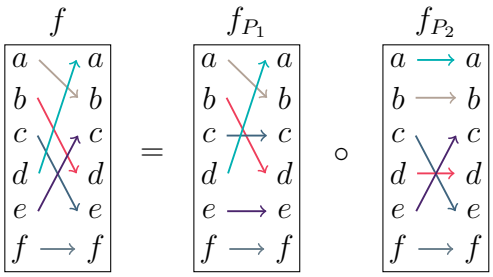


Figure 1.6: Let f be a valid token configuration and $\mathcal{P}(\{a, b, c, d, e, f\}) = \{\{a, b, d\}, \{c, e\}, \{f\}\}$ be a partition of f. This partition is valid, as elements f_{P_1}, f_{P_2} and f_{P_3} are a valid token configurations. Moreover, this partition is also the coarsest partition, as there is no other partition of greater size such that every element are valid. The above example shows that the composition operation of the elements of the partition returns to the original mapping. Note that the mapping f_{P_3} is equivalent to the identity configuration and not shown.

From the configuration f and valid element of the partition $P \in \mathcal{P}(V(G))$, a configuration f_P is built such that $f_P(v) = f(v)$, $\forall v \in P$ and $f_P(v) = v$, $\forall v \in V(G) \setminus P$. The *coarsest* valid partition is defined as the partition where for each other partition of greater size there is at least one subset P that the configuration f_P do not result in a

valid token configuration. For any valid partition $\mathcal{P}(V(G))$ of a TS problem instance, the composition of all element configurations f_P results in the original partition f as seen in Equation 1.2, with $k = |\mathcal{P}(V(G))|$.

$$f_{P_1} \circ f_{P_2} \circ \ldots \circ f_{P_k} = f \tag{1.2}$$

Lemma 1.2.2. *Let $\mathcal{P}(V(G))$ be a coarsest valid partition of a graph G and a configuration f of a TS instance. There is a partition configuration f_P, $P \in \mathcal{P}(V(G))$ if and only if there is a cycle $C \in CS(CG_f)$ such that $V(C) = P$ and $f_P(v) = u$ if and only if $(v, u) \in E(C)$.*

Proof. ($\rightarrow$) Assume any partition $P \in \mathcal{P}(V(G))$ with configuration f_P. As this configuration is valid by the definition, there is a set of cycles $CS(CG_{f_P})$ and every vertex in $V(G) \setminus P$ must be in a self-loop of $CS(CG_{f_P})$. If the subgraph of the vertices of P on CG_{f_P} is disconneted, then each connected subgraph of this subgraph is a disjoint cycle and it is possible to create a smaller partitions of P that are still valid, which is a contradiction on the coarsest valid partition. So, there is exactly one $C \in CS(CG_{f_P})$, such that $V(C) = P$. From the definition of the configuration f_P and the original configuration f, where $f(v) = u$, $f_P(v) = f(v) = u$ for $(v, u) \in E(CG_f) \cap P^2 = E(C)$.

($\leftarrow$) Assume a cycle $C \in CS(CG_f)$. A valid partition of the vertices $P \in \mathcal{P}(V(G))$ can be built from the vertices of each cycle $V(C) = P$, for each $C \in CS(CG_f)$. If this created partition is not the coarsest, then there is a cycle in $CS(CG_f)$ that is not a cycle, but a disjoint union of two or more cycles, which is absurd. By definition, for each $(v, u) \in E(C)$, the related partition configuration will be $f_P(v) = u$ and $f_P(v) = v$ for $v \in V(G) \setminus V(C)$. □

Note that for Equation 1.2 the order of the applications between the functions does not matter. Lemma 1.2.2 proves that this partition representation of a token configuration is equivalent to the cycle set representation. Partitioning swaps can be useful to determine which swaps can be run in parallel in variants of the token swap like Parallel Token Swap. The Corollary 1.2.2.1 shows an interesting interaction between tokens and swaps that can be used to optimize swap sequences.

Corollary 1.2.2.1. *Let S be an optimal sequence of swaps for a graph G and initial configuration f_0. Then, for any two distinct swaps $s_1, s_2 \in S$, they cannot interact with the same two tokens.*

Proof. Suppose that there is two swaps $s_1, s_2 \in S$ that exchange the same two tokens i and j and assume that s_1 appears before on the sequence than s_2. Now, take every swap of S (in-order) that are in-between s_1 and s_2 that exchange either i or j and call this new sequence $S_{i,j}$. Notice that, by removing the swap s_1 from S, every swap on $S_{i,j}$ that interacts with only i will now be interacting with j and vice-versa. Swaps that interact

with both tokens will continue interacting with both, but the position of the tokens in the vertices will be swapped.

Removing this swap does not affect any other token, as it doesn't matter which token is paired with the other tokens in the swaps. As the tokens on s_2 will now be already swapped, it is possible to just remove this swap. This second removal guarantees that the tokens i and j will be in the same position as the original sequence S, creating a new $\dot{S}$ that can solve the instance and have less swaps than S, which is absurd, as S is optimal. □

1.3 Graph Classes and Upper Bounds

This section is a summary of results regarding Token Swapping on specific graph classes and upper bounds on the numbers of swaps. Special focus will be given to graph classes that are related to cographs, as this class is the main focus of this dissertation.

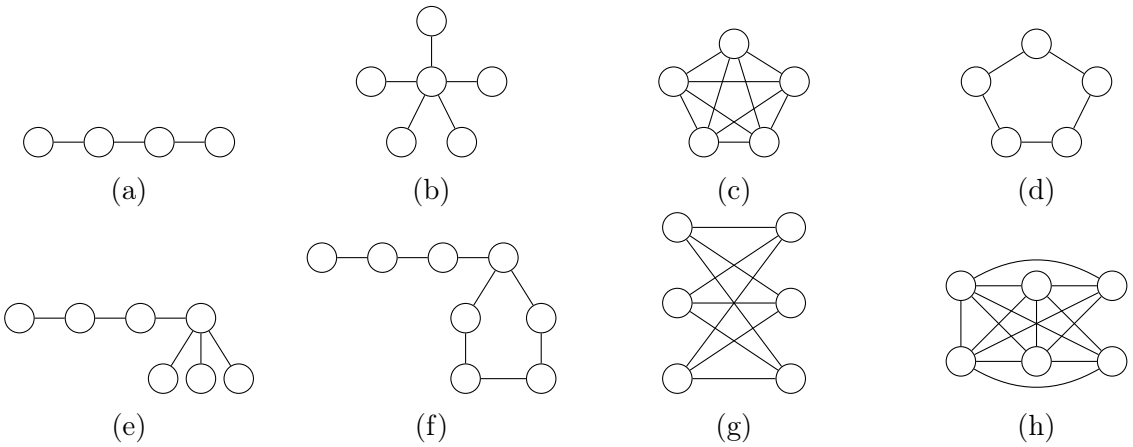


Figure 1.7: Graphs representing graph classes related to efficient algorithms for token swap problems. Figure 1.7a is a path of size 4; Figure 1.7b is a star; Figure 1.7c is a complete graph; Figure 1.7d is a cycle; Figure 1.7e is a broom graph; Figure 1.7f is a lollipop graph; Figure 1.7g is a complete bipartite graph; and Figure 1.7h is a complete split graph.

The problem of swapping tokens on paths, with an example in Figure 1.7a, goes back to the classic problem of ordering an array of integers by the exchange of adjacent numbers, which an efficient algorithm is known for quite some time [20, 24]. This algorithm, detailed by Algorithm 1, is a variation of the well known bubble-sort algorithm and the resulting number of swaps is well defined by the *inversion number*. Let f_0 be a mapping function for a path G and $i, j \in V(G)$. If $i <_M j$ and $f_0(i) >_M f_0(j)$, then either the elements (i, j) or the places $(f_0(i), f_0(j))$ are called an inversion of f_0. The size

of the set of all inversions of f_0 is called the inversion number. For any initial mapping function f_0, the minimum number of swaps is upper bounded by the maximum number of inversions (and thus, the diameter of the corresponding Cayley Graph is also bounded by this same value), which is $\theta(n^2)$.

Algorithm 1: Solving TS on path graphs

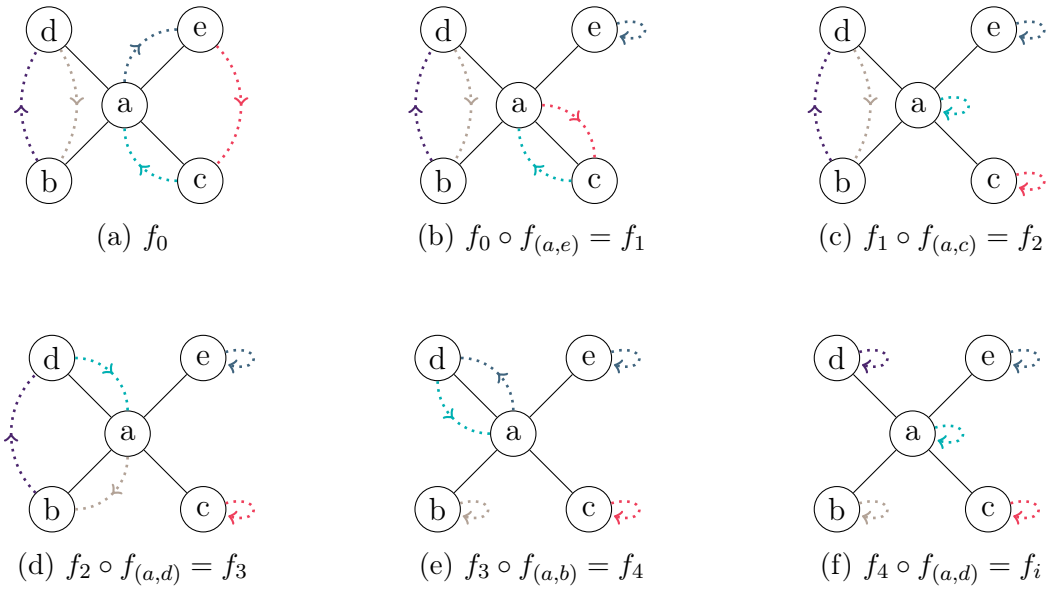
Input: G, f_0
Output: Swap Sequence S
$M = getOrder(G)$;
$bound = |V(G)|$;
$f = f_0$;
$S = ()$;
do
  $t = 0$;
  for $i = 1$ ***to*** $bound - 1$ **do**
    if $f(M^{-1}[j]) > f(M^{-1}[j])$ **then**
      $applySwap(G, f, (j, j+1))$;
      $addToSequence(S, (j, j+1))$;
      $t = j$;
    end
  end
while $t \neq 0$;
return S;

For cycle graphs, shown in Figure 1.7d, the method is an extension of the inversion number method of a path because the inversion number cannot be applied directly, as token can be moved either in a clockwise or anticlockwise manner to its target vertice. The optimal algorithm, presented by [20], first finds a feasible solution through an integer program and proves that this program can be calculated in polinomial time by restricting the search space to optimal solutions and then applying transformations to contract the resulting swap sequence until no other transformation is possible, resulting in a optimal swap sequence.

A broom graph is a graph such that the center of a star is connected to a path and a lollipop is a graph where a node from a cycle is connected to a path and are respectively shown in Figure 1.7e and Figure 1.7f. The proofs of correctness of the polynomial algorithms for brooms and lollipop graphs are similar and based on a evaluation function on the sum of the distances of each token to it's target vertice. Each swap alter this evaluation function by adding or subtracting the distances and the identity function is achieved if and only if the sum is zero. The polynomial algorithms are fairly simple and are presented by [22].

1.3.1 Token Swapping on Stars

A star, shown in Figure 1.7b, is a tree such that every vertice is a leaf except by the *center vertice* that is connected to every leaf vertice. In this graph, as cycles are disjoint, there is exactly one cycle such that the center vertice is a member of and can be solved in exactly the number of leafs in the cycle. For every other cycle, the minimum number of swaps to solve is the number of leafs in the cycle plus one, as it is necessary to move the center token to allow other tokens to achieve their destination. Figure 1.8 is an example of finding an optimal swap sequence on a star graph instance.



(a) f_0 (b) $f_0 \circ f_{(a,e)} = f_1$ (c) $f_1 \circ f_{(a,c)} = f_2$

(d) $f_2 \circ f_{(a,d)} = f_3$ (e) $f_3 \circ f_{(a,b)} = f_4$ (f) $f_4 \circ f_{(a,d)} = f_i$

Figure 1.8: Example of a optimal swap sequence S being applied to a token configuration on a star graph with four leafs. The joint representation identifies each token and its target vertice as a distinct colored arrow.

Based on this, the total number of swaps is given by $y + l$, where y is the total number of incorrect leafs and l is the number of cycles in the configuration that have size greater than 2 and the center vertex is not a part of [3, 7]. Note that, in this case, each cycle is solved separately and no interaction *between* the initial cycles is needed to achieve the identity configuration, also meaning that there is no specific order of swaps between cycles, as long the cycle with the center vertex is solved first or the center vertex is assumed correct for the other cycles. These past results also give an upper-bound on the diameter of the corresponding Cayley Graph.

1.3.2 Token Swapping on Cliques

Clique graphs, also called complete graphs, are graphs such that there is an edge connecting every pair of distinct vertices. The Figure 1.7c is an example of a complete graph with five nodes. In a clique graph with a token configuration, every token is either already correct or adjacent to its target vertex.

Then, for each cycle of size k, it is possible to solve it using exactly $k-1$ swaps, as every swap can move a token to its target vertex, except for the swap in a cycle of size 2 that solves two tokens at the same time. The total swaps needed for any clique is then $n-r$, where n is the total of vertices and r is the total of cycles and the upper bound $n-1$ is given when there is exactly one cycle on the configuration [23].

1.3.3 Token Swapping on Complete Split

A split graph is a graph such that its vertex set can be partitioned into a clique and an independent set, a set of vertices with no edges between any pair of vertices of the set. The Figure 1.7h is an example of a split graph with six nodes: A clique of size four and an independent set of size two. A complete split graph has every possible edge between the vertices of the clique and the independent set. For every cycle of a token swap instance on a complete split graph, the cycle must be either completely contained on the clique, the independent set or having vertices from both structures, as seen in Figure 1.9.

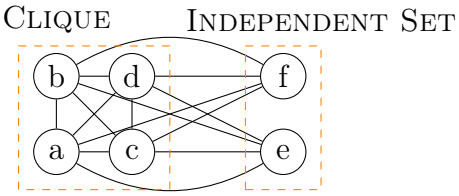


Figure 1.9: Any Token Swap instance with a cycle contained between vertices a, b, c and d can be solved with the same strategy described in Section 1.3.2. For other cycles that involve more than one vertex and vertices f and e, it is necessary to move the tokens from the independent set to the clique first.

Let k be the size of a cycle in a Token Swap instance on a complete split graph. If the cycle is completely contained on the clique, the cycle can be independently solved as a normal clique cycle with $k-1$ swaps. However, cycles contained in the independent set have no edges between them and every token must have to be moved to the clique

to be solved. The process guarantees a $k + 1$ sequence of swaps to solve the cycle. For cycles contained in both structures, it is possible to use a clique vertex to pivot every swap through it, similarly as solving a cycle that contains the center vertex of a star, in $k - 1$ swaps. The entire process is outlined in [40].

1.3.4 Token Swapping on Complete Bipartite

A complete bipartite graph, shown in Figure 1.7g, is a graph such that its vertex set can be partitioned into two independent sets with all edges between them. Similarly to complete split graphs, for an instance of token swap on a complete bipartite graph with partitions X and Y, the cycles can be divided into: a) It is completely contained in partition X; b) It is completely contained in partition Y; c) It has vertices from partitions X and Y; and d) Is a cycle of size one.

Cycles of type c) can be independently solved in $k - 1$ swaps and cycles of type d) are already correct. Although cycles of type a) and b) are contained in separate independent sets and can be solved efficiently in the way showed by Section 1.3.3 in $k+1$ swaps each, it is possible to save swaps by *merging* a cycle of type a) and b) [38]. This notion of merging cycles to save swaps will be generalized into cographs, which form a superclass of complete bipartite graphs, in Section 2.2.

Chapter 2

Solving Token Swap on Cographs

A cograph is defined recursively as follows:

- A graph with a single vertex is a cograph;
- If $G_1, G_2, ..., G_k$ are cographs, then so is their disjoint union;
- if G is a cograph, then so is its complement $\overline{G}$.

Some authors also describe another rule called the *join* operation to build cographs. Given two cographs G_i and G_j, a join $join(G_i, G_j)$ results in a graph $\dot{G}$ such that $V(\dot{G}) = V(G_i) \cup V(G_j)$ and $E(\dot{G}) = E(G_i) \cup E(G_j) \cup \{uv \mid u \in V(G_i), v \in V(G_j)\})$.

This operation can be described by the basic cograph operations $join(G_i, G_j) = \overline{\overline{G_i} \cup \overline{G_j}}$, resulting in a valid operation to build cographs.

A cotree $CT(G)$ of a cograph $G = (V, E)$ is a rooted tree representing its structure. The leaves of $CT(G)$ are exactly V and each internal node is labelled 0 or 1 and will be called 0-node and 1-node, respectively. Each 1-node represents a join and each 0-node represents a disjoint union between the graphs represented by its subtrees.

The children of an 1-node are 0-nodes or leaves and the children of a 0-node are 1-nodes or leaves. Two vertices are adjacent in a cograph if and only if their lowest common ancestor (LCA) in the cotree is a 1-node. The cotree of any particular cograph is unique. Figure 2.1 shows a simple example of a cograph and its cotree representation.

(a) (b)

Figure 2.1: a) Example of a cograph with labeled nodes; b) Cotree trat represents the structure of the cograph 2.1a.

Let G be a cograph and $CT(G)$ the respective cotree. Let f_0 be a token placement on G and CG the related conflict graph with set of permutation cycles CS. Let CS^0 be the set of all cycles $C \in CS$ such that the lowest common ancestor in the cotree of G of the vertices of the cycle is a 0-node of the cotree. Let CS^1 be the set of all cycles $C \in CS$ such that the lowest common ancestor in the cotree of G of the vertices of the cycle is a 1-node of the cotree, or the cycle is a self-loop. These two sets form a partition of the set of cycles of the Conflict Graph and will be called the zero-cycles set and the one-cycles set respectively. Figure 2.2 shows the cotree in Figure 2.1 with a zero-cycle and a one-cycle.

By looking at most of the classes presented at Section 1.3 that have a polinomial time algorithm, it is possible to notice that most of this classes (stars, cliques, complete bipartite and complete split graphs) are subclasses of the cograph class. So, it seemed logical to begin inquiring into other subclasses of cographs for a more definitive pattern that could be used for the more general class. From this intuition, the class of threshold graphs were the first chosen as research subject, as it is both a superclass of complete split and a subclass of the cographs. From the results that were found emerged an interesting pattern that could be generalized for the cograph class and will be presented in this chapter.

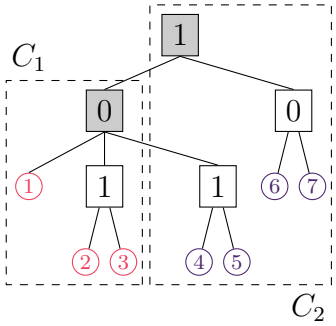


Figure 2.2: The cotree of the graph showed in Figure 2.1 is being used. Let f be a configuration where nodes 1,2,3 are part of a permutation cycle C_1 and nodes 4,5,6,7 are part of another permutation cycle C_2, without paying attention to the exact cycle configuration. The lowest common ancestor of each cycle is denoted as a gray rectangle inside each corresponding labeled rectangle. The respective partitions are $\mathcal{P}(C_1) = \{\{2,3\},\{1\}\}$ and $\mathcal{P}(C_2) = \{\{4,5\},\{6,7\}\}$ respectively.

2.1 Solving Individual Permutation Cycles

The following two lemmas show how an individual permutation cycle C can be solved with $|C| - 1$ if it is a one-cycle and with $|C| + 1$ if it is a zero-cycle without affecting the rest of the configuration.

Lemma 2.1.1. *Any cycle C of CS^1 can be solved in $|C| - 1$ swaps.*

Proof. We proceed by induction on $|C|$. As base case consider $|C| < 3$. If $|C| = 1$ the cycle is already solved and $1 - 1 = 0$ swaps were used. If $|C| = 2$ the cycle can be solved with $2 - 1 = 1$ swap by swapping the two nodes of the cycle as they are adjacent in G.

Now we consider a one-cycle C of size at least 3. Let T be the subtree of $CT(G)$ rooted at $LCA_{G^r}(C)$. Notice that in the at least two subtrees rooted on a child of the root of T that have at least one vertex of C there is a vertex $v \in C$ whose predecessor u in C belongs to a subtree rooted in a different child of T. Then, a swap between u and v (we know they are adjacent in G because their LCA is a 1-node) splits C into one self-loop on v and a cycle $\dot{C}$ with size $|C| - 1$.

To ensure that the obtained $\dot{C}$ is a one-cycle we can choose v in such a way that $LCA_{G^r}(\dot{C}) = LCA_{G^r}(C)$. With that aim, we chose v such that its removal would keep vertices of $\dot{C}$ in at least two subtrees rooted in different children of T. This is always possible as C has at least 3 vertices.

Then, the inductive hypothesis is applied on $\dot{C}$. Since $\dot{C}$ can be solved in $|C| - 2$ swaps, C is solved in $C - 1$ swaps.

□

Lemma 2.1.2. *Any cycle C of CS^0 can be solved in $|C| + 1$ swaps.*

Proof. As G is connected, there must exist at least one ancestor of $LCA_{G^r}(C)$ in $CT(G)$ that is a 1-node and connects the graph (or more locally, the nodes of C). Let the nearest ancestor node be called w and let T be the subtree of $CT(G)$ rooted at w. All the vertices in C are contained in a subtree rooted on a child of T and there must exist a vertex u that belongs to a subtree rooted in a differnet child of T. Then, since T is rooted on a 1-node, u must dominate $V(C)$.

Assume that u already has a correct token in the current configuration. We now merge it into C by using any swap (u, v), $v \in V(C)$, and call this new cycle $\dot{C}$. In this new cycle, the 1-node w is the lowest common ancestor and the technique presented at Lemma 2.1.1 can be used to solve $\dot{C}$ in $|\dot{C}| - 1$ swaps, which is equivalent to $|C|$ swaps, as there is one vertex more. In total, the number of swaps is $|C| + 1$ because of the first swap needed to add u to the cycle.

Notice that as the token of u is assumed to be correct, its token went back to the same place as before when the sequence of swaps is applied. This means that in fact, the token currently in u does not need to be the correct token. We can just assume it is correct for applying the swaps indicated in this Lemma. □

With both of the methods presented, all permutation cycles of any TS instance on a cograph G can be solved with $|V(G)| - |CS^1| + |CS^0|$ swaps. Now, next lemmas will carry on with the analysis of the optimality of each method. First, it is possible to

show that the methods derived form the lemmas achieve the minimum number of swaps for individual cycles by Lemma 2.1.3 and Lemma 2.1.6.

Lemma 2.1.3. *Individually, any cycle C of CS^1 cannot be solved in less than $|C| - 1$ swaps.*

Proof. A solved instance (an identity mapping) has $|C|$ self-loop cycles on the vertices of C. Since each split swap increases the number of cycles by 1, at least $|C| - 1$ split swaps are needed to transform the single cycle C into $|C|$ cycles. □

Lemma 2.1.4. *Let G be a cograph with a token placement f. Let C_1 and C_2 be the two cycles resulting from any split swap on a zero-cycle C. Either C_1 or C_2 must be a cycle with exactly the same lowest common ancestor as the original cycle C.*

Proof. Let T be the subtree of CG rooted on $LCA_{G^r}(C)$. By definition of the LCA, the vertices in C are placed in at least two subtrees rooted on different children of $LCA_{G^r}(C)$ in T. If they belong to more than two of those subtrees, then when C is split into C_1 or C_2, by the pigeon-hole principle [14, 30], either C_1 or C_2 (or both) have vertices in more than one of those subtrees and it has the same lowest common ancestor as C.

Assume now that the vertices in C are placed in exactly two subtrees rooted on children of $LCA_{G^r}(C)$ and that a single inner-swap divides C in such a way that C_1 is fully contained in the subtree rooted on one child of $LCA_{G^r}(C)$ and C_2 is fully contained in another subtree rooted on another child of $LCA_{G^r}(C)$. Notice that this is the only way in which the two new created cycles have lowest common ancestors that differ from the one of C. Such a swap must necessarily involve vertices from both subtress of $LCA_{G^r}(C)$ as swaps between two vertices on the same subtree do not change the number of arcs of the cycle that cross from one child to the other. Now, notice that such a swap is impossible because $LCA_{G^r}(C)$ is a 0-node. □

Lemma 2.1.5. *Let G be a cograph with a token placement f and let C be a zero-cycle. No split-swap can transform the C into two one-cycles.*

Proof. By Lemma 2.1.4 at least one of the two cycles obtained after the split-swap maintains the same least common ancestor of the original cycle. Then it is still a zero-cycle. □

Lemma 2.1.6. *Individually, any cycle $C \in CS^0$ cannot be solved in less than $|C| + 1$ swaps.*

Proof. By Lemma 2.1.5, applying just split swaps to C there will always remain at least one zero-cycle. Then in order to solve the cycle at least one merge-swap is needed. Each merge swap decreases the number of cycles in CG by one. Then, at least one merge swap and $|C|$ splits swaps are needed for the single cycle C to be transformed into $|C|$ self-loops. Therefore, at least $C + 1$ swaps are needed. □

This model of solving cycles can be used to prove the optimality of some subclasses of the cograph class. In complete graphs, every permutation cycle belongs to CS^1, as every vertex is connected to every other, resulting in an empty CS^0 set. In star graphs, the number of cycles $C \in CS^1$ with $|C| > 1$ cannot be more than 1, as all nodes are only neighbors of the center node and the center node can be only part of one-cycle, resulting in every other non-trivial cycle belonging to CS^0. These two classes can be generalized to the class of threshold graphs, that will be considered in Lemma 2.1.9. There are many equivalent definitions for this class, but, for the sake of conciseness, we defined a threshold graph as a graph that can be constructed from a single vertex graph by repeatedly adding either an isolated vertex or a dominating vertex. Also, a threshold graph is a graph free of induced cycles of size four (C_4), induced paths of size four (P_4) and induced two disjoint edges ($2K_2$).

Lemma 2.1.7. *Let G be a threshold graph with an initial token placement f_0. Then, $OPT_G(f_0) \leq |V(G)| - |CS^1(CG_f)| + |CS^0(CG_f)|$.*

Proof. Directly from Lemma 2.1.3 and Lemma 2.1.6. □

Lemma 2.1.8. *Let G be a threshold graph with an initial token placement f_0. Then, $OPT_G(f_0) \geq |V(G)| - |CS^1(CG_{f_0})| + |CS^0(CG_{f_0})|$.*

Proof. Let $p(G, f) = |V(G)| - |CS^1(CG_f)| + |CS^0(CG_f)|$. Notice that $p(G, f_\iota) = 0$ as $|CS^1(CG_{f_\iota})| = |V(G)|$ and $|CS^0(CG_{f_\iota})| = 0$. In the following, we show that any possible swap transforming f into an adjacent configuration $\dot{f}$ cannot decrease the value of $p(G, f)$ in more than 1. That is, $p(G, \dot{f}) \geq p(G, f) - 1$.

Recall that every swap, either split a cycle C or merge two cycles C_u and C_v in CG_f. Then, there are two possibilities to decrease $p(G, \dot{f})$ by more than one: (1) a zero-cycle is split into two one-cycles or (2) two zero-cycles are merged into one one-cycle.

The first case (1) is not possible since contradicts Lemma 2.1.5.

For the second case (2), observe that the fact that the merged cycle is an one-cycle implies that $LCA_{G^r}(C_u) \neq LCA_{G^r}(C_v)$, $LCA_{G^r}(C_u)$ is not an ancestor of $LCA_{G^r}(C_v)$ and $LCA_{G^r}(C_v)$ is not an ancestor of $LCA_{G^r}(C_u)$. In all these cases, the merged cycle would remain a zero-cycle. Let T be the subtree of CG rooted on $LCA_{G^r}(C_u \cup C_v)$.

The vertices of C_u and those of C_v lie in subtrees rooted in different children of T and since the root of T is a 1-node, all vertices in C_u are adjacent to all vertices in C_v in G. Now, take any two non-adjacent vertices x, y from $V(C_u)$ and two non-adjacent vertices w, z from $V(C_v)$ (they must exist on both cycles as they are zero-cycles).

Then, the induced subgraph $G[\{x, y, w, z\}]$ is a cycle of size 4. This is a contradiction on the definition of threshold graphs.

By the above analysis, it is possible to conclude that any token swap decreases $p(G, f)$ by at most one unit for any token placement f and obtain the Inequation 2.1.

$$p(G, \dot{f}) \geq p(G, f) - 1 \tag{2.1}$$

Thus, for any swapping sequence $S = (s_1, s_2, \ldots, s_k)$ that transforms the initial configuration f_0 to the identity configuration f_ι through adjacent configurations $f_1, f_2, \ldots, f_k = f_\iota$, each pair of configurations $p(G, f_{j+1}) \geq p(G, f_j) - 1$ holds from Inequation 2.1 for $j = 1, 2, \ldots, k-1$. Take the sum of these inequations $\sum_j p(G, f_{j+1}) \geq p(G, f_j) - 1$ shown in Inequation 2.2.

$$\begin{array}{lcl} p(G, f_1) & \geq & p(G, f_0) - 1 \\ p(G, f_2) & \geq & p(G, f_1) - 1 \\ \ldots & & \\ p(G, f_{k-1}) & \geq & p(G, f_k) - 1 \\ \hline p(G, f_k) & \geq & p(G, f_0) - |S| \end{array} \tag{2.2}$$

From inequation 2.2 and substituing $p(G, f_k)$ by 0 we get:

$$\begin{array}{lcl} p(G, f_k) & \geq & p(G, f_0) - |S| \\ |S| & \geq & p(G, f_0) - p(G, f_k) \\ |S| & \geq & |V(G)| - |CS^1(CG_f)| + |CS^0(CG_f)| \end{array} \tag{2.3}$$

□

Theorem 2.1.9. *LLet G be a threshold and f_0 an initial token placement for the Token Swap problem. The optimal number of swaps is given by $|V(G)| - |CS^1(CG_f)| + |CS^0(CG_f)|$.*

Proof. Directly from Lemma 2.1.7 and Lemma 2.1.8. □

2.2 Dependencies Betweeen Permutation Cycles

Section 2.1 introduced techniques for solving cycles of the two types individually without changing the configurations of other cycles and proved the optimality of this method for threshold graphs. The proof for general cographs is constructed from the one for threshold graphs, observing that general cographs allow induced cycles of size

4. A merge of two zero-cycles into a one-cycle can save exactly two swaps in the final configuration. Let this type of swap be called a *cutback* swap.

A *Cycle Matching Graph* $H = (CS^0, E_H)$ of a cograph G and initial token configuration f_0 is a graph where each node represents an individual zero-cycle of the current configuration. Let $u, v \in V(H)$ and C_u, C_v be the two related cycles from CS^0. An edge will exist between two nodes u, v if and only if the lowest common ancestor of the union of the vertices $V(C_u) \cup V(C_v)$ is a one-node. This graph will be used to identify the cases where two swaps can be saved.

Lemma 2.2.1. *Let G be a graph and $\mu(G)$ a maximum edge matching of G.*

1. *Adding a vertex connected by edges to the graph G can increase the size of the matching $|\mu(G)|$ in at most one;*

2. *Adding edges to a single vertex can increase the size of the matching $|\mu(G)|$ in at most one;*

3. *Let $\dot{G}$ and $\ddot{G}$ be graphs such that $G \subseteq \dot{G} \subseteq \ddot{G}$, $|V(\dot{G}) \backslash V(G)| = 1$ and $|V(\ddot{G}) \backslash V(\dot{G})| = 1$. Then, $|\mu(\ddot{G})| \leq |\mu(G)| + 2$.*

Proof. The proofs are respectively enumerated below.

1. Let $\dot{G}$ be the graph obtained by adding to G a vertex x connected to any subset of V and let $\mu(\dot{G})$ be a maximum edge matching of $\dot{G}$. Suppose x is saturated in $\mu(\dot{G})$ and $|\mu(\dot{G})| > |\mu(G)|$. $|\mu(\dot{G})|$ must be $|\mu(G)| + 1$, otherwise the size of the maximum matching $|\mu(G)|$ results in a contradiction, as it is possible to use the matching $|\mu(\dot{G})|$ minus the edge involving x, obtaining a larger matching of G. If x is not saturated, then $|\mu(\dot{G})| = |\mu(G)|$.

2. Let $\dot{G}$ be the graph obtained by adding to G any amount of edges involving a vertex $v \in V(G)$ and let $\mu(\dot{G})$ be a maximum edge matching of $\dot{G}$. Assume $|\mu(\dot{G})| > |\mu(G)| + 1$. The matching obtained by removing the edge related to v in $\mu(\dot{G})$ (if it exists) is a valid matching on the original graph G and has size at least $|\mu(G)| + 1$, which contradicts the maximality of the matching of $\mu(G)$.

3. By part (1) of this lemma, the size of the matching $\mu(\dot{G})$ can be at most 1 more than the size of $\mu(G)$, and the size of the matching $\mu(\ddot{G})$ can be at most 1 more than the size of $\mu(\dot{G})$. Combining both inequalities we get:

$$\mu(\dot{G}) \leq \mu(G) + 1 \tag{2.4}$$
$$\mu(\ddot{G}) \leq \mu(\dot{G}) + 1 \tag{2.5}$$
$$\mu(\ddot{G}) \leq \mu(G) + 2 \tag{2.6}$$

□

Let $\mu(H)$ be a maximum edge matching on the graph H. Each element of this matching represents a possible merge swap that can save two swaps in the final swap sequence (by the cost of one swap, two zero-cycles are transformed into one one-cycle), saving in total $2 \times |\mu(H)|$ swaps. Note that after applying these swaps, the resulting configuration will have a totally disconnected Cycle Matching Graph, as only zero-cycles unsaturated by $\mu(H)$ will remain.

Lemma 2.2.2. *Let G be a cograph with an initial token placement f_0. Then, $OPT_G(f_0) \leq |V(G)| - |CS^1(CG_f)| + |CS^0(CG_f)| - 2 \times |\mu(H)|$.*

Proof. Directly from Lemma 2.1.3, Lemma 2.1.6 and the previous discussion. □

Lemma 2.2.3. *Let G be a cograph with an initial token placement f_0. Then, $OPT_G(f_0) \geq |V(G)| - |CS^1(CG_f)| + |CS^0(CG_f)| - 2 \times |\mu(H)|$.*

Proof. Let $p(G, f) = |V(G)| - |CS^1(CG_f)| + |CS^0(CG_f)| - 2 \times |\mu(H)|$. Note that $p(G, f_\iota) = 0$ holds. In the following we show that any possible swap transforming f into an adjacent configuration $\dot{f}$ cannot decrease the value of $p(G, f)$ in more than 1. That is, $p(G, \dot{f}) \geq p(G, f) - 1$.

The following cases encompass every possible split swap of vertices u and v on the given configuration.

- **Case SPLIT-1:** Let u and v belong to the same cycle $C \in CS^1$. This token swap breaks the original cycle into two vertex disjoint cycles C_u and C_v. Assume that $u \in V(C_u)$ and $v \in V(C_v)$.

  Case (A): If $C_u \in CS^1$ and $C_v \in CS^1$, the value of $|CS^1|$ is increased in 1 and the size of the matching $\mu(H)$ does not change, as graph H is not modified, resulting in $p(G, \dot{f}) = p(G, f) - 1$;

  Case (B): If $C_u \in CS^0$ and $C_v \in CS^1$ or if $C_u \in CS^1$ and $C_v \in CS^0$, the value of $|CS^0|$ is increased by 1, increasing the number of vertices in H by one. By Lemma 2.2.1, the matching can increase in at most one unit, resulting in $p(G, f) - 1 \leq p(G, \dot{f}) \leq p(G, f) + 1$;

Case (C): If $C_u \in CS^0$ and $C_v \in CS^0$, the value of $|CS^0|$ is increased by 2 and $|CS^1|$ is decreased by 1, increasing the number of vertices in H in two with a guaranteed additional edge between C_u and C_v as $LCA_{CG(G)}(V(C_u \cup C_v))$ is a 1-node. Then, $|\mu(H)|$ increases at least 1 and by Lemma 2.2.1, the increase is at most 2, resulting in $p(G, f) - 1 \leq p(G, \dot{f}) \leq p(G, f) + 1$.

- **Case SPLIT-0:** Let u and v belong to the same cycle $C \in CS^0$. This token swap breaks the original cycle into two vertex disjoint cycles C_u and C_v. Assume that $u \in V(C_u)$ and $v \in V(C_v)$.

  Case (A): Let $C_u, C_v \in CS^1$. This case is impossible as stated by Lemma 2.1.5.

  Case (B): If $C_u \in CS^0$ and $C_v \in CS^1$ or if $C_u \in CS^1$ and $C_v \in CS^0$, the value of $|CS^1|$ is increased by one and $|CS^0|$ does not change. By Lemma 2.1.4, $LCA_{CG(G)}(V(C_u)) = LCA_{CG(G)}(V(C))$, not changing the graph H, resulting in $p(G, \dot{f}) = p(G, f) - 1$;

  Case (C): If $C_u \in CS^0$ and $C_v \in CS^0$, the value of $|CS^0|$ is increased by 1 and $|CS^1|$ does not change. By Lemma 2.1.4, either C_u or C_v have the same lowest common ancestor as the cycle C. That cycle introduces in H a vertex with the same neighborhood as the vertex corresponding to C that is being removed. Then, in practice, exactly one new vertex is added to H. Lemma 2.2.1 shows that the maximum matching can increase in at most one in this case, resulting in $p(G, f) - 1 \leq p(G, \dot{f}) \leq p(G, f) + 1$.

It is no needed to check the merge swaps in this case, as each of them is just the inverse of a split swap. This means that the previous equations and inequations have inverted plus and minus signals. No equation or inequation show and increase larger than 1, so no merge swap can decrease $p(G, f)$ in more than 1.

By the above analysis, it is possible to conclude that any token swap decreases $p(G, f)$ by at most one for any token placement f and obtain:

$$p(G, \dot{f}) \geq p(G, f) - 1 \tag{2.7}$$

Thus, for any swapping sequence $S = (s_1, s_2, \ldots, s_k)$ that transforms the initial configuration f_0 to the identity configuration f_ι through adjacent configurations $f_1, f_2, \ldots, f_k = f_\iota$, for each pair of configurations $p(G, f_{j+1}) \geq p(G, f_j) - 1$ holds. Taking the sum of these inequations $\sum_j p(G, f_{j+1}) \geq p(G, f_j) - 1$ we get:

$$\begin{array}{lcl} p(G, f_1) & \geq & p(G, f_0) - 1 \\ p(G, f_2) & \geq & p(G, f_1) - 1 \\ \dots & & \\ p(G, f_{k-1}) & \geq & p(G, f_k) - 1 \\ \hline p(G, f_k) & \geq & p(G, f_0) - |S| \end{array} \tag{2.8}$$

From inequation 2.8 and substituting $p(G, f_k)$ for a 0, we get:

$$\begin{array}{lcl} p(G, f_k) & \geq & p(G, f_0) - |S| \\ |S| & \geq & p(G, f_0) - p(G, f_k) \\ |S| & \geq & |V(G)| - |CS^1(CG_f)| + |CS^0(CG_f)| - 2 \times |\mu(H)| \end{array} \tag{2.9}$$

□

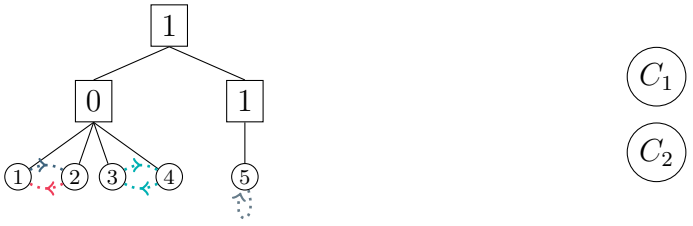
Theorem 2.2.4. *Let G be a cograph and f_0 an initial token placement for the Token Swap problem. The optimal number of swaps is given by $|V(G)| - |CS^1(CG_f)| + |CS^0(CG_f)| - 2 \times |\mu(H)|$.*

Proof. Directly from Lemma 2.2.2 and Lemma 2.2.3. □

Corollary 2.2.4.1. *Let G be a cograph and f_0 an initial token placement for the Token Swap problem. Denote by $g(G, f_0)$ the value associated with this instance as defined in Theorem 2.2.4. Then*

$$g(G, f_0) = O(|V(G)|) \tag{2.10}$$

Proof. Let G be a cograph and f_0 an initial placement for the Token Swap problem, as illustrated in Figure 2.3a. The corresponding Conflict Graph H, shown in Figure 2.3b, is built from cycles numbered from left to right.



(a) (b)

Figure 2.3: Example cotree from a cograph and the related Conflict Graph.

By generalizing the cograph G, it is possible to add an arbitrary number of vertices on the left side of the cotree, while creating the smallest possible zero-cycles between them (size of two or three nodes depending on whether the total number is even or odd), keeping the Conflict Graph disconnected, ensuring that no match is possible. This results in Equation 2.11.

$$\begin{aligned} g(G, f_0) &= |V(G)| - |CS^1(CG_f)| + |CS^0(CG_f)| - 2 \times |\mu(H)| \\ g(G, f_0) &= |V(G)| - 1 + \left\lceil \tfrac{|V(G)|}{2} \right\rceil - 2 \times 0 \\ g(G, f_0) &= |V(G)| + \left\lceil \tfrac{|V(G)|}{2} \right\rceil - 1 \\ g(G, f_0) &= O(|V(G)|) \end{aligned} \tag{2.11}$$

□

Algorithm 2: Solving TS on cographs

Input: G, f
Output: Swap Sequence S

1. Compute the cotree of G.
2. Compute the conflict graph CG_f.
3. For each cycle in CG_f determine if it is a zero-cycle or a one-cycle.
4. Compute H using the zero-cycles from CG_f.
5. Compute a maximum matching in H.
6. For each pair of zero-cycles matched in the computed maximum matching, perform a cutback swap using any edge between vertices of both cycles.
7. Solve the remaining zero-cycles with the procedure implied by the proof of Lemma 2.1.2 and the remaining one-cycles with the procedure implied by the proof of Lemma 2.1.1.
8. Return resulting swap sequence as S.

Theorem 2.2.5. *Token Swap Problem can be solved in polynomial time in cographs.*

Proof. Let G be a cograph and f be a token configuration. By the previous lemmas, the Algorithm 2 solves the Token Swap Problem in G with initial configuration f. Since each step of the algorithm can be performed in polynomial time, the whole algorithm can be executed in polynomial time. □

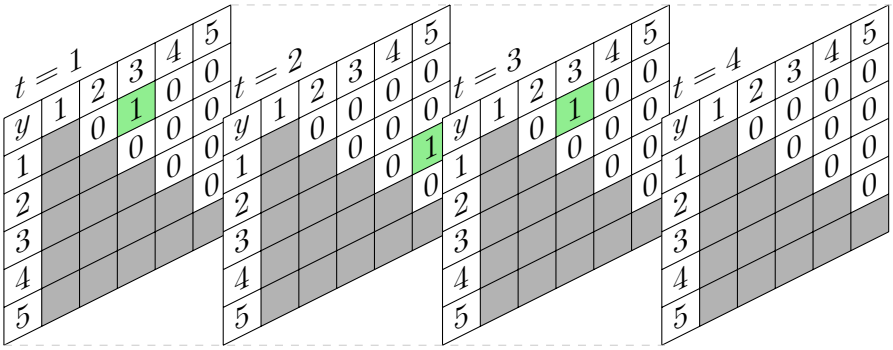
Chapter 3

Integer Linear Programming Models

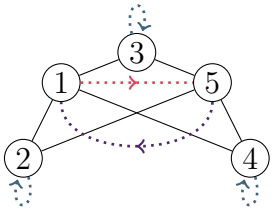
This chapter presents two integer linear programming models for the Token Swap problem and the Parallel Token Swap problem, respectively, along with an explanation of each variable and constraint for each model.

3.1 The TS Problem Formulation as an Integer Programming Problem

The binary variables x_{iut} determine if a token i is at node u in step t. To correct model the bijections between tokens and vertices, Equation 3.3 enforces that any token can be at most in one vertex and Equation 3.4 that a vertex can have at most one token. The binary variables y_{uvt} flags if a swap happened between nodes u and v in step t. It is important to note that swaps are symmetric —i.e., a swap (u, v) for $u, v \in V$ is exactly the same as swap (v, u) —, so variables y_{uvt} and y_{vut} means the same swap. On those grounds, and the fact that our graph is undirected, a technique that creates an ordering $M : V \mapsto [n]$ of the vertices of the graph and use the variables y_{uvt} such that $u <_M v$ can be adopted, halving the number of variables needed to represent a swap. The ordering M is assumed to exist throughout the dissertation for any graph.



(a)



(b)

Figure 3.1: Here is an example of how the variable y_{uvt} can model a sequence of swaps. Let $T = 4$ and the TS instance presented at Figure 3.1b. The minimum number of swaps needed to bring all tokens to its correct positions is three, more precisely $(1, 3), (3, 5), (3, 1)$. At Figure 3.1a it is shown how these swaps are represented (in green) with the fourth panel having no swaps. Note that the swap symmetry is being used to represent swaps using the upper diagonal of the matrix.

The Figure 3.1 is an example of a TS instance and a complete representation of an optimal swap sequence using y_{uvt}. To retrieve the final sequence, it is necessary to traverse each matrix in step order and check each y_{uvt}, appending (u, v) to the swap sequence whenever $y_{uvt} = 1$. Steps with no $y_{uvt} = 1$ can be ignored. The total number of variables is given by Equation 3.1.

$$\underbrace{\frac{T \times n^2 - T \times n}{2}}_{y_{uvt}} + \underbrace{T \times n^2}_{x_{iut}} \tag{3.1}$$

The TS problem model given by Formulation (3.2)-(3.13) search viable solutions of the problem with a given upper-bound in the number of swaps T, allowing at most one swap per step $t \in [T]$. Each step is composed of a set of variables that describe the current configuration, which swap is being selected and Equation 3.11 checks if a swap sequence solves the current instance. The constant T can be calculated by using any of the best approximation algorithms, or by using the trivial upper-bound $O(n^2)$ on the size of an optimal swap sequence mentioned in Chapter 1.

$$\text{minimize} \quad \sum_{\forall uv \in E, u <_M v, \forall t \in [T]} y_{uvt} \tag{3.2}$$

$$\text{subject to} \quad \sum_{\forall u \in V} x_{iut} = 1 \qquad \forall t \in [T], \forall i \in V \tag{3.3}$$

$$\sum_{\forall i \in V} x_{iut} = 1 \qquad \forall t \in [T], \forall u \in V \tag{3.4}$$

$$x_{iut} + x_{ivt+1} \leq y_{uvt} + 1 \qquad \forall i \in V, \forall t \in [T-1], \forall uv \in E, u <_M v \tag{3.5}$$

$$x_{ivt} + x_{iut+1} \leq y_{uvt} + 1 \qquad \forall i \in V, \forall t \in [T-1], \forall uv \in E, u <_M v \tag{3.6}$$

$$x_{iut} + x_{ivt+1} \leq 1 \qquad \forall i \in V, \forall t \in [T-1], \forall uv \notin E \tag{3.7}$$

$$\sum_{\forall u,v \in V, u <_M v} y_{uvt} \leq 1 \qquad \forall t \in [T] \tag{3.8}$$

$$\sum_{\forall uv \in E, u <_M v} y_{uvt} \geq \sum_{\forall uv \in E, u <_M v} y_{vwt+1} \qquad \forall t \in [T-1] \tag{3.9}$$

$$x_{i,f_0(i),0} = 1 \qquad \forall i \in V \tag{3.10}$$

$$x_{iiT} = 1 \qquad \forall i \in V \tag{3.11}$$

$$y_{uvt} \in \{0,1\} \qquad \forall t \in [T], \forall uv \in E \tag{3.12}$$

$$x_{iut} \in \{0,1\} \qquad \forall i \in V, \forall u \in V, \forall t \in [T] \tag{3.13}$$

These approximation algorithms use the same intuitive lower-bound given by

$$\sum_{v \in V} \frac{\delta(f_0(v), v)}{2}$$

where $\delta(u, v)$ is the distance from vertex u to vertex v in the graph G and the division by 2 happens since every swap can decrease the distance between tokens by at maximum two in the sum. This bound is tight for instances of TS that can be solved with only swaps that decrease the distance sum by two [8]. These values give us a good notion of the range of values that T can assume. More detailed explanations of each constraint are presented in the subsequent Section 3.1.1.

3.1.1 Modelling Swaps and Initial Token Configuration

The constraints (3.5)-(3.8) are being used to model the behavior of swaps. For a swap y_{uvt} to be possible, it is required that in step t the token on vertex u is on vertex v

on step $t+1$, given by Inequality 3.5. Similarly, on Inequality 3.6, it was guaranteed that the second token that is on vertex v on step t must be on vertex u on step $t+1$. Note that each pair of vertices $u, v \in V$ were directly chosen from the edge set E, ensuring that an edge must exist. Inequality 3.7 is there to keep the consistency between each pair of vertices that do not have an edge in between, as it is impossible to have a token traverse both in one step. Inequality 3.9 restrict the symmetry of the model, eliminating steps with no swaps followed by a step with a swap. Inequality 3.8 force each step to have at maximum one swap, while also permitting steps with no swaps.

The Inequality 3.10 creates the necessary constraints to represent the initial configuration, setting the necessary variables to one. The step $t = 0$ is fixed on the model for any given configuration f_0, as there must exist one token on each vertice initially. In Section 3.2 a more in-depth discussion about a parallel variant of the TS problem that do not need to guarantee one swap per step will be presented.

3.2 The Parallel TS Problem Formulation as an Integer Programming Problem

The Parallel Token Swap (PTS) problem is a version of TS where swaps can be applied in parallel. For a graph $G := (V, E)$, a *parallel swap* of a set of parallel swaps $P = \{s_1, s_2, \ldots, s_k\}$ is a swap s such that no other swap in P shares a vertex with s. Any swap sequence S that solves an instance of PTS or TS can be partitioned into sequences of parallel swaps $\mathcal{P}(S) = (P^0, P^1, \ldots, P^b)$. The swaps of each partition can be applied to a token placement in any order without changing any intermediate step in S (hence the use of set) or it can be applied by just one representation function f_P, as all swaps in a partition have disjoint vertices. Equation 3.14 shows how this representation function can be built. A partitioned sequence of parallel swaps solves an instance of TS or TSP if and only if the identity function is resulted by applying each partition iteratively, while the swap application order of each partition does not matter.

$$f_P = f_{s_1} \circ f_{s_2} \circ \ldots \circ f_{s_k} \tag{3.14}$$

It is important to note that the size of any partition has a natural upper-bound on the size of the Maximum Matching of the graph G, as it is the maximum number of edges that can be swapped in the graph and still be disjoint. Calculating the maximum matching size in general graphs can be done in polynomial time [13]. Therefore, for graphs with maximum matching size of 1, an instance of PTS has the same set of optimal swap

sequences of the equivalent TS instance. For the general case, the size of a swap sequence that solves a TS instance can be used as an upper-bound for an equivalent PTS instance. This variant of the TS problem has a deeper relation to parallel sorting on SIMD machines consisting of several processors with local memory connected by a network [38, 22] and is better suited for some reconfiguration problems, like Qubit Allocation [32, 31].

One interesting question that naturally arises from this definition is: "what is the swap sequence that solves a given PTS instance and minimizes the number of partitions in $|\mathcal{P}(S)|$?". In the decision version, the existence of at least one swap sequence that solves a PTS instance with k or less partitions is wanted. This version of the problem has already been proven to be **NP-Complete** for general k, but can be calculated in polynomial time if $k \leq 2$ and have an approximation algorithm for paths [22].

Each step is now related to a partition and a new binary variable for each step called s_t were added to keep track of which steps have *at least* one $y_{uvt} = 1$ by using Inequality 3.22. Note that the number of checks is improved by only verifying variables y_{uvt} such that $\{u, v\} \in E$, as swaps can only exist on the edges. These flags help counting how many steps are using at least one swap, while any step that do not need to be used to achieve a minimal swap sequence will not be counted, as $s_t = 0$. This variable and the fact that each step is now allowed to have any number of parallel swaps allows for the minimization of s_t in the minimization rule given by Equation 3.15, which is equivalent to minimizing the number of partitions. The total number of variables is increased by T from Equation 3.1.

$$\text{minimize} \quad \sum_{\forall t \in [T]} s_t \tag{3.15}$$

$$\text{subject to} \quad \sum_{\forall u \in V} x_{iut} = 1 \qquad \forall t \in [T], \forall i \in V \tag{3.16}$$

$$\sum_{\forall i \in V} x_{iut} = 1 \qquad \forall t \in [T], \forall u \in V \tag{3.17}$$

$$x_{iut} + x_{ivt+1} \leq y_{uvt} + 1 \qquad \forall i \in V, \forall t \in [T-1], \forall uv \in E, u <_M v \tag{3.18}$$

$$x_{ivt} + x_{iut+1} \leq y_{uvt} + 1 \qquad \forall i \in V, \forall t \in [T-1], \forall uv \in E, u <_M v \tag{3.19}$$

$$x_{iut} + x_{ivt+1} \leq 1 \qquad \forall i \in V, \forall t \in [T-1], \forall uv \notin E \tag{3.20}$$

$$\sum_{\forall u,v,w \in V, u <_M v} y_{uwt} + y_{wvt} \leq 1 \qquad \forall t \in [T] \tag{3.21}$$

$$y_{uvt} \leq s_t \qquad \forall t \in [T], \forall uv \in E, u <_M v \tag{3.22}$$

$$x_{i,f_0(i),0} = 1 \qquad \forall i \in V \tag{3.23}$$

$$x_{iiT} = 1 \qquad \forall i \in V \tag{3.24}$$

$$s_t \leq s_t \qquad \forall t \in [T-1] \tag{3.25}$$

$$y_{uvt} \in \{0,1\} \qquad \forall t \in [T], \forall uv \in E \tag{3.26}$$

$$x_{iut} \in \{0,1\} \qquad \forall i \in V, \forall u \in V, \forall t \in [T] \tag{3.27}$$

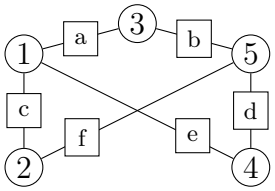
$$s_t \in \{0,1\} \qquad \forall t \in [T] \tag{3.28}$$

The model presented by Formulation (3.15)-(3.28) is an adaptation of the TS model for the PTS problem. The variables y_{uvt} and x_{iut} are used in exactly the same way as the TS model with constraints (3.16)-(3.20) being exactly the same as constraints (3.3)-(3.7). Inequalities 3.25 and 3.23 are respectively equivalent to the Inequalities 3.9 and 3.10. Each new constraint will be explained in detail in Section 3.2.1.

3.2.1 Modelling Parallel Swaps

To allow parallel swaps in the model, an efficient way to check if two swaps are disjoint must be used. If two swaps are not disjoint, then they have exactly one vertex in common, as it is impossible to have repeated swaps in the same step due to how the variables y_{uvt} are modeled. Thus, for any two conflicting swaps, a combination of three

nodes u, v, w could be taken and assumed, without loss of generality, that w is part of these two swaps (u, w), (w, v). Consequently, the variables y_{uwt} and y_{wvt} can not be 1 at the same time in the model, as given by Inequality 3.21. In the case that both edges exist in the graph of an instance, these swaps would share a vertex and could not be used in the same swap. On the cases that these vertices have only one edge or no edge, there is no way two swaps can be used between these three vertices, as Inequalities (3.18)-(3.20) prohibit them.



(a)

Edge Pairings				
a, b	b, c	c, d	d, e	e, f
a, c	b, d	c, e	d, f	
a, d	b, e	c, f		
a, e	b, f			
a, f				

(b)

Figure 3.2: Let Figure 3.2a be a graph with vertice set $V = \{1, 2, 3, 4, 5\}$ and edge set $E = \{a, b, c, d, e, f\}$. Table 3.2b lists all unordered *pairs* of edges that can be swapped in a token placement at the same step with green and red otherwise. The reader can check each pairing by taking the vertice triple of any red pairing and checking in the model. All green edge pairings are vertice disjoint and there is no three edge parallel partition in this graph.

The concept was improved by the use of the inherent vertex ordering M of the graph to remove repeated vertex checks from the model. For any vertex triple u, v, w with a "center" vertex fixed as w, checking variables y_{uwt} and y_{wvt} are equivalent to checking variables y_{vwt} and y_{wvt}. Therefore, checking variables for triples u, w, v such that $u <_M v$ is enough and remove useless checkings. Figure 3.2 give us an example to compare swap sequence representations between TS and PTS models.

Chapter 4

Conclusion

This dissertation presented the Token Swap problem and one parallel variant, the Parallel Token Swap problem. Initially, the work is focused on introducing the necessary mathematical tools for constructing the subsequent proofs, with special emphasis on the crucial notion of *merge* and *split* swaps. Then, these tools are used to build a method for finding an optimal swap sequence for the class of threshold graphs. From this method, a new method is derived to also find an optimal swap sequence for the class of cographs.

Then, to go along with these results, an initial integer linear model for each of the problems is presented together with a detailed discussion about the constraints. The next step would be the coding and solving each of these models to check their viability and compare with other approaches. Moreover, these models could be adapted to other TS variations, such as the Colored Token Swap problem and the Parallel Colored Token Swap Generalizations of the TS problem and PTS problem, respectively, where tokens can have more than one target vertex.

On future works the mathematical notions presented in this dissertation could be used to understand and build different methods for other graph classes. Moreover, the presented integer linear models could still be improved with implementation and testing, as they are an initial model. More constraints would focus on the development of more valid inequalities to optimize the model. Another route would be the creation of specialty models focused in some useful and hard classes of graphs like trees, as more specific constraints could be helpful in getting faster answers.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. On finding lowest common ancestors in trees. In *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing*, STOC '73, page 253–265, New York, NY, USA, 1973. Association for Computing Machinery.

[2] Oswin Aichholzer, Erik D. Demaine, Matias Korman, Jayson Lynch, Anna Lubiw, Zuzana Masárová, Mikhail Rudoy, Virginia Vassilevska Williams, and Nicole Wein. Hardness of token swapping on trees. *CoRR*, abs/2103.06707, 2021.

[3] S. B. Akers and B. Krishnamurthy. A group-theoretic model for symmetric interconnection networks. *IEEE Transactions on Computers*, 38:555–566, 1989.

[4] Stephen Alstrup, Cyril Gavoille, Haim Kaplan, and Theis Rauhe. Nearest common ancestors: A survey and a new algorithm for a distributed environment. *Theory of Computing Systems*, 37:441–456, 2004.

[5] Fred Annexstein, Marc Baumslag, and Arnold L. Rosenberg. Group action graphs and parallel architectures. *SIAM Journal on Computing*, 19:544–569, 1990.

[6] Vineet Bafna and Pavel A. Pevzner. Sorting by transpositions. *SIAM J. Discret. Math.*, 11:224–240, 1998.

[7] Ahmad Biniaz, Kshitij Jain, Anna Lubiw, Zuzana Masárová, Tillmann Miltzow, Debajyoti Mondal, Anurag Murty Naredla, Josef Tkadlec, and Alexi Turcotte. Token swapping on trees. *CoRR*, abs/1903.06981:41, 2019.

[8] Édouard Bonnet, Tillmann Miltzow, and Paweł Rzążewski. Complexity of token swapping and its variants. *Algorithmica*, 80:2656–2682, 2018.

[9] Laurent Bulteau, Guillaume Fertin, and Irena Rusu. Pancake flipping is hard. *Journal of Computer and System Sciences*, 81:1556 – 1574, 2015.

[10] B. Chitturi and T. Indulekha. Sorting permutations with a transposition tree. In *2019 8th International Conference on Modeling Simulation and Applied Optimization (ICMSAO)*, pages 1–4, Bahrain, 2019. IEEE.

[11] Bhadrachalam Chitturi. Upper bounds for sorting permutations with a transposition tree. *Discrete Mathematics Algorithms and Applications*, 5:24, 2013.

[12] Gene Cooperman and Larry Finkelstein. New methods for using cayley graphs in interconnection networks. *Discrete Applied Mathematics*, 37-38:95 – 118, 1992.

[13] Jack Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17:449–467, 1965.

[14] P. Erdös and R. Rado. *A Partition Calculus in Set Theory*, pages 179–241. Birkhäuser Boston, Boston, MA, 1987.

[15] Ashwin Ganesan. Diameter of cayley graphs generated by transposition trees. *CoRR*, abs/1202.5888:11, 2012.

[16] Ashwin Ganesan. On the strictness of a bound for the diameter of cayley graphs generated y transpositions trees. In *Proceedings of the International Conference on Mathematical Modelling & Scientific Computation*, pages 54–61, Cham, 2012. Springer.

[17] Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13:338–355, 1984.

[18] Lenwood Heath and John Vergara. Sorting by short swaps. *Journal of computational biology : a journal of computational molecular cell biology*, 10:775–89, 2003.

[19] Takehiro Ito, Erik D. Demaine, Nicholas J.A. Harvey, Christos H. Papadimitriou, Martha Sideri, Ryuhei Uehara, and Yushi Uno. On the complexity of reconfiguration problems. *Theoretical Computer Science*, 412:1054 – 1065, 2011.

[20] Mark R. Jerrum. The complexity of finding minimum-length generator sequences. *Theoretical Computer Science*, 36:265 – 289, 1985.

[21] Wm. Woolsey Johnson and William E. Story. Notes on the "15" puzzle. *American Journal of Mathematics*, 2:397–404, 1879.

[22] Jun Kawahara, Toshiki Saitoh, and Ryo Yoshinaka. The time complexity of the token swapping problem and its parallel variants. In *WALCOM: Algorithms and Computation*, pages 448–459, Cham, 2017. Springer International Publishing.

[23] Dohan Kim. Sorting on graphs by adjacent swaps using permutation groups. *Computer Science Review*, 22:89–105, 2016.

[24] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., USA, 1998.

[25] Benjamin Kraft. Diameters of cayley graphs generated by transposition trees. *Discrete Applied Mathematics*, 184:178 – 188, 2015.

[26] Tillmann Miltzow, Lothar Narins, Yoshio Okamoto, Günter Rote, Antonis Thomas, and Takeaki Uno. Tight exact and approximate algorithmic results on token swapping. *CoRR*, abs/1602.05150:19, 2016.

[27] Amer E Mouawad. *On Reconfiguration Problems: Structure and Tractability*. PhD thesis, University of Waterloo, 2015.

[28] Naomi Nishimura. Introduction to reconfiguration. *Algorithms*, 11:52, 2018.

[29] Kung-Jui Pai, Ruay-Shiung Chang, and Jou-Ming Chang. Constructing dual-cists of pancake graphs and performance assessment of protection routings on some cayley networks. *The Journal of Supercomputing*, 76:124546, 2020.

[30] AA Razborov. Proof complexity of pigeonhole principles. In *Developments in Language Theory*, pages 100–116, Berlin, Heidelberg, 01 2002. Springer Berlin Heidelberg.

[31] Marcos Yukio Siraichi, Vinícius Fernandes dos Santos, Caroline Collange, and Fernando Magno Quintão Pereira. Qubit allocation as a combination of subgraph isomorphism and token swapping. *Proc. ACM Program. Lang.*, 3:1–29, 2019.

[32] Marcos Yukio Siraichi, Vinícius Fernandes dos Santos, Sylvain Collange, and Fernando Magno Quintao Pereira. Qubit allocation. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pages 113–125, New York, NY, USA, 2018. ACM.

[33] John H Smith. Factoring, into edge transpositions of a tree, permutations fixing a terminal vertex. *Journal of Combinatorial Theory, Series A*, 85:92–95, 1999.

[34] John H Smith. Corrigendum to" factoring, into edge transpositions of a tree, permutations fixing a terminal vertex". *J. Comb. Theory, Ser. A*, 118:726–727, 2011.

[35] Jan van den Heuvel. The complexity of change. In *Surveys in Combinatorics*, pages 127–160, Cambridge, 2013. Cambridge University Press.

[36] Theresa P. Vaughan. Bounds for the rank of a permutation on a tree. *Journal of Combinatorial Mathematics and Combinatorial Computing*, 30:129–148, 1991.

[37] Lei Wang and K. W. Tang. The cayley graph implementation in tinyos for dense wireless sensor networks. In *2007 Wireless Telecommunications Symposium*, pages 1–7, Italy, 2007. 2007 Thyrrenian International Workshop on Digital Communication.

[38] Katsuhisa Yamanaka, Erik D. Demaine, Takehiro Ito, Jun Kawahara, Masashi Kiyomi, Yoshio Okamoto, Toshiki Saitoh, Akira Suzuki, Kei Uchizawa, and Takeaki Uno. Swapping labeled tokens on graphs. *Theoretical Computer Science*, 586:81 – 94, 2015.

[39] Katsuhisa Yamanaka, Takashi Horiyama, J. Mark Neil, David G. Kirkpatrick, Yota Otachi, Toshiki Saitoh, Ryuhei Uehara, and Yushi Uno. Swapping colored tokens on graphs. In *Workshop on Algorithms and Data Structures*, page 16, Victoria, BC, Canada, 2015. Springer.

[40] Gaku Yasui, Kouta Abe, and Katsuhisa Yamanaka. Swapping labeled tokens on complete split graphs. *IEICE technical report. Speech*, 115:87–90, 2015.