International Journal of Networking and Computing – www.ijnc.org
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume 5, Number 1, pages 105–121, January 2015

Self-Stabilizing Algorithms for Maximal 2-packing and General k-packing ($k \geq 2$) with Safe Convergence in an Arbitrary Graph

Yihua Ding, James Wang and Pradip K Srimani
School of Computing, Clemson University
Clemson, SC, 29634, USA

Received: July 9, 2014
Revised: October 22, 2014
Accepted: December 3, 2014
Communicated by Akihiro Fujiwara

Abstract

In a graph or a network $G = (V, E)$, a set $\mathcal{S} \subseteq V$ is a 2-packing if $\forall i \in V : |N[i] \cap \mathcal{S}| \leq 1$, where $N[i]$ denotes the closed neighborhood of node i. A 2-packing is maximal if no proper superset of $\mathcal{S}$ is a 2-packing. This paper presents a safely converging self-stabilizing algorithm for maximal 2-packing problem. Under a synchronous daemon, it quickly converges to a 2-packing (a safe state, not necessarily the legitimate state) in three synchronous steps, and then terminates in a maximal one (the legitimate state) in $O(n)$ steps without breaking safety during the convergence interval, where n is the number of nodes. Space requirement at each node is $O(\log n)$ bits. This is a significant improvement over the most recent self-stabilizing algorithm for maximal 2-packing that uses $O(n^2)$ synchronous steps with same space complexity and that does not have safe convergence property. We then generalize the technique to design a self-stabilizing algorithm for maximal k-packing, $k \geq 2$, with safe convergence that stabilizes in $O(kn^2)$ steps under synchronous daemon; the algorithm has space complexity of $O(kn \log n)$ bits at each node; existing algorithms for k-packing stabilize in exponential time under a central daemon with $O(\log n)$ space complexity.

Keywords: Self-stabilization, Maximal 2-packing, Maximal k-packing, Safe Convergence, Synchronous Daemon

1 Introduction

Self-stabilization: Self-stabilization is an optimistic paradigm to provide decentralized autonomous tolerance against an unlimited number of transient faults (transient faults corrupt data but not the program code) in distributed systems. An algorithm is self-stabilizing iff it reaches some legitimate global state starting from an arbitrary state [1]. In a self-stabilizing algorithm, each node maintains a set of local variables, that determine the *local state* of the node. The *global system state* is made of the union of local states of all nodes in the system. A self-stabilizing algorithm is specified as an uniform set of rules at each node. Each rule consists of a *condition* and an *action* and is written as "**if** condition **then** action". A condition is a boolean predicate involving the local states of the node and its neighbors. A node, at any step of execution, is called *privileged* iff at least one condition is true. The daemon (runtime scheduler) selects node(s) from among the privileged nodes to take an action (also called *move*) at each step. The *central* daemon selects exactly one privileged node to move at each step; the *distributed* daemon selects a non-empty subset of the privileged nodes to move at each step; the *synchronous* daemon selects all the privileged nodes to move at each step. A detailed exposition of self-stabilizing algorithms can be found in [2] and a recent survey of self-stabilizing algorithms for graph theoretic problems is given in [3].

Safe Convergence: Recently, a new concept of *safe convergence* has been introduced in [4]. In a traditional self-stabilizing algorithm, the desired global property (hence, the relevant service in the system) is not guaranteed during the convergence interval starting from an arbitrary state to a legitimate global system state. The concept of safe convergence was introduced to limit this inconvenience to a minimum possible. A self-stabilizing algorithm is said to have safe convergence property iff it first converges to a safe state quickly ($O(1)$ time is expected), and then converges to a legitimate state without breaking safety during the process. A safe state guarantees a minimum quality of service, and the legitimate state guarantees the desired service. Safe convergence property is especially attractive since it provides a measure of safety during the convergence interval of the self-stabilizing algorithm. Various self-stabilizing algorithms with safe convergence have been proposed in the literature, such as minimal independent dominating set, connected dominating set and so on [4, 5, 6, 7]. All of these algorithms use synchronous daemon to reach the safe states in constant time. Recently, two other self-stabilizing algorithms [8, 9] for minimum connected dominating sets in unit disk graphs and (f, g) alliances in arbitrary graphs respectively enjoy the self convergence property using unfair distributed daemon and distinct node IDs; but, starting from an arbitrary initial state the number of steps needed by the algorithms to reach a safe state can be quadratic in n in the worst case, where n is the number of nodes in the graph.

Graph Packing: In this paper, we are interested in the maximal 2-packing of a network graph. The concept of 2-packing and that of packing in general, k-packing, $k \geq 2$, have been used in various applications like network security, facilities location and others [10]. Authors in [11, 12] designed the first two self-stabilizing algorithms for maximal 2-packing, that stabilize in exponential time and $O(n^3)$ time respectively using a central daemon, where n is the number of nodes in the graph. Subsequently, other self-stabilizing algorithms have appeared [13, 14, 15]. The most recent self-stabilizing algorithm for maximal 2-packing is given in [16] that stabilizes in $O(n^2)$ synchronous steps (using a synchronous daemon). All of the above algorithms require $O(\log n)$ bits at each node and they assume that each node has a unique ID; none of them enjoys safe convergence property. Self-stabilizing k-packing algorithms are presented in [13, 14]; both algorithms stabilize in exponential time under a central daemon.

Contribution: In this paper, we assume a synchronous daemon and nodes with unique IDs and propose the first self-stabilizing algorithm with safe convergence to compute the maximal 2-packing of an arbitrary network graph; starting from an arbitrary state, the proposed algorithm first converges to a 2-packing (a *safe* state, not necessarily the legitimate state) in three synchronous steps, and then converges to a maximal one (the *legitimate* state) in $O(n)$ steps without breaking safety rule during the stabilization interval. Space requirement at each node is $O(\log n)$ bits. We then generalize the technique to design a self-stabilizing algorithm for maximal k-packing, $k \geq 2$, with safe convergence that stabilizes in $O(kn^2)$ steps under synchronous daemon; the algorithm has space complexity of $O(kn \log n)$ bits at each node.

2 Model and Terminology

A network or a distributed system is modeled by an undirected graph $G = (V, E)$, where V is the set of nodes, and E is the set of edges. For a node i, $N(i)$, its *open neighborhood*, denotes the set of nodes adjacent to node i; $N[i] = N(i) \cup i$ denotes the **closed neighborhood** of node i. For a node i, $N^{\ell}[i] = \cup_{j \in N[i]} N^{\ell-1}[j], \ell > 1$, where $N^1[i] = N[i]$, its **ℓ-hop closed neighborhood**, denotes the set of nodes that are at most distance of ℓ from node i. Each node $j \in N(i)$ is called a *neighbor* of node i. The distance $\boldsymbol{dist(i, j)}$ is the number of edge(s) in the shortest path between nodes i and j.

Definition 1 In a graph $G = (V, E)$, $|V| = n$ and $|E| = m$, (a) a set $\mathcal{S} \subseteq V$ is a **2-packing** iff $\forall i, j \in \mathcal{S} : dist(i, j) \geq 3$ where $dist(i, j)$ denotes the shortest distance from node i to j; A 2-packing

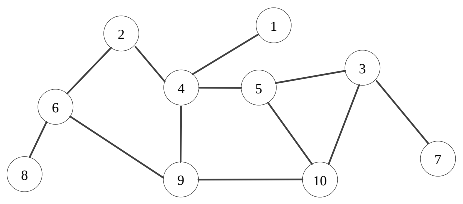

Figure 1: A graph with 10 nodes

is *maximal* if no proper superset of $\mathcal{S}$ is a 2-packing, i.e., $\forall i \in \{V - \mathcal{S}\}\ \exists j \in \mathcal{S} : dist(i,j) \leq 2$ [16]. (b) A set $\mathcal{S} \subseteq V$ is a **k-packing** if $\forall i, j \in \mathcal{S} : dist(i,j) \geq k+1$; $\mathcal{S}$ is a maximal k-packing if no proper superset of $\mathcal{S}$ is a k-packing, i.e., if $\forall i \in \{V - \mathcal{S}\}\ \exists j \in \mathcal{S} : dist(i,j) \leq k$ [14].

As an example, consider the graph shown in Figure 1, where the values inside the nodes give the identifiers. $n = 10$, $m = 12$. $N(2) = \{4, 6\}$, $N[2] = \{2, 4, 6\}$, $dist(2, 9) = 2$. In this graph, $\{1, 7\}$, $\{4, 8\}$, $\{5, 6\}$, and $\{1, 8, 10\}$ all are 2-packings, but only $\{5, 6\}$ and $\{1, 8, 10\}$ are maximal among these four 2-packings; $\{1, 8\}$, $\{6, 7\}$, and $\{7, 8\}$ all are 3-packings, but only $\{6, 7\}$ is maximal among these three 3-packings.

Execution Model: We assume that each node has a unique identifier and the set of identifiers is totally ordered; we assume identifiers are 1 through n, for convenience. The execution of the protocol at each node is managed by a synchronous scheduler (daemon), that selects all privileged nodes in a system state to move synchronously and atomically (we use composite atomicity as opposed to read/write atomicity [17]) in each step; a synchronous step is also called a *round* [4, 5, 6, 7, 16]. Note that such a round is different from the concept of a round used in fair central or distributed daemons. We denote a global system state, the union of the local states of all nodes, by Σ_i, $i = 0, 1, 2, \cdots$, where Σ_0 denotes the initial arbitrary state and Σ_r denotes the system state after the r-th round of the protocol, $r = 1, 2, \cdots$; r-th round executes on Σ_{r-1} to generate Σ_r.

A node is privileged in a given system state iff it is enabled to move by at least one rule of the protocol. The protocol *terminates* in a system state when no node is privileged. The protocol assumes a *shared-memory model* and each node knows only its own state and the local states of its immediate neighbors (distance-one model) as is customary in the most self-stabilizing algorithms.

3 Maximal 2-packing with Safe Convergence

In our proposed self-stabilizing maximal 2-packing algorithm with safe convergence (we call it algorithm `M2PSC`), each node $i, 1 \leq i \leq n$, maintains the following variables:

- A boolean flag $\boldsymbol{s_i}$; at any time (system state) $\mathcal{S}$ is the current set of nodes with $s_i = 1$.
- A nonnegative integer variable $\boldsymbol{c_i}$ to count the number of $\mathcal{S}$ nodes in the closed neighborhood of node i, i.e., $c_i = |N[i] \cap \mathcal{S}|$, at any given system state.
- A pointer $\boldsymbol{p_i}$ (which may be null) that points to a node $j \in N[i]$, indicated by $p_i = j$. If, in a system state, $p_i = i$ for a node i, we say node i has a **self-pointer**.
- A boolean flag $\boldsymbol{d_i}$; node i sets this bit to delay some activity by one round only.

Definition 2

1. A node i is called **consistent** in a (global) system state if $|N[i] \cap \mathcal{S}| \leq 1$.
2. A system state is **safe** if $\mathcal{S} = \{i | i \in V \wedge s_i = 1\}$ denotes a 2-packing, i.e, each node in V is consistent. A system state is **legitimate** if $\mathcal{S}$ denotes a maximal 2-packing.

3. In any system state, $minSP_i$ of a node i, $1 \leq i \leq n$, is defined to be the smallest ID node among the nodes in $N[i]$ with self-pointer, i.e., $minSP_i = \min\{j | j \in N[i] \wedge p_j = j\}$, where $\min\{\} = \text{null}$.

Remark 1 ***(Node Consistency)***

1. The stored variable c_i at node i is a measure of the local consistency of node i in a system state. If c_i is correct in a system state, i.e., $c_i = |N[i] \cap \mathcal{S}|$, node i is **consistent** iff $c_i \leq 1$.
2. If c_i is not known to be correct in a system state, node i is **deemed to be inconsistent** iff $c_i \geq 2$.

The approach underlying the algorithm `M2PSC` is to quickly converge to a safe state, by allowing nodes to exit $\mathcal{S}$ to eliminate all inconsistent nodes in the system state and thereafter to transition through safe states, by allowing nodes only to enter $\mathcal{S}$ appropriately (so that inconsistencies are not introduced), to reach the legitimate state to obtain the maximal 2-packing. We assume a synchronous daemon where at any round all privileged nodes are selected to move. The underlying approach consists of two logical sets of actions:

3.1 Exit $\mathcal{S}$

A node $i \in \mathcal{S}$ exits $\mathcal{S}$ in a round of execution of the protocol iff at least one neighbor $j \in N(i)$ is deemed to be inconsistent (Remark 2.2), as evidenced by the content of the variable c_j; the rationale is that exiting of i may not decrease but never increase the number of inconsistent nodes $j \in N[i]$ even c_j's are erroneous in a system state causing i to exit $\mathcal{S}$ (Remark 2.2). Also, by the same reason, simultaneous exit of multiple nodes from $\mathcal{S}$ cannot increase the number of inconsistent nodes in the system; our objective is to reach a safe state (Definition 2.2) as quickly as possible starting from an illegitimate state.

Definition 3 For a node i, a Boolean predicate $\texttt{nowExit}_i = 1$ iff $i \in \mathcal{S}$ and at least one of its neighbors is deemed to be inconsistent (Remark 2.2):

$$\texttt{nowExit}_i \overset{\text{def}}{\equiv} (s_i = 1) \wedge (\exists j \in N(i) : c_j \geq 2)$$

3.2 Enter $\mathcal{S}$

The protocol requires that *after a node $i \notin \mathcal{S}$ enters $\mathcal{S}$ in a round, node i is guaranteed to be the unique $\mathcal{S}$ node within its 2-hop neighborhood at the end of current round*, and thus each node $j \in N[i]$ remains consistent after node i enters $\mathcal{S}$. To accomplish this requirement we use a locking mechanism and delay technique.

A locking mechanism (implemented by stored variable p_i) is employed such that when a node enters $\mathcal{S}$ in a round, all other nodes in its 2-hop neighborhood are prohibited to enter $\mathcal{S}$. Specifically, when a node i needs to enter $\mathcal{S}$, it first requests a lock by setting self-pointer (i.e., $p_i = i$). We say node i is *locked* or gets the lock iff all nodes in $N[i]$ point to i. The neighbor j of i grants the lock by updating its pointer to i, i.e., $p_j = i$. It is possible that two adjacent nodes request locks simultaneously. In order to break the tie, the node grants the lock (by updating its pointer) to the smallest ID neighbor with self-pointer.

After a node i becomes locked, it sets d bit to delay its enter move by one round only. It should be noted that in delayed round: (a) each node $j \in N[i]$ must have no locked neighbor except i (See Definition 4.2 below) and hence no neighbor entering $\mathcal{S}$; (b) each neighbor $j \in N[i]$ gets chance to update c_j, such that $c_j \geq |N[j] \cap \mathcal{S}|$ are guaranteed when node i enters (it is possible that some neighbor of $j \in N[i]$ exits $\mathcal{S}$ in delayed round, hence the inequality). After the delayed round, node i is guaranteed to be safe to enter (i.e., after node i enters $\mathcal{S}$, it is the unique $\mathcal{S}$ node within its 2-hop neighborhood of node i, and hence each node $j \in N[i]$ remains consistent).

We begin by defining a few predicates in the following two definitions to facilitate the stepwise development of the proposed protocol.

Definition 4 In a system state, a node i can locally compute each of the following Boolean predicates:

1. For a node i, a Boolean predicate $\texttt{needEnter}_i = 1$ iff $i \notin \mathcal{S}$ and there does not exist $\mathcal{S}$ node within distance-2 of node i, as evidenced by the content of the variable c_j on each $j \in N(i)$:

$$\texttt{needEnter}_i \overset{\text{def}}{\equiv} (s_i = 0) \wedge (\forall j \in N(i) : c_j = 0)$$

2. To implement **locking mechanism**, two more Boolean predicates $\texttt{requestLock}_i$ and $\texttt{locked}_i$ on node i are defined as:

$$\texttt{requestLock}_i \overset{\text{def}}{\equiv} \texttt{needEnter}_i \wedge (\forall j \in N[i] : p_j = \mathit{null})$$

$$\texttt{locked}_i \overset{\text{def}}{\equiv} \forall j \in N[i] : p_j = i$$

Note: In a system state, if node i is locked, no node in $N^2(i)$ can be locked in the same state.

3. In any system state, a node i **requests a delay** iff the Boolean predicate $\texttt{requestDelay}_i = 1$ where

$$\texttt{requestDelay}_i \overset{\text{def}}{\equiv} (d_i = 0) \wedge \texttt{needEnter}_i \wedge \texttt{locked}_i$$

4. In any system state, a node i can **enter** $\mathcal{S}$ iff the Boolean predicate $\texttt{nowEnter}_i = 1$ where

$$\texttt{nowEnter}_i \overset{\text{def}}{\equiv} (d_i = 1) \wedge \texttt{needEnter}_i \wedge \texttt{locked}_i$$

Note: In any system state, if a node i is ready to enter ($\texttt{nowEnter}_i = 1$), no node in $N^2(i)$ is ready to enter in the same state since no node in $N^2(i)$ can be locked in that state (Definition 4.2).

Definition 5 For a node i in any system state:

1. The predicate $\texttt{updateC}_i$ is true iff its c_i is not correct, i.e.,

$$\texttt{updateC}_i \overset{\text{def}}{\equiv} c_i \neq |N[i] \cap \mathcal{S}|$$

2. The predicate $\texttt{updateP}_i$ is true iff its p_i is not equal to minSP_i, i.e.,

$$\texttt{updateP}_i \overset{\text{def}}{\equiv} p_i \neq \mathit{minSP}_i$$

3. The predicate $\texttt{clearD}_i$ is true iff $d_i = 1$ and it does not need to enter $\mathcal{S}$ or is not locked, i.e.,

$$\texttt{clearD}_i \overset{\text{def}}{\equiv} (d_i = 1) \wedge \neg(\texttt{needEnter}_i \wedge \texttt{locked}_i)$$

4. The predicate $\texttt{releaseLock}_i$ is true iff it has the self pointer and its pointer is equal to minSP_i but does not need to enter $\mathcal{S}$, i.e.,

$$\texttt{releaseLock}_i \overset{\text{def}}{\equiv} \neg\texttt{needEnter}_i \wedge (p_i = i) \wedge (\mathit{minSP}_i = i)$$

The complete pseudo code of algorithm M2PSC is shown in Figure 2. We highlight a few simple characteristics of the algorithm in the following remark.

Remark 2 In a given round r, $r \geq 1$, of execution:

1. If node i has incorrect c_i in Σ_{r-1}, it must update c_i in round r.

```
RA: if nowExit_i ∨ requestLock_i ∨ releaseLock_i ∨ updateP_i ∨ updateC_i ∨ clearD_i
        then { if nowExit_i
                   then $s_i \leftarrow 0$;                                   [Exit $\mathcal{S}$]
               if requestLock_i
                   then $p_i \leftarrow i$;                                   [Request_Lock]
               if releaseLock_i
                   then $p_i \leftarrow null$;                                [Release_Lock]
               if updateP_i
                   then $p_i \leftarrow minSP_i$;                             [Update_Pointer]
               $c_i \leftarrow |N[i] \cap \mathcal{S}|$;                      [Update_Counter]
               $d_i \leftarrow 0$;                                            [Clear_Delay]
RB: if requestDelay_i
        then { $d_i \leftarrow 1$; $c_i \leftarrow |N[i] \cap \mathcal{S}|$;  [Request_Delay]
RC: if nowEnter_i
        then { $s_i \leftarrow 1$; $p_i \leftarrow null$; $d_i \leftarrow 0$; $c_i \leftarrow |N[i] \cap \mathcal{S}|$;   [Enter $\mathcal{S}$]
```

Figure 2: The Algorithm M2PSC at Node i, $1 \leq i \leq n$

2. For a node i, nowEnter$_i$, requestDelay$_i$ and nowExit$_i$ are pairwise mutual exclusive; requestLock$_i$, releaseLock$_i$ and updateP$_i$ are pairwise mutual exclusive.

3. The membership of node i is changed only by rules RA (Exit $\mathcal{S}$ move) and RC (Enter $\mathcal{S}$ move). If a node i is privileged to make Exit $\mathcal{S}$ move, it must exit $\mathcal{S}$ successfully under synchronous daemon (see part(2)).

4. If node i exits $\mathcal{S}$, its neighboring nodes can concurrently exits $\mathcal{S}$ if they are eligible to do so; If node i enters $\mathcal{S}$, no node $j \in N^2(i)$ can concurrently enter $\mathcal{S}$ in the same round (Definition 4.4).

5. A node i can acquire a self-pointer ($p_i = i$) only by making Request_Lock move in rule RA when it needs to enter $\mathcal{S}$ to maximize $|\mathcal{S}|$ and all its neighbors have *null* pointers. **Note**: a node cannot acquire a self-pointer by making Update_Pointer move in rule RA.

6. A node i releases its self-pointer when it does not need to enter $\mathcal{S}$ (needEnter$_i = 0$) by making Release_Lock move, or when it has at least one smaller ID neighbor with self-pointer by making Update_Pointer move.

7. After a node i with needEnter$_i = 1$ becomes locked, i.e., locked$_i = 1$, it delays its enter move by one round only by making Request_Delay move (setting $d_i = 1$) such that its neighbors have time to correct their c-variables.

8. If $d_i = 1$ in Σ_{r-1}, then node i will clear delay by making either Enter $\mathcal{S}$ or Clear_Delay move such that $d_i = 0$ in Σ_r (Definitions 4.4 and 5.3).

Definition 6 In any system state $\Sigma_r, r \geq 0$:

1. A node i is **privileged** if it is enabled by any of the rules of the algorithm.

2. The execution of the algorithm **terminates** when no node is privileged.

We first prove that $\mathcal{S}$ is a maximal 2-packing when algorithm M2PSC terminates, and then we show the algorithm is safely converging in the sense that starting from an arbitrary state, it first converges to a 2-packing (a safe state) in 3 rounds, and then stabilizes in a maximal one (the legitimate state) in $O(n)$ rounds without breaking safety, where n is the number of nodes.

Lemma 1 *If algorithm M2PSC terminates, then for each node $i \in V$*

(a) c_i is correct, i.e., $c_i = |N[i] \cap \mathcal{S}|$.

(b) $d_i = 0$.

(c) $p_i = null$.

(d) $\texttt{nowExit}_i = 0$ *and* $\texttt{needEnter}_i = 0$.

Proof: (a) This lemma immediately follows from the fact that no node is privileged by the rule RA at the termination of the algorithm.

(b) Assume, by contradiction, there exists some node(s) j with $d_j = 1$. Node j must have $\texttt{needEnter}_j = 1$ and $\texttt{locked}_j = 1$ (otherwise node j is privileged by rule RA to make Clear_Delay move). Thus, node j is privileged by rule RC, a contradiction.

(c) If no node j has self-pointer, then $minSP_i = null$ for all $i \in V$ and hence the lemma holds since $p_i = minSP_i$ (otherwise node i is privileged by rule RA to make Update_Pointer move). So the key point here is to show that there is no node with self-pointer. Assume, by contradiction, there exists some node(s) with self-pointer. Consider the node with minimum ID from among those nodes, say node j; $minSP_k = j$ for each node $k \in N[j]$. Also each node $k \in N[j]$ must have $p_k = minSP_k = j$ (otherwise node k would be privileged by the rule RA to make Update_Pointer move). Thus, node j is locked (i.e., $\texttt{locked}_j = 1$). Also, node j must have $\texttt{needEnter}_j = 1$ (otherwise node j is privileged by rule RA to make Release_Lock move). Thus, we get node j is privileged by rule RB to make Request_Delay move (by part(b)), a contradiction.

(d) No node i is privileged by rule RA to make Exit $\mathcal{S}$ move and Request_lock move; the claim follows from parts (a) and (c). □

Theorem 1 *Starting from an arbitrary system state, if algorithm* `M2PSC` *terminates using synchronous daemon, then* $\mathcal{S}$ *is a maximal 2-packing.*

Proof: First, we show $\mathcal{S}$ is a 2-packing. Assume, by contradiction, $\mathcal{S}$ is not 2-packing, i.e., there exists some node(s) i such that $|N[i] \cap \mathcal{S}| \geq 2$, i.e., $c_i \geq 2$. Thus $\texttt{nowExit}_j = 1$ for each $j \in N(i) \cap \mathcal{S}$; node j is privileged by the rule RA (Exit $\mathcal{S}$), a contradiction. Thus $\mathcal{S}$ is a 2-packing.

Next, we claim $\mathcal{S}$ is maximal. Assume otherwise, i.e., there exist some node(s) $i \in \{V - \mathcal{S}\}$ such that $\nexists j \in \mathcal{S} : dist(i, j) \leq 2$. Thus $\texttt{needEnter}_i = 1$ and node i is privileged by RA to make Request_Lock move (by Lemma 1(c)), a contradiction. □

Lemma 2 *In any system state* Σ_r, $r \geq 1$, *if a node* i *is enabled by the rule* RC *to enter* $\mathcal{S}$, *each node* $j \in N[i]$ *must have* $c_j \geq |N[j] \cap \mathcal{S}|$.

Proof: In the system state Σ_{r-1}, node i must have had $d_i = 0$, $\texttt{needExit}_i = 1$ and $\texttt{locked}_i = 1$; otherwise it is impossible for node i to have $d_i = 1$ in Σ_r (Remark 3.8 and Request_Delay move). Since $\texttt{locked}_i = 1$ in Σ_{r-1}, no node $j \in N^2(i)$ was locked (Definition 4.2) and hence entered $\mathcal{S}$ in round r. Coupled with the fact that each node $j \in N[i]$ corrected its c_j in round r (Remark 3.1), thus, in Σ_r, each node $j \in N[i]$ must have $c_j \geq |N[j] \cap \mathcal{S}|$. **Note:** it is possible that some neighbor of node j exits $\mathcal{S}$ in round r (hence the inequality). □

Lemma 3 *In round* $r \geq 2$, *if a node* i *enters* $\mathcal{S}$ *(by executing rule* RC*), each node* $j \in N[i]$ *has* $|N[j] \cap \mathcal{S}| = 1$ *at the end of round.*

Proof: If node i enters $\mathcal{S}$ in round r, then node $j \in N[i]$ must have had $c_j = 0$ in Σ_{r-1} (Definition 4.4), and thus $|N[j] \cap \mathcal{S}| = 0$ by Lemma 2. Coupled with the fact that no other neighbors of node $j \in N[i]$ can enter $\mathcal{S}$ in the same round by Definition 4.4, the lemma holds. □

Lemma 4 *At the end of round* $r \geq 2$, *if there exists some node(s)* i *such that* $|N[i] \cap \mathcal{S}| \geq 2$, *then* c_i *must be* ≥ 2.

Proof: Consider a node i with $|N[i] \cap \mathcal{S}| \geq 2$ at the end of round $r \geq 2$. We observe that no neighbor of node i entered $\mathcal{S}$ in round r (otherwise $|N[i] \cap \mathcal{S}|$ would be 1 at the end of round r by Lemma 3). But some neighbor(s) of node i may exit $\mathcal{S}$ in round r. Thus node i must have $c_i \geq |N[i] \cap \mathcal{S}| \geq 2$ at the end of round r by Remark 3.1. □

Theorem 2 *Starting from any initial illegitimate state, algorithm* `M2PSC` *converges to a safe state ($\mathcal{S}$ denotes a 2-packing, i.e., each node i has $|N[i] \cap \mathcal{S}| \leq 1$) after 3 rounds.*

Proof: We show that each node i has $|N[i] \cap \mathcal{S}| \leq 1$ after 3 rounds of execution; we consider three cases:

(a) Consider any node i with $|N[i] \cap \mathcal{S}| = 0$ in Σ_2: If any node $j \in N[i]$ enters $\mathcal{S}$ in the round 3, by Lemma 3 node i will still have $|N[i] \cap \mathcal{S}| \leq 1$ after round 3.

(b) Consider any node i with $|N[i] \cap \mathcal{S}| = 1$ in Σ_2: no node $j \in N[i]$ enters $\mathcal{S}$ in the round 3 (Definition 4.4 and Lemma 2), node i will still have $|N[i] \cap \mathcal{S}| \leq 1$ after round 3.

(c) Consider any node i with $|N[i] \cap \mathcal{S}| \geq 2$ in Σ_2: c_i must be ≥ 2 by Lemma 4, each neighbor j of node i has $\texttt{nowExit}_j = 1$. Thus, in the round 3 each neighbor j of node i must exit $\mathcal{S}$ by executing the rule RA (Remark 3.3). It follows that node i will have $|N[i] \cap \mathcal{S}| \leq 1$ after round 3. □

Theorem 3 *After round 3, algorithm* `M2PSC` *maintains safety in all subsequent rounds before converging to a legitimate state.*

Proof: After round 3, we reach a safe state. In any safe state, any node i in V has $|N[i] \cap \mathcal{S}| \leq 1$. If any neighbor of i enters $\mathcal{S}$, node i will remain having $|N[i] \cap \mathcal{S}| \leq 1$ in the next state by Lemma 3, thus we reach another safe state. □

Lemma 5 *Starting from a safe state Σ_r, $r \geq 4$, no node will ever make Exit $\mathcal{S}$ move in subsequent rounds.*

Proof: In a safe state, each node $i \in V$ has $|N[i] \cap \mathcal{S}| \leq 1$. The algorithm `M2PSC` always in the safe state after round 3 (Theorem 3), thus each node i always has $|N[i] \cap \mathcal{S}| \leq 1$ after round 3 and each node i always has $c_i \leq 1$ after round 4 by Remark 3.1. The lemma holds. □

Lemma 6 *In any safe state Σ_r, $r \geq 5$, rules* RA, RB *and* RC *are pairwise mutual exclusive.*

Proof: It is easy to show `requestDelay` and `nowEnter` are pairwise mutual exclusive with each of `nowExit`, `requestLock`, `releaseLock`, `updateP` and `clearD`, we here omit the details. Coupled with the fact that `requestDelay` and `nowEnter` are pairwise mutual exclusive (Remark 3.2). Thus, to prove the lemma, it suffices that show that `requestDelay` and `nowEnter` are pairwise mutual exclusive with `updateC`.

For any node i, if $\texttt{requestDelay}_i = 1$ or $\texttt{nowEnter}_i = 1$ in Σ_r, no node $j \in N[i]$ entered $\mathcal{S}$ in the round r (otherwise node i cannot have self-pointer in Σ_r). Coupled with Lemma 5 and Remark 3.1, c_i must be correct in Σ_r. Thus, $\texttt{requestDelay}_i$ and $\texttt{nowEnter}_i$ are pairwise mutual exclusive with $\texttt{updateC}_i$ in Σ_r. □

Lemma 7 *In any system state Σ_r where $r \geq 1$, two adjacent nodes i and j have self-pointers (i.e., $p_i = i$ and $p_j = j$), then (a) the larger ID node will lose the self-pointer (i.e., if say $i < j$, $p_j \neq j$) in the next rounds; (b) nodes i and j must have concurrently acquired the self-pointers in round r.*

Proof: (a) In Σ_r, node j has $p_j = j \neq minSP_j$ and thus $\texttt{UpdateP}_j = 1$. Coupled with the fact that $\texttt{UpdateP}_j$ is pairwise mutual exclusive with each of $\texttt{requestDelay}_j$ and $\texttt{nowEnter}_j$, in the next round node j must be selected by daemon to make Update_Pointer move in rule RA. (b) If $p_i = i$ but $p_j \neq j$, then node j cannot make Request_Lock move to get the self-pointer in the next state (Definition 4.2). □

Lemma 8 *In a safe state Σ_r, $r \geq 3$, for two adjacent nodes i and j, if $p_i = i$ and $p_j = j$, then* $\texttt{needEnter}_i = \texttt{needEnter}_j = 1$.

Proof: It follows from Lemma 7 that nodes i and j had concurrently made Request_Lock move to get self-pointers in round r. In Σ_{r-1}, $\texttt{needEnter}_i = 1$, $\texttt{needEnter}_j = 1$, $p_i = p_j = \mathit{null}$, and for each $k \in N(i) \cup N(j)$, $p_k = \mathit{null}$ (nodes i and j are enabled for rule RA to make Request_Lock move; Definition 4.2). We argue that:

(a) Node i cannot enter $\mathcal{S}$ (execute rule RC) in round r since $\texttt{locked}_i = 0$ in Σ_{r-1} ($p_i = \mathit{null}$).

(b) Any node $k \in N(i)$ cannot enter $\mathcal{S}$ (execute rule RC) in round r since $\texttt{locked}_k = 0$ in Σ_{r-1} ($p_k = \mathit{null}$).

(c) Any neighbor k' of $k \in N(i)$ cannot enter $\mathcal{S}$ (execute rule RC) in round r since $\texttt{locked}_{k'} = 0$ in Σ_{r-1} ($p_k = \mathit{null}$).

Thus, $\texttt{needEnter}_i$ remains 1 in Σ_r (by similar reasoning, $\texttt{needEnter}_j$ remains 1 in Σ_r). □

Definition 7 In any system state,

1. We define an **island** $\mathcal{I}$ to be a *maximal* set of nodes $\{i \in V | \texttt{needEnter}_i = 1 \wedge p_i = i\}$ such that the subgraph of G induced by the set $\mathcal{I}$ is connected.
2. We use $\boldsymbol{\alpha}$ to denote the number of islands and $\boldsymbol{\beta}$ to denote the number of nodes i with $\texttt{needEnter}_i = 1$.

Remark 3 In any system state:

1. An island may consist of a single or multiple nodes; a node i with $\texttt{needEnter}_i = 1$ and $p_i \neq i$ is not a member of any island.
2. For a node i in an island of size ≥ 2, $\texttt{locked}_i = 0$ since it has a neighbor j with $p_j = j \neq i$ (Definition 4.2).
3. $\alpha \leq \beta$; $\alpha < n$; $\beta \leq n$;
4. After round 4, β is non increasing in subsequent rounds (Definition 4.1 and Lemma 5).
5. When algorithm $\texttt{M2PSC}$ terminates, $\alpha = \beta = 0$.

Lemma 9 *If a node i enters $\mathcal{S}$ (by executing rule* RC*) in a round, node i constitutes a single node island at the beginning of the round.*

Proof: Node i enters $\mathcal{S}$; thus $\texttt{needEnter}_i = 1$ and $\texttt{locked}_i = 1$ (rule RC). Since $p_i = i$, node i belongs to an island (Definition 7.1); node i does not belong to an island of size ≥ 2 (Remark 4.2). □

Lemma 10 *In any round r, $r \geq 5$ (starting from a safe state Σ_{r-1}), (a) α cannot decrease if β remains constant; (b) α decreases at least by 1 and at most by ℓ, if β decreases by ℓ ($1 \leq \ell \leq \beta$).*

Proof: (a) If β remains constant, no node i changes $\texttt{needEnter}_i$ from 1 to 0. (1) Any island $\mathcal{I}$ cannot disappear since the smallest ID node in $\mathcal{I}$, say node i, cannot change its pointer in round r ($p_i = i = \mathit{minSP}_i$ in Σ_{r-1} [no neighbor j of i with $\texttt{needEnter}_j = 0$ has a self-pointer by Lemma 8 and node i does not have any island node neighbor with a smaller ID]). (2) Two islands cannot merge into one: consider any 2 islands $\mathcal{I}_1$ and $\mathcal{I}_2$; since $\mathcal{I}_1 \cup \mathcal{I}_2 = \emptyset$, for the two islands to merge, there must be a node $j \in N(\mathcal{I}_1 \cup \mathcal{I}_2)$ such that $\texttt{needEnter}_j = 1$ in Σ_{r-1} and node j acquires self-pointer in Σ_r (j becomes an island node in Σ_r) by executing rule RA to make Request_Lock move in round r; this is impossible since j has neighbor(s) with non null pointers in Σ_{r-1} (see Definition 4.2).

(b) Starting in a safe state Σ_{r-1}, if β decreases by ℓ in Σ_r, ℓ nodes have changed their $\texttt{needEnter}$ bits from 1 to 0. Consider any node i whose $\texttt{needEnter}_i$ is changed from 1 to 0. At least one of the two must occur in round r (Definition 4.1): (1) node i enters $\mathcal{S}$ by executing rule RC; (2) some neighbor(s) of node $j \in N(i)$ enters $\mathcal{S}$ by executing rule RC such that $c_j > 0$. If all ℓ nodes change their `needEnter` from 1 to 0 because of (1), then α is decreased by ℓ by Lemma 9; If some node(s) i changes $\texttt{needEnter}_i$ from 1 to 0 because of (2), then it is possible that node i does not belong to any island. Although the change of $\texttt{needEnter}_i$ on node i causes β to decrease in Σ_r, it does not cause α to decrease (if node i is not an island node in Σ_{r-1}); but, for the possibilities (2), at least some other node must enter $\mathcal{S}$ (change s bit to 1) by executing rule RC in the round, thereby causing α to decrease (Lemma 9). Thus, α decreases by at most ℓ and at least by 1 in Σ_r. □

Lemma 11 *Starting from a safe state Σ_r, $r \geq 4$, with $\beta \neq 0$,*

(a) either α increases in at most 3 next rounds, if β remains constant;

(b) or β decreases in at most 4 next rounds.

Proof: Starting from a safe state Σ_r, $r \geq 4$, any node i with $\texttt{needEnter}_i = 0$ and $p_i = i$ must either make Update_Pointer or Release_Lock move in round $r+1$ to make $p_i \neq i$ in Σ_{r+1}. Then, Σ_{r+1} does not have a node j with $\texttt{needEnter}_j = 0 \wedge p_j = j$ (otherwise, node j had $\texttt{needEnter}_j = 1$ in Σ_r and hence β has decreased by at least 1 in one round). There are two possibilities:

(1) **There is no island node:** In Σ_{r+1}, each node i has $minSP_i = null$ (no node with self-pointer and Definition 2.3). Also, since $\beta \neq 0$, there must be a node k with $\texttt{needEnter}_k = 1$; node k, in the worst case, must make Update_Pointer move in round $r+2$ and Request_Lock move in round $r+3$ in that sequence to get $p_k = k$. Thus, there is a new island $\{k\}$, i.e., α has increased in at most 3 rounds starting in Σ_r.

(2) **There is at least one island node**: If there are multiple such island nodes, let i be the node with minimum ID among those. In the worst case, each node $j \in N(i)$ makes Update_Pointer move in round $r+2$ to update their pointers to i; node i becomes locked, i.e., $\texttt{locked}_i = 1$ (Definition 4.2), in Σ_{r+2}. Also, if $d_i = 1$ in Σ_{r+1}, node i makes Clear_Delay move in round $r+2$ such that $d_i = 0$ in Σ_{r+2}. Now, there are two possibilities:

(i) At least one $j \in N(i)$ has $minSP_j = k$ in Σ_{r+2} where $k \in N(j)$ (Definition 2.3), $p_k = k$, and $k < i$. Node k must have acquired its self-pointer by making Request_Lock move in round $r+2$ and so, $\texttt{needEnter}_k = 1 \wedge (\forall k' \in N(k) : p_{k'} = null)$ in Σ_{r+1}, i.e., node k is not connected to any island nodes. Thus, $\{k\}$ is a newly formed single node island in Σ_{r+2}, i.e., α increases in at most 2 rounds.

(ii) Each $j \in N(i)$ has $minSP_j = i$ in Σ_{r+2}; in round $r+3$, node i makes Request_Delay move to delay its Enter move by one round only. Thus, node i executes rule RC to enter $\mathcal{S}$ in round $r+4$; so $\texttt{needEnter}_i = 0$ in Σ_{r+4}, i.e., β decreases in at most 4 rounds.

□

Lemma 12 *After round 4, algorithm* `M2PSC` *reaches a safe state with $\alpha = \beta = 0$ in at most $7n$ rounds under a synchronous daemon.*

Proof: In a safe state where $\alpha = \beta$, if β decreases by 1, α must decrease by 1 (Lemma 10); $\beta \leq n$ and β is non-increasing (Remarks 4.3 and 4.4). Recall that β decreases by at least 1 in at most 4 rounds (Lemma 11(b)). Thus, from any safe system state with $\alpha = \beta$, the system will be in a safe state with $\alpha = \beta = 0$ in at most $4n$ rounds. Also, if β remains constant, α must increase by 1 in at most 3 rounds (Lemma 11(a)); in at most $3n$ rounds, α will be equal to β. Thus, the system will be in a safe state with $\alpha = \beta = 0$ in at most $4n + 3n = 7n$ rounds. □

Theorem 4 *Starting in any arbitrary state, the algorithm* `M2PSC` *terminates in at most $O(n)$ rounds under a synchronous daemon.*

Proof: The system reaches a safe state with $\alpha = \beta = 0$ in $7n + 4 = O(n)$ rounds in the worst case (Theorem 2 and Lemma 12). In the next round: all node pointers will be *null*, all c variables will be correct and all d variables will be 0; thus the algorithm terminates. □

4 Maximal k-packing with Safe Convergence

We generalize the basic idea of algorithm `M2PSC` to propose a self-stabilizing maximal k-packing algorithm with safe convergence (we call it algorithm `MKPSC`). As before, starting from an arbitrary state, the system first converges to a k-packing (a safe state) by allowing nodes to exit $\mathcal{S}$ quickly, and thereafter moves through safe states until $\mathcal{S}$ is a maximal k-packing (the legitimate state) by allowing nodes only to enter $\mathcal{S}$ appropriately. We assume a synchronous daemon where in any round all privileged nodes are selected to move. In algorithm `MKPSC`, each node $i, 1 \le i \le n$, maintains the following variables:

- An array of nonnegative integer sets $\boldsymbol{\hat{T}_i[0,...,k-1]}$; $\hat{T}_i[\ell]$, $0 \le \ell < k$, is intended to keep track of the IDs of $\mathcal{S}$-nodes (as a set) in $N^{\ell}[i]$, ℓ-hop closed neighborhood of node i in any system state. In a system state $\mathcal{S}$ is the current set of nodes i with $\hat{T}_i[0] = \{i\}$.
- A pointer array $\boldsymbol{\hat{P}_i[0,...,k-1]}$; $\hat{P}_i[\ell]$ (which may be null) points to a node j, indicated by $\hat{P}_i[\ell] = j$. We say node i has a **self-pointer** iff $\hat{P}_i[0] = i$; $\hat{P}_i[\ell]$, $0 \le \ell < k$, keeps track of the minimum ID node with self-pointer in $N^{\ell}[i]$ at any system state.
- A nonnegative integer variable $\boldsymbol{d_i}$; node i uses this variable to delay some activity by $2k$ rounds.

Remark 4 In any system state, for any node i, (a) $\hat{T}_i$ is **correct** iff (1) $\hat{T}_i[0]$ is either $\{i\}$ or $\emptyset$, and (2) for each $\ell, 1 \le \ell \le (k-1)$, $\hat{T}_i[\ell] = \bigcup\limits_{j \in N[i]} \hat{T}_j[\ell-1]$; (b) $\hat{P}_i$ is **correct** iff (1) $\hat{P}_i[0]$ is either i or *null*, and (2) for each $\ell, 1 \le \ell \le (k-1)$, $\hat{P}_i[\ell] = \min\limits_{j \in N[i]} \hat{P}_j[\ell-1]$.

Definition 8 A system state is **safe** if $\mathcal{S} = \big\{i | i \in V \wedge \hat{T}_i[0] = \{i\}\big\}$ denotes a k-packing, i.e, each node $i \in \mathcal{S}$ is the unique $\mathcal{S}$-node within distance-k of node i. A system state is **legitimate** if $\mathcal{S}$ denotes a maximal k-packing.

The underlying approach consists of two logical sets of actions: (a) **Exit $\boldsymbol{\mathcal{S}}$**: The protocol requires that A node $i \in \mathcal{S}$ exits $\mathcal{S}$ in a round of execution of the protocol iff there exists some $\mathcal{S}$-node(s) within distance-k of node i, as evidenced by the content of the variable $\hat{T}_j[k-1]$ at each node $j \in N(i)$. This quick exit of nodes from $\mathcal{S}$ facilitates quick convergence to a safe state. (b) **Enter $\boldsymbol{\mathcal{S}}$**: Once in a safe state, a node i needs to enter $\mathcal{S}$ without violating safety in the process to eventually converge to a legitimate state ($\mathcal{S}$ is a maximal k-packing). The protocol requires that *after a node $i \notin \mathcal{S}$ enters $\mathcal{S}$ in a round, node i is guaranteed to be the unique $\mathcal{S}$-node within distance-k of node i at the end of current round.* To accomplish this requirement we generalize the concepts of locking mechanism and delay technique in algorithm `M2PSC` such that (1) when a node enters $\mathcal{S}$ in a round, all other nodes in its k-hop neighborhood are prohibited to enter $\mathcal{S}$ in the same round; (2) no $\mathcal{S}$-node exists in k-hop neighborhood of node i at the beginning of the round, i.e., after node i intends to enter $\mathcal{S}$, it must communicate with nodes in its k-hop neighborhood to make sure $|N^{k-1}[j] \cap \mathcal{S}| = 0$ for each $j \in N[i]$ and none of those nodes enter $\mathcal{S}$; (3) in case more than one node in a k-hop neighborhood intends to enter $\mathcal{S}$, a tie resolution mechanism is needed. The entire process takes $2k$ rounds in the proposed protocol and is facilitated by the delay variable d_i. We need to define a few predicates to facilitate the stepwise development of the proposed protocol.

Definition 9 In any system state, a Boolean predicate $\texttt{nowExit}_i = 1$ for a node i is defined as

$$\texttt{nowExit}_i \overset{\text{def}}{\equiv} (\hat{T}_i[0] = \{i\}) \wedge (\exists j \in N(i) : |\hat{T}_j[k-1]| \ge 2)$$

Definition 10 In a system state, for a node i

1. The Boolean predicate $\texttt{needEnter}_i = 1$ iff $i \notin \mathcal{S}$ and there does not exist $\mathcal{S}$-node within distance-k of node i, as evidenced by the content of the variable $\hat{T}_j[k-1]$ on each $j \in N(i)$:

$$\texttt{needEnter}_i \overset{\text{def}}{\equiv} \big(\hat{T}_i[0] \neq \{i\}\big) \wedge \big(\forall j \in N(i) : |\hat{T}_j[k-1]| = 0\big)$$

2. A node i requests a lock (to express its intent to enter $\mathcal{S}$) when its own $\hat{P}_i[0]$ is null as well as all nodes in $N[i]$ have their respective pointers (for distance $k-1$) are null. A node i is locked when it has a self-pointer as well as all nodes in $N[i]$ have their $k-1$-distance pointers pointing to i.

$$\texttt{requestLock}_i \overset{\text{def}}{\equiv} \texttt{needEnter}_i \wedge (\hat{P}_i[0] = \mathit{null}) \wedge (\forall j \in N[i] : \hat{P}_j[k-1] = \mathit{null})$$

$$\texttt{locked}_i \overset{\text{def}}{\equiv} (\hat{P}_i[0] = i) \wedge (\forall j \in N[i] : \hat{P}_j[k-1] = i)$$

3. A node i needs to **request a delay** when it needs to enter $\mathcal{S}$ and is locked and has not yet waited for $2k$ rounds to ensure correctness of $\hat{T}_i$ variables, i.e.,

$$\texttt{requestDelay}_i \overset{\text{def}}{\equiv} (d_i \neq 2k) \wedge \texttt{needEnter}_i \wedge \texttt{locked}_i$$

4. A node i can immediately **enter $\mathcal{S}$** when it has waited for $2k$ rounds maintaining its readiness to enter and locked, i.e., iff the Boolean predicate $\texttt{nowEnter}_i = 1$ where

$$\texttt{nowEnter}_i \overset{\text{def}}{\equiv} (d_i = 2k) \wedge \texttt{needEnter}_i \wedge \texttt{locked}_i$$

Definition 11 For a node i in any system state:

1. The predicate $\texttt{update}\hat{T}_i$ is true iff its $\hat{T}_i$ is not correct (Remark 4), i.e.,

$$\texttt{update}\hat{T}_i \overset{\text{def}}{\equiv} \left(\hat{T}_i[0] \neq \{i\} \wedge \hat{T}_i[0] \neq \emptyset\right) \vee \left(\hat{T}_i[\ell] \neq \bigcup_{j \in N[i]} (\hat{T}_j[\ell-1]) \text{ for some } \ell \in \{1, 2, ..., k-1\}\right)$$

2. The predicate $\texttt{update}\hat{P}_i$ is true iff its $\hat{P}_i$ is not correct (Remark 4), i.e.,

$$\texttt{update}\hat{P}_i \overset{\text{def}}{\equiv} (\hat{P}_i[0] \neq i \wedge \hat{P}_i[0] \neq \mathit{null}) \vee \left(\hat{P}_i[\ell] \neq \min_{j \in N[i]} (\hat{P}_j[\ell-1]) \text{ for some } \ell \in \{1, 2, ..., k-1\}\right)$$

3. The predicate $\texttt{clearD}_i$ is true if the delay indicator $d_i \neq 0$ and either the node i is not eligible to enter $\mathcal{S}$ or it is not locked, i.e.,

$$\texttt{clearD}_i \overset{\text{def}}{\equiv} (d_i \neq 0) \wedge \neg(\texttt{needEnter}_i \wedge \texttt{locked}_i)$$

4. The predicate $\texttt{releaseLock}_i$ is true iff it has the self pointer but it is not eligible to enter $\mathcal{S}$, i.e.,

$$\texttt{releaseLock}_i \overset{\text{def}}{\equiv} \neg\texttt{needEnter}_i \wedge (\hat{P}_i[0] = i)$$

The complete pseudo code of algorithm MKPSC is shown in Figure 3. We highlight a few simple characteristics of the algorithm in the following remark.

Remark 5 In a given round r, $r \geq 1$, of execution:

1. If node i has incorrect $\hat{T}_i$ in Σ_{r-1}, it must update $\hat{T}_i$ in round r.
2. For a node i, $\texttt{nowEnter}_i$, $\texttt{requestDelay}_i$ and $\texttt{nowExit}_i$ are pairwise mutual exclusive.
3. The membership of node i is changed only by rules RA (Exit $\mathcal{S}$ move) and RC (Enter $\mathcal{S}$ move). If a node i is privileged to make Exit $\mathcal{S}$ move, it must exit $\mathcal{S}$ successfully under synchronous daemon (see part(2)).
4. If node i exits $\mathcal{S}$, its neighboring nodes can concurrently exits $\mathcal{S}$ if they are eligible to do so.
5. A node i can acquire a self-pointer ($\hat{P}_i[0] = i$) only by making Request_Lock move in rule RA. **Note**: a node cannot acquire a self-pointer by making Update_$\hat{P}$ move in rule RA.

```
RA: if $\texttt{nowExit}_i \vee \texttt{requestLock}_i \vee \texttt{releaseLock}_i \vee \texttt{update}\hat{P}_i \vee \texttt{update}\hat{T}_i \vee \texttt{clearD}_i$
    then
        if $\texttt{nowExit}_i$
            then $\hat{T}_i[0] \leftarrow \emptyset$;                                        [Exit $\mathcal{S}$]
        if $\texttt{requestLock}_i$
            then $\hat{P}_i[0] \leftarrow i$;                                        [Request_Lock]
        if $\texttt{releaseLock}_i$
            then $\hat{P}_i[0] \leftarrow null$;                                     [Release_Lock]
        if $\texttt{update}\hat{P}_i$
            then
                if $\hat{P}_i[0] \neq i$ then $\hat{P}_i[0] = null$;
                $\hat{P}_i[\ell] \leftarrow \min_{j \in N[i]} (\hat{P}_j[\ell-1])$ for $1 \leq \ell \leq k-1$;      [Update_$\hat{P}$]
        if $\texttt{update}\hat{T}_i$
            then
                if $\hat{T}_i[0] \neq \{i\}$ then $\hat{T}_i[0] = \emptyset$;                     [Update_$\hat{T}$]
                $\hat{T}_i[\ell] \leftarrow \bigcup_{j \in N[i]} (\hat{T}_j[\ell-1])$ for $1 \leq \ell \leq k-1$;
        $d_i \leftarrow 0$;                                                    [Clear_Delay]
RB: if $\texttt{requestDelay}_i$
    then
        $d_i \leftarrow d_i + 1 \bmod (2k+1)$;
        $\hat{T}_i[0] = \emptyset$;                                             [Request_Delay]
        $\hat{T}_i[\ell] \leftarrow \bigcup_{j \in N[i]} (\hat{T}_j[\ell-1])$ for $1 \leq \ell \leq k-1$;
RC: if $\texttt{nowEnter}_i$
    then
        $\hat{T}_i[\ell] \leftarrow \{i\}$ for $0 \leq \ell \leq k-1$;
        $\hat{P}_i[0] \leftarrow null$;                                          [Enter $\mathcal{S}$]
        $d_i \leftarrow 0$;
```

Figure 3: The Algorithm MKPSC at Node i, $1 \leq i \leq n$

6. A node i releases its self-pointer when it does not need to enter $\mathcal{S}$ ($\texttt{needEnter}_i = 0$) by making Release_Lock move.

7. After a node i with $\texttt{needEnter}_i = 1$ becomes locked, i.e., $\texttt{locked}_i = 1$, it delays its enter move by $2k$ rounds by making $2k$ Request_Delay moves such that nodes within its distance-k have time to correct their $\hat{T}$ and $\hat{P}$-variables.

8. Node i either (1) increases d_i by 1 in modulo $2k+1$ by making Request_Delay move (Definitions 10.3), or (2) clears delay by making Enter or Clear_Delay move such that $d_i = 0$ in Σ_r (Definitions 10.4 and 11.3).

Lemma 13 *If algorithm MKPSC terminates, then for each node $i \in V$*

(a) If $\hat{T}_i[0] \neq \{i\}$, then $\hat{T}_i[0] = \emptyset$; $\hat{T}_i[\ell] = \bigcup_{j \in N[i]} (\hat{T}_j[\ell-1])$ for $1 \leq \ell \leq k-1$.

(b) $d_i = 0$.

(c) $\hat{P}_i[\ell] = null$ for $0 \leq \ell \leq k-1$.

(d) $\texttt{nowExit}_i = 0$ and $\texttt{needEnter}_i = 0$.

Proof: (a) This lemma immediately follows from the fact that no node is privileged by rule RA to make Update_$\hat{T}$ move at the termination of the algorithm.

(b) Assume, by contradiction, there exists some node(s) j with $d_j \neq 0$. Node j must have $\texttt{needEnter}_j = 1$ and $\texttt{locked}_j = 1$ (otherwise node j is privileged by rule RA to make Clear_Delay move). Thus, node j is privileged by rule RB or RC, a contradiction.

(c) If no node j has self-pointer (i.e., $\hat{P}_j[0] \neq j$), then $\hat{P}_i[\ell] = null$, $0 \leq \ell \leq k-1$, for all $i \in V$ (otherwise node i is privileged by rule RA to make Update_$\hat{P}$ move). So the key point here is to show that there is no node with self-pointer. Assume, by contradiction, there exists some node(s) with self-pointer. Consider the node with minimum ID from among those nodes, say node i; $\hat{P}_j[\ell] = i$, $1 \leq \ell \leq k-1$, for each node $j \in N[i]$ (otherwise node j would be privileged by the rule RA to make Update_$\hat{P}$ move). Thus, node i is locked (i.e., $\texttt{locked}_i = 1$). Also, node i must have $\texttt{needEnter}_i = 1$ (otherwise node i is privileged by rule RA to make Release_Lock move). Thus, we get node i is privileged by rule RB to make Request_Delay move (by part(b)), a contradiction.

(d) No node i is privileged by rule RA to make Exit $\mathcal{S}$ move and Request_lock move; the claim follows from parts (a) and (c). □

Theorem 5 *Starting from an arbitrary system state, if algorithm MKPSC terminates using synchronous daemon, then $\mathcal{S}$ is a maximal k-packing.*

Proof: First, we show $\mathcal{S}$ is a k-packing. Assume, by contradiction, $\mathcal{S}$ is not k-packing, i.e., there exists two $\mathcal{S}$-node(s) i and j such that the length of the shortest path between i and j, $dist(i,j)$, is $\leq k$. Consider node $p \in N(i)$ on the shortest path, it must have $|\hat{T}_p[k-1]| \geq 2$ (Lemma 13(a)), thus $\texttt{nowExit}_i = 1$; node i is privileged by the rule RA (Exit $\mathcal{S}$), a contradiction. Thus $\mathcal{S}$ is a k-packing.

Next, we claim $\mathcal{S}$ is maximal. Assume otherwise, i.e., there exist some node(s) $i \in \{V - \mathcal{S}\}$ such that $\nexists j \in \mathcal{S} : dist(i,j) \leq k$. Thus $\texttt{needEnter}_i = 1$ and node i is privileged by RA to make Request_Lock move (by Lemma 13(c)), a contradiction. □

Lemma 14 *In any system state Σ_r, $r \geq 2k$, if a node i is enabled by the rule* RC *to enter $\mathcal{S}$, (a) no node in $N^k(i)$ is enabled by the rule* RC *to enter $\mathcal{S}$ in system state Σ_{r-k} to Σ_r; (b) each node $j \in N[i]$ must have $|\hat{T}_j[k-1]| \geq |N^{k-1}[j] \cap \mathcal{S}|$.*

Proof: In the system state Σ_{r-2k}, node i must have had $d_i = 0$, $\texttt{needExit}_i = \texttt{locked}_i = 1$; Also, node i made Request_Delay move in each round from $r-2k+1$ to r [otherwise it is impossible for node i to have $d_i = 2k$ in Σ_r (Remark 6.8 and Request_Delay move)]. Thus, $\texttt{needExit}_i = \texttt{locked}_i = 1$ in system states Σ_{r-2k} to Σ_r.

(a) Node i must be the minimum ID node with self-pointer within its distance-k in Σ_{r-2k}, and no node $j < i$ within distance-k of i got the self-pointer during the rounds $[r-2k+1, r-k]$ (otherwise $\texttt{locked}_i$ cannot remain 1 in system state Σ_{r-2k} to Σ_r); Thus $\hat{P}_j[k-1] \neq j$ for all nodes j in $N^k(i)$ in Σ_{r-k}. The lemma holds.

(b) During the rounds $[r-k+1, r]$, no node j in $N^k(i)$ can Enter $\mathcal{S}$ by Part(a), and each node $j \in N[i]$ corrected its $\hat{T}_j$ (Remark 6.1); thus, in Σ_r, each node $j \in N[i]$ must have $|\hat{T}_j[k-1]| \geq |N^{k-1}[j] \cap \mathcal{S}|$). **Note:** it is possible that some node(s) within distance-k of i exits $\mathcal{S}$ during the rounds $[r-k+1, r]$ (hence the inequality). □

Lemma 15 *In round $r \geq 2k+1$, if a node i enters $\mathcal{S}$ (by executing rule* RC*), node i is guaranteed to be the unique $\mathcal{S}$-node within distance-k of node i at the end of current round.*

Proof: If node i enters $\mathcal{S}$ in round r, then node $j \in N[i]$ must have had $|\hat{T}_j[k-1]| = 0$ in Σ_{r-1} (Definition 10.4), and thus $|N^{k-1}[j] \cap \mathcal{S}| = 0$ by Lemma 14(b). Coupled with the fact that no other nodes in $j \in N^k(i)$ can enter $\mathcal{S}$ in the same round by Lemma 14(a), the lemma holds. □

Lemma 16 *At the end of round $r \geq 2k+1$, if there exists some node(s) $i \in \mathcal{S}$ such that $|N^k[i] \cap \mathcal{S}| \geq 2$, then no nodes in $N^{k-1}[i]$ can make Enter move.*

Proof: This lemma immediately follows from Lemma 15. □

Theorem 6 *Starting from any initial illegitimate state, algorithm MKPSC converges to a safe state ($\mathcal{S}$ denotes a k-packing, i.e, each node $i \in \mathcal{S}$ is the unique $\mathcal{S}$-node within distance-k of node i) after $3k+1$ rounds.*

Proof: Consider any node $i \in \mathcal{S}$ with $|N^k[i] \cap \mathcal{S}| \geq 2$ in Σ_{2k+1}: $\hat{T}_j[k-1]$ on $j \in N(i)$ must be ≥ 2 in Σ_{3k} by Lemma 16 and Remark 6.1, thus node i has $\texttt{nowExit}_i = 1$ in Σ_{3k}. In the round $3k+1$, i must exit $\mathcal{S}$ by executing the rule RA (Remark 6.3). Thus all nodes $i \in \mathcal{S}$ with $|N^k[i] \cap \mathcal{S}| \geq 2$ in Σ_{2k+1} will be out of $\mathcal{S}$ at the end of round $3k+1$. Coupled with the fact that each newly created $\mathcal{S}$-node must be the the unique $\mathcal{S}$-node within its distance-k (Lemma 15), the lemma holds. □

Theorem 7 *After round $3k+1$, algorithm* MKPSC *maintains safety in all subsequent rounds before converging to a legitimate state.*

Proof: After round $3k+1$, we reach a safe state. If any node i enters $\mathcal{S}$, then it is guaranteed to be the unique $\mathcal{S}$-node within distance-k of node i by Lemma 15; thus we reach another safe state. □

Lemma 17 *Starting from a safe state Σ_r, $r \geq 4k$, no node will ever make Exit $\mathcal{S}$ move in subsequent rounds.*

Proof: In a safe state, there are no two $\mathcal{S}$-nodes i and j such that $dist(i,j) \leq k$. The algorithm `MKPSC` always in the safe state after round $3k+1$ (Theorem 7), thus each node i always has $|N^{k-1}[i] \cap \mathcal{S}| \leq 1$ after round $3k+1$ and each node i always has $\hat{T}_i[k-1] \leq 1$ after round $4k$ by Remark 6.1. The lemma holds. □

Lemma 18 *Starting from a safe (not legitimate) state Σ_r, $r \geq 4k$, the number of $\mathcal{S}$-node increases in at most $k(n+4)+2$ rounds.*

Proof: (a) **No node makes Update_$\hat{T}$ moves after round** $r+k$: this is true because there is no Exit move after round $4k$ (Lemma 17) and each node i corrects its $\hat{T}_i$ during the rounds $[r+1, r+k]$.

(b) **No node makes Release_Lock move after round** $r+k+1$: each node i has correct $\hat{T}_i$ at the end of round $r+k$ (part(a)). After another round, all nodes i with $\texttt{needEnter}_i = 0$ release locks simultaneously.

(c) **No node makes Request_Lock and Update_$\hat{P}$ moves after round** $r+k(n+2)+1$: After round $r+k+1$, no node makes Release_Lock move, thus the number of nodes with self-pointer is non-decreasing. Each Request_Lock move increases the number of nodes with self-pointer by 1, so there are at most n Request_Lock moves after round $r+k+1$. In between any two consecutive Request_Lock moves, there are at most k Update_$\hat{P}$ moves.

(d) **No node makes Request_Delay and Clear_Delay moves after round** $r+k(n+4)+1$: After round $r+k(n+2)+1$, Request_Delay and Clear_Delay moves can be made in at most $2k$ subsequent rounds by parts (a), (b) and (c).

Thus, at least one node makes Enter move in round $r+k(n+4)+2$. □

Theorem 8 *Starting at an arbitrary state, algorithm* MKPSC *terminates in $O(kn^2)$ rounds under the synchronous daemon.*

Proof: After round $4k$, the number of $\mathcal{S}$-nodes is non-decreasing by Lemma 17, and the number of $\mathcal{S}$-nodes increases in at most $k(n+4)+2 = O(kn)$ rounds by Lemma 18. Thus, the algorithm terminates in $4k + n \times O(kn) = O(kn^2)$ rounds. □

5 Conclusion

In this paper, we propose a self-stabilizing maximal 2-packing algorithm with safe convergence using a synchronous daemon. Starting at an arbitrary state, it first converges to a 2-packing (a safe state) in three synchronous steps, and then converges to a maximal one (the legitimate state) in $O(n)$ steps without breaking safety during the self-stabilization interval, where n is the number of nodes in the network. Space requirement at each node is $O(\log n)$ bits. This is a significant improvement over the most recent self-stabilizing algorithm for maximal 2-packing in [16], that uses $O(n^2)$ synchronous steps with the same space requirement at each node and that does not have safe convergence property. We then generalize the technique to design a self-stabilizing algorithm for maximal k-packing, $k \geq 2$, with safe convergence that stabilizes in $O(kn^2)$ steps under synchronous daemon; the algorithm has space complexity of $O(kn \log n)$ bits at each node.

We note that a straightforward application of our k-packing algorithm to the case of $k = 2$ will result in a self-stabilizing algorithm of $O(n^2)$ time complexity while that of our first algorithm is $O(n)$. We observe:

- The underlying principle in both of our algorithms `M2PSC` and `MKPSC` is similar: (1) The algorithms, in the first phase, allow a node to execute an "exit $\mathcal{S}$" action if it violates the definition of k-packing ($k \geq 2$) without checking if any other node in its k-hop neighborhood is also executing an "exit $\mathcal{S}$" action, in the same step; this idea does not violate the definition of k-packing. The effect is we reach a safe state (k-packing, but not necessarily maximal) quickly. (2) In the second phase, once we have reached a safe state, we need nodes (not in $\mathcal{S}$) to enter $\mathcal{S}$ towards the goal of making the k-packing maximal such that other nodes in the k-hop neighborhood do not execute the same action concurrently to maintain the safety of the resulting global state (k-packing is maintained); thus, in the second phase, for a node to enter $\mathcal{S}$, it must make sure its k-hop neighborhood is locked (i.e., no such node is eligible to enter $\mathcal{S}$ in the same synchronous state).

- In algorithm `M2PSC` ($k = 2$), locking the 2-hop neighborhood is implemented by using the combination of the single pointer variable and the unique node IDs; this is possible because a node can be given permission to enter $\mathcal{S}$ by all nodes in its 2-hop neighborhood using their $minSP_j$ (determined by unique ID of nodes with self-pointers). This simple "locking mechanism" fails when $k > 2$. In algorithm `MKPSC`, while we generalize the basic concept, we needed more information to be stored at each node resulting in $O(kn \log n)$ space complexity, a different mechanism to update that information and a different locking mechanism; these resulted in increased space and time complexity.

Our time complexity result is only an upper bound; it may be possible to get a tighter upper bound. So, we pose an open question, "design an $O(kn)$ self-stabilizing algorithm for maximal k-packing with safe convergence with $O(k \log n)$ space complexity".

Acknowledgment

The presentation in the paper was improved by detailed comments of the reviewers. This research was partly supported by National Science Foundation [DBI-0960586 and DBI-0960443]; Guangzhou Science and Technology Fund of China [2012J4300038, LCY201206, and 2013J4300061]; and Guangdong Science and Technology Fund of China [2012B091100221].

References

[1] E.W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, Nov. 1974.

[2] S. Dolev. *Self stabilization.* MIT Press, 2000.

[3] N. Guellati and H. Kheddouci. A survey on self-stabilizing algorithms for independence, domination, coloring, and matching in graphs. *Journal of Parallel and Distributed Computing*, 70(4):406–415, 2010.

[4] H. Kakugawa and T. Masuzawa. A self-stabilizing minimal dominating set algorithm with safe convergence. In *20th IEEE International Parallel and Distributed Processing Symposium*, pages 25–29, 2006.

[5] S. Kamei and H. Kakugawa. A self-stabilizing approximation for the minimum connected dominating set with safe convergence. In *Proceedings of the 12th International Conference on Principles of Distributed Systems*, pages 496–511, Luxor, Egypt, 2008. Springer-Verlag.

[6] S. Kamei and H. Kakugawa. A self-stabilizing 6-approximation for the minimum connected dominating set with safe convergence in unit disk graphs. *Theoretical Computer Science*, 428:80–90, April 2012.

[7] Y. Ding, J.Z. Wang, and P.K. Srimani. Self-stabilizing minimal global offensive alliance algorithm with safe convergence in an arbitrary graph. In *11th Annual Conference on Theory and Applications of Models of Computation, LNCS 8402*, pages 366–377, April 2014.

[8] S. Kamei, T. Lzumi, and Y. Yamauchi. An asynchronous self-stabilizing approximation for the minimum connected dominating set with safe convergence in unit disk graphs. In *15th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, volume 8255, pages 251–265, 2013.

[9] F. Carrier, A.K. Datta, S. Devismes, L.L. Larmore, and Y. Rivierre. Self-stabilizing (f, g)-alliances with safe convergence. In *15th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, Osaka, Japan, Nov. 2013.

[10] R. Gallant, G. Gunther, B. Hartnell, and D.F. Rall. Limited packing in graphs. *Discrete Applied Mathematics*, 158:1357–1364, 2010.

[11] M. Gairing, R.M. Geist, S.T. Hedetniemi, and P. Kristiansen. A self-stabilizing algorithm for maximal 2-packing. *Nordic Journal of Computing*, 11:1–11, 2004.

[12] M. Gairing, S.T. Hedetniemi, P. Kristiansen, and A.A. McRae. Distance-two information in self-stabilizing algorithms. *Parallel Processing Letters*, 14:387–398, 2004.

[13] W. Goddard, S.T. Hedetniemi, D.P. Jacobs, and P.K. Srimani. Self-stabilizing global optimization algorithms for large network graphs. *International Journal of Distributed Sensor Networks*, 1(3-4):329–344, 2005.

[14] F. Manne and M. Mjelde. A memory efficient self-stabilizing algorithm for maximal k-packing. In *Proceedings of the 8th International Conference on Stabilization, Safety, and Security of Distributed Systems*, pages 428–439, Berlin, Heidelberg, 2006.

[15] Z. Shi. An updated self-stabilizing algorithm to maximal 2-packing and a linear variation under synchronous daemon. In *2011 International Conference on Parallel and Distributed Processing Techniques and Applications*, volume 1, pages 262–267, 2011.

[16] Z. Shi. A self-stabilizing algorithm to maximal 2-packing with improved complexity. *Information Processing Letters*, 112:525–531, Jul. 2012.

[17] S. Dolev, A. Israeli, and S. Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7(1):3–16, Nov. 1993.