Implementation of Broadcast Domination Algorithms for Interval Graphs, Series-Parallel Graphs, and Trees

Helge Holm
(helgeh@ii.uib.no)

2004

Master Thesis

Department of Informatics
University of Bergen
Norway

Acknowledgments

Thanks to Beate Skogly for support and for being patient while I was working on this thesis.

I would especially like to thank my supervisor, Pinar Heggernes, for always finding time for assistance and constructive criticism.

I also want to thank Harald Barsnes, Halvor Utby, and Gard Vaaler for proofreading and giving valuable feedback.

Contents

Chapter 1

Introduction

In this chapter, we present the motivation behind this work, its goals, and some basic definitions from graph theory.

1.1 Abstract

Domination in graphs is a thoroughly studied subject that has recently been generalised to Broadcast Domination and the Optimal Broadcast Domination problem [7] by D. J. Erwin. Whether the Optimal Broadcast Domination problem is generally solvable in polynomial time is still unknown, but research performed by Jean R. S. Blair, Pinar Heggernes, Steve Horton, and Fredrik Manne has yielded polynomial time algorithms for Optimal Broadcast Domination on interval graphs, series-parallel graphs and trees [1].

The main part of this work has been to implement each of the algorithms for interval graphs, series-parallel graphs, and trees so that it would be possible to use these algorithms in a practical setting, either in research or as part of software applications. We have also used the implementations to test the practical limitations of the algorithms and how the running time turns out in practice on arbitrary interval graphs, series-parallel graphs, and trees, and how likely it is that the optimal dominating broadcast differs from a trivial solution.

Also as a part of this work, we have detected and corrected some errors found in [1], and constructed some simple graph classes that yield easily predictable solutions to the Broadcast Domination problem.

1.2 Implementation

> *Implementation*[22]: In engineering and computer science, an implementation is the practical application of a method or algorithm to fulfill a desired purpose. For example, one might create a computer program that sorts a list of numbers in ascending order. To do so, one would implement a known method of sorting.

Each algorithm is implemented as a stand-alone, command line program written in standard C, and is optimised as much as possible without losing code readability. The preprocessing functions are not optimised much, because their running times are almost unnoticeable compared to the algorithms themselves.

To make the algorithms easy to use, the programs have been designed to be as similar as possible to standard Unix and Linux command line tools by satisfying the following conditions:

- Source code is written in C.
- Programs are runnable from the command line.
- All input is read from the standard input stream and all output is written to the standard output stream.
- Source code is machine-independent.
- Program parameters and input format explained by supplying command line parameter `--help`.

The advantages motivating this approach are:

modularity The programs can easily be used as part of other programs, completely hidden from the end user. For example, a graphical graph editing program can at any time translate its graph to the format accepted by these programs, call up the algorithm in the background, and display the result as a part of its own graphical output.

batch programming All operating systems provide command line batch file programming of some sort. This allows the user to specify that a program should be run multiple times, once on every graph in a given list, and write the results to a text file. This is very useful when the program may take several minutes or hours to run on a graph. Most of the tests in Chapter 7 were (and should be) run by batch files.

usability Because the text-based interfaces of the programs are similar to standard command line tools, anyone familiar with Unix-like systems should be able to understand how to use the programs without having to read any external documentation.

speed The programming language C is renowned as the optimal choice when fast running time and control of memory usage are vital to the program.

All source code for the algorithms, graph generation programs and automated tests, as well as some sample data, is available from `http://www.ii.uib.no/~helgeh/master/` or by e-mailing the author at `helge.holm@gmail.com`.

The source code has not been given as an appendix to the thesis, mainly because C code is notoriously unreadable and as such would not be of much interest to the reader, and also because adding the collected source code would triple the number of pages in the thesis.

1.3 Definitions and terminology

This section will describe the concepts, definitions and terminology of graph theory that will be needed to explain and understand the Broadcast Domination problem described in Chapter 2. The definitions of the three graph classes interval graphs, trees, and series-parallel graphs will be given in Chapters 3, 4, and 5, respectively, along with their corresponding algorithms.

1.3.1 Graphs

Graphs are used to model many real-world or theoretical problems to make them easier to analyse and solve mathematically. Some of the most popular and practical uses are the many graph algorithms for "shortest path", which may be used to find the fastest driving route from one city to another, the minimum number of moves necessary to solve a puzzle and so on.

A *graph* $G = (V, E)$ consists of a set of *vertices* V and a set of *edges* E. A vertex is a single point in the graph, each edge is a connection between two vertices, and each pair of vertices connected by an edge are said to be *adjacent* to each other and *incident* to the edge connecting them. (This will be more formally defined further down.) In the case of a bus transit map (pictured in Figure 1.1), the vertices can be bus stops and each edge can be the buses traveling between two stops. In the case of a puzzle (pictured in Figure 1.2) the vertices can be board configurations and the edges valid moves between configurations.

The size of a graph is given by the number of vertices $n = |V|$ and the number of edges $m = |E|$. Unless otherwise specified, when n is given as part of an expression, for example in $O(n^2)$, it always denotes the number of vertices. Likewise, m denotes the number of edges unless otherwise specified.

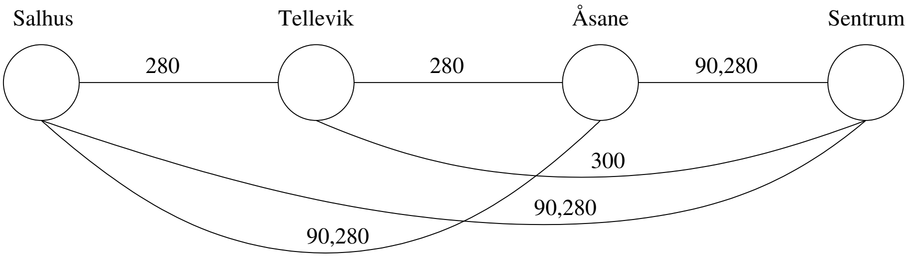


Figure 1.1: A simplified bus transit map

A simplified map of the available bus routes between the locations Salhus and Sentrum in Bergen. If we want to travel from one location to another with the minimum number of buses, we can apply the Shortest Path algorithm to this graph. If we want to travel from Salhus to Sentrum as fast as possible, we can generate all possible routes (avoiding routes that visit the same location twice) from Salhus to Sentrum from the graph, and then use a bus time table to calculate how long each route will take.

A *directed* edge from u to v would mean there was an edge from v to u but not from u to v. However, we will only be dealing with *undirected* graphs where we assume that for each edge from u to v there is also an implicit edge from v to u. There are several popular notations for describing edges. For example uv, (u, v), $\{u, v\}$ are all identical for undirected graphs. We will mostly be using (u, v) because it most resembles the syntax used in programming languages.

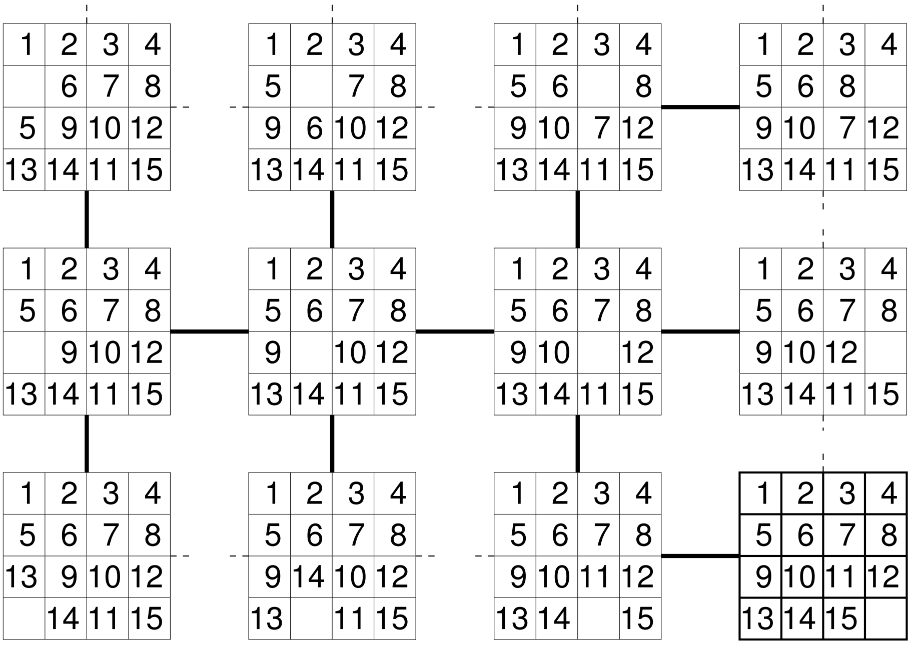


Figure 1.2: A puzzle represented as a graph

A *very small* cut-out of a graph representation of a well-known puzzle toy called a “15-puzzle”. The goal is to put all the numbers in an ascending sequence by repeatedly sliding an adjacent numbered square into the empty square. The vertices are the puzzle configurations, and the edges are valid moves connecting the configurations. The problem of solving a given puzzle can now be reduced to the problem of navigating this maze graph where we start at the configuration matching the one we are given, the exit from the maze is at the solution configuration, and we can only move from configuration to configuration by the edges. Since there already exist many good, proven algorithms for navigating a graph, we will not have to think up, analyse and prove a new algorithm for solving these puzzles; it is sufficient to show how to generate the graph (maze) from the puzzle and prove that a solution to the maze is a solution to the puzzle.

Formally, two vertices u and v are said to be adjacent, or neighbors, if and only if the edge $(u, v) \in E$. Also, both u and v are incident to (u, v). The neighborhood, or set of adjacent vertices, of u is defined as $N(u) = \{v \in V : (u, v) \in E\}$. The neighborhood of a set S of vertices is given by $N(S) = \bigcup_{v \in S} N(v) - S$.

An *induced subgraph* is a graph that can be generated by a subset of the vertices of the original graph, and is defined as: G' is an induced subgraph of G if $V(G') \subseteq V(G)$ and $E(G') = \{(u, v) : u, v \in V(G') \wedge (u, v) \in E(G)\}$. We will be using the notation $G[S]$ to denote an induced subgraph of G where $V(G[S]) = S$.

A *path* of length k is a graph described by a sequence of $k + 1$ vertices $v_1, v_2, \ldots, v_{k+1}$ such that $E = \{(v_i, v_{i+1}) : 1 \leq i \leq k\}$ and $V = \{v_i : 1 \leq i \leq k + 1\}$. The length of a path is thus given by the number of edges, $|E|$. We say that the path given by the sequence $v_1, v_2, \ldots, v_n$ *connects* v_1 and v_n. We will use the notation P_j, where $j > 0$, to denote a path of j vertices and length $j - 1$.

A pair of vertices u, v in a graph G are said to have *distance d* if the shortest path connecting the two nodes has length d, and we define $d(u, v)$ to be the distance from u to v. Note that a shortest path from u to v in G is also an induced subgraph of G.

The *eccentricity*, $e(v)$, of a vertex v is the length of the longest of all the shortest paths from v to any vertex of G, or $e(v) = \max_{u \in V} d(u, v)$. Simply put, the eccentricity of v is the maximum distance to any other vertex.

The *diameter*, $diam(G)$, of a graph G is simply the largest eccentricity in G, or $\max_{v \in G} e(v)$, and the *radius*, $rad(G)$, of G is the smallest eccentricity in G, or $\min_{v \in G} e(v)$. All vertices $v \in V$ where $e(v) = rad(G)$ are called *central vertices*. Note that, by definition, no shortest path between any two vertices in G can be longer than $diam(G)$.

A *cycle* of length k is a graph described by a sequence of $k + 1$ vertices $v_1, v_2, \ldots, v_k, v_1$ such that $E = \{(v_i, v_{i+1}) : 1 \leq i < k\} \cup \{v_k, v_1\}$ and $V = \{v_i : 1 \leq i \leq k\}$. The length of a cycle is given by the number of edges, and we use the notation C_k, where $k \geq 1$, to denote a cycle of k vertices. Unless otherwise specified, a cycle is assumed to be a *simple cycle*, meaning that each vertex in the cycle is incident to exactly two edges. Note the similarity to the definition of a path, and that by adding an edge between the first and last vertex in a path, we get a cycle. Cycles are often induced subgraphs, called *induced cycles*.

A *complete graph* is a graph where each vertex is adjacent to every other vertex, i.e. $\forall_{u,v \in V}(u, v) \in E$. The notation K_k will be used to describe a complete graph containing k vertices.

For a given graph G, a *clique* is a set of vertices $S \subseteq V(G)$ such that $G[S]$ would be a complete graph, i.e. $\forall_{u,v \in S}(u, v) \in E$.

Unless otherwise specified, all graphs in this work are *simple*, meaning they have no more than one edge between any two vertices, and *connected*, meaning that for any pair of vertices there exists a path between them.

1.3.2 Graph classes

A graph class is a set of limitations used to identify graphs that share specific properties. A graph is said to be a member of a graph class if it satisfies the given limitations of the class, and a graph may therefore be a member of several classes. For example, all paths are also interval graphs, series-parallel graphs and trees.

Since graphs often are models of real world situations, they usually have some inherent limitations that can be exploited to make problem solving easier. A very simple example is the path class, as defined above. Almost every problem that is difficult or practically impossible to solve for general graphs are trivial to solve for paths due to their simple structure.

A *recursive graph class* is a class in which any sufficiently large member of the class can be formed by successively joining smaller members of the class at specific vertices called terminals.[1]

This work will mainly deal with the three graph classes interval graphs, trees and series-parallel graphs in Chapters 3, 4 and 5 respectively, but some specially constructed classes will be used for testing and for demonstrating some of the properties of the Broadcast Domination problem. These constructed graph classes will appear in Chapter 6.

1.3.3 P and NP

When discussing the difficulty of solving a problem, it is useful to try to classify the problem with regard to the problem classes *P* and *NP*. Formally, P is the class of languages that are decidable in polynomial time on a deterministic Turing-machine, and NP is the class of languages that are decidable in polynomial time on a non-deterministic Turing-machine[6].

As all computers today are deterministic, and we know of no way to emulate a non-deterministic computer in polynomial time, this translates roughly to P containing practically solvable problems and NP containing practically unsolvable problems.

We know that P$\subseteq$NP, but whether P=NP or P$\subset$NP is unknown. The common belief is that P$\subset$NP.

If a problem B is in NP, and a polynomial time solution to B implies a polynomial time solution to all problems in NP, then B is said to be *NP-complete.*

Whether Broadcast Domination is NP-complete is still an open problem, which will be described in more detail in Chapter 2.

1.4 Outline of the thesis

Chapter 2 will describe the Optimal Broadcast Domination problem and the necessary terminology needed to understand the rest of the chapters. It will also describe some of the many properties of Broadcast Domination.

Chapters 3, 4, and 5 will center around the algorithms for interval graphs, trees, and series-parallel graphs, respectively. They will define and describe their corresponding graph class, present the corresponding algorithm from [1] and which considerations had to be made to create working implementations, and give an example of how the finished algorithm runs.

Chapter 6 will present a few simple graph classes for which the Optimal Broadcast Domination problem is easily solvable, and describe how the optimal broadcast cost can be found in constant time.

Chapter 7 contains the details of and results from the tests we have run on the implementations. We have tested the correctness and performance of the

[1] See [2] for a more formal explanation, or [3] and [4] for examples on how a recursive structure can be used to solve graph problems.

implementations, and also how often the optimal broadcast cost differs from a trivial solution.

Chapter 8 describes some additional algorithms that had to be implemented to create usable programs.

Chapter 9 will sum up the results of all the preceding chapters, and present all the opportunities for further research that have surfaced while working with this thesis.

Chapter 2

Broadcast Domination

In this chapter we will describe in detail the Broadcast Domination problem and the related definitions and terminology needed to understand how the algorithms solve the problem.

The following two sections 2.1 and 2.2 are heavily based on [1], with added examples and figures.

2.1 Introduction and motivation

Domination in graphs is a well known and thoroughly studied subject [8, 9]. A *dominating set* in a graph G is a subset of the vertices of the graph, $V(G)$, such that every vertex in $V(G)$ is either in S or has an element of S as one of its neighbors. Equivalently $S \cup N(S) = V$. The optimal dominating set, i.e. where S is as small as possible is defined as $\gamma(G)$. A real world application of this problem [10] is to view the vertices as cities, and each city must be able to hear a radio station. A city will hear the radio station if it is located in the city itself or in a neighboring city. The standard optimal domination problem seeks a dominating set of minimum cardinality, i.e. S must be as small as possible. See Figure 2.1 for an example.

This has later been generalised to the r*-domination* problem [11, 12], where each vertex in V must either be in the dominating set S or at a distance at most r from some vertex in S. This again translates to radio stations that can reach farther cities than its immediate neighbors. See Figure 2.2 for an example.

Recently, D. J. Erwin introduced the concept of *Broadcast Domination*[7] in which the broadcast stations (i.e., vertices in the dominating set) are permitted to have different transmission powers. This is a more realistic model of broadcast reachability than the standard domination problem, since transmitters are not, in general, identical. For example, FM radio stations are distinguished both by their transmission frequency and by their ERP (Effective Radiated Power). A transmitter with a higher ERP can transmit further, but it is more expensive to build and to operate. Based on this, the broadcast domination problem seeks to compute an integer valued broadcast function f on the vertices, such that every vertex of the graph is at a distance at most $f(v)$ from some vertex v that has $f(v) > 0$. A broadcast domination is optimal if it minimises the sum of the costs of the broadcasts across all vertices in the graph. These costs are typically

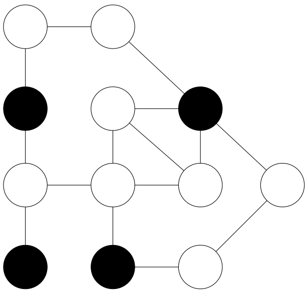


Figure 2.1: Example: dominating set

A dominating set, marked as black vertices. Each vertex v is either dominating (v is black) or has at least one dominating neighbor (v is white).

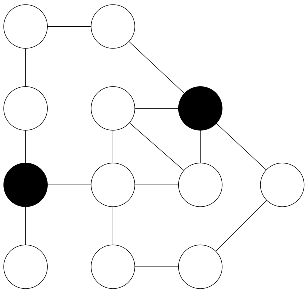


Figure 2.2: Example: r-dominating set

An r-dominating set for $r = 2$.

taken to be the $f(v)$ values, but could more realistically be a function over the $f(v)$ value. Other related broadcast problems are discussed in [13].

[1] also refers to $f(v)$ as the ***broadcast power*** or ***broadcast strength***, of v. Since the two notations mean the same, we will only be using "broadcast power", or simply "*power*".

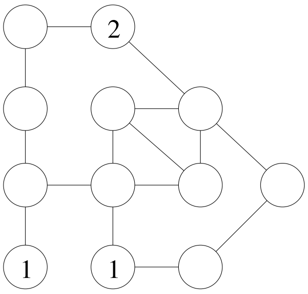


Figure 2.3: Example: a broadcast dominating set

A broadcast domination of a graph, with total cost 4. The broadcast vertices are marked with their corresponding broadcast value.

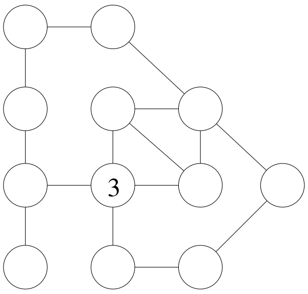


Figure 2.4: Example: minimal-cost broadcast dominating set

An optimal broadcast domination of a graph, with total cost 3. Note that this is also a radial broadcast.

The standard optimal domination problem ($\gamma(G)$) is NP-hard on, for example, planar graphs [14], bipartite graphs [15] and chordal graphs [16], but can be solved in polynomial time on, for example, AT-free graphs [17], permutation graphs [18] and interval graphs [19]. Some variants of the problem, like the ones previously mentioned, have straightforward reductions from the standard domination problem, showing that they are NP-hard on arbitrary graphs. However, the computational complexity of optimal broadcast domination on general graphs is an open problem [13].

The algorithms implemented in this work, all from [1], solve the optimal broadcast domination problem on interval graphs (Chapter 3), trees (Chapter 4), and series-parallel graphs (Chapter 5) in polynomial time.

Easy polynomial time solutions for the broadcast domination problem have been found for paths, cycles, complete graphs, grid graphs, and for some graph classes constructed specifically to have easy polynomial time solutions; 1-caterpillars, teeth and boxes. In each of these cases, there are optimal solutions that either are also solutions to the standard domination problem, or have exactly one non-zero broadcast located at the center of the graph [13]. The classes of graphs addressed in [1] and this work do not exhibit this property. Most of these "easy" classes and their solutions will be described in detail in Chapter 6.

2.2 Definition

A function $f : V \to \{0, 1, \ldots, diam(G)\}$ is a *broadcast* if for every vertex $v \in V$, $f(v) \leq e(v)$. The set of *broadcast dominators* defined by f is the set $V_f = \{v \in V : f(v) > 0\}$. The set of vertices that a vertex v can *hear* is $H_f(v) = \{u \in V_f : d(u,v) \leq f(u)\}$. We will omit the subscript f when the broadcast function is clear from the context. The *cost* of a broadcast f incurred by a set $S \subseteq V$ is $f(S) = \sum_{v \in S} f(v)$. Thus, $f(V)$ is the total cost incurred by the broadcast function f. We say that G has an $f(V)$-*broadcast.*

A broadcast is *dominating* if $|H(v)| \geq 1$ for every vertex v of G. The term $\gamma_b(G)$ denotes the minimum cost of a dominating broadcast on G. We will refer to a dominating broadcast of cost $\gamma_b(G)$ as an *optimal dominating broadcast*, or simply *optimal broadcast.* See figures 2.3 and 2.4 for examples of dominating broadcasts. Although a broadcast function is allowed to assign values from $\{0, 1, \ldots, diam(G)\}$, we never need to assign values larger than $rad(G)$ to any vertex in order to achieve an optimal broadcast. Choosing a central vertex v, i.e. a vertex with minimum eccentricity, and assigning $f(v) = rad(G)$ while assigning $f(w) = 0$ to all other vertices w, defines a dominating broadcast on G. We will call such a broadcast a *radial broadcast.* For some graph classes, a radial broadcast is also an optimal broadcast, and such graphs are called *radial* [13]. However, for interval graphs, series-parallel graphs and trees, radial broadcasts are not necessarily optimal, as can be seen from Figure 2.5.

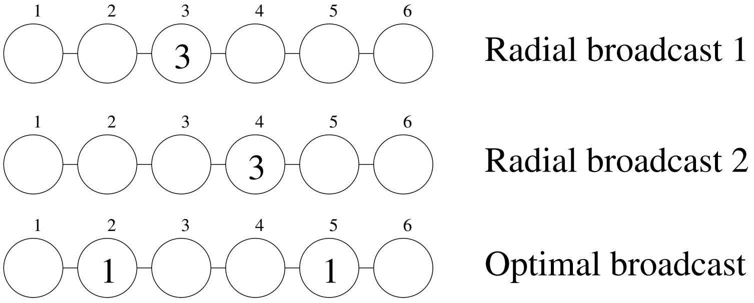


Figure 2.5: Example of radial solution not being optimal

A radial broadcast f requires $f(v_3)$ or $f(v_4)$ to be 3, which is the radius of a path of 6 vertices. However, a dominating broadcast f' of cost 2 can be achieved by assigning $f'(v_2) = f'(v_5) = 1$. Note that a path is also an interval graph, series-parallel graph and a tree (more about this in Chapter 6).

A broadcast is *efficient* if every vertex hears exactly one broadcast, that is, for every v, $|H(v)| = 1$. The following result from [13] is central to several of the results in [1].

Theorem 2.1. *(Dunbar et. al. [13]) Every graph G has a $\gamma_b(G)$-broadcast that is efficient.*

A more restricted problem is NP-complete, which follows immediately by restriction from dominating set. **Restricted Broadcast Domination (RBD):** Given a graph $G = (V, E)$, and positive integers $K, M \leq |V|$, is there a broadcast domination f of G such that:

$$\sum_{v \in V} f(v) \leq K \bigwedge \max_{v \in V} f(v) \leq M$$

2.3 Additional notations

In order to describe the algorithms for this problem, some extra notations will be used. The notations common to more than one algorithm are as follows.

2.3.1 Effective power

As defined above, a broadcast originating at v of power k will dominate any vertex at distance k or less from v. Now, if we look at a neighbor u of v, the same broadcast will dominate any vertex at distance $k-1$ or less from u, and we say that the broadcast from v has *effective power* of $k-1$ at u.

We define effective power as follows: A broadcast of power k, originating at v, will have effective power $k - d(v, w)$ at any vertex w.

2.3.2 Overdominance and underdominance

The algorithms for trees and series-parallel graphs work by utilising *dynamic programming.* This means they are gradually combining possible broadcasts for subgraphs until the optimal solution for the entire graph is calculated. Because the possible broadcasts for a subgraph $G_1 \subset G$ do not have to be neither dominating nor optimal, G_1 may sometimes need broadcasts originating from a vertex $v \in G, v \notin G_1$ to be dominated, and in this case we say that G_1 is *underdominated.* If G_1 is not underdominated, we say that G_1 is *overdominated.* If a broadcast originating from G_1 can be heard outside of G_1, we say that G_1 is *properly overdominated.* If G_1 is overdominated but not properly overdominated, we say that G_1 is *exactly dominated.*

We define $G_1 \subset G$ to have an *underdominance* of i if G_1 needs a broadcast of effective power at least i originating from outside G_1 to be efficiently dominated.

We define $G_1 \subset G$ to have an *overdominance* of i if a broadcast originating in G_1 can be heard by a vertex $v \in G, v \notin G_1$ at minimum distance i from any vertex in G_1 and no broadcast originating in G_1 can be heard by a vertex $v \in G, v \notin G_1$ at minimum distance $i+1$ from any vertex in G_1. Thus, i denotes how far away from G_1 the overdominance can reach.

We define $G_1 \subset G$ to have an *exact dominance*, or an *overdominance* of 0, if G_1 is dominated and no broadcast originating in G_1 can be heard by any vertex $v \in G, v \notin G_1$.

Any overdominance of $i > 0$ is a *proper overdominance.*

For an example of how overdominance and underdominance applies to subtrees, see Figure 2.6.

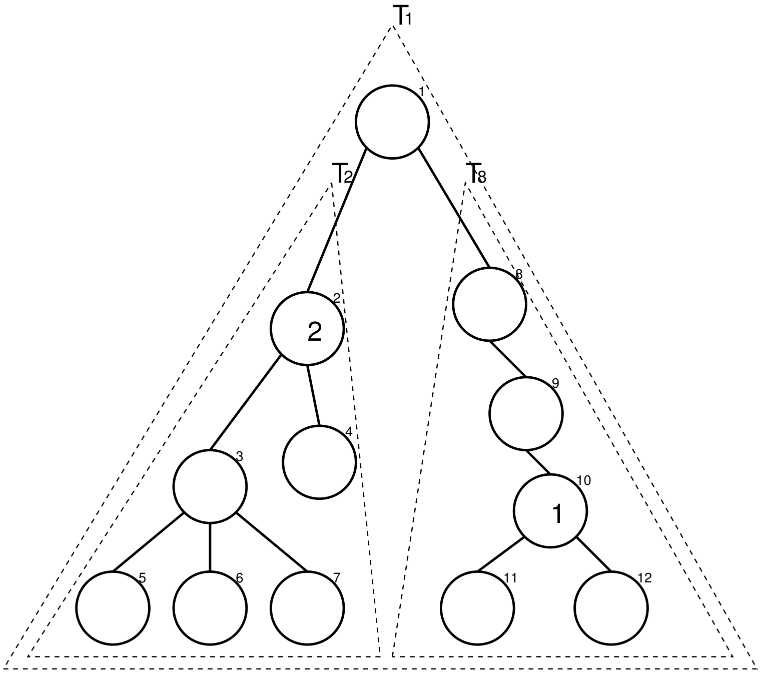


Figure 2.6: Example of overdominance and underdominance in a tree

This figure shows how underdominated and overdominated partial solutions for subtrees can be combined to an optimal solution for the entire tree. A possible solution for the subtree T_2 has an *overdominance* of 2, since the broadcast at vertex 2 can be heard by a vertex at distance 2 from any vertex in T_2. A possible solution for the subtree T_8 has an *underdominance* of 1, since it must hear a broadcast of effective power 1 from outside of T_8 for vertex 8 to be dominated. Because the broadcast from vertex 2 will have an effective power of 1 at a vertex in $N(T_8)$, the two partial solutions can be combined to yield an efficient, optimal solution for T_1.

Overdominance and underdominance can also be specified for each endpoint of G_1, with almost identical definitions:

We define $u \in G_1 \subset G$ to have exact dominance if u is dominated but has an effective power of 0.

We define $u \in G_1 \subset G$ to have an underdominance of i if G_1 needs a broadcast of at least i originating from a neighbor of u outside G_1 to be efficiently dominated.

We define $u \in G_1 \subset G$ to have an overdominance of 0 if it is exactly dominated, or an overdominance of i if a broadcast originating from u can be heard by a vertex $v \in G, v \notin G_1$ at minimum distance i from any vertex in G_1 and no broadcast originating in G_1 can be heard by a vertex $v \in G, v \notin G_1$ at minimum distance $i + 1$ from any vertex in G_1. If a vertex has overdominance of $i > 0$, it is properly overdominated.

For an example of how overdominance and underdominance applies to endpoints of subgraphs, see Figure 2.7.

The overdominance or underdominance of a vertex or subgraph is said to be the *dominance condition* of the vertex or subgraph.

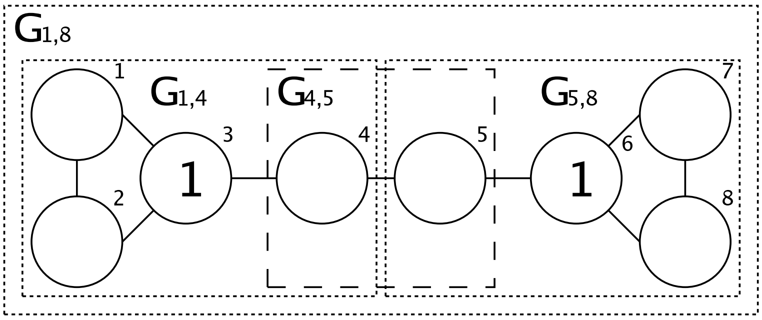


Figure 2.7: Example of overdominance and underdominance in a series-parallel graph

This figure shows how underdominated and exactly dominated partial solutions for subgraphs can be combined to an optimal solution for the entire series-parallel graph.

A possible solution for the subgraph $G_{1,4}$ has exact dominance at both endpoint vertices 1 and 4.

A possible solution for the subgraph $G_{4,5}$ has underdominance of 1 at both endpoint vertices 4 and 5.

A possible solution for the subgraph $G_{5,8}$ has exact dominance at both endpoint vertices 5 and 8.

Combining the solutions for the three subgraphs will yield an effective, optimal broadcast domination for the entire graph $G = G_{1,8}$. Series-parallel graphs, series-parallel construction, and the definitions of left and right terminals will be described in Chapter 5.

Chapter 3

Interval Graphs

In this chapter we will describe the graph class of interval graphs and the corresponding Minimum Broadcast Domination algorithm, an example of how the algorithm runs on a small graph, and the details of turning the algorithm into a working implementation. The algorithm runs in time $O(n^3)$, where n is the number of vertices in the input graph. It also requires $O(n^2)$ space for its two tables.

3.1 Interval graphs

Interval graphs is a graph class modeling overlapping intervals on a real line, for example a timetable of social events (see figures 3.1 and 3.2).

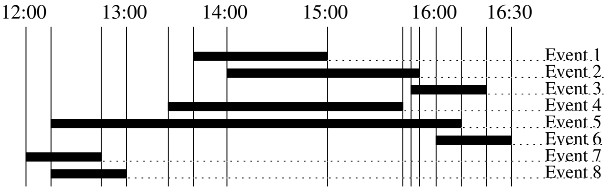


Figure 3.1: Interval Graph example 1

> This representation of an interval graph shows a timetable of available social events. Each black line indicate one event and is a vertex of the interval graph. Horizontally overlapping lines will be adjacent vertices in the graph.

3.1.1 Definition

A graph $G = (V, E)$ is an interval graph if sets of consecutive integers (intervals) can be assigned to each vertex such that the intervals $I(u)$ and $I(v)$ overlap if and only if $(u, v) \in E$.

It is possible to generate the list of edges by knowing only the intervals of the vertices, and to generate a list of the intervals of the vertices by knowing

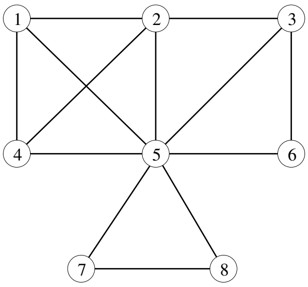


Figure 3.2: Interval Graph example 2

The intervals from Figure 3.1 pictured as a graph. The numbering of the vertices correspond to the ordering of the events, i.e. vertex 3 corresponds to *event 3* and so on.

only the edges. Therefore we will sometimes refer to both interval i and vertex i as v_i.

More formally, a graph $G = (V, E)$ is an interval graph if there exists an interval $I(v)$ for each $v \in V$ and $I(u) \cap I(v) \neq \emptyset$ for all $(u, v) \in E$ and $I(u) \cap I(v) = \emptyset$ for all $(u, v) \notin E$. Interval graphs can be recognised and the corresponding intervals of the vertices of the graph can be constructed in linear time [20]. The interval graph property is hereditary [21], thus induced subgraphs of interval graphs are also interval graphs. A graph G is a *proper* interval graph if G is an interval graph where no interval is completely contained in another vertex, i.e. $\nexists u, v \in V(G) : u \cap v = u$.

We will refer to the lowest and highest interval endpoints of a vertex v as the left $(l(v))$ and right $(r(v))$ endpoints of v, respectively. An interval graph will always have one or more leftmost vertices, the vertices with the lowest left endpoint, and one or more rightmost vertices, the vertices with the highest right endpoint. The *left eccentricity*, $e_l(v)$, of an interval v denotes the length of the shortest path from v to the leftmost vertex, and the *right eccentricity*, $e_r(v)$, of an interval v denotes the length of the shortest path from v to the rightmost vertex. The vertices of an interval graph can be sorted in non-decreasing order of their left endpoints in linear time[1]. Let $V = \{v_1, v_2, \ldots, v_n\}$ be the sorted order of the vertices of a given interval graph $G = (V, E)$. We will always assume that the vertices of a given interval graph are sorted in this manner. We define G_{ij} to be the subgraph of G induced by vertices $v_i, v_{i+1}, \ldots, v_j$, with $1 \leq i \leq j \leq n$.

We will refer to the left endpoint of a set of vertices S as the minimum of all left endpoints of the vertices in S, and vice versa for the right endpoint of S. Thus, $l(V) = \min_{v \in V}\{l(v)\}$ and $r(V) = \max_{v \in V}\{r(v)\}$.

3.1.2 Useful properties

The class of interval graphs has some useful properties that can be exploited both in the algorithm and in the correctness tests. We have observed and proved the following Claim 3.1 and Lemma 3.2 which will be used in Chapter 6.

Claim 3.1. *For any interval graph $G = (V, E)$, there exists a path $P \subseteq$*

G such that $|E(P)| = diam(G)$ and every $v \in V$ overlaps with the interval $[l(V(P)), r(V(P))]$.

Proof. Given an interval graph $G = (V, E)$ sorted by non-decreasing left endpoints, we choose P such that $|E(P)| = diam(G)$ and $r(V(P)) - l(V(P))$ is maximised, and we assume the vertices in P to be numbered $p_1, p_2, \ldots, p_{|P|}$, also ordered by non-decreasing left endpoints. Note that this ordering may not correspond with their order in P.

It is obvious that any vertex v not adjacent to any vertex in P must be outside the endpoints of P. For simplicity, we assume that $r(v) < l(V(P))$. We will now show that we can create a path Q from v to $p_{|P|}$, such that $|E(Q)| > diam(G)$.

We define Q as the path connecting v and $p_{|P|}$. If $|E(Q)| < |E(P)|$, then $|E(P)| \neq diam(G)$, which is a contradiction of the definition of P. If $|E(Q)| = |E(P)|$, then $r(V(Q)) - l(V(Q)) < r(V(P)) - l(V(P))$, which is also a contradiction of the definition of P.

For $l(v) > r(V(P))$, the result is analogous.

Thus, if there exists a v outside the intervals of P, there must also exist a shortest path Q such that $|E(Q)| > |E(P)|$. By definition of $diam(G)$, this would imply that $|E(Q)| = diam(G)$, which again would contradict the fact that $|E(P)| = diam(G)$. □

Lemma 3.2. *For any interval graph G, $rad(G) = \left\lceil \frac{diam(G)}{2} \right\rceil$.*

Proof. Let $G = (V, E)$ be any interval graph and let $P \subseteq G$ be a path such that $|E(P)| = diam(G)$ and every $v \in V$ is adjacent to at least one $p \in P$. By Claim 3.1, such a P exists.

Recall the definition of $rad(G)$ as the minimum $e(v), v \in V$, and that the corresponding v is a central vertex. There exists no vertex $v \in V$ such that $e(v) < e(p)$ for any $p \in P$, because then P would not be a shortest path. We now know that the central vertex c in G must also be the central vertex in P, and thus the radius of G must be equal to the radius of P.

The radius of a path P is simply $\left\lceil \frac{|E(P)|}{2} \right\rceil$, and that gives us $rad(G) = \left\lceil \frac{diam(G)}{2} \right\rceil$. □

Note that for a general graph, Lemma 3.2 does not hold. For example, for a graph $G = C_4$, both the radius and the diameter is 2, as pictured in Figure 3.3.

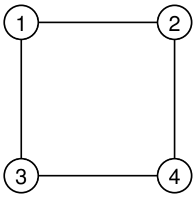


Figure 3.3: Lemma 3.2 does not hold for a C_4

$e(v_1) = e(v_2) = e(v_3) = e(v_4) = 2$, therefore $rad(G) = diam(G) = 2$.

The rest of this section is heavily based on [1], but the example at the end is new.

A broadcast with $f(v_i) > 0$ is, by definition, heard by exactly those intervals v_j where $d(v_i, v_j) < f(v_i)$. Therefore, [1] establishes a method for constructing shortest paths in interval graphs. Assuming that $i < j$, the shortest path $P = \{v_i = p_0, p_1, \dots, p_k = v_j\}$ that we will be using is defined such that p_{t+1} is chosen to be the vertex of $N(p_t)$ with largest right endpoint, for each t between 0 and $k - 2$. A formal algorithm for this process, which is called **SP**, is given below:

```
**Algorithm** Shortest Path (**SP**)
**Input:** An interval graph $G$, two vertices $v_i, v_j$ where $i \leq j$.
**Output:** A path $P$ between $v_i$ and $v_j$ containing a minimum number of edges.
$k = 0$
$p_k = v_i$
$P = \{p_k\}$
**while** $p_k \notin N(v_j)$ **do**
    Choose $p_{k+1}$ to be an interval in $N(p_k)$ with largest right endpoint
    $P = P \cup \{p_{k+1}\}$
    $k = k + 1$
**end while**
$P = P \cup \{v_j\}$
```

Corollary 3.2[1]. *Let f be a dominating broadcast function on G with $v_i \in V_f$. If v_j hears v_i, where $i < j$, then every vertex v_k hears v_i, for $i < k < j$.*

Corollary 3.3[1]. *If $d(v_i, v_k) = d(v_i, v_j) = t$ for some i, k, j satisfying $i < k < j$, then $d(v_i, v_q) = t$ for every q satisfying $k < q < j$.*

A shortest path P from v_i to v_j, where $j < i$, can be found in a similar manner as in algorithm **SP**; the only difference is that one selects an interval with the smallest left endpoint in each step. Due to the fact that we have ordered the intervals by their left endpoints, we do not get a similar result as Corollary 3.2[1] when $j < i$. This is reflected in the following observation:

Observation 3.4[1]. *Let $d(v_i, v_j) = t$ with $j < i$, and let P be a shortest path between v_i and v_j found as described above. Then for any k with $r(v_k) < l(p_{t-1})$, we have $d(v_i, v_k) > t$.*

It follows from Observation 3.4[1] that even if v_j hears v_i, there might exist a v_k with $j < k < i$, such that v_k does not hear v_i. For this situation to happen, we must have $l(v_j) \leq l(v_k)$ and $r(v_k) < l(p_{t-1})$, where $t = d(v_i, v_j)$. Both proper interval graphs and efficient broadcasts in general avoid this situation, as stated in the following corollary.

Corollary 3.5[1]. *Let f be a dominating broadcast function on G with $v_i \in V_f$. If G is proper interval or f is efficient, then the following is true: If v_j hears v_i with $j < i$, then every vertex v_k hears v_i, for $j < k < i$.*

For proper interval graphs we get a result analogous to Corollary 3.3[1].

Corollary 3.6[1]. *If G is proper interval and $d(v_i, v_k) = d(v_i, v_j) = t$ for some k, j, i where $k < j < i$, then $d(v_i, v_q) = t$ for each value q with $k < q < j$.*

[1] also present a result that resolves one of the open problems in [13]:

Theorem 3.7[1]. *For any proper interval graph G, $\gamma_b(G) = \gamma(G)$.*

The following example shows that the above theorem does not apply to interval graphs in general. Let $G = (V, E)$ be the interval graph defined by $V = \{a, b, c, d, e, f\}$ and $E = \{(a, b), (b, c), (c, e), (e, f), (c, d)\}$ (see fig. 3.4). With $f(c) = 2$ an optimal broadcast of total cost 2 is achieved. However, this cannot be achieved by assigning $f(v) = f(w) = 1$ for any pair of vertices $v, w \in V$.

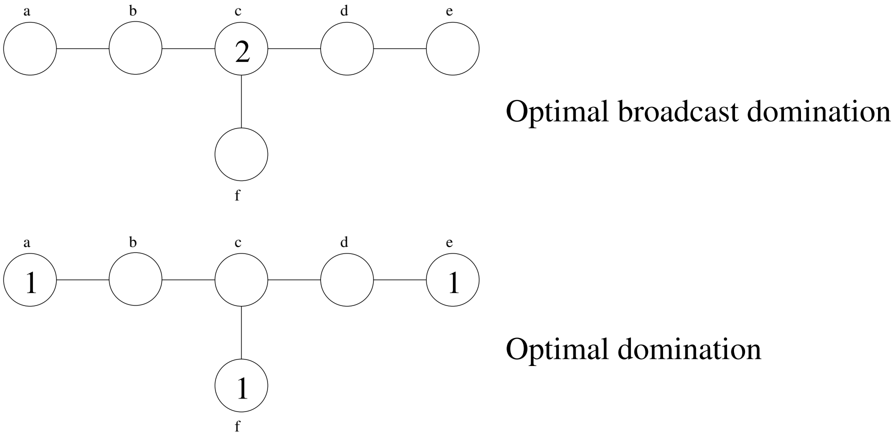


Figure 3.4: Why Theorem 3.7[1] does not apply to a general interval graph.

> We see that there exists a broadcast dominating set of cost $\gamma_b(G) = 2$, whereas no arrangement of the dominators can yield a *dominating set* of cost $\gamma(G) = 2$. Therefore $\gamma_b(G) = \gamma(G)$ does not hold for general interval graphs.

3.2 The algorithm

The following section and Section 3.2.1 are heavily based on [1], with a few minor corrections. The pseudocode for **MLD** and **MinRad**, as well as the examples has been added as part of this work.

We now present the dynamic programming algorithm for computing an optimal efficient broadcast on an interval graph G in $O(n^3)$ time. This time complexity is achieved with the help of an $O(n^2)$ time preprocessing that computes radial broadcasts on all subgraphs G_{ij}. Although here we only compute the cost of an optimal solution, extending the results to compute the broadcast function itself is theoretically straightforward and does not increase the time complexity. In the implementation of the algorithm, we have added functionality for outputting the broadcast function.

It follows from corollaries 3.2[1] and 3.5[1] that an optimal, efficient broadcast consists of a set of broadcast dominators that each dominate exactly one subgraph G_{ij} and nothing outside of G_{ij}. Since the solution is efficient, there is no overlap between these graphs. Noting that in a connected graph we will never have a broadcast dominator that dominates only itself, we get the following recursion for finding a minimum cost *non-radial* efficient solution for G_{ij}:

$$\gamma = \min_{i<k<j} Opt(i,k) + Opt(k+1,j)$$

The minimum cost of dominating G_{ij} and nothing else is then given by

$$Opt(i,j) = \min\{MinRad(i,j), \gamma\}$$

An example of this equation is illustrated in Figure 3.5.

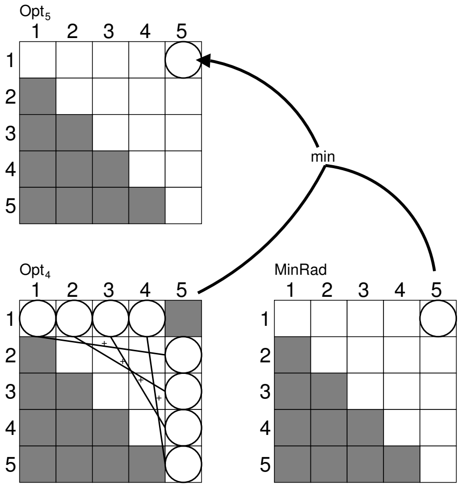


Figure 3.5: How the algorithm calculates the final solution for a graph of 5 vertices.

> For all three tables, the position (row, col) indicates the induced subgraph of all vertices $v_i, row \leq i \leq col$. The tables Opt_5 and Opt_4 are actually the same table *Opt*, but have been separated for illustration purposes. The optimal cost $\gamma_b(G)$ will in this case be the value stored in $Opt(1,5)$. As indicated by the figure, the value in $Opt(1,5)$ is calculated as $Opt(1,5) = \min\{\ Opt(1,1) + Opt(2,5)$, $Opt(1,2) + Opt(3,5)$, $Opt(1,3) + Opt(4,5)$, $Opt(1,4) + Opt(5,5)$, $MinRad(1,5)\ \}$.

Note that it might happen that there does not exist an efficient optimal solution for one or both of G_{ik} and $G_{k+1,j}$ that cannot be heard from outside of these subgraphs. If this is the case we set $Opt(i,j) = \infty$. If we consider G_{ik} then an efficient optimal solution exists if and only if G_{ik} can be decomposed into one or more non-overlapping graphs, each dominated by exactly one broadcast dominator that cannot be heard from outside of this subgraph.

Initially, we determine for each i and j, $i \leq j$, whether there exists a radial broadcast for G_{ij} that cannot be heard from outside of G_{ij}. If a radial solution exists, then its cost is stored in $MinRad(i,j)$. If there is no such solution then $MinRad(i,j) = \infty$. The following $O(n^3)$ time dynamic programming algorithm, which is called Interval Broadcast Domination (**IBD**), progresses by building efficient optimal solutions to all G_{ij} containing l intervals before moving to graphs containing $l+1$ intervals. The correctness and the $O(n^3)$ time complexity of **IBD** follows from the above discussion. Some variables has been renamed from [1] to make the pseudocode clearer.

```
**Algorithm** Interval Broadcast Domination (**IBD**)
**Input:** An interval graph $G$
**Output:** $\gamma_b(G)$
Compute all radial solutions and place them in a table $MinRad$.
**for** $v = 1$ to $n - 1$ **do**
    $Opt(v, v + 1) = MinRad(v, v + 1)$
**end for**
**for** $l = 2$ to $n - 1$ **do**
    **for** $v = 1$ to $n - l$ **do**
        $\gamma = \min_{u=v+1}^{v+l-2}\{Opt(v, u) + Opt(u + 1, v + l)\}$
        $Opt(v, v + l) = \min\{MinRad(v, v + l), \gamma\}$
    **end for**
**end for**
**Return** $Opt(1, n) = \gamma_b(G)$
```

3.2.1 Computing radial solutions

We now describe how $MinRad(i, j)$, a radial broadcast needed to dominate exactly G_{ij}, can be computed efficiently for all i, j. If no such solution exists, i.e. if all radial solutions can be heard from outside of G_{ij}, then $MinRad(i, j) = \infty$.

We first consider what part of the graph would be dominated if we were to set $f(v_k) = t$, $t > 0$. Let v_i and v_j be the lowest and highest numbered vertices that are dominated by $f(v_k)$, respectively. It follows from corollaries 3.2[1] and 3.5[1] that every vertex in G_{ij} must hear $f(v_k)$ if we are going to consider v_k as a candidate for a radial broadcast dominator. We will look at G_{ik} and G_{kj} separately starting with G_{kj}.

Since any sub-path of a shortest path is also a shortest path, it follows that if $v_k = p_0, p_1, \dots, p_t = v_j$ with $t > 1$, is a shortest path from v_k to v_j given by Algorithm **SP**, where $k < j$, then there is a shortest path of length $t-1$ from p_1 to v_j. Similarly, if there is a shortest path of length $t-1$ from p_1 to v_j then there is a shortest path from v_k to v_j of length $t-1$ if $v_k p_1 \in E$; otherwise the shortest length is t. In terms of broadcast domination this means that $f(v_k) = t$ will dominate exactly the same set of vertices in $G_{p_1,n}$ as $f(p_1) = t-1$. Assuming that $k < p_1$, by Corollary 3.2[1], every vertex in G_{k+1,p_1-1} will also hear $f(v_k) = t$. Thus if v_j is the highest numbered vertex that can hear $f(v_{p_1}) = t-1$ then $f(v_k) = t$ will dominate $G_{k+1,j}$ and no vertex in $G_{j+1,n}$. Setting $f(v_k) = 1$ will dominate $G_{k+1,j}$ where v_j is the highest numbered vertex with $l(v_j) \leq r(v_k)$. This value can be found by searching through the neighbors of v_k.

3.2.2 The complete algorithm

The complete algorithm, called Maximal Right Domination (**MRD**), is given below. This algorithm returns a table $R(k, f(v_k))$ containing the maximum value $q \in H(v_k)$ for each $v_k \in V$ and $1 \leq f(v_k) \leq e_r(v_k)$. In order to perform these computations efficiently, the vertices should be processed in order of decreasing left endpoints. For each vertex v_k we search for the highest numbered adjacent vertex in order to determine the reach of $f(v_k) = 1$. We then search for its neighbor v_q with the largest right endpoint. From this we copy the values for how far $f(v_q) = t$ will reach in G_{qn}, $1 \leq t \leq e(v_q)$.

```
Algorithm Maximal Right Domination (MRD)
Input: An interval graph $G$, ordered by nondecreasing left endpoints
Output: A table $R(k, f(v_k)) = \max\{q|v_k \in H(v_q)\}$ with $1 \leq f(v_k) \leq e_r(v_k)$ for each $v_k \in V$.
for $k = n$ downto 1 do
    $R(k, 1) = \max\{q|l(v_q) \leq r(v_k)\}$
    Choose $p_1$ to be an interval in $N(v_k)$ with largest right endpoint.
    for $i = 1$ to $e_r(p_1)$ do
        $R(k, i + 1) = R(p_1, i)$
    end for
end for
Return $R$
```

Determining the value of $R(k, 1)$ for each v_k has an accumulated cost of $O(m) = O(n^2)$. Since at most $n - 1$ values are set for each interval, the overall time for **MRD** is $O(n^2)$.

The computation of which lower numbered vertices will be dominated by $f(v_k) = t$ is similar. The main difference is that not every f-value on a vertex might yield a broadcast that can be used in an efficient solution. Assume $f(v_k) = 1$ and that v_q is the lowest numbered vertex adjacent to v_k, $q < k$. Then, from Corollary 3.5[1] it follows that if there is any vertex in $G_{q+1,k-1}$ not adjacent to v_k then $f(v_k) = 1$ cannot be used in building an efficient solution. To obtain values for $f(v_k) > 1$, it is sufficient to copy these from the same v_q. This follows since v_k will dominate the same vertices using power t as v_q does with power $t - 1$. And if $f(v_q) = t - 1$ cannot be used in an efficient solution then neither can $f(v_k) = t$. The total computation can be carried out in time $O(n^2)$.

Combining the results for which vertices $f(v_k)) = t$ will dominate, we can decide whether this can be used in a radial solution of some G_{ij}. If this is the case and t is lower than the previous lowest $f()$ value used in this graph, we set $MinRad(i, j) = t$. For all values of i and j where there does not exist a radial solution that cannot be heard from outside of G_{ij}, we set $MinRad(i, j) = \infty$. It follows that the whole process can be carried out in time $O(n^2)$.

The algorithm Maximal Left Domination (**MLD**) and the algorithm for calculating the *MinRad* table (**MinRad**) were not formalised in [1] and have been developed according to their textual descriptions:

```
Algorithm Maximal Left Domination (MLD)
Input: An interval graph $G$, ordered by nondecreasing left endpoints
Output: A table $L(k, f(v_k)) = \min\{q|v_k \in H(v_q)\}$ with $1 \leq f(v_k) \leq e_r(v_k)$ for each $v_k \in V$.
for $k = 1$ to $n$ do
    $q = \min\{q|r(v_q) \geq l(v_k)\}$
    $L(k, 1) = v_q$
    for $i = L(k, 1) + 1$ to $k - 1$ do
        if $r(v_i) < l(v_k)$ then
            $L(k, 1) = \infty$
            break loop
        end if
    end for
    for $i = 1$ to $e_l(v_q)$ do
        $L(k, i+1) = L(q, i)$
    end for
end for
Return $L$
```

```
Algorithm Minimal Radial Solutions (MinRad)
Input: An interval graph $G$
Output: A table $MinRad$, containing all radial solutions.
Order $G$ by left endpoints.
$R = MRD(G)$
$L = MLD(G)$
for $i = 1$ to $n$ do
    for $j = i$ to $n$ do
        $MinRad(i, j) = \infty$
    end for
end for
for $k = 1$ to $n$ do
    for $l = \max\{e_l(k), e_r(k)\}$ downto 0 do
        $i = L(k, \min\{l, e_l(k)\})$
        $j = R(k, \min\{l, e_r(k)\})$
        if $MinRad(i, j) > l$ then
            $MinRad(i, j) = l$
        end if
    end for
end for
Return $MinRad$
```

An example of a *MinRad* table is shown in Table 3.1.

3.2.3 Corrections

After implementing the algorithm and testing it on teeth graphs (described in Chapter 6), the algorithm turned out to yield incorrect output. By discussing this problem with Pinar Heggernes, it turned out to be a misprint in the description of the **MRD** algorithm. While [1] stated that the vertices should be ordered by non-decreasing right endpoints, ordering the vertices by non-decreasing left endpoints made the algorithm give correct output.

3.3 Implementation details

In addition to the actual algorithms shown here, we had to implement a method of reading an interval graph from the standard input.

We chose to simply read and store the interval graph as a list of intervals and automatically create an adjacency matrix. Since we will only use the tables *Opt* and *MinRad* in the main algorithm, the interval list and adjacency matrix is freed from memory after the calculation of *MinRad* is complete.

The tables L and R, as specified in the algorithm, are of height n and with different width for each vertex. The width for the table at a vertex v would be $e_l(v)$ for the L table or $e_r(v)$ for the R table. However, this will in practice give

MinRad	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*
1	∞	∞	1	∞	∞	∞	2	1	2	2	3
2		∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
3			∞	∞	∞	1	1	∞	∞	∞	∞
4				∞	∞	∞	∞	∞	∞	∞	∞
5					∞	∞	∞	∞	∞	∞	∞
6						∞	∞	1	∞	∞	∞
7							∞	∞	1	∞	2
8								∞	∞	1	∞
9									∞	∞	1
10										∞	∞
11											∞

Table 3.1: Example of a $MinRad$ table

This is the result from running **MinRad** where the input G is the interval graph from Figure 3.8. For clarity, the fields of $MinRad(i, j)$ where $i > j$ have been left blank. See how this table indicates that there exists a radial solution of cost 3, and that a radial solution on each of the subgraphs $G_{1,8}$ and $G_{9,11}$ is a solution of cost 2.

an overhead in memory management which we would like to avoid. We have thus declared each table to be of constant width $diam(G)$. This will require a bit more memory, but it does not exceed the $O(n^2)$ space usage of the algorithm, and the tables L and R can be discarded after calculating the $MinRad$ table. The calculation of diameter is done in linear time, by a trivial extension of the shortest path algorithm presented in Section 3.1.2.

The added functionality for calculating the broadcast function is done by keeping track of how each value is calculated. In the case of the $MinRad$ table, we are keeping track of which vertex the radial solution originates from, and for each cell (i, j) of the Opt table, which other cells in the Opt table or the $MinRad$ table turned out to give us the value in (i, j). This results in a binary tree structure where the root is the optimal cost and the leaves are references to the $MinRad$ table. By traversing this structure to find the leaves, we will know which exact vertices are used in the optimal dominating broadcast, and how much broadcast power is assigned to each vertex.

3.4 Test run

We will now show an example of how the algorithm will run on a modified version (Figure 3.6) of the interval graph pictured in Figure 3.2.

For clarity, we will first assign arbitrary matching intervals to the vertices and sort the vertices by ascending left endpoints, which results in the list of vertices $v_1 = [1, 1], v_2 = [1, 2], v_3 = [1, 7], v_4 = [3, 4], v_5 = [3, 5], v_6 = [4, 6], v_7 = [6, 7], v_8 = [7, 8], v_9 = [8, 9], v_{10} = [9, 10], v_{11} = [10, 11]$, illustrated in Figure 3.7.

For convenience, we have stored the list of intervals in a file “testgraph”. Now we will run the algorithm, with the $-b$ option to indicate that we also

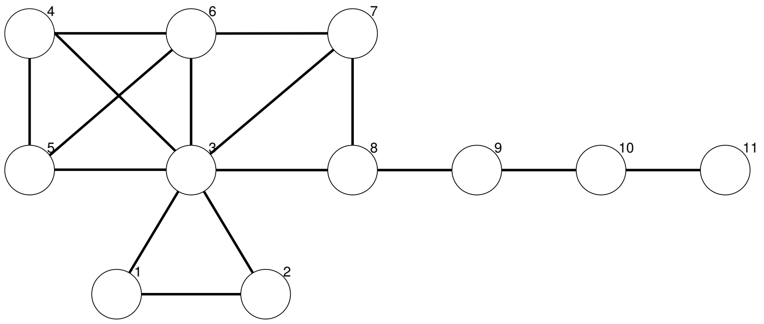


Figure 3.6: Sample input graph

A modified version of the graph from Figure 3.2. The original graph would only need a single radial broadcast of power 1 originating at *event 5* to be optimally dominated, which would not result in a very interesting example. The numbers shown next to the vertices correspond to their ordering by increasing left endpoints.

```
11        // 11 intervals in graph
~
1 1       // o.......... (event 7)
1 2       // oo......... (event 8)
1 7       // o-----o.... (event 5)
3 4       // ..oo....... (event 1)
3 5       // ..o-o...... (event 4)
4 6       // ...o-o..... (event 2)
6 7       // .....oo.... (event 3)
7 8       // ......oo... (event 6)
8 9       // .......oo.. (new)
9 10      // ........oo. (new)
10 11     // .........oo (new)
```

Figure 3.7: Input data for test run

Note that only the numbers on the left side are passed to the program. The input format can be displayed by running the program with the `--help` option.

want the program to output the broadcast function in addition to the optimal cost.

```
# ./interval-graph-broadcast-domination -b < testgraph
Minimum Broadcast Domination: 2
Broadcast function f(v)=Z*:
f(1)=0 f(2)=0 f(3)=1 f(4)=0 f(5)=0 f(6)=0 f(7)=0 f(8)=0
f(9)=0 f(10)=1 f(11)=0
#
```

We can see in Figure 3.8 that this is indeed a dominating broadcast.

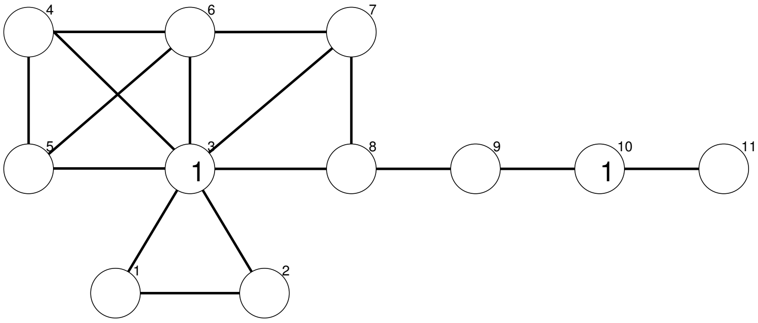


Figure 3.8: Optimal solution to test graph

The results from the test run of the algorithm, plotted into the input graph from Figure 3.6.

Chapter 4

Trees

In this chapter we will start by describing the trees class and the corresponding algorithm for Minimum Broadcast Domination, the details of turning the algorithm into a working implementation, and an example of how the implementation runs on a small tree graph.

The algorithm runs in time $O(nh)$, where n is the number of vertices and h is the height of the input tree. It also requires $O(nh)$ space to store its n tables.

4.1 Trees

Trees are commonly used to represent completely hierarchical structures, such as computer file systems, Internet forums and also robots (see Figure 4.2). However, trees are not inherently hierarchical in structure, and Figure 4.1 shows an unrooted tree.

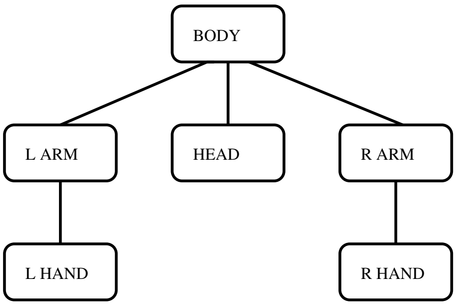


Figure 4.1: Tree example 1

A rooted tree modeling a simple robot.

4.1.1 Definition

A graph G is a tree if and only if it satisfies one of the following equivalent conditions.

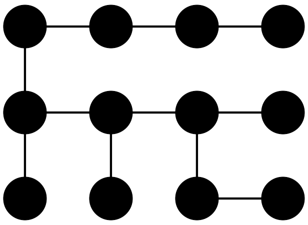


Figure 4.2: Tree example 2

A simple, unrooted tree.

- G is connected and has no simple cycles.
- G has no simple cycles and, if any edge is added to G, then a simple cycle is formed.
- G is connected and, if any edge is removed from G, then it is not connected anymore.
- Any two vertices in G are connected by a unique simple path.
- G contains no cliques of 3 or more vertices.

Note that as all these conditions are equal, it is impossible to satisfy one condition without satisfying all five.

When the tree is made to represent a hierarchical structure, one vertex is selected to be the *root* of the tree. Any tree with a root is a *rooted tree*, and it is assumed that any tree mentioned in this work is a rooted tree unless otherwise specified.

Rooted trees can be defined recursively as: A tree T is either empty or consists of a root vertex and terminal r and a list of subtrees $T_1, T_2, T_3, \ldots, T_k$ where $k \geq 0$. The root vertices of the subtrees are each connected to r by an edge. We call them the *children* of r, and r is the *parent* of its children. Any vertex with no children is called a *leaf*.

We define the *height* of a tree as the eccentricity of the root, i.e. the distance from the root to the farthest leaf. The height of the tree pictured in Figure 4.1 is 2, while the height of the tree pictured in Figure 4.2 can be anything from 4 (the radius of the graph) to 8 (the diameter of the graph) inclusive, depending on which vertex is selected as root.

4.1.2 Useful properties

If the root of the tree is set to be a central vertex, i.e. one with the smallest eccentricity, the tree will be of minimum height. Since the running time is dependent on the tree height, the central vertex should always be chosen as root when passing the input tree to the algorithm.

4.2 The algorithm

The following algorithm was first described in [1] and has been used in this work with no modifications, except for some corrections to the pseudocode.

Let T be a tree with root v_r. We denote the subtree rooted at vertex v by T_v, and hence $T = T_{v_r}$. For a subtree T_v and efficient broadcast domination f of T, we have the following three possibilities for v, the root of T_v: 1) $f(v) > 0$, which will cover v and vertices both in T_v and in other parts of T through the parent edge of v; 2) $f(v) = 0$ and v hears some broadcast originating in T_v; and 3) $f(v) = 0$ and v hears some broadcast originating outside of T_v through the parent edge of v.

Recall from Chapter 2 the notions of underdomination ($dom_k < 0$) and overdomination ($dom_k \geq 0$). Let $cost_v$ be an array associated with vertex v and indexed from $-h$ to h inclusive. The value of $cost_v[i]$ will be the minimum cost for an efficient broadcast domination of T_v with domination condition i. The algorithm will compute the $cost_v$ values in a bottom-up fashion. Elements of $cost_{v_r}[i]$ for $i < 0$ are not considered when reporting the solution since these represent configurations where $T_{v_r} = T$ contains vertices which hear no broadcast.

Suppose v is a leaf. Then the $cost_v[i] = 0$ for $i < 0$, since an efficient broadcast domination f that covers v through its parent will not have a broadcast originating at v. Note that there is no efficient broadcast function on a single vertex that has an exact domination. Hence, we set $cost_v[0] = \infty$. Finally, the only way we can achieve a proper overdomination of value i for leaf v is to set $f(v) = i$. A complete formula for $cost_v$ when v in a leaf is given below:

$$cost_v[i] = \begin{cases} 0 & \text{if } i < 0, \\ \infty & \text{if } i = 0, \\ i & \text{if } i > 1 \end{cases}$$

Now consider computation of the cost vector $cost_v[i]$ for T_v where v has children $v_1, v_2, \ldots, v_c$. When $i < 0$, an underdomination of i in T_v is equivalent to an underdomination of $i + 1$ in each T_{v_k}, $1 \leq k \leq c$ (exact domination if $i + 1 = 0$). Thus, the cost incurred for T_v is the sum of $cost_{v_k}[i + 1]$ over all children v_k of v. When $i = 0$, there cannot be a broadcast originating at v, and since we require f to be efficient, exactly one child subtree of v must have a proper overdomination of 1, and all other child subtrees must have exact domination. If $i > 0$, either $f(v) = i$, or $f(v) = 0$ and exactly one child subtree of v has proper overdomination $i+1$. If $f(v) = i$, then each of the child subtrees must have underdomination of $-i$. If, on the other hand, exactly one child subtree has proper overdomination $i+1$, then all other child subtrees must have underdomination $-i$. These relationships are formalised in the equations given below when v is not a leaf.

$$BestChild_v(i) = \min_{1 \leq k \leq c} \left\{ cost_{v_k}[i+1] + \sum_{1 \leq j \leq c, j \neq k} cost_{v_j}[-i] \right\}$$

$$cost_v[i] = \begin{cases} \sum\limits_{1 \le k \le c} cost_{v_k}[i+1] & \text{if } i < 0, \\ \min\limits_{1 \le k \le c} \left\{ cost_{v_k}[1] + \sum\limits_{1 \le j \le c, j \ne k} cost_{v_j}[0] \right\} & \text{if } i = 0, \\ \min \left\{ \left(i + \sum\limits_{1 \le k \le c} cost_{v_k}[-i] \right), BestChild_v(i) \right\} & \text{if } i > 0 \end{cases}$$

The dynamic programming approach described above results in the following algorithm **TBD**. **TBD** computes the $cost_v$ array of each vertex v in T, starting from the leaves, and processing a vertex only after the cost arrays of its children have been computed.

```
Algorithm Tree Broadcast Domination (TBD)
Input: A tree $T$ rooted at a center vertex $v_r$.
Output: $\gamma_b(T)$.
for every vertex $v$ in $T$ (traversed in post-order) do
  for $i = -h$ to $h$ do
    $Sum(i) = 0$;
    $InfinityCount(i) = 0$;
    for each child $v_k$ of $v$ do
      if $cost_{v_k}[i] = \infty$ then
        $InfinityCount(i) = InfinityCount(i) + 1$;
      else
        $Sum(i) = Sum(i) + cost_{v_k}[i]$;
      end if
    end for
  end for
  $cost_v[-h] = Sum(-h+1)$
  for $i = -h+1$ to $-1$ do
    if $InfinityCount(i+1) \leq 0$ then
      $cost_v[i] = Sum(i+1)$;
    else
      $cost_v[i] = \infty$
    end if
  end for
  $cost_v[0] = \infty$;
  if $InfinityCount(0) \leq 1$ then
    for each child $v_k$ of $v$ do
      if $cost_{v_k}[0] = \infty$ then
        $cost_v[0] = \min\{cost_v[0], cost_{v_k}[1] + Sum(0)\}$;
      else if $InfinityCount(0) = 0$ then
        $cost_v[0] = \min\{cost_v[0], cost_{v_k}[1] + Sum(0) - cost_{v_k}[0]\}$;
      end if
    end for
  end if
  for $i = 1$ to $h-1$ do
    $BestChild(i) = \infty$;
    if $InfinityCount(-i) \leq 1$ then
```

```
      for each child $v_k$ of $v$ do
        if $cost_{v_k}[-i] = \infty$ then
          $BestChild(i)$ =min$\{BestChild(i), cost_{v_k}[i+1] + Sum(-i)\}$;
        else if $InfinityCount(-i) = 0$ then
          $BestChild(i)$ =min$\{BestChild(i), cost_{v_k}[i+1] + Sum(-i) - cost_{v_k}[-i]\}$;
        end if
      end for
    end if
    if $InfinityCount(-i) = 0$ then
      $cost_v[i]$ =min$\{(i + Sum(-i)), BestChild(i)\}$;
    else
      $cost_v[i] = BestChild(i)$;
    end if
  end for
  $cost_v[h] = h + Sum(-h)$;
end for
$\gamma_b(G)$ =min$_{0 \leq i \leq h}\{cost_{v_k}[i]\}$;
Return $\gamma_b(G)$
```

An example of the cost tables for an entire tree is shown in Figure 4.3.

The running time analysis can be found in [1], and will not be repeated here.

4.2.1 Corrections

After implementing and testing the original algorithm, we found that it yielded slightly incorrect answers to the test graphs because of some errors in the pseudocode: When filling in $cost_v$ for $i < 0$ and $i > 0$, the value ∞ as an element of a sum was treated as 0. A value of ∞ in the summation will now always result in the sum being ∞. Also, there were two typographical errors: An i had to be corrected to $-i$ and a v_k was corrected to v_{root}.

The pseudocode was then corrected, and the corrections were accepted by the authors of the algorithm.

4.3 Implementation details

Since the algorithm requires a rooted tree, we can represent the tree as a list of children for each vertex. The input format for the input tree was chosen to be this list, as it would require very little translation of data structures. This implies that the root of the tree must be given beforehand, instead of being optimally chosen by the implementation. The optimal root for an input tree would be a central vertex, since it minimises the height h of the tree. Though finding the central vertex of a tree can be done in linear time by repeatedly remove all leaves from the tree, it would require that the input tree was first given to the program as a general graph and then translated to a child list format before passing it to the algorithm, generating much overhead.

The implementation has been extended to also calculate and return the broadcast function f, if the user requests it. To calculate each $f(v)$, we will need to see which of the cost values for v were used in the optimal value. This can be done by remembering, for each index of each cost vector, which of the values from the cost vectors of the children were used in the final value of the

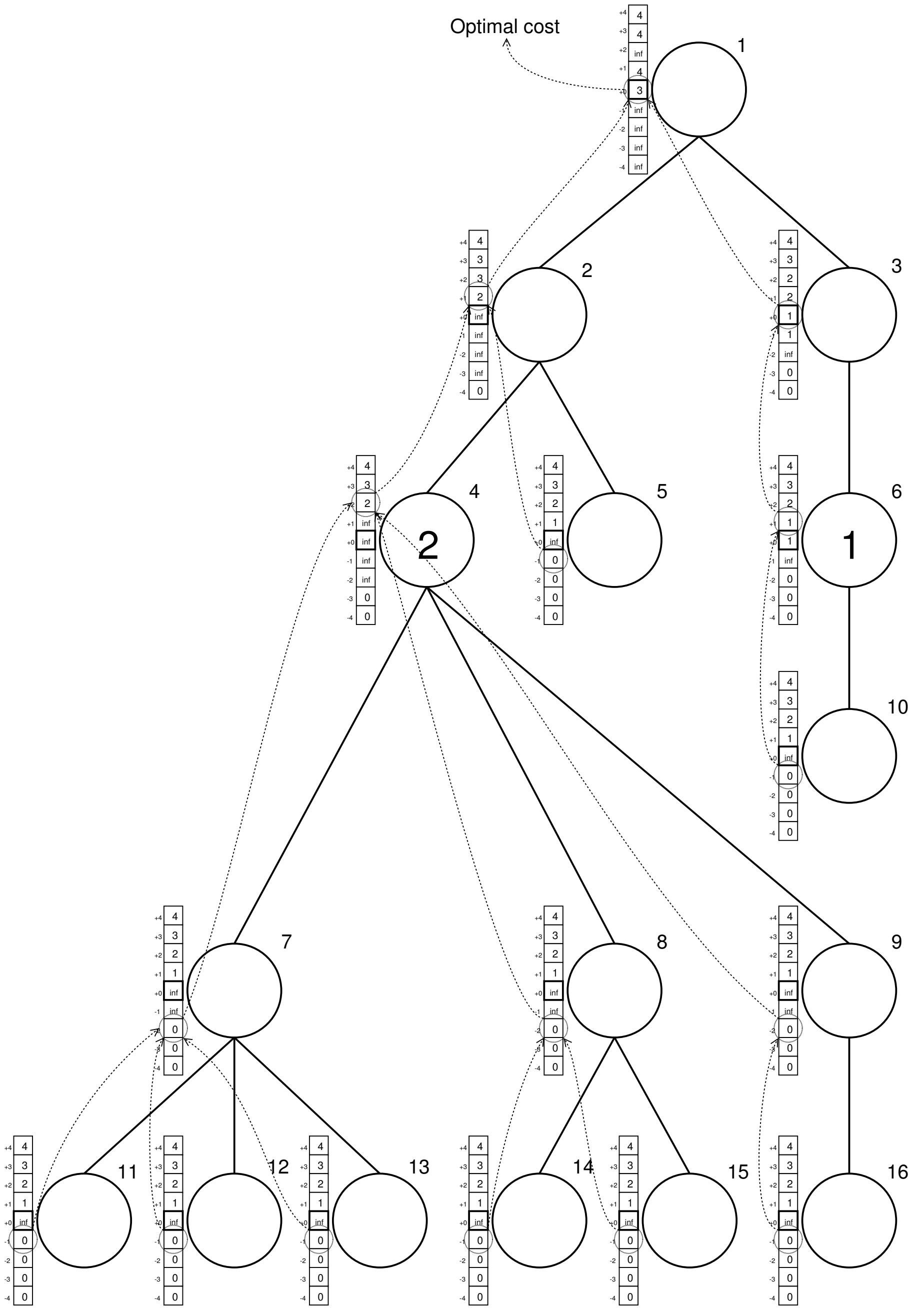

Optimal cost
1
+4 4 +3 4 +2 inf +1 4 +0 3 -1 inf -2 inf -3 inf -4 inf
2
+4 4 +3 3 +2 3 +1 2 +0 inf -1 inf -2 inf -3 inf -4 0
3
+4 4 +3 3 +2 2 +1 2 +0 1 -1 1 -2 inf -3 0 -4 0
4
2
+4 4 +3 3 +2 2 +1 inf +0 inf -1 inf -2 inf -3 0 -4 0
5
+4 4 +3 3 +2 2 +1 1 +0 inf -1 0 -2 0 -3 0 -4 0
6
1
+4 4 +3 3 +2 2 +1 1 +0 1 -1 inf -2 0 -3 0 -4 0
10
+4 4 +3 3 +2 2 +1 1 +0 inf -1 0 -2 0 -3 0 -4 0
7
+4 4 +3 3 +2 2 +1 1 +0 inf -1 inf -2 0 -3 0 -4 0
8
+4 4 +3 3 +2 2 +1 1 +0 inf -1 inf -2 0 -3 0 -4 0
9
+4 4 +3 3 +2 2 +1 1 +0 inf -1 inf -2 0 -3 0 -4 0
11
12
13
14
15
16
+4 4 +3 3 +2 2 +1 1 +0 inf -1 0 -2 0 -3 0 -4 0

Figure 4.3: A tree of size $n = 16$ with calculated cost tables

The cost tables are indexed with the overdominance values in the upper half and the underdominance values in the lower half. The dotted arrows and circled values indicate which values turn out to be part of the optimal solution and which other values they are calculated from, and are used to determine the broadcast function for the tree (also plotted in). Note that the cost vectors are calculated in post-order, and the dotted arrows are calculated afterwards, in pre-order.

current index and whether the value indicates a broadcast originating from the current vertex. Then we will have, for each cost index of the root, a tracing tree whose vertices consist of exactly one cost index from each vertex in the input tree. By traversing the tracing tree rooted at one of the optimal values found at the root, we can extract the broadcast value from each vertex.

To save memory, the tracing trees are not calculated during the execution of the algorithm. Instead, the optimal tracing tree will be calculated on the fly by pretending to calculate the affected table values again from the top down. This will clearly not increase the asymptotic running time of the algorithm, as we will only repeat a small fraction of the original calculations, and tests (Chapter 7) of the implementation show it is practically bounded by memory rather than processing power, so sacrificing speed for memory is necessary to make the implementation able to handle as large problems as possible.

Also, to save memory, we could let the implementation allocate and deallocate calculation tables on the run. Thus, after calculating the tables for vertex v, we can free the tables of the children from the memory. If we also do not allocate memory for the tables of v until we actually get to v, the algorithm will usually be able to perform with less required memory. However, since we need to backtrack through these tables to retrieve the broadcast function, we have not found this optimisation useful enough to implement.

4.4 Test run

We will now run the algorithm on the tree from figure 4.3. We have constructed a list of children for each vertex, as listed in figure 4.4.

The list has been written to a file called “testgraph”. We will run the algorithm with the $-b$ option to indicate that we also want the program to output the broadcast function, not just the optimal cost.

```
# ./tree-broadcast-domination -b < testgraph
Minimum Broadcast Domination: 3
Broadcast function f(v)=Z*:
f(1)=0 f(2)=0 f(3)=0 f(4)=2 f(5)=0 f(6)=1 f(7)=0 f(8)=0 f(9)=0
f(10)=0 f(11)=0 f(12)=0 f(13)=0 f(14)=0 f(15)=0 f(16)=0
#
```

These broadcasts have already been plotted into Figure 4.3, and a careful examination of the figure will show that this is the only optimal dominating broadcast for this tree.

```
16             // 16 vertices in tree
2 2 3          // 1, children: 2, 3
2 4 5          // 2, children: 4, 5
1 6            // 3, children: 6
3 7 8 9        // 4, children: 7, 8, 9
0              // 5, leaf
1 10           // 6, children: 10
3 11 12 13     // 7, children: 11, 12, 13
2 14 15        // 8, children: 14, 15
1 16           // 9, children: 16
0              // 10, leaf
0              // 11, leaf
0              // 12, leaf
0              // 13, leaf
0              // 14, leaf
0              // 15, leaf
0              // 16, leaf
```

Figure 4.4: Input data for test run

This list corresponds to the graph pictured in Figure 4.3.

Only the numbers to the left of the "//" will be passed to the program. The input format is displayed in detail by running the program with the `--help` option.

Chapter 5

Series-Parallel Graphs

In this chapter we will start by describing the graph class of series-parallel graphs, how the definition differs from the one given in [1], and the corresponding algorithm for Minimum Broadcast Domination. The rest of the chapter contains the details of turning the algorithm into a working implementation and an example of how it runs on a small series-parallel graph.

The algorithm runs in time $O(nr^4)$, where n is the number of vertices of the tree decomposition given as input, and r is the radius of the corresponding series-parallel graph. It also requires $O(nr^2)$ space for its $4n$ tables.

5.1 Series-parallel graphs

Series-parallel graphs is a recursively defined graph class which is most commonly used to model electric circuits. Of all the graph classes described in this work, series-parallel graphs are usually the most difficult to understand at first.

5.1.1 Definition

The class of series-parallel graphs is a graph class where each series-parallel graph G has 2 terminals, and each terminal is a vertex in G. To denote the left and right terminals of G, we use $t_L(G)$ and $t_R(G)$, respectively. Every series-parallel graph, except for the base graph K_2, is composed from exactly 2 smaller 2-terminal graphs (this is why it is called a recursive graph class) joined together by either a *parallel* operation or a *series* operation.

Let $G = (V, \{t_L(G), t_R(G)\}, E)$ and let $G_j = (V_j, \{t_L(G_j), t_R(G_j)\}, E_j)$ for $j = 1, 2$ be 2-terminal graphs. Define the *series* operation as $s(G_1, G_2) = G$ if $t_L(G_1) = t_L(G)$, $t_R(G_1) = t_L(G_2)$, and $t_R(G_2) = t_R(G)$. This operation associates $t_R(G_1)$ with $t_L(G_2)$ and then the resulting vertex loses its status as a terminal. Finally, define the *parallel* operation as $p(G_1, G_2) = G$ if $t_L(G_1) = t_L(G_2) = t_L(G)$ and $t_R(G_1) = t_R(G_2) = t_R(G)$. Note that in each case, the resulting graph G has two terminals. See Figure 5.1 for an example of a small series-parallel graph being constructed.

Note that, in [1], two more operations were described, namely *left jackknife* and *right jackknife*. These are not generally found in definitions of series-parallel graphs, and the corresponding formulas in the algorithms were also omitted. Therefore, they have been left out of this work as well.

The reversed construction of a series-parallel graph is called the decomposition. This decomposition can be described by a decomposition tree where the root of the tree is the original graph and the leaves are all K_2s, or edges. The tree identifies two children for each non-leaf vertex of the tree, and an associated operation (series or parallel) that joins the children to create the parent.

Series-parallel graphs can be recognised and a corresponding decomposition tree can be constructed in linear time [5]. See Figure 5.2 for an example of a decomposition tree for a series-parallel graph with 9 edges.

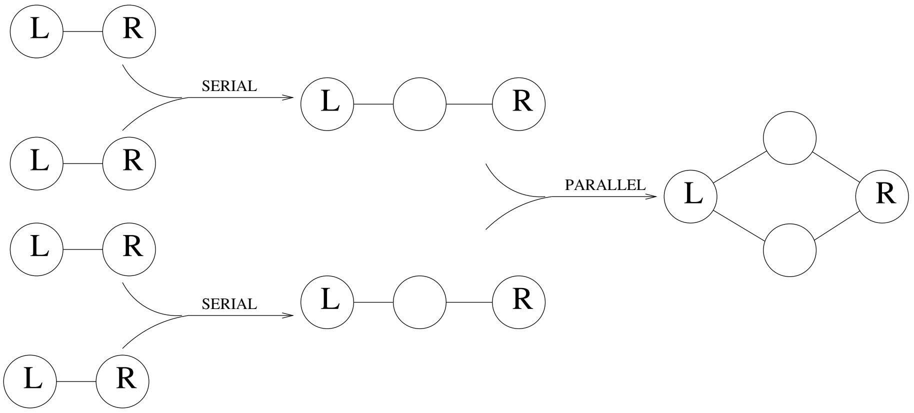


Figure 5.1: Illustration of series-parallel graph construction

A series-parallel graph C_4 constructed by recursively joining smaller series-parallel graphs.

5.1.2 Useful properties

Every series-parallel graph has a corresponding decomposition tree which is a rooted binary tree. The algorithm can work on this tree instead of the original graph, which makes the calculations simpler since it has a much cleaner structure.

5.2 The algorithm

The following section is taken more or less directly from [1] except for some minor corrections. The proofs for the lemmas and the algorithm have not been repeated here.

Given a broadcast function $f : V \to \{0, 1, 2, \ldots, r\}$, we say that dominance condition (dom_L, dom_R) exists if the following holds.

- f dominates G_i except for vertices within distance $-dom_k$ of $t_k(G_i)$ for $k \in \{L, R\}$ whenever $dom_k < 0$ and

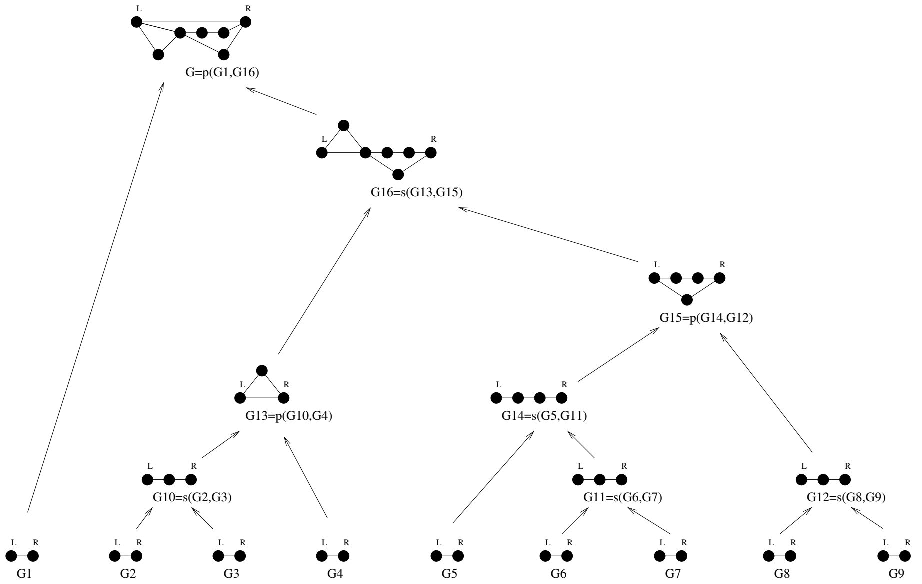


Figure 5.2: Decomposition tree example

A complete decomposition tree for the series-parallel graph G. L and R denote the t_L and t_R, respectively, of each vertex. Note that since we join exactly two terminals at each point, this is a binary tree.

- f would dominate hypothetical paths of length dom_k joined to $t_k(G_i)$ whenever $dom_k \geq 0$.

If $dom_k < 0$ we say $t_k(G_i)$ is *underdominated* and if $dom_k \geq 0$ we say $t_k(G_i)$ is *overdominated*. The definition of a dominance condition allows portions of G_i to be undominated at the current vertex in the decomposition tree only when there is a requirement (indicated by the negative dom_k value) for the underdominance be corrected at some point closer to the root of the decomposition tree.

Now we can define $P_{G_i}(dom_L, dom_R)$ to be the lowest cost of a broadcast function on G_i given that dominance condition (dom_L, dom_R) exists in G_i. P will be either N, L, R, or B to represent *neither*, *left*, *right*, or *both* terminals having a non-zero broadcast originating there, respectively. For example, $L_{G_3}(1, -2)$ represents the lowest cost of a broadcast function in G_3 that provides an overdominance of 1 at the left terminal, has an underdominance of -2 at the right terminal, has a broadcast originating at $t_L(G_3)$, and has no broadcast originating at $t_R(G_3)$.

The following properties of optimal broadcast dominations will be useful in establishing the correctness of our series-parallel algorithm.

Lemma 4.1[1]. *Let f be an optimal broadcast domination of a series-parallel graph G and consider a graph $G_i = (V_i, \{t_L(G_i), t_R(G_i)\}, E_i)$ in the decomposition of G. Let $d_{G_i}(t_L, t_R)$ be the distance in G_i between $t_L(G_i)$ and $t_R(G_i)$.*

- *If f is applied to G_i is such that $dom_L = f(t_L(G_i)) > 0$ and $dom_R < 0 =$*

$f(t_R(G_i))$, then $-(dom_R + d_{G_i}(t_L, t_R) + 1) < f(t_L(G_i))$.

- *If f is applied to G_i is such that $dom_R = f(t_R(G_i)) > 0$ and $dom_L < 0 = f(t_L(G_i))$, then $-(dom_L + d_{G_i}(t_R, t_L) + 1) < f(t_R(G_i))$.*

This means that a negative dom_R can never demand to have the vertices already dominated by dom_L dominated by some other broadcast and vice versa.

Lemma 4.2[1]. *In any optimal broadcast domination f of a graph $G = (V, E)$, if there are two neighbors $v, w \in V$ such that $f(v) > 0$ and $f(w) > 0$, then $f(w) = f(v)$.*

Given a series-parallel graph G, the algorithm computes the values of N_{G_i}, L_{G_i}, R_{G_i}, and B_{G_i} for each graph G_i in a decomposition tree of G, in a bottom-up fashion. Since an optimal broadcast function will have $f(v) \le r$ for all $v \in V$, we restrict the dominance conditions to ranges from $-r$ to r. The next subsection describes how to compute the N, L, R, and B arrays for leaves. Subsections 5.2.2 and 5.2.3 present the recursive relationships that allow the computation of $P_{G_i}(dom_L, dom_R)$ for $P \in \{N, L, R, B\}$ for every non-leaf graph G_i in the decomposition tree. Subsection 5.2.4 presents the main algorithm. The equations described in the three subsections are summarised in Tables 5.1, 5.2, and 5.3.

5.2.1 Initialisation

Recall that each leaf in the decomposition tree of a series-parallel graph corresponds to a K_2. The following discussion of how to initialise the four cost arrays N_{K_2}, L_{K_2}, R_{K_2}, and B_{K_2} is summarised with the equations in Table 5.1.

For the case of N_{K_2} (i.e. $f(t_L(K_2)) = f(t_R(K_2)) = 0$), no broadcast originates from within the K_2. Clearly, then, to lead to a valid broadcast domination of the series-parallel graph, the dominance condition of the K_2 must indicate a demand that will eventually dominate the K_2. The dominance conditions indicating such a demand are identified in the following observation.

Observation 4.3[1]. *Let f be a broadcast domination for a series-parallel graph G and consider an edge $K_2 = (V_{K_2}), \{t_L(K_2), t_R(K_2)\}, E_{K_2})$ corresponding to a leaf in the decomposition of G. If f applied to K_2 is such that $f(t_L(K_2)) = f(t_R(K_2)) = 0$, then $dom_L + dom_R \le -2$.*

We use the value ∞ to represent invalid dominance conditions. All other cases are set to 0, representing zero cost when no broadcast originates from the K_2.

Recall that L_{K_2} represents $f(t_L(K_2)) > 0$ and $f(t_R(K_2)) = 0$, in which case the cost due to the K_2 is simply $f(t_L(K_2))$. Thus, we must enter $f(t_L(K_2))$ for all dominance conditions where $dom_L = f(t_L(K_2))$ and dom_R together with dom_L lead to a valid broadcast domination of the final graph G.

Observation 4.4[1]. *Let f be a broadcast domination for a series-parallel graph G and consider an edge $K_2 = (V_{K_2}, \{t_L(K_2), t_R(K_2)\}, E_{K_2})$ corresponding to a leaf in the decomposition of G. If f is applied to K_2 is such that $f(t_L(K_2)) > 0$ and $f(t_R(K_2)) = 0$, then $dom_R < dom_L$.*

This observation gives us the first line in the equation for L_{K_2}; the second line is a result of Lemma 4.1[1]. The third line is not in the original algorithm

[1], but had to be added to make the algorithm yield correct results (more about this in Section 5.3). Initialisation of R_{K_2} is analogous.

Intuitively, we must set $B_{K_2}(dom_L, dom_R) = \infty$ whenever the dominance condition renders one of the values of dom_L and dom_R superfluous. The conditions when this occurs were formalised in Lemma 4.2[1].

$$N_{K_2}(dom_L, dom_R) = \begin{cases} 0 & \text{if } dom_L + dom_R \leq -2 \\ \infty & \text{if } dom_L + dom_R > -2 \end{cases}$$

$$L_{K_2}(dom_L, dom_R) = \begin{cases} \infty & \text{if } dom_R \geq dom_L \\ \infty & \text{if } dom_R \leq -(dom_L + 2) \\ \infty & \text{if } dom_L \leq 0 \\ dom_L & \text{otherwise} \end{cases}$$

$$R_{K_2}(dom_L, dom_R) = \begin{cases} \infty & \text{if } dom_L \geq dom_R \\ \infty & \text{if } dom_L \leq -(dom_R + 2) \\ \infty & \text{if } dom_R \leq 0 \\ dom_R & \text{otherwise} \end{cases}$$

$$B_{K_2}(dom_L, dom_R) = \begin{cases} dom_L + dom_R & \text{if } dom_L = dom_R > 0 \\ \infty & \text{otherwise} \end{cases}$$

Table 5.1: Initialisations for Algorithm **SPBD**

5.2.2 The series operation

The formulas discussed in this section are shown in Table 5.2.

Consider a vertex G in the decomposition tree that is formed by $G = s(G_1, G_2)$, and consider the computation of L_G. Note that allowable configurations with an originating broadcast at $t_L(G)$ and no originating broadcast at $t_R(G)$ must either come from B_{G_1} and L_{G_2} or from L_{G_1} and N_{G_2}. Other pairs either fail to have the specified originating broadcast, or are incompatible in the sense that $t_R(G_1)$ cannot be associated with $t_L(G_2)$ because one has an originating broadcast and the other does not.

For the case of B_{G_1} and L_{G_2}, $t_R(G_1)$ and $t_L(G_2)$ have an identical originating broadcast i. The formula simply compares these options for all of the relevant i values, and uses one of the minimum values as a candidate value for $L_G(dom_L, dom_R)$. Note that the cost of the originating broadcast at $t_R(G_1) = t_L(G_2)$ is subtracted since its cost is represented in both the cost of dominating G_1 and the cost of dominating G_2, but it only belongs once in the cost of dominating G.

For the case of L_{G_1} and N_{G_2}, no subtraction is needed since $t_R(G_1)$ and $t_L(G_2)$ have no originating broadcast. Now either G_1 provides overdomination that must at least cover any underdomination in G_2, or vice-versa, as represented by the last two lines in the formula for $L_G(dom_L, dom_R)$. When all of these cases are considered, $L_G(dom_L, dom_R)$ is assigned the lowest cost if a broadcast function in G with dominance condition (dom_L, dom_R) that has a broadcast originating at $t_L(G)$ but has no broadcast originating at $t_R(G)$.

Similar arguments justify the recursions given for N_G, R_G, and B_G.

$$N_G(dom_L, dom_R) = \min \begin{cases} \min\limits_{1\le i\le r} R_{G_1}(dom_L, i) + L_{G_2}(i, dom_R) - i \\ \min\limits_{0\le i\le j\le r-1} N_{G_1}(dom_L, j) + N_{G_2}(-i-1, dom_R) \\ \min\limits_{0\le i\le j\le r-1} N_{G_1}(dom_L, -i-1) + N_{G_2}(j, dom_R) \end{cases}$$

$$L_G(dom_L, dom_R) = \min \begin{cases} \min\limits_{1\le i\le r} B_{G_1}(dom_L, i) + L_{G_2}(i, dom_R) - i \\ \min\limits_{0\le i\le j\le r-1} L_{G_1}(dom_L, j) + N_{G_2}(-i-1, dom_R) \\ \min\limits_{0\le i\le j\le r-1} L_{G_1}(dom_L, -i-1) + N_{G_2}(j, dom_R) \end{cases}$$

$$R_G(dom_L, dom_R) = \min \begin{cases} \min\limits_{1\le i\le r} R_{G_1}(dom_L, i) + B_{G_2}(i, dom_R) - i \\ \min\limits_{0\le i\le j\le r-1} N_{G_1}(dom_L, j) + R_{G_2}(-i-1, dom_R) \\ \min\limits_{0\le i\le j\le r-1} N_{G_1}(dom_L, -i-1) + R_{G_2}(j, dom_R) \end{cases}$$

$$B_G(dom_L, dom_R) = \min \begin{cases} \min\limits_{1\le i\le r} B_{G_1}(dom_L, i) + B_{G_2}(i, dom_R) - i \\ \min\limits_{0\le i\le j\le r-1} L_{G_1}(dom_L, j) + R_{G_2}(-i-1, dom_R) \\ \min\limits_{0\le i\le j\le r-1} L_{G_1}(dom_L, -i-1) + R_{G_2}(j, dom_R) \end{cases}$$

Table 5.2: Equations for the $G = s(G_1, G_2)$ operation

5.2.3 The parallel operation

The formulas discussed in this section are shown in Table 5.3.

Consider the computation of $L_G(dom_L, dom_R)$ when $G = p(G_1, G_2)$. When $dom_L \ge 0$, we need to compare configurations in G_1 and G_2 where both graphs have the appropriate originating broadcast at t_L and where one graph provides dom_R at t_R. The other graph can then provide anything between $-(dom_R + 1)$ and dom_R at t_R, thereby guaranteeing that the resulting graph will provide dom_R at t_R. The other half of $L_G(dom_L, dom_R)$ (i.e. when $dom_L < 0$) is computed analogously.

Similar arguments justify the recursions given for N_G, R_G, and B_G.

5.2.4 The main algorithm

We have defined initialisations for leaves in a decomposition tree for a series-parallel graph, and recursive equations for computing the cost of an optimal broadcast domination for the series and parallel operations. An algorithm to compute the cost of an optimal solution for an entire graph G would simply find the lowest cost for which there is no underdominance in the $\{N, L, R, B\}$ arrays at the root of the decomposition tree. The details of this algorithm, which is called **SPBD**, are given below.

```
Algorithm Series-Parallel Broadcast Domination (SPBD)
Input: A series-parallel graph $G$ and a decomposition tree $T_d$ for $G$.
Output: $\gamma_b(G)$.
$r = rad(G)$
for each leaf $G_i$ in $T_d$ do
  Use the formulas in Table 5.1 to compute $N_{G_i}$, $L_{G_i}$, $R_{G_i}$, and $B_{G_i}$.
end for
repeat
  $G_i$ = an unprocessed vertex in $T_d$ whose children have been processed.
  if $G_i = s(G_1, G_2)$ then
    Use the formulas in Table 5.2 to compute $N_{G_i}$, $L_{G_i}$, $R_{G_i}$, and $B_{G_i}$.
  else if $G_i = p(G_1, G_2)$ then
    Use the formulas in Table 5.3 to compute $N_{G_i}$, $L_{G_i}$, $R_{G_i}$, and $B_{G_i}$.
  end if
until the root of $T_d$ has been processed.
$x = \infty$
for $i = 0$ to $r$ do
  for $j = 0$ to $r$ do
    $x = \min\{x, N_G(i,j), L_G(i,j), R_G(i,j), B_G(i,j)\}$
  end for
end for
Return $x = \gamma_b(G)$
```

[1] further proves the correctness of the algorithm and gives an upper bound on the running time with the following results.

Lemma 4.5[1]. *Every possible minimal broadcast function on a series-parallel graph G is considered during execution of Algorithm **SPBD**.*

Theorem 4.6[1]. *Given a series-parallel graph G, Algorithm **SPBD** computes an optimal broadcast domination function of G in $O(nr^4)$ time.*

We will not repeat the proof of the running time $O(nr^4)$ here, but note that, as every edge in the series-parallel graph is a leaf in the decomposition tree, and because the number of vertices in a binary tree containing l leaves is $2l - 1$, the size of the series-parallel decomposition can be described as $2m - 1$, where m is the number of edges in the series-parallel graph. This means we can give the running time of the algorithm as $O(mr^4)$, and consequently, the algorithm uses $O(mr^2)$ space.

$B_G(dom_L, dom_R) = B_{G_1}(dom_L, dom_R) + B_{G_2}(dom_L, dom_R) - dom_L - dom_R$
$L_G(dom_L, dom_R)$ $= \begin{cases} \min\limits_{-(dom_R+1)\le i\le dom_R} \left\{ \begin{array}{l} L_{G_1}(dom_L, dom_R) + L_{G_2}(dom_L, i) - dom_L \\ L_{G_1}(dom_L, i) + L_{G_2}(dom_L, dom_R) - dom_L \end{array} \right\} & \text{if } dom_R \ge 0 \\ \min\limits_{dom_R\le i\le -(dom_R+1)} \left\{ \begin{array}{l} L_{G_1}(dom_L, dom_R) + L_{G_2}(dom_L, i) - dom_L \\ L_{G_1}(dom_L, i) + L_{G_2}(dom_L, dom_R) - dom_L \end{array} \right\} & \text{if } dom_R < 0 \end{cases}$
$R_G(dom_L, dom_R)$ $= \begin{cases} \min\limits_{-(dom_L+1)\le i\le dom_L} \left\{ \begin{array}{l} L_{G_1}(dom_L, dom_R) + L_{G_2}(dom_L, i) - dom_L \\ L_{G_1}(dom_L, i) + L_{G_2}(dom_L, dom_R) - dom_L \end{array} \right\} & \text{if } dom_L \ge 0 \\ \min\limits_{dom_L\le i\le -(dom_L+1)} \left\{ \begin{array}{l} L_{G_1}(dom_L, dom_R) + L_{G_2}(dom_L, i) - dom_L \\ L_{G_1}(dom_L, i) + L_{G_2}(dom_L, dom_R) - dom_L \end{array} \right\} & \text{if } dom_L < 0 \end{cases}$
$N_G(dom_L, dom_R)$ $= \begin{cases} \min\limits_{\substack{-(dom_L+1)\le i\le dom_L \\ -(dom_R+1)\le j\le dom_R}} \left\{ \begin{array}{l} N_{G_1}(dom_L, dom_R) + N_{G_2}(i, j) \\ N_{G_1}(dom_L, j) + N_{G_2}(i, dom_R) \\ N_{G_1}(i, dom_R) + N_{G_2}(dom_L, j) \\ N_{G_1}(i, j) + N_{G_2}(dom_L, dom_R) \end{array} \right\} & \text{if } dom_L, dom_R \ge 0 \\ \min\limits_{\substack{-(dom_L+1)\le i\le dom_L \\ dom_R\le j\le -(dom_R+1)}} \left\{ \begin{array}{l} N_{G_1}(dom_L, dom_R) + N_{G_2}(i, j) \\ N_{G_1}(dom_L, j) + N_{G_2}(i, dom_R) \\ N_{G_1}(i, dom_R) + N_{G_2}(dom_L, j) \\ N_{G_1}(i, j) + N_{G_2}(dom_L, dom_R) \end{array} \right\} & \text{if } dom_L \ge 0 \text{ and } dom_R < 0 \\ \min\limits_{\substack{dom_L\le i\le -(dom_L+1) \\ -(dom_R+1)\le j\le dom_R}} \left\{ \begin{array}{l} N_{G_1}(dom_L, dom_R) + N_{G_2}(i, j) \\ N_{G_1}(dom_L, j) + N_{G_2}(i, dom_R) \\ N_{G_1}(i, dom_R) + N_{G_2}(dom_L, j) \\ N_{G_1}(i, j) + N_{G_2}(dom_L, dom_R) \end{array} \right\} & \text{if } dom_L < 0 \text{ and } dom_R \ge 0 \\ \min\limits_{\substack{dom_L\le i\le -(dom_L+1) \\ dom_R\le j\le -(dom_R+1)}} \left\{ \begin{array}{l} N_{G_1}(dom_L, dom_R) + N_{G_2}(i, j) \\ N_{G_1}(dom_L, j) + N_{G_2}(i, dom_R) \\ N_{G_1}(i, dom_R) + N_{G_2}(dom_L, j) \\ N_{G_1}(i, j) + N_{G_2}(dom_L, dom_R) \end{array} \right\} & \text{if } dom_L, dom_R < 0 \end{cases}$

Table 5.3: Equations for the $G = p(G_1, G_2)$ operation

N	*-2*	*-1*	*0*	*+1*	*+2*
-2	1	2	2	2	∞
-1	1	2	2	2	∞
0	1	2	2	2	∞
+1	2	2	2	4	∞
+2	∞	∞	∞	∞	∞

B	*-2*	*-1*	*0*	*+1*	*+2*
-2	∞	∞	∞	∞	∞
-1	∞	∞	∞	∞	∞
0	∞	∞	∞	∞	∞
+1	∞	∞	∞	3	∞
+2	∞	∞	∞	∞	4

L	*-2*	*-1*	*0*	*+1*	*+2*
-2	∞	∞	∞	∞	∞
-1	∞	∞	∞	∞	∞
0	∞	∞	∞	∞	∞
+1	2	3	3	3	∞
+2	2	3	3	2	∞

R	*-2*	*-1*	*0*	*+1*	*+2*
-2	∞	∞	∞	2	2
-1	∞	∞	∞	2	3
0	∞	∞	∞	2	3
+1	∞	∞	∞	3	2
+2	∞	∞	∞	∞	∞

Table 5.4: Example of the N, L, R, and B tables for a vertex in a graph of radius 2

Rows are indexed by the dom_L value, columns are indexed by the dom_R value. This is actually the final values of the root vertex of the test run in Section 5.4.

Every non-infinite value of non-negative dom_L and dom_R represents a possible dominating broadcast. See how the non-negative values of these tables indicate an optimal solution of cost 2.

5.2.5 Corrections

The original algorithm, as given in [1], was missing some constraints in the calculations of the L and R tables and thus treated some impossible solutions as feasible solutions. Specifically, when a vertex was said to be in both the dominating set and dominating ≤ 0 vertices outward, i.e. $L(dom_L \leq 0, dom_R)$ and $L(dom_L, dom_R \leq 0)$. This made the implementation give some slightly wrong answers to the test graphs, and it was corrected by constraining the L tables to contain the value infinity where $dom_L \leq 0$ and the same for the R tables where $dom_R \leq 0$. This has been confirmed as correct by Prof. Steve Horton, one of the authors of [1].

5.3 Implementation details

Since the algorithm only works on the decomposition tree and not the actual series-parallel graph, we decided to take just the decomposition tree as input to the algorithm, assuming that the first vertex in the input is the root of the decomposition tree. However, we have also created a program for finding a decomposition tree given a series-parallel graph, so that anyone wanting to run the algorithm will not be required to manually decompose a series-parallel graph.

In addition to the actual algorithm and the equations described in this chapter, we also had to create a method for calculating the radius and diameter of a series-parallel graph given its decomposition. We chose to simply create the corresponding series-parallel graph adjacency matrix from the leaves[1], run the Floyd-Warshall All-Pairs-Shortest-Path algorithm on the adjacency matrix to get a distance matrix, and extract the radius and diameter from the distance matrix. It is a bit complicated to compare the running time for the Floyd-Warshall algorithm, which runs in time $O(n^3)$ to the main algorithm, which runs in time $O(nr^4)$ (or $O(mr^4)$), but a little testing showed that the Floyd-Warshall algorithm completed in less than a second on a graph on which the main algorithm took several hours to run. The adjacency matrix and distance matrix are discarded before the algorithm starts, so we use no extra memory. The complete algorithm is given in Chapter 8.

The added functionality for calculating the broadcast function is done similarly to how it is done in the implementation for interval graphs: by keeping track of, for each value x in each table, which other values contribute to the final value x. A simple backtracking from an optimal value to each of the leaves will yield the vertices whose values come from the L, R or B tables of the leaf. The terminals of these leaves will be broadcasting vertices in the series-parallel graph, and the dom_L and dom_R values for the leaves gives the broadcast power.

As with the trees algorithm, we could save memory by letting the implementation allocate and de-allocate calculation tables on the run. Thus, after calculating the tables for vertex v, we can free the tables of the children from the memory. If we also do not allocate memory for the tables of v until we actually get to v, the algorithm will be able to perform with less required memory. However, since the algorithm is bounded by processing time rather than memory, there was no need to implement this optimisation.

[1] Each leaf of the decomposition tree is an edge in the corresponding series-parallel graph.

5.4 Test run

We will now run the algorithm on the series-parallel graph from the top of Figure 5.2. We have arbitrarily numbered the vertices and constructed a list of decomposition tree vertices (see Figure 5.3). The list has been stored in a file named "testgraph".

```
17                 // 17 vertices in decomposition tree
0 17 16 1 7 P      // vertex 1
10 0 0 1 2 L       // vertex 2
10 0 0 2 3 L       // vertex 3
13 0 0 1 3 L       // vertex 4
14 0 0 3 4 L       // vertex 5
11 0 0 4 5 L       // vertex 6
11 0 0 5 7 L       // vertex 7
12 0 0 3 6 L       // vertex 8
12 0 0 6 7 L       // vertex 9
13 2 3 1 3 S       // vertex 10
14 6 7 4 7 S       // vertex 11
15 8 9 3 7 S       // vertex 12
16 10 4 1 3 P      // vertex 13
15 5 11 3 7 S      // vertex 14
16 14 12 3 7 P     // vertex 15
1 13 15 1 7 S      // vertex 16
1 0 0 1 7 L        // vertex 17
```

Figure 5.3: Input data for test run

> The text to the right of the "//" is commenting, and will not be passed to the program. The ordering of the values in each of the vertex lines is "*$parentID_1$, $leftChildID_1$, $rightChildID_1$, $leftTerminalID_2$, $rightTerminalID_2$, Operation,*" where *Operation* is either S, P, or L, for series, parallel and leaf, respectively. ID_1 refers to a vertex ID in the decomposition tree, and ID_2 refers to a vertex ID in the series-parallel graph. Running the program with the `--help` option will also display the requested input format.

We will run the algorithm with the $-b$ option to indicate that we also want the program to output the broadcast function, not just the optimal cost.

```
# ./series-parallel-broadcast-domination -b < testgraph
Minimum Broadcast Domination: 2
Broadcast function f(v)=Z*:
f(1)=0 f(2)=0 f(3)=1 f(4)=0 f(5)=1 f(6)=0 f(7)=0
#
```

The result from the test run is plotted into the graph in Figure 5.4.

This result is very different from the results in chapters 3 and 4, as the returned solution is clearly inefficient (but still optimal). Contrary to the algorithms for trees and interval graphs, the algorithm for series-parallel graphs will consider every possible optimal solution. This means that, although there still does exist an efficient, optimal solution (see Figure 5.5), whether the algorithm will find an efficient optimal solution or a non-efficient solution of the same cost is completely arbitrary.

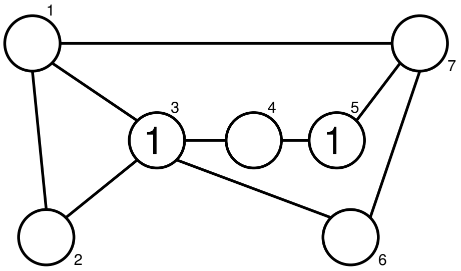


Figure 5.4: Non-efficient, optimal broadcast on example series-parallel graph

This is the result returned by the algorithm. Note that the broadcasts from vertices 3 and 5 overlap at vertex 4.

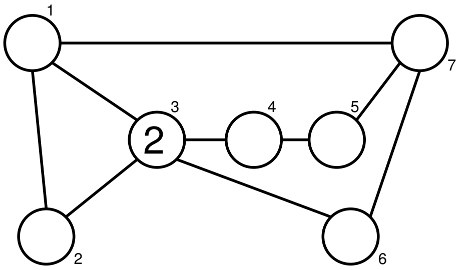


Figure 5.5: Efficient, optimal broadcast on example series-parallel graph

This is one of the alternative optimal solutions the algorithm did not return. It is a simple radial broadcast, so it is clearly efficient.

Chapter 6

Special Case Graphs

To give a better understanding of how the structure of a graph affects its optimal broadcast domination, we have chosen a few simple graph classes that yield easily predictable results.

Because of their predictability, the following graphs will reappear as test graphs in Chapter 7.

6.1 Paths

As is evident from the definition in Chapter 1, paths are a very simple graph class. We will now show that the solution to the Broadcast Domination problem on paths is simple as well. Recall that P_n denotes a path containing n vertices and $n-1$ edges.

Lemma 6.1. *A path P_n has $\gamma_b\left(P_n\right) = \lceil \frac{n}{3} \rceil$.*

Proof. We want to know, for a given broadcast, the ratio for how many vertices can be dominated per broadcast power unit. A broadcast of power 1 on a vertex v will dominate v and the two neighbors of v, and for each added $+1$ power it will dominate two additional vertices. This gives us the formula $CoverRatio(p) = \frac{1+2*p}{p}$. $CoverRatio(1) = 3$ gives the ratio of 3 vertices covered for a broadcast power of 1, and it is easy to see that $CoverRatio(p) < 3$ for all $p > 1$. Therefore, the minimum cost broadcast to dominate a path P_n consists of a number of broadcasts, each of power 1: By setting an evenly spaced third of the vertices to have a broadcast of power 1, we can dominate the entire path with cost $\gamma_b(P_n) = \lceil \frac{n}{3} \rceil$. □

For simplicity, we have defined paths as a series of connected P_3s in the example Figure 6.1. Note that for the broadcast to be efficient, it is sometimes necessary to unite two broadcasts at one end of the path, but this has no effect on the total cost. Figure 6.2 shows how this can be done.

Any graph that is a path is also a series-parallel graph, an interval graph and a tree. This is trivial to verify by reading the definition of the respective graph classes.

A path P_n has the largest radius $r = \lceil \frac{n-1}{2} \rceil$ of all graphs of size n. Assuming we choose the central vertex as the root of a tree, the tree height h is equal

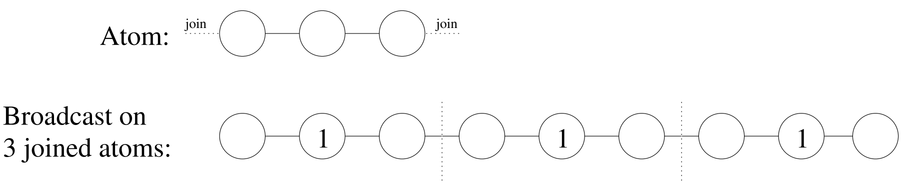


Figure 6.1: Path construction and optimal broadcast domination

The path in the above figure has been simplified to show how the broadcast grows in relation to the graph size.

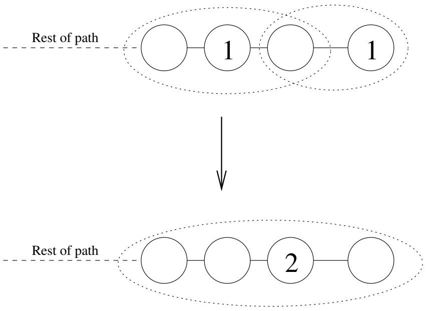


Figure 6.2: Maintaining broadcast efficiency in a path

For any path $P = (V, E)$. When $|V| \mod 3 = 1$, an even distribution of $\left\lceil \frac{|V|}{3} \right\rceil$ broadcasts of power 1 will overlap at exactly one vertex. For simplicity, we assume this to happen at the rightmost end of the path (we can easily rearrange the broadcasting vertices so that this is the case). We can now unite the two rightmost broadcasts such that all 4 rightmost vertices are covered and the broadcast does not overlap with the rest of the graph. The total cost is the same, and the broadcast is now efficient. When $|V| \mod 3 = 2$ we will not get an overlap.

to the radius r. Since r is a factor in the memory usage for the series-parallel algorithm, and h is a factor in the memory usage for the tree algorithm, we can consider paths to be worst case input for these algorithms. The algorithm for interval graphs will will use the same amount of resources for any two graphs of the same size. This means we can use paths to test the practical limitations of the implemented algorithms, i.e. how long paths can we input into the program before it runs out of memory or takes more than, for example, 12 hours to finish.

6.2 Cycles

Since a cycle $C = (V, E)$ is basically a path with one added edge incident to the two endpoints, it is clear that it can also be optimally dominated by $\left\lceil \frac{|V|}{3} \right\rceil$ evenly spaced broadcasts of power 1. However, it is slightly more complicated to maintain efficiency when $|V| \mod 3 \neq 0$. Figures 6.3 and 6.4 show how to maintain efficiency when the entire cycle can not be broken into P_3s. For simplicity, we will assume that $|V| > 6$.

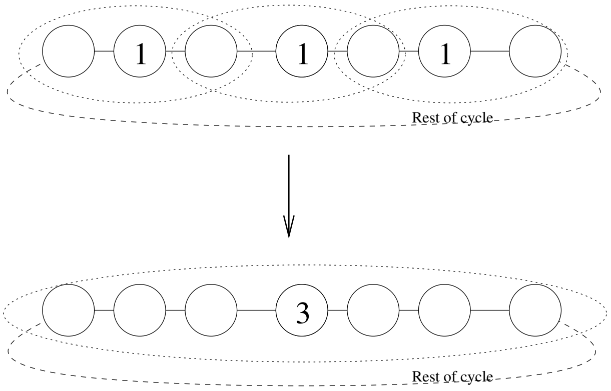


Figure 6.3: Maintaining broadcast efficiency in a cycle case A

When $|V| \mod 3 = 1$, an even distribution of $\left\lceil \frac{|V|}{3} \right\rceil$ broadcasts of power 1 will overlap at exactly two vertices. For simplicity, we assume that the overlapping vertices are at distance 2 from each other (we can easily rearrange the broadcasting vertices so that this is the case) and thus both have one broadcast in common. The vertices covered by the three overlapping broadcasts constitute a P_7, which can be dominated by a single broadcast of power 3. The total cost is the same, and the broadcast is now efficient.

A cycle of size $|V| \geq 3$ is not a tree, since this would imply that there exists at least two different paths between two vertices, contradicting the definition of a tree.

A cycle of size $|V| \geq 4$ is not an interval graph for a similar reason, namely that there would have to exist two paths P_1, P_2 between any two vertices u, v, $u < v$ (thus $right(u) < left(v)$) and no edge joining a vertex $x \in P_1$ to a vertex $y \in P_2$ except where $x \in \{u, v\}$ or $y \in \{u, v\}$. Since the intervals of the vertices

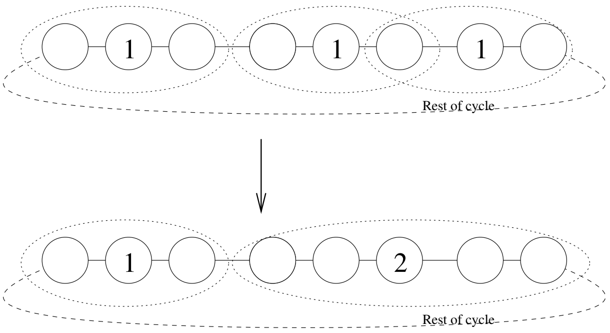


Figure 6.4: Maintaining broadcast efficiency in a cycle case B

When $|V| \mod 3 = 2$, an even distribution of $\left\lceil \frac{|V|}{3} \right\rceil$ broadcasts of power 1 will overlap at exactly one vertex. The vertices dominated by the two overlapping broadcasts constitute a P_5, which can be dominated by a single broadcast of power 2. The total cost is the same, and the broadcast is now efficient.

of any path from $right(u)$ to $left(v)$ has to contain every integer in the interval $[right(u), left(v)]$, and any two vertices with overlapping intervals are adjacent, this means that for any two paths P_1, P_2 from u to v, there exists at least one pair of overlapping intervals $x \in P_1, y \in P_2, x \notin \{u, v\}, y \notin \{u, v\}$. Therefore, there cannot exist a cycle of size $|V| \geq 4$ in an interval graph. A cycle of less than 4 vertices is a clique, which can exist in an interval graph.

All cycles are series-parallel graphs. They can be constructed with $|V| - 2$ concurrent series operations to create a path, followed by one parallel operation to join the two endpoints of the path.

6.3 1-Caterpillars

First, we will define the graph class *star*. A star consists of at least one vertex v_r and edges $E = \{(v_r, v) : v \neq v_r, v \in V\}$, where v_r is evidently the central vertex.

A *caterpillar* is a graph created by joining k arbitrary stars $G_1, G_2, \ldots, G_k$ with edges connecting the central vertex of each $G_i, 1 \leq i \leq k$ such that there exists exactly one path connecting the central vertices of G_1 and G_k. In the case of caterpillars, we call each star a *segment*, and in each segment v_r is called the *body* vertex and all vertices $\{v : v \neq v_r\}$ are called *hair* vertices.

A *1-caterpillar* is a restricted caterpillar where each segment is a P_2. See Figure 6.5 for an example.

Despite its similarity to a path, a 1-caterpillar $G = (V, E)$ can only be optimally dominated by a radial broadcast of total cost $\left\lceil \frac{|V|+2}{4} \right\rceil$, as indicated by Figure 6.5. Thus, 1-caterpillar is a radial graph class.

Lemma 6.3. *A 1-caterpillar $G = (V, E)$ has $\gamma_b(G) = \left\lceil \frac{|V|+2}{4} \right\rceil$.*

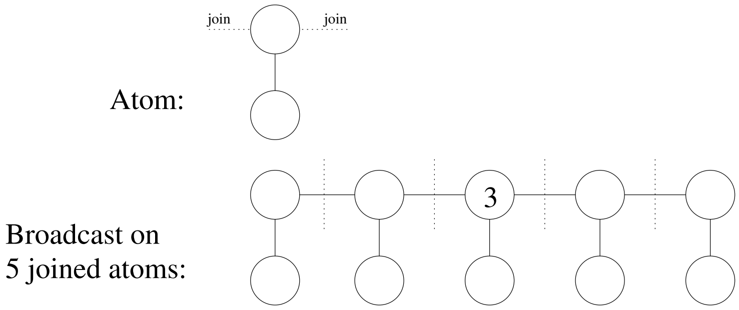


Figure 6.5: 1-Caterpillar construction and optimal broadcast domination

A 1-caterpillar of size $2k$ is created by joining k segments as described in 6.3.

Proof. To prove this, we use the fact that for any broadcast function of optimal cost γ_b there exists at least one efficient broadcast function also of optimal cost γ_b, and show that there exists no efficient broadcast function of cost $\gamma_b < rad(G)$:

Any broadcast not dominating every vertex in G will leave at least one SG where the body is dominated and the hair is undominated. It is clear that to dominate any such hair with any other broadcast we also have to dominate the already dominated body adjacent to the hair, creating an inefficient overlap. Therefore, we have to dominate every vertex in G by a single broadcast for the broadcast domination to be efficient. Since the lowest cost broadcast for dominating every vertex in G is a broadcast of power $rad(G)$ placed at a central vertex, the optimal dominating broadcast function for G has total cost $\gamma_b = rad(G)$. Since $diam(G)$ is the length of the path connecting the hair of SG_1 and the hair of $SG_{\frac{|V|}{2}}$, $diam(G) = \frac{|V|}{2} + 1$, and the radius of any interval graph (see further down how a 1-caterpillar is an interval graph) H is $rad(H) = \left\lceil \frac{diam(H)}{2} \right\rceil$, we have that for any 1-caterpillar G, $\gamma_b(G) = \left\lceil \frac{\frac{|V|}{2}+1}{2} \right\rceil = \left\lceil \frac{|V|+2}{4} \right\rceil$. □

A 1-caterpillar is a tree, since there exists no more than one path between any pair of vertices.

A 1-caterpillar is an interval graph, because the intervals $\bigcup_{1\leq i\leq k}\{[2i+0, 2i+2], [2i+1, 2i+1]\}$ yield a 1-caterpillar of size $|V| = 2k$.

It is impossible to create a 1-caterpillar of size $|V| \geq 6$ by following the rules of series-parallel composition. Since we can only connect two segment by their body vertices, this means at least one segment needs to have the body vertex as both the left and right terminator. This can not be obtained by following the series-parallel composition rules, so therefore 1-caterpillars of size $|V| \geq 6$ are not series-parallel graphs. A 1-caterpillar of size $|V| = 2k < 6$ is a simple P_{2k} and clearly a series-parallel graph.

6.4 Teeth

A *teeth* graph $G = (V, E)$ of size $2k - 1$ is an overlapping series of k C_3s, where $V = \{v_i : 1 \leq i \leq k\}$ and

$$E = \{(v_i, v_{i+1}) : 1 \leq i \leq |V|\} \cup \left\{(v_{2i-1}, v_{2i+1}) : 1 \leq i \leq \left\lfloor \frac{|V|}{2} \right\rfloor \right\},$$

which is probably easier to understand by looking at Figure 6.6. Teeth is also a radial graph class.

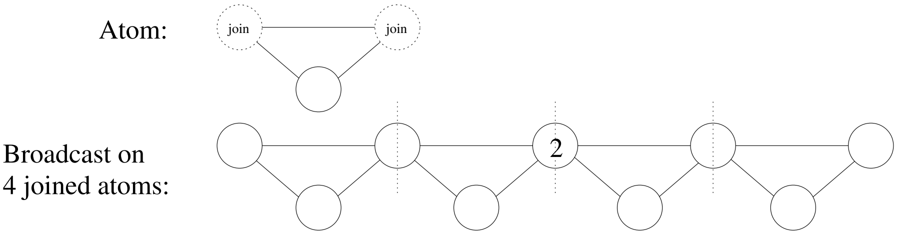


Figure 6.6: Teeth construction and optimal broadcast domination

A teeth graph of size $2k - 1$ is created by joining k C_3s.

Lemma 6.4. *A teeth graph $G = (V, E)$ has $\gamma_b(G) = \left\lceil \frac{|V|-1}{4} \right\rceil = \gamma_b(G)$.*

Proof. Similarly to 1-caterpillars, a teeth graph G can only be dominated efficiently by a single broadcast dominating every vertex of G: For any broadcast that does not dominate every vertex in G, there exists at least one $C_3 \subset G$ where at least one vertex of the C_3 is undominated. This means that for any two broadcasts of any power, there exists at least one C_3 where either the broadcasts overlap, or not every vertex of the C_3 is dominated. In the case where two broadcasts dominate 2 of the vertices of a C_3, the undominated vertex u will be the one with degree 2 in the original graph, and thus a third broadcast can not reach the u without overlapping with at least one of the two other broadcasts. Since any broadcast overlap is inefficient, and any optimal solution has an equivalent efficient broadcast function, a teeth graph can only be optimally dominated by a single, and therefore radial, broadcast.

The diameter of a teeth graph G is $d(v_1, v_{|V|})$. Since the edges of the shortest path connecting v_1 and $v_{|V|}$ is given by the second part of the formula for calculating the edges of G above, $diam(G) = \frac{|V|-1}{2}$. Since the radius of any interval graph (see further down how a teeth graph is an interval graph) H is $rad(H) = \left\lceil \frac{diam(H)}{2} \right\rceil$, we have $rad(G) = \frac{diam(G)}{2} = \left\lceil \frac{|V|-1}{4} \right\rceil$, and consequently $\gamma_b(G) = \left\lceil \frac{|V|-1}{4} \right\rceil$. □

A teeth graph of size $|V| \geq 3$ is trivially not a tree, because it contains at least one C_3.

A teeth graph is an interval graph. The intervals for a teeth graph of size $|V|$ are $\{[1, 2]\} \cup \left\{[2i, 2i + 1], [2i, 2i + 2] : 1 \leq i \leq \frac{|V|-1}{2}\right\}$.

A teeth graph is a series-parallel graph. We can create each C_3 with the operations $parallel(serial(P_2, P_2), P_2)$ and then join the C_3s by *serial* operations to get the teeth graph.

6.5 Boxes

A ***boxes*** graph is essentially a wide path, but its structure is easier to describe as a 1-caterpillar with added edges such that each pair of consecutive segments is a clique, as shown in Figure 6.7.

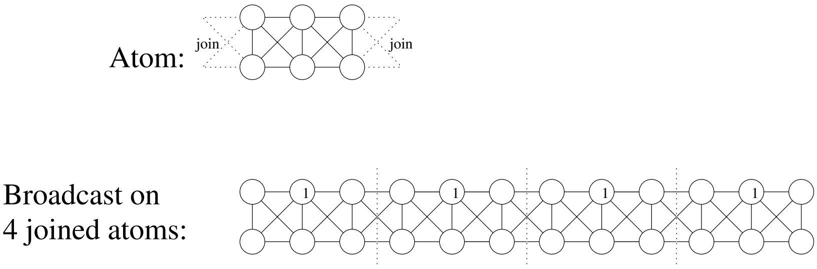


Figure 6.7: Boxes construction and optimal broadcast domination

A boxes graph constructed by joining smaller boxes graphs each of size $|V| = 6$. The construction is simplified to illustrate how the optimal domination cost is related to the graph size. A boxes graph can actually be of any size as long as $|V| \mod 2 = 0$.

Contrary to 1-caterpillars but similar to paths, the optimal broadcast domination of a boxes graph is not one radial broadcast, but a number of evenly spaced broadcasts of power 1.

Lemma 6.5. *A boxes graph* $G = (V, E)$ *has* $\gamma_b(G) = \left\lceil \frac{|V|}{6} \right\rceil$.

Proof. In a boxes graph $G = (V, E)$, any vertex with an originating broadcast of power 1 will dominate itself and its 5 neighbors, for a total of 6 vertices. For each added +1 power it will dominate another 4 vertices. This results in the formula $CoverRatio(p) = \frac{2+4*p}{p}$ describing, for a single broadcasting vertex, how many vertices are dominated per domination power. We can easily see that $CoverRatio(p) < CoverRatio(1)$ for any $p > 1$. Therefore, we will need $\left\lceil \frac{|V|}{6} \right\rceil$ evenly spaced dominators of power 1 to optimally dominate a boxes graph G with total cost $\gamma_b(G) = \left\lceil \frac{|V|}{6} \right\rceil$, as indicated by Figure 6.7. □

For a boxes graph of size $|V| \mod 6 \neq 0$, we can maintain broadcast efficiency in the same way as we would for paths, by joining two overlapping vertices at one end of the graph.

A boxes graph of size $|V| > 2$ is clearly not a tree, since it contains cliques of more than 2 vertices.

A boxes graph of size $|V| > 2$ is not a series-parallel graph, since it is impossible to create a clique of size 4 by following the rules of series-parallel composition. At some time during the construction of a clique of size 4, we must have created a P_3 where the central vertex is not a terminal. Since there is no way to attach a neighbor to a vertex that is not a terminal, we cannot connect the central vertex of the P_3 to a new vertex and create a clique of size 4.

A boxes graph of any size $|V| \mod 2 = 0$ is an interval graph, because the intervals $\bigcup_{1 \leq i \leq \frac{|V|}{2}} \{(2i, 2i+2), (2i, 2i+3)\}$ yield a boxes graph.

Note that, similarly to paths, we could add edges between the vertices of both ends to create a "boxes cycle" which would result in the same optimal broadcast cost as for boxes graphs. However, this would be neither an interval graph, a series-parallel graph nor a tree. Consequently, we can not run any of the algorithms on such a graph, so it is not very relevant to this work.

Chapter 7

Testing

In this chapter, we will describe in detail how the implementations were tested, and the corresponding test results. There are three different types of tests we will perform; correctness tests, performance tests, and how often an optimal broadcast differs from a trivial solution. Each type of test has been covered in a different section.

All tests in this chapter will be performed on a 2.26 MHz Pentium 4 workstation computer with 512 MB RAM and 512 KB level 2 cache.

We will in this chapter sometimes use "algorithm" and "implementation" interchangeably, but it should be clear from the context when there is a need to differentiate between them.

We will only describe the test results from the final, working versions of the implementations.

7.1 Test data generation

To perform a test, we will of course need suitable input for the programs. We have created programs for generating two different *types* of data; the easily predictable special case graphs from Chapter 6, and randomly generated interval graphs, series-parallel graphs and trees.

We will use most of the special case graphs only for correctness testing, but paths will be also be used for testing comparative performance and the practical limits of all three algorithms.

A random instance of each graph class is better suited for more realistic testing, since it potentially represents any conceivable input graph. We will therefore use these for testing performance and for testing how much optimal dominating broadcasts differ from radial solutions.

Even more realistic scenarios would be to find real data of a structure corresponding to series-parallel graphs, interval graphs, and trees, like a computer chip map, a large table of flight times, and an organisation chart for a large corporation, respectively. However, such data were not easily obtainable in a usable format.

7.1.1 Special case graphs

As mentioned in Chapter 6, an optimal broadcast for paths, cycles, 1-caterpillars, teeth graphs and boxes graphs can always be calculated from the graph size. This makes them immediately useful for correctness testing since the result is trivial to verify. In addition to this they are trivial to generate and verify, so there is very little chance of giving the algorithms garbage input[1].

The errors that were detected in the algorithms for interval graphs and series-parallel graphs showed up when testing the implementations on these graphs (see respective chapters 3 and 5 for details). Many programming errors were also detected, and subsequently corrected, during these tests.

7.1.2 Randomly generated graphs

For more realistic test scenarios, we will need to test the implementations on random instances of their corresponding graph classes.

Since randomly generated graphs are a bit more complex than the graphs from the previous section, we also had to test the actual graph generation programs before we could use them, to make sure they did not produce garbage input. These tests were fairly trivial and will not be covered here.

We will mainly be using randomly generated graphs for testing the running time and how the optimal broadcast cost is related to graph size and radius, but have performed some manual correctness tests on small graphs.

7.1.3 Generation of test graphs

The following subsections describe in detail how the test graphs are generated for each graph class.

We have created one stand-alone program for each graph class, and the output is in the same format as the input for the implementations. This means we can create automated tests by ordering a computer to repeatedly pipe[2] a randomly generated graph through the algorithm, and store the result and how much time the algorithm spent in a file.

Special case graphs

The generation of each special case graph class is trivial and has already been described in Chapter 6, so we will not present any pseudocode for the generation of these graphs.

Random interval graphs

The simplest way to generate a random interval graph is to start with an empty interval graph and add n completely random intervals with endpoints in the range $[0, k]$ where k is a chosen integer > 0. However, this tends to generate

[1] Garbage input is a computer term meaning "wrong/bad input". For example, if you wanted a calculator to evaluate $\frac{4322}{7}$ and accidentally typed 4322//7 or 4332/7, you would give the calculator garbage input. This would in turn result in garbage output, in the form of a program crash, an error message, or a useless answer.

[2] Piping is to let one program receive its input from the output of another program. For example, in a Linux shell environment by typing "**progA** | **progB**". **progB** will now take its input from **progA**'s output.

"lumpy" graphs, meaning that there almost always exists a vertex that is incident to over 90% of the rest of the vertices in the graph. Since this means the resulting graphs could generally be dominated at a total cost of 1, 2 or 3 completely regardless of their size, we had to add a restriction: We will in addition require a parameter l denoting the maximum allowable interval width. This proved to result in more interesting graphs.

Now we have one last requirement for the output graph; namely that the graph is connected. For an interval graph to be connected means that there exists at least one path connecting the leftmost vertex to the rightmost vertex. We make use of this definition by first generating a series of randomly overlapping intervals, with interval width in the range $[1, l]$, stretching from leftmost endpoint 0 to rightmost endpoint k. Also, the left and right endpoints of each successive vertex must be at least 1 larger than the corresponding endpoints of the previous vertex.

For simplicity, we want to reduce the number of parameters to the algorithm by eliminating k:

Observation 7.1. *An interval graph of n vertices can always be represented with intervals in the range* $[1...n]$.

Proof. $k = n$ is sufficient to allow for the generated graph to be a path of length n, i.e. a connected graph of size n with the least amount of overlap. Since increasing k will only add the possibility to generate graphs containing less overlap, there is no need for $k > n$. Also, $k < n$ will only yield the possibility to generate graphs containing more overlap, which can also be gained by increasing l. We can therefore safely assume that $k = n$ does not restrict the structure of the generated interval graphs, making the algorithm simpler. □

```
Algorithm Generate Interval Graph
Input: An integer $n$ denoting the size of the output graph and an integer $l$
  denoting the maximum interval length to be used.
Output: An interval graph represented by a set $I$ of $n$ intervals, with max-
  imum allowable interval length of $l$.
// – assert connected graph
let $verticesLeft = n$
$k = n$
$I = \emptyset$
$rightBorder = 0$
$leftBorder = -1$
while $(rightBorder < k - 1)$ do
  $width = randomIntegerInRange([1, l])$
  $right = randomIntegerInRange([1, width]) + rightBorder$
  $left = right - width$
  // assure new interval's left endpoint is larger than the leftBorder value
  // move the interval if necessary
  if $(left < leftBorder + 1)$ then
    $right = right + (leftBorder - left + 1)$
    $left = leftBorder + 1$
  end if
  // truncate interval if it now passes outside the valid range
  if $(right \geq k)$ then
```

```
        $right = k - 1$
    end if
    $I = I \cup \{(left, right)\}$
    $verticesLeft = verticesLeft - 1$
    $leftBorder = left$
    $rightBorder = right$
end while
// – generate the remaining intervals
while ($verticesLeft > 0$) do
    $verticesLeft = verticesLeft - 1$
    $width = randomIntegerInRange([1, l])$
    $left = randomIntegerInRange([0, k - width])$
    $right = left + width$
    $I = I \cup \{(left, right)\}$
end while
Return $I$
```

Random Trees

The simplest way to randomly generate a tree of size n is to start with a tree containing a single vertex (the root) and iteratively add $n - 1$ vertices where each new vertex is set as a child of a random, existing vertex. However, this proved to generally produce trees with a rather large degree close to the root, and low degree close to the leaves. Due to this, we found it necessary to introduce a parameter w which constrains the number of children each vertex can have (constraining the maximum degree of the tree to $w + 1$).

```
Algorithm Generate tree
Input: An integer $n$ denoting the size of the output tree and an integer $w$
   denoting the maximum number of children allowed per vertex.
Output: A tree $T = (V, E)$ of size $n$ with maximum degree $w + 1$.
$V = \{v_0\}$ // the root
$E = \emptyset$
for $i = 1$ to $n$ do
    $parent$ = random vertex from $V$ with $degree < w$
    $V = V \cup \{v_i\}$
    $E = E \cup \{(v_{parent}, v_i)\}$
end for
Return $T = (V, E)$
```

Random Series-Parallel Graphs

Every series-parallel graph can easily be generated from its decomposition tree. Therefore, to generate a series-parallel graph G we start by generating a random binary tree T_d and declaring this to be the decomposition tree of G. Then, we randomly assign either the series operation or the parallel operation to all non-leaf vertices of T_d, with the constraint that a vertex cannot be assigned the parallel operation if none of its children is assigned a series operation. This is to avoid double edges, which have no effect on the broadcast domination of the graph.

Generating the left and right terminals for each vertex is fairly trivial, and it also makes generating a graph from the decomposition tree trivial: Since the leaves in a decomposition tree constitutes all the edges of a graph, we can generate the graph linearly: $V = \{1, 2, \ldots, k\}, E = \{(t_L(G_i), t_R(G_i)) : G_i \text{ is a leaf in } T_d\}$, where k is the number of distinct terminals in T_d. See the pseudocode for **Generate Series-Parallel Decomposition Tree** for details on calculating the terminals.

The part of the algorithm that generates the binary tree is a bit tricky, so we have extracted it from the main algorithm. It starts by creating a random ordering R of the vertices, but assuring that the root is the first element in R, and creating two pointers on the list, p_t and p_p. p_t points to the leftmost vertex that is not yet a part of the tree, and $p_p < p_t$ points to the leftmost vertex that is part of the tree but has not yet been assigned two children. It then inserts $R[0]$ (the root) into the tree and sets $p_p = 0$ and $p_t = 1$. Then, the algorithm will repeat the following until $p_t = n$: Let p be a random number in the range $[p_p, p_t - 1]$, and set $c_1 = R[p_t + 0]$ and $c_2 = R[p_t + 1]$. Add c_1 and c_2 to the tree and assign them to be the children of $R[p]$. Switch $R[p]$ and $R[p_p]$. Increase p_p by 1 and p_t by 2.

```
**Algorithm** Generate Random Binary Tree **GRBT**
**Input:** An integer $n$ denoting the size of the output tree.
**Output:** A binary tree $T_B = (V_{T_B}, E_{T_B})$ of size $n$.
$V_{T_B} = \emptyset$
$E_{T_B} = \emptyset$
$R = list[0, 1, \ldots, n-1]$
$shuffleR[1 \ldots n-1]$
$p_p = 0$
$p_t = 1$
**while** $p_t < n$ **do**
    $c_1 = R[p_t + 0]$
    $c_2 = R[p_t + 1]$
    $p = randomIntegerInRange([p_p, p_t - 1])$
    $V_{T_B} = V_{T_B} \cup \{c_1, c_2\}$
    $E_{T_B} = E_{T_B} \cup \{(R[p], c_1), (R[p], c_2)\}$
    $swap = R[p_p]; R[p_p] = R[p]; R[p] = swap$
    $p_t = p_t + 2$
    $p_p = p_p + 1$
**end while**
**Return** $T_B = (V_{T_B}, E_{T_B})$
```

Because the algorithm for series-parallel graphs only operates on the tree decomposition, we do not need to build and output the resulting graph.

Note that the algorithm will not know the size of the resulting series-parallel graph until it is finished, as it depends on how many vertices are assigned the series operation. This gave us a little trouble during testing, since we wanted graphs of a particular size. To remedy this, we had two choices:

- Keep generating random series-parallel decompositions until the algorithm outputs a graph of the size we want.
- Rework the graph generation algorithm to assure the resultant graph is of a specific size.

As the number of vertices in a series-parallel graph is completely dependent on the number of series operations in the decomposition tree, we would have to create a specified number of series operations, a random number of parallel operations, combine these in such a way that no double edges are created, and add a fitting number of leaf vertices. Since the first option proved to be both quick and painless, there was no need to invent, prove, implement, and test a new algorithm.

```
**Algorithm** Generate Series-Parallel Decomposition Tree
**Input:** An integer $n$ denoting the size of the output tree.
**Output:** A decomposition tree $T_d = (T_B = (V_{T_B}, E_{T_B}), op, t_L, t_R)$, where
  $T_B$ is the binary tree structure, $op$, $t_L$ and $t_R$ are arrays such that $op[v_i]$
  stores the series-parallel operation of $v_i$, $t_L[v_i]$ stores the left terminal of $v_i$
  and $t_R[v_i]$ stores the right terminal of $v_i$.
$op$ = array of size $n$
$t_L$ = array of size $n$
$t_R$ = array of size $n$
$V_{T_B} = GRBT(n)$
// randomly assign operations to vertices
for $i = 0$ to $n - 1$ do
  if vertex $i$ is a leaf in $T_B$ then
    $op[i] = LEAF$
  else
    $setop = randomOf(PARALLEL, SERIAL)$
    if $setop = SERIAL$ then
      $op[i] = SERIAL$
      // the following line assures we will get no double edges
    else if $setop = PARALLEL$ then
      if $leftChild(i) = S$ or $rightChild(i) = S$ then
        $op[i] = PARALLEL$
      else
        $op[i] = SERIAL$
      end if
    end if
  end if
end for
// calculate terminals for vertices
$prd$ = a pre-order traversal of $T_B$.
$t_L[prd[0]] = 0$ // the terminals of...
$t_R[prd[0]] = 1$ // ...the root
$termCount = 2$
for $i = 1$ to $n$ do
  if $op[prd[i]] = SERIAL$ then
    $t_1 = t_L[prd[i]]$
    $t_2 = t_3 = termCount$
    $t_4 = t_R[prd[i]]$
    $termCount = termCount + 1$
    $t_L[leftChild[prd[i]]] = t_1$
    $t_R[leftChild[prd[i]]] = t_2$
    $t_L[rightChild[prd[i]]] = t_3$
    $t_R[rightChild[prd[i]]] = t_4$
  else if $op[prd[i]] = PARALLEL$ then
    $t_1 = t_3 = t_L[prd[i]]$
    $t_2 = t_4 = t_R[prd[i]]$
    $t_L[leftChild[prd[i]]] = t_1$
    $t_R[leftChild[prd[i]]] = t_2$
    $t_L[rightChild[prd[i]]] = t_3$
    $t_R[rightChild[prd[i]]] = t_4$
  end if
end for
Return $T_d = (T_B, op, t_L, t_R)$
```

7.2 Correctness

> *Software testing*[23]: Software testing is a process used to identify the correctness, completeness and quality of developed computer software. Actually, testing can never establish the correctness of computer software. It can only find defects, not prove that there are none. There are a number of different testing approaches that are used to do this ranging from the most informal *ad hoc* testing, to formally specified and controlled methods such as automated testing.

To test the correctness of the implementations, we will use automated tests with the special case graph classes specified in Chapter 6 as input data.

In theory, we would only need to test for those graph sizes where the solution was structurally different. For example, for a 1-caterpillar graph we would have to test for 3 graph sizes: For $|V| = 2$, where $\gamma_b(G) = \gamma(G)$, for $|V| = 4$, where we have 2 feasible solutions, and for $|V| = 6$, where we have one single solution $\gamma_b(G) \neq \gamma(G)$. However, as in all computer programs, there is always a possibility for programming errors in the implementations. Programming errors do not generally follow any conceivable logic, so we wanted to test for a few more graph sizes to increase our confidence in the results. Since the tests could be automated, we tested for every legal graph size up to a specified value chosen such that the tests would not take too long to run.

The results (Table 7.1) show that the algorithms and their implementations perform according to expectations. As the algorithms themselves are mathematically proven to be correct, and the implementations are mathematically trivial translations from the algorithms, we are confident that both the algorithms and the implementations are correct.

We have also performed some ad hoc correctness tests based on randomly generated graphs, but they were very informal and will therefore not be described here.

7.3 Performance

In this section, we will try to give an understanding of the practical performance of the algorithms. Note that since we are running on a single workstation computer, these test results should not be taken very literally, but more as an indication as to how the running times of the algorithms turn out in practice.

class of G	*interval for* $\lvert V(G)\rvert$	*interval graphs*	*SP-graphs*	*trees*
path	$[1, 100]$	pass	pass	pass
cycle	$[3, 100]$	n/a	pass	n/a
1-caterpillar	$[2, 100]$	pass	n/a	pass
teeth	$[3, 99]$	pass	pass	n/a
boxes	$[2, 100]$	pass	n/a	n/a

Table 7.1: Results from automated correctness test

The three rightmost columns indicate how the implementation of the algorithm for the specific graph class performed according to expectations, or n/a where the input graph cannot be represented in the corresponding graph class.

Before we can perform any tests on running time and comparative performance, we need to know which problem sizes can realistically be tested. Informal testing showed us that the implementation of the algorithm on trees was bounded by memory constraints, and that the implementations of the algorithms on interval graphs and series-parallel graphs were bounded by available time. For example, the series-parallel implementation ran for 6 full hours on a path of size 200, and when we tried to run it on a path of size 300, it ran for over 32 hours before it was terminated to make the computer available for more useful tasks. We decided that problem sizes which took longer than 1 hour would be unmanageable for testing.

As described earlier, paths are worst-case input to all algorithms, so we tested how large paths the implementations could handle within the given bounds. The limits are displayed in Table 7.2. Because of the very limited hardware available, we have not attempted to test the performance of the implementations on larger problem sizes, as such results would not be very useful. Any practical applications of the implementations on large problem sizes would most likely want to both parallelise the algorithms and run them on computers more powerful than a mere workstation, yielding a completely different performance.

None of the algorithms contain any noticeably large constants in their asymptotic running times.

7.3.1 Testing all three algorithms on paths

In addition to being worst case input, paths are also a subclass of interval graphs, series-parallel graphs and trees. This means that for any given path, we can run it on all three algorithms and compare the results. However, since the algorithms had radically different running times, there was no common graph size interval which gave useful results. Therefore, we have run each algorithm on 20 different path sizes up to the corresponding limit of the algorithm (as given in Table 7.2) and noted how much time each implementation needed. Since a path of n vertices has a radius of $\lceil \frac{n-1}{2} \rceil$, and $O(n)$ vertices in its decomposition tree, the running times for the interval graph algorithm, the series-parallel graph algorithm and the tree algorithm will be $O(n^3)$, $O(n^5)$, and $O(n^2)$, and the space requirements will be $O(n^2)$, $O(n^3)$, and $O(n^2)$, respectively.

Results

The results are displayed in Table 7.3, and and have also been plotted graphically in Figure 7.1. For each algorithm in the figure, we have also plotted the running times of the other two algorithms as dotted lines, which illustrates how much they differ in running time. Note that the vertical axis of the tree algorithm is measured in seconds, while the other two are measured in minutes.

The results for the tree algorithm show an irregular decrease in running time from the second largest to the largest problem. Such irregularities are not surprising when we are using almost all available system memory on a workstation computer, and subsequent test runs also give the same irregularity. We suspect this to happen because some low-priority, resource-consuming processes which are normally run by the operating system are suspended because there is not enough memory available for them. Thus, our program will get more of the available processor time.

It appears from Figure 7.1 that the algorithm for series-parallel graphs will not be able to handle a significant increase in problem size unless it is drastically optimised (see 9.3.1). However, the interval graph algorithm and the tree algorithm would benefit from adding more processing power and more memory, respectively. Also, the algorithm for trees could benefit from sacrificing some of its speed for memory, as it runs out of available memory long before its running time exceeds 3 minutes.

algorithm	$\|V\|$	*bounded by*
interval graphs	6000	time (1 hour)
series-parallel graphs	150	time (1 hour)
trees	32500	memory

Table 7.2: Practical limitations for each implementation

7.3.2 More general tests

We also want to see how the implementations perform on randomly generated instances of their corresponding graph class, as these make for more realistic results.

Since the running time may vary for several graphs of the same size, we have performed several runs for each graph size on each graph class, and correspondingly reduced the size interval such that none of the implementations would use more than 10 minutes had we run them on paths of the same size. The maximum degree of each tree was constrained to 6 to avoid "bushes", and all intervals were constrained to be of maximum width $\frac{|V|}{20}$ (5% of the maximum possible interval width).

We will perform 5 runs for each graph size on interval graphs and trees, and 10 runs for each graph size on series-parallel graphs, as randomly generated series-parallel graphs turned out to give less predictable running times.

In addition to the size of the input graph and the running time, we also want to know the values affecting the running time. Thus, we have output the height of the trees, the number of vertices of the series-parallel decomposition trees, and the radiuses of the series-parallel graphs (see 8.2 for how the radius is calculated).

$\|V(G)\|$	*time*
300	$0s$
600	$2s$
900	$7s$
1200	$18s$
1500	$35s$
1800	$1m$
2100	$1m36s$
2400	$2m24s$
2700	$3m29s$
3000	$4m53s$
3300	$6m41s$
3600	$8m53s$
3900	$11m38s$
4200	$14m58s$
4500	$19m16s$
4800	$24m11s$
5100	$30m2s$
5400	$36m53s$
5700	$45m15s$
6000	$54m28s$

Interval graphs
time $O(n^3)$
space $O(n^2)$

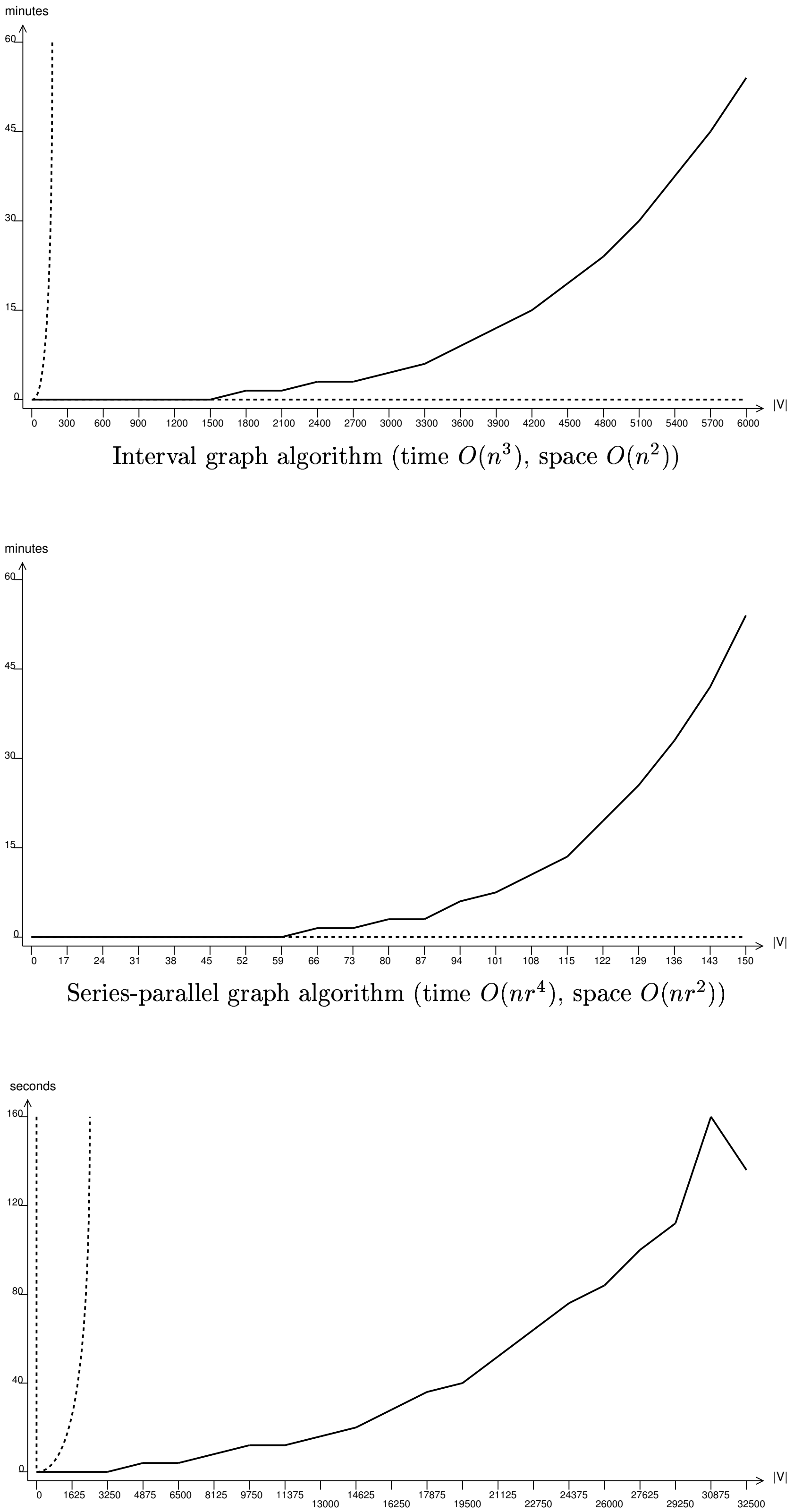
$\|V(G)\|$	*time*
17	$0s$
24	$0s$
31	$1s$
38	$4s$
45	$8s$
52	$17s$
59	$30s$
66	$55s$
73	$1m29s$
80	$2m25s$
87	$3m30s$
94	$5m21s$
101	$7m22s$
108	$10m38s$
115	$14m1s$
122	$19m29s$
129	$25m18s$
136	$33m20s$
143	$41m36s$
150	$54m19s$

Series-parallel graphs
time $O(nr^4)$
space $O(nr^2)$

$\|V(G)\|$	*time*
1625	$0s$
3250	$1s$
4875	$2s$
6500	$4s$
8125	$8s$
9750	$12s$
11375	$12s$
13000	$17s$
14625	$30s$
16250	$28s$
17875	$36s$
19500	$42s$
21125	$53s$
22750	$1m2s$
24375	$1m14s$
26000	$1m25s$
27625	$1m38s$
29250	$1m52s$
30875	$2m40s$
32500	$2m23s$

Trees
time $O(nh)$
space $O(nh)$

Table 7.3: Running times for all algorithms on worst case input



Interval graph algorithm (time $O(n^3)$, space $O(n^2)$)

Series-parallel graph algorithm (time $O(nr^4)$, space $O(nr^2)$)

Tree algorithm (time $O(nh)$, space $O(nh)$)

Figure 7.1: Running times for all algorithms on worst case input

Results

The results are displayed in Table 7.4, and are plotted in Figure 7.2 along with their corresponding running time on paths (in dotted lines).

As expected, the performance of the interval graph algorithm is the same on randomly generated graphs as for paths, as its running time is only dependent on the number of vertices in the input graph.

Also as expected, the performance of the tree algorithm was vastly improved, as the height of the input trees is much lower. The height of a “path tree” of 32500 vertices is 16250, whereas in the randomly generated tree the average height was 23.2.

The performance of the series-parallel algorithm was also significantly better, because the radius of a randomly generated series-parallel graph is much smaller than for a path. The radius of a path of 107 vertices is 53, while for the randomly generated graphs of the same size, the average radius was 15.2.

As is easily seen in the figure, running the algorithm on two series-parallel graphs of the same size can give very different running times. This is mainly because two series-parallel graphs of the same size can be very different, and thus they have different radius values and different decomposition trees of different sizes. Also, the number of parallel versus series vertices in the decomposition tree will affect the running time. By looking at the tables in Chapter 5 we see that the calculation of the parallel operation is much more time consuming than the series operation.

7.4 Optimal cost versus radius

It will also be interesting to see how much the optimal broadcast cost actually differs from the cost of a radial broadcast. If the optimal broadcast cost for one algorithm is nearly always equal to a radial broadcast, that is $\gamma_b(G) = rad(G)$, then it is less probable that we will find much practical use for the algorithm.

To test this, we have run the algorithms on randomly generated graphs up to size 1000 for trees and interval graphs and 100 for series-parallel graphs. We selected 20 equidistant sizes for each graph class, and generated 20 graphs of each class for each size. Since the degree of a tree may affect the optimal dominating broadcast cost, we have tested both for trees of maximum degree 3 and for trees of maximum degree 6. All intervals were constrained to be of maximum width 10, which will result in larger radiuses for larger graphs.

7.4.1 Results

The results of the test are shown in Table 7.5. Contrary to our expectations, trees with degree 3 had the same number of radial broadcasts being optimal as trees of degree 6.

From the results we see that for an arbitrary tree, a radial broadcast is almost always an optimal dominating broadcast. The results from the interval graphs and series-parallel graphs, however, show that the cost of a optimal dominating broadcast is usually less than that of a simple radial broadcast.

We can therefore conclude that, if a practical use for broadcast domination appears, the algorithms for interval graphs and series-parallel graphs are immediately useful, but the algorithm for trees is unlikely to be useful on general

$\lvert V(G)\rvert$	avg. time
185	$0.0s$
370	$0.6s$
555	$1.8s$
740	$4.0s$
925	$8.0s$
1110	$13.8s$
1295	$22.2s$
1480	$33.0s$
1665	$47.4s$
1850	$64.8s$
2035	$86.6s$
2220	$113.4s$
2405	$144.8s$
2590	$181.8s$
2775	$227.0s$
2960	$278.2s$
3145	$339.4s$
3330	$406.2s$
3515	$487.0s$
3700	$579.6s$

Interval graphs
time $O(n^3)$
space $O(n^2)$

$\lvert V(G)\rvert$	avg. time	avg. n	avg. r
12	$0.0s$	29.2	3.8
17	$0.0s$	41.0	5.7
22	$0.0s$	53.8	6.3
27	$0.1s$	67.4	7.1
32	$0.1s$	79.8	8.1
37	$0.1s$	92.8	7.6
42	$0.1s$	104.6	8.9
47	$0.8s$	118.2	10.5
52	$0.8s$	128.8	10.8
57	$1.0s$	145.8	10.4
62	$2.1s$	158.6	11.0
67	$2.2s$	171.6	11.6
72	$4.3s$	182.8	14.0
77	$1.6s$	199.0	11.0
82	$3.3s$	207.4	12.6
87	$2.5s$	226.6	10.6
92	$9.7s$	235.4	13.4
97	$7.4s$	247.4	13.8
102	$9.7s$	259.8	15.4
107	$11.2s$	275.8	15.2

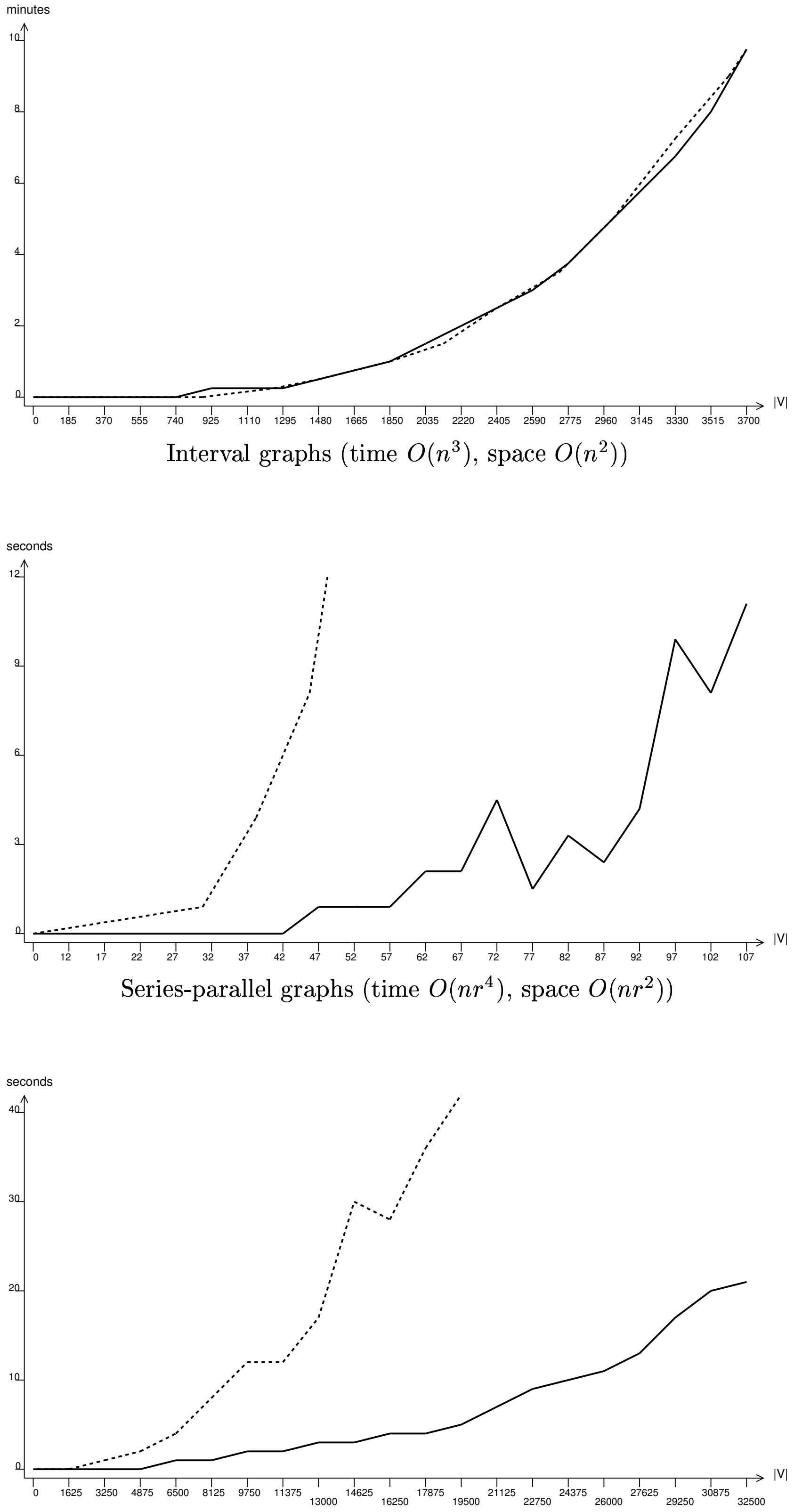
Series-parallel graphs
time $O(nr^4)$
space $O(nr^2)$

$\lvert V(G)\rvert$	avg. time	avg. h
1625	$0.2s$	15.4
3250	$0.4s$	17.0
4875	$0.4s$	19.4
6500	$1.0s$	18.8
8125	$1.0s$	20.0
9750	$1.6s$	19.4
11375	$1.8s$	19.8
13000	$2.6s$	22.0
14625	$3.0s$	21.4
16250	$3.6s$	22.0
17875	$4.2s$	21.4
19500	$5.0s$	21.2
21125	$6.8s$	22.4
22750	$8.6s$	22.8
24375	$9.4s$	22.4
26000	$11.0s$	22.4
27625	$12.8s$	22.2
29250	$17.2s$	24.0
30875	$20.6s$	23.6
32500	$21.4s$	23.2

Trees
time $O(nh)$
space $O(nh)$

Table 7.4: Running times for all algorithms on randomly generated graphs

trees. However, it is possible that trees of a special structure can benefit from the algorithm (see 9.2.3).



Interval graphs (time $O(n^3)$, space $O(n^2)$)

Series-parallel graphs (time $O(nr^4)$, space $O(nr^2)$)

Trees (time $O(nh)$, space $O(nh)$)

Figure 7.2: Running times for all algorithms on randomly generated graphs

$\|V(G)\|$	avg. $rad(G)$	avg. $\gamma_b(G)$	avg. diff.
50	4.15	4.05	0.10
100	7.75	7.50	0.25
150	11.20	10.80	0.40
200	15.00	14.60	0.40
250	18.35	17.95	0.40
300	22.10	21.55	0.55
350	25.65	24.95	0.70
400	29.50	28.60	0.90
450	32.45	31.65	0.80
500	36.20	35.20	1.00
550	40.30	39.30	1.00
600	44.10	42.55	1.55
650	47.05	46.00	1.05
700	50.85	49.75	1.10
750	54.50	53.05	1.45
800	58.60	57.15	1.45
850	61.90	60.30	1.60
900	64.75	63.05	1.70
950	68.40	66.80	1.60
1000	72.15	70.75	1.40

Interval graphs

$\|V(G)\|$	avg. $rad(G)$	avg. $\gamma_b(G)$	avg. diff.
5	2.00	2.00	0.00
10	3.50	2.95	0.55
15	5.10	4.30	0.80
20	5.40	4.55	0.85
25	7.10	5.95	1.15
30	7.95	7.05	0.90
35	8.65	7.60	1.05
40	10.05	8.60	1.45
45	10.05	9.05	1.00
50	10.90	9.75	1.15
55	10.70	9.65	1.05
60	9.75	9.05	0.70
65	11.65	10.80	0.85
70	13.25	12.15	1.10
75	12.95	11.50	1.45
80	13.85	12.65	1.20
85	14.90	13.15	1.75
90	13.30	11.55	1.75
95	14.20	13.05	1.15
100	14.50	13.85	0.65

Series-parallel graphs

$\|V(G)\|$	avg. $rad(G)$	avg. $\gamma_b(G)$	avg. diff.
50	6.05	6.05	0.00
100	7.65	7.60	0.05
150	8.70	8.65	0.05
200	9.30	9.30	0.00
250	9.90	9.90	0.00
300	10.80	10.80	0.00
350	10.55	10.50	0.05
400	10.70	10.70	0.00
450	11.10	11.10	0.00
500	10.80	10.75	0.05
550	11.45	11.45	0.00
600	11.40	11.40	0.00
650	12.15	12.15	0.00
700	11.90	11.85	0.05
750	12.45	12.45	0.00
800	12.60	12.60	0.00
850	13.15	13.15	0.00
900	12.80	12.80	0.00
950	13.00	13.00	0.00
1000	13.00	13.00	0.00

Trees of degree 5

$\|V(G)\|$	avg. $rad(G)$	avg. $\gamma_b(G)$	avg. diff.
50	7.90	7.90	0.00
100	9.65	9.65	0.00
150	11.35	11.30	0.05
200	12.15	12.15	0.00
250	12.95	12.90	0.05
300	13.45	13.45	0.00
350	14.15	14.10	0.05
400	14.20	14.20	0.00
450	14.90	14.90	0.00
500	15.30	15.30	0.00
550	15.65	15.65	0.00
600	15.70	15.70	0.00
650	15.90	15.90	0.00
700	16.65	16.60	0.05
750	16.50	16.50	0.00
800	16.95	16.95	0.00
850	17.25	17.25	0.00
900	17.65	17.60	0.05
950	17.95	17.95	0.00
1000	18.00	18.00	0.00

Trees of degree 2

Table 7.5: Average $rad(G)$, average $\gamma_b(G)$, and average $rad(G) - \gamma_b(G)$ for 20 runs on each graph size.

Chapter 8

Additional Algorithms

This chapter describes two additional algorithms that were *not* presented in [1] but are necessary for the implementations to be useful. They are both sufficiently simple and efficient that it was probably faster (and more fun) to formalise them from scratch rather than trying to find existing, more efficient and still sufficiently simple algorithms.

8.1 Computing the series-parallel decomposition tree

During implementation and debugging it was necessary to create a sub-program that generated the decomposition tree for a given series-parallel graph G, since it was a lot of work to do this manually and the algorithm needed to be tested on graphs consisting of at least a few hundred vertices.

Also, it is assumed that anyone who wants to use the implementation to find the optimal dominating broadcast of a series-parallel graph would not want to decompose the graph manually.

Assume each vertex in G has an associated unique integer in the range $[1 \dots n]$ and that the edges of G are stored in a table $E(1 \dots n, 1 \dots n)$ where $E(u, v)$ denotes the number of edges between u and v. We will also be using a table of lists, $R(1 \dots n, 1 \dots n)$, for storing the reference from an edge to its corresponding vertex in the decomposition tree. Thus, $R(i, j)$ contains a list of the vertices in the decomposition tree corresponding to all the edges between i and j in the currently decomposing graph. After each iteration of the main loop, $E(u, v) = |R(u, v)|$.

The algorithm will, in each step, shrink G by either removing one double edge (reverse parallel operation) or shrinking a P_3 to a P_2 (reverse serial operation).

```
**Algorithm** Series-Parallel Decomposition (SPD)
**Input:** A series-parallel graph $G$ with adjacency table $E$
**Output:** The series-parallel decomposition tree of $G$
$D :=$ new, empty decomposition tree
$R :=$ new table of sets, of size $|V(G)| \times |V(G)|$.
// *initialization phase*
**for** $u = 1$ to $n - 1$ **do**
```

```
    for $v = u + 1$ to $n$ do
        $w$ := new decomposition-vertex(left-terminal= $u$, right-terminal= $v$,
        SP-operation= $NONE$, left-child= 0, right-child= 0)
        $D.add(w)$
        $R(u, v) := \{w\}$
    end for
end for
// decomposition phase
while $|E(G)| > 1$ do
    Find, if possible, a $u \in V(G)$ such that $degree(u) = 2$
    if found $u$ then
        $v_1, v_2 = u$'s two first neighbors.
        $E(v_1, u) := 0$
        $E(u, v_2) := 0$
        $E(v_1, v_2) := E(v_1, v_2) + 1$
        $w$ := new decomposition-vertex(left-terminal= $v_1$, right-terminal= $v_2$,
        SP-operation= $SERIAL$, left-child= $R(v_1, u)$, right-child= $R(u, v_2)$)
        $D.add(w)$
        $R(v_1, v_2) := R(v_1, v_2) \cup \{w\}$
    else
        find $u, v$ such that $E(u, v) > 1$
        $v_1 := u$, $v_2 = v$
        $r_1, r_2$ = two first elements of $R(v_1, v_2)$
        $E(v_1, v_2) := E(v_1, v_2) - 1$
        $w$ := new decomposition-vertex(left-terminal= $v_1$, right-terminal= $v_2$,
        SP-operation= $PARALLEL$, left-child= $r_1$, right-child= $r_2$)
        $D.add(w)$
        $R(v_1, v_2) := R(v_1, v_2) \cup \{w\} - \{r_1, r_2\}$
    end if
end while
Return $D$
```

Proof. At each step of the decomposition phase, the exact part of G that was removed in that step can be constructed from the new vertex of the decomposition tree. Thus, the entire original graph G can be constructed from the final decomposition tree and the algorithm is correct. □

8.1.1 Running time and resource usage

The initialization phase takes time $O(\frac{n(n-1)}{2})$. Since each iteration of the *while* loop will reduce the number of edges in G by one, it will take time $O(|E(G)|)$. Finding a vertex of degree 2 can be done in time $O(n)$ if we keep count of the neighbors of each vertex at all times. Finding a double edge can be done trivially in time $O(|E(G)|)$. Since the number of edges is bounded by n^2, this gives a total running time of $O(n^4)$. Although this is not by any means an ideal running time, it is sufficient for our purposes.

As for memory usage, the only data structures used, apart from the graph and the decomposition, is the table R. Since this table takes no more space than the input graph, the memory used by this algorithm is of no concern.

8.2 Computing the radius and diameter

In 7.4 in the test chapter, and in the algorithm for series-parallel graphs, we needed to know the radius of the input graph.

Since the radius and diameter of a general graph G are defined as $rad(G) = \min\{\max\{d(u,v) : v \in V(G)\} : u \in V(G)\}$ and $diam(G) = \max\{\max\{d(u,v) : v \in V(G)\} : u \in V(G)\}$, the simplest way to calculate them is to first generate a table of the length of the shortest path between each pair of vertices, and then apply the definitions directly to the table. Floyd-Warshall All-Pairs-Shortest-Path (APSP) algorithm creates such a table and its running time of $O(n^3)$ is faster than the main algorithm of the series-parallel algorithm, so it was fast enough for our purposes.

```
Algorithm Radius and Diameter
Input: The edge table $E$ from a general graph $G$
Output: Radius and diameter of $G$
// Floyd-Warshall algorithm
$D_0 := E(G)$
for $1 \leq k \leq n$ do
    for $1 \leq u \leq n$ do
        for $1 \leq v \leq n$ do
            $D_k[u,v] := min\{D_{k-1}[u,v], D_{k-1}[u,k] + D_{k-1}[k,v]\}$
        end for
    end for
end for
// Radius and diameter calculation
$radius := \min_{u \in V}\{\max_{v \in V}\{D_n[u,v]\}\}$
$diameter := \max_{u,v \in V}\{D_n[u,v]\}$
Return $radius, diameter$
```

8.2.1 Proof of correctness

There is no need to prove the Floyd-Warshall algorithm here, and the calculation of radius and diameter are straightforward from their definitions, so the algorithm is correct.

8.2.2 Running time and resource usage

The running time of the Floyd-Warshall APSP algorithm is $O(n^3)$ and the calculations of radius and diameter are both $O(n^2)$, so the overall running time is $O(n^3)$. By discarding the D-tables that are no longer accessed, i.e. for each iteration k, we can discard D_{k-1}, the memory requirements of the Floyd-Warshall APSP algorithm is $2n^2$ integers.

Chapter 9

Conclusion

This chapter will summarise the work that has been described in the preceding chapters, and present the opportunities for further work that have turned up while working on this thesis.

9.1 Summary

The algorithms from [1] that were described in Chapters 3, 4, and 5 have been implemented into usable programs and extended to yield an optimal broadcast function as part of the output. During the implementation and preliminary testing, we detected some small errors in the original algorithms, which have been corrected. All the corrections have been approved by the authors.

Automated correctness tests have given the expected results, and combined with the formal proofs of the algorithms, we see this as sufficient proof that the algorithms are now correct and that the implementations follow the algorithms correctly.

The results from the performance tests have shown how large input data the implementations can be expected to handle when run on a modern workstation, and is also an indication of how large input data the algorithms themselves can reasonably be expected to handle in any situation.

The tests on how the optimal dominating broadcast cost relates to the radius of the input data have shown that for interval graphs and series-parallel graphs, there usually exists a solution which is not radial. Therefore, should a practical use for Optimal Broadcast Domination appear, the algorithms are immediately useful.

The algorithm for trees, however, is practically useless for general trees, as the optimal solution very rarely differs from a radial broadcast. Thus, for the trees algorithm to have practical use, some subclass of trees must be found that has a specific structure where optimal broadcasts are usually not radial (see 9.2.3).

All source code for the algorithms, graph generation programs and automated tests, as well as some sample data, is available from `http://www.ii.uib.no/~helgeh/master/` or by e-mailing the author at `helge.holm@gmail.com`.

9.2 Further work

Anyone who wishes to perform further work in this area may be interested in looking into the problems presented in the following sections.

9.2.1 Parallel implementation

To be able to run the algorithms on very large problem sizes, it would be useful to implement the algorithms in such a way that they could be run on multiprocessor computers.

For example, the trees algorithm and the series-parallel algorithm could cut the input tree (remember that the series-parallel algorithm works on a decomposition tree) at some specific height such that it would produce k different subtrees, pass each subtree to a different processor, and then calculate the remaining top of the tree on a single processor. The algorithm for interval graphs, however, has no obvious way to partition the workload, but research can probably develop such a method.

9.2.2 Broadcast function and partitioning of vertices

Given an efficient, dominating broadcast f on a graph G, we can also partition G such that each subgraph is optimally dominated by a radial broadcast. This is done by numbering all broadcasting vertices $b_1, b_2, \cdots, b_k$, and let each partition V_i contain only b_i and those vertices dominated by b_i. Each induced subgraph $G[V_i]$ is then a graph of radius $f(b_i)$.

Because f is efficient, the distance in G from b_i to any vertex not in V_i is always more than $f(b_i)$.

By investigating this further, it may be possible to find that Optimal Broadcast Domination is related to some partitioning problem.

9.2.3 Tree structure benefiting from Optimal Broadcast Domination

By the above observation that the distance in G from b_i to any vertex not in V_i is more than $f(b_i)$, it may be possible to devise a tree structure where the optimal broadcast will not be radial by first partitioning the vertices and then create the tree from the partitioning.

Although it is not very useful to run the algorithm on such a constructed tree, it may be possible to find some problem where we somehow know that a tree has this structure. Then, the algorithm may be used to find the subtrees corresponding to the original partitions.

9.3 Variable cost function

As mentioned in Chapter 2, a cost function where the cost of providing a broadcast $f(v)$ is not equal to $f(v)$ would yield a more realistic problem specification, and the authors of [1] claim that the algorithms can easily be adapted to a different cost function. However, if the cost function is polynomial we are no longer guaranteed that there exists a broadcast that is both optimal and efficient (see figure 9.1). Since the algorithms for trees and interval graphs are both designed and proven under the assumption that for every optimal dominating broadcast there exists an efficient dominating broadcast of equal cost, a different broadcast cost function can not be used without reworking these algorithms.

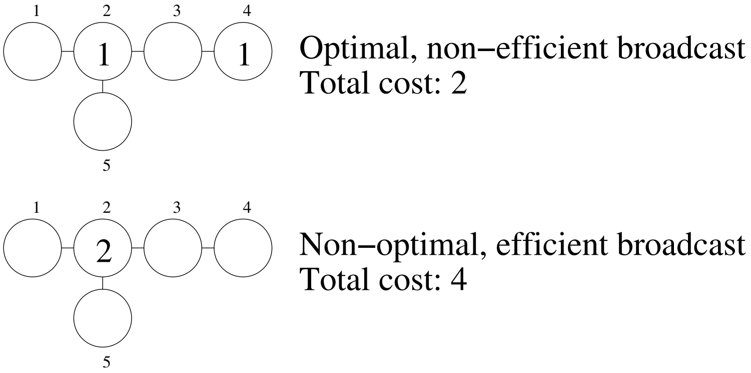

1
2
3
4
1
1
Optimal, non–efficient broadcast
Total cost: 2
5
1
2
3
4
2
Non–optimal, efficient broadcast
Total cost: 4
5

Figure 9.1: Example of non-linear cost function

> Given the graph G above and the cost function $cost(f(v)) = f(v)^2$, we see that the only optimal broadcast will be $f(v_2) = f(v_4) = 1$ with a total cost $\gamma_b(G) = 2$, as nothing less will be able to dominate G, and all other broadcast values than 1 will be larger than the optimal cost 2. In the optimal graph, v_3 can hear both v_2 and v_4, so it is not an efficient broadcast.

The algorithm for series-parallel graphs is not proven under this assumption, so it should still work with other cost functions.

9.3.1 Speeding up the algorithm for series-parallel graphs

The reader may have noticed that the algorithm for series-parallel graphs performs a lot of redundant calculations. Following is a method for eliminating this redundancy, which was unfortunately not implemented in the algorithm due to time constraints.

Since all leaves in the decomposition tree are structurally equal, i.e. they are K_2s, a series-parallel operation on two leaves v_1, v_2 will yield the exact same subgraph as the same series-parallel operation on two other leaves v_3, v_4. This observation can be generalised further: Any two vertices in the decomposition tree whose corresponding subtrees are identical (disregarding all left and right terminals), yield identical subgraphs and thus identical computation tables. For a vertex v, we may exactly describe v by the operations performed in the subtree rooted at v. We will call such a description the *operation history* of v. An example of an operation history for a C_5 is $p\left(s\left(s\left(K_2, K_2\right), s\left(K_2, K_2\right)\right), K_2\right)$, where $p(a, b)$ is a parallel operation on subgraphs a and b, $s(a, b)$ is a series operation on subgraphs a and b, and the K_2s are leaves.

We can store each calculated set of the calculation tables (N, L, R, and B) in a list indexed by operation history, and not re-calculate a set of tables if their corresponding operation history is already stored in the list. Each vertex

v_i in the decomposition tree would then store a reference to the set of tables corresponding to the operation history of v_i.

For simplicity, we will assume that the decomposition trees are balanced (i.e. of minimum height), and we will refer to the calculation of a set of tables N, L, R, and B as one calculation.

For a decomposition tree of height h, this would result in the number of calculations for height $h - 1$ to be at most 2, since there can only exist two different subgraph structures at this height, namely the ones resulting from series operations and the ones resulting from parallel operations. At height $h-2$, we have a maximum of $2^2 = 4$ distinct subgraph structures, and to generalise, at height j, we will have a maximum of 2^{h-j} distinct subgraph structures. The other way around, at height j we will have a maximum of 2^j different vertices, also bounding the number of different operation histories for that height. See Figure 9.2 for a simple illustration.

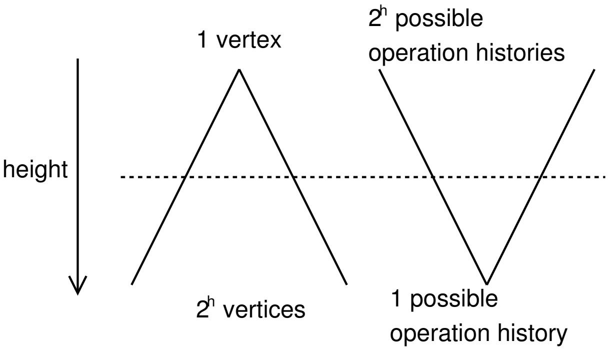


Figure 9.2: Number of calculations needed per tree height

As a result, the maximum number of calculations performed per tree height will be at tree height $\frac{h}{2}$, and the number of calculations will be halved for every 1 height upwards and downwards. In the existing algorithm, the maximum number of calculations performed per tree height is at the bottom of the tree, and the number of calculations are halved for every 1 height upwards. This means the number of calculations needed in the existing algorithm would be $2^{h+1} - 1$, and in the modified algorithm the number would be $2(2^{h/2+1} - 1)$.

Since the decomposition tree is a binary tree with m leaves, where m is the number of edges in the corresponding series-parallel graph, the height is $log_2(m)$. This gives us $2m-1$ calculations for the existing algorithm, and $2((2m-1)^{1/2}-1)$ for the modified algorithm. Asymptotically, this is a reduction from $O(m)$ to $O(\sqrt{m})$ which is a significant optimisation.

Although these calculations are based on the assumption that the decomposition tree is balanced, this method should still significantly speed up the algorithm.

Bibliography

[1] J. R. S. Blair, P. Heggernes, S. Horton, F. Manne, *Broadcast Domination Algorithms for Interval Graphs, Series-Parallel Graphs and Trees*, Reports in Informatics 249, University of Bergen, 2003.

[2] S. B. Horton, R. G. Parker and R. B. Borie, *On minimum cuts and the linear arrangement problem*, Disc. Appl. Math., 103 (2000), pp. 127-139.

[3] S. B. Horton, *The Optimal Linear Arrangement Problem: Algorithms and Approximation*, PhD thesis, School of Industrial and Systems Engineering, Georgia Institute of Technology, Atlanta, 1997.

[4] M. Richey and R. Parker, *On finding spanning Eulerian subgraphs*, Naval Research Logistics Quarterly, 32 (1985), pp. 443-455.

[5] J. Valdes, R. E. Tarjan and E. L. Lawler, *The recognition of series parallel digraphs*, SIAM J. Comput., 11 (1982), pp. 298-313.

[6] M. Sipser, *Introduction to the Theory of Computation*, ISBN 0-534-94727-X, pp. 234-245.

[7] D. J. Erwin, *Dominating broadcasts in graphs*, (2002). Submitted.

[8] T. W. Haynes, S. T. Hedetniemi and P. J. Slater, *Domination in Graphs: Advanced Topics*, Marcel Dekker, New York, 1998.

[9] —, *Fundamentals of Domination in Graphs*, Marcel Dekker, New York, 1998.

[10] C. L. Liu, *Introduction to Combinatorial Mathematics*, McGraw-Hill, New York, 1968.

[11] M. A. Henning, *Distance domination in graphs*, in Domination in Graphs: Advanced Topics, T. W. Haynes, S. T. Hedetniemi and P. J. Slater, eds., Marcel Dekker, New York, 1998, pp. 321-349.

[12] P. J. Slater, *R-domination in Graphs*, J. Assoc. Comput. Mach., 23 (1976), pp. 446-450.

[13] J. E. Dunbar, D. J. Erwin, T. W. Haynes, S. M. Hedetniemi and S. T. Hedetniemi, *Broadcasts in graphs*, (2002), Submitted.

[14] M. R. Garey and D. S. Johnson, *Computers and Intractability*, W. H. Freeman and Co., 1978.

[15] A. K. Dewdney, *Fast Turing reductions between problems in NP*, Tech. Rep. 71, Dept. of Computer Science, University of West Ontario, 1981.

[16] K. S. Booth and J. H. Johnson, *Dominating sets in chordal graphs*, SIAM J. Comput., 11 (1982), pp. 191-199.

[17] D. Kratsch, *Domination and total domination in asteroidal triple-free graphs*, Tech. Rep. Math/Inf/96/25, F.-Schiller-Universität, Jena, 1996.

[18] M. Farber and J. M. Keil, *Domination in permutation graphs*, J. Algorithms, 6 (1985), pp. 309-321.

[19] M. Farber, *Domination, independent domination, and duality in strongly chordal graphs*, Disc. Appl. math., 7 (1984), pp. 115-130.

[20] K. S. Booth and G. S. Lueker, *Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms*, J. Comput. System Sci., 13 (1976), pp. 335-379.

[21] M. C. Golumbic, *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, 1980.

[22] Wikipedia: “Implementation” (22:21, 2004 Aug 10)
`http://en.wikipedia.org/wiki/Implementation`

[23] Wikipedia: “Software testing” (17:49, 2004 Dec 4)
`http://en.wikipedia.org/wiki/Software_testing`