# A Memory Efficient Self-stabilizing Algorithm for Maximal *k*-Packing

Fredrik Manne and Morten Mjelde

Department of Informatics, University in Bergen, Norway
{fredrik.manne, mortenm}@ii.uib.no

**Abstract.** The $k$-packing problem asks for a subset $S$ of the nodes in a graph such that the distance between any pair of nodes in $S$ is greater than $k$. This problem has applications to placing facilities in a network.

In the current paper we present a self-stabilizing algorithm for computing a maximal $k$-packing in a general graph. Our algorithm uses a constant number of variables per node. This improves the memory requirement compared to the previous most memory efficient algorithm [9] which used $k$ variables per node. In addition the presented algorithm is very short and simple.

**Keywords:** self-stabilizing algorithms, $k$-packing.

## 1 Introduction

Facility location problems in a network involve distributing a set of resources such that the entire network is covered. Depending on the objective these can either be minimization problems where one wants to use as few resources as possible while covering the graph or maximization problems where one wants to distribute as many resources as possible under some constraint. There exists a number of such problems and they have been extensively studied in the literature of sequential algorithms [1,10,12,13].

In this paper we present a self-stabilizing distributed algorithm for one such problem, namely the $k$-packing problem. This involves selecting a set $S$ of nodes such that the length of the shortest path between any pair of nodes $(v, w) \in S$ is greater than $k$ (a 1-packing is better known as an independent set). The set $S$ is referred to as black nodes while the remaining nodes are referred to as white. A maximum $k$-packing implies that $S$ is the set with largest cardinality, and finding this is NP-hard on a general graph [6]. The simpler problem of computing a maximal $k$-packing (i.e. no superset of $S$ is also a legal solution) can easily be solved by a sequential greedy algorithm in linear time.

Previous work on developing self-stabilizing algorithms for the $k$-packing problem has resulted in several different algorithms. Gairing et al. gave an algorithm that computed a maximal 2-packing on a general graph [5]. This algorithm used an exponential number of moves and a constant number of variables per node. Goddard et al. subsequently developed a self-stabilizing algorithm for solving maximal $k$-packing on a general graph [9]. This algorithm used an exponential number of

A.K. Datta and M. Gradinariu (Eds.): SSS 2006, LNCS 4280, pp. 428–439, 2006.

moves, and $k$ variables per node. In a recent paper Goddard et al. also presented a self-stabilizing algorithm for the same problem that runs in $n^{O(\log k)}$ moves [8]. However, this algorithm requires that each node stores information about the graph within a radius of $k$ from itself, thus substantially increasing the memory requirements. We also note that there exists a self-stabilizing algorithm for computing a maximum $k$-packing on a tree graph with a moves complexity of $O(n^3)$ [11].

From the above exposition it follows that there is a trade off between the number of moves and the amount of memory used on each node when designing self-stabilizing algorithms for computing a maximal $k$-packing on a general graph. In fact, if one first computed a spanning tree in the graph, a task which is considerable simpler than computing a maximal $k$-packing [7], one could copy the structure of the entire graph into each node in a polynomial number of moves [2] and then solve the maximal $k$-packing problem by a local deterministic sequential algorithm on each node. The same approach could also be used to compute a maximum $k$-packing (although this would require an exponential local running time on every node).

In this paper we fill in one part in this trade off between moves complexity and memory usage. We present an algorithm that computes a maximal $k$-packing for a general graph using only a constant number of variables per node, each of which hold at most $\mathrm{O}(\log n)$ bits. However, the moves complexity of the algorithm is still exponential. In addition to using less memory than other algorithms for this problem the algorithm itself is very short, and thus easy to understand and implement.

Limiting the amount of memory is an important factor in many applications such as in sensor networks where the computational units are small and rely on battery power to operate.

The rest of this paper is organized as follows. In Section 2 we present some background on self-stabilizing algorithms. In Section 3 we present our algorithm and show that any stable solution produced by it is also a legal solution and that it will stabilize in a finite amount of time. Finally, we conclude in Section 4.

## 2 The Self-stabilizing Paradigm

Self-stabilizing algorithms are a variant of distributed systems first introduced by Dijkstra in 1974 [3]. However, the significance of the work was not immediately recognized, and serious work did not begin until the late 1980's. One of the most important properties of any self-stabilizing algorithm is its ability to recover from any transient errors that occurs, and even changes in the graph itself. This ability makes self-stabilizing algorithms extremely fault tolerant.

A self-stabilizing algorithm does not assume the existence of a central leader. Instead, all nodes in the graph are considered equals, and each of them has the same copy of the algorithm. Each node maintains a set of variables that together make up the nodes *local state*. The union of all local states is the graphs *global state*. In the normal self-stabilizing model any node has knowledge only of its own and its neighbors' local states. The algorithm itself is comprised of a set of rules. These are typically written in the form:

**Rule** $i$
**if** $p(v)$
**then** $M$

The function $p(v)$ is called the *predicate*, and $M$ is called the *move*. The predicate takes the node $v$ as a parameter, and becomes true or false based on $v$'s local state and the local state of its neighbors. The move $M$ will change one or more of $v$'s local variables. If the predicate is true the rule is called *privileged*, and only then can it execute its corresponding move. For cases where there are more than one privileged rule in the graph, the self-stabilizing model assumes the existence of a *central daemon* that determines which rules will be permitted to make its move. Various self-stabilizing algorithms employ different daemons, and for the current algorithm we assume an adversarial daemon (as opposed to a fair or random daemon). Regardless of the type of daemon used any self-stabilizing algorithm has to guarantee to reach a solution in a finite number of moves independent of the starting configuration. This is called to *stabilize* and implies that no node in the graph has a privileged rule. For further reading on self-stabilizing algorithms, see [4].

For our algorithm we assume the existence of an undirected graph $G = (V, E)$ where $V$ is the set of nodes and $E$ the set of edges. We further assume that each node has a unique $ID$, and that these $ID$s can be ordered. The $ID$ of a node $v$ is denoted by $ID_v$. The set of nodes $N(v)$ is the open neighborhood of $v$, and contains all the neighbors of $v$.

## 3 The Algorithm

In the following we present and analyze our new algorithm. It is based on each node determining the distance to its two nearest black nodes (possibly including itself). Based on this information a node can then determine if it should be black or white.

### 3.1 The Local Variables

As mentioned in Section 2, each node in a self-stabilizing algorithm maintains a set of local variables that make up the nodes local state. In our algorithm, each node $v \in V$ has two pairs of variables: $(p_v, b_v)$ and $(p'_v, b'_v)$. The intention is that $p_v$ denotes the shortest distance (i.e. number of edges) to $v$'s closest black node $y$ and with $b_v = ID_y$. The pair $(p'_v, b'_v)$ gives the same information about $v$'s second closest black node (assuming it exists). The range of $p_v$ is $[0, \infty]$ while the range of $p'_v$ is $[1, \infty]$. It then follows that in a stable configuration the black nodes are identified by having $p$-values equal to 0 and $p'$-values larger than $k$.

In the case where a node has more than one black node at a minimum distance from it we say that the black node with the smallest $ID$-value is the closest one. Thus the term "closest black node" will always be well defined.

As we will explain in further detail later, the purpose of the $(p', b')$ values is for every black node in the graph to gain knowledge of its closest black node other than itself.

### 3.2 Definitions and Notations

Based on the local variables of each node we now give some definitions and notations that we will be using. This is done to make the ensuing presentation clearer and also more compact.

Two adjacent nodes $v$ and $w$ where $b_v = b_w$ and $p_v = p_w + 1$ belong to the same *domain* and we say that $w$ is a *predecessor* of $v$ in the domain. A node $v$ that does not have a predecessor is a *leader* of the domain. The domain relation is transitive and thus each domain is a connected component of the graph. We will also say that adjacent nodes $v$ and $w$ where either $b'_v = b_w$ and $p'_v = p_w + 1$ or $b'_v = b'_w$ and $p'_v = p'_w + 1$ belong to the same domain. Thus a node can belong to two domains at the same time depending on if we are looking at the $(p_v, b_v)$ or $(p'_v, b'_v)$ values.

A domain $U$ is *proper* if $v$ is a leader of $U$ and there exists a node $w$ such that $b_v = ID_w$. Note that $w$ does not have to be part of $U$ for $U$ to be proper. A domain that is not proper is *improper*.

We define the set $T_v$ for a node $v$ as being the pair $(p_v, b_v)$ and $(p'_v, b'_v)$. We further define the set $T_M$ for some set of nodes $M$ as $\cup_{v \in M} T_v$.

### 3.3 The Algorithm

The algorithm consists of one function and one rule. These are as follows:

**support**$(v)$
$(\alpha, \beta) = \min\{(\gamma, \delta) \in T_{N(v)} : \delta \neq ID_v\}$
$(\alpha', \beta') = \min\{(\gamma, \delta) \in T_{N(v)} : \delta \neq ID_v, \delta \neq \beta\}$

**if** $(\alpha \geq k) \vee (p_v = 0 \wedge \beta > ID_v)$
return $(0, ID_v, \alpha + 1, \beta)$
**else**
return $(\alpha + 1, \beta, \alpha' + 1, \beta')$

**Rule 1**
**if** $(p_v, b_v, p'_v, b'_v) \neq$ support$(v)$
**then** set $(p_v, b_v, p'_v, b'_v) =$ support$(v)$

The purpose of the support function is to return the correct $(p, b)$ and $(p', b')$ values for a node $v$ based on the local state of $v$ and its neighbors. The function starts by selecting a pair $(\alpha, \beta)$ from $T_{N(v)}$ such that $\alpha$ is as small as possible while $\beta \neq ID_v$. In the case of a tie a pair with the smallest $\beta$ value is selected. Next the function selects a pair $(\alpha', \beta')$ in the same manner as above only with the added constraint that $\beta' \neq \beta$. In both of the above cases, if no valid pair can be found in $T_{N(v)}$ the pair $(\infty, \infty)$ will be used.

Based on the selected values the function will now determine if $v$ should be black or not. With the assumption that $(\alpha, \beta)$ represents the distance to the closest black node (other than $v$ itself) it follows that either if $\alpha \geq k$ or if $p_v = 0$ (indicating that $v$ is at present black) and the closest black node has higher $ID$

than $v$ ($\beta > ID_v$) then $v$ can become (or remain) black. In the case where the above condition was met, the function returns $(0, ID_v, \alpha + 1, \beta)$, where the first pair of values indicates that $v$ should be black and the second pair gives the distance and $ID$ of the closest black other than $v$ itself.

If $v$ should not be black then it should become (or remain) white. All it has to do in this case is to gather data about its two closest black nodes. Thus the function returns $(\alpha + 1, \beta, \alpha' + 1, \beta')$.

Rule 1 is the only rule in the algorithm. It simply determines if one or more of the values returned by the support function does not correspond to the node's current values. If this is the case, the node is privileged for a move that corrects them.

### 3.4 Correct Stabilization

We now show that the algorithm, when stable, has solved the maximal $k$-packing problem. To do so we first show that in a stable configuration the values of $p$ and $p'$ will be set to the distance of the nearest and second nearest black node respectively.

We again remind the reader that in the case where a node $v$ has more than one black node at minimum distance we will break ties by defining that which ever has the smallest $ID$ is the one closest to $v$. Note that this is consistent with how the support function operates.

**Lemma 1.** *Let $(p_v, b_v)$ be the local values for a node $v \in V$ in a stable configuration. Then there exists a black node $y$ such that $y$ is the closest black node to $v$ of distance $p_v$ from $v$ and such that $ID_y = b_v$.*

*Proof.* Note first that we cannot have a node $v$ with $p_v > k$ in a stable configuration.

The proof of the claim is by induction on the value of $p_v$. If $p_v = 0$ then $v$ is black, and must have $b_v = ID_v$ in a stable configuration. Assume therefore that the claim is true for every $w \in V$ where $p_w < l$, $1 < l \leq k$, and let $v \in V$ be a node such that $p_v = l$. Then by the construction of the support function there must exist a node $u \in N(v)$ such that $p_u = p_v - 1 = l - 1$ and $b_v = b_u$. From the induction claim it follows that there exists a black node $y$ such that $b_u = ID_y$ where $y$ is the closest black node at a distance $l - 1$ from $u$. We therefore have that there exists a path of length $l$ between $v$ and the black node $y$ where $b_v = ID_y$.

If there was to exist a path from a black node $x$ to $v$ of length less than $l$ or of length $l$ but with $ID_x < b_v$, then again by the induction hypothesis there must exist a node $z \in N(v)$ such that either $p_z + 1 < p_v$ or $p_z + 1 = p_v$ and $b_z < b_v$. In both of these cases $v$ would be privileged for a move. □

**Corollary 1.** *In a stable configuration, the maximum distance from any node to a black node is at most $k$.*

*Proof.* Consider a white node $v$ in a stable solution that has minimum distance $> k$ to the nearest black node. By Lemma 1 it then follows that every $w \in N(v)$ has $p_w \geq k$ in which case the support function will return 0 for $p_v$ thus contradicting the assumption that the configuration is stable. □

We now need to show that we cannot have two black nodes in a stable configuration that are closer to each other than $k$. To do so we first show that the value of $p'_v$ will be set to the distance to the second closest black node of a node $v$.

**Lemma 2.** *Let $(p'_v, b'_v)$ be the local values for some node $v \in V$ in a stable configuration containing at least two black nodes. Then there exists a black node $y$ such that $y$ is the second closest black node to $v$ of distance $p'_v$ from $v$ and such that $ID_y = b'_v$.*

*Proof.* Note first that by the construction of the support function each node that has $0 < p'_v < \infty$ in a stable configuration must have a neighbor $w$ where either $(p_w+1, b_w) = (p'_v, b'_v)$ and $b_w \neq b_v$ or where $(p'_w+1, b'_w) = (p'_v, b'_v)$ and $b_w = b_v$. Thus starting from $v$ there exists a path along decreasing $p$ or $p'$ values such that the $b$-value is unchanged. If the path makes use of a $p$-value then since the $p$-values do not depend on the $p'$-values, the path will lead to a black node. This must eventually happen since a value of $p' = 1$ must have been obtained from a black neighbor. Thus it follows that if $p'_v = l$ then there exists a path of length $l$ from $v$ to a black node $y$ such that $b'_v = ID_y$. It now remains to show that this path is the shortest path from $v$ to a black node different from $b_v$.

Let $v$ be a node with $b_v = ID_x$ in a stable configuration such that among the nodes with $b$-value set to $ID_x$, $v$ has the shortest distance $l$ to a black node $y$ where $y \neq x$. Let $y = w_0, w_1, \ldots, w_{l-1}, v$ be the nodes on this path. Then $y$ must be the closest black node to each $w_i$, $1 \leq i < l$, and by Lemma 1 we must have $p_{w_i} = i$ and $b_{w_i} = ID_y$. The support function applied to $v$ then has the opportunity to return the pair $(l, b_{w_{l-1}} = ID_y)$ for $(p'_v, b'_v)$. If it does not do so then this would indicate that there exists a black node different from $x$ that is closer to $v$ than $y$ is. This is a contradiction and the result follows.

Assume by induction that $p'_v$ and $b'_v$ are set correctly for every node with both $b_v = ID_x$ and with shortest distance $r$, $r \geq l$, to a black node other than $x$. Let $v$ now be a node with shortest distance $r + 1$ to a black node $y$ other than $x$. Then if the shortest path from $v$ to $y$ does not pass through any node with $b$-value set to $x$ the same argument as above shows that we must have $p'_v = r + 1$ and $b'_v = ID_y$. If the shortest path $y = w_0, w_1, \ldots, w_r, v$ does pass through at least one node with $b_{w_i} = ID_x$ then we must have $b_{w_r} = ID_x$. This follows since as soon as the shortest path from $v$ to $y$ leaves the $x$-domain it will not re-enter it. Thus by induction we have $p'_{w_r} = r$ and $b'_{w_r} = ID_y$ and in a stable configuration we must have $p'_v = r + 1$ and $b'_v = ID_y$. □

Note that if there is only one black node $y$ in $G$ then each node $v$ will have $b_v = y$ and the pair $(p'_v, b'_v)$ will be set to $(\infty, \infty)$.

From Lemmas 1 and 2 it follows that in a stable configuration any white node $v$ has $p_v$ equal to the distance to the nearest black node while any black node $w$ has $p'_w$ equal to the distance to the nearest black node other than itself. Thus if $x$ and $y$ are the two closest black nodes and with $ID_x < ID_y$ and distance $l$ from each other where $l \leq k$ then in a stable configuration we will have $p'_y = l$ and $b'_y \leq ID_y$. But with this configuration $y$ cannot keep $p_y = 0$ and is privileged for a move. Thus we have the following result.

**Lemma 3.** *In a stable configuration there does not exist a pair of black nodes where the minimum distance between them is less than or equal to $k$.*

Putting all of this together it is now straightforward to show that a stable solution is also a maximal $k$-packing.

**Theorem 1.** *A stable configuration is a maximal $k$-packing.*

*Proof.* From Lemma 3 it follows that in a stable configuration there cannot exist black nodes within distance $k$ of each other. Further from Corollary 1 we know that there cannot exist a non-privileged white node with distance greater than or equal to $k$ to every black node in the graph. Thus it follows that a stable configuration is a maximal $k$-packing. □

### 3.5 Convergence

Now that we have shown that once the algorithm stabilizes it has reached a valid solution we proceed to show that the algorithm will do so in a finite amount of steps. To reduce the complexity of the presentation we will assume that the algorithm in each move either updates $(p_v, b_v)$ or $(p'_v, b'_v)$ (and not both). While this is not entirely keeping with how Rule 1 functions, making this assumption does not affect the correctness of the analysis. Consider that once the support function for a node $v$ has returned, updating the two pairs $(p_v, b_v)$ and $(p'_v, b'_v)$ can be regarded as two separate moves where one has no bearing on the other.

Starting with $(p_v, b_v)$ we first divide the execution of Rule 1 into three different cases depending on the outcome of the move. These three cases are as follows:

**Black move.** A node is said to make a *black move* if after the move it has changed its color from white to black.

**Decremental move.** A node $v$ is said to make a *d-move* if it has changed $p_v$ to $\bar{p_v}$ and $b_v$ to $\bar{b_v}$ such that either $\bar{p_v} < p_v$ or $\bar{b_v} < b_v \wedge \bar{p_v} = p_v$.

**Incremental move.** A node $v$ is said to make an *i-move* if it has changed $p_v$ to $\bar{p_v}$ and $b_v$ to $\bar{b_v}$ such that either $\bar{p_v} > p_v$ or $\bar{b_v} > b_v \wedge \bar{p_v} = p_v$.

Note that we label a move by the first condition in increasing order that evaluates to true. For example, a node makes a black move if it has become black, even if the move also qualified as a d-move. We note that both the d- and i-moves can be defined for the $(p', b')$-values. It is then straightforward to see that the three different types of moves cover every possible move that the algorithm can make. In the following we will first reason that we cannot have an infinite sequence of d- and i-moves when only applied to the $(p, b)$-values.

To be able to reason about what causes a node to make a move we note that a locally stable node $v$ can only become privileged and make a new move if one of its neighbors $x$ first makes a move. If this is the first move among the neighbors of $v$ that causes $v$ to become privileged we will say that the move made by $x$ *initiated* the subsequent move by $v$. With this definition we can now show the following result.

**Lemma 4.** *A d-move cannot initiate an i-move.*

*Proof.* Consider a locally stable node $v$ with values $p_v$ and $b_v$. Then if $v$ is white there must exist a node $w \in N(v)$ such that $p_w + 1 = p_v$ and $b_w = b_v$. If $v$ is to make an i-move there cannot exist any node $u \in N(v)$ with either $p_u < p_v - 1$ or with $p_u = p_v - 1$ and $b_u \leq b_v$. In the case where $w$ does not make a move this is not true. Also, if $w$ makes a d-move then $w$ must decrease either its $p$-value or $b$-value (or both). In either case the condition for $v$ to make an i-move is not satisfied.

If $v$ is a locally stable black node then it will only make an i-move if some neighbor $w$ has $p_w < k$ and $b_w < ID_v$. But if this is not the case prior to when $w$ makes an i-move it will not be true after the move. □

From Lemma 4 it follows that in a sequence of moves by the nodes of $G$ that consists entirely of d- and i-moves one can analyze the number of i-moves independently from the d-moves. We will do this in the following, but first we show how many consecutive d-moves there can be.

**Lemma 5.** *The number of consecutive d-moves is at most* $O(n^2k)$.

*Proof.* After a node has executed its initial move (which might be a d-move) it will have a $p$-value in the range $[0, k]$. Thus it follows that a node can at most decrement its $p$-value $k$ times before it has to make an i-move. In addition a node can decrease its $b$-value while keeping its $p$-value fixed. Each node in the graph can at most give rise to one unique $b$-value. In addition there might be $n$ additional $b$-values in the graph due to the initial values. Thus for a fixed $p$-value a node might decrease its $b$-value at most $2n$ times. This gives a total of at most $2nk$ d-moves per node. □

Next, we analyze the i-moves and show that any sequence consisting entirely of i-moves must stabilize.

**Lemma 6.** *There cannot be an infinite sequence of i-moves.*

*Proof.* Let $\beta_1, \beta_2, \ldots, \beta_l$, $1 \leq l \leq 2n$, be an increasing sequence containing the set of distinct $b$-values that are used during the execution of the algorithm. This contains the values given by the $ID$s of the nodes as well as any initial $b$-values.

Define a vector $A = [a_{(0,\beta_1)}, a_{(0,\beta_2)}, \ldots, a_{(0,\beta_l)}, a_{(1,\beta_1)}, a_{(1,\beta_2)}, \ldots, a_{(1,\beta_l)}, \ldots, a_{(k,\beta_1)}, a_{(k,\beta_2)}, \ldots, a_{(k,\beta_l)}]$ where entry $a_{(i,\beta_j)}$ is the number of nodes in the graph at any one time with $p$-value equal to $i$ and $b$-value equal to $\beta_j$ that are privileged for an i-move. Note that only a node $v$ with $0 \leq p_v < k$ can be privileged for an i-move, thus every node that is privileged for an i-move is represented in $A$ and the sum of the elements in $A$ is always bounded by $n$. We will now show that if an i-move changes $A$ to $A'$ then $A > A'$ where the comparison is done by viewing each vector as a number consisting of at most $2(k+1)n$ digits.

Consider a node $v$ that makes an i-move and let $p_v, b_v$ be the associated values of $v$ before the move. Then the value in position $(p_v, b_v)$ of $A$ will be reduced by one and since $v$ will not be privileged for a new i-move immediately after this move $v$ will not directly cause any other entry in $A$ to change. In addition, any node $w \in N(v)$ that had $v$ as its only neighboring node with either $p_v < p_w - 1$ or with $p_v = p_w - 1$ and $b_v \leq b_w$ before the move and where either $p_v > p_w - 1$ or $p_v = p_w - 1$ and $b_v > b_w$ is true after the move has now become privileged for an i-move. If this is the case the entry $a_{(p_w,b_w)}$ will increase by one for each such node $w$. But since the initial value of $p_v$ is less than $p_w$ it follows that $A > A'$. To see that any consecutive sequence of i-moves must terminate after a finite number of moves it is sufficient to note that we cannot have negative numbers in $A$ and that the sum of the entries in $A$ is always bounded by $n$. □

The immediate bound obtained from the proof of Lemma 6 is fairly pessimistic as there are an exponential number of distinct configurations of the $A$ vector used in the proof. Still, together with Lemmas 4 and 5 it shows that any sequence of d-moves and i-moves must stabilize. The only time where the $(p', b')$ values might affect the $(p, b)$ values is when making a black node privileged to perform an i-move. But then the node ceases to be black and as long as we don't allow for any black nodes this can at most happen once for each node.

To see that the i- and d-moves on the $(p', b')$ values also must stabilize it is sufficient to note that for fixed $(p, b)$ values the i- and d-moves on $(p', b')$ behaves in the same way as on the $(p, b)$ values. Thus it follows that between each set of i- and d-moves on the $(p, b)$ values we can at most have a finite number of i- and d-moves on the $(p', b')$ values. Thus we have the following result.

**Lemma 7.** *Any sequence of i- and d-moves applied to both the* $(p, b)$ *and* $(p', b')$ *values is bounded.*

It now remains to incorporate the black moves into the analysis. We do this with in the following.

**Lemma 8.** *There cannot be an infinite sequence of black moves.*

*Proof.* Any black node $v$ at the start of the algorithm where $b_v \neq ID_v$ will be corrected by the first move $v$ makes and thus there can at most be $n$ such moves. Thus for the rest of the analysis we assume that $b_v = ID_v$ for every black node.

Similar to in the proof of Lemma 6 we define the vector $A = [a_{(1,\beta_1)}, a_{(1,\beta_2)}, \ldots, a_{(1,\beta_l)}, a_{(2,\beta_1)}, a_{(2,\beta_2)}, \ldots, a_{(2,\beta_l)}, \ldots, a_{(k,\beta_1)}, a_{(k,\beta_2)}, \ldots, a_{(k,\beta_l)}]$ where entry $a_{(j,\beta_i)}$ is the number of nodes in the graph at any one time with $p$-value equal to $j$ and $b$-value equal to $\beta_i$ that are privileged for an i-move on the $(p, b)$ values. Again, we also assume that the different values of $\beta_i$ span all possible values (at most $2n$) and that $\beta_i < \beta_{i+1}$.

It is then clear that only a domain where $\beta_i$ corresponds to the $ID_v$ of some $v \in V$ can contain a black node as a leader and there can only be one such black node at a time (apart from at start up).

Let $v$ be the node with lowest $ID$ among the nodes in $G$. Then $a_{(0,ID_v)}$ is the leftmost position in $A$ that can correspond to a black node. If $v$ is black it can only become white due to a node $w \in N(v)$ with values such that either $p_w < k$ and $b_w < b_v$ or that $p'_w < k$ and $b'_w < b_v$ (in which case $b_w = b_v$). Denote the one of $b_w$ and $b'_w$ that caused this to happen by $b''_w$ and let $U$ be the domain containing $b''_w$. Since $v$ had the lowest ID among the nodes in $G$ it follows that $U$ is improper, and must have a leader $u$ whose $b$-value does not equal the $ID$ of any node.

For $v$ to become black again the value of $p_w$ must increase to at least $k$. This cannot happen until $u$ makes an i-move and increases its $p$-value. Thus between each time $v$ becomes black some node belonging to an improper domain must make an i-move. It follows from the proof of Lemma 6 that this can only happen a finite number of times.

Now assuming that the $r$ nodes with lowest $ID$s, $r \geq 1$, can only change between white and black a finite number of times we will show that this implies that the node $v$ with the $(r+1)$st smallest $ID$ also only can change between white and black a finite number of times.

Let $R$ denote the set of domains with lower $ID$s than $v$. Then using the same argument as above it follows that between each time $v$ changes from black to white some node in $R$ must have executed an i-move. We know that each such move will cause $A < A'$ and that any d-moves will not change any value of $A$. Thus between each time some domain in $R$ executes a black move $v$ can only perform a finite number of black moves. Since by assumption each proper domain in $R$ can only execute a finite number of black moves the result follows. □

Combining the results from lemmas 5 through 8 we now have our main result.

**Theorem 2.** *Algorithm* **Rule 1** *will stabilize in a finite number of moves.*

## 4 Conclusion

We have presented a very simple self-stabilizing algorithm that solves the $k$-packing problem. In doing so it only uses a constant number of variables per node. The main mechanism for solving the problem is a method for a black node to compute the distance to its nearest black node other than itself. We believe that this mechanism can be used in designing self-stabilizing algorithms for other problems that also involves some $k$-distance property. This is something we intend to study further in the future.

We do not believe that this idea can be extended to an anonymous network, since a white node $v$ would not be able to distinguish between black nodes that are not in $N(v)$.

Still, the main open question is to better understand the trade off between memory usage and moves complexity in self-stabilizing algorithms. There are currently few hardness results in terms of moves complexity in the literature on self-stabilizing algorithms and even if some self-stabilizing algorithms require an exponential number of moves there is still room for ranking these like one is currently seeing in the field of exact sequential algorithms.

## References

1. C. Berge, *Theory of Graphs and its Applications*, no. 2 in Collection Universitaire de Mathematiques, Dunod, Paris, 1958.
2. J. Blair and F. Manne, *Efficient self-stabilzing algorithms for tree networks*, in Proceedings of ICDS 2003, The 23rd IEEE International Conference on Distributed Computing Systems, 2003, pp. 20–26.
3. E. W. Dijkstra, *Self-stabilizing systems in spite of distributed control*, CACM, 17 (1974), pp. 643–644.
4. S. Dolev, *Self-stabilization*, MIT press, 2000.
5. M. Gairing, R. M. Geist, S. T. Hedetniemi, and P. Kristiansen, *A self-stabilizing algorithm for maximal 2-packing*, Nordic J. Comput., 11 (2004), pp. 1–11.
6. M. R. Garey and D. S. Johnson, *Computers and Intractability*, W. H. Freeman and Co., 1978.
7. F. Gärtner, *A survey of self-stabilizing spanning-tree algorithms*, Tech. Report IC/2003/38, Swiss Federal Institute of Technology, 2003.
8. W. Goddard, S. Hedetniemi, D. Jacobs, and V. Trevisan, *Distance-k information in self-stabilizing algorithms*, in Proceedings of SIROCCO 2006, 2006. To appear.
9. W. Goddard, S. T. Hedetniemi, D. P. Jacobs, and P. K. Srimani, *Self-stabilizing global optimization algorithms for large network graphs*, Int. J. Dist. Sensor Networks, 1 (2005), pp. 329 – 344.
10. M. A. Henning, *Distance domination in graphs*, in Domination in Graphs: Advanced Topics, T. W. Haynes, S. T. Hedetniemi, and P. J. Slater, eds., Marcel Dekker, New York, 1998, pp. 321–349.

11. M. Mjelde, *$k$-packing and $k$-domination on tree graphs*, master's thesis, Department of Informatics, University of Bergen, Norway, 2004.
12. O. Ore, *Theory of Graphs*, no. 38 in American Mathematical Society Publications, AMS, Providence, 1962.
13. P. J. Slater, *$R$-domination in graphs*, J. Assoc. Comput. Mach., 23 (1976), pp. 446–450.