Efficient Self-stabilizing Graph Searching in Tree Networks

Jean Blair[1], Fredrik Manne[2], and Rodica Mihai[2,⋆]

[1] Department of EE and CS, United States Military Academy, West Point, NY 10996, USA
Jean.Blair@usma.edu
[2] Department of Informatics, University of Bergen, N-5020 Bergen, Norway
{fredrikm,rodica}@ii.uib.no

Abstract. The graph search problem asks for a strategy that enables a minimum sized team of searchers to capture a "fugitive" while it evades and potentially multiplies through a network. It is motivated by the need to eliminate fast spreading viruses and other malicious software agents in computer networks.

The current work improves on previous results with a self-stabilizing algorithm that clears an n node tree network using only $1+\log n$ searchers and $O(n \log n)$ moves after initialization. Since $\Theta(\log n)$ searchers are required to clear some tree networks even in the sequential case, this is the best that any self-stabilizing algorithm can do. The algorithm is based on a novel multi-layer traversal of the network.

1 Introduction

Networks of computing devices enable fast communication, pervasive sharing of information, and effective distributed computations that might otherwise be infeasible. Unfortunately, this comes at the cost of fast spreading viruses and malicious software agents. The fact that modern networks are constantly changing exacerbates the problem. Thus, it is important to regularly search a network in order to eliminate malicious software.

This setting has been formalized as various *graph search* problems where one asks for a strategy that will clear a graph of any unwanted "intruders" typically using as few operations as possible. One can think of a searcher as a separate software agent that must be run on the individual network devices in order to clear it. Thus, minimizing the number of searchers is also important as each searcher uses resources that the system could otherwise have used for productive work. One might further want to limit the number of concurrent searchers when there may be a cost associated with the maximum number used at any given time, for instance due to software licences.

There is a significant body of work focused on sequential algorithms for computing the minimum number of searchers required to search a graph and the

⋆ Now, International Research Institute of Stavanger, N-5008 Bergen, Norway.

S. Dolev et al. (Eds.): SSS 2010, LNCS 6366, pp. 111–125, 2010.

corresponding searching strategy. See [4] for an annotated bibliography. Peng et. al. and Skodinis gave linear-time sequential node and edge searching algorithms for trees [9,10] and in [6,7] it was proven that in general $\Theta(\log n)$ searchers are required for tree networks. Distributed graph searching algorithms have also received considerable attention. See [8] for a list of the most recent works. For tree networks [3] gives a distributed algorithm to compute the minimum number of searchers necessary to clear the edges.

The first self-stabilizing algorithm for solving the node search problem in a tree was introduced in [8]. Given a tree T with n nodes and height h, their algorithm stabilizes after $O(n^3)$ time steps under the distributed adversarial daemon and the number of searchers used to clear the tree T is h.

In this paper we give an efficient self-stabilizing algorithm that improves on the results in [8]. Our algorithm is based on integrating two existing self-stabilizing algorithms with a new search algorithm in order to continuously search a tree network with $n \geq 2$ nodes using only $1 + \lfloor \log n \rfloor$ searchers and $O(n \log n)$ moves after initialization. As our algorithm is non-silent and self-stabilizing it will adapt to any transient faults or changes in the configuration of the network. Moreover, if an intruder is (re)introduced in a cleared network, the algorithm will immediately clear the network again.

We use the leader election algorithm from [2] for rooting the tree and then apply the efficient multiwave algorithm introduced in [1] to initialize the tree. This is then followed by our new search algorithm to clear the tree. The search algorithm works by recursively splitting the graph into smaller components and then clearing each of these. In this way the algorithm behaves as if it was composed of a number of layered algorithms each with its own set of variables. However, the clearing is achieved with just one efficient algorithm and with the number of variables linear in the size of the graph.

The paper is organized as follows. In Section 2 we give preliminaries, followed by a presentation and motivation of our algorithm in Section 3. In Section 4 we show the correctness of our algorithm before concluding in Section 5.

2 Preliminaries

The current focus is on a variant of the graph search problem known as *node searching*. A node search strategy for a graph $G = (V, E)$ consists of a sequence of steps where, at each step, searchers may be both added to and removed from the nodes of G. A node is cleared once a searcher is placed on it and an edge is cleared when searchers occupy both of its endpoints at the same time.[1] A node that has a searcher on it is guarded and cannot be recontaminated as long as the searcher is present on that node. However, cleared edges and cleared nodes without searchers on them are assumed to be recontaminated instantly iff there exists a path of unguarded nodes from them to an uncleared node or edge of the

[1] As will be discussed in the concluding remarks, our results can easily be adapted to solve other graph search variants such as *edge search* or *mixed search*.

graph. The graph search problem then asks for a search strategy that ensures that the entire graph is cleared while using as few searchers as possible.

In our distributed computational model each node of G has a unique identifier and also stores a number of local variables which it can manipulate. A node can read the local variables of its neighbors, i.e. the shared memory model. As is typical, we present our self-stabilizing algorithms as a set of rules where the predicate of each rule only depends on variables local to a node and those of its neighbors. Further, we assume that rules can only be executed during fixed time steps and that a distributed unfair daemon governs which node(s) get to execute a move at any given time step. This means that any non-empty subset of enabled rules may execute during a time step. Note that this is the most general type of daemon. We measure complexity both in terms of the number of rules that have been executed (moves-complexity) and also in terms of the number of rounds, where a round is the smallest sub-sequence of an execution in which every node that was eligible at the beginning of the round either makes a move or has had its predicate disabled since the beginning of the round.

The goal of a self-stabilizing algorithm is for the global system to reach a stable configuration (i.e. where no moves can be made) that conforms to a given specification, independent of its initial configuration and without external intervention. A *non-silent* self-stabilizing algorithm will also reach a configuration that conforms to the specification, but will continue to execute indefinitely once it has done so. Moreover, in the absence of transient faults, the algorithm will continue to conform to the specification.

3 The Algorithm

The overall graph searching algorithm integrates three separate self-stabilizing processes. Initially, we use the leader election algorithm from [2] to elect an $n/2$-separator as the root r (i.e. no component of $G - \{r\}$ contains more than $n/2$ nodes). Moreover, this algorithm ensures that each node knows in which direction r is. Following this, r uses a variant of the multi-wave Propagate-Information-with-Feedback (PIF) process from [1] to get the network ready for searching. This is accomplished by r continuously iterating through a circular list of its children, and for each iteration concurrently initiating two PIF processes. When r sets $child_r$ to point to a node in $N(r)$ the subtree rooted at $child_r$ transitions to the ACTIVE state while all other subtrees are either in the SLEEP state or are in the process of transitioning to SLEEP. Only nodes in the ACTIVE state can participate in the ensuing search process. When $child_r$ signals that the search of its subtree is complete and the next node in r's neighbor list signals that it is SLEEP, r will advance $child_r$ and repeat the process.

The overall process is outlined in Algorithm 1. Loosely speaking, after the system first reaches line 6 the network will have reached a "normal configuration" and thereafter the search process behaves as expected. Note that the algorithm is non-silent.

The entire process uses $5+2\delta(u)$ variables on each node u, where $\delta(u) = |N(u)|$ and $N(u)$ denotes the neighbors of u. For the leader election process we maintain

Algorithm 1. The overall graph searching process

```
 1: Elect an n/2-separator as root r and set all p_u to point towards r; /* L1-L5 */
 2: child_r ← an arbitrary neighbor of r;                          /* R1 */
 3: Signal all v ∈ N(r) − {child_r} to go to SLEEP;      /* consequence of R1 */
 4: loop
 5:   when (Next(child_r) is SLEEP) & (child_r signals "search completed") do
 6:     prev ← child_r; child_r ← Next(child_r);                   /* R2 */
 7:     do in parallel
 8:       Transition subtree rooted at child_r to ACTIVE;          /* T1-T4 */
 9:       Transition subtree rooted at prev to SLEEP;              /* T5 */
10:     Search the subtree rooted at child_r;                      /* S1-S4 */
```

a parent pointer p_u and for each neighbor $v \in N(u)$ a $size_u(v)$ value that will hold the size of the subgraph containing u in the original graph with the edge (u, v) removed. The results in [2] guarantee that once the election process stabilizes, all p_u and $size_u(v)$ values are correct. Since those values are not changed in any other part of the algorithm, in the absence of spurious faults they remain correct throughout. The remaining $4+\delta(u)$ variables are used in the other two processes and will be described in the follow-on subsections.

We use the leader election algorithm, which we refer to as the L-algorithm implemented as rules L1-L5, exactly as specified in [2] and therefore will not further describe it here. The next subsection explains the transition process and our implementation of it. Subsection 3.2 describes the new search process.

3.1 Transition - Lines 8 and 9

The transition processes in lines 8 and 9 together implement a full 2-wave PIF process similar to that designed by Bein, Datta, and Karaata in [1]. For each wave except the last, their algorithm broadcasts information down from the root and then propagates feedback up from the leaves. As presented in [1] the last wave is cut short, only broadcasting down. We, however, include this last propagation of feedback up, since we need the feedback to guarantee that the last broadcast reached the leaves before starting the search process.

The complete state-transition process from the perspective of one node u transitions its variable $state_u$ through a series of five states (SLEEP → AWAKE → CSIZE-UP → CSIZE-DOWN → ACTIVE) as depicted in Figure 1. After initialization, an arbitrary node u rests in the SLEEP state and expects its parent p_u to be in the SLEEP state as well. Only when u sees that p_u is AWAKE will u transition away from SLEEP, moving to the AWAKE state and thereby playing its part in a "wake-up" broadcast. Following this, u waits in the AWAKE state until all of its children transition themselves from SLEEP through AWAKE and finally to CSIZE-UP; only then will u itself transition to CSIZE-UP, thereby providing "I'm awake" feedback to p_u. The leaves are the first to switch from the broadcast down state AWAKE to the feedback up state CSIZE-UP, since they have no children. The switch from this first propagate feedback ("I'm awake") stage to the second broadcast ("get

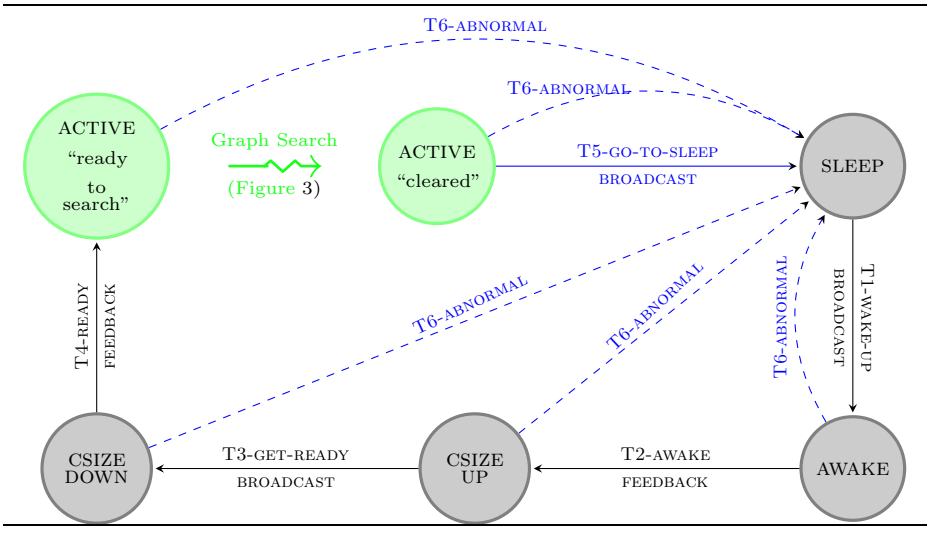

Fig. 1. The T moves process is a PIF process

ready to search") stage occurs at the root, where once its neighbor $child_r$ is in the CSIZE-UP state it effectively moves itself through CSIZE-UP to CSIZE-DOWN, thereby initiating the second wave transitioning all nodes to CSIZE-DOWN once their parent is CSIZE-DOWN. Again, the leaves turn the broadcast down wave into "I'm ready to search" feedback by transitioning through CSIZE-DOWN and to ACTIVE. Only when all of u's children are ACTIVE does u transition itself to ACTIVE. Once ACTIVE, a node is eligible to participate in the search process described in the next subsection.

If at any point during the search process u sees that p_u is no longer ACTIVE (i.e., has transitioned to SLEEP), then u itself will transition to SLEEP. During normal processing this will be initiated as a broadcast down wave from the root only when the entire subtree containing u has been cleared. If, at any point after leader election is complete, the states that p_u, u, and u's children are in do not make sense with respect to this transition process, then u will respond to the abnormal configuration by jumping to SLEEP.

There are three main tasks that we need the transition process to accomplish in order to correctly implement the search using only $\lfloor \log n \rfloor + 1$ searchers. First, the transition process computes in each node the size of the current subtree assuming the edge between the root and the subtree does not exist. These values are collectively stored in the $csize()$ vectors in exactly the same way as the $size()$ vectors holds the size of the subtrees of the entire graph. The $csize()$ values are computed in two waves. During the first feedback ("I'm awake") wave the size of the subtree below each node is calculated using the propagated up $csize()$ values from its children. Then, during the second broadcast ("get ready to search") wave the size of the subtree above each node (i.e., the portion of the subtree reached through the parent) is calculated using the propagated down $csize()$ values from p_u.

function ParentState (u)**:**
  if Root (p_u)
  then if $child_{p_u} = u$
    then if $state_u =$ CSIZE-UP
      then return CSIZE-DOWN
      elseif $state_u =$ SLEEP
      then return AWAKE
      else return $state_u$
    else return SLEEP
  else return $state_{p_u}$

function CalculateComponents (u)**:**
  foreach $v \in N(u)$
    $csize_u(v) \leftarrow 1 + \sum_{x \in N(u)-\{v\}} csize_x(u)$
  return $csize_u$

function ReadyToSearch (u)**:**
  /* check if leaf */
  if $|N(u)| = 1$
  then if $csize_{p_u}(u) = 0$
    then return true
    else return false
  $n \leftarrow csize_u(p_u) + csize_{p_u}(u)$
  if $(\forall v \in N(u),\ csize_v(u) < n/2)$
  then return true
  if $(\exists v \in N(u), csize_v(u) > n/2)$
  then return false
  if $(\exists v \in N(u), csize_v(u) = n/2)$
  then if $(ID_u > ID_v)$
    then return true
    else return false

Fig. 2. Auxilliary functions

The second main task that the transition process accomplishes is to initialize, during the last feedback ("I'm ready to search") wave, the other variables needed to begin the search process. Combining these first two tasks with the transition process from [1] we end up with a six rule implementation that accomplishes the following. (Red font highlights the difference between our implementation and the rules given in [1]).

- T1-WAKE-UP (rule iB in [1]): broadcast down a "wake-up" call (i.e., transition to AWAKE).
- T2-AWAKE (iF, lF): propagate up both "I'm awake" feedback (i.e., transition to CSIZE-UP) and the size of the subtree rooted at u.
- T3-GET-READY (iB): broadcast down both a "get ready to search" call (i.e., transition to CSIZE-DOWN) and the size of the subtree above.
- T4-READY (iF, lF): propagate up "I'm ready to search" feedback (i.e., transition to ACTIVE) and initialize variables needed for the search.
- T5-GO-TO-SLEEP (iB, lB): broadcast down a "go to sleep" call (i.e., transition to SLEEP).
- T6-ABNORMAL (iCa, lCa): jump to SLEEP if any *state* value in the neighborhood does not make sense.

We refer to these six rules as the T-algorithm. Note that for the T-algorithm u uses $1+\delta(u)$ additional variables: the state variable $state_u$ and for each neighbor $v \in N(u)$, $csize_u(v)$.

The third task that the T-algorithm accomplishes is to admit the search process in only one subtree of the root at a time. We accomplish this by using a **ParentState**(u) function to report the state of a node u's parent p_u.

If p_u is not the root, the function simply returns p_u's state. However, when p_u is the root the state it returns is dependent on whether or not $u = child_r$ (see

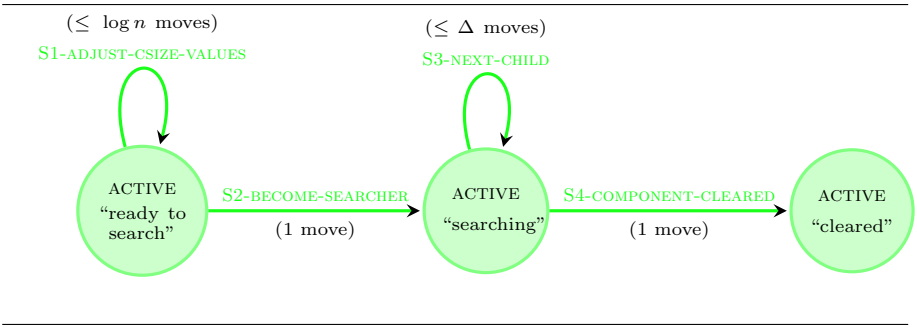

Fig. 3. The S moves process implements graph search

Figure 2 where red font again highlights differences from the implementation in [1]). To see how this works, consider a call to **ParentState**(u) when the parent p_u is the root. Then the return value is SLEEP if u is not the root's current $child_r$; otherwise the return value is the appropriate state in the transition process relative to u's current state. For example, if $u = child_r$ has not yet begun transitioning away from SLEEP then the function returns AWAKE in order to initiate a "wake-up" broadcast in u's subtree.

The results in [1] prove that rules T1 - T6 are executed in a well structured manner, giving the following result.

Lemma 1. *When the root initiates a multi-wave broadcast-feedback process in a subtree C_1 after initialization the following properties hold:*

(a) Every node in C_1 will execute, in order, rules T1, T2, T3, and then T4.
(b) For any $v \in C_1$, when v executes T2, all descendants of v will have already executed T2.
(c) All nodes in C_1 will execute T2 before any node in C_1 executes T3.
(d) For any $v \in C_1$, when v executes T3, all ancestors of v will have already executed T3.
(e) For any $v \in C_1$, when v executes T4, all descendants of v will have already executed T4.

3.2 Search - Lines 2, 3, 5, 6, and 10

The rules used to implement the search process are divided into two sets called the R-algorithm (implementing lines 2, 3, 5, and 6) and the S-algorithm (implementing line 10). The R-algorithm executes only on the root and the S-algorithm executes only on non-root nodes. As mentioned above, the S-algorithm takes place in one subtree of the root at a time. We call that subtree a component of the graph and recursively search components by placing a searcher at the $n'/2$-separator of the component, where n' denotes the number of vertices in the component. We denote such a separator as a *center* node of the component and use this to partition the current component into smaller components, searching

each of these in turn. Once a component is cleared, the last searcher in that component is released before returning control to a previous level in the recursion. The root always hosts a searcher. It is this recursive behavior of splitting the graph at its center that admits a graph searching process that uses only $\lfloor \log n \rfloor + 1$ searchers.

The R- and S-algorithms use the remaining 3 variables. Boolean variables $searcher_u$ and $cleared_u$ are used respectively to indicate the presence of a searcher on u and to signal that after u's most recent T4 move u has been searched and either currently maintains a searcher as a guard or is cleared and guarded somewhere else. Finally, a pointer $child_u$ is used to point to the current neighboring subcomponent that is being searched. We also assume that each node u has a list of its neighbors beginning with **FirstChild**(u) and ending with p_u. The function **Next**$(child_u)$ advances $child_u$ through this list.

The S-algorithm includes five rules that systematically progress through the ACTIVE state as shown in Figure 3. Details are given as Algorithm 2. Rule S1 adjusts the vector $csize_u()$ using the auxiliary function **CalculateComponents** shown in Figure 2. With this the $csize_u()$ values can reflect the reduced size of the current component. When a node u has up-to-date $csize_u()$ values and it

Algorithm 2. Search process on non-root node u (S-algorithm)

```
/* S1 - S4 only possible for ACTIVE non-root with entire neighborhood
   ACTIVE                                                          */
```

S1-ADJUST-CSIZE-VALUES**:**

1: **if** $(\neg\, cleared_u)$ & $(\exists v \in N(u), csize_u(v) \neq 1 + \sum_{x \in N(u)-\{v\}} csize_x(u))$ **then**
2: $\quad csize_u \leftarrow$ **CalculateComponents**(u);

S2-BECOME-SEARCHER**:**

1: **if** (**ReadyToSearch**(u)) & $(\neg\, cleared_u)$ & $(child_u =$ NULL$)$ **then**
2: $\quad searcher_u \leftarrow$ true; $cleared_u \leftarrow$ true; `/* clears u */`
3: $\quad child_u \leftarrow$ **FirstChild**(u); $csize_u(child_u) \leftarrow 0$;

S3-NEXT-CHILD**:**

1: **if** $(child_u \neq p_u)$ & $(child_{child_u} = u)$ & $(\neg\, searcher_{child_u})$ **then**
2: $\quad child_u \leftarrow$ **Next**$(child_u)$; $csize_u(child_u) \leftarrow 0$;

S4-COMPONENT-CLEARED**:**

1: **if** $(child_u = p_u)$ & $(child_{p_u} = u)$ & $(searcher_u)$ & $(searcher_{p_u})$ **then**
2: $\quad searcher_u \leftarrow$ false; `/* edge (u, p_u) is cleared; release u's searcher */`

S5-ABNORMAL**:**

1: **if** $(child_u \notin N(u) \cup \{$NULL$\})$ || $((child_u \in N(u))$ & $(csize_u(child_u) \neq 0))$
$\quad$ || $((child_u =$ NULL$)$ & $(searcher_u$ || $cleared_u))$
$\quad$ || $((child_u \in N(u))$ & $(p_{child_u} = u)$ & $(child_{child_u} = u)$ & $(\neg\, searcher_u))$ **then**
2: $\quad child_u \leftarrow p_u$; $csize_u(p_u) \leftarrow 0$;
3: $\quad searcher_u \leftarrow$ false; $cleared_u \leftarrow$ true;

is the center of the current component, the Boolean function **ReadyToSearch** given in Figure 2 evaluates to true and S2 is enabled on u. An S2 move places a searcher on the node u that is executing the move and starts the search process in one of its neighboring subcomponents (the one pointed to by $child_u$). The S3 move is enabled only after the current $child_u$ subcomponent is cleared, at which time u advances its child pointer to the next child. After u has progressed through each of its non-parent neighbors, S3 is no longer enabled. The S4 move is used to release the searcher on u only after all children components are cleared and the parent p_u also has a searcher on it to guard u's cleared component. Note that just before the S4 move releases the searcher on u, we also know that the edge (u, p_u) has been cleared.

After initialization in the root r's neighborhood (either explicitly through execution of R1 or implicitly with initial values in $N(r)$), the R-algorithm is a continuous series of R2 moves executed by r, each followed by a period of waiting until the current $child_r$ is cleared. An ACTIVE $child_r$ and the subtree rooted at $child_r$ is considered to be cleared when $child_{child_r} = r$ and $child_r$ has released its searcher (i.e., $\neg searcher_{child_r}$). Details are given as Algorithm 3.

Algorithm 3. Search process on root node u (R-algorithm)

```
/* R1 - R2 only possible for root with consistent size values in
   neighborhood                                                    */
```

R1-ROOT-INITIALIZE**:**

1: **if** $(\neg\ searcher_u)$ & $(child_u \notin N(u))$ & $(\exists v \in N(u), csize_u(v) \neq 0)$ **then**
2: $\quad searcher_u \leftarrow$ true; /* clears u */;
3: $\quad \forall v \in N(u),\ csize_u(v) \leftarrow 0$;
4: $\quad child_u \leftarrow$ **FirstChild**(u);

R2-ROOT-NEXT-CHILD**:**

1: **if** $(child_{child_u} = u)$ & $(\neg\ searcher_{child_u})$ & $(state_{child_u} =$ ACTIVE$)$ & $(state_{\mathbf{Next}(child_u)} =$ SLEEP$)$ **then**
2: $\quad child_u \leftarrow$ **Next**$(child_u)$;

4 Correctness

We show the correctness of our algorithm in five parts. The first part is an immediate consequence of the results in [2], where they prove that at most $O(n^2)$ time steps (or h rounds, where h is the diameter of G) contain L moves. Thus if we can show that at any time prior to stabilization of the L-algorithm the R-, T- and S-algorithms will stabilize given that there are no L moves, then it will follow that the L-algorithm will stabilize with the $n/2$-separator as the global root, designated by r, and with every parent pointer p_v set appropriately. The fact that the R-, T- and S-algorithms stabilize will follow from the remainder of the correctness proof.

Next we argue in Subsection 4.1 below that the R-algorithm behaves as expected, assuming the T- and S-algorithms stabilize in the absence of any L- or

R moves. This argument shows that after stabilization of the L-algorithm r will begin executing R2 moves and that just prior to any R2 move, $state_y = \text{SLEEP}$ for $y = \mathbf{Next}(child_r)$. The results in [1] then guarantee that every node in the subtree rooted at y will be in the SLEEP state before it begins the T-algorithm that will be initiated by r with the next R2 move.

We then show in Subsection 4.2 that each R2 move will initiate the T-algorithm in the subtree rooted at the assigned-in-R2 pointer $child_r$ and that each node in that subtree will have the correct initial values for the S-algorithm before it is eligible to execute any S move. Subsection 4.3 then addresses the key correctness result, proving that starting from a normal configuration the S-algorithm correctly searches the subtree rooted at $child_r$ using no more than $\lfloor \log n \rfloor$ searchers at any point in time. For the final portion of the correctness proof Subsection 4.4 argues that the T- and S-algorithms will stabilize correctly in the absence of any L or R moves regardless of the initial configuration and that before the L-algorithm stabilizes, the R-algorithm will stabilize in the absence of any L moves.

Note that the premise required for the first part, that before stabilization of the L-algorithm the R-, T- and S-algorithms stabilize in the absence of any L moves, follows from these results. That is, the results in Subsections 4.2 and 4.3 ensure, respectively, that starting from a normal configuration the T-, and S-algorithms stabilize. The results in Subsection 4.4 ensure that before the L-algorithm stabilizes and/or starting from an abnormal configuration all three algorithms stabilize.

Due to space limitations we do not give the details of most proofs. Details are, however, available upon request.

4.1 The R-algorithm

We focus in this subsection on R moves made after the L-algorithm has stabilized with the $n/2$-separator as r.

Only a node believing it is r is enabled to execute any R move. Moreover, since no T- or S move is privileged on r, R moves are the only possible moves r can make. Furthermore, because the T- and S-algorithms will stabilize (Subsections 4.2 and 4.3), we know that r will be given an opportunity to execute enabled R moves. In light of this, we prove here the following lemma.

Lemma 2. *After the L-algorithm has stabilized:*

(a) r will execute R2 with appropriate variable values and
(b) just before any R2 move, $state_y = \text{SLEEP}$ for $y = \mathbf{Next}(child_r)$, ensuring that every node v in the subtree rooted at y is either in the SLEEP *state or will transition to* SLEEP *before any subsequent S move.*

4.2 The T-algorithm

In this subsection we consider what happens just after r executes an R2 move. Let C be the component of $G - \{r\}$ containing $child_r$ following the R2 move. We

will show that each $v \in C$ will execute the T-algorithm and that, after v's last T move, its variables will be correctly initialized for the start of the S-algorithm. To that end, we say that a $csize_v(x)$ value is *correct-with-respect-to* C if $v \in C$ and $csize_v(x)$ is equal to the size of the component in $C \backslash \{(v, x)\}$ containing v.

From Lemma 2 we know that each node in C will either be in the SLEEP state just after r makes the R2 move, or will subsequently transition to SLEEP before its parent (and hence it) can begin the T-algorithm. Then it follows from Lemma 1 that each node in C will perform T1 through T4 in sequence. This is the foundation needed to prove that the configuration at each $v \in C$ following its T4 move is what is needed before it begins the S-algorithm.

Lemma 3. *Starting from a normal configuration, after a vertex $v \in C$ makes a T4 move and before it makes any S move, the following properties hold:*

(a) $cleared_v = \text{false}$.
(b) $searcher_v = \text{false}$.
(c) $child_v = \text{NULL}$.
(d) For all $x \in N(v)$, $csize_v(x)$ and $csize_x(v)$ are correct-with-respect-to C.

4.3 The S-algorithm

In this subsection we will show that once r has executed an R2 move setting $child_r = v$ the subtree C rooted at v will be searched and cleared. Moreover, the search will stabilize with v's variable values enabling an R2 move on r, thus continuing the search in G. We will also show that the process of searching C never uses more than $\log n$ searchers simultaneously and that the total number of moves is at most $O(|C| \log |C|)$.

It follows from the discussion in the previous section that every node in C will execute the entire T-algorithm before it can start executing the S-algorithm. Thus even though both the T- and S-algorithms might execute concurrently any node that has not yet finished the T-algorithm is eligible to continue executing the T-algorithm. Also, no values that are set by the S-algorithm are used in the predicates of the T-algorithm. Note in particular that $csize()$ values used by the T-algorithm are only influenced by other $csize()$ values from nodes where the T-algorithm is still running. We will therefore assume that the T-algorithm has run to completion on C when the S-algorithm starts. Thus we assume that when the S-algorithm starts we know from Lemma 3 that all $csize()$ values are correct-with-respect-to C, $searcher_v = \text{false}$ and $child_v = \text{false}$ for each $v \in C$. The value of $searcher_v$ can only be set to true if v executes an S2 move. To keep track of the searchers that are used at any given time, we will label a node that executes an S2 move as an L_i searcher where $i - 1$ is the number of searchers in C just prior to the move. For completeness we define $L_0 = r$.

It is conceivable that more than one node in C could execute an S2 move during the same time step and thus we could have more than one L_i searcher at the same time. But, as we will show, only one node in C can execute an S2 move during any time step. We will also show that the searchers are placed and removed in a first-in-last-out fashion. Thus every L_j, $j < i$, will still be a searcher when L_i ceases to be so.

With these assumptions we define the *components* of $G - \{L_0, \ldots, L_{i-1}\}$ that are incident on any L_{i-1} as the C_i components of G. It follows then that C is a C_1 component. Again for completeness we define $C_0 = G$. Additionally, if the value of $child_{L_{i-1}}$ is pointing to a node $v \in C_i \cap N(L_{i-1})$ with $csize_{L_{i-1}}(v) = 0$, then we denote the C_i component containing v as the *active* C_i component. As we will show all active components are in C and are nested within each other.

The *center* of a component C_i, denoted by $center(C_i)$, is the $|C_i|/2$-separator. Ties are broken using the highest ID. Let C_i be a component and let $v \in C_i$ and $x \in N(v)$. Then the *actual size* of the subtree containing v in $C_i - \{x\}$, is denoted by $actual_v^i(x)$.

Again, looking at $C = C_1$ it is clear that until r makes a subsequent R2 move, C_1 remains active and that following the T-algorithm $csize_v(x) = actual_v^1(x)$ for every $v \in C_1$. Before proceeding we need the following result about how S1 moves will update the $csize()$ values in an active component.

Lemma 4. *Let $x_0 = L_{i-1}$ be incident on an active component C_i and let $x_1, x_2 \cdots, x_j$ be any path in C_i where the following properties hold:*

- *$x_1 \in N(x_0)$ and $csize_{x_0}(x_1) = 0$.*
- *$csize_{x_k}(x_{k+1}) > B$ for some constant $B \geq |C_i|$ and each k, $0 < k < j$.*
- *$csize_y(x_k)$ is correct-with-respect-to $C_i - \{x_k\}$ for each x_k, $0 < k \leq j$, and $y \in N(x_k)$ where y is not on the path $x_0, \cdots, x_j$.*

Assume further that the only type of moves that are executed on any node are S1 moves by the nodes $x_1, x_2, \cdots, x_j$. Then for each x_k, $0 < k \leq j$:

(a) The S1 moves will stabilize with $csize_{x_k}(x_{k+1}) = actual_{x_k}^i(x_{k+1})$.
(b) At each time step prior to the move where $csize_{x_k}(x_{k+1})$ obtains its final value $csize_{x_k}(x_{k+1}) > B$.

Lemma 4 identifies the very structured way in which any $csize_v(x)$ value will evolve with a series of S1 moves in the absence of any other S moves; essentially they will decrease, jumping from one actual value to some subsequent actual value, ending when v becomes the center of an active component. As it turns out, this characteristic of the changing $csize_v(x)$ values persists even in the presence of other S moves within the active components.

We will use the following defined characteristic of an active component when we later show how searchers are activated.

Definition 1. *An active component C_i where $|C_i| > 0$ is* ready-for-searching *if the following is true immediately following the S2 or S3 move by an L_{i-1} node that defined C_i:*

(a) Every L_j, $0 \leq j < i$, such that there exists an edge (L_j, v) for some $v \in C_i$ has $child_{L_j} = v$ and $csize_{L_j}(child_{L_j}) = 0$.
(b) For every $v \in C_i$, $cleared_v =$ false.
(c) For each $v \in C_i$ let $x \in N(v)$ be the neighbor of v on the path from v to L_{i-1}. Then $csize_v(x) = actual_v^{i-1}(x)$.

(d) For each $v \in C_i$ let $x \in N(v)$ be a neighbor of v not on the path from v to L_{i-1}. Then $csize_v(x) \geq |C_i|$.

Note that following the T-algorithm on $C = C_1$ all $csize_v()$ for $v \in C_1$ are correct-with-respect-to C_1. Thus the only move that can be executed in C_1 is S2 by $center(C_1)$ which then becomes the first (and currently only) L_1 node. If $|C_1| > 1$ then $child_{L_1}$ is set to some $v \in C_1$ thus defining one or more C_2 components. Since at that time $csize_{L_1}(v) = 0$ it follows that the C_2 component of $C_1 - \{L_1\}$ containing v is also active. Moreover, it is not hard to see that this C_2 component is ready-for-searching. Since the $csize()$ values surrounding any other C_2 components have not changed these will remain stable until L_1 makes a subsequent S3 move, changing a $csize()$ value next to the other C_2 component.

As the next lemma shows, the S-algorithm will recursively continue to create new nested components that are ready-for-searching. Moreover, it will do so in a sequential fashion.

Lemma 5. *When the S-algorithm runs on a ready-for-searching component C_i, $i > 1$:*

(i) The first node in C_i to execute an S2 move will be $center(C_i)$.
(ii) If $|C_i| > 1$ then the C_{i+1} component that $child_{center(C_i)}$ points to just after its S2 move will be ready-for-searching at that time.
(iii) The nodes in $C_i - C_{i+1} - \{center(C_i)\}$ can only execute S1 moves until $center(C_i)$ makes an S3 move.

Since L_1 is the only initial searcher in C_1 it follows from Lemma 5 that the S-algorithm will continue to create new nested components $C_2, \ldots, C_i$ one at a time until either a node L_i is a leaf or a node L_i sets $child_{L_i} = L_j$ where $j < i$. In both cases, L_i then sees $|C_{i+1}| = 0$. In the following we show how the recursion returns and that when it does so each component has been cleared and will not be recontaminated. First we need the following definition of a subtree that has been searched.

Definition 2. *Let $(v, w) \in E$ be where $w = p_v$. The subtree H of $G - \{w\}$ with v as its root is* cleared-and-guarded *if:*

- *For every $x \in V(H) - \{v\}$ $searcher_v =$ true and $searcher_x =$ false,*
- *Every $x \in V(H)$ has $child_x = p_x$, and*
- *Every $x \in V(H)$ has been cleared and no recontamination has occurred.*

Note that it is only the node v closest to r that can possibly make a move in a subtree that is cleared-and-guarded and this can only happen if p_v has $child_{p_v} = v$, at which point v is eligible to execute an S4 move.

We can now show that the S-algorithm, when started correctly on a component C_i, will return with the entire C_i being cleared-and-guarded.

Lemma 6. *Let C_i be a component of G such that C_i is ready-for-searching and every subgraph of $G - C_i$ adjacent to C_i except possibly the one closest to r*

is cleared-and-guarded. Further, let $(v, w) \in E$ *be such that* $v \in C_i$ *and* $w \notin C_i$ *while* $p_v = w$*. Then following the S2 or S3 move that defined* C_i*, the S-algorithm will reach a point where the component of* $G - \{w\}$ *containing* v*, will be cleared-and-guarded. In doing so the algorithm will not use more than* $\lfloor \log |C_i| \rfloor + 1$ *new searchers simultaneously.*

Because each S2 and S3 move designates exactly one of its adjacent subcomponents to next recursively start the S-algorithm, we can prove the following.

Lemma 7. *The number of moves executed by the S-algorithm on a component* C_i *which satisfies the conditions for Lemma 6 is* $O(|C_i| \log |C_i|)$.

Note that the proof of Lemma 7 establishes that, for any $v \in C_i$, the number of times that v executes each S move is as specified in Figure 3.

4.4 Initialization

In this subsection we show that the algorithm will reach a normal configuration. We do this by first showing that in the absence of any L, R and T moves the S-algorithm will stabilize, regardless of the initial configuration. This result, together with the results in [1], ensures that in the absence of any L or R moves, the T- and S-algorithms together will stabilize in a normal configuration. Our other initialization result, Lemma 9, uses this to then show that in the absence of any L moves, the combined R-, T- and S-algorithms stabilize in a normal configuration. Since the results in [2] guarantee that as long as this is the case the L-algorithm will stabilize we will then have proven that the integrated self-stabilizing algorithm that includes the five L rules, the two R rules, the six T rules and the five S rules will reach a normal configuration.

Lemma 8. *Let* $H = (V_H, E_H)$ *be a maximal connected subgraph of* G *such that every* $v \in V_H$ *has* $state_v = \text{ACTIVE}$ *and let* $n = |V_H|$*. Then in the absence of any L, T or R moves, the S-algorithm will stabilize in* H *using* $O(n^2)$ *moves.*

Lemma 9. *Let* $v = \textbf{FirstChild}(r)$ *after an R1 move by* r *and let* C_1 *be the component of* $G - \{r\}$ *containing* v*. Then in the absence of any L moves, the T- and S-algorithms will stabilize in* C_1 *with* $child_v = r$ *and* $searcher_v = \text{false}$.

4.5 The Integrated Algorithm

We are now ready to prove the final results.

Theorem 1. *Starting from an arbitrary configuration the L-, T-, R- and S-algorithms combined reach a point when the* $n/2$*-separator of* G *is* r*, all* p_v *point in the direction of* r*, and the entire network is in a normal configuration when the* r *executes the R2 move.*

Proof. Follows from Lemmas 2, 8, 9 and the results in [2] and [1].

Theorem 2. *After reaching a normal configuration, following any R2 move on* r *the combined L-, T-, R- and S-algorithm will clear all of* G *in* $O(n \log n)$ *moves using no more than* $1 + \lfloor \log n \rfloor$ *searchers.*

5 Concluding Remarks

We have given an efficient non-silent self-stabilizing algorithm for graph searching in trees. The algorithm integrates three separate self-stabilizing processes and ensures that each behaves as expected even in the presence of the other processes.

Although as presented the algorithm addresses the node search problem, it can be easily modified to perform edge searching or mixed searching. In the edge search variant searchers are slid through the edges. Let us consider a node u and its parent v. If v has a searcher on it and u is the next node that will receive a searcher then instead of placing the searcher directly on u we can place the searcher on v and then slide it to u. By doing this we can transform the node search strategy to an edge search strategy.

Due to the sequential nature of our algorithm, the number of rounds for it to execute could be on the same order as the number of moves.

Our algorithm can be used to solve other types of problems that require recursive decomposition of the graph by identifying the centers at each level of the decomposition. For example, it is straight-forward to adapt our algorithm to find a 2-center of a tree using only $O(n)$ moves after initialization, improving over the algorithm given in [5].

References

1. Bein, D., Datta, A.K., Karaata, M.H.: An optimal snap-stabilizing multi-wave algorithm. The Computer Journal 50, 332–340 (2007)
2. Blair, J.R.S., Manne, F.: Efficient self-stabilizing algorithms for tree networks. In: Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems (ICDCS), pp. 912–921 (2003)
3. Coudert, D., Huc, F., Mazauric, D.: A distributed algorithm for computing and updating the process number of a forest. In: Taubenfeld, G. (ed.) DISC 2008. LNCS, vol. 5218, pp. 500–501. Springer, Heidelberg (2008)
4. Fomin, F.V., Thilikos, D.M.: An annotated bibliography on guaranteed graph searching. Theor. Comput. Sci. 399, 236–245 (2008)
5. Huang, T.C., Lin, J.C., Chen, H.J.: A self-stabilizing algorithm which finds a 2-center of a tree. Computers and Mathematics with Applications 40, 607–624 (2000)
6. Kinnersley, N.G.: The vertex separation number of a graph equals its path-width. Inf. Process. Lett. 42, 345–350 (1992)
7. Korach, E., Solel, N.: Tree-width, path-width, and cutwidth. Discrete Appl. Math. 43, 97–101 (1993)
8. Mihai, R., Mjelde, M.: A self-stabilizing algorithm for graph searching in trees. In: Guerraoui, R., Petit, F. (eds.) SSS 2009. LNCS, vol. 5873, pp. 563–577. Springer, Heidelberg (2009)
9. Peng, S., Ho, C., Hsu, T., Ko, M., Tang, C.: Edge and node searching problems on trees. Theor. Comput. Sci. 240, 429–446 (2000)
10. Skodinis, K.: Construction of linear tree-layouts which are optimal with respect to vertex separation in linear time. J. Algorithms 47, 40–59 (2003)