Available online at www.sciencedirect.com

SCIENCE DIRECT®

Discrete Applied Mathematics 141 (2004) 243–253

DISCRETE APPLIED MATHEMATICS

www.elsevier.com/locate/dam

A distributed algorithm to find k-dominating sets

Lucia D. Penso[a,1], Valmir C. Barbosa[a,*]

[a] *Programa de Engenharia de Sistemas e Computação, COPPE, Universidade Federal do Rio de Janeiro, Caixa Postal 68511, 21941-972, Rio de Janeiro, RJ, Brazil*

Received 30 July 2001; received in revised form 16 May 2002; accepted 22 March 2003

Abstract

We consider a connected undirected graph $G(n,m)$ with n nodes and m edges. A k-dominating set D in G is a set of nodes having the property that every node in G is at most k edges away from at least one node in D. Finding a k-dominating set of minimum size is NP-hard. We give a new synchronous distributed algorithm to find a k-dominating set in G of size no greater than $\lfloor n/(k+1)\rfloor$. Our algorithm requires $O(k \log^* n)$ time and $O(m \log k + n \log k \log^* n)$ messages to run. It has the same time complexity as the best currently known algorithm, but improves on that algorithm's message complexity and is, in addition, conceptually simpler.

Keywords: k-Dominating sets; Distributed algorithms; Graph algorithms

1. Introduction

Let $G(n,m)$ be a connected undirected graph with n nodes and m edges. For $k \leqslant n-1$, a *k-dominating set* D in G is a set of nodes with the property that every node in G is at most k edges away from at least one of the nodes of D. The problem of finding k-dominating sets of relatively small sizes is important in a variety of contexts, including multicast systems [11], the placement of servers in a computer network [2], the caching of replicas in database and operating systems [8], and message routing with sparse tables [9].

Finding a k-dominating set in G with the least possible number of nodes is an NP-hard problem [4], so one normally settles for a set of small size that is not

* Corresponding author.
E-mail addresses: lucia@cs.brown.edu (L.D. Penso), valmir@cos.ufrj.br (V.C. Barbosa).
[1] Currently at the Computer Science Department, Brown University, Providence, RI 02912, USA.

0166-218X/$ - see front matter
doi:10.1016/S0166-218X(03)00368-8

necessarily optimal. In general, the small size to be sought is at most $\lfloor n/(k+1)\rfloor$, since it can be argued that a k-dominating set with no more than this number of nodes always exists [6], and likewise that a connected graph on n nodes necessarily exists for which every k-dominating set has at least $\lfloor n/(k+1)\rfloor$ nodes [10].

The argument for $\lfloor n/(k+1)\rfloor$ as an upper bound is instructive in the present context, and goes as follows [6]. Let T be a rooted spanning tree of G and $D_1,\ldots,D_{k+1}$ a partition of its nodes such that, for $0 \leqslant \ell \leqslant k$, every node in $D_{\ell+1}$ is away from the root a number x_ℓ of tree edges such that $x_\ell \bmod (k+1) = \ell$. This partition can be constructed easily by traversing T breadth-first from the root and assigning every new layer of nodes circularly to the sets $D_1,\ldots,D_{k+1}$. Clearly, every one of these sets is a k-dominating set in G. Also, because they partition the graph's node set, and considering that $n \geqslant k+1$, it must be that at least one of them has no more than $\lfloor n/(k+1)\rfloor$ nodes.

Our topic in this paper is finding a k-dominating set in G having no more than $\lfloor n/(k+1)\rfloor$ nodes by means of a synchronous distributed computation on G. The model of distributed computation that we adopt is the standard fully synchronous model [1]. In this model, the nodes of G are processors that function in lockstep at the occurrence of clock pulses, and its edges are bidirectional communication channels that deliver messages between their end nodes before the clock pulse that follows the sending of the message occurs. Time is measured by counting clock pulses.

The current best synchronous algorithm to find a k-dominating set in G is from [6], and is henceforth referred to as Algorithm KP. It proceeds in two stages: the first stage partitions G into the trees of a rooted spanning forest F, each having at least $k+1$ nodes and height $O(k)$; the second stage approaches each tree $U \in F$ as described earlier for the spanning tree T and partitions its nodes into the sets $D_1^U,\ldots,D_{k+1}^U$. If f is the number of trees in F, then the k-dominating set output by the algorithm is $D = D_{\ell_1}^{U_1} \cup \cdots \cup D_{\ell_f}^{U_f}$, where $U_1,\ldots,U_f$ are the trees of F and $D_{\ell_i}^{U_i}$ is the smallest set of $D_1^{U_i},\ldots,D_{k+1}^{U_i}$ for $1 \leqslant i \leqslant f$. If n_U is the number of nodes of $U \in F$, then

$$
\begin{aligned}
|D| &= |D_{\ell_1}^{U_1}| + \cdots + |D_{\ell_f}^{U_f}| \\
&\leqslant \sum_{U\in F} \left\lfloor \frac{n_U}{k+1} \right\rfloor \\
&\leqslant \left\lfloor \frac{\sum_{U\in F} n_U}{k+1} \right\rfloor \\
&= \left\lfloor \frac{n}{k+1} \right\rfloor,
\end{aligned}
$$

since $n_U \geqslant k+1$ for all $U \in F$.

While the second stage of Algorithm KP can be easily implemented within the bounds of $O(k)$ time and $O(n)$ messages, its first stage is based on an arcane combination of previously developed algorithms for related problems [3,5,7], resulting in a time complexity of $O(k \log^* n)$ and a message complexity of $O(m \log k + n^2 \log^* n)$.

The latter, incidentally, is our best estimate of what is really involved, in terms of communication needs, in Algorithm KP—such needs are thoroughly ignored in [6], but this message complexity seems to follow from the message complexities of the algorithm's building blocks.

In this paper, we introduce a new synchronous distributed algorithm for finding a k-dominating set of no more than $\lfloor n/(k+1) \rfloor$ nodes in G. Like Algorithm KP, our algorithm too comprises two subsequent stages, each having the same goal as its counterpart in Algorithm KP. The second stage, in particular, is exactly the same as Algorithm KP's.

Our contribution is the introduction of a new algorithm for the partition of G into the trees of F. When compared to Algorithm KP, our algorithm has the same complexity of $O(k \log^* n)$ time while improving on the message complexity, which in our case is of $O(m \log k + n \log k \log^* n)$. We also find our algorithm to be conceptually simpler than Algorithm KP, which can probably be attributed to the fact that it was designed from scratch with the partitioning problem in mind. While our algorithm simply generates a sequence of "meta-graphs," the last of which has nodes that directly give the rooted trees of F, Algorithm KP reduces the partition problem to other related problems and then combines algorithms for those problems into building a solution to the partition problem. Henceforth, we let the algorithm we introduce be called Algorithm PB.

The following is how the remainder of the paper is organized. The first stage of Algorithm PB is introduced in Section 2 and analyzed for correctness and complexity in Section 3. Concluding remarks are given in Section 4.

2. The algorithm

In this section we introduce the first stage of Algorithm PB. This stage finds a rooted spanning forest F in G, each of whose trees has at least $k+1$ nodes and $O(k)$ height, and is referred to in the sequel as Partition_G.

Partition_G starts by letting the node set of G be the node set of a directed graph $\vec{G}_0$, and proceeds from there in $\lceil \log(k+1) \rceil$ phases. For $0 \leqslant i \leqslant \lceil \log(k+1) \rceil - 1$, phase i first builds the edge set of $\vec{G}_i$ and then begins the transformation of $\vec{G}_i$ into another directed graph, $\vec{G}_{i+1}$, by clustering the nodes of $\vec{G}_i$ together to form the nodes of $\vec{G}_{i+1}$. Each node of $\vec{G}_i$ stands for a rooted tree in G, and this clustering is performed in such a way that not only is each resulting node of $\vec{G}_{i+1}$ also a rooted tree in G, but one that has at least 2^{i+1} nodes and $O(2^{i+1})$ height. After completion of phase $\lceil \log(k+1) \rceil - 1$, the node set of $\vec{G}_{\lceil \log(k+1) \rceil}$ represents the desired rooted spanning forest F (earlier termination is also possible, as we discuss shortly).

If x and y are nodes of $\vec{G}_i$, then we say that x and y are *potential neighbors* in $\vec{G}_i$ if an edge exists in G joining some node in the rooted tree represented by x to some node in the rooted tree represented by y. We say that they are *neighbors* in $\vec{G}_i$ if a directed edge exists between them. A node that has no neighbors is *isolated*. If x and y are neighbors in $\vec{G}_i$, then we use $x \to y$ to indicate that the edge between x and y is directed from x to y. In this case, we say that x is an *upstream* neighbor of y, which in turn is a *downstream* neighbor of x.

Partition_G is based on manipulations of node *identifiers*, which we assume to be a distinct nonnegative integer for every node in G. The identifier of node x in $\vec{G}_i$, denoted by $id(x)$, is the identifier of the root of its tree. If a node's identifier is less than those of all its neighbors, then we call the node a *local minimum*. If it is greater, then we call it a *local maximum*. The following is how Partition_G works during phase i. We use $\log^{(t)} n$ to denote $\log \cdots \log n$, where log is repeated t times.

Step 1 (*Find the edges of* $\vec{G}_i$):

(a) Let each node of $\vec{G}_i$ be *inactive*, if the height of the corresponding rooted tree is at least 2^{i+1}, or *active*, otherwise.
(b) For every active node x of $\vec{G}_i$, find the potential neighbors of x in $\vec{G}_i$. If no potential neighbors are found for any node, then halt and exit Partition_G.
(c) For each active node x of $\vec{G}_i$, let y be the active potential neighbor of x with the least identifier. If x has no active potential neighbors, then let y be the (inactive) potential neighbor of x having the least identifier. Add $x \to y$ to the edge set of $\vec{G}_i$, thus making x and y neighbors in $\vec{G}_i$.

Remark. If no neighbors are found for any node in Step 1(b), then in reality $\vec{G}_i$ has one single node that encompasses all the nodes of G and therefore corresponds to a rooted spanning tree of G. In this case, Partition_G terminates prematurely, that is, before completing all $\lceil \log(k+1) \rceil$ phases.

Remark. At the end of Step 1(c), every active node of $\vec{G}_i$ has exactly one downstream neighbor, while every inactive node has none. Similarly, both active nodes whose downstream neighbor is active and inactive nodes may have between zero and some positive number of upstream neighbors. An active node whose downstream neighbor is inactive has no upstream neighbors.

Step 2 (*Find the nodes of* $\vec{G}_{i+1}$):

(a) If $x \to y$ is an edge of $\vec{G}_i$ such that x is an active node and y an inactive node, then combine x into y by creating a single node whose identifier remains $id(y)$. Let X be the set of active nodes of $\vec{G}_i$ that are not isolated.
(b) For $x \in X$, let $Z(x) \subseteq X$ be the set of upstream neighbors of x. If $Z(x) \neq \emptyset$, then let z be the member of $Z(x)$ having the least identifier. For $y \in Z(x)$ such that $y \neq z$, check whether $Z(y) = \emptyset$. In the affirmative case, combine y into x. Otherwise (i.e., $Z(y) \neq \emptyset$), eliminate edge $y \to x$. Let X be the set of active nodes of $\vec{G}_i$ that are not isolated.
(c) For $x \in X$, if x is a local minimum, then combine its (at most two) neighbors into it and make it isolated by eliminating edges from $\vec{G}_i$ appropriately. Also, combine into the newly formed node any node in X that may have become isolated. Let X be the set of active nodes of $\vec{G}_i$ that are not isolated.
(d) Repeat Step 2(c) for local maxima, then let X be the set of active nodes of $\vec{G}_i$ that are not isolated. For $x \in X$, let $l_x = id(x)$.

(e) For $x \in X$, let

$$l_x^- = \begin{cases} l_y & \text{if } y \to x \text{ is an edge of } \vec{G}_i; \\ l_x - 1 & \text{if } y \to x \text{ is not an edge of } \vec{G}_i \text{ and } l_z > l_x; \\ l_x + 1 & \text{if } y \to x \text{ is not an edge of } \vec{G}_i \text{ and } l_z < l_x, \end{cases}$$

where z is the downstream neighbor of x, and

$$l_x^+ = \begin{cases} l_y & \text{if } x \to y \text{ is an edge of } \vec{G}_i; \\ l_x + 1 & \text{if } x \to y \text{ is not an edge of } \vec{G}_i \text{ and } l_z < l_x; \\ l_x - 1 & \text{if } x \to y \text{ is not an edge of } \vec{G}_i \text{ and } l_z > l_x, \end{cases}$$

where z is the upstream neighbor of x. Now consider the binary representations of l_x^-, l_x, and l_x^+, and let $A(x)$ be the set of positive integers p such that l_x^- and l_x have the same bit at the pth position while l_x and l_x^+ do not. Likewise, let $B(x)$ be the set of numbers p such that l_x^- and l_x have different bits at the pth position while l_x and l_x^+ have the same bit. Assuming that position numbers increase from right to left in a binary representation, let $p^*(x)$ be the greatest member of $A(x) \cup B(x)$. If $x \to y$ is an edge of $\vec{G}_i$ such that $p^*(x) = p^*(y)$, then combine x into y and make the resulting node isolated by eliminating edges appropriately (if any node in X becomes isolated because of this, then combine that node into the newly formed node as well). Now let X be the set of active nodes of $\vec{G}_i$ that are not isolated, then repeat Steps 2(c) and (d) with p^*'s in place of id's, and once again let X be the set of active nodes of $\vec{G}_i$ that are not isolated. If $X \neq \emptyset$, then let $l_x = p^*(x)$ for all $x \in X$ and repeat Step 2(e). If $X = \emptyset$, then let the set of isolated nodes of $\vec{G}_i$ be the node set of $\vec{G}_{i+1}$.

Remark. In Step 2(a), it is possible for more than one x to exist for the same y. In this case, every such x is combined into the single resulting node of identifier $id(y)$. Note that for no such x may there exist a node z such that $z \to x$ or $y \to z$ is an edge of $\vec{G}_i$. This is so, respectively, because x has an inactive downstream neighbor and by Step 1(c) has no active neighbors, and because y, being inactive, has no downstream neighbors. As a consequence, the newly formed node is isolated in $\vec{G}_i$. At the end of Step 2(a), the single downstream neighbor of every member of X is active, and therefore also a member of X.

Remark. In Step 2(b), there may exist more than one $y \in Z(x)$ such that $y \neq z$ and $Z(y) = \emptyset$. Every such y gets combined into node x. At the end of Step 2(b), every member of X has exactly one downstream neighbor and at most one upstream neighbor. That is, the members of X are arranged into groups of nodes, each group having at most one node with no upstream neighbor and exactly one node that has the same neighbor for both upstream and downstream neighbor. Except for these two-node directed cycles, such groups of nodes may be regarded as directed chains.

Remark. At the end of Step 2(d), the members of X are arranged into directed chains of nodes whose identifiers are strictly increasing or decreasing along the chains. Each

such chain has at least two nodes, of which exactly one has no upstream neighbor and exactly one has no downstream neighbor.

Remark. Step 2(e) repeatedly manipulates the node labels l_x so that the finding of minima and maxima, respectively as in Steps 2(c) and (d), can once again be used to combine nodes in X into isolated nodes. Initially, node identifiers are used for labels, but subsequently they get replaced by integers that point into the binary representations of the labels used in the previous iteration. As the iterations progress, these integers have an ever-dwindling range: if $j \geqslant 1$ identifies an iteration within Step 2(e), then the range of labels during iteration j is $0, \ldots, \log^{(j)} n$. Eventually, during a certain iteration $j \leqslant \log^* n$, this range becomes $\{0, 1\}$ and consequently the taking of minima and maxima as in Steps 2(c) and (d) is guaranteed to produce an empty X. At the beginning of each iteration, the members of X are arranged into directed chains whose labels are strictly increasing or decreasing along the chains. Each such chain has at least two nodes, of which exactly one has no upstream neighbor and exactly one has no downstream neighbor.

Steps 1 and 2 specify the ith phase of Partition_G as the manipulation of directed graphs, first to find the edges of $\vec{G}_i$ in Step 1, then in Step 2 to find the nodes of $\vec{G}_{i+1}$. Of course, the realization of such operations on graphs requires communication among the nodes of $\vec{G}_i$, which ultimately translates into communication among the nodes of G. However, the assumption of a synchronous model of distributed computation makes the communication needs of Partition_G rather straightforward to realize [10].

Because each node in $\vec{G}_i$ stands for a rooted tree in G, Steps 1 and 2 can be regarded as being executed by the trees' roots, which in turn coordinate the remaining nodes in their trees in carrying out the various tasks prescribed by the algorithm. For example, Step 1(a) is a broadcast with feedback on tree edges started by the root, which sends out the upper bound of $2^{i+1} - 1$ on the tree height for the ith phase. This is propagated by the other nodes in the tree, which send on what they receive, if nonzero, after decrementing it by one. The feedback is started by the leaves, which clearly never happens if at least one leaf is not reached by the broadcast, thus signaling to the root that the tree is oversized.

In the same vein, by simply letting every node in G that belongs to the same node x in $\vec{G}_i$ have a record of $id(x)$, finding the potential neighbors of x in Step 1(b) and the directed edges incident to it in $\vec{G}_i$ in Step 1(c) are also simple procedures that function by probing the connections of x in G. Whenever an edge is deployed between two nodes in $\vec{G}_i$, a corresponding edge in G, referred to as the *preferred edge* between those two nodes, can also be easily identified and recorded for later use.

All the remaining actions in Partition_G can be realized via communication between the roots of two trees whose nodes in $\vec{G}_i$ are connected by an edge. Whenever a message needs to be sent, it can be routed on tree edges only, except to move from one tree to the other, at which time it must go through the preferred edge between the two trees. This is, for example, the basis for realizing the combination of a node into another: combining a node x into a node y that is connected to it by an edge in $\vec{G}_i$ involves making the preferred edge between them an edge of the new tree and then

propagating through x's tree the information that a new root exists and has identifier $id(y)$.

3. Correctness and complexity

Most of our correctness and complexity arguments hinge on how well Step 2(e) succeeds in breaking directed chains of nodes in $\vec{G}_i$ as needed. It is to the properties of Step 2(e) that we turn first.

Lemma 1. *Let $x \to y$ be an edge at the beginning of an iteration of Step 2(e) of Partition_G. The following holds*:

(i) *If* $p^*(x) \in A(x)$ *and* $p^*(y) \in A(y)$, *then* $p^*(x) \neq p^*(y)$;
(ii) *If* $p^*(x) \in B(x)$ *and* $p^*(y) \in B(y)$, *then* $p^*(x) \neq p^*(y)$;
(iii) *If* $p^*(x) \in B(x)$ *and* $p^*(y) \in A(y)$, *then* $p^*(x) \neq p^*(y)$.

Proof. By Steps 2(a) through (e), edge $x \to y$ is in a chain of nodes whose labels are either strictly increasing or strictly decreasing along the chain. Suppose the former case first. Then $l_x^- < l_x < l_y < l_y^+$.

If $p^*(x) \in A(x)$, then l_x^- and l_x have the same bit at position $p^*(x)$ while l_x and l_y have different bits at that same position. If $p^*(y) \in A(y)$, then l_x and l_y have the same bit at position $p^*(y)$ while l_y and l_y^+ have different bits at that same position. So $p^*(x)$ and $p^*(y)$ cannot be the same position, thus proving (i).

If $p^*(x) \in B(x)$, then l_x^- and l_x have different bits at position $p^*(x)$ while l_x and l_y have the same bit at that same position. If $p^*(y) \in B(y)$, then l_x and l_y have different bits at position $p^*(y)$ while l_y and l_y^+ have the same bit at that same position. So $p^*(x)$ and $p^*(y)$ cannot be the same position, which proves (ii).

We now prove (iii). If $p^*(x) \in B(x)$, then l_x^- and l_x have different bits at position $p^*(x)$ while l_x and l_y have the same bit at that same position. Suppose these bits are 100, respectively, for l_x^-, l_x, and l_y. By definition of $p^*(x)$, at all other positions to the left of $p^*(x)$ in the binary representations of l_x^-, l_x, l_y (that is, positions corresponding to higher powers of two) we must have the same bit for all three labels or bits that differ from l_x^- to l_x and also from l_x to l_y. In other words, the only possibilities are 000, 111, 010, and 101. But these possibilities have all the same bit for l_x^- and l_y, which contradicts the fact that $l_x^- < l_y$. Then the bits for l_x^-, l_x, and l_y at position $p^*(x)$ must be 011.

If $p^*(x) = p^*(y)$, then the bits of l_x and l_y are both 1 at position $p^*(y)$, which is in agreement with the definition of $A(y)$. By this same definition, at position $p^*(y)$ the bit of l_y^+ must be 0. To the left of $p^*(y)$, the possibilities for l_x, l_y, l_y^+ are 000, 111, 010, and 101, again following the definition of $p^*(y)$. This implies the same bit for l_x and l_y^+ at all those positions, which like before contradicts the fact that $l_x < l_y^+$. So $p^*(x) \neq p^*(y)$.

If $x \to y$ is in a chain of nodes whose labels are strictly decreasing along the chain, then $l_x^- > l_x > l_y > l_y^+$. In this case, the arguments that prove (i) and (ii) remain unchanged, while in the proof of (iii) it suffices to complement every bit in the triples we displayed (this leads to contradictions of $l_x^- > l_y$ and $l_x > l_y^+$). □

Lemma 2. *Let $x \to y$ be an edge at the beginning of an iteration of Step* 2(e) *of Partition_G. If $p^*(x) = p^*(y)$, then $p^*(x) \in A(x)$ and $p^*(y) \in B(y)$.*

Proof. If $p^*(x) \in B(x)$ with either $p^*(y) \in B(y)$ or $p^*(y) \in A(y)$, then by Lemma 1, parts (ii) and (iii), $p^*(x) \neq p^*(y)$. If $p^*(x) \in A(x)$ and $p^*(y) \in A(y)$, then by Lemma 1, part (i), $p^*(x) \neq p^*(y)$. Thence the lemma. □

Lemma 3. *Let $x \to y \to z$ be part of a chain at the beginning of an iteration of Step* 2(e) *of Partition_G. If $p^*(x) = p^*(y)$, then $p^*(y) \neq p^*(z)$.*

Proof. By Lemma 2, $p^*(x) \in A(x)$ and $p^*(y) \in B(y)$. By Lemma 1, parts (ii) and (iii), $p^*(y) \neq p^*(z)$. □

Now let I be the last phase of Partition_G in which premature termination in Step 1(b) does not occur. Then $1 \leqslant I \leqslant \lceil \log(k+1) \rceil - 1$ and we have the following.

Lemma 4. *For $i = 0, \ldots, I$, every node of $\vec{G}_{i+1}$ that is not an inactive node of $\vec{G}_i$ is formed by the combination of at least two nodes of $\vec{G}_i$.*

Proof. By Step 2(e), the node set of $\vec{G}_{i+1}$ is the set of isolated nodes in $\vec{G}_i$ at the end of Step 2. The lemma follows easily from the fact that every isolated node produced during Step 2 (that is, isolated nodes that are not inactive during phase i) result from the combination of at least two nodes of $\vec{G}_i$. □

We are now in position to demonstrate that Partition_G does indeed achieve its goals.

Theorem 5. *For $i = 0, \ldots, I+1$, the nodes of $\vec{G}_i$ form a rooted spanning forest of G. Each tree in this forest has at least 2^i nodes and $O(2^i)$ height.*

Proof. The theorem holds trivially for $i = 0$, and we prove it inductively for $i + 1$ with $0 \leqslant i \leqslant I$. The induction hypothesis is that the nodes of $\vec{G}_i$ form a rooted spanning forest of G, each of whose trees having at least 2^i nodes and $O(2^i)$ height.

In order to show that the nodes of $\vec{G}_{i+1}$ do indeed form a rooted spanning forest of G, by the induction hypothesis it suffices to argue that the set X is empty at the end of phase i. The reason for this is that it is the set of isolated nodes at the end of phase i that we take to be the node set of $\vec{G}_{i+1}$, and that $X = \emptyset$ indicates that every node in G is part of the tree represented by some isolated node. But this follows directly from the fact that the range of labels during Step 2(e) decreases steadily as the iterations progress. Eventually, this range becomes such that every label is either 0 or 1, at which time the finding of minima and maxima makes X empty.

Having shown this, we consider the number of nodes and height of each of the trees in the node set of $\vec{G}_{i+1}$. A node of $\vec{G}_{i+1}$ is either an inactive node of $\vec{G}_i$ or results, by Lemma 4, from the combination of at least two nodes of $\vec{G}_i$. In the former case, by Step 1(a) the node corresponds to a rooted tree in G with at least 2^{i+1} nodes. In

the latter case, by the induction hypothesis, it corresponds to a rooted tree in G with at least $q2^i$ nodes for $q \geqslant 2$, that is, at least 2^{i+1} nodes.

As for the height, we reason similarly. If a node in $\vec{G}_{i+1}$ is an inactive node of $\vec{G}_i$, then its height is, by the induction hypothesis, $O(2^i)$, which in turn is $O(2^{i+1})$. If it is a combination of at least two nodes of $\vec{G}_i$, then either this combination takes place in one of Steps 2(a) through (d) or in Step 2(e). In the former case, the combination is either performed over a single edge (Steps 2(a) and (b)), or it is performed over a chain of at most six edges (first in Step 2(c), then in Step 2(d)). In either case, the induction hypothesis leads to a height of $q2^i$ for q a constant, which is $O(2^{i+1})$. The case of Step 2(e) is entirely analogous, since by Lemma 3 it is either performed over a single edge, or else by the taking of minima and maxima, as in Steps 2(c) and (d). □

Corollary 6. *The nodes of $\vec{G}_{I+1}$ form a rooted spanning forest of G, and in this forest each tree has at least $k+1$ nodes and $O(k)$ height.*

Proof. If Partition_G terminates in Step 1(b) of some phase, then $I < \lceil \log(k+1) \rceil - 1$ and the corollary holds, because $\vec{G}$ has in this case one single node encompassing all the $n \geqslant k+1$ nodes of G, and furthermore the height of the tree that corresponds to this single node is by Theorem 5 $O(2^{I+1})$, which is $O(k)$. If Partition_G runs through all the phases, then $I = \lceil \log(k+1) \rceil - 1$ and the corollary follows directly from Theorem 5 with $i = I + 1 = \lceil \log(k+1) \rceil$. □

We now finalize the section by discussing the time and number of messages needed by Partition_G and by Algorithm PB as a whole.

Theorem 7. *Partition_G requires $O(k \log^* n)$ time and $O(m \log k + n \log k \log^* n)$ messages to complete.*

Proof. During the ith phase, $i = 0, \ldots, \lceil \log(k+1) \rceil - 1$, each of Steps 1(a) through 2(d) requires a constant number of communication "rounds," each in turn requiring a number of time units proportional to the height of a rooted tree in that phase, which by Theorem 5 is $O(2^i)$. The same holds for each of the iterations of Step 2(e), of which there are at most $\log^* n$. Then the time required for Partition_G to complete grows with

$$\begin{aligned}
\sum_{i=0}^{\lceil \log(k+1) \rceil - 1} 2^i \log^* n &\leqslant \sum_{i=0}^{\lceil \log(k+1) \rceil - 1} \frac{2^{\log(k+1)}}{2^i} \log^* n \\
&= (k+1) \log^* n \sum_{i=0}^{\lceil \log(k+1) \rceil - 1} \frac{1}{2^i} \\
&< 2(k+1) \log^* n,
\end{aligned}$$

so Partition_G requires $O(k \log^* n)$ time.

The number of messages that Partition_G requires can be estimated likewise for phase i, as follows. The number of messages sent during Step 1 is dominated by Step 1(b) to determine the potential neighbors in G of a node in $\vec{G}_i$, which requires $O(m)$ messages. Steps 2(a) and (b) require $O(n)$ messages, which is the total number of tree edges, because several nodes may be combined into the same node during those steps. However, each of the $O(\log^* n)$ communication "rounds" in Steps 2(c) through (e) is more economical, because the chain structure of $\vec{G}_i$ in those steps allows communication to take place along single paths from the trees' roots, and so the number of messages flowing in each rooted tree is proportional to its height, which during phase i is $O(2^i)$ by Theorem 5. Also by Theorem 5, each rooted tree in $\vec{G}_i$ has at least 2^i nodes, so there are at most $n/2^i$ rooted trees in $\vec{G}_i$. It follows that the number of messages required by Partition_G for completion grows with

$$\sum_{i=0}^{\lceil \log(k+1) \rceil - 1} m + \frac{n}{2^i} 2^i \log^* n = m\lceil \log(k+1) \rceil + n \log^* n \lceil \log(k+1) \rceil,$$

so Partition_G requires $O(m \log k + n \log k \log^* n)$ messages. □

Corollary 8. *Algorithm PB requires* $O(k \log^* n)$ *time and* $O(m \log k + n \log k \log^* n)$ *messages to complete.*

Proof. Immediate from Theorem 7, considering that the algorithm's second stage requires $O(k)$ time and $O(n)$ messages. □

4. Concluding remarks

We have considered the problem of finding a k-dominating set with no more than $\lfloor n/(k+1) \rfloor$ nodes in G, and given a new synchronous distributed algorithm to solve it in $O(k \log^* n)$ time while requiring $O(m \log k + n \log k \log^* n)$ messages. Our algorithm follows the same overall strategy of [6], according to which first a rooted spanning forest is found in G with certain characteristics, and then the desired k-dominating set on that forest.

Our algorithm introduces a new approach to finding the rooted spanning forest, which we think is conceptually simpler than the one of [6], and shares with the algorithm of [6] the additional computation that is required to find the k-dominating set. In both algorithms, the overall complexity is dominated by the forest-finding stage. Both have the same time complexity, but ours has better message complexity.

Acknowledgements

The authors acknowledge partial support from CNPq, CAPES, the PRONEX initiative of Brazil's MCT under contract 41.96.0857.00, and a FAPERJ BBP grant.

References

[1] V.C. Barbosa, An Introduction to Distributed Algorithms, The MIT Press, Cambridge, MA, 1996.
[2] J. Bar-Ilan, G. Kortsarz, D. Peleg, How to allocate network centers, J. Algorithms 15 (1993) 385–415.
[3] J.A. Garay, S. Kutten, D. Peleg, A sub-linear time distributed algorithm for minimum-weight spanning trees, SIAM J. Comput. 27 (1998) 302–316.
[4] M.R. Garey, D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W.H. Freeman & Co., New York, NY, 1979.
[5] A.V. Goldberg, S.A. Plotkin, G.E. Shannon, Parallel symmetry-breaking in sparse graphs, in: Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing, New York, NY, 1987, pp. 315–324.
[6] S. Kutten, D. Peleg, Fast distributed construction of small k-dominating sets and applications, J. Algorithms 28 (1998) 40–66.
[7] A. Panconesi, A. Srinivasan, Improved distributed algorithms for coloring and network decomposition problems, in: Proceedings of the Twenty-Fourth Annual ACM Symposium on Theory of Computing, New York, NY, 1992, pp. 581–592.
[8] D. Peleg, Distributed data structures: a complexity-oriented view, in: Proceedings of the Fourth International Workshop on Distributed Algorithms, 1991, pp. 71–89.
[9] D. Peleg, E. Upfal, A tradeoff between size and efficiency for routing tables, J. ACM 36 (1989) 510–530.
[10] L.D. Penso, A distributed algorithm to find k-dominating sets in graphs, Master's Thesis, Federal University of Rio de Janeiro, December 1999 (in Portuguese).
[11] R. Wittmann, M. Zitterbart, Multicast Communication: Protocols and Applications, Morgan Kaufmann, San Francisco, CA, 2001.