ALGORITMOS DE TEMPO POLINOMIAL E FORMULAÇÕES DE PROGRAMAÇÃO LINEAR INTEIRA PARA O PROBLEMA DE TROCA DE FICHAS EM GRAFOS E SUBCLASSES

CAIO HENRIQUE SEGAWA TONETTI

ALGORITMOS DE TEMPO POLINOMIAL E FORMULAÇÕES DE PROGRAMAÇÃO LINEAR INTEIRA PARA O PROBLEMA DE TROCA DE FICHAS EM GRAFOS E SUBCLASSES

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Vinicius Fernandes dos Santos
Coorientador: Sebastián Urrutia

Belo Horizonte

Fevereiro de 2022

CAIO HENRIQUE SEGAWA TONETTI

POLYNOMIAL TIME ALGORITHMS AND INTEGER LINEAR PROGRAMMING FORMULATIONS FOR THE TOKEN SWAP PROBLEM FOR GRAPHS AND SUBCLASSES

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
February 2022

Segawa Tonetti, Caio Henrique

Polynomial Time Algorithms and Integer Linear Programming Formulations for the Token Swap Problem for Graphs and Subclasses / Caio Henrique Segawa Tonetti. — Belo Horizonte, 2022

xxiii, 44 f. : il. ; 29cm

Dissertação (mestrado) — Federal University of Minas Gerais

Orientador: Vinicius Fernandes dos Santos

1. Graph Theory. 2. Reconfiguration. 3. Integer Programming. I. Título.

CDU

[Folha de Aprovação]

Quando a secretaria do Curso fornecer esta folha,
ela deve ser digitalizada e armazenada no disco em formato gráfico.

Se você estiver usando o `pdflatex`,
armazene o arquivo preferencialmente em formato PNG
(o formato JPEG é pior neste caso).

Se você estiver usando o `latex` (não o `pdflatex`),
terá que converter o arquivo gráfico para o formato EPS.

Em seguida, acrescente a opção `approval={`*nome do arquivo*`}`
ao comando `\ppgccufmg`.

Se a imagem da folha de aprovação precisar ser ajustada, use:
`approval=[`*ajuste*`][`*escala*`]{`*nome do arquivo*`}`
onde *ajuste* é uma distância para deslocar a imagem para baixo
e *escala* é um fator de escala para a imagem. Por exemplo:
`approval=[-2cm][0.9]{`*nome do arquivo*`}`
desloca a imagem 2cm para cima e a escala em 90%.

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

Acknowledgments

I wish to first thank my mother, Liria Yuriko, that always supported my decisions. She was always my teacher, protector and showed much of 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 Lorena Casarotto, for helping and supporting me during this last two years of my masters degree and showing me her 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. Moreover, Eduardo Costa also deserves his mention for being a good company and friend for almost all of my adult life. I owe a lot of my experiences to all of them.

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. And for all those I did wrong, I am sorry.

"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.

Quando se trata de desafios de reconfiguração, os três parâmetros de importância são conexão, diâmetro, e distância. Esta dissertação se concentra no Token Swap, um problema de reconfiguração onde o objetivo é converter uma configuração inicial de fichas de um grafo em uma configuração identidade de fichas que mapeia cada ficha para seu vértice com a menor distância possível.

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, Problemas de Reconfiguração, Pesquisa Operacional.

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.

When it comes to reconfiguration challenges, the three parameters of importance are connection, diameter, and distance. This dissertation focuses on the Token Swap problem, a reconfiguration problem where the goal is to convert an initial token placement on a graph into an identity token placement that maps every node to itself with the shortest distance possible.

The main result of this dissertation is the construction of the necessary mathematical tools and the proof of existence of a 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.

Palavras-chave: Graph Theory, Reconfiguration Problems, Operational Research.

List of Figures

Contents

Chapter 1

Introduction

The reconfiguration framework [Ito et al., 2011] brings the notion of *transformation* in computational problems and new questions arises from the necessity of understanding these changes under various operation and constraints. A simple version of this problem is sorting a list of numbers by adjacent swaps between two elements, in which the swap number is exactly the number of pairs that are out of order, being recognized as the number of *inversions* in a list and two elements are adjacent if they are next to each other in the list. This swap sequence can be calculated in polynomial time by a modified bubble sort algorithm [Knuth, 1998]. Under other conditions, when sorting the lists by *prefix-reversals*, an elementary operation that flips a prefix of the list, finding the minimum number of flips is **NP-complete** [Bulteau et al., 2015]. This problem is called the pancake sorting problem. Another 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 can *slide* a tile to any adjacent empty tile space to achieve a final sorted configuration state. Research of the 15-puzzle version of the n-puzzle dates back to the late 19^{th} century [Johnson and Story, 1879] 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 problems, the interest lies in one of the three following parameters: connectivity, diameter or distance. This parameters translate, respectively, to the following questions: (a) can any configuration be reconfigured into another?; (b) what is the maximum number of required operation steps for any reconfiguration?; and (c) what is the minimum number of operations for any particular reconfiguration? In the examples aforementioned, the distance between two configurations (the initial and the sorted) was being used. For the interested, more information about most of the known reconfiguration problems can be found in the surveys by

[van den Heuvel, 2013; Mouawad, 2015; Nishimura, 2018].

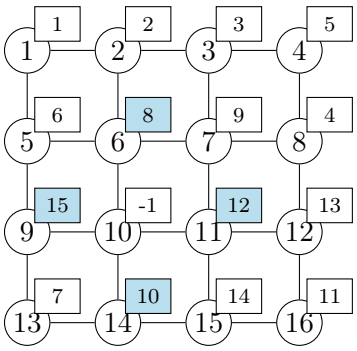


Figure 1.1: Instance of the 15-puzzle problem in a grid graph. Each vertex is represented by a circle and the rectangle is the tile currently positioned in the vertex. The -1 rectangle represents the empty tile. Each light blue colored rectangle is a possible swap operation with the empty tile in the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

This dissertation focus on a reconfiguration problem called the Token Swap problem (TS). Let $G = (V, E)$ be a graph with $n = |V|$ vertices and $|E|$ edges, with distinct tokens placed on it's vertices. The objective is to reconfigure this initial token placement called $f_0 : V \mapsto V$ into the identity token placement f_i that maps every node to itself with minimum distance. The reconfiguration must consist of a token swap sequence S, being each identified by a pair of adjacent graph vertices, meaning that the elementary operation will be restricted to a swap between two tokens placed on vertices that share an edge in the graph. For the decision version of this problem, the aim is to know if it is possible to have a swap sequence S that transforms f_0 to f_i in k or less swaps, such that $k \in \mathbb{N}$. In Figure 1.2, a complete example of a TS instance is shown. The formal mathematical definitions necessary to fully understand this dissertation will be given in Section 1.2.

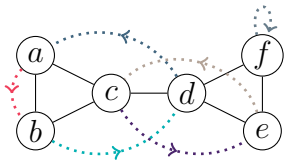


Figure 1.2: Example of a instance of the Token Swap problem with $V = \{a, b, c, d, e, f\}$ and tokens represented by colored dotted arrows to differentiate. The arrows point from the vertice that is originally positioned in mapping f_0 to the target vertice in the identity map f_i.

Every reconfiguration problem can also be formulated as a reconfiguration graph where each node is a possible configuration 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 another [Nishimura, 2018]. In the case of the TS, the reconfiguration graph will be connected as long as the original graph is also connected. This connection guarantees that any configuration can be reconfigured to any other, as it is possible to take any spanning tree of the original graph and move the desired tokens to each corresponding leaf in a way that any chosen final configuration is achieved. This process give us an upper-bound of $O(n^2)$ swaps for any instance of the TS problem [Yamanaka et al., 2015a]. The diameter and distance of this reconfiguration graph is exactly the upper-bound of the reconfiguration problem, and the distance between two configurations —i.e., two vertices in the reconfiguration graph —, respectively. 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 exponential 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 [Jerrum, 1985]. The diameter of the Cayley Graphs has been researched in the context of transposition trees [Akers and Krishnamurthy, 1989; Cooperman and Finkelstein, 1992; Bafna and Pevzner, 1998; Ganesan, 2012a,b; Chitturi, 2013; Kraft, 2015; Chitturi and Indulekha, 2019]—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].

There is discussion about the relation of the TS problem and variants to parallel sorting on a SIMD machine consisting of several processors with local memory connected by a network [Yamanaka et al., 2015a; Kawahara et al., 2017]. Applications

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

of the TS problem encompass a wide range of fields and some examples are: computing efficient interconnection network structures where the maximum delay could be measured by calculating the diameter of the network [Annexstein et al., 1990], computational biology [Bafna and Pevzner, 1998; Heath and Vergara, 2003], model Wireless Sensor Networks (WSS) [Wang and Tang, 2007], protection routing [Pai et al., 2020] and qubit allocation for quantum computers [Siraichi et al., 2018, 2019]. More examples can be found in Aichholzer et al. [2021].

The reconfiguration version of TS was first introduced in 2015 [Yamanaka et al., 2015a], and further generalizations and variations were studied in the same year [Yamanaka et al., 2015b]. The TS problem was first proved **NP-Complete**, and it remains even when restricted to bipartite graphs with degree bounded 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 [Kawahara et al., 2017; Miltzow et al., 2016]. Subsequently, it was proved that the problem remains hard even when both the treewidth and the diameter of the input graph are constant [Bonnet et al., 2018]. There are some special classes of graphs that can be solved through a exact polynomial time algorithm, almost all of them with good references listed in historical order by a study of TS in trees performed by [Biniaz et al., 2019]: cliques, paths, cycles, stars, brooms, lollipop [Kawahara et al., 2017], complete bipartite graphs and complete split graphs. Concerning square of paths, there is a 2-approximation algorithm [Heath and Vergara, 2003]. For trees, two 2-approximation algorithms [Yamanaka et al., 2015a; Miltzow et al., 2016] are known and proven to be tight [Biniaz et al., 2019]. It was showed that these known techniques for approximating TS on trees prevent approximation factors less than 2 [Aichholzer et al., 2021]. Both approximation algorithms can be adapted to general graphs, providing α- and 4-approximation algorithms, respectively, where α is the value of an α-spanner tree of the input graph. Bonnet et al. [2018] conjectured that TS remains **NP-Complete** even in trees and Biniaz et al. [2019] reinforced this conjecture by showing a counterexample of the *Happy Leaf Conjecture* [Vaughan, 1991]. And finally, Aichholzer et al. [2021] showed that TS and other two variations (Weighted Token Swap and Parallel Token Swap) are **NP-Complete** on trees.

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. Chap-

ter 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; and 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_i that maps every element to itself. For a set V, an ordering of the elements of the set is a bijective function $M : V \mapsto \mathbb{N}$ 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 $E \subseteq V^2$. The graph is called *undirected* if and only if the set of edges E have unordered tuples, while for *directed* graphs the set of edges have ordered tuples. The shorthand 'uv' is used to describe a pair (u, v) 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 a *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 $\delta_{out}(v)$. 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 vertice set $V(G)$ such that an edge exists between two distinct vertices if and only if they were not adjacent on G.

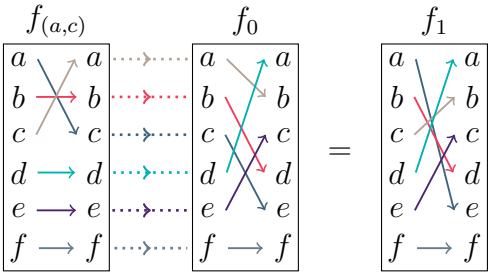


Figure 1.3: Let f_0 be the TS instance represented in Figure 1.2, $f_{(a,c)}$ be the mapping function representation of swap (a, c) and f_1 be the resulting mapping of the operation $f_0 \circ f_{(a,c)}$. Each rectangle represents a function with domain on the left, codomain on the right and color coded 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 a 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 $(u, v) \in E$. 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 $f_s(w) = w$, $\forall w \in V \setminus \{u, v\}$, and 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. It can be said that 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 graph 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.

$$\underbrace{(\underbrace{(\underbrace{(f_0 \circ \underbrace{f_{s_1}}_{f_1}) \circ f_{s_2}}_{f_2}) \circ \ldots) \circ f_{s_k}}_{f_k} = f_i \tag{1.1}$$

It is important to note that 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 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 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. In another 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 graph. 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 Theorem 1.2.1 helps in understanding that the process indeed generates the lowest common ancestor of the subset.

Theorem 1.2.1. *Let G^r be a rooted tree and a subset $\dot{V} \subseteq V(G^r)$. The $LCA_{G^r}(\dot{V})$ gives us the lowest common ancestor of the vertex subset.*

Proof. Let J be the set of all lowest common ancestors resulting from the pairwise node calculation, w be the node of J located the nearest from the root and u,v the nodes of $V(G^r)$ such that $LCA_{G^r}(u, v) = w$. To show that this node is indeed a common ancestor of all nodes in $\dot{V}$, suppose, by contradiction, that there is a node $p \in V(G^r)$ that w is not a ancestor of. Then, if we form a pair from u or v and p, the distance of the lowest common ancestor $LCA_{G^r}(u, p)$ or $LCA_{G^r}(v, p)$ *must* be lower than the distance of w, otherwise w would be a common ancestor, as this node will have to exist in the subtree where w is the root. But this create a contradiction, as w were the nearer node from the root from the set of pairwise lowest common ancestors.

To prove that this node is the *lowest* common ancestor, we assume, by contradiction, that there is another node from J that is lower and is a common ancestor to all nodes in V'. Then, w is not the lowest common ancestor of nodes u and v, creating a contradiction. □

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 [Aho et al., 1973] with an optimally efficient algorithm. Other versions were studied later [Harel and Tarjan, 1984], with many different algorithms and general improvements over the years [Alstrup et al., 2004]. 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 dissertation. 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 the directed graph may contain self-loops when a token is already in the correct vertex. In this graph, configurations can be characterized by a set of directed cycles $CS(CG) = \{C_0, C_1, \ldots, C_p\}$, for $p \in \mathbb{N}$, as seen in Lemma 1.2.2.

Lemma 1.2.2. *[Yamanaka et al., 2015a] Let CG_f be a conflict graph of a graph G. Then, every component in CG is a directed cycle.*

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. Lemma 1.2.4 and Lemma 1.2.3 underline two basic properties of this set of cycles. The cyclic representation $C = (u_1, \ldots, u_k)$ denotes a cycle such that $u_i u_{i+1} \in E(C)$, $\forall i \in [k-1]$ and $u_k u_1 \in E(C)$.

Lemma 1.2.3. *Let CG_f be a conflict graph of a graph G. For each pair of cycles $C_i, C_j \in CS(CG)$, $V(C_i) \cap V(C_j) = \{\emptyset\}$.*

Proof. By definition, as each vertex can have only one token positioned in it and each token can have only one target vertex. □

Lemma 1.2.4. *Let CG_f be a conflict graph of a graph G. The number of cycles in CG_f is bounded by $O(|V(G)|)$ and is exactly $|V(G)|$ only when the current configuration is the identity map f_i.*

Proof. By Lemma 1.2.3, the maximum number of cycles happens when every vertex is part of a distinct disjoint cycle, resulting in one self-loop for each. □

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 in two types: *Merge* and *Split*. 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 $C_i = (u, u_1, \ldots, u_k)$ and $C_j = (v, v_1, \ldots, v_p)$, 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 $C_{ij} = (v, u_1, \ldots, u_k, u, v_1, \ldots, v_p)$. Now, take a permutation cycle $C \in CS(CG_{f_i})$ and assume, without loss of generality, that $C = (u, u_1, \ldots, u_k, v, v_{k+1}, \ldots, v_{k+p})$, for $k, p \in \mathbb{N}_0$. A Split is a swap (u, v) that is applied in one 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 $C_i = (v, u_1, \ldots, u_k)$ and $C_j = (u, v_{k+1}, \ldots, v_{k+p})$.

Lemma 1.2.5. *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. By the Lemma 1.2.2, these vertices can only be arranged in loops or self-loops. As every vertex is on a loop and vertices between cycles are disjoint, as shown in

Lemma 1.2.3, 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 dissertation. 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, $V(C) \subseteq N_G(v)$.

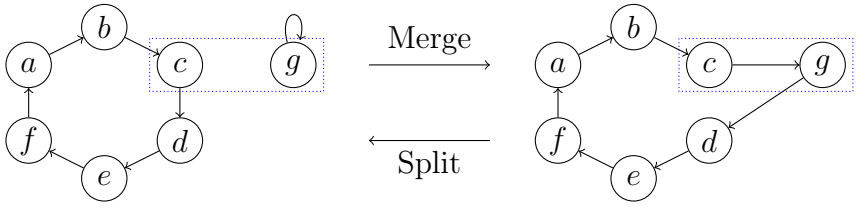


Figure 1.4: Let G be a connected graph with seven vertices where edge (c, g) must exist in $E(G)$. The left image denotes the current configuration as directed cycles and the blue rectangle shows which swap will be applied to achieve the configuration on the right image. This operation *merges* the two 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 its 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 [Smith, 1999] tried to prove, but subsequently found an error [Smith, 2011]. Then, [Vaughan, 1991] 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*[2]. This conjecture was disproved by [Biniaz et al., 2019], as they shown that there is a class of infinite trees that necessarily need (c) swaps to achieve the optimal number of swaps 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. The two approximative algorithms cited in this chapter can only generate swap sequences of category (a) and (b).

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.

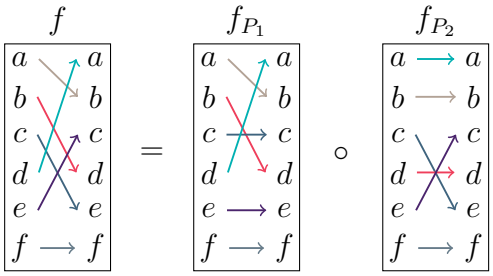


Figure 1.5: Let f be a valid token configuration and $\mathcal{P}(\{a, b, c, d, e, f\}) = \{\{a, b, d\}, \{c, e\}, \{f\}\}$ be a partition of f. This partition is valid, as elements f_{P_1}, f_{P_2} and f_{P_3} are a valid token configurations. Moreover, this partition is also the coarsest partition, 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

[2]The original paper actually claimed a stronger conjecture with the Happy Leaf Conjecture as a special case.

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 1.2.6. *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 1.2.6 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 1.2.6.1 shows an interesting interaction between tokens and swaps that can be used to optimize swap sequences.

Corollary 1.2.6.1. *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.

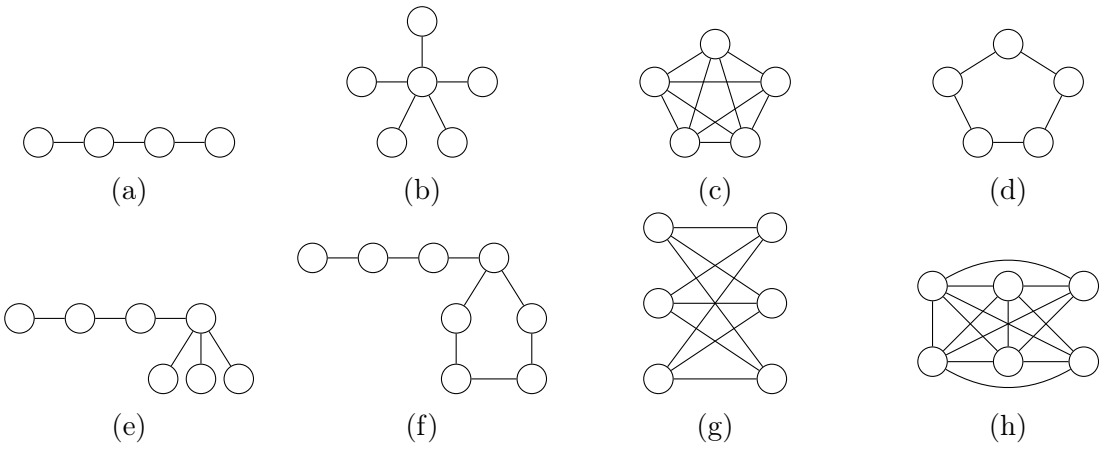


Figure 1.6: Graphs representing graph classes related to efficient algorithms for token swap problems. Figure 1.6a is a path of size 4; Figure 1.6b is a star; Figure 1.6c is a complete graph; Figure 1.6d is a cycle; Figure 1.6e is a broom graph; Figure 1.6f is a lollipop graph; Figure 1.6g is a complete bipartite graph; and Figure 1.6h is a complete split graph.

The problem of swapping tokens on paths, with an example in Figure 1.6a, 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 [Jerrum, 1985; Knuth, 1998]. 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 $O(n^2)$.

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 ≠ 0;
return S;
```

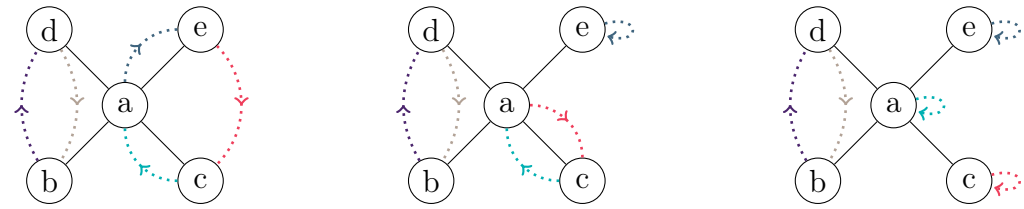
For cycle graphs, shown in Figure 1.6d, the method is an extension of the inversion number method of a path because the inversion number cannot be applied directly, as token can be moved either in a clockwise or anticlockwise manner to its target vertice. The optimal algorithm, presented by Jerrum [1985], 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 1.6e and Figure 1.6f. The proofs of correctness of the polynomial algorithms for brooms and lollipop graphs are similar and based on a eval-

uation 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 Kawahara et al. [2017].

1.3.1 Token Swapping on Stars

A star, shown in Figure 1.6b, 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 1.7 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 1.7: 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 [Akers and Krishnamurthy, 1989; Biniaz et al., 2019]. 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 1.6c 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 [Kim, 2016].

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 1.6h 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.

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 Yasui et al. [2015].

1.3.4 Token Swapping on Complete Bipartite

A complete bipartite graph, shown in Figure 1.6g, 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) [Yamanaka et al., 2015a]. 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:

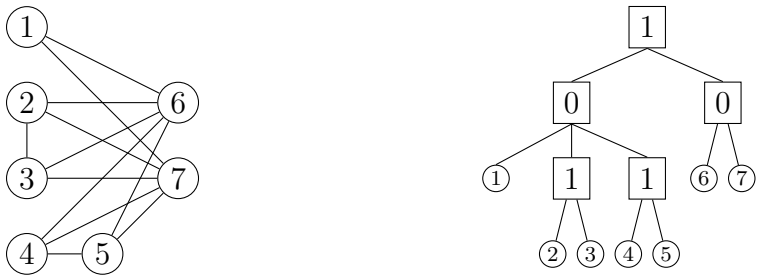
- A graph with a single vertex is a cograph;
- If $G_1, G_2, ..., G_k$ are cographs, then so is their disjoint union;
- if G is a cograph, then so is its complement $\overline{G}$.

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. 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 an 1-node. The cotree of any particular cograph is unique.

Note that 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 $E(\dot{G}) = E(G_i) \cup E(G_j) \cup (V(G_i) \times V(G_j))$. 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. Figure 2.1 introduces a simple example of a cograph and its cotree representation. Note that in a cotree, 1-nodes represent joins and 0-nodes represent disjoint unions between the leaves of their subtrees.

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.



(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 $CS(CG)$. A CS^0 set is 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 u, $LCA_{CT(G)}(V(C)) = u$, with u being a 0-node of the cotree. A CS^1 set is the set of all cycles $C \in CS$ such that $LCA_{CT(G)}(V(C)) = w$, with w being any 1-node of the cotree, or $LCA_{CT(G)}(V(C)) = v$ in the case that $C = (v)$. These two sets are disjoint, their union is exactly the cycle set CS and will be called the zero cyles set and the one cycles set respectively.

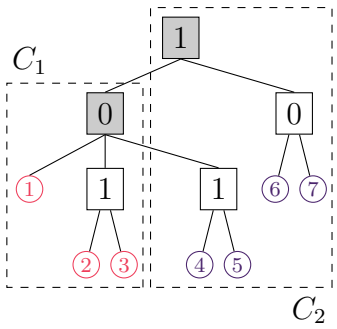


Figure 2.2: The cotree of the graph showed in Figure 2.1 is being used. Let f be a configuration where nodes 1,2,3 are part of a permutation cycle C_1 and nodes 4,5,6,7 are part of another permutation cycle C_2, without paying attention to the exact cycle configuration. The lowest common ancestor of 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

This division of the cycle set CS is important to correctly identify which of the methods to be presented need to be used. Any cycle C of CS have a partition of the vertices $\mathcal{P}(C)$ created by the subtrees of the lowest common ancestor node of the cycle, as seen in Figure 2.2. Per the definition of the cotree, a partition of a cycle of CS^0 have no edges between any of these partitions in G. Similarly, any cycle of the one cycles set CS^1 have a partition of the vertices such that all edges exists between each element of the partition in G. Each partition element represents the leafs in each of the subtrees of the lowest common ancestor node and the union of all partitions of $\mathcal{P}(C)$ is exactly $V(C)$. These are the fundamental properties used in Lemma 2.1.1 and Lemma 2.1.2, in which the proofs presents a method for solving each cycle type with a determined number of swaps. An individual permutation cycle C is a instance of the TS where every other permutation cycle is a cycle of size one in the same graph as the original instance.

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

Proof. Let $\mathcal{P}(C) = \{P_1, P_2, \ldots, P_k\}$ be the partition created by the lowest common ancestor node. Induction on the size of the cycle will be used to prove this lemma. Assume that all other cycles of size m, with $m < |V(C)|$ can be solved with $m - 1$ swaps. The base is a cycle of size one that is already solved, as $1 - 1 = 0$. For the induction step, a argument on the size of the partitions will be used. There are two cases:

Case 1: Let $P = \{v\}$ be a element of size one of $\mathcal{P}(C)$. Then, v dominates the vertices of the $V(C)$ that he is part of, as all edges between partitions *must* exist, by the cograph definition. Swap the token on v to his target vertice, splitting this cycle in one of size one and other of size $m - 1$. The cycle of size one is already solved, while the cycle of size $m - 1$ can be solved in $m - 2$ by the induction hypothesis, resulting in $m - 1$ swaps to solve the original cycle;

Case 2: If there is no element of size one, it is possible to create an element of size one while maintaining the desired number of swaps by iteratively solving separated nodes and removing them from a partition to achieve a partition of size one and go to case number 1. Choose any element of the partition $P \in \mathcal{P}(C)$. By the definition of the cycle, there must exist at least one edge of the cycle that goes from P to some other element of $\mathcal{P}(C)$ and at least one edge that goes to P from some other element of $\mathcal{P}(C)$. Let (u, w) be any of these directed edges of the permutation cycle that goes **to** P. A swap applied along this edge will create two resulting permutation cycles: One

of size one and one of size $|C| - 1$, effectively moving one token to the correct vertex and removing this node from the original cycle C. Let this new cycle be called $\dot{C}$. The partition $\mathcal{P}(\dot{C})$ is exactly the partition of the original C, with the element P with one vertice less, the vertice w. By the induction hypothesis, these two resulting cycles can be solved with the correct number of swaps, resulting in $|C| - 1$ swaps. Note that the resulting cycle of size $|C| - 1$ will fall into either Case 2 or Case 1 eventually. □

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

Proof. Let $\mathcal{P}(C) = \{P_1, P_2, \ldots, P_k\}$ be the partition created by the lowest common ancestor node. By the cograph definition, each element of $\mathcal{P}(C)$ have no edges between them, resulting in no possible path to move tokens from one element to another by only using the nodes $V(C)$. In order to solve C, at least one node must be *merged* to cycle C so it can be solved as a cycle of CS^1.

As G is connected, there must exist at least one ancestor 1-node that connects the graph (or more locally, the nodes of $V(G)$). Let the ancestor node called w and let $NodePartition(w)$ be the element of the leafs of the children subtrees of w in the cotree $CT(G)$. There must exist a $P \in NodePartition(w)$ such that $V(C) \subseteq P$, as w must be a direct ancestor of $LCA_{CT(G)}(C)$. Let u be any node that is an element of any set in $\mathcal{P}(C) \setminus P$. Then, u must dominate $V(C)$.

Now, *assume* that u already has a correct token in the current mapping and merge him to the cycle 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 in $|\dot{C}| - 1$ swaps, which is equivalent to $|C|$ swaps, as there is one vertice more. In total, the number of swaps is $|C| + 1$ because of the first swap needed to add u to the cycle. Note that as the token of u is assumed to be correct, his token went back to the same place as before, not changing any other cycle of the configuration. This means that the token currently in u does not need to be the correct token. □

With both of the methods presented, all permutation cycles of any TS instance on a cograph G can be solved with a value that can be easily calculated and is denoted as $|V(G)| - |CS^1| + |CS^0|$. Now, some exploration can be done about the strength of this characterization and how small are these values of swaps for these cycles. First, it is possible to show that the methods can achieve the minimum number of swaps for individual cycles by Lemma 2.1.3 and Lemma 2.1.5.

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

Proof. Suppose that the cycle C can be solved in less than $|C| - 1$ swaps. Then, suppose an instance of TS where the graph is a complete graph with the same number of vertices and labels of the original cograph. Let the permutation cycle C be the only cycle of size greater than 1 in the initial configuration of this instance. The optimal number of swaps in this instance is described in Section 1.3.2 and is given by $n - r$, where n is the number of vertices and r is the total number of permutation cycles. In our instance, it is possible to see that the minimum number of swaps is $n - 1$ even with a complete topology, showing that the same cycle on a more restricted topology cannot be possibly less than $n - 1$, as the original graph is a subgraph of the complete topology. □

Lemma 2.1.4. *Let G be a graph, f_0 an initial configuration and C any permutation cycle on this instance. Let k be the minimum number of merge swaps needed to solve the individual instance of C in any swap sequence S. Then, $|S| \geq |C| - 1 + 2k$.*

Proof. The reasoning is based on the number of permutation cycles on the instance. Let s be the minimum number of split swaps necessary to solve the instance and $|V(G)|-|C|+1$ the total number of cycles in this instance. The target identity mapping must have $|V(G)|$ cycles to be correct, so, it needes at least $|C|-1$ split swaps, as every split swap must increase the number of cycles by one. However, it is known that at least k merge swaps are needed to solve this instance, which increases the total number of cycles by k at total. Therefore, the number of split swaps must be $s \geq |C| - 1 + k$. Thus, the total number of swaps is bounded by $|S| = s + k \geq |C| - 1 + 2k$. □

Lemma 2.1.5. *Individually, any cycle $C \in CS^0$ cannot be solved in less than $|C| + 1$ swaps.*

Proof. By Lemma 2.1.2, every $C \in CS^0$ can be solved by using exactly one merge swap, as every other swap given by the method on Lemma 2.1.1 are split swaps (swaps along an edge of the permutation cycle). Thus, by Lemma 2.1.4, as the minimum number of merge swaps k needed for cycles of CS^0 is one, each cycle is lower bounded by $|C|-1+2\times 1 = |C|+1$, which is exactly the amount of swaps given by the method shown. □

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 vertice is connected to every other vertice, 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, as seen in Lemma 2.1.8. There are many equivalent definitions for this class, but, for the sake of conciseness, a threshold graph is a graph that can be constructed from the empty graph by repeatedly adding either an isolated vertice or a dominating vertice. 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 2.1.6. *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 2.1.5. □

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

Proof. Let $p(G, f) = |V(G)| - |CS^1(CG_f)| + |CS^0(CG_f)|$. Note that $p(G, f_i) = 0$, the number of swaps needed to solve the identity configuration, holds. First, it is going to be necessary to prove that for any swap on f, the value of $p(G, \dot{f})$ will be decreased by at most one, such that $\dot{f}$ is a adjacent configuration of f. The following cases encompass every possible split interaction between cycles.

- **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 that we will call 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, resulting in $p(G, \dot{f}) = p(G, f) - 1$;

  Case (B): If $C_u \in CS^0$ and $C_v \in CS^1$, the value of $|CS^0|$ is increased in 1, with $|CS^1|$ remaining unchanged, resulting in $p(G, \dot{f}) = p(G, f) + 1$;

  Case (C): Assume $C_u, C_v \in CS^0$. There must exist two elements of the partition P_u, P_v of $\mathcal{P}(C)$ such that $V(C_u) \subseteq P_u$ and $V(C_v) \subseteq P_v$. Take any two disconnected vertices x, y from $V(C_u)$ and two disconnected vertices w, z from $V(C_v)$. Note that they must exist on each cycle as they are a zero cycle. Then, the induced subgraph $G[\{x, y, w, z\}]$ on the original graph is a cycle of size 4. This is a contradiction on the definition of the Threshold graph.

- **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 that we will call C_u and C_v. Assume that $u \in V(C_u)$ and $v \in V(C_v)$.

  Case (A): Let $C_u, C_v \in CS^1$. Then, the total number of swaps given by our model is $|C_u| - 1 + |C_v| - 1 + 1 = |C_u| + |C_v| - 1 = |C| - 1$. This result is a contradiction on Lemma 2.1.5, as this cycle cannot be solved individually in less than $|C| + 1$ swaps;

  Case (B): If $C_u \in CS^0$ and $C_v \in CS^1$, the value of $|CS^1|$ is increased by 1 and $|CS^0|$ continues the same, 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, resulting in $p(G, \dot{f}) = p(G, f) + 1$;

It is not needed to check the merge swaps in this case, as they are just the inverse of a split. This means that the equations have inverted plus and minus signals, and, as no equation has other value greater than one, the proof that $p(G, f)$ decreases by at maximum one for every possible swap holds. 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 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_i through adjacent configurations $f_1, f_2, \ldots, f_k = f_i$, 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.

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

By organizing Inequation 2.2, substituting $p(G, f_k)$ for a 0, as it is the last configuration and this swap sequence solve the instance, and substituting $p(G, f_0)$ by the original assumption, the Inequation 2.3 is found.

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

□

Theorem 2.1.8. *Let G be a threshold graph with an initial token placement f_0. The minimum number of required swaps is given by $|V(G)| - |CS^1| + |CS^0|$.*

Proof. Directly from Lemma 2.1.6 and Lemma 2.1.7. □

2.2 Dependencies Betweeen Permutation Cycles

Section 2.1 gave techniques for solving the two types of cycles individually without changing the configurations of other cycles and proved the optimality of this method for Threshold graphs. The proof for the Cographs is constructed from the Treshold graphs, based on the observation that Cographs allow cycles of size 4. When true, a merge of two zero-cycles to a one-cycle can save exactly two swaps in the final configuration. Let this swap be called a *cutback* swap.

Lemma 2.2.1. *Let G be a cograph with an token placement f. Let C_1 and C_2 be the two cycles resulting from any split swap on a cycle C member of CS^0. Either C_1 or C_2 must be a cycle with exactly the same lowest common ancestor as the original cycle C.*

Proof. Let $V_1, \ldots, V_k$ be elements of a partition $\mathcal{P}(V(C))$ where k is the number of children of the node $LCA_{CG(G)}(V(C))$ in the cotree of G and each element corresponds to the the set of all graph G nodes present in the children subtree. Assume that neither C_1 or C_2 have the same lowest common ancestor as C. Note that by the definition of a 0-cycle this swap must happen between two nodes that are member of the same element of the partition $\mathcal{P}(V(C))$, as there is no edges between each element of the partition, $V(C_1) \cup V(C_2) = V(C)$ and each partition element of must have *at least one* token positioned at a vertice that needs to be moved to another partition element, as these vertices forms a cycle.

If $k > 2$, then by the pigeon-hole principle [Erdös and Rado, 1987; Razborov, 2002], either C_1 or C_2 (or both) have vertices from more than one element of the partition, resulting in the same lowest common ancestor as C. If $k = 2$, then C_1, C_2 must be each an element of $\mathcal{P}(V(C))$. Therefore, there must exist at least one swap between the two elements that perfectly split the two elements to two disjoint cycles. But, as this is a 0-cycle, no such swap can exist. If $k = 1$, then there is a contradiction on the common ancestor, because there is a lower common ancestor to be found. □

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 these cases where two swaps can be saved.

Lemma 2.2.2. *Let G be a graph and $\mu(G)$ a maximum edge matching.*

1. *Adding a vertice connected by edges to the graph G can increase the size of the matching $|\mu(G)|$ in at most one;*
2. *Adding non-existing undirected edges to a vertice can increase the size of the matching $|\mu(G)|$ in at most one;*
3. *Let $\dot{G}$ and $\ddot{G}$ be graphs such that $G \subseteq \dot{G} \subseteq \ddot{G}$, $|V(\dot{G}) \setminus V(G)| = 1$ and $|V(\ddot{G}) \setminus V(\dot{G})| = 1$. Then, $|\mu(\ddot{G})| \leq |\mu(G)| + 2$.*

Proof. The proofs are respectively enumerated below.

1. Let $\dot{G}$ be the graph resulted by adding a vertice x with any adjacency and let $\mu(\dot{G})$ be its maximum edge matching. Suppose x is saturated in $\mu(\dot{G})$ and $|\mu(\dot{G})| > |\mu(G)|$. In special, $|\mu(\dot{G})|$ 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 that x participates as a new bigger matching. If x is not saturated, then $|\mu(\dot{G})| = |\mu(G)|$ by the same reason as above;
2. Let $\dot{G}$ be the graph resulted by adding any amount of non-existing edges to a vertice $v \in V(G)$ and let $\mu(\dot{G})$ be it's maximum edge matching. Let $\mu(\dot{G})$ be matching of size greater than $\mu(G) + 1$. the resulting matching created by removing the edge matching related to v in $\mu(\dot{G})$ is a valid matching on the original graph G and has size at least $\mu(G) + 1$, creating a contradiction on the size of the maximum matching of G.

3. By Lemma 2.2.2, the size of the matching $\mu(\dot{G})$ can increase in at most 1 from $\mu(G)$, as in Inequation 2.4, and the size of the matching $\mu(\ddot{G})$ can increase in at most 1 from $\mu(\dot{G})$, as in Inequation 2.5. By summing these two inequations, the Inequation 2.6 is achieved.

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

□

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, 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 cycles unsaturated by the matching $\mu(H)$ will stay.

Lemma 2.2.3. *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 2.1.5 and the previous discussion. □

Lemma 2.2.4. *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 $p(G, f_i) = 0$ holds. First, it is going to be necessary to prove that for any swap on f, the value of $p(G, \dot{f})$ will be decreased by at maximum one, such that $\dot{f}$ is a adjacent configuration of f. The following cases encompass every possible split interaction between cycles.

- **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 that we will call 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 $C_v \in CS^1$, the value of $|CS^0|$ is increased by 1, increasing the number of vertices in H by one. By Lemma 2.2.2, the matching can increase in at most one, 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. By Lemma 2.2.2, this matching can increase in at most two, 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 that we will call C_u and C_v. Assume that $u \in V(C_u)$ and $v \in V(C_v)$.

  Case (A): Let $C_u, C_v \in CS^1$. Then, the total number of swaps given by our model is $|C_u| - 1 + |C_v| - 1 + 1 = |C_u| + |C_v| - 1 = |C| - 1$. This result is a contradiction on Lemma 2.1.3, as this cycle can be solved individually in less than $|C| + 1$ swaps;

  Case (B): If $C_u \in CS^0$ and $C_v \in CS^1$, the value of $|CS^1|$ is increased by one and $|CS^0|$ does not change. By Lemma 2.2.1, $LCA_{CG(G)}(V(C_u)) = LCA_{CG(G)}(V(C))$, 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 $|CS^1|$ is increased by 1 and $|CS^0|$ does not change. By Lemma 2.2.1, either C_u or C_v have the same lowest common ancestor as the cycle C, not changing the original related vertice on H and adding exactly one additional vertice. Lemma 2.2.2 show that the 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 also not needed to check the merge swaps in this case, as they are just the inverse of a split. This means that the equations have inverted plus and minus signals, and, as no equation has other value greater than one, the proof that $p(G, f)$ decreases by at maximum one for every possible swap holds. 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 the Inequation 2.1.

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

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_i through adjacent configurations $f_1, f_2, \ldots, f_k = f_i$, each pair of configurations $p(G, f_{j+1}) \geq p(G, f_j) - 1$ holds from Inequation 2.7 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 2.8.

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

By organizing Inequation 2.8, substituting $p(G, f_k)$ for a 0, as it is the last configuration and this swap sequence solve the instance, and substituting $p(G, f_0)$ by the original assumption, the Inequation 2.9 is found.

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

□

Theorem 2.2.5. *Let G be a cograph with an initial token placement f_0. The minimum number of required swaps is given by $|V(G)| - |CS^1(CG_f)| + |CS^0(CG_f)| - 2 \times |\mu(H)|$.*

Proof. Directly from Lemma 2.2.3 and Lemma 2.2.4. □

Corollary 2.2.5.1. *Token Swap can be solved in polynomial time in cographs.*

Proof. Directly from Theorem 2.2.5. □

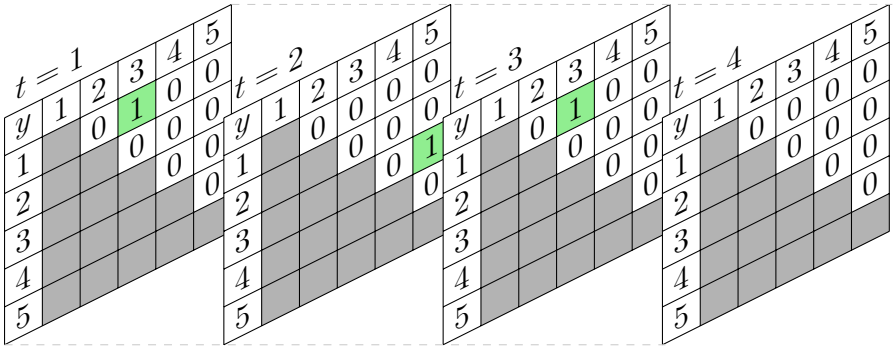
Chapter 3

Integer Linear Programming Models

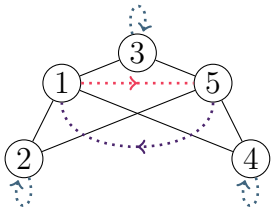
This chapter presents two integer linear programming models for the Token Swap problem and the Parallel Token Swap problem, respectively, along with an explanation of each variable and constraint for each model.

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.

$$
\begin{aligned}
\text{minimize} \quad & \sum\nolimits_{\forall uv \in E, u <_M v, \forall t \in [T]} y_{uvt} && && (3.2)\\
\text{subject to} \quad & \sum\nolimits_{\forall u \in V} x_{iut} = 1 && \forall t \in [T], \forall i \in V && (3.3)\\
& \sum\nolimits_{\forall i \in V} x_{iut} = 1 && \forall t \in [T], \forall u \in V && (3.4)\\
& x_{iut} + x_{ivt+1} \leq y_{uvt} + 1 && \forall i \in V, \forall t \in [T-1], && \\
& && \forall uv \in E, u <_M v && (3.5)\\
& x_{ivt} + x_{iut+1} \leq y_{uvt} + 1 && \forall i \in V, \forall t \in [T-1], && \\
& && \forall uv \in E, u <_M v && (3.6)\\
& x_{iut} + x_{ivt+1} \leq 1 && \forall i \in V, \forall t \in [T-1], && \\
& && \forall uv \notin E && (3.7)\\
& \sum\nolimits_{\forall u,v \in V, u <_M v} y_{uvt} \leq 1 && \forall t \in [T] && (3.8)\\
& \sum\nolimits_{\forall uv \in E, u <_M v} y_{uvt} \geq \sum\nolimits_{\forall uv \in E, u <_M v} y_{vwt+1} && \forall t \in [T-1] && (3.9)\\
& x_{i,f_0(i),0} = 1 && \forall i \in V && (3.10)\\
& x_{iiT} = 1 && \forall i \in V && (3.11)\\
& y_{uvt} \in \{0,1\} && \forall t \in [T], \forall uv \in E && (3.12)\\
& x_{iut} \in \{0,1\} && \forall i \in V, \forall u \in V, && \\
& && \forall t \in [T] && (3.13)
\end{aligned}
$$

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 [Bonnet et al., 2018]. 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 [Edmonds, 1965]. 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 [Yamanaka et al., 2015a; Kawahara et al., 2017] and is better suited for some reconfiguration problems, like Qubit Allocation [Siraichi et al., 2018, 2019].

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 [Kawahara et al., 2017].

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.

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

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

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

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

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

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

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

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

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

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

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

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

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

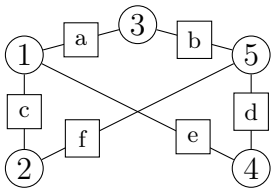
$$s_t \in \{0,1\} \qquad \forall t \in [T] \tag{3.28}$$

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.

Bibliography

Aho, A. V., Hopcroft, J. E., and Ullman, J. D. (1973). 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. Association for Computing Machinery.

Aichholzer, O., Demaine, E. D., Korman, M., Lynch, J., Lubiw, A., Masárová, Z., Rudoy, M., Williams, V. V., and Wein, N. (2021). Hardness of token swapping on trees. *CoRR*, abs/2103.06707.

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

Alstrup, S., Gavoille, C., Kaplan, H., and Rauhe, T. (2004). Nearest common ancestors: A survey and a new algorithm for a distributed environment. *Theory of Computing Systems*, 37:441–456.

Annexstein, F., Baumslag, M., and Rosenberg, A. L. (1990). Group action graphs and parallel architectures. *SIAM Journal on Computing*, 19:544–569.

Bafna, V. and Pevzner, P. A. (1998). Sorting by transpositions. *SIAM J. Discret. Math.*, 11:224–240.

Biniaz, A., Jain, K., Lubiw, A., Masárová, Z., Miltzow, T., Mondal, D., Naredla, A. M., Tkadlec, J., and Turcotte, A. (2019). Token swapping on trees. *CoRR*, abs/1903.06981:41.

Bonnet, É., Miltzow, T., and Rzążewski, P. (2018). Complexity of token swapping and its variants. *Algorithmica*, 80:2656--2682.

Bulteau, L., Fertin, G., and Rusu, I. (2015). Pancake flipping is hard. *Journal of Computer and System Sciences*, 81:1556 – 1574.

Chitturi, B. (2013). Upper bounds for sorting permutations with a transposition tree. *Discrete Mathematics Algorithms and Applications*, 5:24.

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

Cooperman, G. and Finkelstein, L. (1992). New methods for using cayley graphs in interconnection networks. *Discrete Applied Mathematics*, 37-38:95 – 118.

Edmonds, J. (1965). Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17:449–467.

Erdös, P. and Rado, R. (1987). *A Partition Calculus in Set Theory*, pages 179--241. Birkhäuser Boston, Boston, MA.

Ganesan, A. (2012a). Diameter of cayley graphs generated by transposition trees. *CoRR*, abs/1202.5888:11.

Ganesan, A. (2012b). 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. Springer.

Harel, D. and Tarjan, R. E. (1984). Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13:338–355.

Heath, L. and Vergara, J. (2003). Sorting by short swaps. *Journal of computational biology : a journal of computational molecular cell biology*, 10:775–89.

Ito, T., Demaine, E. D., Harvey, N. J., Papadimitriou, C. H., Sideri, M., Uehara, R., and Uno, Y. (2011). On the complexity of reconfiguration problems. *Theoretical Computer Science*, 412:1054 – 1065.

Jerrum, M. R. (1985). The complexity of finding minimum-length generator sequences. *Theoretical Computer Science*, 36:265 – 289.

Johnson, W. W. and Story, W. E. (1879). Notes on the "15" puzzle. *American Journal of Mathematics*, 2:397--404.

Kawahara, J., Saitoh, T., and Yoshinaka, R. (2017). The time complexity of the token swapping problem and its parallel variants. In *WALCOM: Algorithms and Computation*, pages 448--459, Cham. Springer International Publishing.

Kim, D. (2016). Sorting on graphs by adjacent swaps using permutation groups. *Computer Science Review*, 22:89–105.

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

Kraft, B. (2015). Diameters of cayley graphs generated by transposition trees. *Discrete Applied Mathematics*, 184:178 – 188.

Miltzow, T., Narins, L., Okamoto, Y., Rote, G., Thomas, A., and Uno, T. (2016). Tight exact and approximate algorithmic results on token swapping. *CoRR*, abs/1602.05150:19.

Mouawad, A. E. (2015). *On Reconfiguration Problems: Structure and Tractability*. PhD dissertation, University of Waterloo.

Nishimura, N. (2018). Introduction to reconfiguration. *Algorithms*, 11:52.

Pai, K.-J., Chang, R.-S., and Chang, J.-M. (2020). Constructing dual-cists of pancake graphs and performance assessment of protection routings on some cayley networks. *The Journal of Supercomputing*, 76:124546.

Razborov, A. (2002). Proof complexity of pigeonhole principles. In *Developments in Language Theory*, pages 100–116, Berlin, Heidelberg. Springer Berlin Heidelberg.

Siraichi, M. Y., Santos, V. F. d., Collange, C., and Pereira, F. M. Q. a. (2019). Qubit allocation as a combination of subgraph isomorphism and token swapping. *Proc. ACM Program. Lang.*, 3:1–29.

Siraichi, M. Y., Santos, V. F. d., Collange, S., and Pereira, F. M. Q. (2018). Qubit allocation. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pages 113--125, New York, NY, USA. ACM.

Smith, J. H. (1999). Factoring, into edge transpositions of a tree, permutations fixing a terminal vertex. *Journal of Combinatorial Theory, Series A*, 85:92--95.

Smith, J. H. (2011). Corrigendum to" factoring, into edge transpositions of a tree, permutations fixing a terminal vertex". *J. Comb. Theory, Ser. A*, 118:726--727.

van den Heuvel, J. (2013). The complexity of change. In *Surveys in Combinatorics*, pages 127–160, Cambridge. Cambridge University Press.

Vaughan, T. P. (1991). Bounds for the rank of a permutation on a tree. *Journal of Combinatorial Mathematics and Combinatorial Computing*, 30:129–148.

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

Yamanaka, K., Demaine, E. D., Ito, T., Kawahara, J., Kiyomi, M., Okamoto, Y., Saitoh, T., Suzuki, A., Uchizawa, K., and Uno, T. (2015a). Swapping labeled tokens on graphs. *Theoretical Computer Science*, 586:81 – 94.

Yamanaka, K., Horiyama, T., Neil, J. M., Kirkpatrick, D. G., Otachi, Y., Saitoh, T., Uehara, R., and Uno, Y. (2015b). Swapping colored tokens on graphs. In *Workshop on Algorithms and Data Structures*, page 16, Victoria, BC, Canada. Springer.

Yasui, G., Abe, K., and Yamanaka, K. (2015). Swapping labeled tokens on complete split graphs. *IEICE technical report. Speech*, 115:87–90.