Asynchronous Self-Stabilizing Stable Marriage

Marie Laveau

université
PARIS-SACLAY

Asynchronous Self-stabilizing Stable Marriage

Thèse de doctorat de l'université Paris-Saclay

École doctorale n° 580, Sciences et Technologies de l'Information et de la Communication (STIC)
Spécialité de doctorat : Informatique
Unité de recherche : Université Paris-Saclay, CNRS, Laboratoire de recherche en informatique, 91405, Orsay, France
Référent : Faculté des sciences d'Orsay

Thèse présentée et soutenue à Orsay, le 30 Septembre 2020, par

Marie LAVEAU

Composition du jury :

Johanne Cohen Directrice de Recherche, Université Paris-Saclay (LRI)	Présidente
Colette Johnen Professeure, Université Bordeaux (LaBRI)	Rapporteure & Examinatrice
Volker Turau Professeur, Hamburg University of Technology (Institut für Telematik)	Rapporteur & Examinateur
Hugues Fauconnier Professeur, Université de Paris (IRIF)	Examinateur
Sébastien Tixeuil Professeur, Sorbonne Université (LIP6)	Examinateur
Joffroy Beauquier Professeur émérite, Université Paris-Saclay (LRI)	Directeur de thèse
Thibault Bernard Maître de conférences, Université de Reims (Li-PaRAD, UP-Saclay)	Co-encadrant & Examinateur
Janna Burman Maîtresse de conférences (HDR), Université Paris-Saclay (LRI)	Co-encadrante & Examinatrice

Thèse de doctorat

NNT: 2020UPASG008

Résumé

Titre : Mariage Stable Asynchrone et Auto-stabilisant

Mots-clés : Algorithmes Distribués, Modèles Asynchrones, Auto-stabilisation, Mariage Stable, Complexité en Moves, Démon inéquitable, Confidentialité

Le *Problème du Mariage Stable* (SMP) est un problème d'appariement où les participants ont des préférences à propos de leurs partenaires potentiels. L'objectif est de trouver un appariement optimal (stable dans un sens) au regard des préférences. Ce type d'appariement a de très nombreuses applications comme les affectations d'étudiants à des universités (APB ou ParcourSup), celles des internes en médecine aux hôpitaux, les choix des donneurs pour les patients en attente d'organe, la mise en rapport des taxis et de leurs clients ou encore la diffusion de contenu sur Internet. Certaines de ces applications peuvent être traitées de manière centralisée tandis que d'autres, de par leur nature distribuée et la complexité de leurs données, nécessitent un traitement différent. Par exemple, dans le contexte du Cloud-Computing, des machines virtuelles sont émulées par des machines réelles situées sur la terre entière. Un algorithme centralisé causerait des délais considérables dans les prises de décision tout en étant sensible aux défaillances, ce qui est inconcevable pour un service supposé disponible à tout moment.

D'un autre côté, chaque fois que des personnes sont impliquées dans un appariement, elles ont le droit de garder privées leurs données personnelles et en particulier leur liste de préférences, qui peut contenir des informations sensibles. Par conséquent, il est souhaitable que les listes de préférence des personnes ne soient jamais transmises sur Internet, et encore moins rassemblées pour un traitement centralisé. C'est pourquoi la distribution, la tolérance aux défaillances (par auto-stabilisation) et la confidentialité sont les trois principaux mots-clés de cette thèse.

Dans ce contexte, nous proposons deux solutions distribuées *auto-stabilisantes*. De telles solutions tolèrent les défaillances (*e.g.*, corruptions de mémoire ou de messages) transitoires (ou de courte durée) de n'importe quels noeuds. La confidentialité des listes de préférences est garantie par les deux algorithmes que nous proposons : les listes ne sont pas partagées et seules des queries binaires et leurs réponses sont échangés. Une différence entre ces algorithmes est le modèle de communication : le premier algorithme utilise le *modèle à état* tandis que le second algorithme utilise le *modèle à registre* plus général. Dans les deux modèles, les exécutions se déroulent par pas atomiques et un démon (*démon distribué inéquitable*) exprime la notion d'asynchronisme. Avec ce démon, le temps de stabilisation peut être borné en terme de *moves* (pas locaux). Cette mesure de complexité permet d'évaluer avec précision la puissance de calcul nécessaire ou l'énergie dissipée par les exécutions de l'algorithme. Ce n'est pas le cas quand la complexité est évaluée en *rounds*, puisque le nombre de moves effectués dans un round n'est pas nécessairement borné.

Le premier algorithme, basé sur la méthode centralisée de Ackermann *et al.* (SICOMP' 2011), résout le SMP en $O(n^4)$ moves.

Le point de départ du deuxième algorithme est le schéma de détection locale/correction globale de Awerbuch *et al.* (DA' 1994) : un algorithme non auto-stabilisant (devant être

initialisé) mais avec la propriété d'être *vérifiable localement* peut être combiné avec un détecteur et un algorithme de réinitialisation. De cette combinaison résulte un algorithme auto-stabilisant. Malheureusement, la définition de la vérifiabilité locale de DA' 1994 ne s'applique pas à notre cas (en particulier en raison du démon inéquitable). Nous proposons donc une nouvelle définition. De plus, nous concevons un algorithme de réinitialisation (reset) asynchrone, distribué et auto-stabilisant. L'algorithme résultant résout le SMP en $\Theta(n^2)$ moves.

Nous adaptons ces deux algorithms pour résoudre certaines variantes du SMP telles que le mariage stable avec indifférence, avec partenaires inacceptables, *etc.*

Abstract

Title: Asynchronous Self-stabilizing Stable Marriage

Keywords: Distributed Algorithms, Asynchronous Model, Self-stabilization, Stable Marriage, Move Complexity, Unfair Daemon, Privacy

The *Stable Marriage Problem* (SMP) is a matching problem where participants have preferences over their potential partners. The objective is to find a matching that is optimal (stable in certain sens) with regard to these preferences. This type of matching has a lot of widely used applications such as the assignment of children to schools, interns to hospitals, kidney transplant patients to donors, as well as taxi scheduling or content delivery on the Internet. Some applications can be solved in a centralized way while others, due to their distributed nature and their complex data, need a different treatment. For example, when applying this problem to the Cloud-Computing context, virtual machines are emulated by real machines located all over the world. A centralized algorithm would cause unbearable delays and be sensible to failures, which is inconceivable for a service meant to be available at any time.

On the other hand, when humans are to be matched or involved in a matching, they have the right to keep their personal data private and in particular their list of preferences. Consequently, the preference lists should not be transmitted on the Internet, and even less gathered for a centralized treatment. This is why, distribution, fault-tolerance (by self-stabilization) and privacy are the three main keywords of this thesis.

In order to handle these challenges, we provide two distributed *self-stabilizing* solutions. Such solutions tolerate transient (or short-lived) failures (*e.g.*, memory or message corruptions) of any nodes. The privacy of the preference lists is guaranteed by the two proposed algorithms: lists are not shared, only some binary queries and responses are transmitted. One of the differences between the two algorithms is the communication model: the first algorithm uses the *state model* while the second algorithm uses the more general *register model*. In both models, executions proceed in atomic steps and a daemon (*distributed unfair daemon*) conveys the notion of asynchrony. Under this daemon, the *stabilization time* can be bounded in term of *moves* (local computations). This complexity metrics allows to evaluate the necessary computational power or the energy consumption of the algorithm's executions. This is not the case when the stabilization time is measured in *rounds* since an unbounded number of moves may be executed during a round.

The first algorithm, based on the centralized method of Ackermann *et al.* (SICOMP' 2011), solves the problem in $O(n^4)$ moves.

The starting point of the second algorithm is the local detection/global correction scheme of Awerbuch *et al.* (DA' 1994): a non-self-stabilizing algorithm (with initialization) that satisfies the property of *local checkability* can be combined with a detector and a reset algorithms. The result of this composition is a self-stabilizing version of the given algorithm. Unfortunately, local checkability definition of DA '1994 does not apply to our case (in particular due to the unfair daemon). Consequently, we propose a new definition. Furthermore, we design a distributed self-stabilizing asynchronous reset

algorithm. Using it, the resulting composed algorithm solves SMP in $\Theta(n^2)$ moves in a self-stabilizing way.

We adapt both algorithms to solve some variants of SMP such as the stable marriage with indifference, with unacceptable partners, *etc.*

Contents

Chapter 1

Introduction

"Matching under preferences is a topic of great practical importance, deep mathematical structure, and elegant algorithmics."
Kurt Melhorn in [Man13]

Stable Marriage is a matching problem where the participants have preferences over their potential partners. The objective is to find a matching (*i.e.*, an assignment of the participants to one another) that is optimal (stable in a certain sense) with regards to these preferences. This type of matching has a lot of widely used applications such as the assignment of children to schools, interns to hospitals, kidney transplant patients to donors, as well as taxi scheduling or content delivery on the Internet. But in fact, matching under preferences is as old as civilizations. In some cases, the difficulty to solve this problem seemed to have caused serious trouble. As an example, if we go back to antiquity, Danaus had fifty daughters, the Danaides. After Aegyptus, his brother, commanded that his fifty sons should marry the Danaides, Danaus, maybe afraid of the difficulty of getting a matching which would satisfy everybody, elected to flee instead, and to that purpose, he built a ship, the first ship that ever was and fled to Argos. Naturally, the problem is still relevant nowadays. For example, APB (Admission Post-Bac) and ParcourSup, the applications intended to manage the student/university assignments in France use matching under preferences algorithms.

Matching under preferences is an important topic in the domain of economics, especially in market modelization. As a matter of fact, it is easy to relate the notion of preference to the notion of payoff. The preferred choice is associated to the one with the higher payoff and so on. That is why, since the beginning of its algorithmic study fifty years ago, the matching problem with preferences has received a large attention from the economy world. One of the actors in this domain, Lloyd Shapley, received in 2012 the Nobel Prize in economy, partly for his work on matching under preferences. Later, strong relations with game theory have been established: the solutions achieve a pure Nash equilibrium in a cooperative game. Lloyd Shapley and David Gale, in a famous article [GS62], introduced in its current form an instance of matching under preferences by using the metaphor of marriage. There are women and an equal number of men. Each accepts to be married with a person of the other sex. Each participant, woman or man, has a complete list of preferences on the persons of the other sex. The problem is to get them married in such a way that the matching is *stable*. In this example, stable means that there does not exist a pair of a woman and a man, each married to other partners but that prefer each other. If there is no such unmarried pair, there is always somebody in a married pair who has no interest, in terms of preference, in changing her or his partner (see Chap. 3 for formal definition). Gale and Shapley called this problem *Stable Marriage* and this name is still used to define the form of matching under

preferences. Stable marriage is a model for simple markets (like producers/consumers) but when modeling real world interactions, more complicated situations arise. For this reason, many variants of the problem have since been developed and analyzed. We study some of these variants (*e.g.*, stable marriage with ties, with incomplete list, ...) in Chapter 6.

Although being one of the most studied topics in algorithmics during the last decades, matching under preferences stays a very important subject, because of its universality. A deep theory has been developed, numerous theorems have been established, several books on the topic have been published [GI89, RS90, Man13], Donald Knuth wrote a monograph [Knu76] on the topic, and yet the subject is still alive. Indeed, emerging networks and in particular the Internet puts this problem in a completely different perspective and scale. Fifty years ago, the issue was to find algorithmic solutions to handle very hard problems. When one had to match several thousands of medical students, each with its own preferences, to hundreds of hospitals, each with its own criteria, in a way where everybody was satisfied, the solutions by hand were not feasible. Thanks to the advent of computers, a satisfying output was obtained: all the complex data was fed into one machine that calculated the matching. This centralized way of computing the data is still used, but there are new situations (*i.e.*, with distributed data/computation power) that require a different treatment.
As an example, consider a typical Cloud. There are real machines, the computers, located in data centers all over the world, emulating virtual machines and able to give a personalized service to the consumers. An issue is: on which real machine(s) to run the virtual machines in order to have the best performance? But also: where to migrate them depending on the load of the real machines in real time? This type of problem corresponds to the matching under preferences. A centralized solution would cause unbearable delays in the decisions and create bottlenecks. Such a solution would have no failure tolerance, what is inconceivable for a service that is supposed to be available at any time. Due the nature of the problem, solution here must be distributed: the different data centers have to take their decisions locally, from their own information and the information received from other data centers. On the other hand, consider examples of matching under preferences involving humans, like residents-to-hospitals assignments or students-to-universities matchings (*e.g.*, ParcourSup or APB). People have the right to keep their personal data private and in particular their list of preferences: such a list may give a lot of sensible information on a person. The risk is that connections and deductions could be made from participant's preference lists leading to a loss of privacy. Consequently the preference lists of persons should not be transmitted on the Internet, and even less gathered for a centralized treatment.

That is why we are aiming at solutions based on matching under preferences, which are completely distributed, tolerate a certain type of failures and respect the confidentiality of the lists. To the best of our knowledge, it is the first time that these three notions are incorporated together for solving the problem.

In order to handle faults, we provide *self-stabilizing* distributed solutions. Such solutions tolerate transient (or short-lived) failures (*e.g.*, memory or message corruptions) of any number of nodes. That means that after any number of such failures (corrupting nodes' memory), which bring the system into an arbitrary state, the algorithm must recover from these failures (automatically - without any intervention). Meanwhile, the

constants and the code are assumed to be untouched. Thus a self-stabilizing solution solves a problem whatever starting configuration (the global system state) is (see a formal definition in the models and definitions Chapter 3, Section VI). This property is particularly interesting for Cloud and Internet based applications in general, since they frequently require some level of self-stabilization.

We consider two models of communication commonly used for self-stabilizing algorithms: the *state model* (*cf.* [Dij74]) and the *register model* (*cf.* [DIM93]). In both models, executions proceed in atomic *steps* and a *daemon* conveys the notion of asynchrony. We assume the strongest (and the most general) daemon (adversary), the *distributed unfair* daemon. It models a very high level of asynchrony. In particular, it may keep a node from being activated as long as other nodes are *eligible* for activation (have instructions to execute). Though being general, designing and analyzing algorithms under such daemon is more difficult than under fair daemons. The natural time measure with unfair daemon is in terms of *steps* or *moves* (local computations). This is also practically relevant metrics since all local actions are counted until *stabilization*, allowing the computational power and the energy consumption being evaluated (*cf.* [BA16]). Indeed, with other deamons such as the *weakly fair* one (often used for self-stabilizing algorithms), the stabilization time is measured in term of *rounds*. In a round, all eligible nodes are activated at least once. That is, under an unfair daemon, a round may contain an unbounded number of moves. Thus, a round analysis may lead to a too coarse-grained evaluation of the energy and computational consumption. Hence, a polynomial algorithm in term of rounds can be proved exponential in term of steps/moves. For example, a silent leader election with a linear round complexity [DLV11a, DLV11b] has been proved to be exponential [ACD+17] in steps. Huang and Chen's BFS Algorithm [HC92] has an exponential lower bound in steps [DJ16]. Similarly, in [GHIJ19], they prove that the step lower bound of their algorithm [GHIJ14] is exponential, while the upper bound in rounds is linear. Moreover, the step/move distributed analysis can be compared with the analysis of some centralized solutions for stable marriage where the evaluations are in terms of queries or messages (*cf.* [OR15]).

Contributions and Roadmap

The Stable Marriage Problem (SMP) is defined on the complete bipartite graph $K_{n,n}$ (one set of women and one of men). Each node u has a different priority for each node v in the other set, between 1 (the most preferred) and n. The goal is: (i) to match (marry) the women and the men together such that everyone is matched, in a way that (ii) there is no pair of a woman and a man that are not matched to each other, but prefer each other over their current matches. When there are no such pairs, called *blocking pairs* (BPs), the set of marriages is said *stable*.

In this work, we consier *decentralized distributed* settings, where the bipartite graph represents a communication network. Edges represent the communication links and nodes are computing entities (to be matched), having unique identifiers. Each node has only a partial information about the problem instance, contrary to the centralized case. In particular, it is assumed to be initially aware only of its own preferences, but not of the other nodes' preferences. In addition, to ensure confidentiality of the

preferences [BM05] and avoid high message complexity, we follow the previous related studies and rule out a trivial solution where nodes exchange their preference lists and then run a known centralized solution at each node.

Contributions. The present work aims to design algorithms for SMP in a distributed and self-stabilizing fashion but without exchanging preferences. Since a self-stabilizing algorithm runs from any configuration to a configuration satisfying the problem, nodes have to detect and manage the BPs. Unfortunately, in his monograph [Knu76], Knuth notices that resolving locally the BPs one after the other can lead to an infinite cycle of actions. The first step towards self-stabilization was made by Ackermann *et al.* [AGM+11] who gave a centralized algorithm that solves locally the BPs in a particularly synchronized way avoiding cycling. Expressed in a distributed setting, this solution is not self-stabilizing, since even if the initial matching may be arbitrary, some variables are required to be initialized. Our first main contribution is a self-stabilizing solution to SMP in the distributed state model. This solution adopts the principle of (two) synchronized phases present in the Ackemann *et al.* solution, but has to add an additional phase for a distributed synchronization and a local management of variables ensuring the self-stabilization property. We present a formal analysis of the complexity in moves of the solution. The Gale and Shapley's Algorithm (GSA) is known to terminate in $O(n^2)$ rounds, but also in $O(n^2)$ moves. The Ackermann *et al.* centralized solution is of $O(n^2)$ moves too. Our distributed solution is of $O(n^4)$ moves [LMB+17]. This raises the issue of the gap between the relevant centralized and distributed solutions. Note that an independent result [OR15] gives a lower bound for the *communication complexity* of $\Omega(n^2)$ bits, for solving the same problem (in the two parties communication setting using a single bidirectional link). However, this lower bound implies only $\Omega(n^2/\log n)$ moves in the communication model here (assuming constant size registers, used by our solutions).

This leads to the second main contribution of the thesis which is a distributed self-stabilizing solution for SMP in $\Theta(n^2)$ moves. Moreover we tackle this time the problem in a more general communication model of shared registers. In this model, instead of reading the states of the neighbors directly, a node can only access the designated per neighbor shared registers. This solution involves several stages. The starting point is the local detection/global correction scheme of [APSVD94] inspired by [AKY90]. The authors proved that if an algorithm $\mathcal{A}$ with initialization is *locally checkable*, it can be combined with a detector and a reset modules yielding a self-stabilizing algorithm. The basic idea is to launch a reset over the system, when a BP is locally detected (considering it as an abnormal situation). After resetting, the reached configuration is an initial configuration of $\mathcal{A}$. Unfortunately, the local checkability in [APSVD94] does not apply to our case even though BPs are locally detectable. Indeed, in [APSVD94], the local checkability is defined under fair assumptions. But an unfair daemon can choose to keep nodes (*e.g.*, those detecting faults) unactivated as long as other nodes are eligible, and in general even forever, avoiding the detection of faults/"anomalies" preventing the correct stabilization. Thus, we propose a new definition of local checkability with, in particular, a termination condition.

In addition, we design a self-stabilizing distributed asynchronous reset algorithm. This algorithm uses a rooted spanning tree that must be built in a self-stabilizing way

too. Hence, we build a rooted spanning tree of depth 2. The reset algorithm has a stabilization time of $O(n \cdot 6^p)$ moves (where p is the depth of the tree). There exist, in the literature, self-stabilizing reset algorithms functioning over a spanning tree [GM91, AO94, KA98, DJ19]. However, there is no one coming with move complexity analysis while running under unfair scheduler with shared registers communication model.

Finally, we propose an asynchronous version of GSA in the distributed link register model [BBB+18] and we prove its local checkability according to our new definition. Then we put all the modules together and we prove that the algorithm resulting from the composition (with the tree construction, the reset and the detector) is correct and has a stabilization time of $\Theta(n^2)$ moves.

Other contributions of less importance are adaptations of the solutions for SMP to some variants of the problem: sets of unequal size, indifference, incomplete lists, many-to-one and many-to-many matchings. We want to emphasize that the results that we present are obtained under the so called unfair scheduler, which is the most difficult setting to deal with. An unfair scheduler may never choose to activate a process, unless this process is the only one to be eligible.

Roadmap Following this introduction, Chapter 2 proposes an overview of the historical background for SMP and self-stabilization. In Chapter 3, concepts are detailed and the models used in this thesis are defined. In particular, we define the stable marriage problem, self-stabilization and communication models (state reading and link register). Following these two chapters, we present our contributions.

Chapter 4: A solution based on Ackermann *et al.* phases.
In this chapter, we propose a self-stabilizing stable marriage solution, in the model of composite atomicity (state reading model), under an unfair distributed scheduler [LMB+17]. This algorithm is the first self-stabilizing algorithm for SMP. It solves the problem in $O(n^4)$ moves/steps (formal proofs of correctness and complexity are given).

Chapter 5: Local Checkability and Reset.
In this chapter we propose an adapted version of the local checkability. In particular, this definition can be used with the unfair daemon. In order to solve SMP, we provide `Async-GSA`, a distributed asynchronous version of GSA (see Section III.1) in the register model, under an unfair distributed scheduler [BBB+18]. The proof of the local checkability (of `Async-GSA`) is done as well as its correctness and complexity proof. The move and round complexity of `Async-GSA` is of $\Theta(n^2)$.
We also provide a self-stabilizing distributed reset with a self-stabilization time of $O(n)$ moves on a tree of depth 2. Finally, we compose `Async-GSA` with detection and reset modules in order to build a self-stabilizing algorithm for SMP. The composed algorithm solves the problem in $\Theta(n^2)$ moves.

Chapter 6: Extensions to Variants of SMP.
In this last chapter, we extend the two algorithms to some classical variants. The studied variants are basic extensions: unequal sizes of opposite sets, indifference (ties in prefer-

ence lists), unacceptable partners (incomplete lists), many-to-one matching (Hospitals-to-Residents problem) and the more difficult many-to-many matching. The modifications for each case are presented in such a way that they can be simply combined together to obtain a general algorithm solving all the considered variants. Furthermore, we explain why the complexity and correctness proofs remain (almost) the same.

Chapter 2

Related Work

Contents

In 1962, Gale & Shapley introduced the stable marriage problem in their seminal paper [GS62]. After this publication, a lot of works have been published about this problem in economics but also in mathematics and computer science: the problem (and its variants) has many applications in these areas [Bir17]. It can be viewed as a particular formulation of two sided matching markets and has been proved useful in many empirical approaches. For example, it is central for the solution of the large residents-to-hospitals assignment in USA (since 1952) or to the students-to-schools matching in Boston or New York. Later, in computer science, it was used to perform migrations of VMs or schedule taxis [BLAK14, KBW16] . Numerous books [Knu76, RS90, Irv94, Man13], book chapters [KMR16, Bir17, Cse17, Cec17] and surveys [Che19, CCM19] have been written on this topic until now. An International workshop on Matching Under Preferences (MATCH-UP) is devoted to the matching problem each two/three years [MAT]. Finally, a Dagstuhl seminar dedicated to matching under preferences was scheduled in July 2020 [Dag].

We distinguish centralized (Section I) and distributed solutions (Section II). The second section sums up the works regarding two different models: synchronous (Sub-section II.1) and asynchronous (Sub-section II.2).

Finally, since we focus on self-stabilization, we summarize main self-stabilizing results in Section III, first in a centralized context (Sub-section III.1), then concerning transformers (Sub-section III.2).

I - Centralized Solutions

In their paper, Gale and Shapley [GS62] provided a centralized algorithm (working by *deferred acceptances*, see Chapter 5, Section III for explanations) running in $O(n^2)$ time, which is proved to be asymptotically optimal (for centralized algorithms) in [NH90, Seg07]. Gale and Shapley proved also that there always exists at least one stable marriage in any system. The first application of this algorithm was for the allocation of

graduating medical students in US hospitals. Still now, the national Resident Matching program uses (an extension of) this algorithm [NRM] and handle now over 40,000 applicants. Now, stable marriage algorithms are used to assign graduating medical students to residency programs at hospitals also in Canada and Scotland. Similar mechanism are used to assign students to schools and universities in Norway and Singapore (*cf.* [TST99, Gol06]) but also in Boston [APRS05] and New York City [APR05]. More recently, the french students-to-universities matching (APB, ParcourSup) uses also such algorithms.

Notice that, in [Gol06], Golle takles the problem of privacy and cheating in such systems: with the knowledge of all the preference lists, a participant can manipulate the algorithm and change the output matching [GS85, GI89, TST99]. Golle proposes a private algorithm based on the Gale and Shapley algorithm (preference lists are kept secret). Similarly, Doerner *et al.* in [DES16] build a more efficient private algorithm (also based on Gale and Shapley algorithm) in order to apply it on large scale (for matching medical residents).

II - Distributed Solutions

Though different distributed problems have been well studied since decades, works on distributed stable marriage appeared much later than the centralized studies of this problem. Among these studies, theoretical ones consider an idealized *synchronous* distributed communication model, where nodes' progress in a lock-step manner, exchanging information and performing computations *all* together at each step (called *round*). These works focus on round complexity of the problem and its variants. On the contrary, studies with application cases consider an *asynchronous* distributed communication model, where there is no bound on message delivery, channel capacities or relative process speeds. That is, there is no global clock and nodes will eventually perform their computations. These works focus on applications such as Cloud-Computing or content delivery.

II.1 - Synchronous Model

Kipnis and Patt-Shamir [KPS09] prove a lower bound of $\Omega(\sqrt{(n/B \log n)})$ rounds, where B is the number of bits per message, and provide an algorithm that solves the distributed stable marriage in $O(n^2)$ rounds. Searching for better time complexity and conditions that can provide it, many studies consider specific restrictions on the preference lists such as *weighted stable marriage* [AGL10], incomplete or bounded lists [FKPS10, OR15], "almost regular" lists [OR15] and "similarity" in preference lists [KW16]. Still for improving time complexity, approximate versions have been considered (*e.g.*, [KPS09, FKPS10, GNOR15, OR15]), reaching a polylogarithmic time. Furthermore, when assuming strict restrictions on preference lists, approximate stable marriage can be solved even in constant time (*cf.* [FKPS10, OR15]). Notice also several bound results on communication complexity and step complexity (*cf.* [Seg07, CL10, GNOR15]). In particular, Gonczarowski [GNOR15] proved a lower bound on communication complexity: $\Omega(n^2)$ Boolean queries or bits have to be exchanged between two distributed parties, when one

(Alice) holds all the preference lists of the one set (women) and the other party (Bob) holds all the preference lists of the opposite set (men), and they want to solve SMP with these preferences.

II.2 - Asynchronous Model

The first paper considering an asynchronous model in distributed settings is [BM05]. Brito and Meseguer propose an extended version of GSA (with proposals and acceptances), but the participants who receive and accept a proposal delete all the worse ranked participants in its preference list. Thus, the preference lists are not communicated, reenforcing the privacy of the data. This algorithm is intended to work in the message passing model and is asynchronous. It is provided with empirical results but no complexity analysis. In the domain of Cloud computing, stable marriage is used for performing efficient migration of virtual machines to servers (*e.g.*, [XL11a, KL14]). Notice that the Xu and Li's algorithm [XL11a] is an extension of the Brito et Meseguer's algorithm. Content delivery networks that distribute much of the world's content and services have to solve a large and complex stable marriage problem between users and servers [XL11b, MS15]. Recently, in [YAB19], the authors apply the Brito' and Meseguer's algorithm for the scheduling of electric vehicle charging. Since charging takes a long time and the station depends on the demander's route, the scheduling must be done in-advance for efficient resource allocation. Furthermore, due to the frequent charging, driving patterns and preference lists can be divulged if the algorithm exchanges the input data. Thus, the algorithm must be dynamic, private and efficient. To compare their algorithm with the Gale and Shapley's algorithm, they made experiments and observe that their algorithm is more efficient and exchanges less message (than a adapted distributed version Gale and Shapley's algorithm).

III - Self-stabilizing Solutions

Introduced by Dijkstra in [Dij74, Dij86], self-stabilization is a property of distributed algorithms. A self-stabilizing algorithm for a problem P ensures that, from any configuration of the system (resulting from transient faults or not), the algorithm converges to the solution of P. Since many applications of stable marriage, especially in computer science, require failure tolerance[1], it is natural to look at a self-stabilizing solution (one of the failure tolerance schemes).

We first present works on the stable marriage that are related to self-stabilization in Sub-section III.1. There is no distributed, asynchronous and self-stabilizing solution for SMP but some results are helpful for our study.

A way of obtaining a self-stabilizing algorithm for a problem P is to combine a non-self-stabilizing algorithm for P with a transformer. Thus, in Sub-section III.2, we sum up such transformers with their pros and cons.

[1] Notice *e.g.* the tolerance demanding problem of scheduling charging points to electric vehicles on route, described in the previous section).

III.1 - On the Way to Self-stabilizing Stable Marriage Algorithms

Since self-stabilizing algorithms have to solve the problem from any configuration, a central issue is the resolution of blocking pairs (BPs). If a configuration contains an unstable marriage, the algorithm must detect BPs.

Knuth in a famous monograph [Knu76] puts forward this question by providing a problematic example. Indeed, when starting from an arbitrary configuration, it may exist a circular path of BPs resolutions (by matching the blocking pair and the previous partners) involving that an execution may cycle. In the self-stabilizing context, this cycle can lead to a non convergent execution (see Chapter 4, Section I, for explanations). In [RVV90], an open question raised by Knuth is answered: from any configuration containing an unstable matching, there exists at least one path of resolutions (by only matching the BPs' partners) that provides a stable matching. A consequence is that, under fair assumptions, from any unstable marriage there is a finite path of BPs resolutions that reaches a configuration with a stable marriage with probability one.

In [AGM+11], Ackermann *et al.* propose an algorithm for the stable marriage problem, starting from an arbitrary matching. This algorithm works in two phases. The first solves BPs in the same manner as Roth [RVV90], *i.e.* married women start solving their BPs. In the second phase, single women run GSA. This algorithm converges in polynomial time.

Finally, in [Mat07a, Mat07b, Mat09] the author investigates the so-called "self-stabilizing" stable marriage in peer-to-peer networks. The problem is restricted to acyclic preference lists (*i.e.* containing no cycle of peers such that each peer in the cycle prefers its successor to its predecessor). In this case, it is known that exactly one stable marriage exist. Moreover, the fact that there exists no cycle in the preference lists induces that Knuth's cycle cannot happen. Thus, starting in any given marriage, following any strategy of choosing and "fixing" blocking pairs, eventually results in a stable marriage. So, in this particular sense, the case of *acyclic* preferences is self-stabilizing. The authors prove that the number of such "fixes" (using any strategy) till stabilization can be exponential. However, if choosing the fixes in a round-robin fashion over the nodes' set, the number of fixes can be reduced to polynomial. Finally this study presents simulation results in P2P networks.

III.2 - Transformer to Self-stabilizing Solutions

There are several ways to transform a non-self-stabilizing algorithm into a self-stabilizing one. The first transformer was proposed by Katz and Perry [KP90]. It works for an asynchronous message passing system. Using a self-stabilizing snapshot [CL85], the protocol repeatedly evaluates the global state of the system. This evaluation is made by a fixed leader. If a "bad" global state is detected, a reset of the system is launched, restoring a pre-determined global state. This snapshot tool needs some synchronization (round numbers) and exchanges a lot of information (values of local variables and messages). The authors do not provide an upper bound on the number of messages. This approach uses global detection and global correction.

In [AKY90], a new approach is introduced: local detection and global correction. If

a node maintains locally some information so that it can detect bad configurations, a global reset can be launched. When the reset is terminated, the algorithm is restarted from a predefined configuration from which it is correct. If such a local detection is feasible, the algorithm is said to be *locally checkable*. Notice that the communication model uses Read/Write atomicity [DIM93] with a fair daemon.

Later, in [APSV91, Var93], another transformer with a local detection and local correction is proposed. A protocol is said to be *locally correctable* if the global state of the protocol can be corrected to a legitimate global state by applying independent local actions. The protocol uses the Input/Output Automata model with bounded channels [LT89].

Since many algorithms are not both locally checkable and correctable (most of the time only locally checkable), Awerbuch *et al.* [APSVD94] develop the local detection/global correction idea of [AKY90]. The global correction is made with a global reset. They formalize the definitions of local checkability and give an analysis of the combination: the stabilization time of the combined algorithm has an overhead of $O(R)$ asynchronous rounds where R is the response time of the reset.

The global correction is made through a global reset. An example of such an approach is given in [AG90], where a self-stabilizing reset uses a spanning tree built in a self-stabilizing way. The requirements are unique identifiers and the knowledge of a bound on the number of nodes in the network. The stabilization time of this protocol has been proved for asynchronous rounds with a fair daemon.

In this work, we adopt the idea of the latter transformer, by adapting it to our setting with the unfair daemon (Chapter 5). We also design an appropriate reset algorithm. All that with the goal to evaluate the move complexity of the transformed self-stabilizing algorithm to SMP in the link register model under unfair daemon. This allows us to obtain a better time complexity of only $O(n^2)$ moves in a more general (than the state reading) model.

Chapter 3

Models and Definitions

Contents

We first give a formal definition of the *blocking pairs* (BPs) and the *Stable Marriage* problem in Section I.

Then, we define *distributed systems* and *distributed algorithm* (Section II and III) and the notion of an *execution* in Section IV. Then, in Section V, we define the two different communication models. Our algorithm (Chapter 4) is designed for the *state model* while the second one (Chapter 5) is designed for the link register model.

Finally, we define the *self-stabilization* property of distributed systems in Section VI and the associated complexity metrics (Section VII).

I - The Stable Marriage Problem

The *stable marriage problem* has been introduced by Gale and Shapley in their seminal paper in 1962 [GS62]. We consider a set of men (Men) and a set of women (Women) such that each women w is given with a *priority* for each man m, denoted $priority(w, m)$, and reciprocally. The priorities go from 1 to n and the most preferred person has priority 1. In such a system, the goal is to match (marry) women and men together such that everyone is matched and there is no *blocking pair* (BP), *i.e.*, no unmarried pair (w, m) of a woman w and a man m, who both prefer each other to their current matches (partners). When there are no such pairs of nodes, the set of marriages is said to be *stable*. Formal definitions are given below.

Definition 1 (Blocking Pair (BP))**.** *Given a matching* $\mathcal{M} \subset$ Women $\times$ Men*, a pair* (w, m) *is a blocking pair iff the following conditions hold:*

1. *w and m are not matched together, i.e. $(w, m) \notin \mathcal{M}$;*
2. *w is married with m' and prefers m to m', i.e. $\exists\, m' : (w, m') \in \mathcal{M} \wedge priority(w, m) < priority(w, m')$;*
3. *m is married with w' and prefers w to w', i.e. $\exists\, w' : (w', m) \in \mathcal{M} \wedge priority(m, w) < priority(m, w')$.*

Definition 2 (Stable Marriage (SM))**.** *A matching $\mathcal{M} \subset$ Women $\times$ Men is a stable marriage iff $\mathcal{M}$ does not contain any blocking pair and every node is matched.*

Notice that Gale and Shapley proved that there always exists at least one stable marriage for any instance of preference lists, but there are possibly several. In the Table 3.1, we present an example of an SM instance. Let us first consider a matching $M_1 = \{(Jane, Mark), (Anna, Scott), (Zoe, John)\}$. In this matching, the pairs $(Zoe, Scott)$ and $(Jane, Scott)$ are BPs, *i.e.* M_1 is unstable. On the contrary, the matchings $M_2 = \{(Jane, Scott), (Anna, John), (Zoe, Mark)\}$ or $M_3 = \{(Jane, Mark), (Anna, John), (Zoe, Scott)\}$ are stable.

Jane	Scott	Mark	John	John	Zoe	Anna	Jane
Anna	John	Mark	Scott	Scott	Zoe	Jane	Anna
Zoe	Mark	Scott	John	Mark	Jane	Zoe	Anna

Table 3.1: Women' preferences in the right table and men' preferences in the left table

On the Figure 3.1, the instance of Table 3.1 with the unstable marriage M_1 is represented in a distributed system. Bold black edges as well as bold names in the preference lists represent the pairs in the marriage. The blocking pairs $(Jane, Scott)$ and $(Zoe, Scott)$ are represented with the red bold edges and red names.

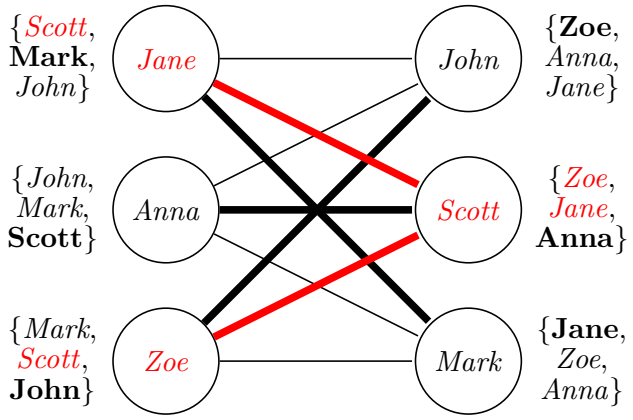


Figure 3.1: Distributed bipartite system with the unstable marriage M_1

Remark 1. *For technical reasons, we use in the proofs a more general definition of blocking pair than the definition given above, as it applies to incomplete matching. In the original definition, a blocking pair has to be a pair of already married persons. In the*

definition of BP used here, man can be unmarried. Formally, a pair (w, m) of a woman w and a man m is blocking iff w is matched to m', m is matched to w' and w and m prefer each other to their actual matching, or, w is matched to m', m is unmatched and w prefer m to m'. Clearly enough, the two notions coincide if the matching is complete. The definition implies that a man prefers to be matched with any woman rather than to stay unmatched.

II - Distributed Systems

A *distributed system* includes a set of computational units called *nodes* (or *processes*). They are connected, *i.e.* able to communicate in a one-to-one fashion and do not have a central memory. Furthermore, there is no centralized control over the nodes.

Each node v can communicate (directly) with a subset of other nodes, called its neighbors and denoted by $\mathcal{N}(v)$ (not including v). Communication is assumed to be bidirectional. Hence, the topology of the system can be represented as a simple undirected graph $G = (V, E)$, where V is the set of nodes and E the set of edges, *i.e.*, communication links. It is assumed that G is the complete bipartite graph $K_{n,n}$, over two subsets of nodes of equal size n.

Following the terminology of [GS62], we call women the n nodes of the first subset (Women) in the bipartite graph and men the n nodes of the second subset (Men). Each node has a unique identifier that can be compared to others. For a clear presentation, we use names to identify nodes. In addition, each node has locally a complete list of n ordered preferences for the nodes of the other set (each woman has a complete list of men and reciprocally) to represent the priorities.

III - Distributed Algorithms

III.1 - Algorithm

A *distributed algorithm* consists of a set of *local algorithms* (one per node). Each node updates its state according to its local algorithm. One way to represent the code of a node v is by a finite set of *guarded rules* of the following form:

Label: (* Comment *)
{Guard}
Actions

The *labels* are used to identify rules. The *guard* of a rule in the code of v is a Boolean expression involving the variables of v and of its neighbors (inside their states or the shared registers, depending on the model). If the guard of some rule evaluates to `True`, then the rule is said to be *enabled* at v. Node v is said to be *eligible* if at least one of its rules is enabled. *Actions* represent a sequence of actions on v's variables. A rule can be *executed* (activated) only if it is enabled. In this case, its execution consists in performing the sequence of actions, using the values of the variables at the time of the guard evaluation.

III.2 - Configurations

The *state* of a node is a vector of the values of its variables. A *configuration* of the system is a vector of states of all nodes. For a given sub-algorithm `Alg`, we denote by $C^{\texttt{Alg}}$ a projection of a configuration C to the variables of `Alg`. By default, in a section dedicated to a particular algorithm `Alg`, when speaking of some configuration we refer to the projection of a configuration to the variables of `Alg`. Furthermore, we use the notation $var(C)$ for the value of var in the configuration C.

IV - Execution of Distributed Algorithms

IV.1 - Scheduler

The asynchrony of the system is modeled by an adversary, called *scheduler* or *daemon*. In a configuration, the scheduler selects a non-empty subset of eligible nodes, then chooses one of the enabled rules per node, then, still atomically, executes the corresponding actions. This is called a *step* (or *transition*) and the activation of each rule in the step is called a *move*, *i.e.*, there are at most n moves per step (one per node). Notice that, since the evaluation of all the rules is made at the same time, moves are causally independent. Such a scheduler is called *distributed* in the literature (contrary to a *central* scheduler, choosing at each step only one enabled node, or to the *synchronous* scheduler that chooses all the enabled nodes). It is convenient to represent a scheduler as a set of sequences of steps or, equivalently as the predicate defining this set. Different types of fairness, limiting the possible decisions of the scheduler, appear in the literature. We do not make any such limitation on the predicate except forcing the scheduler to choose at least one eligible node at each step. This scheduler appears in the literature under the name of *unfair* scheduler. It allows to obtain the strongest results, in particular because some constantly eligible node may stay inactivated for an arbitrary period of time.

IV.2 - Execution

When a step is executed in the configuration C, it leads to a configuration C' and we write $C \rightarrow C'$. We say that C' is *reached* from C, denoted by $C \xrightarrow{*} C'$, if $C \xrightarrow{s} C_1 \xrightarrow{s_1} C_2 \xrightarrow{s_2} \ldots \xrightarrow{s_\pi} C'$. An *execution* is a maximal sequence of pairs (*step*, *configuration*): (s_0,C_0), (s_1,C_1), $\ldots$, (s_k,C_k), $\ldots$ such that $C_i \xrightarrow{s_i} C_{i+1}$ for all $i \geq 0$ and such that the sequence of steps $s_0, s_1, \ldots$ satisfies the predicate of the scheduler.

The term "*maximal*" means that the execution is either infinite or ends in a *terminal configuration*, *i.e.*, a configuration in which no node is eligible.

A distributed algorithm *solves* the stable marriage problem if each of its executions starting from a predefined initial configuration, under the unfair distributed scheduler, reaches a terminal configuration in which there is a stable marriage.

V - Communication Models

Several communication models exist in the literature. In this thesis, we use two models: the state model in Chapter 4 and the register model in Chapter 5. Thus, in the following, this two models are presented.

State model. This is the composite atomicity model of computation (*cf.* [Dij74, Gho14]) in which the nodes communicate by reading the variables', but not constants', values directly in the states of their neighbors. That is, each node can read its own variables and those of its neighbors, but can write only to its own variables. Constants cannot be directly read by a neighbor. This is for being able to keep confidential a sensitive information of a node (like the preference list).

Register model. In Chapter 5, we adopt the link register communication model (*cf.* [DIM93]). Each process is associated with a set of atomic registers, each of size of $O(1)$ bits (our algorithm does not require more space). For each adjacent node u, the node v shares a register $r_{v,u}$ in which v is the only node allowed to write and that u can read. Each register is a record with several fields. The field var in the register $r_{v,u}$ is named $var_{v,u}$.

The Figure 3.2 shows the characteristics of the register model. As for the state model, the variables of the node v are read and written by v. The difference with the state model is the read access of the neighbors. Indeed, j can read $sv1_{v,j}$ and $sv1_{u,j}$ (similarly $sv2_{v,j}$, and $sv2_{u,j}$) while i can read $sv1_{v,i}$ and $sv1_{u,i}$ (similarly, $sv2_{u,i}$ and $sv2_{u,i}$). Notice that, at the starting configuration, a shared variable $sv1_{v,j}$ may be outdated in the correspondence with the local state of v, before v is activated and writes into it. This is impossible in the state reading model, since a neighboring node reads the local memory of v directly.

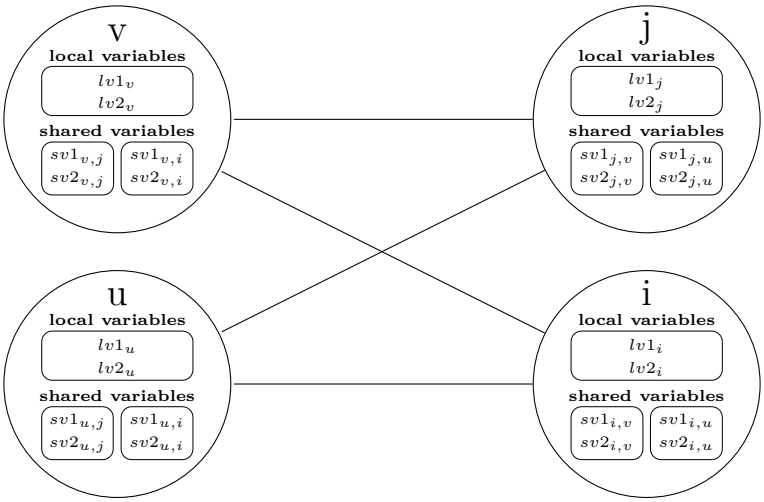


Figure 3.2: The register model

VI - Self-stabilization

The notion of *self-stabilization* [Dij74] is related to transient failure tolerance. Corruptions of variables[1] may put the system in an arbitrary configuration, from which the algorithm has to recover.

Formally let $\mathcal{A}$ be a distributed algorithm, $\mathcal{C}$ the set of its configurations and $\mathcal{E}$ the set of its executions from any configuration in $\mathcal{C}$. A (specification of a) problem is a predicate Prob on executions.

Definition 3. *$\mathcal{A}$ is self-stabilizing for* Prob *(or solves* Prob *in a self-stabilizing way) if and only if there exists a non-empty subset $\mathcal{L}$ of $\mathcal{C}$ of so called* legitimate configurations*, such that:*

1. *(Correctness) any execution in $\mathcal{E}$ starting from a configuration C in $\mathcal{L}$ satisfies* Prob*,*
2. *(Convergence) any execution in $\mathcal{E}$ reaches a configuration in $\mathcal{L}$.*

For any execution of a self-stabilizing algorithm $\mathcal{A}$, whenever Prob is satisfied, we say that $\mathcal{A}$ has *stabilized*.

On the Figure 3.3, the rectangles represent configurations of a system (in $\mathcal{C}$). Green rectangles are legitimate configurations (in $\mathcal{L}$). Orange rectangles are non-legitimate configuration (in $\mathcal{C} \setminus \mathcal{L}$). Arrows are the possible transitions from a configuration to another. From non-legitimate configurations, the execution reaches eventually a legitimate configuration. Furthermore, from a legitimate configuration only legitimate configurations are reachable: this is a particular case of stabilization.

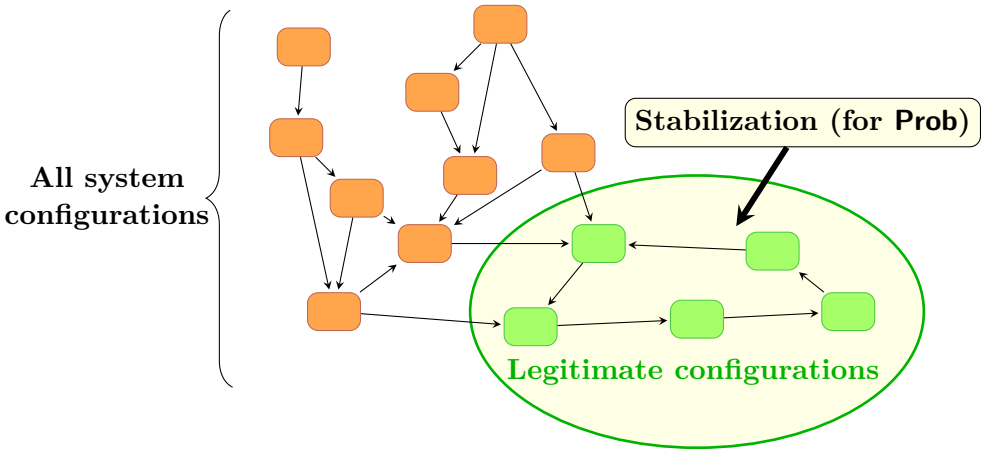


Figure 3.3: Self-stabilization

In the context of stable marriage, the predicate Prob defining the problem is satisfied by an execution iff a) the execution reaches a *terminal* configuration (*i.e.*, a configuration in which no node is eligible), and b) this configuration contains a stable marriage (*i.e.*, all nodes are married and there is no BP).

[1] Notice that constants, like IDs or preference lists, are assumed to be incorruptible.

VII - Time Complexity

The time complexity of a self-stabilizing distributed algorithm can be evaluated in terms of moves, steps or *asynchronous rounds* (see Definition 4). The *stabilization time*, counted in moves (respectively in steps or in rounds), is the maximum number of moves (resp. steps, rounds) until a configuration in $\mathcal{L}$ is reached, starting from an arbitrary configuration. The stabilization time in moves gives an upper bound on the stabilization time in steps and rounds.

Asynchronous rounds. *Asynchronous rounds* (or simply *rounds* in the following) have been introduced in [DIM97] and extended with the concept of *neutralization* [BDPV07]. A process v is said *neutralized* during a step $C_i \rightarrow C_{i+1}$, if v is eligible in configuration C_i but not in configuration C_{i+1}, and is not activated in the step $C_i \rightarrow C_{i+1}$. The rounds are inductively defined as follows.

Definition 4 (Asynchronous rounds)**.** *The first round of an execution* $e = (s_0, C_0), (s_1, C_1), \ldots$ *is the minimal prefix* $e' = (s_0, C_0), \ldots, (s_j, C_j)$*, such that every process that is eligible in* C_0 *either executes a rule or is neutralized during a step of* e'*. Let* e'' *be the suffix* $(s_j, C_j), (s_{j+1}, C_{j+1}), \ldots$ *of* e*. The second round of* e *is the first round of* e''*, and so on.*

Chapter 4

A Solution Based on Ackermann *et al.* Two-Phased Idea

Contents

In this chapter, we present our first solution to SMP. This work is published [LMB+17] and received a *best paper award*. At the time of publication, it was the first *self-stabilizing* distributed solution for general SMP. The algorithm is designed for the model of *composite atomicity* (state reading model), under an *unfair distributed* scheduler (see Chapter 3 for definitions).

We first present an historical background and explain how this algorithm is obtained, in Section I. Then, in Section II, we present the solution. In Section III, the proof analysis is provided in two steps. We first sketch the proof in Sub-section III.1 and in Sub-section III.2, we provide the formal proof of correctness and a time complexity analysis, providing an upper bound in terms of *moves* and *steps*. Finally, we conclude this chapter with some remarks and perspectives (Section IV).

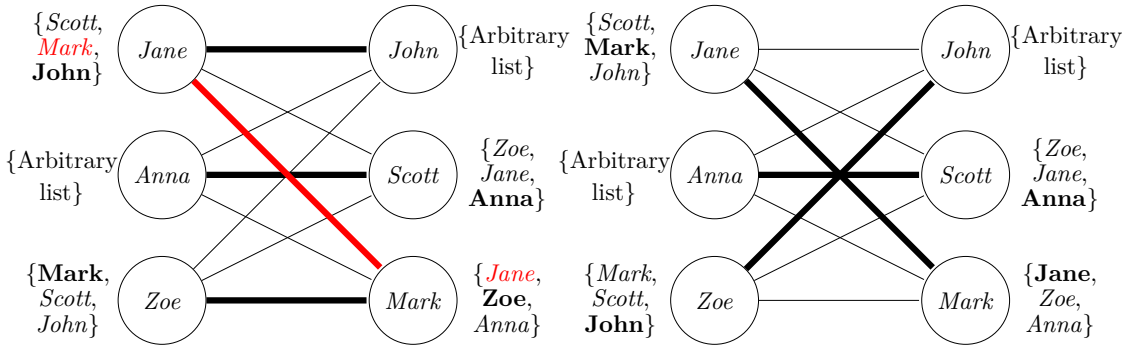
I - Preliminaries and Contribution

Even though the original stable marriage algorithm by Gale and Shapley (GSA) is essentially centralized, it can be interpreted as a distributed one [BM05] and most of the existing distributed algorithms rely on GSA. In general, the algorithm proceeds by iteratively realizing proposals, *e.g.*, by women, and acceptances, *e.g.*, by men (or vice-versa). Intuitively speaking, the algorithm creates matches and *resolves* appearing BPs, when improving iteratively the quality of the matches according to the preferences ("better match" dynamics). But, GSA does not necessarily converge towards correct

configurations from any initial configuration. In other words, it does not naturally tolerate transient failures that can put a system in an arbitrary configuration, *i.e.*, it is not self-stabilizing. In particular, from some configurations, BPs can appear and not been eliminated during the execution. Thus the issue of their elimination in a self-stabilizing context naturally appears.

SMP has received a lot of attention, in particular by Knuth [Knu76]. When investigating combinatorial properties of the problem, Knuth discovered the possibility of cycles when resolving BPs (in a specific way) from some initial configurations with an incomplete or unstable matching.

The Figures below show an example of such a cycle. The pair (*Jane*, *Mark*) is a BP, *i.e.* the marriage is unstable. Knuth proposes to resolve the BP by exchanging the partners of the pair. Here, *Mark* and *Jane* are married together and their former partners, *Zoe* and *John* are married together (Figure 4.1.b).

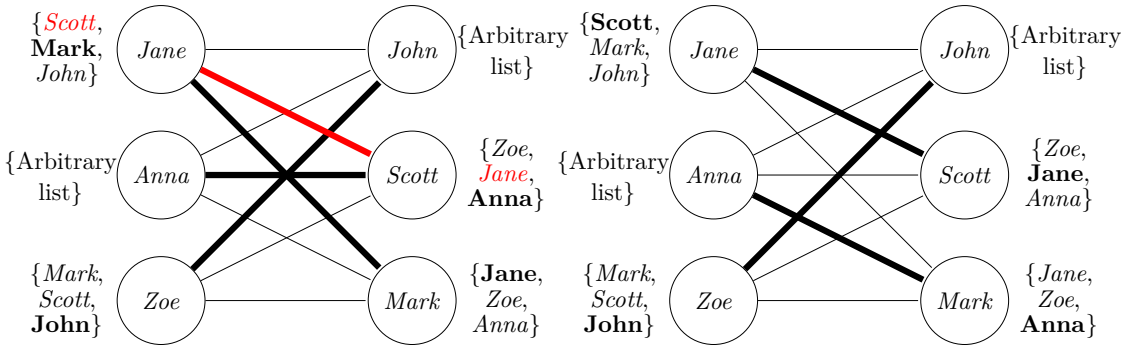


(a) Initial configuration with a BP (*Jane*, *Mark*)

(b) Configuration after solving the BP

Figure 4.1: Knuth's cycle: first resolution

In the reached system, the marriage is still unstable: the pair (*Jane*, *Scott*) is a BP (Figure 4.2.a). After the repair (in the same way as previously), the system is as in Figure 4.2.b.

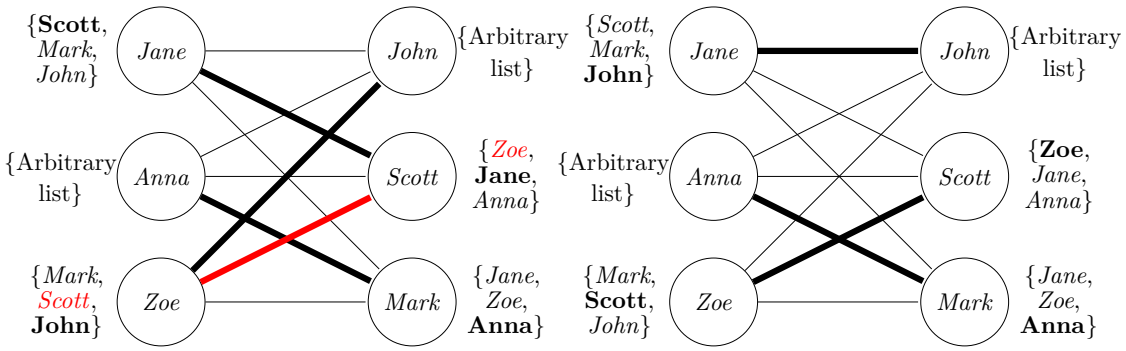


(a) Resulting configuration with an other BP: $(Jane,\ Scott)$

(b) Configuration after solving the BP

Figure 4.2: Knuth's cycle: second resolution

But, there is again a BP: the pair $(Zoe,\ Scott)$ (Figure 4.3.a). Figure 4.3.b shows the system after the repair by exchanging the partners.

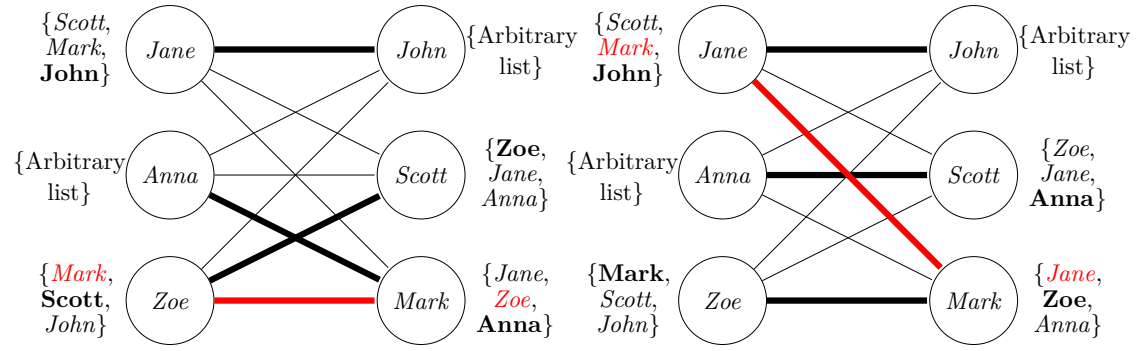


(a) Resulting configuration with an other BP: $(Zoe,\ Scott)$

(b) Configuration after solving the BP

Figure 4.3: Knuth's cycle: third resolution

The pair $(Zoe,\ Mark)$ is a BP, making the matching unstable (Figure 4.4.a). Finally, after the repair, the matching is as in Figure 4.4.b.



(a) Resulting configuration with an other BP: $(Zoe,\ Mark)$

(b) Configuration after solving the BP

Figure 4.4: Knuth's cycle: fourth resolution

Notice that the Figure 4.4.b is the same as the Figure 4.1.a, with the same BP and no improvement in the matchings. This example shows that a self-stabilizing solution cannot be simply obtained by repairing locally the blocking pairs created by transient failures.

After this negative result, a step forward was taken by Roth and Vande Vate [RS90] and by Ackermann *et al.* [AGM+11]. Both works present completely centralized strategies allowing to solve stable marriage starting from any given matching. The strategy proposed by Roth and Vande Vate stores and consults a global access set of previously resolved BPs and thus is inherently centralized. Differently, the strategy by Ackermann *et al.* [AGM+11] works in two phases. In the first one, only married women make proposals for improving their marriages. When no married woman can improve anymore, the second phase starts. In this phase, only unmarried women can make proposals (until they all are matched). At the end of this phase, a stable marriage is obtained (after at most $O(n^2)$ steps). In a distributed context, being able to start from an arbitrary matching is not sufficient to be self-stabilizing, for the reasons that we explain below. Nevertheless, it is a first step towards self-stabilization, and it is the reason why we adopted the idea of the two phases, like in [AGM+11].

Making this idea work in a distributed asynchronous and self-stabilizing way is still very challenging. First, there is a need of a sort of synchronization of phases between the nodes that cannot move all together to the next phase, like in the centralized case. Then, termination detection is needed for detecting the end of the first phase. Furthermore, Ackermann *et al.* supposed "best response" dynamics, contrary to the "better" ones in a distributed GSA. "Best response" dynamics are inherently centralized too, since creation or suppression of a match is not instantaneous (as it is in the centralized case) and the actual matches can change during the delay for realizing these actions. Hence, it is difficult to implement perfect "best response" dynamics. Finally, notice that a distributed matching has to be encoded with pointers that can be badly initialized. This is not taken into account in the algorithm of Ackermann *et al.*.

In addition to these difficulties, we strive to provide a truly decentralized solution using neither leader nor global reset and detecting and correcting faults locally (similarly to the way GSA resolves BPs). This rules out the known self-stabilizing automatic

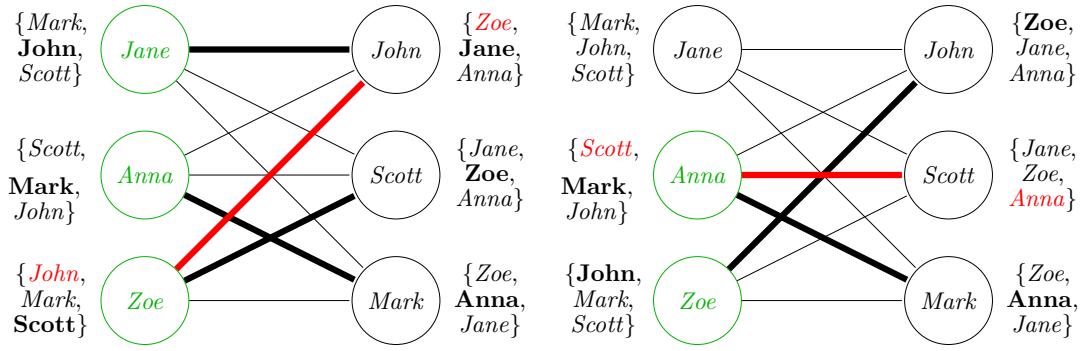
transformers requiring such type of primitives. On the positive side, this allows obtaining more efficient algorithms in terms of time and space. This is also the reason for not using known synchronization techniques (*e.g.*, [AKM+07], [BPV04]). The proposed algorithm works with only one additional phase of synchronization (in addition to the two phases in the strategy of Ackermann *et al.*), while using known synchronization techniques would result in much more additional phases. On top of that, it ensures a sort of confidentiality, in the sense that the preference lists of the nodes are not public[1]. Notice that keeping the complete preference lists of users secret may be an important requirement, for instance in some economic contexts. This goal cannot be achieved by any centralized solution.

The proposed algorithm works under an *unfair* distributed scheduler, *i.e.*, choosing at each step a non-empty subset of nodes that have actions to perform (*i.e.*, *enabled* nodes; see model Section V in Chapter 3 for a formal definition). In spite of all the aforementioned difficulties, we design (Section II) and prove (Section III) such a self-stabilizing stable marriage algorithm (which also guarantees confidentiality of the preference lists). The sketch of its proof is in Sub-section III.1 and the details in Sub-section III.2 The time complexity analysis provides an upper bound of $O(n^4)$ moves (activations changing the state of a node). Straightforwardly, this upper bound applies to steps (activations changing the configuration of the system; see the model section). Note that, in Chapter 6, we also study how the proposed algorithm can be useful for obtaining self-stabilizing solutions to some variants of the stable marriage problem. The results (variants included) have been submitted to a journal.

II - Self-stabilizing Solution to SMP

The solution of Ackermann *et al.* proceeds in two phases. In the first phase, already married women try to improve their marriage as in the example of Figures 4.5 - 4.7. In this example, married women are green. In the initial configuration (Figure 4.5.a), there is only one BP: (Zoe, $John$). In the Ackermann *et al.* algorithm, resolving a BP means to match the BP but, contrary to Knuth, former partners become single. After resolving the BP (Zoe, $John$) leads to a configuration (Figure 4.5.b) with a new BP: ($Anna$, $Scott$).

[1] A node v only communicates to its neighbor u the priority it gives to u, and the priority of its actual spouse.



(a) Unstable marriage: (*Zoe*, *John*) is a BP

(b) After the BP's resolution, a new BP: (*Anna*, *Scott*)

Figure 4.5: First phase of the Ackermann *et al.*'s algorithm

After the resolution of the new BP, the following configuration is reached (Figure 4.6). When no improvement (*i.e.*, no resolution of BP) is possible, phase 2 starts.

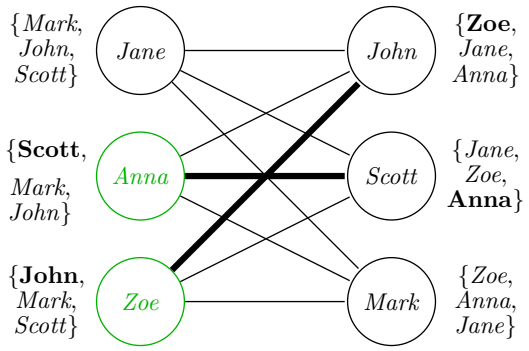
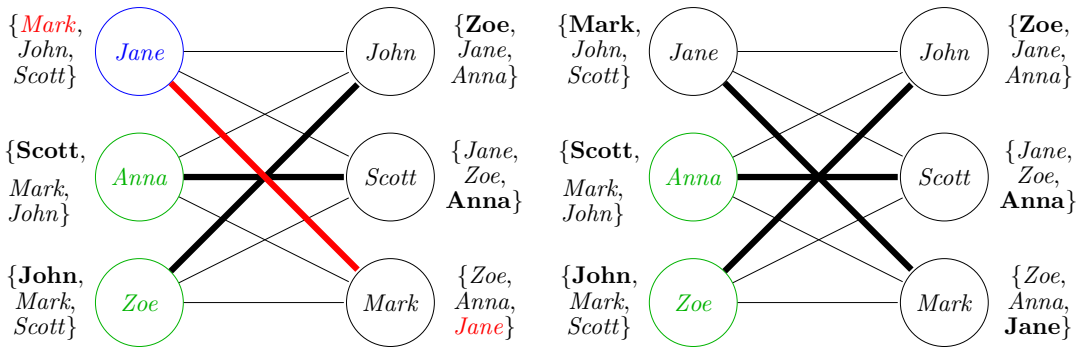


Figure 4.6: Configuration after the first phase

In the second phase, single women (in blue in Figure 4.7.a) try to be married, using the Gale and Shapley's mechanism: they propose to men in their preference list's order. If a married man receives a better proposal, he accepts the proposal and divorces from his actual spouse, who becomes single and, then, proposes in her preference list's order. In this example, *Jane* is the only single woman and she proposes to *Mark*, her first choice. When all single women are married, the final stable marriage is reached (Figure 4.7.b).



(a) The single woman *Jane* proposes to her first choice.

(b) Final matching

Figure 4.7: Second phase of the Ackermann *et al.*'s algorithm

In the first phase, women globally *reduce their regrets*, *i.e.*, increase the quality of their matching, and in the second phase, men do symmetrically the same. The algorithm is correct, even when started from an incomplete matching, but is not self-stabilizing in the strict sense, because all nodes *must start* in phase 1 and change *simultaneously* to phase 2. That is, some possible configurations cannot appear in an execution, while self-stabilization has to recover from any starting configuration. It could be made self-stabilizing using centralization, with the implementation of an incorruptible global phase counter. In a distributed asynchronous setting, things are more difficult. The distributed self-stabilizing solution that we propose takes the idea of two phases, but use a supplementary phase for the purpose of synchronization. We number the phases 1, 1.5 and 2. Phases 1 and 2 play about the same role as in Ackermann *et al.*'s algorithm.

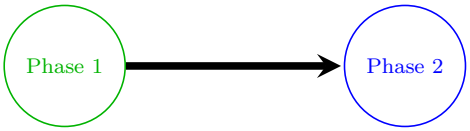


Figure 4.8: Phase transition in the Ackermann *et al.*'s algorithm

Phase 1.5 is an intermediary phase performing synchronization between phase 1 and 2 (due to an erroneous initial configuration). During phases 1 and 2, women have the initiative to propose marriage, men can only choose among the different proposals.

We start by explaining the role of phases when all nodes are initially in phase 1. The transition from phase 1 to phase 1.5 is realized first by women who have checked the lack of BPs. Once all women are in phase 1.5, men can change to phase 1.5 if they do not detect BPs. Otherwise, the detecting man blocks the process (by staying in phase 1). When the woman involved in the BP is activated, it changes its phase to 1 (forcing a come back to phase 1 for all men). It is only when all nodes reach phase 1.5 that women can move to phase 2. Then, men follow by moving

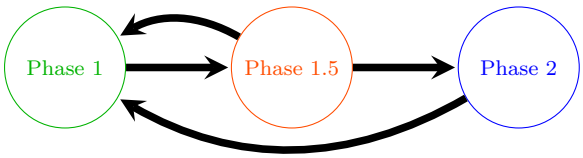


Figure 4.9: Phase transitions in the proposed distributed self-stabilizing version

to the phase 2 too. The verification of the absence of BPs before entering phase 1.5 guarantees their absence at the beginning and during phase 2.

Now we precise how self-stabilization is obtained. When a faulty configuration is detected, nodes can move from phase 2 to phase 1. For example, this happens if it is detected that some pointers are badly initiated, if the phase number of a man is greater than the one of a woman, or if some phase values are not consecutive. This move can also be initiated by a married woman in phase 2, who detects a possible improvement (*i.e.*, a BP since the woman is married). All other nodes will detect the phase change and move to phase 1 too.

We will show that no execution cycles more than one time through the phases 1, 1.5, 2. Similarly to the algorithm of Ackermann *et al.*, we show that, during the last execution of the first phase, the regrets of the married women are globally decreasing. This ensures that no BP exists at the end of this phase. During the last execution of phase 2, it is the same for the regrets of men and, thus, no BP can appear (even though the matching can be still incomplete). At the end, in $O(n^4)$ moves in overall, a complete stable marriage is obtained.

II.1 - Algorithm Implementation

We now make precise the implementation of these ideas. Each node v has variables and constants. The variables can be read by the neighbors, but the access to constants is limited, for the confidentiality reasons that we explained before.

II.1.1 - Variables, Constants, Functions and Predicates (for a node v)

Variables.

- *marriage* $\in \mathcal{N}(v) \cup$ `Null`: if the value is not `Null`, we say that it is the *spouse* of v. Otherwise we say that v is *single*.
- *proposal* $\in \mathcal{N}(v) \cup$ `Null`: if v is a woman, we say that this is the node to whom v has *proposed*; if v is a man, this is the woman whose proposal has been *accepted* by v. If the value is `Null`, we say that there is no proposal or acceptance.
- *phase* $\in \{1,\ 1.5,\ 2\}$: v is said to be *in phase* α if $v.phase\ =\ \alpha$.

Constants.

- *pref*: the v's list of its n neighbors in preference order. The priority of the i^{th} element of the list is i. Then, the first element is the most preferred neighbor and its priority is 1.

Observe that $pref$ is an ordered list that exactly contains all the elements in $\mathcal{N}(v)$. Recall that the lists of preferences are kept secret. A node v only communicates to its neighbor u the *priority* it gives to u and the priority of its actual spouse. If v is single, the latter is $n + 1$.

Functions.

- priority(v,u): returns the priority of u in the preference list of v. Note that if parameter u is evaluated to `Null`, priority(v,u) $= n+1$ (v is single).
- min($\mathcal{A}$): returns the most preferred node in a set $\mathcal{A}$ of nodes

Let $\mathcal{C}_v$ be the set of nodes which prefer v and are preferred by v:

$$\mathcal{C}_v = \{\ u \in v.pref : \text{priority}(v,u) < \text{priority}(v,v.marriage) \\ \wedge\ \text{priority}(u,v) < \text{priority}(u,u.marriage)\ \}$$

The following function is used by women to determine which man to propose to.

- BestMarriage(v): if ($\mathcal{C}_v \neq \emptyset$) then `return` min($\mathcal{C}_v$) else `return Null`

Let $\mathcal{P}_v$ be the set of women who: (a) are preferred by v to his own spouse; (b) prefer v to their own spouse; (c) have proposed to v; (d) are in the same phase as v; (e) are single, if their phase is 2, or with a spouse, if their phase is 1.

$$\begin{aligned}\mathcal{P}_v \quad = \quad & \{u \in \mathcal{C}_v :\ u.proposal = v \wedge u.phase = v.phase \\ \wedge \quad & [(u.marriage \neq \texttt{Null} \wedge u.phase = 1) \\ & \vee\ (u.marriage = \texttt{Null} \wedge u.phase = 2)]\end{aligned}$$

The following function is used only by men to determine which proposal to accept.

- BestProposal(v) = if ($\mathcal{P}_v \neq \emptyset$) then `return` min($\mathcal{P}_v$) else `return Null`

Predicates.
The solution that we propose uses predicates for testing locally some properties.

The predicate Married(v) is used by a woman v for checking whether she is reciprocally *married* (`True`), or not (`False`).

- Married(v) $\equiv$ $(v.marriage \neq \texttt{Null}) \wedge [(v.marriage.marriage = v) \vee (v.marriage.proposal = v)]$
- MarriedM(v) $\equiv$ $(v.marriage \neq \texttt{Null}) \wedge (v.marriage.marriage = v)$

The predicate Response(v) checks if the proposal of v has been *accepted*.

- Response(v) $\equiv$ $(v.proposal \neq \texttt{Null}) \wedge (v.proposal.proposal = v)$

The predicate AlreadyEngaged(v) is used by a man to detect if he already accepted a proposal.

- AlreadyEngaged(v) $\equiv$ $(v.proposal \neq \texttt{Null}) \wedge$ $[(v.proposal.proposal = v) \vee (v.proposal.marriage = v)]$

Since there is an asymmetry between women's proposals and men's acceptances (women ask first for a marriage and then men answer), they have different predicates to verify whether their pointers are correct and, in particular, that their marriages are reciprocal (suffix W in the predicate name refers to women and M to men). Otherwise, the predicate is `False` and pointers are said *incoherent*.

- $\mathsf{IncoherentPointersW}(v) \equiv (v.marriage \neq \texttt{Null}) \wedge [((v.marriage.marriage \neq v) \wedge (v.marriage.proposal \neq v)) \vee (v.marriage = v.proposal) \vee (v.marriage.proposal = v \wedge \mathsf{priority}(v.marriage,v) > \mathsf{priority}(v.marriage,v.marriage.marriage))]$

- $\mathsf{IncoherentPointersM}(v) \equiv (v.marriage \neq \texttt{Null}) \wedge [(v.marriage.marriage \neq v) \vee (v.marriage = v.proposal)]$

Since the definition of BP is asymmetrical (*cf.* Remark 1), there are two predicates for checking the presence of BP (which involves a married woman). Hence, if at least one of these two predicates is `True`, that indicates there is a BP. If a node detects a BP, we say that it is *involved* in a BP.

- $\mathsf{BlockingPairW}(v) \equiv \mathsf{Married}(v) \wedge (\mathcal{C}_v \neq \emptyset)$

- $\mathsf{BlockingPairM}(v) \equiv (\exists u \in \mathcal{C}_v\colon\ u.marriage \neq \texttt{Null})$

The following predicate, $\mathsf{AllCoherentPhase}(v)$, checks some coherence in phases, namely whether v and all its neighbors are in phase 2, or v is in phase 1 and all its neighbors in phases 1 or 1.5. It is used only by men to decide if they can accept a proposal (women verify somewhat different conditions).

- $\mathsf{AllCoherentPhase}(v) \equiv (v.phase = 2 \wedge (\forall u \in \mathcal{N}(v)\colon\ u.phase = 2)) \vee (v.phase = 1 \wedge (\forall u \in \mathcal{N}(v)\colon\ u.phase \in \{1, 1.5\}))$

II.1.2 - Algorithm.

The matching $\mathcal{M}$ built by the presented algorithm is defined by pairs $(w, m) \in E$ such that $w.marriage = m$ and $m.marriage = w$. The algorithm predicate is $\mathsf{Pred2Phases} \equiv [\forall w \in \text{WOMEN}\colon \mathsf{Married}(w) \wedge \neg\mathsf{BlockingPairW}(w) \wedge \neg\mathsf{BlockingPairM}(w.marriage)]$ and for the proof of self-stabilization, we define the legitimate configurations as the configurations satisfying this predicate. Notice that executions satisfying Pred2Phases satisfies also Prob.

The part of the algorithm executed by women (Algorithm 1) has 9 rules. We start by describing intuitively what those rules do.

1. The **Reset** rule performs a reset of marriage and proposal pointers, if these pointers appeared to be incoherent according to IncoherentPointersW.

2. The rule **BadInit** is executed by a woman in phase 2. In this phase a married woman is not supposed to make a proposal. Thus, if her proposal and marriage pointers are not set to `Null` (the only reason for that is a bad initialization), **BadInit** resets the proposal pointer and sets the phase to 1 (to restart the computation of a matching).

3. The rule **Propose1** (respectively **Propose2**) is executed by a married (resp. single) woman in phase 1 (resp. 2). This rule's effect is a proposal to the man who corresponds to the best marriage for her (*i.e.*, best for the woman but also for the man with respect to its actual spouse or single status).

4. The rule **Confirm1** (resp. **Confirm2**) is executed by a married (resp. single) woman in phase 1 (resp. 2), after she has made a proposal to a man and this proposal has been accepted (the man has put the name of the woman in its variable proposal). Then, the woman confirms the marriage, breaking from her previous man (only **Confirm1**) and matching with the new one. The couple is now considered married.

5. The rule **ToPhase1.5** is a phase transition rule from phase 1 to phase 1.5. When a woman in phase 1 cannot make any proposal (no BP is detected or she is single), she has to move to phase 1.5 if all men are in phase 1.

6. The rule **ToPhase2** is also a phase transition rule. A woman in phase 1.5 can change to phase 2 if she does not detect any BP and if all men are in phase 1.5.

7. The rule **ToPhase1** is the third phase transition rule. It is executed by a woman in order to move from phase 2 or phase 1.5 to phase 1. The change happens if the following (faulty) conditions are detected: (a) the woman is in phase 2 but some man is in phase 1 (either a BP has been detected or phase synchronization has not stabilized yet); (b) the woman is in phase 1.5 but a man is in phase 2 (the phase synchronization has not stabilized yet); (c) the woman is married and either in phase 1.5 or 2 but detects a BP.

Remark 2. *If a man m does not answer positively to a proposal from a woman w (it has a better priority proposal), she detects it.* BestMarriage(w) *will not return m any longer and w can change her proposal with* **Propose1** *or* **Propose2**.

Algorithm 1 for $w \in$ WOMEN

Reset : *(* Reset pointers of marriage and proposal *)*
  $\{\mathsf{IncoherentPointersW}(w)\}$
    $w.marriage \leftarrow \texttt{Null}$, $w.proposal \leftarrow \texttt{Null}$
BadInit : *(* Reset the pointer of proposal *)*
  $\{\neg\mathsf{IncoherentPointersW}(w) \wedge w.marriage \neq \texttt{Null}$
  $\wedge\ w.proposal \neq \texttt{Null} \wedge \forall\ v \in \mathcal{N}(w) \cup \{w\} :\ v.phase = 2\}$
    $w.proposal \leftarrow \texttt{Null}$, $w.phase \leftarrow 1$
Propose1 : *(* Propose in phase 1 *)*
  $\{\neg\mathsf{IncoherentPointersW}(w) \wedge \forall\ v \in \mathcal{N}(w) \cup \{w\} :\ v.phase = 1$
  $\wedge\ \mathsf{BestMarriage}(w) \neq w.proposal \wedge \mathsf{Married}(w)\}$
    $w.proposal \leftarrow \mathsf{BestMarriage}(w)$
Confirm1 : *(* Confirm a proposal in phase 1 *)*
  $\{\neg\mathsf{IncoherentPointersW}(w) \wedge \forall\ v \in \mathcal{N}(w) \cup \{w\} :\ v.phase = 1$
  $\wedge\ \mathsf{Response}(w) \wedge \mathsf{Married}(w) \wedge \mathsf{BestMarriage}(w) = w.proposal\}$
    $w.marriage \leftarrow w.proposal$, $w.proposal \leftarrow \texttt{Null}$
Propose2 : *(* Propose in phase 2*)*
  $\{\neg\mathsf{IncoherentPointersW}(w) \wedge \forall\ v \in \mathcal{N}(w) \cup \{w\} :\ v.phase = 2$
  $\wedge\ \mathsf{BestMarriage}(w) \neq w.proposal \wedge w.marriage = \texttt{Null}\}$
    $w.proposal \leftarrow \mathsf{BestMarriage}(w)$
Confirm2 : *(* Confirm a proposal in phase 2 *)*
  $\{\neg\mathsf{IncoherentPointersW}(w) \wedge \forall\ v \in \mathcal{N}(w) \cup \{w\} :\ v.phase = 2$
  $\wedge\ \mathsf{Response}(w) \wedge w.marriage = \texttt{Null}$
  $\wedge\ \mathsf{BestMarriage}(w) = w.proposal\}$
    $w.marriage \leftarrow w.proposal$, $w.proposal \leftarrow \texttt{Null}$
ToPhase1.5 : *(* To phase 1.5 *)*
  $\{\neg\mathsf{IncoherentPointersW}(w) \wedge \forall\ v \in \mathcal{N}(w) \cup \{w\} :\ v.phase = 1$
  $\wedge\ \neg\mathsf{BlockingPairW}(w)\}$
    $w.phase \leftarrow 1.5$, $w.proposal \leftarrow \texttt{Null}$
ToPhase2 : *(* To phase 2 *)*
  $\{\neg\mathsf{IncoherentPointersW}(w) \wedge \forall\ v \in \mathcal{N}(w) \cup \{w\} :\ v.phase = 1.5$
  $\wedge\ \neg\mathsf{BlockingPairW}(w)\}$
    $w.phase \leftarrow 2$, $w.proposal \leftarrow \texttt{Null}$
ToPhase1 : *(* To phase 1 *)*
  $\{\neg\mathsf{IncoherentPointersW}(w) \wedge ($
  $[\exists\ m \in \mathcal{N}(w)\colon (m.phase = 1 \wedge w.phase = 2)$
    $\vee\ (m.phase = 2 \wedge w.phase = 1.5)]$
  $\vee$
  $[w.phase \in \{2, 1.5\} \wedge \mathsf{BlockingPairW}(w)\}$
    $w.phase \leftarrow 1$, $w.proposal \leftarrow \texttt{Null}$

The part of the algorithm executed by men (Algorithm 2) consists of 6 rules:

1. The **Reset** rule resets the marriage pointer of a man and changes its phase to 1. We prove later that this can happen only once for a man in phase 2.

2. The **Accept** rule checks that women are in a consistent phase with respect to the phase of the man executing the rule (AllCoherentPhase), that the best proposal received is different from his actual partner and that he has not accepted another proposal (¬ AlreadyEngaged). Remark that this is a commitment, but the couple is not yet married. If the man is married to another woman, he has to break the marriage since he received a better proposal.

3. The role of the rule **Confirm** is to confirm a marriage. The rule checks that the phases are coherent (AllCoherentPhase) and if the woman has her variable marriage set to the man, he confirms too.

4. The rule **ToPhase1.5** is a phase transition rule from phase 1 to phase 1.5. If all women are in phase 1.5 and no BP is detected, the man changes his phase to 1.5.

5. **ToPhase2** makes men change to phase 2. When all women are in phase 2 and men have checked the lack of BPs, then phase 2 can begin.

6. The **ToPhase1** rule detects a phase synchronization problem (a woman being in phase 1 or 1.5 with the man in phase 2) or a woman willing to change to phase 1 (a BP has been detected) when he is in phase 1.5. Then the man moves to phase 1.

Algorithm 2 for $m \in$ Men

Reset : *(* Reset pointer of marriage *)*
  { IncoherentPointersM(m) }
    $m.marriage \leftarrow$ `Null`, $m.phase \leftarrow 1$
Accept : *(* Accept a proposal except in phase 1.5 *)*
  {$\neg$IncoherentPointersM(m) $\wedge$ AllCoherentPhase(m)
  $\wedge$ BestProposal(m) $\neq$ `Null` $\wedge$ $\neg$AlreadyEngaged(m)}
    $m.proposal \leftarrow$ BestProposal(m)
Confirm : *(* Confirm a marriage *)*
  {$\neg$IncoherentPointersM(m) $\wedge$ $m.proposal \neq$ `Null`
  $\wedge$ $m.proposal.marriage = m \wedge$ AllCoherentPhase(m)
  $\wedge$ priority(m,$m.proposal$) $<$ priority(m,$m.marriage$)}
    $m.marriage \leftarrow m.proposal$, $m.proposal \leftarrow$ `Null`
ToPhase1.5 : *(* To phase 1.5 *)*
  {$\neg$IncoherentPointersM(m) $\wedge \forall\, w \in \mathcal{N}(m)$: $w.phase = 1.5$
  $\wedge$ $m.phase = 1 \wedge \neg$BlockingPairM(m)}
    $m.phase \leftarrow 1.5$, $m.proposal \leftarrow$ `Null`
ToPhase2 : *(* To phase 2 *)*
  {$\neg$IncoherentPointersM(m) $\wedge \forall\, w \in \mathcal{N}(m)$: $w.phase = 2$
  $\wedge$ $m.phase = 1.5 \wedge \neg$BlockingPairM($m$)}
    $m.phase \leftarrow 2$, $m.proposal \leftarrow$ `Null`
ToPhase1 : *(* To phase 1 *)*
  {$\neg$IncoherentPointersM(m) $\wedge$ (
  [$(\exists\, w \in \mathcal{N}(m)$: $w.phase \in \{1.5, 1\}) \wedge m.phase = 2$]
  $\vee$
  [$(\exists\, w \in \mathcal{N}(m)$: $w.phase = 1) \wedge m.phase = 1.5$]) }
    $m.phase \leftarrow 1$, $m.proposal \leftarrow$ `Null`

III - Correctness Proof and Time Complexity Analysis (Algorithms 1 and 2)

We have to prove that Algorithms 1 and 2 are self-stabilizing for the specification Prob of SMP. For that, we prove that any execution (from an arbitrary configuration) terminates in a configuration satisfying Pred2Phases. That establishes convergence and, as Pred2Phases $\Rightarrow$ Prob, this property yields also correctness.

In the following Sub-section III.1, we sketch the proof. The detailed proofs can be found in Sub-section III.2.

III.1 - Sketch

The analysis of the algorithm appears to be complex for several reasons. First, the algorithm has to overcome the unfair adversary that can prevent some enabled nodes from being activated as long as there are other enabled nodes. This may take many moves made by nodes in different states and configurations. Moreover, all these moves

may not contribute to the convergence (*e.g.*, if an existing fault is not yet detected). Still, they have to be taken into account for the correctness and the time analysis. Another reason for the analysis difficulty is the distribution and asynchrony of the solution. For example, as mutual marriage, divorce, and blocking pair detection cannot be done instantaneously, or at least within some timing guarantees (as in synchronous lock-step models), the related results on previous centralized or synchronous solutions cannot be used in our case.

Finally, due to self-stabilization, the analysis has to consider executions starting from an arbitrary configuration. In particular, initially, the phase numbers can be arbitrary. Moreover there are specific rules applying to such or such phase number. The consequence of that is a great number of cases to treat, each case necessitating a particular treatment and special arguments. For classifying the different cases into categories, the following definition is introduced.

Definition 5. *We say that a configuration C is in (A, B, bp) iff, in C, the set of phase numbers of women is A, the set of phase numbers of men is B and there are bp blocking pairs. The configuration C is in $(A, B, bp)^{\times}$ iff, in C, the set of phase numbers of women is included in A, the set of phase numbers of men is included in B and there are bp blocking pairs.*

For example, we have the identity:
$(\{a\}, \{b, c\}, X)^{\times} \equiv (\{a\}, \{b, c\}, X) \bigcup (\{a\}, \{b\}, X) \bigcup (\{a\}, \{c\}, X)$. Furthermore, we denote by $\mathcal{C}^{1W}$ (resp. $\mathcal{C}^{1M}$) the set of configurations having at least one women (resp. one man) in phase 1 and we set $\mathcal{C}^1 = \mathcal{C}^{1W} \bigcup \mathcal{C}^{1M}$.

We denote by $\mathcal{C}^1$ the set of configurations where $\exists\ v \in V\ :\ v.phase = 1$. $\mathcal{C}^{1W}$ and $\mathcal{C}^{1M}$ are sets of configurations in $\mathcal{C}^1$ where, respectively, $v \in$ Women and $v \in$ Men.

We prove the correctness of the algorithm for every possible starting configuration type. We start by giving a skeleton, which allows to skip the countless cases of the detailed proof and which is sufficient for understanding its main ideas. Interested readers will find all the details in the following Sub-section (Sub-sect. III.2).

The first proposition states that any terminal configuration is legitimate (see definition in Sub-sect II.1.2) and in $(2, 2, 0)$.

Proposition 1. *In a terminal configuration, the set of edges $\{(w, m) \in E\ :\ w.marriage = m \wedge m.marriage = w\}$ is non-empty and is a stable matching. This configuration is in $(\{2\}, \{2\}, 0)$.*

Then, we study the convergence. Lemmas 7 - 16 establish that from any configuration in $\mathcal{C}^1$, in $O(n^4)$ moves, an execution reaches a configuration in $(\{1.5\}, \{1.5\}, 0)$. That is made by showing that the sum of the regrets of married women is regularly decreasing. Notice that this property cannot be derived directly from a similar result for the centralized two-phased algorithm of Ackermann *et al.*, since it is based on the "best response" dynamics that are used there (in phase 1). As already explained before, since marriages, divorces and detection of BPs cannot be done instantaneously under a distributed setting, it is difficult and costly to realize such dynamics.

Then, Lemmas 17 - 29 and Proposition 2 below establish that every execution, starting in $(\{1.5\}, \{1.5\}, 0)$, reaches a configuration in $(\{2\}, \{2\}, 0)$. From there, in every

reachable configuration, nodes are in phase 2. Recall that in a configuration $(\{2\}, \{2\}, 0)$, there may be still unmarried nodes.

Proposition 2. *Every execution takes $O(n^4)$ moves to reach a configuration C in $(\{2\}, \{2\}, 0)$. Moreover, every configuration reached from C is in $(\{2\}, \{2\}, X)$ for $X \geq 0$.*

Proposition 2 ensures that the condition required by Proposition 3 (all nodes in phase 2) holds after $O(n^4)$ moves. Proposition 3 concerns precisely a segment of execution, in which nodes stay in phase 2, and establishes a bound $O(n^4)$ on the length of this segment. The bound is obtained by showing that the sum of the regrets of married men is strictly decreasing. Notice that, as before (for phase 1), this property cannot be directly derived from a similar result for the centralized two-phased algorithm of Ackermann *et. al.*. Then, by Proposition 3, from configurations in $(\{2\}, \{2\}, X)$ for $X \geq 0$, a terminal configuration is obtained in additional $O(n^4)$ moves (this is proven through Lemmas 30 - 37 and Corollary 1).

Proposition 3. *Let $\mathcal{E}$ be a sub-execution such that, in every configuration, all nodes are in phase 2. Nodes can execute at most $O(n^4)$ moves in $\mathcal{E}$.*

Finally, Proposition 1, 2 and 3 altogether imply the main theorem below.

Theorem 1. *Any execution takes $O(n^4)$ moves to reach a terminal configuration where the set of edges $\{(w, m) \in E : w.marriage = m \wedge m.marriage = w\}$ is a stable matching.*

III.2 - Detailed Proofs

In this section we prove the technical lemmas needed for Proposition 1 (Lemmas 1 - 6), Proposition 2 (Lemmas 7 - 29) and Proposition 3 (Lemmas 30 - 37). The three propositions yields the main Theorem 1, as described in Sub-sect. III.1.

III.2.1 - Properties of the Terminal Configurations

We start by proving that terminal configurations are configurations in $(2, 2, 0)$ (Lemma 5) by showing that in all other sets of configurations, there is at least one node eligible for a rule. Then, in Proposition 1, we prove that terminal configurations have a set of edges such that $(w, m) \in E : w.marriage = m \wedge m.marriage = w$ is a stable marriage.

Lemma 1. *Let C be a terminal configuration. For a node $m \in$ Men (resp. $w \in$ Women), $\mathsf{IncoherentPointersM}(m)$ (resp. $\mathsf{IncoherentPointersW}(w)$) is* `False`.

Proof. A node satisfying this predicate is eligible for **Reset**, whence a contradiction with the fact that the configuration is terminal. □

Lemma 2. *Let C be a terminal configuration and let $m \in$ Men. Then* $\mathsf{AllCoherentPhase}(m)$ *is* `True` *in C.*

Proof. Assume that AllCoherentPhase(m) is `False`, by contradiction. Then there exists a woman $w \in$ Women such that:

1. if $m.phase = 2$ then $w.phase \in \{1, 1.5\}$
2. if $m.phase = 1$ then $w.phase = 2$

IncoherentPointersW(w) is `False` by Lemma 1. If $w.phase \in \{2, 1.5\}$ then w is eligible for **ToPhase1** since $m \in \mathcal{N}(w)$ and because of points 1 and 2. If $w.phase = 1$ then m is eligible for **ToPhase1**. This contradicts the fact that the configuration is terminal. □

Lemma 3. *Let C be a terminal configuration and v be a node. If there exists a node $u \in \mathcal{N}(v)$ such that $v.marriage = u$ then $u.marriage = v$.*

Proof. Assume first that $v \in$ Men and that $v.marriage = u \in$Women. If $u.marriage \neq v$, then IncoherentPointersM(v) is `True`, which is not possible in a terminal configuration, by Lemma 1.

Assume now that $v \in$ Women, that $v.marriage = u$ with $u \in$Men and that $u.marriage \neq v$. Necessarily $u.proposal = v$ or IncoherentPointersW(v) is `True`, which is not possible by Lemma 1. There are two cases.

- Assume first that $u.marriage =$ `Null`. Then u is eligible for **Confirm** since priority(u,v) < `Null`. This yields a contradiction.
- Assume now that $u.marriage = v_1 \in$ Women. Necessarily priority(u,v) > priority(u,v_1) or u is eligible for **Confirm** (since IncoherentPointersM(u) is `False` by Lemma 1 and that AllCoherentPhase(u) is `True` by Lemma 2). Observe now that the inequality priority(u,v) > priority(u,v_1) implies that IncoherentPointersW(v) is `True`. This is not possible by Lemma 1.

□

Lemma 4. *Let C be a terminal configuration. No node is in phase 1 in C.*

Proof. Assume by contradiction that there exists a node in phase 1, in a terminal configuration C.

We consider first the case in which there exists $w \in$ Women in phase 1. Let $m \in$ Men. IncoherentPointersM(m) and IncoherentPointersW(w) are `False` by Lemma 1. Observe now that m is in phase 1 or eligible for **ToPhase1**. Thus we can assume that all men are in phase 1 as well. There are two different cases for a woman w.

- She is married (Married(w) is `True`) and she forms a BP with some node $m_1 \in$ Men. Assume without loss of generality that m_1 corresponds to BestMarriage(w). Necessarily $w.proposal = m_1$ or w eligible for **Propose1**.

  Observe first that BestProposal(m_1) $\neq$ `Null` since $w.proposal = m_1$ and both are in phase 1.

  - If $m_1.proposal = w$ then w is eligible for **Confirm1** since Response(w) is `True` and IncoherentPointersW(w) is `False`, by Lemma 1. This yields a contradiction.

– Assume now that $m_1.proposal \neq w$. First, AllCoherentPhase(m_1) is `True` by Lemma 2 and BestProposal(m_1) $\neq$ `Null`. If AlreadyEngaged(m_1) is `False` then m_1 is eligible for **Accept**. Therefore assume that AlreadyEngaged(m_1) is `True`. This implies that there exists $w_2 \in$ WOMEN such that $w_2 = m_1.proposal$ and $w_2.proposal = m_1$ or $w_2.marriage = m_1$. By definition of the predicate, $w_2 \neq$ `Null`. Now $m_1.marriage \neq w_2$ or IncoherentPointersM(m_1) is `True`, which is not possible by Lemma 1. There are two cases.

First, if $w_2.marriage = m_1$ and priority(m_1,w_2) $<$ priority($homme_1$,$m_1.marriage$) then m_1 is eligible for **Confirm**. If priority(m_1,w_2) $>$ priority(m_1,$m_1.marriage$) then IncoherentPointersW(w_2) holds and w_2 is eligible for **Reset** which yields a contradiction.

Thus consider the second case in which $w_2.marriage \neq m_1$. Observe first that w_2 is either in phase 1 or 1.5 or AllCoherentPhase(m_1) is `False` since m_1 is in phase 1. This is not possible by Lemma 2. We consider these two cases.

∗ Assume first that w_2 is in phase 1. There are two sub-cases.

· w_2 is married with some node $m_2 \in$ MEN. If BlockingPairW(w_2) is `False` then w_2 eligible for **ToPhase1.5**. If it is `True`, then by definition of the predicate, the set $\mathcal{C}_{w_2}$ is not empty and BestMarriage(w_2) $\neq$ `Null`. Recall also that $w_2.proposal = m_1$. Thus, if BestMarriage(w_2) $= m_1$ then w_2 eligible for **Confirm1**. Otherwise, if BestMarriage(w_2) $\neq m_1$ then w_2 eligible for **Propose1**. This yields a contradiction.

· w_2 is not married by definition, BlockingPairW(w_2) is `False`. In that case, it is eligible for **ToPhase1.5**. This yields a contradiction.

∗ Secondly assume that w_2 is in phase 1.5. If BlockingPairW(w_2) is `False` then it is eligible for **ToPhase2**. If it is `True`, it is eligible for **ToPhase1**. This yields a contradiction.

- She is single or married with no BP: **ToPhase1.5** can be applied. Thus, in a terminal configuration C, women are not in phase 1. Assume now that $\exists m \in$ MEN with $m.phase = 1$ in C. Women can either be in phase 1.5 or 2 (since women are not in phase 1, as proved above).

  If some woman is in phase 2, she is eligible for **ToPhase1** since $m.phase = 1$. Thus we can assume that all women are in phase 1.5. If there exists a woman w such that BlockingPairW(w) is `True`, then she is eligible for **ToPhase1**. Thus, this predicate is `False` for every woman. There are two cases.

  – Assume first that BlockingPairM(m) is `True`. Let w be the woman which forms a BP with m. Then, BlockingPairW(w) is `True`. This is not possible as shown above.

  – Therefore assume BlockingPairM(m) is `False`. Then, m eligible for **ToPhase1.5**, which yields a contradiction.

□

Lemma 5. *A terminal configuration is in* $(\{2\}, \{2\}, 0)$.

Proof. We prove this lemma by contradiction. Let C be a terminal configuration not in $(\{2\}, \{2\}, 0)$. Since by Lemma 4 no node is in phase 1, we have that C is in one of the following sets:

1. $(\{1.5\}, \{1.5\}, X)$
2. $(\{1.5, 2\}, \{1.5\}, X)$
3. $(\{1.5, 2\}, \{2\}, X)$
4. $(\{2\}, \{1.5, 2\}, X)$
5. $(\{1.5\}, \{1.5, 2\}, X)$
6. $(\{1.5, 2\}, \{1.5, 2\}, X)$
7. $(\{1.5\}, \{2\}, X)$
8. $(\{2\}, \{1.5\}, X)$
9. $(\{2\}, \{2\}, X)$, $X > 0$.

In case 1, all nodes are in phase 1.5. If $X = 0$ then women can apply **ToPhase2**. If $X \neq 0$ then the woman in a BP can apply **ToPhase1**.

In case 2, all women are in phase 1.5, men are in phase 1.5 or 2 (with at least one in each phase). Women in phase 1.5 are eligible for **ToPhase1**. The same holds for cases 6 and 8.

For cases 3, all women are in phase 2, men are in phase 1.5 or 2. If there are BPs, women in these pairs are eligible for **ToPhase1**. Otherwise, men in phase 1.5 are eligible for **ToPhase2**.

For cases 4, all men are in phase 2 and women are in phase 1.5 or 2. If there are BPs, women in these pairs are eligible for **ToPhase1**. Otherwise, women in phase 1.5 are eligible for **ToPhase2**.

For cases 5, all men are in phase 1.5 and women are in phase 1.5 or 2. If there are BPs, women in these pairs are eligible for **ToPhase1**. Otherwise, men in phase 1.5 are eligible for **ToPhase2**.

For cases 7, if there are BPs, women in these pairs are eligible for **ToPhase1**. Otherwise men are eligible for **ToPhase2**.

Finally consider configurations in $(\{2\}, \{2\}, X)$, $X > 0$. Women which have a BP are eligible for **ToPhase1**. This concludes the overall proof. □

Lemma 6. *In a terminal configuration, for every woman* w, $\mathsf{Married}(w)$ *holds.*

Proof. Let C be a terminal configuration. By Lemma 5, C is in $(\{2\}, \{2\}, 0)$. Assume by contradiction that there exists $w \in$ WOMEN for which $\mathsf{Married}(w)$ does not hold. This implies that there is at least one man m for which $\mathsf{MarriedM}(m)$ does not hold, since the graph is bipartite complete with the same number of men and women.

There are two cases. First, if $m.marriage \neq$ `Null` then m is eligible for **Reset** since $m.marriage.marriage \neq m$ by definition of $\mathsf{MarriedM}(m)$.

Then if $m.marriage = \texttt{Null}$, it can be also assumed that $w.marriage = \texttt{Null}$. Otherwise, by definition of $\mathsf{Married}(w)$, the disjunction $w.marriage.marriage \neq w \vee w.marriage.proposal \neq w$ holds, which implies that w is eligible for . Altogether, we can assume that $m.marriage = w.marriage = \texttt{Null}$.

We have first that C_w, the set of nodes u such that (w, u) is a BP, is not empty, since both m and w prefer each other to their current (`Null`) marriage. Assume without loss of generality that $m = \mathsf{BestMarriage}(w)$. If $w.proposal \neq m$ then w is eligible for **Propose2**. Thus assume that $w.proposal = m$, implying that $\mathsf{BestProposal}(m) \neq \emptyset$. Thus let $w_1 = \mathsf{BestProposal}(m)$. If $\mathsf{AlreadyEngaged}(m)$ is `False` then m is eligible for **Accept**, which yields a contradiction. Then assume that it is `True`. By definition of this predicate, $m.proposal \neq \texttt{Null}$. Let w_2 such that $w_2 = m.proposal$. We also have by definition of $\mathsf{AlreadyEngaged}(m)$ that $w_2.proposal = m$ or $w_2.marriage = m$.

In the second case, m is eligible for **Confirm** since $\mathsf{priority}(m,w_2) < m.marriage = \texttt{Null}$. In the first case, if $w_2.marriage \neq \texttt{Null}$ then it is eligible for **BadInit**. Thus assume that $w_2.marriage = \texttt{Null}$. If $\mathsf{BestMarriage}(w_2)$ is m then w_2 is eligible for **Confirm2** and otherwise it is eligible for **Propose2**. This concludes the proof. □

Proposition 1. *In a terminal configuration, the set of edges* $\{(w, m) \in E : w.marriage = m \wedge m.marriage = w\}$ *is a stable marriage.*

Proof. By Lemma 5, a terminal configuration is in $(\{2\}, \{2\}, 0)$, which implies that there are no BPs in such a configuration. By Lemma 6 and Lemma 3 all nodes are matched. □

III.2.2 - Convergence Proof

There are multiple techniques for proving the convergence of a self-stabilizing algorithm (Definition 3). Here we adopt a method introduced by Gouda and Multari [GM91], called convergence stairs or sometimes attractors. The image of a stair describes well the method. For reaching the desired set of legitimate configurations, the algorithm proceeds step by step, and the proof consists in showing that, once one step has been reached, the following step will be necessarily reached too. In our case, the steps of the stair correspond to the sets of configurations resulting from Definition 5. This technique allows to compute an upper bound of the stabilization time, by adding the upper bounds of moves, necessary for going from one step to another.

Recall that $\mathcal{C}^{1W}$ (resp. $\mathcal{C}^{1M}$) denotes the set of configurations having at least one women (resp. one man) in phase 1 and that $\mathcal{C}^1 = \mathcal{C}^{1W} \bigcup \mathcal{C}^{1M}$.

III.2.2.1 - Convergence to ({2}, {2}, 0)

Lemma 7. *Let* C *be a configuration in* $\mathcal{C}^{1M}$ *where* $\forall w \in$ WOMEN: $w.phase = 2$. *Any execution starting from* C *reaches a configuration* C' *in* $\mathcal{C}^{1W}$ *in* $O(n)$ *moves.*

Proof. In C, a woman w in phase 2,

(a) either is eligible for **Reset** if $\mathsf{IncoherentPointersW}(w) = \texttt{True}$ (this rule does not change the phase number).

(b) or is eligible for **ToPhase1** since at least one man is in phase 1.

Others rules cannot be applied since at least one man is in phase 1 and w in phase 2.

- A man m in phase 1 is eligible for **Reset** if $\mathsf{IncoherentPointersM}(m) = \texttt{True}$.
- A man in phase 2 is eligible either for **Reset** if $\mathsf{IncoherentPointersM}(m) = \texttt{True}$ (after a **Reset**, $m.phase = 1$), or for **Confirm** and **Accept**. Indeed, if $\forall w \in$WOMEN: $w.phase = 2$, if $m.proposal \neq \texttt{Null} \wedge m.proposal.marriage = m$, m is eligible for **Confirm**. Furthermore, after this move, $\mathsf{AlreadyEngaged}(m) = \texttt{False}$ and, if a better woman is proposing to m, $\mathsf{BestProposal}(m) \neq \texttt{Null}$. Thus, m can also be eligible for **Accept**.
- A man in phase 1.5 is eligible either for **Reset** or **ToPhase2** (all women are in phase 2). Indeed, after a **Reset**, $m.phase = 1$ and m is no more eligible for **ToPhase2**. Reciprocally, after a **ToPhase2**, m cannot be eligible for **Reset** since $\mathsf{IncoherentPointersM}(m) = \texttt{False}$ for **ToPhase2**.

We enumerate all possibles sets of configurations with all women in phase 2 and we ount the number of moves in the corresponding executions:

A. $(\{1\}, \{2\}, X \geq 0)$: first at most n men's and n women's **Reset** (both do not change nodes' phases), then at most n women's **ToPhase1**. After these $O(n)$ moves a configuration in $\mathcal{C}^{1W}$ is reached.

B. $(\{1, 2\}, \{2\}, X \geq 0)$: first, at most n women's **Reset** and $n-1$ men's **Reset** (men in phase 1). Furthermore, men in phase 2 can also be eligible for **Confirm** and/or **Accept**, that is at most $2n$ moves. After this moves, all remaining rules change the phase number. There are two cases that reach both a configuration in $\mathcal{C}^{1W}$:

   (a) all nodes are activated: at most $n-1$ men in phase 2 for **Reset** (if they had not been activated for **Accept**/**Confirm**) and at most n women for **ToPhase1** after $4n-1$ moves, that is $O(n)$ moves.

   (b) only men in phase 2 are activated (if they had not been activated for **Accept**/**Confirm**) for **Reset** and the reached configuration is in $(\{1\}, \{2\}, X)$ and then, in $\mathcal{C}^{1W}$ (see the point A. for explanations) after $O(n)$ moves.

C. $(\{1, 1.5\}, \{2\}, X \geq 0)$: at most n women and at most $n-1$ men in phase 1 can be activated for **Reset**. After that, there is four possible cases:

   (a) If all men in phase 1.5 are activated for **Reset** (at most $n-1$), the reached configuration is in $(\{1\}, \{2\}, X)$ A. and then, in $\mathcal{C}^{1W}$ (see the point A. for explanations) after $O(n)$ moves.

   (b) If all men in phase 1.5 are activated for **ToPhase2** or **Reset**, the reached configuration is in $(\{1, 2\}, \{2\}, X)$ and then in $\mathcal{C}^{1W}$ (see point B. for explanations) after $O(n)$ moves.

   (c) If some men in phase 1.5 are activated for **Reset** and/or some men in phase 1.5 for **ToPhase2** (at most $n-2$ men altogether), a configuration in $(\{1, 1.5, 2\}, \{2\}, X)$ is reached after $O(n)$ moves. From there, $\mathcal{C}^{1W}$ is reached

after at most $O(n)$ moves (see the point D. for explanations except the transition to a configuration in $(\{1, 1.5\}, \{2\}, X)$ since men in phase 2 have been activated for **ToPhase2** and cannot be now eligible for **Reset**).

(d) at most n men are activated (for **ToPhase2** or **Reset**) but also at most n women for **ToPhase1**, the reached configuration is in $\mathcal{C}^{1W}$ after at most $O(n)$ moves.

D. $(\{1, 1.5, 2\}, \{2\}, X \geq 0)$: at most n women and at most $n-2$ men in phase 1 can be activated for **Reset**. Furthermore, at most $n-2$ men in phase 2 can be eligible for **Accept** and/or **Confirm**, that is at most $2n-2$ moves. After that, there is four possible cases:

(a) If all men in phase 1.5 and 2 are activated for **Reset** (at most $n-2$), the reached configuration is in $(\{1\}, \{2\}, X)$ A. and then, in $\mathcal{C}^{1W}$ (see the point A. for explanations) after $O(n)$ moves.

(b) If all men in phase 1.5 are activated for **ToPhase2** or **Reset**, the reached configuration is in $(\{1, 2\}, \{2\}, X)$ and then in $\mathcal{C}^{1W}$ (see point B. for explanations) after $O(n)$ moves.

(c) If all men in phase 2 are activated for **Reset**, the reached configuration is in $(\{1, 1.5\}, \{2\}, X)$ and then in $\mathcal{C}^{1W}$ (see point C. for explanations) after $O(n)$ moves.

(d) at most n men are activated (for **ToPhase2** or **Reset**) but also at most n women for **ToPhase1**, the reached configuration is in $\mathcal{C}^{1W}$ after at most $O(n)$ moves.

To sum up, from any configuration $C \in \mathcal{C}^1$ where $\forall\ w\ \in$ Women : $w.phase\ =\ 2$, any execution takes $O(n)$ moves to reach a configuration in $\in \mathcal{C}^{1W}$. □

Lemma 8. *Let C be a configuration in $\mathcal{C}^1$ where $\forall w \in$Women : $w.phase\ \in\ \{2, 1.5\}$. Any execution starting from C takes $O(n)$ moves to reach a configuration C' in $\mathcal{C}^{1W}$.*

Proof. In these sets of configuration, a women w in phase:

- 1.5 is eligible for **Reset** and **ToPhase1** if $\exists w \in$Women : $w.phase\ =\ 2$ or if BlockingPairW(w) = `True`.
- 2 is eligible for **Reset** and **ToPhase1**.

Since $\exists w_1, w_2 \in$Women : $w_1.phase\ =\ 2\ \wedge\ w_2.phase\ =\ 1.5$, a man m in phase

- 1 is only eligible for **Reset**.
- 2 is eligible either for **Reset** or for **ToPhase1**.
- 1.5 is eligible for **Reset**.

We enumerate all possibles sets of configurations with all women either in phase 2 or 1.5 and count the moves in the corresponding executions:

A. $(\{1\}, \{1.5, 2\}, X \geq 0)$: after at most $2n$ **Reset** (men and women), only at most $n-1$ women are eligible (women in phase 2, for **ToPhase1**). Thus, after at most $3n-1$ moves, that is $O(n)$ moves, a configuration in $\mathcal{C}^{1W}$ is reached.

B. $(\{1, 1.5\}, \{1.5, 2\}, X \geq 0)$: after at most $2n-1$ **Reset** (all men except one in phase 1.5 and n women), there is two cases:

   (a) Either only the last man in phase 1.5 is activated for the **Reset** and a configuration in $(\{1\}, \{1.5, 2\}, X)$ is reached after $2n$ moves. From there, by the point A., after $O(n)$ moves a configuration in $\mathcal{C}^{1W}$ is reached.

   (b) Or at most $n-1$ women (those which are in phase 2) and the last man in phase 1.5 are activated (respectively for **ToPhase1** and **Reset**), that is $3n-1$ moves, *i.e.* $O(n)$ moves. The reached configuration is then in $\mathcal{C}^{1W}$.

   Hence, from $(\{1, 1.5\}, \{1.5, 2\}, X \geq 0)$ and after $O(n)$ moves, a configuration in $\mathcal{C}^{1W}$ is reached.

C. $(\{1, 2\}, \{1.5, 2\}, X \geq 0)$: after at most $2n-1$ **Reset** (all men except one in phase 2 and n women), there are two cases.

   (a) Either at most $n-1$ men in phase 2 are activated for **ToPhase1** or **Reset**, that is $O(n)$ moves, and the reached configuration is in $(\{1\}, \{1.5, 2\}, X)$. By point A., after $O(n)$ moves, a configuration in $\mathcal{C}^{1W}$ is reached.

   (b) Or at most $n-1$ men in phase 2 (for **ToPhase1** or **Reset**) and at most n women are activated (for **ToPhase1**), that is $O(n)$ moves, and the reached configuration is in $\mathcal{C}^{1W}$.

D. $(\{1, 1.5, 2\}, \{1.5, 2\}, X \geq 0)$: after at most $2n-1$ **Reset** (all men except one in phase 2 and n women), there are four cases.

   (a) All men in phase 2 and 1.5 (at most $n-1$) are activated for **ToPhase1** or **Reset**, that is $O(n)$ moves, and the reached configuration is in $(\{1\}, \{1.5, 2\}, X)$. By point A., after $O(n)$ moves, a configuration in $\mathcal{C}^{1W}$ is reached.

   (b) All men in phase 2 (at most $n-2$) are activated for **ToPhase1** or **Reset**, that is $O(n)$ moves, and the reached configuration is in $(\{1, 1.5\}, \{1.5, 2\}, X)$. By point B., after $O(n)$ moves, a configuration in $\mathcal{C}^{1W}$ is reached.

   (c) All men in phase 1.5 (at most $n-2$) are activated for **Reset**, that is $O(n)$ moves, and the reached configuration is in $(\{1, 2\}, \{1.5, 2\}, X)$. By point C., after $O(n)$ moves, a configuration in $\mathcal{C}^{1W}$ is reached.

   (d) At most n men (for **ToPhase1** or **Reset**) and at most n women are activated (for **ToPhase1**), that is $O(n)$ moves, and the reached configuration is in $\mathcal{C}^{1W}$.

To sum up, from any configuration $C \in \mathcal{C}^1$ where $\forall w \in$WOMEN : $w.phase \in \{2, 1.5\}$, any execution takes $O(n)$ moves to reach a configuration in $\in \mathcal{C}^{1W}$. □

Lemma 9. *Let C be a configuration in $\mathcal{C}^1$ where $\forall w \in$WOMEN : $w.phase = 1.5$. Any execution starting from C takes $O(n)$ moves to reach a configuration C' in $\mathcal{C}^{1W}$ or $(\{1.5\}, \{1.5\}, X)$ with $X \geq 0$.*

Proof. Let us suppose, by contradiction, that starting from C, no C' in $\mathcal{C}^{1W}$ or $(\{1.5\},\{1.5\},X)$ with $X \geq 0$ is ever reached. Thus, women are only eligible for **Reset**. Indeed, since $\mathcal{C}^{1W}$ is not reached and women are in phase 1.5, **ToPhase1** is not activated. **ToPhase1.5** is not enabled since women are already in phase 1.5 and **ToPhase2** is not enabled since all men are not in phase 1.5 ($(\{1.5\},\{1.5\},X \geq 0)$ is not reached). Other rules are also not enabled because of women's phases.

Men have several enabled rules relying to their phases and to women's phases (that does not change since they are only eligible for **Reset**). A man m in phase:

- 1 is only eligible for **Reset** and for **ToPhase1.5** if BlockingPairM(m) = `False`.
- 2 is eligible either for **Reset** or for **ToPhase1**.
- 1.5 is eligible for **Reset** or for **ToPhase1** if BlockingPairM(m) = `True`.

Since women cannot change their phases and that there is no configuration in $\mathcal{C}^{1W}$ or $(\{1.5\},\{1.5\},X \geq 0)$, either a terminal configuration is reached or the execution reaches two times the same configuration (since the variables are bounded and if no terminal configuration is reached). By Proposition 1, the terminal configuration is in $(\{2\},\{2\},0)$. But since no man is eligible for **ToPhase2**, this case is impossible. Now consider the configuration that is reached twice. Because all rules change at least one value and the same configuration is reached twice, at least one node v sets the same values to variables twice. Since women are eligible only once for **Reset**, $v \notin$ Women.
If $v.phase = 1$, there are two cases. Either v is first activated for **Reset** and then for **ToPhase1.5**. But after this move, he is no more eligible (**Reset** not enabled because $v.proposal$ = `Null` after **ToPhase1.5**). Or v is only activated for **ToPhase1.5**. In both cases, the variables of v have different values.
If $v.phase = 1.5$, there are also three cases. Either v is first activated for **Reset** and then for **ToPhase1.5**. But after this move, v is no more eligible (**Reset** is not enabled because $v.proposal$ = `Null` after **ToPhase1.5**), or it is only activated for **ToPhase1**, or it is activated for **ToPhase1** and **ToPhase1.5**. In the first two cases, the variables of v have different values. For the third case, v has been activated for **ToPhase1** because of a BP. If he is activated for **ToPhase1.5**, BlockingPairM(m) = `False`. This case happens if the woman w involved in the BP is now considered as single by v after a **Reset** (*i.e.* she was not married, but her marriage pointer was not `Null`). Then, v can have the same state after these two moves but w has now a new pointer value.
If $v.phase = 2$, there are four cases. Node v is either activated for **Reset** and **ToPhase1.5**, for **ToPhase1** and **ToPhase1.5**, only for **Reset** or only for **ToPhase1**. In this four cases, the phase number is not the same, *i.e.* the state is not the same. This lead to a contradiction. Note that, men can reach phases 1 or 1.5 from all phases. Thus, if $\forall m \in$Men: BlockingPairM(m) = `True`, $m.phase = 1.5$ in C and if the two nodes involved in the BPs are not activated in the execution until C', all other men shift to phase 1.5, C' is in $(\{1.5\},\{1.5\},X \geq 0)$. Otherwise, at least a woman is activated and the configuration is in $\mathcal{C}^{1W}$.

Thus, there is no execution not reaching a configuration in $\mathcal{C}^{1W}$ or in $(\{1.5\},\{1.5\},X \geq 0)$. Furthermore, women can only be activated for **Reset** and **ToPhase1** and each man can at most be activated for 2 moves, that is $O(n)$ moves in total. □

Lemma 10. *Let C be a configuration in $\mathcal{C}^1$. Any execution starting from C takes $O(n)$ moves to reach a configuration C' in $\mathcal{C}^{1W}$ or in $(\{1.5\}, \{1.5\}, X \geq 0)$.*

Proof. By definition of $\mathcal{C}^1$, in C, $\exists\, v \in V : v.phase = 1$. Necessarily, $v \in$ Men, otherwise the configuration would be already in $\mathcal{C}^{1W}$. Let us enumerate all possible sets of configuration in which C can be:

- $\mathcal{C}^1$ where $\forall\, w \in$ Women : $w.phase = 2$. By lemma 7, a configuration in $\mathcal{C}^{1W}$ is reached after $O(n)$ moves.
- $\mathcal{C}^1$ where $\forall\, w \in$ Women : $w.phase \in \{2, 1.5\}$. By lemma 8, a configuration in $\mathcal{C}^{1W}$ is reached after $O(n)$ moves.
- $\mathcal{C}^1$ where $\forall\, w \in$ Women : $w.phase = 1.5$. By lemma 9, a configuration in $\mathcal{C}^{1W}$ or in $(\{1.5\}, \{1.5\}, X \geq 0)$ is reached after $O(n)$ moves.

To sum up, from a configuration in $\mathcal{C}^1$, after $O(n)$ moves, a configuration in $\mathcal{C}^{1W}$ or $(\{1.5\}, \{1.5\}, X \geq 0)$ is reached. □

Lemma 11. *Let C be a configuration in $\mathcal{C}^{1W}$. Any execution starting from C takes $O(n^2)$ moves to reach a configuration C' in $(\{1\}, \{1, 1.5\}, X)^{\times} \bigcup (\{1.5\}, \{1.5\}, 0)$.*

Proof. Let w_1 be the woman in phase 1 (because $C \in \mathcal{C}^{1W}$). Let us analyze the moves of other nodes.

Let m be a node in Men. Independently of its phase number, m can be eligible for **Reset**. After the move, $m.phase = 1$. Otherwise, if:

1. $m.phase = 2$ or $m.phase = 1.5$, m is only eligible for **ToPhase1** if he was not eligible for **Reset**: one of its neighbors is in phase 1. If a woman in phase 2 proposes/confirms to m, it cannot accept/confirm (AllCoherentPhase(m) is `False`).

2. $m.phase = 1$, m can be eligible for different rules. If m is eligible for **Accept** or **Confirm**, there are incoherent pointers. Indeed, if a woman w has been activated for **Propose1/2** or **Confirm1/2**, all men are in phase 1. Furthermore, if m is eligible for **Accept** or **Confirm**, all women are in phase 1 or 1.5. Then, the configuration is already C'.

   Now, if m is eligible for **Accept** because a proposal of a woman w, the pointers of w are incoherent and all women are either in phase 1 or 1.5. But men are not all in phase 1. Then, w is not eligible for **Confirm** until no man is in phase 1, that is the configuration C'. No new marriage can be done before reaching C'. If m is eligible for **Confirm** to w, that means that m is already married with w. Then after its move, no new marriage is done.

   Since these moves are possible only because of incoherent pointers and each woman has only two pointers, there are at most n moves (**Accept** or **Confirm**) resulting from their incoherent pointers.

Then, a man is eligible for at most two rules (**Reset** or **ToPhase1** and **Accept** or **Confirm**. That means that in altogether $O(n)$ moves, men are all in phase 1.

Let w be a node in Women. **Reset** is enabled for w. If

- $w.phase = 1.5$, w is eligible for **ToPhase1** if w is involved in a BP or if a man is in phase 2.
- $w.phase = 1$, w is eligible for **Propose**, **Confirm** or **ToPhase1.5** if w is not involved in a BP.
- $w.phase = 2$, w is eligible for **Propose** or **Confirm**, **BadInit** (if no reset) if all men are in phase 2 and for **ToPhase1** if a man is in phase 1 or if w is involved in a BP.

In short, in $O(n^2)$ moves, the execution reaches *C'*.

In the sequel, we consider a particular node and we enumerate its possible interactions with the perspective of determining upper bounds to their number. Naturally, there is a great number of cases to examine, but no one is complicated and its results come from a simple examination of the rules of the algorithm.

Let v be a node in phase 1. Firstly, consider the case of $v \in$ Men. Other men may be in any phase. Let w be in Women. If its pointers are incoherent, w can be eligible for **Reset** . For the other rules, we consider the different sub-cases:

1. $w.phase = 2$: w is only eligible for **ToPhase1** (one of its neighbors is in phase 1), **BadInit** and **Reset**. The first two rules set the phase of w to 1, after a **Reset** if necessary. **Reset** does not affect the phase value.
2. $w.phase = 1.5$: if w is involved in a BP, its is eligible for **ToPhase1**. Otherwise, w is eligible for **Reset**, **ToPhase1** and **ToPhase2**. If w is eligible for **ToPhase2**, all men are in phase 1.5, but v could move to 1.5 from 1 only if all women are in phase 1.5 too, thus the configuration *C'*has been reached. Otherwise, w is eligible either for **Reset** or for **ToPhase1**. **ToPhase1** sets $w.phase$ to 1, after a **Reset** if necessary. **Reset** does not affect the phase value.
3. $w.phase = 1$: if some men are not in phase 1, w has no enabled rule except the **Reset**. Otherwise, w is eligible for:
   - **Reset** only once. Indeed, after **Reset** w is single in phase 1 and cannot be married (propose or confirm) in this phase.
   - **ToPhase1.5** only once and when w is single or married without a BP. In phase 1.5 and after man's **Reset**, w may detect a BP with this man. She is then eligible for **ToPhase1**. To be eligible again for **ToPhase1.5**, w has to propose, a man has to accept, and w then has to confirm. Till confirmation, w has to stay in phase 1 and all men have to be in phase 1. They cannot change their phase till confirmation of w. Moreover, a man can accept only if all women are in phase 1 or 1.5. Thus *C'* is already reached, contradicting the fact that w eligible for **ToPhase1.5** for the second time before *C'* is reached.
   - **Propose1**: since w may propose only once to each man (and not to her spouse), w is eligible for this rule at most $n-1$ times. Indeed, if w propose to a man m, BestMarriage selects the best possible spouse. But if pointers of men are incoherent, w cannot detect the BP with a better spouse m_1 and propose to m. After the activation of m_1 for the **Reset**, she can propose. And this case may happen $n-1$ times.

- **Confirm1**: it is a special case of the previous case. Indeed, since women are not all in phases 1 or 1.5, men cannot accept a proposal. But in the configuration C, the proposal pointer of a man m can be already set to w. Then, if w proposes to this man, w can also be eligible for **Confirm1**. Then, w has resolved a BP (because in the definition of $\mathcal{C}_v$, w checks if its proposal is also more interesting for m). Man m is eligible for **Confirm** when all women are in phase 1 or 1.5, that is the configuration C'.

The worst case is when all women are in phase 1 except one in phase 2 and men all in phase 1, because they can propose to men $n-1$ times. Indeed, each woman is eligible for $O(n)$ moves. That is altogether, $O(n^2)$ moves of women and then all women are either in phase 1 or 1.5.

Now, let us consider the case of $v \in$Women. Let us analyze the other nodes next moves. Let $m \in$ Men. In all cases, m is eligible for **Reset** and then its phase is set to 1. Otherwise, if:

1. $m.phase = 1$, m has nothing to do. In fact, if m is eligible for **Accept** or **Confirm**, all men are in phase 1 because otherwise women could not propose or confirm a marriage. If all women are in phase 1 and there is an incoherent pointer, a woman possibly has her pointer of proposal to m and m is eligible for **Accept**, but the woman will not answer while all men are not in phase 1 (and then, the configuration is C'). Furthermore, if all pointers of a woman are incoherent (the two pointers are set to m for example) m cannot be eligible for these two rules because of the definition of $\mathcal{P}_v$. A man is eligible at most once for one of these rules.

2. $m.phase = 2$ or $m.phase = 1.5$, m is eligible for **ToPhase1** if he was not eligible for **Reset**: one of his neighbors is in phase 1. If a woman in phase 2 proposes to m, m cannot accept (AllCoherentPhase(m) is `False`).

Then, a man is eligible for at most two rules. That is altogether $O(n)$ moves after which men are all in phase 1.

In short, in $O(n^2)$ moves, the system reaches C'. □

In the two following lemmas (Lemmas 12 and 11), we introduce a norm function and we precise conditions in which this function is strictly decreasing. A norm function is a function from a set of configurations into a well ordered set (ordered set with no infinite strictly decreasing sequence). If there exists a norm function on a subset of configurations, an execution on this subset either terminates or reaches a configuration outside the subset. Then exhibiting a norm function allows to prove that an execution does not remain indefinitely at the same step of a convergence stair. The considered norm function is based on the notion of regret ([Knu76, GI89]).

Let MarriedWomen(C) be the set of nodes v in Women that are married in the configuration C. Let $\mathcal{R}_w(C)$ be the sum of the regret of nodes v in MarriedWomen(C):

$$\mathcal{R}_w(C) = \sum_{v \in \text{MarriedWomen}(C)} \text{priority}(v, v.marriage(C))$$

Lemma 12. *Let C be a configuration in $(\{1\},\{1,1.5\},X>0)^{\times}$. Any execution starting from C takes $O(n^2)$ moves to reach a configuration C' in $(\{1\},\{1,1.5\},Y\geq 0)\bigcup(\{1\},\{1\},Y\geq 0)$ such that $\mathcal{R}_w(C) > \mathcal{R}_w(C')$.*

Proof. Let T be the transition $C_1 \rightarrow C_2$ and let us consider moves such that $\mathcal{R}_w(C_1) > \mathcal{R}_w(C_2)$. Note that **Confirm1** is the only rule that decreases $\mathcal{R}_w(C)$ (no rule increases the sum). Indeed, a woman w_0 is married if $[(w_0.marriage.marriage = w_0) \vee (w_0.marriage.proposal = w_0)]$ ($\mathsf{Married}(w_0)$) and **Confirm1** set her marriage pointer to a man m_0 if $m_0.proposal = w_0$ ($\mathsf{Response}(w_0)$). Furthermore, $\mathsf{BestMarriage}(w_0) = w_0.proposal$ checks if $w_0.proposal$ is the best man for w_0 in the current configuration. Then, if a woman is activated for **Confirm1**, she gets married with a new man, better ranked than its actual partner, *i.e.*, decreases $\mathcal{R}_w$.

Now, we consider all enabled rules in C that do not change $\mathcal{R}_w$ and we count how many times each rule is enabled for each node. First, let m be in Men. Relying on its state and the state of the system , m is eligible for:

- **Reset**, once. Indeed, since **Reset** is enabled only if $m.marriage \neq \texttt{Null}$ and sets $m.marriage$ to `Null` between the two activations of **Reset**, $m.marriage$ is set to some value. So m has been activated for **Confirm** (the only rule that sets $m.marriage$). Thus, if a man is eligible twice, a woman has been activated for **Confirm1** and $\mathcal{R}_w$ has decreased. Then, **Reset** is enabled only once.

- **ToPhase1.5**: m is eligible only if all women are in phase 1.5, $\mathsf{BlockingPairM}(m)$ = `False` and $\mathsf{AlreadyEngaged}(m)$ = `False`. Furthermore, m is eligible only once for this rule. Indeed, if m is activated twice for this rule (let us called this two transitions A and B), it means that before A and B all women were in phase 1.5. But between them, m has shift back to phase 1 (otherwise he would not be eligible the transition B). The only reason for which a woman shift to phase 1 between A and B is a BP. Moreover, she was in phase 1.5 for B. Then a BP has been resolved between A and B.

- **ToPhase1**: since men are eligible for **ToPhase1.5** and there are X BP(s), at least one man (involved in a BP $(m_1,\ w_1)$) will stay in phase 1. Woman w_1 will be activated for **ToPhase1** ($\mathsf{BlockingPairW}(w)$ is `True`). After this move, if m is in 1.5 he is eligible for **ToPhase1**, at most once between each resolved BP.

- **Accept**: there is a woman w such that $w.proposal = m$ and her proposal is the best proposal for m in C, *i.e.* $\mathcal{P}_v = w$. Since $\mathcal{P}_v$ is defined with respect to the preferences of the proposing woman and of m, m accepts only if the marriage is beneficial for both of them. If m is activated for **Accept** in C_0such that $C \xrightarrow{*} C_0$ (such that $C_0 \xrightarrow{*} C_1$), m can be activated once again for **Accept** in C_1, if a woman w_1 has been activated for **Propose** for m and that w is proposing to a better ranked man. Then, m is eligible $O(n)$ times (for each woman).

- **Confirm**: when m confirms, m is already considered married (after **Confirm1** of the woman). But he is not eligible twice, because it would implies a new marriage (and then a woman would have been activated for **Confirm1**).

Altogether, a man is eligible for at most $O(n)$ moves, that is $O(n^2)$ moves for all men.

Now, let w be in WOMEN. The 4 following rules may be enabled for w:

- **Reset**: if she is eligible for **Reset**, that means she is not married. She cannot be involved in a BP and she is at most eligible for another move: **ToPhase1.5**.
- **ToPhase1.5**: if she is single or not involved in a BP in phase 1. Woman w can be activated for this rule only once, otherwise it means she has gone back to 1 (because of a BP, see the next point). But if she is eligible again for **ToPhase1.5**, that means there are no more BPs.
- **ToPhase1**: if a woman w involved in a BP ($\mathsf{BlockingPairW}(w)$ is `True`) is in phase 1.5 (only once because of the previous point).
- **Propose1**: if $\mathsf{BlockingPair}(w)$ = `True`: she proposes to the best ranked man in $\mathcal{C}_v$. Woman w can be involved in at most $n-1$ BP(s) (if she is married with the worst ranked man), she can be eligible $n-1$ times for **Propose1**. Indeed, if, for a man m, $\mathsf{IncoherentPointersM}(m)$ = `True`, w cannot detect the BP (w, m). Then, she can propose to a first man m_1 before a man m_2 is activated for **Reset** and she detects the BP because m_2 is better ranked than m_1).

In overall, a woman is eligible for at most $O(n)$ moves, that is $O(n^2)$ moves for all women.

To summarize, nodes are eligible for at most $O(n^2)$ moves before that at least one woman w is eligible for **Confirm1**. Men are in phase 1 (since there was a BP before w's **Confirm1**) and women are either in phase 1 or 1.5. Thus, C' is reached. Note that the number of BPs is now $Y \geq 0$ but not necessarily $Y < X$: a BP (m, w) has been resolved but the previous spouse m_1 of w is now single. New BP(s) involving m_1 can appear after the resolution of the BP. □

Lemma 13. *Let C and C' be configurations in $(\{1\},\{1,1.5\},X)^{\times} \bigcup (\{1,1.5\},\{1.5\},X)^{\times}$ such that $C \xrightarrow{*} C$'. Let w be a woman. If $w.marriage(C) \neq w.marriage(C')$ then* $\mathsf{priority}(w, w.marriage(C)) > \mathsf{priority}(w, w.marriage(C'))$ *or* $w.marriage(C')$ = `Null`. *Thereby, $\mathcal{R}_w(C) > \mathcal{R}_w(C')$. Furthermore, w cannot be married again with $w.marriage(C)$ before being activated for* **ToPhase2**.

Proof. Let w be in WOMEN and m be in MEN. Since women are in phase 1, if $w.marriage(C) \neq w.marriage(C')$, there are two cases:

1. $w.marriage(C) = m$ and $w.marriage(C')$ = `Null` or
2. $w.marriage(C) = m$ and $w.marriage(C') = m_1$.

Indeed, since single women in phase 1 or 1.5 cannot been eligible for **Propose1**, the case $w.marriage(C)$ = `Null` and $w.marriage(C') = m$ is not possible ($\mathsf{Married}(w)$) in C is `False`). The first case appears if there is a BP (w_1, m) for some woman w_1: w_1 proposes to m ($w_1.proposal = m$) and m accepts/confirms the proposal ($m.proposal = w$ and then $m.marriage = w$). Then, w becomes single. We have $\mathcal{R}_w(C) > \mathcal{R}_w(C')$ because w is now single and does not count any longer and w_1 diminishes her regret.

ToPhase1.5 is now enabled for w since she is no more married (and cannot belong to a BP) and, after that, for **ToPhase2**.

The second case happen if (w, m_1) is a BP: m_1 is better ranked by w than m. Woman w resolves the BP by making a proposal ($w.proposal = m_1$) and a confirmation ($w.marriage = m_1$). Then, $\mathcal{R}_w(C) > \mathcal{R}_w(C')$. Woman w cannot be married once again with m implying the existence of a BP (w, m), because m is worse ranked than m_1: w cannot propose to m while she is married with m_1 ($\mathsf{BestMarriage}(w) \neq m$). Then, if there is still a BP involving w, she improves again her regret but can still not be married again with m. When there is no more BP involving w, she is only eligible for **ToPhase1.5** and **ToPhase2** if all men are in phase 1.5. □

In all stages that we considered up to know, the complexity in moves is an $O(n^2)$. Now we arrive at the hard part, which will weigh on the global complexity and which is an $O(n^4)$. The reason is simple: there can be $O(n^2)$ BPs in a configuration and the resolution of a single BP may take $O(n^2)$ steps. These $O(n^2)$ supplementary steps with respect to Ackermann *et al.*'s algorithm is caused by the distribution. While the centralized algorithm resolves a BP in one action, the distribution, together with an adversarial unfair scheduler, makes that there can be some delay before the BP is effectively solved. It seems that nothing can be done against that, but trying a completely different approach. This other approach is the object of the next chapter, but presently, we continue with the analysis of the first algorithm.

Lemma 14. *Let C be in $(\{1\}, \{1, 1.5\}, X)^{\times}$. Any execution starting from C takes $O(n^4)$ moves to reach a configuration C' in $(\{1\}, \{1, 1.5\}, 0)^{\times}$.*

Proof. Remind that $(\{1\}, \{1, 1.5\}, X)^{\times} \equiv (\{1\}, \{1.5\}, X) \bigcup (\{1\}, \{1, 1.5\}, X) \bigcup (\{1\}, \{1\}, X)$. If $X = 0$, there is nothing to prove except if C is in $(\{1\}, \{1\}, 0)$. Women are eligible for at most two rules: **Reset** and **ToPhase1.5**. Men can only be eligible for **Reset**. Then, if $X = 0$, after at most $2n$ **Reset** (men and women) and **ToPhase1.5** (women), that is $O(n)$ moves, the configuration C' is reached.

Now assume $X > 0$. Let us determine an upper bound on X. Since there is a BP $(w,\ m)$ only if w is married, w is involved in at most $n-1$ BP(s). Then, if each women is involved in $n-1$ BP, there are $O(n^2)$ BP(s).

By Lemma 12, one BP is resolved in $O(n^2)$ moves. Since each BP can be resolved at most once (Lemma 13), there is no more BPs after $O(n^4)$ moves. When a woman w is activated to confirm (resolving the last BP (w, m) by setting $w.marriage = m$), all men are in phase 1 and at least w is in phase 1 too ($\mathsf{AllCoherentPhase}(w)$ in **Confirm1**). Other women are either in phase 1 or 1.5 (**ToPhase2** cannot be applied since men are in phase 1). This configuration is in $(\{1\}, \{1, 1.5\}, 0)^{\times}$. □

Lemma 15. *Any execution starting from a configuration C in $(\{1\}, \{1, 1.5\}, 0)$ takes $O(n)$ moves to reach a configuration $C' \in (\{1\}, \{1.5\}, 0)$.*

Proof. Consider the enabled rules in configuration C. Men are only eligible for **Reset**. Indeed, they are not eligible for **Accept** or **Confirm**. Even if there are woman's incoherent pointers, men cannot accept because of the definition of $\mathcal{P}_v$. Indeed, $\mathcal{P}_v$ evaluates if the proposal is more interesting for both, the man and the woman and if the woman is married (if there exists still a BP). This yields to a contradiction with the fact that

there is no more BPs in C by definition. Men are not eligible for **Reset** since they are already in phase 1 (men) and also not for **ToPhase1.5** because of women in phase 1.

Women are eligible for **Reset** and **ToPhase1.5** (women in phase 1). Indeed, since there are no more BP, no woman is eligible for **Propose1** or even **Confirm1**. Concerning **ToPhase1**, women cannot be eligible, since there is no more BPs (women). **ToPhase2** and **BadInit** are not enabled because of nodes' phases.

Thus, after at most $2n$ **Reset**, at most $n-1$ women are eligible for **ToPhase1.5** (and men have no enabled rules), that is $O(n)$ moves, the system reaches a configuration C' in $(\{1\},\{1.5\},0)$. □

Lemma 16. *Any execution starting from a configuration C in $(\{1\},\{1.5\},0)$ takes $O(n)$ moves to reach a configuration $C' \in (\{1.5\},\{1.5\},0)$.*

Proof. Let us consider first women. They have no enabled rule except **Reset** since they are in phase 1.5 and not all men are in phase 1.5.

Now, let us consider men. As women are in phase 1.5 and cannot change their phase, men are only eligible for **ToPhase1.5**. Furthermore, they are eligible for **Reset**. Notice that each man is eligible for **Reset** before **ToPhase1.5** due to IncoherentPointersM.

Then, after $2n$ **Reset** (men and women) and n men's **ToPhase1.5**, that is $O(n)$ moves, the configuration $C' \in (\{1.5\},\{1.5\},0)$ is reached. □

Lemma 17. *In a configuration C in $(\{1.5\},\{1.5\},0) \bigcup (\{1.5\},\{1.5,2\},0)$, women are enabled for rules in the following set {**ToPhase2**, **Reset**} and men are only enabled for **Reset**. Furthermore if $C \to C'$, then the configuration C' is:*

- *in $(\{1.5\},\{1.5\},0) \bigcup (\{1.5\},\{1.5,2\},0)$ if only women's **Reset** are activated in the transition.*
- *in $(\{1.5\},\{1.5,2\},0) \bigcup (\{1.5\},\{2\},0)$ if at least one women's **ToPhase2** and no men's **Reset** is activated in the transition.*
- *in $\mathcal{C}^1$ if at least one men's **Reset** is activated in the transition.*

Proof. Let v be an eligible node in Men. By definition of C, $v.phase = 1.5$. Then, **Accept**, **Confirm** and **ToPhase1.5** cannot be applied. Since there is no woman in phase 1 and v is in phase 1.5, v cannot be eligible for **ToPhase1**. Furthermore, as there exists at least one woman in phase 1.5, **ToPhase2** is also not enabled. Thus, v is only eligible for **Reset**, if its pointers are incoherent.

Now, let v be in Women. Because men's phase is 1.5, v cannot be eligible for **Propose1**, **Confirm1**, **Propose2**, **Confirm**2, **BadInit** and **ToPhase1.5**. **ToPhase1** is also not enabled: there is no BP and there is no man in phase 1. Thus, the only possible rules for v are **ToPhase2** (if v is in phase 1.5) and **Reset** (if its pointers are incoherent).

To sum up, if in the transition $C \to C'$ at least one man (**Reset**) is activated, the configuration C' is in $\mathcal{C}^1$. Otherwise, if only women are activated, C' is in $((\{1.5\},\{1.5,2\},0) \bigcup (\{1.5\},\{2\},0)$ if at least one women's **ToPhase2** is activated or in $(\{1.5\},\{1.5\},0) \bigcup (\{1.5\},\{1.5,2\},0)$ if only women's **Reset** are activated. □

Lemma 18. *In a configuration C in $(\{1.5\},\{2\},0) \bigcup \{1.5,2\},\{2\},0)$, women are only enabled for* **Reset** *and men are enabled for rules in* {**ToPhase2**, **Reset**, **Accept**, **Confirm**}. *Furthermore if $C \to C'$, then the configuration C' is:*

- *in $(\{1.5\},\{2\},0) \bigcup \{1.5,2\},\{2\},0)$ if only* **Accept** *(men),* **Confirm** *(men) and* **Reset** *(women) are activated in the transition.*
- *in $(\{1.5,2\},\{2\},0) \bigcup (\{2\},\{2\},0)$ if at least one men's* **ToPhase2** *and no men's* **Reset** *are activated in the transition.*
- *in $\mathcal{C}^1$ if in C if at least one men's* **Reset** *is activated in the transition.*

Proof. Let v be an eligible node in WOMEN. By definition of C, there exists at least one man in phase 1.5. Then, **Propose1**, **Propose2**, **Confirm1**, **Confirm**2 and **BadInit** cannot be applied. Since women are already in phase 2 and there is no BP, **ToPhase2**, **ToPhase1.5** and **ToPhase1** are also not enabled. Then, if the pointer of v is incoherent, v is eligible only for **Reset** (the marriage is not reciprocal), otherwise, women have no enabled rules.

Now, let v be in MEN. Because $v.phase \neq 1$, v cannot activate **ToPhase1.5**. Since women are in phase 2, **ToPhase1** is also not enabled. Let us consider **Accept** and **Confirm** and the two possible cases:

- $v.phase = 1.5$. Because of $\mathsf{AllCoherentPhase}(v)$, these rules cannot be applied.
- $v.phase = 2$. If there is a woman w such that $w.proposal = v$, that $\mathsf{BestProposal}(v) \neq \texttt{Null}$ and if $\neg\mathsf{AlreadyEngaged}(v)$, v is eligible for **Accept**. But if v accepts the proposal and since the woman cannot answer in this configuration, there is no new marriage. Furthermore, if v is eligible for **Confirm**, he was already married (the woman had its identifier in her pointer of marriage). In any cases, that does not create a marriage and thereby also not a BP. Moreover, the phase of nodes activated for these rules is still 2.

Finally, we consider C', the new configuration after the transition from C. If at least one man has been activated for **Reset**, C' is in $\mathcal{C}^1$. If nodes have only been activated for **Accept** (men), **Confirm** (men) or **Reset** (women), C' is in $(\{1.5\},\{2\},0) \bigcup \{1.5,2\},\{2\},0)$. Otherwise, if at least one man has been activated for **ToPhase2** and none for **Reset**, C' is in $(\{1.5,2\},\{2\},0) \bigcup (\{2\},\{2\},0)$. □

Lemma 19. *Any execution starting from a configuration C in $(\{1.5\},\{1.5\},0) \bigcup (\{1.5\},\{1.5,2\},0) \bigcup (\{1.5\},\{2\},0) \bigcup (\{1.5,2\},\{2\},0)$ takes $O(n)$ moves to reach a configuration C' in $(\{2\},\{2\},0)$ or in $\mathcal{C}^1$ after at least one men's* **Reset**.

Proof. Let C be in $(\{1.5\},\{1.5\},0) \bigcup (\{1.5\},\{1.5,2\},0)$. By Lemma 17, men are eligible only for **Reset** and women for **Reset** and **ToPhase2**. If at least one man is activated, the reached configuration is C' in $\mathcal{C}^1$. Otherwise, women are eligible for **Reset** (each woman at most once, since women cannot set their pointers) and for **ToPhase2** (also at most once for each woman, since in phase 2 women are not eligible for **ToPhase2**). Thus, after $O(n)$ moves, either a configuration in $(\{1.5\},\{2\},0)$ or in $\mathcal{C}^1$ is reached.

From $(\{1.5\},\{2\},0) \bigcup (\{1.5,2\},\{2\},0)$, by Lemma 18, men are eligible for **ToPhase2**, **Reset**, **Accept** and **Confirm** and women are eligible only for **Reset**. If at least one

man is activated for **Reset**, the reached configuration is C' in $\mathcal{C}^1$. Otherwise, at most n men are eligible for **ToPhase2** and then, $n-1$ for **Accept** and **Confirm**. Furthermore, women are eligible for **Reset** (each woman at most once, since women cannot set again their pointers). Thus, after $O(n)$ moves, a configuration in $(\{2\},\{2\},0)$ or in $\mathcal{C}^1$ is reached.

In overall, from C, after $O(n)$ moves, a configuration either in $(\{2\},\{2\},0)$ or in $\mathcal{C}^1$ (if a man is activated for **Reset**) is reached. □

Lemma 20. *Let C be a configuration in $(\{1.5\},\{1.5\},X)$ where $X>0$. Any execution starting from C takes $O(n)$ moves to reach a configuration C' in $\mathcal{C}^{1W}$.*

Proof. Let us consider first a node v in Women. Since v is in phase 1.5 and all men are in phase 1.5, v is not eligible for **Propose1/2**, **Confirm1/2** or **ToPhase1.5**. Since there are X BP(s), some women are involved in these BPs. Women involved in a BP are eligible for **ToPhase1** and the others are eligible for **ToPhase2** (because of BlockingPairW). Once they are in phase 2, while all men are in phase 1.5, they can do nothing. Node v is also eligible for **Reset**.

Let us consider now a node v in Men. Since v is in phase 1.5, he cannot be eligible for **Accept**, **Confirm** and **ToPhase1.5**. **ToPhase1** and **ToPhase2** are also not enabled: there are some women in phase 1.5 and others in phase 2 (after **ToPhase2**). Then v can only be eligible for **Reset**.

Thus, after at most n **Reset** (women) + $(n-1)$ women's **ToPhase2** the only enabled rule is **ToPhase1** of a woman involved in a BP or a man's **Reset**. If only men are activated, $\mathcal{C}^1$ is reached, but, after this shift to phase 1, men that are not involved in a BP can shift back to phase 1.5 (at most $n-1$). Women involved in a BP are still eligible. Thus, after at most n **Reset** (women) + $(n-X)$ women's **ToPhase2** + n men's **Reset** + $n-1$ **ToPhase1.5**, women involved in a BP are eligible and activated for **ToPhase1**. Then, after $O(n)$ moves, a configuration C' in $\mathcal{C}^{1W}$ is reached. □

Lemma 21. *Let C be a configuration in $(\{1.5,2\},\{1.5\},X) \bigcup (\{1.5,2\},\{1.5,2\},X) \bigcup (\{2\},\{1.5\},X)$ with $X \geq 0$. Any execution starting from C takes $O(n)$ moves to reach a configuration C' in $\mathcal{C}^{1W}$ or in $(\{1.5\},\{1.5\},0)$.*

Proof. A common feature to the configurations specified in the statement is: $\exists\ w \in \text{Women} \wedge \exists\ m \in \text{Men} :\ w.phase = 1.5 \wedge m.phase = 2$

Let us consider first a node v in Men. Independently of phases, **Reset** can be enabled. Notice that if a man is activated for **Reset**, the configuration is immediately in $\mathcal{C}^1$. By lemma 10 and 20, after $O(n)$ moves, the reached configuration is $\mathcal{C}^{1W}$ or $(\{1.5\},\{1.5\},0)$. Since there exists at least one woman in phase 1.5 and a man in phase 2, AllCoherentPhase(v) is `False` and **Accept** and **Confirm** are not enabled. For the same reason, **ToPhase2** is also not enabled. Since nodes can only be in phase 1.5 or 2, **ToPhase1.5** cannot be activated. Concerning **ToPhase1**, it can be enabled only if the node v is in phase 2 (because there exists a woman in phase 1.5) or if v is involved in a BP. In short, men in phase 1.5 are only eligible for **Reset** and men in phase 2 are eligible for **Reset** or **ToPhase1** (any of this two rules is sufficient to reach a configuration in $\mathcal{C}^1$).

Now, let us consider a node v in Women. Independently of phases, **Reset** may be enabled. Then v cannot be eligible for **Propose1**, **Confirm1**, **ToPhase2** and

ToPhase1.5 because of men in phase 2. Since v and all men are not in phase 2 together, **Propose2** and **Confirm**2 are not enabled. But if v is in phase 2, v may be activated for **BadInit** (and then the system reaches a configuration in $\mathcal{C}^{1W}$). Concerning **ToPhase1**, since there is at least a man in phase 2, all women in phase 1.5 are eligible. Married women involved in a BP in phase 2 are also eligible for this rule. To summarize, a woman v is eligible for

- **Reset**.
- **ToPhase1** (if v is in phase 1.5 or in phase 2 with a BP).
- **BadInit** (if v in phase 2).

So, after at most n **Reset** (women), women are only eligible for **ToPhase1** or **BadInit** and men for **Reset** and **ToPhase1**. Then, after the next activation, that is altogether $O(n)$ moves, the configuration is in $\mathcal{C}^1$. Furthermore, after lemma 10 and 20, we know that the execution will reach a configuration in $\mathcal{C}^{1W}$ or in $(\{1.5\}, \{1.5\}, 0)$. □

Lemma 22. *Let C be a configuration in $(\{2\}, \{1.5, 2\}, X \geq 0)$. Any execution starting from C takes $O(n)$ moves to reach a configuration C' in $\mathcal{C}^{1W}$ or in $(\{1.5\}, \{1.5\}, 0)$.*

Proof. Let us consider first a node u in Men. Independently of phases, **Reset** can be enabled. Note that if a man is activated for **Reset**, the system reaches immediately a configuration in $\mathcal{C}^1$. Since there exists at least one woman in phase 1.5 and no node in phase 1, AllCoherentPhase(u) is `False` and then **Accept** and **Confirm** are not enabled. Since men are in phase 2, **ToPhase2** and **ToPhase1.5** are also not enabled. Concerning **ToPhase1**, it may be enabled because a woman is in phase 1.5.

In short, men are only eligible for **Reset** or **ToPhase1**. (These two rules lead to a configuration in $\mathcal{C}^1$).

Now, let us consider a node u in Women. Independently of phases, **Reset** may be enabled. Let $u.phase = 1.5$. Then u cannot be eligible for **Propose1/2**, **Confirm1/2**, **BadInit** and **ToPhase1.5** because of the phase of u. Moreover, because men in phase 2, **ToPhase2** is also not enabled. Then, u is eligible for **ToPhase1**.

Now, let $u.phase = 2$. Then, u cannot be eligible for **Propose1**, **Confirm1**, **ToPhase2** and **ToPhase1.5** (because of u's phase). In case of incoherence between proposal and marriage pointers, u is eligible for **BadInit**. But if she is activated for this rule, the system reaches a configuration in $\mathcal{C}^1$. Concerning **Propose2**, **Confirm**2 and **ToPhase1**, there are several cases:

- u is married: **Propose2** and **Confirm**2 are not enabled. However, u can be activated for **ToPhase1** if u is involved in a BP.
- u is single: u cannot be activated for **ToPhase1**, but for **Propose2** and **Confirm**2. We know that men cannot apply **Accept** or **Confirm**. But if *proposal* pointers are incoherent, a woman may propose to a man u and then confirm to u the marriage because of u's incoherent *proposal* pointer. Each woman may propose and confirm only once. Otherwise it would mean that m_1, a man better ranked for u, has been discovered after u's **Propose2** or **Confirm**2. But when u made her proposal to m, m_1 wasn't interesting for u (better marriage or incoherent

pointers). In any case, this means that m_1 has been activated for **Reset** and then should be in phase 1. That is in contradiction with the fact that all men are in phase 2 and the system is now in a configuration in $\mathcal{C}^1$. Furthermore, this new marriage between m and u can create a new BP, but we will see later in this proof that the system will reach a configuration where a node is in phase 1.

To summarize, a woman u is eligible for

- **Reset**
- **BadInit** if u is in phase 2 with incoherence between pointers but the system reaches a configuration in $\mathcal{C}^1$.
- **ToPhase1** if u is in phase 1.5 or in phase 2 with a BP.
- **Propose2** and **Confirm**2 if u is in phase 2.

So, after at most n **Reset** (women), and $n-1$ **Propose2** and **Confirm**2 (there is at most one woman in phase 1.5) moves, nodes are only eligible for rules that set their phase to 1 (**ToPhase1**, men's **Reset** and **BadInit**).

Then, after at most $O(n)$ moves, the system reaches a configuration in $\mathcal{C}^1$. After Lemmas 10 and 20, from there, a configuration in $\mathcal{C}^{1W}$ or in $(\{1.5\},\{1.5\},0)$ is reached in $O(n)$ moves.

Altogether, that is $O(n)$ moves to reach a configuration in $\mathcal{C}^{1W}$ or in $(\{1.5\},\{1.5\},0)$ from C. □

Lemma 23. *Any execution starting from a configuration C in $(\{1.5\},\{1.5,2\},X>0)$ takes $O(n)$ moves to reach a configuration C' in $\mathcal{C}^1 \bigcup (\{1.5\},\{2\},X>0)$.*

Proof. Let us consider first a node v in Men. Since there is at least one woman in phase 1.5 and one in phase 2, v can be only eligible for **Reset**. After his move, the reached configuration is in $\mathcal{C}^1$.

Now, let us consider a node v in Women. There are several cases:

- v is eligible for **Reset** if she has incoherent pointers ($\mathsf{IncoherentPointersW}(v) =$ `True`).
- If $\mathsf{BlockingPairW}(v) =$ `True` and $v.phase \in \{1.5,2\}$: v is eligible for **ToPhase1**. This rule change v' phase.
- If $\mathsf{BlockingPairW}(v) =$ `False` and $v.phase = 1.5$, she is eligible for **ToPhase2**. This rule sets 2 in $v.phase$.

Let us say first that all women in phase 1.5 (at most $n-1$) are not involved in a BP. They are enabled for **ToPhase2**. There are two cases:

- All women not involved in a BP (at most $n-1$) are activated first for **Reset**. Subsequently, women in phase 1.5 (at most $n-1$) are activated for **ToPhase2**. Thus, a configuration in $(\{1.5\},\{2\},X)$ is reached after at most $2n-2$ moves, that is $O(n)$ moves.

- If other nodes are activated in the same transition (of the previous point) that women are activated for **ToPhase2** (*i.e.* at most $n-1$ men for **Reset** and the rest of women, *i.e.* 1 here, for **ToPhase1**), a configuration in $\mathcal{C}^1$ is reached after at most $3n-2$ moves, that is $O(n)$ moves.

Now, let us say that women in phase 1.5 are involved in a BP. These women cannot shift to phase 2. As previously, women not involved in a BP (at most $n-1$) are activated first for **Reset**. And after that, in the same time, $n-1$ men are eligible for **Reset**, 1 for **ToPhase1** and $n-1$ women for **ToPhase2**. A configuration in $\mathcal{C}^1$ is reached after at most $3n-2$ moves, that is $O(n)$ moves. □

Lemma 24. *Any execution starting from a configuration C in $(\{1.5\},\{2\},X>0)$ takes $O(n)$ moves to reach a configuration C' in $\mathcal{C}^1$ or in $(\{1.5,2\},\{2\},X>0)$.*

Proof. Let us consider first a node v in MEN. Since, in $C, \forall\, w \in$ WOMEN, $w.phase = 2$, men are eligible for two rules: either **Reset** or **ToPhase2** (if BlockingPairM = `False`). Both rules change nodes' phase (to 2 or 1): if a man is activated for one of these rules, the reached configuration is either in $\mathcal{C}^1$ (one **Reset** has been executed) or in $(\{1.5,2\},\{2\},X)$ (one **ToPhase2** and no **Reset** have been executed).

Now, let us consider a node v in WOMEN. Node v can be eligible for **Reset** or, if v's BlockingPairW is `True`, **ToPhase1**, but not for both in the same time. **ToPhase1** changes nodes' phase. After at most n women's moves, the reached configuration is in $\mathcal{C}^1$ ($n-1$ **Reset** and one **ToPhase1**).

Then, after at most $3n$ moves, (n women's **Reset**, n women's **ToPhase1** and n men's **Reset** or **ToPhase2**), that is $O(n)$ moves, the configuration is either in $\mathcal{C}^1$ (if at least a man is activated for **Reset** or a woman for **ToPhase1**) or in $(\{1.5,2\},\{2\},X>0)$ (if women are only activated for **Reset** and men for **ToPhase2**). □

Lemma 25. *Any execution starting from a configuration C in $(\{1.5,2\},\{2\},X>0)$ takes $O(n)$ moves to reach a configuration C' in $\mathcal{C}^1$ or in $(\{2\},\{2\},X>0)$.*

Proof. Let us consider first a node v in MEN. If $v.phase = 1.5$, v is eligible for several rules: **Reset** and **ToPhase2** if BlockingPairM = `True`. If $v.phase = 2$, v is eligible for several rules: **Reset**, **Accept** and **Confirm**. In fact, if a woman w is proposing to v, v is eligible for **Accept** if w is the best proposal regarding his preference lists. In this case, since w cannot confirm in this configuration (at least one man is in phase 1.5), there is no new marriage. If v is eligible for **Confirm**, $w.marriage = v$ and $v.proposal = w$ *i.e.* they are already married (Married(w) = `True`). In any cases, that does not create a marriage and thereby also not a BP. Moreover, the phase of men activated for these rules is still 2. After the move of a man with one of these rules, the reached configuration is in $\mathcal{C}^1$ (at least one **Reset**) or in $(\{1.5,2\},\{2\},X)$ (only **ToPhase2**, **Accept** and **Confirm**).

Now, let us consider a node v in WOMEN. Node v is eligible for **Reset** or **BadInit** if she has incoherent pointers, but not for both. Indeed, if v is activated for **Reset**, then her guard of **BadInit** is `False` ($v.marriage$ and $v.proposal$ are set to `Null`). And if **BadInit** is applied, that means that **Reset** was not enabled and after **BadInit** is still not enabled. There are two cases for other rules: 1. if BlockingPairW(v): v is eligible for

ToPhase1, 2. otherwise, v is eligible for any rule. In any case, since there is at least one man in phase 1.5, v is not eligible for **Confirm**2 and **Propose2**.

Then, after at most n **Reset** of women, the enabled rules are men's **Reset** (at most n), men's **ToPhase2** (at most $n - X$), women's **ToPhase1** and **BadInit**. Then, either only **ToPhase2** of men are applied (at most $n - 1$) and the reached configuration is in $(\{2\}, \{2\}, X)$ or at least one of the following rules are activated: men's **Reset**, men's **ToPhase1**, women's **ToPhase1** or **BadInit**. In this case, the reached configuration is in $\mathcal{C}^1$.

In short, after $O(n)$ moves, the reached configuration is either in $(\{2\}, \{2\}, X)$ or in $\mathcal{C}^1$. □

Lemma 26. *Let C be a configuration in $(\{2\}, \{2\}, X > 0)$. Any execution starting from a configuration C takes $O(n^2)$ moves to reach a configuration C' either in $\mathcal{C}^1$ or, if no man has been activated for* **Reset** *or woman for* **BadInit** *or* ***ToPhase1****, in $(\{2\}, \{2\}, 0)$.*

Proof. Let us consider a woman w with $\neg$BlockingPairW. There are two cases:

- w is married without BP (*i.e.* Married(v) $\wedge$ $\mathcal{C}_v = \emptyset$). Only **BadInit** can be executed if $w.proposal \neq$ `Null`.
- w is single (*i.e.* $v.marriage =$ `Null`). Then, w is eligible for the **Reset**, **Propose2** and **Confirm**2.

After her **Reset** (if she needs one), she is eligible for **Propose2**. There is now also two cases. She proposes to m and we assume that w is the best proposal for m. Then m accepts the proposal and both confirm one after the other. In all cases, because of the definition of $\mathcal{C}_v$ and $\mathcal{P}_v$, m decreases his regret (either he was single and is now married or he was married and is now with a better ranked spouse). If m was involved in a BP (w_1, m), after this new marriage, the BP may be resolved. Indeed, if w has a better priority for m than w_1, there is no more BP (w_1, m). Note that the pair (w, m) was not a BP because w was single.

If each BP in C is resolved by a single woman, the number of BPs decreases and it cannot grow since men are only improving their marriage (BestProposal sets). Since there are $O(n^2)$ possible matches, there are $O(n^2)$ BPs. If all women make their proposals to each man in a BP, in at most $O(n^2)$ moves, there is no more BPs and the configuration is then in $(\{2\}, \{2\}, 0)$. If before resolving all the BPs, a married woman involved in one of them is activated (for **ToPhase1**) or a woman for **BadInit** or a man is activated for **Reset**, the system reaches a configuration in $\mathcal{C}^1$. □

Consider a subset of configurations $\mathcal{C}^2$ in $(\{2\}, \{2\}, 0)$ such that no man is eligible for **Reset** and $\forall w \in$Women, $w.proposal =$ `Null` $\vee$ ($w.marriage =$ `Null` $\wedge$ $w.proposal \neq$ `Null`).

Lemma 27. *Let C be a configuration in $(\{1.5\}, \{1.5\}, 0)$, such that in C no man is eligible for* **Reset** *and $\forall w \in$Women: $w.marriage.marriage = w \Rightarrow w.marriage \neq w.proposal$. Every segment of execution starting from C reaches a configuration C' in $\mathcal{C}^2$.*

Proof. From C, by Lemma 19, a configuration C_2 in $(\{2\},\{2\},0)$ or in $\mathcal{C}^1$ (after the first man's **Reset**) is reached.

Let us first consider that C_2 is in $\mathcal{C}^1$ after a man's **Reset**. Thus, there is configuration C_1 such that $C_1 \rightarrow C_2$ and, during this transition, a man m has been activated for **Reset**. Let us prove by contradiction that it is not possible. First, by definition of C, no man is eligible for **Reset** and there is no BP. Thus,

1. in C, $m.marriage = \texttt{Null}$ or $m.marriage \neq \texttt{Null}$ and both $m.marriage.marriage = m$ and $m.marriage \neq m.proposal$ and
2. in C_1, $m.marriage \neq \texttt{Null}$ and either $m.marriage.marriage \neq m$ or $m.marriage = m.proposal$.

Consequently, there are several cases. Let w be a woman. Let T be the transition that reaches C_1 from a configuration C_0 where:

1. $m.marriage = \texttt{Null}$. Thus, in T, one of the following actions is made: (a) $m.marriage \leftarrow w$ with $w.marriage \neq m$ or, (b) $m.marriage \leftarrow w$ and $m.proposal \leftarrow w$ or, (c) $m.marriage \leftarrow m.proposal$ with $m.proposal \neq \texttt{Null}$ after the transition;
2. $m.marriage \neq \texttt{Null}$ and $m.marriage.marriage = m$. Thus, in T, one of the following actions is made: (a) $m.marriage.marriage \leftarrow \texttt{Null}$ or, (b) $m.marriage.marriage \leftarrow m_1$ or, (c) $m.proposal \leftarrow m.marriage$.

First case, sub-case 1a, in T, $m.marriage \leftarrow w$ with $w.marriage \neq m$. The only rule that set the marriage pointer of m is **Confirm**. But in the guard of this rule, $w.marriage = m$ must be `True`. Therefore, the first case cannot happen.
Sub-case 1b, in T, $m.marriage \leftarrow w$ and $m.proposal \leftarrow w$. But no rule can set the two pointers to a value at the same time.
Sub-case 1c, in T, $m.marriage \leftarrow m.proposal$ with $m.proposal \neq \texttt{Null}$ after the transition. **Confirm** is the only rule that set $m.marriage \leftarrow m.proposal$ but also $m.proposal \leftarrow \texttt{Null}$. Thus, this transition is not possible.

Second case, sub-case 2a, in T, $m.marriage.marriage \leftarrow \texttt{Null}$. The only rule resulting in this action is the **Reset**. Thus, in C_0, w is eligible for **Reset** because $(w.marriage = w.proposal)$ (the condition $(w.marriage.proposal = w \wedge \mathsf{priority}(w.marriage,w) > \mathsf{priority}(w.marriage,w.marriage.marriage))$ cannot be `True` since $m.marriage.marriage = m$ in C_0). Since in C, this condition is not `True` for w, this two cases have been constructed performing one or several actions. The first case, as previously for men, no rule can set both pointers to the same value at the same time, or in a consecutive fashion (guard of **Confirm1/2**). Sub-case 2b, in T, $m.marriage.marriage \leftarrow m_1$. As for the previous sub-sub-case, women can assign a value to their *marriage* pointers only if all nodes are in phase 2, and thus, the configuration is already C_2. Sub-case 2c, in T, $m.proposal \leftarrow m.marriage$ with $m.marriage \neq \texttt{Null}$. The only rule that set $m.proposal$ to a value with possibly $m.marriage \neq \texttt{Null}$ is **Accept**. But, by definition of $\mathsf{BestProposal}(m)$, the value returned by this function cannot be the value of $m.marriage$ if $m.marriage \neq \texttt{Null}$.

Thus, by contradiction, C_2 is in $(\{2\},\{2\},0)$.

Now, let us prove that C_2 is in $\mathcal{C}^2$, *i.e.* in $(\{2\},\{2\},0)$ with the two following conditions: (a) no man is eligible for **Reset** and (b) $\forall\ w\ \in$ WOMEN, $w.proposal\ =$ `Null` $\vee\ (w.marriage\ =$ `Null` $\wedge\ w.proposal\ \neq$ `Null`). Case (a), since there cannot have a transition to $\mathcal{C}^1$ with a man's **Reset** before C_2, in C_1 (such that $C_1 \to C_2$), no man is eligible for **Reset**. Furthermore, men are eligible only for **ToPhase2** and women for no action until all men are in phase 2. **ToPhase2** sets only $m.proposal$ to `Null`. Thus, m are not eligible for **Reset** after the transition, in C_2 (guard of **Reset** depend on $m.marriage$). Finally, since in phase 2, no woman is eligible for any rule (until all men are also in phase 2 *i.e.* in C_2) and that the last actions of these women are **ToPhase2**, $m.proposal\ =$ `Null`.

Thus, C_2 is in $\mathcal{C}^2$. □

Lemma 28. *Let C be a configuration in $\mathcal{C}^{1W}$. Every segment of execution starting from C reaches a configuration D in $\mathcal{C}^2$.*

Proof. From C, by Lemmas 11, 14, 15 and 16, a configuration C_2 in $(\{1.5\},\{1.5\},0)$ is reached.

But since in C, at least one woman w is in phase 1, and since w is in phase 1.5 in C_2, she has been activated for **ToPhase1.5** in a transition T: $C_0 \to C_1$ such that $C \xrightarrow{*} C_0 \to C_1 \xrightarrow{*} C_2$. Furthermore, since the condition for **ToPhase1.5** is $\forall\ v\ \in \mathcal{N}(w) \cup \{w\}\ :\ v.phase\ =\ 1$, in C_0 all men are in phase 1. Thus, in the sub-execution $C_1 \xrightarrow{*} C_2$, men are activated for **ToPhase1.5** (they are in phase 1.5 in C_2). Then, C_2 is in $(\{1.5\},\{1.5\},0)$ and men have been already activated and are no more eligible for **Reset**. But, notice that women are possibly eligible for **Reset**. If a woman is eligible because of the condition $w.marriage.marriage\ =\ w \wedge\ w.marriage\ \neq\ w.proposal$, after her **Reset**, the man $w.marriage$ is eligible for **Reset**. Other conditions does not break a reciprocal marriage and, thus, men are not eligible for **Reset** afterward. Then, man $w.marriage$ shifts back to phase 1 (possibly with a woman that was in phase 2) and goes back again to phase 1.5. Thus, there is a loop between these configurations until there is no more woman eligible for **Reset** because of this condition. This can happen at most n times because women's **Reset** set pointers to `Null`. Indeed, after a **Reset**, women are in phase 1.5 and cannot set their pointers until phase 2.

Therefore, let us consider the last configuration in $(\{1.5\},\{1.5\},0)$ where $\forall\ w\ \in$ WOMEN : $w.marriage.marriage\ =\ w \Rightarrow\ w.marriage\ \neq\ w.proposal$ and no man is eligible for **Reset**. From this configuration, by Lemma 27, a configuration D in $\mathcal{C}^2$ is reached. □

Lemma 29. *Every configuration reached from a configuration C in $\mathcal{C}^2$ is in $\mathcal{C}^2$.*

Proof. Let us prove the result by induction. Let T be a transition $C \to C'$. Notice first that in this transition, no node can change its phase value: men are not eligible for **Reset**, women are not eligible for **BadInit** nor **ToPhase1** (there are no BP or incoherent phases). Other rules (men's **ToPhase1**, **ToPhase1.5** and **ToPhase2**) are not enabled because of nodes' phases.

Now, let us analyze all possible moves in T and their effect. First, since women are in phase 2, they are eligible for either **Propose2**, **Confirm**2 or **Reset**.
Let w be a woman. Since there is no BP and if w is (reciprocally) married, she is not

eligible for any rule.
If w is single, there are two cases for C: either 1. $w.proposal = \texttt{Null}$ or 2. $w.marriage = \texttt{Null} \wedge w.proposal \neq \texttt{Null}$.
In case 1, w can be eligible for **Reset** or **Propose2**. If w is activated for **Reset** in C, $w.marriage \neq \texttt{Null}$ and either the marriage is not reciprocal ($(v.marriage.marriage \neq v) \wedge (v.marriage.proposal \neq v)$) or the man has a better marriage ($v.marriage.proposal = v \wedge \mathsf{priority}(v.marriage,v) > \mathsf{priority}(v.marriage,v.marriage.marriage)$) (other conditions cannot be fulfilled since $w.proposal = \texttt{Null}$). After this move, $w.proposal = w.marriage = \texttt{Null}$. Notice that no marriage has been broken: in both cases, the man is still married or single, but his *marriage* pointer does not point to w and, thus, man's **Reset** conditions are not `True` after w's **Reset**. Thus, the man is not eligible for **Reset** in C'.
If w is not eligible for **Reset** and is activated for **Propose2** after this move, the following conditions are still fulfilled: $w.marriage = \texttt{Null} \wedge w.proposal \neq \texttt{Null}$ (see conditions of **Propose2** ($w.marriage = \texttt{Null}$) and how this rule sets $w.proposal$).
In the case 2, w is only eligible for **Confirm**2. After this move, the reached configuration is in $\mathcal{C}^2$ (**Confirm**2 contains the following action: $w.proposal \leftarrow \texttt{Null}$). Since $\mathsf{BestMarriage}(w) = w.proposal$ is in the guard of the rule, w does not create any BP. Indeed, because all pointers are coherent, the predicate checks whether w is proposing to the best choice. If not, w is not eligible for **Confirm**2, thus, does not create a BP.

Now, let us analyze men's moves. Men are eligible for **Accept** and **Confirm**. Let m be a man. Notice that both rules cannot create any BP. Indeed, a man involved in a BP can be married or not. Thus, **Accept** or **Confirm** set *proposal* or *marriage* pointers but do not create any BP. But since, by Lemma 32, men are always improving their marriage, a better one cannot create a new BP.
Furthermore, after a transition where one of these two moves has been applied, $\mathsf{IncoherentPointersM}(m)$ is still not `True`. Indeed, in C, $m.marriage = \texttt{Null}$ and, after **Accept**, this is still `True`. And if the move is **Confirm**, $m.marriage \neq \texttt{Null}$ but $m.marriage \neq m.proposal$ (effect of the rule: $m.proposal$ is set to `Null`). And $m.marriage.marriage \neq m$ is `False` since in C, $m.proposal.marriage = m$ and women cannot change their marriage when they are already married.

Thus, C' is in $\mathcal{C}^2$ and by induction, all the reachable configurations from C are also in $\mathcal{C}^2$. □

Proposition 2. *Every execution takes $O(n^4)$ moves to reach a configuration C in $(\{2\},\{2\},0)$. Moreover, every configuration reached from C is in $(\{2\},\{2\},X)$ with $X \geq 0$.*

Proof. For each set of configurations $\mathcal{C}' = (\{1.5,2\},\{1.5,2\},X)^{\times}$ with $X \geq 0$ listed below, we show how any execution starting from a configuration in $\mathcal{C}'$ reaches a configuration in $\mathcal{C}^1$ or in $(\{2\},\{2\},0)$. For doing that, we indicate the lemmas justifying the reachability from one set of configurations to another. Note that each such sub-execution takes $O(n^2)$ moves.

1. From $(\{1.5\},\{1.5\},X)$ to:
   - $\mathcal{C}^{1W}$, for $X > 0$: Lemma 20.

- $\mathcal{C}^1$ (if there is at least one man's incoherent pointer) or $(\{2\},\{2\},0)$, for $X = 0$: Lemma 19.

2. From $(\{1.5\},\{1.5,2\},X)$ to $\mathcal{C}^1$ or
   - $(\{1.5\},\{2\},X)$, for $X > 0$: Lemma 23,
   - $(\{2\},\{2\},0)$, for $X = 0$: Lemma 19.
3. From $(\{1.5\},\{2\},X)$ to $\mathcal{C}^1$ or :
   - $(\{1.5,2\},\{2\},X)$, for $X > 0$: Lemma 24,
   - $(\{2\},\{2\},0)$, for $X = 0$: Lemma 19.
4. From $(\{1.5,2\},\{2\},X)$ to: $\mathcal{C}^1$ or $(\{2\},\{2\},X)$:
   - for $X > 0$: Lemma 25,
   - for $X = 0$: Lemma 19.
5. From $(\{1.5,2\},\{1.5\},X)$ to $\mathcal{C}^{1W}$ or $(\{1.5\},\{1.5\},0)$, for $X \geq 0$: Lemma 21.
6. From $(\{1.5,2\},\{1.5,2\},X)$ to $\mathcal{C}^{1W}$ or $(\{1.5\},\{1.5\},0)$, for $X \geq 0$: Lemma 21.
7. From $(\{2\},\{1.5\},X)$ to $\mathcal{C}^{1W}$ or $(\{1.5\},\{1.5\},0)$, for $X \geq 0$: Lemma 21.
8. From $(\{2\},\{1.5,2\},X)$ to $\mathcal{C}^{1W}$ or $(\{1.5\},\{1.5\},0)$, for $X \geq 0$: Lemma 22.
9. From $(\{2\},\{2\},X)$ to $\mathcal{C}^1$ or $(\{2\},\{2\},0)$, for $X > 0$: Lemma 26.

Now, we consider a configuration C' in $\mathcal{C}^1$. By Lemma 10, any execution starting from C' takes $O(n)$ moves to reach a configuration C_1 in $\mathcal{C}^{1W}$ or in $(\{1.5\},\{1.5\},X)$ with $X \geq 0$.

If C_1 is in $(\{1.5\},\{1.5\},0)$, the case is listed above (item 1): by Lemma 19, a configuration either in $\mathcal{C}^1$ (if there is at least one man's incoherent pointer) or in $(\{2\},\{2\},0)$ is reached. Note that can lead to cycle between $(\{1.5\},\{1.5\},0)$ and $\mathcal{C}^1$ because of men's incoherent pointers. But since each man can have at most once incoherent pointers, this cycle can only last until there is no more incoherent pointers, that is n times. After that, from $(\{1.5\},\{1.5\},0)$, the reached configuration is in $(\{2\},\{2\},0)$ in $O(n)$ moves, by Lemma 19.
From $\mathcal{C}^{1W}$, by Lemma 28, a configuration C_2 in $\mathcal{C}^2$ is reached. Moreover, by Lemmas 11, 14, 15, 16 and 19, this configuration is reached in $O(n^4)$ moves.

Thus, starting from C', any execution reaches a configuration C_2 in $\mathcal{C}^2$ or in $(\{2\},\{2\},0)$. By Lemma 29, from C_2 in $\mathcal{C}^2$, every configuration is in $\mathcal{C}^2$. If C_2 is in $(\{2\},\{2\},0)$, either a configuration in $\mathcal{C}^1$ is reached or all reachable configurations are in $(\{2\},\{2\},X)$ (with $X \geq 0$). Indeed, nodes in phase 2 can only shift to phase 1. Then, either from C_2 all configurations are in $(\{2\},\{2\},X \geq 0)$ or there exists a transition (with either **ToPhase1**, **BadInit** or men's **Reset**) to $C_3 \in \mathcal{C}^1$ (in $O(n^4)$ moves, by Lemma 3). From C_3, a configuration C_4 in $\mathcal{C}^2$ is reached (see the above case where C' is in $\mathcal{C}^1$) and all reachable configurations are in $\mathcal{C}^2$.

In summary, we have listed above all possible types of configurations and shown that, in each case, a configuration C" in $(\{2\},\{2\},0)$ is reached in $O(n^4)$ moves and that every configurations reached from C" are in $(\{2\},\{2\},X)$ with $W \geq 0$.

□

III.2.2.2 - Convergence to a Terminal Configuration

Lemma 30. *Let $\mathcal{E}$ be a sub-execution such that in every configuration of $\mathcal{E}$, all nodes are in phase 2. Assume that in some transition $D_0 \to D_1$ in $\mathcal{E}$ a woman w executes a rule.*

1. *The activated rule belongs to* {**Reset**, **Propose2**, **Confirm2**}*;*
2. $\neg\mathsf{Married}(w)$ *holds in D_0;*
3. *If $w.marriage$ $=$ `Null` in D_0, then the activated rule is either* **Propose2** *or* **Confirm2***;*
4. *If $w.marriage$ $\neq$ `Null`, then the activated rule is* **Reset**.

Proof. Since w is in phase 2 in D_0 (by hypothesis), w is not enabled for any rule in {**ToPhase1.5**, **ToPhase2**, **Confirm1**, **Propose1**}. Moreover, since w remains in phase 2 in D_1, w cannot execute **ToPhase1** and **BadInit**. If it is the case, then w will be in phase 1 in D_1, which yields a contradiction. This proves point 1.

The point 2 holds according to the guard of **Reset**, **Propose2** and **Confirm2**.

Assume that w executes a rule in $D_0 \to D_1$. We consider two cases. First, if $w.marriage$ $=$ `Null` in D_0, then according to the algorithm, w is eligible for **Propose2** and **Confirm2** in D_0 but not for **Reset**, which proves the point 3.

Second, if $w.marriage$ $\neq$ `Null` in D_0, then w is not eligible for **Propose2** neither **Confirm2** according to the guard of these rules, which proves the point 4. □

Lemma 31. *Let $\mathcal{E}$ be a sub-execution such that in every configuration of $\mathcal{E}$, all nodes are in phase 2. Assume that in some transition $D_0 \to D_1$ in $\mathcal{E}$ a man m executes a rule.*

1. *The activated rules is either* **Accept** *or* **Confirm**.
2. *If* $\mathsf{AlreadyEngaged}(m)$ *holds in D_0, then the activated rule is* **Confirm**.

Proof. Assume that m executes a rule in $D_0 \to D_1$. By definition of $\mathcal{E}$, m does not execute **ToPhase1**, **ToPhase1.5**, **ToPhase2** and **Reset** during $D_0 \to D_1$.

Assume that $\mathsf{AlreadyEngaged}(m)$ holds in D_0. According to the **Accept** guard, m cannot execute **Accept** in $D_0 \to D_1$. □

Lemma 32. *Let m be in* Men. *Let $\mathcal{E}$ be a sub-execution such that in every configuration of $\mathcal{E}$, all nodes are in phase 2. Let $D_0 \to D_1$ and $F_0 \to F_1$ be two transitions corresponding to two consecutive activation by m of* **Confirm**.
We have: m executes at least one **Accept** *between D_1 and F_0.*

Proof. We have: $m.proposal = \texttt{Null}$ in D_1 and $m.proposal \neq \texttt{Null}$ in F_0 according to **Confirm**. So, m has to execute a rule that writes a non-null value in $m.proposal$ between D_1 and F_0. Since $\mathcal{E}$ is a sub-execution such that for each configuration in $\mathcal{E}$, all nodes are in phase 2, m can execute only **Accept** or **Confirm** by Lemma 31. Among these two rules, there is only one rule doing that: **Accept**. Thus, m executes this rule at least once between D_1 and F_0. This concludes the proof. □

Lemma 33. *Let m be in* Men. *Let $\mathcal{E}$ be a sub-execution such that in every configuration of $\mathcal{E}$, all nodes are in phase 2. Let $D_0 \to D_1$ and $F_0 \to F_1$ be two transitions corresponding to two consecutive activations by m of* **Confirm**.
We have: $\mathsf{priority}(m,m.marriage(D_1)) > \mathsf{priority}(m,m.marriage(F_1))$.

Proof. We prove the first point. First, let $D_0 \to D_1$ and $F_0 \to F_1$ be two transitions corresponding to two consecutive **Confirm** executed by m.

From Lemma 32, there exists at least one transition between D_1 and F_0 in which m executes **Accept**. Let $A \to B$ be the last such transition between D_1 and F_0. Thus, m only executes some **Accept** in $D_1 \xrightarrow{*} A$ (from Lemma 31 and since m does not execute any **Confirm** between D_1 and F_0). **Accept** does not write in $m.marriage$, so $m.marriage$ remains constant between D_1 and F_0. Let $m.marriage = w_1$ in D_1. Thus $m.marriage = w_1$ in A. From the definition of **Accept**, we have: $\mathsf{BestProposal}(m) \neq \texttt{Null}$ in A. Let $w_2 = \mathsf{BestProposal}(m)$ in A. According to the predicate we have: $w_2 = \mathsf{min}(\mathcal{P}_m)$ and so $\mathsf{priority}(m,w_2) < \mathsf{priority}(m,m.marriage)$ with $m.marriage = w_1$ in A. Thus $\mathsf{priority}(m,w_2) < \mathsf{priority}(m,w_1)$. Moreover, since $w_2 = \mathsf{BestProposal}(m)$ in A, then $m.proposal = w_2$ in B. Observe that m does not execute any rule between B and F_0. Thus $m.proposal = w_2$ in F_0. Since m executes **Confirm** in $F_0 \to F_1$ then $m.marriage = w_2$ in F_1.

Finally, we have: $m.marriage(D_1) = w_1$ and $m.marriage(F_1) = w_2$ and $\mathsf{priority}(m,w_2) < \mathsf{priority}(m,w_1)$ which concludes the proof. □

Corollary 1. *Let m be in* Men. *Let $\mathcal{E}$ be a sub-execution such that in every configuration of $\mathcal{E}$, all nodes are in phase 2. Man m can execute at most $n+1$* **Confirm** *in $\mathcal{E}$.*

Proof. By Lemma 31, m upgrades its marriage between two consecutive **Confirm**. There is at most n distinct possible marriages and there exists a total order among all these possibilities. □

Lemma 34. *Let w be in* Women *and let m be in* Men. *Let $\mathcal{E}$ be a sub-execution such that in every configuration of $\mathcal{E}$, all nodes are in phase 2. Let A_1 and A_2 be two configurations of $\mathcal{E}$ such that $A_1 \xrightarrow{*} A_2$ and w does not execute any rule between A_1 and A_2.*

1. *If $m \in \mathcal{C}_w$ in A_1 and $m \notin \mathcal{C}_w$ in A_2 then m executes* **Confirm** *in $A_1 \xrightarrow{*} A_2$;*
2. *If $m \notin \mathcal{C}_w$ in A_1 and $m \in \mathcal{C}_w$ in A_2 then m executes* **Confirm** *in $A_1 \xrightarrow{*} A_2$.*

Proof.

1. $m \in \mathcal{C}_w$ in A_1 implies that $\mathsf{priority}(m,w) < \mathsf{priority}(m,m.marriage(A_1))$. Furthermore, $m \notin \mathcal{C}_w$ in A_2 implies that $\mathsf{priority}(m,w) \geq \mathsf{priority}(m,m.marriage(A_2))$, since w does not change its *marriage* variable. So m changes its *marriage* value in $A_1 \xrightarrow{*} A_2$. According to Lemma 31, it can only do that executing **Confirm**.

2. $m \in \mathcal{C}_w$ in A_2 implies that $\mathsf{priority}(m,w) < \mathsf{priority}(m,m.marriage(A_2))$. Furthermore, $m \notin \mathcal{C}_w$ in A_1 implies that $\mathsf{priority}(m,w) \geq \mathsf{priority}(m,m.marriage(A_1))$, since w does not change its *marriage* variable. So m changes its *marriage* value in $A_1 \xrightarrow{*} A_2$. According to Lemma 31, it can only do that executing **Confirm**.

□

Lemma 35. *Let w be in* Women. *Let $\mathcal{E}$ be a sub-execution such that in every configuration of $\mathcal{E}$, all nodes are in phase 2. Between two consecutive executions of* **Propose2** *by w, there exists a man $m \in$ Men which executes* **Confirm**.

Proof. Let $D_0 \to D_1$ and $F_0 \to F_1$ be two transitions corresponding to two consecutive activations of **Propose2** by w. Assume that $m_2 = \mathsf{BestMarriage}(w)$ in F_0. Assume that $m_1 = \mathsf{BestMarriage}(w)$ in D_0.

There are two cases: either w does not execute any rule between D_0 and F_0or w does execute some rules. We start with the first case.

Observe that $w.proposal = m_1$ in D_1 and in F_0. Moreover according to the **Propose2** guard, $\mathsf{BestMarriage}(w) \neq w.proposal$ in F_0. Thus $\mathsf{BestMarriage}(w) \neq m_1$ in F_0 meaning that $m_1 \neq m_2$. There are now three sub-cases.

(a) $m_2 = \texttt{Null}$. Then we have: $m_1 \neq \texttt{Null}$ and $m_1 \in \mathcal{C}_w$ in D_1. Moreover, since $m_2 = \mathsf{BestMarriage}(w)$ in F_0 then $m_1 \notin \mathcal{C}_w$ in F_0. Thus, according to Lemma 34, m_1 executes **Confirm** in $D_1 \xrightarrow{*} F_0$.

(b) $m_2 \in \mathcal{C}_w$ in D_1. Thus $\mathsf{priority}(w,m_1) < \mathsf{priority}(w,m_2)$ and so $m_1 \neq \texttt{Null}$. Since $m_2 = \mathsf{BestMarriage}(w)$ in F_0 then $m_1 \notin \mathcal{C}_w$ in F_0 while $m_1 \in \mathcal{C}_w$ in D_1. Thus according to Lemma 34, m_1 executes **Confirm** in $D_1 \xrightarrow{*} F_0$.

(c) $m_2 = \texttt{Null}$ and $m_2 \notin \mathcal{C}_w$ in D_1. Since $m_2 = \mathsf{BestMarriage}(w)$ in F_0 then $m_2 \in \mathcal{C}_w$ in F_0. Thus according to Lemma 34, m_2 executes **Confirm** in $D_1 \xrightarrow{*} F_0$.

□

Lemma 36. *Let $\mathcal{E}$ be a sub-execution such that in every configuration of $\mathcal{E}$, all nodes are in phase 2. Let w be in* Women. *Let $C_0 \to C_1$, $C_2 \to C_3$, $C_4 \to C_5$ be three transitions corresponding to three consecutive rules executed by w. Then w executes* **Propose2** *once between C_0 and C_5.*

Proof. Using Lemma 30, w can only execute **Propose2**, **Reset**, and **Confirm2** in these three transitions.

Assume first that in $C_0 \to C_1$, w executes **Confirm2**. Then, in C_1, there exists a man m such that $w.marriage(C_1) = m$.

Since w does not execute any rule between C_1 and C_2, then in C_2, we have $w.marriage(C_2) = m$. Using point four of Lemma 30, w executes **Reset** in $C_2 \to C_3$. Moreover, since in C_3, $w.marriage(C_3) = \texttt{Null}$ and $w.proposal(C_3) = \texttt{Null}$, Lemma 30 implies that w executes **Propose2** in transition $C_4 \to C_5$.

Consider the second case in which w executes **Propose2** in $C_0 \to C_1$. In that case, the lemma holds.

The third case is when w executes **Reset** in $C_0 \to C_1$. In configuration C_1 we have that $w.proposal = w.marriage = \texttt{Null}$. The same holds in configuration C_2 since

w does not execute any rule between these two configurations, by hypothesis. In C_2, woman w can only execute **Propose2** or **Confirm2** by point 3 of Lemma 30. Since we also have that $w.proposal = \texttt{Null}$ in configuration C_2 then $\mathsf{Response}(w)$ does not hold in C_2 and thus w is only eligible for **Propose2**, which concludes the proof. □

Lemma 37. *Let m be in* Men. *Let $\mathcal{E}$ be a sub-execution such that in every configuration of $\mathcal{E}$, all nodes are in phase 2. Let $D_0 \to D_1$ and $F_0 \to F_1$ be two transitions corresponding to two consecutive activations by m of* **Accept**. *If m does not execute any rule between D_1 and F_0 then there exists a woman which executes a move between D_1 and F_0.*

Proof. Assume by contradiction that this is not the case. $\mathsf{BestProposal}(m) \neq \texttt{Null}$ in D_0 so there exists $w \in$ Women such that $w = \mathsf{BestProposal}(m)$ in D_0 and so $w.proposal = m$ in D_0. Thus in D_1, $\mathsf{AlreadyEngaged}(m)$ holds or otherwise w executed a rule in $D_0 \to D_1$ which yields a contradiction. By definition of **Accept**, $\mathsf{AlreadyEngaged}(m)$ does not hold in F_0. Since this predicate holds in D_1 and since m does not execute any rule between D_1 and F_0 then necessarily w executes a rule, which yields a contradiction. □

Proposition 3. *Let $\mathcal{E}$ be a sub-execution such that, in every configuration, all nodes are in phase 2. Nodes can execute at most $O(n^4)$ moves in $\mathcal{E}$.*

Proof. Let m be a man and let w be a woman. According to Lemma 31, m can only execute **Confirm** or **Accept** in $\mathcal{E}$. In the same way, according to Lemma 30, w can only execute **Reset**, **Confirm2** or **Propose2** in $\mathcal{E}$. We count the maximum number of rules that can be executed.

Confirm: according to Lemma 32, between two consecutive executions of **Confirm** by m, m updates its marriage preference. Thus a man executes $O(n)$ **Confirm**. So the number of **Confirm** in $\mathcal{E}$ is in $O(n^2)$.

Propose2: according to Lemma 35, between two consecutive executions of **Propose2** by w, there exists a man that executes **Confirm**. So, a woman executes $O(n^2)$ **Propose2**. Thus the number of **Propose2** in $\mathcal{E}$ is in $O(n^3)$.

The **Reset** and **Confirm2**: according to Lemma 36, between three consecutive executions of any rule of w, w executes at least one **Propose2**. So, a woman executes $O(n^2)$ **Reset** and $O(n^2)$ **Confirm2**. Thus the number of **Reset** in $\mathcal{E}$ is in $O(n^3)$ and the number of **Confirm2** in $\mathcal{E}$ is in $O(n^3)$.

The **Accept**: we consider all **Accept** executed by m in $\mathcal{E}$. They can be divided into three types: (i) the first **Accept**, (ii) an **Accept** such that m executed at least one rule between this **Accept** and *the* previous one and (iii) an **Accept** such that m did not execute any rule between this **Accept** and *the* previous one.
We count the number of **Accept** of each type.
First, **Accept** of type (i) appears once.
Accept of type (ii): By Lemma 31, the rule m executed between this **Accept** and the previous one is a **Confirm**. The number of **Confirm** executed by a man is in $O(n)$, thus the number of **Accept** of type (ii) executed by a man is in $O(n)$ too.
Accept of type (iii): By Lemma 37, between this **Accept** and the previous one, both executed by m, there exists a woman that has executed a rule. The number of rules executed by some woman is in $O(n^3)$ so the number of **Accept** of type (iii) executed

by men is in $O(n^3)$ too.
We finally obtain that the number of **Accept** executed by men is in $O(n^3)$ and so the number of **Accept** is in $O(n^4)$.

Then the number of rules executed in $\mathcal{E}$ is in $O(n^4)$. □

IV - Conclusion

In this chapter, we presented the first asynchronous self-stabilizing algorithm [LMB+17] for SMP. This algorithm (Alg. 1 and 2) is a distributed asynchronous self-stabilizing adaptation of the Ackermann *et al.*'s algorithm [AGM+11]. Their algorithm works with two different phases in which different nodes are eligible to perform actions. It implies that the algorithm needs some synchronization between nodes. This is the main difficulty especially since the unfair distributed daemon can chose nodes that do not make the system progress. Thus, useless moves can delay the convergence to the stable marriage. In particular, we notice that in the first phase women detect blocking pairs and solve them by local repairs (in getting married with the blocking partner). Since the system is asynchronous and distributed, $O(n^2)$ moves are needed to resolve one BP. An instance of SMP contains $O(n^2)$ BPs. Thus, we proved a time complexity of $O(n^4)$ moves and this is also an upper bound in term of rounds.

The lower bound of $\Omega(n^2)$ boolean queries [GNOR15] rises the following question: is it possible to build an algorithm with a better time complexity? This question is relevant since we show that resolving a BP locally in distributed settings is costly. In an other hand, the local detection is efficient: in one move, a node is aware of its involvement in a BP. Since applications need low time complexity, it is pertinent to search time improvement on BP's repairs. One way to solve this problem is to repair globally the system after a local BP detection. This is the subject of the next chapter.

Chapter 5

An Approach by Local Checkability and Reset

Contents

The complexity gap between the original Gale and Shapley's Algorithm (GSA) [GS62] and the distributed version using Ackermann *et al.*'s phases is quite large. There would be multiple reasons for that: distribution, asynchrony and in particular the very adversarial scheduler, self-stabilization or simply a wrong approach to the problem. The difference between $O(n^2)$ and $O(n^4)$ is big and it seems important to understand the real reasons for it. A way for attacking this issue could be to start from the optimal version, GSA, and to add step by step supplementary constraints in order to localize where the discrepancies come from. It is what we do in the present chapter. The result of our investigation is somewhat surprising: as a matter of fact, the gap disappears when considering the problem in the right way. Indeed, as we show it in the previous chapter, the cost of the local detection of "errors" (*e.g.*, BPs) is negligible, but the local resolutions of these "errors" may require a huge (and even infinite) number of actions. Thus, we now aim at resolving these errors/faults globally.

We adopt the technique introduced in [AKY90, APSVD94] under the name of local checkability and global repair. As it will be explained later, we were lead at defining a slightly different definition of local checkability [APSVD94], adapted to our purposes. It worth noting that the new solution in this chapter works under the more general and practical communication model of link register (equivalent to message passing; see the definition in the Model Chapter 3, Section V). A preliminary version of this work has been published in [BBB+18].

The chapter is organized as follows. We informally present the steps to obtain the new self-stabilizing solution to SMP in $\Theta(n^2)$ moves, in Section I. Then, in Section II, we describe the local checkability scheme and propose our definition. In Section III, the non-self-stabilizing but distributed asynchronous algorithm solving SMP, called `Async-GSA`, is designed and its correctness and local checkability property are proved. Afterward, in Section IV, we propose a reset algorithm `ResetAlg` that runs on a tree (built by a proposed algorithm `TreeAlg`). Finally, all algorithms are combined in Section V to get the self-stabilizing algorithm `CompAlg`, which solves SMP in $\Theta(n^2)$ moves. We conclude this chapter in the last Section VI.

I - Introduction

The issue of transforming a classical distributed algorithm, with a particular initialization, into a self-stabilizing one, has been studied for a very long time and different types of approaches have been proposed. These approaches focus on the notions of locality and globality and follow the same pattern. If an algorithm deviates from a correct behavior (because of transients faults or bad initialization), the incorrectness has first to be detected, and then corrected. For most problems, detection can be made locally. It appears this is also the case for the stable marriage, because an inconsistency implies either an incoherence between the variables of two neighbors or the existence of a blocking pair. Both can be detected by exchanging information only at the local level. Once an incoherence has been detected, one has to repair it, and the reparation can be either local or global. In the case of stable marriage, Knuth's cycle [Knu76] suggests that local repair does not work. That is why we considered techniques based on local detection

and global repair.

Global repair means that a node detecting an inconsistency is activated for propagating a reset wave, which sets each node in the network into an initial state in a synchronized way. Then, staring from such a re-initialized configuration a non-self-stabilizing but distributed algorithm will produce a correct result. However, things are not so simple, having in mind to get an optimal complexity and to keep the preference lists private. First, since the final algorithm has to be self-stabilizing, it must detect faults at any time, in any component of the final combination. Thus, the reset component and the distributed version of GSA have to be executed in parallel and each should stabilize in at most $O(n^2)$ moves. Notice that (to our knowledge) there are no studies on the move complexity of such algorithms in the considered model. So we have to provide it to achieve our goal.

Moreover, to obtain an efficient move complexity of the whole composition under an unfair daemon appears to be quite challenging. Such an adversarial scheduler can choose to activate in priority GSA part of the composition, when a reset is ongoing (and conversely), resulting in the multiplication of the move complexities of the two submodules. However, the advantage of the bipartite communication graph coupled with a system of priorities between the algorithm modules allow to propose a combination in which complexity is additive and not multiplicative.

Our starting point is a well known technique introduced in [AKY90, APSVD94] under the name of *local checkability and global repair*. It was proven in [APSVD94] that, if an initialized solution satisfies some specific properties, it can be transformed in a self-stabilizing solution. Although [APSVD94] assumes the message passing model, the transformation applies to the link register model. Indeed in [APSVD94] the channel capacity is 1 message and is equivalent to a register in read/write atomicity ([KK15, AKY97]). For the transformation to be correct, the non self-stabilizing algorithm has to be *locally checkable* [AKY90], *i.e.*, nodes can locally detect if a configuration is incorrect. Correct configurations are in particular those reached by an execution of the given non-self-stabilizing algorithm (here `Async-GSA`) starting from a correct configuration (here $C_{init}^{\texttt{Async-GSA}}$ - the initial configuration of `Async-GSA`). Notice however that a configuration in which there is any stable marriage is also correct, even if it cannot be reached by an execution from $C_{init}^{\texttt{Async-GSA}}$ (see Figure 5.4 for an example of a stable marriage that cannot be reached by GSA). Checking is made periodically, locally by each node, verifying the consistency of its state with the values in the shared registers written by its neighbors. Once an incorrect configuration is locally detected, a global reset is launched, setting each variable to a predefined initial (reset) value, while ensuring the required synchronization allowing to reach the initial configuration $C_{init}^{\texttt{Async-GSA}}$. Then the algorithm behaves as if it has been started from a correct configuration and reaches a terminal configuration with a stable marriage.

Note that, in Chapter 6, we also study how the proposed algorithm can be useful for obtaining self-stabilizing solutions to some variants of the stable marriage problem.

II - Local Checkability

For obtaining a self-stabilizing algorithm computing a stable marriage (or solving any other problem) in a general communication model under an unfair demon, weaker than in [APSVD94], we have to strengthen the original definition (property) of their local checkability in several ways (given in Definition 6 below). By abusing the notation, we keep the name of local checkability for this more restricted property.

To explain the property and the restrictions we add, let us start with the original definition. Basically, in [APSVD94], an algorithm `Alg` is locally checkable for a global predicate Π if: (i) Π can be defined by local predicates $\mathsf{LP}_{i,j}$ (for every directed link (i,j)) on the state of node i and the shared register value of j that can be read by i, (ii) no action of `Alg` can turn $\mathsf{LP}_{i,j}$ from being satisfied to not, and (iii) there is a configuration in the set of configurations of `Alg` satisfying Π. The conditions (i) and (ii) correspond to 1 and 3 in Def. 6.

Contrary to (iii), the condition 2 in Def. 6 is restricted to the initial configuration of `Alg`. Both conditions provide configurations to be restored by the reset. Notice however that (iii) is too weak. For example, `Alg` reaches configurations containing stable marriage satisfying Π (thus satisfied by (iii)), but it is difficult to reset the variables of every node to obtain the corresponding configuration with a stable marriage. It requires to solve the problem itself in advance. On the contrary, since `Alg` is an initialized algorithm, it is known how to set the local variables of each node to obtain it, it suffices only to ask that this configuration satisfies Π. One can chose another such configuration reachable in `Alg`, but the local states (values of the variables) of that configuration should be known in advance (for being able designing the transformer).

The conditions 4 and 5 are completely newly added restrictions in Def. 6, comparing to those in [APSVD94]. The condition 5 is introduced for dealing with an unfair daemon. This condition prevents from this adversary to retain the correction process of the reset module, by constantly privileging the actions of `Alg`. This becomes impossible with such a condition, since `Alg` is asked to be terminating.

Finally, condition 4 ensures that from any configuration satisfied by Π, even one that cannot be reached in executions of `Alg`, the problem solved by `Alg` is nevertheless solved. This condition is required since a reset is not launched in configurations satisfying Π and Π is asked to be stable. Put another way, first notice that condition 4 reduces the class of algorithms to which the reset is applicable. These algorithms must have the special property to be correct, not only when started from their initial configuration, but also from any configuration satisfying the predicate Π. Such configurations may be unreachable from the initial configuration but, as Π is stable, the executions from them may never activate the reset. Then `Alg`, on top of being correct from its initial configuration, must have the supplementary property to be also correct from all configurations satisfying Π. This condition is lacking in [APSVD94], resulting in a weaker transformation to behaviors of `Alg` from any configuration in Π, and not necessary for solving the problem solved by `Alg`.

Definition 6 (extended from [AKY90, APSVD94])**.** *[Local Checkability] Let* `Alg` *be a solution to a problem specification* Prob *and* Π *a global predicate on the configurations of* `Alg`*.* `Alg` *is locally checkable for* Π *iff the following conditions hold.*

1. *There exists a set* $\mathcal{LP}$ *of local predicates* $\mathsf{LP}_{i,j}$ *for each* i *and* j *where* $(i,\ j) \in E$ *such that*

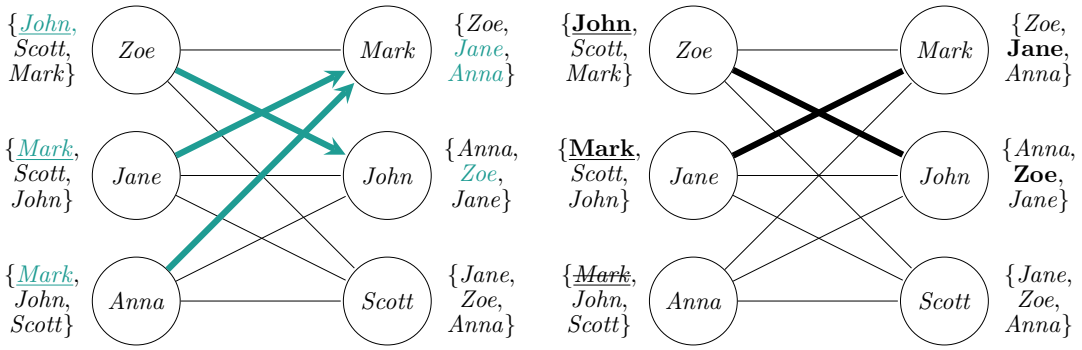
$$\Pi \ = \ \bigwedge_{\forall (i,j) \in E} \mathsf{LP}_{i,j}.$$

2. *The initial configuration of* `Alg` *satisfies* Π.
3. *Each* $\mathsf{LP}_{i,j}$ *is* stable, *that is, if* C *is a configuration satisfying* $\mathsf{LP}_{i,j}$ *and* $C \to C'$ *is a transition of* `Alg`, *then* C' *satisfies also* $\mathsf{LP}_{i,j}$.
4. *Any execution from a configuration satisfying* Π *satisfies* Prob.
5. *From any configuration,* `Alg` *terminates.*

III - Towards a Distributed Asynchronous Version of GSA

We start by presenting the non-distributed algorithm of Gale and Shapley, on which we base our distributed solution with initialization. Then, in Sub-section III.1, we present this non-self-stabilizing distributed solution - algorithm `Async-GSA`. In Sub-section III.2, we define the local predicates required to prove the local checkability of `Async-GSA`, together with its correctness and the move complexity upper bound.

Gale and Shapley's algorithm executes successive rounds. In every round, each woman proposes to her favorite man (Figure 5.1.a, proposals are represented with turquoise arrows). Doing so ensures that no blocking pair appears in the final matching (and during the whole execution). Each man who receives proposals accepts the best one (according to his preferences) and rejects all the others (Figure 5.1.b). On the figures, accepted marriages are represented with bold edges and names. A woman rejected by a man crosses out his name on her list.

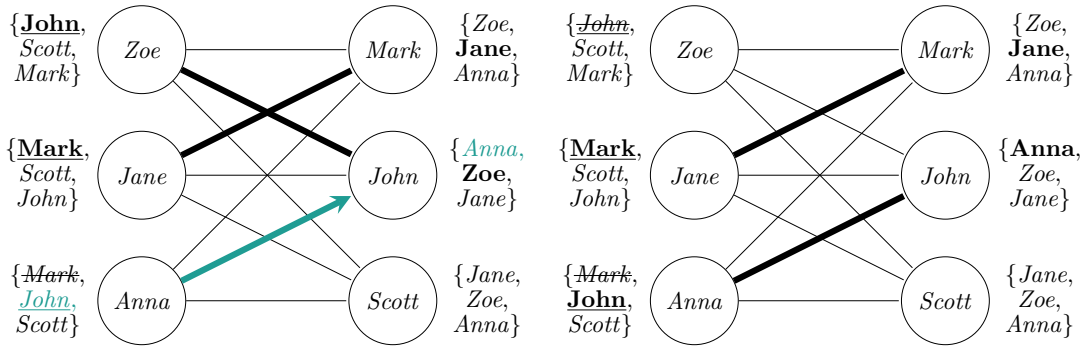


(a) Women make proposals to men (b) Configuration after men's acceptances

Figure 5.1: First round of Gale & Shapley's algorithm

Then, the first round is finished and the second starts. Single women propose to their following choice in their preference list (Figure 5.2.a). Again, men accept their

best proposals and refuse the others (Figure 5.2.b). If a proposal is better than its current match, the man rejects its partner and accepts the proposal.

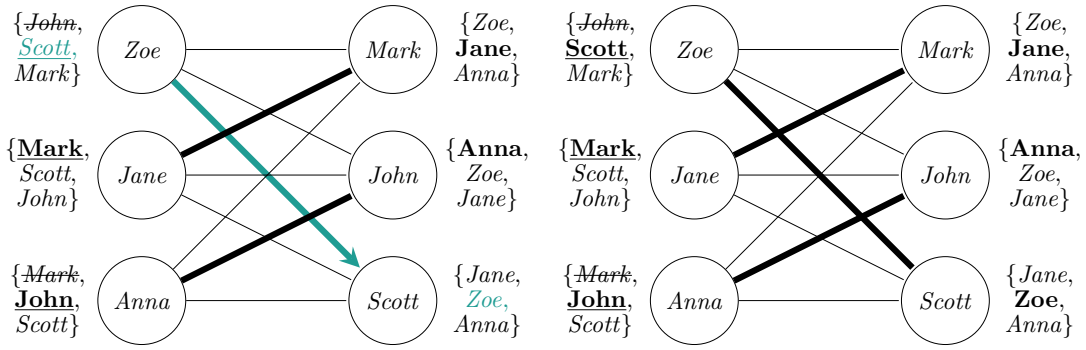


(a) The single woman proposes to its next choice

(b) The man accepts the best proposal

Figure 5.2: Second round of Gale & Shapley's algorithm

Then, the second round ends. Proposals and acceptances steps are repeated until all nodes are matched (Figure 5.3.a & 5.3.b). Notice that a married man stays married but can possibly improve (regarding his preferences) his marriage. The stable matching is obtained after at most $O(n^2)$ rounds. This method received the name of *deferred acceptance*.



(a) The single woman proposes to its next choice

(b) Final Matching

Figure 5.3: Final round of Gale & Shapley's algorithm

Note that the obtained matching is not the only stable marriage. There are others that cannot be obtained by this algorithm. See an example in Figure 5.4.

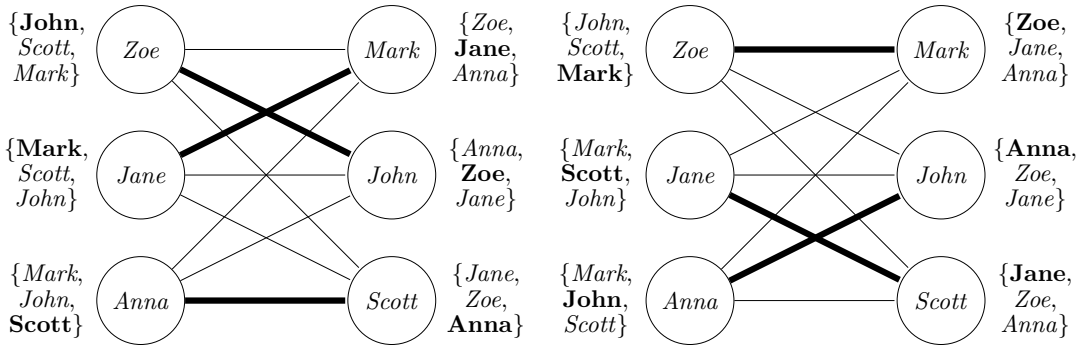


Figure 5.4: Two other stable marriages in the same system

III.1 - Distributed Asynchronous Version of GSA: `Async-GSA`

Based on the Gale and Shapley's ideas [GS62], we propose a distributed and asynchronous algorithm `Async-GSA` that solves the stable marriage problem. This algorithm is not self-stabilizing, but constitutes a module of the self-stabilizing solution composed with the scheme described in the previous section. Like GSA, `Async-GSA` works with proposals and acceptances, from an initial configuration $C_{init}^{\texttt{Async-GSA}}$ defined by variables set to a specific value (see Sub-section III.1.1).

However, the resulting algorithm works very differently from the original version, due to asynchrony. In GSA, executions proceed in synchronous rounds. In alternating rounds, women propose in a round and in the other round men answer. When men answer, they choose right away the best proposal in GSA. But in the proposed algorithm, they consider each received proposition separately at each step, not synchronously with the other received proposals in the same round as in GSA, depending on the particular schedule chosen by the adversary. Furthermore, since GSA is centralized, if a married man accepts a better proposal, the previous marriage is canceled right away and the previous spouse can propose to another man in the next round. With an asynchronous distributed scheduler, information can be delayed and the proposed solution has to take care of that.

Regarding the optimally of the solution, it has been proven in [GNOR15] that the communication complexity ([Yao79]) of the stable marriage problem is $\Omega(n^2)$ bits. This result implies an $\Omega(n^2/\log n)$ bound in moves in our model (assuming constant size communication registers). This is because in the model here each dedicated link directed from every participant in one set to another allows to not incorporate identities into messages. Communication complexity concerns the amount of information that has to be transferred only over a single bidirectional link between the two parties, called Alice and Bob. In our case, Alice holds the instance input (preference lists) of women and Bob, of men. In any algorithm functioning in such a setting it is required to include the ID of the concerned participant in a transferred message; while this information is encoded in our setting of bipartite network and thus should not be transferred. Nevertheless, the algorithm proposed here can be considered as near optimal. Moreover, we believe that there is a better lower bound of $\Omega(n^2)$.

III.1.1 - Variables, Constants, Registers and Functions (for a node v)

Variables & Constants.

- $pref$: v's constant list of its n neighbors in preference order. The priority of the element is the rank, *i.e.*, the i^{th} element has priority i. Thus, the first element is the most preferred neighbor and its priority is 1.
  As before $pref$ is a constant list and is kept secret.

- $marriage\_pref \in \mathcal{N}(v) \bigcup \texttt{Null}$: if v is a woman, the node to whom v has *proposed* (and her *spouse* if additional conditions are satisfied; see below); if v is a man, his *spouse* identifier. In $C_{init}^{\texttt{Async-GSA}}$, for men, the value of $marriage\_pref$ is `Null` and for women, the first element of v's $pref$.

Registers.
Recall that $var_{v,u}$ can be read and written by v but only read by u.

- $request_{v,u} \in \{\texttt{None}, \texttt{Proposal}, \texttt{Yes}, \texttt{No}\}$:
  - `None`: initial value of the variable (in $C_{init}^{\texttt{Async-GSA}}$).
  - `Proposal`: only for a woman v to *propose* to a man u.
  - `Yes`: used by a man to *accept* a proposal or by a woman to confirm a marriage. Two nodes v and u are said to be *married* iff $request_{v,u} = request_{u,v} = \texttt{Yes}$.
  - `No`: used by a man to *refuse* a proposal or by a woman to confirm a refusal.

Notice that in the algorithm, v may want access to a variable $var_{v,\texttt{Null}}$ or $var_{\texttt{Null},v}$. This variable does not exist and the value `None` is returned.

Functions

- $\mathsf{next}(v)$: returns the element after $marriage\_pref$ in the preference list of v and returns `Null` if $marriage\_pref$ is the last element or if $marriage\_pref = \texttt{Null}$.

- $\mathsf{priority}(v,u)$: returns the priority ($\in [1,n]$) of u in the preference list of v. Note that if u is evaluated to `Null`, $\mathsf{priority}(v,u) = n+1$.

III.1.2 - `Async-GSA`'s Algorithm Predicate

The matching $\mathcal{M}$ is defined by so-called *married* pairs $(v,u) \in E$ such that $request_{v,u} = \texttt{Yes} \wedge request_{u,v} = \texttt{Yes} \wedge marriage\_pref_v = u \wedge marriage\_pref_u = v$.
The predicate associated to `Async-GSA` is

$$\mathsf{Pred}_{\texttt{Async-GSA}} \equiv [\forall v \in V : \mathsf{Married}(v) \wedge \neg \mathsf{BlockingPair}(v)]$$

where

- $\mathsf{Married}(v) \equiv (request_{v,marriage\_pref_v} = \texttt{Yes}) \wedge (request_{marriage\_pref_v,v} = \texttt{Yes}) \wedge (marriage\_pref_{marriage\_pref_v} = v)$

- BlockingPair(v) $\equiv$ $\exists u \in \mathcal{N}(v)$: priority($v$,$marriage\_pref_v$) $>$ priority(v,u) $\wedge$ priority(u,$marriage\_pref_u$) $>$ priority(u,v).

For a node u that makes BlockingPair(v) satisfied, (v, u) is called a blocking pair.

Proposition 4. *A configuration C satisfies* PredAsync-GSA *iff C contains a stable marriage.*

Proof. Let us first prove by contradiction the direct implication: if PredAsync-GSA is `True` in C, then C contains a stable marriage.
First, since a node v can only be married with the node $marriage\_pref_v$, v cannot be married twice. Now, let v be single. In this case, $request_{v,marriage\_pref_v} = \texttt{Yes} \wedge request_{marriage\_pref_v,v} = \texttt{Yes}$ is `False` since $request_{v,\texttt{Null}} = request_{\texttt{Null},v} =$ `None`. Then, each node is married with exactly one node. Furthermore, the marriage is reciprocal. Indeed, since $request_{marriage\_pref_v,v} = \texttt{Yes}$ and the predicate is `True` for the node $marriage\_pref_v$, then $marriage\_pref_{marriage\_pref_v} = v$.
Now, by contradiction, assume that v participates to a blocking pair. So, there exist node u and v which are not married together but prefer each other to their current spouse. But PredAsync-GSA is `True`, *i.e.* BlockingPair(v) is `False` so that u and v do not prefer each other. This leads to a contradiction and thus there is no blocking pair in C.
Thus C contains a stable marriage.

Now, we prove that if a configuration C contains a stable marriage, it satisfies PredAsync-GSA. Two nodes u and v are married if $request_{v,u} = \texttt{Yes} \wedge request_{u,v} = \texttt{Yes} \wedge marriage\_pref_v = u \wedge marriage\_pref_u = v$. So, $\forall u \in V$, Married(v) is `True`. Furthermore, in a stable marriage, there is no blocking pair. Then, there is no pair (u, v) such that u prefer v to its current spouse and vice versa: the predicates BlockingPair(u) and BlockingPair(v) are false. Hence, PredAsync-GSA is `True` in C.

□

III.1.3 - Algorithm

The part of the algorithm executed by women (Algorithm 3) has 3 rules. We start by describing intuitively what those rules do.

- The rule **Propose** is executed by a woman to propose to the man in her $marriage\_pref$ pointer.
- The rule **Confirm** checks if the man $marriage\_pref$ to whom the woman has proposed, has answered positively. If he has, her register is set to `Yes`.
- The rule **Refusal_Management** is enabled if the woman's proposal has been rejected by the man $marriage\_pref$. In this case, the value `No` is set in the register and the $marriage\_pref$ pointer is set to the next man in the woman's preference list[1].

[1] If the last man refuses the proposal, this rule cannot be enabled.

Algorithm 3 `Async-GSA` for $w \in$ WOMEN:

Propose : *(* Proposes to the man pointed by marriage_pref *)*
  $\{\exists m \in \mathcal{N}(w)$: $request_{w,m} \notin \{$`Proposal`, `Yes`, `No`$\}$
  $\wedge\ marriage\_pref = m\}$
    $request_{w,m} \leftarrow$ `Proposal`

Confirm : *(* Confirms her proposal *)*
  $\{\exists m \in \mathcal{N}(w)$: $request_{w,m} =$ `Proposal`
  $\wedge\ marriage\_pref = m \wedge request_{m,w} =$ `Yes`$\}$
    $request_{w,m} \leftarrow$ `Yes`

Refusal_Management : *(* Manages a refusal *)*
  $\{\exists m \in \mathcal{N}(w)$: $request_{w,m} \in \{$`Proposal`, `Yes`$\}$
  $\wedge\ marriage\_pref = m \wedge request_{m,w} =$ `No`$\}$
    $request_{w,m} \leftarrow$ `No`
    $marriage\_pref \leftarrow \mathsf{next}(w)$

The part of the algorithm executed by men (Algorithm 4) consists of 2 rules:

- The rule **Accept** is enabled if a woman is proposing to the man and if this woman is preferred over the actual spouse of m, *i.e.*, the woman pointed by his *marriage_pref* pointer. In this case, the man sets its request variable (in the shared register) to `Yes` and updates his *marriage_pref* pointer to the identifier of the woman.
- The role of **Refuse** is the opposite of **Accept**: if a proposal is received from a less preferred woman than his actual spouse, the man sets its request variable to `No`.

Algorithm 4 `Async-GSA` for $m \in$ MEN:

Accept : *(* Accepts a proposal *)*
  $\{\exists w \in \mathcal{N}(m)$: $request_{w,m} =$ `Proposal`
  $\wedge\ \mathsf{priority}(m,w) < \mathsf{priority}(m,marriage\_pref)\}$
    $marriage\_pref \leftarrow w$
    $request_{m,w} \leftarrow$ `Yes`

Refuse : *(* Refuses a proposal *)*
  $\{\exists w \in \mathcal{N}(m)$: $request_{w,m} \in \{$`Proposal`, `Yes`$\} \wedge request_{m,w} \neq$ `No`
  $\wedge\ \mathsf{priority}(m,w) > \mathsf{priority}(m,marriage\_pref)\ \}$
    $request_{m,w} \leftarrow$ `No`

Correctness and complexity. The correctness proof and the complexity analysis of `Async-GSA` are in Section III.2, in relation with the proof of condition 4 of local checkability.

III.2 - Local Checkability of `Async-GSA`

We prove that `Async-GSA` is locally checkable (according to Def. 6), by constructing the local predicate (named $\mathsf{LP}_{m,w}$) that is checked by each man m. For that, w communicates to m whether it prefers m to its current spouse. With this information, m is able to detect a blocking pair on the edge (m, w) but also an incoherence in the variables. We prove it in Sub-section III.2.2. Notice that the exchange of information between w and m is limited and respects the privacy: preference lists are not communicated.

III.2.1 - Local Predicates

The local predicate $\mathsf{LP}_{m,w}$ must detect any deviation in the execution of `Async-GSA`. That is why it is built in relation with the guarded rules of `Async-GSA`. We use the specificity of the communication graph to define the local predicate only on the edges (m, w) where $m \in$Men and $w \in$Women: $\mathsf{LP}_{m,w}$ is checked by (the man) m on his edge (m, w). We build $\mathsf{LP}_{m,w}$ step by step. In the sequel we use the terms "to the right" (rep. "to the left") for indicating that the pointer of $marriage\_pref_w$ has been shifted towards a less (resp. more) preferred man. Note that in the clauses of $\mathsf{LP}_{m,w}$ we sometimes have a term $marriage\_pref_w = m$ or $\mathsf{priority}(w, marriage\_pref_w) < \mathsf{priority}(w, m)$. To enable man m to evaluate these terms we assume that each woman w shares the result of these comparisons (for every neighboring man m) in its shared registers. This assumption is implicit and not implemented in `Async-GSA`.

The complete local predicate checked by m is:

$$\mathsf{LP}_{m,w} \equiv (\mathbf{P}^{0}_{m,w} \vee \mathbf{P}^{Propose}_{m,w} \vee \mathbf{P}^{Accept}_{m,w} \vee \mathbf{P}^{Confirm}_{m,w} \vee \mathbf{P}^{Refuse}_{m,w} \vee \mathbf{P}^{R\_M}_{m,w}) \wedge \neg \mathbf{P}^{BP}_{m,w}$$

We describe each sub-predicate in the following.

$\mathbf{P}^{0}_{m,w}$ is a predicate satisfied locally in a configuration C where no proposal/refusal/acceptance has been made by m or w on the registers of the edge (m, w) and in all configurations reached from it, as long as no rule has been applied by m or w on the registers. In other words, it is satisfied in a configuration where $request_{w,m} = request_{m,w} = \texttt{None}$, and in all subsequent configurations as long as no rule is applied on (m, w). In these latter configurations, $request_{w,m}$ and $request_{m,w}$ have not been modified but $marriage\_pref_w$ and $marriage\_pref_m$ may have been updated by rules applied on other links ($marriage\_pref_w$ cannot be shifted to the right).

$$\mathbf{P}^{0}_{m,w} \equiv request_{w,m} = request_{m,w} = \texttt{None} \\ \wedge \mathsf{priority}(w, marriage\_pref_w) < \mathsf{priority}(w, m)$$

Notice that $\mathbf{P}^{0}_{m,w}$ is `True` in $C^{\texttt{Async-GSA}}_{init}$.

$\mathbf{P}^{Propose}_{m,w}$ is a predicate related to a situation in which a proposal has been made by a woman w. Proposals are made in a configuration satisfying $\mathbf{P}^{0}_{m,w}$ with **Propose**. This rule sets $request_{w,m}$ to the value `Proposal` and $marriage\_pref_w$ to m.

$$\mathbf{P}^{Propose}_{m,w} \equiv request_{w,m} = \texttt{Proposal} \wedge request_{m,w} = \texttt{None} \\ \wedge marriage\_pref_w = m$$

$\mathbf{P}^{Accept}_{m,w}$ is a predicate related to a situation in which a proposal has been made by w to m and m has accepted, *i.e.*, from a configuration in which $\mathbf{P}^{Propose}_{m,w}$ is `True`,

an acceptance is made by m with **Accept**. The predicate allows to check the priority of $marriage\_pref_w$: after **Accept**, either $marriage\_pref_m$ points to w or, if m has accepted a new better proposal, to a better ranked woman. Thus, the priority of $marriage\_pref_m$ is better than that of w.

$$\mathbf{P}^{Accept}_{m,w} \equiv request_{w,m} = \texttt{Proposal} \wedge request_{m,w} = \texttt{Yes} \\ \wedge \mathsf{priority}(m,w) \geq \mathsf{priority}(m,marriage\_pref_m) \wedge marriage\_pref_w = m$$

$\mathbf{P}^{Confirm}_{m,w}$ is a predicate related to a situation in which both nodes have set their request variable to `Yes`, meaning that they are married. A configuration, in which $\mathbf{P}^{Confirm}_{m,w}$ is `True` is obtained from a configuration satisfying $\mathbf{P}^{Accept}_{m,w}$ after a transition with **Confirm**. For the same reason than for $\mathbf{P}^{Accept}_{m,w}$, the priority of $marriage\_pref_m$ is checked.

$$\mathbf{P}^{Confirm}_{m,w} \equiv request_{w,m} = \texttt{Yes} \wedge request_{m,w} = \texttt{Yes} \\ \wedge \mathsf{priority}(m,w) \geq \mathsf{priority}(m,marriage\_pref_m) \wedge marriage\_pref_w = m$$

$\mathbf{P}^{Refuse}_{m,w}$ is a predicate related to a situation where m is activated for **Refuse** to refuse w's proposal (from a configuration where $\mathbf{P}^{Propose}_{m,w}$ was `True`). The rule **Refuse** is enabled in two different kinds of configuration. First if w has a priority worse than $marriage\_pref_m$, then it sets $request_{m,w}$ to `No`. Second, from a configuration satisfying $\mathbf{P}^{Confirm}_{m,w}$, if m is activated for **Refuse** after having accepted. **Refuse** is enabled if w has a worse priority than $marriage\_pref_m$. It sets $request_{m,w}$ to `No`. This case is possible if, after having accepted the proposal of w, another better ranked woman proposes to m.

$$\mathbf{P}^{Refuse}_{m,w} \equiv request_{w,m} \in \{\texttt{Proposal}, \texttt{Yes}\} \wedge request_{m,w} = \texttt{No} \\ \wedge marriage\_pref_w = m \wedge \mathsf{priority}(m,w) > \mathsf{priority}(m,marriage\_pref_m)$$

Recall that, w shares only the result of $\mathsf{priority}(w,m) < \mathsf{priority}(w,marriage\_pref_w)$ to the man m using a bit. $\mathbf{P}^{R\_M}_{m,w}$ is satisfied when w is activated for **Refusal_Management** in a configuration in which $\mathbf{P}^{Refuse}_{m,w}$ is `True`. Thus, this predicate is related to a situation in which w and m have both refused to be married together.

$$\mathbf{P}^{R\_M}_{m,w} \equiv request_{w,m} = \texttt{No} \wedge request_{m,w} = \texttt{No} \\ \wedge \mathsf{priority}(w,m) < \mathsf{priority}(w,marriage\_pref_w) \\ \wedge \mathsf{priority}(m,w) > \mathsf{priority}(m,marriage\_pref_m)$$

$\mathbf{P}^{BP}_{m,w}$ is a predicate used for detecting if the subsystem contains a blocking pair. Recall that there exists a blocking pair if both $marriage\_pref$ of w and m variables are not pointing to each other but nodes prefer each other to their actual $marriage\_pref$, for all values of $request$.

$$\mathbf{P}^{BP}_{m,w} \equiv \mathsf{priority}(w,m) < \mathsf{priority}(w,marriage\_pref_w) \\ \wedge \mathsf{priority}(m,w) < \mathsf{priority}(m,marriage\_pref_m)$$

Notice that if a woman reaches the end of her preference list with no partner, the situation is detected by this predicate. Indeed, at least one man is single and so a blocking pair is formed (since the nodes' sets are of equal size).

III.2.2 - Proof of `Async-GSA`'s Local Checkability.

Now we prove the local checkability of `Async-GSA` for

$$\Pi = \bigwedge_{\substack{\forall (m,w)\in E \ \wedge \\ m\in \textsc{Men} \ \wedge \ w\in \textsc{Women}}} \mathsf{LP}_{m,w}.$$

First, we consider the condition 3 of stability in Def. 6. The property that $\mathbf{P}^{0}_{m,w}$, $\mathbf{P}^{Propose}_{m,w}$, $\mathbf{P}^{Accept}_{m,w}$, $\mathbf{P}^{Confirm}_{m,w}$, $\mathbf{P}^{Refuse}_{m,w}$ and $\mathbf{P}^{R\_M}_{m,w}$ are stable comes directly from their construction and their relation to the transitions of `Async-GSA`. Furthermore, we prove in the following lemma that $\neg\mathbf{P}^{BP}_{m,w}$ is also stable.

Lemma 38. *The predicate* $\neg\mathbf{P}^{BP}_{m,w}$ *is stable for* `Async-GSA`.

Proof. Assume that there is an edge (w, m) in C that does not satisfy $\mathbf{P}^{BP}_{m,w}$.

If m is activated for **Accept**, $\mathsf{priority}(m,w) < \mathsf{priority}(m,marriage\_pref_m)$ is `True` in C. But, since $\mathbf{P}^{BP}_{m,w}$ is not satisfied in C, we have $\mathsf{priority}(w,m) > \mathsf{priority}(w,marriage\_pref_w)$ in C and this is still `True` in C' (**Accept** does not change $marriage\_pref_w$). Thus, $\mathbf{P}^{BP}_{m,w}$ is also `False` in C'.

The rules **Propose**, **Confirm** and **Refuse** do not change the value of $marriage\_pref$ of w and m. Thus, in C', $\mathbf{P}^{BP}_{m,w}$ is still `False`.

If w is activated for **Refusal_Management**, in C' $marriage\_pref_w$ is shifted to the next element to the right. This rule is enabled only if $request_{m,w}$ = `No`. Thus, in the previous transition m has set her variable to `No` with the **Refuse** rule, *i.e.*, $\mathsf{priority}(m,w) > \mathsf{priority}(m,marriage\_pref_m)$ is `False` in C and is still `True` in C'. Hence, $\mathbf{P}^{BP}_{m,w}$ is still `False` in C'.

□

Corollary 2. $\mathsf{LP}_{m,w}$ *is stable.*

Proof. This corollary is the direct consequence of:

1. the construction of $\mathbf{P}^{0}_{m,w}$, $\mathbf{P}^{Propose}_{m,w}$, $\mathbf{P}^{Accept}_{m,w}$, $\mathbf{P}^{Confirm}_{m,w}$, $\mathbf{P}^{Refuse}_{m,w}$ and $\mathbf{P}^{R\_M}_{m,w}$ (related to the transitions of `Async-GSA`) and,
2. the lemma 38 states that $\neg\mathbf{P}^{BP}_{m,w}$ is stable.

□

Now, we prove the condition 4 of Definition 6.

Lemma 39. *From any configuration, a woman* w *can only shift her pointer* $marriage\_pref$ *one by one to the right with the rule* **Refusal_Management**.

Proof. The only rule updating $marriage\_pref$ of w is **Refusal_Management**. This rule sets the pointer to the next element in the list after $marriage\_pref$ and if $marriage\_pref$ is the last element, to `Null`. Notice that if $marriage\_pref$ is set to `Null`, the rule **Refusal_Management** is not enabled ($\forall v \in$ V, `Null` $\notin \mathcal{N}(v)$). Thus, the pointer of the women cannot be moved.

□

Corollary 3. *From any configuration, a woman w is activated in an execution at most n times for* **Refusal_Management**.

Proof. By lemma 39, each woman can only be activated once for **Refusal_Management** for each element in her preference list. Furthermore, since preference lists have n elements, each woman can be activated at most n times for **Refusal_Management**. □

Lemma 40. *From any configuration, a woman w is eligible in an execution for at most two moves (one* **Propose** *and one* **Confirm***) if she is not activated for any* **Refusal_Management**.

Proof. The guards of these two rules contain $marriage\_pref = m$, allowing their activation on the edge (w, m) only (since w is not activated for **Refusal_Management**). Furthermore, the condition $request_{w,m} =$ `Proposal` in the guards of **Confirm**, resp. $request_{w,m} \notin \{$`Proposal`, `Yes`, `No`$\}$ in **Propose**'s guard, implies that after its activation, **Confirm**, resp. **Propose**, is no more enabled. Finally, after the activation of w for **Propose**, **Confirm** may be enabled, but after the activation for **Confirm**, **Propose** is not enabled.

Thus, **Confirm** and **Propose** are activated at most once each on the edge (w, m) if w is not activated for **Refusal_Management**. □

Corollary 4. *From any configuration, a woman w is activated in an execution for at most two moves between two activations for* **Refusal_Management**.

Proof. By Corollary 3, w can be activated twice for **Refusal_Management** and by Lemma 40, w can be activated only twice between these two **Refusal_Management**. □

Lemma 41. *From any configuration, after at most $3n - 2$ of her own moves in an execution, a woman w is no more activated for any* `Async-GSA`*'s rule.*

Proof. Let w be a woman. Assume by contradiction that there is a cycle of activations. By Corollary 3 and Corollary 4, this is not possible if **Refusal_Management** is activated (after two of her own moves, w is no more eligible). Furthermore, by Lemma 40, w can be activated for only two moves without being activated for **Refusal_Management**.

Thus, after at most $3n-2$ of her own moves, w is no more eligible for any `Async-GSA` rule. □

Lemma 42. *From any configuration, after at most $2n - 1$ of his own moves in an execution, a man m is not any more activated for any* `Async-GSA`*'s rule.*

Proof. Let us consider the $marriage\_pref$ pointer of m. It can be set only by the rule **Accept** and, since in the guard there is the priority condition ($\mathsf{priority}(m,w) < \mathsf{priority}(m,marriage\_pref)$), m can only move its pointer to the left. Thus, since there are n elements in his preference list, m can be activated at most n times for **Accept**.

Now let us consider the rule **Refuse**. Since $request_{m,w} \neq$ `No`, m cannot be activated for **Refuse** twice in a row. Furthermore, it cannot be activated in alternation with **Accept** since $\mathsf{priority}(m,w) > \mathsf{priority}(m,marriage\_pref)$ for **Refuse** and

$marriage\_pref$ can only be shifted to the left by **Accept**. That is why m can only be activated $n-1$ times for **Refuse**.

Thus, after at most $2n-1$ of his own moves, m cannot be activated any more for any rule of `Async-GSA`. □

Corollary 5. *From any configuration, after $O(n^2)$ moves in an execution, no rule is enabled, i.e. the configuration is terminal.*

Proof. The corollary is a direct consequence of Lemmas 41 and 42. □

Recall that

$$\mathsf{PredAsync\text{-}GSA} \equiv [\forall\ v\ \in\ V:\ \mathsf{Married}(v)\ \wedge\ \neg\mathsf{BlockingPair}(v)]$$

where

- $\mathsf{Married}(v) \equiv (request_{v,marriage\_pref_v} = \texttt{Yes}) \wedge (request_{marriage\_pref_v,v} = \texttt{Yes}) \wedge (marriage\_pref_{marriage\_pref_v} = v)$
- $\mathsf{BlockingPair}(v) \equiv \exists\ u\ \in \mathcal{N}(v)$: $\mathsf{priority}(v,marriage\_pref_v) > \mathsf{priority}(v,u) \wedge \mathsf{priority}(u,marriage\_pref_u) > \mathsf{priority}(u,v)$.

Lemma 43. *A terminal configuration of* `Async-GSA` *satisfying* Π*, satisfies the algorithm predicate* PredAsync-GSA.

Proof. Let C be a terminal configuration satisfying Π. In C, for all edges (m,w), $\mathbf{P}^{Propose}_{m,w}$, $\mathbf{P}^{Accept}_{m,w}$ and $\mathbf{P}^{Refuse}_{m,w}$ are `False` (otherwise a rule would be enabled but C is terminal). Thus, since Π is satisfied, $\forall (m,w) \in E : (\mathbf{P}^{Confirm}_{m,w} \vee \mathbf{P}^{R\_M}_{m,w} \vee \mathbf{P}^{0}_{m,w}) \wedge \neg\mathbf{P}^{BP}_{m,w}$ is necessarily `True`.

But, since $\mathbf{P}^{BP}_{m,w}$ is false on all edges (m,w), $\mathsf{BlockingPair}(v)$ is `False` for all v. This implies that no node has $marriage\_pref = \texttt{Null}$. Notice also that if $\mathbf{P}^{Confirm}_{m,w}$ is satisfied on an edge, we have $\mathsf{priority}(m,w) = \mathsf{priority}(m,marriage\_pref_m)$ (and not $\mathsf{priority}(m,w) \geq \mathsf{priority}(m,marriage\_pref_m)$, otherwise m would be eligible for **Refuse**). Hence, each man and woman has one and only one incident edge that satisfies $\mathbf{P}^{Confirm}_{m,w}$, *i.e.*, each node satisfy Married.

Thus, a terminal configuration satisfying Π satisfies also the algorithm predicate PredAsync-GSA of `Async-GSA`. □

Recall that the problem specification Prob defining SMP (defined in Chap. 3, Sec. VI) is satisfied by an execution iff

a) the execution reaches a *terminal* configuration (*i.e.*, a configuration in which no node is eligible), and

b) this configuration contains a stable marriage.

Furthermore, notice that this following lemma proves also the correctness of `Async-GSA`.

Lemma 44. *The point 4 of Definition 6 is satisfied, i.e. from any configuration C_0 satisfying* Π*, any execution of* `Async-GSA` *satisfies* Prob.

Proof. By Corollary 5, from any configuration satisfying Π, a terminal configuration C_1 is reached. Furthermore from Lemma 38 and the fact that $\mathsf{LP}_{m,w}$ is stable, C_1 satisfies Π. Finally, by Lemma 43, C_1 satisfies the algorithm predicate $\mathsf{PredAsync\text{-}GSA}$, *i.e.*, contains a stable marriage by Proposition 4.

Thus, from any configuration satisfying Π, any execution of `Async-GSA` satisfies Prob. □

Now we can prove that `Async-GSA` is locally checkable. Recall that the initial configuration $C_{init}^{\texttt{Async-GSA}}$ is a configuration where $\forall (m,w) \in E,\ request_{w,m} = request_{m,w} = \texttt{None} \wedge marriage\_pref_m = \texttt{Null} \wedge \mathsf{priority}(w, marriage\_pref_w) = 1$.

Theorem 2. `Async-GSA` *is locally checkable for* Π.

Proof. There are five conditions in Def. 6.
First, the condition 1 is satisfied by definition of Π and $\mathsf{LP}_{m,w}$.
Second, $C_{init}^{\texttt{Async-GSA}}$ satisfies the condition 2 ($\mathbf{P}^0_{m,w}$ is satisfied for all (m,w)).
Third, by Corollary 2, the condition 3 is satisfied.
Fourth, by Lemma 44, the condition 4 is satisfied.
Finally, by Lemmas 41 and 42, the condition 5 is satisfied. □

So, a man can detect, using the local detection detailed in [APSVD94], whether or not its `Async-GSA` state satisfies $\mathsf{LP}_{m,w}$. If not (*cf.* Section IV), it can take some actions. The global composition is presented in Section V.

III.3 - Time Complexity

`Async-GSA`'s Time Complexity. Notice that Corollary 5 proves the worst case complexity of the `Async-GSA` module: $O(n^2)$ moves. This induces also a complexity of $O(n^2)$ rounds. Indeed, the definition of a round (see Def. 4) captures the execution rate of the slowest processor in any computation. Since there is at least one move in each round, an upper bound for the move complexity is an upper bound for the round complexity. Thus, the final configuration is reached in at most $O(n^2)$ rounds.
In the following, we illustrate an execution scenario that reaches this bound, proving that our round and move complexity is tight. Consider a system with n women and n men denoted by $w_1, w_2, \ldots, w_n$ and $m_1, m_2, \ldots, m_n$ (Figure 5.5). The preference list of w_x for $x > 1$ is: $[m_x, \ldots, m_n, m_2, \ldots, m_{x-1}, m_1]$ and w_1 has the same list as w_2. The preference list of m_x is: $[w_{x+1}, \ldots, w_1, w_n, \ldots, w_{x+2}]$, with $x+1 = 1$ and $x+2 = 2$ if $x = n$ and $x+2 = 1$ if $x = n-1$.

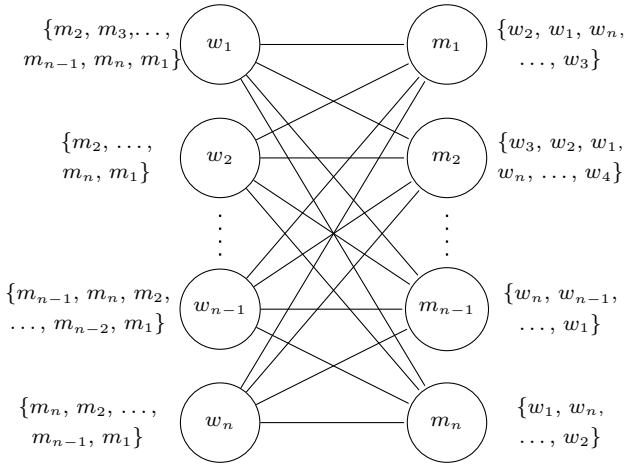
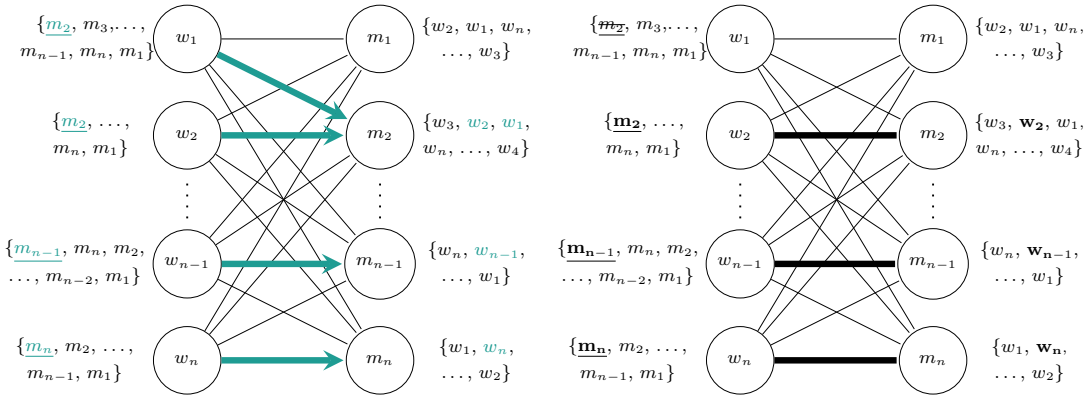


Figure 5.5: System before running `Async-GSA`

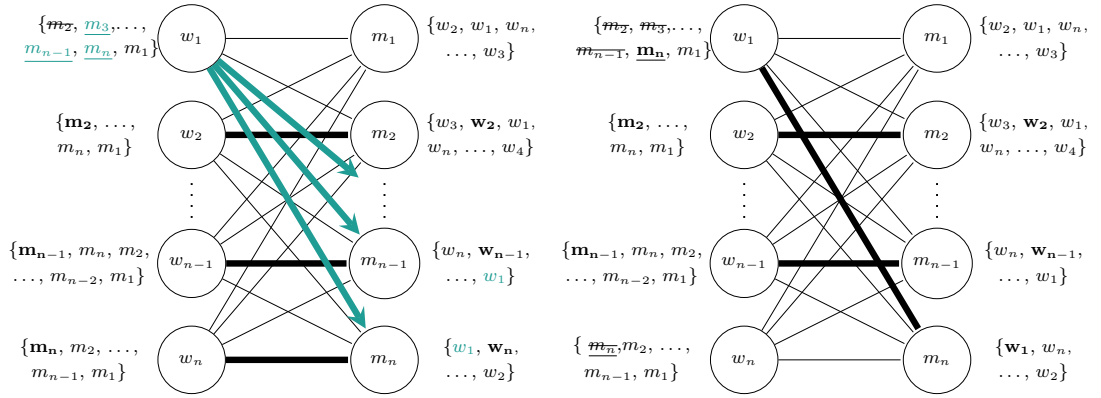
In the first round each w_x proposes to m_x except w_1 that proposes to m_2 (Figure 5.6.a). Thus, m_2 receives two proposals, accepts in the second round w_2's proposal and refuses that from m_1. The other men accept the proposal (Figure 5.6.b).



(a) Women's proposals (b) After the acceptances

Figure 5.6: First two rounds of `Async-GSA`

In the $(2 \cdot (n-3))$ following rounds w_1 proposes to $m_3, \ldots, m_{n-1}$ but is refused each time (Figure 5.7.a). Hence, in the $(2n-3)^{th}$ round, w_1 proposes to m_n and is accepted in the next round while w_n is refused (Figure 5.7.b).



(a) w_1's $n-2$ next proposals (b) w_1 matching

Figure 5.7: Next $(2n-4)^{th}$ rounds, w_1'proposals before finding its partner

Now, w_n is single and proposes to $m_2, \ldots, m_{n-1}$ until being accepted by m_{n-1}. So, after $2 \cdot (n-2)$ rounds, w_n is married to m_{n-1} and w_{n-1} is single. Thus, each woman makes proposals with the same pattern in $2 \cdot (n-2)$ rounds and the last woman w_2 ends up (after its own $2 \cdot (n-2)$ rounds) by proposing to m_1 (Figure 5.8). This leads to a final complexity of $\Theta(n^2)$ rounds.

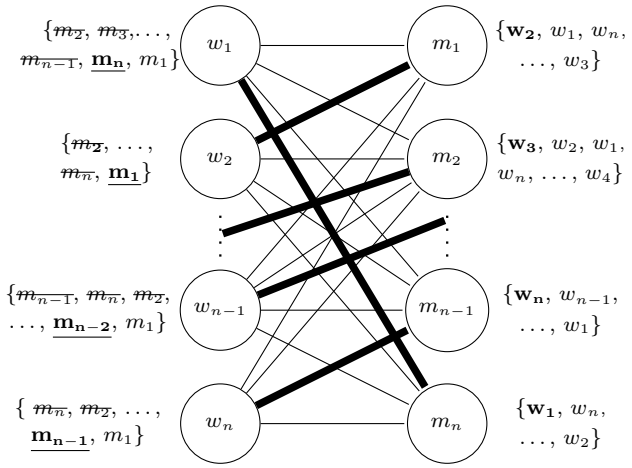


Figure 5.8: Final matching

IV - Reset

We propose a self-stabilizing distributed and asynchronous reset algorithm (`ResetAlg`) that, upon request of a node, resets globally a requesting algorithm to a specific configuration. More precisely, this is ensured only if `ResetAlg` itself is already stabilized to a "correct" configurations. Being self-stabilizing, this algorithm reaches such configurations eventually. We prove that these, as well as the following restored configuration after a reset request, are reached in $O(n)$ moves each. In a similar way to PIF (Propagation of Information with Feedback) algorithms, `ResetAlg` works with waves propagating

up and down a rooted tree values and actions to execute, in a way coordinated (synchronized) by the root. When a node requests a reset, a wave goes up to inform the root. Then, a freeze wave goes down to the leaves to freeze the nodes for the requesting algorithm (such that they are not eligible for it and also for requesting the reset again, as long as the ongoing reset operation is accomplished). Once the leaves are reached, nodes are reset by a wave going up to the root. Finally, the last wave releases the nodes, which have been all reset, and the specific reset configuration reached.

Although the general ideas described above, concerning the `ResetAlg` functioning, apply to any spanning tree, it is necessary to consider a specific tree to obtain such a low complexity, the issue coming from the adversarial unfair scheduler. Indeed, it has the capacity of initiating recursively resets in size increasing sub-trees, terminating none of them but the last one. Thus, the $O(n)$ move complexity is obtained over a bipartite communication graph justified by the definition of SMP. Nevertheless, we present a general complexity analysis of `ResetAlg`, for any given underlying tree.

The section is organized as follows. In Sub-section IV.1, we present and prove a self-stabilizing spanning tree construction algorithm `TreeAlg` (Algorithm 5) that builds in $O(n)$ moves a rooted tree of depth 2 on a bipartite graph $K_{n,n}$. Then, the reset algorithm `ResetAlg` is presented in Sub-section IV.2. This algorithm is proved to satisfy the specification of the self-stabilizing reset problem (Definition 7) in $O(n)$ moves on a tree of depth 2.

IV.1 - Tree Algorithm `TreeAlg`

Since a node has a complete preference list, it has the identifiers of the nodes in the opposite set. That allows to build a double fan-shaped tree. The root of the tree is the woman with the minimum identifier, W_{min}. All men are children of W_{min}: this is the first fan. The second fan is composed of the other women that are children of the man with the minimum identifier: M_{min}. Thus the depth of the tree is 2. Since each node holds the identifiers of the other subset in its preference list, W_{min} may learn that she is the root once all men provide her with the correct information regarding the minimality of her identifier. Unfortunately, due to a bad initialization, this information may be incorrect (corrupted). Indeed the construction being an element in the final composition has to be self-stabilizing. That is why we do not consider that the spanning tree is built from the start.

The following figure represents the constructed tree on a bipartite graph $K_{3,3}$. W_{min} is w_1 and M_{min} is m_1. On the left, the tree is represented on the bipartite graph (bold edges are the links in the tree). On the right, the tree is displayed such that the double-fan construction is visible.

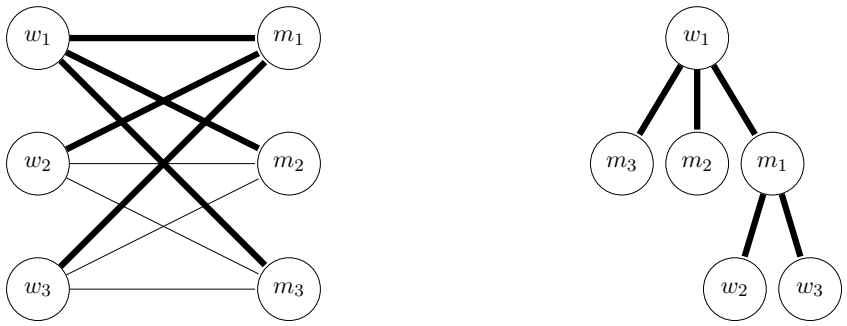


Figure 5.9: Example of a tree on a graph $K_{3,3}$

IV.1.1 - Variables, Constants, Registers and Functions (for a node v)

Variables & Constants.

- *parent*: the identifier of the parent of v. If $v = W_{min}$, eventually $parent_v = v$,
- $pref$: list of n neighbors in preference order (see the variables of the previous algorithm for more details).

Registers.

- $min_{v,u} \in \{\texttt{True}, \texttt{False}\}$: `True` represents the fact that u is the minimum in the $pref_v$, `False` otherwise.

Functions.

- $\mathsf{min}(list) \rightarrow$ identifier: returns the minimum identifier in the list *list*.

IV.1.2 - Tree Algorithm Predicate

The algorithm builds a spanning tree rooted in W_{min}. The eventually constructed tree is encoded in the local variables *parent*. The root of the tree w has $parent_w = w$. All women and men are leaves, except M_{min} and W_{min}. M_{min} is on the paths between W_{min} and the other women.

The predicate of the rooted tree construction is:

$$\begin{aligned}\mathsf{PredT} \equiv\ \forall\, v \in V, \forall u \in \mathcal{N}(v):\ & min_{v,u} = (u = \mathsf{min}(pref_v)) \\ & \wedge [(parent_v = \mathsf{min}(pref_v) \wedge \neg min_{u,v}) && ① \\ & \quad \vee (parent_v = \mathsf{min}(pref_v) \wedge min_{u,v} \wedge v \in \textsc{Men}) && ② \\ & \quad \vee (parent_v = v \wedge min_{u,v} \wedge v \in \textsc{Women})] && ③\end{aligned}$$

Notice that this definition implies that there is exactly one woman satisfying ③ $parent_v = v \wedge min_{u,v}$, *i.e.*, exactly one W_{min}. Thus, the other women satisfy ① $parent_v = \mathsf{min}(pref_v) \wedge \neg min_{u,v}$ and are children of M_{min}. M_{min} satisfies ② $parent_v = \mathsf{min}(pref_v) \wedge min_{u,v}$. Similarly, there is exactly one M_{min}. All other men are children of W_{min} and satisfy ①. We define the legitimate configurations for the tree algorithm as the configurations satisfying PredT.

IV.1.3 - Algorithm

The self-stabilizing tree algorithm is composed of three rules:

1. The rule **I_am_not_root** may be enabled for all men and all women except the root. It sets the local variable *parent* to the minimum identifier of the other set.
2. The rule **I_am_root** may be enabled for v if $v \in$WOMEN and if all v's neighbors u have their communication variable $min_{u,v}$ set to `True`. In this case, v knows that she is the root of the tree, *i.e.*, W_{min}, and sets her parent pointer to her own identifier.
3. The rule **Update** updates the link register variables of the adjacent links so that the neighbors could learn whether or not they have the minimum identifier in their set.

Algorithm 5 Tree construction `TreeAlg` for $v \in V$

1: **I_am_not_root** : *(* v is not root *)*
2: $\{parent \neq \mathsf{min}(pref) \wedge [(\exists u \in \mathcal{N}(v):\ \neg min_{u,v}) \ \vee\ v \in$MEN$]\}$
3: $parent \leftarrow \mathsf{min}(pref)$
4:
5: **I_am_root** : *(* v is W_{min} *)*
6: $\{v \in$WOMEN $\wedge\ parent \neq v\ \wedge\ (\forall u \in \mathcal{N}(v):\ min_{u,v})\}$
7: $parent \leftarrow v$
8:
9: **Update** : *(* Updates $min_{v,u}$ *)*
10: $\{\exists u \in \mathcal{N}(v):\ min_{v,u} \neq (\mathsf{min}(pref) = u)\}$
11: $\forall u \in \mathcal{N}(v):\ min_{v,u} \leftarrow (\mathsf{min}(pref) = u)$

Using the constructed tree, the following function computes the children set of a node. This function can be used by any algorithm running on the tree.

- $\mathsf{children}(v)$: returns the set of identifiers: returns the v's set of children' identifiers.

```
if(∀u ∈N(v): min_{u,v} = True)
        return N(v)−parent
else
        return ∅
```

IV.1.4 - Correctness and Complexity Analysis of `TreeAlg` (Algorithm 5)

Lemma 45. *In an execution from any configuration C, each man m is activated at most once for* **I_am_not_root** *and at most once for* **Update**.

Proof. A man is eligible only for two rules: **Update** and **I_am_not_root**. Indeed, **I_am_root** is eligible only for women. If m is activated for **I_am_not_root**, *parent* takes the value $\mathsf{min}(pref)$. Notice that **Update** does not change *parent*. Thus, **I_am_not_root** cannot be enabled once again.
A similar reasoning holds for **Update**. □

Lemma 46. *In an execution from any configuration C, each woman w is activated at most once for each rule (***I_am_not_root***,* **I_am_root** *and* **Update***).*

Proof. First the condition of activation of **Update** is the existence of a variable $min_{u,v}$ (for $u \in \mathcal{N}(v)$) that is not equal to $(\mathsf{min}(pref) = u)$. Second **Update** is the only rule setting min. Hence, after one activation of **Update**, no register satisfies the guard again.

Now, notice that a woman cannot be activated twice in a row for **I_am_not_root** because of the conditions on the variable *parent* (the same argument holds for **I_am_root**). Assume that this rules are activated by w more than once but by alternation. Let **I_am_not_root** be the first activated rule. Thus, before this activation, $\exists u_1 \in \mathcal{N}(w)$: $\neg min_{u_1,w}$ but when **I_am_root** is activated, we know that $\forall u \in \mathcal{N}(w)$: $min_{u,w}$. Furthermore, between the two activations, at least u_1 has been activated for **Update**. Now assume that w is activated again for **I_am_not_root**. This induces that a man u_2 (not u_1, by Lemma 45) has changed its $min_{u_2,w}$. But, since u_1 has set $min_{u_1,v}$ to 1 and all men have the same preference list, u_2 cannot be activated for **Update** to set the value of $min_{u_2,w}$ to 0. Thus, w cannot be activated again for **I_am_not_root**. The same contradiction can be found if w is activated first for **I_am_root** and then for **I_am_not_root**.

□

Recall that configurations satisfying PredT are legitimate configurations for `TreeAlg` and that:

$$\begin{aligned}\mathsf{PredT} \equiv\ \forall\, v \in V, \forall u \in \mathcal{N}(v):\ & min_{v,u} = (u = \mathsf{min}(pref_v)) \\ & \wedge [(parent_v = \mathsf{min}(pref_v) \wedge \neg min_{u,v}) \\ & \quad \vee (parent_v = \mathsf{min}(pref_v) \wedge min_{u,v} \wedge v \in \textsc{Men}) \\ & \quad \vee (parent_v = v \wedge min_{u,v} \wedge v \in \textsc{Women})]\end{aligned}$$

Lemma 47. *Any terminal configuration satisfies* PredT*, i.e. it is legitimate.*

Proof. In a terminal configuration C, $\forall\, v \in V, \forall u \in \mathcal{N}(v)$: $min_{v,u} = (u = \mathsf{min}(pref_v))$ is necessarily `True`, otherwise **Update** would be enabled.

Now, assume that C does not satisfy PredT. This implies that at least one edge (v, u) does not satisfies one of these following conditions:

- $parent_v = \mathsf{min}(ref_v) \wedge \neg min_{u,v}$
- $parent_v = \mathsf{min}(pref_v) \wedge min_{u,v} \wedge v \in \textsc{Men}$
- $parent_v = v \wedge min_{u,v} \wedge v \in \textsc{Women}$

Thus (v, u) satisfies at least one of these following conditions:

1. $parent_v = v \wedge \neg min_{u,v}$
2. $parent_v = \mathsf{min}(pref_v) \wedge min_{u,v} \wedge v \in \textsc{Women}$
3. $parent_v = v \wedge min_{u,v} \wedge v \in \textsc{Men}$

Case 1 is not possible in a terminal configuration: v would be eligible for **I_am_not_root**.

Case 2 is only for women. There are two sub-cases: either all neighbors of v have their *min* set to 1 or there is at least one neighbor's *min* variable set to 0. In both sub-cases, a node is eligible: either v is eligible for **I_am_root** or a neighbor is eligible for **Update** (u or an other man).

Finally, in case 3, v is eligible for **I_am_not_root**.

Hence, a terminal configuration satisfies PredT. □

Theorem 3. *From any configuration C, after $O(n)$ moves, a terminal and legitimate configuration is reached.*

Proof. By Lemmas 45 and 46, each nodes is eligible at most once for each rule. Thus, after $O(n)$ moves, a terminal configuration is reached.

Finally, by Lemma 47, this terminal configuration is legitimate. □

IV.2 - Reset Algorithm `ResetAlg`

In this section, we propose a technique for resetting (under some conditions) a given (so called) *basic algorithm* `BasicAlg` into a configuration $C_{reset}^{\texttt{BasicAlg}}$. The technique uses the reset algorithm `ResetAlg` (Algorithm 6). To request a reset, a *reset signal* is *generated/launched* (at any node) by `BasicAlg` (or some other module controlling the execution of `BasicAlg`). This triggers the local module of `ResetAlg`. Launching a signal represents the writing of `True` by `BasicAlg` in a boolean variable shared with `ResetAlg` (called here *signal*). The reception of the signal is the reading of `True` in this boolean variable.

Being self-stabilizing, `ResetAlg` can be started in an arbitrary configuration, but is designed to reach a set of "good/correct" configurations (see $\mathcal{C}_{safe}^{\texttt{ResetAlg}}$ set defined below). From any such configuration, if finitely many signals are launched by the basic algorithm (and at least one), then the basic algorithm eventually reaches a configuration in $C_{reset}^{\texttt{BasicAlg}}$.

Remark 3. *As it is stated in* [KA98]*, whenever arbitrary states can be reached in the presence of faults, it is impossible to ensure (to design a resetting algorithm such) that every "resetting operation" (invoked by the reset signal in our case) is correct, i.e., necessary results in the predefined resetting configuration ($C_{reset}^{\texttt{BasicAlg}}$). This is because "the faults may perturb the resetting module (the reset algorithm) to a configuration where the reset operation has completed prematurely". Furthermore, notice that all known self-stabilizing reset algorithms (*[GM91, KA98, Var93, APSVD94, AO94, AH93, DH95]*) have a similar specification.*

The precise specification of the reset problem that we solve is given below.

Recall that, for a given sub-algorithm `Alg`, we denote by $C^{\texttt{Alg}}$ the projection of a configuration C to the variables of `Alg`.

We assume that there is a non-empty set $\mathcal{C}_{normal}^{\texttt{ResetAlg}}$ of terminal configurations contained in the set of "good" configurations $\mathcal{C}_{safe}^{\texttt{ResetAlg}}$ and where each node is in a *normal* status (*i.e.*, executing no reset actions locally) - see Def. 8.

We define a configuration C in $\mathcal{C}_{safe}^{\texttt{ResetAlg}}$ satisfying the following conditions: 1. nodes are consistent (see Sub-section IV.2.1.2 for the definition), 2. either the status of any node is normal and it has no signal or, the status is *reset* (just executed a reset locally) and, 3. any node with the status normal has only parents with status normal and nodes with status reset have children with only status reset. The formal definition is given in Definition 9.
We also assume that a configuration $C_{reset}^{\texttt{BasicAlg}}$ of `BasicAlg` has been precisely defined.

We give the definition of the problem under the form of conditions on executions.

Definition 7 (Specification of the Self-stabilizing Reset Problem)**.**

1. *(Convergence) Starting from an arbitrary configuration, eventually a configuration in $\mathcal{C}_{safe}^{\texttt{ResetAlg}}$ is reached.*
2. *(Termination) If a finite number of signals are launched, a configuration in $\mathcal{C}_{normal}^{\texttt{ResetAlg}}$ is reached.*
3. *(Reset) Starting from any configuration in $\mathcal{C}_{safe}^{\texttt{ResetAlg}}$, if a finite number of signals and at least one are launched, $C_{reset}^{\texttt{BasicAlg}}$ is reached.*

Remark 4. *Notice that $\mathcal{C}_{safe}^{\texttt{ResetAlg}}$ contains $\mathcal{C}_{normal}^{\texttt{ResetAlg}}$ in the current self-stabilizing implementation of* `ResetAlg`*. We later prove (Proposition 5) that it contains only one configuration $C_{normal}^{\texttt{ResetAlg}}$ (Def. 8) and it is terminal (if reset signals, and faults, cease).*

IV.2.1 - `ResetAlg`

In `ResetAlg`, nodes communicate over a rooted tree. Although we use only the double-fan tree of depth 2 for solving SMP, we present a reset algorithm for any rooted spanning tree. Each node has a variable *status*. `BasicAlg` *launches* or *generates* a signal by setting a specific boolean variable (*signal*) to `True`. When receiving a reset signal, a node, only if its *status* is set to *normal*, changes for *initiate* to inform its parent that a reset is in progress, and so on up to the root. Then, successive waves are initialized by the root. First a freeze wave (changing *status* to $freeze$) goes down to the leaves. During this wave, nodes are inhibited, *i.e.* they are not eligible for any rule and wait for the next wave. The second wave goes from the leaves to the root: nodes are activated to reset the requested values (through a function `reset_BasicAlg_variables()` provided by `BasicAlg`) and switch to *reset* status to inform their parents. Once the wave reaches the root, all nodes have been activated for the reset and $C_{reset}^{\texttt{BasicAlg}}$ configuration is reached (if the initial reset signal was launched in $\mathcal{C}_{safe}^{\texttt{ResetAlg}}$). Nodes are now ready to return to the normal status: the third wave (*normal* status) is initiated by the root. When this wave ends, all nodes are in the *normal* status and do not change until a possible next reset request.

Notice that during this process, when $status \neq normal$, the rules of `BasicAlg` are not enabled, neither request reset signals are accepted. This is important to allow reaching $C_{reset}^{\texttt{BasicAlg}}$. Indeed, if `BasicAlg` could be executed when a node has its *status* variable set to *reset* (*i.e.* just after the execution of the reset function), `BasicAlg`'s variables could be modified before all nodes were activated for the function (because of the asynchronous unfair demon). In this case, $C_{reset}^{\texttt{BasicAlg}}$ would never be reached. Thus,

`BasicAlg`'s rules can be executed at a node only when this node has its *status* variable set to *normal*.

IV.2.1.1 - Variables, Registers, Predicates, Functions and Procedures (for a node v)

Variables.

- *status*: in $\{normal, initiate, freeze, reset\}$.
  - *normal* indicates that the node is not aware of any ongoing reset, *i.e.* the node may be eligible for `BasicAlg`.
  - *initiate* is used to inform the parent that a node wants to *launch a reset*. This information goes from the initiator of the reset to the root.
  - $freeze$ is used by the root to freeze the nodes down to the leaves.
  - *reset* is used to inform the parent (and finally the root) that descendants of the node v have already performed the reset.

  A node with $status = normal$ is called a *normal*-node. Similarly, we derive a $freeze$-node, a *reset*-node and an *initiate*-node.

- *parent*: written by the spanning tree algorithm and only read by the reset algorithm.

- *signal*: boolean written by the environment (`BasicAlg` or other external module) to request a reset (if `True`). We say that a signal is *generated/launched* when this variable is set to `True`.

Registers.

- $st_{v,u} \in \{normal, initiate, freeze, reset\}$: aiming to have the copy of the v's *status* value in the shared register read by u.

Predicates.
The three following predicates are used by the node v to distinguish between root, internal node or leaf :

- $\mathsf{I\_am\_Root} \equiv parent_v = v$.
- $\mathsf{I\_am\_Leaf} \equiv \mathsf{children}(v) = \emptyset$ (see the next section for the definition of $\mathsf{children}(v)$).
- $\mathsf{I\_am\_Internal} \equiv \neg\mathsf{I\_am\_Root} \wedge \neg\mathsf{I\_am\_Leaf}$.

Functions & Procedures.

- reset_BasicAlg_variables(): resets variables of the basic algorithm to their values in $C_{reset}^{\texttt{BasicAlg}}$.
- children(v): returns the set of children of v. Children are determined by the tree algorithm (Algorithm 5).
- update_variables(new_status): updates the local and shared variables.
  $status \leftarrow new\_status$
  if($\neg$I_am_Root) : $st_{v,parent} \leftarrow new\_status$
  if($\neg$I_am_Leaf) : $\forall u \in$children(v), $st_{v,u} \leftarrow new\_status$
  This procedure is used all along the algorithm to propagate the waves over the tree.

IV.2.1.2 - Additional Definitions

A *node* v is said *consistent* (with its shared registers) if $\forall u \in \mathcal{N}(v) : \ st_{v,u} = status$. A *configuration* is said *consistent* if every node is consistent. Notice that this property is checked by a node using the rule **Variables_Consistency**.

Definition 8 ($C_{normal}^{\texttt{ResetAlg}}$)**.** *$C_{normal}^{\texttt{ResetAlg}}$ is the configuration in which all nodes have status = normal, are consistent and no boolean signal is* `True`*. We denote by* PredRTerm *be the predicate defining $C_{normal}^{\texttt{ResetAlg}}$.*

Definition 9 (Safe configurations $\mathcal{C}_{safe}^{\texttt{ResetAlg}}$)**.** *The set of* `ResetAlg`*'s safe configurations $\mathcal{C}_{safe}^{\texttt{ResetAlg}}$ is defined by the following predicate :* $(status_{root} = normal \wedge \neg signal) \wedge (\forall v \in V \setminus \{root\} : [(status_v = status_{parent_v} = normal) \vee (\forall u \in \text{children}(v): status_v = status_u = reset)] \wedge [(status_v = normal \wedge \neg signal) \vee status_v = reset])$ *and all nodes are consistent.*

For example, the figure 5.10 represents a safe configuration on a tree of depth 2, assuming that it is also consistent. Nodes with $status = normal$ are denoted by letter with a n and those with $status = reset$ are denoted by a r.

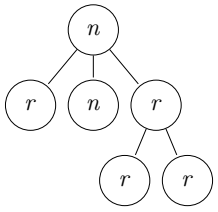


Figure 5.10: A safe configuration (assuming that the configuration is consistent)

IV.2.1.3 - Algorithm

The reset algorithm (Algorithm 6) is specified by the rules described below. They are presented by decreasing priority order, **Reset_Launch** having the highest priority and **Back_Normal** the lowest. If several rules are enabled for the same node v in a configuration C, the node is only eligible for the rule with the higher priority.

- The rule **Reset_Launch** is used by a *normal*-node to check whether the reset signal has been launched by the composition (if *signal* is `True`). In this case, variables are set to *initiate* to launch the waves (explained in the beginning of Section IV).

- The rule **Variables_Consistency** is used to check whether all shared variables *st* of the node contain *v*'s *status*, *i.e.*, this rule checks the consistency of the node. If enabled, the rule sets local and shared status variables to *initiate* to report the inconsistency.
- The rule **Neighbors_Coherence** is used by a non-leaf *reset*-node to check whether its children have also status *reset*, *i.e.* to check whether the *reset* wave is from the leaf to the root. If not, the rule sets local and shared status variables to *initiate* to report the *incoherence* between neighbors.
- The rule **Initiate** is executed either by the root to register that a reset is requested, or by internal *normal*-nodes to transmit to their parents the reset request of a child by setting its variable $st_{v,parent}$ to *initiate*.
- The rule **Freeze** is executed by a node to transmit down the $freeze$ wave. Since the $freeze$ wave goes from root to leaves, a node checks if its children are not already in status $freeze$ or *reset*. Indeed, if a child has $status \in \{freeze, reset\}$ this means that there is a rest of an older wave and if the node is activated now for **Freeze**, there is no guarantee that its children will be activated for **Reset**. Thus, the node is blocked until all its children have either *normal* or *initiate* status.
- The rule **Reset** is executed by a $freeze$-node if either it is a leaf or if all its children are in the status *reset* (for the root and internal nodes), *i.e.*, have already been activated for **Freeze**. This rule executes the procedure `reset_BasicAlg_variables()` of the basic algorithm and propagates the *reset* wave up to the root.
- The rule **Back_Normal** is executed by a *reset*-node to return to the *normal* status after the *reset* wave. Thus, it checks if its children have also status *reset*. This unfreeze wave is launched by the root and goes down to the leaves. Notice that the *signal* variable is set to `False` during this transition to delete remaining signals that have not been receipted.

Algorithm 6 Reset Algorithm `ResetAlg` for $v \in V$

1: **Reset_Launch** :
2: $\{status = normal \wedge signal\}$
3: update_variables($initiate$)
4:
5: **Variables_Consistency** : *(* Checks local and shared variables' coherence *)*
6: $\{(\neg$I_am_Root $\wedge\ st_{v,parent} \neq status)$
7: $\vee\ (\neg$I_am_Leaf $\wedge\ (\exists u \in$children(v): $st_{v,u} \neq status))\}$
8: update_variables($initiate$)
9:
10: **Neighbors_Coherence** : *(* Checks the neighbors' status coherence *)*
11: $\{status = reset \wedge \neg$I_am_Leaf $\wedge\ (\exists u \in$children(v): $st_{u,v} \neq reset)\}$
12: update_variables($initiate$)
13:
14: **Initiate** : *(* Propagates the initiate wave to the root *)*
15: $\{status = normal \wedge \neg$I_am_Leaf $\wedge\ (\exists u \in$children(v): $st_{u,v} = initiate)\}$
16: update_variables($initiate$)
17:
18: **Freeze** : *(* Propagates the freeze wave *)*
19: $\{status \notin \{freeze, reset\} \wedge (\forall u \in$children$(v)$: $st_{u,v} \notin \{freeze, reset\}) \wedge$
20: $[(\neg$I_am_Root $\wedge\ st_{parent,v} = freeze) \vee ($I_am_Root $\wedge\ status = initiate)]\}$
21: update_variables($freeze$)
22:
23: **Reset** : *(* Resets the variables and propagates the reset wave *)*
24: $\{status = freeze \wedge [$I_am_Leaf
25: $\vee\ (\neg$I_am_Leaf $\wedge\ (\forall u \in$children(v): $st_{u,v} = reset))]\}$
26: update_variables($reset$)
27: reset_BasicAlg_variables()
28:
29: **Back_Normal** : *(* Returns to the normal status *)*
30: $\{status = reset \wedge ([(\forall u \in$children$(v)$: $st_{u,v} = reset) \wedge$ I_am_Root$]$
31: $\vee\ [\neg$I_am_Root $\wedge\ st_{parent,v} \notin \{freeze, reset\}])\}$
32: update_variables($normal$)
33: $signal \leftarrow$ `False`

IV.2.2 - Correctness and Complexity Analysis of `ResetAlg` (Algorithm 6)

A sketch of the proof is in Sub-section IV.2.2.1 and the detailed proof in Sub-section IV.2.2.2.

IV.2.2.1 - Sketch

Assume that a rooted spanning tree of any depth has been built. In this section, we sketch the correctness (see Definition 7 for the self-stabilizing reset specification) and complexity proofs of `ResetAlg`. The full proof can be found in Section IV.2.2.2. Recall that $C_{normal}^{\texttt{ResetAlg}}$ is defined in Definition 8 and $\mathcal{C}_{safe}^{\texttt{ResetAlg}}$ is defined in Definition 9.

First, we prove in Sub-section IV.2.2.2.1 Lemmas 48, 49 and Corollary 6 used all along the proof. These technical lemmas concern some properties of the rules **Variables_Consistency** and **Neighbors_Coherence**.

Then, we prove in Sub-section IV.2.2.2.2 that a configuration is terminal iff no node has *status* or its shared variables set to $freeze$, *reset* or *initiate* (Lemmas 51, 52 and 53). Hence, a configuration is terminal iff nodes have *status* and shared variables set to *normal* (Proposition 5).

After this technical lemmas, the convergence and complexity of `ResetAlg` are proved.

In Sub-section IV.2.2.2.3, we prove the convergence of `ResetAlg` from any configuration to $\mathcal{C}_{safe}^{\texttt{ResetAlg}}$. The main idea is to show that the number of moves that a node can make depends on the moves of its parent, *i.e.* recursively, the total number of moves depends on the number of root's moves. Then, we focus on the root and prove the properties yielding the convergence to $\mathcal{C}_{safe}^{\texttt{ResetAlg}}$.

Thus, we first show that a node is activated for at most 6 moves between two activations of its parent (Lemma 54). Hence in an infinite execution (not terminated), the root is activated infinitely many times too. During such an execution, after the root is activated twice for **Reset**, a specific configuration in which all nodes are coherent and have status *reset* is reached (Lemma 55 and Corollary 7). From this configuration, after one root's **Back_Normal**, a configuration in $\mathcal{C}_{safe}^{\texttt{ResetAlg}}$ is reached. Hence, during this execution, the root has been activated a constant number of times. Furthermore, since between each two moves of the root, its children are eligible for 6 moves and, recursively, its grandchildren are eligible for 6^2 moves, *etc.*, $\mathcal{C}_{safe}^{\texttt{ResetAlg}}$ is reached after $O(n \cdot 6^p)$ moves, where p is the depth of the tree (Lemma 56).

Finally, in Sub-section IV.2.2.2.4, the termination of `ResetAlg` is proved. Using some lemmas of the previous section and with similar arguments, we show that, if there is no signal launched, from any configuration, $C_{normal}^{\texttt{ResetAlg}}$ is reached after $O(n \cdot 6^p)$ moves (Lemma 58). Indeed, from any configuration, after at most $O(n \cdot 6^p)$ moves, a terminal configuration is reached, *i.e.* $C_{normal}^{\texttt{ResetAlg}}$ is reached by Proposition 5.

In Sub-section IV.2.2.2.5, we focus on the third point of the definition of the reset problem. Lemma 59 proves that the configuration $C_{reset}^{\texttt{BasicAlg}}$ is reached in $O(n)$ moves from any configuration in $\mathcal{C}_{safe}^{\texttt{ResetAlg}}$ in which a signal is launched. Indeed, the waves proceed normally: the *initiate* wave reaches the root and then, $freeze$ and *reset* waves are initiated.

Finally, Theorem 4 concludes by proving that `ResetAlg` satisfies every condition of the reset specification in $O(n \cdot 6^p)$ moves. More precisely, in every of the three conditions, the configurations specified to be reached are reached in $O(n \cdot 6^p)$ moves. Notice that on a tree of depth 2, this move complexity is of $O(n)$ moves.

IV.2.2.2 - Detailed Proof

Recall that we assume that a rooted tree of any depth has been built. The proof is divided in five parts.

First, in Sub-section IV.2.2.2.1, technical lemmas (Lemmas 48-50 and Corollary 6) about some properties of the rules **Variables_Consistency** and **Neighbors_Coherence** are presented.

Then, in Sub-section IV.2.2.2.2, we establish that in the terminal configuration, all

nodes are *normal*-nodes (Lemmas 52 - 53). Thus, Proposition 5 states that $C_{normal}^{\texttt{ResetAlg}}$ is the only terminal configuration of `ResetAlg`.

In Sub-sections IV.2.2.2.3, IV.2.2.2.4 and IV.2.2.2.5, we prove the tree conditions of the reset specification.

Finally, the main Theorem 4 in Sub-section IV.2.2.2.6 states that `ResetAlg` satisfies each of the three conditions in the reset specification in $O(n \cdot 6^p)$ moves.

IV.2.2.2.1 - Preliminary Lemmas

This sub-section contains basic lemmas used throughout the proof. The first (Lemma 48) states that any node v is activated at most once for **Variables_Consistency** in any execution and, if v is not eligible at some point for **Variables_Consistency**, it never becomes such.
Then, we prove that, from a configuration C in which no node has $status = initiate$ and no node is eligible for **Variables_Consistency** and **Neighbors_Coherence**, no node becomes eligible for these two rules, as long as a reset signal is launched (Corollary 6). This is done using Lemma 49 in which we prove that, from C, the guard condition of **Neighbors_Coherence**'s rule cannot be `True`.

Finally Lemma 50 proves that nodes cannot be activated more than once for **Neighbors_Coherence**.

Lemma 48. *In any execution,*

1. *if a node v is not eligible for* **Variables_Consistency***, it will never be eligible for* **Variables_Consistency** *and,*

2. *a node v is activated at most once for* **Variables_Consistency***.*

Proof. The only way for a node to change its status and the values of its shared registers is to execute the function update_variables(). Any activation of this function makes the rule **Variables_Consistency** no more enabled. The function is executed at each rule activation. Thus a node that is eligible for **Variables_Consistency** becomes not eligible for this rule after its activation and a node that is not eligible remains not eligible after activation of any rule. That proves the lemma. □

Lemma 49. *Let C be a configuration in which no node is eligible for* **Variables_Consistency** *and* **Neighbors_Coherence***. In any execution from C, the rule* **Neighbors_Coherence** *is never enabled.*

Proof. We recall that the condition for **Neighbors_Coherence** to be enabled is $Cond = \exists v : \neg \mathsf{I\_am\_Leaf} \wedge status_v = reset \wedge (\exists c \in \mathsf{children}(v): st_{c,v} \neq reset)$.

Notice that **Variables_Consistency** is not enabled in any execution from C (from Lemma 48 and the lemma's assumptions). We prove the lemma by contradiction and consider the first configuration C' in which $Cond$ is `True` for a node v.

There are three cases to consider, according to the last move before reaching C'.

1. The child c has $status_c \neq reset$ and v has executed a rule that sets $status$ to $reset$,

2. v has $status_v = reset$ and children c has executed a rule that sets its status to a different value than $reset$, or

3. both c and v are activated in the same step: v switches to $reset$ while c takes a status different from $reset$.

Case 1. The only rule that sets the status to $reset$ is **Reset**. But v is eligible for this rule only if all its children are $reset$-nodes. Thus Case 1 is not possible.

Case 2. Since v is not eligible for **Neighbors_Coherence** in C, $status_c = reset$. The only rules that make c's status different from $reset$ are **Neighbors_Coherence** or **Back_Normal**. But, **Neighbors_Coherence** is not enabled in C. Furthermore, since $status_v = reset$, c cannot be eligible for **Back_Normal**. Thus Case 2 is not possible.

Case 3. The only possibility for v to set $status$ to $reset$ is when executing the rule **Reset** with $status = freeze$. If v is eligible for **Reset**, $status_c = reset$. Thus, c is possibly eligible only for **Back_Normal** or **Neighbors_Coherence**. But it cannot be eligible for **Back_Normal** because v has status reset and it cannot be eligible for **Neighbors_Coherence** because C' is the first configuration in which this rule is enabled. Thus Case 3 is not possible.

That ends the proof of Lemma 49. □

Corollary 6. *Let C be a configuration where $\forall v : status_v \neq initiate$ and no node is eligible for* **Variables_Consistency**, **Reset_Launch** *and* **Neighbors_Coherence**. *From C, in any execution in which no node is eligible for* **Reset_Launch**, *no node is eligible for* **Variables_Consistency** *and* **Neighbors_Coherence**.

Lemma 50. *In any execution, a node is activated at most once for* **Neighbors_Coherence**.

Proof. Let us prove this lemma by contradiction: suppose that a node v is activated twice for **Neighbors_Coherence** from C_1 and C_5. Notice that v cannot be a leaf since leaves are not eligible for **Neighbors_Coherence**. This implies two facts: a) in both configurations, at least one v 's child is an $initiate$-node and b) between C_1 and C_5, v is activated for **Freeze** and **Reset** in this order. Let C_2 be the configuration from which v is activated for **Freeze** and C_3 the one from which v is activated for **Reset**. In C_2, $\forall c \in \mathsf{children}(v)$, $st_{c,v} \in \{freeze, reset\}$ and in C_3, $\forall c \in \mathsf{children}(v)$, $st_{c,v} = reset$. Thus, between C_2 and C_3, all v's children are activated for **Freeze** and **Reset** in this order. Recursively, the argument applies up to the leaves: each descendant (children, grandchildren, *etc.*) of v is activated for **Freeze** and **Reset**, in that order, between the two activations (**Freeze** and **Reset**) of its parent. This implies that all nodes have been activated at least once before C_3, *i.e.* they are no more eligible for **Variables_Consistency** (by Lemma 48).

Let us now focus on the execution between C_3 and C_5 and suppose that c, a child of v, is an $initiate$-child in C_5. Since before C_3, c has been activated for **Reset** and in C_5, c is an $initiate$-node, c is activated for **Neighbors_Coherence** before C_5, from C_4. Indeed, others rules that set the status to $initiate$ cannot be enabled (**Back_Normal** not enabled). Similarly, at least one c's child cc has been activated for **Neighbors_Coherence** between c activation for **Reset** and C_4. Recur-

sively, the argument applies on internal nodes (leaves cannot be activated for **Neighbors_Coherence**). Thus, the contradiction arises when an internal node has only leaves as children. It has no child u eligible for **Neighbors_Coherence** and there is no other possibility to set its status to *initiate* in this configuration from the status *reset*.

Thus, nodes are not eligible twice for **Neighbors_Coherence**. □

IV.2.2.2.2 - $C_{normal}^{\texttt{ResetAlg}}$ is the Unique Terminal Configuration of `ResetAlg`

In this sub-section, we prove that, unless a node becomes eligible for **Reset_Launch**, $C_{normal}^{\texttt{ResetAlg}}$ is terminal and there is no other terminal configuration. In Lemmas 51 - 53, we prove that nodes in a terminal configuration are not in status *freeze*, *reset* or *initiate*. This implies Proposition 5: terminal configurations are those with node's status and shared variables set to *normal*. Nodes are consistent, otherwise **Variables_Consistency** would be enabled and the configuration wouldn't be terminal. $C_{normal}^{\texttt{ResetAlg}}$ is the unique configuration satisfying this properties.

Lemma 51. *In a terminal configuration, no node has status* $=$ *freeze.*

Proof. Assume that there exists a node v with $status_v = freeze$ in a terminal configuration C. In C, nodes are coherent (*i.e.* shared and local variables are equal) otherwise **Variables_Consistency** would be enabled.

First, let us consider that v is a leaf. Node v is eligible for **Reset** no matter its parent's status. Thus, if $status_v = freeze$ in C, v cannot be a leaf.

Consider now that v is either the root or an internal node. Since C is terminal, v is not eligible for **Reset**. Thus, $\exists c \in \mathsf{children}(v) : st_{c,v} \neq reset$. There are two sub-cases: (a) $status_c \in \{normal, initiate\}$ or (b) $status_c = freeze$.

Case (a) is possible only if c is not eligible for **Freeze**, *i.e.* c is an internal node (if c is a leaf, it would be eligible for **Freeze** with no condition) with a child cc with $status_{cc} \in \{freeze, reset\}$. If cc is a *reset*-node, it is eligible for **Back_Normal** with no condition on its children. Hence, if c is not eligible for **Freeze**, a c's child cc has $status_{cc} = freeze$ and is an internal node (leafs cannot have $status = freeze$, see the previous paragraph). This leads to the same case as for v, cc is an internal node with at least one child ccc with : $st_{ccc,cc} \neq reset$. This pattern of *status*' values can be repeated with some case (b) in between, but none of the node of the pattern can be a leaf. Hence, case (a) is possible only if the branch of *freeze*-nodes ends with at least one case (b) (a node and one of its child with $status = freeze$). But since leafs cannot have $status = freeze$, this is not possible.

Similarly, case (b), is not possible. First, c cannot be a leaf by the previous paragraph. Hence, v has $status_v = freeze$ only if v is the root or an internal node and if at least one of its children c is an internal node and has $status_c = freeze$. Recursively, c has at least one child satisfying also these conditions and so on. But since the tree has a finite depth, this is not possible: there is at least one internal node with only leaf children.

Thus, no node has $status = freeze$ in a terminal configuration. □

Lemma 52. *In a terminal configuration, no node has status* $=$ *reset.*

Proof. Assume that there exists a node v with $status_v = reset$ in a terminal configuration C. In C, nodes are coherent (*i.e.* shared and local variables are equal) otherwise **Variables_Consistency** would be enabled.

First, let us consider that v is the root. Since C is terminal, $\exists c \in \mathsf{children}(v) : st_{c,v} \neq reset$, otherwise v would be eligible for **Back_Normal**. But c cannot have $status \in \{normal, initiate, freeze\}$ because of **Neighbors_Coherence**. Thus, v cannot be the root in C.

Now, let us consider that v is an internal node or a leaf. Let p be its parent. $st_{p,v} \in \{reset, freeze\}$, otherwise **Back_Normal** would be enabled for v. But, by Lemma 51, $status_p \neq freeze$. Hence, v has $status_v = reset$ only if its parent has $status_p = reset$. Recursively, p's parent pp has also $status_{pp} = reset$ and so on. But the last parent is the root r and by the previous paragraph, $status_r \neq reset$. Thus, this case is also impossible.

Thus, no node has $status = reset$ in a terminal configuration. □

Lemma 53. *In a terminal configuration, no node has* $status = initiate$.

Proof. Assume that there exists a node v with $status_v = initiate$ in a terminal configuration C. In C, nodes are coherent (*i.e.* shared and local variables are equal) otherwise **Variables_Consistency** would be enabled. Furthermore, from Lemmas 51 and 52, no node has $status \in \{freeze, reset\}$ in a terminal configuration.

If v is the root. v's children may have only status *normal* and *initiate*. In both cases, v is eligible for **Freeze**. Thus, v cannot be the root in C.

Now, let us consider that v is an internal node or a leaf. If the parent p of v has $status_p = normal$, p is eligible for **Initiate**. Thus, p has $status_p = initiate$. Recursively, p's parent pp has also $status_{pp} = initiate$ and so on. But the last parent is the root r and by the previous paragraph, $status_r \neq initiate$. Thus, this case is also impossible.

Thus, no node may have $status = initiate$ in a terminal configuration. □

Proposition 5. *A configuration C is terminal iff it satisfies* $\mathsf{PredRTerm} \equiv \forall v \in V : status_v = normal \wedge$ *nodes are consistent* $\wedge \neg signal_v$, *i.e. it is* $C^{\texttt{ResetAlg}}_{normal}$.

Proof. First, notice that $C^{\texttt{ResetAlg}}_{normal}$ is terminal. Indeed, it is easy to see that no rule is enabled.

Let C be a terminal configuration. Since **Variables_Consistency** checks the consistency of nodes, they are consistent with there shared registers. By Lemmas 51, 52 and 53 no node has status *initiate*, *freeze* or *reset*. Then they are *normal*-nodes. Finally, there is no signal: a *normal*-node with a signal would be eligible for **Reset_Launch**. Thus, C satisfies $\mathsf{PredRTerm}$ *i.e.*, C is $C^{\texttt{ResetAlg}}_{normal}$. □

IV.2.2.2.3 - Convergence Properties of `ResetAlg` (Condition 1 of Def. 7)

In this section, we focus on the convergence property of `ResetAlg` as defined in the reset specification (Definition 7):

(Convergence) Starting from an arbitrary configuration, eventually a configuration in $C^{\texttt{ResetAlg}}_{safe}$ *is reached.*

This part is proved counting the number of moves made by each node. Indeed, we prove first that a node can be activated at most 6 times before its parent is activated once (Lemma 54). We also prove that the root can be activated at most 9 times (Corollary 57) using the fact that it can be activated twice for **Reset** (Lemma 55). Thus, a node at depth p can be activated $9 \cdot 6^p$ times. This results in an overall complexity of $O(n \cdot 6^p)$ moves (Lemma 56). From the configuration reached by the second root's **Reset**, we prove that a configuration in $\mathcal{C}_{safe}^{\texttt{ResetAlg}}$ is reached in 1 move.

We first prove that if a node is not activated, its children are activated for at most 6 moves.

Lemma 54. *Let C be any configuration and C' be a configuration from which a node v is activated for any rule. In the execution $C \xrightarrow{*} C'$, v's children are activated for at most 6 rules.*

Proof. Since some rule check the register of the parent and other do not, let us analyze the longest sequence of activations before C' of one v's child c (the figure 5.11 represents the possible transitions from each *status* value).

Thus, if $st_{v,c} = initiate$, c can be activated for at most 3 moves (in the order: **Reset** then **Back_Normal** and then **Initiate** or **Reset_Launch**, when $status_c = freeze$ and c is coherent).

If $st_{v,c} = reset$, c can be activated for at most 2 moves (first **Reset** and then **Neighbors_Coherence**, when $status_c = freeze$ and c is coherent). Indeed, in these cases, **Back_Normal** nor **Freeze** can be enabled since $st_{v,c} = reset$ satisfies none of their conditions.

If $st_{v,c} = freeze$, c can be activated for at most 6 moves (**Initiate** or **Variables_Consistency** or **Reset_Launch** or **Neighbors_Coherence**, then **Freeze**, then **Reset**, then **Neighbors_Coherence**, then **Freeze** and finally **Reset**, for any value of $status_c$). Notice that after the second **Reset**, c cannot be activated again for **Neighbors_Coherence**. Indeed, since **Freeze** and **Reset** check the *st* values of the children, between **Freeze** and **Reset**, c children have been also activated for **Freeze** and **Reset**. Thus, recursively, all children of c are coherent and have $status = reset$ when c is activated for **Reset**, *i.e.* none of them can change their *status* to an other *value* and allowing its father to be activated for **Neighbors_Coherence**.

Finally, if $st_{v,c} = normal$, c can be activated for at most 3 moves (**Reset**, **Back_Normal** and **Initiate**, when $status_c = freeze$ and c is coherent).

Thus, if v is activated from C, each of its children may be activated for at most 6 moves. □

In the following figures, the possible transitions (from a status to another) are depicted. Edges represent the transitions and the nodes of this diagram the resulting status value after a transition. The rules that can be applied to make a transition are giver over every edge. Some rules are labeled by "△" and "1". "1" means that the rule is activated only once (for example **Variables_Consistency**, by Lemma 48 or **Neighbors_Coherence** by Lemma 50). "△" means that this transition is not possible for the leaves.

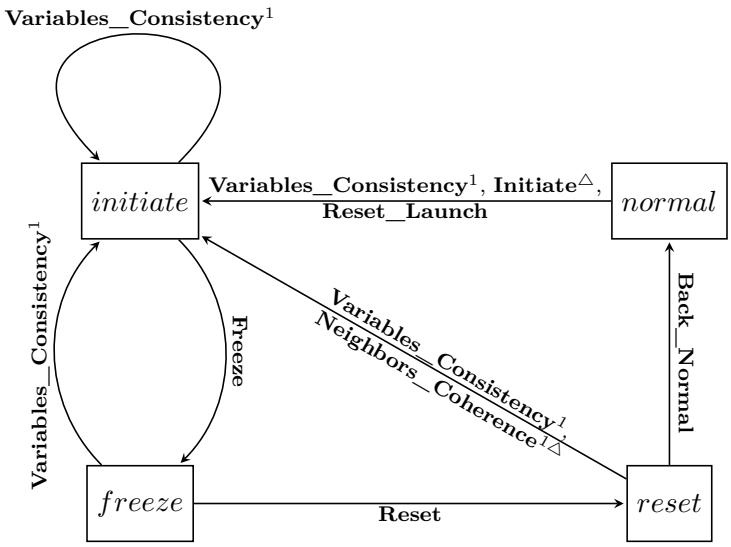


Figure 5.11: Rules activations of a node depending on its *status*

We now consider the root r: if r is activated twice for **Reset**, we can deduce the configuration reached after r is activated for **Reset** the second time. In this configuration, all nodes v are coherent and have $status_v = reset$ (Lemma 55 and Corollary 7). An illustration of the execution mentioned in the following lemma and the corollary is given in Figure 5.12.

Lemma 55. *Let C and C' two configurations from which the root r is activated for* **Reset**. *Let $\mathcal{E}$ be an execution such that $C \xrightarrow{*} C'$. Then in C', any node v but the root r satisfies $status_v = reset \wedge (\forall u \in \mathsf{children}(v)\colon st_{v,u} = reset) \wedge st_{v,parent} = reset$ and in consequence, r is the only eligible node in C'.*

Proof. Since **Reset** is enabled at r in C and C', in both configurations we have $(\forall c \in \mathsf{children}(r)\colon st_{c,r} = reset \wedge st_{r,c} = freeze) \wedge status_r = freeze$.

First, since $status_r = freeze$ in C', the last activation of r before C' is for **Freeze** in a configuration C_1. Thus in C_1, $\forall c \in \mathsf{children}(r)\colon st_{c,r} \notin \{freeze, reset\}$ is `True` but in C', $\forall c \in \mathsf{children}(r)\colon st_{c,r} = reset$. Hence, between C_1 and C', all children of r have been activated to change their variables and the last activation of each of them is for **Reset** (notice that in C', children are coherent since they have been activated at least once). To be activated for **Reset**, a node must be a $freeze$-node, then its last activation before **Reset** is for **Freeze**.

Recursively, the children of a r's child c is also only activated for **Freeze** and **Reset** (in that order) between the two activation of its parent for the same reasons. This induces that a node is activated for **Reset** only if its children are in status $reset$. Thus, in C', all nodes are in status $reset$ (**Neighbors_Coherence** not enabled since one of its children must be in status *initiate* and **Back_Normal** is enabled if its parent is in status *normal* and it is in status $freeze$).

The following execution is depicted in Figure 5.12. For example, let us consider a tree of depth 2 and a child c of r. Let C_2, respectively C_5, be the configuration in which

c is activated for the last time for **Freeze**, resp., for **Reset**. If c is a leaf, between C_5 and C' c is not eligible, that is, $status_c = reset$ and c is coherent in C'.

If c is an internal node, in C_2 $\forall cc \in \mathsf{children}(c)$: $st_{cc,c} \notin \{freeze, reset\}$ is `True` and in C_5, $\forall cc \in \mathsf{children}(c)$: $st_{cc,c} = reset$ is `True`. As for r, the two last activations of any child cc of c before C_5, are for **Freeze** and **Reset**.

As the tree is of depth 2, cc is a leaf. Let C_3, respectively C_4, be the configuration from which cc has been activated for **Freeze**, resp. for **Reset**. Thus, in C_5, $status_{cc} = st_{cc,c} = reset$ holds, *i.e.* cc is coherent. Between C_5 and C', cc is not eligible (no rule can be enabled). After c's **Reset** from C_5, c has $status_c = reset$ and is coherent. From this configuration, it cannot be eligible for any rule. □

Corollary 7. *In the conditions of Lemma 55, let C" be the configuration reached from C' by activating* **Reset** *at r. In C", all nodes v are coherent and have $status_v = reset$.*

Proof. Direct consequence of the r's **Reset** activation in C'. □

Figure 5.12 is a schematic representation of the necessary successive activations analyzed in the proof of Lemma 55 and Corollary 7. For simplicity, we illustrate the scenario only over the double-fan tree, but the proofs work for any rooted tree. The three levels of the double-fan tree are represented vertically (r stands for the root W_{min}, c for its children and cc for the children of M_{min}). Time runs from left to right and the sequence of configurations considered in the proof are depicted in a chronological order (on the top of the figure). The conditions allowing the activation of the rules are indicated vertically in red: re is for $reset$, f for $freeze$ and i for $initiate$. Thus, for example, on the first horizontal line, the first left bended arrow represents r's activation for **Reset** during the transition from C. If the reached configuration is used in the proof, it appears in the figure, otherwise it is not.

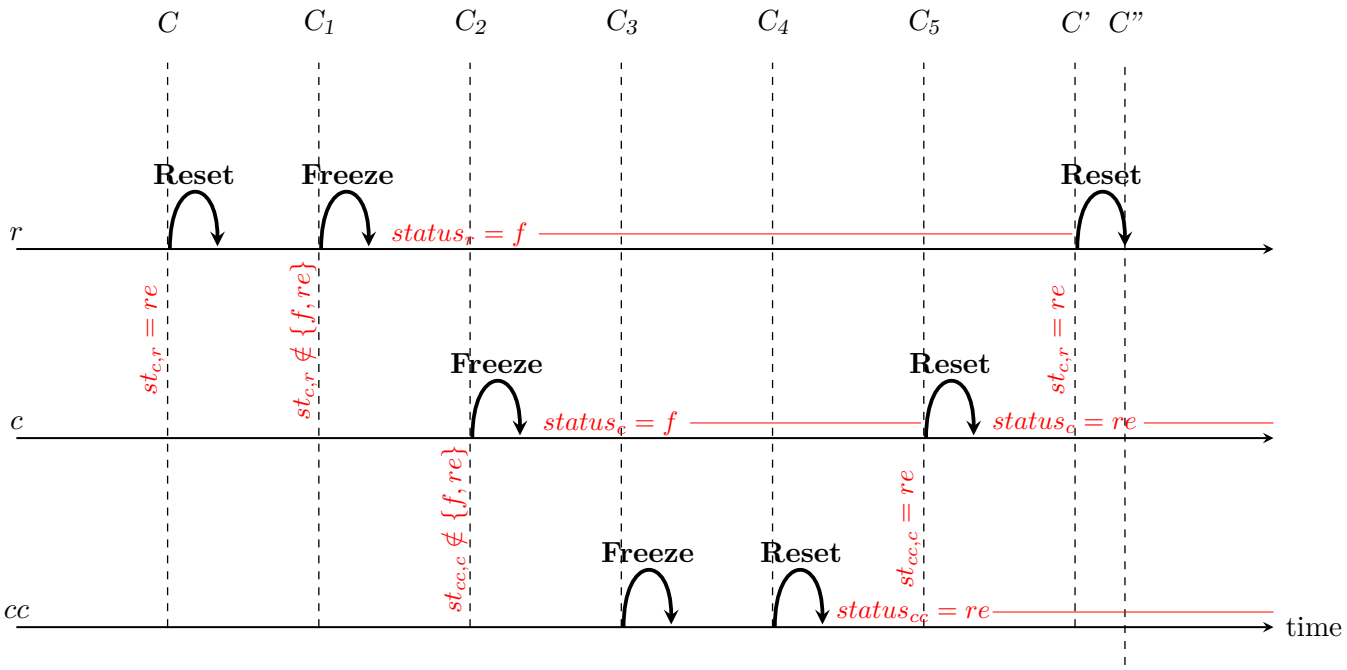


Figure 5.12: Illustration of the execution of `ResetAlg` in Lemma 55 and Corollary 7

Finally, we can prove that `ResetAlg` converges after $O(n \cdot 6^p)$ moves from any configuration to $\mathcal{C}_{safe}^{\texttt{ResetAlg}}$: from the configuration reached by the second root's **Reset** (by Lemma 55 and Corollary 7), $\mathcal{C}_{safe}^{\texttt{ResetAlg}}$ is reached after 1 more.

Lemma 56. *From any configuration C, any execution of* `ResetAlg` *reaches $\mathcal{C}_{safe}^{\texttt{ResetAlg}}$ after $O(n \cdot 6^p)$ moves.*

Proof. First, by Lemma 55, the root r is activated at most twice for **Reset** from C. Let C_1 be the configuration from which r is activated the first time and C_2 the configuration from which r is activated the second time (such that $C_1 \xrightarrow{*} C_2$). By Corollary 7, all nodes v (except the root) are coherent and have $status_v = reset$ in C_2. This configuration is reached after at most 8 moves: before C_1, r can be eligible for at most 3 moves (**Back_Normal**, **Initiate** or **Reset_Launch** and then **Freeze**) and between C_1 and C_2, r can be activated at most for the 3 same moves.

Since by Lemma 54, children of a node v are activated at most 6 times before one v's activation, each children of r are activated at most $8 \cdot 6 = 48$ times. Recursively, the grand children are activated at most $8 \cdot 6 \cdot 6 = 288$ times. More generally, a node at depth p is activated at most $8 \cdot 6^p$ times.

By Lemma 55, r is the only eligible node in C_2. After its activation, all nodes v (r included) are coherent and have $status_v = reset$. In this configuration, r is again the only eligible node (for **Back_Normal**). After its activation, $status_r = normal \wedge \neg signal \wedge \forall v \in V \setminus \{r\} : status_v = reset$ (**Back_Normal** sets *signal* to `False`) and all nodes are coherent (they all have been activated at least once for a rule). Hence, this configuration is in $\mathcal{C}_{safe}^{\texttt{ResetAlg}}$.

This leads to the fact that the algorithm converges to $\mathcal{C}_{safe}^{\texttt{ResetAlg}}$ in $O(n \cdot 6^p)$ moves from any configuration. □

IV.2.2.2.4 - Convergence Properties of `ResetAlg` in the Absence of Signals (Condition 2 of Def. 7)

In this section, we prove the termination property of `ResetAlg` as defined in the reset specification (Definition 7):

(Termination) If a finite number of signals are launched, a configuration in $\mathcal{C}_{normal}^{\texttt{ResetAlg}}$ is reached.

Thus, we consider executions in which there is no reset signal launched, *i.e.*, the environment does not set *signal* to `True`. By Lemma 55, the root can be activated twice for **Reset** in any execution (with or without signal). Nodes in the reached configuration after the second **Reset**'s activation are all *reset*-nodes and they are coherent. This allows us to prove that, from any configuration, the root can be activated at most 9 times (Lemma 57 - the figure 5.13 represents the possible transitions after each rule) if there is no new signal. Using the fact that a node is activated for at most 6 moves between 2 successive moves of its parent, we show that a node at depth p can be activated $9 \cdot 6^p$ times. This results in an overall complexity of $O(n \cdot 6^p)$ moves (Lemma 58). This implies that `ResetAlg` converges and thus reaches a terminal configuration (where no node is eligible), which is necessarily $C_{normal}^{\texttt{ResetAlg}}$ by Proposition 5.

Lemma 57. *From any configuration C, the root r is activated for at most 9 rules in any execution with no new signal.*

Proof. By Lemma 55 and Corollary 7, from any configuration, r can be activated twice for **Reset** (in C and C_1) and after the second activation, all nodes v are coherent and have $status_v = reset$. Before C, r can be activated for at most 3 rules (**Back_Normal**, **Initiate** and **Freeze** - the figure 5.13 represents the possible sequence of rules). Between C and C_1, r is activated for at most 3 rules (**Back_Normal**, **Initiate** and **Freeze**). From this configuration, this is easy to see that, since there is no new signal, each node is activated once for **Back_Normal**. Since no node is anymore eligible, this is a terminal configuration, *i.e.* the terminal configuration $C_{normal}^{\texttt{ResetAlg}}$ (Proposition 5).

Thus, from any configuration, the root is activated at most 9 times. □

We provide the following figure5.13 adapted to the proof of Lemma 57. It depicts the sequence of rule's activations. Similar labels as in Figure 5.11 are used. "1" means that the rule is only activated once (for example **Variables_Consistency**, by Lemma 48 or **Neighbors_Coherence** by Lemma 50). "△" means that this transition is not possible for the leaves. We add a label "□" to denote a transition which is not applied to the root.

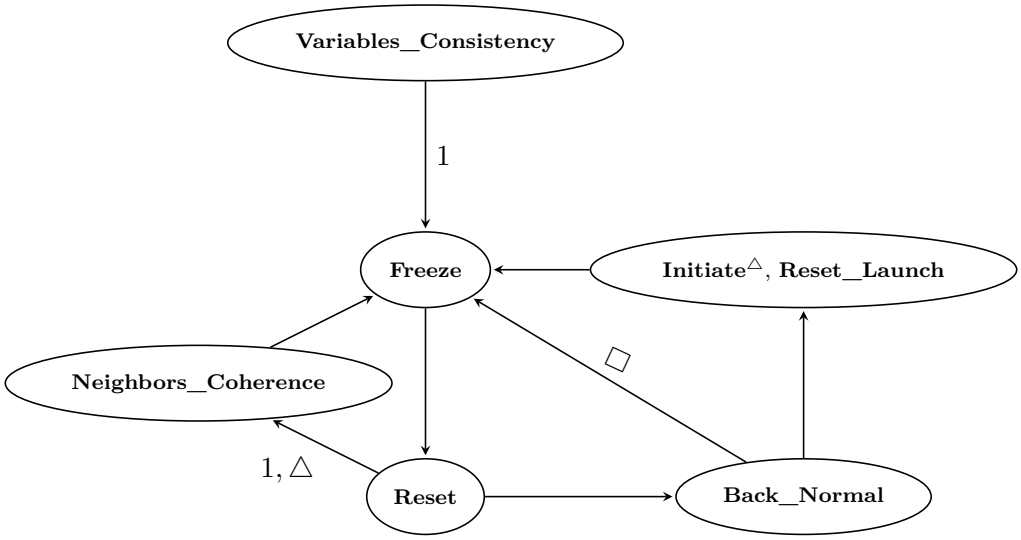


Figure 5.13: Possible sequence of rule's activations

Finally, we can prove that `ResetAlg` converges from any configuration to $C_{normal}^{\texttt{ResetAlg}}$ in $O(n \cdot 6^p)$ moves, in the absence of reset signals.

Lemma 58. *From any configuration C, any execution of* `ResetAlg`*, in which there is no new reset signal, reaches $C_{normal}^{\texttt{ResetAlg}}$ after $O(n \cdot 6^p)$ moves.*

Proof. First, by Lemma 55, the root is activated at most twice for **Reset** from C. Let C_1 be the configuration from which the root is activated the first time and C_2 the configuration from which the root is activated the second time (such that $C_1 \xrightarrow{*} C_2$). By Corollary 7, all nodes v are coherent and have $status_v = reset$ in C_2. Altogether, by Corollary 57, the root is activated at most for 9 rules.

Since by Lemma 54, children of a node v are activated at most 6 times before one v's activation, each children of the root are activated at most $9 \cdot 6 = 54$ times. Recursively, the grand children are activated at most $9 \cdot 6 \cdot 6 = 324$ times. More generally, a node at depth p is activated at most $9 \cdot 6^p$ times.

Notice that nodes could be eligible after the last activation of the root but since in C_2 all nodes v are coherent and have $status_v = reset$, the only possible activation for each node is once for **Back_Normal** (in absence of new signal).

This leads to the conclusion that, if there is no new signal, the algorithm converges to $C_{normal}^{\texttt{ResetAlg}}$ in $O(n \cdot 6^p)$ moves from any configuration. □

IV.2.2.2.5 - Re-initialization of `BasicAlg` (Condition 3 of Def. 7)

In this section, we focus on the reset property of `ResetAlg` as defined in the reset specification (Definition 7):

(Reset) Starting from any configuration in $\mathcal{C}_{safe}^{\texttt{ResetAlg}}$, if a finite number of signals and at least one are launched, $C_{reset}^{\texttt{BasicAlg}}$ is reached.

Recall that when the variable *signal* is set to `True`, we say that a signal is generated. Possible moves are analyzed in Lemma 59 in order to prove that, if from any configuration in $\mathcal{C}_{safe}^{\texttt{ResetAlg}}$ at least one signal is generated, then $C_{reset}^{\texttt{BasicAlg}}$ is reached (*i.e.*, each node has executed reset_BasicAlg_variables()). The following lemma is illustrated by Figure 5.14 on a tree of depth 2, but is proven for any rooted tree.

Lemma 59. *From any configuration $C \in \mathcal{C}_{safe}^{\texttt{ResetAlg}}$, if at least one reset signal is generated, then $C_{reset}^{\texttt{BasicAlg}}$ is necessarily reached after $O(n)$ moves.*

Proof. Let v be a node receiving a reset signal and C_1 be the configuration reached from C. In C_1, v is necessarily a *normal*-node and others nodes are *normal*- or *reset*-nodes distributed on the tree as in a configuration in $\mathcal{C}_{safe}^{\texttt{ResetAlg}}$. But notice that C_1 is not in $\mathcal{C}_{safe}^{\texttt{ResetAlg}}$since v has *signal* set to `True`. Furthermore, in C_1 nodes are consistent (by Lemma 48, since they are consistent in C).

In C_1, v is eligible for **Reset_Launch**. *normal*-node are not eligible and *reset*-nodes are eligible for **Back_Normal** if their parent are *normal*-nodes. Thus, after at most $O(n)$ **Back_Normal**, v is activated for **Reset_Launch**: this is the start of an initiate wave. If several nodes have received a reset signal and have been activated for **Reset_Launch**, the *initiate* wave comes from several parts of the tree but an *initiate*-node cannot be activated several times for **Initiate**. Since v's parent are all *normal*-nodes, this wave reaches the root after at most $O(n)$ **Initiate**. Then, if all children of the root are *normal*-nodes, the root may transform the *initiate* wave into a *freeze* wave. Notice that, if its children are still *reset*-nodes, the root waits till its children are activated for **Back_Normal** (since there is a finite number of other possible rules, its children are activated at some point). The *freeze* wave is broadcasted with the same mechanism if some children are *reset*-nodes. When the *freeze* wave reaches the leafs, the *reset* wave begins and is convergecasted. Hence, after $O(n)$ more moves, a configuration C" in which nodes are all *reset*-nodes after having executed **Reset** is reached. Indeed, *reset*-nodes are neither eligible for **Back_Normal** (their parents must be *normal*-nodes) nor for **Neighbors_Coherence** (Lemma 49). Since all nodes have executed **Reset** as last rule before C" and that `BasicAlg` is not eligible while nodes are not *normal*-nodes, $C_{reset}^{\texttt{BasicAlg}}$ is reached in C". □

The following figure illustrates the execution of Lemma 59: from a configuration in which a node has *signal* set to `True`, $C_{reset}^{\texttt{BasicAlg}}$ is reached. We use the same notations as in Figure 5.12.

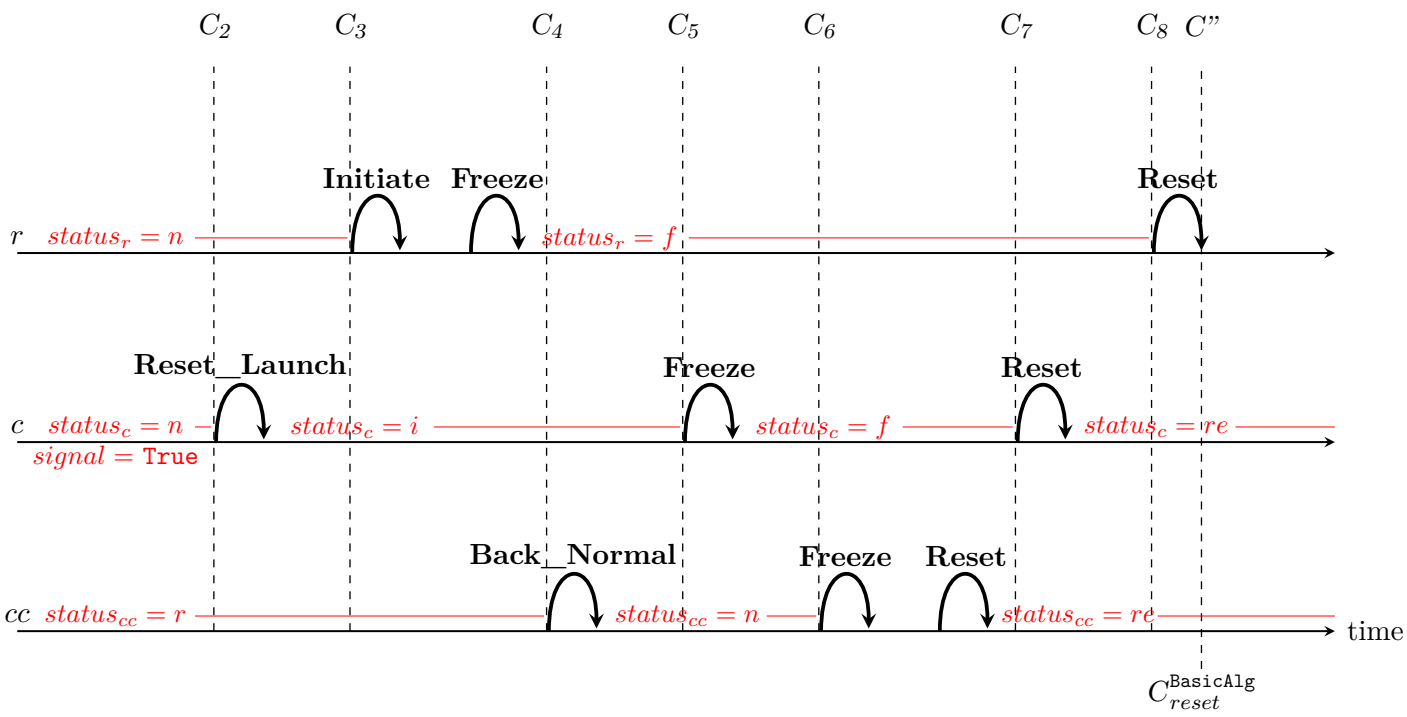


Figure 5.14: Illustration of the execution of `ResetAlg` in Lemma 59

IV.2.2.2.6 - Main Theorem : `ResetAlg` Correctness and Complexity

Recall that the self-stabilizing reset specification is in Definition 7, the configuration $C^{\texttt{ResetAlg}}_{normal}$ in Definition 8 and the safe configuration $\mathcal{C}^{\texttt{ResetAlg}}_{safe}$ in Definition 9.

Theorem 4. *Assuming that a tree of depth p is built,* `ResetAlg` *satisfies the self-stabilizing reset specification (as specified in Definition 7) in $O(n \cdot 6^p)$ moves, i.e., the configurations specified to be reached in every corresponding condition of this specification are each reached in $O(n \cdot 6^p)$.*

Proof. First, Lemmas 56 proves that from any configuration, a configuration in $\mathcal{C}^{\texttt{ResetAlg}}_{safe}$ is reached in $O(n6^p)$ moves, *i.e.* point 1 (Convergence) of the definition is satisfied.

The Lemma 58 proves that, if there is a finite number of signals, `ResetAlg` terminates (in $C^{\texttt{ResetAlg}}_{normal}$) in $O(n6^p)$ moves: point 2 (Termination) is satisfied.

Finally, the point 3 (Reset) is proved to be satisfied by Lemma 59 in $O(n)$ moves. □

Though this upper bound is high, assuming that the underlying tree is constructed by the algorithm `TreeAlg` proposed in Section IV.1, that is with $p = 2$, `ResetAlg` converges and performs the reset in $O(n)$ moves.

V - Composition

In this section, we explicitly present the final combination of `Async-GSA` (Alg. 3 and 4) with the reset module `ResetAlg` (Alg. 6) composed itself with the tree construction al-

gorithm `TreeAlg` (Alg. 5). All this, for obtaining a self-stabilizing version of `Async-GSA` (Alg. 7) following the transformation technique by local checking and global reset presented in Section II. Notice that `ResetAlg` resets an algorithm `BasicAlg` to a specific configuration called $C_{reset}^{\texttt{BasicAlg}}$ in the previous section. For `Async-GSA`, this configuration is $C_{init}^{\texttt{Async-GSA}}$.

The main goal of this section is to analyze the move complexity of the final combination. This analysis is made especially difficult by the assumption of an unfair scheduler. When considering rounds (synchronous or not) under a fair scheduler, the analysis is easier because the notion of round hides the useless actions. That is not the case under an unfair scheduler. A rough worst case analysis in terms of moves results in the product of the stabilization times of the composition modules taken separately, *i.e.*, in $O(n^4)$. This is because an unfair scheduler may choose as long as possible to activate nodes in a "wrong order" of modules' rules they are eligible to execute. That is, it may constantly privilege activating nodes executing a module A, while A can correctly execute only after the stabilization of another module B. This "spends" moves which do not "contribute" to advance the stabilization of the whole composition (and implies a multiplication of stabilization times of modules). See the following Section V.2 for more detailed description on this issue.

Despite this difficulty, we make a thorough complexity analysis and obtain, in the worst case, only $\Theta(n^2)$ moves till the stabilization to a stable marriage, starting from any configuration. We obtain this by establishing priorities between the guarded rules (locally at each node - see Alg. 7), and by relying on the bipartite topology of the underlying graph.

V.1 - Composition Algorithm `CompAlg` (Alg. 7)

V.1.1 - Variables and Predicates (for a node v)

Variables.

- *status*: variable of the reset algorithm `ResetAlg` (Alg. 6), read only in `CompAlg`. $status \in \{initiate, freeze, normal, reset\}$. `CompAlg` verify whether or not the value of this variable is normal. That is, the node does not participate in the reset procedure. Only in this case, reset signals can be launched and the rules of `Async-GSA` executed (lines 12 and 17).

Predicates.

- GuardsTreeAlg: the disjunction of all `TreeAlg`'s rule guards.
- GuardsResetAlg: the disjunction of all `ResetAlg`'s rule guards.
- GuardsAsync-GSA: the disjunction of all `Async-GSA`'s rule guards.
- $\mathsf{LP}_{m,w}$: a predicate defined in Definition 6 to prove the local checkability of `Async-GSA`; used by men to trigger a reset signal, whenever not satisfied.

- Tree_LC $\equiv$ $\forall u_1, u_2 \in \mathcal{N}(v)$: $min_{u_1,v} = min_{u_2,v}$. A predicate used to check the coherence of the shared registers of `TreeAlg` in all the neighbors of v. This is to prevent v from executing other than `TreeAlg`'s rules while tree is not yet stabilized, thus improving move complexity.

V.1.2 - Algorithm

The composition algorithm manages the activations of its modules through the implementation of priorities. The rules are presented and executed in the decreasing priority order: the first rule **Comp_Tree** has priority over the others and the last rule **Comp_SM** can be enabled only if neither `ResetAlg` nor `TreeAlg` has an enabled rule. These priorities are implemented by mutually exclusive rule guards. Here are the guarded rules of `CompAlg`:

- **Comp_Tree** is enabled to execute the rules of `TreeAlg` if the tree is locally incorrect (*i.e.*, at least on of its rules is enabled), according to GuardsTreeAlg.
- If the tree is locally correct according to GuardsTreeAlg and Tree_LC, **Comp_Reset** is enabled if GuardsResetAlg is also satisfied.
- The rule **Comp_Local_Check** is only enabled for men since $\mathsf{LP}_{m,w}$ is only checked by men. Moreover, it should be with $status = normal$, with no tree or reset modules' rules enabled. Note that the checking "w *in* Women" is equivalent to $w \in \mathcal{N}(v)$ since v is a men.
- Finally, **Comp_SM** is enabled if all others modules, except `Async-GSA`'s, are locally correct (*i.e.*, their rules are not enabled) and if $status = normal$.

Algorithm 7 Composition algorithm `CompAlg` for $v \in V$

Comp_Tree : *(* If a tree's rule is enabled *)*
  {GuardsTreeAlg}
    TreeAlg

Comp_Reset : *(* If a reset's rule is enabled *)*
  {¬GuardsTreeAlg ∧ Tree_LC ∧ GuardsResetAlg}
    ResetAlg

Comp_Local_Check : *(* Only for men: local checking and reset *)*
  {¬GuardsTreeAlg ∧ Tree_LC ∧ ¬GuardsResetAlg
  ∧($v \in$Men ∧ $\exists w \in$Women: ¬$\mathsf{LP}_{v,w}$) ∧ $status = normal$}
    launch a reset signal to ResetAlg

Comp_SM : *(* If no other rule is enabled *)*
  {¬GuardsTreeAlg ∧ Tree_LC ∧ ¬GuardsResetAlg ∧ $status = normal$
  ∧(($v \in$Men ∧ $\forall w \in$Women: $\mathsf{LP}_{v,w}$) ∨ $v \in$ Women) ∧ GuardsAsync-GSA}
    Async-GSA

V.2 - Correctness and Complexity Analysis of CompAlg (Algorithm 7)

Contrary to the classical time analysis in terms of rounds (asynchronous or synchronous) where the overall complexity of a composition is normally obtained by summing up the complexities of the modules, in case of moves, this complexity is upper-bounded by the multiplication of the modules' complexities. This is because with a distributed (asynchronous), and especially unfair, daemon, some nodes can be retained from being activated for a very long time (even though, they are eligible). Imagine a composition of two algorithms A_1 and A_2, stabilizing in $O(f_1(n))$ and $O(f_2(n))$, respectively, while executed in a stand alone mode. Moreover, to stabilize, A_2 assumes that A_1 has already stabilized to a correct configurations (satisfying the specification). That is, the composition of A_1 and A_2 stabilizes only after A_2 has stabilized, following the stabilization of A_1. Now, notice that the considered daemon may privilege to activate the nodes eligible for A_2 rules, while retaining from the execution those eligible for A_1, as long as possible. In a distributed setting, nodes executed for A_2 may not be aware that A_1 is not yet stabilized. Their moves thus may add an overhead of $O(f_1(n))$ to each move executed by A_1 (each such move may restart the computation of A_2). That is why the rough move complexity analysis of the composition results in the multiplication $O(f_1(n) \cdot f_2(n))$.

Following this reasoning, the rough move complexity of `CompAlg` is $O(n^4)$ moves ($O(\texttt{TreeAlg}) \cdot O(\texttt{ResetAlg}) \cdot O(\texttt{Async-GSA}) = O(n) \cdot O(n) \cdot O(n^2)$). We will perform a tighter time analysis proving an $O(n^2)$ move complexity for `CompAlg`. This analysis is complex due to several reasons. One is because we cannot ignore moves that are done out of the order needed by the modules for their respective stabilization (like those of A_2 above, executed before A_1 has been stabilized). In addition, the communication model by registers allows nodes to communicate different information to their neighbors, making the stabilization analysis more complex.
Another reason is the quite complex interleaving of the `CompAlg`'s modules: `ResetAlg` runs on a stabilized tree (assuming PredT is satisfied) and the rule **Comp_Local_Check** checks the variables of `Async-GSA` (local predicate LP) to trigger a reset signal sent to `ResetAlg`.

The latter reason leaded us to analyze the `CompAlg`'s convergence in two parts. We first prove that, from any configuration C_0, a configuration C_T in which PredT is satisfied, *i.e.*, where the tree is stabilized, is reached, while analyzing the corresponding time complexity. In the second part, we analyze the stabilization time from C_T until a terminal configuration C_M in which PredAsync-GSA (global predicate for a stable marriage with `Async-GSA`), PredT (global predicate of the tree built by `TreeAlg`) and PredRTerm (global predicate of the terminal configuration of `ResetAlg`), are all satisfied.

In the first part of the analysis, we consider two types of nodes: (1) nodes that will be activated for `TreeAlg` between C_0 and C_T and (2) nodes that are already stabilized for the tree (they are no anymore eligible for any `TreeAlg`'s rule). Notice that nodes which are activated for the last time for a `TreeAlg`'s rule change from type (1) to type (2). The main idea of the proof is to analyze how much moves can be made by a node before C_T, depending on its type. Figure 5.15 illustrates (using similar conventions as in Figure 5.12) a possible coexistence of such types of nodes during the segment between C_0 and C_T. The last activated node for `TreeAlg` is v so, it is of type (1) (during the red

execution segment in the figure), while w is of type (1) in the beginning, and of type (2) after its activation for `TreeAlg` in some configuration C_x (blue execution segment in the figure). A node u is of type (2) from the beginning.

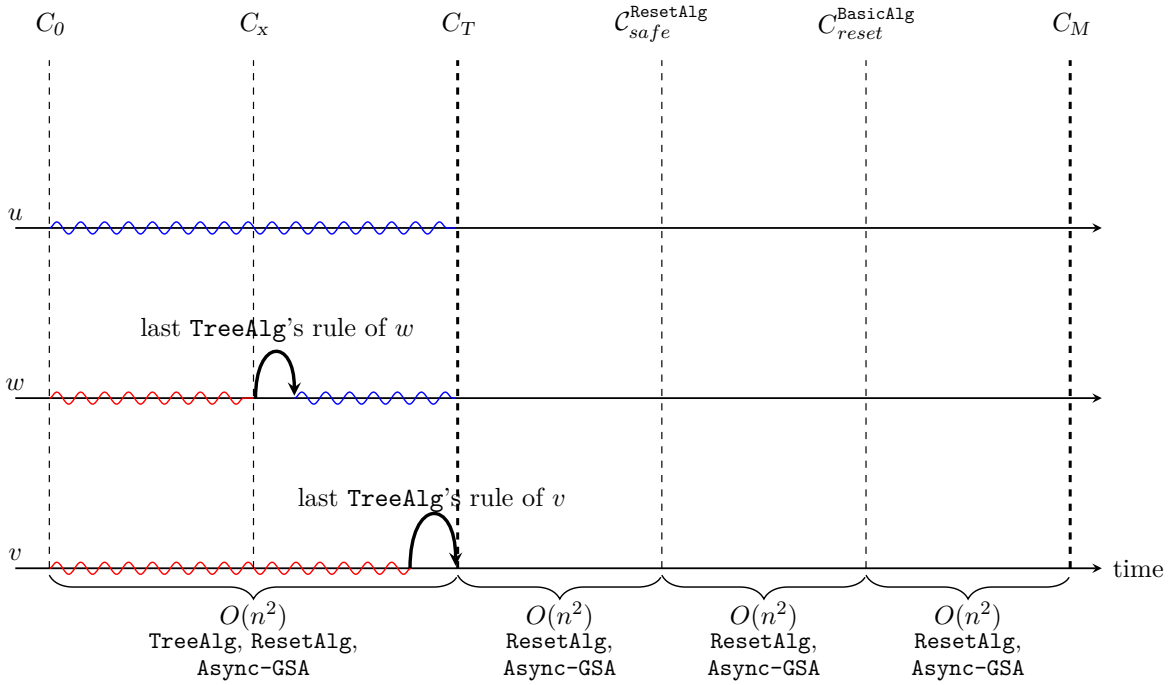


Figure 5.15: Illustration of the execution of `CompAlg`
Nodes are of type (1) during the red execution segments, and of type (2) during the blue ones.

We first prove that nodes of type (1) (like w or v) can be activated for a constant number of moves before their last activation for `TreeAlg` (Lemmas 60 - 66). Thus, after a constant number of its own moves, a node switches to type (2) (like w in C_x).

Then, we prove that a node of type (2) (like u or w after its last activation for `TreeAlg`) is eligible for $O(1)$ moves of `ResetAlg` and $O(n)$ moves of `Async-GSA` before C_T is reached, *i.e.*, till PredT becomes satisfied (Lemma 67). Concerning the $O(1)$ moves of `ResetAlg`, the main argument is that `ResetAlg` proceeds in "waves" (of broadcast and convergecast) propagated over a tree of depth 2 and coordinated by the root. But since the tree is not stabilized yet before C_T, waves cannot be propagated, leading to a constant number of `ResetAlg`'s moves. This implies that each type (2) node can be activated a constant number of times for $O(n)$ moves of `Async-GSA`, since a node can perform $O(n)$ moves in `Async-GSA` and partial `ResetAlg` can reset `Async-GSA`'s variables of the node a constant number of times. Thus, C_T is reached after $O(n^2)$ moves (Lemma 67).

Concerning the second part, after the tree has been built, in the worst case, an execution is divided into three additional sub-parts: i) an initial sub-part in which a partial reset (propagated on the tree) is executed, ii) a second sub-part during which a reset is performed after being triggered in a safe configuration (see Definitions 7

(self-stabilizing reset problem) and 8 (safe configuration)) and iii) a third sub-part, which corresponds to an execution of `Async-GSA` with the correct initialization. We discuss upper bounds for each of these three parts.

We first consider the sub-part i) with the partial reset. By Lemma 56, `ResetAlg` converges to $\mathcal{C}_{safe}^{\texttt{ResetAlg}}$ in $O(n)$ moves from any configuration (with stabilized tree of depth 2). This induces that **Reset** is executed a constant number of time: `Async-GSA` do not start again and again because of the reset of its variables. Thus after $O(n^2)$ moves from C_T, $\mathcal{C}_{safe}^{\texttt{ResetAlg}}$ is reached.

The sub-part ii) begins from $\mathcal{C}_{safe}^{\texttt{ResetAlg}}$ in which the reset can be triggered by some incoherent nodes (regarding its state and its neighbors' registers) or nodes involved in a blocking pair. The other nodes simply execute rules of `Async-GSA` (Algorithms 3 (women) and 4 (men)). The longest execution segment of this part is obtained when the unfair scheduler chooses to ignore the incorrect nodes (from executing the enabled rules of the incoherence detection - rule **Comp_Local_Check**). This may take at most $O(n^2)$ moves (of the `Async-GSA` rules): after a partial stable marriage has been built with the correct nodes (those that do not detect any incoherence) using `Async-GSA`, these nodes are no more eligible, at least because no woman can make a new proposal, as the end of her preference list has been reached or a partner has been found. The task of building a partial stable marriage takes $O(n^2)$, still from Lemma 44. Then, an incorrect node is activated, triggering a reset. The triggered reset ends after $O(n)$ moves, resetting `Async-GSA` to $C_{reset}^{\texttt{BasicAlg}}$ and `Async-GSA` can be executed again.

Finally, consider the sub-part iii), Lemma 44 gives the $O(n^2)$ moves upper bound in terms of moves from $C_{reset}^{\texttt{BasicAlg}}$. Furthermore, since `Async-GSA` est locally checkable, no reset signal can be launched from $C_{reset}^{\texttt{BasicAlg}}$. Thus, `ResetAlg` converges to $C_{normal}^{\texttt{ResetAlg}}$ in $O(n)$ moves.

Then, from C_T, after $O(n^2)$ moves, a configuration C_M (in which `PredAsync-GSA` is satisfied) is reached (Lemma 68).

This justifies the overall complexity of $O(n^2)$ moves (Theorem 5). This bound is also correct in terms of rounds and is tight (see the scenario reaching it in Sub-section III.3).

V.2.1 - Stabilization of the Tree (to PredT)

Lemma 60. *From any configuration, a man v is activated for at most* 1 *move (of* `TreeAlg`*) before its last activation for* `TreeAlg`*.*

Proof. Let C be any configuration. In C, v is either eligible for `TreeAlg` or not eligible for `TreeAlg`. Notice that, by Lemma 45, men are eligible for at most 2 moves: **Update** and **I_am_not_root**.

If v is not eligible, it cannot become eligible for these two rules. Indeed, these rules check only v's variables (for the men): if these variables are already with the 'good' values, no other rule can change them. Thus, v is never eligible for these rules.

If v is eligible for `TreeAlg`'s rules, it is for **Update** and/or **I_am_not_root**. If it is only for one rule, it becomes not eligible after its activation and v is in the previous case: it cannot become eligible for the other rule. If v is eligible for both, it cannot be activated for another rule between the activations. This is because, if **I_am_not_root** is activated, it is already eligible in C and because the actions of **I_am_not_root**

rule do not affect the variables verified by the guard of **Update** (and vise versa). Thus, a man v can be activated for at most 1 move before its last `TreeAlg`'s activation. □

Lemma 61. *Let C be any configuration in which there is a woman w not eligible for* `TreeAlg` *and with* Tree_LC *not satisfied. From C, w is not activated for any move before its last activation for* `TreeAlg`*.*

Proof. Since w is not eligible for any `TreeAlg`'s rule in C, **Update** is never enabled for w (shared registers are already consistent and not other rule change their values). Furthermore, since Tree_LC is not satisfied and v is not eligible for `TreeAlg` in C, **I_am_not_root** is not the first `TreeAlg`'s move of w.

Hence, w is activated for **I_am_root**, when all neighbors agree on the *min* value `True`. Between the last **Update** activation of w's neighbors and the activation for **I_am_root**, Tree_LC is satisfied. But w is already eligible for **I_am_root**, *i.e.*, w cannot be activated for any other rule before **I_am_root** (rules's priorities). Furthermore, since men have been activated for **Update**, this induces that their *min* value are correct: w is the root. Thus, it cannot be activated for **I_am_not_root** after **I_am_root**. I_am_Root is its last `TreeAlg`'s activation, after no other move. □

Lemma 62. *Let C be any configuration in which there is a woman w eligible for* `TreeAlg` *and with* Tree_LC *not satisfied. From C, w is activated for at most* 2 *moves (of* `TreeAlg`*) before its last activation for* `TreeAlg`*.*

Proof. In C, w is eligible for **Update** and/or for **I_am_not_root** (Tree_LC not satisfied). After these activations, either w is no more activated for `TreeAlg` (the last activation for `TreeAlg` already happened) or there is a last activation for **I_am_root** (women are eligible for at most 3 moves, once for each, by Lemma 46). We consider the case in which w will be activated for **I_am_root**. Since in C, Tree_LC is not satisfied, neighbors are activated for **Update** and will agree on the *min* value `True` in a configuration C_1. Any other rule is enabled for w during this time: Tree_LC is not satisfied. In C_1, w is eligible for **I_am_root** and this is its last activation for `TreeAlg`.

Thus, after at most 2 moves of `TreeAlg`, w's next activation is for its last `TreeAlg`'s rule. □

Lemma 63. *Let C be any configuration with a woman w that is not eligible for* `TreeAlg` *and that satisfies* Tree_LC*. From C, before the first activation of one of its neighbors, w can make at most* 5 *moves.*

Proof. In C, since Tree_LC is satisfied for w, all w's neighbors agree on a *min* value.

We analyze first the number of moves that w can make with `ResetAlg` before any activation of its neighbors.
If neighbors' *min* values are `True`, w "thinks" that she is the root. The longest sequence of `ResetAlg`'s activations that does not require a shared variable update of the neighbors is either: i) **Reset** and then **Back_Normal** or ii) **Reset_Launch** or **Variables_Consistency** or **Initiate** and then **Freeze** Thus, w can be activated at most for 2 moves.
If neighbors' *min* values are `False`, w "thinks" she is a leaf. Node w can be activated

for at most 3 moves: for **Variables_Consistency** or **Reset_Launch**, for **Freeze** and then, for **Reset** (see Figures 5.11 and 5.13).

Furthermore, if $status_w = normal$ (after **Back_Normal** or before **Reset_Launch** or **Variables_Consistency** or **Initiate**, for example), v can be activated for 2 moves of `Async-GSA`: **Refusal_Management** and then either **Propose** or **Refuse**.

Comp_Local_Check is not enabled since w is a woman.

Thus, after at most 5 moves (3 of `ResetAlg` + 2 of `Async-GSA`), w cannot anymore be activated if its neighbors are not activated. □

Lemma 64. *Let C be any configuration in which there is a woman w not eligible for* `TreeAlg` *and with* Tree_LC *satisfied. From C, w is activated for at most 7 moves before its last activation for* `TreeAlg`*.*

Proof. Since w is not eligible for `TreeAlg`'s rules in C, its shared registers are consistent and thus **Update** is never enabled for w. Moreover, since Tree_LC is satisfied in C, its neighbors are coherent: their shared variables *min* have the same value `True` or `False`. If w's neighbors are not activated for **Update**, w is not anymore eligible for `TreeAlg`. If some w's neighbors are activated for **Update**, this means that all its neighbors' *min* have a wrong value: all w's neighbors will be activated for **Update**. In this case, w will be activated either for **I_am_root** (if in C the values are `False`) or **I_am_not_root** (if in C the values are `True`) in a configuration C_1. The execution can be divided into two parts: before the first w's neighbors' activation and after this activation. In the following, we analyze each part.

Before the first neighbor's activation, by Lemma 63, w can make at most 5 moves.

After at least one w's neighbors' activation for **Update**, w is either i) still not eligible for `TreeAlg` and with Tree_LC not satisfied (*min* values were `False` and not all men have been activated) or ii) eligible for `TreeAlg` (**I_am_not_root**) and with Tree_LC not satisfied (*min* values were `True` and not all men have been activated) or iii) eligible for `TreeAlg` (**I_am_root** or **I_am_not_root**) and with Tree_LC satisfied (all men have been activated for **Update** in the same step). Notice that the case in which w is still not eligible for `TreeAlg` and with Tree_LC satisfied is not possible.

If w is in case i), by Lemma 61, it cannot make any move before C_1.

If w is in case ii), by Lemma 62, it can make at most 2 moves before C_1.

Finally, if w is in case iii), it is eligible either for **I_am_root** (if *min* values are `True`) or **I_am_not_root** (if *min* values are `False`). Since by Lemma 46 men are activated once for **Update**, the *min* values will not change again. Thus, w's next activation can only be for its last `TreeAlg`'s rule.

Thus, after at most 7 moves, w's next activation can only be for its last `TreeAlg`'s rule. □

Lemma 65. *Let C be any configuration in which there is a woman w eligible for* `TreeAlg` *and with* Tree_LC *satisfied. From C, w is activated for at most 9 moves before its last activation for* `TreeAlg`*.*

Proof. In C, w is eligible for any of the 3 `TreeAlg`'s rules and, by Lemma 46, it is eligible at most once for each rule. The order of activation depends on the neighbors shared values.

In C, w is eligible for at most 2 moves: either one **Update** and one **I_am_root** (if neighbors' *min* values are `True` in C) or **Update** and **I_am_not_root** (if neighbors' *min* values are `False`).
Let suppose that w is activated for **Update** and is also eligible for **I_am_root** or **I_am_not_root**. After this move, Tree_LC may be not satisfied if at least one neighbor has also been activated for **Update** in the same transition. In this case, by Lemma 61 and Lemma 62, after at most 2 moves, w is activated for its last activation.
If after the w's activation for **Update**, Tree_LC is still satisfied, w is still eligible for either **I_am_root** or **I_am_not_root**. Notice that, before the next w's activation, Tree_LC can be not anymore satisfied. If so, again by Lemma 61 and Lemma 62, after at most 2 moves, w is activated for its last `TreeAlg`'s rule. Consider now that w is activated for either **I_am_root** or **I_am_not_root**. After this activation, there are several cases.

- Either Tree_LC is still satisfied and w not eligible for `TreeAlg` (no neighbor's **Update** during the same transition). By Lemma 64, after at most 7 moves, w is activated for it last `TreeAlg`'s rule.
- Or Tree_LC is not satisfied and w not eligible for `TreeAlg` (during the same transition, w is activated for **I_am_not_root** and at least one of its neighbors is (and not all)activated for **Update**). By Lemma 61, w is only eligible for it last `TreeAlg`'s activation.
- Or Tree_LC is not satisfied and w eligible for `TreeAlg` (during the same transition, w is activated for **I_am_root** and at least one of its neighbors (and not all) is activated for **Update**). By Lemma 62, after at most 2 moves, w is activated for it last activation.
- Or Tree_LC is satisfied and w eligible for `TreeAlg`. This last case is only possible if during the same transition, w is activated (for **I_am_not_root** or **I_am_root**) and all its neighbors are activated for **Update**. Thus, w is eligible for **I_am_root** if it has been activated for **I_am_not_root** earlier or reciprocally. This move is the last `TreeAlg`'s activation since women may be activated at most once for each `TreeAlg`'s rule.

Notice that the same argumentation leads to the same result if w is activated first for **I_am_root** or **I_am_not_root** and then for **Update**.
Thus, after at most 9 moves (2 first `TreeAlg`'s rules + 7 of Lemma 64), w's next activation can only be for its last `TreeAlg`'s rule. □

Lemma 66. *From any configuration C, a node v is activated for at most 9 moves before its last activation for* `TreeAlg`.

Proof. In C, v can be of four different types:

1. Tree_LC is not satisfied but v is not eligible for `TreeAlg`,
2. Tree_LC is not satisfied and v is eligible for `TreeAlg`,

3. Tree_LC is satisfied and v is not eligible for `TreeAlg` or,

4. Tree_LC is satisfied but v is eligible for `TreeAlg`.

First, by Lemma 60, if v is a man, no matter the type, it can make at most 1 move before its last activation.

Now, let v be a woman. Women of type 1, 2, 3 and 4, make 0, 2, 7 and 9 moves respectively, by Lemma 61, Lemma 62, Lemma 64 and Lemma 65, respectively.

Thus, after at most 9 of it own moves, v is only eligible for its last `TreeAlg`'s rule. □

Lemma 67. *Let C be any configuration in which* PredT *is not satisfied. From C, a configuration C_1 in which* PredT *is satisfied is reached in at most $O(n^2)$ moves.*

Proof. Assume that such a configuration C_1 is never reached. There are tree possibilities:

1. either no node is eligible for `TreeAlg` but PredT is not satisfied, or

2. some nodes are infinitely often activated for `TreeAlg`, but PredT is never reached, or

3. at least one node u that is eligible (or will be eligible) for `TreeAlg`, is never activated.

By Lemma 47 (any terminal configuration (of `TreeAlg`) satisfies PredT) and the fact that `TreeAlg` is activated in **Comp_Tree** with no restriction (first priority in `CompAlg`), case 1 is not possible.

Furthermore, by Lemmas 45 and 46, each node is eligible at most 3 times for `TreeAlg` and by Lemma 47, when all nodes have been activated for `TreeAlg`, C_1 is reached. Thus, case 2 is also not possible.

Now, let us consider case 3. We show that, between C and C_1, the set of other (than u) nodes (those not activated for `TreeAlg`) are eligible for at most $O(n^2)$ moves in overall. This implies that u cannot stay unactivated, *i.e.* C_1 (in which PredT is satisfied) is reached.

Let us thus analyze the move complexity, during the execution segment from C till C_1, of the set S of nodes which are never eligible for `TreeAlg` rules, *i.e.*, for which GuardsTreeAlg is not satisfied during the whole execution. These nodes form a forest that is part of the future tree. Other nodes, eligible for `TreeAlg`, may join the sub-trees (forest), but a cycle (of parent pointers) can never be formed, since this requires for a node in S to be activated for `TreeAlg`. Such joins are made after the last `TreeAlg`'s move of those nodes, *i.e.* after their $O(1)$ moves (of any type). Thus, a leaf v still eligible for `TreeAlg` is not in the forest. Notice that `ResetAlg` uses convergecasts and broadcasts to propagate the waves. Thus, M_{min} and W_{min} wait for answers from their children before being eligible, *i.e.*, v's parent cannot run the algorithm to convergecast a wave. Similarly, if M_{min} or W_{min} are not in the tree, broadcast waves cannot be completed. Hence, while not all nodes are in this tree (PredT is not satisfied), waves are blocked. But when nodes (eligible for `TreeAlg`) are activated for their last action (for `TreeAlg`), they "take their place" in the tree and participate to the current blocked wave. Furthermore, when joining the tree, nodes cannot "disturb" the waves: either the nodes already have the required (by the current wave) status or it shifts to *initiate* (**Variables_Consistency**,

Reset_Launch or **Neighbors_Coherence**). In this case, the *initiate* wave may possibly reach the root (if it is not blocked at some point) and be transformed in a *freeze* broadcast wave. But this wave cannot end and be transformed in a successfully *reset* wave since the tree is not complete. Furthermore, the next join cannot shift the *freeze*-nodes back to *initiate*.

Furthermore, women that are activated for `TreeAlg` may be activated for a constant number of `Async-GSA`'s and `ResetAlg`'s moves before their last `TreeAlg`'s activation (men cannot by Lemma 60). But these moves do not influence nodes in S. Indeed, those women are activated for `Async-GSA`'s and **Reset** only in the context of Lemmas 64 and 65, *i.e.* they are eligible for those moves when men on the other side are all eligible for `TreeAlg` thus, are not in S.

Hence, over this (dynamic) forest structure where GuardsTreeAlg is not satisfied (and thus PredT is satisfied) for every node (in the segment from C to C_1), at most $O(n^2)$ moves are executed by the nodes, by Lemma 68 (including the joining nodes after their own $O(1)$ moves of any type, by Lemma 66). Thus, there is no infinite cycle and after $O(n^2)$ moves, the last eligible node for `TreeAlg` is activated and a configuration C_1 satisfying PredT is reached. □

V.2.2 - Convergence after PredT is satisfied

Lemma 68. *Starting from a configuration C where* PredT *is satisfied,* `CompAlg` *reaches a terminal configuration in $O(n^2)$ moves where* PredAsync-GSA *is satisfied too.*

Proof. Since in C PredT is satisfied, by Lemma 56, in $O(n)$ moves, a configuration C_1 in $\mathcal{C}_{safe}^{\texttt{ResetAlg}}$ is reached.
Between C and C_1, men with $status = normal$ and with a local predicate LP false can be activated for **Comp_Local_Check**. This new signal does not interrupt the convergence of `ResetAlg` to $\mathcal{C}_{safe}^{\texttt{ResetAlg}}$: Lemma 56 is proven with no condition on signals. Since each node is activated a constant number of times for `ResetAlg`'s rules and thus is a constant number of times with $status = normal$, each man can be activated a constant number of times for **Comp_Local_Check** (**Comp_Local_Check** changes the status to *initiate*). That is $O(n)$ additional moves for **Comp_Local_Check** before C_1.
Furthermore, between C and C_1, every *normal*-node (and with all LP `True` for men) can also be activated for `Async-GSA`, and all such nodes together for $O(n^2)$ moves (by Lemmas 42 and 41). Notice that, since `ResetAlg` is activated only for $O(n)$ moves, **Reset** is activated a constant number of times for each node. The activation of a node for **Reset** does not influence the `Async-GSA`'s moves of other nodes: when a man is activated for **Reset**, the only incidence is that women may be stuck in the process of proposal/acceptance (since the *request*'s value does not correspond anymore to the process and thus to the rules' guards) and when a woman is activated for **Reset**, men may be eligible for **Comp_Local_Check**. Hence, a node may not be restarted for `Async-GSA` more than a constant number of times. Thus, after $O(n^2)$ moves (of all algorithms together) C_1 is reached.

When C_1 is reached, if **Comp_Local_Check** is activated (at most once per man, since reset signal launch changes the status of a node to *initiate* and that `ResetAlg` satisfies the specification 7), a reset is launched and $C_{reset}^{\texttt{BasicAlg}}$ is reached in additional

$O(n)$ moves (by Lemma 59). During this convergence to $C^{\texttt{BasicAlg}}_{reset}$, *normal*-nodes may be activated for `Async-GSA` for $O(n^2)$ moves together (by Lemmas 42 and 41 and the *freeze* wave is propagated normally - the reset and the tree are stabilized - from the root to the leafs interrupting all nodes before resetting). By definition of $C^{\texttt{BasicAlg}}_{reset}$ (here, $C^{\texttt{BasicAlg}}_{reset}$ is $C^{\texttt{Async-GSA}}_{init}$), Π is now satisfied and since `Async-GSA` is locally checkable, **Comp_Local_Check** cannot be enabled anymore. Then, in $O(n)$ moves, $C^{\texttt{ResetAlg}}_{normal}$ - a terminal configuration projected on `ResetAlg` - is reached. From $C^{\texttt{BasicAlg}}_{reset}$, by Corollary 5, a terminal configuration (a projection on `Async-GSA`) C_3 satisfying `PredAsync-GSA` is reached after $O(n^2)$ additional moves (since `Async-GSA` is locally checkable and by Lemma 43). At that moment, a terminal configuration of the whole composition `CompAlg` is reached. □

Theorem 5. *The algorithm* `CompAlg` *solves the stable marriage problem in* $O(n^2)$ *moves.*

Proof. By Lemma 67, from any configuration in which `PredT` is not satisfied, after $O(n^2)$ moves, a configuration C in which `PredT` is satisfied is reached. From C, by Lemma 68, after $O(n^2)$ moves, a terminal configuration where `PredAsync-GSA` is satisfied is reached. □

VI - Conclusion

In this chapter, we have presented our second asynchronous self-stabilizing algorithm for SMP. This algorithm is built using the local checkability scheme of [APSVD94] that we adapted to our model and purpose (Definition 6). Local checkability is a well-know method allowing to compose a non-self-stabilizing algorithm, a local detector and a reset components in order to build a self-stabilizing version of the given initialized algorithm. Though the technique is well known, it had to be adapted in a delicate way. Moreover, neither it provides the move complexity of the composition, nor the components themselves are provided automatically.

For being able to use the adapted version of this technique we first designed a non-self-stabilizing algorithm to be transformed. This is an asynchronous distributed version of GSA (`Async-GSA` - Alg. 3 and 4) that solves SMP in $\Theta(n^2)$ moves. Since no self-stabilizing reset algorithm running under an unfair daemon in a link register model was designed and analyzed (for move complexity) before this work, we design also a reset algorithm (`ResetAlg` - Alg. 6) executed over an underlying rooted tree. Thus, we propose also a tree construction algorithm (`TreeAlg` - Alg. 5) that builds a rooted tree of depth 2 in $O(n)$ moves. The reset algorithm runs on this tree and its move complexity is $O(n)$.

Then, we compose those algorithms together (`Async-GSA`, `TreeAlg`, `ResetAlg` and the detector into `CompAlg` - Alg. 7) in a way providing a self-stabilizing version of `Async-GSA` that solves SMP in $\Theta(n^2)$ moves. The complexity analysis reaching this bound is intricate and constitutes one of the main contributions of this thesis. Finally note that all algorithms use registers of only few bits, and the confidentiality of the preference lists is always kept.

Chapter 6

Extensions to Variants of SMP

Contents

Many variants of the original stable marriage problem were studied. Each corresponds to a particular application domain. We consider some of them and discuss the possibility to obtain self-stabilizing solutions based on the algorithms we proposed (Chapters 4 and 5) for the basic problem. In each case, we study how these algorithms can be adapted. First, the definitions of preference list and consequently of blocking pair and stable matching must be extended. In some cases, the solution is an easy adaptation of the solutions for the basic stable marriage. In some other cases, the algorithms must be more deeply modified, as well as their proofs, but stay relevant. Finally, there are cases where, due to the particularity of the proposed algorithms (*e.g.*, the extensively used phase breakdown or the local checkability), suitable simple adaptations are inappropriate and different approaches are required. These cases are discussed as perspectives in Chapter 7, Section II.

The variants, for which we present a solution, are all mentioned in the reference book of Gusfield and Irving [GI89]. Their names are: stable matching with unequal sizes of opposite sets, stable matching with indifference (ties in preference lists), stable matching with unacceptable partners (incomplete lists), many-to-one matching (Hospitals/Residents problem) and many-to-many matching. In the following, each variant is considered separately. It is explained and the required changes to the algorithm and to the proof are indicated. Notice that the modifications for each case are presented in such a way that they can be simply combined together to obtain a general algorithm covering all the different variants. It is also interesting to notice that the changes (when needed) are very similar for both algorithms. This observation comes from the fact that in both cases we use local checking.

We start by extending the basic definition of stable matching, to have a uniform framework for the considered variants.

First, to allow many-to-many (and many-to-one) matchings, each node v must be matched to at most $b \geq 1$ partners. In this case, b is called the *capacity* and v is said to be *b-matched* if it is matched with exactly b partners[1]

[1]For simplicity, we have a constant b for every node, but this can be easyly adapted to the case where each node u has its individual capacity b_u.

Then, the definition of preference list is extended to allow unacceptable partners. Each node v has a preference list of k neighbors ($0 \leq k \leq n$) in preference order. A node u is *acceptable* for v iff u is in v's preference list, *i.e.*, $u \in v.pref$. A node prefers to stay single rather than to be matched to an unacceptable partner.

The definition of (extended) stable marriage/matching still relies on the absence of BP, whose definition must be extended too.

Definition 10 (Blocking Pair (BP))**.** *Given an extended matching $\mathcal{M}$, a pair (w, m) is a blocking pair iff the following conditions are satisfied:*

1. *w and m are not matched together, i.e. $(w, m) \notin \mathcal{M}$;*
2. *w and m are acceptable for each other;*
3. *w is not b-matched $\vee$ w prefers m to at least one of her b partners, i.e. $\exists\ m' :\ (w,\ m') \in \mathcal{M} \wedge \mathsf{priority}(w,m) < \mathsf{priority}(w,m')$;*
4. *m is not b-matched $\vee$ m prefers w to at least one of his b partners, i.e. $\exists\ w' :\ (w',\ m) \in \mathcal{M} \wedge \mathsf{priority}(m,w) < \mathsf{priority}(m,w')$.*

I - Subsets of unequal Size, $b = 1$

In this variant, sets of women and men can be of different cardinality. Furthermore, the nodes' capacity is 1. As a consequence, some nodes can be single in the final matching.

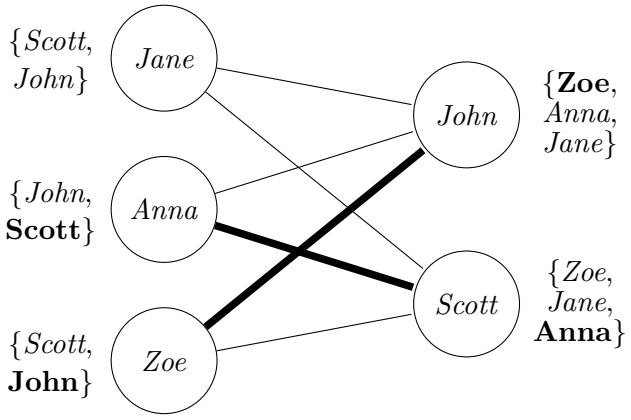


Figure 6.1: Stable marriage in a system with subsets of unequal size

Adaptation of the Two Phase Algorithm. This is the simplest variant, since there is nothing to change in the basic algorithm. Indeed, if a woman detects no BP and reaches the end of her preference list without finding any match, she is single and no more enabled. Symmetrically, if a man does not receive any proposal, he stays single and not enabled. The time complexity analysis for this extension stays relevant, but depends on the size of the largest subset.

Adaptation of `CompAlg`. This is also the easiest variant for `CompAlg` since it needs no change. Indeed, the only problematic point is the the presence of singles (a woman reaching the end of its list or a man receiving no proposal) in the final matching. In this situation no reset signal should be launched since matching is stable. In this matching, nodes in the opposite set of the singles are all matched to a better ranked partner than the singles (since the matching is stable). Hence, as each time preferences are checked using the local predicates, no reset signal can indeed be launched.

II - Stable Matching with Indifference (preference lists with ties), $b = 1$

In this variant, ordering in the preference lists is not required to be strict. In the literature studying this variant (*e.g.*, [IMM99, BM06, IMO09]), several sub-variants are considered. These variants affect the stability of the required matching, and depend on the definition of BP. There are three ways to extend Definition 10 of a BP (w, m). In all of them, points 1 and 2 in Def. 10 remain the same. Points 3 and 4 may be kept or slightly modified:

(a) w and m strictly prefer each other (like in Def. 10),

(b) w strictly prefers m and w is not worse than the actual partner of m, or reciprocally (that is replacing in 3 or 4 $<$ should be replaced respectively by $\leq$),

(c) m is not worse than the actual respective partner of w and reciprocally (that is replacing in 3 and 4 $<$ by $\leq$).

Depending on the choice, the type of matching stability changes. With (a) it is called weakly stable, with (b) strongly stable and with (c) super stable. With (b) and (c), there is no guarantee that a stable marriage exists [Irv94].

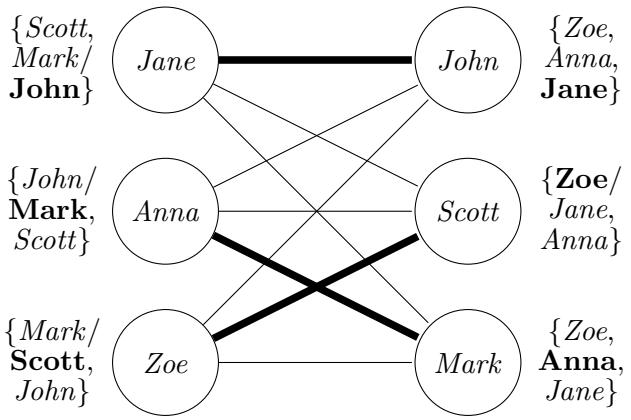


Figure 6.2: Weakly stable matching with indifference

Adaptation of both Algorithms to Condition (a). It suffices to remark that, when breaking the ties arbitrarily, a matching stable with condition (a) is a stable marriage (as noticed in [Irv94]). Thus only a slight modification of the proposed algorithms is needed. The ties are arbitrarily broken in advance, and the corresponding preference lists are appropriately adjusted. There is no change in the complexity analysis of both algorithms.

Conditions (b) and (c) Both cases raise a general issue concerning problems for which there is not always a stable matching. If their specification only concerns terminating executions (in a configuration with a stable matching), both solutions are still relevant. If their specification asks also for detecting the absence of stable matching, the issue is complicated (in particular, non-terminating executions are possible). We delay its discussion to the last chapter on perspectives (Chapter 7, Section II).

III - Unacceptable Partners (incomplete preference lists), $b = 1$

In this variant, some partners can be *unacceptable* for some nodes. This means that a node prefers to stay single rather than to be matched to an unacceptable partner. This is expressed in the preference list so that unacceptable partners are absent. In other words, they are not ranked. As mentionned in [Irv94], some nodes can be single in a final stable marriage. However, a stable marriage always exists (possibly with single nodes) [GI89].

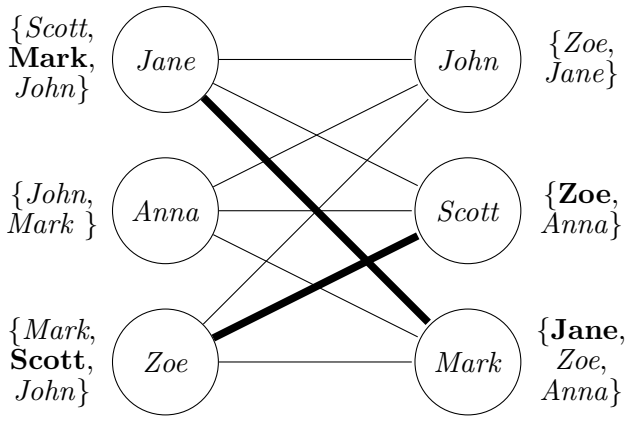


Figure 6.3: Stable matching with unacceptable partners

Adaptation of the Two Phase Algorithm. Adapting the algorithm to this variant requires minor changes. First, because to be single is a better choice than to be married to an unacceptable partner, the priority function (see Chapter 4, Section II) is adjusted: for a node $u \in \mathcal{N}(v)$, $\mathsf{priority}(v,u) = n+2$ iff $u \notin v.pref$ (recall that if $u = v.marriage =$

`Null`, *i.e.*, v is single, $\mathsf{priority}(v,u) = n+1$).[2] This simple adaptation ensures that, for a node v, an unacceptable node u always has the lowest possible priority (even comparing to being single). Thus, u will never be a part of the $\mathcal{C}_v$ set (set of mutually preferred neighbors of v), used in other predicates of the basic algorithm. Notice that the values of the priority function p are used in $\mathcal{C}_v$ (Sect. II). This implies that u will be excluded from any action contributing to the creation of matchings, in all the rules of type **Propose**, women's **Confirm** and **Accept**. Note that **Confirm** for men does not contribute to the creation of a matching, since a man eligible for this rule is already considered as married ($m.proposal.marriage = m$). Hence, new unacceptable matchings (involving an unacceptable partner) cannot be created. Thus the proof of the basic solution stays relevant, if no unacceptable matching exists due to a "bad" initialization. Notice that the original proof does not make any assumption on the size of the preference list. Moreover, the part of the proof asserting that a terminal configuration with a stable marriage is reached is still valid if some nodes end single.

But what if some unacceptable matchings exist at initialization? They have to be reset using **Reset**. To ensure that the actions of these rules are launched, the incoherent pointers' predicates (constituting the guards of **Reset** rules) has to be adapted. This is for detecting whether the pointers of a node v, $v.marriage$ and $v.proposal$ (used to indicate and create matchings) point to unacceptable partners. Thus, both predicates $\mathsf{IncoherentPointersW}(v)$ and $\mathsf{IncoherentPointersM}(v)$ are modified, each in the same way, by adding the following disjunction:

$$\mathsf{priority}(v,v.marriage) = n+2 \vee \mathsf{priority}(v,v.proposal) = n+2.$$

Due to this modification and by the correctness of the original algorithm, in at least $O(n^4)$ moves, every node is activated. Thus, all incoherent pointers are reset and never become incoherent again (in the way defined above), as it is explained in the previous paragraphs. Then, a stable matching is reached in at most additional $O(n^4)$ moves.

Adaptation of `CompAlg`. As for the Ackermann *et al.* based algorithm, the priority function (see Chapter 5, Section II) must be adapted in the same way to handle unacceptable partners. Hence, for a node $u \in \mathcal{N}(v)$, $\mathsf{priority}(v,u) = n+2$ iff $u \notin v.pref$ (recall that if $u = v.marriage\_pref = \texttt{Null}$, *i.e.*, v is single, $\mathsf{priority}(v,u) = n+1$). This induces that single nodes prefer to stay single instead of being matched with an unacceptable node.

Note that `Async-GSA` cannot build a pair with unacceptable partners: a woman cannot propose to an unacceptable partner (he is not in her list) and a man cannot accept a proposition from an unacceptable woman ($\mathsf{priority}$ returns $n+2$).

As for the first algorithm, there is still a problem if a pair with an unacceptable partner is already present in the initial configuration. The local predicate must detect it but it does not as it is, even with the new value of $\mathsf{priority}$. Hence, we add the following local predicate in a conjunction with the original $\mathsf{LP}_{m,w}$ (as defined in Section III.2).

$$\mathbf{P}_{m,w}^{Unacceptable} \equiv \neg[(\mathsf{priority}(w,m) = n+2 \wedge request_{w,m} \neq \texttt{None})$$

[2] Recall that it is assumed that node u communicates to $v \in \mathcal{N}(u)$ the value of $\mathsf{priority}(u,v)$. Thus, together with the previous assumptions, it is required that u communicates the priority $\mathsf{priority}(u,v) = n+2$ to an unacceptable node v.

$$\vee\ (\mathsf{priority}(m,w) = n+2\ \wedge\ request_{m,w}\ \notin\ \{\texttt{None}, \texttt{No}\})]$$

The first part of this predicate detects the cases where the woman is proposing to an unacceptable partner. Indeed, if a man is unacceptable for a woman, the woman cannot propose to this man. Thus, if $request_{w,m} \neq \texttt{None}$, this is because of a bad initialization. The second part allows to detect when men accept unacceptable partners. Note that, since $\mathbf{P}_{m,w}^{Unacceptable}$ only read local variables or registers and the `Async-GSA` rules cannot create states not satisfying this predicate, `Async-GSA` is still locally checkable. Furthermore, this does not change `Async-GSA`'s complexity, neither the global complexity of `CompAlg`.

IV - Many-to-One (Hospitals-to-Residents Problem)

This variant, introduced by Gale and Shapley in their seminal work [GS62], considers two sets of members, hospitals and residents, of different sizes. Each hospital can be matched with several residents, up to its capacity $b \geq 1$, and each resident can be matched to a single hospital. The sum of capacities has not to be equal to the number of residents.

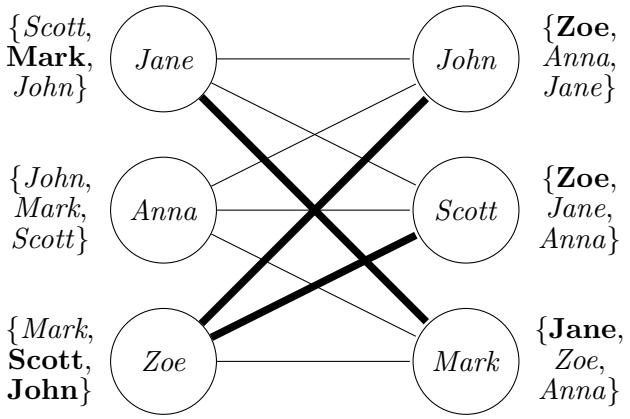


Figure 6.4: Many-to-one matching with $b = 2$

Adaptation for both Algorithms. A reduction from the many-to-one case to the one-to-one case is known [GI89]. At the algorithm level, each occurrence of a hospital h in the preference list of a resident is replaced by the sequence $h[1], h[2]\ \ldots,\ h[b]$ of hospital instances (the ties between this instances are broken arbitrarily)[3]. Then, at each activation of a hospital h with capacity b, b similar instances, $h[1], h[2]\ \ldots,\ h[b]$, of the algorithm are executed sequentially. In this transformation, a hospital node is eligible if at least one of the rules of any of its b instances is enabled. Then, it can be activated, if chosen by the scheduler. The resulting stable matching corresponds

[3]One assumes that the capacity b is known in advance by the residents or that they can read it in a register.

to the many-to-one matching. For the time complexity of this extension, refer to the complexity of the more general many-to-many variant.

V - Many-to-Many

The many-to-many extension of the stable marriage and its variants have been intensively-studied in the literature (*e.g.*, [Sot99, BAM03, MMNO04, BAM07, FGS10, EO16]). Maybe the reason is that the formulation of the problem corresponds to real situations involving markets. There are many variations of the many-to-many stable matching. Some formulations use preference lists of subsets and do not assume any compatibility (for instance the subset $\{v_1\}$ can be more preferred than the larger subset $\{v_1, v_2\}$). In all generality, a stable matching does not always exist. These variants are far from the solution that we gave and we will only focus here on the case of *responsive preferences* [RS90, Alk99], that is when preferences over subsets are consistent with the individual preferences over nodes (*e.g.*, for example, $\{v_1, v_2\}$ is more preferred than $\{v_1\}$ or $\{v_2\}$ and $\{v_1, v_2\}$ is more preferred than $\{v_1, v_3\}$ if v_2 is more preferred than v_3). Thus, simple sequences of nodes specify preference lists. This also implies that our extended definition of stable marriage (in this chapter) includes this variant of SMP with responsive preferences. It is known that a stable marriage always exist with responsive preferences, and in the case where initialization of the nodes is allowed, an iterative variant of the Gale and Shapley algorithm can be used (see [Alk99]). Here, we focus on the basic case having possibly two different size sets and unacceptable partners. Their members are commonly named workers and firms and each one can be matched multiple times (up to its capacity $b \geq 1$).

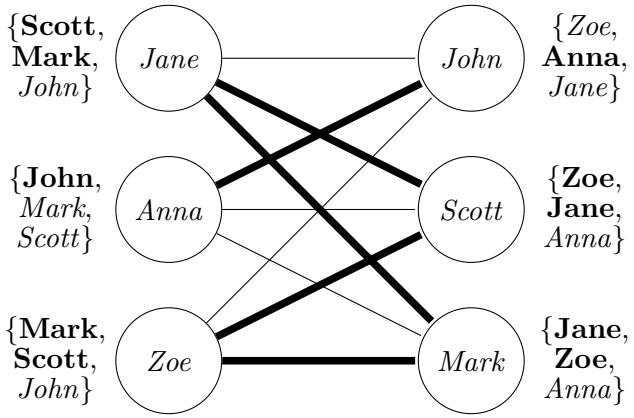


Figure 6.5: Many-to-many matching with $b = 2$

Adaptation of the Two Phase Algorithm. The solution we give for this case is adapted to all the variants presented till now. It is based on a similar transformation as for the many-to-one case, applied this time on both sides. Basically, each occurrence of v in the preference list of a member on the opposite side is replaced by the sequence $v[1], v[2] \ldots, v[b]$ of instances. Then, at each activation of a member v with capacity b, b similar instances of the algorithm are executed one after the other. However, problems

can appear. For example, let v and u be two members of different sets. Two instances of v can be or can become matched with two instances of u. If the instances of v are the most preferred for u, then no BP will ever appear and change the situation, leaving two same nodes matched twice together.

To avoid this situation, an instance $v[i]$ of v should be prevented to propose to and become matched with a neighboring instance $u[k]$ already matched or having a proposal from another instance $v[j]$ of the same node v. For that, each instance of v computes the set of such "occupied" neighboring instances, denoted $\mathsf{Occupied}$, and excludes them from the set of mutually preferred instances $\mathcal{C}$ (see Chapter 4, Section II), at every activation. Let us denote the currently considered instance of v by $v[this]$. Then,

$$\mathsf{Occupied}_{v[this]} = \{u[k] \in \mathcal{N}(v[this]) \mid \exists j \neq this, \exists i : (v[j].marriage = u[i] \vee v[j].proposal = u[i])\}.$$

This implies that the set of instances $\mathsf{Occupied}_{v[this]}$ will be excluded from $\mathcal{C}_{v[this]}$ at any activation of $v[this]$, and thus from any action contributing to creation of matchings, in all the rules of type **Propose**, **Confirm** and **Accept** (as is necessary by the explanation above).

This modification ensures that no new "double" matchings between the same two nodes are created. However, similarly to the solution for the variant with unacceptable partners, such "double" matchings, existing at initialization, have to be deleted (using **Reset**). Here again, this can be done by adapting $\mathsf{IncoherentPointersW}(v[this])$ and $\mathsf{IncoherentPointersM}(v[this])$. We present this adaptation together with the adaptation needed to tolerate unacceptable partners (so that the result works for all the considered variants). Each of the two predicates are modified in the same way, by adding the following disjunction:

$$[\mathsf{priority}(v[this],v[this].marriage) = n+2 \vee \mathsf{priority}(v[this],v[this].proposal) = n+2] \vee [\exists i \neq this : (v[this].marriage = x[j] \wedge v[this].proposal = y[k] \wedge v[i].marriage = z[l] \wedge v[i].proposal = w[m]) \wedge \{x,y\} \cup \{z,w\} \nsubseteq \{\texttt{Null}\}].$$

Due to this modification and by the correctness of the original algorithm, in at least $O(N^4)$ moves, every node is activated, where N is the overall number of the instances. Thus, all incoherent pointers are reset (by **Reset**) and never become incoherent again (in the way defined above). Then, a stable matching is reached in at most additional $O(N^4)$ moves.

Adaptation of `CompAlg`. As for the first algorithm, we apply the transformation described in the previous section to both sides. Hence, each occurrence of v in the preference list of a member on the opposite side is replaced by the sequence $v[1], v[2] \ldots, v[b]$ of instances. Then, at each activation of a member v with capacity b, b similar instances of the algorithm are executed one after the other.

The same problem also appears with this algorithm: two instances of a node v may be matched to two different instances of the same node u. Hence, `Async-GSA` must be modified for avoiding such double matches in the same way. For that, each instance of v computes the set of "occupied" neighboring instances (already matched with v), denoted $\mathsf{Occupied}$, and excludes them from consideration in the preference list $pref$ and

from $\mathcal{N}(v)$ (see Chapter 5, Section III), at every activation. Let us denote the currently considered instance of v by $v[this]$ and b_u be the capacity of b. Then,

$$\mathsf{Occupied}_{v[this]} = \{u[k] \in \mathcal{N}(v[this]) \mid \exists j \neq this, \exists i : v[j].marriage\_pref = u[i]\}.$$

Hence, the set of instances $\mathsf{Occupied}_{v[this]}$ will be excluded from $pref$ and from $\mathcal{N}(v)$ at any activation of $v[this]$ for `Async-GSA`, and thus from any action contributing to creation of matchings in particular since $\mathsf{next}(pref)$ cannot anymore return these instances.

But, as explained before, this "double matchings" may be already build at initialization and the local predicate must detect it. Thus, we add the following local predicate in a conjunction with the original $\mathsf{LP}_{m,w}$ (as defined in Section III.2). We include the modification for unacceptable partners to give a global solution for all the variants treated till now.

$$\begin{aligned}\mathbf{P}^{Extensions}_{m[i],w[j]} \equiv\ & \neg[(\mathsf{priority}(w[j],m[i]) = n+2 \ \wedge\ request_{w[j],m[i]} \neq \texttt{None}) \\ & \vee\ (\mathsf{priority}(m[i],w[j]) = n+2 \ \wedge\ request_{m[i],w[j]} \notin \{\texttt{None}, \texttt{No}\}) \\ & \vee\ (\exists k \neq i :\ m[i].marriage\_pref = x[j] \ \wedge\ m[k].marriage\_pref = z[l] \\ & \wedge\ x = z)]\end{aligned}$$

Note that similarly to this predicate, all other local predicates has to be modified using similar notations to manage the different instances.

Since `Async-GSA` cannot fall in such a problematic case defined by $\mathbf{P}^{Extensions}_{m[i],w[j]}$, it remains locally checkable and thus can be composed into `CompAlg`. Furthermore, the analysis complexity is now depend on the total number N of instances but stays of the same order (since b is constant).

Chapter 7

Conclusion

Contents

I - Summary

During the design of asynchronous self-stabilizing distributed algorithms solving SMP, we faced different issues. The first is asynchrony, in relation with our goal of efficiency concerning the proposed algorithms. The second issue is the requirement of privacy for the participant's information, leading to keep their preference lists secret. The third issue is the assumption of a strong adversary, under the form of an unfair demon, still in relation with the best possible performances.

Nevertheless, we proposed two self-stabilizing distributed algorithms that solves SMP in an asynchronous communication model with an unfair daemon and respecting the privacy (only some binary queries and responses about the preference lists are transmitted).

Based on the Ackermann *et al.*'s idea (of two phases), we proposed a first algorithm (Chapter 4) that solves SMP in $O(n^4)$ moves and rounds in the state reading model. The two phases mechanism uses local detection and correction of BPs. Hence, when a partial stable marriage is already obtained, the algorithm may keep the matched pairs. This can be a huge advantage in the case of a distributed storage of the matching, when only some sites have been hit by failures. But the counterpart is a rather high worst case complexity. This work has been published in [LMB+17] and is submitted to a journal.

Since the worst case complexity was far from the theoretical lower bound of $\Omega(n^2/\log n)$ moves, our next step was to look for an algorithm with a better complexity. As local correction appeared to be costly (due to our first solution) costly, we tried to implement a global correction, by triggering a reset each time a problem was detected locally. For that, we had to modify the original local detection conditions of [APSVD94], while still using the same global reset scheme. The resulting algorithm (Chapter 5) solves SMP in $\Theta(n^2)$ moves and rounds in the link register model. A preliminary version of this work has been published in [BBB+18]. The full version is in preparation for submission.

Finally, we studied in Chapter 6 some classical variants of SMP and propose the minor changes to both algorithms to obtain a self-stabilizing algorithm that solves all these variants.

II - Perspectives

We now discuss some perspectives related to the results of the thesis.

Complexity. In [GNOR15], it has been proven that the communication complexity of SMP is $\Omega(n^2)$ bits inducing a lower bound of $\Omega(n^2/\log n)$ moves in our model (see Section III.1 for explanations). In this work, the best algorithm (`CompAlg`) has a complexity of $\Theta(n^2)$ moves, letting a gap open between the best know solution and the lower bound. A natural issue is to close the gap. That can be done either by finding a still more efficient algorithm in terms of moves or by increasing the lower bound.

In [GNOR15] the lower bound proof uses a particular case of preference lists that can be represented using only $O(n^2)$ bits altogether. However in a general case, the preference lists require $\Theta(n^2 \log n)$ bits to be represented in the two-party setting assumed in the communication complexity analysis (the preference list of each SMP participant is coded by $\Theta(n \log n)$ bits). We believe it is possible to provide a proof for a more general case of preferences represented by $\Omega(n^2 \log n)$ bits, resulting in $\Omega(n^2)$ lower bound of moves in our model. Nevertheless, this is still an open question.

Reset Algorithm. Numerous reset algorithms were known before this work, but no one was precisely designed for the link register model with an unfair scheduler. The reset algorithm that we proposed is adapted to this model and has a move complexity of $O(n \cdot 6^p)$ (where p is the depth of the rooted spanning tree on which it runs). Hence, if the depth of the tree is known and does not depend on any system parameter (like our tree of depth 2), it has a linear complexity. Otherwise, the complexity is exponential. Natural issues are the questions of the tightness of this bound and the possibility of a better solution. For the tightness of our algorithm's bound, we conjecture that it cannot be improved by other analysis techniques. The reason for the latter is the following.

There are possible cycles of moves that a children of a node in a tree can make from any configuration. These cycles are due to faults that may cause a node to change for *initiate* status several times, before its parent makes a move, leading to the exponential move complexity. These cycles are not broken until the *initiate* wave comes to the root and a *freeze* wave is launched. Hence, in order to get a better algorithm, an idea would be to avoid such cycles by using more restricted guards.

Self-stabilizing Transformer. In Chapter 5, we have presented the result of the composition of `TreeAlg`, `ResetAlg` and `Async-GSA` as a whole algorithm. As `TreeAlg` and `ResetAlg` are largely independent of `Async-GSA`, one can ask whether such a composition is applicable to other algorithms than `Async-GSA`. Underlying this approach, there can be proven a general composition theorem, stating that if an initialized algorithm satisfies some properties, related to local checkability, and solves a graph problem Prob, then it can be automatically (syntactically) transformed into a self-stabilizing version (self-stabilizing for Prob). Such a composition theorem would establish the existence of what can be called a self-stabilizing transformer, transforming automatically an algorithm with initialization into a self-stabilizing algorithm solving the same problem. There already exist several such transformers in the literature ([APSV91, APSVD94],

...), and we base our second solution on them. The originality of the one that we suggest is the consideration of the *unfair demon*, as well as the fact that a tight bound on the *moves* complexity for the transformed algorithm could be obtained (as a function of the move complexity of the initialized algorithm). In the context of this thesis, a privileged application domain of such a syntactic transformer would be the different variants of SMP that have been studied, and received solutions, for some of them, in a centralized setting. Most of them seem to satisfy the local checkability property and thus be liable for the general transformation. An issue appears with variants that do not always have a solution. We discuss them in the next paragraph.

Problematic Variants. Beside the variants that we listed in Chapter 6, there are other variants, for which getting a self-stabilizing solution seems unattainable, with the techniques that we proposed. We list in the following variants that seem difficult to solve by a simple adaptation of any of the two types of solutions we gave (two phases and composition with a reset), thus presenting open questions. These variants are stable roommates, incomplete bipartite graph, 3-dimensional matching, strongly and super stable matching with indifference. They appear in the reference book [GI89] and are among the twelve research directions suggested by Knuth [Knu76]. We explain the general reasons why the solutions we developed cannot be applied to these variants.

Strongly and super stable matching with indifference. These two variants are described in Chapter 4, Sub-section II. The Figure 7.1 illustrate a strongly but not super stable marriage (because of the pair (*Zoe*, *Scott*) in red; bold edges and names represent the marriages).

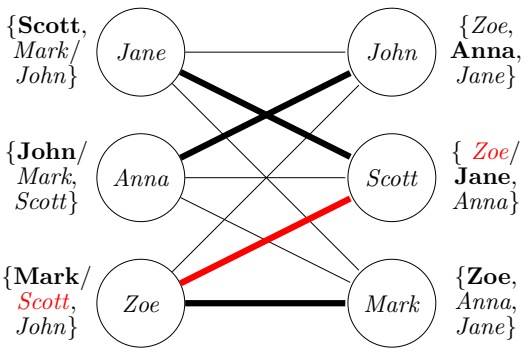


Figure 7.1: Strongly but not super stable marriage

Recall that, by [Irv94], stable marriage is not guaranteed in these cases (there is no super stable marriage in the system of Figure 7.1). Section II describes a simple self-stabilizing solution that does not detect the case lacking a stable marriage. Implementing such detection in a self-stabilizing manner is indeed not simple and previous existing solutions are not appropriate.

They are centralized, initialized and detect the lack of a stable marriage as follows. If a node exhausts all the possible partners in its preference list during an execution, it reaches a state s, encoding the fact it can conclude that no stable marriage exists [Irv94]. However, when self-stabilization is required (convergence from any initial configuration), such an approach is inappropriate. For example, if a node starts in state s, it can incorrectly conclude that no stable marriage exists. Hence, both our solutions do not extend to strongly and super stable matching with indifference (but, if only instances in which a stable matching exists are proposed, it seems feasible to extend them).

Amira10

Stable roommates problem. Contrary to the stable marriage, this problem is defined on a complete communication graph, where each node has a preference list over all other nodes. The problem consists to build a stable matching, which is a complete matching without blocking pairs. In this context, a pair (u, v) is a blocking pair iff $(u, v) \notin \mathcal{M}$ and u and v prefer each other to their actual matching, with no restriction over sets. Building a stable matching is not always possible, like in the following example (the matching is represented in black and the blocking pairs in red):

Id of node:	Preference list
1:	3, 2, 4
2:	1, 3, 4
3:	2, 1, 4
4:	1, 2, 3

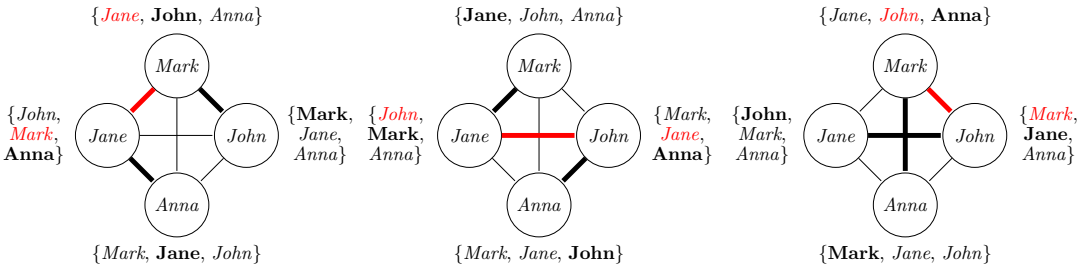


Figure 7.2: Stable roommates problem with no stable matching

In [GI89], a centralized algorithm is given to determine whether a given instance admits a stable matching, and if so finds one. Intuitively, this determination is done similarly to the technique used in [Irv94] for detecting the lack of stable marriage, by verifying that a node has exhausted all the possible partners in its preference list. As before, this approach is inappropriate for self-stabilization.

Moreover, the communication graph being complete, the two phases technique of Ackermann *et al.*, based on two distinct sets, is not appropriate either.

3-dimensional matching. This last variant has been proposed by Knuth in [Knu76] as a generalization of the stable marriage problem to three dimensions (*e.g.*, to the 3-gender stable matching problem with men, women and dogs). In [Alk88], Alkan defines the problem (participants are allowed to express preferences over all pairs they could possibly join), shows an example for which there is no solution and generalizes the result to k-dimensional graphs. Ng and Hirschberg

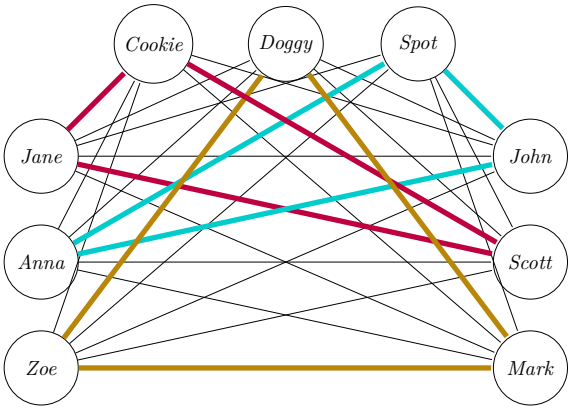


Figure 7.3: 3-D Matching

in [NH91] prove that this problem is NP-complete.

As for the stable roommates problem, the communication graph is not bipartite. Thus, the Ackermann *et al.*'s technique based on two distinct sets is not appropriate. Furthermore, there is no known distributed algorithm for this problem, *i.e.* the basic element in the composition is missing.

Incomplete bipartite graph. Several types of systems can be considered under this variant. In the main studied version (considered in [KPS09] for example), preferences are related to the graph topology, in the sense that two non-neighbors are unacceptable partners. In this case , even if the graph is not connected, a solution is feasible, but requires an important modification to the given solutions. We start by describing briefly the modifications to bring to the two phase algorithm of Chapter 4.

For ensuring correct transitions between phases, nodes have to know in which phases the opposite set nodes are. But, as a complete broadcast cannot be directly (in one step) performed if the graph is incomplete, one solution would be to combine the presented algorithm with a self-stabilizing rooted tree construction (*e.g.*, [AG90, AKM+07]) in each connected component. This is for propagating an information along the tree (upward to and downward from the root), regarding the types of phases' states (for each of the two opposite sets). In this way every node could have (a possibly outdated) information about the type of phases in the other set. But the changes in phases cannot be propagated immediately, causing a different behavior comparing to the original algorithm variant. Neither the proof nor the complexity analysis of the basic algorithm apply to this case.

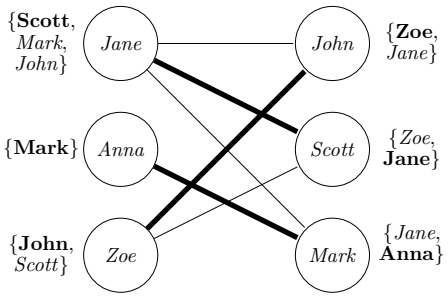


Figure 7.4: Stable matching in an incomplete bipartite graph

Concerning the solution based on local checkability and reset, the main issue is the tree. Indeed, since the graph is not complete, in the worst case, the tree is of depth n, increasing considerably the global complexity. Any known spanning tree construction can be used to obtain this tree.

Bibliography

[ACD+17] K. Altisen, A. Cournier, S. Devismes, A. Durand, and F. Petit. Self-stabilizing leader election in polynomial steps. *Information and Computation*, 2017. Preliminary version in SSS 2014. (Cited on page 3.)

[AG90] A. Arora and M. Gouda. Distributed Reset. In *Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, 1990. (Cited on pages 11 and 133.)

[AGL10] N. Amira, R. Giladi, and Z. Lotker. Distributed Weighted Stable Marriage Problem. In *International Colloquium on Structural Information and Communication Complexity (SIROCCO)*, 2010. (Cited on page 8.)

[AGM+11] H. Ackermann, P. W. Goldberg, V. S. Mirrokni, H. Röglin, and B. Vöcking. Uncoordinated Two-Sided Matching Markets. *SIAM Journal on Computing*, 2011. (Cited on pages 4, 10, 24 and 66.)

[AH93] E. Anagnostou and V. Hadzilacos. Tolerating transient and permanent failures (extended abstract). In *International Workshop on Distributed Algorithms (WDAG)*, 1993. (Cited on page 89.)

[AKM+07] B. Awerbuch, S. Kutten, Y. Mansour, B. Patt-Shamir, and G. Varghese. A Time-Optimal Self-Stabilizing Synchronizer Using A Phase Clock. *IEEE Transactions On Dependable And Secure Computing*, 2007. (Cited on pages 25 and 133.)

[AKY90] Y. Afek, S. Kutten, and M. Yung. Memory-efficient self stabilizing protocols for general networks. In *International Workshop on Distributed Algorithms (WDAG)*, 1990. (Cited on pages 4, 10, 11, 68, 69 and 70.)

[AKY97] Y. Afek, S. Kutten, and M. Yung. The Local Detection Paradigm and Its Applications to Self-stabilization. *Theoretical Computer Science*, 1997. (Cited on page 69.)

[Alk88] A. Alkan. Nonexistence of stable threesome matchings. *Mathematical Social Sciences*, 1988. (Cited on page 132.)

[Alk99] A. Alkan. On the properties of stable many-to-many matchings under responsive preferences. In *Current Trends in Economics: Theory and Applications*, pages 29–39. Springer Berlin Heidelberg, 1999. (Cited on page 125.)

[AO94] B. Awerbuch and R. Ostrovsky. Memory-Efficient and Self-Stabilizing Network RESET (Extended Abstract). In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 1994. (Cited on pages 5 and 89.)

[APR05] A. Abdulkadiroğlu, P. A. Pathak, and A. E. Roth. The New York City High School Match. *American Economic Review*, 2005. (Cited on page 8.)

[APRS05] A. Abdulkadiroglu, P. A. Pathak, A. E. Roth, and T. Sönmez. The Boston Public School Match. *American Economic Review*, 2005. (Cited on page 8.)

[APSV91] B. Awerbuch, B. Patt-Shamir, and G. Varghese. Self-stabilization by local checking and correction. In *Annual Symposium of Foundations of Computer Science (FOCS)*, 1991. (Cited on pages 11 and 130.)

[APSVD94] B. Awerbuch, B. Patt-Shamir, G. Varghese, and S. Dolev. Self-stabilization by local checking and global reset. In *International Workshop on Distributed Algorithms (WDAG)*, 1994. (Cited on pages 4, 11, 68, 69, 70, 82, 89, 118, 129 and 130.)

[BA16] M. W. Baidas and M. M. Afghah. Energy-efficient partner selection in cooperative wireless networks: a matching-theoretic approach. *International Journal of Communication Systems*, 2016. (Cited on page 3.)

[BAM03] V. Bansal, A. Agrawal, and V. S. Malhotra. Stable Marriages with Multiple Partners: Efficient Search for an Optimal Solution. In *International Colloquium on Automata, Languages, and Programming (ICALP)*, 2003. (Cited on page 125.)

[BAM07] V. Bansal, A. Agrawal, and V. S. Malhotra. Polynomial time algorithm for an optimal stable assignment with multiple partners. *Theoretical Computer Science*, 2007. (Cited on page 125.)

[BBB+18] J. Beauquier, T. Bernard, J. Burman, S. Kutten, and M. Laveau. Brief announcement: Time efficient self-stabilizing stable marriage. In *Stabilization, Safety, and Security of Distributed Systems (SSS)*, 2018. (Cited on pages 5, 68 and 129.)

[BDPV07] A. Bui, A. K. Datta, F. Petit, and V. Villain. Snap-stabilization and PIF in tree networks. *Distributed Computing*, 2007. (Cited on page 19.)

[Bir17] P. Biró. Applications of Matching Models under Preferences. In *Trends in Computational Social Choice*, chapter 18. AI Access, 2017. (Cited on page 7.)

[BLAK14] R. Bai, J. Li, J. A. D. Atkin, and G. Kendall. A novel approach to independent taxi scheduling problem based on stable matching. *Journal of the Operational Research Society*, 2014. (Cited on page 7.)

[BM05] I. Brito and P. Meseguer. Distributed stable marriage problem. In *Workshop on Distributed Constraint Reasoning at IJCAI*, 2005. (Cited on pages 4, 9 and 21.)

[BM06] I. Brito and P. Meseguer. Distributed Stable Matching Problems with Ties and Incomplete Lists. In *Principles and Practice of Constraint Programming (CP)*, 2006. (Cited on page 121.)

[BPV04] C. Boulinier, F. Petit, and V. Villain. When graph theory helps self-stabilization. In *Symposium on Principles of Distributed Computing (PODC)*, 2004. (Cited on page 25.)

[CCM19] K. Cechlárová, Á. Cseh, and D. Manlove. Selected open problems in Matching Under Preferences. *Bulletin of the EATCS*, 2019. (Cited on page 7.)

[Cec17] K. Cechlárová. School Placement of Trainee Teachers: Theory and Practice. In *Trends in Computational Social Choice*, chapter 19. AI Access, 2017. (Cited on page 7.)

[Che19] J. Chen. Computational Complexity of Stable Marriage and Stable Roommates and Their Variants. Technical report, University of Warsaw, 2019. (Cited on page 7.)

[CL85] K. Mani Chandy and Leslie Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transaction on Computer Systems*, 1985. (Cited on page 10.)

[CL10] J.-H. Chou and C.-J. Lu. Communication requirements for stable marriages. In *International Conference on Algorithms and Complexity (CIAC)*. Springer Berlin Heidelberg, 2010. (Cited on page 8.)

[Cse17] Á. Cseh. Popular Matchings. In *Trends in Computational Social Choice*, chapter 6. AI Access, 2017. (Cited on page 7.)

[Dag] Schloss dagstuhl – leibniz-zentrum für informatik. matching under preferences: Theory and practice, web document available at https://www.dagstuhl.de/en/program/calendar/semhp/?semnr=20301 (accessed 24/07/19). (Cited on page 7.)

[DES16] J. Doerner, D. Evans, and A. Shelat. Secure Stable Matching at Scale. In *Computer and Communications Security (CCS)*, 2016. (Cited on page 8.)

[DH95] S. Dolev and T. Herman. SuperStabilizing Protocols for Dynamic Distributed Systems. In *Symposium on Principles of Distributed Computing (PODC)*, 1995. (Cited on page 89.)

[Dij74] E. W. Dijkstra. Self-stabilizing Systems in Spite of Distributed Control. *Communication of the ACM*, 1974. (Cited on pages 3, 9, 17 and 18.)

[Dij86] E. W. Dijkstra. A belated proof of self-stabilization. *Distributed Computing*, 1986. (Cited on page 9.)

[DIM93] S. Dolev, A. Israeli, and S. Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 1993. Preliminary version published in PODC' 90. (Cited on pages 3, 11 and 17.)

[DIM97] S. Dolev, A. Israeli, and S. Moran. Uniform Dynamic Self-Stabilizing Leader Election. *IEEE Transactions on Parallel Distributed Systems*, 1997. Preliminary version published in WDAG' 91. (Cited on page 19.)

[DJ16] S. Devismes and C. Johnen. Silent self-stabilizing BFS tree algorithms revisited. *Journal of Parallel and Distributed Computing*, 2016. (Cited on page 3.)

[DJ19] S. Devismes and C. Johnen. Self-Stabilizing Distributed Cooperative Reset. In *International Conference on Distributed Computing Systems (ICDCS)*, 2019. (Cited on page 5.)

[DLV11a] A. K. Datta, L. L. Larmore, and P. Vemula. An O(n)-time self-stabilizing leader election algorithm. *Journal of Parallel and Distributed Computing*, 2011. (Cited on page 3.)

[DLV11b] A. K. Datta, L. L. Larmore, and P. Vemula. Self-stabilizing leader election in optimal space under an arbitrary scheduler. *Theoretical Computer Science*, 2011. (Cited on page 3.)

[EO16] F. Echenique and J. Oviedo. A Theory of Stability in Many-to-Many Matching Markets. *Theoretical Economics*, 2016. (Cited on page 125.)

[FGS10] A. Fagebaume, D. Gale, and M. Sotomayor. A note on the multiple partners assignment game. *Journal of Mathematical Economics*, 2010. (Cited on page 125.)

[FKPS10] P. Floréen, P. Kaski, V. Polishchuk, and J. Suomela. Almost Stable Matchings by Truncating the Gale-Shapley Algorithm. *Algorithmica*, 2010. (Cited on page 8.)

[GHIJ14] C. Glacet, N. Hanusse, D. Ilcinkas, and C. Johnen. Disconnected Components Detection and Rooted Shortest-Path Tree Maintenance in Networks. In *Stabilization, Safety, and Security of Distributed Systems (SSS)*, 2014. (Cited on page 3.)

[GHIJ19] C. Glacet, N. Hanusse, D. Ilcinkas, and C. Johnen. Disconnected Components Detection and Rooted Shortest-Path Tree Maintenance in Networks. *Journal of Parallel and Distributed Computing*, 2019. (Cited on page 3.)

[Gho14] S. Ghosh. *Distributed Systems: An Algorithmic Approach.* Chapman & Hall/CRC, 2nd edition, 2014. (Cited on page 17.)

[GI89] D. Gusfield and R. W. Irving. *The Stable Marriage Problem: Structure and Algorithms.* Foundations of computing series. MIT Press, 1989. (Cited on pages 2, 8, 47, 119, 122, 124, 131 and 132.)

[GM91] M. G. Gouda and N. J. Multari. Stabilizing Communication Protocols. *IEEE Transactions on Computers*, 1991. (Cited on pages 5, 40 and 89.)

[GNOR15] Y. A. Gonczarowski, N. Nisan, R. Ostrovsky, and W. Rosenbaum. A Stable Marriage Requires Communication. In *Symposium on Discrete Algorithms (SODA)*, 2015. (Cited on pages 8, 66, 73 and 130.)

[Gol06] P. Golle. A Private Stable Matching Algorithm. In *International Conference on Financial Cryptography and Data Security (FC)*, 2006. (Cited on page 8.)

[GS62] D. Gale and L. S. Shapley. College Admissions and the Stability of Marriage. *The American Mathematical Monthly*, 1962. (Cited on pages 1, 7, 13, 15, 68, 73 and 124.)

[GS85] D. Gale and M. Sotomayor. Ms. machiavelli and the stable matching problem. *The American Mathematical Monthly*, 1985. (Cited on page 8.)

[HC92] S.-T. Huang and N.-S. Chen. A self-stabilizing algorithm for constructing breadth-first trees. *Information Processing Letters*, 1992. (Cited on page 3.)

[IMM99] K. Iwama, S. Miyazaki, and Y. Morita. Stable Marriage with Incomplete Lists and Ties (Extended Abstract). In *International Colloquium on Automata, Languages, and Programming (ICALP)*, 1999. (Cited on page 121.)

[IMO09] R. W. Irving, D. F. Manlove, and G. O'Malley. Stable marriage with ties and bounded length preference lists. In *Journal of Discrete Algorithms*, 2009. (Cited on page 121.)

[Irv94] R. W. Irving. Stable marriage and indifference. *Discrete Applied Mathematics*, 1994. (Cited on pages 7, 121, 122, 131 and 132.)

[KA98] S. S. Kulkarni and A. Arora. Multitolerance in Distributed Reset. Technical report, The Ohio State University, 1998. (Cited on pages 5 and 89.)

[KBW16] M. Kuemmel, F. Busch, and D. Z.W. Wang. Taxi Dispatching and Stable Marriage. *Procedia Computer Science*, 2016. (Cited on page 7.)

[KK15] A. Korman and T. Kutten, S.and Masuzawa. Fast and compact self-stabilizing verification, computation, and fault detection of an MST. *Distributed Computing*, 2015. (Cited on page 69.)

[KL14] G. Kim and W. Lee. Stable Matching with Ties for Cloud-assisted Smart TV Services. In *International Conference on Consumer Electronics (ICCE)*, 2014. (Cited on page 9.)

[KMR16] B. Klaus, D. F. Manlove, and F. Rossi. Matching under Preferences. In *Handbook of Computational Social Choice*. Cambridge University Press, 2016. (Cited on page 7.)

[Knu76] D. E. Knuth. *Mariages stables et leurs relations avec d'autres problemes combinatoires*. Les Presses de l'Université de Montréal, 1976. English translation in *Stable Marriage and its Relation to Other Combinatorial Problems*, volume 10 of CRM Proceedings and Lecture Notes, American Mathematical Society, 1997. (Cited on pages 2, 4, 7, 10, 22, 47, 68, 131 and 132.)

[KP90] S. Katz and K. Perry. Self-stabilizing Extensions for Message-passing Systems. In *Symposium on Principles of Distributed Computing (PODC)*, 1990. (Cited on page 10.)

[KPS09] A. Kipnis and B. Patt-Shamir. A Note on Distributed Stable Matching. In *International Conference on Distributed Computing Systems (ICDCS)*, 2009. (Cited on pages 8 and 133.)

[KW16] P. Khanchandani and R. Wattenhofer. Distributed Stable Matching with Similar Preference Lists. In *Principles of Distributed Systems (OPODIS)*, 2016. (Cited on page 8.)

[LMB+17] M. Laveau, G. Manoussakis, J. Beauquier, T. Bernard, J. Burman, J. Cohen, and L. Pilard. Self-stabilizing Distributed Stable Marriage. In *Stabilization, Safety, and Security of Distributed Systems (SSS)*, 2017. (Cited on pages 4, 5, 21, 66 and 129.)

[LT89] N. A. Lynch and M. R. Tuttle. An Introduction to Input/Output Automata. *CWI Quarterly*, 1989. (Cited on page 11.)

[Man13] D. Manlove. *Algorithmics Of Matching Under Preferences*. Theoretical computer science. World Scientific Publishing, 2013. (Cited on pages 1, 2 and 7.)

[MAT] International work- shop on matching under preferences (match-up), web document available at http://www.optimalmatching.com/MATCHUP (accessed 24/07/19). (Cited on page 7.)

[Mat07a] F. Mathieu. Self-stabilization in preference-based systems. *Peer-to-Peer Networking and Applications*, 2007. (Cited on page 10.)

[Mat07b] F. Mathieu. Upper bounds for stabilization in acyclic preference-based systems. In *Stabilization, Safety, and Security of Distributed Systems (SSS)*, 2007. (Cited on page 10.)

[Mat09] F. Mathieu. *Autour du pair-à-pair : distribution de contenus, réseaux à préférences acycliques*. Habilitation à diriger des recherches, Université Pierre et Marie Curie - Paris VI, 2009. (Cited on page 10.)

[MMNO04] R. Martínez, J. Massó, A. Neme, and J. Oviedo. An algorithm to compute the full set of many-to-many stable matchings. *Mathematical social sciences*, 2004. (Cited on page 125.)

[MS15] B. M. Maggs and R. K. Sitaraman. Algorithmic Nuggets in Content Delivery. *Computer Communication Review*, 2015. (Cited on page 9.)

[NH90] C. Ng and D. S. Hirschberg. Lower Bounds for the Stable Marriage Problem and Its Variants. *SIAM Journal on Computing*, 1990. (Cited on page 7.)

[NH91] C. Ng and D. S. Hirschberg. Three-dimensional Stable Matching Problems. *Journal on Discrete Mathematics*, 1991. (Cited on page 133.)

[NRM] The match, national resident matching program, web document available at http://www.nrmp.org/matching-algorithm (accessed 24/07/19). (Cited on page 8.)

[OR15] R. Ostrovsky and W. Rosenbaum. Fast Distributed Almost Stable Matchings. In *Principles of Distributed Computing (PODC)*, 2015. (Cited on pages 3, 4 and 8.)

[RS90] A. E. Roth and M. A. O. Sotomayor. *Two-sided Matching: A Study in Game-Theoretic Modeling and Analysis*. Cambridge University Press, 1990. (Cited on pages 2, 7, 24 and 125.)

[RVV90] A. Roth and J. H. Vande Vate. Random Paths to Stability in Two-Sided Matching. *Econometrica*, 1990. (Cited on page 10.)

[Seg07] I. Segal. The communication requirements of social choice rules and supporting budget sets. *Journal of Economic Theory*, 2007. (Cited on pages 7 and 8.)

[Sot99] M. Sotomayor. Three remarks on the many-to-many stable matching problem. *Mathematical Social Sciences*, 1999. (Cited on page 125.)

[TST99] C.-P. Teo, J. Sethuraman, and W.-P. Tan. Gale-shapley stable marriage problem revisited: Strategic issues and applications (extended abstract). In *Integer Programming and Combinatorial Optimization (IPCO)*, 1999. (Cited on page 8.)

[Var93] G. Varghese. *Self-stabilization by Local Checking and Correction*. PhD thesis, Massachusetts Institute of Technology, 1993. (Cited on pages 11 and 89.)

[XL11a] H. Xu and B. Li. Egalitarian Stable Matching for VM Migration in Cloud Computing. In *Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2011. (Cited on page 9.)

[XL11b] H. Xu and B. Li. Seen as stable marriages. In *International Conference on Computer Communications (INFOCOM)*, 2011. (Cited on page 9.)

[YAB19] F. Yucel, K. Akkaya, and B. Bulut. Efficient and privacy preserving supplier matching for electric vehicle charging. *Ad Hoc Networks*, 2019. (Cited on page 9.)

[Yao79] A. C.-C. Yao. Some Complexity Questions Related to Distributive Computing (Preliminary Report). In *Symposium on Theory of Computing (STOC)*, 1979. (Cited on page 73.)

Aknowledgement

Writing a thesis is a tough process. This is why I would like to thank all those who have helped me in one way or another in this task. Even more so in this complicated context of the pandemic, each support has been precious.

First and foremost, I would like to thank my supervisors, Joffroy Beauquier, Janna Burman and Thibault Bernard, for their unfailing support and their unvaluable advices during these years. They guided my with patience and taught me the methodology of many things, from writing research papers, to presenting my ideas and results. Thanks to them, I learned to always aim better, without being satisfied with what has already been achieved. I will try in the future to apply their teachings and to produce quality research.

I would like to express my gratitude to Colette Johnen and Volker Tureau for reviewing my manuscript and for their relevant and insightful reviews. I am also thankful to the others members of my thesis committee, Johanne Cohen, Hugues Fauconnier and Sébastien Tixeuil, for attending my defense. I would also like to thank Johanne Cohen, Laurence Pilard and George Manoussakis, the co-authors of my first paper, for our discussions about the stable marriage and the work we produced. I am extremely grateful towards Shay Kutten for hosting me twice at the Technion during my thesis and for our inspiring discussions.
At the beginning of my Master, I wasn't sure what I wanted to do. Thus, I am very thankful to Hervé Bredin and Claude Barras for welcoming me at the LIMSI and for introducing me to research during my first internship.

Also, I would like to thank my family and especially my parents for their unfailing support and their faith in me and my aunt and uncle in particular for their logistical support.
I want to express a special thank to Fabien Dufoulon with whom I shared our advisors but also an office and a coffee machine. We spent (and we still spend) a lot of great time, from lively political debates to precise technical discussions, from incredible games to quiet evenings. I want to thank Chuan for taking part of these times and for putting up with our endless debates.
Furthermore, I would like to thank my colleagues (and friends) Amal, Aygul, Antoine, Ian, Yushan, Oguz, Laercio, Timothée, David, Pierre, George and Hay and also my friends "from outside", Marie B., Hélène, Simon, Lucille, Ye, Margaux, Yohann and Marie S..

Finally, I would also like to thank all the colleagues of the LRI and especially the administration team as well the technical support team, for their availability, their kindness and their help.

université PARIS-SACLAY

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

Titre : Mariage Stable Asynchrone et Auto-stabilisant

Mots clés : Algorithmes Distribués, Modèles Asynchrones, Auto-stabilisation, Mariage Stable, Complexité en Moves, Démon inéquitable, Confidentialité

Résumé : Le *Problème du Mariage Stable* (SMP) est un problème d'appariement où les participants ont des préférences à propos de leurs partenaires potentiels. L'objectif est de trouver un appariement optimal (stable dans un sens) au regard des préférences. Ce type d'appariement a de très nombreuses applications comme les affectations d'étudiants à des universités (APB ou ParcourSup), celles des internes en médecine aux hôpitaux, les choix des donneurs pour les patients en attente d'organe, la mise en rapport des taxis et de leurs clients ou encore la diffusion de contenu sur Internet. Certaines de ces applications peuvent être traitées de manière centralisée tandis que d'autres, de par leur nature distribuée et la complexité de leurs données, nécessitent un traitement différent.

Dans ce contexte, nous proposons deux solutions distribuées *auto-stabilisantes* ($i.e.$ qui tolèrent les défaillances transitoires (ou de courte durée) de n'importe quels noeuds). Pour ces deux algorithmes, les exécutions se déroulent par pas atomiques et un démon (*démon distribué inéquitable*) exprime la notion d'asynchronisme. Avec ce démon, le temps de stabilisation peut être borné en terme de *moves* (pas locaux). Cette mesure de complexité permet d'évaluer avec précision la puissance de calcul nécessaire ou l'énergie dissipée par les exécutions de l'algorithme.

Le premier algorithme, basé sur la méthode centralisée de Ackermann *et al.* (SICOMP' 2011), résout le SMP en $O(n^4)$ moves.

Le point de départ du deuxième algorithme est le schéma de détection locale/correction globale de Awerbuch *et al.* (DA' 1994). Malheureusement, la définition de la vérifiabilité locale de DA' 1994 ne s'applique pas à notre cas (en particulier en raison du démon inéquitable). Nous proposons donc une nouvelle définition. De plus, nous concevons un algorithme de réinitialisation (reset) asynchrone, distribué et auto-stabilisant. L'algorithme résultant résout le SMP en $\Theta(n^2)$ moves.

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: Asynchronous Self-stabilizing Stable Marriage

Keywords: Distributed Algorithms, Asynchronous Model, Self-stabilization, Stable Marriage, Move Complexity, Unfair Daemon, Privacy

Abstract: The *Stable Marriage Problem* (SMP) is a matching problem where participants have preferences over their potential partners. The objective is to find a matching that is optimal (stable in certain sens) with regard to these preferences. This type of matching has a lot of widely used applications such as the assignment of children to schools, interns to hospitals, kidney transplant patients to donors, as well as taxi scheduling or content delivery on the Internet. Some applications can be solved in a centralized way while others, due to their distributed nature and their complex data, need a different treatment.

In order to handle this challenge, we provide two distributed *self-stabilizing* solutions ($i.e.$, that tolerate transient (or short-lived) failures ($e.g.$, memory or message corruptions) of any nodes). The privacy of the preference lists is guaranteed by the two proposed algorithms: lists are not shared, only some binary queries and responses are transmitted. For both algorithms, executions proceed in atomic steps and a daemon (*distributed unfair daemon*) conveys the notion of asynchrony. Under this daemon, the *stabilization time* can be bounded in term of *moves* (local computations). This complexity metrics allows to evaluate the necessary computational power or the energy consumption of the algorithm's executions.

The first algorithm, based on the centralized method of Ackermann *et al.* (SICOMP' 2011), solves the problem in $O(n^4)$ moves.

The starting point of the second algorithm is the local detection/global correction scheme of Awerbuch *et al.* (DA' 1994). Unfortunately, local checkability definition of DA '1994 does not apply to our case (in particular due to the unfair daemon). Consequently, we propose a new definition. Furthermore, we design a distributed self-stabilizing asynchronous reset algorithm. Using it, the resulting composed algorithm solves SMP in $\Theta(n^2)$ moves in a self-stabilizing way.

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