A Self-stabilizing Algorithm for Maximal 2-packing

Martin Gairing[1,*], Robert M. Geist[2], Stephen T. Hedetniemi[2], and Petter Kristiansen[3,*]

[1] Faculty of Computer Science, Electrical Engineering and Mathematics, University of Paderborn, 33102 Paderborn, Germany
gairing@uni-paderborn.de

[2] Department of Computer Science, Clemson University, Clemson, SC 29634, USA
{rmg, hedet}@cs.clemson.edu

[3] Department of Informatics, University of Bergen, N-5020 Bergen, Norway
petterk@ii.uib.no

Abstract

In the self-stabilizing algorithmic paradigm for distributed computing each node has only a local view of the system, yet in a finite amount of time the system converges to a global state, satisfying some desired property. In a graph $G = (V, E)$, a subset $S \subseteq V$ is a 2-packing if $\forall u \in V : |N[u] \cap S| \leq 1$. In this paper we present an ID-based, self-stabilizing algorithm for finding a maximal 2-packing, a non-local property, in an arbitrary graph. We also show how to use Markov analysis to analyse the behaviour of a non-ID-based version of the algorithm.

CR Classification: C.2.4, D.1.3, F.2.2, G.2.2, G.3
Key words: self-stabilizing algorithms, 2-packing, Markov analysis

1 Introduction

A distributed system can be modelled with an undirected graph $G = (V, E)$, with node set V and edge set E. If i is a node, then $N(i)$, its *open neighbourhood*, denotes the set of nodes to which i is adjacent. Every node $j \in N(i)$ is called a *neighbour* of node i. $N[i]$, the *closed neighbourhood* of i, is defined as $N[i] = N(i) \cup \{i\}$. If S is a set of nodes, then $N(S) = \bigcup_{i \in S} N(i)$. The *boundary* of a set of nodes S is defined as $\partial(S) = N(S) \setminus S$. The *distance* $\text{dist}(i, j)$ between two nodes i and j is the number of edges in a shortest $i - j$ path. A graph $G' = (V', E')$ is a *subgraph* of G, written $G' \subseteq G$, if $V' \subseteq V$ and $E' \subseteq E$. If $G' \subseteq G$ and E' contains all the edges $ij \in E$ with $i, j \in V'$, then G' is an *induced subgraph* of G; we say that V' *induces* G', and write $G' = G[V']$. We

*This work was done while the author was visiting Clemson University.

say that a set of nodes in a graph is *independent* if no edge exists between any pair of nodes in the set.

A subset S of nodes in a graph $G = (V, E)$ is called a *2-packing* [9] if $\forall i \in V : |N[i] \cap S| \leq 1$. A 2-packing S is *maximal* if no proper superset of S is a 2-packing.

Self-stabilization is a relatively new paradigm for distributed systems that allows a system to achieve a desired global state, even in the presence of faults. The concept was introduced in 1974 by Dijkstra [1], but serious work on self-stabilizing algorithms did not start until the late 1980s. (See [11, ch. 15] for an introduction to self-stabilizing algorithms.) In this algorithmic paradigm, each node executes the same set of instructions, maintains its own set of local variables, and changes the values of its local variables based on the current values of its variables and those of its neighbours. A node can not access information beyond its closed neighbourhood, and can only update its own local variables. The contents of a node's local variables determine its *local state*. The system's *global state* is the union of all local states.

When a node changes its local state, it is said to make a *move*. Our algorithm is given as a set of rules of the form **Rk: if** $p(i)$ **then** M, where $p(i)$ is a predicate for node i, and M is a move which involves the change in value of one or more of i's local variables. Node i becomes *privileged* if $p(i)$ is true. When a node becomes privileged, it may execute the corresponding move. We assume there exists a daemon, an adversarial oracle, as introduced in [1], which selects one of the privileged nodes. The selected node then makes a move. Following the serial model, we assume that no two nodes move at the same time. The goal of the daemon is to keep the algorithm running for as long as possible. When no further moves are possible, we say that the system is *stable*, or is in a stable state. We say that a self-stabilizing algorithm is *correct* if, when the system executes the algorithm,

1) every stable state it can reach is legitimate, that is, every stable state has the desired global property, and

2) it always reaches a stable state after a finite number of moves.

Problems that can be solved by a straightforward greedy method in the conventional algorithmic model often require a far more clever approach in the self-stabilizing model. For example, finding a maximal matching (i.e. a set of disjoint edges that cover all remaining edges) in a graph is trivial: starting with an empty set of edges, keep adding another disjoint edge to the set as long as one exists. To describe and prove the correctness of a self-stabilizing algorithm for the same problem takes considerably more effort [7, 10, 5].

In this paper we present a so-called ID-based, self-stabilizing algorithm for finding a maximal 2-packing in an arbitrary graph. The term *ID-based* refers to the fact that our algorithm assumes a unique identifier is associated with each node in the network. In practical networks, nodes usually have some form of identification, e.g. an IP-address, totally anonymous networks are more of a 'mathematical nicety'. 2-packings are vertex subsets with some inherent

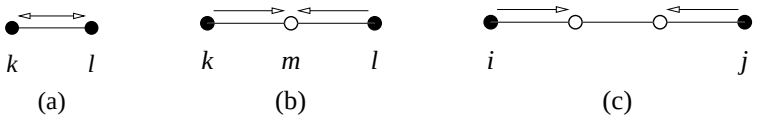

Figure 1: Diagrams (a), (b), and (c) depict exchange of membership information in the network. (Black colour indicates membership, arrows indicate possible flow of membership information.)

non-locality, in that no two nodes in the 2-packing can have overlapping neighbourhoods. This prevents nodes in a 2-packing from having direct knowledge of each other. A self-stabilizing algorithm for maximal independent sets, i.e. 1-packings, was presented by Hedetniemi et al. in [4]. Due to the inherent non-locality of 2-packings, the algorithm we present here is considerably more complex, both in design and verification. With 1-packings two nodes that lie to close, i.e. are neighbours, both have direct knowledge of each other, allowing for a much simpler, and faster, algorithm.

We believe that the algorithm presented here is interesting, not only in its own right, as a continuation of the study of self-stabilizing algorithms for finding maximal independent sets, and similar vertex subsets in graphs, but also as a possible subroutine in algorithms for more complex problems where one needs to ensure mutual exclusion between nodes with overlapping neighbourhoods during execution. One example of an algorithm for such a problem is the algorithm for finding a minimal $\{k\}$-dominating function, presented by Gairing et al. in [2]. (Given a graph $G = (V, E)$, a $\{k\}$-*dominating function* is a function $f : V \to \{0, 1, 2, \ldots, k\}$, such that $\sum_{j \in N[i]} f(j) \geq k$, for all $i \in V$, see e.g. [3].) The algorithm in [2] uses a slight modification of the algorithm presented here as a subroutine.

2 Maximal 2-packing and Self-stabilization

The facts below follow directly from the definition of a maximal 2-packing.

Fact 1. Every 2-packing S is an independent set, that is, no two nodes in S are neighbours.

Fact 2. If S is a 2-packing, then $\forall i, j \in S : \text{dist}(i, j) \geq 3$.

Fact 3. If S is a maximal 2-packing, then $\forall i \in V \setminus S \; \exists j \in S : \text{dist}(i, j) \leq 2$.

Fact 2 states that for every pair of nodes i, j in a 2-packing S, we must have $\text{dist}(i, j) \geq 3$. In a self-stabilizing algorithm nodes can only access information in their respective neighbourhoods. This implies that node i is unable to correctly convey membership information to node j, and vice versa. Figure 1 (c) depicts this situation. However, given a set T that is not a 2-packing, we know there

exists a pair of nodes k, l in T with $\text{dist}(k,l) \leq 2$. There are two cases to consider: 1) $\text{dist}(k,l) = 1$, and 2) $\text{dist}(k,l) = 2$. In case 1) both k and l can detect the error by accessing membership information in their respective neighbourhoods. This is depicted in Figure 1 (a). One of the nodes can then change its local state to correct the error. In case 2) there will exist a node $m \in N(k) \cap N(l)$ such that $m \notin T$. Node m will detect the error, and can in turn inform either k or l of the violation, so that this node can correct the error. This is depicted in Figure 1 (b).

The above discussion shows that, given a proposed 2-packing, i.e. a subset of nodes in a network, all nodes that violate the 2-packing condition can be detected by at least one node in the network; even when the nodes are only able to access membership information in their respective neighbourhoods. But merely detecting violations is not enough. Figure 1 (b) shows that some mechanism is needed to inform the violating nodes that they must change their local states, when they are unable to detect the violation themselves. In the algorithm described below we use pointers to devise such a mechanism. Assume S is the set we are trying to alter into a maximal 2-packing. A node $i \in V \setminus S$ with one or more neighbours in S will point to exactly one of these neighbours, say j. This indicates to all nodes in $(N(i) \cap S) \setminus \{j\}$ that they are violating the 2-packing condition and should therefore change their local states, and to all nodes in $N(i) \cap (V \setminus S)$ that i already has a neighbour in S. Nodes without neighbours in S, and nodes in S, should point to NULL.

Any self-stabilizing algorithm for finding a maximal 2-packing S must obviously incorporate a rule by which nodes not in S can become members of S. (Our system may for instance have an initial global state in which S is empty.) Since the nodes are only able to access information in their respective neighbourhoods, the algorithm shall let any node $i \in V \setminus S$ become a member of S if, by accessing information (both membership information and the pointers described above) in its neighbourhood, i is unable to determine that it will violate the 2-packing condition by doing so.

To avoid infinite loops in the execution of our algorithm, we need to augment the pointer mechanism described above. Figure 2 illustrates how the daemon can make an algorithm repeat a global state in a four-cycle infinitely many times, by cleverly choosing where each node points, if the algorithm uses only membership information and the described pointer mechanism. In the algorithm described below we use pointers in conjunction with unique identifiers on the nodes. This allows us to design an algorithm where no global state can be repeated. With unique identifiers we can limit the daemon's freedom to choose where each node points, thereby avoiding infinite loops.

3 An ID-based 2-packing Algorithm

We now formally present our ID-based, self-stabilizing algorithm for identifying a maximal 2-packing in an arbitrary graph. Note that we are not trying to obtain a maximal 2-packing of largest cardinality, merely a maximal one. The

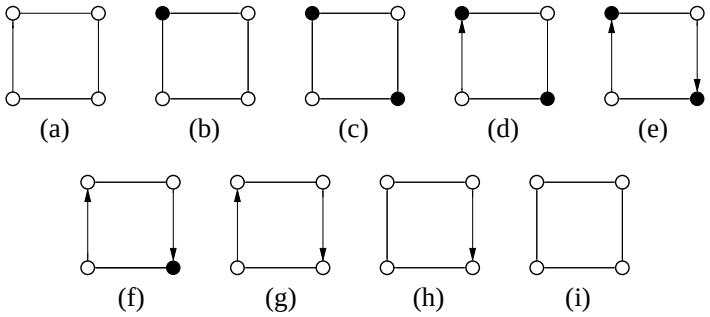

Figure 2: Diagrams (a) – (i) depict an infinite loop on a four-cycle. (Black colour indicates membership, and arrows indicate pointers.)

problem of finding a maximal 2-packing of largest cardinality was shown to be $\mathcal{NP}$-hard by Hochbaum and Schmoys in [6], finding a maximal one is easily done in linear time (with the standard RAM model).

We assume that each node i in the network has a unique identifier $\mathrm{id}(i)$, and that there exists a total ordering of these identifiers. In our algorithm, each node i has a boolean variable $x(i)$ indicating membership in the 2-packing we are trying to construct. That is, $x(i) = 1$ if node i is an element of the 2-packing we are trying to construct, and $x(i) = 0$ otherwise. Furthermore, each node i has a pointer that can point to any neighbour $j \in N(i)$ or to NULL. The nodes not in the 2-packing use this pointer to limit their number of neighbours in the 2-packing. We use the notation $i \to j$ and $i \to \mathrm{NULL}$ to denote that the pointer of i points to j and that the pointer of i points to NULL, respectively. The algorithm consists of six rules, **R1** – **R6**, and is identically stored and executed in each node i.

Algorithm 1: Maximal 2-packing

---

uses $\mathrm{id}(\_)$ *identifier*;
local $x(\_)$ *boolean*,
$\_\to\_$ *pointer*;

R1: **if** $x(i) = 0 \ \wedge \ \forall j \in N(i) : x(j) = 0 \wedge (j \to i \vee j \to \mathrm{NULL})$
then $\begin{cases} x(i) = 1; \\ i \to \mathrm{NULL}; \end{cases}$

R2: **if** $x(i) = 0 \ \wedge \ i \to j \ \wedge \ \forall k \in N(i) : x(k) = 0$
then $\begin{cases} i \to \mathrm{NULL}; \\ \textbf{if } \forall l \in N(i) : l \to i \vee l \to \mathrm{NULL} \\ \quad \textbf{then } x(i) = 1; \end{cases}$

R3: **if** $x(i) = 0 \ \wedge \ ((i \to \text{NULL}) \vee (i \to j \wedge x(j) = 0)) \ \wedge$
$\exists k \in N(i) \ : \ x(k) = 1$
then $i \to k$, where $\text{id}(k) = \min\{\text{id}(l) : l \in N(i) \wedge x(l) = 1\}$;

R4: **if** $x(i) = 1 \ \wedge \ \exists j \in N(i) \ : \ x(j) = 1$
then $\begin{cases} x(i) = 0; \\ i \to j, \text{ where } \text{id}(j) = \min\{\text{id}(l) : l \in N(i) \wedge x(l) = 1\}; \end{cases}$

R5: **if** $x(i) = 1 \ \wedge \ \forall j \in N(i) \ : \ x(j) = 0 \ \wedge \ \exists k \in N(i) \ : \ k \to l, l \neq i$
then $\begin{cases} x(i) = 0; \\ i \to \text{NULL}; \end{cases}$

R6: **if** $x(i) = 1 \ \wedge \ \neg(i \to \text{NULL}) \ \wedge$
$\forall j \in N(i) \ : \ x(j) = 0 \wedge (j \to i \vee j \to \text{NULL})$
then $i \to \text{NULL}$;

We shall use the term *x-move* if a move changes the x-value of a node, and *pointer-move* if a move only changes the pointer-value of a node. Rules **R1**, **R2** if the second then-statement is executed, **R4**, and **R5** are x-moves, whereas rules **R2** if the second then-statement is not executed, **R3**, and **R6** are pointer-moves.

4 Verification

For simplicity we shall in the following say that node i is *black* if $x(i) = 1$, and *white* if $x(i) = 0$. We say that a black node is a *singleton black* if all its neighbours are white. A black node i is *locked* if all nodes in $N(i)$ are white and point to i, and $i \to \text{NULL}$. Note that if a node i is locked, then no node in $N[i]$ will make another move until the algorithm stabilizes. For the purpose of this paper we define a *black enclave*[1] to be a maximal set V' of black nodes, such that $|V'| \geq 2$ and $G[V']$ is connected. A *white enclave* is a maximal set V'' of nodes such that $G[V'']$ is connected and V'' does not contain any adjacent black nodes. (Note that while the black enclaves only contain black nodes, we allow the white enclaves to contain singleton black nodes.)

The system is in a *legitimate state* if and only if 1) the x-values correctly reflect a maximal 2-packing, 2) all black nodes point to NULL, and 3) all white nodes point to their only black neighbour, if they have one, and to NULL otherwise.

To prove correctness of Algorithm 1 we need to show each of the following:

I. If the system reaches a stable state, then it is in a legitimate state, or

[1]The Oxford English Dictionary defines an enclave to be a portion of territory entirely surrounded by foreign dominions.

equivalently, if the system is not in a legitimate state, then at least one node can make a move.

II. Regardless of the initial state and regardless of the sequence of moves selected by the daemon, the system is guaranteed to reach a stable state after a finite number of moves.

The following two lemmata show that the proposed algorithm satisfies requirement I.

Lemma 1. *If the system reaches a legitimate state, then no node can make a move.*

Proof. This is obvious since no predicate of any rule **R1** – **R6** will be true. □

Lemma 2. *If the system is in an illegitimate state, then there exists at least one node that can make a move.*

Proof. Assume the system is in an illegitimate state. There are three cases to consider: 1) the x-values do not reflect a 2-packing, 2) the x-values reflect a non-maximal 2-packing, and 3) the x-values reflect a maximal 2-packing, but the pointers are incorrect. We must show that in each case at least one node can make a move.

In case 1) there will either exist a black node adjacent to another black node or a white node adjacent to two or more black nodes. In the former sub-case one of the black nodes can execute rule **R4**, and in the latter rule **R5** (possibly after the white node has executed rule **R3**).

In case 2) there will exist a white node with no black nodes within a distance of two. In this case the white node can execute rule **R1**, or a neighbour of it rule **R2**.

In case 3) there will be a black node not pointing to NULL, a white node pointing to a white node, or a white node not pointing to its black neighbour. The black node can execute rule **R6**, the white node pointing to a white node rule **R2**, and the white node not pointing to its black neighbour rule **R3**. □

To prove that our algorithm meets requirement II, we first show that all black enclaves must shrink until no adjacent pair of black nodes exists in the network, at which point the entire network can be viewed as a white enclave. We then show that the algorithm will arrange the singleton black nodes into a legitimate global state and stabilize.

The following lemma shows that we can concentrate on the x-moves when analysing the behaviour of Algorithm 1.

Lemma 3. *There can be at most $|V|$ consecutive pointer-moves without intervening x-moves.*

Proof. Immediately after making a pointer-move, node i is unprivileged, none of the predicates of the six rules can be true for i after any of the tree possible pointer-moves. No subsequent change of the pointer in any neighbour of i can

make the predicates of rules **R3** or **R6** true for i. A pointer move by a neighbour of i may make the predicate of rule **R2** true for i, but only if the condition of the second if-statement is true. The next move by node i must therefore be an x-move. This, of course, applies to all nodes, and in a sequence of consecutive pointer-moves, each node can make at most one move. □

The following three lemmata show that all black enclaves must shrink in size.

Lemma 4. *If C is a white enclave, and no node in $\partial(C)$ makes a move, then the nodes in C can only make a finite number of moves.*

Proof. Let $B \subset C$ be the set of singleton black nodes in C. If $B \neq \emptyset$ and a node $b \in B$ has a white neighbour pointing away from it, then the daemon can force b to change from black to white, using rule **R5**.

When a new singleton black node c is introduced into $B \subset C$, by $c \in C$ executing either rule **R1** or **R2**, we know that no node in $N(c)$ is pointing away from it. Therefore, c can only change back to white if a node in $N(c)$ makes a pointer-move, and points to another black node c'. This can only happen if $\mathrm{id}(c') < \mathrm{id}(c)$. Every node $d \in \partial(C)$ can possibly force all $b \in B$ to change to white, if d lies at the end of chain of nodes changing from black to white. However, each $d \in \partial(C)$ can only force such a move $|N(d) \cap C|$ times, as a new pointer to d will be added every time.

Assume B, as a whole, has changed to white $|N(\partial(C)) \cap C| + 1$ times (once initially, and once for each neighbour of $\partial(C)$ in C, as described above). At this point we know that when a new singleton black node is introduced into $B \subset C$, we will always have $B \neq \emptyset$. A singleton black node b can only be replaced by another singleton black node b' when $\mathrm{id}(b') < \mathrm{id}(b)$. This implies that a singleton black node $b^* \in B \subset C$ at some point will become minimal, in the sense that no node $c \in C$ with $\mathrm{id}(c) < \mathrm{id}(b^*)$ can change to black and force b^* to make an x-move. Singleton black nodes can be introduced into $B \subset C$ in this way until no further x-moves are possible in C. At this point all neighbours of every minimal node b^*, not already pointing to b^*, will execute rule **R3** and point to b^*, so that b^* becomes locked. Since b^* is locked, all black nodes in $N(N(b^*))$ are forced to change from black to white. After doing so, no node in $N(N(b^*))$ can make any further x-moves. We can therefore disregard all nodes in $N(N(b^*))$, for each of the locked nodes b^*, and make a similar recursive argument for the rest of C. □

Lemma 5. *No black enclave can increase in size, and no white enclave can decrease in size.*

Proof. The only rule that can be executed by a node in a black enclave is rule **R4**, which decreases the size of the black enclave by one. And it is obvious from the predicates of rules **R1** and **R2**, that Algorithm 1 never can create two adjacent black nodes. Thus, no black enclave can increase in size, and consequently, no white enclave can decrease in size. □

Lemma 6. *The size of all black enclaves must shrink.*

Proof. It follows from Lemma 4 that only a finite number of moves can be made by the nodes of the white enclaves, before a move by a node in a black enclave is forced. Lemma 5 states that no black enclave can increase in size. This implies that the size of all black enclaves must shrink. □

Lemma 7. *If all black nodes are singletons, then Algorithm 1 stabilizes.*

Proof. When all black nodes are singletons, the whole network can be viewed as a white enclave. Let B be the set of singleton black nodes. If $B \neq \emptyset$ and any node $b \in B$ has a white neighbour pointing away from it, then the daemon can force b to change from black to white, using rule **R5**.

Assume B, as a whole, has changed to white once. At this point we know that when a new singleton black node is introduced into B, we will always have $B \neq \emptyset$. A singleton black node b can only be replaced by another singleton black node b' when $\mathrm{id}(b') < \mathrm{id}(b)$. This implies that a singleton black node $b^* \in B$ at some point will become minimal, in the sense that no node c with $\mathrm{id}(c) < \mathrm{id}(b^*)$ can change to black and force b^* to make an x-move. Singleton black nodes can be introduced into B in this way until no further x-moves are possible. At this point all neighbours of every minimal node b^*, not already pointing to b^*, will execute rule **R3** and point to b^*, so that b^* becomes locked. Since b^* is locked, all black nodes in $N(N(b^*))$ are forced to change from black to white. After doing so no node in $N(N(b^*))$ can make any further x-moves. We can therefore disregard all nodes in $N(N(b^*))$, for each of the locked nodes b^*, and make a similar recursive argument for the rest of the network. □

The above discussion shows that Algorithm 1 finds a maximal 2-packing in an arbitrary graph within finite time.

Theorem 1. *Algorithm 1 stabilizes with a maximal 2-packing in finite time.*

Proof. It follows from Lemmata 1 and 2 that no node can make a move if the system has stabilized, and that at least one node can make a move otherwise. And it follows from Lemmata 6 and 7 that the algorithm terminates in finite time. □

The above discussion shows that we can guarantee termination, albeit in exponential time. (For a path on n nodes with labels increasing from left to right a lower bound of $\Omega(2^{O(n)})$ moves is easily shown.) Simple simulations however, show that Algorithm 1 almost always terminates in linear time when run with a randomized daemon, instead of with the usual adversarial oracle.

5 Markov Analysis

In Algorithm 1 we used IDs to ensure termination. Here we briefly discuss a non-ID-based variant of the algorithm, and show how Markov analysis may be used to analyse its behaviour. We now assume the daemon only makes random

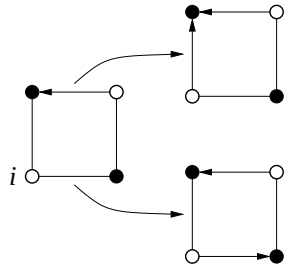

Figure 3: Non-determinism in rule **R3′**

choices when choosing a privileged node, and where a node points when it makes a pointer-move. Assuming the daemon is randomized, rather than an adversarial oracle, may be more relevant from a practical viewpoint. We can now simplify rules **R3** and **R4**, as follows:

R3′: **if** $x(i) = 0 \ \wedge \ ((i \rightarrow \text{NULL}) \vee (i \rightarrow j \wedge x(j) = 0)) \ \wedge$
$\exists k \in N(i) \ : \ x(k) = 1$
then $i \rightarrow k$;

R4′: **if** $x(i) = 1 \ \wedge \ \exists j \in N(i) \ : \ x(j) = 1$
then $\begin{cases} x(i) = 0; \\ i \rightarrow j; \end{cases}$

The non-determinism in rules **R3′** and **R4′** can lead to cycles in the state space, and it is important to quantify the potential ill-effects. As long as each state is *transient*, i.e., has at least one non-zero probability path to the termination state, the algorithm will terminate with probability 1. Lemma 2, which also holds with the above modifications to rules **R3** and **R4**, ensures this tranciency for any graph. Nevertheless, the time to termination may be arbitrarily long, and it is useful to determine its expected value.

Consider the four-cycle. Rule **R3′** can be applied to node i of the state shown in the left hand side of Figure 3 to yield either of the states shown on the right. The upper transition is the 'correct' choice, in that this path will lead to termination after one more step. The lower transition is the 'incorrect' choice, in that it may lead to termination or it may, after five more steps, return to the state on the left. The ill-effects of the incorrect choice can be completely quantified. The reduced state space for the four-cycle, with associated transitions, is shown in Figure 4. (The state space is reduced by grouping together states with identical stochastic behaviour.) Labels on edges show non-unit transition probabilities, wherever such can occur. Probability q is the probability of making the 'incorrect' choice in the application of rule **R3′**. We also assume a single, invisible 'regeneration' edge from the termination state T to the start state S, an analytic trick to convert a transient chain with single absorbing state into an

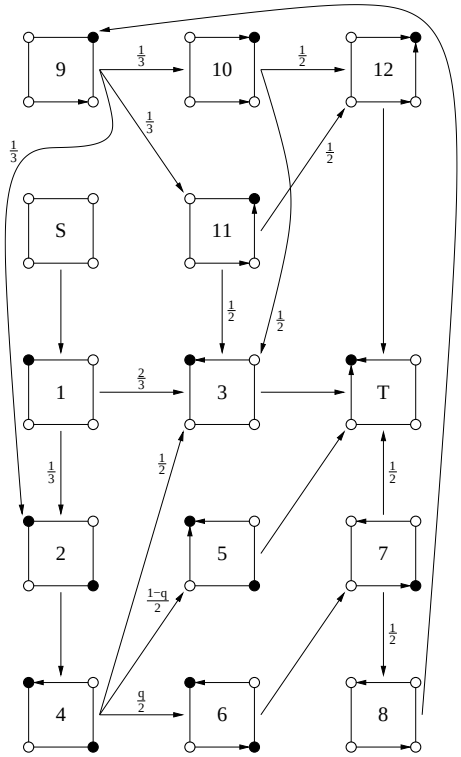

Figure 4: State space for the four-cycle

ergodic chain, i.e., one in which any state can be reached from any other state, and the number of transitions required to reach it is aperiodic.

The fundamental theorem for ergodic chains (see e.g. [8, 12]) states that if $p_{m,n}$ is the probability of transition from state m to state n in an ergodic Markov chain, and P the corresponding matrix, then the chain has a unique steady-state distribution π given by

$$\pi = \pi P \ ,$$

subject to $\Sigma_m \pi_m = 1$. Further, the expected number of transitions required to return to state m, the so-called *mean recurrence time*, is the reciprocal of the steady state probability, $1/\pi_m$.

In the case at hand, the matrix P is sparse, and the equation is easily solved. In particular, the mean recurrence time for the start state S is

$$\frac{1}{\pi_S} = \frac{168 + 5q}{36 - 3q} \ .$$

Now, because it requires exactly one transition in the ergodic chain to go from the termination state T to the start state S we have determined that the ex-

pected time to termination is

$$\frac{1}{\pi_S} - 1 = \frac{132 + 8q}{36 - 3q} \ .$$

Thus, in the optimistic case ($q = 0$), we expect 3.67 steps to termination, and, in the pessimistic case ($q = 1$), we expect 4.24 steps to termination, a relatively small increase.

It is clear that a similar analysis can be carried out for any specific graph, but the state spaces become large and the task becomes arduous very quickly. An automated, numerical Markovian analysis might be developed for arbitrary graphs, but it is unclear whether such would offer any benefits over straightforward simulation.

6 Concluding Remarks

In this paper we have presented an ID-based, self-stabilizing algorithm for finding a maximal 2-packing in an arbitrary graph. We have seen that 2-packing has some inherent non-local properties, but that it still is possible, at the expense of running time, to design a self-stabilizing algorithm for finding a maximal 2-packing in an arbitrary graph. We have also briefly shown how to use Markov analysis to analyse the behaviour of a non-ID-based version of the algorithm on small graphs. We believe the algorithm presented here may be useful as a subroutine in other more complex algorithms, where mutual exclusion between nodes with overlapping neighbourhoods is required during execution. One such example is presented in [2].

References

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

[2] Martin Gairing, Stephen T. Hedetniemi, Petter Kristiansen, and Alice A. McRae. Self-stabilizing algorithms for $\{k\}$-domination. In Shing-Tsaan Huang and Ted Herman, editors, *Self-Stabilizing Systems, 6th International Symposium, SSS 2003, San Fransico, CA, USA, June 24–25, Proceedings*, volume 2704 of *Lecture Notes in Computer Science*, pages 49–60. Springer-Verlag, 2003.

[3] Theresa W. Haynes, Stephen T. Hedetniemi, and Peter J. Slater. *Domination in Graphs*. Marcel Dekker, 1998.

[4] Sandra M. Hedetniemi, Stephen T. Hedetniemi, David P. Jacobs, and Pradip K. Srimani. Self-stabilizing algorithms for minimal dominating sets and maximal independent sets. *Computers and Mathematics with Applications*, to appear.

[5] Stephen T. Hedetniemi, David P. Jacobs, and Pradip K. Srimani. Maximal matching stabilizes in time $O(m)$. *Information Processing Letters*, 80(5):221–223, 2001.

[6] Dorit S. Hochbaum and David B. Schmoys. A best possible heuristic for the k-center problem. *Mathematics of Operations Research*, 10(2):180–184, 1985.

[7] Su-Chu Hsu and Shing-Tsaan Huang. A self-stabilizing algorithm for maximal matching. *Information Processing Letters*, 43(2):77–81, 1992.

[8] H. Kobayashi. *Modeling and Analisys*. Addison Wesley, Massachusetts, 1978.

[9] A. Meir and J. W. Moon. Relations between packing and covering numbers of a tree. *Pacific Journal of Mathematics*, 61(1):225–233, 1975.

[10] Gerard Tel. Maximal matching stabilizes in quadratic time. *Information Processing Letters*, 49(6):271–272, 1994.

[11] Gerard Tel. *Introduction to distributed algorithms*. Cambridge University Press, second edition, 2000.

[12] K. S. Trivedi. *Probability and Statistics with Reliability, Queueing, and Computer Science Applications*. Prentice Hall, Englewood Cliffs, NJ, 1983.