K-packing and K-domination on tree graphs

Morten Mjelde
mortenm@ii.uib.no

Department of Informatics
University of Bergen
Norway

Cand. Scient. Thesis
2004

Acknowledgment

I would like to thank my supervisor Fredrik Manne for his help and advice with this thesis.

I also want to thank my family for their support and patience over the years.

And lastly, I would like to thank my friends for making my time as a student as memorable as it has been.

Contents

1 Introduction **6**

1.1 Introduction 6

1.2 Overview of the problems 7

1.2.1 Maximum K-packing 7

1.2.2 Minimum K-domination 8

1.3 Existing algorithms 8

1.3.1 K-packing 8

1.3.2 K-domination 9

1.4 The basics 9

1.4.1 Assumptions 9

1.4.2 Definitions 10

1.4.3 The basic principle 10

2 Maximum K-Packing on trees **12**

2.1 Introduction 12

2.1.1 Description of K-packing 12

2.2 Phase 1: Calculate the table 12

2.2.1 Leaf nodes 13

2.2.2 Non-leaf nodes: $i = K$ 13

2.2.3 Non-leaf nodes : $i = K - 1, 1$ 13

2.2.4 Non-leaf nodes: $i = 0$ 14

2.2.5 Example 14

2.2.6 History Checks 15

2.2.7 More on heirs 16

2.2.8 Phase 1 in words 16

2.3 Phase 2: Setting the final-distance 17

2.4 Proving correctness 18

2.4.1 Proof for Phase 1 19

2.4.2 Proof for Phase 2 21

2.5 Implementation and running time 21

2.5.1 Implementation 21

2.5.2 Running time 22

2.6 Final example 23

2.7 The algorithm in pseudo-code 24

3 Minimum K-Domination on trees **27**
3.1 Description of K-Domination 27
3.1.1 K-Domination vs K-Packing 27
3.2 The basic principle 28
3.3 Phase 1: Calculating the table 28
3.3.1 Heirs and sibling-distance 28
3.3.2 History Checks 29
3.3.3 Example 30
3.3.4 The four cases 30
3.3.5 Leaf nodes 30
3.3.6 Non-leaf nodes: $i = 0$ 31
3.3.7 Non-leaf nodes: $1 \leq i \leq K$ 32
3.3.8 Non-leaf nodes: $K + 1 \leq i \leq 2K$ 32
3.3.9 Phase 1 in words 32
3.4 Phase 2: Setting the final-distance 32
3.5 Proving correctness 33
3.5.1 Proof for Phase 1 34
3.5.2 Proof for Phase 2 36
3.6 Implementation and running time 36
3.6.1 Implementation 36
3.6.2 Running time 37
3.7 The algorithm in pseudo-code 38

4 Self-stabilizing Algorithms **41**
4.1 General Introduction 41
4.1.1 Example of Self-stabilizing algorithms 42
4.1.2 Multistage Self-stabilizing algorithms 43
4.1.3 Multistage Self-stabilizing algorithms on trees 44
4.1.4 Leader election 45
4.1.5 The two phases 46
4.2 Self-stabilizing K-Packing 48
4.2.1 Phase 1 48
4.2.2 Phase 2 50
4.3 Self-stabilizing K-Domination 52
4.3.1 Phase 1 52
4.3.2 Phase 2 53
4.4 Combining the three algorithms 55
4.5 Conclusion 55

5 Overall conclusion **56**
5.1 Further development 57
5.2 Open problems 57
5.2.1 Other problems 57
5.2.2 Tree-decomposition 58

Chapter 1

Introduction

1.1 Introduction

There are many problems common in computer science that have known polynomial algorithms for solving them (commonly referred to as problems in P). There are also a great many problems that have non-polynomial algorithms for solving them (referred to as problems in NP). It has long been known that every problem in P is also in NP, however, whether or not the reverse is true remains an open question. At present, any algorithm for solving a problem not known to be in P could possibly take years to provide a solution, even on relatively small inputs. Since the difference between P and NP was initially hinted on in the mid 1960's, much effort has gone into proving whether or not P = NP ([9] is highly recommended for further reading about this topic). This research has spawned a number of subsets of the problems in NP, NP-complete and NP-hard being two common, but as of yet, no definitive answer to the question if P = NP has appeared. Because of this, and because there are many practical situations that requires a solution to problems currently not known to be in P, many have sought out ways to try to solve these problems in a reasonable amount of time. Among the most common ways of doing this is restricting the problem to certain instances, using approximation algorithms, randomized algorithms, parametrized complexity algorithms and so on (for further reading about randomized and approximation algorithms I recommend [16], and [17] for more information on parameterized complexity). In this thesis, we will look at two problems with no known polynomial algorithms, and restrict them such that they can be solved in polynomial time.

Self-stabilizing algorithm, a variant of distributed algorithms, is a relatively new concept in computer science. Although it was first introduced in 1974, serious work did not begin until the late 1980's. The details of self-stabilizing algorithms will be described later, but in short, each node in a graph is an independent unit, without any knowledge of the rest of the graph, save itself and its neighbors. In other words, no single, all knowing entity can be assumed to exist, and the nodes will have to solve a given problem in unison. This makes for an often complicated process of finding a self-stabilizing algorithm. However self-stabilizing algorithms is also an area of study that will no doubt receive much attention in years to come, because of its inherit applications on any type of wireless network, which are becoming more and more common both in various businesses and in the day to day lives of countless people. The most readily available examples of common wireless situations are mobile phones, and two standards for wireless communication: Bluetooth and the 802.11 standard. Bluetooth is already commonly used in smaller, short-ranged units, such as speakers, headphones, microphones and so on, and the 802.11 standard is becoming more and more common in wireless computer networks both in office environments and in homes. There are many challenges inherit to wireless networks (frequency allocation for example), and self-stabilizing algorithms provides a theoretical basis for studying and overcoming these challenges.

The most obvious reason for solving a problem, any problem, is of course "because it is there"[1]. Beyond that however, the purpose of this thesis is two-fold: First, develop a self-stabilizing algorithm that runs in polynomial time for solving maximum K-packing (an NP-hard problem) on tree graphs, and second, show that the same method can be applied to another similar problem, namely minimum K-domination (which is NP-complete), also on tree graphs (relations between these two problems are studied in great detail in [13]). There are many practical situations that resembles the above two problems (more on this later), therefor solving them is not solely due to academic interests. The following chapters will first describe two sequential algorithms, and then two self-stabilizing algorithms. The reason for this ordering is that since most readers are no doubt more familiar with sequential algorithms than self-stabilizing algorithms, it ultimately proved easier to first show a sequential version, and then a self-stabilizing version.

1.2 Overview of the problems

To start of, a small introduction of the two problems we will be working on might be in order, as well as examples of practical applications for a solution to them. This section will only give a brief overview of the problem. A more specific description will be given in the following chapters.

1.2.1 Maximum K-packing

K-packing on a graph means that we select a set of the nodes in the graph (denoted black nodes), such that no pair of black nodes has less then K nodes between them. Finding a maximum K-packing on a graph means finding the maximum number of black nodes that can legally exist in the graph. For a general graph, this problem is NP-hard.

[1]This was the answer given by George Mallory in 1923 when asked why he wanted to climb Mount Everest. Mallory and Andrew Irvine died during the climb the following year, and even to this day, no-one knows if they ever reached the summit.

There are several practical applications of an algorithm for solving maximum K-packing. Imagine a case where we have a wireless network where there are several base stations that needs to be placed around a given area. Suppose we prioritize the speed of the network, and that means that we have to limit the workload of each base station. In other words, assuming that every station will receive approximately the same amount of users, we have to place as many of them as we can, without any of them interfering with each other. If the transmitting power of one base station was $K/2$, that means that the distance between any two base stations has to be K or greater. As we see, if black nodes in the graph are the base stations, this is exactly the problem maximum K-packing, set in a practical situation.

1.2.2 Minimum K-domination

K-domination means finding a set of nodes in the graph (black nodes) such that no node in the graph has a distance equal or greater then K from at least one black node. Minimum K-domination means that the number of black nodes is the smallest possible if the above condition is to be met for every node in the graph. For a general graph, this problem is NP-complete.

Just as with maximum K-packing, minimum K-domination has several practical applications. Imagine the same situation as described in the above section, where we have a wireless network, and a number of base stations to place around an area, thus allowing users in that area to access the wireless network. However, in this case, base stations are expensive, and we want to use as few of them as possible, while still covering the entire area. In this case, every station has a transmitting power of K, and if the base stations are the black nodes, we see that this situation is very similar to minimum K-domination.

1.3 Existing algorithms

Before we begin solving the problems given in the above section, it might be interesting to consider what has been achieved by others for the problems maximum K-packing and minimum K-domination.

1.3.1 K-packing

As stated above, maximum K-packing is known to be NP-hard, and therefore any known algorithm for solving the problem on a general graph would be expected to run in exponential time, even if the graph happens to be a tree. The same is true for any self-stabilizing algorithm. Gairing et. al. shows in [12] a self-stabilizing algorithm for solving a variant of the problem, maximal 2-packing (note that maximal 2-packing means that no proper super-set of the black nodes constitutes a legal solution, whereas maximum 2-packing means that no better solution over all possible sets of black nodes exists), on a general graph. As would be expected, it runs in exponential time. A reference search for K-packing algorithms on trees turned up very little, and it would seem as if little work has been done on the subject.

The well known problem maximum Independent Set (which is NP-hard) is another sub-problem of maximum K-packing. Maximum Independent Set has polynomial algorithms for solving it on certain special graphs, such as bipartite graphs [5] and chordal graphs [6]. An algorithm for solving maximum Independent Set on a tree runs in linear time[2], and is trivial to the point where actually finding a reference to it proved difficult.

1.3.2 K-domination

As with K-packing, any known algorithm for solving minimum K-domination on a general graph is expected to run in exponential time. This would also be the case for self-stabilizing algorithms. Gairing et. al. shows a minimal K-domination algorithm in [11] that runs in exponential time (they also show a self-stabilizing algorithm for solving 2-domination that runs in linear time). A reference search for K-domination on trees turned up little, so it would appear that this subject has received little attention.

As seen in the above two sections, there is evidently much room for improvement, and in this thesis, we shall see how the above two problems can be solved in polynomial time on a tree, using either sequential or self-stabilizing algorithms (In addition to the references mentioned above, see [20], [21] and [22] for more information about the two problems).

1.4 The basics

In the following chapters four algorithms will be presented. Two of them are sequential algorithms on trees for maximum K-packing and minimum K-domination, respectively. The other two are remakes of the first two into self-stabilizing algorithms that solve the same two problems.

Before we get into the details of how the algorithms functions, a few basic definitions and assumptions that are common in tree algorithms will be presented.

1.4.1 Assumptions

In order for the presented algorithms to work, the following basis is required: We are given an undirected graph $T = (V, E)$ that is a tree. It has a root r, and each of the nodes in the tree knows which of its neighbors is closest to r. It is not uncommon for an algorithms running time to vary depending on which node in the tree graph is chosen

[2]To find the maximum independent set on a tree, the algorithm first removes every leaf node in the tree, and then adds these to the set S. Next it removes every newly formed leaf, and this process is repeated until the tree is empty, at which point the given set S is the maximum independent set

as the root. But as we shall see later, the root may be arbitrarily chosen for all four algorithms in the following chapters. From here on, it is always assumed that the graph is a tree complying with the above.

1.4.2 Definitions

The following common graph definitions are used: The *parent* of a node is the neighbor that is closest to the root. The *children* of a node are all the nodes that it is a parent to (here after denoted as C_v for a node v), and *siblings* are the nodes that a node shares its parent with. The number *d(v,u)* is the number of nodes in the shortest path between the nodes v and u[3]. The tree *T[v]* is the subtree of T that has the node v as its root. The nodes in *N(i)* is the open neighborhood of the node i, containing all the nodes that i is a neighbor to, and $N[i]$ is the closed neighborhood of i such that $N[i] = N(i) \cup i$. A *leaf* is a node that has no children. Ancestor and descendant are terms used to described the relationship of two nodes that are not necessarily neighbors. Given two nodes v and u such that $u \in T[v]$, v is an ancestor of u and u is a descendant of v. A *level* in a tree is a set of nodes that are all exactly the same distance from the root.

1.4.3 The basic principle

The two sequential algorithms presented in the two following chapters are in many ways very similar, and therefor a general overview could be helpful in understanding the relationship between them. The algorithms are based on dynamic programming. They go through two phases. The first phase starts at the leaves, and works its way to the root. The second phase starts at the root and works its way back to the leaves. After the calculations has been completed in Phase 1 for a node v, the algorithm knows the size of the optimal solution for $T[v]$. Thus, at the end of Phase 1, the calculations has been completed for the root, and the size of the optimal solution has been found for the entire tree.

The second phase makes another pass through the tree, this time from the root down. During this phase the information from Phase 1 is used to set the details of the optimal solution found in Phase 1.

This approach to solving problems on trees is not uncommon, and it has been used before, such as in [1] and for weighted independent set on trees, both of which requires two passes through the tree in order to find an optimal solution. It should be noted that for both the following problems there does not have to be only one optimal solution for a given graph. Multiple solution could present equally good results, and the choice between them becomes arbitrarily.

[3]This definition of distance in a graph is not often used. In most cases distance refers to the number of edges, not nodes, between two nodes. The latter is used here because it eases the understanding of both the problems and the algorithms

The following two chapters will each present an algorithm to solve the problems of maximum K-packing and minimum K-domination on trees. The fourth chapter will show how to modify these two algorithms into self-stabilizing algorithms.

Chapter 2

Maximum K-Packing on trees

2.1 Introduction

The following chapter present an algorithm for solving maximum K -packing on a tree graph. A general definition of K -packing will first be presented, followed by the various details in the algorithm.

2.1.1 Description of K-packing

The goal of a K-Packing algorithm is to find a subset S of nodes in a graph so that no pair of nodes in S has a distance less than K from one another (K is a positive integer). We will denote the nodes in S as *black nodes*, while all other are denoted as *white nodes*. In a maximum K-Packing the number of nodes in S is the maximum number possible over all possible K-Packings for the graph in question. A special case of maximum K-Packing, called maximum 2-Packing, was proven NP-Hard in [4] on a general graph. Maximum 1-Packing is the same as the well known problem maximum Independent Set, which is NP-hard [9]. Obviously, maximum K-Packing must also be at least NP-hard, and therefore no polynomial algorithm is likely to exist. The goal of this chapter is to develop a polynomial algorithm that solves maximum K-Packing on a graph that is a tree, that is, a graph without cycles.

2.2 Phase 1: Calculate the table

The first phase of the algorithm is intended to let each node v compute the maximum number of black nodes $T[v]$ can have regardless the choice any of the nodes in $V - T[v]$ can make with regards to color. To accomplish this, we introduce the following: Each node v has a table $M_v[0 : K]$. Each position i in this table contains the maximum number of black nodes the subtree $T[v]$ can have provided that there are at least i levels of white nodes beneath v, including v itself (that is, the distance from v to any black node in its subtree has to be $(i-1) mod(K+1)$ or greater). So if $i = 0$, it would imply that the node v may become black. If $i = 1$ then v must be white, but a child of v may become black, and so on. Any position except 0 implies that v is white. During the first phase of the algorithm, it will compute this table for every node in the tree, starting with $i = K$ and working its way up to $i = 0$. The purpose of $M_v[]$ is to provide the parent of v with the information needed for it to compute its own table. When the root r finishes Phase 1, $M_r[0]$ gives the number of black nodes in an optimal solution.

There are two types of nodes to consider, leaf nodes, and non-leaf nodes. In addition, we consider three different cases for each non-leaf node, depending on i: Either $i = K$, $i = (K-1) : 1$, or $i = 0$. In the following sections, all four cases will be dealt with. Recall that the purpose of Phase 1 is to compute the table $M_r[]$, and to do this, it must be computed for every node in the tree, starting with the leafs. The following four sections will give a somewhat loose description of the difficulties associated with each of the four cases. A proof will not be given in these four sections, instead a complete proof will be given near the end of the chapter.

2.2.1 Leaf nodes

The leaf nodes are the simplest to compute. The table $M_v[]$ is simply set to 1 at $M_v[0]$ and 0 at every other position. This is fairly obvious, since the only way $T[v]$ can have a black node is if v itself is that node.

2.2.2 Non-leaf nodes: $i = K$

In this case, the node v is not a leaf, and we wish to compute $M_v[K]$ (meaning that $i = K$). Recall that the table index stated how many levels of white nodes has to exist beneath v (including v itself). So in this case, v's children and the next $K-2$ levels has to be white. So in order to calculate $M_v[K]$, we can simply sum up the values in all the children's tables at position $K-1$. We can give this as the following equation:

$$M_v[K] = \sum_{c \in C_v} M_c[K-1] \tag{2.1}$$

Note that the distance from the first potential black node in one child's subtree to the first potential black node in another child's subtree is $2 \cdot (K-1) + 1 = 2K - 1 \geq K$ for every $K > 0$. So there is no risk of any two black nodes from different subtrees will come in conflict with each other.

2.2.3 Non-leaf nodes : $i = K - 1, 1$

Here, we describe how the table from $M_v[K-1]$ up to $M_v[1]$ is computed for a non-leaf node. In many of these positions, the equation for calculating the value is identical to Equation 2.1 (of course replacing K with the appropriate table position). However, this is not always the case. Note that the distance from the node v in $T[v]$ to the first possible black node in one of its children's subtree is $i - 1$. Thus the distance from this potentially black node to the first potentially black node in a different child's subtree will be $2 \cdot (i-1)+1 = 2 \cdot i - 1$. So for some values of i, there is the possibility that two black nodes in the subtree $T[v]$ may have a distance less then K from each other. Note that if $i = 0$, there is no risk of a conflict as the one above, since by definition the distance would then be $2 \cdot K + 1$. We can now formulate the following condition:

$$\textbf{if } (2 \cdot i - 1 < K \ \& \ i \neq 0) \tag{2.2}$$

So for every value of i where Condition 2.2 is true, we can not simply use Equation 2.1. In these cases, we have to select one child h from which the parent reads the $i-1$ table position, and for all the other children, the parent has to use a value that has be modified to compensate for the potentially black node in the selected child's subtree. The selected child h is called an *heir*, and the modified value of i is called a *sibling-distance*. Heirs and sibling-distances will be covered in more detail later, but in short the algorithm has to try every child as an heir, and see which one will give the best results. The heir is stored in the table $heir_v[i]$ and the sibling-distance in $sd_v[i]$. The value i and node v has the same meaning as in $M_v[i]$. The non-modified value of i is here on after referred to as a *parent-distance*. The sibling-distance itself is given with the following equation:

$$sd_v[i] = K - (i-1) \tag{2.3}$$

The specific details on finding the heir and sibling-distance will be covered later.

2.2.4 Non-leaf nodes: $i = 0$

Finally, when $i = 0$, the node v may become black. In this case the value $M_v[0]$ can simply be set to the sum of $M_c[1]$ for every $c \in C_v$ and add one for v.

2.2.5 Example

Now that we have a working understanding of most of the algorithm, we can look at the example in Figure 2.1a. There are three nodes present in the figure: v, the root in the subtree, and its two children u and w. $K = 2$ for this example. The node u is a leaf node, while w is a non-leaf node, with a child that is not visible in the figure. The tables $M_u[]$ and $M_w[]$ have already been computed, and only $M_v[]$ remains.

As described above, we start with $M_v[K]$ ($i = K = 2$). Here we can simply use Equation 2.1. Figure 2.1b shows the outcome of this calculation. Next we have position $i = 1$. We need to start by checking if an heir is needed. We see that Equation 2.2 becomes true for $K = 2$ and $i = 1$, so we need to first find the sibling-distance. Using Equation 2.3, we find $sd_v[1] = 2$. So now we can try both u and w and see which of them works best as an heir. As illustrated in Figure 2.1c, we see that using w as an heir gives only a single black node, while u as an heir gives 2. So we can set $M_v[1] = 2$ and $heir_v[1] = u$. This only leaves us with $M_v[0]$ left to compute. As stated in Section 2.2.4, we can simply take the sum of $M_c[2]$ for every $c \in C_v$ and add one for v. As seen in Figure 2.1d, this gives $M_v[0] = 1$.

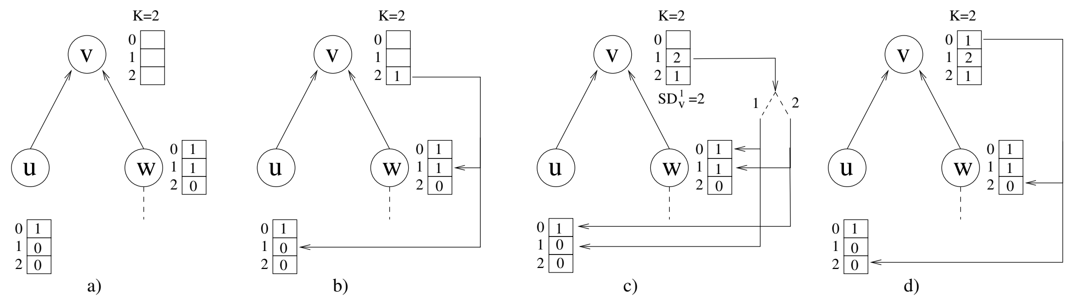


Figure 2.1: K-packing example 1

As one may have noticed, the finished table $M_v[]$ in the above example does not comply with the definition of the table as stated in Section 2.3, since $M_v[0]$ is in fact not an optimal solution, which is 2, as found for $i = 1$. One may also have noticed that while Section 2.3.2 gave an equation tailored to the situation discussed in the section, neither Section 2.3.3 nor 2.3.4 did the same. Both of these "oversights" will be addressed in the following sections.

2.2.6 History Checks

As stated above, the result in $M_v[0]$ in Figure 2.1d is not an optimal solution, as it was intended to be. For while 2 is an optimal solution for $M_v[1]$, it is also an optimal (and legal) solution for $M_v[0]$. It can be easily shown that this mistake can carry on to v's parent, if it had any, and potentially distort the values in several nodes in the tree. So in order to counter this, each position i in $M_v[]$ (for every $i < K$) has to be compared with position $i + 1$ in $M_v[]$, to see if it has a better solution. This means that in the above example, $M_v[1]$ has to be compared to $M_v[2]$, and $M_v[0]$ is compared to $M_v[1]$. In the first case, the original solution was the better one. However, $M_v[1]$ is indeed greater then $M_v[0]$, and thus gives a better solution.

So we see that for every $i < K$ the algorithm has to compare $M_v[i]$ against $M_v[i + 1]$ for every $v \in V$ (except the leaf nodes). This is called a *history check*. The original solution computed for position i is referred to as the *intuitive solution*. We can now formulate the equation that was left out in Section 2.2.4. For any non-leaf node:

$$M_v[0] = max\{M_v[1], 1 + \sum_{c \in C_v} M_c[K]\} \tag{2.4}$$

The effect this would have on the example in Figure 2.1 is that $M_v[0]$ would be set to 2, not 1. This means that the algorithm decides that v is not to become black, even though it could, and instead allows two of its descendants to become black in its place, thereby producing a better solution.

Obviously, there is no need for the algorithm to perform a history check more then one table position further down, since the $i+2$ position has already been checked when position $i+1$ was computed and so on. This means that the table $M_v[]$ for any node $v \in V$ is non-decreasing as i decreases. So the top position, $M_v[0]$ always has the best solution (although it might not be the only one that has it, as in the above example). In order to keep track of where the best solution originated, the following definition is introduced: The number $pd_v[i]$ for any node v is the position in the $M_v[]$ table where the best solution for $M_v[i]$ was found. Naturally $pd_v[i]$ has to be greater or equal to i, since if it was less then i, a black node could appear at a level where it would violate the rules of K-packing. If $pd_v[i] = i$ then the optimal solution was the intuitive solution, if $pd_v[i] = i+1$ then the optimal solution was the intuitive solution at position $i+1$ and so on. So whenever a history check is completed for position i, $pd_v[i]$ is set to the position where an optimal solution was found. In the example in Figure 2.1, $pd_v[2]$ is 2, $pd_v[1]$ is 1, but $pd_v[0]$ is also 1. Note that the default setting for every position in the pd table is the same as the position. Ie. $pd_v[i] = i$ unless otherwise stated.

2.2.7 More on heirs

Heirs have been barely touched on, and this section will elaborate further. Recall that we used heirs when there was the potential risk that two children's subtrees could have black nodes that where closer than K to one another (Equation 2.2). When the need for an heir has been identified, we use Equation 2.3 to find the sibling-distance. Now we are ready to actually find the heir. Assuming that v is the node for which the algorithm is currently computing the tables, any of its children (the nodes in C_v) could be the optimal choice for an heir, and the algorithm will have to check them one at a time, essentially selecting one child c, using the value at $M_c[i-1]$ and adding the sum of $M_y[sd_v[i]-1]$ for every $y \in C_v/c$. This is then repeated for each $c \in C_v$, and this result is compared to the previous c, seeing which of them is the greater. When completed, the node that was found to give the best solution is stored in $heir_v[i]$, where it can be referenced in Phase 2, and possibly during a history check. Note that while the heirs for two or more different values of i may be the same, this is not guaranteed, and the process of finding an heir must be repeated for every i that satisfies Equation 2.2. We can now formulate the final equation to complete Phase 1. We simply expand Equation 2.1, replacing K with i and taking into account both heirs and history checks. So for any $i = K-1; 1$ we use:

$$M_v[i] = max\left\{M_v[i+1], max_{w \in C_v}\left\{M_w[i-1] + \sum_{c \in C_v/w} M_c[max\{K-1, K-(i-1)-1\}]\right\}\right\} \tag{2.5}$$

2.2.8 Phase 1 in words

As we have seen, the algorithm computes for each node the contents in its table, starting at position K. If the node is a leaf, the first position the table M is set to 1, and every other is set to 0, as described in Section 2.2.1. For any non-leaf node, there are three possibilities: Either $i = K$, in which case Equation 2.1 is used. If $i = K - 1; 1$ we need to include heirs and history checks, and therefor use Equation 2.5. Or $i = 0$, in which case Equation 2.4 is used. These four cases covers every possible circumstance in Phase 1, and provides the root with all the information it needs in order to start Phase 2.

2.3 Phase 2: Setting the final-distance

Now that Phase 1 is finished, the algorithm is ready to start Phase 2. But first we need the following definition: The value $final_v$ (called a *final-distance*) is the position in $M_v[]$ which the algorithm decides that node v will use as the solution for $T[v]$. As stated before, the table $M_v[i]$ is non-decreasing as i decreases, and therefor position $M_r[0]$ (for the root r) holds the optimal solution for the whole tree. Recall from Section 2.2.6 that this did not imply that r had to be black. Instead it meant that $pd_r[0]$ stored where in the table the optimal solution was found. So for the root r, finding $final_r$ is simply setting it to $pd_r[0]$. Now the algorithm knows the minimum number of levels of white nodes has to exist beneath the root, and the algorithm can start setting $final_c$ for every $c \in C_r$. For each of the children it checks if the child is the roots heir (if an heir is needed for the selected value of $final_r$). If the child is the heir the algorithm can use the roots final-distance directly to compute the child's final-distance, and if the child is not the heir it uses the sibling-distance to find the child's final-distance.

In more precise terms, for every $c \in C_r$, the algorithm will have to first check if $heir_r[final_r] == c$. If it is, the child is an heir, and $final_c$ is set to $pd_c[(final_r - 1) mod(K + 1)]$. If it is not an heir, $final_c$ is set to $pd_c[(sd_r[final_r] - 1) mod(K + 1)]$. This is then repeated for each of the roots children, and eventually for every node in the tree, in preorder. The modulo operator is used since we want the final-distance to return to K instead of becoming negative if $final_r = 0$. For each node $v \in V$, if $final_v = 0$, the node becomes black and if not, it becomes white. We see that there are 3 cases to consider when setting the final-distance: either *(i)* the node is the root, *(ii)*the node is its parents heir or the parent has no heirs, or *(iii)* the nodes parent has an heir, but the node is not it. We can now formulate three new equations for these three cases:

$$\textbf{if}(v == r) final_v = pd_v[0] \tag{2.6}$$

$$\textbf{if}(heir_p[final_p] == v \bigvee heir_p[final_p] == null) final_v = pd_v[(final_p - 1) mod(K + 1)] \tag{2.7}$$

$$\textbf{if}(heir_p[final_p] \neq v \bigwedge heir_p[final_p] \neq null) final_v = pd_v[(sd_p[final_p] - 1) mod(K + 1)] \tag{2.8}$$

It might be of interest to note that after Phase 1 has ended, the root knows how many black nodes there are in an optimal solution. So if this is the only goal, the algorithm can be terminated there, forgoing the process of actually finding the black nodes. Also, for this algorithm, there is no optimal choice for the root. Since $final_r$ for the root r can be any value between 0 and K, which node that actually performs the function of a root has no impact on the outcome or running time. The only restriction is that there is only one root, and all the nodes know which of their neighbors is closest to it.

Now that we have a detailed description of Phase 2, an example might be in order. Figure 2.2 shows the same tree as in Figure 2.1, only that Phase 1 has been completed (and a history check has been performed), and the algorithm is ready to begin Phase 2.

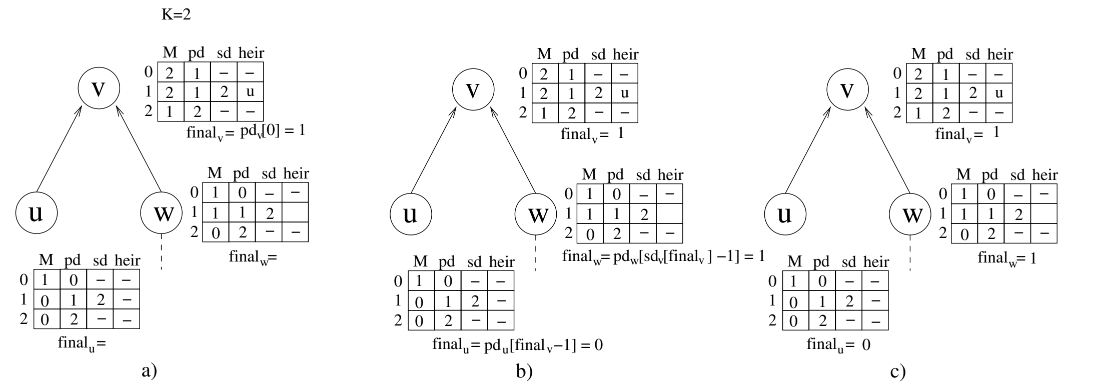


Figure 2.2: K-packing example 2

Figure 2.2a show the root setting its final-distance. As described above, it uses the value found at $pd_v[0]$ (Equation 2.6). Figure 2.2b shows the two children u and w setting their final distance. If we take node u first, the algorithm checks u's parent's *heir* table, and sees that u is the heir for the parents final distance. It can then use $final_v$ directly, and using u's own *pd* table to take into account the result of the history check (Equation 2.7). The value $final_u$ is then found to be 0 (meaning that u becomes black). For the other child, w, the algorithm also begins with checking if it is its parents heir. Since it is not, it cannot use $final_v$ directly, and instead has to use the sd_v table to find the sibling-distance. The algorithm can then use w's own *pd* table to set $final_w$, which becomes 1 (Equation 2.8). Figure 2.2c shows the final outcome of Phase 2 for these three nodes.

2.4 Proving correctness

Now that the algorithm has been described in full detail, the next logical question would be if it really works, ie. will it produce an optimal solution for the maximum K-Packing problem on a tree graph? Since the algorithm has two phases, the proof for the algorithm will also be two-fold. Starting with Phase 1, it can be proven correct with two induction proofs. The first induction proof will prove that the table $M_v[]$ for some node v is correctly computed, and the second one will address the bottom-up part of Phase 1. The proof for Phase 2 follows more or less directly once we have proven Phase 1, as we shall see.

2.4.1 Proof for Phase 1

Before we begin on the actual proofs, we need the following three lemmas, which will be used in the coming two induction proofs:

Lemma 1 The tables $M_v[]$, $pd_v[]$, $sd_v[]$ and $heir_v[]$ for some leaf node v will be correctly computed by the given algorithm.

> **Proof:** In Section 2.2.1 we see that for a leaf node v every position in $M_v[]$ is set to 0, except position 0 which is set to 1. This is obviously correct since the only way the subtree $T[v]$ can have a black node is if v is it. The table $pd_v[]$ at every position i is set to i, since a history check is not needed. The node has no children and therefor $heir_v[]$ can be empty, and without any heirs, $sd_v[]$ will never be used. It follows therefor that every leaf node in the tree will be correctly computed, thus proving the lemma.

Lemma 2 For a node $v \in V$, the sibling-distance for some position i in the table $M_v[]$ is always larger than i if Equation 2.2 is true.

> **Proof:** If we assume $2 \cdot i - 1 = K$ and insert this into Equation 2.3 we get the sibling-distance of $2 \cdot i - 1 - (i - 1) = i$. Now if $2 \cdot i - 1 < K$ we know that there exists some positive x such that $2 \cdot i - 1 + x = K$. Inserting this into Equation 2.3 we get the sibling-distance $(2 \cdot i - 1 + x) - (i - 1) = i + x > i$, thus proving the lemma.

Lemma 3 If, for some node v and some position i, Equation 2.2 is true then the distance between two black nodes in different children of v's subtrees is greater or equal to K.

> **Proof:** Here there are two cases: either *(i)* one of the above mentioned children is the heir, or *(ii)* neither of them are. For the first case the sum of the distance between the two black nodes is at least $(i - 1) + (K - (i - 1) - 1) + 1 = (i - 1) + K - (i - 1) = K$. For case *(ii)* we know from Lemma 2 that if Equation 2.2 is true the sibling-distance is greater than i and therefor the sum of two sibling-distances is greater than K.

Now we are ready to start proving Phase 1. The following two theorems will show that Phase 1 will indeed find an optimal solution for maximum K-packing on a tree.

Theorem 1 The table $M_v[]$ for some node v will be correctly computed provided that the M-table for every $c \in C_v$ has already been correctly computed.

Proof: The proof is by induction over the position i in the M-table. From Lemma 1 we know that a leaf node is correctly computed, so we only need to show that Theorem 1 is true for a non-leaf node

Basis: The base case is if $i = K$. In this case Equation 2.1 is used. It is obviously correct, since it uses the values from the children's M-tables at position $K - 1$. Due to the history check, this will also include position K, and therefor every possible solution is included in the computation.

Induction step: We assume that every $M_v[j]$ where $K \leq j \leq i+1$ has been correctly computed. For every $i < K$, there are three possibilities: Either *(i)* $(K - 1 \leq i \leq \lfloor (K+1)/2 \rfloor + 1)$, in which case the algorithm includes a history check, but there is no need for an heir. Or *(ii)* $(\lfloor (K + 1/2 \rfloor \leq i \leq 1)$, in which case a history check is performed and an heir is needed. And finally *(iii)* $i = 0$ where we need to add one for the node itself, and include a history check.

We see that if i falls within the limits of Case *(i)*, Equation 2.4 becomes identical to Equation 2.1, except that it includes a history check. The algorithm will use the position $i - 1$ in the children's M-tables, but also every other position up to K due to the history check in the children. The history check in the node itself covers the possibility if some position $j > i$ is a better solution.

If i falls within the limits of Case *(ii)* we need to modify the above discussion to include heirs. We know from lemmas 2 and 3 that for any i where Equation 2.2 is true, we can guarantee that no two black descendants from different children of the same node will come into conflict with each others, and that the minimum distance between two such nodes is K. We also see from the discussion of Case *(i)* that a history check will compare the intuitive solution against previous solutions for the node in question, thus making sure that the solution the algorithm finally decides upon is the optimal solution for the i in question.

And finally, Case *(iii)* is just a simple extension of Case *(i)*, going through the exact same argument, only adding 1 to the intuitive solution. That this is correct follows directly from the discussion for Case *(i)*.

Combining these three cases with the basis, we see that every value of i is correctly dealt with and thus Theorem 1 is proven. ■

We see from Theorem 1 that every the table $M_v[]$ for some $v \in V$ will be correctly computed provided that $M_c[]$ for every $c \in C_v$ has already been correctly computed.

Theorem 2 The algorithm as presented will be able to find an optimal solution for maximum K-packing on a tree.

> **Proof:** Proof is by induction over the height of the tree.
>
> **Basis:** The base case is a leaf node, which was shown to be correctly computed in Lemma 1
>
> **Induction step:** This proof follows automatically from Theorem 1, since if every value in the M-tables for every node at the $(h-1)^{th}$ level is correct, every node at the h^{th} will also be able to compute correct M tables. And because of the history check, the table will have the properties stated in Section 2.2. ■

2.4.2 Proof for Phase 2

That Phase 2 is correct is, due to the above discussion, almost self-evident. Since we know that the optimal solution is found in Phase 1, and the tables pd, sd and $heir$ are therefor also correct, we know that $pd_r[0]$ for the root r is the optimal solution. Phase 2 then uses information from the pd, sd and $heir$ to set the final-distance for the roots children, and from there each of the children's children and so on. Thus, since Phase 2 does little more then use information computed in Phase 1, it is obviously correct.

2.5 Implementation and running time

Now that the algorithm has been described in detail, we can look at the more practical issues of implementation and running time. The following two sections will address these issues.

2.5.1 Implementation

The algorithm as described up to this point, while giving an optimal solution, does not have an optimal running time. This section will look at modifications that can be done to improve performance, and to cut down on the amount of code. The final pseudo code for the algorithm is presented in Section 2.7.

First, there is Equation 2.5. If we omit the history check, for each child $c \in C_v$ for a node v, the equation would sum up the value of every $a \in C_v/c$ using the sibling-distance, and then add the value using i for c. This would take time $|C_v - 1|$ for each $c \in C_v$, thus taking time $|C_v| \cdot (|C_v| - 1)$ in total, which is $O(|C_v|^2)$. However this can be done faster. The algorithm can first sum up the values from every child of v, using the sibling-distance, and then for every child, one by one, subtract the value that used the sibling-distance, and add the value that uses the original i. Both steps takes time $|C_v|$, and therefor the total time is $O(|C_v|)$, which is better then $O(|C_v|^2)$. The history check is only a single comparison, and therefor putting it back into the equation has no effect on the running time.

Looking at Equations 2.1 and 2.4, we see that if we omit the history check, they are identical up to the point when Equation 2.4 adds one. So these can be combined into a single sum, and just adding one if $i = 0$.

A history check is performed for all values of i except for $i = K$. So the history check can be removed from each individual equation, and added as a common check at the end for all $i \neq K$.

In the algorithm as described up to this point, there has been one *sd* table for every node in the graph. This, however, is somewhat redundant, since the values in the *sd* table is independent of any specific node. Therefor we need only a single *sd* table for the entire graph. The pseudo code in Section 2.7 will include an *sd* table for each node, for reasons that will become clear in later chapters.

2.5.2 Running time

Now that the algorithm has been modified to improve performance, we can analyze the running time. Each phase will we addressed separately.

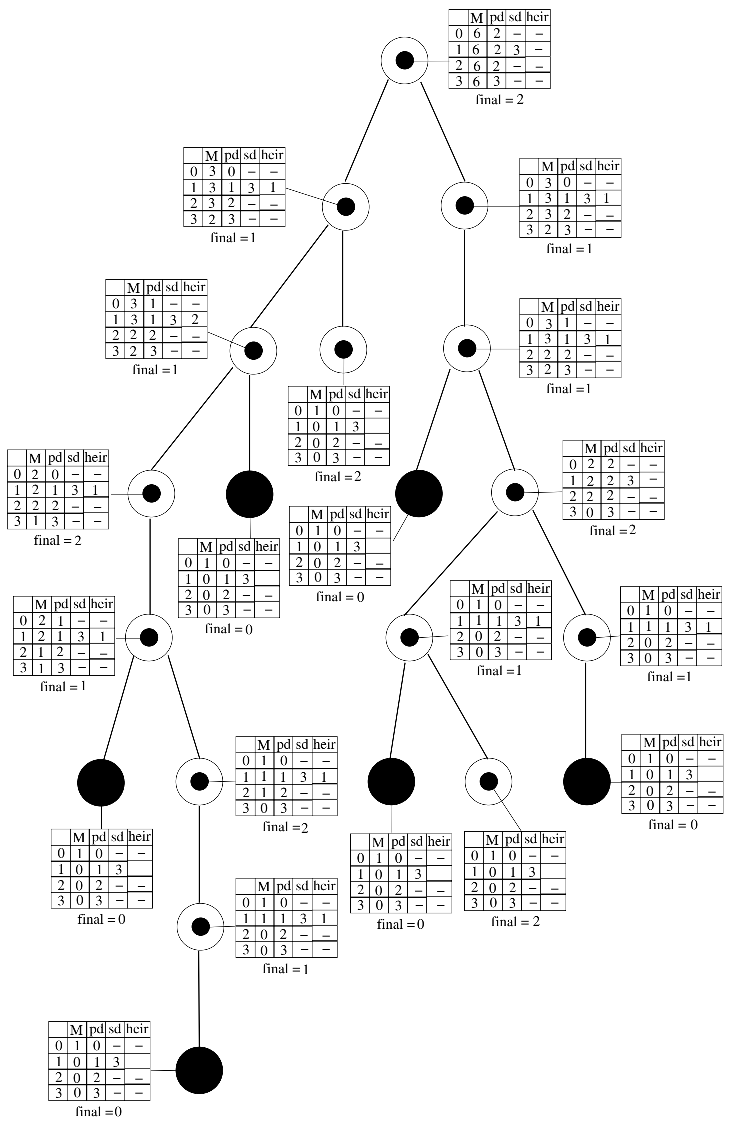
In Phase 1 we have already seem that Equation 2.5 uses time $O(|C_v|)$ for some node v. Both Equations 2.1 and 2.4 can be easily shown to use the same time. We know that for each node the algorithm computes a table of size $K + 1$. From the above discussion it is clear that each position in the table will take $O(|C_v|)$ time. Therefor every node v will use a total time of $O(K \cdot |C_v|)$. So for the entire Phase 1, the algorithm uses time $\sum_{v \in V} K \cdot |C_v|$. Since the total number of children in the graph is $n - 1$ (since every node except the root is someones child), the total running time for Phase 1 is $O(K \cdot n)$.

Phase 2 has a much simpler running time. For each node the algorithm need only perform a fixed amount of computations regardless of the value of K, and therefor takes a total time $O(n)$.

As a conclusion, we see that the total running time of the algorithm is $O(K \cdot n + n) = O(K \cdot n)$. We also saw that the original equations, while giving an optimal solution, where not optimal with regards to running time and amounts of code. The final code can be found in Section 2.7. This pseudo code does not deal with the problem of representing the tree in an efficient way, however [7] describes an effective recursive representation of trees which may be of use in any practical implementation of this algorithm.

2.6 Final example

Now that the algorithm has been described in full detail, a larger example might be in order. Figure 2.3 shows the final outcome of the algorithm, after both Phase 1 and 2 are finished. In order to simplify the figure, none of the nodes have names, and instead, whenever an heir is referred to, the number in the *heir* table tells which child, from left to right is the heir (that is, if the *heir* table says 2, then the second child from the left is the heir). Those nodes that are finally chosen to be a part of S are colored black in the figure. For this example $K = 3$.



	M	pd	sd	heir
0	6	2	–	–
1	6	2	3	–
2	6	2	–	–
3	6	3	–	–

final = 2

	M	pd	sd	heir
0	3	0	–	–
1	3	1	3	1
2	3	2	–	–
3	2	3	–	–

final = 1

	M	pd	sd	heir
0	3	0	–	–
1	3	1	3	1
2	3	2	–	–
3	2	3	–	–

final = 1

	M	pd	sd	heir
0	3	1	–	–
1	3	1	3	2
2	2	2	–	–
3	2	3	–	–

final = 1

	M	pd	sd	heir
0	3	1	–	–
1	3	1	3	1
2	2	2	–	–
3	2	3	–	–

final = 1

	M	pd	sd	heir
0	1	0	–	–
1	0	1	3	
2	0	2	–	–
3	0	3	–	–

final = 2

	M	pd	sd	heir
0	2	2	–	–
1	2	2	3	–
2	2	2	–	–
3	0	3	–	–

final = 2

	M	pd	sd	heir
0	2	0	–	–
1	2	1	3	1
2	2	2	–	–
3	1	3	–	–

final = 2

	M	pd	sd	heir
0	1	0	–	–
1	0	1	3	
2	0	2	–	–
3	0	3	–	–

final = 0

	M	pd	sd	heir
0	1	0	–	–
1	0	1	3	
2	0	2	–	–
3	0	3	–	–

final = 0

	M	pd	sd	heir
0	1	0	–	–
1	1	1	3	1
2	0	2	–	–
3	0	3	–	–

final = 1

	M	pd	sd	heir
0	1	0	–	–
1	1	1	3	1
2	0	2	–	–
3	0	3	–	–

final = 1

	M	pd	sd	heir
0	2	1	–	–
1	2	1	3	1
2	1	2	–	–
3	1	3	–	–

final = 1

	M	pd	sd	heir
0	1	0	–	–
1	1	1	3	1
2	1	2	–	–
3	0	3	–	–

final = 2

	M	pd	sd	heir
0	1	0	–	–
1	0	1	3	
2	0	2	–	–
3	0	3	–	–

final = 0

	M	pd	sd	heir
0	1	0	–	–
1	0	1	3	
2	0	2	–	–
3	0	3	–	–

final = 0

	M	pd	sd	heir
0	1	0	–	–
1	0	1	3	
2	0	2	–	–
3	0	3	–	–

final = 0

	M	pd	sd	heir
0	1	0	–	–
1	0	1	3	
2	0	2	–	–
3	0	3	–	–

final = 2

	M	pd	sd	heir
0	1	0	–	–
1	1	1	3	1
2	0	2	–	–
3	0	3	–	–

final = 1

	M	pd	sd	heir
0	1	0	–	–
1	0	1	3	
2	0	2	–	–
3	0	3	–	–

final = 0

Figure 2.3: K-packing final example

2.7 The algorithm in pseudo-code

```
$K$-**Packing** $(G = (V, E))$
//Starts by calling FillTab() for every node in the graph, from the leaves up.
for $(v \in V, in\ postorder)$
{
    FillTab(v)
}

//The root can now set its final distance, and becomes black if need be.
$final_r = pd_r[0]$

if $(final_r == 0)$
  $color_r = black$

// All the other nodes can now begin setting their final distance, from top to bottom.
for $(v \in \{V - r\}, in\ preorder)$
{
    $p = parent_v$

    //If the current node is its parents heir, or the parent has no heirs.
    if $(Heir_p[final_p] == v \bigvee Heir_p[final_p] == null)$
    {
      $final_v = pd_v[(final_p - 1) mod (K + 1)]$
    }


    //If the parent has an heir, but the current node is not it.
    else
    {
      $final_v = pd_v[(sd_p[final_p] - 1) mod (K + 1)]$
    }

    if $(final_v == 0)$
      $color_v = black$
}
```

```
FillTab($v$)
    For every position in the table for node $v$
    for ($i = K : 0$)
    {
        $p = parent_v$
        $sum = 0$
        This provided that an heir is needed for this position
        if ($2 \cdot i - 1 < K \,\&\, i \neq 0$)
        {
            $sdist_v[i] = K - (i - 1)$
            $bigsum = 0$
            $M_v[i] = 0$
            $heir_v[i] = null$
            $pd_v[i] = i$

            for ($l \in \{N(v) - p\}$)
                $bigsum+ = M_l[(sd_v[i] - 1) mod(K + 1)]$

            for ($m \in \{N(v) - p\}$)
            {
                $sum = bigsum - M_m[(sd_v[i] - 1) mod(K + 1)] + M_m[pd_m[i - 1]]$

                if ($sum > M_v[i]$)
                    $M_v[i] = sum$
                    $heir_v[i] = m$
            }
        }

        For every other position
        else
        {
            for ($n \in N(v) - p$)
                $M_v[i]+ = M_m[pd_m[i - 1]]$

            If the table position implies that the node can be black
            if (i = 0)
                $M_v[i]+ = 1$

            $heir_v[i] = null$
            $pd_v[i] = i$
        }

        Here the history check is preformed
        if ($T_v[i] < T_v[i + 1] \,\&\, i \neq K$)
    {
        $M_v[i] = M_v[i+1]$
        $heir_v[i] = heir_v[i+1]$
        $pd_v[i] = pd_v[i+1]$
        $sd_v[i] = sd_v[i+1]$
    }

}
```

Chapter 3

Minimum K-Domination on trees

3.1 Description of K-Domination

K-Domination is a problem that are in many ways similar to K-Packing. It is defined as follows: A subset S (denoted as black nodes) of the nodes in a graph $G = (V, E)$ is selected so that for every node $v \in V$, there exists at least one node $s \in S$ such that $d(s, v) < K$. In other words, every node in the graph has at most $K - 1$ nodes between it and at least one node in S. Minimum K-Domination is NP-complete on a general graph [11], so the goal of this chapter is to modify the maximum K-Packing algorithm for a tree graph so that it will work for minimum K-Domination on a tree graph.

3.1.1 K-Domination vs K-Packing

There are many similarities between K-Domination and K-Packing. They both require that a subset of the nodes in the graph are selected, and as we shall see, they can be solved in very similar ways on trees.

The two biggest differences between the problems are 1) that one is a maximum problem, while the other is a minimum problem. And 2) in a K-Packing the black nodes are prevented from being within a certain distance from each other, while the black nodes in K-Domination maybe be as close to each other as the algorithm sees fit. On the same note, in K-Packing, the black nodes can be as far away from each others as the algorithm choses, while in K-Domination every node in S has to have a distance of $2 \cdot K$ or less to some other node in S. This means that an empty S is a valid solution to the K-Packing problem, and $S = V$ is a valid solution for the K-Domination problem (of course, neither of them are particular good solutions, but they are valid). With these differences in mind, we can start changing the maximum K-Packing algorithm for trees to work for minimum K-Domination on a tree.

3.2 The basic principle

The algorithm for solving minimum K-Domination on a tree works for the most part the same way as the algorithm for solving maximum K-Packing on a tree: It goes through two phases. In Phase 1, it works its way up from the leaves to the root. After the computations for a node v is finished, the algorithm knows how many black nodes the optimal solution has for the subtree $T[v]$. So when Phase 1 is finished, the algorithm knows how many black nodes the optimal solution has for the entire tree.

Phase 2 starts at the root and works its way down to the leaves, setting the appropriate nodes as black as it goes. When it has completed for all the leaves, the algorithm terminates, and the black nodes in the graph constitutes an optimal solution.

This algorithm makes the same assumptions about the properties of the tree-graph as was stated in Chapter 1.

3.3 Phase 1: Calculating the table

As in the K-Packing algorithm, a table $M_v[]$ is computed for every node $v \in V$. The table is now of size $2 \cdot K + 1$ however. The tables $Heir_v[], pd_v[], sd_v[]$ are also present in this algorithm, with much the same uses. One new definition is used: The *domination field* of a black node s is the subset of the nodes in the graph such that for each node v in that subset $d(s, v) < K$. In other words, the domination field consists of all the nodes that is dominated by s.

The value at $M_v[i]$ now contains the minimum number of black nodes the subtree $T[v]$ can have provided that the closest black node further down the tree is *at most* i nodes away. If $i = 0$ then it is implied that the node is black. If $1 \leq i \leq K$ then the node is dominated from some black node further down the tree, and if $K+1 \leq i \leq 2K$ then the node has to be dominated by a black node further up the tree (or in a siblings subtree, as we shall see later). As in the K-Packing algorithm, the table $M_v[]$ is computed for every node, and the parents table is computed using information from the children's tables and so on until the roots table is completed, at which point Phase 1 is finished. The table is computed starting at $i = 0$ and up to $i = 2 \cdot K$.

3.3.1 Heirs and sibling-distance

In the K-Packing algorithm, heirs were used whenever a given i for a node would lead to a conflict between black nodes in two or more children's subtrees. In this algorithm, heirs are used to avoid redundancy, ie. that two or more black nodes dominate the same node. Recall that K-Domination is a minimization problem, and therefor we want to keep the number of nodes in S as small as possible. Therefor whenever the algorithm discovers that two or more black descendants of a node in different children's subtrees may overlap in their domination fields, it has to take steps to ensure that it does not happen, provided that there is an advantage in avoiding it (more on this later).

First, let us look at when a situation may arise that requires the use of heirs. From Section 3.3 we know that if for some node v $1 \leq i \leq K$ then v is dominated from below, and if $K+1 \leq i \leq 2K$ it is dominated from above (or rather, the algorithm is expecting it to be). Obviously, the only time an heir is needed is if $1 \leq i \leq K$ for the parent node. So we can use the following condition:

$$\mathbf{if}(i \leq K \,\&\, i \neq 0) \tag{3.1}$$

In these cases, the parent computes a sibling-distance, and in the same manner as in the K-Packing algorithm, it checks each of its children to see which one will give the best result if it is allowed to use i, and all the other children uses the sibling-distance. The sibling-distance itself is given by:

$$sd_v[i] = (2 \cdot K + 1) - i \tag{3.2}$$

3.3.2 History Checks

In the maximum K-Packing algorithm we saw that there were situations where it was advantageous that the distance between two black nodes was more then K, despite the intuitive notion that it would be pointless. The same is the case in minimum K-Domination, only here there are cases where two black nodes may both dominate the same nodes, thus giving a better solution. So in order to compensate for this, we use a history check.

To see how this is done, recall that in minimum K-Domination, there is no minimum distance between two black nodes, but there is a maximum distance of $2 \cdot K$. So whenever the value at $M_v[i]$ for some node v is computed, it is implied that at least one black node further down the tree is no more then i nodes away. It cant be more than i without the algorithm running the risk of an illegal solution (ie. not every node is dominated by a black node) when the parent is computed, but it can be less than i. As stated above, having two black nodes closer than $2 \cdot K$ may be advantageous in some situations, and therefor the possibility has to be explored.

So for each value in the $M_v[]$ table that is computed, a comparison is made with one or more other legal solutions, to see if one of them can offer a better solution. Recall that a position i in the table $M_v[]$ implied that at least one black node in $T[v]$ is no more then i nodes away from v. So if $i = 0$ the node in question has to be black, or else an illegal solution might occur. If $i = 1$ then the intuitive solution would be that one or more children of v would be black, but v itself could become black, either instead or in addition to its children. This solution might seem somewhat redundant, but it is perfectly legal. If $i = 2$ then the node may use either the intuitive solution given by

$i = 2$ or it could use the ones computed for $i = 1$ or $i = 0$, both of which would be legal based on the definition given for the table M in Section 3.3.

So for every $i > 0$, the intuitive solution has to be compared to every position $j < i$. In other words, $M_v[i]$ is compared to $M_v[i-1]$, $M_v[i-2]$ and so on. If one of these gives a better solution (that is, a smaller number of black nodes), it is used instead of the intuitive solution. The table $pd_v[]$ is used to keep track of where a solution originated. If $pd_v[i] = i$ then the intuitive solution was used, if $pd_v[i] = j$ then the solution in position j was used. As with the K-Packing algorithm, the intuitive solution in position i need only be compared to position $i - 1$, since position $i - 1$ has already been compared to position $i - 2$ and so on. The default value for the pd table at any position is $pd_v[i] = i$.

3.3.3 Example

Now that a general overview of the algorithm is done, a small example might be in order. Figure 3.1 shows the same graph as in Figure 1 and 2. Also here $K = 2$, and the table M has already been computed for the two nodes u and w. Thus only node v remains. Figure 3.1a) shows the graph before the calculations for node v begins. Note that the history check is not performed during the computations of the M table. Instead the final figure will show the table after the history check.

Figure 3.1b) shows the calculation for $M_v[0]$. We see that it uses the sum of children's M-tables at position 4 and adds one (since it will become black if $final_v = 0$). This gives a sum of 1. Figure 3.1c) shows the computations for $M_v[1]$. Here a history check is performed, and we see that if w is chosen as the heir the value at position 1 becomes 2, while if u is chosen as an heir, the result becomes 1. Obviously the latter is the better choice. Position 2 is computed in a similar way as position 1 and is not shown. Figure 3.1d) shows the computations for $M_v[3]$ and $M_v[4]$. In both cases no heir is needed, and the algorithm simply uses the values at the children's M values at position 2 and 3, respectively. Now we see from Figure 3.1e) that after the history check is performed, the intuitive solutions in positions 2 and 3 in $M_v[]$ was found to not be the optimal solution. Instead the optimal solution was found at position 1 for both cases. Thus we are left with the M table for node v where only a single black node was the optimal solution for every position.

3.3.4 The four cases

Now we see that there are four different cases to be considered in Phase 1 for a node v: Either v is a leaf node, or it is not a leaf node and $i = 0$, $1 \leq i \leq K$ or $K + 1 \leq i \leq 2K$. Each of these four cases will be dealt with in the following sections.

3.3.5 Leaf nodes

In the K-Packing algorithm the leaf nodes where simple cases. Here they become a little more complicated. $M_v[0]$ is still set to one, and for every $K + 1 \leq i \leq 2 \cdot K$, $M_v[i]$ is

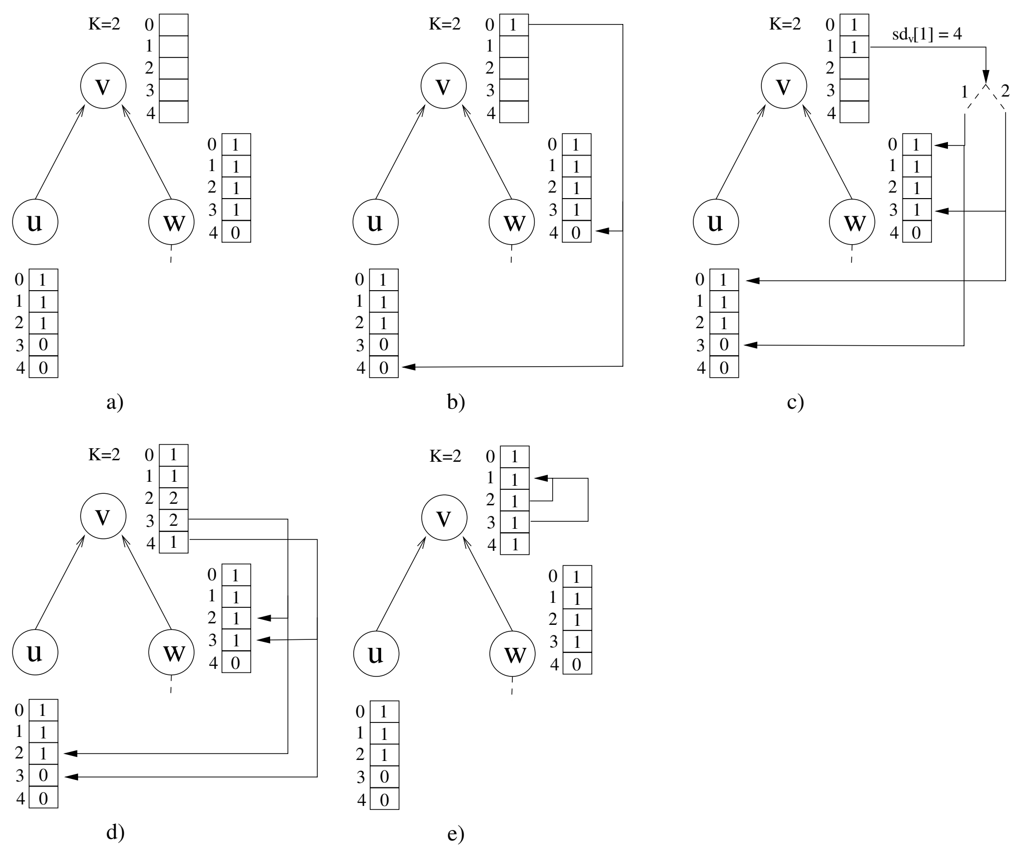


Figure 3.1: K-domination example

set to 0. However, recall from Section 3.3 that i implied that there is a black node at most i nodes further down the tree. But this is obviously not the case, since there are no nodes further down from a leaf. Any i such that $K+1 \leq i \leq 2 \cdot K$ will not cause a problem, since it is implied that the leaf node will then be dominated from above. But if $1 \leq i \leq K$ it is implied that the node is dominated from below, which is obviously not going to happen. So for those i's, $M_v[i]$ has to be set as infinite. However, the history check will come into play here, since any position $1 \leq i \leq K$ will be compared to $i-1$ which has been compared to $i-2$ and so on, and therefor each of them will use the solution for $i=0$. So the final M table for every leaf node will be $M_v[i]=1$ for every $0 \leq i \leq K$ and 0 for every other i.

3.3.6 Non-leaf nodes: $i = 0$

This is the simplest case to compute, since it only needs to use the children's values at position $2 \cdot K$. No history check or heir is needed:

$$M_v[0] = 1 + \sum_{c \in C_v} M_c[2K]\} \tag{3.3}$$

3.3.7 Non-leaf nodes: $1 \leq i \leq K$

In these cases, the node is dominated from below, and an heir may be needed since Equation 3.1 is true. A history check is required.

$$M_v[i] = min\{M_v[i-1], min_{c \in C_v}\{M_c[i-1] + \sum_{a \in C_v/c} M_a[2K-i]\}\} \tag{3.4}$$

3.3.8 Non-leaf nodes: $K + 1 \leq i \leq 2K$

Here, heirs are not needed, but a history check is still required.

$$M_v[i] = min\{M_v[i-1], \sum_{c \in C_v} M_c[i-1]\} \tag{3.5}$$

3.3.9 Phase 1 in words

As we have seen, Phase 1 goes trough all the nodes $v \in V$, bottom to top, and computes all the values in the $M_v[]$ table, starting at position 0 and going up to $2K$. A history check is performed when necessary, and heirs are used whenever Equation 3.1 is true. Section 3.3.5 describes the table for leaf nodes, and Equations 3.3 to 3.5 describes the three different cases for non-leaf nodes. As with the maximum K-Packing algorithm, after Phase 1 is finished, the root knows how many black nodes there are in an optimal solution, and if this is the only goal, the algorithm can terminate here.

3.4 Phase 2: Setting the final-distance

The purpose of Phase 2 is to find all the nodes that are black in an optimal solution. This is done by starting at the root, and after setting its color, goes through all the nodes, top to bottom, and setting the appropriate color for each node.

As in the K-Packing algorithm, the table $pd_v[i]$ for a node v and position i has the position where the solution in $M_v[i]$ first originated. If the intuitive solution came out on top after the history check, $pd_v[i]$ is set to i. But if the history check finds a better solution at position $i-1$, $pd_v[i]$ is set to $pd_v[i-1]$. Also, as in the K-Packing algorithm, the value $final_v$ for some node v is set during Phase 2 to identify the position where the optimal solution was found. In addition, the *heir* table is used to check whether or not a node is its parent's heir. If it is the heir, the algorithm can use the nodes parent final-distance directly, and if it is not, the algorithm uses the parent's *sd* table to get the sibling-distance.

As seen in Section 3.3.5, any value of i such that $1 \leq i \leq K$ constitutes an invalid solution for a leaf, since it implies that the node is dominated from below, which is an impossibility since there are no nodes below a leaf. The root works in much the same way: any i such that $K+1 \leq i \leq 2K$ implies that the node will be dominated from above, which is of course not possible since there are no nodes above the root. So just as some values for i where invalid for a leaf, any value of i such that $K+1 \leq i \leq 2K$ is invalid for the root. This does not mean that those positions in the M table are not to be computed, but they can not be used as a solution. In other words, only $0 \leq i \leq K$ can be a legal final-distance for the root. Since the history check guaranties that the values in every M table is non-increasing as i increases, the solution in $M_r[K]$ is always the optimal one for the root r.

As in the K-Packing algorithm, Phase 2 has to traverse every node in the graph, top to bottom, starting with the root. We saw above how the final-distance for the root is computed. For all the other nodes, the algorithm uses information from the nodes parent to compute the final-distance. This is done in the following way: if the node in question is its parents heir, the algorithm can use the parents final-distance directly, and using the nodes own *pd* table to take into account the result of the history check. If the node in question is not its parents heir, the algorithm has to use the parents *sd* table to find the sibling-distance for the parents final-distance, and then use the nodes own *pd* table. We see that there are 3 cases to consider when setting the final-distance: either *(i)* the node is the root, *(ii)* the node is its parents heir or the parent has no heirs, or *(iii)* the nodes parent has an heir, but the node is not it. We can now formulate three new equations for these three cases:

$$\textbf{if}(v == r)\ final_v = pd_v[K] \tag{3.6}$$

$$\textbf{if}(heir_p[final_p] == v \bigvee heir_p[final_p] == null)\ final_v = pd_v[(final_p - 1) mod (2K+1)] \tag{3.7}$$

$$\textbf{if}(heir_p[final_p] \neq v \bigwedge heir_p[final_p] \neq null)\ final_v = pd_v[(sd_r[final_p] - 1) mod (2K+1)] \tag{3.8}$$

Just as in the K-Packing algorithm, which node that acts as the root of the tree has no bearing on the outcome or running time of the algorithm.

3.5 Proving correctness

Now that we have a full understanding of the algorithm, a proof that it does indeed solve minimum K-Domination on a tree would be in order. This proof will for the most part follow the same lines as the proof for the K-Packing algorithm. Beginning with

Phase 1, we need two induction proofs to show that it is correct. The proof for Phase 2 follows more or less automatically once Phase 1 is proven.

3.5.1 Proof for Phase 1

Before we begin on the actual proofs, we need the following three lemmas, which will be used in the coming two induction proofs:

Lemma 1 The tables $M_v[]$, $pd_v[]$, $sd_v[]$ and $heir_v[]$ for some leaf node v will be correctly computed by the given algorithm.

> **Proof:** In Section 3.3.5 we see that for a leaf node v, positions i such that $0 \leq i \leq K$ is set to 1 and every i such that $K+1 \leq i \leq 2 \cdot K$ is set to 0. This is obviously correct, since the only way the subtree $T[v]$ can have a black node is if v is it. Positions $1 \leq i \leq K$ is set to 1 due to the history check, since any such i implied that v is dominated from below, which is of course not possible. If $K+1 \leq i \leq 2 \cdot K$ then v will be dominated from above, and no black node is needed in the subtree $T[v]$. There are no children present, so $heir_v[]$ can be empty, and therefor $sd_v[]$ will never be accessed. The table $pd_v[]$ on the other hand has to be set to 0 for every $1 \leq i \leq K$, and the default value for every other i.

Lemma 2 For a node $v \in V$, the sibling-distance for some position i in the table $M_v[]$ is always larger than i if Equation 3.1 is true.

> **Proof:** If we assume that $i = K$ and insert this into Equation 3.2 we get the sibling-distance of $2 \cdot i + 1 - i = i + 1 > i$. If $i < K$ we know that there exists some positive x such that $i + x = K$. Again inserting this into Equation 3.2 we get the sibling-distance $2 \cdot (i+x)+1-i = 2 \cdot i + 2 \cdot x + 1 - i = i + 2 \cdot x + 1 > i$, thus proving the lemma.

Lemma 3 If, for some node v and some position i, Equation 3.1 is true then the distance from the intuitive solution between two black nodes in different children's subtrees is greater or equal to $2 \cdot K$.

> **Proof:** Here there are two cases: either *(i)* one of the above mentioned children is the heir, or *(ii)* neither of them are. For the first case the sum of the distance between the two nodes is $2 \cdot K + 1 - i - 1 + i = 2 \cdot K$. For case *(ii)* we know from Lemma 2 that if Equation 3.1 is true the sibling-distance is greater than i and therefor the sum of two sibling-distances is greater than $2 \cdot K$.

Now we are ready to start proving Phase 1. The following two theorems will show that Phase 1 will indeed find an optimal solution for the problem minimum K-domination on a tree.

Theorem 1 The table $M_v[]$ for some node v will be correctly computed provided that the M-table for every $c \in C_v$ is correctly computed.

Proof: The proof is by induction over the position i in the M-table. From Lemma 1 we know that a leaf node is correctly computed, so we only need to show that Theorem 1 is true for a non-leaf node

Basis: The base case is if $i = 0$. In this case Equation 3.3 is used. It is obviously correct, since it uses the values from the children's M-tables at position $2 \cdot K$ and adds one for the node itself. No history check or heir is needed.

Induction step: We assume that every $M_v[j]$ where $j < i$ has been correctly computed. For every $i > 0$, there are two possibilities: Either *(i)* $1 \leq i \leq K$, in which case the algorithm includes a history check, and an heir is needed. Or *(ii)* $K + 1 \leq i \leq 2 \cdot K$, in which case a history check is performed but an heir is not needed.

We see that if i falls within the limits of Case *(i)*, Equation 3.4 is used. It will compute the intuitive solution using Equation 3.2 as sibling-distance. From Lemma 3 we know that this will always give a distance between two black nodes in different siblings subtrees greater or equal to $2 \cdot K$. When we add the history check we know that the possibility that some $j < i$ will be a better solution is be covered.

If i falls within the limits of Case *(ii)* Equation 3.5 is used. Compared to Equation 3.4 it is identical with the exception of the need for an heir. Thus we can remove the heir computations from the equation, only leaving the base summation from the children plus the history. That this is correct follows from the above discussion.

Combining these two cases with the basis, we see that every value of i is correctly dealt with and thus Theorem 1 is proven. ■

We see from Theorem 1 that every the table $M_v[]$ for some $v \in V$ will be correctly computed provided that $M_c[]$ for every $c \in C_v$ has already been correctly computed.

Theorem 2 The algorithm as presented will be able to find an optimal solution for maximum K-packing on a tree.

Proof: The proof is by induction over the height of the tree.

Basis: The base case is a leaf node, which was shown to be correctly computed in Lemma 1

Induction step: This proof follows automatically from Theorem 1, since if every value in the M-tables for every node at the $(h-1)^{th}$ level is correct, every node at the h^{th} will also be able to compute correct M tables. And because of the history check, the table will have the properties stated in Section 3.3. ■

3.5.2 Proof for Phase 2

That Phase 2 is correct is, due to the above discussion, almost self-evident. Since we know that the optimal solution is found in Phase 1, and the tables pd, sd and $heir$ are therefor also correct, then we know that $pd_r[K]$ for the root r is the optimal solution. Phase 2 then uses information from the pd, sd and $heir$ to set the final-distance for the roots children, and from there each of the children's children and so on. Since Phase 2 does little more then use information computed in Phase 1, it is obviously correct.

3.6 Implementation and running time

Now that we have a complete understanding of the modifications required to remake the algorithm from the previous chapter into a maximum K-Domination algorithm for tree graphs it would be a good time to consider ways to reduce the amount of code, and improve running time.

3.6.1 Implementation

As with the K-Packing algorithm, there are a few ways the algorithm can be modified to both reduce the running time and the amount of code. The final code is in Section 3.7.

In the algorithm as described, there has been one sd table for every node in the graph. This, however, is somewhat redundant, since the values in the sd table is independent of any specific node. Therefor we need only a single sd table for the entire graph. The pseudo code in Section 3.7 will include an sd table for each node, for reasons that will become clear in later chapters.

In the K-Packing algorithm we were able to reduce the running time of the part of the algorithm that dealt with finding heirs. The same improvements can be made here, since that part of the algorithm is virtually identical in both the K-Packing algorithm and the K-Domination algorithm. In short, instead of selecting one child as an heir, and adding the sum of all the of all the others using the sibling-distance, the algorithm can add all the children together using the sibling-distance, and one by one subtract the value that used the sibling-distance, and add the value that uses i. This will reduce the running time from $\mathrm{O}(|C_v|^2)$ to $\mathrm{O}(|C_v|)$.

The equations 3.3, 3.4 and 3.5 are in many ways similar. Therefor we can divide Phase 1 first up into those value of i that needs an heir and those who do not. After that add one if $i = 0$, and perform a history if not.

3.6.2 Running time

From the above section it follows almost automatically that the running time for Phase 1 is $\mathrm{O}(K \cdot n)$. As in the K-Packing algorithm each position i in $M_v[i]$ for node v takes time $|C_v|$, and since there are $2 \cdot K + 1$ possible values for i, this makes the running time for a single node v $\mathrm{O}(|C_v| \cdot (2K + 1)) = \mathrm{O}(|C_v| \cdot K)$. The total number of children in the entire tree is $n - 1$ and therefor the total running time for Phase 1 is $\mathrm{O}(K \cdot n)$.

The running time for Phase 2 is much simpler. For each node in the graph only a fixed amount of operations are performed, and the running time is therefor $\mathrm{O}(n)$. So the combined running time of the algorithm is $\mathrm{O}(K \cdot n)$, just as with the K-Packing algorithm.

3.7 The algorithm in pseudo-code

```
$K$-Domination $(G = (V, E))$
//Starts by calling FillTab() for every node in the graph, from the leaves up.
for $(v \in V, in\ postorder)$
{
    FillTab(v)
}

//The root can now set its final distance, and becomes black if need be.
$final_r = pd_r[K]$

if $(final_r == 0)$
   $color_r = black$

// All the other nodes can now begin setting their final distance, from top to bottom.
for $(v \in \{V - r\}, in\ preorder)$
{
    $p = parent_v$

    //If the current node is its parents heir, or the parent has no heirs.
    if $(Heir_p[final_p] == v \bigvee Heir_p[final_p] == null)$
    {
       $final_v = pd_v[(final_p - 1) mod (2K + 1)]$
    }

    //If the parent has an heir, but the current node is not it.
    else
    {
       $final_v = pd_v[(sd_p[final_p] - 1) mod (2K + 1)]$
    }

    if $(final_v == 0)$
       $color_v = black$
}
```

```
**FillTab**($v$)
**for** $(i = 0 : 2K)$
{
    $M_v[i] = 0$
    $heir_v[i] = null$
    $pd_v[i] = i$

    //If an heir is needed
    **if** $(i \leq K \,\&\, i \neq 0)$
    {
        $sd_v[i] = 2 \cdot K + 1 - i$
        $bigsum = 0$

        **for** $(l \in C_v)$
            $bigsum+ = M_l[(sd_v[i] - 1) mod (2k + 1)]$

        **for** $(m \in C_v)$
        {
            $sum = bigsum - M_m[(sd_v[i] - 1) mod (2k + 1)] + M_m[i]$

            **if** $(sum < M_v[i])$
                $M_v[i] = sum$
                $heir_v[i] = m$
        }
    }

    //If an heir is not needed
    **else**
        **for** $(c \in C_v)$
            $M_v[i]+ = M_c[i - 1]$

    // If $v$ is a leaf and not the root
    **if** $(C_v == 0 \,\&\, v \neq r)$
        **for** $(j = 1 : K)$
            $M_v[j] = 1)$

    //Here the history check is performed ...
    **if** $(i \geq 1)$
    {
        **if** $(M_v[i - 1] < M_v[i])$
            $M_v[i] = M_v[i - 1]$
            $pd_v[i] = pd_v[i - 1]$
    }
    // ... and if a history check is not required, $i = 0$ and one is added to the result.
    **else**
        $M_v[0] += 1$
}
```

Chapter 4

Self-stabilizing Algorithms

4.1 General Introduction

A self-stabilizing algorithm is a concept first introduced by Dijkstra in 1974 [10], but serious work did not begin until the late 1980s. The basic idea is that each node in a graph is an independent unit, and it has no knowledge about the graph, save its neighbors and it self. This means that no single node in the graph is guaranteed to know what the entire graph looks like. Each node has a copy of the algorithm, and using only information from their neighbors and it self, each node will reach a state that is guaranteed to comply with one or more set limitations. An example could be Independent Set, in which case each node will decide to become either black or white, based solely on information from its neighbors.

A self-stabilizing algorithm is comprised of a set of rules, where each rule has a predicate $p(v)$ (v is a node) and a move M. A rule would commonly look like this:

```
Ri:
    if:   $p(v)$
    then:   $M$
```

The predicate becomes true if one or more conditions are met, and the move can then be executed. If the predicate of the rule i is true, the rule is called privileged. A node may at any time have several privileged rules. There are many models that govern which of the privileged rules in the graph is executed. Two common are as follows: A central daemon is used, and it selects a privileged rule, and that rule will be allowed to make its move, while all the other privileged rules wait for the first rule to finish. Then the daemon selects another rule, and so on. The daemon may select rules to execute either based on priority, or simply randomly. Another common model makes no assumptions about a central daemon, and two or more rules, in the same or different nodes, may make their moves at the same time, provided none of them are trying to access or write to a value that another rules is already writing to. Regardless of the daemon (or lack thereof), the algorithm becomes stable when there are no privileged nodes. The graph should now comply with the specifics of the problem in question (such as Independent Set, in which case no black nodes are neighbors). A self-stabilizing algorithm has to be able to reach a stable condition regardless of the beginning state of the graph, regardless of any changes that occur in the graph during the execution (however, if the graph does indeed change, the algorithm will possibly have to start over again), and it has to do so in a finite number of moves. This makes a self-stabilizing algorithm very fault tolerant, but it also means that even problems with fairly simple sequential solutions may require clever and often complex self-stabilizing algorithms (for more information on self-stabilizing algorithms, see [23]).

Running time for self-stabilizing algorithms is often handled somewhat differently then for sequential algorithms. In the latter case running time refers to the maximum number of operations that is required for the algorithm to provide a solution, regardless of the input. In self-stabilizing algorithms the running time refers to the maximum number of rules that has to fire before the algorithm reaches a stable state, regardless of the initial state of the graph. This means that any change in the graph, regardless of what changed and what caused it, could lead to the maximum number of rules firing again.

4.1.1 Example of Self-stabilizing algorithms

To give a better understanding of self-stabilizing algorithms, an example might be in order. The following algorithm, due to Hedentniemi et. al., solves the maximal independent set on a general graph. The complete algorithm with proof can be found in [14] and only an overview will be given here.

The algorithm consists of two rules, $R1$ and $R2$, which are as follows:

```
R1:
    if:   $s(i) = 0 \bigwedge (\forall j \in N(i))\ s(j) = 0$
    then:   $set\ s(i) = 1$

R2:
    if:   $s(i) = 1 \bigwedge (\exists j \in N(i))\ s(j) = 1$
    then:   $set\ s(i) = 0$
```

We see that the rules uses a variable $s(i)$ for every $i \in V$. This refers to the color of the node. If $s(i) = 0$ then node i is white and if $s(i) = 1$ the node is black. We can formulate both rules in words. $R1$ reads: *If i is white, and every neighbor is also white, then i becomes black.* $R2$ reads: *If i is black, and there exists at least one neighbor that is also black, then i becomes white.* This means that whenever a white node notices that it has no black neighbors, it becomes black, and whenever a black node notices that it has a black neighbor, it becomes white.

To further illustrate the algorithm, the following example shows a possible execution of the algorithm. Note that this algorithm employs a random daemon, that is, only one node executes at a time, and that node is chosen randomly. Figure 4.1 shows the example:

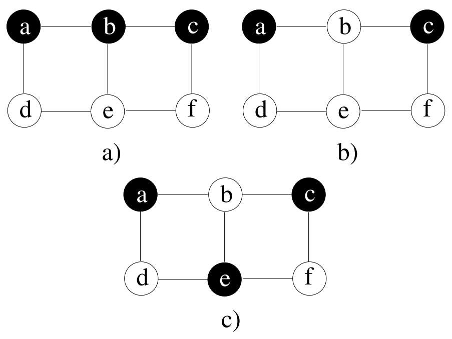


Figure 4.1: Maximal Independent Set example

As we see from the figure, the starting position has nodes a, b and c colored black, while d, e and f are white. Thus rule $R2$ is privileged in the three first nodes. Since the daemon is random, any one of them can be chosen to make a move, but lets assume b makes it, and becomes white. Now we see that since e no longer has any black neighbors, $R1$ becomes privileged for it. And since it is neither a or c have any black neighbors left, they are no longer privileged. Thus e makes its move, and becomes black, as seen in Figure 4.1c). Now we see that every white node has at least one black neighbor, and no black nodes have any black neighbors. Thus no rules are privileged, and the algorithm is stable.

Note that if the choice of which node where to make a move in Figure 4.1a) had been different, the the final solution could have been 2 black nodes, not 3. A possible order in which that could happen is if first nodes a and b becomes white, and then node d becomes black. Now every node in the graph is either black and has no black neighbors, or it is white, and has one black neighbor. In other words the algorithm is stable, but there are only 2 black nodes in the graph. This illustrates an important concept about self-stabilizing algorithms, since we in most cases have little control over the order that rules are allowed to execute, and therefor the creator of a self-stabilizing algorithm has to take every possible scenario into consideration

4.1.2 Multistage Self-stabilizing algorithms

It is not uncommon for two or more self-stabilizing algorithms to function in a graph at the same time. If these algorithms do not interfere with one another (that is, they do not read or write to the same values) the total number of moves will be no more then the sum of each individual algorithms number of moves. However, things are not always that simple. Imagine what would happen if two algorithms (Algorithm A and Algorithm B with running time $O(f(A))$ and $O(f(B))$, respectively) function in the same graph at the same time and one of them writes to a value that another one reads from. This would mean that whenever Algorithm A writes to said value, Algorithm B could end up having to recompute everything, potentially using $O(f(B))$ moves. Since Algorithm A can in theory make some change to the graph $O(f(B))$ times, and each time it does Algorithm could use $O(f(B))$ moves, the total running time for both algorithms is $O(f(A)) \cdot O(f(B))$.

4.1.3 Multistage Self-stabilizing algorithms on trees

Intuitively there is no reason to believe that tree graphs should be any different than a general graph when the above situation occurs. However Blair and Manne was able to show in [2] that, for a tree, if one combines l self-stabilizing algorithms that comply to set limitations (more on this later) the running time would not be the product of the running time for each of the algorithms, but instead $O(n^{l+1})$. However, for reasons that will become clear, this chapter will only repeat what is needed to combine two such algorithms.

Recall that there are two algorithms A and B, and Algorithm B uses information that Algorithm A may be constantly changing, thus forcing Algorithm B to recompute its own values. In order for these to algorithm to fit into the context of [2], Algorithm A has to take this form:

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

In other words, for every node i there is a value $f_i(j)$ for every $j \in N(i)$ that take input from every neighbor of i except j. It was proven in [2] that any algorithm that takes this form will run in time $O(n^2)$. The following figure shows how the values in one node is computed using values in its neighbors.

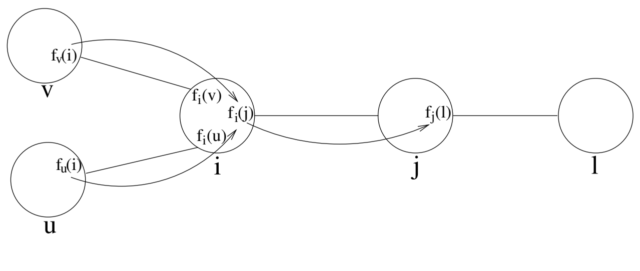


Figure 4.2: Example for $G1$

We see that the value $f_i(j)$ in the node i is computed using the values $f_v(i)$ and $f_u(i)$. The value $f_j(l)$ is in turn computed using $f_i(j)$ and so on.

Algorithm B on the other hand uses information not only from itself, but also from Algorithm A. It has to take this shape:

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

From these two rules we see that the value f belongs to Algorithm A and value q belongs to Algorithm B. We also see that the rule G1 for some node i uses only f in its computations, while rule H1 uses both f and q. It does not however alter f, so if G1 at some node fires, H1 can become privileged in one of the neighboring nodes, but not vice versa. It is shown in [2] that H1 will also run in $O(n^2)$ time, and G1 and H1 combined will run in $O(n^3)$ time.

4.1.4 Leader election

A common problem in many algorithms for trees, self-stabilizing or otherwise, is the process of finding and/or creating a root. There are numerous algorithms that requires the existence of some node that it can call a root, and other algorithms that will run better or faster if the tree is balanced (ie. the root has an approximate equal number of nodes in each of its children's subtrees). The most readily available example of this is a quick-sort algorithm (for example the one given in [15]), where a balanced tree will give a running time of $O(n \cdot log(n))$, where as an unbalanced tree could potentially give a running time of $O(n^2)$. In the two algorithms for K-Packing and K-Domination from the two previous chapters a root was required, but it could be simply chosen arbitrarily, since which node that functioned as the root would have no bearing on the output or running time of the algorithms. Things are not quite that simple in the self-stabilizing world. Since the graph itself is dynamic, and since no single node can be sure whether or not it has knowledge of the entire graph, a root can not simply be chosen at random. Instead some sort of self-stabilizing algorithm will have to be used to determine if a node is the root, and if not, which of its neighbors are closest to the root (ie. which neighbor is the nodes parent).

A self-stabilizing algorithm that does just that is given by Blair and Manne in [3]. The full details of this algorithm will not be presented here, just a brief overview: In short, the algorithm work by finding a $n/2$ separator (that is, a node that can be removed such that none of the remaining connected component has more then $n/2$ nodes in it). This node is then selected as the root, since it can be shown that there exists at least one and at most two such nodes a tree (in the case of a tie, the node with the node with the the largest "name" is chosen as the root). In order to find the $n/2$ separator, each node i has values called $size_i(j)$ for every $j \in N(i)$. When the algorithm has stabilized,

$size_i(j)$ contains the number of nodes in the connected component $G - j$ containing i. If $size_j(i) < n/2$ for every $j \in N(i)$ of a node i then the node can be set as a root (in which case it becomes its own parent), and if there is some $j \in N(i)$ where $size_j(i) > n/2$ then j becomes the parent of i. It can be easily shown that this algorithm fits perfectly into the frame set for G1 in [2].

4.1.5 The two phases

Observant readers may have noticed by now that the two phases in the K-Packing and K-Domination algorithms from the previous chapters looks in many ways similar to two self-stabilizing algorithms, one that computes Phase 1 and the other computes Phase 2. This would certainly fit within the frame presented in [2], but if we combine these two algorithms with the root finding algorithm in [3] we get a total of 3 algorithms running simultaneously, and since Phase 1 uses values from the root finding algorithm and Phase 2 uses values from Phase 1, the total running time would be $O(n^4)$. However, this can be improved upon.

The way the two algorithms from Chapters 2 and 3 are presented, it is clear that both phases rely heavily on knowledge of which of a nodes neighbors is its parent. However, Phase 1 in both algorithms can be modified to work around the need for a root: For any single node there is only a limited number of candidates for the role as parent. One would think that that there are $|N(i)|$ possible parents for a node i. However, since a node is its own parent if it is the root of the tree (from the root finding algorithm in [3]), there are in fact $|N(i)| + 1 = |N[i]|$ possible parents. So, if instead of computing a single set of the tables $M, heir, pd$ and sd for every node, the algorithm can compute one set of these tables for each possible parent of the node in question. In this way Phase 1 in both the K-Packing and K-Domination algorithms can be modified to work around the need for a known parent.

Figure 4.3 shows a small example to further illustrate this point. The figure shows 3 nodes, v, u and w. The example shows the values used in computing the values in the M-table for node v.

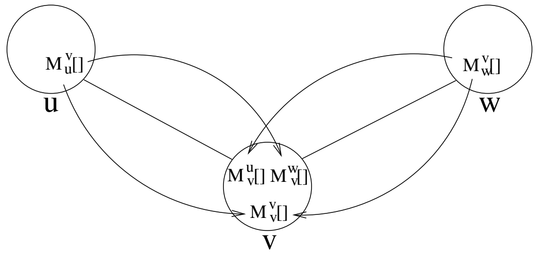


Figure 4.3: Example for Section 4.1.5

From the above discussion we see that there are three possible parents for the node, either u, w or v itself (ie. v is the root of the tree). For the M-table where it is assumed that u is the parent ($M_v^u[]$), $M_w^v[]$ is used in the computations, and for the table where it is assumed that w is the parent ($M_v^w[]$), $M_u^v[]$ is used. If the node v is to be the root of the tree, it will have to be its own parent, and for that possibility the table $M_v^v[]$ is used. In these computations the algorithm uses M-tables from both its children.

Phase 2 is another matter entirely. The need for a known parent in every node in the tree is too heavily woven into the algorithms to be easily removed. But there is no need to remove it anyway, since we already have an algorithm to find the root of the tree (the algorithm in [3]). Thus, Phase 2 in both algorithms can operate as a self-stabilizing algorithm that uses values from the root finding algorithm and the Phase 1 algorithm.

It follows from the above discussion that a self-stabilizing algorithm for Phase 1 can run in parallel with the root finding algorithm, and a self-stabilizing algorithm for Phase 2 can make use of both values from the root finding algorithm and values the Phase 1 algorithm. If we can show that both phases in the K-packing and K-domination can be turned into self-stabilizing algorithms that fit into the frame described in [2] we can show that not only will each phase run in $\mathrm{O}(n^2)$ time, but combined with the root finding algorithm an optimal solution for either problem can be found in $\mathrm{O}(n^3)$ time. The next two sections will show just that.

4.2 Self-stabilizing K-Packing

This section will show how the maximum K-packing algorithm for tree graphs from Chapter 2 can be used as a basis for creating a self-stabilizing algorithm that solves the same problem. The resulting algorithm will fit into the framework from [2], which will make proving correctness and running time much easier. For the duration of this section, it is assumed that the graph the algorithm is running on is a tree graph, where every node may or may not know which of its neighbors is its parent, and there may at any one time be none, a single or several roots. The root finding algorithm from [3] is used to solve the problem of finding both a nodes parent, and as long as it is stable, every node in the graph knows which of the nodes in the closed neighborhood is its parent. As with the sequential algorithm from Chapter 2, this algorithm will also have two phases, and each of which can be described as a separate rule.

4.2.1 Phase 1

Recall from Chapter 2 that the purpose of Phase 1 is to compute the tables $M_v[], pd_v[], sd_v[]$ and $heir_v[]$ for every node $v \in V$. This is still the case here, however there are two problems inherit to self-stabilizing algorithms that has to be overcome first:

Recall from Section 4.1.5 that a node may not at any one time know which of its neighbors is its parent, and even if it does, the parent could be changed to a different node due to either the node in question or one of its neighbors making a move at some point. As noted in Section 4.1.5, this meant that either Phase 1 would have to hold of making any computations for a node until the node knew who its parent was, or it had to work around the need for this knowledge. As discussed in Section 4.1.5, the latter was the best option. Therefor, every node has to have one set of the before mentioned tables for each possible parent. This includes the node itself, since it is its own parent if it is the root of the tree. In other words: for every $v \in V$ there exists a set $M_v^c[], sd_v^c[], pd_v^c[]$ and $heir_v^c[]$ for every $c \in N[v]$.

The other problem inherit to self-stabilizing algorithms that will interfere with the algorithm as described in Chapter 2 is the fact that we have no control over the order in which the various nodes in the graph executes their rules. Assuming that the root finding algorithm is stable (every node knows their parent, and there is only a single root), this means that after a node v has completed its commutations, one of its children can make a move that forces v to redo some of the before mentioned calculations. If another child makes a move after that v has to once again start over and so on. This means that the eventual rule that will make up Phase 1 will for some node v have to not only run once, but potentially as many times as there are nodes in the subtree $T[v]$. This means that while the running time for the sequential algorithm was linear for Phase 1, that can not be the case here.

We can now formulate a rule for Phase 1 that will fit into the framework set in [2]:

R1:
if: $\exists p \in N[v] : (M_v^p[], heir_v^p[], pd_v^p[] \text{ or } sd_v^p[]) \neq FillTab(v,p)$
then: $FillTab(v,p)$

This rule makes use of the function *FillTab(v,p)*, where v is the node in question and p one of the candidates for the role as parent. The function $FillTab(v,p)$ is as follows:

```
FillTab($v, p$)
    //For every position in the table for node $v$
    for ($i = K : 0$)
    {
        $sum = 0$
        //This provided that an heir is needed for this position
        if ($2 \cdot i - 1 < K \,\&\, i \neq 0$)
        {
            $sdist_v[i] = K - (i-1)$
            $bigsum = 0$
            $M_v^p[i] = 0$
            $heir_v^p[i] = null$
            $pd_v^p[i] = i$

            for ($l \in \{N(v) - p\}$)
                $bigsum+ = M_l^v[(sd_v[i] - 1) mod(K+1)]$

            for ($m \in \{N(v) - p\}$)
            {
                $sum = bigsum - M_m^v[(sd_v^p[i] - 1) mod(K+1)] + M_m^v[pd_m^v[i-1]]$

                if ($sum > M_v^p[i]$)
                    $M_v^p[i] = sum$
                    $heir_v^p[i] = m$
            }
        }

        //For every other position
        else
        {
            for ($n \in N(v) - p$)
                $M_v[i]+ = M_m^v[pd_m[i-1]]$

            //If the table position implies that the node can be black
        if (i = 0)
            $M_v^p[i] += 1$

        $heir_v^p[i] = null$
        $pd_v^p[i] = i$
    }

    //Here the history check is preformed
    if ($T_v^p[i] < T_v^p[i+1]$ & $i \neq K$)
    {
        $M_v^p[i] = M_v^p[i+1]$
        $heir_v^p[i] = heir_v^p[i+1]$
        $pd_v^p[i] = pd_v^p[i+1]$
        $sd_v^p[i] = sd_v^p[i+1]$
    }

}
```

We see that rule $R1$ and function $FillTab(v,p)$ combined are in principle identical to the frame $G1$ from Section 4.1.3, with the exception that $G1$ only allowed for potential parents to be selected from v's open neighborhood, while $R1$ selects from v's closed neighborhood. This however has no effect on the correctness or running time of the algorithm. Since $R1$ does indeed fit into the framework set in [2] it follows that it will stabilize, and it will do so in at most O(n^2) moves.

4.2.2 Phase 2

Recall that the point of Phase 2 is to use the information from Phase 1 to set the final-distance, and now that we have a self-stabilizing rule for Phase 1, we can move on to Phase 2. As stated in Section 4.1.5, the need for a root is too heavily woven into Phase 2 to be easily removed. In addition Phase 2 relies on values computed in Phase 1, so there is no way to make it independent in the same way we did with Phase 1. In other words, any rule for Phase 2 will need to use information from both the root finding algorithm and from $R1$. We can now formulate the following rule:

R2:
  if: $!R1 \bigwedge CorrectPointer(v) \bigwedge final_v \neq SetFinal(v)$
  then: $final_v = SetFinal(v)$

The boolean function *CorrectPointer(i)* refers to the root finding algorithm from [3], If the function is true, none of the rules in the root finding algorithm are privileged. In other words the node v knows, for the time being at least, which neighbor is its parent. Also notice that the rule will not become privileged if $R1$ is privileged. These two conditions are two obvious requisites for $R2$ to become privileged, since there is no point in finding the final-distance if either one of the tables M, pd, sd or *heir* is not correct, or if the node in question does not have the correct parent. The function *SetFinal(v)* will find the final-distance and is as follows:

```
SetFinal($v$)
{
    //If the node is the root, it can set the final-distance directly.
    if $(v \to v)$
        $final_v = pd_v^v[0]$

    else
    {
        $p = parent_v$

        //If the current node is its parents heir, or the parent has no heirs.
        if $(Heir_p^{parent_p}[final_p] == v \bigvee Heir_p^{parent_p}[final_p] == null)$
        {
            $final_v = pd_v^p[(final_p - 1) mod(K+1)]$
        }


        //If the parent has an heir, but the current node is not it.
        else
        {
            $final_v = pd_v^p[(sd_p^{parent_p}[final_p] - 1) mod(K+1)]$
        }
    }

    if $(final_v == 0)$
        $color_v = black$
}
```

From Section 4.1.3 we see that $R2$ fits into the frame $H1$. It uses information from both rule $R1$ and from the root finding algorithm. It follows therefor that $R2$ will reach a stable state in at most $\mathrm{O}(n^2)$ moves provided that both the root finding algorithm and $R1$ does not make any moves during that time and the graph does not change. Running time and correctness will be shown in Section 4.4

4.3 Self-stabilizing K-Domination

This section will show how to remake the sequential algorithm for solving the problem minimum K-domination for tree graphs from Chapter 3 into a self-stabilizing algorithm. Doing this will for the most part be identical to the previous section, where the K-packing algorithm from Chapter 2 was turned into a self-stabilizing algorithm. We will still need to make use of the root finding algorithm from [3], and each of the phases in the sequential algorithm will be given as separate rules.

4.3.1 Phase 1

As in the previous section, Phase 1 will have to be able to function without knowledge without a node knowing which of its neighbors is its parent. This means that every node in the graph needs one set of the tables M, pd, sd and $heir$ for every possible parent. In other words: for every $v \in V$ there exists a set $M_v^c[], sd_v^c[], pd_v^c[]$ and $heir_v^c[]$ for every $c \in N[v]$. We can now formulate the following rule:

```
R1:
    if: $\exists p \in N[v] : (M_v^p[], heir_v^p[], pd_v^p[] \; or \; sd_v^p[]) \neq FillTab(v,p)$
    then: $FillTab(v,p)$
```

The rule $R1$ makes use of the function $FillTab(v,p)$, where v is the node in question and p is the assumed parent for this set of the before mentioned tables:

```
FillTab$(v,p)$
for $(i = 0 : 2K)$
{
    $M_v^p[i] = 0$
    $heir_v^p[i] = null$
    $pd_v^p[i] = i$

    //If an heir is needed
    if $(i \leq K \,\&\, i \neq 0)$
    {
        $sd_v^p[i] = 2 \cdot K + 1 - i$
        $bigsum = 0$

        for $(l \in C_v)$
            $bigsum+ = M_l^v[(sd_v[i] - 1) mod(2k+1)]$

        for $(m \in C_v)$
        {
            $sum = bigsum - M_m^v[(sd_v^p[i] - 1) mod(2k+1)] + M_m^v[i]$
            if $(sum < M_v[i])$
                $M_v^p[i] = sum$
                $heir_v^p[i] = m$
        }
    }

    //If an heir is not needed
    else
        for $(c \in C_v)$
            $M_v^p[i] += M_c^v[i-1]$

    //If the node is a leaf and the assumed parent is not itself (ie. $v$ is not the root)
    if $(|N(i)| = 1$ & $v \neq p)$
        for $(j = 1 : K)$
            $M_v^p[j] = 1)$

    //Here the history check is performed ...
    if $(i \geq 1)$
    {
        if $(M_v^p[i-1] < M_v^p[i])$
            $M_v^p[i] = M_v^p[i-1]$
            $pd_v^p[i] = pd_v^p[i-1]$
    }
    // ... and if a history check is not required, $i = 0$ and one is added to the result.
    else
        $M_v^p[0] += 1$
}
```

As with its counterpart in Section 4.2, we see that the $FillTab(v,p)$ function does not make use of the root finding algorithm, and we also see that it fits into the frame $G1$ from Section 4.1.5, since it only reads values from v's neighbors. This is of course with the exception of the history check, where values that were computed previously in the function was used. This, however, does not cause a problem, since it will only cause additional computations internally in a node, and will never cause additional rules to fire. As before, we know that since $R1$ fits into the framework from [2] it will stabilize in no more then $O(n^2)$ moves.

4.3.2 Phase 2

Now that we have a rule for Phase 1, we can move on to Phase 2. Recall that Phase 2 has to make use of $R1$ but also the root finding algorithm from [3]. We can give $R2$ as follows:

R2:

if: $!R1 \bigwedge CorrectPointer(v) \bigwedge final_v \neq SetFinal(v)$

then: $final_v = SetFinal(v)$

As before, there is no need for $R2$ to become privileged as long as R is, and as long as the node v does not have the correct parent. The boolean function $CorrectPointer(v)$ as before only becomes true when none of the rules in the root finding algorithm are privileged. $R2$ makes use of the function $SetFinal$ which is as follows:

```
SetFinal($v$)
{
    //If the node is the root, it can set the final-distance directly.
    if ($v \rightarrow v$)
        $final_v = pd_v^v[K]$

    else
    {
        $p = parent_v$

        //If the current node is its parents heir, or the parent has no heirs.
        if ($Heir_p^{parent_p}[final_p] == v \bigvee Heir_p^{parent_p}[final_p] == null$)
        {
            $final_v = pd_v^p[(final_p - 1) mod (2 \cdot K + 1)]$
        }


        //If the parent has an heir, but the current node is not it.
        else
        {
            $final_v = pd_v^p[(sd_p^{parent_p}[final_p] - 1) mod (2 \cdot K + 1)]$
        }
    }

    if ($final_v == 0$)
        $color_v = black$
}
```

From Section 4.1.3 we see that $R2$ fits into the frame $H1$. It uses information from both rule $R1$ and from the root finding algorithm. It follows therefor that $R2$ will reach a stable state in at most $O(n^2)$ moves provided that both the root finding algorithm and $R1$ does not make any moves during that time and the graph does not change. Running time and correctness will be shown in Section 4.4

4.4 Combining the three algorithms

We now have two rules: $R1$ for Phase 1 and $R2$ for Phase 2 for both algorithms. If we combine these with the root finding algorithm we see that $R1$ does not have any exchange of information with the root finding algorithm and $R2$ uses information from the two others. This may seem to not fit into the framework set down in [2], but since the root finding algorithm and $R1$ has no contact with each other and since they can at most use $O(n^2)$ number of moves to stabilize, the number of times $R2$ can "de-stabilize" due to one of them making a move only increases by a factor of two. This can be viewed as if the Phase 1 algorithm where to also compute the root (instead of the root finding algorithm), the only effect that would have was increasing the number of values that h function in $H1$ has to use in its calculations. In other words, $R1$ and the root finding algorithm combined still runs in $O(n^2)$ moves, and $R2$ still runs in $O(n^2)$, and as proven in [2], combined, they will solve the maximum K-packing problem or minimum K-domination on tree graphs in at most $O(n^3)$ moves, so long as the graph itself does not change.

It may be of interest to note that the rules $R1$ and $R2$ for both algorithms are identical. They only differ in the functions $FillTab()$ and $SetFinal()$. This, perhaps above all else, shows how similar the two problems maximum K-packing and minimum K-domination are. This also means that the two problems can be solved simultaneously on the same tree graph, provided that each problem has its own set of variables. The total running time for wold only increase by a constant factor, thus the asymptotic running time remains the same.

4.5 Conclusion

We have now seen that the two algorithms from Chapter 2 and 3 can be modified to become self-stabilizing algorithms. We have also seen that the self-stabilizing algorithms will stabilize in a polynomial number of moves so long as the graph itself does not change. However, note that the total time spent for the self-stabilizing algorithms is indeed larger than the running time for the sequential algorithms (since there are at most $O(n^3)$ moves, and any one move has to at least perform a fixed amount of calculations). It is not at all uncommon for self-stabilizing algorithms to have a greater running time than their sequential counter parts (sometimes self-stabilizing algorithms even run in exponential time, even if the sequential algorithm runs in polynomial time), and it would indeed have been surprising if this had not been the case here.

Chapter 5

Overall conclusion

We have now seen how the two problems maximum K-packing and minimum K-domination can be solved on tree graphs in time $O(K \cdot n)$ for sequential algorithms and in $O(n^3)$ number of moves for the self-stabilizing algorithms. It is far from uncommon for sequential algorithms on tree graphs to have linear running time, and since the two presented in Chapters 2 and 3 are linear for a constant K, the result found in the two chapters is not unexpected.

It is also worth noting that the algorithms presented here will be able to solve a weighted variant of the two problems. Weighted graphs refers to the cases where every edge and/or node has a weight associated with it. This weight gives the "cost" of selecting the node or edge to be part of a set (meaning that if we select three nodes that weigh 2, 4 and 1, respectively, the total cost would be 7). In the algorithms presented here, remaking them to solve the problems on weighted tree graphs is merely adding the weight of the node v instead of one in the equations 2.4 and 3.3 and the function *FillTab*() for both the self-stabilizing algorithms. This would have no effect on the running time of any of the algorithms.

If we compare the two pairs of algorithms from the above chapters to existing algorithms, we see that we have indeed improved upon them. We have seen polynomial time sequential algorithms for solving problems that up to this point only had exponential time algorithms (which means that these problems can now be solved efficiently on trees). We have also modified the two sequential algorithms, remaking them as self-stabilizing algorithms that run in polynomial time. As mentioned in the previous chapters, there already existed self-stabilizing algorithms for solving maximal 2-packing [12] and minimal K-domination [11], both of which ran in exponential time. The algorithms presented in Chapter 4 improves upon these in several ways: First, the number of moves has been reduced from exponential to polynomial (this due to the fact that the algorithms presented here were designed specifically for a tree graph). Next, the algorithm in this thesis solves K-packing for a general K, not just when $K = 2$. And finally, the algorithms given here finds a maximum/minimum solution, not a maximal/minimal solution.

5.1 Further development

Now that we have seen how all four algorithms perform, is there any room for improvement? Starting with the sequential algorithms, we see that they run in linear time for a constant K. Sub-linear running time would most likely not be possible, since at the very least, the color of every node in the graph has to be set at one point or another. Nor does it seem likely that K could be removed from the equation, since the minimum or maximum distance between two black nodes has to be taken into account during the computations. Thus it is reasonable to conjecture that a running time less than $O(K \cdot n)$ is not possible.

The self-stabilizing algorithms on the other hand is another story. Since they are a relatively new concept, there may indeed be methods for reducing the number of moves that will appear in years to come. It would be interesting to revisit these two problems some time in the future to see if new development has revealed ways to reduce the running time.

5.2 Open problems

5.2.1 Other problems

As has been hinted on several places in the previous chapters, the method used to create the algorithms for maximum K-packing and minimum K-domination on trees may very well be applied on other, similar problems. Recall from Chapter 4 that the rules $R1$ and $R2$ are identical in both the self-stabilizing algorithms. They only differ in the details of the functions $FillTab()$ and $SetFinal()$, and even there they both employ the same tables and variables. So there is evidently the possibility to define a group of problems that would all be solvable using the same method and variables, where only the specifics of the computation of these variables would be different between these problems.

In the two problems dealt with in the previous chapters, a subset of the nodes are selected to form a solution. While it may seem as if every node is therefore selected to have one of two states (black or white), there is more at work here. In fact, there are $K + 1$ states for the K-packing algorithms and $2 \cdot K + 1$ states for the K-domination algorithms (the states are mutually exclusive). That is, a node is selected to have one of these states depending on the algorithms choice of a final-distance for the node in question. Each choice of a state is associated with a cost (the size of the set S for the subtree), and the state which is chosen for a node v has repercussions for the other nodes in $T[v]$. A large part of identifying the specifics for a given problem that is believed to be solvable using the above method would be to identify every state that a node may have, and also how the different states relates to each other. I have not actively searched for problems that can be solved with this method, but I would conjecture that there are several.

5.2.2 Tree-decomposition

As mentioned above, it is not uncommon for a problem with no known polynomial algorithm for a general graph to be solvable in polynomial (and maybe even linear) time on a tree graph. Because of this, a method to "transform" a graph, called tree-decomposition has received much attention in recent years (a sub-variant of tree-decompositions, path-decompositions is also a much studied topic). In short, a tree-decomposition means that we place every node in the graph into one or more "super-nodes" such that every node in the original graph is in at least one super-node, for every pair of nodes with an edge between them there exists at least one super-node where they are both a member, and two super-nodes has en edge between them if and only if at least one node is a member of both super-nodes. In addition, for every node v in the graph, every set of super-nodes that contains v forms a connected component, and the super-nodes themselves form a tree (ie. no cycles). The width of a tree-decomposition is the number of nodes in the largest super-node minus one, and the tree-width of a graph is the smallest width over all tree-decompositions for that graph (for more information about tree-decompositions and related topics, see [18] and [19]). There are many algorithms for otherwise intractable problems that can take advantage of tree-decomposition of a graph to solve the problem in question much faster than conventional algorithms could. In these cases, the running time is usually polynomial for the size of the graph, but increases exponential as the width of the tree-decomposition increases.

It would be an interesting study to see if the algorithms presented in this thesis could be modified to take advantage of a tree-decomposition of a graph to solve the problems maximum K-packing and minimum K-domination on a general graph, only with a running time that increases exponentially with the width of the tree-decomposition, not the size of the graph. If this were indeed possible, it would mean that the two problems could be solved relatively efficiently on a great many graphs where that was previously not possible.

Bibliography

[1] Jean Blair, Pinar Heggernes, Steve Horton, and Fredrik Manne, *Broadcast Domination Algorithms for Interval Graphs, Series-Parallel Graphs, and Trees.* Reports in Informatics 249, University of Bergen, 2003

[2] Jean R. S. Blair and Fredrik Manne. *Efficient Multi-Stage Self-stabilizing for Tree Networks.*

[3] Jean R. S. Blair and Fredrik Manne. *Efficient Self-stabilizing algorithms for Tree Networks.*, Presented at ICDCS'03, The 23rd International Conference on Distributed Computing Sustems, 2003

[4] Dorit S. Hochbaum and David B. Schmoys. *A best possible heuristic for the k-center problem.* Mathematics of Operations Research, 10(2):180-184, 1985.

[5] F. Harary. *Graph Theory.* Addison Wesley, Reading , MA, 1969.

[6] F. Garvil. *Algorithms for minimum coloring, maximum clique, minimum covering by cliques and maximum independent set of a chordal graph.* SIAM J. Comput. 1 1972, 180-187

[7] S.L. Mitchell, E.J. Cockayne and S.T. Hedetniemi, *Linear algorithms on recursive representations of trees*, J. Comput. System Sci. 18 , 76-85 (1979).

[8] E.O Hare, W.R. Hare, *k-Packing of Pm x Pn*, At the 22nd Southeastern International Conference on Combinatorics, Graph Theory and Computing, Feb. 11 - 15, 1991, Baton Rouge, LA

[9] M.R. Garey, D.S. Johnson, *Computers and Intractability, A guide to the Theory of NP-Completeness*, W.H. Freeman and company, San Francisco.

[10] E. W. Dijkstra. *Self-stabilizing systems in spite of distibuted control.* Communications of the ACM, 17:643-644, 1974.

[11] Martin Gairing, Stephen T. Hedetniemi, Peter Kristiansen and Alice A. McRae: *Self-Stabilizing Algorithms for $\{k\}$-domination*, Proceedings of the 6th Symposium on Self-Stabilization (SSS 2003), Springer LNCS 2704, pages 49-60, 2003..

[12] Martin Gairing, Robert M. Geist, Stephen T. Hedetniemi and Petter Kristiansen: *A self-stabilizing algorithm for maximal 2-packing*, Technical Report 230, Department of Informatics, University of Bergen, 2002.

[13] A. Meir and J. W. Moon: *Realtions between packing and covering numbers of a tree.* Pacific journal of mathematics, 1975, 61 nr.1, 225-233.

[14] S.M. Hedentniemi, S.T. Hedentniemi, D.P. Jacobs and P.K. Srimani: *Self-stabilizing algorithms for minimal dominating sets and maximal independent sets.* 2001

[15] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein: *Introduction to algorithms*, The MIT press, 2001.

[16] Mikhail J. Atallah, *Algorithms and Theory of Computation Handbook*, CRC press, 1998.

[17] R. G. Downey, M.F. Fellows, M. R. Fellows, F. Schneider, *Parameterized Complexity*, Springer Verlag, 1998.

[18] Hans L. Bodlaender, *A Tourist Guide through Treewidth*, 1992 (revised 1993)

[19] Pinar Heggernes, *Advanced Graph Theoretical Topics*, Partial curriculum in I238/INF334 Algorithms II, Department of Informatics, University of Bergen, Norway, 2001

[20] Hong-Gwa Yeh and Gerard J. Chang, *Weighted connected k-domination and weighted k-dominating clique in distance-hereditary graphs*, Theoretical Computer Science 263 (2001) 3-8.

[21] J. Cheriyan, T. Jordn and R. Ravi, *On 2-coverings and 2-packings of laminar families*, Proceedings of the 7th Annual European Symposium on Algorithms, p.510-520, July 16-18, 1999

[22] Gerard J. Chang and George l: Nemhauser, *The k-domination and k-stability problems on sun-free chordal graphs*, Siam J. Algebraic Discrete Methods, 5, 332-345, 1984.

[23] Ted Herman, *A Comprehensive Bibliography on Self-Stabilization*, Available at **ftp://ftp.cs.uiowa.edu/pub//selfstab/bibliography/index.html**