5-2007

A LINEAR-TIME ALGORITHM FOR BROADCAST DOMINATION IN A TREE

John Dabney
Clemson University, jdabney@clemson.edu

A LINEAR-TIME ALGORITHM FOR BROADCAST DOMINATION IN A TREE

A Thesis
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science
Computer Science

---

by
John Randolph Dabney Jr.
May 2007

Accepted by:
Dr. Brian C. Dean, Committee Chair
Dr. Sandra M. Hedetniemi
Dr. Robert M. Geist III

ABSTRACT

The broadcast domination problem is a variant of the classical minimum dominating set problem in which a transmitter of power p at vertex v is capable of dominating (broadcasting to) all vertices within distance p from v. Our goal is to assign a broadcast power $f(v)$ to every vertex v in a graph such that $\sum_{v \in V} f(v)$ is minimized, and such that every vertex u with $f(u) = 0$ is within distance $f(v)$ of some vertex v with $f(v) > 0$. The problem is solvable in polynomial time on a general graph [15], and Blair et al. [6] gave an $O(n^2)$ algorithm for trees. In this paper, we provide an $O(n)$ algorithm for trees. Our algorithm is notable due to the fact that it makes decisions for each vertex v based on "non-local" information from vertices far away from v, whereas almost all other linear-time algorithms for trees only make use of local information.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ALGORITHMS

Chapter 1

Introduction

Many variants of domination and covering problems in graphs have been studied over the years, and as a rule of thumb, most of these tend to be NP-hard. A remarkable exception is the *broadcast domination problem* [11, 12], in which a transmitter of power p at some vertex v is capable of dominating (i.e., broadcasting to) any vertex within distance p from v. The goal of the problem is to assign a broadcast power $f(v)$ to every vertex v in a graph, at minimum total cost $\sum_{v \in V} f(v)$, so that every vertex u with $f(u) = 0$ is within distance $f(v)$ of some vertex v with $f(v) > 0$ (i.e., u can hear the broadcast of v). As shown by Heggernes and Lokshtanov [15], this problem is solvable in polynomial time on a general graph with n vertices. Their algorithm runs in $O(n^6)$ time, although much faster algorithms have been constructed for special types of graphs. Blair et al. [6] give an $O(n^3)$ algorithm for interval graphs[1], an $O(nr^4) = O(n^5)$ algorithm for series-parallel graphs with radius $r = rad(G)$, and an $O(nr) = O(n^2)$ algorithm for trees. One can also solve the minimum broadcast domination problem in a tree using a "primal-dual" method [13, 16, 18], although a straightforward implementation of this approach requires $O(n^3)$ time. In this paper, we focus on the broadcast domination problem in a tree, and devise an $O(n)$ algorithm.

The existence of a new linear-time algorithm for trees is typically not so interesting to the algorithmic community, since hundreds of such algorithms are known. The vast majority of these algorithms can even be generated in a mechanical fashion using general-purpose techniques for constructing dynamic programming algorithms on trees and graphs of bounded treewidth — for example, "table building" approaches [5, 19], or by express-

[1] A simple reformulation of the dynamic programming approach used in [6] improves the running time for interval graphs to $O(n^2)$.

ing a problem using monadic second order logic [1, 8, 9, 10]. The broadcast domination problem, however, seems to be one of the rare problems that does not fit into any of these methodologies, primarily because a dynamic programming algorithm for this problem must make decisions based on "non-local" information. Whereas for many problems (e.g., minimum dominating set, minimum vertex cover, minimum maximal irredundant set, etc.) one can solve subproblems for a particular vertex v based solely on subproblem solutions for v's children or immediate descendants, the broadcast domination problem requires information from much larger distances in order to determine the optimal amount of power to allocate to a transmitter at vertex v. Accordingly, in order to achieve a linear running time, our algorithm must incorporate several novel and sophisticated techniques.

Chapter 2

Preliminaries

Consider a tree $T = (V, E)$ with $n = |V|$ vertices. Let $dist(u, v)$ denote the distance between vertices u and v in T, let $diam(T) = \max\{dist(u, v) : u, v \in V\}$ be the diameter of T, and let $B(v, r) = \{u : dist(u, v) \leq r\}$ denote the ball of radius r around v. A function $f : V \to \{0, 1, 2, \ldots, diam(T)\}$ is called a *broadcast*, where a vertex v with $f(v) > 0$ is interpreted as a transmitter of power $f(v)$, whose transmission reaches every vertex in $B(v, f(v))$. A broadcast f is *dominating* if every vertex u with $f(u) = 0$ hears the broadcast of at least one other vertex, and it is *efficient* if every vertex u with $f(u) = 0$ hears the broadcast from exactly one other vertex. For an efficient dominating broadcast f, we define its *ball graph* $B(f)$ as the graph obtained by contracting the vertices in every ball $B(v, f(v))$ for $f(v) > 0$ down to a single vertex.

The *minimum-cost broadcast domination problem* involves finding a dominating broadcast f that minimizes $cost(f) = \sum_{v \in V} f(v)$. One can certainly conceive of more elaborate cost functions to minimize as an objective. For example a more realistic cost for building a transmitter at v might have the form $c_{fixed} + c_{power}(f(v))$, where c_{fixed} is a fixed one-time cost and c_{power} is an increasing function of the broadcast power (both of which may conceivably depend on v). A prototypical application in this setting would be installing a minimum-cost collection of facilities in a graph, where each facility involves some fixed cost for its initial installation plus a variable cost that depends on the size of the facility (a larger facility being able to service a larger geographic area). Many other applications are conceivable; any problem where one wishes to cover a graph with variable-sized "ball-shaped" regions (larger regions being more expensive to install than smaller regions) can be modeled as a broadcast domination problem. For consistency with the existing literature, we focus in this paper on the simpler objective of minimizing $\sum_{v \in V} f(v)$. In our concluding remarks in Section 6, we discuss the extension of our results to more complicated cost

functions.

The broadcast domination problem was first introduced by Erwin [12] and subsequently studied by Horton et al. [17], Blair et al. [6], and Dunbar et al. [11] before its complexity was finally resolved in a breakthrough paper by Heggernes and Lokshtanov [15], who gave a polynomial time algorithm for broadcast domination on a general graph. This result is quite surprising since, anecdotally, most domination and covering problems on general graphs tend to be NP-hard. The algorithm of [15] depends highly on two important structural properties of optimal dominating broadcasts:

Lemma 1 [11] For any graph G, one can find an optimal dominating broadcast of G that is efficient.

Lemma 2 [15] For any graph G, one can find an optimal dominating broadcast f such that (i) f is efficient and (ii) every vertex in $B(f)$ has maximum degree 2.

Corollary 3 For any tree T, one can find an optimal dominating broadcast f such that (i) f is efficient, and (ii) $B(f)$ is a path.

A dominating broadcast that uses a single transmitter at the center of T of power $rad(T)$ is called a *radial* broadcast. It is an interesting open question to characterize the class of trees for which an optimal dominating broadcast is radial. Since one can easily compute an optimal radial broadcast in a tree in linear time, we henceforth assume that T has an optimal efficient dominating broadcast f in which $B(f)$ is a path with at least two vertices. Once we assign a root to T, this orients $B(f)$ so that we can speak of one ball in f being "above" or "below" another ball (i.e., ball B_1 is above ball B_2 if B_1 contains a vertex of lower depth than all the vertices in B_2). Every ball except the one containing the root will have exactly one neighboring ball above it and at most one below it.

The *overdomination* of vertex v in a non-zero broadcast f is defined as $dom_f(v) = \max\{f(u) - dist(u,v) : f(u) > 0\}$, and represents the excess power available at v from its strongest nearby transmitter. If $dom_f(v) > 0$, then we say v is *overdominated*. Note that f is dominating if and only if $dom_f(v) \geq 0$ for all v.

2.1 Precomputation on a Rooted Tree

Our algorithm will root T at two different vertices over the course of its operation, and each time T is rooted it spends $O(n)$ time performing some useful precomputation. We first compute (in linear time) the depth $d(v)$ and height $h(v)$ of every vertex v, where the depth of the root is zero and the height of a leaf is zero. For simplicity of notation, we write $d(v)$ and $h(v)$ rather than $d_r(v)$ and $h_r(v)$ where r is the root, with the understanding that $d(v)$ and $h(v)$ are relative to the current tree root (these quantities and any others dependent on the root are recomputed after the single root change).

Let $\pi(v)$ denote the parent of v and $\pi_k(v)$ denote the kth level ancestor of v — that is, the ancestor of v of depth $d(v) - k$. Using a data structure initially due to Berkman and Vishkin [4] and then substantially simplified by Bender and Farach-Colton [3], we can compute $\pi_k(v)$ in only $O(1)$ time for any k and v, after initially expending $O(n)$ preprocessing time and space. Let $LCA(u, v)$ denote the lowest common ancestor of u and v, the vertex of highest depth in the tree that is an ancestor of both u and v. Otherwise stated, if p_u and p_v are the respective paths from the root down to u and v, then $LCA(u, v)$ is the deepest vertex in common between p_u and p_v. Using a data structure initially due to Harel and Tarjan [14] and again substantially simplified by Bender and Farach-Colton [2], we can compute $LCA(u, v)$ in only $O(1)$ time for any u and v, after initially expending $O(n)$ preprocessing time and space.

As a further preprocessing step, we greedily partition T into vertex-disjoint paths as follows. Initially, in $O(n)$ processing time we compute for each vertex v a pointer to $deepest(v)$, a leaf of maximum depth in v's subtree. This is done recursively: $deepest(v) = v$ if v is a leaf, otherwise $deepest(v) = deepest(u)$ where u is a child of v with maximum height $h(u)$. Next, we take the rooted path from the root r down to $deepest(r)$ and remove the vertices and edges along this path from T. This potentially splits T into several disjoint rooted subtrees, which are themselves then decomposed into vertex-disjoint rooted paths in the same fashion. The entire process takes $O(n)$ time if implemented appropriately. For each vertex v, we denote by P_v the path in our decomposition containing v. In order to facilitate certain future operations along these paths, we store each path in an array, and we maintain a

pointer from each vertex v to the array P_v containing v. Note that v also knows its index within P_v, as this can be computed in constant time by taking the difference between $d(v)$ and the depth of root of P_v.

Finally, we compute for each vertex v its second height $h_2(v)$ and third height $h_3(v)$, where $h_2(v)$ is defined as the maximum value of $1 + h(u)$ over all of v's children $u \notin P_v$. Similarly, $h_3(v)$ is the maximum of $1 + h(u)$ over all v's children $u \notin P_v$, excluding the child used to define $h_2(v)$. In English, $h_2(v)$ is the length of a deepest path descending from v that branches off P_v at v, and $h_3(v)$ is the length of a second-deepest such path. In order to locate these paths efficiently, we maintain a pointer from v to $c_1(v)$ (the child of v on P_v), $c_2(v)$ (the child of v that defines $h_2(v)$), and $c_3(v)$ (the child of v that defines $h_3(v)$). We set $h_3(v) = 0$ if v has fewer than 3 children, and $h_2(v) = 0$ if v has only 1 child. Implemented appropriately, all of this information takes $O(n)$ time to compute.

Chapter 3

The High-Level Algorithm

In a tree T with root r, we say that a broadcast f is *proper* if (i) f is dominating and efficient, (ii) $B(f)$ is a path, (iii) the ball containing r is an endpoint of $B(f)$, and (iv) r is not overdominated. Let $cost_r[v]$ denote the optimal cost of a proper broadcast for just the subtree rooted at v. If such a proper broadcast does not exist (for example, if v is a leaf), we set $cost_r[v] = +\infty$. The bulk of our algorithm is focused on computing $cost_r[v]$ in $O(1)$ amortized time for each $v \in V$.

If f satisfies only conditions (i) and (ii) above, we say it is *semi-proper*. Due to Lemmas 1 and 2, every tree has an optimal broadcast that is semi-proper. Moreover, since we are assuming that T is optimally dominated by some efficient *non-radial* broadcast f^*, we can decompose f^* into a union of two proper sub-broadcasts as follows. We first compute a diameter D of T in linear time, whose endpoints we denote by a and b. Next, we root T at a and compute $cost_a[v]$ for every v in $O(n)$ time, and then we root T at b and compute $cost_b[v]$ for every v in $O(n)$ time. Since f^* is not radial, there must be some edge $uv \in D$ that crosses the boundary between two balls in f^*. By examining all edges on D, we are guaranteed to find such an edge, from which we can compute

$$cost(f^*) = \min_{\substack{uv \in D \\ dist(a,u) < dist(a,v)}} \{cost_a[v] + cost_b[u]\}. \tag{3.1}$$

Therefore, to compute $cost(f^*)$ in $O(n)$ time, all we need is an $O(n)$ procedure for computing $cost_r[v]$ for all vertices v in a rooted tree. By restricting our focus only to proper broadcasts, we simplify this task quite a bit. We can assume when computing $cost_r[v]$ that v is dominated by the broadcast emanating from a single transmitter t with $f(t) = dist(v, t)$, where $B(t, f(t))$ has at most one neighboring ball below it (whereas in a semi-proper solution we would need to have considered the added possibility of two neighboring balls

beneath $B(t, f(t))$). Let us now distinguish two important cases: we say that v is *externally dominated* by a transmitter t if $f(t) \geq h(t)$. In this case, we call t an *external* transmitter, since its transmission reaches all the way to the bottom of its own subtree. If v is not externally dominated, we say it is *internally dominated* by an *internal* transmitter t with $f(t) < h(t)$. We define $ecost_r[v]$ as the optimal cost of a proper broadcast that dominates only v's subtree, such that v is externally dominated, and we define $icost_r[v]$ similarly for the internal case (if no such broadcast exists, then we set $ecost_r[v] = +\infty$ or $icost_r[v] = +\infty$).

During a single postorder scan of T, we will compute both $ecost_r[v]$ and $icost_r[v]$ for every vertex v and then set $cost_r[v] = \min(ecost_r[v], icost_r[v])$. In the next two chapters, we show how to compute $ecost[v]$ and $icost[v]$ each in $O(1)$ amortized time per vertex (since the root is fixed for these computations, we will drop the subscript r from our notation).

Our algorithm can be construed as a dynamic programming algorithm, and as with most dynamic programming algorithms we focus our exposition on computing the cost of an optimal solution f^*, rather than the structure of this solution. If structure is desired, it is relatively easy to augment our algorithm to compute $f^*(v)$ for every vertex v.

Chapter 4

The External Case

We call a pair (t, p) where $f(t) = p$ a *configuration* that describes a potential external transmitter. Let $N(t, p) = \{v : dist(t, v) = p + 1\} \backslash \{\pi_{p+1}(t)\}$ denote the set of all neighboring vertices below the ball $B(t, p)$. We call the configuration (t, p) *valid* if and only if $p \in \{h(t), h(t) + 1\}$, $p \leq d(t)$, and $|N(t, p)| \leq 1$. Invalid configurations can be safely ignored for the external case. If $p < h(t)$ then t cannot be an external transmitter. If $p \geq h(t) + 2$, we could move t to $\pi(t)$ and decrease p by one to obtain a better dominating broadcast. If $p > d(t)$, then $B(t, p)$ would overdominate the root, so the resulting broadcast would not be proper. If $|N(t, p)| > 1$, then the ball $B(t, p)$ would have at least two neighboring balls below it, so the resulting broadcast would not be proper. In a moment, we will describe a method for testing whether a configuration (t, p) is valid in only $O(1)$ worst-case time.

The key idea behind the external case is now the following:

Lemma 4 If a configuration (t, p) is valid, then it is only relevant for the computation of $ecost[v]$ for the single vertex $v = \pi_p(t)$.

Proof. The vertex $v = \pi_p(t)$ is the only one satisfying the properties (i) $B(t, p)$ is contained in v's subtree, and (ii) $dist(v, t) = p$, so v is dominated but not overdominated by t. □

As a preprocessing step after rooting T, we spend $O(n)$ time iterating over all $2n$ transmitter configurations (t, p) with $p \in \{h(t), h(t) + 1\}$. For each such configuration that is valid, we add the pair (t, p) to a linked list L_v attached to $v = \pi_p(t)$. Afterwards, during the main algorithm we can compute $ecost[v]$ for any vertex v by minimizing over the possible choices in L_v:

$$ecost[v] = \min_{(t,p) \in L_v} \left\{ p + \sum_{u \in N(t,p)} cost[u] \right\}. \tag{4.1}$$

Note that $cost[u]$ will already have been computed by the time we compute $ecost[v]$, since

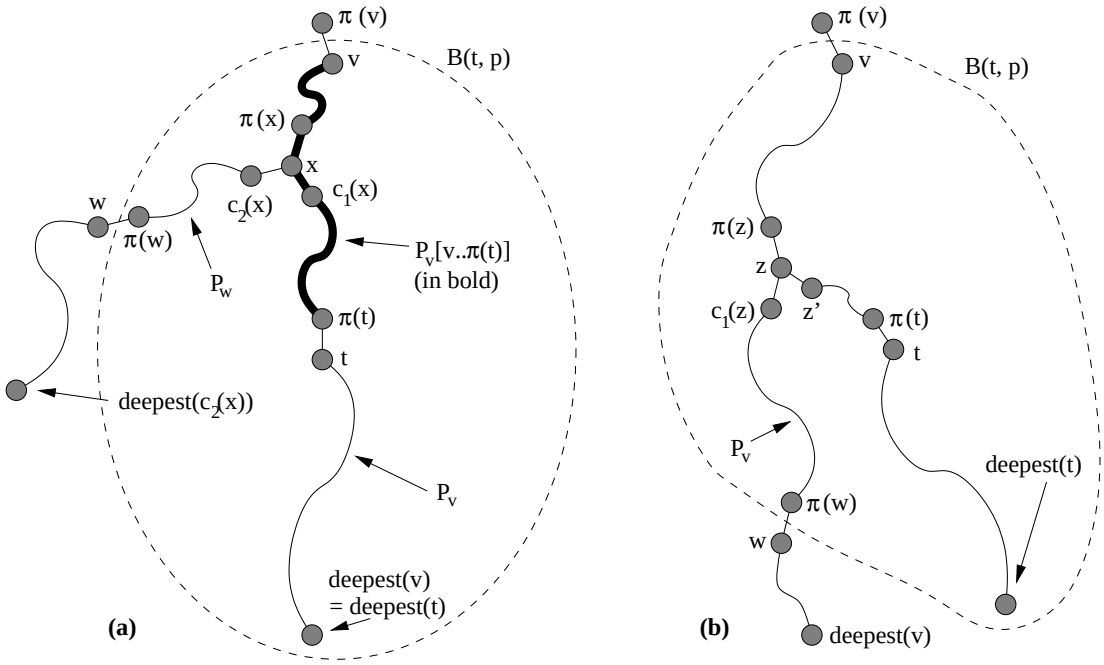

Figure 4.1: Searching for vertices in $N(t, p)$.

the sole vertex (if any) in $N(t, p)$ belongs to v's subtree. Since $|N(t, p)| \leq 1$, evaluating the formula above takes $O(|L_v|)$ time for a particular vertex v, and $O(n)$ time in total to compute $ecost[v]$ for every vertex v.

4.1 Checking a Configuration for Validity

Consider a particular configuration (t, p) with $p \in \{h(t), h(t) + 1\}$. In order to check (t, p) for validity, it is easy to check whether $p > d(t)$, so we assume that $p \leq d(t)$ and focus on the more challenging question of determining whether or not $|N(t, p)| \leq 1$. Let $v = \pi_p(t)$ denote the unique vertex for which (t, p) is relevant for computing $ecost[v]$. Since $p \geq h(t)$, the ball $B(t, p)$ engulfs the entire subtree rooted at t and so we can restrict our search for vertices in $N(t, p)$ to those vertices in v's subtree that are not in t's subtree. Consider now two cases:

1. $t \in P_v$. Recall that the deepest path descending from v, P_v, is stored in an array. Let $P_v[v \ldots \pi(t)]$ denote the segment of this array corresponding to the path from v down to $\pi(t)$. In order to simplify our exposition, we are abusing notation slightly by using

Check-Validity(t, p)**:**

1. If $p > d(t)$, Then Stop. (t, p) is invalid.
2. $v = \pi_p(t)$
3. If $t \in P_v$, Then
4. Find $x \in P_v[v \ldots \pi(t)]$ maximizing $\alpha_1(x)$ $(\alpha_1(x) = h_2(x) - d(y))$
5. If $\alpha_1(x) \leq p - d(t)$, Then Stop. (t, p) is valid. (since $N(t, p) = \emptyset$)
6. $k = \alpha_1(x) - d(t) - (p + 1)$; $w = \pi_k(deepest(c_2(x)))$
7. If $h_3(x) + d(t) - d(x) > p$
8. or $\max\{\alpha_1(y) : y \in P_v[v \ldots \pi(x)]\} > p - d(t)$
9. or $\max\{\alpha_1(y) : y \in P_v[c_1(x) \ldots \pi(t)]\} > p - d(t)$
10. or $\max\{\alpha_2(y) : y \in P_w[c_2(x) \ldots \pi(w)]\} > p - d(t) + 2d(x)$, $(\alpha_2(y) = h_2(y) + d(y))$
11. Then Stop. (t, p) is invalid. (since $|N(t, p)| > 1$)
12. (t, p) is valid, and $N(t, p) = \{w\}$.
13. Else
14. $k = h(t) + d(t) - d(z) - 1$; $z' = \pi_k(deepest(t))$
15. If $t \notin P_{z'}$, Then Stop. (t, p) is invalid. (since $|N(t, p)| > 1$)
16. If $(z' = c_2(z)$ And $h_3(z) + d(t) - d(z) > p)$
17. or $(z' \neq c_2(z)$ And $h_2(z) + d(t) - d(z) > p)$
18. or $\max\{\alpha_1(x) : x \in P_v[v \ldots \pi(z)]\} > p - d(t)$
19. or $\max\{\alpha_2(x) : x \in P_v[c_1(v) \ldots \pi(w)]\} > p - d(t) + 2d(z)$
20. or $\max\{\alpha_1(x) : x \in P_{v'}[v' \ldots \pi(t)]\} > p - d(t)$
21. Then Stop. (t, p) is invalid. (since $|N(t, p)| > 1$)
22. (t, p) is valid, and $N(t, p) = \{w\}$.
23. Endif

Algorithm 4.1: Checking if (t, p) is valid in $O(1)$ time.

vertices as array indices; in an actual implementation, we would need to figure the integer index corresponding to a vertex v based on $d(v)$ and the depth of the highest vertex on P_v. Any vertices in $N(t, p)$ can be located by checking for the presence of paths branching off $P_v[v \ldots \pi(t)]$ that are sufficiently long to escape from $B(t, p)$, as shown in Figure 4.1(a). We first try to locate one such path, branching off at a vertex $x \in P_v[v \ldots \pi(t)]$, and crossing out of $B(t, p)$ along the edge $(w, \pi(w))$, as shown in the figure. The vertex w will be our first entry in $N(t, p)$. If our search for w succeeds, then we launch another search for a second "escaping path" (not shown in the figure). If a second such path exists, then (t, p) is not valid.

In order to find x and w in $O(1)$ time, we employ a data structure for range maximum queries (RMQs). The RMQ problem involves preprocessing an array $A[1 \ldots n]$ so that subsequent queries for the maximum element in a subarray $A[i \ldots j]$ can be answered quickly. By building a Cartesian tree from A in linear time, we can translate

an instance of the RMQ problem into an equivalent instance of the LCA problem (see [2]), so by using fast LCA data structures [14, 2] one can answer RMQ queries in $O(1)$ time after initially expending $O(n)$ preprocessing time and space. In our case, we assign each vertex v the value $\alpha_1(v) = h_2(v) - d(v)$ and preprocess each of the arrays P_v into which we have decomposed our tree. This takes $O(n)$ total time since these arrays collectively have n elements. Returning to the problem of finding an $x \in P_v[v \ldots \pi(t)]$ from which a long path branches off, note that the second height $h_2(x)$ gives the length of the longest path branching off P_v at x. Therefore, all we need to do is locate a vertex $x \in P_v[v \ldots \pi(t)]$ for which $h_2(x) + d(t) - d(x) > p$. We can rewrite this condition as $\alpha_1(x) > p - d(t)$, so by performing an RMQ for the $x \in P_v[v \ldots \pi(t)]$ maximizing $\alpha_1(x)$ and comparing against $p - d(t)$, we can locate a suitable x (if it exists) in $O(1)$ time. If no such x exists, (t, p) is valid and we are finished. Otherwise, we can find w in $O(1)$ time, since $w = \pi_k(deepest(c_2(x)))$, where $k = \alpha_1(x) + d(t) - (p + 1)$. The final check for a second long branching path (that would render (t, p) invalid) requires 3 more RMQs for the subpaths $P_v[v \ldots \pi(x)]$, $P_v[c_1(x) \ldots \pi(t)]$, and $P_w[c_2(x) \ldots \pi(w)]$, in addition to a check based on $h_3(x)$ to see if such a path branches off x. Complete details are given in pseudocode in Algorithm 4.1.

2. $t \notin P_v$. As shown in Figure 4.1(b), t resides on a path that branches off P_v at the vertex $z = LCA(deepest(v), t)$. Note that $B(t, p)$ cannot dominate all the way down to $deepest(v)$, so the vertex $w \in N(t, p)$ always exists. If it were the case that $deepest(v) \in B(t, p)$, then we would have $p \geq h(z)+1 \geq h(t)+2$, which contradicts our assumption that $p \in \{h(t), h(t) + 1\}$. In $O(1)$ time, we can locate $w = \pi_k(deepest(v))$, where $k = h(z) + d(t) - d(z) - p$. Note that $dist(v, z) = dist(\pi(w), z)$. To ensure that (t, p) is valid, we now only need to check whether or not $N(t, p)$ contains any vertices other than w. This is done as in the previous case using a small number of RMQs, the details of which are given in pseudocode in Algorithm 4.1. The only subtle observation one needs to make is that if $t \notin P_{z'}$ (with z' defined as in the pseudocode in Algorithm 4.1), then (t, p) is not valid because this implies that $P_{z'}$ would contribute a

second vertex to $N(t, p)$ as it descends downward from $B(t, p)$ (by the same argument we used to show that P_v contributes the vertex w to $N(t, p)$).

Chapter 5

The Internal Case

In the external case, we have the advantage of knowing what power to assign to a transmitter. This no longer holds in the internal case, but fortunately we can rely on another useful structural property instead. As we see in the following lemma, the internal case reduces to a somewhat complicated one-dimensional dynamic programming problem along the paths in our deepest path decomposition.

Lemma 5 Consider the computation of $icost[v]$ for some vertex v. If $icost[v] \neq +\infty$, then we can write $icost[v] = f(t) + cost[w]$, where $t \in P_v$ is an internal transmitter and $w \in P_v$ is the unique vertex in $N(t, f(t))$.

Proof. Assume that $icost[v] \neq +\infty$, and consider an optimal proper broadcast f that dominates only v's subtree, such that v is internally dominated. Let t be the unique transmitter responsible for dominating v. Since f is proper we have $|N(t, f(t))| \leq 1$, and since t is an internal transmitter we also have $|N(t, f(t))| \geq 1$, in particular since P_v descends downward out of $B(t, f(t))$. Let w denote the unique vertex in $N(t, f(t))$, contributed by P_v (see Figure 5.1(a) for a picture of the arrangement of v, t, and w). If $t \notin P_v$, then since t is not an external transmitter we know that $f(t) < h(t)$, from which we infer that P_t would contribute another element to $N(t, f(t))$ (distinct from w) as it descends downward out of $B(t, f(t))$, again contradicting the fact that $|N(t, f(t))| = 1$. □

Note that if we are given v and a particular w (as defined by the preceding lemma), we can compute $f(t) = (d(w) - d(v) - 1)/2$ and $t = \pi_{f(t)+1}(w)$. Similarly, we can compute w given t, albeit with a slightly more cumbersome formula. Since $icost[v]$ is completely determined once we know w, we can obtain $icost[v]$ by minimizing over different choices

for w below v on P_v:

$$\begin{aligned} icost[v] &= \min_{w \in F_v} \left\{ \frac{d(w) - d(v) - 1}{2} + cost[w] \right\} && (5.1) \\ &= -(d(v)+1)/2 + \min_{w \in F_v} \{ d(w)/2 + cost[w] \} && (5.2) \\ &= -(d(v)+1)/2 + \min_{w \in F_v} \{ val(w) \} && (5.3) \end{aligned}$$

where we define $val(w) = d(w)/2 + cost[w]$, and where $F_v \subset P_v$ denotes the set of *feasible choices* for w when computing $icost[v]$ according to (5.3). In order to characterize the set F_v, we first introduce some additional notation. Let $B_{vw} = B(t, f(t))$, where t and $f(t)$ are computed as above based on v and w. Note that for the ball B_{vw} to exist at all, its radius $f(t) = (d(w) - d(v) - 1)/2$ must be a positive integer, so $d(w)$ must have different parity from $d(v)$, and also $w \neq c_1(v)$. If B_{vw} exists, we also define $N_{vw} = N(t, f(t))$ to be its lower neighbor set. We can now give a simple characterization for F_v: $w \in F_v$ if and only if (i) B_{vw} exists, and (ii) $N_{vw} = \{w\}$. Condition (i) is easy to satisfy as long as we search for w among the set of vertices below $c_1(v)$ on P_v for which $d(w)$ and $d(v)$ differ in parity. Condition (ii) is slightly more complicated. It disqualifies a vertex w if there is a sufficiently long path that branches off P_v inside B_{vw} and escapes from B_{vw}, contributing another lower neighbor to N_{vw} in addition to w. We call such paths *overhangs*, and discuss their exact behavior in a moment.

We are now ready to describe our algorithm for the internal case. At a high level, it computes $icost[v]$ using (5.3) for every vertex v during the same postorder scan of T in which we compute $ecost[v]$ and $cost[v]$. We can think of this computation a collection of independent computations, one for each of the "deep paths" into which we have decomposed T. Along each such path, we are scanning upward and computing $icost[v]$ for each vertex v in sequence. We henceforth restrict our focus to just one such path.

A direct application of (5.3) would spend at least $O(n)$ time computing $icost[v]$ for a particular v. To improve this, we note that the minimization in (5.3) behaves in a very "stable" fashion: the same w will achieve the minimum as long as F_v does not change for successive vertices v. In our case, $F_v \cap F_{\pi(v)} = \emptyset$ due to parity constraints, but F_v and $F_{\pi(\pi(v))}$ are highly related. Using appropriate data structures, we will show how to compute

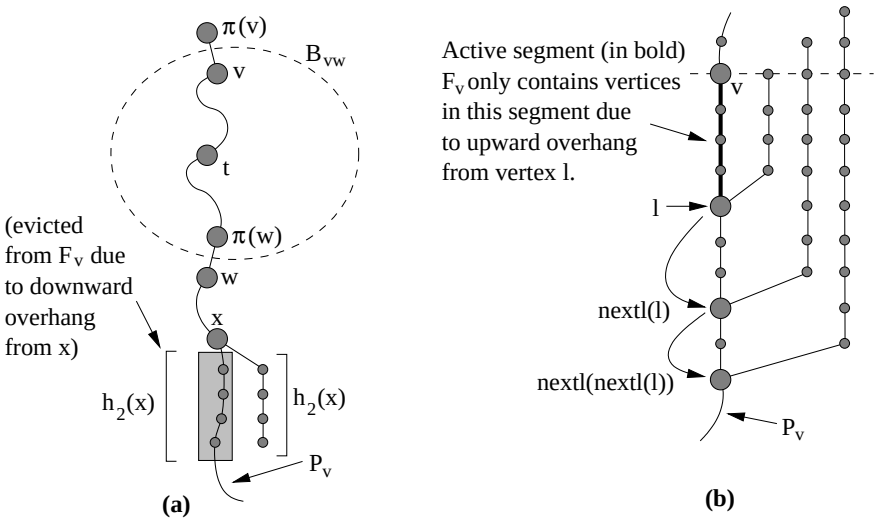

Figure 5.1: Illustration of the downward and upward overhang conditions. In (a), the downward overhang of length $h_2(x)$ branching off x permanently disqualifies the $h_2(x)$ vertices in its "shadow" below x on P_v from belonging to F_v. In (b), the upward overhang from l (actually a downward path that branches off and descends from l, which we visualize by stretching it out upward) prevents any choice of w below l from belonging to F_v. When our algorithm steps from v to $\pi(v)$, we will "escape" from this upward overhang and be constrained instead by the larger upward overhang branching off $next(l)$. The current stack of nested upward overhangs branching off l, $next(l)$, and so on, forms what we call the l-list.

$icost[\pi(\pi(v))]$ in only $O(1)$ amortized time after already having computed $icost[v]$.

5.1 Overhang Conditions

Recall that we must have $N_{vw} = \{w\}$ in order for w to belong to F_v. That is, the ball B_{vw} can have only w itself as a lower neighbor, so any branch off P_v inside B_{vw} must therefore be completely contained within B_{vw}. The following two *overhang conditions* use this fact to restrict F_v.

- **The Downward Overhang Condition.** For every vertex $x \in P_v[v \ldots deepest(v)]$, none of the $h_2(x)$ vertices immediately below x on P_v can belong to F_v.
- **The Upward Overhang Condition.** For every vertex $x \in P_v[v \ldots deepest(v)]$, if $h_2(x) > d(x) - d(v)$, then none of the vertices below x on P_v can belong to F_v.

Let us briefly try to develop an intuitive notion of these two conditions. An overhang is a path branching off $P_v[v \ldots]$ that restricts the feasible set F_v. Each such branching path

plays the role of both a downward overhang and an upward overhang. As shown in Figure 5.1(a), consider taking the longest path branching off vertex $x \in P_v[v \ldots]$ (of length $h_2(x)$), and stretching this path out downward alongside P_v. According to the downward overhang condition, all vertices $w \in P_v$ in its "shadow" are disqualified from belonging to the feasible set F_v. Note also that these vertices are permanently disqualified since this same downward overhang constraint will apply for successive vertices v as the algorithm continues scanning upward along P_v. Consider now taking a similar path (for example, branching off vertex l in Figure 5.1(b)), and stretching it out upward alongside P_v. If this path extends as high as v, then l satisfies the upward overhang condition and no vertex below l can belong to F_v. However, as opposed to downward overhang constraints, upward overhang constraints are only temporary, since eventually we will "escape" from an upward overhang once we finally step upward to a vertex v that is high enough along P_v. Upward overhangs form a nesting structure (shown in the figure), where an escape from one upward overhang may leave us constrained by a larger upward overhang that branches off P_v from an even deeper vertex. In a moment we will introduce a data structure called the l-list for maintaining the nested structure of these upward overhangs as our algorithm proceeds to scan upward along P_v.

Lemma 6 For a given v and $w \in P_v$, if B_{vw} exists, then $N_{vw} = \{w\}$ (i.e., $w \in F_v$) if and only if w is not disqualified by the downward or upward overhang conditions.

Proof. Suppose that w violates the downward overhang condition with respect to some vertex x. As one can see in Figure 5.1(a), a choice of w from any of the $h_2(x)$ vertices below x leads to a ball B_{vw} with $|N_{vw}| > 1$, since the path branching off x would escape from B_{vw} and contribute a second lower neighbor in addition to w. Now suppose that w violates the upward but not the downward overhang condition with respect to x, as illustrated in Figure 5.1(b). Here, w cannot be chosen from the $h_2(x)$ vertices immediately below x, so let us assume that w has even lower depth $d(w) > d(x) + h_2(x)$. The transmitter at the center of B_{vw} would then satisfy $d(t) = d(w) - 1 - p(w) > d(x)$, and so its transmission would travel upward to x and then branch, reaching v before it reaches the end of the path branching off x. Consequently, the path of height $h_2(x)$ branching off x would escape from B_{vw} and

contribute an extra lower neighbor to N_{vw}. Finally, suppose that N_{vw} contains some vertex z in addition to w, and let $x = LCA(w, z) \in P_v$. If $d(x) \geq d(t)$ for the transmitter t of B_{vw} on P_v, then we must have $h_2(x) \geq d(w) - d(x)$, so w violates the downward overhang condition with respect to x. On the other hand, if $d(x) < d(t)$ then it must be the case that $h_2(x) > d(x) - d(v)$, so w violates the upward overhang condition with respect to x. □

Using overhang conditions, we now have a means of characterizing the feasible set F_v over which we are minimizing at each step of our algorithm. We now introduce some simple data structures that allow us to keep track of relevant overhang constraints (and thus maintain F_v implicitly) over the course of the algorithm as it scans upward along P_v.

5.2 Managing Upward Overhangs: The *l*-list

When computing F_v for a particular vertex v, there might be several vertices $x \in P[v \ldots]$ that qualify for the upward overhang condition. The highest of these along P_v is the most interesting, since it provides the tightest bound on the feasible set F_v. We call this vertex l, since it provides a lower bound on the range of valid choices for $w \in F_v$ that captures every upward overhang constraint. As we move up the tree during our algorithm, every time we encounter a vertex v from which $h_2(v) > 0$, this introduces a new upward overhang, and as a result we must reset l to point to v. However, whenever this happens, we remember the previous location of l by first setting a pointer $nextl(v) = l$. The resulting chain of pointers forms a linked list that we call the l-list, shown in Figure 5.1(b). The elements in the l-list partition $P_v[v \ldots]$ into regions that we call *segments*. The highest of these, $P_v[v \ldots l]$, is called the *active segment*, the next segment is $P_v[c_1(l) \ldots nextl(l)]$, and so on. Due to the upward overhang condition, we know that F_v contains only vertices in the active segment.

As we see in Figure 5.1(b), the l-list encodes a set of upward overhangs that is "nested", so we can rightfully think of the l-list as a stack. As we scan upward along P_v during the course of the algorithm, we adjust the l-list by popping off any vertices from whose upward overhangs we manage to "escape". That is, at each vertex v, we check if $d(l) - d(v) > h_2(l)$. If so, we have "escaped" from the upward overhang branching off l, and so we accordingly pop l off the l-list (setting $l = nextl(l)$) and enlarge the active segment. Since it is possible that we

can escape from several overhangs in a single step, we may continue to pop entries off the top of the l-list until we finally reach a vertex l that provides a valid upward overhang. It is possible for l to scan all the way to the end of the l-list, in which case we set l to $deepest(v)$ since all of $P_v[v \ldots deepest(v)]$ should now be the active segment (l is also initialized to $deepest(v)$ at the very beginning of our scan up P_v).

All updates to the l pointer require $O(1)$ amortized time per vertex v we examine, so we now have a convenient and efficient way to track the exact set of vertices $P_v[v \ldots l]$ surviving the upward overhang condition.

5.3 Maintaining the Minimum: The m-lists

We maintain two more doubly linked lists embedded within the vertices of our current path P_v, which we call the m-lists (m for "minimum"), one of which is shown in Figure 5.2(a). An m-list contains a set of all vertices that could conceivably be the optimal $w \in F_v$ used for computing $icost[\cdot]$ of v or one of its ancestors according to (5.3). Since vertices v of odd depth allow only for vertices $w \in F_v$ of even depth and vice versa, we maintain two m-lists. The *even* m-list contains only even-depth vertices and is used for computing $icost[v]$ where v has odd depth, and vice versa for the *odd* m-list. We maintain pointers m_{even} and m_{odd} to the highest vertices on the even and odd m-lists. When computing $icost[v]$ for a particular vertex v, we will sometimes speak of "the" m-list, by which we mean the m-list having the opposite parity of $d(v)$ (i.e., the one relevant for computing $icost[v]$). We use m to refer generically to the top entry in this list (either m_{even} or m_{odd}).

Consider the sequence of $val(x)$ values for vertices x along an m-list. For both m-lists, we maintain the invariant that this sequence will be monotonically increasing from top to bottom within each segment of the l-list. In particular, this means the vertex w minimizing (5.3) can be found by looking at the topmost vertex in the m-list within the active segment of the l-list — in other words, m. This allows us to compute $icost[v] = -(d(v) + 1)/2 + val[m]$ in $O(1)$ time. If it so happens that there are no entries in the m-list within the active segment of the l-list (i.e., if $d(m) > d(l)$ or if the m-list is completely empty), then we instead set $icost[v] = +\infty$.

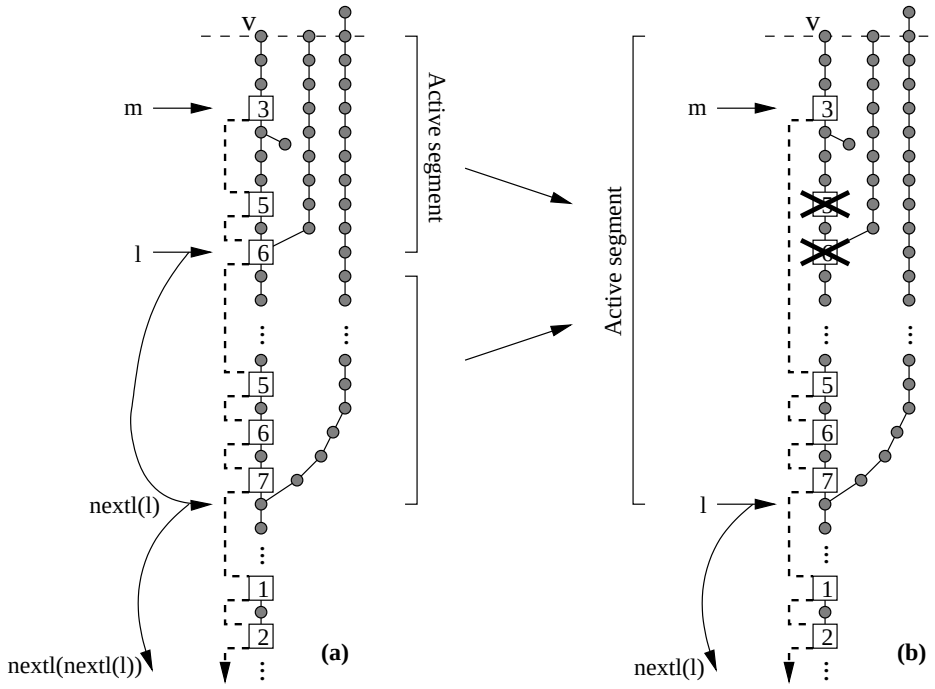

Figure 5.2: Illustrations of (a) an *m*-list (drawn with dotted links), and (b) the operation of splicing together the portions of the *m*-list belonging to the top two segments of the *l*-list. This event happens whenever *v* "escapes" from the current upward overhang from *l* (upward overhangs are emphasized in the figure by stretching these paths upward). The other *m*-list of different parity is not pictured, but it would also be spliced in the same fashion. The boxed number shown at a vertex *x* is $val(x)$.

Vertices are removed from *m*-lists whenever it is clear that they are no longer relevant choices for *w* for minimizing (5.3) for the current *v* or any ancestor of *v* on P_v that we will encounter in the future. For example, when we reach a vertex *v* with $h_2(v) > 0$, the downward overhang from *v* disqualifies all vertices up to $h_2(v)$ steps below *v* from inclusion in the current F_v and all future F_v's. Therefore, as long as $d(m_{even}) \leq v + h_2(v)$, we remove m_{even} from the head of the even *m*-list, and we do the same for the odd *m*-list. Since vertices are never re-admitted to an *m*-list, we spend at most $O(1)$ time per vertex in total for all *m*-list deletions. Finally, note that deletion of the top part of any *m*-list due to a downward overhang preserves the invariant that the *m*-lists are monotonically increasing from top to bottom in each *l*-list segment.

When our algorithm advances to some vertex *v*, we add *v*'s great-grandchild on P_v, $g = c_1(c_1(c_1(v)))$, as a new candidate on the top of the *m*-list. Recall that *v* and $c_1(c_1(v))$ are not in F_v since they share the same depth parity as *v*, and $c_1(v) \notin F_v$ since then B_{vw} would

not be defined (its radius would be zero). We therefore plan to introduce g as the new topmost element on the m-list. However, if m and g are in the same segment of the l-list (which we can easily check in $O(1)$ time) and $val[m] \leq val[g]$, then we forgo adding g to the m-list, since from now on, m will always be at least as good a candidate for minimizing (5.3) whenever g is a candidate. Any future deletion (say, due to downward overhang) that removes m will also removes g, since g is above m. Therefore, it is safe to omit g from the m-list. Moreover, since we only add a new vertex to the m-list if its value is lower than the current topmost vertex, this preserves our invariant that the m-list is monotonically increasing from top to bottom within each segment.

The only remaining update to the m-list we need to describe is a *splice* operation that takes place any time we escape from an upward overhang. Recall that here the active segment of the l-list is repeatedly merged with the segment immediately below it as we remove vertices from the top of the l-list. Whenever two adjacent segments s_1 and s_2 in the l-list are merged, we need to splice together the monotonically increasing portions of both the even and odd m-lists in s_1 and in s_2 to form one large monotonically increasing list over $s_1 \cup s_2$, as shown in Figure 5.2(b). This is done as follows: starting from the top of an m-list in s_2 (the lower of the two segments), scan upward into the s_1 and delete vertices from the bottom of the m-list in s_1 until the two m-lists form a single monotonic chain. Due to the same reasoning as above, these vertices are safe to delete since they are each "dominated" by a more optimal vertex lower in the same l-list segment. Since deletions are permanent and only contribute $O(1)$ running time per vertex, the only operation we must treat with some care in this case is that of finding the topmost m-list vertex in a segment. This is done by maintaining bidirectional pointers from each vertex x in the l-list to the topmost elements on the odd and even m-lists within the segment below x. With appropriate care, these pointers require only $O(1)$ time to update per vertex v that we visit.

This concludes the description of the algorithm for computing $icost[v]$. Its running time is clearly $O(n)$ in total since we spend $O(1)$ time per vertex v visited, plus an additional $O(1)$ time per vertex to possibly remove it in the future from an m-list.

Chapter 6

Extensions and Future Directions

Our algorithm easily generalizes (with no degradation in running time) to handle the extended problem variant where there is a fixed cost for building each transmitter in addition to a linear cost based on its power. This is possible because Lemmas 1 and 2 generalize to this case, so the structure of an optimal solution can still be assumed to be a path of disjoint balls. It is important, however, that the fixed cost must be independent of transmitter location; otherwise, the lemmas fail to apply.

Since a concave function f satisfies $f(x+y) \leq f(x) + f(y)$ for $x, y \geq 0$, Lemmas 1 and 2 also generalize to the case where the cost of a transmitter at vertex v is of the form $c_{fixed} + c_{power}(f(v))$, where c_{fixed} is a fixed one-time cost and c_{power} is a concave function of the broadcast power, and where c_{fixed} and c_{power} do not depend on v (this also implies that the algorithm of Heggernes and Lokshtanov [15] for broadcast domination of a general graph extends automatically to this more sophisticated problem variant). Concave transmitter costs are generally realistic, due to economies of scale. In our algorithm, the added difficulty with concave transmitter costs is that in the internal case, the optimal vertex among the elements in an m-list might move down the list monotonically over time, rather than staying at the top of the list as it does now. However, we can still adapt our algorithm without any running time penalty, if we incorporate additional logic that monitors and removes the top element from an m-list in the active segment whenever it becomes sub-optimal.

Since they no longer satisfy Lemmas 1 and 2, convex transmitter costs seem much more difficult to accommodate in a highly-efficient fashion on trees (and for a general graph, these make the problem NP-hard since the minimum dominating set problem is a special case of this problem where the transmitter cost jumps from zero to infinity at power level 2). A similarly-behaving problem (NP-hard on general graphs for the same reason) is the distance-attenuated case where a transmitter gradually loses power according to some

decreasing function of transmission distance, and every vertex must hear at least some minimum amount of total transmission power (possibly from a combination of transmitters). Other interesting variants include the problem variant with non-unit edge lengths, the *multi-cover* variant where some nodes must hear multiple broadcasts for fault-tolerance purposes, and the partial cover problem where we can leave certain vertices uncovered at a cost (one can think of such a vertex as a zero-power transmitter in the fixed cost model). The partial cover variant is particularly interesting as it satisfies both Lemmas 1 and 2, and yet still seems perhaps difficult to solve in polynomial time on a general graph as well as in linear time on a tree (it can be solved in polynomial time on a tree; see [13, 16, 18]).

As shown in [13, 16], the natural fractional relaxation of the broadcast domination problem to a linear program always admits an integer-valued optimal solution, and so does its dual. The dual problem of broadcast domination is actually interesting in its own right and in its integer version does not appear to have been studied previously. Here our goal is to select a maximum-cardinality subset of vertices S so that $|S \cap B(v, r)| \leq r$ for all $v \in V$ and $r \geq 1$. In other words, we would like to locate facilities throughout a graph subject to a "density" constraint that prohibits too many facilities from being opened in any particular radial region. One can solve this problem in $O(n^3)$ time in trees [13, 16], but it is not known if one can do so in nearly-linear time (say, by adapting the result from this paper).

Blair and Horton [7] study the related problem of *broadcast covering* in a graph, where the goal is to find a broadcast that covers all the edges, rather than all the vertices of a graph. This problem is much easier than the broadcast domination problem since in any graph one can find an optimal broadcast cover that is radial, allowing for a simple $O(n)$ algorithm on trees. For the broadcast domination problem, it remains a challenging open problem (even in the case of trees) to characterize which inputs admit optimal radial solutions.

BIBLIOGRAPHY

[1] S. Arnborg, J. Lagergren, and D. Seese, *Easy problems for tree-decomposable graphs*, J. Algorithms **12** (1991), 308–340.

[2] M.A. Bender and M. Farach-Colton, *The LCA problem revisited*, Proceedings of the 4th Latin American Theoretical Informatics (LATIN) Conference, 2000, pp. 88–94.

[3] _____, *The level ancestor problem simplified*, Theor. Comput. Sci. **321** (2004), no. 1, 5–12.

[4] O. Berkman and U. Vishkin, *Finding level-ancestors in trees*, J. Comput. Sys. Sci. **48** (1994), no. 2, 214–230.

[5] M.W. Bern, E.L. Lawler, and A.L. Wong, *Linear-time computation of optimal subgraphs of decomposable graphs*, J. Algorithms **8** (1987), no. 2, 216–235.

[6] J.R.S. Blair, P. Heggernes, S.B. Horton, and F. Maine, *Broadcast domination algorithms for interval graphs, series-parallel graphs, and trees*, Proceedings of the 35th Southeastern International Conference on Combinatorics, Graph Theory, and Computing (CGTC), 2004.

[7] J.R.S. Blair and S.B. Horton, *Broadcast covers in graphs*, Congressus Numerantium **173** (2005), 109–115.

[8] R.B. Borie, R.G. Parker, and C.A. Tovey, *Automatic generation of linear-time algorithms from predicate calculus descriptions of of problems on recursively-constructed graph families*, Algorithmica **7** (1992), 555–582.

[9] B. Courcelle, *The monadic second order logic of graphs I: Recognizable sets of finite graphs*, Infor. and Comput. **85** (1990), 12–75.

[10] _____, *The monadic second-order logic of graphs III: treewidth, forbidden minors and complexity issues*, Informatique Théoretique **26** (1992), 257–286.

[11] J.E. Dunbar, D.J. Erwin, T.W. Haynes, S.M. Hedetniemi, and S.T. Hedetniemi, *Broadcasts in graphs*, Disc. Appl. Math. **154** (2006), no. 1, 59–75.

[12] D.J. Erwin, *Dominating broadcasts in graphs*, Bull. Inst. Combin. Appl. **42** (2004), 89–105.

[13] M. Farber, *Domination, independent domination, and duality in strongly chordal graphs*, Disc. Appl. Math. **7** (1984), 115–130.

[14] D. Harel and R.E. Tarjan, *Fast algorithms for finding nearest common ancestors*, SIAM J. Comput. **13** (1984), no. 2, 338–355.

[15] P. Heggernes and D. Lokshtanov, *Optimal broadcast domination in polynomial time*, Disc. Math. **306** (2006), no. 24, 3267–3280.

[16] A.J. Hoffman, M. Sakarovich, and A. Kolen, *Totally balanced and greedy matrices*, SIAM J. Alg. Disc. Meth. **6** (1985), 721–730.

[17] S.B. Horton, C.N. Meneses, A. Mukherjee, and M.E. Uluçakli, *A computational study of the broadcast domination problem*, Tech. Report 2004-45, DIMACS, 2004.

[18] A. Lubiw, *Doubly lexical orderings of matrices*, SIAM J. Comput. **16** (1987), no. 5, 854–879.

[19] T.V. Wimer, S.T. Hedetniemi, and R.C. Laskar, *A methodology for constructing linear graph algorithms*, Proceedings of the Sundance Conference on Combinatorics and Related Topics (Sundance, Utah), vol. 50, 1985, pp. 43–60.