HAL
open science

Polynomial Silent Self-Stabilizing p-Star Decomposition

Mohammed Haddad, Colette Johnen, Sven Köhler

To cite this version:

Mohammed Haddad, Colette Johnen, Sven Köhler. Polynomial Silent Self-Stabilizing p-Star Decomposition. 2016. hal-01514323

HAL Id: hal-01514323

https://hal.science/hal-01514323v1

Preprint submitted on 26 Apr 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Polynomial Silent Self-Stabilizing p-Star Decomposition[1]

Mohammed Haddad

Université Claude Bernard Lyon 1, Villeurbanne, France

Colette Johnen

University of Bordeaux, Talence Cedex, France

Sven Köhler

University of Freiburg, Freiburg im Breisgau, Germany

Abstract

We present a silent self-stabilizing distributed algorithm computing a maximal p-star decomposition of the underlying communication network. Under the unfair distributed scheduler, the most general scheduler model, the algorithm converges in at most $12\Delta m + \mathcal{O}(m+n)$ moves, where m is the number of edges, n is the number of nodes, and Δ is the maximum node degree. Regarding the move complexity, our algorithm outperforms the previously known best algorithm by a factor of Δ. While the round complexity for the previous algorithm was unknown, we show a $5\left\lfloor\frac{n}{p+1}\right\rfloor + 5$ bound for our algorithm.

1. Introduction

Fault-tolerance is among the most important requirements for distributed systems. Self-stabilization is a fault-tolerance technique that deals with transient faults. It was first introduced by Dijkstra [1]. Starting in an arbitrary configuration, a self-stabilizing distributed system converges to a legitimate configuration in finite time by itself, i.e., without any external intervention. This makes self-stabilization an elegant approach for non-masking fault-tolerance [2].

An H-decomposition of a graph G subdivides a graph into disjoint components which which are isomorphic to H. A p-star is a complete bipartite graph $K_{1,p}$ with one center node and p leaves. One of the famous and well studied graph decompositions in literature is star decomposition [3, 4, 5]. A decomposition of a graph into stars is a way of expressing the graph as the union of disjoint stars [6]. The problem of star decomposition has several applications including scientific computing, scheduling, load balancing and parallel computing [7], important nodes detection in social networks [8]. Decomposing a graph into stars is also used in parallel computing and programming. This decomposition offers similar feature as Master-Slave paradigm, used in grids [9] and P2P infrastructures [10].

[1]This study has been carried out with financial support from the French State, managed by the French National Research Agency (ANR) in the frame of the "Investments for the future" Programme IdEx Bordeaux - CPU (ANR-10-IDEX-03-02) and by the Sustainability Center Freiburg, Germany. The Sustainability Center Freiburg is a cooperation of the Fraunhofer Society and the University of Freiburg and is supported by grants from the Baden-Württemberg Ministry of Economics and the Baden-Württemberg Ministry of Science, Research and the Arts.

Email addresses: `mohammed.haddad@univ-lyon1.fr` (Mohammed Haddad), `johnen@labri.fr` (Colette Johnen), `koehlers@informatik.uni-freiburg.de` (Sven Köhler)

Related Work

Self-stabilizing algorithms have been proposed for a large variety of graph theoretical problems such as finding minimal dominating sets [11], maximal matchings [12, 13], independent sets [14], spanning trees [15], *etc.* The graph decomposition problem is defined on a graph $G = (V, E)$, where V is the set of nodes and E is the set of edges, such that the graph G is decomposed into smaller components having specific properties. These properties are often defined on the size of the partitions (clusters), on their shape (subgraphs) or both (patterns). Graph partitioning into clusters was considered by [16] and [17]. In [18], authors considered a particular graph partitioning problem that consists of decomposing the graph into partitions of order k. [18] considered a particular graph decomposition problem that consisted in partitioning a graph with k^2 nodes into k partitions of order k. The proposed algorithm relies on self-stabilizing spanning tree construction and converges within $3(h + 1)$ steps where h is the height of the spanning tree. Furthermore, [16], [17] and [19] focused on decomposing the graph into clusters while [20] considered decomposition of graphs into triangles. Other self-stabilizing algorithms were proposed for graph colorings [21, 22] that can be considered as decompositions into independent sets.

Observe that the maximal node-disjoint p-star decomposition problem when restricted to $p = 1$ is equivalent to maximal matching problem in graphs. Thus the general problem when $p \geq 1$ is NP-complete since generalized matching and general graph factor problems were proved to be NP-complete in [23] and [24] respectively. The maximal matching problem has received much interest due the abundant number of applications in fields as diverse as transversal theory, assignment problems, network flows, and scheduling. Many studies have addressed this problem even in the field of self-stabilization [12, 25, 13, 26]. The best known move complexity for maximal matching problem is $\mathcal{O}(m)$ and was obtained by [27].

The first self-stabilizing algorithm for the p-star decomposition problem was proposed in [28]. It finds a maximal decomposition into node-disjoint p-stars. The decomposition is maximal in the sense that the nodes not part of any p-star cannot form a p-star. However, the algorithm proposed in [28] always converges to a unique legitimate configuration according to the input graph and does not guarantee a polynomial move complexity. An improvement was proposed in [29] where authors dealt with the uniqueness of legitimate state and proved their algorithm to converge within $O(\Delta^2 m)$ moves under the unfair distributed scheduler where m is the number of edges and Δ is maximum node degree in the graph. A bound on the round complexity of the algorithm was not given.

Our Results

In this paper, we improve the move complexity of the previous algorithm to $12\Delta m + \mathcal{O}(m+n)$ and prove an $\mathcal{O}(n)$ bound on the round complexity. The algorithm is proven correct and analyzed under the unfair distributed scheduler, the most powerful adversary. The algorithm does not converge to a unique legitimate configuration. In fact, there is a legitimate configuration for any valid maximal p-star decomposition.

The paper is organized as follows : Computation model is defined in Section 2, then details of the p-star decomposition algorithm are presented in Section 3. Proofs of correctness and convergence are given in Sections 4 and 5, respectively.

2. Model of Computation

We model the distributed system as a simple undirected graph $G = (V, E)$, where V is a set of nodes, and E is a set of edges representing the communication links. We denote by $N(v)$ the set of all neighbors of v in G, i.e., $N(v)$ contains all nodes that can communicate with v. We call $N(v)$ the *open neighborhood* and denote the *closed neighborhood* by $N[v] = N(v) \cup \{v\}$.

A distributed algorithm defines a set of *shared variables* and a set of *rules* for each node. In this paper, we only discuss *uniform algorithms*, i.e., each nodes has the same set of variables and executes the same set of rules. The (ordered) tuple of the values assigned to the variables of a node is called the *local state*. The (ordered) tuple of all local states constitutes the current *configuration* of the distributed system. Each rule is given in the form $guard(v) \rightarrow action$, where $guard(v)$ is a Boolean predicate over the variables within the closed neighborhood of node v. The action may read all shared variables within the closed neighborhood

but may modify only the variables of v itself. We say that a rule is *enabled*, if its guard evaluates to true. We say that a node is *enabled*, it any of its rules is enabled. For the algorithm given in this paper, at most one rule per node is enabled at a time.

Executions of a distributed algorithm are represented by a sequence of configurations $e = \langle c_0, c_1, c_2, \ldots \rangle$, where c_0 is called the initial configuration. Executions are organized in *steps*. The i-th step, where c_i is reached from c_{i-1}, denoted by $c_{i-1} \to c_i$, consists of three phases. In the first phase, an adversarial *scheduler* selects a non-empty set S_i of nodes that are enabled in c_{i-1}. In the second phase, all selected nodes perform the action of an enabled rule and compute their new local state. The computed local state is then made visible to neighboring nodes in the third phase. This model is called *composite atomicity*. If a node was selected, then we say that the node has made a *move* during that step. The sequence $\langle S_1, S_2, S_3, \ldots \rangle$ is called the *schedule* of e.

Three scheduler models are commonly discussed in the literature. The *central scheduler* selects exactly one enabled node per step. The *synchronous scheduler* selects all enabled nodes in each step. The most general model is the *distributed scheduler*, where the adversary may chose any non-empty of enabled nodes. A scheduler may satisfy certain fairness properties. We say that a scheduler is *fair*, if it selects any continuously enabled process after a finite number of steps. An *unfair* scheduler may indefinitely delay the move of a node as long as there are other enabled nodes. The algorithm given in this paper is proven correct and analyzed under the most general model, i.e., the unfair distributed scheduler.

A set CS of configurations is *closed* if any step from a configuration of CS reaches a configuration of CS. By extension, a predicate Pr on configurations is said *closed* if the set of configurations verifying Pr is closed.

A distributed algorithm is called *self-stabilizing*, if it satisfies the following requirements:

- Any execution reaches a legitimate configuration after a finite number of steps. (Convergence)
- The set of legitimate configurations is closed. (Closure)

The legitimacy of a configuration typically expresses that the output of the algorithm is correct.

The algorithm given in this paper is *silent*, i.e., any execution eventually reaches a *terminal configuration* in which no node is enabled. Note that for any silent self-stabilizing algorithm, terminal configurations are also legitimate. Its runtime complexity is analyzed with respect to two different metrics: the move and the round complexity. The *move complexity* is defined as $\sum_i |S_i|$. The *round complexity* is obtained by partitioning the execution e into *asynchronous rounds* as follows: The first round is the minimal prefix $\langle c_0, c_1, \ldots, c_x \rangle$ of e such that $V \setminus D_0 \subseteq \bigcup_{i=1}^{x} D_i \cup S_i$, where D_i is the set of nodes that are not enabled in c_i. The second round of e is the first round of the remaining suffix $\langle c_x, c_{x+1}, c_{x+2}, \ldots \rangle$, and so forth. One asynchronous round allows each node that was enabled at the beginning of the round to make a move, unless the node becomes disabled due to a move by a neighbor.

3. The p-Star Decomposition Algorithm

In this section we describe the implementation of the p-star decomposition algorithm. First we give an overview of how the algorithm works. Assume that some nodes are already a *member* of a star, while others are not. Nodes indicate to their neighbors whether they are a member of a star or not. In addition, each node indicates whether it may be *viable center* of a new star. That is the case only if the node itself and p of its neighbors are not a member of a star, yet.

Nodes keep track of the viable centers within the closed neighborhood. Unless they are a member of a star, they invite the viable center having the minimum identifier to form new p-star. The invitation is updated as needed if the set of viable centers within the closed neighborhood changes. Directing the invitation at the viable center with the minimum identifier makes sure that no deadlocks can occur. Eventually, a viable center is invited by itself and at least p neighbors. Such a viable center v then picks p neighbors as the leaves of the star and assigns them to a new p-star centered at v.

In the remainder of this section, we present the shared variables and predicates used by the algorithm. Then, the rules are presented.

3.1. Variables and Predicates

Algorithm 1 gives an overview over the variables, their domain, their meaning, and the predicates that the algorithm uses. Node v is the center of a p-star if and only if v verifies $correctCenter(v)$. The set stored in the shared variable $leaves(v)$ contains the p neighbors of v belonging to the star centered in v. So $star(v)$, as defined below, is the set of nodes in the p-star centered in v.

Definition 1. $star(v) = \emptyset$ if $leaves(v) = \emptyset$ and $star(v) = \{v\} \cup leaves(v)$ otherwise.

For any p-star centered in v, the following statements are proven (Observation 1, Lemma 2):
- The shared variable *center* is equal to v for every node in $star(v)$;
- Every node in $star(v)$ verifies the predicate *correctLeaf* or *correctCenter*;
- The star is well formed, i.e., the star contains v and p neighbors of v.

The values of the shared variables *center* and *leaves* in its neighborhood are enough to allow a node v to determine the value of the predicates $correctCenter(v)$ and $correctLeaf(v)$; so they suffice to compute the value of $isInStar(v)$.

Algorithm 1 Shared Variables, Predicates, Macros and Guard predicates

Shared variables of each node $v \in V$
- $center(v)$ — a node identifier or $\perp$
  The center of the p-star that v belongs to or the viable center that v invites to form new p-star. The value $\perp$ is used if v is not a member of a p-star and is not inviting any node.
- $leaves(v)$ — a set of up to p node identifiers
  The set is empty if v is not the center of a p-star.
  Otherwise it contains the leaves of the p-star.
- $inStar(v) \in Boolean$
  Indicates whether v is a member of a p-star.
- $viableCenter(v) \in Boolean$
  Indicates whether v is a viable center for a new p-star.
- $lockedCenter(v) \in Boolean$
  Indicates whether the value of $center(v)$ is locked or not.

Predicates
- $isCenter(v) \equiv |leaves(v)| = p$
- $incorrectCenter(v) \equiv (leaves(v) \neq \emptyset) \wedge$
  $((center(v) \neq v) \vee (\exists u \in leaves(v) : center(u) \neq v) \vee$
  $\neg isCenter(v) \vee (leaves(v) \not\subseteq N(v)))$
- $correctLeaf(v) \equiv (center(v) \in N(v)) \wedge$
  $isCenter(center(v)) \wedge (v \in leaves(center(v))$
- $correctCenter(v) \equiv isCenter(v) \wedge \neg incorrectCenter(v)$
- $isInStar(v) \equiv correctLeaf(v) \vee correctCenter(v)$
- $isViableCenter(v) \equiv \neg isInStar(v) \wedge (|\{\ u \in N(v) \mid \neg inStar(u)\ \}| \geq p)$

Macros
- $bestCenter(v)$ is the element of $\{u \in N[v] \mid viableCenter(u) \wedge leaves(u) = \emptyset\}$
  having the smallest identifier or $\perp$ if the set is empty
- $potentialLeaves(v)$ is the set $\{u \in N(v) \mid center(u) = v \wedge lockedCenter(u)\}$

Guard Predicates
- $starToUpdate(v) \equiv \neg isInStar(v) \wedge (|potentialLeaves(v)| \geq p) \wedge (v = center(v))$
- $centerToUpdate(v) \equiv \neg isInStar(v) \wedge (center(v) \neq bestCenter(v) \vee \neg lockedCenter(v))$
- $variablesToUpdate(v) \equiv (inStar(v) \neq isInStar(v)) \vee (viableCenter(v) \neq isViableCenter(v)) \vee$
  $incorrectCenter(v)$

Observation 1.
$leaves \neq \emptyset \wedge \neg incorrectCenter(v) \Rightarrow correctCenter(v)$
$leaves(v) = \emptyset \Rightarrow \neg incorrectCenter(v)$
$variablesToUpdate(v) \Rightarrow v$ *is enabled*
$incorrectCenter(v) \Rightarrow v$ *is enabled*

Algorithm 2 : p-star rules on v

Procedures

- $updateBooleans(v)$: $inStar(v) := isInStar(v)$;
  $viableCenter(v) := isViableCenter(v)$;
- $updateVariables(v)$: if $incorrectCenter(v)$ then $leaves(v) := \emptyset$;
  $updateBooleans(v)$;

Rules

RA(v) : $starToUpdate(v) \longrightarrow$
$leaves(v) :=$ subset of $potentialLeaves(v)$ with exactly p elements;
$updateBooleans(v)$;

RI(v) : $\neg starToUpdate(v) \wedge centerToUpdate(v) \wedge \neg lockedCenter(v) \longrightarrow$
$lockedCenter(v) :=$ true;
$center(v) := bestCenter(v)$;
$updateVariables(v)$;

RGI(v) : $\neg starToUpdate(v) \wedge centerToUpdate(v) \wedge lockedCenter(v) \longrightarrow$
$lockedCenter(v) :=$ false;
$updateVariables(v)$;

RU(v) : $\neg starToUpdate(v) \wedge \neg centerToUpdate(v) \wedge variablesToUpdate(v) \longrightarrow$
$updateVariables(v)$;

Lemma 2. *Every node v verifying $\neg incorrectCenter(v)$ and $star(v) \neq \emptyset$ satisfies the following assertions:*

$$|star(v)| = p + 1 \tag{1}$$
$$star(v) \subseteq N[v] \tag{2}$$
$$center(u) = v \qquad \forall u \in star(v) \tag{3}$$
$$correctLeaf(u) = \text{true} \qquad \forall u \in leaves(v) \tag{4}$$
$$isInStar(u) = \text{true} \qquad \forall u \in star(v) \tag{5}$$

Proof. We have $leaves(v) \neq \emptyset$ and $\neg incorrectCenter(v)$. According to Observation 1, v verifies $correctCenter(v)$.
By definition of $correctCenter(v)$, we have $leaves(v) \subseteq N(v)$, $|leaves(v)| = p$, so assertions (1) and (2) are verified.
Assertion (3) is verified by v because we have $\neg incorrectCenter(v)$. We conclude that assertion (4) is verified. So assertion (5) is also verified. □

3.2. p-Star Construction

The shared variable *inStar* allows nodes to determine whether they are a viable center of a new p-star. If a node v and at least p neighbors of v are not a member of a p-star, then v may become the center of a new p-star (i.e., v verifies the predicate *isViableCenter*).

The shared variable $viableCenter(v)$ informs neighbors of node v that in the current configuration, v may become the center of an new p-star. The shared variables *inStar* and *viableCenter* are updated by any rule during the execution of procedure *updateBooleans*. Note that if the value of $inStar(v)$ or $viableCenter(v)$ is not accurate, then v is enabled (Observation 1).

In the remainder of the algorithm presentation, we will study the behavior of nodes (v, u) that are not in a p-star (i.e., nodes in CS).

Definition 2.
Let S be the set of nodes belonging to a p-star, i.e., $S = \bigcup_{v \in V} star(v)$. Then $CS = V \setminus S$ is the complement of S, i.e., the nodes that do not belong to a p-star.

The macro $bestCenter(v)$ returns the viable center that v should invite to create a new p-star. If v is not connected to a viable center, then $\perp$ is returned. More precisely, $bestCenter(v)$ chooses the node having the smallest identifier among the viable centers at a distance less than 2 from v. In particular, we have $viableCenter(u)$ and $leaves(u) = \emptyset$ if $bestCenter(v) = u$.

To invite u, node v sets its shared variable *center* to u via rule RI. Rule RI is the only rule modifying the value of $center(v)$. If rule RI is enabled for node v, then $\neg lockedCenter(v)$ is verified. Node v is enabled if $lockedCenter(v) \neq$ true or $center(v) \neq bestCenter(v)$ because $centerToUpdate(v)$ is verified. So v will invite node $bestCenter(v)$ to be the center of its star or it will set $center(v)$ to $\perp$ if it is isolated (rule RI).

The set $potentialLeaves(v)$ contains the neighbors of v that can safely be in the p-star centered at v if the p-star is created during the next step. Any node of this set invites v to be the potential center and they cannot update their variable *center* during the next step because the value of their shared variable *lockedCenter* is true. To change the value of *center*, a node with must first execute rule RGI before it can change *center* using rule RI in a subsequent step.

Let v be a node of CS. If the size of $potentialLeaves(v)$ is at least p and $center(v) = v$ then v can form a p-star centered in itself by executing rule RA.

3.3. p-Star Stability

The execution of rule RA by some node v builds a p-star. After this move, v is the center of a p-star containing p members of $potentialLeaves(v)$, as proven in Lemma 4. The p-star stays unchanged (i.e., the set $star(v)$ stays unchanged) along any execution, which is proven in Lemma 3.

Lemma 3. *Let c be a configuration where v verifies $correctCenter(v)$. Along any execution from c, the predicate $correctCenter(v)$ is verified. The set $leaves(v)$ stays unchanged along any execution from c.*

Proof. Only a move of a node in $star(v)$ may change the value of the predicates $isCenter(v)$ and $incorrectCenter(v)$ or the set $leaves(v)$.

According to Lemma 2 and the definitions of $correctCenter(v)$ and $star(v)$, the nodes of $star(v)$ verify the predicate *isInStar*. For these nodes, only rule RU may be enabled.

The action of rule RU does not change the value of $leaves(v)$ and $center(v)$ since $incorrectCenter(v) =$ false in c. For a node u of $leaves(v)$, the action of rule RU does not change the value of $center(u)$.
So the values of $leaves(v)$, $isCenter(v)$, and $incorrectCenter(v)$ stay unchanged. □

Lemma 4. *If a node v executes rule RA during the step $c_1 \to c_2$, then $correctCenter(v)$ is verified in c_2.*

Proof. In c_1, $potentialLeaves(v)$ is a set of at least p nodes of $N(v)$ and any node u of $potentialLeaves(v)$ verifies $center(u) = v$ and $lockedCenter(u)$.

Let u be a neighbor of v with $u \in potentialLeaves(v)$ in c_1. In c_1, u is disabled with respect to rule RI. Hence $center(u)$ cannot change during the step $c_1 \to c_2$. So in c_2, $leaves(v)$ is a set of p distinct nodes of $N(v)$ and any node of $leaves(v)$ verifies $center(u) = v$. Also $center(v) = v$ in c_2, according to rule RA. Thus, $correctCenter(v)$ is verified in c_2. □

4. Correctness of Terminal Configurations

In this section we establish that any terminal configuration represents a maximal p-star decomposition of the graph. More precisely, we prove that S contains only well formed disjoint p-stars and S is maximal, i.e., no p-star can be added to S.

Lemma 5. *Let u, v be two nodes verifying the predicate $\neg incorrectCenter$. Then the sets $star(v)$ and $star(u)$ are disjoint.*

Proof. The claim is trivial if either $star(u)$ or $star(v)$ is empty. According to Lemma 2, $center(w) = u$ (resp. $center(x) = v$) for all $w \in star(u)$ (resp. for all $x \in star(v)$). Thus $star(u)$ and $star(v)$ cannot intersect. □

Definition 3. PS denotes the set of nodes of CS having at least p neighbors in CS (definition 2), i.e., $PS = \{z \in CS \mid \ |N(z) \cap CS| \geq p\}$.

Lemma 6. *In any terminal configuration, we have*
- $leaves(z) = \emptyset$, *for any node z in CS*
- $isInStar(u) = false$ *for any node x in CS*
- $viableCenter(z) = true$ *for any node z in PS*
- $viableCenter(x) = false$ *for any node x not in PS*

Proof. Let w be a node of CS. According to the definition of S, $w \notin \bigcup_{v \in V} leaves(v)$ and $leaves(w) = \emptyset$. So, we have $\neg isCenter(w)$ and $\neg correctLeaf(w)$. According to Observation 1, for any node w of CS, $isInStar(w) = inStar(w) =$ false because w is disabled.

Let z be a node of PS. z has at least p neighbors having their shared variable *inStar* set to false. Thus $isViableCenter(z)$ is verified. According to Observation 1, $viableCenter(z) =$ true, because z is disabled.

Let CPS denote the set of nodes of CS having at most $p - 1$ neighbors in CS (i.e., $CPS = CS \setminus PS = \{z \in CS \mid \ |N(z) \cap CS| < p\}$). Let cz be a node of CPS. Node cz does not have p neighbors having their shared variable *inStar* set to false. $\neg isViableCenter(cz)$ is verified. So, $viableCenter(cz) = false$.
Let u be a node of S. According to Lemma 2, $isInStar(u) = inStar(u) = true$. So, $viableCenter(u) = false$ because u is disabled.
We conclude that for any node not in PS, $viableCenter(x) = false$. □

PS is the set of nodes that could be the center of a *star* set containing only nodes that are not yet in a p-star.

Lemma 7. *In any terminal configuration, PS is empty.*

Proof. Assume that PS is not empty. Let cz be the node of PS having the smallest identifier. Consider any node $w \in CS \cap N[cz]$. We have $bestCenter(w) = cz$ and $isInStar(w) = false$ are according to Lemma 6. As w is disabled, $\neg centerToUpdate(w)$ is verified (Observation 1). We have $lockedCenter(w) =$ true and $center(w) = bestCenter(w) = cz$.
As $|CS \cap N(cz)| \geq p$, $starToUpdate(cz)$ is verified; so node cz is enabled with respect to rule RA. That is a contradiction to the assumption that the configuration is terminal. Thus PS is empty. □

Theorem 8. *In any terminal configuration, $S = \bigcup_{v \in V} star(v)$ is a valid maximal p-star decomposition.*

Proof. According to Observation 1, any node v verify the predicate $\neg incorrectCenter(v)$ in a terminal configuration.

According to Lemma 2, and the definition of *star*, $star(v)$ is empty or is a p-star centered to v in a terminal configuration. According to Lemma 5, the *star* sets are pairewise disjoint.

According to Lemma 7 and the definitions of PS, and S is maximal. □

5. Move Complexity

5.1. Number of RA and RU moves per node

Lemma 9. *Every node v executes rule RA at most once.*

PROOF. After v executes rule RA, it verifies the predicate *correctCenter*(v) (Lemma 4). From a configuration where *correctCenter*(v) is verified this predicate stays verified along any execution (Lemma 3). So along any execution, v may perform only the rule RU, because *isInStar*(v) is always verified (Lemma 2). □

Lemma 10. *For every node v, $\neg$incorrectCenter(v) is a closed predicate.*

PROOF. Assume $\neg$*incorrectCenter*(v) is verfied in configuration c. According to Observation 1, in c, we have *leaves*$(v) = \emptyset$ or *isCenter*(v). Assuming that *isCenter*(v) in c, the claim follows from Lemma 3. Assume that *leaves*$(v) = \emptyset$ in c. Only the action of rule RA sets *leaves*(v) to a non-empty set. The claim follows from Lemmas 3 and 4. □

Lemma 11. *If a node v makes a move, then $\neg$incorrectCenter(v) is verified by the configuration reached immediately after this move.*

PROOF. Consider a step $c_1 \to c_2$ that contains a move of v. If v executes rule RA, then the claim follows from Lemma 4. If $\neg$*incorrectCenter*(v) is verified in c_1 the claim follows from Lemma 10

Assume that v executes rules RI, RGI, or RU from a configuration c_1 where *incorrectCenter*(v) is verified. The procedure *updateVariables*(v) empties the set *leaves*(v) during the step $c_1 \to c_2$. So, in c_2 we have *leaves* $= \emptyset$. According to Observation 1, $\neg$*incorrectCenter*(v) is verified in c_2. □

Lemma 12. *Consider a step $c_1 \to c_2$ in which correctLeaf(v) changes from* false *to* true*. Let u be the node such that center$(v) = u$ in c_2. Then node u executes rule RA during the step.*

PROOF. The proof is by contradiction, i.e., assume that node u has not executed rule RA during the step. The action of rules RI, RGI, or RU, if they change *leaves*(u), assign the empty set to *leaves*(u). As $v \in$ *leaves*(u) in c_2, we conclude that u does not update its set *leaves* during this step.

So *correctLeaf*(v) could change from false to true during this step only if *center*(v) changes and *leaves*(u) does not change. That implies that v executed rule RI during the step. However, rule RI can only set *center*(v) to u if *bestCenter*$(v) = u$ which implies that *leaves*$(u) = \emptyset$ in c_1 and in c_2. That is a contradiction to $v \in$ *leaves*(u) in c_2. □

Lemma 13. *Consider a step $c_1 \to c_2$ in which correctLeaf(v) changes from* false *to* true*. Along any execution starting in c_2, correctLeaf(v) stays verified.*

PROOF. Let u be the node such that $u =$ *center*(v) in c_2. During the step $c_1 \to c_2$, u performs the rule RA (Lemma 12). So in c_2, node $u =$ *center*(v) verifies the predicate *correctCenter*(u) (Lemma 4). According to Lemma 3, v belongs to *leaves*(u) along any execution from c_2. In c_2, *star*$(v) \neq \emptyset$ and $\neg$*incorrectCenter*(v), so *correctLeaf*(v) stays verified along any execution (Lemma 2). □

Lemma 14. *For each node v, the value of isInStar(v) changes at most 2 times.*

PROOF. Consider a step $c_1 \to c_2$ in which *isInStar*(v) changes from false to true. This is either due to the fact that *correctLeaf*(v) became true, or *correctCenter*(v) became true.

correctCenter(v) is closed (Lemma 3). *correctLeaf*(v) stays verified after that step (Lemma 13).

So *isInStar*(v) may change from false to true at most once; then it keeps the value true. Thus $\langle \text{true}, \text{false}, \text{true} \rangle$ is the maximum sequence of values that *isInStar*(v) may assume. □

Accounting for an initial inconsistencies, we obtain the following result.

Corollary 15. *Each node v changes the variable inStar(v) at most* 3 *times.*

Lemma 16. *For each node v, the value of isViableCenter(v) changes at most $3\deg(v) + 2$ times.*

PROOF. *isViableCenter*(v) may change its value each time a neighbors $u \in N(v)$ change its value of *inStar*(v). By Corollary 15, this may occur at most $3\deg(v)$ times. Furthermore, *isViableCenter*(v) may also change its value if *isInStar*(v) changes. By Lemma 14, this can occur at most 2 times. □

Accounting for an initial inconsistencies, we obtain the following result.

Corollary 17. *Each node v changes the variable viableCenter(v) at most $3\deg(v)+3$ times.*

Lemma 18. *Each node v executes the rule RU at most $3\deg(v)+5$ times.*

Proof. By definition of *variablesToUpdate*(v), rule RU is only enabled if variables *inStar*(v) or *viableCenter*(v) need updating or if *incorrectCenter*(v) = true. The first execution of rule RU may be triggered by initial inconsistencies. Note that by Lemmas 11 and 10, *incorrectCenter*(v) never becomes true after the first move of v.

Thus all further executions of RU are caused by changes of *isInStar*(v) and *isViableCenter*(v). The latter can occur at most $3\deg(v)+2+2$ times by Lemmas 14 and 16. □

5.2. Number of RGI and RI moves per node

Lemma 19. *Per node, the variable leaves(v) changes at most 2 times.*

Proof. Rule RA is the only rule assigning a non-empty set to *leaves*(v). By Lemma 9, rule RA can only be executed once by v and afterwards, *leaves*(v) does not change. All other rules set *leaves*(v) to the empty set if they modify *leaves*(v). Thus *leaves*(v) can only change twice: once to the empty set and once to a non-empty set. □

Lemma 20. *For every node v, the value of bestCenter(v) changes at most $\deg(v)(3\Delta+5)+3\deg(v)+5$ times.*

Proof. *bestCenter*(v) changes only if the set of nodes $u \in N[v]$ satisfying the condition (i) *viableCenter*$(u) \wedge$ *leaves*$(u) = \emptyset$ changes. By Corollary 17 and Lemma 19, the condition (i) changes its value at most $3\deg(u)+3+2 \leq 3\Delta+5$ times on any execution. The claim follows as $N[v]$ contains exactly $\deg(v)$ nodes plus v. □

Lemma 21. *Rule RGI is executed at most $\deg(v)(3\Delta+5)+3\deg(v)+5+1$ times.*

Proof. By definition of *centerToUpdate*(v), rule RGI is enabled only if *center*$(v) \neq$ *bestCenter*(v) and *lockedCenter*(v) = true. When the algorithm assigns true to *lockedCenter*(v) (rule RI), then it also assigns *bestCenter*(v) to *center*(v). So *center*$(v) \neq$ *bestCenter*(v) when *lockedCenter*(v) = true is either due to initial inconsistencies or due to the fact that *bestCenter*(v) has changed. By Lemma 20, the latter can only happen $\deg(v)(3\Delta+5)+3\deg(v)+5$ times. □

Lemma 22. *Rule RI is executed at most $\deg(v)(3\Delta+5)+3\deg(v)+5+2$ times.*

Proof. For rule RI to be enabled, it must holds that *lockedCenter*(v) = false. When executed, rule RI changes *lockedCenter*(v) to true and only rule RGI set to false this variable. So any execution of rule RI is either due to an initial inconsistency or must be preceded by an execution of rule RGI. The latter can happen at most $\deg(v)(3\Delta+5)+3\deg(v)+5+1$ times by Lemma 21. □

Theorem 23 (Upper bound on the number of moves). *The algorithm terminates after at most $12\Delta m + \mathcal{O}(m+n)$ moves.*

Proof. By Lemma 9, rule RA can be executed at most 1 time. By Lemma 18, rule RU is executed at most $3\deg(v)+5$ times. Rule RGI can be executed at most $\deg(v)(3\Delta+5)+3\deg(v)+6$ times by Lemma 21, and 22 rule RI is executed at most $\deg(v)(3\Delta+5)+3\deg(v)+7$ times. So each node v makes at most

$$2\deg(v)(3\Delta+5)+9\deg(v)+19 = 6\deg(v)\Delta+19\deg(v)+19 \text{ moves.}$$

The total number of moves over all nodes is then bounded by

$$\textstyle\sum_{v\in V}(6\Delta\deg(v)+19\deg(v)+19) = 12\Delta m+38m+19n$$

□

6. Round Complexity

The proofs of the round complexity utilize the notion of *attractors* [30]. An attractor A is a set of configuration that is closed under the execution of the algorithm. That is, for any execution, configurations subsequent to a configuration of A are also in A. A self-stabilizing always has two attractors: the set of all configurations and the set of all legitimate configurations.

Let $\mathcal{C}$ denote the set of all configurations. First we will establish that attractor A_0, as defined below, is reached in a single round.

Definition 4. $A_0 = \{c \in \mathcal{C} \mid \forall v \in V : \neg incorrectCenter(v)\}$

6.1. Properties of A_0

The following corollary is a direct consequence of the Observation 1 and Lemmas 10 and 11.

Corollary 24. *After at most one round of execution starting in any configuration, A_0 has been reached and A_0 is closed under any execution of the algorithm.*

6.2. Properties of executions starting in A_0

In the following, we establish that in any configuration of A_0, at most *nbMaxStars* nodes verify the predicate *isCenter*.

Definition 5. $nbMaxStars = \lfloor \frac{n}{p+1} \rfloor$.

The following corollary follows from the definition of A_0, and *star* and Lemmas 3 and 2.

Corollary 25. *In A_0, isCenter is a closed predicate. In A_0, if isCenter(v) is verified, then $|star(v)| = p+1$.*

Corollary 26. *In any execution from A_0, the rule RA is performed at most nbMaxStars times.*

Proof. Let c be a configuration of A_0 According to Corollary 25, if $isCenter(v)$ is verified in c then $|star(v)| = p + 1$. According to Lemma 5, in c, if $u \neq v$ then $star(u)$ is disjoint of $star(v)$. So, in c, the number of nodes satisfying the predicate *isCenter* is bounded by *nbMaxStars*.
According to Corollary 25, in A_0 the predicate *isCenter* is closed.
Thus, the rule RA is performed at most *nbMaxStars* times along any execution in A_0. □

6.3. Properties of RA-restricted executions starting in A_0

We call a step $c_1 \to c_2$ *RA-restricted*, if no node executes rule RA during that step. Similarly, we say that a round is *RA-restricted* if it consists of only RA-restricted steps. In the remainder we study executions starting in configurations of A_0. We show that any such execution contains at most 4 consecutive RA-restricted rounds, i.e., after 4 consecutive RA-restricted rounds, either a terminal configuration is reached or in the following round at least one node executes rule RA. In any execution starting in A_0, there are at most *nbMaxStars* RA-unrestricted rounds (Corollary 26). So the number total of rounds, in any execution from A_0 is at most $5nbMaxStars + 4$.

First, we establish that the value of $leaves(v)$ and $isInStar(v)$ for any node v in a configuration of A_0 stays unchanged during any execution till no node performs the RA action.

Lemma 27. *Consider an RA-restricted step $c_1 \to c_2$ with $c_1 \in A_0$. For any node v, the value of $leaves(v)$ does not change during this step.*

Proof. The rules RI, RGI and RU modify $leaves(v)$ if $incorrectCenter(v) =$ true in c_1. We have $incorrectCenter(v) =$ false in $c_1 \in A_0$. Thus, $leaves(v)$ does not change during this step. □

Lemma 28. *Consider a step $c_1 \to c_2$ with $c_1 \in A_0$. If $correctCenter(v)$ changes from* false *to* true*, then node v executed rule RA during this step.*

Proof. The proof is by contradiction, i.e., assume that node v has not executed rule RA during the step. $\neg incorrectCenter(v)$ and $\neg correctCenter(v)$ are verified in c_1. So $leaves(v) = \emptyset$ in c_1 (according to Observation 1). According to Lemma 27, in c_2, we have $leaves(v) = \emptyset$. So $isCenter(v)$ is not verified in c_2. □

Lemma 29. *$isInStar(v)$ is a closed under any RA-restricted step $c_1 \to c_2$ with $c_1 \in A_0$.*

Proof. Let v be a node that verifies $isInStar(v)$ in c_1. Either $correctCenter(v)$ or $correctLeaf(v)$ is verified, by definition of $isInStar$.

If $correctCenter(v)$ is verified in c_1, then according to Lemma 3 $correctCenter(v)$ is also verified in c_2. So $isInStar(v)$ is verified in c_2 (Lemma 2).

Let $u = center(v)$. If $correctLeaf(v)$ is verified, then $leaves(u) \neq \emptyset$ in c_1. So by definition of A_0 and Observation 1 we have $correctCenter(u)$ in c_1 and thus in c_2 (Lemma 3). Since $v \in leaves(u)$ in c_2 by Lemma 27, we have $correctLeaf(v)$ in c_2 by Lemma 2. □

Lemma 30. *Consider an RA-restricted step $c_1 \to c_2$ with $c_1 \in A_0$. For every node v, the value of $isInStar(v)$ does not change during this step.*

Proof. Let v be a node. A change of $isInStar(v)$ from false to true implies that either $correctLeaf(v)$ or $correctCenter(v)$ changes from false to true. According to Lemmas 12 and 28, such a change occurs only if a node performs the rule RA during this step. Since this step is RA-restricted, this cannot occur.
According to Lemma 29, a change of the value of $isInStar(v)$ from true to false cannot happen in A_0. □

Definition 6. $A_1 = \{c \in A_0 \mid \forall v \in V : inStar(v) = isInStar(v)\}$

6.4. Properties of A_1

Lemma 31. *Let c be a configuration of A_0 and $c1$ the configuration reached after one RA-restricted round from c. We have $c1 \in A_1$ and A_1 is closed under any RA-restricted step.*

Proof. By Corollary 24, the set A_0 is closed and thus $c1 \in A_0$. Let v be a node with $inStar(v) \neq isInStar(v)$ in c. Node v is enabled as long as this inequality holds and no node executes rule RA. Thus node v performs a move during the first RA-restricted round. After the move of v, we have $inStar(v) = isInStar(v)$. This equality stays verify until a node does the RA action (Lemma 30). □

Lemma 32. *Consider an RA-restricted step $c_1 \to c_2$ with $c_1 \in A_1$. For every node v, the value of $isViableCenter(v)$ does not change during this step.*

Proof. The value of $isViableCenter(v)$ may only changes when the value of $inStar(u)$ for some node $u \in N(v)$ changes. As $c_1 \in A_1$ we have $inStar(u) = isInStar(u)$ for all nodes $u \in V$. So no node changes the value of its $inStar$ variable during this step. □

Definition 7. $A_2 = \{c \in A_1 \mid \forall v \in V : \neg variablesToUpdate(v)\}$

6.5. Properties of A_2

Lemma 33. *Let c be a configuration of A_0 and $c2$ the the configuration reached after two RA-restricted rounds from c. We have $c2 \in A_2$ and A_2 is closed under any RA-restricted step.*

Proof. After the first RA-restricted round from c, the configuration $c1 \in A_1$ is reached (Lemma 31). Since A_1 is closed we have $c2 \in A_1$.

Let v be a node with $isViableCenter(v) \neq viableCenter(v)$ in $c1$. As long as this inequality holds and no node performs RA action, the node v is enabled. Thus node v performs a move during the second RA-restricted round. After the move of v, we have $isViableCenter(v) = viableCenter(v)$. This equality stays verify until a node does the RA action (Lemma 32). □

Lemma 34. *Consider an RA-restricted step $c_1 \to c_2$ with $c_1 \in A_2$. For every node v, the value of $bestCenter(v)$ does not change during this step.*

Proof. The value $bestCenter(v)$ can only change if $viableCenter(u)$ or $leaves(u)$ change for some $u \in N[v]$. Since $c_1 \in A_2$ we have $viableCenter(u) = isViableCenter(u)$ for all nodes $u \in V$. By Lemma 32 we know that the value of $isViableCenter(u)$ does not change during this step for all nodes $u \in V$. Also, by Lemma 27, we know that the set $leaves(u)$ does change for any node $u \in V$. Therefore, the value $bestCenter(v)$ does change during this step. □

Definition 8. $A_3 = \{c \in A_2 \mid \forall v \in V : \neg centerToUpdate(v)\}$

6.6. Properties of A_3

Lemma 35. *Let c be a configuration of A_0 and $c3$ be the configuration reached after four RA-restricted rounds from c. We have $c3 \in A_3$ and A_3 is closed under any RA-restricted step.*

Proof. After two RA-restricted rounds from c, the configurations $c2 \in A_2$ is reached (Lemmas 33). Since A_2 is closed the have $c3 \in A_2$.

Let v be a node with $centerToUpdate(v) = true$ in $c2$. Without loss of generality, assume that $lockedCenter(v) =$ true in $c2$. Node v is enabled as long as $centerToUpdate(v) =$ true holds and no node performs RA action. Thus node v makes an RGI move during the third RA-restricted round. After the move of v, we have $lockedCenter(v) =$ false. So node v remains enabled until it performs another move or RA action is performed.

Let v be a node with $centerToUpdate(v) = true$, and $lockedCenter(v) =$ false in $c3$ (the configuration reached after 3 RA-restricted rounds from c). Before the end of the fourth RA-restricted round, v executes rule RI which updates $center(v)$ with the value of $bestCenter(v)$ and sets $lockedCenter(v)$ to true. After that move, $centerToUpdate(v) =$ false holds as $bestCenter(v)$ does not change (Lemma 34). □

Lemma 36. *Consider a configuration of A_3. Each node is either disabled or only enabled with respect to rule RA.*

Proof. Let v be a node. By definition of A_3 we have $centerToUpdate(v) =$ false. Thus v is disabled with respect to rules RI and RGI. Also, by definition of $A_2 \supseteq A_3$ we have $variablesToUpdate(v) =$ false. Thus rule RU is disabled. So v is either disabled or enabled with respect to rule RA. □

Combining Lemmas 35, and 36 yields the following result.

Corollary 37. *Any execution starting in A_0 contains at most 4 consecutive RA-restricted rounds.*

Theorem 38 (Upper bound on the number of rounds). *The algorithm terminates after at most $5(nbMaxStars + 1) = 5(\left\lfloor \frac{n}{p+1} \right\rfloor + 1)$ rounds.*

Proof. By Corollary 24, any non-restricted execution reaches A_0 after at most 1 round. Afterwards, an execution of rule RA occurs at least every 5 rounds by Corollary 37. Also by Corollary 37, any trailing suffix not containing any executions of rule RA can be at most 4 rounds in length.

By Corollary 26, there are at most $nbMaxStars$ rounds having a RA action along any execution starting in A_0. This completes the proof. □

7. Conclusion

We studied in this paper the problem of decomposing a graph into node-disjoint p-stars from a self-stabilization point of view. This problem is a generalization of maximal matching problem in graphs. We proposed a new self-stabilizing algorithm for maximal p-star decomposition problem performing better than both previously proposed algorithms. In fact, we improve the move complexity to $\mathcal{O}(\Delta m)$ instead of $\mathcal{O}(\Delta^2 m)$ in [29] and solve the uniqueness legitimate configuration problem that [28] suffered from, without losing linearity of round complexity. As future work, we aim to generalize the proposed algorithm to the weighted p-star decomposition problem.

References

[1] E. W. Dijkstra, Self-stabilizing systems in spite of distributed control, Commun. ACM 17 (11) (1974) 643–644.
[2] S. Dolev, Self-stabilization, MIT Press, 2000.
[3] P. Cain, Decomposition of complete graphs into stars, Bull. Austral. Math. Soc. 10 (1974) 23–30.
[4] H.-C. Lee, C. Lin, Balanced star decompositions of regular multigraphs and *lambda*-fold complete bipartite graphs, Discrete Mathematics 301 (2-3) (2005) 195–206.
[5] E. E. R. Merly, N. Gnanadhas, Linear star decomposition of lobster, Int. J. of Contemp. Math. Sciences 7 (6) (2012) 251–261.
[6] D. E. Bryant, S. I. El-Zanati, C. V. Eynden, Star factorizations of graph products, J. Graph Theory 36 (2) (2001) 59–66.
[7] K. Andreev, H. Räcke, Balanced graph partitioning, in: 16th annual ACM symposium on Parallelism in algorithms and architectures, 2004, pp. 120–124.
[8] S. Lemmouchi, M. Haddad, H. Kheddouci, Robustness study of emerged communities from exchanges in peer-to-peer networks, Computer Communications 36 (10–11) (2013) 1145 – 1158.
[9] M. Mezmaz, N. Melab, E.-G. Talbi, A grid-based parallel approach of the multi-objective branch and bound, in: 15th Euromicro International Conference on PDP, 2007, pp. 23–30.
[10] A. Bendjoudi, N. Melab, E.-G. Talbi, P2p design and implementation of a parallel branch and bound algorithm for grids, Int. J. Grid Util. Comput. 1 (2) (2009) 159–168.
[11] W. Y. Chiu, C. Chen, S.-Y. Tsai, A $4n$-move self-stabilizing algorithm for the minimal dominating set problem using an unfair distributed daemon, Inf. Process. Lett. 114 (10) (2014) 515–518.
[12] S.-C. Hsu, S.-T. Huang, A self-stabilizing algorithm for maximal matching, Inf. Process. Lett. 43 (2) (1992) 77–81.
[13] S. T. Hedetniemi, D. P. Jacobs, P. K. Srimani, Maximal matching stabilizes in time $O(m)$, Inf. Process. Lett. 80 (5) (2001) 221–223.
[14] N. Guellati, H. Kheddouci, A survey on self-stabilizing algorithms for independence, domination, coloring, and matching in graphs, J. Parallel Distrib. Comput. 70 (4) (2010) 406–415. doi:10.1016/j.jpdc.2009.11.006.
[15] L. Blin, M. G. Potop-Butucaru, S. Rovedakis, S. Tixeuil, Loop-free super-stabilizing spanning tree construction, in: 12th SSS, 2010, pp. 50–64.
[16] E. Caron, A. K. Datta, B. Depardon, L. L. Larmore, A self-stabilizing k-clustering algorithm using an arbitrary metric, in: 15th International Euro-Par Conference, 2009, pp. 602–614.
[17] D. Bein, A. K. Datta, C. R. Jagganagari, V. Villain, A self-stabilizing link-cluster algorithm in mobile ad hoc networks, in: 8th International Symposium on Parallel Architectures,Algorithms and Networks, 2005, pp. 436–441.
[18] F. Belkouch, M. Bui, L. Chen, A. K. Datta, Self-stabilizing deterministic network decomposition, J. Parallel Distrib. Comput. 62 (4) (2002) 696–714.
[19] C. Johnen, L. H. Nguyen, Robust self-stabilizing clustering algorithm, in: 10th International Conference on Principles of Distributed Systems, 2006, pp. 410–424.
[20] B. Neggazi, M. Haddad, H. Kheddouci, Self-stabilizing algorithm for maximal graph partitioning into triangles, in: 14th SSS, 2012, pp. 31–42.
[21] S. Ghosh, M. H. Karaata, A self-stabilizing algorithm for coloring planar graphs, Distributed Computing 7 (1) (1993) 55–59.
[22] S. T. Hedetniemi, D. P. Jacobs, P. K. Srimani, Linear time self-stabilizing colorings, Information Processing Letters 87 (5) (2003) 251–255.
[23] D. G. Kirkpatrick, P. Hell, On the completeness of a generalized matching problem, in: 10th annual ACM symposium on Theory of computing, 1978, pp. 240–245.
[24] D. Kirkpatrick, P. Hell, On the complexity of general graph factor problems, SIAM Journal on Computing 12 (3) (1983) 601–609.
[25] G. Tel, Maximal matching stabilizes in quadratic time, Inf. Process. Lett. 49 (6) (1994) 271–272.
[26] W. Goddard, S. T. Hedetniemi, D. P. Jacobs, P. K. Srimani, Self-stabilizing protocols for maximal matching and maximal independent sets for ad hoc networks, in: 17th International Parallel and Distributed Processing Symposium, 2003, p. 14.
[27] F. Manne, M. Mjelde, L. Pilard, S. Tixeuil, A new self-stabilizing maximal matching algorithm, Theor. Comput. Sci. 410 (14) (2009) 1336–1345.
[28] B. Neggazi, V. Turau, M. Haddad, H. Kheddouci, A self-stabilizing algorithm for maximal p-star decomposition of general graphs, in: 15th SSS, 2013, pp. 74–85.
[29] B. Neggazi, M. Haddad, H. Kheddouci, A new self-stabilizing algorithm for maximal p-star decomposition of general graphs, Information Processing Letters 115 (11) (2015) 892–898.
[30] M. G. Gouda, N. J. Multari, Stabilizing communication protocols, IEEE Transactions on Computers 40 (4) (1991) 448–458. doi:10.1109/12.88464.