International Journal of Engineering, Mathematical and Physical Sciences
ISSN: 2517-9934
Vol:4, No:3, 2010

A New Self-stabilizing Algorithm for Maximal 2-packing

Zhengnan Shi
Dept of Mathematics and Computer Science
University of Wisconsin Whitewater
Whitewater, WI, 53190
USA

Abstract—In the self-stabilizing algorithmic paradigm, each node has a local view of the system, in a finite amount of time the system converges to a global state with desired property. In a graph $G = (V, E)$, a subset $S \subseteq V$ is a 2-packing if $\forall i \in V$: $|N[i] \cap S| \leq 1$. In this paper, an ID-based, constant space, self-stabilizing algorithm that stabilizes to a maximal 2-packing in an arbitrary graph is proposed. It is shown that the algorithm stabilizes in $O(n^3)$ moves under any scheduler (daemon). Specifically, it is shown that the algorithm stabilizes in linear time-steps under a synchronous daemon where every privileged node moves at each time-step.

Keywords—self-stabilization, 2-packing, distributed computing, fault tolerance, graph algorithms

I. Introduction and Related Work

Self-stabilization is a strong and desirable fault-tolerance property. The self-stabilizing approach is introduced by Dijkstra [3]. A distributed network is defined as a connected, undirected graph G with node set V and edge set $E \subseteq V \times V$. Let $n = |V|$ and $m = |E|$. Two nodes joined by an edge are said to be *neighbors*. Notation $N(i)$ is used to denote the set of neighbors of node i—its (open) *neighborhood*. $N[i]$, the *closed neighborhood* of i, is defined as $N[i] = N(i) \cup \{i\}$. The *distance* $dist(i, j)$ between two nodes i and j is the number of edges in a shortest i-j path. The contents of a node's local variables are defined as its *local state*. The system's *global state* is the union of all local states. If you take an arbitrary distributed algorithm and start it in a state where its variables have been set to a random value from its domain, the behavior is usually not predictable. However, starting from *any* initial configuration and in *every* execution, self-stabilizing systems are required to recover to a set of legal states.

A. Self-stabilizing Algorithms

A self-stabilizing algorithm is presented as a set of rules, each with a boolean predicate and an action. The rules of these algorithms are of the form $p \rightarrow M$, where p is a Boolean *predicate*, and M is a *move* which changes local variable(s). A node is said to be *privileged* if the predicate p is true. If a node becomes privileged, it may execute the corresponding move M. In the shared-variable version of this paradigm, every node executes the same set of self-stabilizing rules, and maintains and changes its own set of local variables based on the current values of its variables and those of its *neighbors*.

email shiz@uww.edu, telephone (262)472-5006, fax (262)472-1372.

It is assumed that there exists a *daemon*, an adversarial oracle as introduced in [3], [9], which at each time-step selects one or more of the privileged nodes to move. In the *serial*, also known as the *central daemon* model, no two nodes move at the same time. In the *distributed daemon* model, the daemon can choose any subset of privileged nodes to move simultaneously. A special case of this is the *synchronous daemon* where every privileged node moves at each step.

Self-stabilizing algorithms can be designed for networks that are either *ID-based* or for the networks that are *anonymous*. In an ID-based network, each node has a unique ID. In an anonymous network, the nodes lack unique IDs, so there is not a priori way of distinguishing them. Anonymous self-stabilizing algorithms are not as powerful as the ID-based ones. However, they do not require the maintenance overhead of the ID's in the network. It is known that, given IDs, any algorithm for the central daemon can be transformed into one for the distributed daemon (see for example [2]). However, the resulting protocols are not as fast. For a complete discussion of self-stabilization, see the books by Dolev [4] or Tel [15].

When no further state change is possible, it is claimed that the system is in a *stable configuration*. A self-stabilizing algorithm must satisfy:

1) From any initial illegitimate state it reaches a legitimate state after a finite number of moves; and
2) For any legitimate state and for any move allowed by that state, the next state is a legitimate state.

The complexity of a self-stabilizing algorithm is measured by the upper bound of the number of moves and/or *time-steps* [4], [5]. A *time-step* is the minimum period of time where every node that is continually privileged moves at least once. In general, the number of moves is an upper bound on the number of time-steps.

Several graph problems arise naturally in distributed systems. For example, distributed algorithms for finding matchings, independent sets, dominating sets and colorings have been studied [7], [9], [11]–[13]. Synchronous algorithms have been considered in [1], [6], [14] inter alia.

In this paper, an ID-based, constant space, self-stabilizing algorithm that stabilizes to a maximal 2-packing in an arbitrary graph is proposed. It is shown that the algorithm stabilizes in $O(n^3)$ moves under any scheduler (daemon). Specifically, it is shown that the algorithm stabilizes in linear time-steps under

a synchronous daemon where every privileged node moves at each step.

B. the 2-packing Problem

A subset S of nodes in a graph $G = (V,E)$ is called a 2-packing [10] if $\forall i \in V : |N[i] \cap S| \leq 1$. A 2-packing is maximal if no proper superset of S is a 2-packing.

The 2-packings are subsets of nodes with some inherent non-locality, in that no two nodes in the 2-packing set can have overlapping neighborhoods. This prevents a node in a 2-packing from having direct knowledge of any other node in the set. A self-stabilizing algorithm for maximal independent sets (1-packing) was produced by Hedetniemi et al. in [7]. Because of the non-locality of 2-packings, this algorithm is more complex both in design, correctness and complexity proofs. The facts below follow directly from the definition of 2-packing.

1) Every 2-packing S is an independent set.
2) If S is a 2-packing, then $\forall i,j \in S : dist(i,j) \geq 3$.
3) If S is maximal 2-packing, then $\forall i \in V \backslash S \exists j \in S : dist(i,j) \leq 2$.

The goal of this self-stabilizing algorithm is to find a maximal 2-packing. The problem of finding a maximum 2-packing (a maximal 2-packing of largest cardinality) is shown to be NP-hard by Hochbuam and Schmoys in [8], finding a maximal one is easily done in linear time with the standard RAM model.

II. A Self-stabilizing Algorithm for Maximal 2-packing

Now a self-stabilizing algorithm which finds a Maximal 2-packing is formally introduce. It is referred to as Algorithm $Maximal\ 2\text{-}Packing$. It is assumed that each node in the network has a unique ID. Without loss of generality, let the ID's be between 1 and n (the order of the graph). When node i is referenced, variable i holds the ID of the node. Local variables of Algorithm MP are listed next.

1) Variable c is an enumerate type of 3 possible values: 0, 1 or 2.
2) Variables p (parent) and r (root) point to neighbors. They hold either the ID's of nodes or 0. Subscription is used to denote the ID of the hosting node of a variable. Since an ID is at least 1, let $c_0 = p_0 = r_0 = null$.

The following notations are used to abridge the presentation of Algorithm MP.

1) Notation r-p denotes a node of minimum c in set $\{j \in N(i), r_j > r_i \wedge c_j < 2\}$. If there are more than one node, r-p is the one with the maximum r. If there is no such node, let r-p be $null$.
2) Notation min-p denotes a node of minimum c in set $\{j \in N(i), c_j < 2\}$. If there are more than one node, min-p is the one with the maximum r. If there is no such node, let min-p be $null$.

The rules of this self-stabilizing algorithm are represented by a list of if-then statements. To make a rule concise, the negation of the desired state (the then statement) is omitted from the condition. If the desired state is already achieved, the node shall not be privileged by the rule. Assume the algorithm runs on node i.

Algorithm 1: $Maximal\ 2\text{-}Packing$

c-Decrease
  if $min\text{-}p \neq null \wedge c_i > c_{min\text{-}p} + 1$
    then $c_i = c_{min\text{-}p} + 1 \wedge r_i = r_{min\text{-}p} \wedge p_i = min\text{-}p$
Join
  if $\forall j \in N(i), c_j = 2$
    then $c_i = 0 \wedge r_i = i \wedge p_i = i$
Leave
  if $c_i = 0 \wedge r\text{-}p \neq null$
    then $c_i = c_{r\text{-}p} + 1 \wedge r_i = r_{r\text{-}p} \wedge p_i = r\text{-}p$
c-Orphan
  if $c_i = 1 \wedge (r_i \neq r_{min\text{-}p} \vee r_i \neq r_{p_i} \vee c_{p_i} \neq 0)$
    then $c_i = c_{min\text{-}p} + 1 \wedge r_i = r_{min\text{-}p} \wedge p_i = min\text{-}p$

Algorithm 1: Maximal 2-Packing

Upon stabilization, the maximal 2-packing set S is identified by nodes of $c = 0$.

A. A Brief Explanation of the Rules

In this section, the reasons behind the design of the rules of Algorithm $Maximal\ 2\text{-}Packing$ are described. A key local variable on each node is c. Variable c is used to record the shortest distance to a node in the 2-packing set S. If $c = 0$, the node is in the set S. If $c = 1$, then the node is adjacent to a node of $c = 0$. If $c = 2$, then the node is adjacent to a node of $c = 1$, and possibly other nodes of $c = 1$ or $c = 2$; but it is not adjacent to any node of $c = 0$. Through neighbors' variable c's, a node can extract information on whether it is within distance 2 from a node in the 2-packing set S.

The c-Decrease Rule applies to a node of $c = 2$. If a node i sees a neighbor, min-p with $c_{min\text{-}p} = 0$, then i decreases its variable c_i from 2 to 1. The root r_i and parent p_i variables of node i are also adjusted to these of the node min-p. After the move, variable c_i reflects the shortest distance from i to a node in set S.

If node i is surrounded by node(s) of $c = 2$, i considers itself to be of distance 2 to any node in the 2-packing set S. The node i can move by the Join Rule to join set S. Its root and parent variables are both set to the ID of itself. Recall that the negation of the desired state (the then statement) is omitted from the condition. If node i has never moved and $c_i = 0$ with $r_i \neq i$ or $p_i \neq i$, node i can move by the Join Rule to correct variables r_i and p_i.

The Leave Rule uses ID's to resolve conflicts when the distance between two nodes in the set S is less than 3. If a node i of $c_i = 0$ is adjacent to a node r-p of $c_{r\text{-}p} < 2$ and $r_i < r_{r\text{-}p}$, then i changes its c to a nonzero value. The Leave Rule ensures that the distance between any pair of nodes in S is at least 3. For the Leave Rule to function properly, any node j of $c_j = 1$ must have r_j equals the largest ID of a neighboring node in the set S. This is guaranteed by the last rule of Algorithm $Maximal\ 2\text{-}Packing$, the c-Orphan Rule.

A node i of $c_i = 1$ must be adjacent to a node, pointed to by its parent variable, p_i with $c_{p_i} = 0$. The root variable of node i must be equal to the root variable of its parent p_i. The c-Orphan Rule is designed to ensure this property. By definition,

notation *min-p* denotes a node of minimum c in set $\{j \in N(i), c_j < 2\}$. If there are more than one node, *min-p* is the one with the maximum r. The c-Orphan Rule also updates variable r_i to the largest ID possible (without increasing c). This ensures that the Leave Rule functions properly.

III. Correctness

It is first shown that Algorithm *Maximal 2-Packing* produces a maximal 2-packing upon stabilization. The correctness of Algorithm MP is proved by Lemma 1.

Lemma 1: Upon stabilization, Algorithm *Maximal 2-Packing* produces a Maximal 2-packing set S in which every node has $c = 0$.

Proof: Let nodes i and j have $c = 0$ upon stabilization. By the Leave Rule, i and j are not adjacent. If $dist(i, j) < 3$, let k be the node on the shortest path between i and j. Ensured by the c-Decrease Rule, c_k is 1. Without loss of generality, let $i > j$. Ensured by the c-Orphan Rule, $r_k \geq i > j$ which is the largest ID of nodes i and j. Then j is privileged by the Leave Rule; a contradiction. Hence, the self-stabilizing algorithm yields a 2-packing.

Next, it is shown that set S is a maximal 2-packing. Variable c can have three possible values: 0, 1 and 2. By the c-Orphan Rule, a node of $c = 1$ must be adjacent to a node of $c = 0$. A node of $c = 2$ must be adjacent to a node of $c = 1$, and possibly other nodes of $c = 1$ or $c = 2$, since it is not privileged by the Join Rule. But the node must not be adjacent to any node of $c = 0$ by the c-Decrease Rule. Hence any node not in the 2-packing set is within distance 2 from a node in the set. The result is maximal. ■

IV. Convergence and Complexity Analysis

In this section, the complexity of Algorithm *Maximal 2-Packing* is bounded. The number of moves is used to evaluate the complexity of the algorithm except for the analysis under a synchronous daemon. With a synchronous daemon, every privileged node moves at each *time-step*. The concept *time-step* is used to measure complexity for this case. A *time-step* is the minimum period of time where every node that is continually privileged moves at least once. The concept of a *dirty* node is first define.

Definition. *If a node has never moved since the start of the self-stabilization process, it is called a* dirty *node.*

The number of *dirty* nodes is at most n. Once a node moves, it is no longer dirty.

Definition. *Let i be a node and S be the 2-packing set. $count(i, S)$ is used to denote the number of nodes in S whose ID's are greater than i.*

The convergence is shown next. First the number of moves by the Leave Rule is examined. The analysis looks at the number of moves under a central daemon. This number of moves is an upper bound of the complexity of Algorithm *Maximal 2-Packing* under any daemon (scheduler).

Lemma 2: The number of moves by the Leave Rule is of $O(n^2)$.

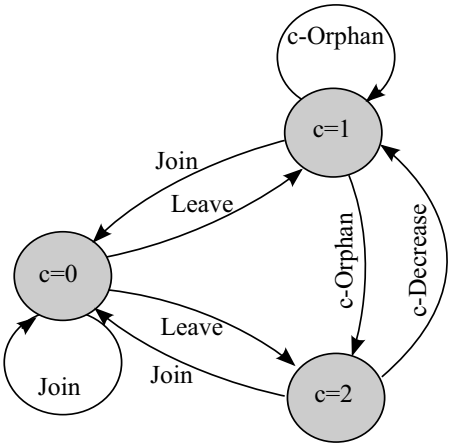

Fig. 1. State Transition Diagram

Proof: A node i may move by the Leave Rule because it is dirty. Otherwise node i has moved before, its last move must be by the Join Rule since the Leave Rule requires $c_i = 0$. Each neighbor of i had $c = 2$ at the last move. Now, when i moves by the Leave Rule, $r_{r\text{-}p} > r_i = i$ by definition. Let j denote *r-p*. Node j could be dirty when i moved by the Join Rule. Because j must move before i moves by the Leave Rule, the number of dirty nodes reduces by one. If j has moved before, j was not adjacent to any node of $c = 0$ at its last move. Before i moves by the Leave Rule,at the last move of j, j was adjacent to a node k with $c_k = 0 \wedge k > i$. If $c_k = 0$ when i moves by the Leave Rule, $count(i, S)$ has increased by one. If $c_k > 0$ when i move by the Leave Rule, then k must have moved at least once by the Leave Rule. Note that only the Leave Rule may decrease $count(i, S)$. For node k, the same argument for i is used: either the number of dirty nodes has decreased, or $count(k, S)$ has increased. Since $k > i$, $count(i, S)$ has also increased. The number of dirty nodes is at most n and $count(i, S) < n$. There are $O(n)$ moves by the Leave Rule on each node, hence the result. ■

Next, the number of moves by the c-Orphan Rule is examined.

Lemma 3: The number of moves by the c-Orphan Rule is of $O(n^3)$.

Proof: Let i be a node moves by the c-Orphan Rule. If i is a dirty node, it is no longer dirty after the one move. There is $O(n)$ moves by dirty nodes.

If node i has moved and $c_i = 1$, then p_i was a neighbor of i with $c_{p_i} = 0$ when i made its last move. Since $c_i = 1$, no neighbor of i can move by the Join Rule. Node p_i must move by the Leave Rule before i moves by the c-Orphan Rule. By Lemma 2, the number of moves by the Leave Rule is of $O(n^2)$. Node p_i can be the parent of $O(n)$ nodes. Hence the number of moves by nodes of $c = 1$ is of $O(n^3)$. ■

Theorem 1: Algorithm *Maximal 2-Packing* stabilizes with a maximal 2-packing in $O(n^3)$ moves under any daemon(scheduler).

Proof: Figure 1 is the state transition diagram with regard to variable c. A move by the c-Decrease, Join or Leave Rules

changes the value of c. A move by the c-Orphan Rule may result in the same value of c but a different value of r.

By Lemma 2, the total number of moves by the Leave Rule is of $O(n^2)$. By Lemma 3, the total number of moves by the c-Orphan Rule is of $O(n^3)$. These restrictions bound the number of moves on each arc in the diagram to be of $O(n^3)$. Hence, the total number of moves by Algorithm $Maximal\ 2\text{-}Packing$ is $O(n^3)$. ■

Algorithm $Maximal\ 2\text{-}Packing$ converges under a synchronous daemon is second shown.

Lemma 4: If the distance between two nodes of $c = 0$ (in set S) is less than 3, at least one of the two nodes moves out of set S within the next 2 time-steps under a synchronous daemon.

Proof: Let nodes i and j have $c = 0$ at time-step t. Without loss of generality, let $i > j$. If i and j are adjacent, then node j moves by the Leave Rule at time-step $t+1$ under a synchronous daemon. If $dist(i, j) = 2$, let k be the node on the shortest path between i and j. Ensured by the c-Decrease Rule, c_k is 1 at time-step $t+1$. Ensured by the c-Orphan Rule, $r_k \geq i$ which is the largest ID of nodes i and j. Then j moves by the Leave Rule at time-step $t + 2$ under a synchronous daemon. ■

Lemma 5: If a node i is not within distance 2 from any node of $c = 0$ (in the set S), the node or its neighbor(s) within distance 2 from i will join the set S within the next 2 time-steps under a synchronous daemon.

Proof: Assume node i is not within distance 2 from any node of $c = 0$ at time-step t. In time-step $t+1$, node i and its neighbors must move by the Join Rule or the c-Orphan Rule. If i and its neighbors all move by the c-Orphan Rule, they all have $c = 2$. Hence node i moves by the Join Rule in time-step $t + 2$. ■

Theorem 2: Algorithm $Maximal\ 2\text{-}Packing$ stabilizes at a maximal 2-packing in linear time-steps under a synchronous daemon.

Proof: Under a synchronous daemon, the algorithm executes in parallel. For proposition of the upper bound, this process is described in order.

By Lemma 5, every node is within distance 2 from a node in set S by time-step 3. Through these time-steps, the cardinality of the set S increases. By Lemma 4, the distance between two nodes of set S is at least 3 by time-step 6. Through these time-steps, the cardinality of the set S decreases from the end of time-step 3, but increases from time-step 1. By time-step 6, any neighbor of a node joined the set S by the end of time-step 3 has $c = 1$. Hence, if the distance between two nodes of set S is less than 3, both nodes have joined the set S after time-step 3. If the 6 time-steps is considered a process, it repeats $O(n)$ time-steps because the cardinality of S increases after each process. Hence, the algorithm stabilizes in linear time-steps. ■

V. Concluding Remarks

In this paper, an ID-based, constant space, self-stabilizing algorithm for finding a maximal 2-packing in an arbitrary graph have been presented. It have been seen that the problem of 2-packing has inherent non-local properties. However, it is possible to design a reasonably fast self-stabilizing algorithm for finding a maximal 2-packing in arbitrary graph. It is shown that the algorithm stabilizes in $O(n^3)$ moves under any scheduler (daemon). Specifically, it is shown that the algorithm stabilizes in linear time-steps under a synchronous daemon where every privileged node moves at each step.

References

[1] Y. Afek and S. Dolev. Local stabilizer. *J. Parallel Distrib. Comput.*, 62(5):745–765, 2002.
[2] J. Beauquier, A. K. Datta, M. Gradinariu, and F. Magniette. Self-stabilizing local mutual exclusion and daemon refinement. In *International Symposium on Distributed Computing*, pages 223–237, 2000.
[3] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Comm. ACM*, 17(11):643–644, Jan. 1974.
[4] S. Dolev. *Self-Stabilization*. MIT Press, 2000.
[5] S. Dolev, A. Israeli, and S. Moran. Uniform dynamic self-stabilizing leader election. *IEEE Trans. on Parallel and Distributed Systems*, 8(4), 1995.
[6] W. Goddard, S. T. Hedetniemi, D. P. Jacobs, and P. K. Srimani. Self-stabilizing algorithms for orderings and colorings. *International Journal of Foundations of Computer Science*, 16:19–36, 2005.
[7] S. M. Hedetniemi, S. T. Hedetniemi, D. P. Jacobs, and P. K. Srimani. Self-stabilizing algorithms for minimal dominating sets and maximal independent sets. *Comput. Math. Appl.*, 46(5-6):805–811, 2003.
[8] D. S. Hochbaum and D. B. Schmoys. A best possible heuristic for the k-center problem. *Mathematics of Operations Research*, 10(2):180–184, 1985.
[9] S.-C. Hsu and S.-T. Huang. A self-stabilizing algorithm for maximal matching. *Inform. Process. Lett.*, 43:77–81, 1992.
[10] A. Meir and J. W. Moon. Relations between packing and covering numbers of a tree. *Pacific Journal of Mathematics*, 61(1):225–233, 1975.
[11] A. Panconesi and A. Srinivasan. The local nature of δ-coloring and its algorithmic applications. *Combinatorica*, 15:225–280, 1995.
[12] S. Rajagopalan and V. Vazirani. Primal-dual RNC approximation algorithms for set cover and covering integer programs. *SIAM J. Comput.*, 28:525–540, 1998.
[13] Z. Shi, W. Goddard, and S. T. Hedetniemi. an anonymous self-stabilizing algorithm for 1-maximal independent set in trees. *Information Processing Letters*, 91(2):77–83, 2004.
[14] S. Shukla, D. Rosenkrantz, and S. Ravi. Observations on self-stabilizing graph algorithms for anonymous networks. In *Proceedings of the Second Workshop on Self-Stabilizing Systems*, pages 7.1–7.15, 1995.
[15] G. Tel. *Introduction to Distributed Algorithms, Second Edition*. Cambridge University Press, Cambridge UK, 2000.