Combinatorial Algorithms and Optimization

George Manoussakis

université PARIS-SACLAY

NNT : 2017SACLS517

UNIVERSITÉ PARIS SUD
Comprendre le monde, construire l'avenir

Combinatorial Algorithms and Optimization.

Thèse de doctorat de l'Université Paris-Saclay
préparée à l'Université Paris Sud.

École doctorale n°580 : sciences et technologies de l'information et de la communication (STIC)
Spécialité de doctorat : informatique

Thèse présentée et soutenue à Orsay, le 15 novembre 2017, par

George Manoussakis

Composition du Jury :

Christina Bazgan Professeure, Université Paris Dauphine (LAMSADE)	Présidente
Ralf Klasing Directeur de Recherche, CNRS (LaBRI)	Rapporteur
Dmitrii Pasechnik Professeur, Pembrook College Oxford	Rapporteur
Lionel Pournin Professeur, Université Paris 13 (Lipn)	Rapporteur
Michèle Sebag Directrice de recherche, CNRS (LRI)	Examinatrice
Ioan Todinca Professeur, Université d'Orléans (LIFO)	Examinateur
Johanne Cohen Directrice de Recherche, CNRS (LRI)	Co-Directrice de thèse
Antoine Deza Directeur de Recherche, CNRS (LRI)	Directeur de thèse

Thèse de doctorat

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre : Algorithmes combinatoires et Optimisation.

Mots clés : algorithmique, graphes, énumération, diamètre, polytope

Résumé : Nous nous intéressons à trois questions principales dans ce document. Les deux premières concernent des problèmes d'algorithmique de graphe. Nous introduisons d'abord la classe des graphes k-dégénérés qui est souvent utilisée pour modéliser des grands graphes éparses issus du monde réel, tels que les réseaux sociaux. Nous proposons de nouveaux algorithmes d'énumération pour ces graphes. Le but d'un algorithme d'énumération est de trouver tous les sous-graphes isomorphes à un graphe donné en entrée du problème. En particulier, nous construisons un algorithme énumérant tous les cycles simples de tailles fixés dans ces graphes, en temps optimal. La détection de cycle peut donner des indices sur les propriétés de connexité du graphe. Nous proposons aussi un algorithme dont la complexité dépend de la taille de la solution pour le problème d'énumération des cliques maximales de ces graphes. Ce problème modélise la question de la détection de communautés. Une communauté est formée par un ensemble d'individus qui
interagissent plus souvent entre eux qu'avec les autres. Ils s'agit donc de groupes d'individus qui ont tissés des liens plus forts ou qui ont des affinités communes. L'intérêt de la détection de communautés est multiple : identifier des profils types, effectuer des actions ciblées, mieux ajuster les recommandations, etc. La seconde question que nous étudions est aussi une question d'algorithmique de graphes, bien que le contexte soit différent. Nous considérons les graphes en tant que systèmes distribués. Chaque sommet a une capacité de calcul et peut communiquer avec ses voisins, à travers des canaux d'échanges modélisés par les arêtes du graphe. Dans ce contexte, nous nous intéressons à des questions liées à la notion de couplage. Un couplage est un ensemble d'arêtes dans le graphe tel que les arêtes de cet ensemble ne partagent pas de sommets deux à deux. Un couplage peut être utilisé, par exemple, dans les problèmes d'affectation des tâches et ce pour avoir une efficacité maximale, par exemple, chaque tâche est attribuée à une seule machine ou vice versa.

La recherche d'un couplage maximum dans un graphe biparti est d'ailleurs appelée le problème d'affectation. Nos algorithmes fonctionnent sans avoir à faire de supposition sur l'état initial du système, qui peut donc être correct ou incorrect. Des algorithmes distribués fonctionnant sous ces hypothèses sont appelés auto-stabilisant puisqu'ils doivent converger, ou stabiliser, vers un état correct après une initialisation quelconque. Cette approche est utile pour résoudre des problèmes qui peuvent apparaitre dans des systèmes où une intervention humaine n'est pas toujours possible, comme les satellites dans l'espace, par exemple. Dans ce cadre nous proposons un algorithme retournant une deux tiers approximation du couplage maximum. Nous proposons aussi un algorithme retournant un couplage maximal quand les communications sont restreintes de telle manière à simuler le paradigme du passage de message. Le troisième objet d'étude n'est pas directement lié à l'algorithmique de graphe, bien que quelques techniques classiques de ce domaine soient utilisées pour obtenir certains de nos résultats. Nous introduisons et étudions certaines familles de polytopes, appelées Zonotopes Primitifs, qui peuvent être décrits comme la somme de Minkowski de vecteurs primitifs. Ces vecteurs primitifs sont obtenus en considérant, de façon gloutonne, tous les vecteurs à norme bornée ayant des coordonnée première deux à deux. Nous prouvons certaines propriétés combinatoires de ces polytopes et illustrons la connexion avec le plus grand diamètre possible de l'enveloppe convexe de points à coordonnées entières à valeurs dans $[k]$, en dimension d. Cette question du diamètre est liée à l'étude de l'algorithme du Simplexe, puisque borner ce paramètre donne immédiatement une borne inférieure sur la complexité pire cas de cette algorithme. Dans un second temps, nous étudions des paramètres de petites instances de Zonotopes Primitifs, tels que leur nombre de sommets, entre autres.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France

université PARIS-SACLAY
ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Title: Combinatorial Algorithms and Optimization.

Keywords : algorithmics, graphs, enumeration, diameter, polytopes

Abstract: We start by studying the class of k-degenerate graphs which are often used to model sparse real-world graphs. We focus on enumeration questions for these graphs. That is, we try and provide algorithms which must output, without duplication, all the occurrences of some input subgraph. We investigate the questions of finding all cycles of some given size and all maximal cliques in the graph. Our two contributions are a worst-case output size optimal algorithm for fixed-size cycle enumeration and an output sensitive algorithm for maximal clique enumeration for this restricted class of graphs. In a second part we consider graphs in a distributed manner. We investigate questions related to finding matchings of the network, when no assumption is made on the initial state of the system. These algorithms are often referred to as self-stabilizing. Our two main contributions are an algorithm returning an approximation of the maximum matching and a new algorithm for maximal matching when communication simulates message passing. Finally, we introduce and investigate some special families of polytopes, namely primitive zonotopes, which can be described as the Minkowski sum of short primitive vectors. We highlight connections with the largest possible diameter of the convex hull of a set of points in dimension d whose coordinates are integers between 0 and k. Our main contributions are new lower bounds for this diameter question as well as descriptions of small instances of these objects.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France

Contents

Chapter 1

Introduction

We start this thesis by introducing some fundamental notions about graphs and graph algorithms in this chapter. This choice is motivated by the fact that, even though the three topics discussed in this thesis are quite different from one another, they all borrow tools and concepts from classical graph theory, to some extent.

After the introduction chapter, we will study graph algorithms in Part I of the document. Chapters 2 and 3 are dedicated to the study of some enumeration problems on restricted classes of graphs. Many real world graphs are sparse, that is, have few edges. Sparsity is often correlated to some other graphs parameter as degeneracy for instance. It turns out that many real world graphs are sparse and have small degeneracy, see [127] for example. Thus, it may be useful to design algorithms which rely on the exploitation of this parameter when the input is some large real world sparse graph. Enumeration questions are central in many domains such as computational biology and text mining [100, 151, 82] for instance and have been extensively studied from a theoretical and computational point of view. We first describe in Chapter 2 an algorithm listing all cycles of some given constant size in graphs with small degeneracy. Enumeration of cycles plays an important role in many practical problems. It is useful in the analysis of the World Wide Web and social networks, as the number of cycles can be used to identify connectivity patterns in a network. In fact, it has been shown that induced cycles effectively characterize connectivity structures of networks as a whole [120]. Enumeration of cycles is also used to understand ecological networks structures, such as food webs [129]. Another application is the nature of structure-property relationships in some chemical compounds that are related to the presence of chordless cycles [67].

In Chapter 3 we will study the problem of enumerating all the maximal cliques of k-degenerate graphs. We provide the first algorithm which requires only some polynomial function (of the degeneracy) time on average per clique. Clique finding has applications in many important problems. Finding cliques was first studied in the framework of social network analysis, as a way of finding closely-interacting communities [73]. In bioinformatics, clique finding procedures have been used to find recurring patterns in protein structures [69, 97, 98]. It has also been used to predict the structures of proteins from their molecular sequences [123], and to find similarities in shapes that may indicate relationships between proteins [63]. Other applications of clique finding problems include information retrieval [10], computer vision [80], computational topology [154], and electronic commerce [152].

Sometimes it may be also useful to see graphs as networks. In that context, the vertices or nodes are entities which have some computing power. They can communicate together and must solve a problem on the graph using only local knowledge. We may also require that the network is not perfect and presents errors

or failures. Thus, we want to formally introduce robustness properties and design algorithms which guarantee such properties. In the scope of this thesis, we study distributed algorithms that, from any initial state of the system (correct or incorrect) will converge to a correct state. These algorithms are called self-stabilizing since they can stabilize from any state to a correct state. In Chapters 4 and 5 we describe new self-stabilizing algorithms for matching related problems.

Finally, in Chapter 6 we study a special family of polytopes which have interesting properties regarding the maximum diameter of polytopes with integer coordinates. The notion of diameter is the one one expects, that is the diameter of the underlying polytope graph. To prove these new diameter bounds, standard counting techniques and classical combinatorial results concerning the number of disjoint perfect matchings are used. One central problem related to the diameter of polytopes (not necessarily with integer vertices) is the Hirsch conjecture formulated in 1957 which bounds the diameter of any polyhedron by a function of some of its parameters (number of facets and dimension). This question has drawn a lot of attention and recently the conjecture has been disproved by Santos' counterexample [124]. After that, researchers have been trying to determine the new upper bound since Santos' counterexample only violates Hirsch's conjecture by some small threshold. In the context of this thesis we shift our attention to specific polytopes, that is convex polytopes (and bounded) with integer coordinates. Our main contributions are constructive new lower bounds for the diameter of these polytopes as well as a broader description of the properties of the objects (called Primitive Zonotopes) that we introduce.

This first chapter is organized as follows. We give basic definition and notations regarding graphs in Section 1.1. Then we introduce matchings in Section 1.2 and degeneracy in Section 1.3. In Section 1.4.1 we describe, briefly, concepts about enumeration algorithms. Finally, in Section 1.4.2 we discuss some simple notions regarding self-stabilizing algorithms. The definitions and notions that we use in the chapter concerning lattice polytopes can be found in Section 6.1.

1.1 Basic graph concepts

In this section, we introduce basic notions for graphs that we use throughout the thesis. For further information about graphs, one can refer to [22, 148, 39] and references therein. We introduce now some fundamentals notions about graphs. Graphs are mathematical objects used to formally model real life concepts. Very simply put, a graph is a set of vertices and a set of edges between pairs of these vertices. For instance, social networks can be described as graphs: users are the vertices, and an edge exists between two users if they are socially connected on the network. Graphs were first introduced by Euler in his paper on the Seven Bridges of Königsberg [55]. The city of Königsberg in former Prussia was set across the Pregel River, and included two large islands. The mainland part of the city and the two islands were connected to each other by seven bridges. The problem was to find a walk through the city that would cross each of those bridges once and only once. Euler's solution to the problem is considered to be the first theorem of graph theory. Since then, graphs have found numerous applications in many domains such as Physics, Chemistry, Biology and Telecommunications, among others.

A *graph*, denoted $G = (V, E)$ is a set of *vertices* V and a (multi)-set of *edges* E. Every edge connects a pair of vertices. If the vertices are the same then the edge is often called a loop. See Figure 1.1 for an illustration. If an edge has an orientation then the graph is said to be *oriented* or directed. Otherwise it is *undirected*. In this

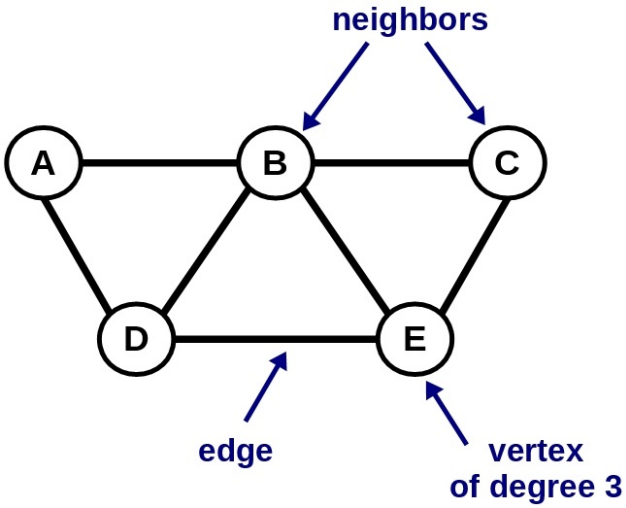


Figure 1.1 – A small connected graph. It has five vertices and seven edges. Vertex E has degree three since it is has three neighbors D, B and C.

thesis we will be using both definitions. For instance in Chapter 2 we will consider directed graphs. Often the set of edges is noted $E(G)$ and the set of vertices $V(G)$. If E is a set then the graph is simple, if it is a multiset the graph is a multigraph. By convention $|V(G)| = n$ and $|E(G)| = m$ are the number of vertices and edges, respectively. A graph with n vertices is said to be of *order* n.

An edge $e \in E$ will be noted $\{u, v\}$ where u and v are the vertices of the edge. When two vertices u and v belong to an edge we say that u and v are *adjacent*. For a certain vertex $v \in V(G)$, the set $adj(v) = \{w \in V(G) | \{u, w\} \in E(G)\}$ is called the *adjacency list* of v. Equivalently, the *open neighbourhood* of v, noted $N(v)$, is equal to $adj(v)$. The *closed neighbourhood* of v, noted $N[v]$ is the set $\{N(v) \cup \{v\}\}$. A graph G is usually represented either by its adjacency lists or by an adjacency matrix. The *adjacency lists* corresponds to the set $\{adj(v), v \in V(G)\}$. The *adjacency matrix* is an $n \times n$ matrix M such that M_{ij} is one when there is an edge from vertex i to vertex j, and zero if there is no edge.

The *degree* of a vertex $v \in V(G)$ is the number of edges that contain v as an endpoint. The *maximum degree* of a graph, denoted Δ, corresponds to the maximum degree among all the degrees of vertices in the graph. Recall that if the edges of a graph have a direction the graph is directed (equivalently oriented). Thus for a directed graph, when writing edge $e = \{u, v\}$ it is assumed that e is directed from u to v. It is an outgoing edge for vertex u and an incoming edge for vertex v. The *in-degree* of a node v is the number of incoming edges for vertex v and equivalently the *out-degree* is the number of its outgoing edges. If the out-degree of any vertex of the graph is bounded by some integer d, then the graph will be said to have a d-bounded orientation.

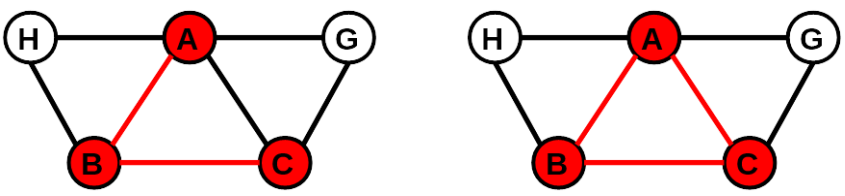


Figure 1.2 – We consider, in red, subgraphs of the graph on five vertices and seven edges. On the left we have a subgraph (not induced) since edge A, C is omitted. On the right, the subgraph on vertices A, B and C is induced.

A subgraph of a graph $G = (V, E)$ is a graph H such that the set of vertices $V(H) \subseteq V(G)$ and that the set of edges $E(H) \subseteq E(G)$. An induced subgraph is such a subgraph that contains all possible edges of graph G between pairs of vertices of $V(H)$. Thus when describing an induced subgraph H of a graph G it is enough to specify $V(H)$. See Figure 1.2 for an illustration. A subgraph of a graph is strict if it is not the graph itself.

A *path* is a graph whose vertices can be listed in order $v_1, v_2, ..., v_n$ such that the edges are $\{v_i, v_{i+1}\}$ for $i \in \{1, 2, ..., n-1\}$. A graph is *connected* if there exists a path between any two of its vertices. Otherwise it is *disconnected*. A path $v_1, v_2, ..., v_n$ such that $\{v_1, v_n\}$ is an edge is called a *cycle*. The length p of a cycle is the number of its vertices. See Figure 1.3 for an example. A cycle with p vertices is often noted C_p. A path or cycle with no repeated vertices is said *simple*. A graph in which there are no cycles is called a *forest*. A cycle is *chordless* or induced if is has no edge connecting two of its vertices. See Figure 1.3.

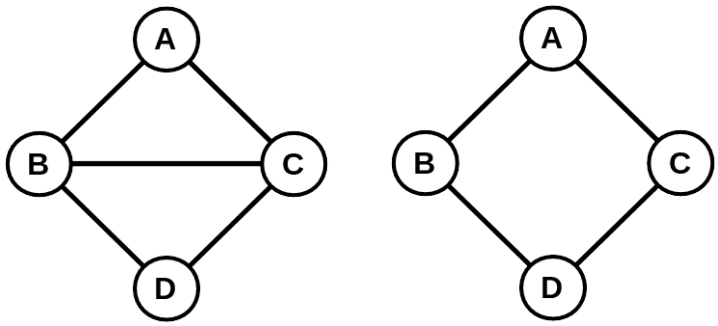


Figure 1.3 – A cycle on four vertices is presented on the left. A chordless one in the middle.

In an oriented graph, a directed cycle is a cycle in which all the edges are oriented in the same direction. If there is no such subgraph we say that the orientation is acyclic. For instance, in Chapter 2 we will consider graphs with an acyclic orientation in which all vertices have outdegree bounded by some integer k, called degeneracy. This notion is introduced in Section 1.3.

A *clique* K is a graph such that every two distinct vertices of $V(K)$ are adjacent. Given some graph G, a clique is *maximal* in G if it cannot be extended by including one more adjacent vertex. See Figure 1.4 for an example. It is *maximum* if there is no other clique of G with more vertices. Conversely, a graph in which no two pairwise vertices are adjacent is called an independent set. Similarly to cliques, an independent set can be maximal or maximum. In Chapter 3 we will study the question of finding all maximal cliques of some given graph.

A graph is said *dense* if its number of edges is close to the maximal number of edges. For a n-order graph, observe that the maximal number of edges is $n(n-1)/2$. On the opposite, a graph with few edges is said to be *sparse*. The distinction between sparse and dense graphs often depends on the context. In Section 1.3, we introduce degeneracy which is a graph parameter often used to measure the density of the graph.

1.2 Matchings

Chapters 4 and 5 are dedicated to the design of algorithms for matching problems in some specified distributed setup. A *matching* M in a graph G is a subset of the

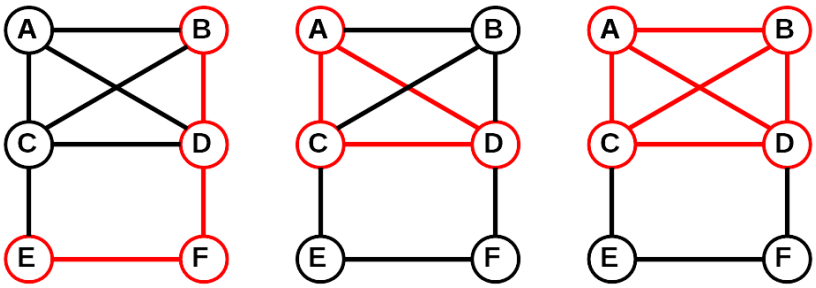


Figure 1.4 – We present different notions for cliques. On the left, the subgraph in red is not a clique since, for instance, vertices B and E are not adjacent. In the middle we have clique. It is not maximal (thus not maximum either) since adding vertex B would yield a larger clique. On the right we have a maximal clique which is also maximum in the graph.

edges of G which do not have a common end vertex. A matching is *maximal* if no proper superset of M is also a matching whereas a *maximum* matching is a maximal matching with the highest cardinality among all possible maximal matchings. If all the vertices of the graph belong to some edge in the matching, we say that it is *perfect*. See Figure 1.6 for an illustration.

Given a real number k, we say that a matching is a *k-approximation* of the maximum matching if it contains at least $k \times Max$ edges where Max is the number of edges of the maximum matching. In this thesis we are interested in two problems concerning matchings. In some distributed setup (introduced in Section 1.4.2) we want to design an algorithm returning a 2/3-approximation of a maximum matching of the input graph (Chapter 4) and also an algorithm returning a maximal matching (Chapter 5). A 2/3-approximation of the maximum matching (also called a 1-maximal matching) is expected to have more edges than a maximal matching, which only guarantees a 1/2-approximation.

Our algorithm of Chapter 4 which returns a 2/3-approximation is essentially based on the exploitation of augmenting paths of the input graph. When not considering a distributed setup, most linear time approximation algorithms for the maximum matching problem are based on a simple greedy strategy exploiting *augmenting paths*. An *augmenting path* is a path, starting and ending in an unmatched node, and where every other edge is either unmatched or matched; *i.e.* for each consecutive pair of edges, exactly one of them must belong to the matching. Let us consider the example in Figure 1.5. In this figure, B and C are matched nodes and A, D are unmatched nodes. The path (A, B, C, D) is an augmenting path of length 3 (written 3-*augmenting path*).

Figure 1.5 – A small example of an augmenting path. The edge in bold is the matched edge, the other edges are unmatched.

It is well known [79] that given a graph $G = (V, E)$ and a matching $M \subseteq E$, if there is no augmenting path of length $2k-1$ or less, then M is a $\frac{k}{k+1}-$approximation of the maximum matching. See [50] for the weighted version of this theorem. The greedy strategy in [50, 121] consists in finding all augmenting paths of length ℓ or less and by switching matched and unmatched edges of these paths in order to improve the maximum matching approximation.

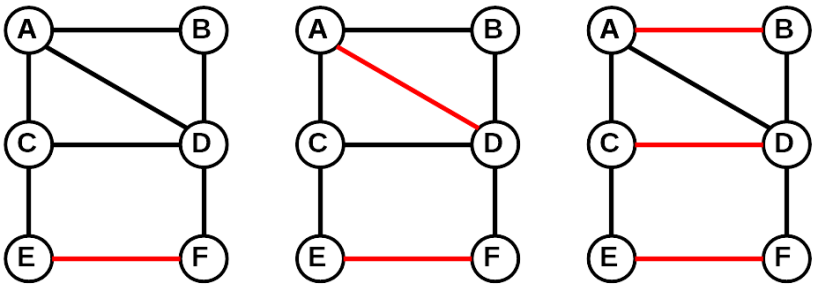


Figure 1.6 – We present the different matching notions on the given graph. We put the matched edges in red and the normal edges in black. On the left, we have a matching. In the middle we have a maximal matching as adding any other edge would yield two edges with some same endpoint. On the right we have a maximum matching, which is also perfect, as all the vertices are some endpoints of its edges.

The algorithm presented in Chapter 5 returns a maximal matching in some distributed setup (introduced in detail in the chapter and in Section 1.4.2). The strategy which we follow in our approach is essentially the basic greedy strategy which returns a maximal matching in classical graphs which we adapt for our distributed setup. It works as follows. Initially, the matching is just the empty set. Then edges are considered one by one and added to the matching if they do not have any vertex in common with the current matching. It is easy to see that this procedure yields a maximal matching.

1.3 Degeneracy of a graph

In Chapters 2 and 3 we will study algorithmic problems where the input is not any general graph but a restricted family of graphs, namely k-degenerate graphs. The *degeneracy* of a graph is a common measure of its sparsity. Degenerate graphs have been extensively studied as real life graphs are often sparse and have low degeneracy, as well as other important classes of graphs. For instance, the World Wide Web graph, citation networks, and collaboration graphs have low degeneracy [68]. The Barabàsi-Albert model of preferential attachment [12], frequently used as a model for social networks, produces graphs with bounded degeneracy. Furthermore, planar graphs have degeneracy at most five [102]. A graph is planar if it can be drawn on the plane in such a way that its edges intersect only at their endpoints. In other words, it can be drawn in such a way that no two edges cross each other. Degenerate graphs have been introduced by Lick *et al.* [102]:

Definition 1.3.1 ([102]). *The degeneracy of a graph G is the smallest integer k such that every subgraph of G contains a vertex of degree at most k.*

A graph with degeneracy 3 is presented in Figure 1.7. Degeneracy is also known as the k-core number [14], width [58], and linkage [93] of a graph. A k-degenerate graph has maximum clique size less than $k+1$ as all the vertices of a clique of size more than $k+1$ have degree strictly more that k. A graph which has degeneracy k has also a degeneracy ordering, or k-degenerate ordering, which is an ordering such that each vertex has k or fewer neighbors that come later in the ordering. Figure 1.7 shows a possible degeneracy ordering for the example graph. This ordering can be constructed by removing iteratively a vertex of G with degree k or less. Since, by assumption, the graph is k-degenerate, observe that such a vertex always exists. Conversely, if G has an ordering with this property, then it is k-degenerate, since for

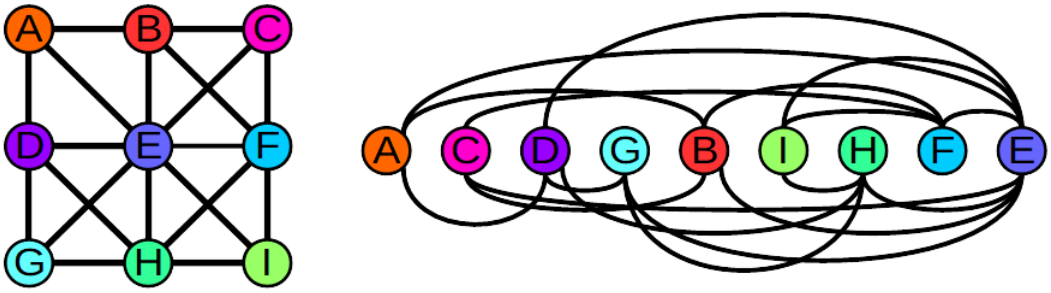


Figure 1.7 – On the left, a small graph with degeneracy 3. On the right, we have a 3-degenerate ordering of the graph: $A, C, D, G, B, I, H, F, E$.

any subgraph H of G, the vertex of H that comes first in the degeneracy ordering has k or less neighbors in H. This yields the following proposition, proved by Lick *et al.* [102]:

Proposition 1.3.2 ([102]). *A graph G is k-degenerate if and only if it has a k-degenerate ordering.*

Equivalently, a third definition is that degeneracy is the smallest integer k such that the graph has an acyclic orientation with out-degree bounded by k [31]. Such an orientation can be found by orienting each edge from its earlier endpoint to its later endpoint in a degeneracy ordering. Conversely if such an orientation is given then a degeneracy ordering may be found from that orientation.

Degeneracy is a robust measure of sparsity: it is within a constant factor of other popular measures of sparsity such as arboricity and thickness. Degeneracy, along with a degeneracy ordering, can be very easily computed by a simple greedy strategy of iteratively removing a vertex with smallest degree (and incident edges) from the graph until it is empty. The degeneracy is the maximum of the degrees of the vertices at the time they are removed from the graph, and the degeneracy ordering is the order in which vertices are removed from the graph [87]. Computing the degeneracy and a degenerate ordering of a graph can be done in time $\mathcal{O}(n+m)$ [14].

It is possible to have a more efficient adjacency representation (than adjacency lists) of a graph using its degeneracy. We introduce this notion in the next definition and then prove that it can be constructed relatively fast in Lemma 1.3.4.

Definition 1.3.3 (folklore). *Let $G = (V, E)$ be a k-degenerate graph and assume it is directed with a k-bounded orientation. Assume that G is given by the adjacency list for each vertex. The degenerate adjacency list of a vertex $x \in V$ is its adjacency list in which every vertex that is pointing towards x has been deleted and its sorted degenerate adjacency lists is its degenerate adjacency lists which has been sorted.*

This adjacency structure will be used in both our new algorithms for sparse graphs in Chapters 2 and 3. It is not too costly to build and adjacency queries can be sped up.

Lemma 1.3.4. *The sorted degenerate adjacency lists of a n-order k-degenerate graph G can be computed in time $\mathcal{O}(nk \log k)$ and adjacency queries can be done in time $\mathcal{O}(\log k)$ using these modified lists.*

Proof. Assume that we have the adjacency lists of G. Let $x \in V$ and let d_x be its degree and d_x^+ its out-degree. In time $\mathcal{O}(d_x)$ remove all vertices from its adjacency list that are pointing towards it. This takes total time $\mathcal{O}(m)$ and yields the degenerate adjacency list. Then we can sort this new adjacency list in time $\mathcal{O}(d_x^+ \log d_x^+) =$

$\mathcal{O}(k \log k)$ since a k-degenerate graphs has an acyclic orientation with out-degree at most k. Repeat the procedure for all the vertices of the graph. This is done in total time $\mathcal{O}(nk \log k + m) = \mathcal{O}(nk \log k)$. These new adjacency lists are of size at most k and they are sorted. This yields the proof for the time complexity of adjacency queries.

□

1.4 Graph algorithms

1.4.1 Enumeration algorithms

Both our contributions of Chapters 4 and 5 are enumeration algorithms. The design of such algorithms listing all possible solutions of a given problem dates back to the 1950s [65, 139]. Simply put, the goal of enumeration algorithms is to output all the solutions of a given problem. They were first studied in the area of complexity and optimization [90, 145], and then found applications in several other domains, including bioinformatics, machine learning, network analytics, and social analysis [2, 111, 126]. For instance, the enumeration of triangles has drawn a lot of attention [21, 4, 85, 125, 16] as well as their generalizations such as cliques and other dense subgraphs [23, 27, 30, 37, 53, 76, 104, 117, 137, 143].

The number of solutions of many enumeration problems is usually exponential in the size of the instance, which implies that enumeration algorithms require often at least exponential time. On the other hand, when the number of solutions is polynomial, one should expect the algorithm to be polynomial as well.

In this context, algorithms which have time complexity depending on the number of solutions (referred to as output size) have been developed. They are categorized in the following way [90]:

Definition 1.4.1. *An enumeration algorithm is polynomial total time if the time required to output all the solutions is bounded by a polynomial in the size of the input and the number of solutions.*

For example, the algorithm of Paull and Unger [119] enumerates all the independent sets of a n-order graph in polynomial total time $\mathcal{O}(n^2\alpha)$ where α is the number of independent sets of the graph.

Definition 1.4.2. *An enumeration algorithm is polynomial delay if it generates the solutions, one after the other in some order, in such a way that the delay until the first is output, and thereafter the delay between any two consecutive solutions, is bounded by a polynomial in the input size.*

Simply put, polynomial total time means that the delay between any output of two consecutive solutions has to be polynomial on the average, while the polynomial delay implies that the maximum delay has to be polynomial. Hence, Definition 1.4.2 implies Definition 1.4.1. One can refer to [146] for a catalogue of classified enumeration algorithms. Another paradigm for enumeration problems are *worst-case output size optimal* solutions. Roughly put, these algorithms guarantee a complexity which matches the maximum possible size of the output. For instance, in a general n-order graph, the maximum possible number of maximal cliques is $\mathcal{O}(3^{n/3})$ [25, 112]. Tomita *et al.* [137] proved an algorithm enumerating all maximal cliques of a general graph in time $\mathcal{O}(3^{n/3})$. Thus, it is worst-case output size optimal since there exist graphs on which one cannot hope to do better, as even printing their maximal cliques would require $\Omega(3^{n/3})$ time.

There exist many techniques for enumeration questions in graphs. One can for instance consider the general backtracking approach to enumeration problems. Simply put, a backtracking algorithm starts from an empty set and then elements are added recursively to the solution by a sequence of candidate extension steps. The possible candidates are represented as the nodes of a tree structure, the potential search tree. Each candidate is the parent of the candidates that differ from it by a single extension step; the leaves of the tree are the partial candidates that cannot be extended any further. The backtracking algorithm traverses this search tree recursively in depth-first order. At each node c, the algorithm checks whether c can be completed to a valid solution. If it cannot, the sub-tree rooted at c is pruned. Otherwise, the algorithm checks whether c itself is a valid solution, and if so reports it to the user; and then recursively enumerates all sub-trees of c. Therefore, the actual search tree that is traversed by the algorithm is only a part of the potential tree. The total cost of the algorithm is the number of nodes of the actual tree times the cost of obtaining and processing each node. Other classical tools include binary search which is a branch and bound like recursive partition algorithm and reverse search which is, simply put, searching on a traversal tree defined by some parent-child relation.

In the scope of this thesis we will present two enumeration algorithms. The first one enumerates all maximal cliques of a k-degenerate graph in polynomial total time. We conjecture that currently, it is not polynomial time delay. The second algorithm lists all fixed-size simple cycles in k-degenerate graphs in worst-case output size optimal time. They are presented in Chapters 3 and 2, respectively. These results do not really use classical techniques. The algorithm on cycles is mostly relying on decomposition of the cycles into smaller easily enumerable parts which is an idea that has been used in other papers related to cycles finding and listing such as [5] and [99]. The other contribution relies mostly on ideas similar to kernelization techniques used for the design of fixed-parameter tractable algorithms. Essentially, we do not want to solve a large problem but rather try, through a preprocessing phase, to reduce the question to some smaller instance. In our contribution of Chapter 5, instead of solving a problem on our n-order graph input, we show that it is in fact equivalent to solving n problems on smaller graphs.

In the next section we want to introduce notions related to some other graph algorithmic problems where the input is a distributed environment in which vertices play an important role in the solution.

1.4.2 Self-stabilizing Algorithms

A distributed system is a model in which components such as computers, processes or other entities cooperate in order to achieve a common goal. They communicate together through exchange of messages or by sharing memory. This system can be represented by a graph in which the nodes are the computing entities and in which the edges represent the communication channels between them. The difficulty of solving problems in this model arises from the fact that the nodes only have a local vision of the system and that immediate communication is only possible with the neighbors.

Distributed systems can either be synchronous or asynchronous. They are synchronous if the existence of a global clock is assumed. The processors communicate simultaneously at each tick of this clock. Implicitly, this yields a bound on the message delay. On the other hand, a system is asynchronous if there is no global clock. Thus, it does not rely on the strict arrival times of messages and coordination is

achieved through other means.

A distributed algorithm is a piece of code that is run on all the nodes of the distributed system. Based on the knowledge the nodes gather through communication with their neighbors, they must cooperate in order to solve a global problem. Communication is usually modeled in one of the three following ways, as described by Dolev [47]:

— The *shared memory model*. In this model, nodes can read the registers of all their neighbors in one atomic step. They can only write in their own registers.

— The *read-write model*. Nodes can execute a single read or write operation at each atomic step. They can only write in their own registers.

— The *message passing model*. Nodes can either receive or send a message at each atomic step (not both simultaneously).

In the shared memory and read-write models two neighbors share a common memory whereas in the message passing model nodes exchange messages. In this thesis, we will present two algorithms for matching problems where we use the first two communication models. We also assume that the system is not *anonymous*. All nodes can be distinguished through identifiers, for instance. In an anonymous system, processors or nodes are identical.

Faults can happen in a distributed system. For instance, a computer may shutdown, or a communication link may break. Various external and internal events may disturb the system and lead to unwanted behaviors. Three main types of faults are distinguished [135].

— *Transient faults*. They occur once and then disappear.

— *Permanent faults*. They can occur at any time and they stay permanently.

— *Intermittent faults*. They can occur at any time in the execution.

Solutions have been proposed to deal with these possible faults and assure a correct functioning of the system. These solutions can essentially be categorized into two families [135]:

— *Robust Algorithms*. They typically need redundant components and information. They assume that with a bounded number of faults, the system will maintain a proper behavior. An exhaustive list of the expected faults is required.

— *Self-stabilizing Algorithms*. Starting from any state of the system, these algorithms guarantee that the system will eventually reach a correct state (*convergence property*). This behavior simulates the fact that memory and registers may hold incorrect values after a fault. They are also resilient to topological changes of the network since the system configuration after a topology change can be treated just like any other arbitrary starting configuration.

In the next section, we introduce in detail the notion of Self-stabilization. These explanations will be useful for the understanding of the algorithms and proofs given later in the thesis, in Chapters 4 and 5.

1.4.2.1 General properties of self-stabilizing algorithms

The concept of self-stabilization was introduced by Dijkstra [46] and popularized by Lamport [101]. For surveys about self-stabilizing algorithms, one can refer to [47, 70, 62]. Self-stabilization has many applications in systems where human intervention is not always possible after an error occurs. Such systems include sensor networks, computer networks or satellites.

A system is self-stabilizing if it is allowed to start from any possible state (equivalently configuration) of its entities and converges to a correct state in finite time, without any external intervention. Intuitively, a state of the system is simply the values of all the variables of the nodes. This behavior is refereed to as the *convergence* property. Another desirable behavior is the *closure* property. After reaching a correct state, the system is guaranteed to remain in a correct state. These two definitions are described precisely by Arora *et al.* [6]. We define formally these notions in the next section.

In most of the literature, self-stabilizing algorithms assume that the faults happening in the system are transient, or convergence may not be guaranteed if faults happen intermittently during the convergence phase. A self-stabilizing algorithm may be *silent* or *not silent*. It is silent if and only if the nodes keep their registers unmodified once a correct state has been reached. This is often the case with self-stabilizing algorithms for graph problems: for instance, once a maximum independent set has been computed in a system, the nodes do not need to perform any other modification of their variables. When considering silent algorithms, the *correctness* property is often mentioned: the final configurations must verify the specification of the algorithm. For instance, if the goal of the algorithm is to compute a maximal matching, then, in a correct state, the set of nodes of the system must form a maximal matching. More formally, every final configuration must be *legitimate* with regard to the predicate P specifying the problem meaning that every final configuration must satisfy P.

In the scope of this thesis, we will consider self-stabilizing silent algorithms and assume that the faults are of transient type.

1.4.2.2 Algorithms description

Formally, the considered system consists of a set of processes where two adjacent processes can communicate with each other. The communication relation is represented by an undirected graph $G = (V, E)$ where $|V| = n$ and $|E| = m$. Each process corresponds to a node in V and two processes u and v are adjacent if and only if $(u, v) \in E$. The set of neighbors of a process u is denoted by $N(u)$ and is the set of all processes adjacent to u, and Δ is the maximum degree of G. For basic definitions and notations regarding graphs, the reader can refer to Section 1.1.

Each process maintains a set of variables that specify the local state of the process. A *configuration* (or state) C is the local states of all processes in the system. Each process executes the same algorithm that consists of a set of rules. Each rule is of the following form:

$$< name >:: if < guard > then < command >.$$

The *name* is the name of the rule. The *guard* is a predicate over the variables of both the process and its neighbours. The *command* is a sequence of actions assigning new values to the local variables of the process.

A rule is *activable* in a configuration C if its guard in C is true. A process is *eligible* for the rule $\mathcal{R}$ in a configuration C if its rule $\mathcal{R}$ is activable in C and we say the process is *activable* in C. An *execution* is an alternate sequence of configurations and actions $\mathcal{E} = C_0, A_0, \ldots, C_i, A_i, \ldots$, such that $\forall i \in \mathbb{N}^*$, C_{i+1} is obtained by executing the command of at least one rule that is activable in C_i (a process that executes such a rule makes a *move*). More precisely, A_i is the non empty set of activable rules in C_i that has been executed to reach C_{i+1} and such that each process has at most one of its rules in A_i. We use the notation $C_i \mapsto C_{i+1}$

to denote this transition in $\mathcal{E}$. We also need an inclusion notion for executions. Let $\mathcal{E}' = C'_0, A'_0, \cdots, C'_k$ be a finite execution. We say $\mathcal{E}'$ is a *sub-execution* of $\mathcal{E}$ if and only if $\exists t \geq 0$ such that $\forall j \in [0, \cdots, k]$:$(C'_j = C_{j+t} \wedge A'_j = A_{j+t})$. Since an execution is a sequence of configurations, if C and C' are two such configurations in some execution $\mathcal{E}$, then we write $C \leq C'$ if and only if C appears before C' in $\mathcal{E}$ or if $C = C'$. Moreover, we write $\mathcal{E} \backslash C$ to denote all configurations of $\mathcal{E}$ except configuration C.

An *atomic operation* is such that no change can take place during its run, we usually assume that an atomic operation is instantaneous.

1.4.2.3 Daemons

A *daemon* or scheduler is a predicate on the executions. Roughly, the daemon has the role to select nodes which are eligible for some rule to execute their moves (called enabled or activable nodes). This mechanism plays the role of both scheduler and adversary against the stabilization of the algorithm. It tries to prevent the convergence of the algorithm by scheduling the worst possible execution. The two most common daemons are the following:

— The *central* daemon : An unique enabled node is selected for execution at each transition.

— The *distributed* daemon: Any non-empty subset of the enabled nodes is selected for execution at each transition.

A daemon is *fair* if every node which is eligible for some rule is eventually selected. On the other hand, a daemon is *unfair* if the execution of an activable process is delayed as long as there are other activable nodes. Notice that the distributed unfair daemon is the most general one since an algorithm which works under this daemon will also work correctly under all the other ones.

1.4.2.4 Complexity measures

An algorithm's efficiency is measured through its *complexity*. It often corresponds to the maximum (or worst-case) resource requirements (such as time, space, among others) of the algorithm. It usually depends on the size of the input. In case of distributed systems, the complexity is given as a function of the parameters of the underlying network graph, such as the number of processors or the number of communication links. For a survey on the complexity of distributed algorithm one can refer to [9].

When restricted to self-stabilizing algorithm, an introduction to their complexity analysis can be found in [47]. For these algorithms, a standard complexity measure is the number of *moves* which are executed by the nodes before the system reaches a correct state. A move is defined formally as follows.

Definition 1.4.3. *Node v does a move when it executes a rule for which it is activable. It corresponds to a transition from the state C_0 in which v was activable to a new state C_1 after it executed the enabled rule.*

Other complexity measures have also been considered in the literature. For instance, the *step* complexity corresponds to a time frame within which each node can make at most one move and such that all nodes make their move simultaneously. More formally :

Definition 1.4.4. *A step is a tuple (C_0, C_1), where C_0 and C_1 are configurations, such that all nodes doing moves during this step are enabled in configuration C_0 and such that C_1 is the configuration reached after these nodes have made their moves simultaneously.*

From this definition follows the definition of a *round*. It corresponds to the minimum number of steps required such that every node makes at least a move. We can now define the time complexity of a self-stabilizing algorithm.

Definition 1.4.5. *The time complexity of a self-stabilizing algorithm is the maximum (worst case) number of moves, steps or rounds the algorithm executes before reaching a correct state, independently of the starting configuration.*

In the scope of this thesis, we will design algorithms and prove complexity with respect to their maximum possible number of moves.

1.4.2.5 Composition

When studying algorithmic problems, it is often desirable to be able to reuse existing algorithms and build upon them to obtain new results. For instance, if one wishes to construct a 2/3-approximation of a maximum matching, one could first use an algorithm to compute a maximal matching in the system, and, in a second phase, design another algorithm taking as input this maximal matching, modify it, and obtain the wanted result. In that context, the notion of composition of algorithms arises.

In classical distributed systems composition can be achieved by essentially executing sequentially the algorithms. When the first algorithm is finished, the second one starts. In a self-stabilizing setup however, detecting termination of the first algorithm is not possible [133]. Nevertheless, composition can still be achieved [77, 133]. Roughly put, two algorithms A_1 and A_2 can be composed if they do not share variables (they do not write and read in the same set of variables). Algorithm A_2 can read the variables of A_1 while it is stabilizing. Since, by the convergence property, these variables will be correct at some point, then algorithm A_2 will be correct as well at some point. Formally, the composition of two self-stabilizing algorithms is defined as follows.

Definition 1.4.6. *Let A_1 and A_2 be two self-stabilizing algorithms that do not share any variables. The composition of A_1 and A_2 is the algorithm which consists of all variables and rules of both A_1 and A_2.*

To prove that the composition of A_1 and A_2 is self-stabilizing, it must be shown, intuitively, that they do not block each other. Formally put:

Theorem 1.4.7. *[77, 133] Let A_1 and A_2 be two self-stabilizing algorithms with specifications P_1 and P_2. The composition of A_1 and A_2 is self-stabilizing if:*

- *When algorithm A_1 has stabilized, P_1 holds forever*
- *If P_1 holds, algorithm A_2 stabilizes*
- *Once P_1 holds, the variables of A_1 read by A_2 do not change*
- *The daemon is fair with respect to A_1 and A_2*

Thus, the move complexity of the composition of two self-stabilizing algorithms is the product of the complexities of each algorithm [47].

Part I

Enumeration algorithms for k-degenerate graphs

Chapter 2

Fixed-size cycles enumeration

In this chapter we present our algorithm for enumerating all fixed length simple cycles in k-degenerate graphs, that can be found in our published paper [108].

2.1 Introduction

The question of finding *fixed* length simple induced and non induced cycles in planar and k-degenerate graphs has been extensively studied. Among other contributions, Papadimitriou *et al.* [118] presented an algorithm finding C_3's in planar graphs. Chiba *et al.* [30] and Chrobak *et al.* [31] proposed simpler linear time algorithms to find C_3's and the first of these papers also presents an algorithm finding C_4's. Both papers also apply their techniques to k-degenerate graphs. Richardson [122] gave an $\mathcal{O}(n \log n)$ algorithm finding C_5's and C_6's in planar graphs.

For any fixed cycle length, Alon *et al.* [5], gave algorithms for both general and k-degenerate graphs. Cai *et al.* [24] proposed algorithms finding induced cycles of any fixed size.

For the problem of finding all occurrences of any p-length simple cycle in planar graphs, assuming some constant bound on p, Eppstein [51] proposes an algorithm running in time $\mathcal{O}(n + \alpha)$ where α is the number of simple p-length cycles in the graph. His algorithm works for any subgraph of fixed size p and solves in fact the more general problem of *subgraph isomorphism in planar graphs*. His result has been later improved by Dorn [49], who reduces the time dependence in p. For short cycles of size six or less, Kowalik [99], proposes an algorithm listing all occurrences of these cycles in time $\mathcal{O}(n + \alpha)$. His algorithm is faster in practice than the one of Eppstein for planar graphs and also works for k-degenerate graphs, with complexity $\mathcal{O}(k^2 m + \alpha)$, for cycles of size up to five. He also proves that the maximal number of simple p-length cycles in a planar graph is $\Theta(n^{\lfloor p/2 \rfloor})$. More recently, Meeks [110] proposed a randomized algorithm, given a general graph G, enumerating any p-sized subgraph H in time $\mathcal{O}(\alpha f)$ where f is the time needed to find one occurrence of H in G. This result, together with the one of Alon *et al.* [5] for instance, yields an $\mathcal{O}(\alpha n^{\mathcal{O}(1)} k^{\mathcal{O}(1)})$ time algorithm finding all occurrences of a p-length simple cycle in k-degenerate graphs, assuming p constant.

Other contributions have also been made for general graphs. For the problem of finding *all cycles* (any length), Tarjan [132] gives an $\mathcal{O}(nm\alpha)$ time algorithm, where α is the total number of cycles of the graph. This complexity has been improved to $\mathcal{O}((n + m)\alpha)$ by Johnson [89]. More recently, Birmelé *et al.* [19] proposed an $\mathcal{O}(c)$ time algorithm where c is the number of edges of all the cycles of the graph. Uno *et al.* [144] proposed an $\mathcal{O}((n+m)\alpha)$ algorithm finding all chordless cycles. We are not sure whether these algorithms can be easily adapted to output exactly all

p-length simple cycles in k-degenerate or general graphs with similar complexities but where α would be the number of p-length simple cycles (instead of the number of all cycles). For the problem of counting all cycles of size less than some constant p, Giscard *et al.* [66] propose an algorithm running in time $\mathcal{O}(\Delta|S_p|)$ where Δ is the maximum degree and $|S_p|$ the number of induced subgraphs with p or less vertices in the graph. Alon *et al.* [5] proposed an algorithm counting cycles of size less than 7 in time $\mathcal{O}(n^\omega)$ where ω is the exponent of matrix multiplication (the smallest value for which there is a known $\mathcal{O}(n^\omega)$ matrix-multiplication algorithm). Williams *et al.* [150] and Björklund *et al.* [20] also give algorithms which can be used to count fixed-size cycles.

Our contribution is a simple algorithm listing all p-length simple cycles in a n-order k-degenerate graph in time $\mathcal{O}(n^{\lfloor p/2 \rfloor} k^{\lceil p/2 \rceil} \log k)$, assuming that the graph is stored in an adjacency list data structure. If we have its adjacency matrix the time complexity can be improved to $\mathcal{O}(n^{\lfloor p/2 \rfloor} k^{\lceil p/2 \rceil})$. We then show that this complexity is worst-case output size optimal by proving that the maximal number of p-length simple cycles in an n-order k degenerate graph is $\Theta(n^{\lfloor p/2 \rfloor} k^{\lceil p/2 \rceil})$. These results also hold for induced cycles. To the best of our knowledge, this is the first such algorithm. It differs from the one of Meeks described before since it is deterministic, self-contained and can have better or worst time complexity depending on the number of simple p-length cycles of the input graph. Further improvements are discussed in the conclusion.

Our complexities are given assuming a constant bound on p. The exact dependence in p is described later but is exponential. Our approach for the main algorithm of this section is the following. We first show that in a k-degenerate graph, any p-length cycle can be decomposed into small special paths, namely t-paths, introduced in Definition 2.2.2. We then prove that these t-paths can be computed and combined efficiently to generate candidate cycles. With some more work, we can then output exactly all p-length simple cycles of the graph.

The organization of rest of the section is as follows. In Section 2.2 we introduce notations and definitions. In Section 2.3 we prove preliminary results for paths, t-paths and cycles. Using these results, we describe and prove algorithms in Section 2.4. The correctness and time complexity of the main algorithm is proved in Theorem 2.4.3 and the bound for the number of cycles in Theorem 2.4.5.

2.2 Definitions

We introduce and recall notations and definitions that we use throughout the section. By k we will denote the degeneracy of the graph and by p the length of the cycles we seek to list. When not specified, $G = (V, E)$ is a simple connected graph. As mentioned in Section 1.3, given a k-degenerate graph G, one can compute in time $\mathcal{O}(m)$ its degeneracy ordering [14]. This ordering also yields an acyclic orientation of the edges such that every vertex has out-degree at most k. From now on we will consider k-degenerate graphs as oriented acyclic graphs with out-degree bounded by k. If (x, y) is an edge of some oriented graph G we will write $x \to y$ and say that x is oriented towards y if edge (x, y) is oriented towards y.

Definition 2.2.1. *An oriented path $P : p_1, p_2, ..., p_x$ is* increasing *(resp.* decreasing*) with respect to p_1 if $p_1 \to p_2 \to ... \to p_x$ (resp. $p_1 \leftarrow p_2 \leftarrow ... \leftarrow p_x$).*

In the next definition, we introduce t-paths, which is the main ingredient of our algorithm for cycle enumeration.

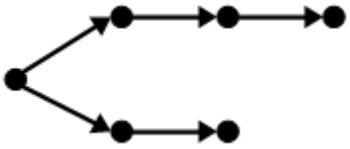


Figure 2.1 – A strict t-path of size $(2, 3)$.

Definition 2.2.2. *Let G be an oriented graph and $i, j \in \mathbb{N}$. A t-path $\mathcal{P}$ of size (i, j) is a path of $i + 1 + j$ vertices, $v_1, v_2, ..v_i, r, u_1, u_2, ...u_j$ such that $\mathcal{P}_l : r, v_i, v_{i-1}, ..., v_1$ and $\mathcal{P}_r : r, u_1, u_2, ..., u_j$ are increasing paths with respect to r. Vertices v_1 and u_j are called the end vertices of $\mathcal{P}$, vertex r its center. If $i, j \in \mathbb{N}^+$, we say that $\mathcal{P}$ is a strict t-path. See Figure 2.2.2.*

In a k-degenerate graph, it is easy to count and enumerate all t-paths of some given size. We prove that fact in the next lemma and its two corollaries.

Lemma 2.2.3. *Let $G = (V, E)$ be a k-degenerate graph. Assume that we have the sorted degenerate adjacency lists of G. Given a vertex $x \in V$ and $i \in \mathbb{N}$, we can compute all increasing paths of size i, starting with x in graph G in time $\mathcal{O}(k^i)$. There are at most $\mathcal{O}(k^i)$ such paths.*

Proof. Start with the sorted degenerate adjacency list of vertex x. By definition it is of size at most k. For every vertex y in this list, construct a candidate path of size one containing x and y. Note that there are at most k such candidate paths. For each such path, generate all k candidates paths of size two where the third vertex is a vertex of the degenerate adjacency list of the second vertex of the path. There are k^2 such candidates paths of size two in total. Go on in this fashion until all paths of size i have been generated. This procedure takes time $\mathcal{O}(k^i)$: at step $h < i$ we must consider at most k vertices from each of the previously computed k^{h-1} degenerate adjacency lists.

□

Corollary 2.2.4. *Let $G = (V, E)$ be a k-degenerate graph. Assume that we have the sorted degenerate adjacency lists of G. Given a vertex $x \in V$ and $i, j \in \mathbb{N}$, we can compute all t-paths of size (i, j) with center x, in G, in time $\mathcal{O}(k^{i+j})$. There are at most $\mathcal{O}(k^{i+j})$ such t-paths.*

Corollary 2.2.5. *Let $G = (V, E)$ be a k-degenerate graph. Assume that we have the sorted degenerate adjacency lists of G. Given $i, j \in \mathbb{N}$, we can compute all t-paths of size (i, j) in G, in time $\mathcal{O}(nk^{i+j})$. There are at most $\mathcal{O}(nk^{i+j})$ such t-paths.*

We need a few more definitions to be able to describe formally how we decompose cycles into t-paths. This decomposition is proposed in the next section in Lemma 2.3.1.

Definition 2.2.6. *Let G be an oriented graph and $\mathcal{P}_1$, $\mathcal{P}_2$ two t-paths of G. They are adjacent if they do not have any vertex in common but one or two of their end vertices.*

Definition 2.2.7. *Let G be an oriented graph and $x = (i, j) \in \mathbb{N}^2$. We say that we can associate a t-path to x in G if there exists a t-path of $i + 1 + j$ vertices in G.*

Definition 2.2.8. *Let x, y, z be three consecutive nodes in $\mathcal{C}$, an oriented simple cycle. Node y is a root of $\mathcal{C}$ if $x \leftarrow y \rightarrow z$.*

Observation 2.2.9. *An acyclic oriented simple cycle has a root.*

2.3 Basic Results

We prove simple results concerning paths, t-paths and cycles. These lemmas are used to prove the correctness and time complexity of Algorithm 2, presented in Section 2.4. In the next lemma, we show that, essentially, any cycle can be decomposed into t-paths.

Lemma 2.3.1. *Let $\mathcal{C}$ be a simple cycle of size p. For any possible acyclic orientation of $\mathcal{C}$, we can compute a list $\mathcal{L}$ of strictly positive integer pairs $(l_1, l_2), ..., (l_{r-1}, l_r)$ in time $\mathcal{O}(p)$ with the following properties: (See Figure 2.2.)*

(i) To each pair (l_i, l_{i+1}) we can associate a strict t-path such that $\mathcal{C}$ can be decomposed into $|\mathcal{L}|$ strict adjacent t-paths: one for each pair of $\mathcal{L}$.

(ii) If $|\mathcal{L}| > 2$, two strict t-paths associated to pairs of $\mathcal{L}$ are adjacent if and only if their associated pairs are consecutive in $\mathcal{L}$ (modulo $|\mathcal{L}|$).

(iii) If $|\mathcal{L}| > 2$, two t-paths t_1 and t_2 associated to two consecutive pairs (l_i, l_{i+1}) and (l_{i+2}, l_{i+3}) (modulo $|\mathcal{L}|$) have one common vertex: it is the end vertex of the increasing path $P_{l_{i+1}}$ of t_1 and the end vertex of the increasing path $P_{l_{i+2}}$ of t_2.

Proof. We proceed as follows. The first step is to find in time $\mathcal{O}(p)$ a root r of $\mathcal{C}$. Let x and y be its two neighbours in $\mathcal{C}$. Find the two longest increasing paths P_1 and P_2 with respect to r in $\mathcal{C}$ going through x and y. Note that paths P_1 and P_2 are well defined. This can be done in time $\mathcal{O}(|P_1| + |P_2|)$. After that put the corresponding pair in $\mathcal{L}$. If the end vertices of P_1 and P_2 are the same, we are done: the cycle itself is a t-path and $\mathcal{L} = (|P_1|, |P_2|)$. Observe that in that case property (i) is verified.

Otherwise, if the end vertices of P_1 and P_2 are not the same, we proceed with step two. Start from the end vertex v_2 of P_2 and find the longest decreasing path P_3 in $\mathcal{C}$ with respect to v_2. Observe that P_3 exists necessarily, by definition of P_2. Finding the vertices of P_3 is done in $\mathcal{O}(|P_3|)$. The end vertex v_3 of P_3 is a root, by definition of P_3. Observe also that v_3 and v_1 (the end vertex of P_1) are distinct. If that was not the case then P_1 would not have been the longest increasing path starting from r, which is a contradiction by construction of P_1. Now find the longest increasing path with respect to v_3 going in the other direction than P_3, call it P_4. It exists since v_3 and v_1 are distinct. If its end vertex v_4 is equal to v_1, then $\mathcal{C}$ can be decomposed into two strict adjacent t-paths and $\mathcal{L} = ((|P_1|, |P_2|), (|P_3|, |P_4|))$. Observe that the three properties for $\mathcal{L}$ are verified in that case.

Otherwise, if v_4 and v_1 are distinct, we proceed exactly as in step 2, but starting this time with the end vertex v_4 of P_4. We proceed in this fashion until we reach vertex v_1.

□

In the next three lemmas we show bounds on the number of t-paths that can decompose a cycle, their total size and their total number in a given graph. We first prove that a cycle cannot be decomposed into too many t-paths.

Lemma 2.3.2. *Let $\mathcal{C}$ be a simple cycle of size p with an acyclic orientation. Let $\mathcal{L} : (l_1, l_2), ..., (l_{r-1}, l_r)$ be the list of integer pairs associated to $\mathcal{C}$, as defined in Lemma 2.3.1. List $\mathcal{L}$ is at most of size $\lfloor p/2 \rfloor$.*

Proof. Assume first that p is even. Assume by contradiction that list $\mathcal{L}$ is of size $s > p/2$. By construction of $\mathcal{L}$, each pair (l_j, l_{j+1}) of $\mathcal{L}$ can be associated to a strict t-path of G. By Definition 2.2.2 we have that $l_j, l_{j+1} \in \mathbb{N}^*$. Thus each t-path associated to a pair in $\mathcal{L}$ has at least three vertices. This implies that the s many

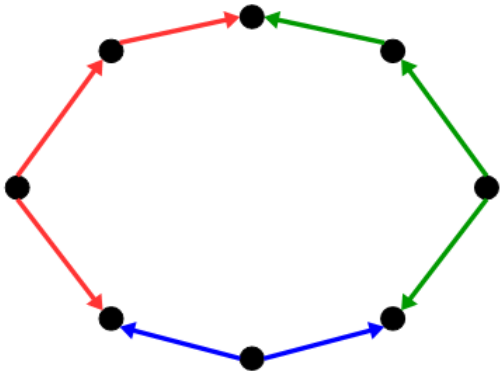


Figure 2.2 – An example of t-path decomposition of a cycle. The associated list is $\mathcal{L} = (1,2),(2,1),(1,1)$ assuming that pair $(1,2)$ is associated to the red t-path.

t-paths have $3s$ vertices altogether. But two consecutive t-paths in $\mathcal{L}$ have also a common vertex. Thus, in total, the s many t-paths have at least $3s - s = 2s > p$ vertices which gives the contradiction in that case.

Assume now that p is odd. We first show that there exists at least one strict t-path associated to some pair in $\mathcal{L}$ which has four or more vertices. By definition a strict t-path can not have only one or two vertices. Thus assume by contradiction that all s pairs in $\mathcal{L}$ are associated to strict t-paths of size three. As in the previous case, the total number of vertices of these t-paths is $3s - s = 2s$. By definition, this number is equal to p the size of $\mathcal{C}$ which is odd in that case, thus we have a contradiction. This implies that there exists at least one strict t-path associated to some pair in $\mathcal{L}$ which has four or more vertices. Assume now by contradiction that $s > \lfloor p/2 \rfloor = \frac{p-1}{2}$. Since at least one t-path of $\mathcal{L}$ has size four, then the s many t-paths (to which we remove the common vertices) have in total at least $3(s-1)+4-s = 2s+1$ vertices. To conclude the proof observe that $2s+1 > 2\frac{p-1}{2}+1 = p$, which yields the contradiction.

□

Lemma 2.3.3. *Let $\mathcal{C}$ be a simple cycle of size p with an acyclic orientation. Let $\mathcal{L} : (l_1,l_2),...,(l_{r-1},l_r)$ be the list of integer pairs associated to $\mathcal{C}$, as defined in Lemma 2.3.1. We have that $\sum_{j=1}^{r} l_j = p$*

Proof. As defined in Lemma 2.3.1, $\mathcal{C}$ can be decomposed into $|\mathcal{L}|$ t-paths, one for each pair of $\mathcal{L}$ such that two consecutive pairs (modulo $|\mathcal{L}|$) correspond to adjacent t-paths with one end vertex in common. By Definition 2.2.7, a t-path associated to a pair (l_i, l_{i+1}) is of size $l_i + 1 + l_{i+1}$. This implies that the t-paths of $\mathcal{L}$ have total size $(\sum_{j=1}^{r} l_j) + r$. Since cycle $\mathcal{C}$ can be decomposed into the t-paths associated to list $\mathcal{L}$, that two consecutive t-paths have one end vertex in common and that there are r such t-paths, we get that $(\sum_{j=1}^{r} l_j) + r = p + r$, which completes the proof.

□

2.4 Algorithm

We prove Theorem 2.4.3 which is the main result the section. For the sake of clarity, we first start by describing a simpler algorithm, namely Algorithm 2

and prove in Theorems 2.4.1 and 2.4.2 that it solves the problem of finding all p-length simple cycles in time $\mathcal{O}(n^{\lfloor p/2 \rfloor} k^p)$. Then we show how to modify it to get the claimed $\mathcal{O}(n^{\lfloor p/2 \rfloor} k^{\lceil p/2 \rceil} \log k)$ and $\mathcal{O}(n^{\lfloor p/2 \rfloor} k^{\lceil p/2 \rceil})$ complexities, in Theorem 2.4.3. Roughly put, Algorithm 2 works as follows. First, it computes all possible t-path decompositions of a p-length simple cycle. Then it finds all the corresponding t-paths in the graph and combines them to form candidate cycles. The length p of the cycles is supposed of constant size in the section.

Algorithm 1:

Data: A graph G and $p \in \mathbb{N}$.
Result: All simple p-length cycles of G.

1 Compute k the degeneracy of G and a degeneracy ordering σ_G. Construct an acyclic orientation of G with bounded out-degree k.
2 Compute the sorted degenerate adjacency lists of G.
3 Initialize $\mathcal{D} : 1, 2, ..., p$ a simple cycle.
4 Compute all acyclic orientations of $\mathcal{D}$.
5 **for** *each such orientation* **do**
6 Compute the ordered list $\mathcal{L} = p_1, ..., p_r$ of pairs associated to the strict t-paths of $\mathcal{D}$.
7 **for** $j = 1$ **to** r **do**
8 Compute all possible strict t-paths associated to pair p_j in G, put them in a set $\mathcal{S}_j$.
9 **end**
10 Compute all possible lists $\mathcal{C} = c_1, ..., c_r$ with $c_i \in \mathcal{S}_i$
11 **for** *each such list* $\mathcal{C}$ **do**
12 Check if it is a simple cycle:
 — check if vertices are unique except for the end vertices of the t-paths
 — check if two consecutive t-paths have a common end vertex
  if *yes* **then**
  Let s be the string obtained by sorting the vertices of $\mathcal{C}$ by increasing identifier.
  Search string s in T.
  if *it does not exist in* T **then**
  Output s.
  Insert it in T.
  end
  end
13 **end**
14 **end**

Theorem 2.4.1. *At the end of Algorithm 2, all p-length simple cycles of the graph G have been outputted exactly once.*

Proof. Assume first by contradiction that there exists a p-length simple cycle $\mathcal{C}_1 : c_1, c_2, ..., c_p$ of G which has not been outputted by Algorithm 2. Without loss of generality assume that c_2 has lowest ranking in the degeneracy ordering. As defined and proved in Lemma 2.3.1, depending on the orientation of $\mathcal{C}_1$, we can compute a list of integer pairs $\mathcal{L}_1 : p_1, p_2, ..., p_{r-1}, p_r$ that corresponds to the sizes of adjacent strict t-paths such that $\mathcal{C}_1$ can be decomposed into these t-paths. Up to renaming the vertices, the orientation of $\mathcal{C}_1$ has been generated at Line 4 of Algorithm 2. This

implies that there exists a list $\mathcal{L}$ computed in Line 6 which is equal to $\mathcal{L}_1$. Since in Line 8 all t-paths associated to the pairs of $\mathcal{L}$ are computed, this implies that there exists some list $\mathcal{C}$ generated in Line 10 which contains all the t-paths of $\mathcal{C}_1$, in the same order as they appear in $\mathcal{C}$. This implies in fact that cycle $\mathcal{C}_1$ is outputted at some point by Algorithm 2, which yields the contradiction.

The test done after 12 ensure that Algorithm 2 outputs simple cycles and that they are unique.

□

Theorem 2.4.2. *Algorithm 2 runs in time $\mathcal{O}(n^{\lfloor p/2 \rfloor}k^p)$.*

Proof. Computing the degeneracy and the degeneracy ordering of G can be done in $\mathcal{O}(m)$ [14]. Computing the degenerate adjacency lists of G in Line 2 can be done in time $\mathcal{O}(nk \log k)$, by Lemma 1.3.4. Computing all acyclic permutations of $\mathcal{D}$ can be done in total time $\mathcal{O}(2^p)$. Using Lemma 2.3.1, Line 6 can be done in $\mathcal{O}(p)$ for each orientation. Since there are $\mathcal{O}(2^p)$ of them this takes total time $\mathcal{O}(p2^p)$. By Corollary 2.2.5, given a pair $l_j = (a_j, b_j)$, we can compute all t-paths of size $(a_j + b_j)$ in G in time $\mathcal{O}(nk^{a_j+b_j})$. Thus for each list $\mathcal{L} : (l_1 = a_1, b_1), l_2 = (a_2, b_2), ..., l_{r-1} = (a_{r-1}, b_{r-1}), l_r = (a_r, b_r)$, Line 8 can be done in time $\mathcal{O}(n \sum_{(a_j,b_j)\in\mathcal{L}} k^{a_j+b_j}) = \mathcal{O}(nk^p)$ since $\sum_{(a_j,b_j)\in\mathcal{L}} a_j + b_j = p$ as proved in Lemma 2.3.3. By Corollary 2.2.5, each set S_j associated to a pair $l_j = (a_j, b_j)$ is of size $\mathcal{O}(nk^{a_j+b_j})$. If there are r such sets, computing all possible lists $\mathcal{C}$ in Line 10 can be done in time $\mathcal{O}(n^r k^p)$ since $\sum_{(a_j,b_j)\in\mathcal{L}} a_j + b_j = p$ as proved in Lemma 2.3.3. Now by Lemma 2.3.2, $r \leq \lfloor p/2 \rfloor$, thus Line 10 takes at most $\mathcal{O}(n^{\lfloor p/2 \rfloor}k^p)$ time, per orientation. To conclude, in Line 12, checking if a list $\mathcal{C}$ is a simple cycle can be done in time $\mathcal{O}(p)$. Since $\mathcal{C}$ is of size at most $\mathcal{O}(p + \lfloor p/2 \rfloor) = \mathcal{O}(p)$ we can check the uniqueness of the vertices and the condition on the end vertices in time $\mathcal{O}(p)$. Finally searching, sorting a string s of size $\mathcal{O}(p)$ or inserting it in a radix tree can be done in time $\mathcal{O}(p \log p)$, see [113]. In total, Algorithm 2 runs in time $\mathcal{O}(|E| + nk \log k + p2^p + p2^p n^{\lfloor p/2 \rfloor} k^p \log p) = \mathcal{O}(n^{\lfloor p/2 \rfloor} k^p)$, assuming p constant.

□

Theorem 2.4.3. *Algorithm 2 can be modified to run in time $\mathcal{O}(n^{\lfloor p/2 \rfloor} k^{\lceil p/2 \rceil} \log k)$ if the graph is stored in adjacency lists or $\mathcal{O}(n^{\lfloor p/2 \rfloor} k^{\lceil p/2 \rceil})$ time if it is stored in an adjacency matrix.*

Proof. We modify Algorithm 2 in the following two ways. Assume list $\mathcal{L} : l_1{=}(a_1, b_1), l_2{=}(a_2, b_2), ..., l_r{=}(a_r, b_r)$ has been computed in Line 6.

- For each $j \in [1, r]$, transform $l_j = (a_j, b_j)$ into $(a_j, b_j - 1)$. This can be done in $\mathcal{O}(p)$.
- When checking if a list $\mathcal{C}$ is a simple cycle after line 12 we check the uniqueness of all the vertices and check if two consecutive t-paths have adjacent end vertices (before, we had to check if they had common vertices). This can be done in time $\mathcal{O}(p \log k)$ using the degenerate adjacency lists computed in line 2 or $\mathcal{O}(1)$ if we have the adjacency matrix of the graph.

We show that Algorithm 2, modified in this way, has the claimed complexity. When we decrease the values of the pairs in $\mathcal{L}$ as described in the first modification and assuming $|\mathcal{L}| = r$, Line 8 can be computed in total time $\mathcal{O}(nk^{p-r})$ since $\sum_{(a_j,b_j)\in\mathcal{L}} a_j + b_j = p - r$. All possible lists in Line 10 can be computed in time $\mathcal{O}(n^r k^{p-r})$ and there are $\mathcal{O}(n^r k^{p-r})$ of them. Since $r \leq \lfloor p/2 \rfloor$ as proved in Lemma 2.3.2 and since $k \leq n$ then $\mathcal{O}(n^r k^{p-r}) = \mathcal{O}(n^{\lfloor p/2 \rfloor} k^{\lceil p/2 \rceil})$. Thus the total time complexity is $\mathcal{O}(n^{\lfloor p/2 \rfloor} k^{\lceil p/2 \rceil} p2^p \log p \log k){=}\mathcal{O}(n^{\lfloor p/2 \rfloor} k^{\lceil p/2 \rceil} \log k)$ with these modifications

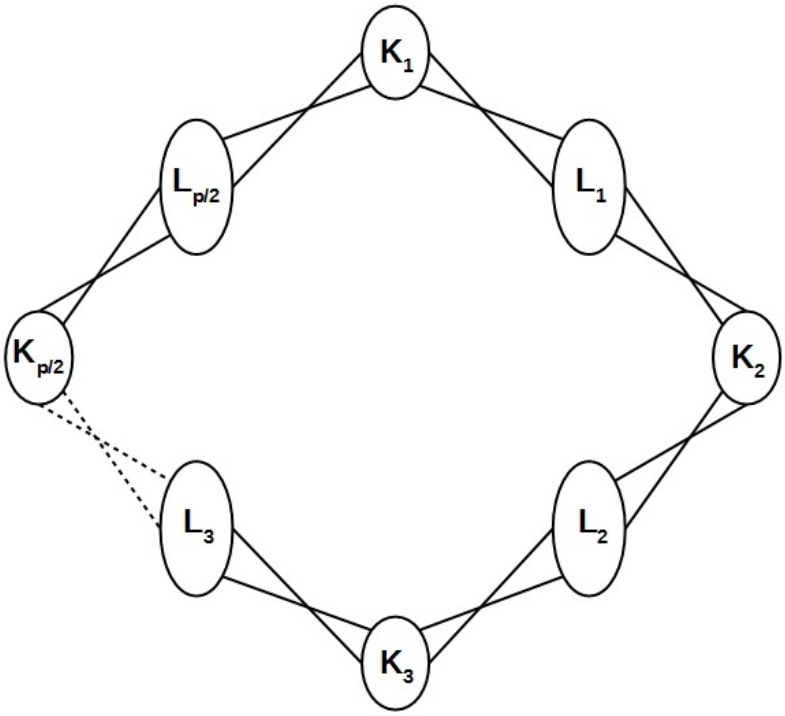


Figure 2.3 – A k-degenerate graph with with the maximal number of p-length simple cycles.

if adjacency queries can be done in $\mathcal{O}(\log k)$ or $\mathcal{O}(n^{\lfloor p/2 \rfloor} k^{\lceil p/2 \rceil})$ if they can be done in $\mathcal{O}(1)$, which completes the complexity analysis.

We prove now the correctness of Algorithm 2 when modified in this way. As shown in Lemma 2.3.1, a simple cycle $\mathcal{C}_1$, given an orientation of its vertices, can be decomposed into strict adjacent t-paths. Consider now all pairs of such adjacent t-paths. If we remove the common vertex from one t-path from each pair, cycle $\mathcal{C}_1$ can still be decomposed into these new t-paths. The previous adjacent t-paths which had a common end vertex now become new smaller t-paths that have adjacent end vertices. The two modifications described above reflect this property: by changing the value of the pairs in the list $\mathcal{L}$ we generate t-paths with one less vertex but we have now to check that, if they appear consecutively in list $\mathcal{C}$ at Line 10, they have adjacent end vertices.

□

Corollary 2.4.4. *With same complexities, we can output exactly all p-length induced cycles.*

Proof. Consider Algorithm 2 modified as in Theorem 2.4.3. Before outputting a cycle we can check if it is induced in time $\mathcal{O}(p^2 \log k)$ with the sorted degenerate adjacency lists: for every pair of vertices of the cycle check if they are adjacent. If we have the adjacency matrix this can be done in time $\mathcal{O}(p^2)$.

□

We prove in the rest of the section that the algorithm is optimal. That is, we construct, for any given degeneracy k and cycle length p, k-degenerate graphs which have the maximum possible p-length simple cycles and show that this number matches the algorithm's complexity.

Theorem 2.4.5. *The maximal number of p-length simple cycles in a k-degenerate graph is $\Theta(n^{\lfloor p/2 \rfloor} k^{\lceil p/2 \rceil})$.*

Proof. Algorithm 2 modified as described in the proof of Theorem 2.4.3 generates at most $\mathcal{O}(n^{\lfloor p/2 \rfloor} k^{\lceil p/2 \rceil})$ candidate cycles, assuming a constant bound on p. Since this algorithm outputs all cycles of the graph, this yields the upper bound.

We now prove the lower bound. We construct for any k, p and $n \geq kp$ a n-order k-degenerate graph with $\Omega(n^{\lfloor p/2 \rfloor} k^{\lceil p/2 \rceil})$ simple p-length cycles.

Assume first that p and k are even. Consider $p/2$ independent sets $K_1, K_2, ..., K_{p/2}$ of size $k/2$ and $p/2$ independent sets $L_1, L_2, ...L_{p/2}$ of size $l \geq k$. Connect all the vertices of set K_i to all the vertices of set L_i for every i. Connect all the vertices of set L_i to all the vertices of set $K_{i+1 \bmod p}$. (See Figure 2.3). This graph is k-degenerate. To see that, observe that every vertex has degree at least k, thus the graph cannot be $(k-1)$-degenerate. Every edge has one endpoint in a set K_i and the other in a set L_j. Thus orienting every edge towards its vertex which is in some K set yields an acyclic orientation with out-degree bounded by k. This graph has $n = \frac{p}{2}\frac{k}{2} + \frac{p}{2}l$ vertices which implies $l = \frac{2n}{p} - \frac{k}{2}$. The total number of simple p-length cycles is $l^{p/2}\frac{k}{2}^{p/2} = (\frac{2n}{p} - \frac{k}{2})^{p/2}\frac{k}{2}^{p/2}$. Observe that $n \geq kp$ implies that $(\frac{2n}{p} - \frac{k}{2})^{p/2}\frac{k}{2}^{p/2} = \Omega(n^{p/2}k^{p/2})$

Assume now that p is even and k odd. The construction is similar except that set K_i has $\lfloor \frac{k}{2} \rfloor$ vertices if i is odd and $\lceil \frac{k}{2} \rceil$ otherwise. If $p/2$ is even the proof is exactly the same as for the case in which p and k are even. If $p/2$ is odd we only prove that the graph is k-degenerate, the proof for the number of simple p-length cycles being the same. The graph is not $(k-1)$-degenerate since the subgraph induced by the vertices of sets K_1, L_1 and K_2 has no vertex of degree less than k. Orienting the edges as in the case in which p and k are even yields an acyclic orientation with out-degree bounded by k.

Assume now that p is odd. The construction is similar. We consider $\lceil p/2 \rceil$ independent sets $K_1, K_2, ..., K_{\lceil p/2 \rceil}$and $\lfloor p/2 \rfloor$ independent sets $L_1, L_2, ..., L_{\lfloor p/2 \rfloor}$. Connect all the vertices of K_i to all the vertices of L_i for every $i < \lceil p/2 \rceil$. Connect all the vertices of L_i to all the vertices of K_{i+1}. Finally connect all vertices of $K_{\lceil p/2 \rceil}$ to all the vertices of K_1. The proof that this graph is k-degenerate is the same as before. If k is even, take the sets K_i of size $k/2$, otherwise take set K_i of size $\lfloor k/2 \rfloor$ if i odd or of size $\lceil k/2 \rceil$ if i even. Now every vertex has degree at least k so the graph cannot be $(k-1)$-degenerate. Orient every edge between the sets $K_{\lceil p/2 \rceil}$ and K_1 arbitrarily. Every other edge has one vertex in some K set and the other in some L set. Thus orienting every edge from its vertex which is in some L set towards its vertex which is in some K set yields an acyclic orientation with out-degree bounded by k. The proof for the number of cycles is exactly the same as for the previous cases.

□

Corollary 2.4.6. *The maximal number of p-length induced cycles in a k-degenerate graph is $\Theta(n^{\lfloor p/2 \rfloor} k^{\lceil p/2 \rceil})$.*

Proof. The upper bound is a consequence of Theorem 2.4.3 and Corollary 2.4.4. Observe that the cycles constructed in Theorem 2.4.5 are induced, which completes the proof.

□

2.5 Conclusion

Given a n-order k-degenerate graph, we presented an algorithm running in time $\mathcal{O}(n^{\lfloor p/2 \rfloor} k^{\lceil p/2 \rceil})$ enumerating all its p-length simple cycles. We then proved that this algorithm is worst-case output size optimal by constructing for any k, p and

$n \geq kp$ a n-order k-degenerate graph with $\Theta(n^{\lfloor p/2 \rfloor} k^{\lceil p/2 \rceil})$ simple p-length cycles. The complexity of the algorithm is given assuming it is stored in an adjacency matrix data structure. If instead it is stored in an adjacency list data structure, the complexity becomes $\mathcal{O}(n^{\lfloor p/2 \rfloor} k^{\lceil p/2 \rceil} \log k)$. Thus the first question we ask is whether or not we can achieve the optimal complexity when the graph is given through its adjacency lists. A second improvement would be to prove an output sensitive algorithm, similar to the one Kowalik presented for cycles of size less than five, see [99]. That is, we ask whether it is possible to achieve an $\mathcal{O}(n^{\mathcal{O}(1)} k^{\mathcal{O}(1)} + \alpha)$ complexity for this problem where α is the number of p-length simple cycles in the graph, assuming p constant. Kowalik essentially shows that small cycles can be broken into few small special paths with at most 3 or 4 vertices and proves bounds on the number of these paths that can have common vertices. Can we extend his approach using t-paths decompositions?

Chapter 3

Maximal cliques enumeration

We present in this chapter our second contribution concerning enumeration algorithms for k-degenerate graphs. It has been presented at IPEC 2017 conference.

3.1 Introduction

For many applications, it is useful to report not one but all the maximal cliques of the graph. General graphs can contain an exponential number of cliques [112], thus any algorithm enumerating maximal cliques has exponential worst case time complexity. In practice however, graphs with this worst-case behavior are not typical. Most of the encountered graphs are sparse [68]. Thus, handling correctly the sparsity of the graph can be an efficient approach. Indeed, it has long been known that certain sparse graph families, such as planar graphs and graphs with low degeneracy, contain only a linear number of cliques, and that all maximal cliques in these graphs can be listed in linear time [30, 31]. Thus it seems pertinent to design algorithms for maximal clique enumeration that take into account some sparsity measure of the graph. In our case, this measure is degeneracy. We give now some of the existing results.

There have been many contributions for the problem of listing all maximal cliques in general and k-degenerate graphs. We can essentially distinguish between two families of algorithms, which have been introduced in Section 1.4.1.

On one side, worst-case output size algorithms have been proposed. Tomita *et al.* [137] prove an algorihtm enumerating all maximal cliques of a general n-order graph in time $\mathcal{O}(3^{n/3})$. This is worst-case output size optimal in general graphs as for instance the Moon-Moser graphs have $\Theta(3^{n/3})$ cliques [25, 112]. Similarly, for k-degenerate graphs, Eppstein *et al.* [53] prove a $\mathcal{O}((n-k)3^{k/3})$ bound on the maximal number of maximal cliques and then show an algorithm running in time $\mathcal{O}(k(n-k)3^{k/3})$. The two algorithms described above rely on ideas of the Bron-Kerbosch algorithm [23]. These results are summarized in the first three rows of Table 3.1. The basic form of the Bron–Kerbosch algorithm is a recursive backtracking algorithm that searches for all maximal cliques in a given graph G. Given three sets R, P, and X, it finds the maximal cliques that include all of the vertices in R, some of the vertices in P, and none of the vertices in X. In each call to the algorithm, P and X are disjoint sets whose union consists of those vertices that form cliques when added to R. In other words, $P \cup X$ is the set of vertices which are joined to every element of R. When P and X are both empty there are no further elements that can be added to R, so R is a maximal clique and the algorithm outputs R. The recursion is initiated by setting R and X to be the empty set and P to be the vertex set of the

Algorithm	Setup	Enumeration	Space
Bron-Kerbosch [23]	$\mathcal{O}(m)$	unbounded	$\mathcal{O}(n+q\Delta)$
Tomita *et al.* [137]	$\mathcal{O}(m)$	$\mathcal{O}(3^{n/3})$	$\mathcal{O}(n+q\Delta)$
Eppstein *et al.* [53]	$\mathcal{O}(m)$	$\mathcal{O}(k(n-k)3^{k/3})$	$\mathcal{O}(n+k\Delta)$
Johnson *et al.* [90]$^+$	$\mathcal{O}(mn)$	$\alpha\mathcal{O}(mn)$	$\mathcal{O}(\alpha n)$
Tsukiyama *et al.* [140]$^+$	$\mathcal{O}(n^2)$	$\alpha\mathcal{O}((n^2-m)n)$	$\mathcal{O}(n^2)$
Chiba *et al.* [30]$^+$	$\mathcal{O}(m)$	$\alpha\mathcal{O}(mk)$	$\mathcal{O}(m)$
Makino *et al.* [104]$^+$	$\mathcal{O}(mn)$	$\alpha\mathcal{O}(\Delta^4)$	$\mathcal{O}(m)$
Chang *et al.* [27]$^+$	$\mathcal{O}(m)$	$\alpha\mathcal{O}(\Delta h^3)$	$\mathcal{O}(m)$
Makino *et al.* [104]$^+$	$\mathcal{O}(n^2)$	$\alpha\mathcal{O}(n^{2.37})$	$\mathcal{O}(n^2)$
Comin *et al.* [37]$^+$	$\mathcal{O}(n^{5.37})$	$\alpha\mathcal{O}(n^{2.09})$	$\mathcal{O}(n^{4.27})$
Conte *et al.* [38]$^+$	$\mathcal{O}(m\log^{\mathcal{O}(1)}(m+n))$	$\alpha\mathcal{O}(qd(\Delta+qd)\log^{\mathcal{O}(1)}(m+n))$	$\mathcal{O}(q)$
Conte *et al.* [38]$^+$	$\mathcal{O}(m\log^{\mathcal{O}(1)}(m+n))$	$\alpha\mathcal{O}(\min\{mk, qk\Delta\}\log^{\mathcal{O}(1)}(m+n))$	$\mathcal{O}(k)$

Δ = max degree $\quad$ k = degeneracy $\quad$ q = largest clique size

α = number of maximal cliques

h = smallest integer such that $|\{v \in V : |N(v)| \geq h\}|$, where $k \leq h \leq \Delta$.

Table 3.1 – Bounds for maximal clique enumeration where $q-1 \leq k \leq \Delta \leq n-1$. $^+$: these are polynomial time delay algorithms. Their delay is equal to their enumeration time divided by the number of maximal cliques α. The space bounds do not include the space needed to store the graph.

graph. Within each recursive call, the algorithm considers the vertices in P in turn; if there are no such vertices, it either reports R as a maximal clique (if X is empty), or backtracks. For each vertex v chosen from P, it makes a recursive call in which v is added to R and in which P and X are restricted to the neighbor set $N(v)$ of v, which finds and reports all clique extensions of R that contain v. Then, it moves v from P to X to exclude it from consideration in future cliques and continues with the next vertex in P.

Another family is the one of polynomial delay algorithms. Their time complexities can be divided into a preprocessing phase followed by an enumeration phase. During the enumeration phase, maximal cliques of the graph are outputted with polynomial delay: the wait between the output of two maximal cliques is bounded by some polynomial in the parameters of the graph. For example, the algorithm of Johnson *et al.* [90] has time setup $\mathcal{O}(mn)$ and polynomial time delay $\mathcal{O}(mn)$. Thus after the setup phase, this algorithm requires $\alpha\mathcal{O}(mn)$ time, α being the number of maximal cliques, to output all the maximal cliques of the graph. It is *output sensitive* since the enumeration time depends on the number of maximal cliques of the graph. All the algorithms that fall into this category are listed in the last nine rows of Table 3.1. For these specific algorithms the time delay is equal to the enumeration time divided by α, the number of maximal cliques.

Our contribution is, given a k-degenerate graph, a polynomial time algorithm with setup time $\mathcal{O}(n(k^2+s(k+1)))$ and enumeration time $\alpha\mathcal{O}((k+1)f(k+1))$, where $s(k+1)$ (resp. $f(k+1)$) is the preprocessing time (resp. enumeration time) for maximal clique enumeration in a general $(k+1)$-order graph. For example, using the algorithm of Makino *et al.* [104] which has setup time $\mathcal{O}((k+1)^2)$ and enumeration time $\mathcal{O}((k+1)^{2.37})$ for a $(k+1)$-order graph, our algorithm has setup time $\mathcal{O}(n(k^2+(k+1)^2))$ and enumeration time $\alpha\mathcal{O}((k+1)(k+1)^{2.37})$. Since in a k-order graph all the graph parameters (number of edges and vertices, the maximum degree, the clique size, etc.) are bounded by some function of k, our algorithm will always have enumeration time depending only on the degeneracy of the graph,

whatever output sensitive algorithm of Table 3.1 we use. This is the first such algorithm. Observe that, even graphs with constant degeneracy can have maximum degree $\Delta = \Theta(n)$: this holds for the graph which has a vertex connected to an independent set of size $(n-1)$. Its degeneracy is one but it has maximum degree $n-1$.

On the downside, we were not able to prove that our algorithm has polynomial time delay. It also requires that the maximal cliques be stored. Thus, since the maximal cliques of a k-degenerate graph are of size at most $k+1$, our algorithm needs $\mathcal{O}((k+1)\alpha)$ space, besides the space needed to store the graph (in our case, the graph can be stored using adjacency lists). Further improvements are discussed in the conclusion.

The organization of the section is as follows. In Section 3.2 we introduce some notations and definitions. In Section 3.3 we prove basic results. These results are used in Section 3.4 to prove the correctness and time complexity of Algorithm 2, which is the main contribution of the section.

3.2 Definitions

For basic notations and definitions, the reader can refer to Section 1.1. We assume that the mentioned graphs are simple and connected. Given an ordering $v_1, ..., v_n$ of the vertices of a graph G, V_i is the set of vertices following v_i including itself in this ordering, that is, the set $\{v_i, v_{i+1}, ..., v_n\}$. By G_i we denote the induced subgraph $G[N[v_i] \cap V_i]$. Recall that a graph is *k-degenerate* if there is an ordering $v_1, ..., v_n$ of its vertices such that for all i, $1 \leq i \leq n$, $|N(v_i) \cap V_i| \leq k$. Given a graph G, we denote by σ_G its degeneracy ordering and if $x \in V(G)$ then $\sigma_G(x)$ will be the ranking of x in σ_G.

At some point in the algorithm we'll need to store maximal cliques and their suffixes. One way to do that is to transform the cliques into words and store them in some special data structure which is introduced in Section 3.4. To do so we need to give some basic definitions regarding strings and their suffixes. Let Σ be an alphabet, that is, a non-empty finite set of symbols. Let a string s be any finite sequence of symbols from Σ; s will be a substring of a string t if there exists strings u and v such that $t = usv$. If u or v is not empty then s is a proper substring of t. It will be a *suffix* of t if there exists a string u such that $t = us$. If u is not empty, s is called a *proper suffix* of t.

3.3 Basic results

When not specified, we always assume that, given a k-degenerate graph G, we have its degeneracy ordering, denoted by σ_G. When referring to an ordering of the vertices, we always refer to σ_G. The family of subgraphs $G_i, i \in [n]$ described in Section 3.2 will always be constructed following the degeneracy ordering of G. Thus, these graphs have at most $k+1$ vertices, since in a degeneracy ordering $v_1, ..., v_n$ of the vertices of G, the inequality $|N[v_i] \cap V_i| \leq k+1$ holds.

We first prove that the family of induced subgraphs $G_i, i \in [n]$ can be constructed quite efficiently.

Lemma 3.3.1. *Given a k-degenerate graph G, there is an algorithm constructing the induced subgraphs $G_i, i \in [n]$ in time $\mathcal{O}(nk^2)$ and $\mathcal{O}(m)$ space.*

Proof. Compute the degenerate adjacency lists of G. This is done in time $\mathcal{O}(m)$ by Lemma 1.3.4. Observe first that the vertex set of graph $G_i, i \in [n]$ corresponds to the i-th vertex of σ_G plus all the vertices of its degenerate adjacency list. Thus, it only remains to show how to compute the adjacency lists of each of these graphs. We proceed as follows. For every vertex $x \in V(G_i)$, go through its degenerate adjacency list and remove vertices which are not in the vertex set $V(G_i)$. Observe that this can be done in $\mathcal{O}(k)$ by coloring the vertices of $V(G_i)$ blue and removing non blue vertices from the degenerate adjacency list of x. This procedure takes time $\mathcal{O}(k^2)$ for each graph $G_i, i \in [n]$, thus in total, we need time $\mathcal{O}(nk^2 + m) = \mathcal{O}(nk^2)$, as $m = \mathcal{O}(nk)$.

□

In the remainder of the section, we study the relationship between the family of induced subgraphs constructed in Lemma 3.3.1 and the maximal cliques of the graph. We show that the cliques which are maximal in these subgraphs but not in the general graph have a very particular structure. We also show that every maximal clique of the general graph appears exactly once in one of the subgraphs.

Lemma 3.3.2. *Let G be a k-degenerate graph, σ_G its degeneracy ordering, and let K be a maximal clique of an induced subgraph $G_i, i \in [n]$. Clique K is not a maximal clique of G if and only if there exists a maximal clique C of G which is an induced subgraph of a G_j with $j < i$ and such that K is a strict induced subgraph of C.*

Proof. Let σ_G be the degeneracy ordering of G. Assume that K is a maximal clique of an induced graph G_i for $i = 1, ..., n-k$ but is not a maximal clique of G. Observe that $v_i \in V(K)$ since, by definition, v_i is connected to all the vertices of $V(G_i) \backslash v_i$. Since K_i is a clique which is not maximal, then there exists a set A of vertices such that $A \cap V(K) = \emptyset$ and the graph induced on $V(K) \cup A$ is a maximal clique of G. Let v_j be the vertex of A with lower ranking in σ_G. We have that $\sigma_G(v_j) < \sigma_G(v_i)$ since v_j is connected to v_i but does not appear in $V(G_i)$. (It does not appear, otherwise $A \cap V(K) \neq \emptyset$). Let C be the maximal clique induced on $V(K) \cup A$. Clique C is an induced subgraph of G_j with $j < i$. Observe that K does not have v_j in its vertex set. Therefore K is a strict induced subgraph of C.

Conversely, assume that K is a maximal clique of G_i and C a maximal clique of $G_j, j < i$ such that K is an induced subgraph of C. Since K is a strict induced subgraph of a maximal clique of G then K can not be a maximal clique of G.

□

Corollary 3.3.3. *Let G be a k-degenerate graph and let K be a maximal clique of an induced subgraph $G_i, i \in [n]$ such that K is not maximal in G. Let C be a maximal clique of G which is a subgraph of some graph $G_j, j < i$ and such that K is a subgraph of C. Let $W(K)$ and $W(C)$ be the words obtained from the vertices of cliques K and C which have been ordered following σ_G. Then $W(K)$ is a proper suffix of $W(C)$.*

Proof. Observe first that by Lemma 3.3.2, clique C is well defined. Since K is a strict subgraph of C then $V(K) \subset V(C)$. Recall that by definition, graph $G_i = G[N[v_i] \cap V_i]$ where v_i is the i-th vertex of the degeneracy ordering. Observe that since v_i is the vertex of $V(K)$ with smallest ranking in σ_G then v_i appears first in $W(K)$. We also have that $v_i \in V(C)$. Assume now by contradiction that $W(K)$ is not a proper suffix of $W(C)$. This implies that there exists at least a vertex $x \in V(C) \backslash V(K)$ that appears after vertex v_i in $W(C)$. If that was not the case then $W(K)$ would have been a proper suffix of $W(C)$. This implies that vertex x

appears after vertex v_i in σ_G. Observe now that x is connected to all the vertices of K since $x \in V(C)$ and $V(K) \subset V(C)$. Thus $G[V(K) \cup \{x\}]$ is a maximal clique of G_i, which is a contradiction by maximality of K.

□

Lemma 3.3.4. *Let G be a k-degenerate graph. Every clique which is maximal in some subgraph $G_i, i \in [n]$ is not maximal in any subgraph G_j with $j \neq i$.*

Proof. Let K be a maximal clique of some subgraph $G_i, i \in [n]$. Assume by contradiction that there exists a $j \in [n]$ with $j \neq i$ such that K is maximal in G_j. Assume first that $i < j$. Since vertex v_i is connected to all the vertices of graph G_i, necessarily $v_i \in V(K)$ or K is not maximal in G_i. Since we assumed $i < j$ then $v_i \notin V(G_j)$. This implies that K cannot be a subgraph of G_j, which gives a contradiction in that case. Thus assume now that $j < i$. The proof is similar. Vertex v_j which is connected to all the vertices of G_j does not belong to graph G_i. Since K is maximal in G_i and since $v_j \notin V(K)$ then K cannot be maximal in G_j.

□

Lemma 3.3.5. *Let G be a k-degenerate graph, σ_G its degeneracy ordering. Every maximal clique of G is a subgraph of exactly one graph $G_i, i \in [n]$.*

Proof. let K be some maximal clique of G. We first prove that K is a subgraph of at least a subgraph $G_i, i \in [n]$. Let $x \in V(K)$ be the vertex of K which has minimum ranking in σ_G. Observe now that clique K is a subgraph of graph $G_{\sigma_G(x)}$. The fact that clique K is a subgraph of at most one graph $G_i, i \in [n]$ is a consequence of Lemma 3.3.4.

□

We can now bound the number of cliques which are maximal in graphs $G_i, i \in [n]$ but not maximal in the graph itself. This is done using the characterization of their structure in Lemmas 3.3.2, 3.3.4 and 3.3.5.

Lemma 3.3.6. *Let G be a k-degenerate graph. Let G_i, $i \in [n]$ be the family of induced subgraphs as defined in Section 3.2 and constructed in Lemma 3.3.1. Let α denote the number of maximal cliques of G and α_i the number of maximal cliques of graph G_i. We have that $\sum_{j=1}^{n} \alpha_j \leq \alpha(k+1)$.*

Proof. Let max_i denotes the number of maximal cliques of $G_i, i \in [n]$ which are maximal in G and $Nmax_i$ the number of maximal cliques of $G_i, i \in [n]$ which are not maximal in G. We have that $\alpha_i = max_i + Nmax_i$. By Lemma 3.3.5, every maximal clique of G is a subgraph of exactly one graph $G_i, i \in [n]$. This implies that $\sum_{j=1}^{n} max_j = \alpha$. Let X be the set of cliques which are maximal in some graph $G_i, i \in [n]$ but not maximal in G and let $x \in X$. By Corollary 3.3.3, the word obtained from the vertices of x which have been ordered following σ_G is a proper suffix of the word obtained from ordering the vertices, following σ_G, of some maximal clique of G. This implies that X is of size at most $k\alpha$ since a maximum clique of a k-degenerate graph has at most k vertices and since a word with k letters has at most k proper suffixes. To conclude the proof, Lemma 3.3.4 implies that clique x is maximal in a unique graph $G_i, i \in [n]$ which implies that $\sum_{j=1}^{n} Nmax_j \leq k\alpha$. Thus in total $\sum_{j=1}^{n} \alpha_j \leq \alpha + \alpha k = \alpha(k+1)$.

□

We are now ready to describe the algorithm for maximal clique enumeration in the next section. It is essentially based on Corollary 3.3.3 and Lemmas 3.3.5 and 3.3.6.

3.4 Algorithm for maximal clique enumeration

Before we describe the algorithm, we introduce suffix trees. We need a data structure to store the proper suffixes of all maximal cliques. Given a word of size n, we can construct a suffix tree containing all its suffixes in space and time $\mathcal{O}(n)$, see [109, 142, 147]. For a set of words $X = \{x_1, x_2, ..., x_n\}$, it is possible to construct a generalized suffix tree containing all the suffixes of the words in X, in an online fashion, in space and time $\mathcal{O}(\sum_{i=1}^{n} |x_i|)$, see [72, chapter 6] and [142] for instance.

The outline of the algorithm is the following. We start by computing the induced subgraphs $G_i, i \in [n]$. Then we consider each such subgraph, starting from from G_1 up to G_n. We find all its maximal cliques and try to find them in a generalized suffix tree. If there is a match, the clique is rejected, otherwise it is outputted and its proper suffixes are inserted into the generalized suffix tree. The procedure is described in Algorithm 2. Its correctness is proved in Theorem 3.4.1 and its time complexity in Theorem 3.4.2.

Algorithm 2:

Data: A graph G.
Result: All the maximal cliques of G.

```
1  Compute the degeneracy k of G and σ_G.
2  Construct the graphs G_i, i ∈ [n].
3  Initialize T an empty generalized suffix tree.
4  for j = 1 to n do
5  |  Compute all maximal cliques of graph G_j.
6  |  for every maximal clique K of graph G_j do
7  |  |  Order the vertices of K following σ_G
8  |  |  Search for K in T.
9  |  |  if there is a match then
10 |  |  |  Reject it.
11 |  |  end
12 |  |  else
13 |  |  |  Insert the proper suffixes of K in T.
14 |  |  |  Output K.
15 |  |  end
16 |  end
17 end
```

Theorem 3.4.1. *Given a graph G, Algorithm 2 outputs exactly all its maximal cliques, without duplication.*

Proof. By Lemma 3.3.5, every maximal clique of the graph is a subgraph of exactly one graph $G_i, i \in [n]$. Thus every maximal clique K of the graph is considered exactly once in Line 6 of the algorithm. If K is matched in the generalized suffix tree at Line 7 then the vertices of K ordered following σ_G form a proper suffix of some clique of the graph. This contradicts the fact that K is maximal in G. Thus every maximal clique is outputted exactly once. Moreover all the proper suffixes of all the maximal cliques are stored in the generalized tree. By Corollary 3.3.3 the word obtained from a maximal clique in some graph $G_i, i \in [n]$ which is not maximal in G forms a proper suffix of the world obtained from some maximal clique of G. Thus all such cliques will be rejected in Line 9 of Algorithm 2. In conclusion, we proved that only the maximum cliques of G are outputted, without duplication.

□

Theorem 3.4.2. *Given a graph G, Algorithm 2 has setup time $\mathcal{O}(n(k^2 + s(k+1))$ and enumeration time $\alpha\mathcal{O}((k+1)f(k+1))$ where α is the number of maximal cliques of G and $s(k+1)$ (resp. $f(k+1)$) is the preprocessing time (resp. enumeration time) of maximal clique enumeration in a general $(k+1)$-order graph.*

Proof. Computing the degeneracy of G in Line 1 is done in $\mathcal{O}(m)$ time. Constructing the graphs $G_i, i \in [n]$ in Line 2 is done in $\mathcal{O}(nk^2)$, by Lemma 3.3.1. To compute all the maximal cliques of every graph $G_i, i \in [n]$, we can use any algorithm of Table 3.1. The chosen algorithm has preprocessing time $s(|V(G_i)|) = \mathcal{O}(s(k+1))$ and enumeration time $f(|V(G_i)|) = \mathcal{O}(f(k+1))$ for each graph $G_i, i \in [n]$ since these graphs have at most $(k+1)$ vertices. We first preprocess every such graph G_i in total time $\mathcal{O}(n \times s(k+1))$. Thus the preprocessing phase takes time $\mathcal{O}(nk^2 + m + n \times s(k+1)) = \mathcal{O}(n(k^2 + s(k+1)))$. Then we enumerate all the maximal cliques of the graphs $G_i, i \in [n]$ in total time $(\sum_{j=1}^{n} \alpha_j) \times \mathcal{O}(f(k+1))$ where α_j is the number of maximal cliques of graph G_j. By Lemma 3.3.6, $\sum_{j=1}^{n} \alpha_j \leq (k+1)\alpha$. Thus enumerating all the maximal cliques of the graphs $G_i, i \in [n]$ takes total time $\alpha\mathcal{O}(f(k+1)(k+1))$. Searching and inserting the generated cliques in the suffix tree takes total time $\alpha\mathcal{O}((k+1)^2)$. In conclusion, Algorithm 2 has preprocessing time $\mathcal{O}(n(k^2 + s(k+1)))$ and enumeration time $\alpha\mathcal{O}((k+1)f(k+1))$, as claimed.

□

3.5 Conclusion

We presented the first output sensitive algorithm for maximal clique enumeration whose enumeration time depends only on the degeneracy of the graph. We were not able to prove that it has polynomial time delay. Our intuition is that in its current state, our algorithm has time delay $\mathcal{O}(k\alpha)$. Thus we first ask whether this is true or not and if yes, if there is a way to modify our approach to get a polynomial time delay. The second question that we ask is whether or not we can improve the space complexity. In its current state, our algorithm requires that the maximal cliques be stored. Can we modify our approach as to avoid that ?

Part II

Self-stabilizing algorithms

Chapter 4

A 2/3-approximation for maximum matching

In this chapter we present the first or our two self-stabilizing algorithms. It approximates, in polynomial number of moves, a maximum matching in an input graph, with ratio 2/3. It has been presented at the OPODIS 2016 conference [33].

4.1 Introduction

Matching problems have received a lot of attention in different areas. Dynamic load balancing and job scheduling in parallel and distributed networks can be solved by algorithms using a matching set of communication links [17, 64]. Moreover, the matching problem has been recently studied in the algorithmic game theory. Indeed, the seminal problem relative to matching introduced by Knuth is the stable marriage problem [96]. This problem can be modeled as a game used in social networks [78] and in wireless networks [138].

In this section, we present a self-stabilizing algorithm for finding a 1-maximal matching that uses a greedy strategy. Our algorithm stabilizes after $O(m \times n^2)$ moves under the adversarial distributed daemon.

For the maximum matching problem, self-stabilizing algorithms have been designed for particular topologies. In anonymous tree networks, a self-stabilizing algorithm converging in $O(n^4)$ moves under the sequential adversarial daemon is given by Karaata and Saleh [92]. Recently, Datta *et al.* [42] improve this result, and give a silent self-stabilizing protocol that converges in $O(n^2)$ moves. For anonymous bipartite networks, a self-stabilizing algorithms converging in $O(n^2)$ rounds under the sequential daemon is given by Chattopadhyay et al. [29]. So, this algorithm converges in $\Omega(n^3)$ moves (since one round corresponds to at least n moves).

In unweighted or weighted general graphs, self-stabilizing algorithms for computing maximal matchings have been designed in various models (anonymous network [8] or not [141], see [71] for a survey). For an unweighted graph, Hsu and Huang [81] gave the first self-stabilizing algorithm and proved a bound of $O(n^3)$ on the number of moves under a sequential adversarial daemon. Hedetniemi *et al.* [75] completed the complexity analysis proving a $O(m)$ move complexity. Manne *et al.* [106] gave a self-stabilizing algorithm that converges in $O(m)$ moves under a distributed adversarial daemon. Cohen *et al.* [34] extend this result and propose a randomized self-stabilizing algorithm for computing a maximal matching in an anonymous network. The complexity is $O(n^2)$ moves with high probability, under the adversarial distributed daemon.

Manne *et al.* [107] and Asada and Inoue [8] presented some self-stabilizing algorithms for finding a 1-maximal matching. Manne *et al.* gave an exponential upper bound on the stabilization time of their algorithm ($O(2^n)$ moves under a distributed adversarial daemon). However, they didn't show that this upper bound is tight. Asada and Inoue [8] gave a polynomial algorithm but under the adversarial sequential daemon. Recently, Inoue *et al.* [84] gave a modified version of [8] that stabilizes after $O(m)$ moves under the distributed adversarial daemon for networks without cycle whose length is a multiple of three.

In a weighted graph, Manne and Mjelde [105] presented the first self-stabilizing algorithm for computing a weighted matching of a graph with a $\frac{1}{2}$-approximation of the optimal solution. They established that their algorithm stabilizes after at most an exponential number of moves under any adversarial daemon (*i.e.*, sequential or distributed). Turau and Hauck [141] gave a modified version of the previous algorithm that stabilizes after $O(nm)$ moves under any adversarial daemon.

Figure 4.1 compares features of the aforementioned algorithms and our result.

We are then interested in the following problem: how to efficiently build a 1-maximal matching in an identified graph with a general topology, using an adversarial distributed daemon and in a self-stabilizing way? In this section, we present two algorithms solving this problem. The first one is the well-known algorithm from Manne *et al.* [107] that was the only one until now that solved this problem. The second algorithm is our contribution. It is known that the Manne *et al.* algorithm reaches a sub-exponential complexity [32]. We prove that our algorithm is polynomial (in $O(m \times n^2)$). This section is an extended version of the conference paper [33], where we present our polynomial algorithm (but with a sketch of the proof only) and the technical report [32]. In the conference paper [33], we obtained a $O(n^3)$ convergence time assuming an already built maximal matching. In this section, under the same assumption, we obtain a $O(n^2)$ convergence time. Thus, as we will develop this scheme in Sections 4.3 and 5.4.3, using a classical composition [47] of the self-stabilizing maximal matching algorithm given by Manne *et al.* [106] and of our algorithm, we obtain a $O(m \times n^2)$ move complexity.

Matching	Topology	Identifiers	Daemon	Complexity (moves)	Work
Maximum	Tree Bipartite	Global Anonymous	Adver. Sequential	$O(n^2)$ $\Omega(n^3)$	[92, 42] [29]
Maximal	Arbitrary	Global	Adver. Sequential Adver. Distributed	$O(m)$ $O(m)$	[81, 75] [106]
		Anonymous	Adver. Sequential Adver. Distributed	$O(n^2)$ $O(n^2)$ whp	[81] [34]
1-Maximal	Arbitrary without cycle with multiple of 3 length	Anonymous	Adver. Sequential Adver. Distributed	$O(m)$ $O(m)$	[8] [84]
	Arbitrary	Global	Adver. Distributed	$\mathbf{\Omega(2^{\sqrt{n}})}$ $\mathbf{O(m.n^2)}$	[107] **Here**

Figure 4.1 – Best results in maximum matching approximation. In bold, our contributions.

In the rest of the chapter, we present the model (Section 4.2), then we give the strategy based on a 3-augmenting path deletion that is used to build a 1-maximal matching (Section 4.3). This strategy is used by both algorithms presented next. In Section 4.4, we precisely describe the Manne *et al.* algorithm [107]. Next, we give our polynomial algorithm in Section 4.5, its correctness proof in Section 4.5.5 followed by its convergence proof in Section 4.5.6.

4.2 Model

For the communication, we consider the *shared memory model*. In this model, as described in Section 1.4.2, each process maintains a set of *local variables* that makes up the *local state* of the process. A process can read its local variables and the local variables of its neighbors, but it can write only in its own local variables.

We consider the *adversarial distributed daemon*, which is the most general one. It can select any non-empty subset of activable nodes for execution. The algorithm presented here, is *silent*, meaning that once the algorithm has stabilized, no process is activable. In other words, all executions of our algorithm are finite and end in a stable configuration. A non silent self-stabilizing algorithm has at least one infinite execution with a suffix only containing legitimate configurations, but not stable ones.

4.3 Common strategy to build a 1-maximal matching

In what follows, we present two algorithms. The first one, denoted by ExpoMatch, is the Manne *et al.* algorithm [107]. The second one, called PolyMatch, is the main contribution of this section. These two algorithms share different elements and this section is devoted to give these main common points.

Both algorithms operate on an undirected graph, where every node has a unique identifier. They also assume an adversarial distributed daemon and that there exists an already built maximal matching, noted $\mathcal{M}$. Based on $\mathcal{M}$, the two algorithms build a 1-maximal matching. To perform that, nodes search and delete any 3-augmenting paths they find in $\mathcal{M}$.

4.3.1 3-augmenting path

We recall the definition of an augmenting path given in Section 1.2. An augmenting path is a path in the graph, starting and ending in an unmatched node, and where every other edge is either unmatched or matched.

Definition 4.3.1. *Let $G = (V, E)$ be a graph and M be a maximal matching of G. (x, u, v, y) is a* 3-augmenting path *on (G, M) if:*

1. *(x, u, v, y) is a path in G (so all nodes are distincts);*
2. *$\{(x, u), (v, y)\} \subset E \setminus M$;*
3. *$(u, v) \in M$*

Once an augmenting path is detected, nodes rearrange the matching accordingly, *i.e.*, transform this path with one matched edge into a path with two matched edges (see Figure 4.2). This transformation leads to the deletion of the augmenting path and increases by one the cardinality of the matching. Both algorithms will stabilize when there are no augmenting paths of length three left. Thus the hypothesis of Karps's theorem [79] eventually holds, giving a $\frac{2}{3}$-approximation of the maximum matching (and so a 1-maximal matching).

4.3.2 The underlying maximal matching

In the rest of the section, $\mathcal{M}$ is the underlying maximal matching. This underlying matching is locally expressed by variables m_v for each node v. If $(u, v) \in \mathcal{M}$

(a) A 3-augmenting path (one matched edge)

(b) The path after being exploited. (two matched edges)

Figure 4.2 – An example of exploitation of an augmenting path. On the left we have the initial matching and on the right the new matching after exploitation of the augmenting path.

then u and v are *matched nodes* and we have: $m_u = v \wedge m_v = u$. If u is not incident to any edge in $\mathcal{M}$, then u is a *single node* and $m_u = null$. For a set of nodes A, we define $single(A)$ and $matched(A)$ as the set of single and matched nodes in A, accordingly to the underlying maximal matching $\mathcal{M}$. Since we assume $\mathcal{M}$ to be stable, a node membership in $matched(V)$ or $single(V)$ will not change throughout an execution, and each node u can use the value of m_u to determine which set it belongs to.

Note that $\mathcal{M}$ can be built with any silent self-stabilizing maximal matching algorithm that works for general graphs and with an adversarial distributed daemon. We can then use, for instance, the self-stabilizing maximal matching algorithm from [106] that stabilizes in $O(m)$ moves. Observe that this algorithm is silent, meaning that the maximal matching remains constant once the algorithm has stabilized.

4.3.3 Augmenting paths detection and exploitation

Both algorithms ExpoMatch and PolyMatch are based on two phases for each edge (u, v) in $\mathcal{M}$: (1) *detecting* augmenting paths and (2) *exploiting* the detected augmenting paths. Node u keeps track of four variables. The pointer p_u is used to define the final matching. The variables α_v, β_u are used to detect augmenting paths and contain single neighbors of u. Also, s_u is a boolean variable used for the augmenting path exploitation. We will see in section 4.5 that algorithm PolyMatch uses a fifth variable named end_u.

In the rest of the section, we will call $\mathcal{M}^+$ the final 1-maximal matching built by any of the two algorithms. $\mathcal{M}^+$ is defined as follows:

Definition 4.3.2. *The built set of edges is:*

$$\mathcal{M}^+ = \{(u, v) \in \mathcal{M} : p_u = p_v = null\} \cup \{(a, b) \in E \setminus \mathcal{M} : p_a = b \wedge p_b = a\}$$

The first set in the union is pairs of nodes that do not perform any rematch. These pairs come from $\mathcal{M}$. The second set in the union is pairs of nodes that were not matched together in $\mathcal{M}$, but after a 3-augmenting path detection and exploitation, they matched together.

Augmenting path detection. First, every pair of matched nodes u, v (v=m_u and u=m_v) tries to find single neighbors they can rematch with. These single neighbors have to be *available*, in particular, they should not be married in a final way with another matched node. We will see in the next sections, that the meaning of being available is not the same in PolyMatch and ExpoMatch. We say that a single node x is a *candidate* for a matched node u if x is an available single neighbor of u. Note that u and v need to have a sufficient number of candidates to detect a 3-augmenting path: each node should have at least one candidate and the sum

of the number of candidates for u and v should be at least 2. In both algorithms, the *BestRematch* predicate is used to compute candidates of a matched node u, writing in α_u and β_u. Then, the condition below is used in both algorithms – in the *AskFirst* predicate – to ensure the number of candidates is sufficiently high to detect if u belongs to a 3-augmenting path.

$$\alpha_u \neq null \wedge \alpha_{m_u} \neq null \wedge 2 \leq Unique(\{\alpha_u, \beta_u, \alpha_{m_u}, \beta_{m_u}\}) \leq 4$$

where $Unique(A)$ returns the number of unique elements in the multi-set A.

Augmenting path exploitation. The exploitation is done in a sequential way. First, two nodes matched together u and v agree on which one starts to build a rematch and which one ends. This local consensus is done using *AskFirst* and *AskSecond* predicates. Observe that these predicates are exactly the same in both algorithms. These predicates use the local state of u and v to assign a role to these two nodes. If *AskFirst(u)* is *True* then u starts to rematch and v ends. If *AskSecond(u)* is *True* then v starts to rematch and u ends.

Observe that there are only three distinct possible values for the quadruplet $(AskFirst(u), AskSecond(u), AskFirst(v), AskSecond(v))$ for any couple $(u, v) \in \mathcal{M}$ and whatever the α and β values are. These are: $(null, null, null, null)$ or $(x, null, null, y)$ or $(null, x, y, null)$, with x and y are two distincts single nodes. The first case means that there is no 3-augmenting path that contains the couple (u, v). The two other cases mean that (x, u, v, y) is a 3-augmenting path. The second case occurs when $x < y$, otherwise we are in the third case. Node u is said to be *First* if $AskFirst(u) \neq null$. In the same way, u is *Second* if $AskSecond(u) \neq null$. So, if a 3-augmenting path is detected though (u, v), the roles of u and v depend on the identifiers of single nodes (candidates) in the augmenting path, *i.e.*, u is *First* iff its single neighbor in the augmenting path has a smaller identifier that the single neighbor of v in the augmenting path.

4.3.4 Graphical convention

We will follow the above conventions in all the figures: matched nodes are represented with thick circles and single nodes with thin circles. Node identifiers are indicated inside the circles. Moreover, all edges that belong to the maximal matching $\mathcal{M}$ are represented with a thick line, whereas the other edges are represented with a simple line. In the same way, all matched nodes are represented with a thick line, whereas single nodes are represented with a simple line. We illustrate the use of the p-values by an arrow, and the absence of the arrow or symbol '⊤' mean that the p-value of the node equals to null. A prohibited value is first drawn in grey, then scratched out in black. For instance, in Figure 4.4, node 10 is single, nodes 9 and 8 are matched and the edge $(8, 9) \in \mathcal{M}$.

4.4 Description of the algorithm ExpoMatch

In this section, we precisely describe the algorithm ExpoMatch [107]. The algorithm itself is shown in Figure 4.3.

4.4.1 Augmenting paths detection and exploitation

Augmenting path detection. In this algorithm, a single node x is a *candidate* for a matched node u if it is not involved in another augmenting path exploitation,

i.e., if $p_x = null \vee p_x = u$.

Augmenting path exploitation. A 3-augmenting path is exploited in two phases. These two phases are performed in a sequential way. Recall that node u is said to be *First* if $AskFirst(u) \neq null$ and node u is *Second* if $AskSecond(u) \neq null$. Let us consider two nodes u and v such that $(u, v) \in \mathcal{M}$. Let us assume that u and v detect an augmenting path.

1. The *First* node starts : Exactly one node among u and v attempts to rematch with one of its candidates. This phase is complete when the first node, let say u, is such that $s_u = True$ and this indicates to the *Second* node (v) that the first phase is over.
2. The *Second* node continues: only when the first node succeeds will the second node attempt to rematch with one of its candidates.
   (a) If this also succeeds, the exploitation is done and the augmenting path is said to be *fully exploited*
   (b) Otherwise the rematch built by the *First* node is deleted and candidates α and β are computed again, allowing then the detection of some new augmenting paths.

4.4.2 Rules description

There are four rules for matched nodes. The *Update* rule is the rule with the highest priority. This rule allows a matched node to update its α and β variables, using the *BestRematch* predicate. Then, predicates *AskFirst* and *AskSecond* are used to define the role the node will have in the 3-augmenting path exploitation. If the node is *First* (resp. *Second*), then it will execute *MatchFirst* (resp. *MatchSecond*) several times for this 3-augmenting path exploitation. The *ResetMatch* rule is performed to reset bad initialization and also to reset an augmenting path exploitation that did not terminate. For instance, this case happens when the single candidate of the *Second* node rematches with some other node in the middle of the exploitation path process.

Let us consider $(u, v) \in \mathcal{M}$ and assume that u and v detect an augmenting path and u is *First*. The *MatchFirst* rule is used by u to build its rematch. The rule is performed a first time by u to propose a rematch to its candidate x (u sets p_u to x). Then, if x accepts ($p_x = u$), u performs this rule a second time to communicate to v that its rematch attempt is a succeed (u sets s_u to $True$). The *MatchSecond* rule is used by the node v to build its rematch. This rule can only be performed if $s_u = True$. Then, the rule is performed once by v to propose a rematch to its candidate y (v sets p_v to y). Then, if y accepts ($p_y = v$), the path is fully exploited and will not change during the rest of the execution.

There is only one rule for single nodes, called *SingleNode*. Recall that all neighbors of a single node are matched, since $\mathcal{M}$ is a maximal matching. A single node should always point to its smallest neighbor that points to it. This rule allows to point to such a neighbor but also to reset a bad p-value to *null*. Observe that a single node x cannot perform this rule if $p_{p_x} = x$, which means that if x point to some neighbor that points back to x, then x is locked.

4.4.3 An execution example of the ExpoMatch algorithm

Now, we give a possible execution of Algorithm ExpoMatch under a distributed adversarial daemon. Figure 4.4.(a) shows the initial configuration of the execution.

——————— **Rules for each node u in single($\mathbf{V}$)**

SingleNode ::
if $(p_u = null \wedge Lowest\{v \in N(u) \mid p_v = u\} \neq null) \vee p_u \notin matched(N(u)) \cup \{null\} \vee$
$(p_u \neq null \wedge p_{p_u} \neq u)$
then $p_u := Lowest\{v \in N(u) \mid p_v = u\}$

——————— **Rules for each node u in matched($\mathbf{V}$)**

Update ::
if $(\alpha_u > \beta_u) \vee (\alpha_u, \beta_u \notin single(N(u)) \cup \{null\}) \vee (\alpha_u = \beta_u \wedge \alpha_u \neq null) \vee p_u \notin single(N(u)) \cup \{null\} \vee$
$((\alpha_u, \beta_u) \neq \text{BestRematch}(u) \wedge (p_u = null \vee p_{p_u} \notin \{u, null\}))$
then $(\alpha_u, \beta_u) := \text{BestRematch}(u)$
$(p_u, s_u) := (null, false)$

MatchFirst ::
Let $x = \text{AskFirst}(u)$
if $x \neq null \wedge (p_u \neq x \vee s_u \neq (p_{p_u} = u))$
then $p_u := x$
$s_v := (p_{p_u} = u)$

MatchSecond ::
Let $y = \text{AskSecond}(u)$
if $y \neq null \wedge s_{m_u} = true \wedge p_u \neq y$
then $p_u := y$

ResetMatch ::
if $\text{AskFirst}(u) = \text{AskSecond}(u) = null \wedge (p_u, s_u) \neq (null, false)$
then $(p_u, s_u) := (null, false)$

——————— **Predicates and functions**

BestRematch$(u) \equiv$
$a := Lowest\ \{v \in single(N(u)) \wedge (p_v = null \vee p_v = u)\}$
$b := Lowest\ \{v \in single(N(u)) \setminus \{a\} \wedge (p_v = null \vee p_v = u)\}$
return (a, b)

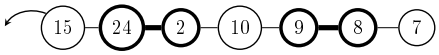
AskFirst$(u) \equiv$
if $\alpha_u \neq null \wedge \alpha_{m_u} \neq null \wedge 2 \leq Unique(\{\alpha_u, \beta_u, \alpha_{m_u}, \beta_{m_u}\}) \leq 4$
then if $\alpha_u < \alpha_{m_u} \vee (\alpha_u = \alpha_{m_u} \wedge \beta_u = null) \vee (\alpha_u = \alpha_{m_u} \wedge \beta_{m_u} \neq null \wedge u < m_u)$
then return α_u
else return $null$

AskSecond$(u) \equiv$
if $AskFirst(m_u) \neq null$
then return $Lowest(\{\alpha_u, \beta_u\} \setminus \{\alpha_{m_u}\})$
else return $null$

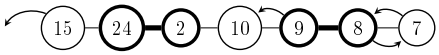
$Unique(A)$ returns the number of unique elements in the multi-set A.
$Lowest(A)$ returns the node in A with the lowest identifier. If $A = \emptyset$, then $Lowest(A)$ returns $null$.

Figure 4.3 – ExpoMatch algorithm

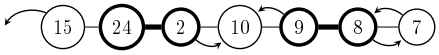
The topology is a path of seven vertices. The underlying maximal matching represented by bold edges contains two edges $(24, 2)$ and $(9, 8)$. Then nodes 24, 2, 9 and 8 are *matched* nodes (in $matched(V)$) and nodes 15, 10 and 7 are *single* nodes (in $single(V)$). There are two 3-augmenting paths: $(15, 24, 2, 10)$ and $(10, 9, 8, 7)$.



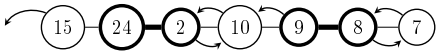
(a) Initial configuration.



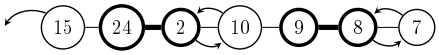
(b) The exploitation of $(10, 9, 8, 7)$ starts.



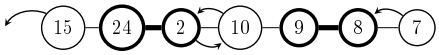
(c) Node 2 executes a *MatchFirst* move.



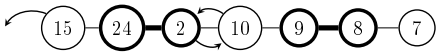
(d) Node 10 executes a *SingleNode* move.



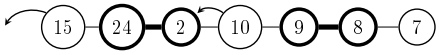
(e) Node 9 executes an *Update* move.



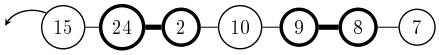
(f) Node 8 executes a *ResetMatch* move.



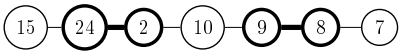
(g) Node 7 executes a *SingleNode* move.



(h) Node 2 executes a *ResetMatch* move.



(i) Node 10 execute a *SingleNode* move.



(j) Node 15 executes a *SingleNode* move.

Figure 4.4 – An execution of Algorithm ExpoMatch

The initial configuration (Figure 4.4.(a)). In the initial configuration, we assume that α-*values* and β-*values* are defined as follows: $(\alpha_8, \beta_8) = (7, null)$, $(\alpha_9, \beta_9) = (10, null)$ and $(\alpha_{24}, \beta_{24}) = (15, null)$ and $(\alpha_2, \beta_2) = (10, null)$. We also assume all s-*values* are well defined: $s_8 = s_9 = s_2 = s_{24} = false$. At this step, node 15 has its p-*values* such that: $p_{15} \notin \{null, 24\}$.

Observe that in the initial configuration, we only have two wrong values: p_{15} and α_{24}. We are going to show that these two faulty nodes can generate the destruction of an augmenting path exploitation, even if this exploited path does not contain any faulty node. This scenario is the fundamental reason why the ExpoMatch algorithm is sub-exponential.

The 3-augmenting path exploitation of $(10, 9, 8, 7)$ starts (Figure 4.4.(a-b)). Nodes 8 and 9 can start to exploit their augmenting path: node 8 is *First* because $\alpha_8 < \alpha_9$, so node 8 executes a *MatchFirst* move and sets $p_8 = 7$. At this point, node 8 waits for an answer of node 7. Node 7 accepts to take part of this path exploitation setting $p_7 = 8$ (by performing a *SingleNode* rule). Afterwards, node 8 can tell node 9 that it can start its exploitation too. Thus node 8 executes again a *MatchFirst* move and sets $s_8 = True$. Now, node 9 can start his exploitation. Assume that it does by executing a *MatchSecond* move. Then node 9 waits for an answer of node 10 and we are in configuration drawn in Figure 4.4.(b).

The 3-augmenting path exploitation of $(15, 24, 2, 10)$ starts (Figure 4.4.(b-d)). Now, we focus on the other 3-augmenting path $(15, 24, 2, 10)$. At this moment, since $2 \leq Unique(\{\alpha_{24}, \beta_{24}, \alpha_2, \beta_2\}) \leq 4$, node 2 detects a 3-augmenting path and starts to exploit it. Since node 2 is *First* ($AskFirst(2) = 10$), node 2 can execute a *MatchFirst* move. Let us assume it does (see Figure 4.4.(c)).

Since both nodes 9 and 2 are pointing to node 10, node 10 can choose the node to match with from these two nodes. Node 10 makes this choice by executing a

SingleNode move: since $Lowest\{u \in N(10) \mid p_u = 10\} = 2$, node 10 chooses node 2 (see Figure 4.4.(d)).

The 3-augmenting path $(10, 9, 8, 7)$ exploitation is destroyed (Figures 4.4.(d-g)). Node 9 considers that node 10 belongs to another 3-augmenting path because $p_{10} \notin \{null, 9\}$. Moreover, since $(\alpha_9, \beta_9) \neq BestRematch(9)$, node 9 can execute an *Update* move. Let us assume it does. Figure 4.4.(e) shows the configuration obtained after this move: $(\alpha_9, \beta_9) = (null, null)$ and $(p_9, s_9) = (null, false)$. This will cause $AskFirst(8) = AskSecond(8) = null$. Then node 8 executes a *ResetMatch* move (see configuration after this move in Figure 4.4.(f)). This will cause node 7 to execute a *SingleNode* move and sets $p_7 = null$ as seen in Figure 4.4.(g).

Focus on the 3-augmenting path $(15, 24, 2, 10)$ (Figures 4.4.(g-j)). Let us assume now that the faulty node 24 is activated. It executes an *Update* move (because $(\alpha_{24}, \beta_{24}) \neq BestRematch(24)$) and sets $(\alpha_{24}, \beta_{24}) = (null, null)$. After this move, node 2 detects that it does not belong to any 3-augmenting path since $AskFirst(2) = AskSecond(2) = null$. So, node 2 executes a *ResetMatch* move and sets $(p_2, s_2) = (null, false)$ (see Figure 4.4.(h)). Afterward, node 10 executes a *SingleNode* move to set p_{10} to *null* (see Figure 4.4.(i)). Now, only node 15 is activable and it executes a *SingleNode* move in order to set p_{15} to *null* (see Figure 4.4.(j)). At this moment, the two exploitation processes for the two 3-augmenting paths can start again.

4.5 Our algorithm PolyMatch

The algorithm presented in this section is called PolyMatch, and is based on the algorithm presented by Manne *et al.* [107], called ExpoMatch. As in ExpoMatch, PolyMatch assumes there exists an underlying maximal matching, called $\mathcal{M}$.

4.5.1 Variables description

Our algorithm has the same set of local variables as in ExpoMatch plus one additional boolean variable, called *end*. As in ExpoMatch, for a matched node u, the pointer p_u refers to a neighbor of u that u is trying to (re)match with, and pointers α_u and β_u refer to two candidates for a possible rematching with u. And again, s_u is a boolean variable that indicates if u has performed a successful rematching with its candidate. Finally, the new variable end_u is a boolean variable that indicates if *both* u and m_u have performed a successful rematching or not. For a single node x, only one pointer p_x and one boolean variable end_x are needed. p_x has the same purpose as the p-variable of a matched node. The *end*-variable of a single node allows the matched nodes to know whether it is *available* or not. A single node x is *available* for a matched node u if it is possible for x to eventually rematch with u, *i.e.*, $p_x = u \vee end_x = False$ (see *BestRematch* predicate).

4.5.2 Augmenting paths detection and exploitation

Augmenting path detection. In this algorithm, a single node x is a candidate for a matched node u if it is not involved in another augmenting path that is fully exploited, *i.e.*, if $end_x = False \vee p_x = u$.

Augmenting path exploitation. A 3-augmenting path is exploited in three phases. These phases are performed in a sequential way. Let us consider two nodes

u and v such that $(u, v) \in \mathcal{M}$. Let us assume that u and v detect a 3-augmenting path.

1. The *First* node starts (same as in ExpoMatch): The *First* node, let say u, tries to rematch with its candidate. This phase is complete when $s_u = True$ and this indicate to the *Second* node (v) that the first phase is over.
2. The *Second* node continues: only when the first node succeeds will the second node attempt to rematch with one of its candidates. This phase is complete when $end_v = True$ and this indicates to v's neighbors that the second phase is over.
3. All nodes in the path set their *end* variable to *True*: the *end* value of v is propagated in the path. The goal of this phase is to write *True* in the *end* variables of the two single nodes in the path in order to make them unavailable for other married nodes. Indeed, the *end* variable is used to compute the candidates of a matched node.

The scenario for an augmenting path exploitation when everything goes well is given in the following. Node u starts trying to rematch with x performing a *MatchFirst* move and $p_u := x$. If x accepts the proposition, performing an *UpdateP* move and $p_x := u$, then u will inform v of this first phase success, once again by performing a *MatchFirst* move and $s_u :=$*True*. Observe that at this point, x cannot change its p-value since $p_{p_x} = x$. Finally, node v tries to rematch with y, performing a *MatchSecond* move and $p_v := y$. If y accepts the proposition, performing an *UpdateP* move and $p_y := v$, then v will inform u of this final success, by performing a *MatchSecond* move again and $end_v :=$*True*. This completes the second phase. From then, all nodes in this 3-augmenting path will set their *end*-variable to *True*: u by performing a last *MatchFirst* move, and x and y by performing an *UpdateEnd* move. From this point, none of these nodes x, u, v, or y will ever be eligible for any move again. Moreover, once single nodes have their *end*-variables set to *True*, they are not available anymore for any other matched nodes.

4.5.3 Rules description

There are four rules for matched nodes. As in ExpoMatch, the *Update* rule allows a matched node to update its α and β variables, using the *BestRematch* predicate. Then, predicates *AskFirst* and *AskSecond* are used to define the role the node will have in the 3-augmenting path exploitation. If the node is *First* (resp. *Second*), then it will execute *MatchFirst* (resp. *MatchSecond*) for this 3-augmenting path exploitation. The *ResetMatch* rule is performed to reset bad initialization and also to reset an augmenting path exploitation that did not terminate.

The *MatchFirst* rule is used by the node when it is *First*. Let u be this node. The rule is performed three times in a usual path exploitation:

1. The first time this rule is performed, u seduces its candidate setting (end_u, s_u, p_u) to $(False, False, AskFirst(u))$.
2. Then this rule is performed a second time after u's candidate has accepted u's proposition, *i.e.*, when $AskFirst(u)$ has set its p-variable to u. So the second *MatchFirst* execution sets (end_u, s_u, p_u) to $(False, True, AskFirst(u))$. Now, variable s_u is equal to $True$, allowing node m_u that is *Second* to seduce its own candidate.
3. Finally, the *MatchFirst* rule is performed a third time when m_u completed is own rematch, *i.e.*, when $end_{m_u} = True$. Observe that when there is no bad information due to some bad initializations, then $end_{m_u} = True$ means $p_{m_u} =$

$AskSecond(m_u) \wedge p_{p_{m_u}} = m_u$ (see the third line of the *MatchSecond* rule). So this third *MatchFirst* execution sets (end_u, s_u, p_u) to $(True, True, AskFirst(u))$, meaning that the 3-augmenting path has been fully exploited.

In the *MatchFirst* rule, observe that we make the s_u affectation before the p_u affectation, because the s_u value must be computed accordingly to the value of p_u before activating u. Indeed, when u executes *MatchFirst* for the first time, it allows to set p_u from $\perp$ to $AskFirst(u)$ while s_u remains $False$. Then when u executes *MatchFirst* for the second time, s_u is set from $False$ to $True$ while p_u remains equal to $AskFirst(u)$. For the same argument, we make the end_u affectation before the s_u affectation. Thus, the "normal" values sequence for (p_u, s_u, end_u) is: $((\perp, False, False), (AskFirst(u),\ False, False), (AskFirst(u),\ True,\ False), (AskFirst(u), True, True))$.

The *MatchSecond* rule is used by the node when it is *Second*. This rule is performed only twice in a usual path exploitation. For the first execution, u has to wait for m_u to set its s_{m_u} to *True*. Then u can perform *MatchSecond* and update its p-variable to $AskSecond(u)$. When u's candidate has accepted his proposition, u can perform *MatchSecond* for the second time, setting s_u and end_u to *True*. As in the *MatchFirst* rule, we set the *end* and s affectations before the p affectation.

There are three rules for single nodes. The *ResetEnd* rule is used to reset bad initializations. In the *UpdateP* rule, the node updates its p-value according to the propositions done by neighboring matched nodes. If there is no proposition, the node sets its p-value to *null*. Otherwise, p is set to the minimum identifier among all proposals. Afterward, the p-value can only change when the proposition is canceled. When a single node u has accepted a proposition, its end value should be equal to the end value of p_u. The *UpdateEnd* rule is used for this purpose.

4.5.4 Execution examples

We give two different executions of algorithm PolyMatch under the adversarial distributed daemon. The first execution points out the main differences between our algorithm PolyMatch and algorithm ExpoMatch. In the second execution, we focus on the *end* variable role for the exploited path process.

Main difference between PolyMatch and ExpoMatch algorithms. We will consider the same example as in Section 4.4.3: we assume that we are in the configuration drawn in Figure 4.4.(d). We assume that all *end-values* equal to $False$. We also assume that all α*-values* and β*-values* are defined as follows: $(\alpha_8, \beta_8) = (7, null)$, $(\alpha_9, \beta_9) = (10, null)$ and $(\alpha_{24}, \beta_{24}) = (15, null)$ and $(\alpha_2, \beta_2) = (10, null)$. At this moment, we assume that the two 3-augmenting paths are partially exploited: $p_2 = p_9 = 10$, $p_{10} = 2$, $p_{15} \neq 24$, $p_{15} \neq null$, $p_8 = 7$, and $p_7 = 8$.

Let us focus on node 24. Node 24 considers that it does not belong to a 3-augmenting path because $end_{15} = True$ means that node 15 is not rematched. Thus, it is eligible to execute a *Update* move.

In our algorithm, even after node 10 has chosen node 2, node 9 still waits for an acceptation of node 10, and will do so while end_{10} remains *False* except for node 15. However, at this point, in Manne *et al.* algorithm, node 9 can destroy the augmenting path construction. This is the main difference that allows our algorithm to prevent from exponential executions.

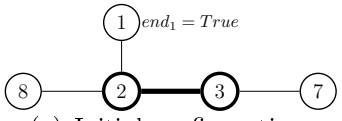
So, at this point there is a binary choice for node 9: destroy or not its augmenting path construction. In the Manne *et al.* algorithm, the choice is to destroy, thus the destruction of a partially exploited augmenting path can be done while no fully exploited augmenting path has been built. In our algorithm, we do the other choice

which is: do not destroy while there is still hope to exploit the augmenting path. So, if node 9 breaks a partially exploited augmenting path, then node 10 belongs to a fully exploited augmenting path. Thus, this destruction implies one 3-augmenting path has been fully exploited, and thus the matching size has been increased by 1.

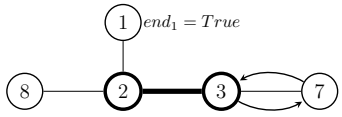
This difference is implemented in the algorithm through the *BestRematch* predicate. The condition $p_x = null$ in Manne *et al.* algorithm has been replaced by the condition end_x =*False* in our algorithm. Then, in our algorithm, $BestRematch(9)$ remains constant when node 10 chooses node 2, while it does not in Manne *et al.* algorithm, making node 9 eligible for *Update*.

How to handle the *end*-variable.

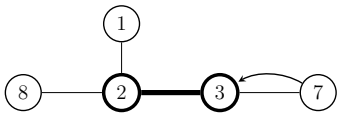
Second, we consider the first execution in order to illustrate the rule of local *end*-variables. Figure 4.5(a) shows the initial state of the execution. The underlying maximal matching contains one edge, $(2, 3)$. Then nodes 2, 3 are *matched* nodes, and nodes 1, 7, and 8 are *single* nodes. At the beginning, there are two 3-augmenting paths: $(1, 2, 3, 7)$ and $(8, 2, 3, 7)$.



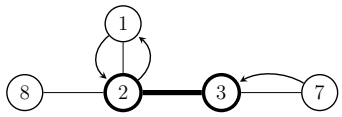
(a) Initial configuration.



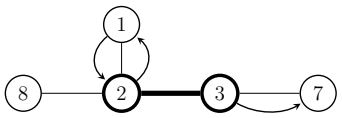
(b) 3 executes *MatchFirst*, then 7 executes *UpdateP* and chooses 3.



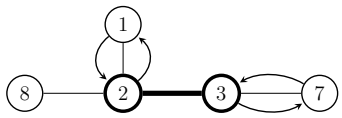
(c) 1 executes *ResetEnd*, then 2 executes *Update* and becomes *First*. Finally, 3 executes *ResetMatch*.



(d) 2 executes *MatchFirst*, then 1 executes *UpdateP* and accepts the proposition of 2. Finally, 2 executes *MatchFirst* (s_2:=*True*).



(e) In parallel 7 and 3 execute *UpdateP* and *MatchSecond* respectively.



(f) 7 executes *UpdateP*, then 3 executes *MatchSecond*, then the *True* value of end_3 is propagated in the path $(1, 2, 3, 7)$.

Figure 4.5 – An execution of Algorithm PolyMatch (Only the *True* value of the *end*-variables are given)

The initial configuration (Figure 4.5(a)):

In the initial configuration, we assume that all α-*values* and β-*values* are defined as follows: $(\alpha_2, \beta_2) = (8, null)$, and $(\alpha_3, \beta_3) = (7, null)$. We also assume all *s-values* are well defined (all other *s*-values are *False*) whereas all *end*-values are *False* but end_1 is *True*. At this moment, node 2 considers that since end_1 =*True*, node 1 already belongs to a fully exploited 3-augmenting path: $BestRematch(2) = (8, null)$.

The 3-augmenting path is $(7, 3, 2, 8)$. Node 2 considers that node 1 is not available because end_1 =*True*. Since $2 \leq Unique(\{\alpha_2, \beta_2, \alpha_3, \beta_3\}) \leq 4$, nodes 2 and 3 detect a 3-augmenting path and start to exploit it. Since node 3 is *First* ($AskFirst(3) = 7$ and $AskFirst(2) = null$), node 3 may execute a *MatchFirst* move. Let us assume it does.

The 3-augmenting path exploitation starts (Figure 4.5(b)):

Node 3 executes here a *MatchFirst* move and points to node 7. Since node 3 is pointing to node 7, node 7 is the only activable node among all nodes except node 1. Node 7 points to node 3 by executing a *UpdateP* move.

Let us focus on node 1. Its *end*-value is not well defined since $end_1 = True$ while node 1 does not belong to a fully exploited augmenting path. Thus, node 1 is eligible for the *ResetEnd* rule. Let us assume it makes this move. After this move, we have $end_1 = False$. This implies that $BestRematch(2) = (1, 8)$ and thus $(\alpha_2, \beta_2) = (8, null) \neq BestRematch(2)$. So, only node 2 is activable, and is eligible for the *Update* rule. Thus, after this mode, node 2 is *First*. This implies that node 3 is *Second*, and it is eligible for *ResetMatch* because $AskSecond(3) \neq null \wedge p_3 \neq null \wedge s_2 = False$. Let us assume it does it.

A second 3-augmenting path exploitation starts (Figure 4.5(d)):

Let us consider node 2. It is *First* and it can execute a *MatchFirst* rule. After this activation, it sets $p_2 = 1$ and $s_2 = end_2 = False$. Now, node 1 accepts the node 2 proposition by executing a *UpdateP* move. After this activation, node 1 points to node 2 ($p_1 = 2$). Now, node 2 is eligible for executing a *MatchFirst* rule. It sets $p_2 = 1$ and $s_2 = True$. This implies that node 3 becomes eligible for *MatchSecond*.

In the configuration shown in Figure 4.5(e), node 3 can propose to node 7 with a *MatchSecond*. Note that node 7 is also eligible for *UpdateP* since $p_3 \neq 7$. Let us assume these two nodes do the move in parallel. Figure 4.5(e) shows the configuration obtained after theses moves: $p_3 = 7$, $p_7 = null$. Note that after these activations, we have $s_3 = False$ since, *before* these activations, the p-values of nodes 3 and 7 are not as follow: $p_3 = 7$ and $p_7 = 3$. This kind of transitions, where a matched node proposition is performed in parallel with a single node abandonment, is the reason why we make the s-affectation, then the p-affectation in the *MatchFirst* rule. This trick allows to obtain after a *MatchFirst* rule: $s_u = True$ implies $p_{p_u} = u$. Finally, observe at this step that node 3 still waits for an answer of node 7.

The path $(1, 2, 3, 7)$ becomes fully exploited (Figure 4.5(f)):

Now, node 7 can choose 3 by executing *UpdateP*. Assume that it does. Since $end_3 \neq (p_3 = 7 \wedge p_3 = AskSecond(3) \wedge p_2 = AskFirst(2))$, node 3 is eligible for a *MatchSecond* rule to set end_3 to *True* and then to make the other nodes aware that the path is fully exploited. Assume node 3 executes a *MatchSecond* move. This will cause node 7 (resp. 2) to execute an *UpdateEnd* move (resp. a *MatchFirst* move) and sets $end_7 = True$ (resp. $end_2 = True$). Now, it is the turn of node 1 to execute an *UpdateEnd* move. As the *end*-value of nodes 1, 2, 3, and 7 are equal to *True*, the 3-augmenting path is fully exploited. The system has reached a stable configuration (see Figure 4.5(f)). Thus, the size of the matching is increasing by one and there is no 3-augmenting path left.

Now, we present the proof of our algorithm.

4.5.5 Correctness Proof

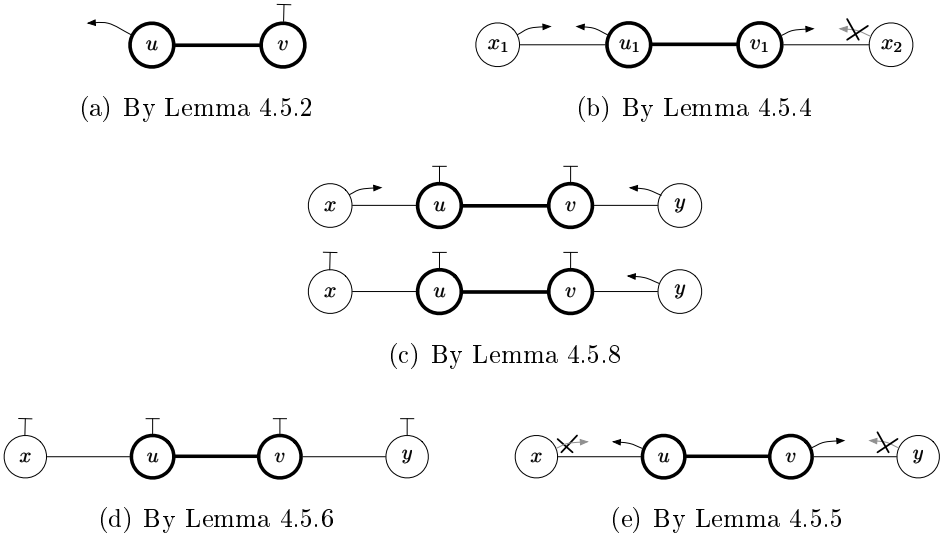
A natural way to prove the correction of PolyMatch algorithm could have been to follow the approach below. We consider a stable configuration C in PolyMatch and we prove that C is also stable in the Manne *et al.* algorithm. As we use the exact

same variables but the *end*-variable and because the matching is only defined on the common variables, the correctness follows from the Manne *et al.* paper. Moreover, we can easily show that if C is stable in PolyMatch, then no rule from the Manne *et al.* algorithm but the *Update* rule can be performed in C. Unfortunately, it is not straightforward to prove that the *Update* rule from the Manne *et al.* algorithm cannot be executed in C. Indeed, our *Update* rule is more difficult to execute than the one of Manne *et al.* in the sense that some possible *Update* in Manne *et al.* are not possible in our algorithm. By the way, this is why our algorithm has a better time complexity since the number of partially exploited augmented path destruction in our algorithm is smaller than in the Manne *et al.* algorithm. In particular, we have to prove that in a stable configuration, for any matched node, if $p_u \neq null$, then $end_{p_u} = True$. To prove that, we need Lemmas 4.5.1, 4.5.2, 4.5.3, 4.5.4 and a part of the proof from Theorem 4.5.7. Observe that from these results, the correctness is straightforward without using the Manne *et al.* proof.

We first introduce some notations. A matched node u is said to be *First* if $AskFirst(u) \neq null$. In the same way, u is *Second* if $AskSecond(u) \neq null$. Let $Ask : V \to V \cup \{null\}$ be a function where $Ask(u) = AskFirst(u)$ if $AskFirst(u) \neq null$, otherwise $Ask(u) = AskSecond(u)$. We will say a node makes a *match* rule if it performs a $MatchFirst$ or $MatchSecond$ rule.

Recall that the set of edges built by our algorithm PolyMatch is $\mathcal{M}^+ = \{(u,v) \in \mathcal{M} : p_u = p_v = null\} \cup \{(a,b) \in E \setminus \mathcal{M} : p_a = b \wedge p_b = a\}$.

For the correctness part of the proof, we prove that in a stable configuration, $\mathcal{M}^+$ is a 2/3-approximation of a maximum matching on graph G. To do that we demonstrate that there is no 3-augmenting path on $(G, \mathcal{M}^+)$. In particular we prove that for any edge $(u,v) \in \mathcal{M}$, we have either $p_u = p_v = null$, or u and v have two distincts single neighbors they are rematched with, *i.e.*, $\exists x \in single(N(u)), \exists y \in single(N(v))$ with $x \neq y$ such that $(p_x = u) \wedge (p_u = x) \wedge (p_y = v) \wedge (p_v = y)$. In order to prove that, we show that every other case for (u,v) is impossible. The main studied cases are shown in Figure 4.6. Finally, we prove that if $p_u = p_v = null$ then (u,v) does not belong to a 3-augmenting-path on $(G, \mathcal{M}^+)$.



(a) By Lemma 4.5.2

(b) By Lemma 4.5.4

(c) By Lemma 4.5.8

(d) By Lemma 4.5.6

(e) By Lemma 4.5.5

Figure 4.6 – Impossible situations in a stable configuration.

Lemma 4.5.1. *In any stable configuration, we have the following properties:*

— $\forall u \in matched(V) : p_u = Ask(u)$;

— $\forall x \in single(V)$: *if* $p_x = u$ *with* $u \neq null$, *then* $u \in matched(N(x)) \wedge p_u = x \wedge end_u = end_x$.

Proof. First, we will prove the first property. We consider the case where $AskFirst(u) \neq null$. We have $p_u = AskFirst(u)$, otherwise node u can execute rule $AskFirst$. We can apply the same result for the case where $AskSecond(u) \neq null$. Finally, we consider the case where $AskFirst(u) = AskSecond(u) = null$. If $p_u \neq null$, then node u can execute rule $ResetMatch$ which yields a contradiction. Thus, $p_u = null$.

Second, we consider a stable configuration C where $p_x = u$, with $u \neq null$. $u \in matched(N(x))$, otherwise x is eligible for an $UpdateP$ rule. Now there are two cases: $p_u = x$ and $p_u \neq x$. If $p_u \neq x$, this means that $p_{p_x} \neq x$. Thus, x is eligible for rule $UpdateP$, and this yields to a contradiction with the fact that C is stable. Finally, we have $end_u = end_x$, otherwise x is eligible for rule $UpdateEnd$. □

Lemma 4.5.2. *Let (u, v) be an edge in $\mathcal{M}$. Let C be a configuration. If $p_u \neq null \wedge p_v = null$ holds in C (see Figure 4.6(a)), then C is not stable.*

Proof. By contradiction. We assume C is stable. From Lemma 4.5.1, we have $p_u = Ask(u) \neq null$ and $p_v = Ask(v)$. So, by definition of predicates $AskFirst$ and $AskSecond$, $Ask(u) = x \neq null$ implies that $Ask(v) \neq null$. This contradicts the fact that $p_v = Ask(v) = null$. □

Lemma 4.5.3. *Let (x, u, v, y) be a 3-augmenting path on $(G, \mathcal{M})$. Let C be a stable configuration. In C, if $p_x = u$, $p_u = x$, $p_v = y$ and $p_y = u$, then $end_x = end_u = end_v = end_y = True$.*

Proof. From Lemma 4.5.1, $p_u = Ask(u)$ (resp. $p_v = Ask(v)$) thus $Ask(u) \neq null$ and $Ask(v) \neq null$. Without loss of generality, we can assume that $AskFirst(u) \neq null$. We have $s_u = True$, otherwise u can execute the $MatchFirst$ rule. Now, as $s_u = True$, we must have $end_v = True$, otherwise v can execute the $MatchSecond$ rule. As $s_u = end_v = True$, we must have $end_u = True$, otherwise u can execute the $MatchFirst$ rule. From Lemma 4.5.1, we can deduce that $end_x = end_u = end_v = end_y = True$ and this concludes the proof. □

Lemma 4.5.4. *Let (x_1, u_1, v_1, x_2) be a 3-augmenting path on $(G, \mathcal{M})$. Let C be a configuration. If $p_{x_1} = u_1 \wedge p_{u_1} = x_1 \wedge p_{v_1} = x_2 \wedge p_{x_2} \neq v_1$ holds in C (see Figure 4.6(b)), then C is not stable.*

Proof. By contradiction. We assume C is stable. From Lemma 4.5.1, $Ask(u_1) = x_1$ and $Ask(v_1) = x_2$.

First we assume that $AskSecond(u_1) = x_1$ and $AskFirst(v_1) = x_2$. The local variable s_{v_1} is $False$, otherwise v_1 would be eligible for executing the $MatchFirst$ rule. Since $AskSecond(u_1) \neq null \wedge p_{u_1} \neq null \wedge s_{v_1} = False$, this implies that u_1 is eligible for the $ResetMatch$ rule which is a contradiction.

Second, we assume that $AskFirst(u_1) = x_1$ and $AskSecond(v_1) = x_2$. We have $s_{u_1} = True$, otherwise u_1 can execute the $MatchFirst$ rule. This implies that $end_{v_1} = False$, otherwise v_1 can execute the $MatchSecond$ rule. As $end_{v_1} = False$, then $end_{u_1} = False$, otherwise u_1 can execute the $MatchFirst$ rule. From Lemma 4.5.1, $end_{x_1} = end_{u_1} = end_{v_1} = False$. Since $Ask(v_1) = x_2$, we have $x_2 \in \{\alpha_{v_1}, \beta_{v_1}\}$. Let us assume $end_{x_2} = True$. Then $x_2 \notin BestRematch(v_1)$ and then v_1 is eligible for an $Update$. Thus $end_{x_2} = False$.

Therefore, C is a configuration such that u_1 is $First$ and v_1 is $Second$ with $end_{x_1} = end_{u_1} = end_{v_1} = end_{x_2} = False$. Now we are going to show that there exists

another augmenting path (x_2, u_2, v_2, x_3) with $end_{x_2} = end_{u_2} = end_{v_2} = end_{x_3} = False$ and $p_{u_2} = x_2$, $p_{x_2} = u_2$, $p_{v_2} = x_3$ and $p_{x_3} \neq v_2$ such that u_2 is $First$ and v_2 is $Second$ (see Figure 4.7).

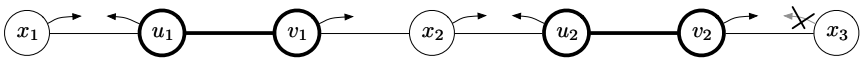


Figure 4.7 – A chain of 3-augmenting paths.

$p_{x_2} \neq null$ otherwise x_2 is eligible for an *UpdateP* rule. Thus there exists a vertex $u_2 \neq v_1$ such that $p_{x_2} = u_2$. From Lemma 4.5.1, $u_2 \in matched(N(x_2))$ and $p_{u_2} = x_2$. Therefore, there exists a node $v_2 = m_{u_2}$. From Lemma 4.5.2, we can deduce that $p_{v_2} \neq null$ and there exists a node x_3 such that $p_{v_2} = x_3$. $x_3 \in single(N(v_2))$ otherwise x_2 is eligible for an *Update* rule. Finally, if $p_{x_3} = v_2$, then Lemma 4.5.3 implies that $end_{x_2} = end_{a_2} = end_{b_2} = end_{x_3} = True$. This contradicts the fact $end_{x_2} = False$. So, we have $p_{x_3} \neq v_2$.

We can then conclude that (x_2, u_2, v_2, x_3) is a 3-augmenting path such that $p_{x_2} = u_2 \wedge p_{u_2} = x_2 \wedge p_{v_2} = x_3 \wedge p_{x_3} \neq v_2$. This augmenting path has the exact same properties than the first considered augmenting path (x_1, u_1, v_1, x_2) and in particular u_1 is First.

Now we can continue the construction in the same way. Therefore, for C to be stable, it must exist a chain of 3-augmenting paths $(x_1, u_1, v_1, x_2, u_2, v_2, \ldots, x_i, u_i, v_i, x_{i+1}, \ldots)$ where $\forall i \geq 1 : (x_i, u_i, v_i, x_{i+1})$ is a 3-augmenting path with $p_{x_i} = u_i \wedge p_{u_i} = x_i \wedge p_{v_i} = x_{i+1} \wedge p_{x_{i+1}} = v_{i+1}$ and u_i is First. Thus, $x_1 < x_2 < \ldots < x_i < \ldots$ since u_i will always be First. Since the graph is finite, some x_k must be equal to some x_ℓ with $\ell \neq k$ which contradicts the fact that the identifier's sequence is strictly increasing. □

Lemma 4.5.5. *Let (x, u, v, y) be a 3-augmenting path on $(G, \mathcal{M})$. Let C be a configuration. If $p_u = x \wedge p_x \neq u \wedge p_v = y \wedge p_y \neq v$ holds in C (see Figure 4.6(e)), then C is not stable.*

Proof. By contradiction, assume that C is stable. From Lemma 4.5.1, $Ask(u) = x$. Assume to begin with, that $AskFirst(u) \neq null$. Because $p_{p_u} \neq u$ we have $s_u = False$, otherwise u is eligible for $MatchFirst$. Since $AskSecond(v) \neq null$ and $s_{m_v} = s_u = False$ then v can apply the $ResetMatch$ rule which yields a contradiction. Therefore assume that $AskSecond(u) \neq null$. The situation is symmetric (because now $AskFirst(v) \neq null$) and therefore we get the same contradiction as before. □

Lemma 4.5.6. *Let (x, u, v, y) be a 3-augmenting path on $(G, \mathcal{M})$. Let C be a configuration. If $p_y = p_u = p_v = p_y = null$ holds in C (see Figure 4.6(d)), then C is not stable.*

Proof. By contradiction, assume that C is stable. $end_x = False$ (resp. $end_y = False$), otherwise x (resp. y) is eligible for a $ResetMatch$. $(\alpha_u, \beta_u) = BestRematch(u)$ (resp. $(\alpha_v, \beta_v) = BestRematch(v)$), otherwise u (resp. v) is eligible for an *Update*. Thus, there is at least an available single node for u and v and so $Ask(u) \neq null$ and $Ask(v) \neq null$. Then, this contradicts the fact that $Ask(u) = null$ (see Lemma 4.5.1). □

Theorem 4.5.7. *In a stable configuration we have, $\forall (u, v) \in \mathcal{M}$:*

— $p_u = p_v = null$ *or*

— $\exists x \in single(N(u)), \exists y \in single(N(v))$ *with* $x \neq y$ *such that* $p_x = u \wedge p_u = x \wedge p_y = v \wedge p_v = y$.

Proof. We will prove that all cases but these two are not possible in a stable configuration. First, Lemma 4.5.2 says the configuration cannot be stable if exactly one of p_u or p_v is not *null*.

Second, assume that $p_u \neq null \wedge p_v \neq null$. Let $p_u = x$ and $p_v = y$. Observe that $x \in single(N(u))$ (resp. $y \in single(N(v))$), otherwise u (resp. v) is eligible for *Update*.

Case $x \neq y$: If $p_x \neq u$ and $p_y \neq v$ then Lemma 4.5.5 says the configuration cannot be stable. If $p_x = u$ and $p_y \neq v$ then Lemma 4.5.4 says the configuration cannot be stable. Thus, the only remaining possibility when $p_u \neq null$ and $p_v \neq null$ is: $p_x = p_u$ and $p_y = v$.

Case $x = y$: $Ask(u) \neq null$ (resp. $Ask(v) \neq null$), otherwise u (resp. v) is eligible for a *ResetMatch*. Without loss of generality, let us assume that u is First. $x = AskFirst(u)$ (resp. x=AskSecond(v)), otherwise u (resp. v) is eligible for *MatchFirst* (resp. *MatchSecond*). Thus $AskFirst(u) = AskSecond(v)$ which is impossible according to these two predicates. □

Lemma 4.5.8. *Let x be a single node. In a stable configuration, if $p_x = u, u \neq null$ then there exists a 3-augmenting path (x, u, v, y) on $(G, \mathcal{M})$ such that $p_x = u \wedge p_u = x \wedge p_v = y \wedge p_y = v$.*

Proof. By lemma 4.5.1, if $p_x = u$ with $u \neq null$ then $u \in matched(N(x))$ and $p_u = x$. Since $p_u \neq null$, by Theorem 4.5.7 the result holds. □

Observe that according to this Lemma, cases from Figure 4.6(c) are impossible.

Thus, in a stable configuration, for all edges $(u, v) \in \mathcal{M}$, if $p_u = p_v = null$ then (u, v) does not belong to a 3-augmenting-path on $(G, \mathcal{M}^+)$. In other words, we obtain:

Corollary 4.5.9. *In a stable configuration, there is no 3-augmenting path on $(G, \mathcal{M}^+)$ left.*

4.5.6 Convergence Proof

This section is devoted to the convergence proof. In the following, μ will denote the number of matched nodes and σ the number of single nodes.

The first step consists in proving that the values of s and *end* represent the different phases of the path exploitation. Recall that $s_u = True$ means $p_{p_u} = u$. Moreover $end_u = True$ means that the path is fully exploited. We can easily prove that after one activation of a matched node u, $s_u = True$ implies $p_{p_u} = u$:

Lemma 4.5.11. *Let u be a matched node. Consider an execution $\mathcal{E}$ starting after u executed some rule. Let C be any configuration in $\mathcal{E}$. In C, if s_u =True then $\exists x \in single(N(u)) : p_u = x \wedge p_x = u$.*

However, a bad initialization of end_{m_u} to *True* can induce u to wrongly write *True* in end_u. But this can appear only once and thus, the second times u writes *True* in end_u means that a 3-augmenting path involving u has been fully exploited.

Theorem 4.5.18. *In any execution, a matched node u can write end_u :=True at most twice.*

We now count the number of destruction of partially exploited augmenting paths. Recall that in the Manne *et al.* algorithm, for one fully exploited augmenting path, it is possible to destroy a sub-exponential number of partially exploited ones.

In our algorithm, observe that for a path destruction, the set of single neighbors that are candidates for a matched edge has to change and this change can only occur when a single node changes its *end*-value. Such a change induces a path destruction if a matched node takes into account this modification by applying an *Update* rule. So, we first count the number of times a single node can change its *end*-value (Lemma 4.5.24) and then we deduce the number of times a matched node can execute *Update* (Corollary 4.5.27). Finally, we conclude we destroy at most $O(n^2)(= O(\Delta(\sigma + \mu)))$ partially exploited augmenting path.

The rest of the proof consists in counting the number of moves that can be performed between two *Update*, allowing us to conclude the proof (Theorem 4.5.35).

In the following, we detailed point by point the idea behind each result cited above.

Since single nodes just follow orders from their neighboring matched nodes, we can count the number of times single nodes can change the value of their *end* variable. There are σ possible modifications due to bad initializations. A matched node u can write *True* twice in end_u, so end_u can be *True* during 3 distinct sub-executions. As a single node x copies the *end*-value of the matched node it points to ($p_x = u$), then a single node can change its *end*-value at most 3 times as well. And we obtain 6μ modifications.

Lemma 4.5.24. *In any execution, the number of transitions where a single node changes the value of its end variables (from True to False or from False to True) is at most* $\sigma + 6\mu$ *times.*

We count the maximal number of *Update* rule that can be performed in any execution. To do that, we observe that the first line of the *Update* guard can be *True* at most once in an execution (Lemma 4.5.12). Then we prove for the second line of the guard to be *True*, a single node has to change its *end* value. Thus, for each single node modification of the *end-value*, at most all matched neighbors of this single node can perform an *Update* rule.

Corollary 4.5.27. *Matched nodes can execute at most* $\Delta(\sigma + 6\mu) + \mu$ *times the* Update *rule.*

Third, we consider two particular matched nodes u and v and an execution with no *Update* rule performed by these two nodes. Then we count the maximal number of moves performed by these two nodes in this execution. The idea is that in such an execution, the α and β values of u and v remain constant. Thus, in these small executions, u and v detect at most one augmenting path and perform at most one rematch attempt. We obtain that the maximal number of moves of u and v in these small executions is 12. By the previous remark and Corollary 4.5.27, we obtain:

Theorem 4.5.35. *In any execution, matched nodes can execute at most* $12\Delta(\sigma + 6\mu) + 18\mu$ *rules.*

Finally, we count the maximal number of moves that single nodes can perform, counting rule by rule. The *ResetEnd* is done at most once. The number of *UpdateEnd* is bounded by the number of times single nodes can change their *end*-value, so it is at most $\sigma + 6\mu$. Finally, *UpdateP* is counted as follows: between two consecutive *UpdateP* executed by a single node x, a matched node has to make a move. The total number of executed *UpdateP* is then at most $12\Delta(\sigma + 6\mu) + 18\mu + 1$.

Corollary 4.5.40. *The algorithm* PolyMatch *converges in $O(n^2)$ moves under the adversarial distributed daemon and in a general graph, provided that an underlying maximal matching has been initially built.*

The Manne *et al.* algorithm [106] builds a self-stabilizing maximal matching under the adversarial distributed daemon in a general graph, in $O(m)$ moves. This leads to a $O(m.n^2)$ moves complexity to build a 1-maximal matching with our algorithm without any assumption of an underlying maximal matching.

Now, the next section is devoted to the description of the technical proof.

A matched node can write *True* in its *end*-variable at most twice

The first three lemmas are technical lemmas.

Lemma 4.5.10. *Let u be a matched node. Consider an execution $\mathcal{E}$ starting after u executed some rule. Let C be any configuration in $\mathcal{E}$. If $end_u = True$ in C then $s_u = True$ as well.*

Proof. Let $C_0 \mapsto C_1$ be the transition in $\mathcal{E}$ in which u executed a rule for the last time before C. Observe that C may be equal to C_1. The executed rule is necessarily a *match* rule, otherwise end_u could not be $True$ in C_1. If it is a $MatchSecond$ the lemma holds since in that case s_u is a copy of end_u. Assume now it is a $MatchFirst$. For end_u to be $True$ in C_1, $p_u = AskFirst(u) \wedge p_{p_u} = u \wedge p_{m_u} = AskSecond(m_u)$ must hold in C_0, according to the guard of $MatchFirst$. This implies that u writes $True$ in s_u in transition $C_0 \mapsto C_1$. □

Lemma 4.5.11. *Let u be a matched node. Consider an execution $\mathcal{E}$ starting after u executed some rule. Let C be any configuration in $\mathcal{E}$. In C, if $s_u = True$ then $\exists x \in single(N(u)) : p_u = x \wedge p_x = u$.*

Proof. Consider transition $C_0 \mapsto C_1$ in which u executed a rule for the last time before C. The executed rule is necessarily a *match* rule, otherwise s_u could not be $True$ in C_1. Observe now that whichever *match* rule is applied, $Ask(u) \neq null$ – let us assume $Ask(u) = x$ – and $p_u = x$ and $p_x = u$ must hold in C_0 for s_u to be $True$ in C_1. $p_u = x$ still holds in C_1 and until C. Moreover, x must be in $single(N(u))$, otherwise u would have executed an *Update* instead of a match rule in $C_0 \mapsto C_1$, since *Update* has the highest priority among all rules. Finally, in transition $C_0 \mapsto C_1$, x cannot execute $UpdateP$ nor $ResetEnd$ since $p_x \in matched(N(x)) \wedge p_{p_x} = x$ holds in C_0. Thus in C_1, $p_u = x$ and $p_x = u$ holds. Using the same argument, x cannot execute $UpdateP$ nor $ResetEnd$ between configurations C_1 and C. Thus $p_u = x \wedge p_x = u$ in C. □

Lemma 4.5.12. *Let u be a matched node and $\mathcal{E}$ be an execution containing a transition $C_0 \mapsto C_1$ where u makes a move. From C_1, the predicate in the first line of the guard of the Update rule will ever hold from C_1.*

Proof. Let C_2 be any configuration in $\mathcal{E}$ such that $C_2 \geq C_1$. Let $C_{10} \mapsto C_{11}$ be the last transition before C_2 in which u executes a move. Notice that by definition of $\mathcal{E}$, this transition exists. Assume by contradiction that one of the following predicates holds in C_2.

(1) $(\alpha_u > \beta_u) \vee (\alpha_u, \beta_u \notin (single(N(u)) \cup \{null\})) \vee (\alpha_u = \beta_u \wedge \alpha_u \neq null)$

(2) $p_u \notin (single(N(u)) \cup \{null\})$

By definition between C_{11} and C_2, u does not execute rules. To modify the variables α_u, β_u and p_u, u must execute a rule. Thus one of the two predicates also holds in C_{11}.

We first show that if predicate (1) holds in C_{11} then we get a contradiction. If u executes an *Update* rule in transition $C_{10} \mapsto C_{11}$, then by definition of the *BestRematch* function, predicate (1) cannot hold in C_{11} (observe that the only way for $\alpha_u = \beta_u$ is when $\alpha_u = \beta_u = null$). Thus assume that u executes a *match* or *ResetMatch* rule. Note that these rules do not modify the value of the α_u and β_u variables. This implies that if u executes one of these rules in $C_{10} \mapsto C_{11}$, predicate (1) not only holds in C_{11} but also in C_{10}. Observe that this implies, in that case that u is eligible for *Update* in $C_{10} \mapsto C_{11}$, which gives the contradiction since *Update* is the rule with the highest priority among all rules.

Now assume predicate (2) holds in C_{11}. In transition $C_{10} \mapsto C_{11}$, u cannot execute *Update* nor *ResetMatch* as this would imply that $p_u = null$ in C_{11}. Assume that in $C_{10} \mapsto C_{11}$ u executes a *match* rule. Since in C_{11}, $p_u \notin (single(N(u)) \cup \{null\})$ this implies that in C_{10}, $Ask(u) \notin (single(N(u)) \cup \{null\})$. This implies that $\alpha_u, \beta_u \notin (single(N(u)) \cup \{null\})$ in C_{10}. Thus u is eligible for *Update* in transition $C_{10} \mapsto C_{11}$ and this yields the contradiction since *Update* is the rule with the highest priority among all rules.

Since these two predicates cannot hold in C_2, this concludes the proof. □

Now, we focus on particular configurations for a matched edge (u, v) corresponding to the fact that they have completely exploited a 3-augmenting path.

Lemma 4.5.13. *Let (u, v) be a matched edge, $\mathcal{E}$ be an execution and C be a configuration of $\mathcal{E}$. If, in C, we have:*

1. $p_u \in single(N(u)) \wedge p_u = AskFirst(u) \wedge p_{p_u} = u$;
2. $p_v \in single(N(v)) \wedge p_v = AskSecond(v) \wedge p_{p_v} = v$;
3. $s_u = end_u = s_v = end_v = True$;

then neither u nor v will ever be eligible for any rule from C.

Proof. Observe first that neither u nor v are eligible for any rule in C. Moreover, p_u (resp. p_v) is not eligible for an *UpdateP* move since u (resp. v) does not make any move. Thus p_{p_u} and p_{p_v} will remain constant since u and v do not make any move and so neither u nor v will ever be eligible for any rule from C. □

The configuration C described in Lemma 4.5.13 is called a $stop_{uv}$ configuration. From such a configuration neither u nor v will ever be eligible for any rule.

In Lemmas 4.5.16 and 4.5.17, we consider executions where a matched node u writes $True$ in end_u twice, and we focus on the transition $C_0 \mapsto C_1$ where u performs its second writing. Lemma 4.5.16 shows that, if u is First in C_0, then C_1 is a $stop_{um_u}$ configuration. Lemma 4.5.17 shows that, if u is Second in C_0, then either C_1 is a $stop_{um_u}$ configuration or there exists a configuration C_3 such that $C_3 > C_1$, u does not make any move from C_1 to C_3 and C_3 is a $stop_{um_u}$ configuration.

Lemma 4.5.14 and Corollary 4.5.15 are required to prove Lemmas 4.5.16 and 4.5.17.

Lemma 4.5.14. *Let (u, v) be a matched edge. Let $\mathcal{E}$ be some execution in which v does not execute any rule. If there exists a transition $C_0 \mapsto C_1$ in $\mathcal{E}$ where u writes $True$ in end_u, then u is not eligible for any rule from C_1.*

Proof. To write $True$ in end_u in transition $C_0 \mapsto C_1$, u must have executed a *match* rule. According to this rule, $(p_u = Ask(u) \wedge p_{p_u} = u)$ holds in C_0 with $p_u \in$

$single(N(u))$, otherwise u would have executed an *Update* instead of a *match* rule. Now, since in $C_0 \mapsto C_1$, p_u cannot execute *UpdateP*, then it cannot change its p-value and since v does not execute any move then it cannot change $Ask(u)$. Thus, $(p_u = Ask(u) \wedge p_{p_u} = u)$ holds in both C_0 and C_1.

Assume now by contradiction that u executes a rule after configuration C_1. Let $C_2 \mapsto C_3$ be the next transition in which it executes a rule. Recall that between configurations C_1 and C_2 both u and v do not execute rules. Observe also that p_u is not eligible for *UpdateP* between these configurations. Thus $(p_u = Ask(u) \wedge p_{p_u} = u)$ holds from C_0 to C_2. Moreover the following points hold as well between C_0 and C_2 since in $C_0 \mapsto C_1$, u executed a *match* rule and v does not apply rules in $\mathcal{E}$:

— α_u, α_v, β_u and β_v do not change.

— The values of the variables of v do not change.

— $Ask(u)$ and $Ask(v)$ do not change.

— If u was *First* in C_0 it is *First* in C_2 and the same holds if it was *Second*.

Using these remarks, we start by proving that u is not eligible for *ResetMatch* in C_2. If it is *First* in C_2, this holds since $AskFirst(u) \neq null$ and $AskSecond(u) = null$. If it is *Second* then to be eligible for *ResetMatch*, $s_v = False$ must hold in C_2 since $AskSecond(u) \neq null$. Since u executed $end_u = True$ in $C_0 \mapsto C_1$ and since u was *Second* in C_0, then necessarily $s_v = True$ in C_0 and thus in C_2 (using remark 2 above). So u is not eligible for *ResetMatch* in C_2.

We show now that u is not eligible for an *Update* in C_2. The α and β variables of u and v remain constant between C_0 and C_2. Thus if any of the three first disjunctions in the *Update* rule holds in C_2 then it also holds in C_0 and in $C_0 \mapsto C_1$ u should have executed an *Update* since it has higher priority than the *match* rules. Moreover since in C_2 $(p_u = Ask(u) \wedge p_{p_u} = u)$ holds, the last two disjunctions of *Update* are *False* and we can state u is not eligible for this rule.

We conclude the proof by showing that u is not eligible for a *match* rule in C_2. If u was *First* in C_0 then it is *First* in C_2. To write *True* in end_u then $(p_u = AskFirst(u) \wedge p_{p_u} = u \wedge s_u \wedge p_{m_u} = AskSecond(m_u) \wedge end_{m_u})$ must hold in C_0. Since in $C_0 \mapsto C_1$ v does not execute rules, it also holds in C_1. The same remark between configurations C_1 and C_2 implies that this predicate holds in C_2. Thus in C_2, all the three conditions of the *MatchFirst* guard are *False* and u not eligible for *MatchFirst*. A similar remark if u is *Second* implies that u will not be eligible for *MatchSecond* in C_2 if it was *Second* in C_0. □

Corollary 4.5.15. *Let (u, v) be a matched edge. In any execution, if u writes True in end_u twice, then v executes a rule between these two writings.*

Lemma 4.5.16. *Let (u, v) be a matched edge and $\mathcal{E}$ be an execution where u writes True in its variable end_u at least twice. Let $C_0 \mapsto C_1$ be the transition where u writes True in end_u for the second time in $\mathcal{E}$. If u is First in C_0 then the following holds:*

1. *in configuration C_0,*
   (a) $s_v = end_v = True$;
   (b) $p_u = AskFirst(u) \wedge p_{p_u} = u \wedge s_u = True \wedge p_v = AskSecond(v)$;
   (c) $p_u \in single(N(u))$;
   (d) $p_v \in single(N(v)) \wedge p_{p_v} = v$;
2. *v does not execute any move in $C_0 \mapsto C_1$;*
3. *in configuration C_1,*

(a) $s_u = end_u = True$;

(b) $p_u \in single(N(u)) \wedge p_v \in single(N(v))$;

(c) $s_v = end_v = True$;

(d) $p_u = AskFirst(u) \wedge p_v = AskSecond(v)$;

(e) $p_{p_u} = u \wedge p_{p_v} = v$.

Proof. We prove Point 1a. Observe that for u to write $True$ in end_u, end_v must be $True$ in C_0. By Lemma 4.5.10 this implies that s_v is True as well. Now Point 1b holds by definition of the $MatchFirst$ rule. As in C_0, u already executed an action, then according to Lemma 4.5.12, Point 1c holds and will always hold. By Corollary 4.5.15, u cannot write $True$ consecutively if v does not execute moves. Thus at some point before C_0, v applied some rule. This implies that in configuration C_0, since $s_v = True$, by Lemma 4.5.11, $\exists x \in single(N(v)) : p_v = x \wedge p_x = v$. Thus Point 1d holds.

We now show that v does not execute any move in $C_0 \mapsto C_1$ (Point 2). Recall that v already executed an action before C_0, so by Lemma 4.5.12, line 1 of the $Update$ guard does not hold in C_0. Moreover, by Point 1d, line 2 does not hold either. Thus, v is not eligible for $Update$ in C_0. We also have that $s_u = True$ and $AskSecond(v) \neq null$ in C_0, thus v is not eligible for $ResetMatch$. Observe now that by Points 1a, 1b and 1d, v is not eligible for $MatchSecond$ in C_0. Finally v cannot execute $MatchFirst$ since $AskFirst(v) = null$. Thus v does not execute any move in $C_0 \mapsto C_1$ and so Point 2 holds.

In C_1, end_u is True by hypothesis and according to Point 1b, u writes $True$ in s_u in transition $C_0 \mapsto C_1$. Thus Point 3a holds. Points 3b holds by Points 1c and 1d. Points 3c holds by Points 1a and 2. $AskFirst(u)$ and $AskSecond(v)$ remain constant in $C_0 \mapsto C_1$ since neither u nor v executes an $Update$ in this transition. Moreover p_v remains constant in $C_0 \mapsto C_1$ by Point 2 and p_u remains constant also since it writes $AskFirst(u)$ in p_u in this transition while $p_u = AskFirst(u)$ in C_0. Thus Points 3d holds. Observe that nor p_u neither p_v is eligible for an $UpdateP$ in C_0, thus Point 3e holds. □

Now, we consider the case where u is Second.

Lemma 4.5.17. *Let (u, v) be a matched edge and $\mathcal{E}$ be an execution where u writes $True$ in its variable end_u at least twice. Let $C_0 \mapsto C_1$ be the transition where u writes $True$ in end_u for the second time in $\mathcal{E}$. If u is Second in C_0 then the following holds:*

1. *in configuration C_0,*
   (a) $s_v = True \wedge p_v = AskFirst(v)$;
   (b) $p_v \in single(N(v)) \wedge p_{p_v} = v$;
2. *in transition $C_0 \mapsto C_1$, v is not eligible for Update nor ResetMatch;*
3. *in configuration C_1,*
   (a) $s_u = end_u = True$;
   (b) $p_v \in single(N(v)) \wedge p_v = AskFirst(v) \wedge p_{p_v} = v$;
   (c) $p_u \in single(N(u)) \wedge p_u = AskSecond(u) \wedge p_{p_u} = u$;
   (d) $s_v = True$;
4. *u is not eligible for any move in C_1;*
5. *If $end_u = False$ in C_1 then the following holds:*
   (a) *From C_1, v executes a next move and this move is a MatchFirst;*

(b) *Let us assume this move (the first move of v from C_1) is done in transition $C_2 \mapsto C_3$. In configuration C_3, we have:*

i. $s_u = end_u = True$;

ii. $p_v \in single(N(v)) \wedge p_v = AskFirst(v) \wedge p_{p_v} = v$;

iii. $p_u \in single(N(u)) \wedge p_u = AskSecond(u) \wedge p_{p_u} = u$;

iv. $s_v = True$;

v. *u does not execute moves between C_1 and C_3;*

vi. $end_v = True$;

Proof. We show Point 1a. For u to write $True$ in transition $C_0 \mapsto C_1$, u executes a $MatchSecond$ in this transition. Thus $s_v = True$ must hold in C_0 and $p_v = AskFirst(v)$ as well. By Corollary 4.5.15, u cannot write $True$ consecutively if v does not execute any move. Thus at some point before C_0, v applied some rule. Thus, and by Lemma 4.5.11, $\exists x \in single(N(v)) : p_v = x \wedge p_x = v$ in configuration C_0, so Point 1b holds.

As $AskFirst(v) \neq null$ in C_0, v is not eligible for $ResetMatch$ in C_0. We prove now that v is not eligible for $Update$. By Corollary 4.5.15 and Lemma 4.5.12, line 1 of the $Update$ guard does not hold in C_0. Finally, according to Point 2b, the second line of the $Update$ guard does not hold, which concludes Point 2.

We consider now Point 3a. In C_1, $s_u = end_u = True$ holds because, executing a $MatchSecond$, u writes $True$ in end_u and writes end_u in s_u during transition $C_0 \mapsto C_1$.

We now show Point 3b. $AskFirst(v)$ and $AskSecond(u)$ remain constant in $C_0 \mapsto C_1$ since neither u nor v execute an $Update$ in this transition. Moreover, the only rule v can execute in $C_0 \mapsto C_1$ is a $MatchFirst$, according to Point 2. Thus v does not change its p-value in $C_0 \mapsto C_1$ and so $p_v = AskFirst(v)$ in C_1. Now, in C_0, $v \in matched(N(p_v)) \wedge p_{p_v} = v$ thus p_v cannot execute $UpdateP$ in $C_0 \mapsto C_1$ and thus it cannot change its p-value. So, $p_{p_v} = v$ in C_1.

Point 3c holds since after u executed a $MatchSecond$ in $C_0 \mapsto C_1$, observe that necessarily $p_u = AskSecond(u)$ in C_1. Moreover, $s_u = True$ in C_1 so, according to Lemma 4.5.11, $\exists y \in single(N(u)) : p_u = y \wedge p_y = u$ in C_1.

$p_v = AskFirst(v)$ and $p_{p_v} = v$ hold in C_0, according to Points 2a and 2b. Moreover, $p_u = AskSecond(u)$ holds in C_0 since u writes $True$ in end_u while executing a $MatchSecond$ in $C_0 \mapsto C_1$. Finally, by Point 2, v can only execute $MatchFirst$ in $C_0 \mapsto C_1$, thus variable s_v remains $True$ in transition $C_0 \mapsto C_1$ and Point 3d holds.

We now prove Point 4. If $end_v = True$ in C_1, then according to Lemma 4.5.13, u is not eligible for any rule in C_1. Now, let us consider the case $end_v = False$ in C_1. By Points 3c and 3d, u is not eligible for $ResetMatch$. By Point 3c and Lemma 4.5.12, u is not eligible for $Update$. By Points 3a, 3b and 3c, u is not eligible for $MatchSecond$. Finally, since u is Second in C_1, u is not eligible for $MatchFirst$ neither and Point 4 holds.

Now since between C_1 and C_2, v does not execute any rule (by Point 5b), and since p_u (resp. p_v) is not eligible for $UpdateP$ while u (resp. v) does not move (because $p_{p_u} = u$ (resp. $p_{p_v} = v$)), then $Ask(u)$, $Ask(v)$, p_{p_u} and p_{p_v} remain constant while u does not make any move. And so, properties 3a, 3b, 3c and 3d hold for any configuration between C_1 and C_2, thus u is not eligible for any rule between C_1 and C_2 and u will not execute any move from C_1 to C_3. Moreover, the end_v-value is the same from C_1 to C_2.

If $end_v = False$ in C_2, then v is eligible for a $MatchFirst$ and that it will write $True$ in its end_v-variable while all properties of Point 3 will still hold in C_3. Thus Point 5 holds. □

Theorem 4.5.18. *In any execution, a matched node u can write $end_u := True$ at most twice.*

Proof. Let (u, v) be a matched edge and $\mathcal{E}$ be an execution where u writes $True$ in its variable end_u at least twice. Let $C_0 \mapsto C_1$ be the transition where u writes $True$ in end_u for the second time in $\mathcal{E}$. If u is First (resp. Second) in C_0 then from Lemmas 4.5.13 and 4.5.16, (resp. 4.5.17), from C_1, neither u nor v will ever be eligible for any rule. □

The number of times single nodes can change their *end*-variable

In the following, μ denotes the number of matched nodes and σ the number of single nodes.

Lemma 4.5.19. *Let x be a single node. If x writes $True$ in some transition $C_0 \mapsto C_1$ then, in C_0, $\exists u \in matched(N(x)) : p_x = u \wedge p_u = x \wedge end_x = False \wedge end_u = True$.*

Proof. To write True in its *end* variable, a single node must apply *UpdateEnd*. Observe now that to apply this rule, the conditions described in the Lemma must hold. □

Lemma 4.5.20. *Let u be a matched node. Consider an execution $\mathcal{E}$ starting after u executed some rule and in which end_u is always $True$, except for the last configuration D of $\mathcal{E}$ in which it may be $False$. Let $\mathcal{E}\backslash D$ be all configurations of $\mathcal{E}$ but configuration D. In $\mathcal{E}\backslash D$, the following holds:*

- *$p_u \in single(N(u))$;*
- *p_u remains constant.*

Proof. Since $end_u = True$ in $\mathcal{E}\backslash D$, the last rule executed before $\mathcal{E}$ is necessarily a *Match* rule. So, at the beginning of $\mathcal{E}$, $p_u \in single(N(u))$, otherwise, u would not have executed a *Match* rule, but an *Update* instead.

We prove now that in $\mathcal{E}\backslash D$, p_u remains constant. Assume by contradiction that there exists a transition in which p_u is modified. Let $C_0 \mapsto C_1$ be the first such transition. First, observe that in $\mathcal{E}\backslash D$, u cannot execute *ResetMatch* nor *Update* since that would set end_u to $False$. Thus u must execute a *Match* rule in $C_0 \mapsto C_1$. Since the value of p_u changes in this transition, this implies that $Ask(u) \neq p_u$ in C_0. Thus, whatever the *Match* rule, observe now that in C_1, end_u must be $False$, which gives a contradiction and concludes the proof. □

Definition 4.5.21. *Let u be a matched node. We say that a transition $C_0 \mapsto C_1$ is of type* "a single copies True from u" *if there exists a single node x such that $(p_x = u \wedge p_u = x \wedge end_x = False)$ in C_0 and $end_x = True$ in C_1. Note that by Lemma 4.5.19, $end_u = True$ in C_0 and $x \in single(N(u))$.*

If a transition $C_0 \mapsto C_1$ is of type "a single node copies True from u" *and if x is the single node with $(p_x = u \wedge p_u = x \wedge end_x = False)$ in C_0 and $end_x = True$ in C_1, then we will say x* copies $True$ from u.

Lemma 4.5.22. *Let u be a matched node and $\mathcal{E}$ be an execution. In $\mathcal{E}$, there are at most three transitions of type* "a single copies $True$ from u".

Proof. Let $\mathcal{E}$ be an execution. We consider some sub-executions of $\mathcal{E}$.

Let $\mathcal{E}_{init}$ be a sub-execution of $\mathcal{E}$ that starts in the initial configuration of $\mathcal{E}$ and that ends just after the first move of u. Let $C_0 \mapsto C_1$ be the last transition of $\mathcal{E}_{init}$. Observe that u does not execute any move until configuration C_0 and executes its

first move in transition $C_0 \mapsto C_1$. We will write $\mathcal{E}_{init} \setminus C_1$ to denote all configurations of $\mathcal{E}_{init}$ but the configuration C_1. We prove that there is at most one transition of type *"a single copies True from u"* in $\mathcal{E}_{init}$.

There are two possible cases regarding end_u in all configuration of $\mathcal{E}_{init} \setminus C_1$: either end_u is always $True$ or end_u is always $False$. If $end_u = False$ then by Definition 4.5.21, no single node can copy $True$ from u in $\mathcal{E}_{init}$, not even in transition $C_0 \mapsto C_1$, since no single node is eligible for such a copy in C_0. If $end_u = True$, once again, there are two cases: either (i) $(p_u = null \vee p_u \notin single(N(u)))$ in all configuration of $\mathcal{E}_{init} \setminus C_1$, or (ii) $(p_u \in single(N(u)))$ in $\mathcal{E}_{init} \setminus C_1$. In case (i) then by Definition 4.5.21 no single node can copy $True$ from u in $\mathcal{E}_{init}$, not even in $C_0 \mapsto C_1$. In case (ii), observe that p_u remains constant in all configurations of $\mathcal{E}_{init} \setminus C_1$, thus at most one single node can copy $True$ from u in $\mathcal{E}_{init}$.

Let $\mathcal{E}_{true}$ be a sub-execution of $\mathcal{E}$ starting after u executed some rule and such that: for all configurations in $\mathcal{E}_{true}$ but the last one, $end_u = True$. There is no constraint on the value of end_u in the last configuration of $\mathcal{E}_{true}$. According to Lemma 4.5.20, $p_u \in single(N(u))$ and p_u remains constant in all configurations of $\mathcal{E}_{true}$ but the last one. This implies that at most one single can copy $True$ from u in $\mathcal{E}_{true}$.

Let $\mathcal{E}_{false}$ be an execution starting after u executed some rule and such that: for all configurations in $\mathcal{E}_{false}$ but the last one, $end_u = False$. There is no constraint on the value of end_u in the last configuration of $\mathcal{E}_{false}$. By Definition 4.5.21, no single node will be able to copy $True$ from u in $\mathcal{E}_{false}$.

To conclude, by Corollary 4.5.18, u can write $True$ in its *end* variable at most twice. Thus, for all executions $\mathcal{E}$, $\mathcal{E}$ contains exactly one sub-execution of type $\mathcal{E}_{init}$, and at most two sub-executions of type $\mathcal{E}_{true}$ and the remaining sub-executions are of type $\mathcal{E}_{false}$. This implies that in total, we have at most three transitions of type "*a single copies True from u*" in $\mathcal{E}$. □

Lemma 4.5.23. *In any execution, the number of transitions where a single node writes True in its end variable is at most* 3μ.

Proof. Let $\mathcal{E}$ be an execution and x be a single node. If x writes $True$ in end_x in some transition of $\mathcal{E}$, then x necessarily executes an *UpdateEnd* rule and by Definition 4.5.21, this means *x copies True from some matched node* in this transition. Now the lemma holds by Lemma 4.5.22. □

Lemma 4.5.24. *In any execution, the number of transitions where a single node changes the value of its end variables (from True to False or from False to True) is at most* $\sigma + 6\mu$ *times.*

Proof. A single node can write $True$ in its *end* variable at most 3μ times, by Corollary 4.5.23. Each of these writings allows one writing from $True$ to $False$, which leads to 6μ possible modifications of the *end* variables. Now, let us consider a single node x. If $end_x = False$ initially, then no more change are possible, however if $end_x = True$ initially, then one more modification from $True$ to $False$ is possible. Each single node can do at most one modification due to this initialization and thus the Lemma holds. □

How many *Update* in an execution?

Definition 4.5.25. *Let u be a matched node and C be a configuration. We define* $Cand(u, C) = \{x \in single(N(u)) : (p_x = u \ \vee \ end_x = False)\}$ *which is the set of vertices considered by the function BestRematch(u) in configuration C.*

Lemma 4.5.26. *Let u be a matched node that has already executed some rule. If there exists a transition $C_0 \mapsto C_1$ such that u is eligible for Update in C_1 and not in C_0, then there exists a single node x such that $x \in Cand(u, C_0) \backslash Cand(u, C_1)$ or $x \in Cand(u, C_1) \backslash Cand(u, C_0)$. Moreover, in transition $C_0 \mapsto C_1$, x flips the value of its end variable.*

Proof. Since u has already executed some rule, to become eligible for *Update* in transition $C_0 \mapsto C_1$, necessarily the second disjonction in the *Update* rule must hold, by Lemma 4.5.12. This implies that $(\alpha_v, \beta_v) \neq BestRematch(v)$ must become $True$ in $C_0 \mapsto C_1$. Now either $Lowest(Cand(u, C_0)) \notin Cand(u, C_1)$ or $\exists x \notin Cand(u, C_0)$ such that $x = Lowest(Cand(u, C_1))$. This proves the first point.

For the second point we first consider the case $x \in Cand(u, C_1)$ and $x \notin Cand(u, C_0)$. Necessarily $end_x = True \wedge p_x \neq u$ in C_0 and $end_x = False \vee p_x = u$ in C_1. If $p_x = u$ in C_1 then in transition $C_0 \mapsto C_1$, x has executed an $UpdateP$ and the second point holds. Assume now that $p_x \neq u$ in C_1. Necessarily $end_u = False$ in C_1 and the Lemma holds.

We consider the second case in which $x \notin Cand(u, C_1)$ and $x \in Cand(u, C_0)$. Necessarily in C_1, $p_x \neq v$ and $end_x = True$. Thus if $end_x = False$ in C_0 the lemma holds. Assume by contradiction that $end_x = True$ in C_0. This implies $p_x = u$ in C_0. But since in C_1 $p_x \neq u$ then x executed either $UpdateP$ or $UpdateEnd$ in $C_0 \mapsto C_1$ which implies $end_x = False$ in C_1, a contradiction. This completes the proof. □

Corollary 4.5.27. *Matched nodes can execute at most $\Delta(\sigma + 6\mu) + \mu$ times the Update rule.*

Proof. Initially each matched node can be eligible for an *Update*. Now, let us consider only matched nodes that have already executed a move. For such a node to become eligible for an *Update* rule, at least one single node must change the value of its *end* variable by Lemma 4.5.26. Thus, each change of the *end* value of a single node can generate at most Δ matched nodes to be eligible for an *Update*. By Lemma 4.5.24, the number of transitions where a single node changes the value of its *end* variables is at most $\sigma + 6\mu$ times. Thus we obtain at most $\Delta(\sigma + 6\mu)$ *Update* generated by a change of the *end* value of a single node and the Lemma holds. □

A bound on the total number of moves in any execution

Definition 4.5.28. *Let (u, v) be a matched edge. In the following, we call $\mathcal{F}$ a finite execution where neither u nor v execute the Update rule. Let $D_{\mathcal{E}}$ be the first configuration of $\mathcal{F}$ and $D'_{\mathcal{E}}$ be the last one.*

Observe that in the execution $\mathcal{F}$, all variables α and β of nodes u and v remain constant and thus, predicates *AskFirst* and *AskSecond* for these two nodes remain constant too.

Lemma 4.5.29. *If $Ask(u) = Ask(v) = null$ in $\mathcal{F}$, then u and v can both execute at most one $ResetMatch$.*

Proof. Recall that in the execution $\mathcal{F}$, by definition, u and v do not execute the *Update* rule. Moreover, these two nodes are not eligible for *Match* rules since $Ask(u) = Ask(v) = null$. Thus they are only eligible for *ResetMatch*. Observe now it is not possible to execute this rule twice in a row, which completes the proof. □

Lemma 4.5.30. *Assume that in $\mathcal{F}$, u is First and v is Second. If s_u is False in all configurations of $\mathcal{F}$ but the last one, then v can execute at most one rule in $\mathcal{F}$.*

Proof. Since $s_u = False$ in all configurations of $\mathcal{F}$ but the last one, node v which is *Second* can only be eligible for $ResetMatch$. Observe that if v executes $ResetMatch$, it is not eligible for a rule anymore and the Lemma holds. □

Lemma 4.5.31. *Assume that in $\mathcal{F}$, u is First and v is Second. If s_u is False throughout $\mathcal{F}$, then u can execute at most one rule in $\mathcal{F}$.*

Proof. Node u can only be eligible for $MatchFirst$. Assume u executes $MatchFirst$ for the first time in some transition $C_0 \mapsto C_1$, then in C_1, necessarily, $p_u = AskFirst(u)$, $s_u = False$ (by hypothesis) and $end_u = False$ by Lemma 4.5.10. Let $\mathcal{F}_1$ be the execution starting in C_1 and finishing in $D'_{\mathcal{E}}$. Since in $\mathcal{F}_1$, there is no $Update$ of nodes u and v, observe that $p_u = AskFirst(u)$ remains $True$ in this execution. Assume by contradiction that u executes another $MatchFirst$ in $\mathcal{F}_1$. Consider the first transition $C_2 \mapsto C_3$ after C_1 when it executes this rule. Notice that between C_1 and C_2 it does not execute rules. Thus, in C_2, $p_u = AskFirst(u)$, $s_u = False$ and $end_u = False$ hold. Now, if u executes $MatchFirst$ in C_2 it is necessarily to modify the value of s_u or end_u. By definition, it cannot change the value of s_u. Moreover it cannot modify the value of end_u as this would imply by Lemma 4.5.10 that $s_u = True$ in C_3. This completes the proof. □

Lemma 4.5.32. *Let (u, v) be a matched edge. Assume that in $\mathcal{F}$, u is First, v is Second and that u writes True in s_u in some transition of $\mathcal{F}$. Let $C_0 \mapsto C_1$ be the transition in $\mathcal{F}$ in which u writes True in s_u for the first time. Let $\mathcal{F}_1$ be the execution starting in C_1 and finishing in $D'_{\mathcal{E}}$. In $\mathcal{F}_1$, u can apply at most 3 rules and v at most 2.*

Proof. We first prove that in $\mathcal{F}_1$, s_u remains $True$. Observe that u cannot execute $Update$ nor $ResetMatch$ since it is $First$. So, u can only execute $MatchFirst$ in $\mathcal{F}_1$. For u to write $False$ in s_u, it must exists a configuration in $\mathcal{F}_1$ such that $p_u \neq AskFirst(u) \vee p_{p_u} \neq u \vee p_v \notin \{AskSecond(v), null\}$. Let us prove that none of these cases are possible.

Since u executed $MatchFirst$ in transition $C_0 \mapsto C_1$ writing $True$ in s_u then, by definition of this rule, $p_u = AskFirst(u) \wedge p_{p_u} = u \wedge p_v \in \{AskSecond(v), null\}$ holds in C_0. As there is no $Update$ of u and v in $\mathcal{F}$, then $AskFirst(u)$ and $AskSecond(v)$ remain constant throughout $\mathcal{F}$ (and $\mathcal{F}_1$). So each time u executes a $MatchFirst$, it writes the same value $AskFirst(u)$ in its p-variable. Thus $p_u = AskFirst(u)$ holds throughout $\mathcal{F}_1$. Moreover, each time v executes a rule, it writes either $null$ or the same value $AskSecond(v)$ in its p-variable. Thus $p_v \in \{AskSecond(v), null\}$ holds throughout $\mathcal{F}_1$. Now by Lemma 4.5.11, in C_1 we have, $\exists x \in single(N(u)) : p_u = x \wedge p_x = u$, since $s_u = True$. This stays $True$ in $\mathcal{F}_1$ as p_u remains constant and x will then not be eligible for $UpdateP$ in $\mathcal{F}_1$. Thus $p_{p_u} = u$ holds throughout $\mathcal{F}_1$. Thus, $p_u = AskFirst(u) \wedge p_{p_u} = u \wedge p_v \in \{AskSecond(v), null\}$ holds throughout $\mathcal{F}_1$ and so $s_u = True$ throughout $\mathcal{F}_1$.

This implies that in $\mathcal{F}_1$, v is only eligible for $MatchSecond$. Consider the first time it executes this rule in some transition $B_0 \mapsto B_1$, with $B_1 \geq C_1$. Then, in B_1, $p_v = AskSecond(v)$, $s_v = end_v$ and this will hold between B_1 and $D'_{\mathcal{E}}$. If $end_v = True$ in B_1 then this will stay $True$ between B_1 and $D'_{\mathcal{E}}$. Indeed, p_v is not eligible for $UpdateP$ and we already showed that $p_u = AskFirst(u)$ holds in $\mathcal{F}_1$. In that case, between B_1 and $D'_{\mathcal{E}}$, v will not be eligible for any rule and so v will have executed at most one rule in $\mathcal{F}_1$. In the other case, that is $end_v (= s_v) = False$ in B_1, since $p_v = AskSecond(v)$ holds between B_1 and $D'_{\mathcal{E}}$, necessarily, the next time v executes a $MatchSecond$ rule, it is to write $True$ in end_v. After that observe that v is not eligible for any rule. Thus, v can execute at most 2 rules in $\mathcal{F}_1$.

To conclude the proof it remains to count the number of moves of u in $\mathcal{F}_1$. Recall that we proved that s_u is always $True$ in $\mathcal{F}_1$. Thus whenever u executes a $MatchFirst$, it is to modify the value of its end variable. Observe that this value depends, in fact, on the value of end_v and on p_v since we proved $p_u = AskFirst(u) \wedge p_{p_u} = u \wedge s_u \wedge p_v \in \{AskSecond(v), null\}$ holds throughout $\mathcal{F}_1$. Since we proved that in $\mathcal{F}_1$, v can execute at most two rules, this implies that these variables can have at most three different values in $\mathcal{F}_1$. Thus u can execute at most 3 rules in $\mathcal{F}_1$. □

Lemma 4.5.33. *Assume that in $\mathcal{F}$, u is First and v is Second. If s_u is True throughout $\mathcal{F}$ and if u does not execute any move in $\mathcal{F}$, then v can execute at most two rules in $\mathcal{F}$.*

Proof. By Definition 4.5.28, v cannot execute $Update$ in $\mathcal{F}_1$. Since we suppose that in $\mathcal{F}_1$, $s_u = True$ then v is not eligible for $ResetMatch$. Thus in $\mathcal{F}_1$, v can only execute $MatchSecond$. After it executed this rule for the first time, $p_v = AskSecond(v)$ and $s_v = end_v$ will always hold, since v is only eligible for $MatchSecond$. Thus the second time it executes this rule, it is necessarily to modify its end_v and s_v variables. Observe that after that, since u does not execute rules, v is not eligible for any rule. □

Lemma 4.5.34. *In $\mathcal{F}$, u and v can globally execute at most 12 rules.*

Proof. If $Ask(u) = Ask(v) = null$, the Lemma holds by Lemma 4.5.29. Assume now that u is $First$ and v $Second$. We consider two executions in $\mathcal{F}$.

Let $C_0 \mapsto C_1$ be the first transition in $\mathcal{F}$ in which u executes a rule. Let $\mathcal{F}_0$ be the execution starting in $D_{\mathcal{E}}$ and finishing in C_0. There are two cases.

If $s_u = False$ in $\mathcal{F}_0$ then v is only eligible for $ResetMatch$ in this execution. Observe that after it executes this rule for the first time in $\mathcal{F}_0$, it is not eligible for any rule after that in $\mathcal{F}_0$.

If $s_u = True$ in $\mathcal{F}_0$ then by Lemma 4.5.33, v can execute at most two rules in this execution. In transition $C_0 \mapsto C_1$, u and v can execute one rule each.

Let $\mathcal{F}_1$ be the execution starting in C_1 and finishing in $D'_{\mathcal{E}}$. Whatever rule u executes in transition $C_0 \mapsto C_1$ observe that u either writes $True$ or $False$ in s_u. If u writes $True$ in s_u in transition $C_0 \mapsto C_1$, then by Lemma 4.5.32, u and v can execute at most five rules in total in $\mathcal{F}_1$.

Consider the other case in which u writes $False$ in C_1. Let $C_2 \mapsto C_3$ be the first transition in $\mathcal{F}_1$ in which u writes $True$ in s_u. Call $\mathcal{F}_{10}$ the execution between C_1 and C_3 and $\mathcal{F}_{11}$ the execution between C_3 and $D'_{\mathcal{E}}$. By definition, s_u stays $False$ in $\mathcal{F}_{10} \backslash C_3$. Thus in $\mathcal{F}_{10} \backslash C_3$, u can execute at most one rule, by Lemma 4.5.31. Now in $\mathcal{F}_{10}$, u can execute at most two rules. By Lemma 4.5.30, v can execute at most one rule in $\mathcal{F}_{10}$. In total, u and v can execute at most three rules in $\mathcal{F}_{10}$. In $\mathcal{F}_{11}$, u and v can execute at most five rules by Lemma 4.5.32. Thus in $\mathcal{F}_1$, u and v can apply at most eight rules. □

Theorem 4.5.35. *In any execution, matched nodes can execute at most $12\Delta(\sigma + 6\mu) + 18\mu$ rules.*

Proof. Let k be the number of edges in the underlying maximal matching, with $k = \frac{\mu}{2}$. For $i \in [1, .., k]$, let $\{(u_i, v_i) = a_i\}$ be the set of matched edges. By $Update(a_i)$ we denote an $Update$ rule executed by node u_i or v_i. By Lemma 4.5.34, between two $Update(a_i)$ rules, nodes u_i and v_i can execute at most 12 rules. By Corollary 4.5.27, there are at most $\Delta(\sigma + 6\mu) + \mu$ executed $Update$ rules. Thus in

total, nodes can execute at most $\sum_{i=1}^{k} 12 \times (\#Update(a_i) + 1)$

$= 12\sum_{i=1}^{k} \#Update(a_i) + 12\sum_{i=1}^{k} 1 \leq 12(\Delta(\sigma + 6\mu) + \mu) + 12k = 12\Delta(\sigma + 6\mu) + 18\mu$ rules. □

Lemma 4.5.36. *In any execution, single nodes can execute at most σ times the ResetEnd rule.*

Proof. We prove that a single node x can execute the *ResetEnd* rule at most once. Assume by contradiction that it executes this rule twice. Let $C_0 \mapsto C_1$ be the transition when it executes it the second time. In C_0, $end_x = True$, by definition of the rule. Since x already executed a *ResetEnd* rule, it must have, at some point, wrote $True$ in end_x. This is only possible through an execution of *UpdateEnd*. Thus consider the last transition $D_0 \mapsto D_1$ in which it executed this rule. Observe that $D_1 \leq C_0$. Since between D_1 and C_0, end_x remains $True$, observe that x does not execute any rule between these two configurations. Now since in D_1, $p_x \neq null$ and this holds in C_0 then x is not eligible for *ResetEnd* in C_0, which gives the contradiction. This implies that single nodes can execute at most $\mathcal{O}(\sigma)$ times the *ResetEnd* rule. □

Lemma 4.5.37. *In any execution, single nodes can execute at most $\sigma + 6\mu$ times the UpdateEnd rule.*

Proof. By Lemma 4.5.24, single nodes can change the value of their *end* variable at most $\sigma + 6\mu$ times. Thus, they can apply *UpdateEnd* at most $\sigma + 6\mu$ times, since in every application of this rule, the value of the *end* variable must change. □

Lemma 4.5.38. *In any execution, single nodes can execute at most $12\Delta(\sigma + 6\mu) + 18\mu + 1$ times the UpdateP rule.*

Proof. Let x be a single node. Let $C_0 \mapsto C_1$ be a transition in which x executes an *UpdateP* rule and let $C_2 \mapsto C_3$ be the next transition after C_1 in which x executes an *UpdateP* rule. We prove that for x to execute the *UpdateP* rule in $C_2 \mapsto C_3$, a matched node has to execute a move between C_0 and C_2.

In C_1 there are two cases: either $p_x = null$ or $p_x \neq null$. Assume to begin that $p_x = null$. This implies that in C_0 the set $\{w \in N(x) | p_w = x\}$ is empty. In C_2, $p_x = null$, since between C_1 and C_2, x can only apply *UpdateEnd* or *ResetEnd*. Thus if it applies *UpdateP* in C_2, necessarily $\{w \in N(x) | p_w = x\} \neq \emptyset$. This implies that a matched node must have executed a *Match* rule between C_1 and C_2 and the lemma holds in that case.

Consider now the case in which $p_x = u$ with $u \neq null$ in C_1. By definition of the *UpdateP* rule, we also have $u \in matched(N(x)) \wedge p_u = x$ holds in C_0. In C_2 we still have that $p_x = u$ since between C_1 and C_2, x can only execute *UpdateEnd* or *ResetEnd*. Thus if x executes *UpdateP* in C_2, necessarily $p_{p_x} \neq x$. This implies that $p_u \neq x$ and so u executed a rule between C_0 and C_2.

Now, the lemma holds by Theorem 4.5.35. □

Corollary 4.5.39. *In any execution, nodes can execute at most $\mathcal{O}(n^2)$ moves.*

Proof. According to Lemmas 4.5.36, 4.5.37 and 4.5.38, single nodes can execute at most $\mathcal{O}(n^2)$ moves. Moreover, according to Theorem 4.5.35, matched nodes can execute at most $\mathcal{O}(n^2)$ moves. □

Corollary 4.5.40. *The algorithm* PolyMatch *converges in* $O(n^2)$ *moves under the adversarial distributed daemon and in a general graph, provided that an underlying maximal matching has been initially built.*

Recall that the algorithm PolyMatch assumes an underlying maximal matching. As we said in section 4.3, we can use the self-stabilizing maximal matching algorithm of Manne *et al.* [106] that stabilizes in $O(m)$ moves. Then, using a classical composition of these two algorithms [47], we obtain a total time complexity of $O(m \times n^2)$ moves under the adversarial distributed daemon.

Chapter 5

Maximal Matching in the Link Register Model

In this chapter we present our second self-stabilizing algorithm. It finds a maximal matching in the Link Register model. See [35] for a preliminary version of the paper.

5.1 Introduction

The algorithm of the previous section, as well as the self-stabilizing algorithms for matching problems which can be found in the literature, that we described in Section 4.1, work in a state model, in which nodes can directly access the variables of adjacent nodes. This model was introduced by Dijkstra in [46]. However, this model fails to capture the aynchronous phenomena that happen in many real-life distributed systems, which led [48] to introduce the link-register model (or read-write model). In this model, communications are abstracted by registers in which nodes can write and read values. Atomicity conditions define the granularity of the algorithm. A variety of atomicity conditions exist (see [86] for a survey and some results on the strength of the different atomicities).

Read/write atomic registers are registers associated to each (directed) link, in which one of the nodes can write, and the other read; each read or write operation on the register is atomic (meaning that it cannot be interrupted), but a read in the register can happen arbitrarily long after the previous write, forcing the reading process to act based on outdated values (as opposed to what happens in Dijkstra-type state model). Read/write atomic registers can be implemented over message-passing models (at a large cost however), as in [114] for instance. This kind of atomicity is the strongest, meaning that algorithm written under this model also solve the problem under the other classical models.

The possible occurrence of faults in the execution of the algorithm is taken into account with the paradigm of self-stabilization, as defined in Section 1.4.2. A fault (or a sequence of faults) can lead the system to an arbitrary configuration of the processes and registers, starting from which the execution (seen as a completely new execution starting from this arbitrary configuration) must eventually resume a correct behavior in finite time.

To the best of our knowledge, there exists only one algorithm dealing with the self-stabilizing construction of a maximal matching under the link-register model with read/write atomicity. Chattopadhyay *et al.* [28] present this solution in a general anonymous network. Their algorithm assumes a fair distributed daemon and reaches a linear round complexity. As described in Section 1.4.2, a round of

computation is the shortest partial computation in which each enabled processor executes at least one full iteration of its algorithm. Processor performs as many as $\Delta + 1$ atomic operations (or steps) in one complete iteration, with Δ the maximum degree of the network. Thus a round costs at least Δn atomic steps and so the atomic step complexity is $\Omega(\Delta n^2)$. However this is just a lower bound. Observe that it is not possible to deduce an upper bound from this result since, according to the definition, in a round, the number of activations of each process has to be finite but not bounded.

In this chapter we propose a new distributed self-stabilizing algorithm solving the matching problem in a link-register model with read/write atomicity. This is the first algorithm working under the *adversarial* distributed daemon. The algorithm is presented under the form of guarded rules (usual for state model algorithms, but as far as we are aware of, never used before for link-register algorithms). This allows to underscore the granularity of the model, each configuration being the result of the application of an arbitrary subset of guarded rules to the previous configuration.

Our solution assumes unique identifiers in the system while the Chattopadhyay *et al.* solution assumes an anonymous network. However, Chattopadhyay *et al.* assume a fair distributed daemon and the knowledge on an upper bound on the size of the system while we do not assume any fairness neither any size knowledge in our solution. Finally, our solution gives an $O(\Delta m)$ step complexity while there is no upper bound for the Chattopadhyay *et al.* solution.

In the literature, some transformers from a communication model to another exist. To the best of our knowledge, the Beauquier *et al.* [15] transformer is the only transformer from the state model with the composite atomicity to the link-register model with the read/write atomicity and that deals with the adversarial distributed daemon.

We could have applied this transformer to the Manne *et al.* algorithm [106] in order to obtain a self-stabilizing algorithm for a maximal matching construction in the link-register model with read/write atomicity. However, no complexity is given for the Beauquier *et al.* transformer, so our solution stabilizes more quickly ($O(\Delta m)$) than this transformation would guaranty (only finite).

Another transformer between two communication models is presented in the Dolev book [47]. Dolev presents a transformer from the state model with composite atomicity to the link-register model with read/write atomicity. The difference with the Beauquier *et al.* transformer is the fair sequential daemon assumption. The Dolev transformer is proved to have an exponential round complexity at worst. So, once again, if we compare our solution with the composition of the Dolev transformer and the Manne *et al.* algorithm, not only our solution assumes a stronger daemon but it also stabilizes more quickly.

Another approach would be to use communication primitives giving some nice properties on read and write atomic actions and leading to the simulation of the state model with composite atomicity. A good start to make such a simulation is the Johnen *et al.* communication primitives [88] that guarantees some properties between two consecutive Write performed by a unique node. The strongest one guarantees that between two Write, all neighbors read exactly once the latest written value. The property is obtained by blocking the write of a node until all its neighbors perform their read.

However, these primitives cannot be trivially used as a base for a transformer. Indeed, under the state model, a node reads the state of all its neighbors and changes its own state. Thus the model guarantees each read value corresponds to the last written value. In particular, this prevents a node from performing a move based on

the last written value of one neighbor (let say a) and the last but one written value of another neighbor (let say b). In other words, there is no incoming transition in such a configuration. Of course these a and b values can appear in a configuration (if the algorithms allows it), but if so these two values cannot correspond to the last written value for a and a past value for b. Thus, if the algorithm does not create such a couple of values (for instance in the case of the use of a common technique of a local agreement where each process progresses accordingly to the progression of one of its neighbor in a two-phase locking process for instance, as we can find in the Manne *et al.* algorithm, in the Goddard one, and finally in our contribution), then such a configuration is not reachable under the state model with composite atomicity, while it is in the model used in this chapter. Thus, if we use the Johnen *et al.* primitives with the Manne *et al.* algorithm, we obtain some new executions compared to the ones one obtains without the primitive but assuming the composite atomicity. In particular, an execution that infinitely often reaches the aforementioned kind of configurations.

The chapter is organized as follows. We first present the model under which the algorithm is designed in the next Section. Section 5.3 is devoted to the algorithm, the convergence and correctness of which are proven in the last section.

5.2 Model

For definitions and notations about models, the reader can refer to Section 1.4.2. In this contribution, we consider the link register communication model. Considering two adjacent processors u and v, there exists a register r_{uv} in which u is the only process allowed to write, and that v can read.

All nodes have the same variables; if var is a variable, var_u denotes the instance of this variable on process u. Each node u has a unique id id_u; in the following, for the sake of simplicity, we do not distinguish between u and id_u.

Each node u maintains a variable $p_u \in N(u) \cup \{null\}$ indicating the neighbor it is married or attempting to marry.

A configuration solves the maximal matching if it is such that $\forall u, (p_u \neq null \Rightarrow p_{p_u} = u) \wedge (p_u = null \Rightarrow \forall v \in N(u), p_v \neq null)$. The first part of this specification means that if a node points to one of its neighbor, this neighbor points to it; the second one implies maximality: if a node points to no other node, none of its neighbors is in the same situation, since they could marry and create a larger matching.

The algorithm is presented under the form of a set of guarded rules. To respect read/write atomicity, if a guard refers to the value of a neighbor's register (which implies the reading of this register), the associated action cannot write in a register. In particular, we decided to introduce the $Write$ guarded rule, that writes the adequate value in a register; other actions never write in any register. Thus, all actions consist either in readings in neighbor's registers and taking local actions, or in writing in its own register, which respects the read/write atomicity. A guarded rule is activable in a configuration if its guard is true.

Link-register algorithms are generally presented as an infinite loop of readings, local actions and writings for each node, and the chosen atomicity allows to decide the points in the algorithm at which the execution of a node can be suspended to let another node take over. We choose to show more explicitly the points at which a node suspends action, and thus the result in terms of configuration of each of its atomic actions: the next configuration in an execution is obtained by applying one or several actions of guarded rules whose guards are true.

The moment when a process u writes in register r_{uv} is the time starting from which the written value r_{uv} is available to v, thus, the writing is analogous to a message reception by v in a message-passing model. A node u reads in its register r_{uv} in all guards. This allows it to check that the writing register of a node has reached its correct value. This can be paralleled with an acknowledgement.

Recall, as introduced in Section 1.4.2.2 that by $C_i \mapsto C_{i+1}$ we denote a transition in some execution. We extend this notation with $C_i \mapsto^* C_j$ if $i \leq j$, and $C_i \mapsto^+ C_j$ if $i < j$; in this case, we say for any action $(R, u) \in A_k$ with $i \leq k \leq j$ that there has been a transition (R, u) between C_i and C_j.

5.3 Algorithm

The presented algorithm is based on the algorithm by Manne *et al* [106] written under the state model. A marriage is contracted in two phases: (1) *the selection* of the edge to add to the matching and (2) *the confirmation* or the *lock* of the edge in the matching in three steps. For the selection phase, the lowest id node of a pair proposes to its neighbor, that accepts (or not) the proposition. In the acceptation case, the marriage is confirmed in three steps. This scheme can be interrupted at any point, either because of another marriage being concluded, or because of a faulty initialization. Once the marriage is reached, the married nodes do not do any more moves, and the algorithm is eventually silent. At worst, the algorithm has to take $O(m\Delta)$ moves before reaching a maximal matching.

5.3.1 Variables description

Each node u has two local variables. Variable p_u is the identifier of the node u points to: nodes u and v are said to *be married* to each other if and only if u and v are neighbors, p_u points to v, and p_u points to u. We also use a variable m_u indicating the progress of u's marriage: $m_u \in \{0, 1, 2, 3\}$.

Each node u has a four bits register r_{uv} for each of its neighbors v. The first two bits $r_{uv}.p$ can take the value Idle if u points to *null* (ie $p_u = null$), You if it points to v, and Other if it points to a node $\neq v$. The last two bits $r_{uv}.m$ can be 0, 1, 2 or 3, indicating the progress of u's marriage.

5.3.2 Algorithm description

The *Seduction* and *Marriage* rules implement the selection of an edge for the matching: they set the p variable of a node to a candidate for a marriage. First, the node with the smallest id in a pair executes the *Seduction* rule, to which the node with the highest id can respond by executing the *Marriage* rule. This asymmetrical process avoids situations with nodes trying to seduce neighbors in a cycle, that could be reproduced by an adversarial daemon.

Under read/write atomicity, an offset is possible between the value of the local variables of a node (p, m) and the value of its registers. In order to avoid infinite executions during which the distributed daemon lets a node u attempt to marry a neighbor v just at the same time when v abandons its attempt to marry u, and then conversely at the next step, it is necessary to design a mechanism locking progressively a marriage. We achieve that with variable m, which takes values in $\{0, 1, 2, 3\}$: except for faulty initialization, $m_u = 0$ means that u did not start locking any marriage, and $m_u \geq 1$ means that u has a neighbor v such that $p_u = v \wedge p_v = u$.

If $m_u = 1$ or $m_u = 2$, then the marriage lock is in progress and if $m_u = 3$, then the lock is done. m is incremented in the execution of rule *Increase*.

The *Reset* rule ensures that local variables p and m for a node u have consistent values. It is executed when predicate $PRabandonment(u)$ or $PRreset(u)$ is true. $PRabandonment(u)$ means that u's marriage process should be restarted if u is trying to marry a node $v < u$ that is not seducing it, or if u is trying to seduce a node that is already married (at least, when the registers of v indicate that). This last case can happen when v is responding to several proposals at the same time, while the first one is provoked by "bad" initializations. $PRreset(u)$ indicates a discrepancy between the steps taken in the locking mechanism by the two processes involved in it. This is due to "bad" initializations.

5.3.3 Algorithm

5.3.3.1 Predicates and functions

$$\begin{aligned}
Correct_register_value(u,a) \equiv\ & \text{if } p_u = null \text{ then return } (Idle, 0)\\
& \text{else if } p_u = a \text{ then return } (You, m_u)\\
& \text{else return } (Other, m_u)
\end{aligned}$$

$$\begin{aligned}
PRabandonment(u) \equiv [\, & p_u \neq null \wedge (r_{p_u u}.p \neq You \wedge (u > p_u \vee m_u \neq 0))\\
& \vee (r_{p_u u} = (Other, 3) \wedge u < p_u)\,]
\end{aligned}$$

$$\begin{aligned}
PRreset(u) \equiv\ & (p_u \neq null) \wedge (r_{p_u u}.p = You) \wedge (\\
& (|m_u = 0 - r_{p_u u}.m| \geq 2)\\
& \vee\ (m_u = 0 \wedge r_{p_u u}.m = 1 \wedge u > p_u)\\
& \vee\ (m_u = 1 \wedge r_{p_u u}.m = 0 \wedge u < p_u)\\
& \vee\ (m_u = 1 \wedge r_{p_u u}.m = 2 \wedge u < p_u)\\
& \vee\ (m_u = 2 \wedge r_{p_u u}.m = 1 \wedge u > p_u)\\
& \vee\ (m_u = 3 \wedge r_{p_u u}.m = 2 \wedge u > p_u)\\
& \vee\ (m_u = 2 \wedge r_{p_u u}.m = 3 \wedge u < p_u))
\end{aligned}$$

5.3.3.2 Rules for each node u

$\forall \mathbf{a} \in \mathbf{N(u)}$,

$$\begin{aligned}
&\textbf{Write(a)} && :: r_{ua} \neq Correct_register_value(u,a)\\
& && \quad \rightarrow r_{ua} := Correct_register_value(u,a)\\
&\textbf{Seduction(a)} && :: p_u = null \ \wedge r_{ua} = Correct_register_value(u,a)\\
& && \quad \wedge\ r_{au} = (Idle, 0) \wedge (u < a) \quad \rightarrow (p_u, m_u) := (a, 0)\\
&\textbf{Marriage(a)} && :: p_u = null \ \wedge r_{ua} = Correct_register_value(u,a)\\
& && \quad \wedge\ r_{au} = (You, 0) \wedge (u > a) \quad \rightarrow (p_u, m_u) := (a, 0)
\end{aligned}$$

$$\begin{aligned}
\textbf{Increase} :: \ & p_u \neq null \wedge r_{up_u} = Correct_register_value(u, p_u)\\
& \wedge (r_{p_u u}.p = You) \wedge (\\
& \quad (m_u = 0) \wedge [\, (u < p_u \wedge r_{p_u u}.m = 1) \vee (u > p_u \wedge r_{p_u u}.m = 0)\,]\\
& \quad \vee\ (m_u = 1) \wedge [\, (u < p_u \wedge r_{p_u u}.m = 1) \vee (u > p_u \wedge r_{p_u u}.m = 2)\,]\\
& \quad \vee\ (m_u = 2) \wedge [\, (u < p_u \wedge r_{p_u u}.m = 2) \vee (u > p_u \wedge r_{p_u u}.m = 3)\,]\\
&) \rightarrow\\
& \qquad m_u := m_u + 1
\end{aligned}$$

$$\begin{aligned}
\textbf{Reset} :: \ & p_u \neq null \ \wedge r_{up_u} = Correct_register_value(u, p_u)\\
& \quad \wedge\ (PRabandonment(u) \vee PRreset(u)) \quad \rightarrow (p_u, m_u) := (null, 0)
\end{aligned}$$

5.3.4 About the rules

Let $\deg(u)$ be the degree of node u. A node u has $3\deg(u) + 2$ rules: one rule *Write*, *Seduction* and *Marriage* for each neighbor, plus one rule *Increase* and one

Reset. Observe that rules *Seduction* and *Marriage* can be executed when the node is pointing to *null*, so these rules can be executed over some "good " candidates. However, rules *Reset* and *Increase* can only be executed over the one neighbor the node is pointing to. Given a neighbor v of u, u can be activable only for rule $Marriage(v)$ or $Seduction(v)$, because the first one needs that $u > v$ and the second that $u < v$.

Definition 5.3.1 (v-*Increase*/v-*Reset*, v-rule and $R(-)$ rule)**.** *Let u and v be two nodes. We say that node u is activable in the configuration C for a v-Increase (resp. v-Reset) rule if, in C, u is activable for an Increase (resp. Reset) rule and $p_u = v$. We say that node u is eligible in the configuration C for a v-rule if u is eligible for one of the following rule: $\{Write(v), Seduction(v), Marriage(v), v\text{-}Increase, v\text{-}Reset\}$ in C. Finally, let R be any rule among Write, Marriage and Seduction. We say that u is eligible for a $R(-)$ rule, if there exists a neighbor of u, say a, such that u is eligible for a $R(a)$ rule.*

Observation 5.3.2. *Let u be a node and C be a configuration. In C, we have:*

1. *if $p_u = null$ then:*
   - *u is not eligible for Increase nor Reset;*
   - *$\forall v \in N(u)$: u is eligible for at most one rule among the set of rules $\{Write(v)$, $Marriage(v)$, $Seduction(v)\}$;*
2. *if $p_u \neq null$ then:*
   - *u is not eligible for $Marriage(-)$ nor $Seduction(-)$;*
   - *u is eligible for at most one rule among the set of rules $\{Write(p_u)$, Increase, Reset$\}$; moreover, if this rule is an Increase (resp. Reset), this is necessarily a p_u-Increase (resp. p_u-Reset);*
   - *and $\forall x \in N(u) \setminus \{p_u\}$: among all the x-rules, u can only be eligible for $Write(x)$;*

5.3.5 Execution examples

Below is an execution of the algorithm under the adversarial distributed daemon. Figure 5.1a shows the initial state of the execution. Node identifiers are indicated inside the circles. In this execution, we take $s < t$. Black arrows show the content of the local variable p and the absence of arrow means that $p = null$. s:$Write(t)$ means that node s executes the $Write(t)$ rule.

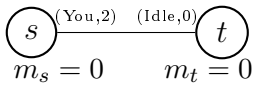
Consider the initial configuration (Figure 5.1a) in which variable and register values are as follows: $(p_s, m_s) = (p_t, m_t) = (null, 0)$, $r_{st} = (You, 2)$ and $r_{ts} = (Idle, 0)$. Thus, initially, nodes s and t are not matched. Since the local variables of s are not consistent with register r_{st}, node s executes $Write(t)$ in order to set $r_{st} = (Idle, 0)$ (Figure 5.1b).

Now, since s and t are not matched they can start a selection process in order to marry. The node with the smallest identifier, s, starts the process and thus, since $r_{ts} = (Idle, 0)$, s executes a $Seduction(t)$ rule in order to set p_s to t. Then s is eligible to execute a series of $Write(v)$ rules to update its registers (Figure 5.1c).

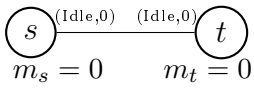
Once register r_{st} is updated, node t answers to the proposition of s by executing a $Marriage(s)$ rule, setting $p_t = s$. It is then eligible to execute a series of $Write(v)$ rules to update its registers. As long as t does not update its r_{ts} register, the process of locking the marriage cannot start since s needs $r_{ts}.p = You$ in order to start increasing its m variable. So assume t updates its r_{ts} register with a $Write(s)$ (Figure 5.1d).

From this point, both nodes point towards each other. The locking process of the marriage can start from this point. First, the node with the highest identifier, that is t, sets its m variable to 1 with a *s-Increase* and then updates its registers (see Figure 5.1e). Then node s executes a *t-Increase* and sets $m_s = 1$ followed by a $Write(t)$ rule to update its register (Figure 5.1f). After, it executes another *t-Increase* rule to set $m_s = 2$ (Figure 5.1g). This execution of two consecutive *t-Increase* by s guarantees that t has correct register values. Now, it is the turn of t to execute a *s-Increase* rule to set $m_t = 2$ followed by a $Write(s)$ in Figure 5.1h. In Figure 5.1i, s executes *t-Increase* and a $Write(t)$ in order to update its register. Finally, t executes a last *s-Increase* rule to set $m_t = 3$. At this point the matching of s and t is locked.

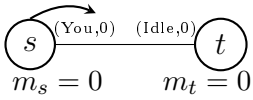
One might wonder why the *Increase* rule is not alternately executed by s and t. Indeed, s executes two consecutive *t-Increase* to set its m variable from 0 to 1 and then to 2, while t does not change its m value. Actually, the algorithm does not converge if nodes would perform the *Increase* rule alternately. These two consecutive *Increase* are a key point on the lock process of a marriage and this will be discussed in Section 5.3.6.



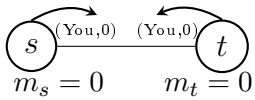
(a) Initial configuration



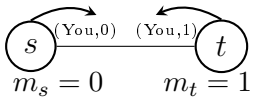
(b) $s : Write(t)$



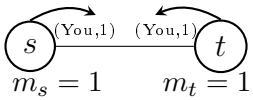
(c) $s : Seduction(t)$ then $s : Write(v), \forall v \in N(s)$



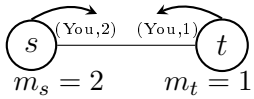
(d) $t : Marriage(s)$ then $t : Write(s)$



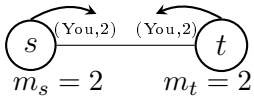
(e) t : *s-Increase* then $t : Write(s)$



(f) s : *t-Increase* then $s : Write(t)$



(g) s : *t-Increase* then $s : Write(t)$

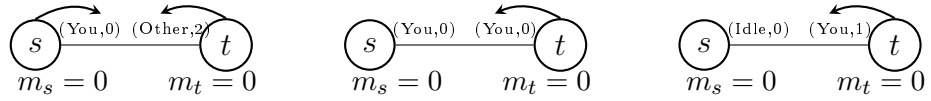


(h) t : *s-Increase* then $t : Write(s)$

Figure 5.1 – A typical execution of the algorithm. The absence of arrow means that the p-variable is equal to *null*.

Now, consider another execution, in which the algorithm deals with some inconsistencies between registers and local variable. In the initial configuration (Figure 5.2a), variables and registers have the following values: $(p_s, m_s) = (t, 0)$, $r_{st} = (You, 0)$, $(p_t, m_t) = (s, 0)$ and $r_{ts} = (Other, 3)$. Thus, the register r_{st} is badly initialized. Then s can execute *Reset* to restart a process of a new marriage. In parallel, t updates its registers by executing $Write(s)$ (Figure 5.2b). From now on, t starts the lock process by executing *Increase* and s updates its registers by executing $Write(t)$. Once again let us assume they both do so. Then t updates its register, leading to the configuration in Figure 5.2c. At this moment, node t has to give up the process of the marriage lock since $r_{st} = (Idle, 0)$, so that t is activable for a

Reset. Once t has executed this *Reset* and updated its register, the configuration is the one in Figure 5.1b.



(a) Initial configuration (b) t:$Write(s)$ // s:*t-Reset* (c) t:*s-Incr.*//s:$Write(s)$ then $t : Write(s)$.

Figure 5.2 – How to deal with incoherent values between registers and local variables?

5.3.6 Lock mechanism analysis

Figure 5.3 gives the ordered actions but the $Write$ rule, executed by two neighboring nodes s and t (with $s < t$) in order to completely get married after both nodes have performed a *Reset*. The *Increase* rule is the kernel of the lock mechanism allowing the two nodes to progress together.

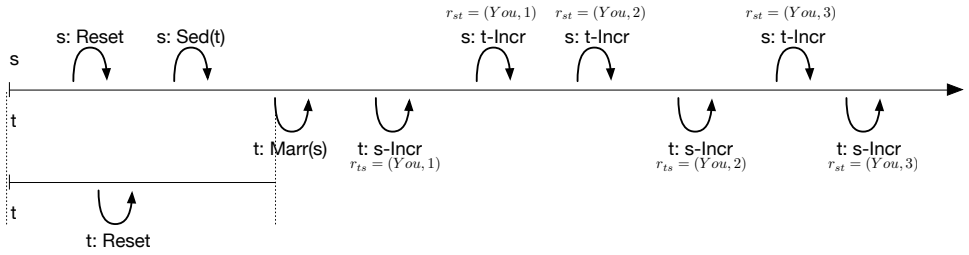
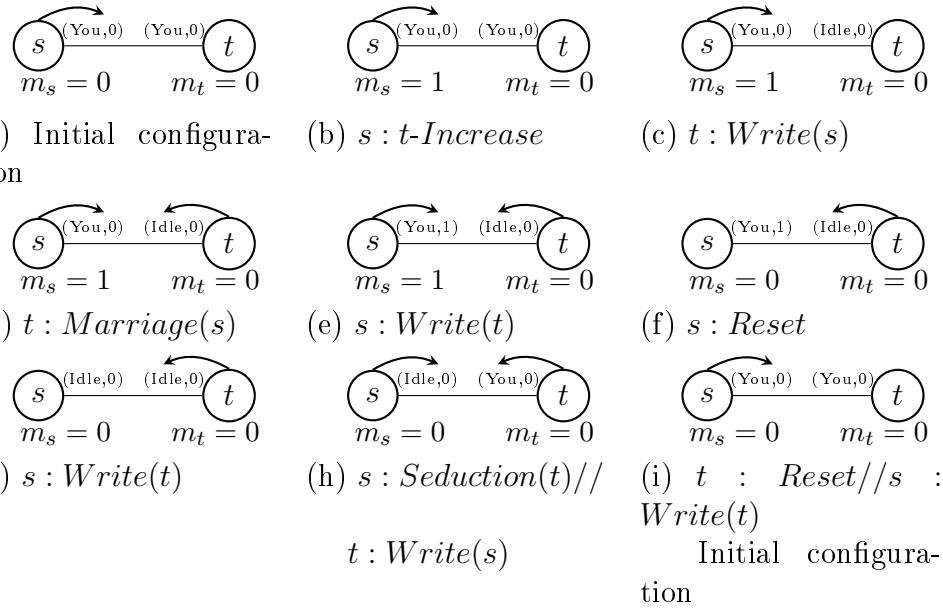


Figure 5.3 – Matching progress between s and t after both nodes reset. Actions performed by s (resp. t) are drawn above (resp. below) the execution line. In this execution, node t should execute a *Reset* before executing $Marriage(s)$ (see the second execution line).

Observe that our algorithm does not allow that s and t executes alternately the *Increase* rules because this would avoid our algorithm to stabilize. Using the same algorithm with only one modification in the *Increase* rule where s makes the first *Increase* while t does the second, we can exhibit a infinite execution that does not stabilize. This execution contains a loop on an incorrect configuration. Figure 5.4 gives this loop.

As Figure 5.4 illustrates it, t should initiate the locking mechanism by executing the first *Increase*. Thus, t executes two moves (*Marriage* and then *Increase*) in a row without s performing any *Increase*, allowing then to avoid incorrect execution. We use the same mechanism when it is the turn of s to perform its lock with the *Increase* rule. Thus, s executes two *Increase* in a row without t performing any *Increase*.

Observe that a node should be able to reset a proposition to a locked married node and so it has to be able to detect this lock. The lock progress is given in the m-value of a node, and the chosen value for a locked marriage should be the same for both married nodes in order for their neighbor to be able to perform this detection. We choose the value 3 instead of 2 to represent a locked marriage (called the *locked marriage value*) for the following reason.



(a) Initial configuration (b) $s : t\text{-}Increase$ (c) $t : Write(s)$

(d) $t : Marriage(s)$ (e) $s : Write(t)$ (f) $s : Reset$

(g) $s : Write(t)$ (h) $s : Seduction(t)//$ $t : Write(s)$ (i) $t : Reset//s : Write(t)$ Initial configuration

Figure 5.4 – Counter-example: why alternating actions creates loop in executions. The absence of arrow means that the p-variable is equal to *null*.

A node can write $m = 2$ $O(\Delta)$ times while it can only write $m = 3$ twice in an execution. Observe that when a node writes the locked marriage value in its m-variable, this can generate $O(\Delta)$ *Reset* (because of the $PRAbandonment$ predicate). Thus, if the locked marriage value is 2, the complexity contains a Δ^2 factor (see [35] for the complexity computation) while if it is 3, the complexity only contains a Δ factor.

5.3.7 Local impact after a topological change

In this section we give an analysis on the impact of a topological change on our algorithm. In particular, we give which node will have to take part of the matching reconstruction after a topological change. First observe that if a topological change occurs in a stable configuration, then a matched node can change its marriage if only if its married neighbor has disappeared.

Now, let us assume that a single node x disappears. Since the configuration is stable just before the topological change then all neighbors of x are matched and so the configuration remains stable. If a new node x appears, then all single neighbors of this new node will attempt a marriage with it. And all married neighbors of this new node will just perform one Write in order to update their new register. But there will be no impact on the nodes at distance 2 from x since single nodes at distance 1 from x only have married nodes neighbors and all married nodes at distance 1 from x prevent their single neighbors to get married.

Finally, if a node u disappears while u was married with node v. Observe that married neighbors of u or v won't remain activable after this topological change. We consider two sets: the set U, the single neighbor of u that are not neighbor of v and the set V, the single neighbors of v. Nodes in U only have married neighbors since there are single and u disappears. So, these nodes wont' remain activable after this topological change. However, nodes in V had only married neighbors before the topological change, but have now one single neighbor v, that just lost his spouse. So all node in V will attempt a marriage with v. But there will be no impact on the nodes at distance 2 from u or v for same reason than before.

In conclusion, the reconstruction generated by a topological change only concerns

nodes at distance at most two from the change. All the other nodes in the system will not execute any move due to this change.

On the other hand, in the Chattopadhyay et al. solution, the system can be globally affected with only one topological change. For instance, if a node with the smallest id appears in the system, and since the matching is built based on some local minimum id, then this new smallest id node can modify the local minimum id nodes at distance in $O(n)$ from the new node. And so, the matching reconstruction can concern a number of nodes that depends on the size of the system.

5.4 Proof of the Algorithm

5.4.1 State of an edge

We first focus on an edge (s,t) of the matching $\mathcal{M} = \{(a,b) \in E : p_a = b \wedge p_b = a\}$ built by our algorithm when $s < t$. In particular, we focus on values of local variables and registers of this edge in some chosen configurations.

Definition 5.4.1. *Let (s,t) be an edge with $s < t$. We say that in a configuration C, the edge (s,t) is in state (You, α, β) if* $(p_s = t \wedge p_t = s) \wedge (m_s = \alpha \wedge m_t = \beta)$

If an edge (s,t) is in state (You, α, β), then this edge belongs to the matching. Unfortunately, due to some "bad" initialization for instance, this edge can be removed from the matching at some point of the execution. In the following, we characterize an edge that belongs to the matching and that will forever remain in it.

A correct state corresponds to the situations appearing in the Figure 5.1 starting from step (d). Starting from a configuration where edge (s,t) is in a correct state, the two nodes, one after the other, execute *Increase* and $Write$ rules. A link is in an updated correct state when all registers of the edge are updated (and so exactly one node among s and t is eligible for an *Increase*), while it is toUpdate when the register of one of the two nodes is not up to date (and so exactly one node among s and t is eligible for a $Write$).

Definition 5.4.2 (Updated correct state)**.** *Consider an edge (s,t) with $s < t$ in state (You, α, β) in a configuration C. This link is in an* updated correct state *if*
$(r_{st} = (You, \alpha) \wedge r_{ts} = (You, \beta)) \wedge (\alpha, \beta) \in \{(0,0),(0,1),(1,1),(2,1),(2,2),(3,2),(3,3)\}$

Definition 5.4.3 (toUpdate correct state)**.** *Let (s,t) be an edge with $s < t$ in the state (You, α, β) in a configuration C. This state is said to be a* toUpdate correct state *if*

$$[(\alpha,\beta) \in \{(0,1),(2,2),(3,3)\} \wedge (r_{st} = (You,\alpha) \wedge r_{ts} = (You, \beta - 1))]$$
$$\vee\ [(\alpha,\beta) \in \{(1,1),(2,1),(3,2)\} \wedge (r_{st} = (You,\alpha - 1) \wedge r_{ts} = (You, \beta))]$$

Definition 5.4.4 (Correct state)**.** *Let (s,t) be an edge with $s < t$ in the state (You, α, β) in a configuration C. This state is said to be* correct *if the state is an updated or a toUpdate correct state.*

All four previous definitions deal with an edge in which the first node has a smaller identifier than the second node. In the following, we will write (s,t) to denote such an edge. This constraint is due to the fact that nodes execute their *Increase* rule one after the other in a specific order, and a link in state $(You, 0, 1)$ can be correct while a link in state $(You, 1, 0)$ never is. When we do not make any assumption on which node has the smallest identifier in an edge, we use the notation (u,v).

We now state that a node in an edge in a correct state is only activable for *Increase* and *Write* and that an edge in a correct state will forever remain in a correct state.

Lemma 5.4.5. *Let (s,t) be an edge with $s < t$. Let C be a configuration. If (s,t) is in a correct state (You, α, β) in C then neither s nor t is eligible for Seduction$(-)$, Marriage$(-)$ or Reset in C;*

Lemma 5.4.6. *Let (s,t) be an edge with $s < t$. Let C be a configuration. If (s,t) is in the correct state (You, α, β) in C then*

- *$\forall C'$ such that $C \mapsto C'$ is a possible transition we have: (s,t) is in a correct state in C';*
- *neither s nor t is eligible for any rule in C iff (s,t) is in the updated correct state $(You, 3, 3)$ in C.*

Corollary 5.4.7. *Let (s,t) be an edge with $s < t$. Let C be a configuration. If (s,t) is in the correct state (You, α, β) in C then neither s nor t is eligible for Seduction$(-)$, Marriage$(-)$ or Reset from C.*

5.4.2 Correctness Proof

Definition 5.4.8. *A configuration is called* stable *if no node can execute a rule in this configuration.*

In particular, a configuration is stable iff all guards are false.

We now show that if our algorithm reaches a stable configuration then p-values define a maximal matching. The matching built by the algorithm is $\mathcal{M} = \{(u,v) \in E : p_u = v \wedge p_v = u\}$.

Lemma 5.4.9. *Let u be a node. In any stable configuration*
$$\forall v \in N(u) : r_{uv} = Correct\_register\_value(u,v)$$

Lemma 5.4.10. *Let u be a node. In any stable configuration:*
$$p_u = null \Rightarrow \forall v \in N(u) : p_v \notin \{null, u\}$$

Proof. Let C be a stable configuration where $p_u = null$. Consider a neighbor v. After Lemma 5.4.9, $r_{uv} = (Idle, 0)$.

If $p_v = null$ then we can assume without loss of generality that $u < v$. Then, according to Lemma 5.4.9, $r_{vu} = (Idle, 0)$. Thus u is eligible for a $Seduction(v)$ rule and C is not stable.

If $p_v = u$ and $u < v$ then $PRabandonment(v)$ holds since $r_{uv} \neq You$ and then v is eligible for a *Reset* rule and C is not stable. Finally, if $p_v = u$ and $v < u$ then, according to Lemma 5.4.9, $r_{vu} = (You, m_v)$. Then either $m_v = 0$ and so u is eligible for a $Marriage(v)$ rule, or $m_v \neq 0$ and so $PRabandonment(v)$ holds and then v is eligible for a *Reset* rule. In both cases, C is not stable. □

Lemma 5.4.11. *Let (s,t) be an edge with $s < t$. In any stable configuration, if $p_s = t$ and $p_t = s$ then edge (s,t) is in the updated correct state $(You, 3, 3)$.*

Proof. Consider the state (You, α, β) of edge (s,t) in a stable configuration C. Observe that if an edge (s,t) is in a correct state, then edge (s,t) is in the updated correct state $(You, 3, 3)$ from Lemma 5.4.6, point 2.

We now prove by contradiction that this is the only possible case. Assume that the edge (s,t) is not in a correct state. First observe that from Lemma 5.4.9, we

have $r_{st} = Correct\_register\_value(s,t)$, and node s (resp. t) is not eligible for a $Write(t)$ (resp. $Write(s)$) rule. This implies that if (s,t) is in the state (You, α, β), then $r_{st} = (You, \alpha)$ and $r_{ts} = (You, \beta)$. Thus, according to Definition 5.4.2, the only remaining possibilities for (α, β) are in set $\{(2,0),(3,0),(3,1),(0,2),(0,3),(1,3),(1,0),(1,2),(2,3)\}$. In all of these cases, s is activable for a *Reset* rule which contradicts the fact that C is a stable configuration.

□

Lemma 5.4.12. *Let (u,v) be an edge. In any stable configuration, $p_u = v$ if and only if $p_v = u$.*

Proof. Assume, by contradiction, that $p_u = v$ and $p_v \neq u$. By Lemma 5.4.10, $p_v \neq null$, thus $\exists v_1 \in N(v) : p_v = v_1$ with $v_1 \neq u$.

If $u > v$, after Lemma 5.4.9, $r_{vu}.p = Other$ (*i.e.*, $\neq You$). So the predicate $PRabandonment(u)$ holds and node u is eligible for a *Reset* rule. Thus the configuration is not stable, which is impossible. If $u < v$ and $m_v = 3$ then Lemma 5.4.9 implies that $r_{vu} = (Other, 3)$. Then the predicate $PRabandonment(u)$ holds and u is eligible for a *Reset* rule. Thus the configuration is not stable neither.

Thus $(p_u = v \wedge p_v \neq u) \Rightarrow (u < v \wedge m_v \in \{0,1,2\} \wedge \exists v_1 \in N(v) \setminus \{u\} : p_v = v_1)$.

Suppose by contradiction $p_{v_1} = v$. From Lemma 5.4.11, the edge (v_1, v) is in updated correct state $(You, 3, 3)$. This implies that $m_v = 3$ which contradicts the fact $m_v \in \{0,1,2\}$. So, $p_{v_1} \neq v$. Using the same argument for edge (u,v), we can deduce: $v < v_1 \wedge m_{v_1} \in \{0,1,2\} \wedge \exists v_2 \in N(v_1) \setminus \{v\} : p_{v_1} = v_2)$. Now we can continue the construction in the same way. We construct a path $(u, v, v_1, v_2, \ldots, v_r, \ldots)$ where $\forall i \geq 1 : p_{v_i} = v_{i+1} \wedge v_i < v_{i+1} \wedge m_{v_i} \in \{0,1,2\}$. Since the number of nodes is finite, there exists a node v_{y_1} that appears at least twice in the path. Thus, this path contains the cycle $(v_{y_1}, v_{y_2} \ldots, v_r, v_{y_1})$ and by construction, we have $v_r < v_{y_1}$ and $v_{y_1} < v_r$. This gives the contradiction. □

From these Lemmas, we deduce:

Theorem 5.4.13. *In any stable configuration, the set of edges $\mathcal{M} = \{(u,v) \in E : p_u = v \wedge p_v = u\}$ is a maximal matching.*

5.4.3 Overview of the Convergence Proof

In this section, we present a sketch of the convergence proof as it is slightly long and technical. We present the most important lemmas and theorems which are needed to bound the number of moves before convergence. The complete proof is given in the next section.

The three following lemmas put in relation the number of moves of all rules except the $Write$ rule.

Lemma 5.4.14. *Let (s,t) be an edge with $s < t$. Let $\mathcal{E}$ be an execution containing two transitions $C_0 \mapsto C_1$ and $D_0 \mapsto D_1$ with $C_1 \mapsto^* D_0$ where t executes a $Marriage(s)$ rule. Then s executes a $Seduction(t)$ rule between C_1 and D_0.*

Lemma 5.4.15. *Let u be a node. Let $\mathcal{E}$ be an execution where u executes at least two Reset moves. Let $C_0 \mapsto C_1$ and $C_2 \mapsto C_3$ be two transitions corresponding to two consecutive Reset rule executed by u. Then u executes a rule in $\{Seduction(-), Marriage(-)\}$ once between C_1 and C_2.*

Lemma 5.4.16. *Let u be a node. Let $\mathcal{E}$ be an execution where u executes at least four Increase moves. Let $C_0 \mapsto C_1$, $C_2 \mapsto C_3$, $C_4 \mapsto C_5$ and $C_6 \mapsto C_7$ be four transitions corresponding to four consecutive Increase rules executed by u. Then u executes a Reset rule once between C_0 and C_7.*

These lemmas bound the number of *Marriage* (Lemma 5.4.14), *Reset* (Lemma 5.4.15) and *Increase* (Lemma 5.4.16) in function of the number of *Seduction*. Then an upper bound on the number of $Write$ follows since one modification of the local variables of u leads u to execute at most $deg(u)$ $Write$. So, in the following, we present the sketch of the proof leading to an upper bound on the number of *Seduction*.In the Lemma below, we state that the number of $Seduction(t)$ by s is strongly connected to the number of times that t writes 3 in m_t.

Lemma 5.4.17. *Let (s,t) be an edge with $s < t$. Let $\mathcal{E}$ be an execution containing three transitions $D_0 \mapsto D_1$, $D_2 \mapsto D_3$ and $D_4 \mapsto D_5$ with $D_1 \mapsto^* D_2$ and $D_3 \mapsto^* D_4$ and where s executes a $Seduction(t)$ rule. Then there exists a transition $D \mapsto D'$ between D_2 and D_4 where t executes a $Write(s)$ rule and with in D: $p_t \neq null$ and $m_t = 3$.*

From the previous Lemma, we know that t has to write 3 in its m-variable for s to reset a previous $Seduction(t)$. The next Theorem gives conditions where the value 3 in a m-variable corresponds to a locked marriage and thus yields a situation where s cannot seduce t anymore.

Theorem 5.4.18. *Let (u,v) be an edge. Let $\mathcal{E}$ be an execution. If $\mathcal{E}$ contains one transition $A_0 \mapsto A_1$ and a configuration A_3, with $A_1 \mapsto^* A_3$ such that :*

- *in $A_0 \mapsto A_1$, u executes a Reset rule;*
- *and in A_3, $(p_u, m_u) = (v, 3)$;*

then the edge $(min(u,v), max(u,v))$ is in a correct state in A_3.

Based on these two previous results, we can bound the number of *Seduction* that can be applied by node s. We show that if s seduces t, then this seduction can be reseted once because of a "bad" initialization, and then, after another *Seduction*, this can be reseted a second time. But according to Lemma 5.4.17, this second *Reset* means that $p_t \neq null$ and $m_t = 2$. Then according to Theorem 5.4.18, at this point, t belongs to an edge in a correct state. And so s cannot seduce t anymore. Finally, we obtain that s cannot seduce a neighbor more than twice and the following theorem holds.

Theorem 5.4.19. *Let (s,t) be an edge with $s < t$. Node s can execute the $Seduction(t)$ rule at most 3 times in any execution.*

As we said before, using Lemmas 5.4.14, 5.4.15, 5.4.16, we can conclude on the time complexity of the algorithm.

Theorem 5.4.20. *The algorithm stabilizes in $O(m\Delta)$ moves where Δ is the maximum degree of G.*

5.4.4 The Complete Convergence Proof

5.4.4.1 Property of the state of an edge

Lemma 5.4.5. *Let (s,t) be an edge with $s < t$. Let C be a configuration. If (s,t) is in the correct state (You, α, β) in C then neither s nor t is eligible for $Seduction(-)$, $Marriage(-)$ or Reset in C;*

Proof. Neither s nor t are eligible for a $Seduction(-)$ or a $Marriage(-)$ rule in C, since $p_s \neq null$ and $p_t \neq null$ in C. Since $p_s = t$, then s is not eligible for a x-*Reset* if $x \neq t$. We now study the case of a t-*Reset*. $PRabandonment(s)$ is False since $r_{ts}.p =$

You. In C, we have: $(m_s, r_{ts}.m) \in \{(0,0),(0,1),(1,1),(2,1),(2,2),(3,2),(3,3)\}$ and $r_{ts}.p = You$ and $s < t$. So, $PRreset(s)$, does not hold in C. Thus, s is not eligible for $Reset$ in C. Since $p_t = s$, then t is not eligible for a x-$Reset$ if $x \neq s$. We now study the case of a s-$Reset$. $PRabandonment(t)$ is False since $r_{st}.p = You$. In C, we have: $(m_s, r_{ts}.m) \in \{(0,0),(1,0),(1,1),(1,2),(2,2),(2,3),(3,3)\}$ and $r_{st}.p = You$ and $s < t$. So, $PRreset(t)$, does not hold in C. Thus, t is not eligible for $Reset$ in C. □

Lemma 5.4.6. *Let (s,t) be an edge with $s < t$. Let C be a configuration. If (s,t) is in the correct state (You, α, β) in C then*

- *$\forall C'$ such that $C \mapsto C'$ is a possible transition we have: (s,t) is in a correct state in C';*
- *neither s nor t is eligible for any rule in C iff (s,t) is in the updated correct state $(You, 3, 3)$ in C.*

Proof. First, observe that the correct state definition only depends on the value of the following variables/registers $(p_s, p_t, m_s, m_t, r_{st}, r_{ts})$. Only s and t can write in these variables/registers. So, whatever a node $x \notin \{s,t\}$ executes in $C \mapsto C'$, this move cannot change the state of (s,t) in C'. From Lemma 5.4.5 s and t can only execute $Write$ or $Increase$, so the p-values will not change in $C \mapsto C'$ and then the state of (s,t) in C' only depends on the value of the following quadruplet $(m_s, m_t, r_{st}.m, r_{ts}.m)$ in C'. Moreover, observe that if s executes an $Increase$, then it is a t-$Increase$ since $p_s = t$ in C. In the same way, if t executes an $Increase$, then it is a s-$Increase$ since $p_t = s$ in C. Finally, observe that if s or t executes a $Write(x)$ rule, with $x \notin \{s,t\}$, then this move cannot change the state of (s,t) in C'.

Thus, we are now going to perform a case study: we check for all possible correct states in C, which rules s (resp. t) is eligible for, among the rules t-$Increase$ and $Write(t)$ (resp. s-$Increase$ and $Write(s)$). We call these rules the *relevant* rules since these are the only one that can change the state of (s,t). For all possible transitions $C \mapsto C'$ that contains at least one of these rules, we prove that (s,t) is in a correct state in C'. The two following tables present this case study.

The α and β values in C are given in Column 1. Column 2 gives the values of the quadruplet $(m_s, m_t, r_{st}.m, r_{ts}.m)$, according to the values of α and β. Columns 3 and 4 give rules s and t are eligible for.

Observe that at each line, there is at most one node among s and t that is eligible for a relevant rule in C. If this node does not perform any rule in the transition starting in C, then the state of edge (s,t) remains constant and the proof is done. Otherwise, we obtain the considered configuration in the tables, called D. Thus, in Columns 5 and 6, we give the state that is reached after s or t performs its rule. Observe that the last line of Table 1 does not contain any value in the last two columns because there is no such a D configuration since neither s nor t is eligible for a relevant rule.

In the following table, we assume (s,t) is in an updated correct state in C: ($toUpdateCS$ means toUpdate correct state)

in C				in D	
(α, β)	$(m_s, m_t, r_{st}, r_{ts})$	relevant rules eligibility for s	relevant rules eligibility for t	$(m_s, m_t, r_{st}, r_{ts})$	state of (s,t)
$(0,0)$	$(0,0,0,0)$	$\emptyset$	s-Increase	$(0,1,0,0)$	toUpdateCS $(You, 0, 1)$
$(0,1)$	$(0,1,0,1)$	t-Increase	$\emptyset$	$(1,1,0,1)$	toUpdateCS $(You, 1, 1)$
$(1,1)$	$(1,1,1,1)$	t-Increase	$\emptyset$	$(2,1,1,1)$	toUpdateCS $(You, 2, 1)$
$(2,1)$	$(2,1,2,1)$	$\emptyset$	s-Increase	$(2,2,2,1)$	toUpdateCS $(You, 2, 2)$
$(2,2)$	$(2,2,2,2)$	t-Increase	$\emptyset$	$(3,2,2,2)$	toUpdateCS $(You, 3, 2)$
$(3,2)$	$(3,2,3,2)$	$\emptyset$	s-Increase	$(3,3,3,2)$	toUpdateCS $(You, 3, 3)$
$(3,3)$	$(3,3,3,3)$	$\emptyset$	$\emptyset$	-	-

In Table 2, we assume (s,t) is in a toUpdate correct state in C: (*updatedCS* means updated correct state)

in C				in D	
(α, β)	$(m_s, m_t, r_{st}, r_{ts})$	relevant rules eligibility for s	relevant rules eligibility for t	$(m_s, m_t, r_{st}, r_{ts})$	state of (s,t)
$(0,1)$	$(0,1,0,0)$	$\emptyset$	$Write(s)$	$(0,1,0,1)$	updatedCS $(You, 0, 1)$
$(1,1)$	$(1,1,0,1)$	$Write(t)$	$\emptyset$	$(1,1,1,1)$	updatedCS $(You, 1, 1)$
$(2,1)$	$(2,1,1,1)$	$Write(t)$	$\emptyset$	$(2,1,2,1)$	updatedCS $(You, 2, 1)$
$(2,2)$	$(2,2,2,1)$	$\emptyset$	$Write(s)$	$(2,2,2,2)$	updatedCS $(You, 2, 2)$
$(3,2)$	$(3,2,2,2)$	$Write(t)$	$\emptyset$	$(3,2,3,2)$	updatedCS $(You, 3, 2)$
$(3,3)$	$(3,3,3,2)$	$\emptyset$	$Write(s)$	$(3,3,3,3)$	updatedCS $(You, 3, 3)$

□

5.4.4.2 Convergence Proof

Lemma 5.4.21. *Let u be a node. Let $\mathcal{E}$ be an execution containing two transitions $C_0 \mapsto C_1$ and $C_2 \mapsto C_3$ with $C_1 \mapsto^* C_2$ where u executes a rule.*

1. *If u executes an Increase rule during these two transitions and if $m_u = 1$ in C_3, then u executes a Reset rule between C_1 and C_2.*
2. *If u executes a Seduction$(-)$ or a Mariage$(-)$ rule during these two transitions, then u executes a Reset rule between C_1 and C_2.*

Proof. We start by proving the first point. According to the *Increase* rule, $p_u \neq null$ in C_0 and u writes 1, 2 or 3 in m_u during the transition $C_0 \mapsto C_1$. So $m_u \neq 0$ and $p_u \neq null$ in C_1. Since $m_u = 1$ in C_3, then $m_u = 0$ in C_2. Thus either u sets its m variable to 0 executing a *Reset* rule between C_1 and C_2 and the proof is done, or u executes a $Marriage(-)$ or a $Seduction(-)$ rule, let say in transition $C \mapsto C'$. Then we have $p_u = null$ in C. Since $p_u \neq null$ in C_1, then u must execute a *Reset* rule between C_1 and C.

We now prove the second point. According to both rules *Seduction* and *Marriage*, $p_u \neq \perp$ in C_1 and $p_u = \perp$ in C_2. Thus u must execute a *Reset* rule between C_1 and C_2 in order to set p_u to $\perp$. □

Lemma 5.4.22. *Let (s,t) be an edge with $s < t$. Let $\mathcal{E}$ be an execution containing two transitions $C_0 \mapsto C_1$ and $D_0 \mapsto D_1$ with $C_1 \mapsto^* D_0$ where s executes a Seduction(t) rule. We have: both s and t execute Reset between C_1 and D_0.*

Proof. From the $Seduction(t)$ rule, we have $r_{ts} = (Idle, 0)$ in C_0 and D_0. Moreover, according to Lemma 5.4.21, s executes a *Reset* rule between C_1 and D_0. Let $C_4 \mapsto C_5$ be the transition where it does for the first time.

We now study two cases: in C_4, r_{ts} is either equal or different from $(Idle, 0)$.

If it is different then there exists two transitions $C_2 \mapsto C_3$ and $C_6 \mapsto C_7$, with $C_0 \mapsto^* C_2 \mapsto^+ C_4$ and $C_4 \mapsto^* C_6 \mapsto^+ D_0$ such that: (i) in $C_2 \mapsto C_3$, t executes $Write(s)$ to set r_{ts} to a couple $\neq (Idle, 0)$ and so $p_t \neq null$ in C_2; and (ii) in $C_6 \mapsto C_7$, t executes $Write(s)$ to set r_{ts} to $(Idle, 0)$ and so $p_t = null$ in C_6. Thus, there exists a transition between C_3 and C_6 where t executes a *Reset* move.

Now, if $r_{ts} = (Idle, 0)$ in C_4. Then, $PRreset(s)$ is false and according to the *Reset* rule and in particular to the $PRabandonment(s)$ predicate, $m_s \neq 0$ in C_4. Moreover, from the $Seduction(t)$ rule, $m_s = 0$ in C_1. Thus there exists a transition $C_2 \mapsto C_3$ between C_1 and C_4 where s executes an *Increase* rule. Since s executes its first *Reset* from C_1 in $C_4 \mapsto C_5$, then $p_s = t$ from C_1 to C_4, and so $p_s = t$ in C_2. According to the *Increase* rule, in C_2 $r_{ts} = (You, 1)$. Since $r_{ts} = (Idle, 0)$ in C_0 and C_4, then there exists two transitions $B_0 \mapsto B_1$ and $B_2 \mapsto B_3$, with $C_0 \mapsto^* B_0 \mapsto^+ C_2$ and $C_2 \mapsto^* B_2 \mapsto^+ C_4$ such that: (i) in $B_0 \mapsto B_1$, t executes $Write(s)$ to set r_{ts} to $(You, 1)$ and so $(p_t, m_t) \neq (s, 1)$ in B_0; and (ii) in $B_2 \mapsto B_3$, t executes $Write(s)$ to set r_{ts} to $(Idle, 0)$ and so $(p_t, m_t) = (\perp, 0)$ in B_2. Thus, there exists a transition between B_1 and B_2 where t executes a *Reset* move. □

Lemma 5.4.23. *Let (s, t) be an edge with $s < t$. Let $\mathcal{E}$ be an execution. If $\mathcal{E}$ contains two configurations L_0 and L_1 with $L_0 \mapsto^* L_1$ and such that:*

- *t executed at least one s-rule before L_0;*
- *$r_{ts} = (Idle, 0)$ in L_0;*
- *$(p_t, m_t) = (s, 1)$ in L_1;*

then, t executes a s-Increase to set $m_s = 1$ between L_0 and L_1.

Proof. In L_0 there are two cases concerning the value of p_t: either $p_t = s$ or not. We consider the first case where $p_t = s$. Let $D_0 \mapsto D_1$ be the last transition before $\mathcal{E}$ in which t executes a s-rule. Thus we have $r_{ts} = (Idle, 0)$ and $p_t = s$ in D_1.

According to the s-rule t executes in $D_0 \mapsto D_1$, we can deduce its m_t value in D_1:

- $Write(u)$: not possible since this would imply $p_t = s$ in D_0 and then $r_{ts}.p = You$ in D_1. But $r_{ts} = (Idle, 0)$ in D_1.
- $Reset(s)$: not possible since this would imply $p_t = \perp$ in D_1.
- $Increase(s)$: not possible since this would implies in D_0: $\neg PRwriting(s)$ and $p_s = t$ and $r_{ts} = (Idle, 0)$ (since only m_s would be modified). And these three conditions are not compatible.
- $Seduction(s)$: t cannot perform this rule since $s < t$.
- $Marriage$: $(p_t, m_t) = (s, 0)$ in D_1.

Thus $(p_t, m_t) = (s, 0)$ and $r_{ts} = Idle, 0$ in D_1. Since this is the last s-rule t executes before L_0 then observe that p_t cannot be modified. Thus between D_1 and L_0 t cannot execute *Seduction*, $Marriage$ or *Reset* rules. Since t is not eligible for $Write(s)$ then r_{ts} remains equal to $(Idle, 0)$ until L_0. Observe also that since $p_t = s$ between D_1 and L_0, t cannot execute $Increase(x)$ for any node x. Thus $m_t = 0$ remains $True$ between these configurations. This implies that $m_t = 0$ in L_0. We obtain that $m_t = 0$ in L_0 and since $(p_t, m_t) = (s, 1)$ in L_1 then t must execute an $Increase(s)$ writing $m_t := 1$ between L_0 and L_1.

We now study the second case where $p_t \neq s$ in L_0. Since in L_1, $p_t = s$ by assumption, then t must execute a $Marriage(s)$ rule in some transition $C_0 \mapsto C_1$ between L_0 and L_1 and so $m_t = 0$ in C_1. Since $(p_t, m_t) = (s, 1)$ in L_1, then t must execute a s-*Increase* to set $m_t := 1$ between C_1 and L_1. Finally, the proof is done because $L_0 \mapsto^+ C_1 \mapsto^+ L_1$. □

Lemma 5.4.24. *Let (u, v) be an edge. Let $\mathcal{E}$ be an execution containing two consecutive transitions $A_1 \mapsto A_2$ and $A_3 \mapsto A_4$ with $A_2 \mapsto^* A_3$ where u executes a v-Increase rule. If node u does not execute any Reset rule between A_1 and A_4 and $(p_u, m_u) = (v, x)$ with $x \in \{0, 1\}$ in A_1, then there exists a transition $D_1 \mapsto D_2$ with $A_1 \mapsto^* D_1$ and $D_2 \mapsto^* A_4$ such that*

- *$r_{uv} = (You, x)$ between A_1 and D_1;*
- *$r_{uv} = (You, x + 1)$ between D_2 and A_2;*
- *$(p_u, m_u) = (v, x + 1)$ between D_2 and A_3;*

Proof. Since u does not perform any *Reset* from A_1 to A_4 by assumption, then p_u remains constant from A_1 to A_4. Since u performs a v-*Increase* rule in $A_1 \mapsto A_2$, $p_u = v$ between A_2 and A_3.

Since $(p_u, m_u) = (v, x)$ in A_1, we also have $r_{uv} = (You, x)$ otherwise u would have executed a $Write(v)$ instead of an Increase in $A_1 \mapsto A_2$.

According to the v-*Increase* rules, $r_{uv} = (You, x)$ in A_1, and $(p_u, m_u) = (v, x+1)$ in A_2. Observe that between A_2 and A_3, we have by assumption: u does not perform any *Reset*, any v-Increase, *Seduction* nor *Marriage* (since $p_u = v$). Thus, the only rule u can perform between A_2 and A_3 are $Write(-)$. Thus, (p_u, m_u) remains constant between A_2 and A_3.

Since u executes a v-*Increase* rule in $A_3 \mapsto A_4$, $r_{uv} = (You, x + 1)$ in A_3, according to the v-*Increase* rules. Thus u updates its registers by executing $Write(v)$ between A_2 and A_3.

Let $D_1 \mapsto D_2$ be the first transition with $A_2 \mapsto^* D_1$ and $D_2 \mapsto^* A_3$ in which u executes a $Write(v)$ rule. Thus, we have $r_{uv} = (You, x + 1)$ in D_2.

By definition of transition $D_1 \mapsto D_2$, r_{uv} remains constant between A_1 and D_1. So, since $r_{uv} = (You, x)$ in A_1, $r_{uv} = (You, x)$ between A_1 and D_1 .

Since u can only execute $Write(-)$ rule between $A_2 \mapsto^* A_3$, this implies that (p_u, m_u) remains constant between A_2 and A_3. Thus, $(p_u, m_u) = (v, x + 1)$ between D_2 and A_3. Moreover, since $r_{uv} = (You, x + 1)$ in D_2, $r_{uv} = (You, x + 1)$ between D_2 and A_2. □

Lemma 5.4.25. *Let (s, t) be an edge with $s < t$. Let $\mathcal{E}$ be an execution. If $\mathcal{E}$ contains two transitions $A_0 \mapsto A_1$, $A_2 \mapsto A_3$ and a configuration A_4, with $A_1 \mapsto^* A_2$ and $A_3 \mapsto^* A_4$ and such that :*

- *in $A_0 \mapsto A_1$, s executes a Reset rule;*
- *and in A_3, $(p_s, m_s) = (t, 3)$;*

then the edge (s, t) is in a correct state in A_3.

Proof. Let $C_0 \mapsto C_1$ be the last *Reset* executed by s between A_0 and A_3 (we can have $C_0 = A_2$). In C_1, $p_s = null$ and in A_3, $p_s = t$, with $t > s$. Thus s must execute a $Seduction(t)$ rule between C_1 and A_3 to set $p_t = s$.

Let $C_2 \mapsto C_3$ be the last $Seduction(-)$ rule executed by s between C_1 and A_3. Since s does not perform any *Reset* from C_3 to A_3 by construction, then p_s remains constant from C_3 to A_3. $p_t = s$ in A_3, thus $p_s = t$ in C_3 and so s performs a $Seduction(t)$ rule in $C_2 \mapsto C_3$.

Since s performs a $Seduction(t)$ rule in $C_2 \mapsto C_3$, we have $r_{ts} = (Idle, 0)$ in C_2.

Observe that between C_3 and A_3, we have by construction: s can perform between C_3 and A_3 are $Write(-)$ and t-*Increase* (since $p_t = s$). So the value of m_s can only change by a $+1$ incrementation between C_3 and A_3 . In C_3, $m_s = 0$ and in A_3, $m_s = 3$. Thus, beside the $Write$ rule, s executed exactly three t-*Increase* between C_3 and A_3.

Let $C_4 \mapsto C_5$, $C_6 \mapsto C_7$ and $C_8 \mapsto C_9$ be these three *t-Increase* executed by s, with $C_5 \mapsto^* C_6$ and $C_7 \mapsto^* C_8$.

In $C_4 \mapsto C_5$, s sets $m_s = 1$. According to the *Increase* rule, we have: in C_4, $r_{ts} = (You, 1)$. Since $r_{ts} = (Idle, 0)$ in C_2, t performs a $Write(s)$ rule between C_2 and C_4 to set $r_{st} = (You, 2)$. Let $W_0 \mapsto W_1$ be this transition. We thus have $(p_t, m_t) = (s, 2)$ in W_0 with $C_2 \mapsto^+ W_1$ and $W_1 \mapsto^* C_4$.

In $C_6 \mapsto C_7$, s sets $m_s = 2$, and in $C_8 \mapsto C_9$, s sets $m_s = 3$. According to the *Increase* rule, we have: in C_6, $r_{ts} = (You, 1)$ and in C_8, $r_{ts} = (You, 3)$. Thus, t performs a $Write(s)$ rule between C_6 and C_8 to set $r_{ts} = (You, 3)$.

Let $W_2 \mapsto W_3$ be this transition. We thus have $(p_t, m_t) = (s, 3)$ in W_2 with $C_6 \mapsto^* W_2$ and $W_3 \mapsto^* C_8$. In order to increase its m-variable, t must execute a *s-Increase* rule to set $m_t = 3$.

Let $D_0 \mapsto D_1$ be the last transition where t executes a *s-Increase* rule to set $m_t = 3$. between W_1 and W_2. We will prove that the edge (s, t) is in the updated correct state $(You, 2, 3)$ in D_0. First, observe that since $C_2 \mapsto^+ W_1$ and $D_1 \mapsto^* W_2$, we can deduce that $C_2 \mapsto^* D_0 \mapsto D_1 \mapsto^* W_2$.

Using (twice) Lemma 5.4.24, we can deduce that there exist two transitions $A_0 \mapsto A_1$ and $B_0 \mapsto B_1$ such that

— $r_{st} = (You, 0)$ between C_4 and A_1;

— $r_{st} = (You, 1)$ between A_2 and C_6, and between C_6 and B_0;

— $r_{st} = (You, 2)$ between B_0 and C_7;

Since $r_{st} = (You, 2)$ in D_0, t executes a *s-Increase* in $D_0 \mapsto D_1$ between B_0 and C_7. Using Lemma 5.4.24, we have $(p_s, m_s) = (t, 2)$ between B_0 and C_7;

We finally obtain for D_0: $(p_s, m_s) = (s, 2)$, $(p_t, m_t) = (s, 2)$, $r_{st} = (You, 2)$ and $r_{ts} = (You, 2)$. Thus the edge (s, t) is in the updated correct state $(You, 2, 2)$ in D_0. As $D_0 \mapsto^+ C_8 \mapsto^+ A_3$ and according to Lemma 5.4.6, the edge (s, t) is in a correct state in A_3. □

Lemma 5.4.26. *Let (s, t) be an edge with $s < t$. Let $\mathcal{E}$ be an execution. If $\mathcal{E}$ contains two transitions $A_0 \mapsto A_1$, $A_2 \mapsto A_3$ and a configuration A_4, with $A_1 \mapsto^* A_2$ and $A_3 \mapsto^* A_4$ and such that :*

— *in $A_0 \mapsto A_1$, t executes a s-rule;*

— *in $A_2 \mapsto A_3$, s executes a Reset rule;*

— *and in A_4, $(p_s, m_s) = (t, 1)$;*

then the edge (s, t) is in a correct state in A_4.

Proof. Let $C_0 \mapsto C_1$ be the last *Reset* executed by s between A_2 and A_4 (we can have $C_0 = A_2$). In C_1, $p_s = null$ and in A_4, $p_s = t$, with $s < t$. Thus s must execute a $Seduction(t)$ rule between C_1 and A_4 to set $p_s = t$

Let $C_2 \mapsto C_3$ be the last $Seduction(-)$ rule executed by s between C_1 and A_4. Since s does not perform any *Reset* from C_3 to A_4 by construction, then p_s remains constant from C_3 to A_4. $p_s = t$ in A_4, thus $p_s = t$ in C_3 and so s performs a $Seduction(t)$ rule in $C_2 \mapsto C_3$.

Observe that between C_3 and A_4, we have by construction: s does not perform any *Reset* nor *Seduction* and $p_s = t$. Thus, s cannot perform any *Marriage* rule neither. So, the only rule s can perform between C_3 and A_4 are $Write(-)$ and *t-Increase* (since $p_s = t$). So the value of m_s can only change by a $+1$ incrementation between C_3 and A_4 . In C_3, $m_s = 0$ and in A_4, $m_s = 1$. Thus, beside the $Write$ rule, s executed exactly one *t-Increase* between C_3 and A_4.

Let $C_4 \mapsto C_5$ be this *t-Increase* executed by s. In $C_4 \mapsto C_5$, s sets $m_s = 1$. So, in C_4, $(p_s, m_s) = (t, 0)$ and then $r_{ts} = (You, 1)$. Moreover, according to the

Seduction(t) rule, in C_2, $r_{ts} = (Idle, 0)$. So there exists a transition $D_0 \mapsto D_1$ between C_2 and C_4 where t executes a $Write(s)$ rule to set $r_{ts} = (You, 1)$. Thus, in D_0, $(p_t, m_t) = (s, 1)$.

From Lemma 5.4.23 – by setting $C_2 = L_0$ and $D_0 = L_1$ and considering that t executed a s-rule in $A_0 \mapsto A_1$ – there exists a transition $F_0 \mapsto F_1$ between C_2 and D_0 where t executes a *s-Increase* rule to set $m_t = 1$.

We are now going to prove that the edge (s, t) is in the updated correct state $(You, 0, 0)$ in F_0.

Since t executes a *s-Increase* in $F_0 \mapsto F_1$ with $m_t = 1$ in F_1, then in F_0 we have: $(p_t, m_t) = (s, 0)$ and, according to the *Increase* rule, $r_{st} = (You, 0)$. We also have $r_{ts} = (You, 0)$ otherwise t would have perform a $Write(s)$ instead of an *s-Increase* in $F_0 \mapsto F_1$.

Recall, that $C_2 \mapsto^* F_0 \mapsto^+ D_0$. We already know that $r_{ts} = (Idle, 0)$ in C_2, so $C_3 \mapsto^* F_0 \mapsto^+ D_0$.

Moreover, we know that between C_3 and C_4, s does not perform any rule but some $Write(-)$ rule. So (p_s, m_s) remains constant between C_3 and C_4. s executes a *Seduction*(t) rule in $C_2 \mapsto C_3$, so $(p_s, m_s) = (t, 0)$ in C_3. Moreover, $C_3 \mapsto^* D_0 \mapsto^+ C_4$, so $(p_s, m_s) = (t, 0)$ in D_0.

We finally obtain for F_0: $(p_t, m_t) = (s, 0)$, $(p_s, m_s) = (t, 0)$, $r_{ts} = (You, 0)$ and $r_{st} = (You, 0)$. Thus the edge (s, t) is in the updated correct state $(You, 0, 0)$ in F_0. As $F_0 \mapsto^+ C_4 \mapsto^+ A_4$ and according to Lemma 5.4.6, the edge (s, t) is in a correct state in A_4.

□

Lemma 5.4.27. *Let (s, t) be an edge with $s < t$. Let $\mathcal{E}$ be an execution. If $\mathcal{E}$ contains two transitions $A_0 \mapsto A_1$, $A_2 \mapsto A_3$ and a configuration A_4, with $A_1 \mapsto^* A_2$ and $A_3 \mapsto^* A_4$ and such that :*

- *in $A_0 \mapsto A_1$, t executes a Reset rule;*
- *and in A_3, $(p_t, m_t) = (s, 3)$;*

then the edge (s, t) is in a correct state in A_3.

Proof. Let $C_0 \mapsto C_1$ be the last *Reset* executed by t between A_0 and A_3 (we can have $C_0 = A_2$). In C_1, $p_t = null$ and in A_3, $p_t = s$, with $t > s$. Thus t must execute a $Marriage(s)$ rule between C_1 and A_3 to set $p_t = s$.

Let $C_2 \mapsto C_3$ be the last $Marriage(-)$ rule executed by t between C_1 and A_3. Since t does not perform any *Reset* from C_3 to A_3 by construction, then p_t remains constant from C_3 to A_3. $p_t = s$ in A_3, thus $p_t = s$ in C_3 and so t performs a $Marriage(s)$ rule in $C_2 \mapsto C_3$.

Observe that between C_3 and A_3, we have by construction: t does not perform any *Reset* nor *Marriage* and $p_t = s$. Thus, t cannot perform any *Seduction* rule neither. So, the only rule t can perform between C_3 and A_3 are $Write(-)$ and *s-Increase* (since $p_t = s$). So the value of m_t can only change by a $+1$ incrementation between C_3 and A_3 . In C_3, $m_t = 0$ and in A_3, $m_t = 3$. Thus, beside the $Write$ rule, t executed exactly three *s-Increase* between C_3 and A_3.

Let $C_4 \mapsto C_5$, $C_6 \mapsto C_7$ and $C_8 \mapsto C_9$ be these three *s-Increase* executed by t, with $C_5 \mapsto^* C_6$ and $C_7 \mapsto^* C_8$.

In $C_4 \mapsto C_5$, t sets $m_t = 1$ and in $C_6 \mapsto C_7$, t sets $m_t = 2$. So in C_4, $m_t = 0$ and in C_6, $m_t = 1$. According to the *Increase* rule, we have: in C_4, $r_{st} = (You, 0)$ and in C_6, $r_{st} = (You, 2)$. Thus, s performs a $Write(t)$ rule between C_4 and C_6 to set $r_{st} = (You, 2)$.

Let $W_0 \mapsto W_1$ be this transition. We thus have $(p_s, m_s) = (t, 2)$ in W_0 with $C_4 \mapsto^* W_0$ and $W_1 \mapsto^* C_6$.

In $C_6 \mapsto C_7$, t sets $m_t = 2$ and in $C_8 \mapsto C_9$, t sets $m_t = 3$. According to the *Increase* rule, we have: in C_6, $r_{st} = (You, 2)$ and in C_8, $r_{st} = (You, 3)$. Thus, s performs a $Write(t)$ rule between C_6 and C_8 to set $r_{st} = (You, 3)$.

Let $W_2 \mapsto W_3$ be this transition. We thus have $(p_s, m_s) = (t, 3)$ in D_4 with $C_6 \mapsto^* W_2$ and $W_3 \mapsto^* C_8$.

To sum, we have a relationship among these following configurations:

$$C_4 \mapsto^* W_0 \mapsto W_1 \mapsto^* C_6 \mapsto^* W_2 \mapsto W_3 \mapsto^* C_8.$$

We focus which rules s executes between W_1 and W_2. In order to increase its m-variable, s must execute a *t-Increase* rule to set $m_s = 3$. Let $D_0 \mapsto D_1$ be a transition where s executes a *t-Increase* rule between W_1 and W_2.

We will prove that the edge (s, t) is in the updated correct state $(You, 2, 2)$ in D_0.

Since s executes a *t-Increase* in $D_0 \mapsto D_1$ with $m_s = 3$ in D_1, then in D_0 we have: $(p_s, m_s) = (t, 2)$ and, according to the *Increase* rule, $r_{ts} = (You, 2)$.

Since transition $D_0 \mapsto D_1$ is between W_1 and W_2, transition $D_0 \mapsto D_1$ is between C_4 and C_7. Using (twice) Lemma 5.4.24, we can deduce that there exist two transitions $A_0 \mapsto A_1$ and $B_0 \mapsto B_1$ such that

— $r_{ts} = (You, 0)$ between C_4 and A_1;

— $r_{ts} = (You, 1)$ between A_2 and C_6, and between C_6 and B_0

— $r_{ts} = (You, 2)$ between B_0 and C_7;

Since $r_{ts} = (You, 2)$ in D_0, s executes a *t-Increase* in $D_0 \mapsto D_1$ between B_0 and C_7. Using Lemma 5.4.24, we have $(p_t, m_t) = (s, 2)$ between B_0 and C_7;

We finally obtain for D_0: $(p_s, m_s) = (s, 2)$, $(p_t, m_t) = (s, 2)$, $r_{st} = (You, 2)$ and $r_{ts} = (You, 2)$. Thus the edge (s, t) is in the updated correct state $(You, 2, 2)$ in D_0. As $D_0 \mapsto^+ C_8 \mapsto^+ A_3$ and according to Lemma 5.4.6, the edge (s, t) is in a correct state in A_3. □

Theorem 5.4.18. *Let (u, v) be an edge. Let $\mathcal{E}$ be an execution. If $\mathcal{E}$ contains one transition $A_0 \mapsto A_1$ and a configuration A_3, with $A_1 \mapsto^* A_3$ such that :*

— *in $A_0 \mapsto A_1$, u executes a Reset rule;*

— *and in A_3, $(p_u, m_u) = (v, 3)$;*

then the edge $(min(u, v), max(u, v))$ is in a correct state in A_3.

Proof. If $u < v$, we conclude immediately using Lemma 5.4.25. Otherwise, we can apply Lemma 5.4.27.

□

Lemma 5.4.17. *Let (s, t) be an edge with $s < t$. Let $\mathcal{E}$ be an execution containing three transitions $D_0 \mapsto D_1$, $D_2 \mapsto D_3$ and $D_4 \mapsto D_5$ with $D_1 \mapsto^* D_2$ and $D_3 \mapsto^* D_4$ and where s executes a Seduction(t) rule. Then there exists a transition $D \mapsto D'$ between D_2 and D_4 where t executes a $Write(s)$ rule and with in D: $p_t \neq null$ and $m_t = 3$.*

Proof. According to Lemma 5.4.21.(2), s executes a *Reset* between D_1 and D_2. Let $C_0 \mapsto C_1$ be the transition where s does so for the first time. So s executes a *t-Reset* rule in $C_0 \mapsto C_1$. Moreover, according to Lemma 5.4.22, t executes a *Reset* between D_1 and D_2.

Observe now that the execution starting in configuration D_2 reaches all the assumptions made in Lemmas 5.4.26. Indeed, before D_2, t executes a s-rule and then s executes a *Reset* rule.

According to Lemma 5.4.22, s executes a *Reset* rule between D_3 and D_4. Let $C_2 \mapsto C_3$ be the transition where s does so for the first time. So s executes a *t-Reset* in $C_2 \mapsto C_3$. In C_2, either $m_s > 0$ or $m_s = 0$.

If $m_s > 0$ in C_2 and since $m_s = 0$ in D_3 by the *Seduction* rule, then s performs $m_s := 1$ between D_3 and C_2, let say in transition $B \mapsto B'$. In this transition, s executes an *Increase* rule. Recall that $p_s = t$ in D_3 since s executes a *Seduction*(t) rule, and that s does not execute any *Reset* between D_3 and C_2. Thus $p_s = t$ in B and so $(p_s, m_s) = (t, 1)$ in B'. By Lemma 5.4.26, the edge (s, t) is in a correct state in B'. Thus, by Corollary 5.4.7, from this configuration, s cannot execute any *Reset*, which contradict the fact that it does in $C_2 \mapsto C_3$. Then $m_s = 0$ in C_2.

Since $(p_s, m_s) = (t, 0)$ in C_2 with $s < t$ then, according to the *Reset* rule, $r_{ts}.m = 3$ in C_2. Moreover, since $r_{ts}.m = 0$ in D_2 by the *Seduction*(t) rule, then t executes a *Write*(s) rule between D_2 and C_2. This is the transition $D \mapsto D'$ and we have $m_t = 3$ and $p_t \neq null$ in D. This conclude the proof. □

Theorem 5.4.19. *Let (s, t) be an edge with $s < t$. Node s can execute the Seduction(t) rule at most 3 times in any execution..*

Proof. By contradiction. Let $\mathcal{E}$ be an execution where s executes a *Seduction*(t) rule at least 4 times. We will prove that such an execution is not possible since after the 3^{rd} *Seduction*(t) execution, s cannot perform any other *Seduction*(t) rule.

For $1 \leq i \leq 4$, let $A_i \mapsto B_i$ be the transition where s executes its i^{th} *Seduction*(t) rule in $\mathcal{E}$.

According to Lemma 5.4.17, between each couple of configurations (B_j, A_{j+1}) where $1 \leq j \leq 3$, there exists a transition $C_j \mapsto D_j$ where t executes a *Write*(s) rule and with $p_t \neq null$ and $m_t = 3$ in C_j.

Since s executes two *Seduction*(t) rules in $A_2 \mapsto B_2$ and $A_3 \mapsto B_3$ and by Lemma 5.4.22, t executes a *Reset* between B_2 and A_3. Recall that $m_t = 3$ in C_2 with $B_2 \mapsto^+ C_2 \mapsto^* A_3$. Thus by Theorem 5.4.18, the edge $(min(t, p_t), max(t, p_t))$ is in a correct state in C_2. From Corollary 5.4.7, from this configuration t cannot execute any *Reset*. However, since s executes two *Seduction*(t) rules in $A_3 \mapsto B_3$ and $A_4 \mapsto B_4$ and by Lemma 5.4.22, t executes a *Reset* between B_3 and A_4 which leads the contradiction.

□

Lemma 5.4.14. *Let (s, t) be an edge with $s < t$. Let $\mathcal{E}$ be an execution containing two transitions $C_0 \mapsto C_1$ and $D_0 \mapsto D_1$ with $C_1 \mapsto^* D_0$ where t executes a Marriage(s) rule. Then s executes a Seduction(t) rule between C_1 and D_0.*

Proof. From the *Marriage*(s) rule, we have $r_{st} = (You, 0)$ in C_0 and D_0. Moreover, according to Lemma 5.4.21, t executes a *Reset* rule between C_1 and D_0. Let $C_4 \mapsto C_5$ be the transition where it does for the first time. Then $p_t = s$ from C_1 to C_4.

In the first step of the proof, we will prove that there exists two configurations γ and γ' between C_0 and D_0 and with $\gamma \mapsto^+ \gamma'$, such that $(p_s, m_s) \neq (t, 0)$ in γ and $(p_s, m_s) = (t, 0)$ in γ'. Then, we will prove that s must execute a *Seduction*(t) rule between γ and γ'. For the first step of the proof, we study two cases according to the value of r_{st} in C_4.

First, assume that $r_{st} \neq (You, 0)$ in C_4. So, there exists two transitions $C_2 \mapsto C_3$ and $C_6 \mapsto C_7$, with $C_0 \mapsto^* C_2 \mapsto^+ C_4$ and $C_4 \mapsto^* C_6 \mapsto^+ D_0$ such that: (i) in $C_2 \mapsto C_3$, s executes *Write*(t) to set r_{st} to a couple $\neq (You, 0)$ and so $(p_s, m_s) \neq (t, 0)$ in C_2; and (ii) in $C_6 \mapsto C_7$, s executes *Write*(t) to set r_{st} to $(You, 0)$ and so $(p_s, m_s) = (t, 0)$ in C_6. We then have $\gamma = C_2$ and $\gamma' = C_6$.

Second, if $r_{st} = (You, 0)$ in C_4, then $PRabandonment(t)$ is false and according to the *Reset* rule and in particular to the $PRreset(t)$ predicate, $m_t \geq 2$ in C_4 (it is the only possible case as $s < t$ and $r_{st}.m = 0$). Since $m_t = 0 < 2$ in C_1, there exists a transition $C_2 \mapsto C_3$ between C_1 and C_4 where t executes an *Increase* rule in order to write 2 in its m-variable. Since $p_t = s$ from C_1 to C_4, then $p_t = s$ in C_2 and then, according to the *Increase* rule, in C_2 $r_{st} = (You, 2)$. Since $r_{st} = (You, 0)$ in C_0 and C_4, then there exists two transitions $B_0 \mapsto B_1$ and $B_2 \mapsto B_3$, with $C_0 \mapsto^* B_0 \mapsto^+ C_2$ and $C_2 \mapsto^* B_1 \mapsto^+ C_4$ such that: (i) in $B_0 \mapsto B_1$, s executes $Write(t)$ to set r_{st} to $(You, 2)$ and so $(p_s, m_s) = (t, 2)$ in B_0; and (ii) in $B_2 \mapsto B_3$, s executes $Write(t)$ to set r_{st} to $(You, 0)$ and so $(p_s, m_s) = (t, 0)$ in B_2. We then have $\gamma = B_0$ and $\gamma' = B_2$.

We now prove the second step of the proof, that is s must execute a $Seduction(t)$ rule between γ and γ'. If $p_s \neq t$ in γ, and since $s < t$, then s must execute a $Seduction(t)$ rule between γ and γ' in order to set t in its p-variable. Otherwise, we have $p_s = t \wedge m_s \neq 0$ in γ. Let us assume that s does not perform any $Seduction(t)$ rule between γ and γ'. Thus, the only two rules to write 0 in its m-variable are $Marriage(-)$ and *Reset*. Since $s < t$, s cannot execute a $Marriage(t)$ rule, thus after writing 0 in its m variable, $p_s \neq t$ and we go back to the case 1, leading to the conclusion that s must perform a $Seduction(t)$ rule in order to write t in its p-variable. Finally, in any cases, s must execute a $Seduction(t)$ rule between γ and γ'. □

Lemma 5.4.15. *Let u be a node. Let $\mathcal{E}$ be an execution where u executes at least two Reset moves. Let $C_0 \mapsto C_1$ and $C_2 \mapsto C_3$ be two transitions corresponding to two consecutive Reset rule executed by u. Then u executes a rule in $\{Seduction(-), Mariage(-)\}$ once between C_1 and C_2.*

Proof. According to the *Reset* rule, $p_u = null$ in C_1 and $p_u \neq null$ in C_2. so, u has to execute a rule between C_1 and C_2 to set a neighbor identifier in its p-variable. There are only two rules doing that: the *Seduction* and the *Marriage* rules. Thus, u executes such a rule at least once between C_1 and C_2. Now, assume that node u executes such a rule more than once between C_1 and C_2. Then, from Lemma 5.4.21, u executes a *Reset* rule between C_1 and C_2. This contradicts the fact that node u does not execute any *Reset* rule between C_1 and C_2. Thus, the lemma holds. □

Lemma 5.4.16. *Let u be a node. Let $\mathcal{E}$ be an execution where u executes at least four Increase moves. Let $C_0 \mapsto C_1$, $C_2 \mapsto C_3$, $C_4 \mapsto C_5$ and $C_6 \mapsto C_7$ be four transitions corresponding to four consecutive Increase rules executed by u. Then u executes a Reset rule once between C_0 and C_7.*

Proof. We now prove this lemma, by contradiction. Let us assume node u does not executes a *Reset* rule a between C_0 and C_7.

According to the *Increase* rule, $m_u \in \{1, 2, 3\}$ in C_1, C_3 and C_5. According to Lemma 5.4.21, if $m_u = 1$ in C_3 (resp. C_5) then u executes a *Reset* rule between C_1 and C_2 (resp. C_3 and C_4). This contradicts the fact that node u does not execute any *Reset* rule between C_0 and C_7. Thus, $m_u = 2$ or $m_u = 3$ in C_3 and C_5.

Assume that $m_u = 3$ in C_3. Since $m_u = \{1, 2, 3\}$ in C_5, this implies that $m_u < 3$ in C_4. There is only one way to decrease the value of an m variable between C_3 and C_4: to write 0. Thus u has to execute a *Reset* rule between C_3 and C_4 in order to decrease the value m_u. However this contradicts the fact that node u does not execute any *Reset* rule between C_0 and C_5.

Assume that $m_u = 2$ in C_3. Since node u does not execute any *Reset* rule between C_3 and C_4, m_u remains constant between C_3 and C_4. So $m_u = 2$ in C_4 and $m_u = 3$ in C_5. Using the same argument as previously, this contradicts the fact

that node u does not execute any *Reset* rule between C_5 and C_7. So the proof is completed.

□

Theorem 5.4.20. *The algorithm stabilizes in $O(m\Delta)$ moves where Δ is the maximum degree of G.*

Proof. Let $deg(u)$ be the degree of node u. Let (s,t) be an edge with $s < t$. By Theorem 5.4.19, node s can execute the $Seduction(t)$ rule $O(1)$ times. By Lemma 5.4.14, between two executions of the $Marriage(s)$ rule by node t, node s must execute a $Seduction(t)$ rule. This implies that t can execute $O(1)$ $Marriage(s)$ rules. In total, a node u can $O(1)$ $Seduction(-)$ and $Marriage(-)$ rules per neighbor, which gives a $\mathcal{O}(deg(u))$ total number of these rules, per node u.

Let now u be a node. By Lemma 5.4.15, between two executions of the *Reset* rule by node u, it must execute a $Marriage(-)$ or $Seduction(-)$ rule. Since we proved that it can do so $\mathcal{O}(deg(u))$ times, then it can execute the *Reset* rule $\mathcal{O}(\Delta^2)$ times as well. Now by Lemma 5.4.16, between three executions of the *Increase* rule, node u must execute the *Reset* rule. As a consequence, u can execute the *Increase* rule $\mathcal{O}(deg(u))$ times. Altogether, a node can execute at most $\mathcal{O}(deg(u))$ times $Seduction(-)$, $Marriage(-)$, *Increase* and *Reset* rules. Let's call such rules high level rules. So, all nodes can, in total, execute $O(m)$ high level rules (since the sum of the degrees of the vertices is twice the number of edges).

Each time it executes such a high level rule, a node u may execute a $Write(-)$ rule $\mathcal{O}(deg(u))$ times to update all its registers. If it does not execute any high level rule, a node can execute at most $\mathcal{O}(deg(u))$ *Write* rules. Finally, nodes can execute at most $O(m)$ high level rules and $O(m\Delta)$ *Write* rules. □

5.4.5 Conclusion

The link-register model, by introducing a delay between the time an action is taken and the time an adjacent node is informed of the resulting modification, allows to study the asynchronism induced by communications in distributed systems. Read/write atomicity is the most restrictive model in this category, by allowing only node-to-node communications. The adversarial distributed daemon tops it off by being able to postpone a communication arbitrarily long (precisely, until no other move can be taken by the algorithm).

In this chapter, we propose the first algorithm to solve the matching problem in this setting. This algorithm is self-stabilizing, and takes at worst $O(m\Delta)$ moves before converging from the worst possible initialization, with the worst possible scheduling of communications. We prove this algorithm and compute its complexity by bounding the number of times a marriage process can be interrupted, either due to a bad initialization, or to another marriage of one of the two processes.

In this algorithm, we impose that a register r_{uv} is up to date before allowing u to take any action but *Write* regarding v. This restriction is similar to an acknowledgement from v that it has received the new value and will use it from now on. We are working to remove this condition that introduces some synchronism in the model. The link-register model is a step toward message-passing models, and we are aiming at studying the effects of these various hypotheses on the communications between nodes on the implementation of the algorithm presented in this chapter.

Part III

Lattice polytopes

Chapter 6

New bounds for the diameter of lattice polytopes

In this chapter we describe some of our contributions regarding the characterization of the diameter of lattice convex polytopes. These results have appeared in a journal paper [44] and a in forthcoming book chapter.

6.1 Introduction

Polytopes have drawn interest from ancient ages. Stone models of polytopes dated back to the neolithic have been found in Scotland, even though their function remains unknown as of today [56]. The Etruscans were the first to use cubic and dodecahedral dices for gambling purposes. It seems also that The Egyptians were aware of the regular tetrahedron and octahedron. The Greeks were probably the first to mathematically study polytopes and regular solids. Among others, Pythagoras is considered the inventor of the regular dodecahedron, Euclid studied regular polyhedra and the Platonic Solids in his book XIII of The Elements [7] and Archimedes gave the list of the 13 semi-regular polyhedra of the first kind. These objects have also been considered in higher dimensions and Schläfli is considered to be the first mathematician to have studied such objects, around 1850. Many natural questions arise regarding properties of these objects. For instance, crystallographers have been studying their symmetries and have also been interested in tiling problems.

In this chapter, we want to study the behavior of one parameter, the diameter, of some restricted class of polytopes, (convex polytopes with integer coordinates) with respect to their dimension and grid embedding size. Finding a good bound on the maximal edge-diameter of a polytope has recently received a lot of attention. A lot of research has been dedicated to bounding the diameter of polytopes in terms of their dimension and the number of their facets. It is not only a natural question of discrete geometry, but also historically closely connected with the theory of the simplex method. One central question related to the diameter of polytopes is the Hirsch conjecture which states that the edge-graph of a n-facet d-dimensional polytope has diameter no more than $n - d$. The conjecture was first put forth in a letter by Hirsch to Dantzig in 1957 [40] and was motivated by the analysis of the simplex method in linear programming, as the diameter of a polytope provides a lower bound on the number of steps needed by the simplex method. The Hirsch conjecture was proven for $d < 4$ and for various special cases. It is now known to be false in general, as shown by Santos' counterexample [124]. Allamigeon, Benchimol, Gaubert, and Joswig have also exhibited a counterexample to a continuous analogue of the polynomial Hirsch conjecture [3]. Bounds have also been proved. Kalai and

Kleitman's upper bound for the diameter of polytopes [91] was strengthened by Todd [136], and then by Sukegawa [131].

More closely related to the context of this thesis, bounds for the diameter have also been studied in the case in which the considered polytopes are lattice. One of the first such result is the upper bound of Kleinschmidt and Onn for the diameter of lattice polytopes [94]. It was strengthened by Del Pia and Michini [43], and then by Deza and Pournin [45]. Additionally, new exact values have been computationaly found by Chadder And Deza [26]. These results are discussed in a more detailed manner in Section 6.6.

In the next section, we introduce basic definitions. In Sections 6.3 and 6.4 we highlights connections between zonotopes, Minkowski sums and hyperplane arrangement. In Section 6.5 we introduce Primitive Zonotopes and some of their properties. In Section 6.6 we derive lower bounds for the diameter of lattice polytopes. Finally in Section 6.7 we describe some small instances of Primitive Zonotopes.

6.2 Basic notions

We introduce and define notions about *lattice convex polytopes*, which are the main objects of interest in this thesis chapter. A bounded d-dimensional convex polytope is the convex hull of a finite set of point in $\mathcal{R}^d$. When restricted to points in $\mathbb{Z}^d$, the polytope is said to be *lattice*. A set K is *convex* if, for each pair of distinct points x, y in K, the closed segment with endpoints x and y is contained within K. A polytope is *full-dimensional* if it is a full dimensional object in $\mathbb{R}^d$. A convex polytope can also be represented as the intersection of a finite number of halfspace. Assuming, that there are m halfspaces defining polytope P, a concise way to describe P is $Ax \leq b$ where A is an $m \times d$ matrix, x is a $m \times 1$ matrix and b is an $m \times 1$ matrix representing the inequalities of the m halfspaces.

A *face* of a convex polytope is any intersection of the polytope with a halfspace such that none of the interior points of the polytope lie on the boundary of the halfspaces. If a polytope is d-dimensional, its facets are its $(d-1)$-dimensional faces, its vertices are its 0-dimensional faces, its edges are its 1-dimensional faces. The polytope is its d-dimensional face and the null polytope (the empty set) is considered to be its (-1)-dimensional face. See Figure 6.1 for an illustration.

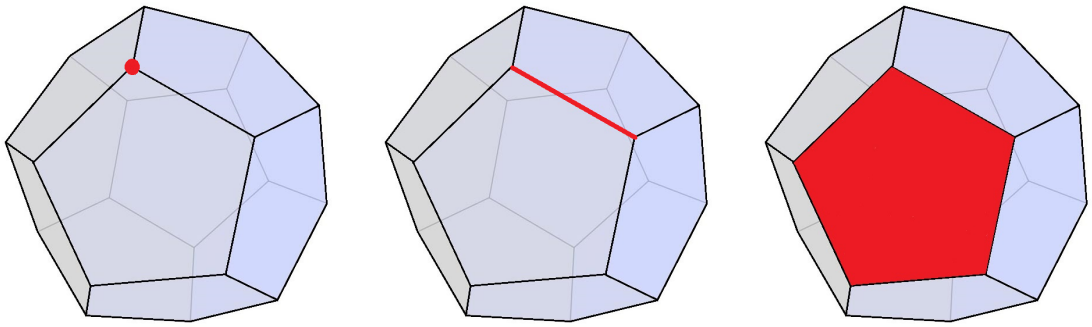


Figure 6.1 – A three dimensional polytope. In red, a vertex or 0-dimensional face is represented on the left, an edge or 1-dimensional face is drawn in the middle picture and a face or 2-dimensional face is drawn on the right.

A *lattice* is a partially ordered set in which every two elements have a join (least upper bound) and a meet (greatest lower bound). The faces of a convex polytope

form a lattice called its *face lattice*, where the partial ordering is by set containment of faces. See Figure 6.2 for a small example. The given definition of a face ensures that every pair of faces has a join and a meet in this lattice. The whole polytope is the unique maximum element of the lattice, and the empty set is the unique minimum element of the lattice. Two polytopes are called *combinatorially isomorphic* if their face lattices are isomorphic. The dual of a polytope is a polytope where the vertices of one correspond to the faces of the other and the edges between pairs of vertices of one correspond to the edges between pairs of faces of the other.

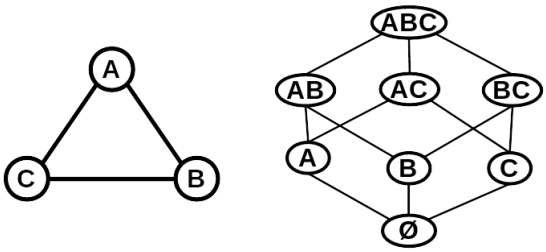


Figure 6.2 – The face lattice of a triangle. Each face is labeled by its vertex set in the diagram. Each line represents a containment relation between faces: the face which is up contains the face which is down.

The *polytope graph* (also graph of the polytope, 1-skeleton or edge-graph) is the graph associated to the vertices and edges of the polytope, ignoring higher-dimensional faces. See Figure 6.3 for an illustration. By a result of Whitney [149] the face lattice of a three-dimensional polytope is determined uniquely by its graph. Because these polytopes' face lattices are determined by their graphs, the problem of deciding whether two three-dimensional are combinatorially isomorphic can be formulated equivalently as a special case of the graph isomorphism problem.

The *grid embedding size* of a d-dimensional polytope P is the is the length of an edge of the smallest d-dimensional cube which contains P. See Figure 6.4 for an illustration.

The *diameter* of a polytope is the diameter of its associated graph (the longest shortest path between any two vertices). See Figure 6.3 for an illustration. Finding good bounds on the maximal diameter of a polytope in terms of its dimension and the number of its facets is an important question closely connected with the theory of the simplex method. The simplex method is a popular algorithm, introduced by Dantzig [41], for linear programming. Simply put, the algorithm iterates over the vertices of a convex polytope representing the feasible region of the input problem. It

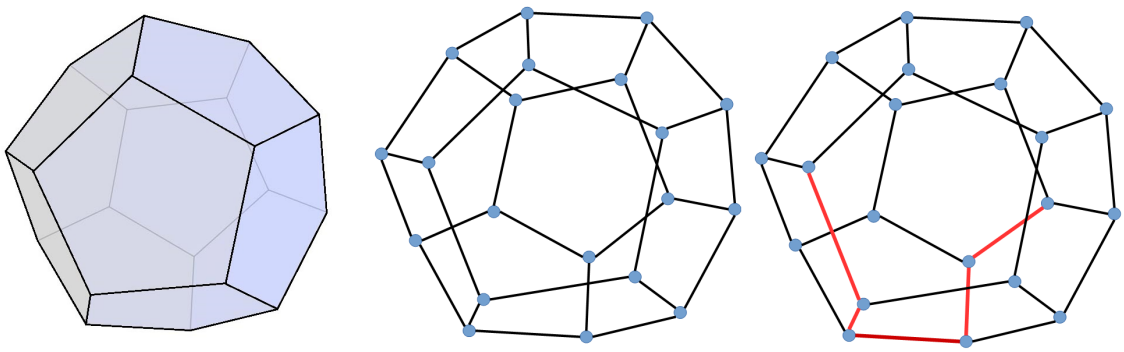


Figure 6.3 – A three dimensional polytope, its edge graph and a path achieving its diameter

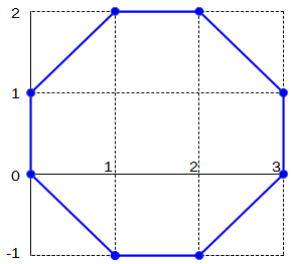


Figure 6.4 – A two dimensional polytope with grid embedding size three.

starts from a vertex which is a feasible solution of the problem and at each iteration, it tries to find a new vertex improving on the objective value. Thus a lower bound on the worst case complexity of this method is the diameter of the intrinsic polytope.

6.3 Zonotopes as Minkowski sums

To introduce the notion of *zonotope*, we need to define *Minkowski sums*. The Minkowski sum of the polytopes $P_1, P_2, ..., P_r \subset \mathbb{R}^d$ is defined as

$$P_1 + P_2 + ... + P_n = \{x_1 + x_2 + ... + x_r : x_j \in P_j\}$$

For instance, the Minkowski sum of the unit cube whose left bottom corner is the origin and the line segment $[\binom{0}{0}, \binom{2}{1}]$ is the hexagon whose vertices are $\binom{0}{0}, \binom{0}{1}, \binom{1}{0}, \binom{3}{1}, \binom{3}{2}$ and $\binom{2}{2}$, see Figure 6.5:

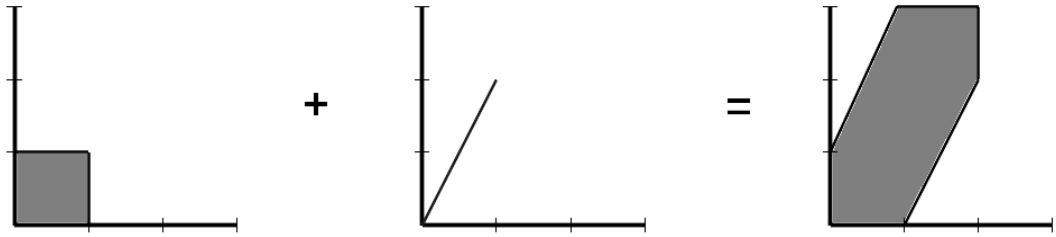


Figure 6.5 – The Minkowski sum of the unit cube and the line segment $[\binom{0}{0}, \binom{2}{1}]$ is the hexagon on the right.

If we consider now m line segments in $\mathbb{R}^d$ which have one endpoint at the origin and the other located at some vector $u_j \in \mathbb{R}^d$ for $j \in [m]$ then their Minkowski sum is called a *zonotope* and is by definition:

$$\mathsf{Ms}(u_1, u_2, ..., u_m) = \{\lambda_1 u_1 + \lambda_2 u_2 + ... + \lambda_m u_m \text{ with } \lambda_j \in [0,1]\}$$

which we also write as $\sum[0,1]\{u_j, j \in [m]\}$. In fact a zonotope can be seen equivalently as a projection of the unit cube $[0,1]^m$. To observe that, we rewrite the above equation as follows:

$$\mathsf{Ms}(u_1, u_2, ..., u_m) = \{\lambda_1 u_1 + \lambda_2 u_2 + ... + \lambda_m u_m \text{ with } \lambda_j \in [0,1]\}$$
$$= \left\{ (u_1 u_2 \cdots u_m) \begin{pmatrix} \lambda_1 \\ \lambda_2 \\ \vdots \\ \lambda_3 \end{pmatrix} : \lambda_j \in [0,1] \right\}$$

$$= A[0,1]^m$$

with A being a $(d \times m)$ matrix whose i-th column is u_j. A bit more generally, a zonotope can be any translate $A[0,1]^m + b$, with $b \in \mathbb{R}^d$. To summarize, a zonotope can be seen either as a Minkowski sum of some m line segments or as the projection of the unit cube $[0,1]^m$. In this thesis, some of the zonotopes are modified so the origin becomes their new center of mass. To do that, we dilate matrix A by a factor two and translate the resulting image to the origin, as follows:

$$\begin{aligned} & 2A[0,1]^m - (u_1 + u_2 + \cdots + u_n) \\ &= \{2u_1\lambda_1 + \cdots + 2u_n\lambda_n - (u_1 + \cdots + u_n) : \lambda_j \in [0,1]\} \\ &= \{(2\lambda - 1)u_1 + \cdots + (2\lambda - 1)u_n : \lambda_j \in [0,1]\} \\ &= \{\lambda'_1 u_1 + \cdots + \lambda'_1 u_n : \lambda'_j \in [-1,1]\} \\ &= A[-1,1]^m \end{aligned}$$

Which we also write as $\sum[-1,1]\{u_j, j \in [m]\}$. Thus we get the image of a larger cube centred at the origin. Observe that such a zonotope P is symmetric about the origin. Assuming that $P = A[-1,1]^m$ then if $y \in P$ and $y = Ax$ with $x \in [-1,1]^m$ then $-y \in P$ since $A(-x) \in P$ as well. Such a transformation will be useful in the context of this thesis as it will allow us to derive simpler proofs regarding the symmetries of the considered polytopes.

6.4 Zonotopes and hyperplane arrangements

We explain intuitively the duality between zonotopes and *hyperplane arrangements*. This parallel is useful in explaining why the diameter of (primitive) zonotopes can be easily derived from the vector set that defines them. An *hyperplane* is a subspace of one dimension less than its ambient space. A finite set $\{H_i,\ i \in [m]\}$ of hyperplanes in $\mathbb{R}^d$ is called an *arrangement* of hyperplanes. Given an arrangement of hyperplanes A, it is possible to delimit regions of the space relatively to its hyperplanes, in the following way, assuming that the i-th hyperplane of A has equation $A_i x = b$:

— $H_i^- = \{x : A_i x \leq b_i\}$,
— $H_i^0 = \{x : A_i x = b_i\}$ and
— $H_i^+ = \{x : A_i x \geq b_i\}$

With this partition it is possible to associate to to each point in $\mathbb{R}^d$ a sign vector describing its position relatively to the hyperplanes of A, as follows: (see also Figure 6.6)

$$\sigma(x) \in \{-, 0, +\}^m \text{ and } \sigma(x) = \begin{cases} + \text{ if } x \in H_i^+ \\ - \text{ if } x \in H_i^- \\ 0 \text{ if } x \in H_i^0 \end{cases}$$

The set of points with a given sign vector are called the *faces* of the arrangement. The full dimensional faces are called the *regions* or *cells* of the arrangement. The facial incidence can be described through a binary relation among sign vectors. Let $X, Y \in \{-, 0, +\}^m$ be two sign vectors. They are *incident* if $i \in [m]$ and $X_i \neq 0$ implies $X_i = Y_i$. The poset $\{\sigma(x) : x \in \mathbb{R}^d\}$ ordered by the incidence relation is called the *face poset* and is denoted $\mathcal{F}(A)$. If we assume that all the hyperplanes

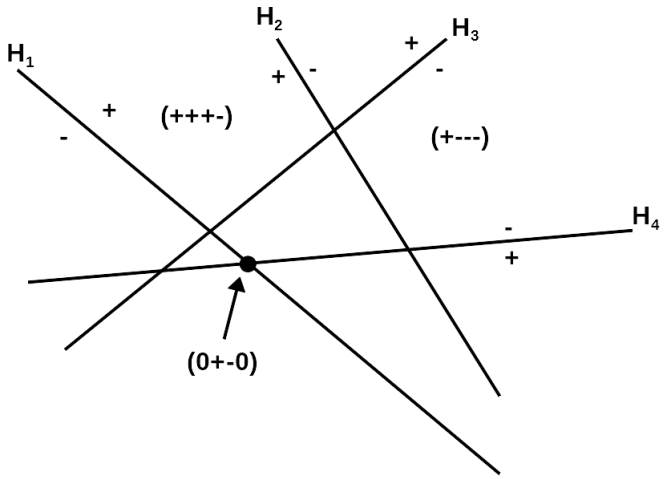


Figure 6.6 – An arrangement of hyperplanes. Each hyperplane delimits the space in two denoted by "+" and "−". The position of any point in the space can then be given by a sign vector.

contain the origin then $\mathcal{F}(A)$ contains a unique minimum element which is the zero vector, and is symmetric with respect to the origin since if a vector X is in $\mathcal{F}(A)$ then $-X$ is also in the face poset. Let M be the matrix representing the central arrangement with $H_i = \{x : M_i x = 0\}$, $i \in [m]$. Adding an artificial greatest element $\mathbf{1}$ to $\mathcal{F}(A)$ yields in fact the face lattice of the following polytope [60]:

Definition 6.4.1. $P_M = \{x : y^T M x \leq 1, \forall y \in \{-1, +1\}^m\}$.

Now, the polar $(P_M)^* = \text{conv}\{y^T M : y \in \{-1, 1\}\} = \{y^T M : y \in [-1, 1]\}$ is the zonotope whose generators are the m line segments $[-M_i, M_i]$. Simply put, the polar $P*$ of a polytope P is a polytope whose facets are the vertices of P and vice-versa.

From this duality of zonotopes and hyperplane arrangements one can derive that the diameter of a zonotope is exactly the number of its generators which are not pairwise colinear. Two parallel generators correspond to the same hyperplanes (with possibly opposite orientations). Because the vertices of the zonotopes corresponds to the full dimensional cells in the hyperplane arrangement then the diameter of the zonotope is the number of distinct hyperplanes. This holds since we need at least this number of edges to reach the vertex which corresponds to all minus signs to the vertex corresponding to all the plus signs.

6.5 Primitive zonotopes

6.5.1 Definitions

We study lattice polytopes whose vertices are drawn from $\{0, 1, \ldots, k\}^d$ to which we refer as lattice (d, k)-polytopes. For simplicity, we only consider full dimensional lattice (d, k)-polytopes. Searching for lattice polytopes with a large diameter for a given k, natural candidates include zonotopes generated by short integer vectors in order to keep the grid embedding size relatively small. In addition, we restrict to integer vectors which are pairwise linearly independent in order to maximize the diameter. Thus, for $q = \infty$ or a positive integer, and d, p positive integers, we consider the following Minkowski sum:

$$H_q(d, p) = \sum [0, 1]\{v \in \mathbb{Z}^d : \|v\|_q \leq p\,,\ \gcd(v) = 1\,,\ v \succ 0\},$$

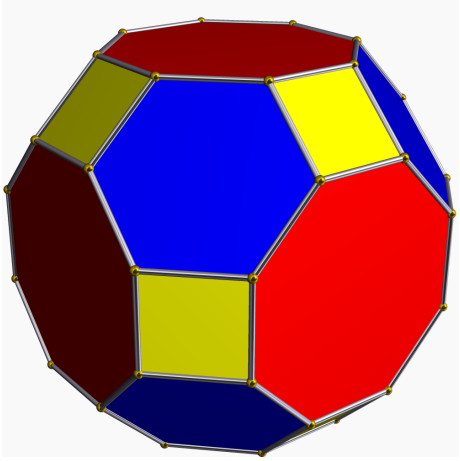


Figure 6.7 – $Z_1(3,2)$ is congruent to the truncated cuboctahedron

where $\gcd(v)$ is the largest integer dividing all entries of v, and $\succ$ the lexicographic order on $\mathbb{R}^d$, i.e. $v \succ 0$ if the first nonzero coordinate of v is positive. Similarly, we consider the *primitive zonotope* $Z_q(d,p)$, which is, up to translation the image of $H_q(d,p)$ by a homothety of factor 2: (see section 6.3 for the construction)

$$Z_q(d,p) = \sum [-1,1]\{v \in \mathbb{Z}^d \,:\, \|v\|_q \leq p\,,\ \gcd(v) = 1\,,\ v \succ 0\}$$

We also consider the *positive primitive zonotope* $Z_q^+(d,p)$ defined as the zonotope generated by the primitive integer vectors of q-norm at most p with nonnegative coordinates:

$$Z_q^+(d,p) = \sum [-1,1]\{v \in \mathbb{Z}_+^d \,:\, \|v\|_q \leq p\,,\ \gcd(v) = 1\}$$

where $\mathbb{Z}_+ = \{0,1,\ldots\}$. Similarly, we consider the Minkowski sum of the generators of $Z_q^+(d,p)$:

$$H_q^+(d,p) = \sum [0,1]\{v \in \mathbb{Z}_+^d \,:\, \|v\|_q \leq p\,,\ \gcd(v) = 1\}.$$

We illustrate the primitive zonotopes with a few examples:

(*i*) For finite q, $Z_q(d,1)$ is generated by the d unit vectors and forms the $\{-1,1\}^d$-cube. $H_q(d,1)$ is the $\{0,1\}^d$-cube.

(*ii*) $Z_1(d,2)$ is a known polytope, namely the permutahedron of type B_d [57, 83], and thus, $H_1(d,2)$ is, up to translation, a lattice $(d, 2d-1)$-polytope with $2^d d!$ vertices and diameter d^2. For example, $Z_1(2,2)$ is generated by $\{(0,1),(1,0),(1,1),(1,-1)\}$ and forms the octagon whose vertices are $\{(-3,-1),(-3,1),(-1,3),(1,3),(3,1),(3,-1),(1,-3),(-1,-3)\}$. $H_1(2,2)$ is, up to translation, a lattice $(2,3)$-polygon. $Z_1(3,2)$ is congruent to the truncated cuboctahedron – which is also called great rhombicuboctahedron, see figure 6.7 for an illustration, and is the Minkowski sum of an octahedron and a cuboctahedron, see for instance Eppstein [52]. $H_1(3,2)$ is, up to translation, a lattice $(3,5)$-polytope with diameter 9 and 48 vertices.

(*iii*) $H_1^+(d,2)$ is the Minkowski sum of the permutahedron with the $\{0,1\}^d$-cube. Thus, $H_1^+(d,2)$ is a lattice (d,d)-polytope with diameter $\binom{d+1}{2}$.

(*iv*) $Z_\infty(3,1)$ is congruent to the truncated small rhombicuboctahedron, see Figure 6.8 for an illustration, which is the Minkowski sum of a cube, a truncated octahedron, and a rhombic dodecahedron, see for instance Eppstein [52].

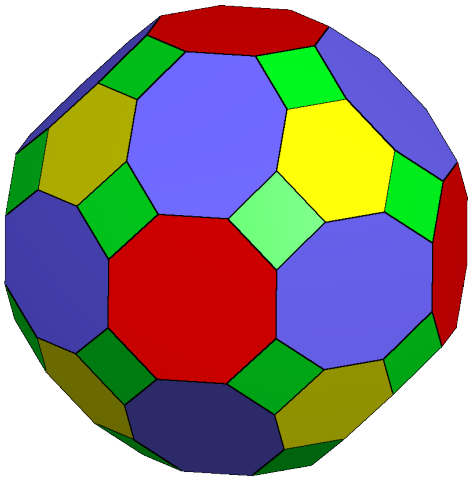


Figure 6.8 – $Z_\infty(3,1)$ is congruent to the truncated small rhombicuboctahedron

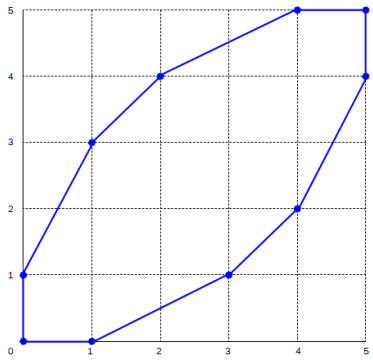


Figure 6.9 – $H^+_\infty(2,2)$

$H_\infty(3,1)$ is, up to translation, a lattice $(3,9)$-polytope with diameter 13 and 96 vertices.

(*v*) $Z^+_\infty(2,2)$ is generated by $\{(0,1),(1,0),(1,1),(1,2),(2,1)\}$ and forms the decagon whose vertices are $\{(-5,-5),(-5,-3),(-3,-5),(-3,1),(-1,3),(1,-3),(3,-1),(3,5),(5,3),(5,5)\}$. $H^+_\infty(2,2)$ is a lattice $(2,5)$-polygon, see Figure 6.9.

6.5.2 Combinatorial properties

We provide properties concerning $Z_q(d,p)$ and $Z^+_q(d,p)$, and in particular their symmetry group, diameter, and vertices. The properties listed in this section are extensions to $Z_q(d,p)$ of known properties of $Z_1(d,2)$.

Property 6.4.1.

(*i*) *$Z_q(d,p)$ is invariant under the symmetries induced by coordinate permutations and the reflections induced by sign flips.*

(*ii*) *The sum $\sigma_q(d,p)$ of all the generators of $Z_q(d,p)$ is a vertex of both $Z_q(d,p)$ and $H_q(d,p)$. The origin is a vertex of $H_q(d,p)$, and $-\sigma_q(d,p)$ is a vertex of $Z_q(d,p)$.*

(*iii*) *The coordinates of the vertices of $Z_q(d,p)$ are odd. Thus, the number of vertices of $Z_q(d,p)$ is a multiple of 2^d.*

(*iv*) *$H_q(d,p)$ is, up to translation, a lattice (d,k)-polytope where k is the sum of the first coordinates of all generators of $Z_q(d,p)$.*

(*v*) *The diameter of $Z_q(d,p)$, respectively $Z_q^+(d,p)$, is equal to the number of its generators.*

Proof. We first prove point (i). Note that if the set G of generators of a zonotope Z is invariant under coordinate permutation or sign flip, then the same holds for Z. Let π denote a permutation or a sign flip, and consider a signed sum $\sum_{g\in G} \epsilon_g g$. Then, $\pi(\sum_{g\in G} \epsilon_g g) = \sum_{g\in G} \epsilon_g \pi(g)$ is also a signed sum of generators since G is permutation and sign flip invariant. In other words, the set of all signed sums is invariant under permutations and sign flips, and thus the same holds for the convex hull Z of all signed sums. Let $J_q(d,p)$ be the set of all $-g$ for $g \in G_q(d,p)$. The zonotope $\tilde{Z}_q(d,p)$ generated by $G_q(d,p) \cup J_q(d,p)$ is the image of $Z_q(d,p)$ by a homothety of factor 2, and thus shares the same symmetry group. One can check that the set of generators of $\tilde{Z}_q(d,p)$ is invariant under coordinate permutation or sign flip, thus the same holds for $\tilde{Z}_q(d,p)$, and consequently holds for $Z_q(d,p)$.

We now prove point (ii). Consider the minimization problem $\{\min c^T x : x \in H_q(d,p)\}$ or, equivalently, $\min c^T x$ over all integer valued points of $H_q(d,p)$. Set $c = (d!\bar{x}^d, (d-1)!\bar{x}^{d-1}, \ldots, \bar{x})$ where $\bar{x} = (2p+1)^{d+1}$. Assuming that x is not the origin, let x_{i_0} denotes the first nonzero coordinate of x. Note that $x_{i_0} \geq 1$ by definition of $G_q(d,p)$, and $|x_i| \leq \bar{x}$. Thus, we have the following:

$$c^T x \geq (d+1-i_0)!\bar{x}^{d+1-i_0} - \bar{x} \sum_{i_0 < i \leq d} (d+1-i)!\bar{x}^{d+1-i} > 0$$

. In other words, the origin is the unique minimizer of a linear optimization instance over $H_q(d,p)$; that is, the origin is a vertex of $H_q(d,p)$. As $Z_q(d,p) = 2H_q(d,p) - \sigma_q(d,p)$, the point $-\sigma_q(d,p)$ is a vertex of $Z_q(d,p)$. By item (i) of Proposition 6.4.1, the point $\sigma_q(d,p)$ is a vertex of $Z_q(d,p)$, and thus $(\sigma_q(d,p) + \sigma_q(d,p))/2$ is a vertex of $H_q(d,p)$.

We now prove point (iii). We first show that the coordinates of the vertex $\sigma_q(d,p)$ are odd. As noted in the proof of item (iii) of Property 6.4.3, the i-th coordinate of $\sigma_q(d,p)$ is equal to the first coordinate of $\sigma_q(d-i+1,p)$. Thus, it is enough to show that the first coordinate of $\sigma_q(d,p)$ is odd. Except for the first unit vector $(1,0,\ldots,0)$, any generator g of $Z_q(d,p)$ with nonzero first coordinate can be paired with the generator $\bar{g}$ where $\bar{g}_1 = g_1$ and $\bar{g}_i = -g_i$ for $i \neq 1$. Thus, the sum of the first coordinates of the generators of $Z_q(d,p)$, excluding the first unit vector, is even. Hence, the first coordinate of $\sigma_q(d,p)$ is odd, and thus all the coordinates of $\sigma_q(d,p)$ are odd. Consider a vertex $v = \sum_{g\in G_q(d,p)} \epsilon(g) g$ of $Z_q(d,p)$. Since flipping the sign of an $\epsilon(g)$ does not change the parity of a coordinate of v, the coordinates of v have the same parity as the ones of $\sigma_q(d,p)$; i.e. are odd. In particular, the coordinates of a vertex of $Z_q(d,p)$ are nonzero and item (i) of Proposition 6.4.1 implies that the number of vertices of $Z_q(d,p)$ is a multiple of 2^d.

We now prove point (iv). Let Z be a zonotope generated by integer-valued generators $m^j : j = 1,\ldots,m(Z)$. Then, Z is, up to translation, a lattice (d,k)-polytope with $k \leq \max_{i=1,\ldots,d} \sum_{1\leq j\leq m(Z)} |m_i^j|$. Item (i) of Property 6.4.1 implies that the integer range of its coordinates is independent of the chosen coordinate. The same holds for $H_q(d,p)$, and, thus to determine the integer range of $H_q(d,p)$, it is enough to consider the first coordinates of its generators. Since the origin is a vertex of

$H_q(d,p)$ and the first coordinate of its generator is nonnegative, the integer range of $H_q(d,p)$ is the sum of the first coordinates of its generators.

For item (v), recall that the diameter of a zonotope is at most the number of its generators, and this inequality is satisfied with equality if no pair of generators are linearly dependent – which is the case for $Z_q(d,p)$ and $Z_q^+(d,p)$. □

Property 6.4.2.

(i) *$Z_q^+(d,p)$ is centrally symmetric and invariant under the symmetries induced by coordinate permutations.*

(ii) *The sum $\sigma_q^+(d,p)$ of all the generators of $Z_q^+(d,p)$ is a vertex of both $Z_q^+(d,p)$ and $H_q^+(d,p)$. The origin is a vertex of $H_q^+(d,p)$, and $-\sigma_q^+(d,p)$ is a vertex of $Z_q^+(d,p)$.*

Proof. Consider a generator $g \in G_q^+(d,p)$ and a coordinate permutation π. Since $\pi(g) \in G_q^+(d,p)$, $\pi(Z_q^+(d,p)) = \pi(\sum[-1,1]G_q^+(d,p)) = \sum[-1,1]\pi(G_q^+(d,p)) = \sum[-1,1]G_q^+(d,p) = Z_q^+(d,p)$. As in the proof of item (ii) of Property 6.4.1, one can check that the origin is the unique minimizer of $\{\min c^T x : x \in H_q(d,p)\}$ with $c = (1,1,\ldots,1)$. Thus, the origin is a vertex of $H_q^+(d,p)$. As $Z_q^+(d,p) = 2H_q^+(d,p) - \sigma_q(d,p)$, the point $-\sigma_q(d,p)$ is a vertex of $Z_q^+(d,p)$. Since $Z_q^+(d,p)$ is invariant under the symmetries induced by coordinate permutations, $\sigma_q(d,p)$ is a vertex of $Z_q^+(d,p)$, and thus $(\sigma_q(d,p) + \sigma_q(d,p))/2$ is a vertex of $H_q^+(d,p)$. □

A vertex v of $Z_q(d,p)$ is called *canonical* if $v_1 \geq \cdots \geq v_d > 0$. Property 6.4.1 item (i) implies that the vertices of $Z_q(d,p)$ are all the coordinate permutations and sign flips of its canonical vertices.

Property 6.4.3.

(i) *A canonical vertex v of $Z_q(d,p)$ is the unique maximizer of $\{\max c^T x : x \in Z_q(d,p)\}$ for some vector c satisfying $c_1 > c_2 > \cdots > c_d > 0$.*

(ii) *$Z_1(d,2)$ has $2^d d!$ vertices corresponding to all coordinate permutations and sign flips of the unique canonical vertex $\sigma_1(d,2) = (2d-1, 2d-3, \ldots, 1)$.*

(iii) *For $q = \infty$ or $p \neq 1$, $Z_q(d,p)$ has at least $2^d d!$ vertices including all coordinate permutations and sign flips of the canonical vertex $\sigma_q(d,p)$.*

(iv) *$Z_\infty^+(d,1)$ has at least $2 + 2d!$ vertices including the $2d!$ permutations of $\pm\sigma(d)$ where $\sigma(d)$ is a vertex with pairwise distinct coordinates, and the 2 vertices $\pm\sigma_\infty^+(d,1)$.*

Proof. We prove items (i) and (ii). Given a canonical vertex v of $Z_q(d,p)$, let c be a vector such that v is the unique maximizer of $\{\max c^T x : x \in Z_q(d,p)\}$. Up to infinitesimal perturbations, we can assume that the coordinates of c are pairwise distinct and nonzero. Note that each coordinate c_i of c is positive as otherwise flipping the sign of $v_i > 0$ would yield a point in $Z_q(d,p)$ with higher objective value than v. Assume that $c_i < c_j$ for some $i < j$. Then, $v_i = v_j$ as otherwise permuting v_i and v_j would yield a point in $Z_q(d,p)$ with higher objective value than v. Let $\pi_{ij}(c)$ be obtained by permuting c_i and c_j. Then, one can check that v is the unique maximizer of $\{\max \pi_{ij}(c)^T x : x \in Z_q(d,p)\}$. Assume, by contradiction, that $v' \in Z_q(d,p)$ satisfies $\pi_{ij}(c)^T v' \geq \pi_{ij}(c)^T v$. Then, $c^T \pi_{ij}(v') = \pi_{ij}(c)^T v' \geq \pi_{ij}(c)^T v = c^T v$ which implies $\pi_{ij}(v') = v$, and hence $v' = v$, since v is the unique maximizer of $\{\max c^T x : x \in Z_q(d,p)\}$. Thus, successive appropriate permutations yield a vector $\pi(c)$ with $\pi(c)_1 > \cdots > \pi(c)_d > 0$ such that v is the unique maximizer of $\{\max c^T x : x \in Z_q(d,p)\}$. For item (ii), one can check that $\sigma_1(d,2) = (2d-1, 2d-3, \ldots, 1)$ is the

unique maximizer of $\{\max c^T x : x \in Z_1(2,p)\}$ for any c satisfying $c_1 > \cdots > c_d > 0$. Thus, by item (i) of Property 6.4.3, $\sigma_1(d,2)$ is the unique canonical vertex of $Z_1(d,2)$ and the vertices of $Z_1(d,2)$ are the $2^d d!$ coordinate permutations and sign flips of $\sigma_1(d,2)$.

We prove item (iii). We first note that the i-th coordinate of $\sigma_q(d,p)$ is equal to the first coordinate of $\sigma_q(d-i+1,p)$. The statement trivially holds for $i=1$. For $i>1$, consider a generator g of $Z_q(d,p)$ with $g_i \neq 0$ and $g_{i_0} > 0$ for some $i_0 < i$, then g can be paired with the generator $\bar{g}$ where $g_i = -\bar{g}_i$ and $g_{i_0} = \bar{g}_{i_0}$. Thus, the sum of all the i-th coordinates of the generators of $Z_q(d,p)$ is equal to the sum of the generators of $Z_q(d,p)$ such that the first $i-1$ coordinates are zero. In other words, the i-th coordinate of $\sigma_q(d,p)$ is equal to the first coordinate of $\sigma_q(d-i+1,p)$. For example, for finite q, $\sigma_q(d,1) = (1,\ldots,1)$ and $Z_q(d,1)$ is the $\{-1,1\}^d$-cube. Then, note that for $q=\infty$ or $p \neq 1$ the first coordinate of $\sigma_q(d-i+1,p)$, which is the grid embedding size of $H_q(d-i+1,p)$, is strictly decreasing with i increasing. Thus, the action of the symmetry group of $Z_q(d,p)$ on $\sigma_q(d,p)$ generates $2^d d!$ distinct vertices of $Z_q(d,p)$. For instance, one can check the i-th coordinate of $\sigma_\infty(d,1)$ is 3^{d-i}.

We prove item (iv). The statement trivially holds for $d=1$. For $d \geq 2$, we show by induction that the vertices of $Z^+_\infty(d,1)$ include $\sigma(d)$ satisfying $0 = \sigma_1(d) < \cdots < \sigma_d(d) = 2^{d-1}$. The base case holds for $d=2$ as $\sigma(2) = (0,2)$ is a vertex of $Z^+_\infty(2,1)$. Assume such a vertex $\sigma(d)$ exists, and thus $\sigma(d) = \sum_{g \in G^+_\infty(d,1)} \epsilon(g) g$ for some $\epsilon(g)$ and $\sigma(d)$ is the unique maximizer of $\{\max c(d)^T x : x \in Z^+_\infty(d,1)\}$ for some $c(d)$. The generators of $Z^+_\infty(d+1,1)$ consist of the 2^d-1 vectors $(g,0)$ obtained by appending 0 to a generator of $Z^+_\infty(d,1)$, the 2^d-1 vectors $(g,1)$ obtained by appending 1, and the unit vector e_{d+1}. Consider the point $s(d+1) = e_{d+1} + \sum_{g \in G^+_\infty(d,1)} (g,1) - \sum_{g \in G^+_\infty(d,1)} \epsilon(g)(g,0) = (2^{d-1},\ldots,2^{d-1},2^d) - (\sigma(d),0)$; that is, $s(d+1) = (2^{d-1}-\sigma_1(d),\ldots,2^{d-1}-\sigma_{d-1}(d),0,2^d)$. Thus, the coordinates of $s(d+1)$ are pairwise distinct and a suitable permutation of $s(d+1)$ yields a point $\sigma(d+1)$ satisfying $0 = \sigma_1(d+1) < \cdots < \sigma_{d+1}(d+1) = 2^d$. To show that $\sigma(d+1)$ is a vertex of $Z^+_\infty(d+1,1)$, one can check that $\sigma(d+1)$ is the unique maximizer of $\{\max c(d+1)^T x : x \in Z^+_\infty(d+1,1)\}$ where $c(d+1) = (-c(d), c_{d+1})$ for sufficiently large c_{d+1}. Thus, for $d \geq 2$, a point $\sigma(d)$ satisfying $0 = \sigma_1(d) < \cdots < \sigma_d(d) = 2^{d-1}$ is a vertex of $Z^+_q(d,p)$. Zonotopes being centrally symmetric, $-\sigma(d)$ is a vertex of $Z^+_q(d,p)$ and the same holds for the distinct $2d!$ permutations of $\pm\sigma(d)$. □

6.6 Large diameter

Let $\delta(d,k)$ be the maximum possible edge-diameter over all lattice (d,k)-polytopes. Naddef [116] showed in 1989 that $\delta(d,1) = d$, Kleinschmidt and Onn [94] generalized this result in 1992 showing that $\delta(d,k) \leq kd$. In 2016, Del Pia and Michini [43] strengthened the upper bound to $\delta(d,k) \leq kd - \lceil d/2 \rceil$ for $k \geq 2$, and showed that $\delta(d,2) = \lfloor 3d/2 \rfloor$. Pursuing Del Pia and Michini's approach, Deza and Pournin [45] showed that $\delta(d,k) \leq kd - \lceil 2d/3 \rceil - (k-3)$ for $k \geq 3$, and that $\delta(4,3) = 8$. Then, Chadder and Deza [26], showed with a computer assisted proof that $\delta(3,4) = 7$ and $\delta(3,5) = 9$. Del Pia and Michini conclude their paper noting that the current lower bound for $\delta(d,k)$ is of order $k^{2/3}d$ and ask whether the gap between the lower and upper bounds could be closed, or at least reduced. The order $k^{2/3}d$ lower bound for $\delta(d,k)$ is a direct consequence of the determination of $\delta(2,k)$ which was investigated independently in the early nineties by Thiele [134], Balog and Bárány [11], and Ack-

eta and Žunić [1]. In this section, we highlight that $H_1(2,p)$ is the unique polygon achieving $\delta(2,k)$ for a proper k, and that a Minkowski sum of a proper subset of the generators of $H_1(d,2)$ achieves a diameter of $\lfloor (k+1)d/2 \rfloor$ for all $k \leq 2d-1$.

6.6.1 $H_1(2,p)$ as a lattice polygon with large diameter

Finding lattice polygons with the largest diameter; that is, to determine $\delta(2,k)$, was investigated independently in the early nineties by Thiele [134], Balog and Bárány [11], and Acketa and Žunić [1]. This question can be found in Ziegler's book [153] as Exercise 4.15. The answer is summarized in Proposition 6.4.1 where $\phi(j)$ is the Euler totient function counting positive integers less than or equal to j and relatively prime with j. Note that $\phi(1)$ is set to 1.

Proposition 6.4.1. *$H_1(2,p)$ is, up to translation, a lattice $(2,k)$-polygon with $k = \sum_{j=1}^{p} j\phi(j)$ where $\phi(j)$ denotes the Euler totient function. The diameter of $H_1(2,p)$ is $2\sum_{j=1}^{p}\phi(j)$ and satisfies $\delta(H_1(2,p)) = \delta(2,k)$. Thus, $\delta(2,k) = 6(\frac{k}{2\pi})^{2/3} + O(k^{1/3}\log k)$.*

Note that lattice polygons can be associated to sets of integer-valued vectors adding to zero and such that no pair of vectors are positive multiples of each other. Such sets of vectors forms a $(2,k)$-polygon with $2k$ being the maximum between the sum of the norms of the first coordinates of the vectors and the sum of the norms of the second coordinates of the vectors. Then, for $k = \sum_{j=1}^{p} j\phi(j)$ for some p, one can show that $\delta(2,k)$ is achieved uniquely by a translation of $H_1(2,p)$. For $k \neq \sum_{j=1}^{p} j\phi(j)$ for any p, $\delta(2,k)$ is achieved by a translation of a Minkowski sum of a proper subset of the generators of $H_1(2,p)$ including all generators of $H_1(2,p-1)$ for a proper p. For the order of $\sum_{j=1}^{p}\phi(j)$, respectively $\sum_{j=1}^{p} j\phi(j)$, being $\frac{3p^2}{\pi^2} + O(p\ln p)$, respectively $\frac{2p^3}{\pi^2} + O(p^2\ln p)$, we refer to [74]. The first values of $\delta(2,k)$ are given in Table 6.1.

p of $H_1(2,p)$	1		2						3								4
k	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
$\delta(2,k)$	2	3	4	4	5	6	6	7	8	8	8	9	10	10	10	11	12

Table 6.1 – Relation between $H_1(2,p)$ and $\delta(2,k)$

6.6.2 $H_1(d,2)$ as a lattice polytope with large diameter

We first show how to obtain a lower bound of $kd/2$ for $\delta(d,k)$ for some $k < d$ by considering some special graphs, namely :

Definition 6.4.2. *A n-order graph G is circulant if its vertices can be numbered such that, if some two vertices x and $(x+r) \bmod n$ are adjacent then every vertices numbered z and $(z+r) \bmod n$ are adjacent as well.*

For instance, every cycle graph is circulant, as every vertex x is adjacent to $x+1 \bmod n$ and $x-1 \bmod n$, see Figure 6.10 for an example. To a given graph G, one can associate a graphical zonotope H_G as follows. To each edge $m = (u,v)$ we associate a n dimensional vector which has zeros everywhere except for the coordinates corresponding to u and v, see Figure 6.10 for an example. The Minkowski sum of all these line segments yields what is called a graphical zonotope. Thus, assume that for a given k there exists a n-order circulant graph such that each vertex has degree k. The associated graphical zonotope has $nk/2$ (the number of edges)

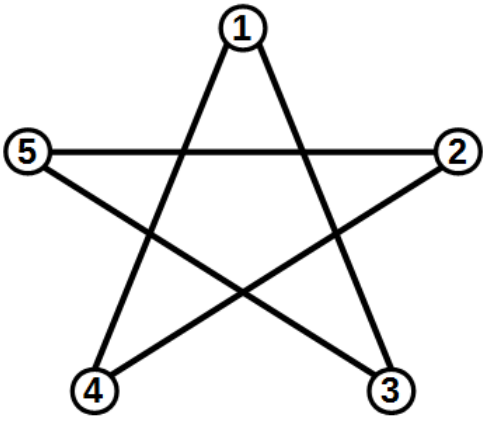


Figure 6.10 – A small instance of a circulant graph on 5 vertices. The graph is circulant since every vertex numbered x is adjacent to the vertices $x+2 \bmod 5$ and $x+3 \bmod 5$. To each edge we associate a vector of size 5 as follows. For edge $(1,3)$ it is the vector $(1,0,1,0,0)$, for edge $(1,4)$ it is the vector $(1,0,0,1,0)$ and so on.

non-colinear generators, since they corresponds to $(0,1)$ vectors with no duplicates. They have grid-embedding size k as there are at most k vectors which have a one for any given coordinate. Pursuing this approach, we show that a Minkowski sum of a proper subset of the generators of $H_1(d,2)$ yields $\delta(d,k) \geq \lfloor (k+1)d/2 \rfloor$ for all $k \leq 2d-1$.

Theorem 6.4.3. *For $k \leq 2d-1$, there is a subset of the generators of $H_1(d,2)$ whose Minkowski sum is, up to translation, a lattice (d,k)-polytope with diameter $\lfloor (k+1)d/2 \rfloor$. So for $k \leq 2d-1$ we have $\delta(d,k) \geq \lfloor (k+1)d/2 \rfloor$. For instance, $H_1^+(d,2)$ is a lattice (d,d)-polytope with diameter $\binom{d+1}{2}$, and $H_1(d,2)$ is, up to translation, a lattice $(d,2d-1)$-polytope with diameter d^2.*

Proof. We first note that the number of generators of $H_1(d,2)$ is d^2. The generators of $H_1(d,2)$ are $\{-1,0,1\}$-valued d-tuples: d permutations of $(1,0,\ldots,0)$, $\binom{d}{2}$ permutations of $(1,1,0,\ldots,0)$, and $\binom{d}{2}$ permutations of $(1,-1,0,\ldots,0)$. Thus, $\delta(H_1(d,2)) = d^2$ by Property 6.4.1 item (v). As the sum of the first coordinates of the generators of $H_1(d,2)$ is $2d-1$, $H_1(d,2)$ is, up to translation, a lattice $(d,2d-1)$-polytope by Property 6.4.1 item (iv). Consider first the case when d is even. The first $d-1$ subsets are obtained by removing from the current subset of generators of $H_1(d,2)$ a set of $d/2$ generators taken among the $\binom{d}{2}$ permutations of $(1,-1,0,\ldots,0)$. The removed $d-1$ subsets correspond to $d-1$ disjoint perfect matchings of the complete graph K_d where the nonzero i^{th} and j^{th} coordinates of a generator $(\ldots,1,\ldots,-1,\ldots)$ correspond to the edge $[i,j]$ of K_d. The first perfect matching is $[1,2],[3,d],[4,d-1],\ldots,[d/2+1,d/2+2]$. The next perfect matching is obtained by changing d to 2, and i to $i+1$ for all other entries except 1 which remains unchanged. See Figure 6.11 for an illustration. This procedure yields $d-1$ disjoint perfect matchings as, placing the vertices 2 to d on a circle around 1 where the edge $[1,2]$ is vertical and the edges $[3,d],[4,d-1],\ldots,[d/2+1,d/2+2]$ are horizontal, the procedure corresponds to the $d-1$ rotations of the initial perfect matching, see [18, Chapter 12]. As these $d-1$ perfect matchings correspond to all the generators of $H_1(d,2)$ which are permutations of $(1,-1,0,\ldots,0)$, the procedure ends with a subset of the generators of $H_1(d,2)$ forming the $\binom{d+1}{2}$ generators of $H_1^+(d,2)$. We can then repeat the same procedure where the nonzero i^{th} and j^{th} coordinates of a generator $(\ldots,1,\ldots,1,\ldots)$ correspond to the edge $[i,j]$ of K_d, and similarly obtain $d-1$ disjoint perfect matchings. The procedure now ends with a

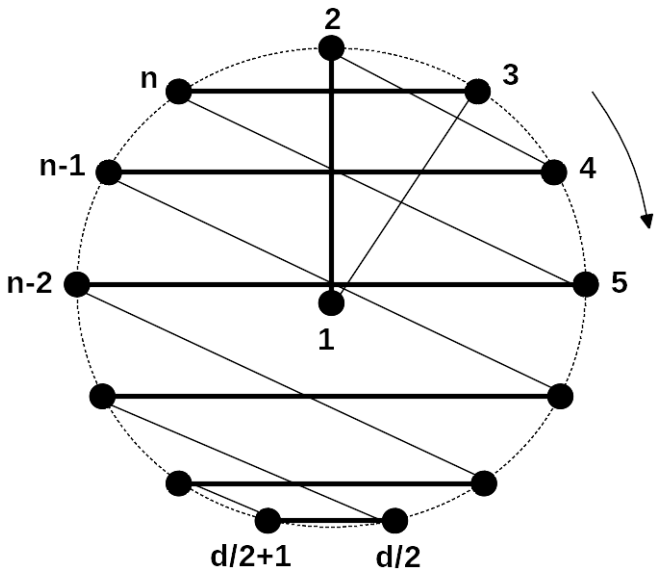


Figure 6.11 – Assuming that d is even, we place the d vertices of the clique K_d on a circle. In bold lines we have the initial perfect matching. The other disjoint perfect matchings are obtained by rotating this first matching. Such a new matching is shown in light lines.

subset of the generators of $H_1(d,2)$ forming $H_1(d,1)$; that is the unit cube. One can check that if the Minkowski sum H of the current subset of generators of $H_1(d,2)$ is a lattice (d,k)-polytope of diameter $\delta(H)$, removing the $d/2$ generators corresponding to a perfect matching yields a lattice $(d,k-1)$-polytope of diameter $\delta(H)-d/2$. Thus, starting from $H_1(d,2)$ which is a $(d,2d-1)$-polytope with diameter d^2, we obtain a (d,k)-polytope with diameter $(k+1)d/2$ for all $k \leq 2d-1$. The case when d is odd is similar. The removed subsets are of alternating sizes $\lceil d/2 \rceil$ and $\lfloor d/2 \rfloor$. Adding a dummy vertex $d+1$ to K_d, we consider the d disjoint perfect matching of K_{d+1} described for even d. The first subset consists of the $\lceil d/2 \rceil$ edges where $[3,d+1]$ is replaced by $[3,5]$, the second subset consists of the $\lfloor d/2 \rfloor$ edges where $[5,d+1]$ is removed, the third subset consists of the $\lceil d/2 \rceil$ edges where $[7,d+1]$ is replaced by $[7,9]$, and so forth. As for even d, one can check that if the Minkowski sum H of the current subset of generators of $H_1(d,2)$ is a lattice (d,k)-polytope of diameter $\delta(H)$, removing the described $\lceil d/2 \rceil$, respectively $\lfloor d/2 \rfloor$, generators yields a lattice $(d,k-1)$-polytope of diameter $\delta(H)-\lceil d/2 \rceil$, respectively $\delta(H)-\lfloor d/2 \rfloor$. Thus, starting from $H_1(d,2)$ which is a $(d,2d-1)$-polytope with diameter d^2, we obtain a (d,k)-polytope with diameter $\lfloor (k+1)d/2 \rfloor$ for all $k \leq 2d-1$. □

Conjecture 6.4.4. *$\delta(d,k) \leq \lfloor (k+1)d/2 \rfloor$, and $\delta(d,k)$ is achieved, up to translation, by a Minkowski sum of lattice vectors.*

Note that Conjecture 6.4.4 holds for all known values of $\delta(d,k)$ given in Table 6.2, and hypothesizes, in particular, that $\delta(d,3)=2d$. Note that $\delta(d,3)=2d$ for $d \leq 4$, $2d \leq \delta(d,3) \leq \lfloor 7d/3 \rfloor - 1$ when $d \not\equiv 2 \mod 3$, and $\delta(d,3) \leq \lfloor 7d/3 \rfloor$ when $d \equiv 2 \mod 3$, see [45].

Soprunov and Soprunova [130] considered the Minkowski length of a lattice polytope P; that is, the largest number of lattice segments whose Minkowski sum is contained in P. For example, the Minkowski length of the $\{0,k\}^d$-cube is kd. We consider a variant of the Minkowski length and the special case when P is the $\{0,k\}^d$-cube. Let $L(d,k)$ denote the largest number of pairwise linearly independent lattice segments whose Minkowski sum is contained in the $\{0,k\}^d$-cube. One can check that the

		k								
	$\delta(d,k)$	1	2	3	4	5	6	7	8	9	10
d	1	1	1	1	1	1	1	1	1	1	1
	2	2	3	4	4	5	6	6	7	8	8
	3	3	4	6	7	9					
	4	4	6	8							
	$\vdots$	$\vdots$	$\vdots$								
	d	d	$\lfloor 3d/2 \rfloor$								

Table 6.2 – Largest diameter $\delta(d,k)$ over all lattice (d,k)-polytopes

generators of $H_1(d,2)$ form the largest, and unique, set of primitive lattice vectors which Minkowski sum fits within the $\{0,k\}^d$-cube for $k = 2d-1$; that is, for k being the sum of the first coordinates of the d^2 generators of $H_1(d,2)$. Thus, $L(d,2d-1) = \delta(H_1(d,2)) = d^2$. Similarly, the constructions used in Proposition 6.4.1 and Theorem 6.4.3 imply that $L(2,k) = \delta(2,k)$ for all k, and $L(d,k) = \lfloor (k+1)d/2 \rfloor$ for $k \leq 2d-1$.

6.7 Small primitive zonotopes $H_q(d,p)$ and $H_q^+(d,p)$

In this section we provide the number of vertices, the diameter; that is, the number of generators, and the grid embedding size for $H_q(d,p)$ and $H_q^+(d,p)$ for small d and p, and $q = 1, 2$, and ∞. We recall that, up to translation, $Z_q(d,p)$, respectively $Z_q^+(d,p)$, is the image of $H_q(d,p)$, respectively $H_q^+(d,p)$, by a homothety of factor 2. Thus $Z_q(d,p)$ and $H_q(d,p)$, respectively $Z_q^+(d,p)$ and $H_q^+(d,p)$, have the same number of vertices and the same diameter, while the grid embedding size of the $Z_q(d,p)$, respectively $Z_q^+(d,p)$, is twice the one of $H_q(d,p)$, respectively $H_q^+(d,p)$. Since both $H_q(d,1)$ and $H_q^+(d,1)$ are equal to the $\{0,1\}^d$-cube for finite q, both are omitted from the tables provided in this section. The Euler totient function counting positive integers less than or equal to j and relatively prime with j is denoted by $\phi(j)$. Note that $\phi(1)$ is set to 1.

These values have been computed using the `C++` programming language on a standard desktop computer. Simply put, the code work as follows. If we just stick to the definition and try and compute the convex hull of all subset sums of the generators, this yields an exponential algorithm which cannot be used in practice. Instead, there is a simple polynomial algorithm. Assuming that we are given m generators $\{z_1, z_2, ..., z_m\}$ which are d-dimensional, we proceed as follows. We compute iteratively a sequence of zonotopes $Z_1, Z_2, ..., Z_m$ and their vertex sets $V_1, V_2, ..., V_k$ by considering the generators one by one. We start with Z_1 which is the interval $[0, z_1]$ with vertices $V_1 = \{0, z_1\}$. Now, supposing that we have computed Z_k and V_k we add the next generator z_{k+1}. Let $U_k := \{v + z_{k+1} : v \in V_k\}$, that is, we add z_{k+1} to each v in V_k. The next zonotope satisfies $Z_{k+1} = conv(V_k \cup U_k)$, that is, the convex hull of the union of V_k and U_k. We compute it using a convex hull algorithm (namely we did it using both the Quickhull algorithm [13] and the cdd algorithm [59]) and its set of vertices V_{k+1}. We keep going in this fashion until all vertices have been considered.

Enumerative questions concerning $H_q(d,p)$ and $H_q^+(d,p)$ have been studied in various settings. We list a few instances, and the associated OEIS sequences, see [128] for details and references therein. By f_0 we denote the function giving the number

of vertices of the polytope while function δ will denote its diameter.

(*i*) $f_0(H^+_\infty(d,1))$ corresponds to the OEIS sequence A034997 giving the number of generalized retarded functions in quantum field theory. The value of $f_0(H^+_\infty(d,1))$ was determined till $d=8$.

(*ii*) $f_0(H_\infty(d,1))$, which is the number of regions of hyperplane arrangements with $\{-1,0.1\}$-valued normals in dimension d, corresponds to the OEIS sequence A009997 giving $f_0(H_\infty(d,1))/(2^d d!)$. The value of $f_0(H_\infty(d,1))$ was determined till $d=7$.

(*iii*) $\delta(H^+_\infty(d,p))$ corresponds to the OEIS sequence A090030 with further cross-referenced sequences for $d \leq 7$ and $p \leq 8$.

(*iv*) $\delta(H^+_1(3,p))$, respectively $\delta(H^+_2(2,p))$, $\delta(H_\infty(d,2))$, $\delta(H_\infty(2,p))/4$, $\delta(H_2(2,p))/2$, $\delta(H^+_1(d,3))$, and $\delta(H^+_2(d,2))$, corresponds to the OEIS sequence A048134, respectively A049715, A005059, A002088, A175341, A008778, and A055795.

(*v*) the grid embedding size of $H_2(d,2)$, respectively $H_\infty(d,2)$ and $H^+_1(d,3)$, corresponds to the OEIS sequence A161712, respectively A080961 and A052905.

Small primitive zonotopes $H_q(d,p)$

In Tables 6.3, 6.4, and 6.5, the number of vertices $f_0(H_q(d,p))$ is divided by $2^d d!$ and followed by its diameter $\delta(H_q(d,p))$ and grid embedding size. For instance, the entry 26 (49, 53) for $(q,d,p)=(1,3,4)$ in Table 6.3 indicates that $H_1(3,4)$ has $26 \times 2^3 3! = 1248$ vertices, diameter 49, and is, up to translation, a lattice $(3,53)$-polytope.

Small primitive zonotopes $H_1(d,p)$

Property 6.4.1.

(*i*) *$H_1(d,1)$ is the $\{0,1\}^d$-cube,*

(*ii*) *$H_1(d,2)$ is, up to translation, a lattice (d,k)-polytope with $k=2d-1$, and diameter d^2, and $2^d d!$ vertices,*

(*iii*) *$H_1(d,3)$ is, up to translation, a lattice (d,k)-polytope with $k=2d^2+2d-3$, and diameter $d(d+2)(2d-1)/3$,*

(*iv*) *$H_1(d,4)$ is, up to translation, a lattice (d,k)-polytope with $k=\binom{d-1}{0}+16\binom{d-1}{1}+20\binom{d-1}{2}+8\binom{d-1}{3}$, and diameter $d(d^3+2d^2+2d-2)/3$,*

(*v*) *$H_1(2,p)$ is, up to translation, a lattice $(2,k)$-polygon with $k=\sum_{1\leq j\leq p} j\phi(j)$, and diameter $2\sum_{1\leq j\leq p}\phi(j)$.*

		p				
	$H_1(d,p)$	2	3	4	5	6
	2	1 (4,3)	2 (8,9)	3 (12,17)	5 (20,37)	6 (24,49)
d	3	1 (9,5)	7 (25,21)	26 (49,53)	102 (97,133)	227 (145,229)
	4	1 (16,7)	40 (56,37)	531 (136,117)	6 741 (312,337)	? (560,709)
	5	1 (25,9)	339 (105,57)	? (305,217)	? (801,713)	? (1 681,1 769)

Table 6.3 – Small primitive zonotopes $H_1(d,p)$

Proof. One can check that the generators of $H_1(d,2)$ consist of $\binom{d}{1}$ unity vectors and $2\binom{d}{2}$ vectors $\{\ldots,1,\ldots,\pm1,\ldots\}$. Thus, the diameter of $H_1(d,2)$ is $\binom{d}{1}+2\binom{d}{2}=d^2$. Similarly, one can check that the sum of the first coordinates of the generators of $H_1(d,2)$ is $2d-1$. Note that $H_1(d,2)$ is the permutahedron of type B_d. Then, one can check that, in addition to the previously determined generators of $H_1(d,2)$, the generators of $H_1(d,3)$ consist of $2\binom{d}{2}$ vectors $\{\ldots,1,\ldots,\pm2,\ldots\}$, $2\binom{d}{2}$ vectors $\{\ldots,2,\ldots,\pm1,\ldots\}$, and $4\binom{d}{3}$ vectors $\{\ldots,1,\ldots,\pm1,\ldots,\pm1,\ldots\}$. Thus, the diameter of $H_1(d,3)$ is $\binom{d}{1}+6\binom{d}{2}+4\binom{d}{3}=d(d+2)(2d-1)/3$. Similarly, one can check that the sum of the first coordinates of the generators of $H_1(d,3)$ is $\binom{d-1}{0}+8\binom{d-1}{1}+4\binom{d-1}{2}=2d^2+2d-3$. Furthermore, one can check that, in addition to the previously determined generators of $H_1(d,3)$, the generators of $H_1(d,4)$ consist of $2\binom{d}{2}$ vectors $\{\ldots,1,\ldots,\pm3,\ldots\}$, $2\binom{d}{2}$ vectors $\{\ldots,3,\ldots,\pm1,\ldots\}$, $4\binom{d}{3}$ vectors $\{\ldots,1,\ldots,\pm1,\ldots,\pm2,\ldots\}$, $4\binom{d}{3}$ vectors $\{\ldots,1,\ldots,\pm2,\ldots,\pm1,\ldots\}$, $4\binom{d}{3}$ vectors $\{\ldots,2,\ldots,\pm1,\ldots,\pm1,\ldots\}$, and $8\binom{d}{4}$ vectors $\{\ldots,1,\ldots,\pm1,\ldots,\pm1,\ldots,\pm1,\ldots\}$. Thus, the diameter of $H_1(d,4)$ is $\binom{d}{1}+10\binom{d}{2}+16\binom{d}{3}+8\binom{d}{4}=d(d^3+2d^2+2d-2)/3$. Similarly, one can check that the sum of the first coordinates of the generators of $H_1(d,4)$ is $\binom{d-1}{0}+16\binom{d-1}{1}+20\binom{d-1}{2}+8\binom{d-1}{3}$. Finally, item (v) corresponds to Proposition 6.4.1. □

Small primitive zonotopes $H_2(d,p)$

Property 6.4.2.

(*i*) *$H_2(d,1)$ is the $\{0,1\}^d$-cube,*

(*ii*) *$H_2(d,2)$ is, up to translation, a lattice (d,k)-polytope with $k=\sum_{0\le j\le 3}2^j\binom{d-1}{j}$, and diameter $\sum_{0\le j\le 3}2^j\binom{d}{j+1}$.*

		p			
	$H_2(d,p)$	2	3	4	5
	2	1 (4,3)	2 (8,9)	4 (16,27)	6 (24,51)
d	3	2 (13,9)	26 (49,57)	126 (109,161)	443 (205,377)
	4	14 (40,27)	1 427 (192,193)	? (592,795)	? (1 424,2 411)
	5	273 (105,65)	? (641,577)		

Table 6.4 – Small primitive zonotopes $H_2(d,p)$

Proof. One can check that the generators of $H_2(d,2)$ consist of $\binom{d}{1}$ unity vectors, $2\binom{d}{2}$ vectors $\{\ldots,1,\ldots,\pm1,\ldots\}$, $4\binom{d}{3}$ vectors $\{\ldots,1,\ldots,\pm1,\ldots,\pm1,\ldots\}$, and $8\binom{d}{4}$ vectors $\{\ldots,1,\ldots,\pm1,\ldots,\pm1,\ldots,\pm1,\ldots\}$. Thus, the diameter of $H_2(d,2)$ is $\sum_{0\le j\le 3}2^j\binom{d}{j+1}$. Similarly, one can check that the sum of the first coordinates of the generators of $H_2(d,2)$ is $\sum_{0\le j\le 3}2^j\binom{d-1}{j}$. □

Small primitive zonotopes $H_\infty(d,p)$

Property 6.4.3.

(*i*) *$H_\infty(d,1)$ is, up to translation, a lattice (d,k)-polytope with $k=3^{d-1}$, and diameter $(3^d-1)/2$,*

(*ii*) $H_\infty(d,2)$ *is, up to translation, a lattice* (d,k)*-polytope with* $k = 3\times 5^{d-1} - 2\times 3^{d-1}$*, and diameter* $(5^d-3^d)/2$*,*

(*iii*) $H_\infty(2,p)$ *is, up to translation, a lattice* $(2,k)$*-polygon with diameter* $4\sum_{1\le j\le p}\phi(j)$*.*

(*iv*) *The number* $m(d,1)$ *of vertices of* $H_\infty(d,1)$ *satisfies*

$$2^d d! \le m(d,1) \le 2\sum_{0\le i\le d-1}\binom{(3^d-3)/2}{i} - 2\binom{(3^{d-1}-3)/2}{d-1}.$$

		p			
	$H_\infty(d,p)$	1	2	3	4
d	2	1 (4,3)	2 (8,9)	4 (16,27)	6 (24,51)
	3	2 (13,9)	26 (49,57)	228 (145,249)	910 (289,633)
	4	14 (40,27)	4 333 (272,321)	? (1 120,1 923)	? (2 928,6 459)
	5	516 (121,81)			
	6	12 4187 (364,243)			
	7	214 580 603 (1 093,729)			

Table 6.5 – Small primitive zonotopes $H_\infty(d,p)$

Proof. One can check that $H_\infty(d,1)$ has $(3^d-1)/2$ generators consisting of all $\{-1,0,1\}$-valued vectors which first nonzero coordinate is positive. Out of the 5^d $\{-2,-1,0,1,2\}$-valued vectors, 3^d are $\{-2,0,2\}$-valued. Thus, keeping the ones which first nonzero coordinate is positive, $H_\infty(d,2)$ has $(5^d-3^d)/2$ generators. Similarly, one can check that the sum of the first coordinates of the generators of $H_\infty(d,2)$ is $3\times 5^d - 5\times 3^d$. The generators (i,j) of $H_\infty(2,p)$ such that $||(i,j)||_\infty \le 1$ are $(1,0),(0,1),(1,1)$ and $(1,-1)$. For a given $i>1$, there are $2\phi(i)$ generators (i,j) such that $||(i,j)||_\infty > 1$ and $j<i$. Thus, there are $4\sum_{2\le j\le p}\phi(j)$ generators (i,j) such that $||(i,j)||_\infty > 1$. Thus, the diameter of $H_\infty(2,p)$ is $4\sum_{1\le j\le p}\phi(j)$.

We prove now point *iv*. The first inequality restates item (*iii*) of Property 6.4.3 where (q,d,p) is set to $(\infty,d,1)$. The second inequality is obtained by exploiting the structure of the generators of $H_\infty(d,1)$. One can check that $H_\infty(d,1)$ has $(3^d-1)/2$ generators and that removing the first zero of the generators of $H_\infty(d,1)$ starting with zero yields exactly the $(3^{d-1}-1)/2$ generators of $H_\infty(d-1,1)$. We recall that the number of vertices $f_0(Z)$ of a d-dimensional zonotope Z generated by m generators is bounded by $\bar{f}(d,m) = 2\sum_{0\le i\le d-1}\binom{m-1}{i}$ [61]. By duality, the number $f_0(Z)$ of vertices of a zonotope Z is equal to the number $f_{d-1}(\mathcal{A})$ of cells of the associate hyperplane arrangement $\mathcal{A}$ where each generator m^j of Z corresponds to an hyperplane h^j of $\mathcal{A}$. The inequality $f_0(Z) \le \bar{f}(d,m)$ is based on the inequality $f_{d-1}(\mathcal{A}) \le f_{d-1}(\mathcal{A}\setminus h^j) + f_{d-1}(\mathcal{A}\cap h^j)$ for any hyperplane h^j of $\mathcal{A}$ where $\mathcal{A}\setminus h^j$ denotes the arrangement obtained by removing h^j from $\mathcal{A}$, and $\mathcal{A}\cap h^j$ denotes the arrangement obtained by intersecting $\mathcal{A}$ with h^j. This last inequality and the duality between zonotopes and hyperplane arrangements are detailed, for example, in [61]. Recursively applying this inequality to the arrangement $\mathcal{A}_\infty(d,1)$ associated to $H_\infty(d,1)$ till the remaining $(3^{d-1}-1)/2$ hyperplanes form a $(d-1)$-dimensional arrangement equivalent to $\mathcal{A}_\infty(d-1,1)$ yields: $f_{d-1}(\mathcal{A}_\infty(d,1)) \le \bar{f}(d,(3^d-1)/2) - \big(\bar{f}(d,(3^{d-1}-1)/2) - \bar{f}(d-1,(3^{d-1}-1)/2)\big)$ which completes the

proof since $f_{d-1}(\mathcal{A}_\infty(d,1)) = f_0(H_\infty(d,1))$ and $\bar{f}(d,m) - \bar{f}(d-1,m) = 2\binom{m-1}{d}$. In other words, the inequality is based on the inductive build-up of $H_\infty(d,1)$ starting with the $(3^{d-1}-3)/2$ generators with zero as first coordinate, and noticing that these $(3^{d-1}-3)/2$ generators belong to a lower dimensional space.

□

6.7.1 Small positive primitive zonotopes $H_q^+(d,p)$

In Tables 6.6, 6.7, and 6.8, the number of vertices $f_0(H_q^+(d,p))$ is followed by its diameter $\delta(H_q^+(d,p))$ and grid embedding size. For instance, the entry 1082 (15, 5) for $(q,d,p) = (1,5,2)$ in Table 6.6 indicates that $H_1^+(5,1)$ has 1082 vertices, diameter 15, and is a lattice $(5,5)$-polytope.

Small positive primitive zonotopes $H_1^+(d,p)$

Property 6.4.4.

(*i*) $H_1^+(d,1)$ *is the* $\{0,1\}^d$*-cube,*

(*ii*) $H_1^+(d,2)$ *is a lattice* (d,k)*-polytope with* $k=d$*, and diameter* $\binom{d+1}{2}$*,*

(*iii*) $H_1^+(d,3)$ *is a lattice* (d,k)*-polytope with* $k = (d^2+5d-4)/2$ *and diameter* $d(d^2+6d-1)/6$*.*

(*iv*) $H_1^+(2,p)$ *is a lattice* $(2,k)$*-polygon with* $k = 1 + \sum_{2\le j\le p} j\phi(j)/2$*, and diameter* $1 + \sum_{1\le j\le p} \phi(j)$*.*

	$H_1^+(d,p)$	p = 2	3	4	5	6
d	2	6 (3,2)	10 (5,5)	14 (7,9)	22 (11,19)	26 (13,25)
	3	26 (6,3)	110 (13,10)	314 (22,22)	1 022 (40,52)	1 970 (55,82)
	4	150 (10,4)	2 194 (26,16)	17534 (51,41)	145 198 (103,106)	593 402 (161,193)
	5	1 082 (15,5)	71 582 (45,23)	2 062 682 (100,67)	? (221,188)	? (386,386)
	6	9 366 (21,6)	? (71,31)	?(176,106)		

Table 6.6 – Small positive primitive zonotopes $H_1^+(d,p)$

Proof. One can check that the generators of $H_1^+(d,2)$ consist of $\binom{d}{1}$ unity vectors and $\binom{d}{2}$ vectors $\{\ldots,1,\ldots,1,\ldots\}$. Thus, the diameter of $H_1^+(d,2)$ is $\binom{d}{1} + \binom{d}{2} = \binom{d+1}{2}$. Similarly, one can check that the sum of the first coordinates of the generators of $H_1^+(d,2)$ is d. Note that $H_1^+(2,p)$ is the Minkowski sum of the permutahedron with the $\{0,1\}^d$-cube. One can check that, in addition to the previously determined generators of $H_1^+(2,p)$, the generators of $H_1^+(d,3)$ consist of $\binom{d}{3}$ vectors $\{\ldots,1,\ldots,1,\ldots,1,\ldots,\}$, $\binom{d}{2}$ vectors $\{\ldots,1,\ldots,2,\ldots\}$, and $\binom{d}{2}$ vectors $\{\ldots,2,\ldots,1,\ldots\}$. Thus $H_1^+(d,3)$ has $\binom{d}{3} + 3\binom{d}{2} + \binom{d}{1}$ generators. Similarly, one can check that the sum of the first coordinates of the generators of $H_1^+(d,3)$ is $\binom{d-1}{2} + 4\binom{d-1}{1} + \binom{d}{0}$. Out of the generators of $H_1(2,p)$, $\sum_{2\le j\le p} \phi(j)$ have a negative coordinate. Thus, the diameter of $H_1^+(2,p)$ is $1 + \sum_{1\le j\le p} \phi(j)$. Similarly, one can check that the sum of the first coordinates of the generators of $H_1^+(2,p)$ is $1 + \sum_{2\le j\le p} j\phi(j)/2$. □

Small positive primitive zonotopes $H_2^+(d,p)$

Property 6.4.5.

(*i*) $H_2^+(d,1)$ *is the* $\{0,1\}^d$*-cube,*

(*ii*) $H_2^+(d,2)$ *is a* (d,k) *polytope with* $k = \binom{d}{1} + \binom{d}{3}$*, and diameter* $\binom{d+1}{2} + \binom{d+1}{4}$.

		p			
	$H_2^+(d,p)$	2	3	4	5
	2	6 (3,2)	10 (5,5)	18 (9,14)	26 (13,26)
d	3	32 (7,4)	212 (19,19)	1 010 (40,54)	3 074 (70,120)
	4	370 (15,8)	19 438 (55,49)	362 962 (141,170)	3 497 862 (299,462)
	5	10 922 (30,15)	? (136,108)	? (441,487)	

Table 6.7 – Small positive primitive zonotopes $H_2^+(d,p)$

Proof. One can check that the generators of $H_2^+(d,2)$ consist of $\binom{d}{i}$ vectors with exactly i ones for $i = 1, 2, 3$, and 4. Thus, the diameter of $H_2^+(d,2)$ is $\binom{d+1}{2} + \binom{d+1}{4}$. Similarly, one can check that the sum of the first coordinates of the generators of $H_2^+(d,2)$ is $\binom{d}{1} + \binom{d}{3}$. □

Small positive primitive zonotopes $H_\infty^+(d,p)$

Property 6.4.6.

(*i*) $H_\infty^+(d,1)$ *is, a lattice* (d,k)*-polytope with* $k = 2^{d-1}$*, and diameter* $2^d - 1$,

(*ii*) $H_\infty^+(d,2)$ *is a lattice* (d,k)*-polytope with* $k = 3^d - 2^d$*, and diameter* $3^d - 2^d$,

(*iii*) $H_\infty^+(2,p)$ *is a lattice* $(2,k)$*-polygon with diameter* $1 + 2\sum_{1 \le j \le p} \phi(j)$.

		p			
	$H_\infty^+(d,p)$	1	2	3	4
	2	6 (3,2)	10 (5,5)	18 (9,14)	26 (13,26)
d	3	32 (7,4)	212 (19,19)	1 418 (49,76)	4 916 (91,184)
	4	370 (15,8)	27 778 (65,65)	1 275 842 (225,344)	? (529,1 064)
	5	11 292 (31,16)	? (211,211)	? (961,1456)	? (2 851,5 716)
	6	1 066 044 (63,32)			
	7	347 326 352 (127,64)			
	8	419 172 756 930 (255,128)			

Table 6.8 – Small positive primitive zonotopes $H_\infty^+(d,p)$

Proof. One can check that $H_\infty^+(d,1)$ has $2^d - 1$ generators consisting of all $\{0,1\}$-valued vectors except the origin. Thus, the diameter of $H_\infty^+(d,1)$ is $2^d - 1$. Similarly, one can check that the sum of the first coordinates of the generators of $H_\infty^+(d,1)$ is 2^{d-1}. Out of the 3^d $\{0,1,2\}$-valued vectors, 2^d are $\{0,2\}$-valued. Thus, the diameter of $H_\infty^+(d,2)$ is $3^d - 2^d$. Similarly, one can check that the sum of the first coordinates of the generators of $H_\infty^+(d,2)$ is $3^d - 2^d$. The generators (i,j) of $H_\infty^+(2,p)$ such that $||(i,j)||_\infty \le 1$ are $(1,0)$, $(0,1)$, and $(1,1)$. For a given $i > 1$, there are $\phi(i)$ generators (i,j) such that $||(i,j)||_\infty > 1$ and $j < i$. Thus, there are $2\sum_{2 \le j \le p} \phi(j)$ generators (i,j) such that $||(i,j)||_\infty > 1$. Thus, the diameter of $H_\infty^+(2,p)$ is $1 + 2\sum_{1 \le j \le p} \phi(j)$. □

6.7.2 Open problems

The zonotopes $H^+_\infty(d,1)$ can be related to hypergraphs. An *hypergraph* is simply a graph in which edges can span more than two vertices. Formally, an hypergraph $H = (X, E)$ has vertex set $X = \{v_1, v_2, ..., v_d\}$ and edge set $E \subseteq \{0,1\}^d$, see Figure 6.12 for an illustration.

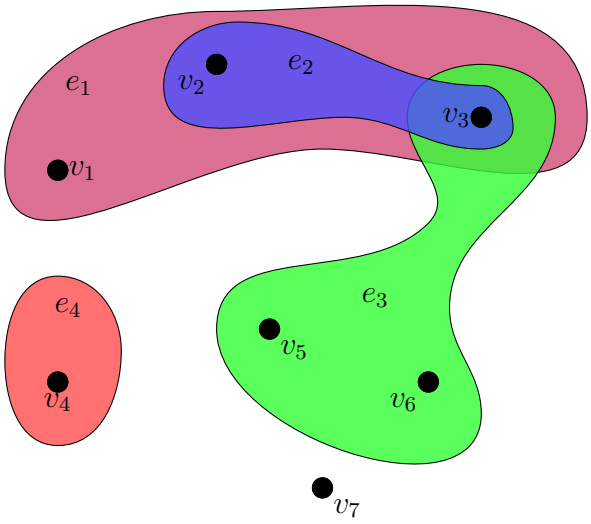


Figure 6.12 – An hypergraph with seven vertices and five edges, each corresponding to a color. We give the edges as vectors as follows. $e_1 = \{1,1,1,0,0,0,0\}$ $e_2 = \{0,1,1,0,0,0,0\}$ $e_3 = \{0,0,1,0,1,1,0\}$ $e_4 = \{0,0,0,1,0,0,0\}$.

The open questions deal with a reformulation in term of degree sequence of hypergraphs. The question is presented within the context of $H^+_q(d,p)$ but could be adapted to $H_q(d,p)$. Each subset $E \subseteq \{0,1\}^d$ can be associated to the edge set of a hypergraph with ground set $[d]$. The vector $\sum_{e \in E} e$ is called the *degree sequence* of E (and of the associated hypergraph), and the convex hull of the degree sequences of all hypergraphs with ground set $[d]$ is called the *hypergraph polytope* D_d; and thus $D_d = H^+_\infty(d,1)$. Considering only k-uniform hypergraphs; that is, subsets $E \subseteq \{0,1\}^d$ where all vectors in E have k nonzero entries, one obtains the *k-uniform hypergraph polytope* $D_d(k)$ as the convex hull of the degree sequences of all k-uniform hypergraphs. The k-uniform hypergraph polytope, in particular $D_d(2)$ and $D_d(3)$, have been extensively studied, see [36, 54, 95, 115] and references therein. A natural question raised in the literature asks for suitable necessary and sufficient conditions to check whether a vector $h \in D_d(k) \cap \mathbb{Z}^d$ is the degree sequence of some k-uniform hypergraph. For $k = 2$; that is for graphs, the celebrated Erdös-Gallai Theorem [54] shows that the trivial necessary condition is also sufficient. For $k = 3$; that is for 3-uniform hypergraphs, the question was raised by Klivans and Reiner [95]. Liu [103] exhibited counterexamples by constructing *holes* for $d \geq 16$; that is, vectors h in $D_d(3) \cap \mathbb{Z}^d$ such that the sum of the coordinates of h is a multiple of 3, but h is not the degree sequence of a 3-uniform hypergraph.

We call a vector in $H^+_q(d,p) \cap \mathbb{Z}^d$ a *hole* if it cannot be written as the sum of a subset of the generators of $H^+_q(d,p)$. It would be interesting to explicitly find such holes and better understand them. A natural follow-up question, provided there are holes, is "*For given fixed positive integers p and q, what is the complexity of deciding if a given vector $h \in H^+_q(d,p) \cap \mathbb{Z}^d$ is a hole, and if not, of writing h as the sum of a subset of generators of $H^+_q(d,p)$?*".

Chapter 7

Perspectives

Few questions have been answered, many have emerged, as noted in the conclusion sections of the chapters. We will not recall these open problems here. We want to discuss some future directions that are being currently investigated.

Our contribution of Chapter 3 shed light on an interesting phenomenon regarding the relationship between the maximal cliques and induced properties of the graph. A property is induced on G if there exists an ordering of the vertices $v_1, ..., v_n$ of the vertices of G such that property P holds on $G[N(v_i) \cap V_i]$. We recall that V_i is the set of vertices following v_i including itself in this ordering, that is, the set $\{v_i, v_{i+1}, ..., v_n\}$. Following that definition, a graph is k-degenerate if it has inductive property P_k: "the vertex set is of size less or equal than k". What we essentially proved is that if we can solve the maximal cliques enumeration problem on a graph with property P_k then we can solve it on a graph with inductive property P_k. It seems to us that, in fact, this holds for any property. Thus we are currently looking into this question and try to extend this approach to other clique related problems.

Our contribution of Chapter 5 is one of the first self-stabilizing algorithms considering the communication model simulating the message passing paradigm (link register). What we would like to do next is find a way to transform systematically any given algorithm of the standard communication model used in self-stabilizing algorithms (shared memory model) into the link register model. Such transformers exist but very little is known about their complexity and some of them are restricted (for instance at the daemon level). We recall that these transformers have been discussed in the introduction of Chapter 5. Thus we would like to design a general transformer whose complexity we could analyze (and which would have polynomial transformation cost, ideally). To do that, we first need to design a self-stabilizing algorithm solving the maximal independent set problem in the link register paradigm. This is the first step which is currently under investigation.

Finally, concerning the diameter of lattice polytopes, our goal is double. First, we are trying to compute some missing values presented in the tables of Chapter 6. To achieve that, the algorithm that we presented for computing small zonotopes is not efficient. New theoretical tools are needed in order to understand the structure of the polytopes and derive from that faster algorithms computing them. The second question under investigation is the one of holes. We are currently trying to extend Liu's approach [103] to our framework, one problem being that the author gave little to no intuition on how he actually found the holes. Is a computational or theoretical approach necessary for this question?

Bibliography

[1] D. Acketa and J. Žunić. On the maximal number of edges of convex digital polygons included into an $m \times m$-grid. *Journal of Combinatorial Theory A*, 69:358–368, 1995.

[2] N. K. Ahmed, J. Neville, R. A. Rossi, and N. Duffield. Efficient graphlet counting for large networks. In *Data Mining (ICDM), 2015 IEEE International Conference on*, pages 1–10. IEEE, 2015.

[3] X. Allamigeon, P. Benchimol, S. Gaubert, and M. Joswig. Long and winding central paths. *arXiv:1405.4161*, 2014.

[4] N. Alon, R. Yuster, and U. Zwick. Color-coding. *Journal of the ACM (JACM)*, 42(4):844–856, 1995.

[5] N. Alon, R. Yuster, and U. Zwick. Finding and counting given length cycles. *Algorithmica*, 17(3):209–223, 1997.

[6] A. Arora and M. Gouda. Closure and convergence: A foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering*, 19(11):1015–1027, 1993.

[7] Benno Artmann. *Euclid—the creation of mathematics*. Springer Science & Business Media, 2012.

[8] Y. Asada and M. Inoue. An efficient silent self-stabilizing algorithm for 1-maximal matching in anonymous networks. In *WALCOM: Algorithms and Computation - 9th International Workshop*, pages 187–198. Springer International Publishing, 2015.

[9] H. Attiya and J. Welch. *Distributed computing: fundamentals, simulations, and advanced topics*, volume 19. John Wiley & Sons, 2004.

[10] J. G. Augustson and Jack Minker. An analysis of some graph theoretical cluster techniques. *Journal of the ACM (JACM)*, 17(4):571–588, 1970.

[11] A. Balog and I. Bárány. On the convex hull of the integer points in a disc. In *Proceedings of the Seventh Annual Symposium on Computational Geometry*, pages 162–165, 1991.

[12] A.-L. Barabási and R. Albert. Emergence of scaling in random networks. *science*, 286(5439):509–512, 1999.

[13] C. B. Barber, D. P. Dobkin, and H. Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Trans. Math. Softw.*, 22(4):469–483, December 1996.

[14] V. Batagelj and M. Zaversnik. An $\mathcal{O}(m)$ algorithm for cores decomposition of networks. *arXiv preprint cs/0310049*, 2003.

[15] J. Beauquier, A. K. Datta, M. Gradinariu, and booktitle=DISC pages=223–237 year=2000 organization=Springer Magniette, F. Self-stabilizing local mutual exclusion and daemon refinement.

[16] M. Bentert, T. Fluschnik, A. Nichterlein, and R. Niedermeier. *Parameterized Aspects of Triangle Enumeration*, pages 96–110. Springer Berlin Heidelberg, Berlin, Heidelberg, 2017.

[17] P. Berenbrink, T. Friedetzky, and R. A. Martin. On the stability of dynamic diffusion load balancing. *Algorithmica*, 50(3):329–350, 2008.

[18] C. Berge. *Graphes*. Gauthier-Villars, 1983.

[19] E. Birmelé, R. Ferreira, R. Grossi, Andrea Marino, Nadia Pisanti, Romeo Rizzi, and Gustavo Sacomoto. *Optimal Listing of Cycles and st-Paths in Undirected Graphs*, pages 1884–1896.

[20] A. Björklund, P. Kaski, and L. Kowalik. Counting thin subgraphs via packings faster than meet-in-the-middle time. In *Proceedings of the twenty-fifth annual ACM-SIAM symposium on Discrete algorithms*, pages 594–603. Society for Industrial and Applied Mathematics, 2014.

[21] A. Björklund, R. Pagh, V. V. Williams, and U. Zwick. *Listing Triangles*, pages 223–234. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[22] J. A. Bondy and U. S. R. Murty. *Graph theory with applications*, volume 290. Macmillan London, 1976.

[23] C. Bron and J. Kerbosch. Algorithm 457: Finding all cliques of an undirected graph. *Commun. ACM*, 16(9):575–577, September 1973.

[24] L. Cai, S. M. Chan, and S. O. Chan. *Random Separation: A New Method for Solving Fixed-Cardinality Optimization Problems*, pages 239–250. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[25] F. Cazals and C. Karande. A note on the problem of reporting maximal cliques. *Theoretical Computer Science*, 407(1-3):564–568, 2008.

[26] N. Chadder and A. Deza. Computational determination of the largest lattice polytope diameter. *CoRR*, abs/1704.01687, 2017.

[27] L. Chang, J. X. Yu, and L. Qin. Fast maximal cliques enumeration in sparse graphs. *Algorithmica*, 66(1):173–186, 2013.

[28] S. Chattopadhyay, L. Higham, and K. Seyffarth. Dynamic and self-stabilizing distributed matching. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 290–297. ACM, 2002.

[29] Subhendu Chattopadhyay, Lisa Higham, and Karen Seyffarth. Dynamic and self-stabilizing distributed matching. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 290–297. ACM, 2002.

[30] N. Chiba and T. Nishizeki. Arboricity and subgraph listing algorithms. *SIAM Journal on Computing*, 14(1):210–223, 1985.

[31] M. Chrobak and D. Eppstein. Planar orientations with low out-degree and compaction of adjacency matrices. *Theoretical Computer Science*, 86(2):243 – 266, 1991.

[32] J. Cohen, J. Lefevre, K. Maamra, G. Manoussakis, and L. Pilard. Polynomial self-stabilizing algorithm and proof for a 2/3-approximation of a maximum matching. *CoRR*, abs/1611.06038, 2016.

[33] J. Cohen, K. Maâmra, G. Manoussakis, and L. Pilard. Polynomial self-stabilizing maximum matching algorithm with approximation ratio 2/3. In *International Conference On Principles Of Distributed Systems, OPODIS*, 2016.

[34] Johanne Cohen, Jonas Lefevre, Khaled Maâmra, Laurence Pilard, and Devan Sohier. A self-stabilizing algorithm for maximal matching in anonymous networks. *Parallel Processing Letters*, 26(04):1650016, 2016.

[35] Johanne Cohen, George Manoussakis, Laurence Pilard, and Devan Sohier. A self-stabilizing algorithm for maximal matching in link-register model in $o(nΔˆ3)$ moves. *CoRR*, abs/1709.04811, 2017.

[36] C. Colbourn, W. Kocay, and D. Stinson. Some NP-complete problems for hypergraph degree sequences. *Discrete Applied Mathematics*, 14:239–254, 1986.

[37] C. Comin and R. Rizzi. An improved upper bound on maximal clique listing via rectangular fast matrix multiplication. *arXiv preprint arXiv:1506.01082*, 2015.

[38] A. Conte, R. Grossi, A. Marino, and L. Versari. Sublinear-space bounded-delay enumeration for massive network analytics: Maximal cliques. In *43rd International Colloquium on Automata, Languages, and Programming (ICALP 2016)*, volume 148, pages 1–148, 2016.

[39] T. H. Cormen. *Introduction to algorithms*. MIT press, 2009.

[40] G. Dantzig. *Linear programming and extensions*. Princeton university press, 2016.

[41] G. B. Dantzig. *Origins of the simplex method*. ACM, 1990.

[42] A. K. Datta, L. L. Larmore, and T. Masuzawa. Maximum matching for anonymous trees with constant space per process. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 46. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[43] A. Del Pia and C. Michini. On the diameter of lattice polytopes. *Discrete and Computational Geometry*, 55:681–687, 2016.

[44] A. Deza, G. Manoussakis, and S. Onn. Primitive zonotopes. *Discrete & Computational Geometry*, Feb 2017.

[45] A. Deza and L. Pournin. Improved bounds on the diameter of lattice polytopes. *arXiv:1610.00341*, 2016.

[46] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, 1974.

[47] S. Dolev. *Self-Stabilization*. MIT Press, 2000.

[48] S. Dolev, A. Israeli, and S. Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *dc*, 7:3–16, 1993.

[49] F. Dorn. Planar Subgraph Isomorphism Revisited. In *27th International Symposium on Theoretical Aspects of Computer Science*, volume 5 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 263–274, Dagstuhl, Germany, 2010. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[50] D. E. Drake and S. Hougardy. A simple approximation algorithm for the weighted matching problem. *Inf. Process. Lett.*, 85(4):211–213, 2003.

[51] D. Eppstein. Subgraph isomorphism in planar graphs and related problems. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '95, pages 632–640, Philadelphia, PA, USA, 1995. Society for Industrial and Applied Mathematics.

[52] D. Eppstein. Zonohedra and zonotopes. *Mathematica in Education and Research*, 5:15–21, 1996.

[53] D. Eppstein, M. Löffler, and D. Strash. Listing all maximal cliques in large sparse real-world graphs. *J. Exp. Algorithmics*, 18:3.1:3.1–3.1:3.21, November 2013.

[54] P. Erdös and T. Gallai. Graphs with prescribed degrees of vertices (in Hungarian). *Matematikai Lopak*, 11:264–274, 1960.

[55] L. Euler. Solutio problematis ad geometriam situs pertinentis. *Commentarii Academiae Scientiarum Imperialis Petropolitanae*, 8:128–140, 1736.

[56] J. Evans. *Ancient Stone Implements, Weapons, and Ornaments, of Great Britain*. Cambridge Library Collection - Archaeology. Cambridge University Press, 2015.

[57] S. Fomin and N. Reading. Root systems and generalized associahedra. *arXiv preprint math/0505518*, 2005.

[58] E. C. Freuder. A sufficient condition for backtrack-free search. *Journal of the ACM (JACM)*, 29(1):24–32, 1982.

[59] K. Fukuda. cdd homepage. `https://www.inf.ethz.ch/personal/fukudak/cdd_home`.

[60] K. Fukuda. Lecture notes on oriented matroids and geometric computation. *Computational Complexity*, 2:7.

[61] K. Fukuda. Lecture notes: Polyhedral computation. `http://www-oldurls.inf.ethz.ch/personal/fukudak/lect/pclect/notes2015/`.

[62] F. C. Gaertner. A Survey of Self-Stabilizing Spanning-Tree Construction Algorithms. Technical report, 2003.

[63] E. J. Gardiner, P. Willett, and P. J. Artymiuk. Graph-theoretic techniques for macromolecular docking. *Journal of Chemical Information and Computer Sciences*, 40(2):273–279, 2000.

[64] B. Ghosh and S. Muthukrishnan. Dynamic load balancing by random matchings. *J. Comput. Syst. Sci.*, 53(3):357–370, 1996.

[65] E. N. Gilbert. Enumeration of labelled graphs. *Canad. J. Math*, 8(1):05–411, 1956.

[66] P.-L. Giscard, N. Kriege, and R. C. Wilson. A general purpose algorithm for counting simple cycles and simple paths of any length. *arXiv preprint arXiv:1612.05531*, 2016.

[67] P. M. Gleiss. Short cycles. 2001.

[68] G. Goel and J. Gustedt. Bounded arboricity to determine the local structure of sparse graphs. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 159–167. Springer, 2006.

[69] H. M. Grindley, P. J. Artymiuk, D. W. Rice, and P. Willett. Identification of tertiary structure resemblance in proteins using a maximal common subgraph isomorphism algorithm. *Journal of molecular biology*, 229(3):707–721, 1993.

[70] N. Guellati and H. Kheddouci. A survey on self-stabilizing algorithms for independence, domination, coloring, and matching in graphs. *Journal of Parallel and Distributed Computing*, 70(4):406 – 415, 2010.

[71] N. Guellati and H. Kheddouci. A survey on self-stabilizing algorithms for independence, domination, coloring, and matching in graphs. *J. Parallel Distrib. Comput.*, 70(4):406–415, 2010.

[72] D. Gusfield. *Algorithms on strings, trees and sequences: computer science and computational biology*. Cambridge University Press, 1997.

[73] F. Harary and I. C. Ross. A procedure for clique detection using the group matrix. *Sociometry*, 20(3):205–215, 1957.

[74] G. Hardy, E. Wright, D. Hearth-Brown, and J. Silverman. *An introduction to the theory of numbers*. Clarendon Press Oxford, 1979.

[75] S. T. Hedetniemi, D. Pokrass Jacobs, and P. K. Srimani. Maximal matching stabilizes in time o(m). *Inf. Process. Lett.*, 80(5):221–223, 2001.

[76] C. J. Henry and S. Ramanna. Maximal clique enumeration in finding near neighbourhoods.

[77] T. R. Herman. *A comprehensive bibliography on self-stabilization*. 2002.

[78] M. Hoefer. Local matching dynamics in social networks. *Inf. Comput.*, 222:20–35, 2013.

[79] J. E. Hopcroft and R. M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973.

[80] R. Horaud and T. Skordas. Stereo correspondence through feature grouping and maximal cliques. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(11):1168–1180, 1989.

[81] S.-C. Hsu and S.-T. Huang. A self-stabilizing algorithm for maximal matching. *Inf. Process. Lett.*, 43(2):77–81, 1992.

[82] J. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraphs in the presence of isomorphism. In *Data Mining, 2003. ICDM 2003. Third IEEE International Conference on*, pages 549–552. IEEE, 2003.

[83] J. Humphreys. *Reflection groups and Coxeter groups*. Cambridge Studies in Advanced Mathematics. Cambridge University Press, 1990.

[84] M. Inoue, F. Ooshita, and S. Tixeuil. An efficient silent self-stabilizing 1-maximal matching algorithm under distributed daemon without global identifiers. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems*, pages 195–212. Springer, 2016.

[85] A. Itai and M. Rodeh. Finding a minimum circuit in a graph. *SIAM Journal on Computing*, 7(4):413–423, 1978.

[86] Colette J. and L. Higham. Fault-tolerant implementations of atomic registers by safe registers in networks. In *Proceedings of the Twenty-Seventh Annual ACM Symposium on Principles of Distributed Computing, PODC 2008, Toronto, Canada, August 18-21, 2008*, page 449, 2008.

[87] T. R. Jensen and B. Toft. *Graph coloring problems*, volume 39. John Wiley & Sons, 2011.

[88] C. Johnen, I. Lavallee, and C. Lavault. Reliable self-stabilizing communication for quasi rendezvous. *arXiv preprint arXiv:1005.5630*, 2010.

[89] D. B. Johnson. Finding all the elementary circuits of a directed graph. *SIAM Journal on Computing*, 4(1):77–84, 1975.

[90] D. S. Johnson, M. Yannakakis, and C. H. Papadimitriou. On generating all maximal independent sets. *Information Processing Letters*, 27(3):119 – 123, 1988.

[91] G. Kalai and D. Kleitman. A quasi-polynomial bound for the diameter of graphs of polyhedra. *Bulletin of the American Mathematical Society*, 26:315–316, 1992.

[92] M. H. Karaata and K. A. Saleh. Distributed self-stabilizing algorithm for finding maximum matching. *Comput Syst Sci Eng*, 15(3):175–180, 2000.

[93] L. M. Kirousis and D. M. Thilikos. The linkage of a graph. *SIAM Journal on Computing*, 25(3):626–647, 1996.

[94] P. Kleinschmidt and S. Onn. On the diameter of convex polytopes. *Discrete Mathematics*, 102:75–77, 1992.

[95] C. Klivans and V. Reiner. Shifted set families, degree sequences, and plethysm. *Electronic Journal of Combinatorics*, 15 (1), 2008.

[96] D. Knuth. *Marriages stables et leurs relations avec d'autres problèmes combinatoires*. Les Presses de l'Université de Montréal, 1976.

[97] I. Koch. Enumerating all connected maximal common subgraphs in two graphs. *Theoretical Computer Science*, 250(1-2):1–30, 2001.

[98] I. Koch, T. Lengauer, and E. Wanke. An algorithm for finding maximal common subtopologies in a set of protein structures. *Journal of computational biology*, 3(2):289–306, 1996.

[99] L. Kowalik. Short cycles in planar graphs. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 284–296. Springer, 2003.

[100] M. Koyutürk, A. Grama, and W. Szpankowski. An efficient algorithm for detecting frequent subgraphs in biological networks. *Bioinformatics*, 20(suppl_1):i200–i207, 2004.

[101] L. Lamport. Solved problems, unsolved problems and nonproblems in concurrency. pages 1–11, August 1984.

[102] D. R. Lick and A. T. White. d-degenerate graphs. *Canad. J. Math.*, 22:1082–1096, 1970.

[103] R. Liu. Nonconvexity of the set of hypergraph degree sequences. *Electronic Journal of Combinatorics*, 20 (1), 2013.

[104] K. Makino and T. Uno. New algorithms for enumerating all maximal cliques. Springer.

[105] F. Manne and M. Mjelde. A self-stabilizing weighted matching algorithm. In *9th Int. Symposium Stabilization, Safety, and Security of Distributed Systems (SSS)*, Lecture Notes in Computer Science, pages 383–393. Springer, 2007.

[106] F. Manne, M. Mjelde, L. Pilard, and S. Tixeuil. A new self-stabilizing maximal matching algorithm. *Theoretical Computer Science (TCS)*, 410(14):1336–1345, 2009.

[107] F. Manne, M. Mjelde, L. Pilard, and S. Tixeuil. A self-stabilizing 2/3-approximation algorithm for the maximum matching problem. *Theoretical Computer Science (TCS)*, 412(40):5515–5526, 2011.

[108] G. Manoussakis. Listing all fixed-length simple cycles in sparse graphs in optimal time. In *Fundamentals of Computation Theory - 21st International Symposium, FCT 2017, Bordeaux, France, September 11-13, 2017, Proceedings*, pages 355–366, 2017.

[109] E. M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, April 1976.

[110] K. Meeks. Randomised enumeration of small witnesses using a decision oracle. In *11th International Symposium on Parameterized and Exact Computation, IPEC 2016, August 24-26, 2016, Aarhus, Denmark*, pages 22:1–22:12, 2016.

[111] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. Network motifs: simple building blocks of complex networks. *Science*, 298(5594):824–827, 2002.

[112] J. W Moon and L. Moser. On cliques in graphs. *Israel journal of Mathematics*, 3(1):23–28, 1965.

[113] D. R. Morrison. Patricia-practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, 1968.

[114] A. Mostéfaoui, M. Petrolia, M. Raynal, and C. Jard. Atomic read/write memory in signature-free byzantine asynchronous message-passing systems. *Theory Comput. Syst.*, 60(4):677–694, 2017.

[115] N.L. Bhanu Murthy and Murali K. Srinivasan. The polytope of degree sequences of hypergraphs. *Linear Algebra and its Applications*, 350:147–170, 2002.

[116] D. Naddef. The Hirsch conjecture is true for $(0,1)$-polytopes. *Mathematical Programming*, 45:109–110, 1989.

[117] L. Pan and E. E. Santos. An anytime-anywhere approach for maximal clique enumeration in social network analysis. In *Systems, Man and Cybernetics, 2008. SMC 2008. IEEE International Conference on*, pages 3529–3535. IEEE, 2008.

[118] C. H. Papadimitriou and M. Yannakakis. The clique problem for planar graphs. *Information Processing Letters*, 13(4):131 – 133, 1981.

[119] M. C. Paull and S. H. Unger. Minimizing the number of states in incompletely specified sequential switching functions. *IRE Transactions on Electronic Computers*, (3):356–367, 1959.

[120] J. L. Pfaltz. Chordless cycles in networks. *2013 IEEE 29th International Conference on Data Engineering Workshops (ICDEW)*, pages 223–228, 2013.

[121] R. Preis. Linear time 1/2-approximation algorithm for maximum weighted matching in general graphs. In *16th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, Lecture Notes in Computer Science, pages 259–269. Springer, 1999.

[122] D. Richards. Finding short cycles in planar graphs using separators. *Journal of Algorithms*, 7(3):382 – 394, 1986.

[123] R. Samudrala and J. Moult. A graph-theoretic algorithm for comparative modeling of protein structure. *Journal of molecular biology*, 279(1):287–302, 1998.

[124] F. Santos. A counterexample to the Hirsch conjecture. *Annals of Mathematics*, 176:383–412, 2012.

[125] T. Schank and D. Wagner. Finding, counting and listing all triangles in large graphs, an experimental study. Springer.

[126] N. Shervashidze, S. Vishwanathan, T. Petri, K. Mehlhorn, and K. Borgwardt. Efficient graphlet kernels for large graph comparison. In *Artificial Intelligence and Statistics*, pages 488–495, 2009.

[127] K. Shin, T. Eliassi-Rad, and C. Faloutsos. Patterns and anomalies in k-cores of real-world graphs with applications. *Knowledge and Information Systems*, pages 1–34, 2017.

[128] N. Sloane (editor). The on-line encyclopedia of integer sequences. `https://oeis.org`.

[129] N. Sokhn, R. Baltensperger, L.-F. Bersier, J. Hennebert, and U. Ultes-Nitsche. Identification of chordless cycles in ecological networks. Springer.

[130] I. Soprunov and J. Soprunova. Eventual quasi-linearity of the Minkowski length. *European Journal of Combinatorics*, 58:110–117, 2016.

[131] N. Sukegawa. Improving bounds on the diameter of a polyhedron in high dimensions. *arXiv:1604.04338*, 2016.

[132] R. Tarjan. Enumeration of the elementary circuits of a directed graph. *SIAM Journal on Computing*, 2(3):211–216, 1973.

[133] G. Tel. *Introduction to distributed algorithms*. Cambridge university press, 2000.

[134] T. Thiele. Extremalprobleme für Punktmengen. *Diplomarbeit, Freie Universität Berlin*, 1991.

[135] S. Tixeuil. Self-stabilizing Algorithms. In *Algorithms and theory of computation handbook*, pages 26.1–26.45. Chapman & Hall/CRC, 2009. 50 pages.

[136] M. Todd. An improved Kalai-Kleitman bound for the diameter of a polyhedron. *SIAM Journal on Discrete Mathematics*, 28:1944–1947, 2014.

[137] E. Tomita, A. Tanaka, and H. Takahashi. The worst-case time complexity for generating all maximal cliques and computational experiments. *Theor. Comput. Sci.*, 363(1):28–42, October 2006.

[138] M. Touati, R. El-Azouzi, M. Coupechoux, E. Altman, and J. M. Kelif. Controlled matching game for user association and resource allocation in multi-rate wlans? In *2015 53rd Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pages 372–380, Sept 2015.

[139] H. M. Trent. A note on the enumeration and listing of all possible trees in a connected linear graph. *Proceedings of the National Academy of Sciences*, 40(10):1004–1007, 1954.

[140] S. Tsukiyama, M. Ide, H. Ariyoshi, and I. Shirakawa. A new algorithm for generating all the maximal independent sets. *SIAM Journal on Computing*, 6(3):505–517, 1977.

[141] V. Turau and B. Hauck. A new analysis of a self-stabilizing maximum weight matching algorithm with approximation ratio 2. *Theoretical Computer Science (TCS)*, 412(40):5527–5540, 2011.

[142] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[143] T. Uno. An efficient algorithm for solving pseudo clique enumeration problem. *Algorithmica*, 56(1):3–16, 2010.

[144] T. Uno and H. Satoh. An efficient algorithm for enumerating chordless cycles and chordless paths. In *International Conference on Discovery Science*, pages 313–324. Springer, 2014.

[145] L. G. Valiant. The complexity of computing the permanent. *Theoretical computer science*, 8(2):189–201, 1979.

[146] K. Wasa. Enumeration of enumeration algorithms. *arXiv preprint arXiv:1605.05102*, 2016.

[147] P. Weiner. Linear pattern matching algorithms. In *Switching and Automata Theory, 1973. SWAT '08. IEEE Conference Record of 14th Annual Symposium on*, pages 1–11, Oct 1973.

[148] D. B. West et al. *Introduction to graph theory*, volume 2. Prentice hall Upper Saddle River, 2001.

[149] H. Whitney. Congruent graphs and the connectivity of graphs. *American Journal of Mathematics*, 54(1):150–168, 1932.

[150] V. V. Williams and R. Williams. Finding, minimizing, and counting weighted subgraphs. *SIAM Journal on Computing*, 42(3):831–854, 2013.

[151] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *Data Mining, 2002. ICDM 2003. Proceedings. 2002 IEEE International Conference on*, pages 721–724. IEEE, 2002.

[152] M. J. Zaki, S. Parthasarathy, M. Ogihara, W. Li, et al. New algorithms for fast discovery of association rules.

[153] G. Ziegler. *Lectures on Polytopes*. Graduate Texts in Mathematics. Springer, 1995.

[154] A. Zomorodian. The tidy set: a minimal simplicial set for computing homology of clique complexes. In *Proceedings of the twenty-sixth annual symposium on Computational geometry*, pages 257–266. ACM, 2010.