A self-stabilizing algorithm for finding a spanning tree in a polynomial number of moves

Adrian Kosowski and Łukasz Kuszner

Gdańsk University of Technology, Poland
Department of Algorithms and System Modeling
{kosowski,kuszner}@sphere.pl

Abstract. In the self-stabilizing model each node has only local information about the system. Regardless of the initial state, the system must achieve a desirable global state. We discuss the construction of a solution to the spanning tree problem in this model. To our knowledge we give the first self-stabilizing algorithm working in a polynomial number of moves, without any fairness assumptions. Additionally we show that this approach can be applied under a distributed daemon. We briefly discuss implementation aspects of the proposed algorithm and its application in broadcast routing and in distributed computing.

1 Introduction

A distributed system consists of nodes that are pairwise connected by communication channels. Each node maintains variables which determine its *local state*. The *global state* of the system is the union of all local states. Such a model is seen to be a good abstraction for real objects such as peer-to-peer networks.

The system is constructed in such a way as to guarantee that it works correctly, i.e. persists in a legitimate state, even though some perturbations can bring it to an illegitimate state. It is desirable that it returns to a legitimate state without any external intervention. Self-stabilization, a concept introduced by Dijkstra [4] in 1974, can be thought of as a technique for designing such resilient systems. A *self-stabilizing system* is one which is able to achieve a legitimate global state starting from any possible global state.

A distributed system can be modeled by a connected graph $G = (V, E)$, where vertex set V corresponds to system nodes and the set of edges E denotes communication links between them. A vertex can change its local state by making a *move*. The algorithm for each vertex v is given as a set of rules of the form **if** $p(v)$ **then** A, where $p(v)$ is a predicate over local states of v and its neighbors, and A is an action changing a local state of v (a move of v). A vertex v becomes *active* when $p(v)$ is true, otherwise v is *stable*. If all vertices in a graph are stable, we say that the system is stable. In order to measure the time complexity of self-stabilizing algorithms we often use the number of moves or, more commonly in synchronized systems, the number of rounds. All aspects of self-stabilization are described in detail in [5].

Finding a spanning tree in such a system can be the basis of many complex distributed protocols like routing, token circulation or code assignment. Predictably, many publications on the subject have appeared in recent years. A self-stabilizing algorithm for a BFS spanning tree in a semi-uniform system with a central daemon under read/write atomicity was described in [7]. Afek, Kutten and Yung in [1] gave a similar algorithm but for a uniform network, which stabilizes in $O(n^2)$ asynchronous rounds. Another algorithm was given by Arora and Gouda in [3] as a part of a more complex algorithm. In this paper authors assumed unique identifiers for vertices and a bound on the graph size known to all nodes. A very simple algorithm was presented by Huang and Chan in [9]. Yet another, which also needs a bound on n known to all vertices, was invented by Sur and Srimani in [11].

Later on, many other papers appeared on the subject, describing various approaches, considering time effectiveness [2], memory requirements (see e.g. [6, 10]) or communication costs. But, to the best of our knowledge, no algorithm working in a polynomial number of moves without any assumptions on scheduler fairness has ever before been described in literature. As a matter of fact, we generalize this model even further, by allowing the scheduler to select an arbitrary, non-empty subset of active nodes which then all perform a move in parallel (this is known as a single *action*). The problem was open even for the case when the scheduler selects exactly one of the active nodes at a time to make a move.

2 An algorithm for finding a spanning tree in $O(n\,\mathrm{diam}(G))$ moves

Let $G = (V, E)$ be a system graph, where vertex set V corresponds to system nodes and the set of edges E denotes communication links between them. By $n = |V|$ and $m = |E|$ we denote the number of vertices and the number of edges, respectively. In addition, let $N(v) = \{u : (u, v) \in E\}$ be the *open neighborhood* of v, and let $\deg(v) = |N(v)|$ be the *degree* of v. A spanning tree $T = (V, E')$ of $G = (V, E)$ is a subgraph of G consisting of the same set of nodes V, but only a subset $E' \subseteq E$ of edges such that there exists exactly one path between every pair of nodes in T. To ensure the existence of a spanning tree, graph G must be connected, so in this paper we restrict our considerations to connected graphs only.

In our first approach we provide a simple semi-uniform algorithm, in which each node has only one local variable f, a non-negative integer. Semi-uniformity means that exactly one of the nodes, called a *root*, needs to be distinguished. We will denote it by r. The interpretation of state variable f is as follows. Consider node v and let us choose u such that $f(u) = \min_{w \in N(v)} f(w)$ and u is the first[1] vertex among the neighbors of v with such a property. If $f(u) < f(v)$ then we say that u is the *parent* of v. For the distinguished vertex $f(r)$ is permanently equal to 0.

We now proceed to show an algorithm which will guarantee that the parent of each vertex (with the exception of the root) is uniquely defined, thus implying an ordering of the vertices equivalent to a spanning tree.

[1] To be able to say "first" we must assume that the neighbors are somehow ordered. This is not a strong assumption as long as a node is able to distinguish between its neighbors. For example, if the neighbors of v are stored in the form of a list, the order can be given according to the list sequence.

Algorithm 1:

R: **if** $v \neq r \wedge f(v) \leq \min_{u \in N(v)} f(u)$
then $f(v) = \max_{u \in N(v)} f(u) + 1$

Let us consider an arbitrary spanning tree $T = (V, E_T)$ of G. Let us consider an edge $e = \{u, v\} \in E_T$, where vertex u is closer to root r in tree T than vertex v, $d_T(u,r) < d_T(v,r)$ ($d_T(u,v)$ is the length of path $[u,v]_T$). We will call edge e *correctly directed* if $f(u) < f(v)$. When analyzing Algorithm 1 it is useful to bear in mind the following observation.

Corollary 1 *If for a given state of the system running Algorithm 1 there exists a spanning tree T such that all its edges are correctly directed, then the system is stable.*

Theorem 2 *Algorithm 1 stabilizes in $O(n \operatorname{diam}(G))$ moves.*

Proof. Let us consider an arbitrary spanning tree T of G and the following function:

$$S_T(v) = \sum_{(w,w') \in E([r,v]_T)} \max\{f(w) - f(w'), 0\} \tag{1}$$

Intuitively, $S_T(v)$ can be thought of as the number of edges lying on the path $[r,v]_T$ which are correctly directed. Obviously:

$$0 \leq S_T(v) \leq d_T(r,v) \tag{2}$$

Let us consider the effect of a parallel action of a set of vertices $X = \{x_1, \ldots, x_k\} \subset V$ on the values $S_T(v)$. As the root r cannot make any move, thus $r \notin X$. Without loss of generality we can assume that the subgraph H of G induced by X is connected, since otherwise the same actions can be performed by the scheduler in several successive actions, without any time gain. The structure of Algorithm 1 implies that before the action, for all i we have $f(x_i) \leq \min_{u \in N(x)} f(u)$, and consequently $f(x_i) = f(x_j)$ for all i, j. Consider an arbitrary connected component P of the forest $T \cap H$. The change of values of f within component P affects which edges of T are correctly directed. Notice that the action does not affect the edges of $T \setminus P$. Let e_P be the edge connecting P with the component of $T \setminus P$ containing root r. Before the action, no edge of P was correctly directed, and edge e_P was not correctly directed either. After the action, edge e_P is correctly directed. Taking into account the fact that for any vertex $v \in V$ the value $S_T(v)$ depends only on which edges of the path $[r,v]_T$ are correctly directed, we immediately obtain that the value $S_T(x)$ increases by at least 1 for every vertex $x \in P$, and does not decrease for any other vertex of T. Since this reasoning can be repeated for all other connected components of $T \cap H$, it is evident that the value $S_T(x)$ increases by at least 1 for every vertex which is making a move, and does not decrease for any other vertex.

Thus, by inequality (2) and Corollary 1 we obtain that a vertex v may make at most $d_T(r,v)$ moves while the algorithm is stabilizing. Since the above reasoning holds for any tree T, it is also correct for the BFS spanning tree T_B of graph G rooted at vertex r. The following inequality is true for any vertex v: $d_{T_B}(r,v) \leq \operatorname{diam}(G)$. All vertices become stable after making at most $\operatorname{diam}(G)$ moves each, which completes the proof.

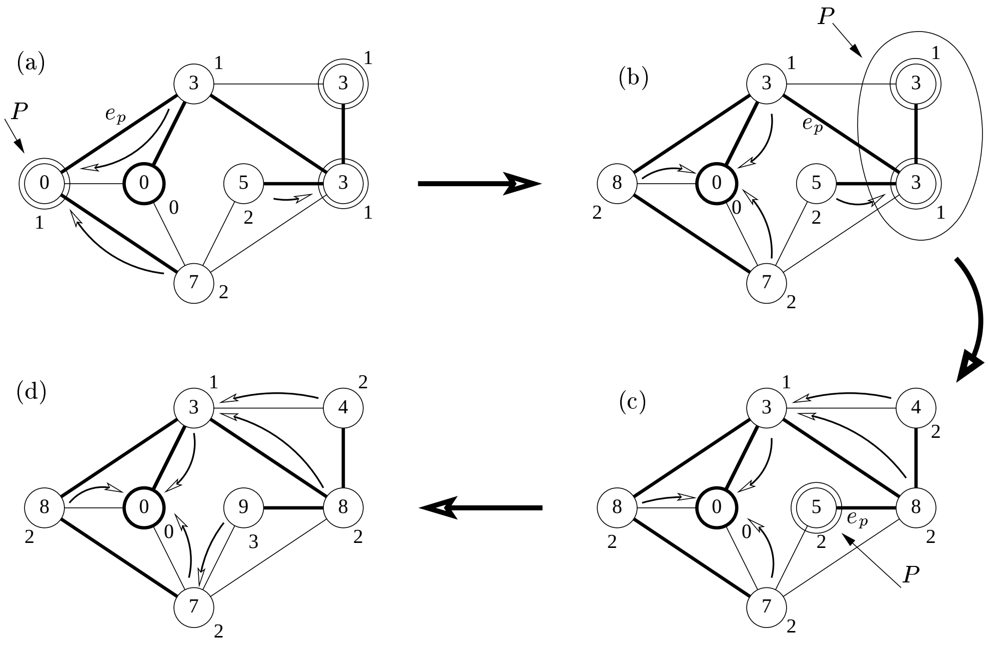

Fig. 1. Illustration of a process of Algorithm 1 (consult Subsection 2.1)

Having in mind Theorem 2 we know that the system stabilizes. Now, we must consider the result of the process.

Theorem 3 *Algorithm 1 finds a spanning tree.*

Proof. Suppose that the system is stable, so each node is stable. Hence according to our definition of the parenthood relation every vertex except root r has a unique parent node (otherwise, such a vertex would have a locally minimal value and rule R would be active for it). By including all edges $\{u, v\}$ such that v is a parent of u we obtain graph T, a subgraph of $G = (V, E)$. It is easy to observe that r is a vertex of T, thus T has n vertices and $n - 1$ edges. Moreover, by definition of parenthood, T may not contain any cycles, so T is a spanning tree of G.

The algorithm given above is extremely simple and fast. However, at this point we can observe certain inconveniences. First, we cannot give an upper bound on the value of $f(v)$ and secondly, we do not provide sufficient local state information to allow a vertex to recognize its child nodes in the tree (only its parent). The former problem is addressed in detail in Section 4. The latter may be easily solved by adding a simple rule to Algorithm 1, which has no effect on its polynomial pessimistic stabilization time.

2.1 Example

In Figure 1 each of the pictures illustrates the states of the vertices in successive moves. The root vertex r is distinguished by a bold circle and active vertices are marked with double circles. The state value f of each vertex is given inside the circle, while the value S_T, used in the proof of theorem 2, is marked close to the circle. An exemplary chosen spanning tree T is denoted by bold lines, while additional arcs show the parenthood relation between vertices. A small pointer indicates the vertices to perform the next action. Symbols P and e_p have the same meaning as in the proof. The algorithm computes a spanning tree in three actions, and the result is visible (in the form of the parenthood relation) in Figure 1(d).

3 Fault containment for limited perturbations

In practical applications of spanning tree construction it is often desirable for the stabilization time of the algorithm to be dependent on the severity of the faults which appeared in the system, i.e. a minor perturbation ought to result in quicker stabilization. This notion is referred to as *fault-containing* spanning tree construction, and was studied in the round-based model by Ghosh, Gupta and Pemmaraju [8].

Algorithm 1 proves extremely stable when considered from the point of view of fault containment. We say that the system starts in a *k-faulty state* if the initial state can be transformed into a state representing some valid spanning tree of G (rooted at r) by changing local vertex states of not more than k vertices.

First, we observe that if the initial state of the system corresponds to some correct spanning tree, Algorithm 1 will not make any actions.

Corollary 4 *If the system starts in a 0-faulty state, then Algorithm 1 stabilizes without making any actions.*

The algorithm also stabilizes quickly for perturbations on a limited number of vertices.

Property 5 *If the system starts in a k-faulty state, then Algorithm 1 stabilizes in $O(kn)$ moves.*

Proof. The proof is based on a similar method as that applied in the proof of Theorem 2. Suppose that the system starts in a k-faulty state with tree T used as the reference solution. It suffices to notice that the introduction of a single fault results in a change of values $S_T(v)$ by no more than 1 for all vertices. Consequently, in a k-faulty state the initial value of $S_T(v)$ is never smaller than the final value of $S_T(v)$ by more than k. Since every move of a vertex increases its current value $S_T(v)$ by at least 1, no vertex will ever move more than k times while Algorithm 1 is stabilizing. Of course, the solution obtained by the algorithm need not be the same as the reference tree T.

Finally, it is interesting to observe the way in which Algorithm 1 builds its spanning tree. At every stage of execution, each vertex of the graph is capable of indicating its direct parent or stating that in the current arrangement it has no parent (at the end of the process the only vertex left without a parent is the root r). This feature is of some importance in networking applications, since it guarantees that at every stage of execution the temporary solution is a directed spanning forest, and is always acyclic.

4 Memory usage for local states

The local state of Algorithm 1 consists of one local variable f and one implied pointer, indicating the parent of the vertex. The values $f(v)$ are non-negative in the initial state and increase throughout the operation of the algorithm. It is easy to show that if their initial value is polynomial with respect to n, then their final value is also polynomial with respect to n. Indeed, let F_t denote the maximum of $f(v)$ taken over all vertices, for the state of the system after exactly t actions. Initially, $F_t = F_0$ and with every action F_t increases by at most 1. Since by Theorem 2 the algorithm stabilizes after f actions, for some $f \leq n \operatorname{diam}(G)$, the final value F_f fulfills the inequality $F_f \leq F_0 + n^2$.

However, in practical applications it is often useful to give up the classical model of a local state understood as a memory cell of dynamically expandable size, and assume that the size of the local state is limited by physical storage constraints. In order to achieve this, we propose the following modification of Algorithm 1, in which the local state variable $f(v)$ is understood as a memory cell capable of storing any integer from the range $[0, N]$, for some known constant value $N \geq n$. The only exception is the local state of the root, as the value $f(r)$ is always fixed and equal to 0 (as in the case of Algorithm 1).

Algorithm 2:

R: **if** $v \neq r \wedge f(v) \leq \min_{u \in N(v)} f(u) \wedge \max_{u \in N(v)} f(u) < N$
then $f(v) = \max_{u \in N(v)} f(u) + 1$

R1: **if** $v \neq r \wedge f(v) > \max_{u \in N(v)\,:\, f(u) < f(v)} (f(u) + 1)$
then $f(v) = \max_{u \in N(v)\,:\, f(u) < f(v)} (f(u) + 1)$

Theorem 6 *Algorithm 2 stabilizes in $O(Nn \operatorname{diam}(G))$ moves and requires $O(\log N)$ storage space per vertex.*

Proof. The $O(\log N)$ storage space required by the algorithm is an obvious consequence of the structure of the local state. We will concentrate on proving the bound on the number of moves required by the algorithm.

Rule *R* of Algorithm 2 is a copy of rule *R* of Algorithm 1, with the constraint that the rule will only activate provided the resultant value $f(v)$ does not exceed the imposed upper bound N. Rule *R1* reduces the value $f(v)$ for the active vertex v to the smallest possible value which does not make any correctly directed edge in graph G (with respect to any spanning tree) lose its correct direction (consult the proof of Theorem 2), even if such a rule is executed in parallel with rule *R* on other vertices.

It suffices to show that before the algorithm stabilizes rule *R* activates in at most $n \operatorname{diam}(G)$ moves, and rule *R1* activates in at most $O(Nn \operatorname{diam}(G))$ moves. The first part of the statement is true by Theorem 2, since the bound on the number of activations of rule *R* is only dependent on the correct directions of edges in some spanning tree (proof of Theorem 2), and the number of such moves made by a single vertex is bounded by $\operatorname{diam}(G)$. To prove the second part of the statement, we consider the sum Σ_t of values $f(v)$ taken over all vertices, in the state of the system after exactly t actions. The change of value Σ_t from Σ_0 in the first considered state to Σ_f in the last considered state is the result of the total change $\Delta\Sigma_{\mathtt{R}}$ caused by moves using rule

R and the total change $\Delta\Sigma_{\mathtt{R1}}$ caused by activations of rule *R1*, i.e.:

$$\Sigma_f = \Sigma_0 + \Delta\Sigma_{\mathtt{R}} + \Delta\Sigma_{\mathtt{R1}} \tag{3}$$

Since at every stage of the algorithm all values $f(v)$ are bounded ($0 \leq f(v) \leq N$), we have $0 \leq \Sigma_0 \leq Nn$ and $0 \leq \Sigma_f \leq Nn$. A single move using rule *R* increases Σ_t by not more than N, and the number of activations of rule *R* is bounded by $n\,\mathrm{diam}(G)$, hence $0 \leq \Delta\Sigma_{\mathtt{R}} \leq Nn\,\mathrm{diam}(G)$. From (3) and the above observations we have:

$$|\Delta\Sigma_{\mathtt{R1}}| \leq |\Sigma_f| + |\Sigma_0| + |\Delta\Sigma_{\mathtt{R}}| \leq Nn(\mathrm{diam}(G) + 2) \in O(Nn\,\mathrm{diam}(G)) \tag{4}$$

By studying rule *R1* it is easy to observe that any activation of this rule strictly decreases the value Σ, and consequently the number of activations of this rule does not exceed $|\Delta\Sigma_{\mathtt{R1}}|$. By applying (4) we obtain the desired bound on the number of moves.

Theorem 7 *Algorithm 2 finds a spanning tree.*

Proof. By studying the proof of Theorem 3, we observe that rule *R* is inactive for all vertices in one of two cases: (1) the current state is already stable, or (2) there is a vertex v with all edges directed towards it which has a neighbor u such that $f(u) = N$. It suffices to prove that the latter case is not a final state of Algorithm 2. The proof proceeds by contradiction. Suppose that rule *R1* is also inactive for all vertices. This means that for any vertex w we either have $f(w) = 0$, or there exists a neighbor x of w such that $f(w) = f(x) + 1$. Consequently, for any vertex w the inequality $f(w) < n$ holds, a contradiction with $f(u) = N \geq n$.

It is interesting to study the effect of the chosen bound N on the performance of Algorithm 2. If N is polynomial with respect to n, $N \in O(\mathrm{poly}(n))$, then the memory required for storing a local state is $O(\log n)$, and the number of moves made by Algorithm 2 is $O(\mathrm{poly}(n))$. In particular, if N is a constant-factor bound on the value of n, i.e. $N \in O(n)$, then the number of moves of Algorithm 2 may be written as $O(n^2\,\mathrm{diam}(G))$. On the other hand, it has to be remembered that the value N needs to be stored in all vertices, and consequently selecting a value too close to n may decrease the scalability of the system.

5 Final remarks

When considering the model of self-stabilization without fairness guarantees, the spanning tree algorithm presented in this paper has numerous and profound implications.

First of all, the presented algorithm may be easily and efficiently applied in a routing protocol for a distributed network. In such a network, broadcast packets may only be routed along edges of some spanning tree (to avoid infinite transmission loops) and the spanning tree may need to be dynamically recreated in case of temporary malfunction without the intervention of a central agent. A spanning tree suitable for such a protocol can be constructed using an algorithm based on Algorithm 2 with one additional rule (allowing a vertex to know not only its parent, but also its children), whose polynomial stabilization time is immediately evident.

Moreover, the existence of the discussed algorithm shows that it is possible to use an algorithm stabilizing in a polynomial number of moves to determine a structure defined by a global property in the graph, but using only local neighborhood information. Our algorithm immediately implies the existence of an algorithm stabilizing in polynomial time for decomposing a bipartite graph with a distinguished root vertex into two independent sets (a problem posed in [12]). Such an approach opens up new possibilities for polynomial-time self-stabilizing distributed computing without fairness assumptions.

References

1. Y. Afek, S. Kutten and M. Yung, *Memory efficient self-stabilizing protocols for general networks*, Proceedings of the 4th International Workshop on Distributed Algorithms, 1991, 15–28.
2. S. Aggarwal and S. Kutten, *Time optimal self-stabilizing spanning tree algorithms*, LNCS **761** (1993), 400–410.
3. A. Arora and M. Gouda, *Distributed reset*, IEEE Transactions on Computers, **43** (1994), 1026 – 1038.
4. E. W. Dijkstra, *Self-stabilizing systems in spite of distributed control*, Communication of the ACM **17** (1974), 643–644.
5. S. Dolev, *Self-stabilization*, MIT Press, 2000.
6. S. Dolev, M. Gouda and Marco Schneider, *Memory requirements for silent stabilization.* Acta Informatica **36** (1999), 447–462
7. S. Dolev, A. Israeli and S. Moran, *Self-stabilization of dynamic system assuming only read/write atomicity*, Distributed Computing **7** (1993), 3–16.
8. S. Ghosh, A. Gupta, and S. Pemmaraju, *A fault-containing self-stabilizing algorithm for spanning trees*, Journal of Computing and Information **2** (1996), 322-338.
9. S. Huang and N. Chen, *A self-stabilizing algorithm for constructing breadth-first trees*, Inform. Process. Lett. **41** (1992), 109–117.
10. C. Johnen *Memory Efficient, Self-Stabilizing algorithm to construct BFS spanning trees*, Proc. of the third Workshop on Self-Stabilizing System (1997), 125–140,
11. S. Sur and P. K. Srimani, *A self-stabilizing distributed algorithm to construct BFS spanning trees of a symmetric graph*, Parallel Process. Lett. **2** (1992), 171–179.
12. S. Sur and P. K. Srimani, *A self-stabilizing algorithm for coloring bipartite graphs*, Inform. Sci. **69** (1993), 219–227.