Efficient Multi-Stage Self-Stabilizing Algorithms for Tree Networks

Jean R. S. Blair[*] Fredrik Manne[†]

Abstract

In [1] a new efficient self-stabilizing algorithm for leader election in trees was proposed and shown to have a moves complexity of $O(n^2)$. In the current work we combine two such algorithms where the variables of one of the algorithms drives the predicates of the other. In this way we obtain a general framework for developing self-stabilizing algorithms on trees where some kind of feedback mechanism is needed. Contrary to what one might expect we show that the moves complexity of this new algorithm is not the multiplicative moves complexity of the two algorithms, but in fact $O(n^3)$.

A number of previously published self-stabilizing algorithms are shown to fit into our framework and therefore obtain the improved bound not only in terms of reducing the number of moves but also in that the required model is less restrictive than the earlier published algorithms.

We further show that our approach can be generalized to design k-stage self-stabilization algorithms on trees giving a combined moves complexity of $O(n^{k+1})$.

1 Introduction

Two self-stabilizing algorithms A and B can be combined so that local stabilized values resulting from the execution of algorithm A drive the predicates of algorithm B. In this way one can build up more complex algorithms. One example of such a 2-stage self-stabilizing algorithm could be to combine a self-stabilizing algorithm for finding a spanning tree in a general network with a self-stabilizing algorithms that operates on the resulting spanning tree.

In the current paper we investigate the possibility of converting sequential tree algorithms that require *more* than a single bottom-up pass to solve a problem into multi-stage self-stabilizing algorithms. In particular, we focus on algorithms that first require that a node (or pair of nodes) with a certain property be designated as the leader in a tree network and then pass information about that leader back into the network before subsequent stages in which the particular problem is solved.

Several such self-stabilizing algorithms have been proposed previously in the literature. Examples include finding a 2-coloring of a tree network [10], solving various problems that are applicable to dynamic programming such as maximum weighted independent set [3], and finding a 2-center in a tree [8].

[*]Department of Electrical Engineering and Computer Science, United States Military Academy, West Point, New York 10996, USA, `Jean-Blair@usma.edu`

[†]Department of Informatics, University of Bergen, N-5020 Bergen, Norway, `Fredrik.Manne@ii.uib.no`

Although none of these algorithms contain a detailed analysis of their moves complexity we note that it is fairly straight forward to show that they could use $\Omega(n^5)$ moves. This is partly because they rely on an $O(n^3)$ algorithm for rooting the tree. In addition some of these algorithms demand the existence of a fair daemon where no eligible rule will wait indefinitely long before it is applied.

In the current paper we combine two instances of the leader election algorithm from [1] to develop a general framework that can be shown to solve all of the above mentioned problems. Since the moves complexity of the leader election algorithm is $O(n^2)$ one might expect that the moves complexity of this combined algorithm will be their multiplicative complexity of $O(n^4)$. However, we show that it is in fact $O(n^3)$.

Although it might seem intuitive for general algorithms A and B with moves complexity $m(A)$ and $m(B)$ respectively, that their combined moves complexity (where A drives B) should be $O(m(A)m(B))$ (i.e. B must stabilize after each A move) we argue that it is in fact $O(m(A)m(B)\Delta)$ where Δ is the maximum number of neighbors of any node in the underlaying graph. This should again be contrasted to the presented moves complexity of $O(n^3)$ moves which is an order of magnitude less than the multiplicative moves complexity.

We further show that our approach can be generalized to design k-stage self-stabilization algorithms on trees giving a combined moves complexity of $O(n^{k+1})$.

For the rest of this paper we assume that G is a tree with n nodes. If H is a subgraph of G then we let $G - H$ denote the induced subgraph found by removing H and all edges incident to H from G. The only exception to this is if (v, w) is an edge of G, then we write $G - (v, w)$ for the (disconnected) graph obtained by removing only the edge (v, w) (and not v and w) from G. When G is a tree and $(i, j) \in E$ we write $G_i(j)$ for the component of $G - \{i\}$ containing node j.

If i is a node, then $N(i)$ denotes the set of nodes to which i is adjacent. We assume that each node has a unique identifier.

We assume read-write atomicity, but make no assumptions about a central demon. Thus, two or more nodes can make simultaneous moves, as long as no node is accessing a value that is being written by another node.

In the next section we review the self-stabilizing algorithm that will be the building block of our multi-stage algorithm. In Section 3 we present and analyze our generic 2-stage algorithm. Various applications of this algorithm is presented in Section 4. In Section 5, we develop a generic k-stage algorithm before concluding in Section 6.

2 A 1-stage tree algorithm

In this section we look at a generic algorithm for tree-networks that will be our building block when we later design multistage algorithms. The algorithm was first presented in [1] where it was used to perform leader election in a tree network and also to implement various bottom-up type algorithms on trees to solve problems such as maximum independent set and maximum matching. All of these algorithms were shown to have a moves complexity of $\theta(n^2)$.

In the algorithm each node i has a separate value $f_i(j)$ for each neighbor j. Let further $g()$ be a function defined on values of f. The main rule is then as follows:

```
G1:   if   $(\exists j \in N(i)$ such that $f_i(j) \neq g(\cup_{k \in N(i)-\{j\}} f_k(i))$
```

then $\quad (f_i(j) \leftarrow g(\cup_{k\in N(i)-\{j\}} f_k(i)))$

Rule **G1** specifies that in order to set $f_i(j)$ the only input that will be used are values $f_k(i)$, $k \in N(i) - \{j\}$ while $f_i(j)$ again will only be used as input to compute values $f_j(r)$, $r \neq i$. This is as illustrated in Figure 1.

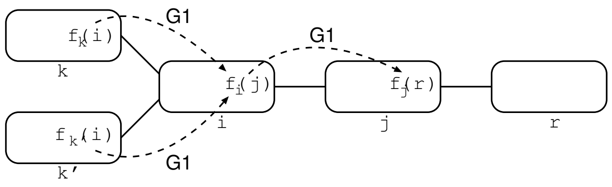

Figure 1: The input to **G1** moves

In the following we will give an alternative proof of the moves complexity of **G1** than the one given in [1]. The reason for this is that we will need some of the intermediate results when we analyze what happens when we combine several such algorithms. Also, this gives a good review of how the algorithm works.

In [1] the moves complexity was derived by showing that the values of a particular pair $f_i(j)$ and $f_j(i)$ can change at most n times during the execution of **G1**. The result then follows since there are $n-1$ such pairs. The proof we present here will instead show how many subsequent moves an initial value of $f_i(j)$ can lead to on other nodes. To do this we first need a formal definition of what is meant by “a value leads to a move”. We introduce the notation $G_i^T(j)$ to denote the application of **G1** on the variable $f_i(j)$ at time T. We also denote the value of the variable $f_i(j)$ at time T by $f_i^T(j)$. Note that $T = 0$ indicates the starting time of the algorithm. The $g()$ function on a leaf does not depend on any other f-value. To make the analysis simpler we assign a dummy value $f_i(i)$ to each leaf i. This value will be used as input to $g()$ on i but will neither alter its value nor will it be updated by **G1**. Our definition then becomes:

Definition 2.1 *Let $G_{i_1}^{T_1}(i_2), G_{i_2}^{T_2}(i_3), \ldots, G_{i_r}^{T_r}(i_{r+1})$ be a sequence of moves such that for each pair of adjacent moves $G_{i_k}^{T_k}(i_{k+1}), G_{i_{k+1}}^{T_{k+1}}(i_{k+2})$ the following holds:*

1. $T_k < T_{k+1}$
2. $i_k \neq i_{k+2}$
3. *There is no move $G_{i_{k+1}}^{T'}(i_{k+2})$ such that $T_k < T' < T_{k+1}$*

Then if $f_{i_0}^{T_1}(i_1)$ is a variable that is used as input to $G_{i_1}^{T_1}(i_2)$ we write $f_{i_0}^{T_1}(i_1) \rightsquigarrow G_{i_r}^{T_r}(j_j)$.

What Definition 2.1 says is that there exists a chain of moves starting with the variable $f_{i_0}^{T_1}(i_1)$ as input and (possibly) ending with the move $G_{i_r}^{T_r}(i_{r+1})$, where each move uses the outcome of the previous move as input. Note that it is not necessarily the value $f_{i_k}(i_{k+1})$ set in move $G_{i_k}^{T_k}(i_{k+1})$ that triggers the next move $G_{i_{k+1}}^{T_{k+1}}(i_{k+2})$. It is only required that $G_{i_{k+1}}^{T_{k+1}}(i_{k+2})$ is the first subsequent move that makes use of this value as input when altering the value of $f_{i_{k+1}}(i_{k+2})$.

We now proceed to bound the number of moves that a particular value $f_i^T(j)$ can lead to.

Lemma 2.2 *Let i and j be neighboring nodes in G and let $size_i(j)$ be the number of nodes in $G_i(j)$. Then for a particular value T there can at most be $size_i(j) - 1$ moves $G_{i_r}^{T_r}(j_r)$ such that $f_i^T(j) \rightsquigarrow G_{i_r}^{T_r}(j_r)$ is true.*

Proof. Note first that from Definition 2.1 a sequence of moves that causes $f_i^T(j) \rightsquigarrow G_{i_r}^{T_r}(j_r)$ to be true is unique and cannot contain moves performed on the same node more than once. Thus it is sufficient to show that the number of values in $G_i(j)$ that could be changed due to $f_i^T(j)$ is bounded by $size_i(j) - 1$.

The rest of the proof is by induction on $size_i(j)$. If $size_i(j) = 1$ then j is a leaf and will not use $f_i^T(j)$ to update any values. Assume therefore that the result is true for all neighboring nodes k and l where $size_k(l) < p$ and that $size_i(j) = p > 1$. Then the node j will have $N(j) > 1$ neighbors. For each $k \in N(j) - \{i\}$ there can at most be one application of **G1** to update $f_j(k)$ at some time $T_k > T$. Since $size_j(k) < p$ it follows that each such updated value $f_j^{T_k}(k)$ will again cause at most $size_j(k)$ subsequent moves. Thus the total number of moves caused by $f_i^T(j)$ is bounded by $|N(j)| - 1 + \sum_{k \in N(j)-\{i\}}(size_j(k) - 1) = \sum_{k \in N(j)-\{i\}} size_j(k) = size_i(j) - 1$ and the result follows. ■

What Lemma 2.2 says is that each value $f_i^T(j)$ can at most cause a "ripple" effect that will reach each non-leaf node in $G_i(j)$ at most once. This gives us the following corollary.

Corollary 2.3 *Let i be a node at a given time T. Then the total number of moves such that $f_i^T(j) \rightsquigarrow G_k^{T'}(l)$ summed over all $j \in N(i)$ is at most $n - |N(i)| - 1$.*

Proof. From Lemma 2.2 we know that for a particular value of j there can at most be $size_i(j) - 1$ moves such that $f_i^T(j) \rightsquigarrow G_k^{T'}(l)$. Thus summing over all values of $j \in N(i)$ we get the desired result $\sum_{j \in N(i)}(size_i(j) - 1) = n - |N(i)| - 1$. ■

To be able to bound the total number of moves we now show that every move originates from some initial f-value.

Lemma 2.4 *For any move $G_i^T(j)$ there exists some value $f_k^0(l)$ such that $f_k^0(l) \rightsquigarrow G_i^T(j)$ is true.*

Proof. Let $f_q^T(i)$ be a value that is used as input to $G_i^T(j)$. If $f_q^T(i) = f_q^0(i)$ then we are done. If not, starting with $f_q^T(i)$, repeatedly select a new f-value that was used as input to the move that set the current f-value. Since the fact that the underlying graph is a tree this process will never enter the same node twice and must therefore terminate with some value $f_k^{T'}(l)$ (possibly with $k = l$) which has not been set by any **G1** move. Thus we must have $T' = 0$ and the result follows. ■

Theorem 2.5 *The moves complexity of the* **G1** *algorithm is $O(n^2)$.*

Proof. By Lemma 2.4 it is sufficient to sum the number of moves such that $f_i^0(l) \rightsquigarrow G_i(j)$ over all values of i and l. From Corollary 2.3 it follows that if $i \neq l$ then for any specific node this sum is at most $n-1$. Since there are n nodes this gives at most $n(n-1)$ moves. On a leaf node $f_i(i)$ might contribute another n moves. But since there are at most n leafs in G the upper bound still remains at $O(n^2)$. ■

It is shown in [1] that the bound from Theorem 2.5 is also a lower bound thus the moves complexity of algorithm **G1** is $\Theta(n^2)$.

In the next section we will use these results to look at what happens when the $g()$ function not only depends on f-values, but also on input from another self-stabilizing algorithm.

3 Combining two algorithms

We will in this section describe and analyze what happens when we combine two rules as described in Section 2 such that one of the rules uses values from the other rule as part of its input. In particular we will show that our proposed combined algorithm has a moves complexity of $O(n^3)$. The purpose of this new rule is to be able to perform some calculation on the graph G that depends on the stabilized values of **G1**. We will show several such examples in Section 4.

We consider a rule **H1** similar to **G1** that sets a local value $q_i(j)$ in the same way as **G1** sets $f_i(j)$. The only major change from **G1** is that we have replaced the $g()$ function with an $h()$ function that in addition to the appropriate q-values also takes $f_i(j)$ and $f_j(i)$ as input for all $j \in N(i)$.

```
H1:  if    $(\exists j \in N(i)$ such that $q_i(j) \neq h(\cup_{k \in N(i)-\{j\}} q_k(i), \cup_{k \in N(i)}(f_k(i), f_i(k)))$
     then  $(q_i(j) \leftarrow h(\cup_{k \in N(i)-\{j\}} q_k(i), \cup_{k \in N(i)}(f_k(i), f_i(k)))$
```

Figure 2 shows the values that are used as input to **H1** when altering the value of $q_i(j)$. Note in particular that the structure of which q-values are used is the same as for the f-values that are used by **G1**. Moreover, the f-values are only used as input and are never altered by **H1**.

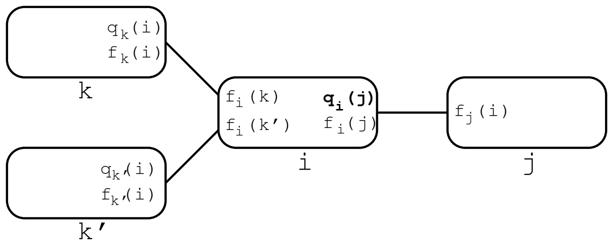

Figure 2: The input to **H1** moves

The formulation of **H1** is as general as possible and we will see more specific examples of how it can be instantiated in the following sections. We now proceed to analyze the combined moves complexity of **G1** and **H1**. First we show that Lemma 2.2 also applies to rule **H1**.

Lemma 3.1 *Let i and j be neighboring nodes in G and let $size_i(j)$ be the number of nodes in $G_i(j)$. Then for a particular value T there can at most be $size_i(j) - 1$ moves $H_{i_r}^{T_r}(j_r)$ such that $q_i^T(j) \rightsquigarrow H_{i_r}^{T_r}(j_r)$ is true.*

Proof. The key observation to realizing that this is true is that f-values that are altered do not influence the $\rightsquigarrow$ relation for **H1** moves.

In particular if $q_i^T(j) \rightsquigarrow H_j^T(k)$ and subsequently some f-value changes such that $q_j(k)$ must be recomputed with a move $H_j^{T'}(k)$, $T < T'$, then $q_i^T(j) \rightsquigarrow H_j^{T'}(k)$ is not true since $q_i^T(j)$ has already been used to set the value of $q_j(k)$. Once this is clear the proof is identical to the proof of Lemma 2.2. ∎

Lemma 3.1 states that independent of where a value $q_i^T(j)$ originates from it can at most cause $size_i(j)$ subsequent moves. We cannot prove a similar result as Lemma 2.4 for **H1** stating that every **H1** move must originate from some initial q-value since an **H1** move can be triggered both by a changing q-value and also by a changing f-value. Thus for each move $H_i^{T_r}(j)$ either there exists some q such that $q_i^T(j) \rightsquigarrow H_i^{T_r}(j)$ is true or it is a direct consequence of some $f_i(j)$ or $f_j(i)$ value. We can now give the desired bound for the number of moves caused by the combined **G1** $-$ **H1** algorithm.

Lemma 3.2 *The algorithm* **G1** $-$ **H1** *can at most make $O(n^3)$ moves.*

Proof. We first note that by Theorem 2.5 there can at most be $O(n^2)$ **G1** moves. Thus it remains to count the number of **H1** moves.

Initially we consider all $H1$ moves $H_k^{T_r}(l)$ such that $q_i^0(j) \rightsquigarrow H_k^{T_r}(l)$ is true for some i and j. That is, the moves that can be traced back to some initial q value. Since **H1** follows the same structure as **G1** there can at most be $O(n^2)$ such moves.

Finally, we are left with the **H1** moves that can only be traced back to q-values that were altered due to changing f-values. To bound these we consider a particular node i. If some $f_i(j)$ changes then this might lead to $|N(i)|$ **H1** moves on node i and also to $|N(j)|$ **H1** moves on node j. The q-values that are changed on node i can from Lemma 3.1 again at most cause at most $n-1$ new **H1** moves throughout the graph. The same is true for the q-values on node j that were changed due to $f_i(j)$. Thus it follows that any change of an $f_i(j)$ value will at most lead to $2n$ new **H1** moves. From Theorem 2.5 we know that the total number of times any f-value changes is $O(n^2)$. This gives the desired bound of $O(n^3)$ **H1** moves. ∎

4 Applications

We will in this section give three examples of algorithms or types of algorithms that fit the framework we developed in Sections 2 and 3 and which therefore achieve a moves complexity of $O(n^3)$. All of these algorithms are based on using algorithm **G1** to root the tree while algorithm **H1** computes some value based on knowledge of in which direction the root is.

For ease of notation and to make the presentation clearer we will subdivide the computation of $h()$ using separate functions that compute various properties based on the input parameters to $h()$. Most important among these functions is the computation of a *leader*. Since this is a common task for all of the resented algorithms we present this first.

4.1 Leader election

As discussed in [1] the leader election algorithm can be implemented with several different criteria for determining the leader. In the version presented here a 1-center node is elected as the leader. This is a node such that the maximum distance to a leaf is as small as possible. As shown in [6] a tree has either one unique 1-center node or two adjacent ones.

We set $g(\cup_{k \in N(i)-\{j\}} f_k(i)) = 1 + \max_{k \in N(i)-\{j\}} f_k(i)$. Then when algorithm **G1** stabilizes $f_i(j)$ will contain the maximum distance from node i to a leaf in $G_j(i)$. The node can now determine its maximum distance to a leaf as $\max\{f_i(j), f_j(i)+1\}$. By testing if $f_i(j) > f_j(i)$ for all $j \in N(i)$ a node can easily determine if it is a unique one center. If it is not a 1-center, then there will be exactly one neighbor j such that $f_i(j) < f_j(i)$ and the 1-center(s) must lie in $G_i(j)$ If there are two 1-centers then there will be exactly one neighbor j such that $f_i(j) = f_j(i)$ and i and j are the two 1-centers. In case we only need one root we can break a tie between two 1-centers by using the nodes unique id's (ID_i). For more details about the leader election algorithm we refer to [1].

In order to determine that the algorithm is stable from its local perspective node i uses a predicate $sizeCorrect_i$ defined as follows:

$$sizeCorrect_i = (\forall j \in N(i),\ f_i(j) = 1 + \max_{k \in N(i)-\{j\}} f_k(i))$$

The predicate $sizeCorrect_i$ simply states that **G1** cannot be executed on node i. The following function p_i can now be used to return a pointer to the neighbor of i that is closest to the root of G. If the algorithm is not locally stable it returns *null*.

```
Integer Function $p_i()$
    if $(\neg sizeCorrect_i)$
        return $null$;
    if $(\forall j \in N(i)\ f_i(j) > f_j(i))$
        return $i$;
    if $(\exists j \in N(i)$ such that $f_i(j) < f_j(i))$
        return $j$;
    Let $j \in N(i)$ be such that $f_i(j) = f_j(i))$
    if $(ID_i > ID_j)$
        return $i$;
        else return $j$;
```

Note that the function $\max_{k \in N(i)-\{j\}}$ is defined to have a value of 0 when i is a leaf, i.e., when $N(i) = \{j\}$.

If we do not need to break a tie between two 1-centers we can change the test for determining if a node is a 1-center to $f_i(j) \geq f_j(i))$. That is, as long as a node is one of the 1-centers in a tree, it will return a pointer to itself. Then when a node is not a 1-center there will always exist a neighbor j such that $f_i(j) < f_j(i))$ and we can omit the last if test.

The input values used by the function p_i are all contained among the values used by $h()$. Thus we can make use of p_i when defining new $h()$ functions.

4.2 Distance from the root and 2-coloring of trees

As a first simple example of how Algorithm **H1** can be instantiated to compute properties that rely on knowledge about the leader node, we show how to compute, at each node in the network, the distance from any node to the root node. This algorithm can also easily be modified to compute a 2-coloring on G.

The algorithm assumes that q_i is an integer value and uses the following h function:

```
Integer Function h()
    if ($p_i = i$)
        return 0;
    else
        return $q_{p_i} + 1$;
```

On the root node $h()$ will always return the value 0 and on all other nodes it will return one more than the q-value of its parent node.

Given that Algorithm **G1** is correct and has stabilized it is straight forward to show that with this $h()$ function Algorithm **H1** will stabilize with each q_i containing the distance from node i to the leader node. The moves complexity of $O(n^3)$ follows directly from Lemma 3.2.

If we wish to compute an optimal 2-coloring of a tree we can modify $h()$ by setting q_i to be a boolean variable and replace the return value 0 with *true* and $q_{p_i} + 1$ with $\neg q_{p_i}$. For this algorithm it would have been sufficient to only use one q_i value for each node. But as this does not influence the asymptotic moves complexity and in order to maintain consistency we choose to maintain one $q_i(j)$ value for each $j \in N(i)$.

We note that the information that is being sent back from the root to the rest of the graph could also be a fixed value giving for example the (unique) identity of the root or its spatial coordinates.

4.3 Dynamic Programming

Several graph-theoretic optimization problems have linear time dynamic programming solutions when restricted to trees. These include among others, maximum weighted matching, maximum weighted independent set, minimum weighted edge covering, and minimum weighted dominating set. All of these algorithms are based on performing a postorder traversal of the rooted tree and for each node computing a fixed number of values based on the values of its children. When the algorithm terminates, the size of the global solution will be known at the root node and a second phase starting from the root and moving down to the leaves can be used to find the actual solution.

Self-stabilizing algorithms for these types of problems were first presented in [3] where an algorithm was presented that computes a solution using $r + 1$ "rounds" where r is the radius of the graph and a round consists of a minimal sequence of moves such that every node that could possibly become eligible makes a move. This was later improved on in [1] where algorithms similar to **G1** where presented with a moves complexity of $O(n^2)$. It should be noted that these complexity results do not include the computation of the actual solution which, as described in [3], requires a second stage.

It is fairly straight forward to show that algorithm **H1** can be used to compute the actual solution thus resulting in a moves complexity of $O(n^3)$. As an example of this we show how the solution to the maximum weighted independent set problem on a tree can be obtained using **H1**. This algorithm is adapted from [3].

For ease of presentation we make the initial asumption that the graph is rooted. Let $w(i)$ be the integer weight of node i. Let further $f^+(i)$ be the weight of a maximum weight independent set that includes i of the subtree rooted at i, $f^-(i)$ be the weight of a maximum weight independent set that does not include i, and $f(i)$ be the maximum weight of an independent set of the subtree rooted at i regardless of weather i is in it or not. Then $f(i) = \max\{f^+(i), f^-(i)\}$ and $f^+(i)$ and $f^-(i)$ can be computed using the following formulas:

$$f^+(i) = \sum_{j \in N(i)-\{p_i\}} f^-(j) + w(i)$$

$$f^-(i) = \sum_{j \in N(i)-\{p_i\}} f(i)$$

If we let f be an array containing two values we can convert the computation of $f^+(i)$ and $f^-(i)$ into a $g()$ function for **G1** as follows:

```
Integer [2] Function $g()$
    $f_i^+(j) = \sum_{k \in N(i)-\{j\}} f_k^-(i) + w(i)$;
    $f_i^-(j) = \sum_{k \in N(i)-\{j\}} \max\{f_k^-(i), f_k^+(i)\}$;
    return $(f_i^+(j), f_i^-(j))$;
```

This function will compute $f_i^+(j)$ and $f_i^-(j)$ for every node i for every possible root except the case where node i itself is the root. To determine the root of the tree one can as described in [1] run two separate copies of **G1**, one containing the above $g()$ function to evaluate weights of different solutions depending on the placement of the root and one copy of **G1** with a $g()$ function to root the tree. These algorithms will not interact and will stabilize in $O(n^2)$ moves.

We now use the output of both of these algorithms as input to **H1** to determine which nodes should be in an optimal solution. Let q_i to be a boolean value which indicates if node i is in the maximum weight independent set, let p_i be based on the **G1** algorithm that roots the tree, and let the f-values be based on the above $g()$ function. Then we get the following $h()$ function.

```
Boolean Function $h()$
    if $(p_i = i)$
        $t_i^+ = \sum_{k \in N(i)} f_k^-(i) + w(i)$;
        $t_i^- = \sum_{k \in N(i)} \max\{f_k^-(i), f_k^+(i)\}$;
        return $t_i^+ > t_i^-$;
    return $(\neg q_{p_i}(i) \wedge f_i^+(p_i) > f^-(p_i))$;
```

The first *if* statement makes the root calculate f-values that includes every neighbor. If this results in that the solution with the root included is the heaviest then the root is in the solution otherwise not. Any other node can only be in the solution if its parent node is not in and its solution with itself included is larger than the one without itself. When the algorithms stabilizes the value of $q_i(p_i)$ will be true if and only if node i is in the maximum weight independent set. Using two separate **G1** rules will only change the proof of Lemma 3.2 so that the number of **H1** moves that can be traced back to changing values in the two **G1** will at most double. Thus the moves complexity of this algorithm is also $O(n^3)$.

4.4 2-centers in trees

A k-center of a graph G is a k-tuple of nodes $x_1, x_2, \ldots, x_k \in V$ such that the maximum minimum distance from any node in G to a node in the k-tuple is as small as possible. Thus a k-center minimizes the distance between any node in G and a node in the k-center. As a particular case note that a 1-center is a node where the maximum distance to a leaf is as small as possible. In [6] it is shown that a tree has either one 1-center or two 1-centers which are neighbors.

In [8] a self-stabilizing algorithm for computing a 2-center of a tree G was presented and proved correct. The algorithm relies on the following scheme to find a 2-center of G:

Algorithm 2C

1. Find the 1-center(s) of G
2. (a) If G has two 1-centers r_1 and r_2 then split G into two connected components G_1 and G_2 by removing the edge (r_1, r_2).
   (b) If G has exactly one 1-center r then there are at least two nodes a and b at maximum distance from r such that a and b are in different connected components C_a and C_b of $G - \{r\}$. Let u be the node in C_a such that $(r, u) \in E$. Create connected components $G_1 = C_a \cup (r', u)$ where r' is a clone of r and $G_2 = G - C_a$.
3. Find one 1-center in G_1 and one in G_2 and return these as a 2-center of G.

Figure 3 illustrates the two cases in Step 2 of the algorithm.

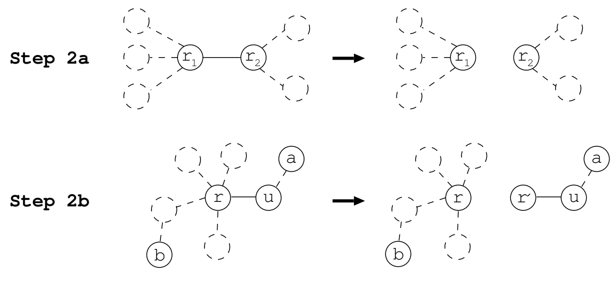

Figure 3: The two cases in Step 2 of Algorithm **2C**

Our algorithm progresses in the same interleaved three stages as Algorithm **2C**. In the following we present each stage separately.

As described in [1] algorithm **G1** can be used to find the 1-center(s) of G. We then sett the value of $f_i(j)$ to be the maximum distance from node i to a leaf in $G_i(j)$. This is achieved with the following g function:

```
**Integer Function** $g()$
    **return** $1 + \max_{k \in N(i)-\{j\}} f_k(i)$;
```

Note that when i is a leaf we define $g()$ to return 0, i.e. when $N(i) = \{j\}$. When the algorithm stabilizes the 1-center node(s) can be identified by the fact that they will have $f_i(j) \geq f_j(i)$ for all $j \in N(i)$ and if there exists a $j \in N(i)$ such that $f_i(j) < f_j(i)$ then the 1-center(s) must lie in $G_i(j)$.

The *sizeCorrect* predicate and the $p()$ function must also be updated appropriately to reflect that the $g()$ function now calculates the maximum distance to a leaf.

```
**Integer Function** $p_i()$
    **if** $(!sizeCorrect_i)$
        **return** $null$;
    **if** $(\forall j \in N(i)\ f_i(j) < f_j(i))$
        **return** $i$;
    **if** $(\exists j \in N(i)$ such that $f_i(j) < f_j(i))$
        **return** $j$;
```

The second stage of the algorithm should cut the graph into two components as specified in Step 2 of Algorithm 2C and then proceed to run a 1-center algorithm on each component returning one 1-center for each component. In order to point to where the graph should be cut we define a function cut_i to be used on the 1-center node(s).

```
**Integer Function** $cut_i()$
    **if** $(p_i = i)$
        **return** $\max_{k \in N(i)} f_k(i)$;
    **else**
        **return** $null$
```

With this definition it is clear that for a stabilized system $cut_i()$ will on the 1-center node(s) point to a neighbor j such that $G_i(j)$ contains a leaf at a maximum distance from i. If there are two 1-centers this implies that they will both set their *cut* values to point to the other 1-center.

Before we proceed we consider how the value of cut_i should be used on the 1-center(s) to achieve the effect of Step 2 and 3 of algorithm **2C**.

In Step 2a it is clear that G will be cut appropriately if each component of $G - \{r_1, r_2\}$ disregards the values of $q_{r_1}(cut_{r_2})$ and $q_{r_2}(cut_{r_1})$ and runs the 1-center algorithm again. In Step 2b this will also work for the component $G_u(r)$. To get the desired result on the component $G_r(u)$ node r must set $q_r(u) = 0$. This will give node u the impression that r is a leaf. Note that this will not effect the case when there are two 1-centers.

The following h function for calculating the values to determine the new 2-centers reflects these values of cut_i on the 1-center node(s).

```
**Integer Function** $h()$
    **if** $(j = cut_i)$
        **return** $0$;
    **return** $1 + \max_{k \in N(i) - \{j, cut_i\}} q_k(i)$;
```

We can define a parent pointer on each node to indicate the closest 2-center in a similar fashion to p_i the only difference is that on the 1-center node(s) we ignore the node pointed to by cut_i. The function that calculates this pointer must also break ties in the case that any component contains two 1-centers.

5 Multi-stage algorithms

In this section we consider how algorithm **G1-H1** presented in Sections 2 and 3 can be further extended into a general k-level algorithm where the input to the rule on level l, $l > 1$ can make use of the variables of level $l - 1$.

Such an algorithm could for instance be used to extend the 2-center algorithm of Section 4.4 into a "2^{k-1}-center" algorithm. If one instead had used an $\frac{n}{2}$-separator as the center node this algorithm could have been modified to compute the nodes on the first k levels of an elimination tree of height $\log n$ [9].

Before we consider how the rules should be designed in detail we look at the moves complexity of a general multi-stage self-stabilizing algorithm. Assume that we have algorithms A and B with moves complexity $M(A)$ and $M(B)$ respectively and that certain variables set in A are used as input to the rules of B. Then a variable a_i set by A on node i can at most be used as input to rules of B on node i and the neighbors of i. It is reasonable to assume that B will only use each a_i once on each node i and set all its dependent variables in one move. However, we have no knowledge of in which order the nodes will make use of a_i. Thus it could be that after a B move that uses a_i there could be as many as $M(B)$ moves before the next B move that reads a_i is made. Thus in the worst case B could make $(deg(i)+1)M(B)$ moves before the next A move, where $deg(i)$ is the number of neighbors of i in G. This shows that in general the best moves complexity we can give for the combined algorithm $A-B$ is $\Delta M(A)M(B)$ where $\Delta = \max_{i \in V} deg(i)$.

In spite of this rather pessimistic observation we note that we were able to show that the combined moves complexity of algorithm **G1-H1** was $O(n^3)$ even though a particular $f_i(j)$ or $f_j(i)$ value can be used up to $2deg(i)$ times in an application of **H1**. But looking more closely at the proof of Lemma 3.2 reveals that each f-value could again cause a total of $2(n-1)$ **H1** moves. Including the moves complexity of **G1** and of the **H1** moves that can be accounted to initial q-values a more precise bound on the moves complexity would then be $2n(n-1)^2 + 2n(n-1)$. If we were to recursively combine k such algorithms this would result in a moves complexity of $O(2^{k-1}n^{k+1})$. In the following we show how to develop an algorithm with a moves complexity of $O(n^{k+1})$.

From the above discussion it follows that in order to achieve this bound on the moves complexity each value of level l, $l < k$, should at most cause n moves on level $l+1$. To be more concrete we study the relationship between the f-values of **G1** (which will be our Level 1) and the number of (i.e. Level 2) **H1** moves they can cause. In the current **H1** algorithm a node j can read a value $f_i(j)$ directly from node i and then perform $size_i(j)$ **H1** moves to propagate the effect of this value throughout $G_i(j)$. At a later stage node i can make use of $f_i(j)$ to alter the value of $q_i(j)$ which could again cause $size_i(j)$ new moves in $G_i(j)$. The same is true for $G_j(i)$ where there could be $size_j(i)$ moves initiated by node i reading $f_i(j)$. Then in a second sweep node j could alter $q_j(i)$ and again cause $size_j(i)$ moves before **H1** stabilizes.

This effect can be accredited to the fact that there are two different paths between $f_i(j)$ and each $q_s(t)$, except $q_i(j)$ and $q_j(i)$, along which moves can propagate. See Figures 4a and 4b for an illustration of how $f_i(j)$ can influence the values of **H1**.

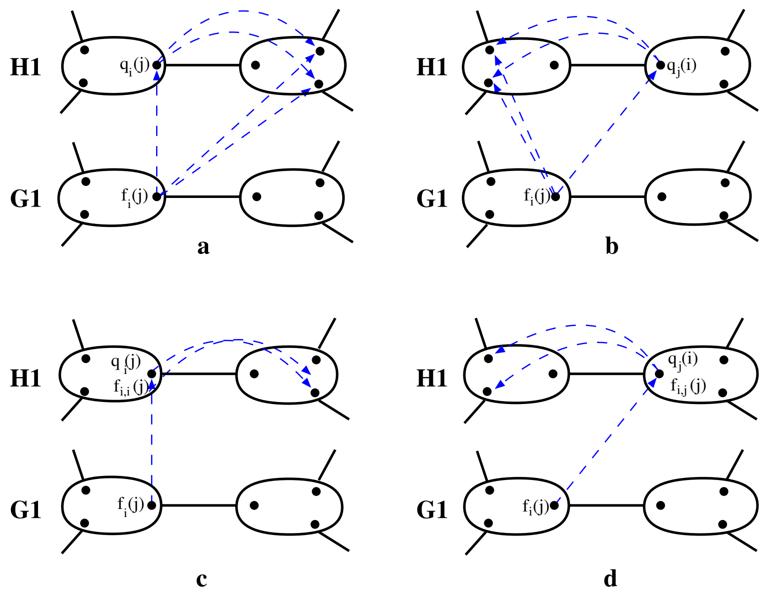

Figure 4: The influence of $f_i(j)$ on **H1** moves.

To avoid the possibility of two **H1** sweeps throughout G due to one $f_i(j)$ value we must ensure that $q_j(i)$ is updated using $f_i(j)$ before a sweep throughout $G_j(i)$ is performed and similarly that $q_i(j)$ is updated with $f_i(j)$ before a sweep of $G_i(j)$. This can be achieved if we create two new variables $f_{i,i}(j)$ and $f_{i,j}(j)$ which the value of $f_i(j)$ is to be copied into. The variable $f_{i,i}(j)$ resides on node i together with $q_i(j)$ and $f_{i,j}(j)$ on node j with $q_j(i)$. When node i updates the value of $q_i(j)$ using $f_i(j)$ it simultaneously makes sure that the value of $f_{i,i}(j)$ is set equal to $f_i(j)$ and similarly when node j updates $q_j(i)$ it makes sure that $f_{i,j}(j)$ is set equal to $f_i(j)$. Then when node i updates a value $q_i(r)$, $r \neq j$ it makes use of $f_{i,j}(j)$ (instead of $f_i(j)$) and similarly when node j updates $q_j(t)$, $t \neq i$ the value of $f_{i,i}(j)$ is used. This is as shown in Figures 4c and 4d. In this way the value of $q_i(j)$ will be updated with $f_i(j)$ before a propagation of moves through $G_i(j)$ and similarly will $q_j(i)$ be updated with $f_i(j)$ before a propagation of moves through $G_j(i)$. The resulting effect is that there will only be one path along which moves can be propagated from $f_i(j)$ to any q-value and thus there will at most be n **H1** moves initiated by $f_i(j)$. Note that with this setting there will be two extra values associated with each $q_i(j)$ namely $f_{i,i}(j)$ and $f_{j,i}(i)$. Both of these must be updated accordingly whenever $q_i(j)$ is updated.

For the k-level algorithm let $q_i^l(j)$, $j \in N(i)$, be a value on node i that is used on level l of the algorithm and let $h^l()$ be the function that is used for updating the value of $q_i^l(j)$. Then for $l > 1$ the following parameters can be used as input to $h^l()$:

- $\cup_{k \in N(i)-\{j\}} q_k^l(i)$
- $q_i^{l-1}(j), q_j^{l-1}(i)$
- $\cup_{k \in N(i)-\{j\}} (q_{i,k}^{l-1}(k), q_{k,k}^{l-1}(i))$

For $l = 1$ only the values $\cup_{k \in N(i)-\{j\}} q_k^l(i)$ are used as input. In addition to updating $q_i^l(j)$ we must also at the same time for $l > 1$ make sure that the values of $q_{i,i}^{l-1}(j)$ and $q_{j,i}^{l-1}(i)$ are updated. This can be achieved as follows. (The input parameters to $h^l()$ has been omitted for brevity.)

```
$H^l$:  **if**  $(\exists j \in N(i)$ such that $q_i^l(j) \neq h^l())$ **or** $(q_{i,i}^{l-1}(j) \neq q_i^{l-1}(j))$ **or** $(q_{j,i}^{l-1}(i) \neq q_j^{l-1}(i))$
        **then**
            **if**  $(\exists j \in N(i)$ such that $q_i^l(j) \neq h^l())$
                 $q_i(j) \leftarrow h^l()$
            **if**  $(q_{i,i}^{l-1}(j) \neq q_i^{l-1}(j))$
                 $q_{i,i}^{l-1}(j) \neq q_i^{l-1}(j)$
            **if**  $(q_{j,i}^{l-1}(i) \neq q_j^{l-1}(i))$
                 $q_{j,i}^{l-1}(i) \neq q_j^{l-1}(i)$
```

For $l = 1$ one would only test if $q_i^1(j) \neq h^1()$ and then set it appropriately. It follows from the above discussion that the total moves complexity of this algorithm is $O(n^{k+1})$.

6 Concluding remarks

We have described a generic 2-stage self-stabilizing algorithm to solve problems on a tree network where some kind of feedback mechanism is needed. Although each stage of the algorithm has a moves complexity of $O(n^2)$ we show that their combined moves complexity is $O(n^3)$. The generic algorithm is then instantiated to solve a number of problems on tree networks including 2-coloring, dynamic programming on trees, and finding a 2-center. Although self-stabilizing algorithms for each of theses problems have been presented before in the literature our algorithms reduce the moves complexity by at least a factor of $O(n^2)$ and also requires a less restrictive model.

We also generalize our algorithm into a k-level self-stabilizing algorithm with moves complexity $O(n^{2k})$.

One interesting research direction from this work would be to investigate if there are other multistage algorithms on more general topologies that also leads to a moves complexity less than the product of the individual algorithms.

References

[1] J. R. Blair and F. Manne, *Efficient self-stabilizing algorithms for tree networks.* To be presented at ICDCS'03, The 23rd International Conference on Distributed Computing Systems, 2003.

[2] S. C. Bruell, S. Ghosh, M. H. Karaata, and S. V. Pemmaraju, *Self-stabilizing algorithms for finding centers and medians of trees*, SIAM J. Comput., 29 (1999), pp. 600–614.

[3] S. Ghosh, A. Gupta, M. H. Karaata, and S. V. Permmaraju, *Self-stabilizing dynamic programming algorithms on trees*, in Proceedings of the Second Workshop on Self-Stabilizing Systems, 1995, pp. 11.1–11.15.

[4] S. Ghosh and M. H. Karaata, *A self-stabilizing algorithm for coloring planar graphs*, Distributed Computing, 7 (1993), pp. 55–59.

[5] M. Gradinariu and S. Tixeuil, *Self-stabilizing vertex coloring of arbitrary graphs*, in Proc. OPODIS'2000, 4th international conference on principles of distributed systems, 2000, pp. 55–70.

[6] F. Harary, *Graph Theory*, Addison-Wesley, 1972.

[7] S. T. Hedetniemi, D. P. Jacobs, and P. K. Srimani, *Self-stabilizing maximal independent set and grundy coloring in linear time.* Submitted.

[8] T. C. Huang, J.-C. Lin, and H.-J. Chen, *A self-stabilizing algorithm which finds a 2-center of a tree*, Computers and Mathematics with Applications, 40 (2000), pp. 607–624.

[9] J. W. H. Liu, *The role of elimination trees in sparse factorization*, SIAM J. Matrix Anal. Appl., 11 (1990), pp. 134–172.

[10] S. K. Shukla, D. Rosenkrantz, and S. S. Ravi, *Observations on self-stabilizing graph algorithms for anonymous networks*, in Proceedings of the Second Workshop on Self-stabilizing Systems, 1995, pp. 7.1–7.15.

[11] S. Sur and P. K. Srimani, *A self-stabilizing algorithm for coloring bipartite graphs*, Information Sciences, 69 (1992), pp. 219–227.