NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents

Partition Algorithm for the Dominating Set Problem

Ka Leung Ma

A Thesis
in
The Department
of
Computer Science

Presented in Partial Fulfilment of the Requirements for
the Degree of Master of Computer Science at
Concordia University
Montréal, Québec, Canada

July 1990

ISBN 0-315-59150-1

Abstract

Partition Algorithm for the Dominating Set Problem

K. L. Ma

An algorithm to find the dominating sets of a specific size in a given graph was developed. The algorithm will recursively select a group of vertices, called a cell, to partition into several subcells and then partition the content of the cell among its subcells. Two tests, namely the Coverage Test and the Wastage Test, were used to cut down the search tree generated by the algorithm. A program was developed based on this algorithm. Using the program, we confirmed that the minimum dominating number of the 8×8 grid graph is 16, the minimum domination number of the 8×9 grid graph is greater than 17, and the minimum domination number of the 9×9 grid graph is greater than 19. We also obtained the dominating sets of several grid graphs and knight's graphs in an $m \times n$ chessboard with various test sizes.

Acknowledegments

I would like to express my sincere gratitude to Dr. Clement Lam, my thesis supervisor, for his guidance and valuable insight throughout this research. His support and patience were invaluable in the preparation of this thesis.

I would like to thank the staff of the Department of Computer Services and the Department of Computer Science at Concordia University for their permission to use their computer facilities.

I am indebted to my parents and my brother for their patience, understanding and encouragement throughout this research.

At various stages of this research, a number of my friends have given me encouragement, valuable suggestions, and different kinds of support. In this regard, I owe a debt of gratitude to all of them.

This research was supported by a scholarship from the Natural Sciences and Engineering Research Council of Canada.

Contents

List of Figures

List of Tables

Chapter 1

INTRODUCTION

The study of the domination number in various combinatorial problems dates back to the 1800's when questions concerning the optimum placement of chess pieces on a chessboard were first published in [1,10]. Two earlier discussions can be found in [3,24]. A more detailed discussion, which contains a quick review of results and applications concerning dominating sets in graphs, can be found in [9]. Different combinatorial problems such as the domination number of grid graphs [7], knight's domination number of a chessboard [14], and the football pool problem [18,19] can be solved by representing the problem by a graph and obtaining its domination numbers. Solving these combinatorial problems will contribute to areas like networking, VLSI, and database management. For example, grid graphs have been used to model a variety of routing problems in street networks and processor interconnections in multiprocessor VLSI systems. One can refer to [11,12,15,20,25,27,28,29] for some other applications of grid graphs. In [21], the author discusses the application of the domination number to communications in a network, where a dominating set represents a set of cities which, acting as transmitting stations, can transmit messages to every city in the network. Some basic concepts concerning graphs and the dominating set problem are discussed in [2,4,5,13,22,26].

In this thesis we present a partitioning algorithm we developed to obtain all the dominating sets of a graph with a specified domination number s. Our algorithm was motivated by [19] in which the authors used a similar method, but in different terminology, to prove that 27 bets are required for the football pool problem with 5

matches. Based on our algorithm, we implemented a program which takes a graph and an integer s as inputs. The program will find all the dominating sets of size s for that graph. Using the program, we have confirmed that the best known upper bound of the minimum domination number of a 8×8 grid graph published in [7] is indeed optimal. We have also confirmed that the minimum domination number of the 8×9 grid graph is greater than 17, and the minimum domination number of the 9×9 grid graph is greater than 19. Moreover, we have used the program to verify that the knight's dominating sets in a 3×11, a 4×6, and a 5×12 chessboard published in [14] are indeed complete. Theorems developed in [6,8,16,23] may be verified using our program.

The layout of the thesis is as follows. Chapter 2 discusses basic definitions and mathematical preliminaries for the dominating set problems. Chapter 3 talks about the methodology used in the program to reduce the computational time. Chapter 4 discusses results obtained when we applied this program to find the the dominating sets of a grid graph and the dominating sets of a knight's graph in an $m \times n$ chessboard. Finally, Chapter 5 gives a conclusion and suggestions about further work which can be carried out in this project.

Chapter 2

MATHEMATICAL PRELIMINARIES

In this chapter some basic concepts and definitions about the dominating set problem will be presented. The method used to solve the dominating set problem and the theorems involved will also be explained.

2.1 Basic Definitions

Given a graph $G(V, E)$, a vertex u is said to be *covered* or *dominated* by a vertex v if and only if $(u, v) \in E$ or $v = u$. The *extended edge set* of E, denoted E', is equal to $E \cup \{(u, u) : \forall u \in V\}$. For all graphs, we work on the extended edge set. A graph, $G(V, E)$, is said to be covered by a set of vertices V' if and only if the union of the sets of vertices covered by each vertex in V' is equal to V. V' is a *dominating set* of G. The *domination number* of G is the number of vertices in V'. The *minimum domination number* is the cardinality of the smallest V' that covers G. Given a graph $G(V, E)$, one of its domination numbers is $|V|$ and all its domination numbers are less than or equal to $|V|$. Figure 2.1 shows an example of a 4×4 grid graph. The minimum domination number is 4 and the largest domination number is 16. One of the dominating sets with domination number 4 is $\{b, h, i, o\}$.

Our approach in solving the dominating set problem is to use a partition algorithm to partition a graph recursively until all the solutions are obtained. A *solution* is a dominating set of size s, where s is an integer input to the algorithm. The vertices

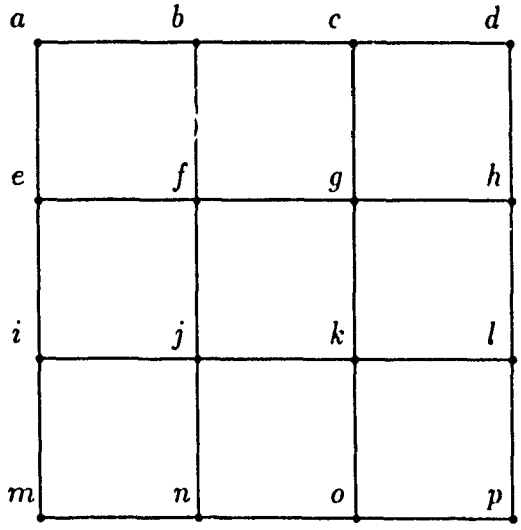

Figure 2.1: A 4×4 grid graph.

of the graph are partitioned into *cells* where each cell is a set of vertices. The *size* of a cell, $size(C)$, is the number of vertices in the cell C. Initially, all the vertices are in one single cell. The *refinement* of a cell is the partitioning of the vertices in the cell into subcells. If a cell C is refined into a set of subcells, $\{C_1, \ldots, C_n\}$, we call C the *parent* of $C_1, \ldots, C_n$, and $C_1, \ldots, C_n$ are called the *children* of C. Two cells are *siblings* of one another if they have the same parent.

In the dominating set problem, a vertex is either in or not in the dominating set. We say a vertex is *on* if a vertex is in the dominating set, otherwise it is *off*. The *content* of a cell is the number of vertices in the cell that are on. We use $\sigma(C)$ to denote the content of a cell C. Clearly

$$0 \leq \sigma(C) \leq size(C).$$

It is also clear that when a cell is refined, the total content of the subcells is equal to the content of the original cell. This will lead us to the following theorem:

Theorem 2.1 *If a cell C is refined into subcells $C_1, \ldots, C_n$, then*

$$\sum_{i=1}^{n} \sigma(C_i) = \sigma(C).$$

We will call the theorem above the *Conservation Rule*.

2.2 The Partition Algorithm

The partition algorithm will take a graph $G(V, E)$ and an integer s, called the *test size*, as its inputs. Starting with V as a single cell and s as its content, the algorithm will recursively refine the cell and partition its content. A solution can be obtained only when the algorithm comes to a *complete refinement*. A complete refinement is the last refinement of the cells which leads to $|V|$ cells each of size 1 and the total content of all the $|V|$ cells must be equal to s.

In order to cut down the amount of computation time, before each refinement of the cells one has to check whether further refinement will lead to a solution. Two tests were developed to achieve this purpose. The first test is called the *Coverage Test* and the second one is called the *Wastage Test*. They restrict the possible partitions of the parent's content among its children cells. The two tests guarantee that those partitions which are not considered will never lead to a solution. The idea of partitioning the vertices was used in [19], and the Coverage Test is a generalized version of the method used there.

2.2.1 The Coverage Test

Before we discuss the Coverage Test, we have to first introduce the following definitions. A cell C_i is said to be a *neighbour* of a cell C_j if there exists an edge (u, v), where $(u, v) \in E'$ with $u \in C_i$ and $v \in C_j$. We use $C_i \sim C_j$ to denote C_i is a neighbour of C_j and $C_i \not\sim C_j$ to denote C_i is not a neighbour of C_j. We will call C_i the *neighbour cell* and C_j the *master cell*. As we work on the the extended edge set E', each cell C_i is a neighbour of itself. The Coverage Test can be informally stated as follows:

> The content of a cell C and its neighbours must be large enough to cover all the vertices in C.

To be more formal, we need to state clearly the interaction between one cell and the other. With every ordered pair $< C_i, C_j >$ of cells, we define the *upper influence function* of C_i on C_j, with a parameter m, to be the upper bound on the number

of edges (u,v) in E' with $u \in C_i, v \in C_j$ and the number of distinct u equals to m. The upper influence function is denoted as $U_{C_i,C_j}(m)$ or $U_{ij}(m)$ in short, where m is an integer between 0 and $size(C_i)$. If $\sigma(C_i) = m$, then the maximum number of vertices in C_j covered by m vertices in C_i is $U_{ij}(m)$. One way to compute $U_{ij}(m)$, is by sorting the vertices of C_i descendingly according to the number of edges in the extended edge set going from a vertex $u \in C_i$ to the vertices in C_j and then counting the total number of edges going into C_j in the first m vertices.

Let us consider an example using the 4 × 4 grid graph in Figure 2.1. Assume $C_1 = \{a, b, c, d, e, f, g, h\}$ and $C_2 = \{i, j, k, l, m, n, o, p\}$, then the descending order of the vertices in C_1 according to the number of edges in the extended edge set going from a vertex $u \in C_1$ to the vertices in C_2 is e, f, g, h, a, b, c, and d, with the number of edges equal to 1, 1, 1, 1, 0, 0, 0, and 0, respectively. Using these data, we can generate U_{12}. The following will show some examples of different upper influence functions:

$$U_{12} = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 0 & 1 & 2 & 3 & 4 & 4 & 4 & 4 & 4 \end{bmatrix} = U_{21},$$

and

$$U_{11} = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 0 & 4 & 8 & 12 & 16 & 19 & 22 & 25 & 28 \end{bmatrix} = U_{22}.$$

The upper row in the array represents the parameter m in the function and the lower row in the array represents the value of the function, $U_{ij}(m)$. In general, U_{ij} is nonlinear, but one can approximate the function by letting $U_{ij}(x) \leq U_{ij}(1) \times x$ where x is any integer from 0 to $size(C_i)$. In this example, $U_{ij} = U_{ji}$, but in general they are not equal. Consider the following example. Choose $C_1 = \{a, d, m, p\}$ (the four corners), $C_2 = \{b, c, e, h, i, l, n, o\}$ (the vertices on the four edges except the corners), and $C_3 = \{f, g, j, k\}$ (the rest of the vertices), then

$$U_{12} = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 \\ 0 & 2 & 4 & 6 & 8 \end{bmatrix} = U_{32},$$

$$U_{21} = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{bmatrix} = U_{22} = U_{23},$$

$$U_{13} = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} = U_{31},$$

$$U_{11} = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 \\ 0 & 1 & 2 & 3 & 4 \end{bmatrix},$$

and

$$U_{33} = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 \\ 0 & 3 & 6 & 9 & 12 \end{bmatrix}.$$

Notice that the largest m in $U_{12}(m)$ is different from the largest m in $U_{21}(m)$. This is because the former m is $size(C_1)$ while the latter m is $size(C_2)$.

Suppose that there are n cells. The Coverage Test requires that for each cell C_j, $\sum_{i=1}^{n} U_{ij}(\sigma(C_i)) \geq size(C_j)$. Notice that this is not a linear inequality. In addition, if C_i and C_j are not neighbours, then U_{ij} is the zero function. Hence this lead to the following theorem:

Theorem 2.2 *For each cell C_j, the necessary condition for C_j to be covered is*

$$\sum_{C_i \sim C_j} U_{ij}(\sigma(C_i)) \geq size(C_j).$$

Unfortunately, while we are trying to find a dominating set, we seldom know exactly what the $\sigma(C_i)$ are. Quite often, we only know the upper bound and the lower bound of the content of a cell. Based on these bounds, there may be several ways to partition the content after each refinement of a cell. The partition algorithm will try all these ways and see if any of them will lead to a solution when it comes to a complete refinement.

In order to apply the Coverage Test in the partition algorithm, we have to further refine Theorem 2.2. Let us define the upper bound of the content of a cell C as $\overline{\sigma}(C)$ and the lower bound of the content of a cell C as $\underline{\sigma}(C)$. In short, we will call $\overline{\sigma}(C)$ as the *upper content* of C and $\underline{\sigma}(C)$ the *lower content* of C. Obviously, $\underline{\sigma}(C) \leq \sigma(C) \leq \overline{\sigma}(C)$. Finally, we will define S_{kj}, the *S-slack* of a cell C_k relative to a cell C_j, by

$$S_{kj} = U_{kj}(\overline{\sigma}(C_k)) - U_{kj}(\underline{\sigma}(C_k)).$$

Notice that if $\overline{\sigma}(C_i) = \sigma(C_i) = \underline{\sigma}(C_i)$, then $S_{kj} = 0$. Using the S-slack, the Coverage Test is defined in the next theorem.

Theorem 2.3 *For each cell C_j and for all neighbour cells C_k of C_j, in order for C_j to be covered, the following inequality should hold:*

$$\left[\sum_{C_i \sim C_j} U_{ij}(\overline{\sigma}(C_i))\right] - size(C_j) \geq S_{kj}.$$

Proof: In order to cover C_j, the best case is when each edge from the neighbour of C_j covers one distinct vertex in C_j. If all the neighbours except C_k have upper content, then the lower content of C_k must be large enough to cover all the uncovered vertices in C_j. Based on this idea, we obtained the following inequality,

$$U_{kj}(\underline{\sigma}(C_k)) \geq size(C_j) - \sum_{(C_i \sim C_j) \wedge (i \neq k)} U_{ij}(\overline{\sigma}(C_i)).$$

Subtracting $U_{kj}(\overline{\sigma}(C_k))$ from both sides of the inequality, we will get

$$\begin{aligned} & U_{kj}(\underline{\sigma}(C_k)) - U_{kj}(\overline{\sigma}(C_k)) \geq size(C_j) - \sum_{C_i \sim C_j} U_{ij}(\overline{\sigma}(C_i)) \\ \Leftrightarrow \quad & -S_{kj} \geq size(C_j) - \sum_{C_i \sim C_j} U_{ij}(\overline{\sigma}(C_i)) \\ \Leftrightarrow \quad & \left[\sum_{C_i \sim C_j} U_{ij}(\overline{\sigma}(C_i))\right] - size(C_j) \geq S_{kj}. \ \square \end{aligned}$$

If the inequality is not satisfied, $\underline{\sigma}(C_k)$ must be raised so that $U_{ij}(\underline{\sigma}(C_k))$ is greater than or equal to the number of uncovered vertices in C_j.

After a refinement, suppose there are q subcells in total. According to Theorem 2.3, for each fixed j, these q subcells $C_1, \ldots, C_q$ will lead to q inequalities: $\left[\sum_{C_i \sim C_j} U_{ij}(\overline{\sigma}(C_i))\right] - size(C_j) \geq S_{kj}$, for $k = 1$ to q. In the beginning, we initialize $\underline{\sigma}(C_k)$ to $\sigma(C_k)$ if it is defined, otherwise $\underline{\sigma}(C_k)$ is initialized to 0. After applying the Coverage Test, if any of the inequalities is not satisfied, the corresponding S-slack, S_{kj}, will be reduced by raising the lower content of C_k. Hence, by applying the Coverage Test, one can improve $\underline{\sigma}(C_k)$. If the application of the Coverage Test leads to a $\underline{\sigma}(C_k)$ which is greater than $\overline{\sigma}(C_k)$, this means that it is impossible to cover all the vertices in C_j, hence, further refinement will not lead to any solution and the partition algorithm would backtrack to previous stage. The algorithm to implement the Coverage Test is summarized below.

The Coverage Test algorithm

for each C_j

total upper influence $\leftarrow 0$;

for each $C_i \sim C_j$

total upper influence $\leftarrow$ *total upper influence* $+ U_{ij}(\overline{\sigma}(C_i))$;

check value $\leftarrow$ *total upper influence* $- size(C_j)$;

for each $C_k \sim C_j$

exit $\leftarrow$ false;

repeat

work out S_{kj};

if *check value* $< S_{kj}$ **then**

if $\overline{\sigma}(C_k) > \underline{\sigma}(C_k)$ **then**

increase $\underline{\sigma}(C_k)$ by 1;

else

report no solution in this partition of content;

return;

else

exit $\leftarrow$ true;

until *exit*;

2.2.2 The Wastage Test

A vertex is said to have a *wastage* of t if from all its neighbours there is a total of $t+1$ distinct edges connected to this vertex. Again, we should remember that we are working on the extended edge set. The *maximum possible wastage* of a cell C_j, $W(C_j)$, is defined by

$$W(C_j) = \left[\sum_{C_i \sim C_j} U_{ij}(\sigma(C_i)) \right] - size(C_j)$$

where $\sum_{C_i \sim C_j} U_{ij}(\sigma(C_i))$ represents the maximum possible coverage for the cell C_j. Hence, after subtracting the size of C_j, one will get the maximum possible wastage.

In order to work out the maximum possible wastage of each cell, we have to reach a state where the content of each cell is defined. Notice that the maximum possible wastage is not conserved. Consider the 4×4 grid graph in Figure 2.1. Assume $C = \{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p\}$, $C_1 = \{a, b, c, d, e, f, g, h\}$, $C_2 = \{i, j, k, l, m, n, o, p\}$, $\sigma(C) = 6$, $\sigma(C_1) = 3$, and $\sigma(C_2) = 3$. Then $W(C) = (4 \times 5 + 2 \times 4) - 4 \times 4 = 12$, whereas $W(C_1) = (3 \times 4 + 3 \times 1) - 4 \times 2 = 7$ and $W(C_2) = (3 \times 4 + 3 \times 1) - 4 \times 2 = 7$. In the process of working out $W(C_1)$, the 3×4 is $U_{11}(\sigma(C_1))$ and the 3×1 is $U_{21}(\sigma(C_2))$.

Before we define the Wastage Test, we need to define a function to state clearly the interaction between one cell and the other. With every ordered pair of cells $< C_i, C_j >$, we define the *lower influence function* of C_i on C_j, with a parameter m, to be the lower bound of the number of edges $(u, v) \in E'$ with $u \in C_i, v \in C_j$ and the number of distinct u being m. The lower influence function is denoted as $L_{C_iC_j}(m)$ or $L_{ij}(m)$ in short, where m is an integer between 0 and $size(C_i)$. If $\sigma(C_i) = m$, then the minimum number of edges between C_j and C_i is $L_{ij}(m)$. First, the vertices of C_i are sorted ascendingly according to the number of edges in the extended edge set that connect a vertex $u \in C_i$ to the vertices in C_j. $L_{ij}(x)$ is then obtained by counting the total number of edges going into C_j from the first x vertices in C_i.

Let us consider the 4×4 grid graph example which is shown in Figure 2.1. Assume $C_1 = \{a, b, c, d, e, f, g, h\}$ and $C_2 = \{i, j, k, l, m, n, o, p\}$, then the ascending order of the vertices in C_1 according to the number of edges in the extended edge set going from a vertex $u \in C_1$ to the vertices in C_2 is a, b, c, d, e, f, g, and h, with the number of edges equal to 0, 0, 0, 0, 1, 1, 1, and 1, respectively. Using these data, we can generate L_{12}. The following are some examples of different lower influence functions:

$$L_{12} = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 0 & 0 & 0 & 0 & 0 & 1 & 2 & 3 & 4 \end{bmatrix} = L_{21},$$

and

$$L_{11} = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 0 & 3 & 6 & 9 & 12 & 16 & 20 & 24 & 28 \end{bmatrix} = L_{22}.$$

The upper row in the array represents the parameter m in the function and the lower row in the array represents the value of the function, $L_{ij}(m)$. In general, L_{ij}

is nonlinear, but one can approximate the function by letting $L_{ij}(x) \geq L_{ij}(1) \times x$ where x is any integer from 0 to $size(C_i)$. In this example, $L_{ij} = L_{ji}$, but in general they are not equal. Consider the following example. Choose $C_1 = \{a,\ d,\ m,\ p\}$ (the four corners), $C_2 = \{b,\ c,\ e,\ h,\ i,\ l,\ n,\ o\}$ (the vertices on the four edges except the corners), and $C_3 = \{f,\ g,\ j,\ k\}$ (the rest of the vertices), then

$$L_{12} = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 \\ 0 & 2 & 4 & 6 & 8 \end{bmatrix} = L_{32},$$

whereas

$$L_{21} = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{bmatrix} = L_{22} = L_{23},$$

$$L_{13} = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} = L_{31},$$

$$L_{11} = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 \\ 0 & 1 & 2 & 3 & 4 \end{bmatrix},$$

and

$$L_{33} = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 \\ 0 & 3 & 6 & 9 & 12 \end{bmatrix}.$$

Notice that the largest m in $L_{12}(m)$ is different from the largest m in $L_{21}(m)$. This is because the former m is $size(C_1)$ while the later m is $size(C_2)$. Also notice that, $\forall i, j : L_{ij}(size(C_i)) = U_{ij}(size(C_i))$, $L_{ij}(0) = U_{ij}(0) = 0$, and $C_i \not\sim C_j$ iff $\forall x(L_{ij}(x) = U_{ij}(x) = 0)$ where x is any integer from 0 to $size(C_i)$. Hence in the last example, $C_1 \not\sim C_3$.

The *unavoidable wastage* of a cell C_j, denoted by $\underline{W}(C_j)$, is defined as follows:

$$\underline{W}(C_j) = maximum\ of \begin{cases} 0 \\ \left[\sum_{C_i \sim C_j} L_{ij}(\underline{a}(C_i))\right] - size(C_j). \end{cases}$$

The term $\sum_{C_i \sim C_j} L_{ij}(\underline{a}(C_i))$ will give the best minimum coverage C_j can get from all its neighbours. Hence, if $\left[\sum_{C_i \sim C_j} L_{ij}(\underline{a}(C_i))\right] - size(C_j)$ is greater than or equal to zero, it will give the the unavoidable wastage of the cell C_j. Since $\left[\sum_{C_i \sim C_j} L_{ij}(\underline{a}(C_i))\right] - size(C_j)$ may be negative, one has to take the maximum between this value and zero. Let us denote C_j as a child of C by $C_j \dashv C$. The Wastage Test can be stated in the following theorem.

Theorem 2.4 *The necessary condition for a cell C to be covered is*

$$\sum_{C_j \dashv C} \underline{W}(C_j) \leq W(C).$$

Informally, the above Theorem means that the known unavoidable wastage among all the children of a cell C cannot be bigger than the maximum possible wastage of C. Based on the lower influence function, we will define Z_{kj}, the *Z-slack* of a cell C_k relative to a cell C_j to be

$$Z_{kj} = L_{kj}(\overline{\sigma}(C_k)) - L_{kj}(\underline{\sigma}(C_k)).$$

By choosing a cell C_k to have the upper content, or equivalently by defining

$$\underline{W_k}'(C_j) = max \left\{ \begin{array}{l} 0 \\ \left[\sum_{C_i \sim C_j} L_{ij}(\underline{\sigma}(C_i))\right] - size(C_j) + Z_{kj} \end{array} \right. ,$$

then the Wastage Test can be restated in the following theorem:

Theorem 2.5 *Let the cell C_k have its upper content, then the necessary condition for a cell C to be covered is*

$$\sum_{C_j \dashv C} \underline{W_k}'(C_j) \leq W(C).$$

Proof: If the upper content of the cell C_k is used, then by the definition of the unavoidable wastage,

$$\begin{aligned} \underline{W}(C_j) &= max \left\{ \begin{array}{l} 0 \\ \left[\sum_{(C_i \sim C_j) \wedge (i \neq k)} L_{ij}(\underline{\sigma}(C_i))\right] - size(C_j) + L_{kj}(\overline{\sigma}(C_i)) \end{array} \right. \\ &= max \left\{ \begin{array}{l} 0 \\ \left[\sum_{C_i \sim C_j} L_{ij}(\underline{\sigma}(C_i))\right] - size(C_j) + Z_{kj}. \end{array} \right. \end{aligned}$$

When the upper content of the cell C_k is used, $\underline{W}(C_j)$ will become $\underline{W_k}'(C_j)$, hence, Theorem 2.4 can be rewritten as:

$$\sum_{C_j \dashv C} \underline{W_k}'(C_j) \leq W(C). \ \square$$

Suppose we are at a stage where a refinement is just finished, and there are q subcells in total. Each time when we choose a different subcell C_k to have the upper

content $\overline{\sigma}(C_k)$, it will lead to q unavoidable wastage, $\underline{W_k}'(C_j)$, where j varies from 1 to q. Tracing back the parents of all the subcells, different inequalities can be set up. In the beginning, we initialize $\overline{\sigma}(C_k)$ to $\sigma(C_k)$ if it is defined, otherwise $\overline{\sigma}(C_k)$ is initialized to the minimum of $size(C_k)$ and $\sigma(C)$, where $C_k \dashv C$. After applying the Wastage Test, if any of the inequalities is not satisfied, the Z-slack, Z_{kj}, will be reduced by lowering the upper content of C_k. If this leads to a $\overline{\sigma}(C_k)$ which is smaller than $\underline{\sigma}(C_k)$, then it means that at least one of the parents' maximum possible wastage is less than the known unavoidable wastage among all its children. Hence, further refinement will not lead to any solution. The partition algorithm would backtrack to previous stage. The following is an algorithm for the Wastage Test.

The Wastage Test algorithm

for all C_j after the latest refinement, work out the wastage of its parent $W(C)$;
work out all the $\underline{W}(C_j)$;
repeat
    select a new k;
    repeat
        pass ← true;
        work out all Z_{kj};
        work out all $\underline{W_k}'(C_i)$;
        for each parent C
            if $\sum_{C_i \dashv C} \underline{W_k}'(C_i) > W(C)$
            then if $\overline{\sigma}(C_k) > \underline{\sigma}(C_k)$
                then decrease $\overline{\sigma}(C_k)$ by 1;
                    pass ← false;
                else report no solution in this partition of content;
                    return;
    until *pass*;
until no new k;

Chapter 3

IMPLEMENTATION DETAILS

Based on the theorems in Chapter 2, we have developed a program to find the dominating set of a graph $G(V, E)$, with a given test size s. The program is implemented using the language C. C is chosen because it is closely associated with the UNIX operating system on which the program is run. In addition, it can support some frequently used operations like dynamic array allocation and logical operations.

We define a *depth* as a group of cells. It is numbered starting from zero. Depth zero is the cell which represents the whole graph. The program recursively selects a cell and partitions it into subcells. Once a cell is selected and partitioned, all its subcells together with those cells in the current depth which are not partitioned form a new depth. This new depth becomes the *latest depth*. The latest depth indicates how close the program is from a complete refinement. The depth number shows the level of the recursion. When the program reaches a new depth, it try to partition the available content amongst all the newly generated subcells. Besides using the Coverage Test and the Wastage Test to reduce the number of ways to partition the content, special designs are used in the program to reduce the computational time of the frequently used operations. The rest of this chapter talks about some of these special designs.

3.1 Data Structure

The data structure of a cell is implemented as a structure to store all the frequently referenced information. In order to save memory space and facilitate the partition

of a cell, the set of vertices is represented by an array of bits, and is named a *bit array*. The existence of a vertex is indicated by setting its corresponding bit to one. We will number the vertices starting from zero. If a graph contains r vertices and a computer word has w bits, then the bit array uses $\lceil \frac{r}{w} \rceil$ words. From now on, we will use the word *word* to mean a computer word. For example, if a word is 32 bits long, and the graph contains 40 vertices, then the bit array is two words long. Given a cell containing three vertices 0, 5, and 34, the bit array representing the cell will have positions 0, and 5 of the first word and position 2 of the second word set to one and the rest set to zero. The following is the data structure of a cell defined in the program and Figure 3.1 shows a pictorial view of such a cell. Notice that each cell contains an array of words to indicate which vertex in the graph is in this cell.

```
struct cell_node
  {
     struct cell_node *parent;       /* parent cell's address */
     struct cell_node *sibling;      /* next sibling's address */
     struct cell_node *first_child; /* first child's address */
     unsigned *vertices_set;         /* an array of computer words
                                        used to represent the
                                        vertices in this cell */
     int prev_upper_content,         /* previous upper content */
         upper_content,              /* current upper content */
         prev_lower_content,         /* previous lower content */
         lower_content,              /* current lower content */
         actual_content,             /* content of the cell */
         total_children,             /* number of children */
         total_vertices,             /* number of vertices */
         max_poss_wastage;           /* maximum possible wastage */
  };
```

To understand why the data structure of a cell_node is made as such, let us look at some of the ways the different fields are used. The parent, sibling, first_child, and

cell_node	data structure
parent	pointer to cell_node
sibling	pointer to cell_node
first_child	pointer to cell_node
vertices_set	bit array
prev_upper_content	integer
upper_content	integer
prev_lower_content	integer
lower_content	integer
actual_content	integer
total_children	integer
total_vertices	integer
max_poss_wastage	integer

Figure 3.1: The cell node.

max_poss_wastage fields are used in the wastage test. The vertices_set field, which represents all the vertices contained in this cell, is used to find the neighbours of this cell. When the program backtracks, the prev_lower_content and the prev_upper_content fields are used to remember and then restore the values of the lower and upper contents of the cell. The total_children, upper_content, and lower_content fields are used by the program to generate all the partitions of the parent's content. The actual_content field is used to work out the maximum possible wastage of this cell. Finally, the total_vertices field is used when the program is generating the lower and upper influence functions.

When a cell C is partitioned into n subcells, $C_1, \ldots, C_n$, a new depth is reached, and this depth is one level deeper than the last depth. The subcells contained in this depth are the cells in the last depth except C, which is replaced by $C_1, \ldots, C_n$. Since the lower and upper contents of each of the newly created subcells will be improved by using the Coverage and Wastage Tests, all the subcells at this depth should be stored in a way that they can be easily accessed. The data structure of the depth is designed in a way that all the subcells in the same depth can be easily accessed. The

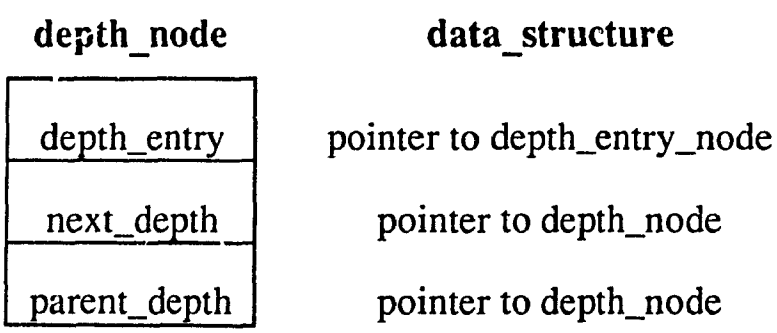

Figure 3.2: The depth node.

following are the data structures defined in the program. The pictorial views of a depth_node, a depth_entry_node, a neigh_info_node, and an influ_func_node are given in Figure 3.2 to Figure 3.5.

```
/* Depth node is the header node of each depth. */

struct depth_node
  {
     struct depth_entry_node *depth_entry;
     struct depth_node *next_depth, *parent_depth;
  };


/* Depth entry node contains a pointer to the master cell
   and a pointer to the neighbour information node. */

struct depth_entry_node
  {
     struct cell_node *corr_master_cell;
     struct neigh_info_node *neigh_info;
     struct depth_entry_node *next_entry;
  };
```

depth_entry_node	data_structure
corr_master_cell	pointer to cell_node
neigh_info	pointer to neigh_info_node
next_entry	pointer to depth_entry_node

Figure 3.3: The depth entry node.

neigh_info_node	data_structure
corr_neigh_cell	pointer to cell_node
other_neigh_info	pointer to neigh_info_node
influ_func	pointer to influ_func_node

Figure 3.4: The neighbour information node.

```
/* The NEIGHbour INFOrmation NODE contains a pointer
   to a neighbour of the master cell, a pointer to other
   neighbour information node, and a pointer to the
   corresponding influence function. */

struct neigh_info_node
  {
     struct cell_node *corr_neigh_cell;
     struct neigh_info_node *other_neigh_info;
     struct influ_func_node *influ_func;
  };

/* The INFLUence FUNCtion NODE contains a sorted list of
```

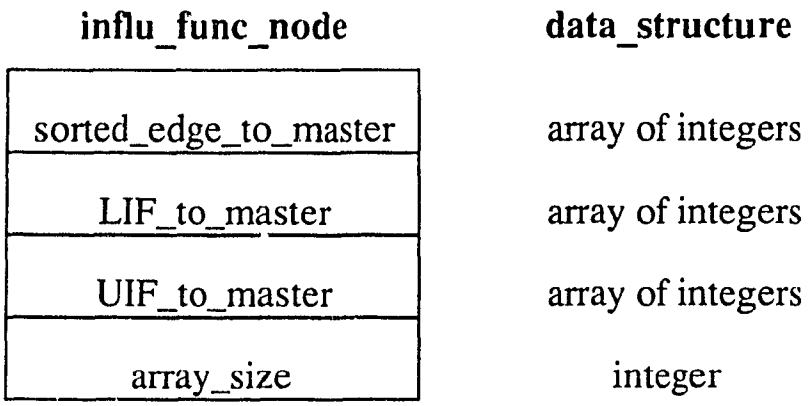

Figure 3.5: The influence function node.

```
   the number of edges from each vertex of a neighbour to the
   master cell. Based on this list, a list of Lower Influence
   Function and Upper Influence Function are also kept.
   Notice that in SUN 4, an integer contains 32 bits, hence
   it can be used to represent 32 vertices. */

struct influ_func_node
  {
     int *sorted_edge_to_master, /* an array of integers */
         *LIF_to_master,         /* an array of integers */
         *UIF_to_master,         /* an array of integers */
         array_size;             /* the size of the array */
  };
```

We design the data structure corresponding to a depth to facilitate the calculation of the Coverage Test, the Wastage Test, and the working out of the maximum possible wastage of each cell. It is implemented to make backtracking easy. Figure 3.6 shows how the data structure of the cells and the depths are related after a 2×2 grid graph with test size two is partitioned into two subcells of two vertices with each subcell having content one. Notice that a negative value in the prev_lower_content and the

prev_upper_content fields means that these values are undefined.

The depth nodes are the header nodes for each depth. They are linked as a doubly linked list using the fields next_depth and parent_depth. The latter field is used for backtracking. The depth_entry field of the depth_node points to the first depth_entry_node in this depth. There is one depth_entry_node for each cell in this depth. This depth_entry_node is linked as a single linked list by the next_entry field. The corr_master_cell field points to the cell in this depth associated with the given depth_entry_node. The neigh_info_field points to the first neighbour's information of this master cell. Since we are working on the extended edge set, a cell is a neighbour of itself, hence, the first neighbour of the master cell is itself. The other neighbour cells are linked as a singly linked list via the other_neigh_info field. The corr_neigh_cell field points to its actual cell. The influ_func field is used to point to the influ_func_node which contains all the influence functions whose master cell is the cell pointed to by the corr_master_cell field in the depth_entry_node and the neighbour cell is pointed to by the corr_neigh_cell field in the neigh_info_node. In the influ_func_node, the sorted_edge_to_master field contains an ascending array of integers, according to the number of edges going from a vertex in the neighbour cell to the vertices in the master cell. The LIF_to_master and the UIF_to_master are two arrays of integers. They represent the lower influence function and the upper influence function respectively. The array_size field contains the size of the neighbour cell which is also the size of all the arrays in the influ_func_node.

3.2 Table Look-Up

While we are generating the information for a new depth, we often encounter the following operations:

1. Given two cells C_i and C_j, determine whether C_i is a neighbour of C_j.

2. Given a vertex v and a cell C, determine the number of distinct edges from v to C.

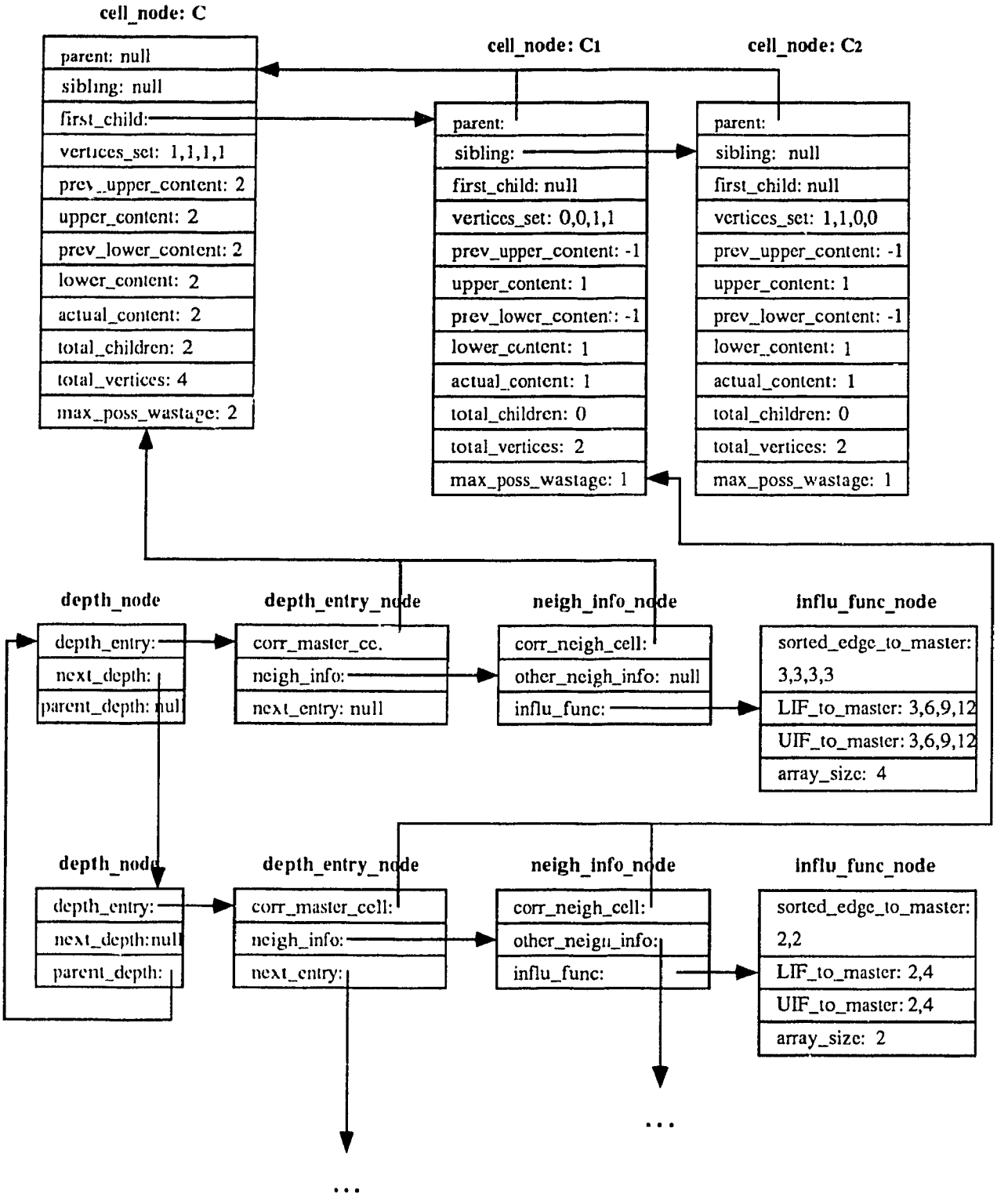

Figure 3.6: An overall structure between the cells and the depths.

In order to speed up these operations, we need one special representation for the graph and another one for the vertices in a cell. The representation of the vertices in a cell have already been explained on page 15. Before explaining the data structure of the graph, let us look at another definition. The *adjacency matrix* A of an undirected graph is a matrix with the following properties. If vertices i and j are connected, then $A[i,j] = A[j,i] = 1$, otherwise $A[i,j] = A[j,i] = 0$. Given a graph of r vertices, the graph will be represented by an adjacency matrix A of r rows, where each row i is represented by an array of computer words to indicate which vertices in the graph are connected to the vertex i. The vertices of the graph will be numbered from 0 to $r-1$. We use the notation $A[i,j]$ to indicate the i-th row and j-th bit of the adjacency matrix, whereas, $A[i]$ will mean the i-th row of the adjacency matrix. Notice that the representation of the vertices in a cell is exactly the same as the representation of a row in the adjacency matrix. Since we are working on the extended edge set, the following result should hold: $A[i,i] = 1, \forall i = 0$ to $r-1$.

The following algorithm will determine whether C_i is a neighbour of C_j.

An algorithm to determine the neighbourhood of two cells

for each vertex $u \in C_i$

  if (((*A*[*u*]) **and** (the vertex set of C_j)) $\neq \emptyset$)

    then return true;

return false;

The **and** operation in the above algorithm is a logical 'and' operation. To use the above algorithm, one has to find an efficient way to pick the vertices from a cell one at a time. One way to achieve this is to use a look-up table.

The method of table look-up uses an array where each element *table*[i] is a structure with two fields, the first field specifies the location of the first one (an 'on' bit) in the binary expansion of i, and the second field contain number of ones in this binary expansion. Let us number the right-most bit of a ord as position 0. To initialize this table, the first position of one can be determined by shifting each bit of the word one position to its right until position 0 is 'on' or if the left-most bit of

the binary expansion of i has been shifted to position 0. A similar idea can be use to find out the number of ones in the binary expansion of i. We call this array the *table*. The data structure of the table is as follows.

```
/* Data structure of the table. This
   table is used for table look-up. */

struct table_node
  {
     int first_position; /* the first position of the
                            word which is equal to one */
     int numberofones;   /* the number of
                            ones in the word */
  };
```

If a word is 32 bits long, then the maximum size of the table has 2^{32} entries. Because 2^{32} entries will require too much memory space, we only create a table with 2^{16} entries, indexed form 0 to $2^{16} - 1$. We will split a word into two parts, the *lower sixteen bits*, which are the bits from positions 0 to 15, representing the lower half of the word, and the *upper sixteen bits*, which are the bits from positions 16 to 31, representing the upper half of the word.

Using this look-up table, we can compute the number of ones in a word by two steps. The first step is to obtain the number of ones in the lower and the upper sixteen bits. The second step is to sum up the two totals. The number of ones in the lower sixteen bits can be obtained by setting the upper sixteen bits to zero and then performing a table look-up. The number of ones in the upper sixteen bits can be obtained by setting the lower sixteen bits to zero, shifting the upper half of the word to the lower half of the word, and then looking up the table.

To obtain the first position of one in the word, we will try to find the first position of one in the lower sixteen bits. If the lower sixteen bits are all equal to zero, then we try to obtain the first position of one in the upper sixteen bits.

By looking up the table, one can obtain all the vertices in a cell C by the following algorithm.

An algorithm to pick all the vertices in a cell
make a temporary copy of the vertex set of C;
while the temporary vertex set of $C \neq \emptyset$
    get the first position of one in the temporary vertex set of C;
    report this first position of one;
    set the first position of one in the temporary vertex set of C to zero;
discard the temporary copy of the vertex set of C;

Given a vertex u and a cell C, we can determine the number of distinct edges from u to C by doing a logical 'and' operation between $A[u]$ and the vertex set of C. Then based on this result, we use the table to look up the number of ones in it.

3.3 Assumption On Neighbourhood

Since this work deals with undirected graphs, we use this assumption while generating the neigh_info_node. When the program recognizes that C_j is a neighbour of C_i, then it will also assume that C_i is a neighbour of C_j. Hence, when generating the neighbour information of C_j, the program will only check those cells C_k, where $k \geq j$, and thus saving some computation time.

3.4 Partition Content

When a cell C is partitioned into a number of subcells $C_1, \ldots, C_n$, a new depth is created. The actual content of C is then partitioned amongst $C_1, \ldots, C_n$. Based on the lower and upper contents of each subcell, we generate all the possible partitions of $\sigma(C)$. The generation of all the possible partitions is summarized in the procedure **Generate**. It is a recursive procedure where each invocation **Generate** (C_i) is responsible for generating the content of the subcell C_i. Based on the Conservation

Rule, it uses a global variable *available content* to store the size of the content available for the cells $C_1 \ldots C_n$. Initially, *available content* is set to $\sigma(C)$. Before each recursion, the procedure will make sure two conditions are satisfied. The first condition is that the available content should be used up if all the remaining subcells are assigned their upper contents. The second condition is that the available content should be sufficient to satisfy the lower contents of the remaining cells. These two tests depend on computing the sums $\sum_{j>i} \overline{\sigma}(C_j)$ and $\sum_{j>i} \underline{\sigma}(C_j)$. For fast access, the summations, $\sum_{j>i} \overline{\sigma}(C_j)$ for $i = 1$ to n, are stored in an array of size n, called *upper limit distribution*, and the summations, $\sum_{j>i} \underline{\sigma}(C_j)$ for $i = 1$ to n, are stored in another array of size n, called *lower limit distribution*.

The content partition algorithm

procedure **Generate** (C_i)

$\sigma(C_i) \leftarrow \underline{\sigma}(C_i)$;

while $\sigma(C_i) \leq \overline{\sigma}(C_i)$ **do**

available content $\leftarrow$ (*available content* $- \sigma(C_i)$);

if ((*available content* $\leq (\sum_{j>i} \overline{\sigma}(C_j))$) **and**

(*available content* $\geq (\sum_{j>i} \underline{\sigma}(C_j))$)) **then**

if $i = n$ **then**

if *available content* $= 0$ **then**

report a good set of partition is obtained;

return;

else Generate (C_{i+1});

else return;

reset *available content*;

increase $\sigma(C_i)$ by 1;

Chapter 4

Results

Another important part of the program is how we partition a cell. Although the final result will be the same, a different method of partitioning may require a different amount of time. Since no method is the best for all graphs, we try to use a method that is good for most graphs. In the rest of this chapter, we will use the word 'partition' to means the partition of a cell.

We used two methods of partition. Before we discuss these two methods, let us discuss some of the common points about them. In both methods, a new depth $l+1$ is created by choosing a cell C at depth l and partitioning it into subcells. A cell will be chosen to be partitioned if and only if the total number of vertices in this cell is greater than one. Just before it is partitioned, we store away its current lower and upper contents. Later on, when the program backtracks to that depth l, we restore back the last lower and upper contents of cell the C. Another common point is the partitioning of a cell with zero content. Suppose we have a cell with r vertices, where $r > 1$, and zero content. When this cell is selected for partitioning, it will be partitioned into r subcells of one vertex each with zero content.

4.1 Partition By Half

The first method is to partition a cell C by half. If $size(C) = r$, then the first $\lfloor \frac{r}{2} \rfloor$ vertices in C will be in the first subcell, and the rest of the vertices in C will be in the second subcell.

4.2 Partition From A Reservoir

The second method is to partition the vertices from a reservoir. A *reservoir* is a cell with size greater than one. Its function is to supply suitable vertices for partitioning into different cells of size one. The basic idea of partitioning from a reservoir is to select some vertices from the reservoir based on certain constraints and then partition them into *discrete subcells.* A discrete subcell is a cell with size one. We define *maxsubcells* as a predefined value which is used to limit the maximum number of subcells that can be partitioned from the reservoir. In the beginning, the algorithm will look for any cell with size one which is not covered. If there exists such a cell C, the algorithm will get all its neighbour vertices from the reservoir. After removing all the neighbour vertices from the reservoir, each neighbour vertices will form a discrete subcell. In the case where there is no neighbour vertex of C in the reservoir, this means that C will not be covered when the program comes to a complete refinement, and the program will backtrack to previous stage. If there is no uncovered cell with size one, then at most *maxsubcells* vertices with minimum degree among all the vertices in the reservoir will be selected and partitioned into discrete subcells. It can be summarized by the following algorithm. Initially, the reservoir is the cell representing the input graph. Function **Mindegree**(C) will return the minimum degree among all the vertices in the cell C and function **Deg**(v) will return the degree of vertex v. The constraints used in this algorithm are to select at most *maxsubcells* vertices which have a minimum degree among all the vertices in the reservoir.

The partition from a reservoir algorithm

if $\exists\ C$ such that $((size(C) = 1) \wedge (\forall v(v \in C \wedge v$ is not covered$)))$ **then**
  if the reservoir contains some neighbour vertices of C **then**
    get and remove all neighbours of v form the reservoir;
    partition each neighbour of v as a discrete subcell;
  else
    report no solution in this partition of content;
    return;

else if reservoir exists **then**

totalsubcells $\leftarrow$ 0;

mindeg $\leftarrow$ **Mindegree** (reservoir);

while $((\exists v(v \in$ reservoir $\wedge$ **Deg** $(v) =$ *mindeg*$)) \wedge$

($totalsubcells < maxsubcells$)) **do**

partition v as a discrete subcell;

remove v from the reservoir;

increase *totalsubcells* by 1;

Let us clarify the idea of the algorithm by an example. Suppose the input graph is Figure 2.1, the test size is 3, and *maxsubcells* is 4. At the beginning, the reservoir is a cell C containing all the vertices in the input graph and it is the only cell in this depth. When the partition from a reservoir algorithm is applied, it partitions C into $C_1 = \{b, c, e, f, g, h, i, j, k, l, n, o\}$, $C_2 = \{a\}, C_3 = \{d\}, C_4 = \{m\}$, and $C_5 = \{p\}$. After the content of C is partitioned into its subcells, C_1 to C_5, if any of the the discrete subcells, say C_2, is not covered, the two neighbour vertices of a, namely b and e, will be partitioned from the reservoir C_1 as discrete subcells. The cells in this depth will be $C_1' = \{c, f, g, h, i, j, k, l, n, o\}$, C_2, C_3, C_4, C_5, $C_6 = \{b\}$, and $C_7 = \{e\}$. On the other hand, if all the discrete subcells C_2, C_3, C_4, and C_5 are covered, the first four vertices b, c, e, and h, with minimum degree among all the vertices in the reservoir C_1 will be partitioned as four discrete subcells. The cells in this depth will be $C_1' = \{f, g, i, j, k, l, n, o\}$, C_2, C_3, C_4, C_5, $C_6 = \{b\}$, $C_7 = \{c\}$, $C_8 = \{e\}$, and $C_9 = \{h\}$.

4.3 Output

The program is tested and run on a SUN 4/280s SPARC machine at Concordia University. Two kinds of graphs are tested. The first is the grid graph and the second is the knight's graph in an $m \times n$ chessboard.

4.3.1 Grid graph

An $m \times n$ *grid graph* has a vertex set $V = \{(i,j)|i = 1,\ldots,m,\ j = 1,\ldots,n\}$ and vertices (i,j) and (i',j') are adjacent if and only if they are consecutive on a row or column, that is to say, if $i = i'$ and $j = j' \pm 1$ or $i = i' \pm 1$ and $j = j'$. An example of a 4×4 grid graph is shown in Figure 2.1.

In [7], the authors gave a list of minimum domination numbers of some $m \times n$ grid graphs where $2 \leq n,m \leq 12$. In order to decide whether s is the minimum domination number for a graph G, we have to make sure that firstly $s-1$ is not a domination number of G and secondly s is a domination number of G. The CPU time required to decide whether s is the minimum domination number of G is the sum of the CPU time needed to finish running the program with the input graph G and test sizes s and $s-1$. Of course, in the case of test size s, we can stop once a dominating set of size s is found. It had taken the authors in [7] approximately 20 hours of CPU time using an IBM 3081 computer to decide that the minimum domination number of the 7×7 grid graph is 12. Besides the 7×7 grid graph, the authors did not mention timing results in other grid graphs.

Tables 4.1 and 4.2 show some of the timing results obtained when the partition by half algorithm and the partition from a reservoir algorithm are applied to different grid graphs. All the results obtained agree with those published in [7]. We confirm that the minimum domination number of the 8×8 grid graph is 16, whereas the authors in [7] can only give a lower and upper bounds for the minimum domination number of the 8×8 grid graph. In addition, we confirm that the minimum domination number of the 8×9 grid graph is greater than 17 and the minimum domination number of the 9×9 grid graph is greater than 19.

4.3.2 Knight's graph in an $m \times n$ chessboard

A knight's graph in an $m \times n$ chessboard, or a knight's graph in short, has a vertex set $V = \{(i,j)|i = 1,\ldots,m,\ j = 1,\ldots,n\}$ and vertices (i,j) and (i',j') are adjacent if and only if the square (i',j') of an $m \times n$ chessboard can be occupied in one move by a knight in the square (i,j).

m	n	*test size*	*CPU time (s)*	*number of solutions*	*stop at first solution*
1	72	24	9.883	1	no
5	5	7	46.417	22	no
7	7	11	147.717	0	no
7	7	12	1365.983	1	yes
8	8	14	270.050	0	no

Table 4.1: Timing results of the partition by half algorithm on different grid graphs.

m	n	*maxsubcells*	*test size*	*CPU time (s)*	*number of solutions*	*stop at first solution*
8	8	5	14	253.567	0	no
8	8	5	15	4924.467	0	no
8	8	5	16	1188.200	1	yes
8	8	6	14	241.367	0	no
8	8	7	14	195.617	0	no
8	8	8	14	222.067	0	no
8	9	7	17	57905.800	0	no
9	9	7	19	183588.267	0	no

Table 4.2: Timing results of the partition from a reservoir algorithm on different grid graphs.

m	n	*test size*	*CPU time (s)*	*number of solutions*	*stop at first solution*
3	11	8	13822.983	20	no
4	6	4	6.117	1	no

Table 4.3: Timing results of the partition by half algorithm on different knight's graphs in an $m \times n$ chessboard.

m	n	*marsubcells*	*test size*	*CPU time (s)*	*number of solutions*	*stop at first solution*
4	6	6	4	4.850	1	no
4	6	7	4	4.800	1	no
4	6	8	4	4.783	1	no
4	6	9	4	4.950	1	no
5	12	4	10	68555.850	4	no

Table 4.4: Timing results of the partition from a reservoir algorithm on different knight's graphs in an $m \times n$ chessboard.

In [14], the authors developed an algorithm to obtained the minimum domination number of some knight's graphs in the range where $3 \leq m \leq 10$ and $3 \leq n \leq 12$.

Tables 4.3 and 4.4 show some of the timing results obtained when the partition by half algorithm and the partition from a reservoir algorithm are applied to different knight's graphs. All the dominating sets we obtained agree with those published in [14]. Since the authors did not mention any timing results, we cannot do any timing comparison with them.

Chapter 5

CONCLUSIONS

The program is fully developed. When we apply the program to find the dominating sets or to confirm the minimum domination number of certain grid graphs and knight's graphs in a chessboard, the results obtained agree with those published in [7] and [14]. These results strongly indicates the correctness of the partition algorithm. Because we cannot find any software package that can solve the dominating set problem for graphs, we cannot make many timing comparisons. When we compare it with the only timing result of the 7×7 grid graph mentioned in [7], our program is significantly faster.

The program has its drawbacks. As the size of the graphs increases, the time required to obtain a solution becomes very long. In [17], the author stated that the computation of the domination number for graphs is an NP-complete problem. We tried to obtain the dominating set of a 9×9 grid graph with a test size of twenty. Using the method of partition from a reservoir, the program was still running after 200 minutes and no solution has been obtained.

The algorithm may be improved in several ways. Other methods of partitioning can be used. Currently, when a cell is partitioned, all the contents of its children will be defined. Maybe if we define some of the contents of the children later, the algorithm may still have enough information to decided whether a solution is possible. Finally, isomorph rejection can be added into the program to cut down the search tree.

Bibliography

[1] W. W. R. Ball. *Mathematical Recreations and Problems of Past and Present Times.* MacMillan, London, 1892.

[2] C. Berge. *Graphs and Hypergraphs.* North-Holland, Amsterdam, 1973.

[3] C. Berge. *Theory of Graphs and its Applications.* Methuen, London, 1962.

[4] R. G. Busacker, and T. L. Saaty. *Finite Graphs and Networks: An Introduction with Applications.* McGraw-Hill, New York, 1965.

[5] B. Carré. *Graphs and Networks.* Clarendon Press, Oxford, 1979.

[6] E. J. Cockayne, B. Gamble, and B. Shepherd. *An Upper Bound for the k-Domination Number of a Graph.* Journal of Graph Theory, vol. 9, 1985, pp. 533–534.

[7] E. J. Cockayne, E. O. Hare, S. T. Hedetniemi, and T. V. Wimer. *Bounds for the Domination Number of Grid Graphs.* Congressus Numerantium 47, 1985, pp. 217–228.

[8] E. J. Cockayne, and S. T. Hedetniemi. *Note on the Diagonal Queens' Domination Problem.* Journal of Combinatorial Theory, series A 42, 1986, pp. 137–139.

[9] E. J. Cockayne, and S. T. Hedetniemi. *Towards a Theory of Domination in Graphs.* Networks 7, 1977, pp. 247–261.

[10] C. F. deJaenisch. *Applications de l'Analyse Mathematique au Jeu des Echecs.* Petrograd, 1862.

[11] A. Farley, and S. T. Hedetniemi. *Broadcasting in Grid Graphs.* Proc. Ninth S. E. Conf. on Combinatorics, Graph Theory, and Computing, Utilitas Mathematica, Winnipeg, 1978, pp. 275–288.

[12] W. M. Gentleman. *Some Complexity Results for Matrix Computations on a Parallel Processor.* JACM, 25, 1987, pp. 112–115.

[13] F. Harary. *Graph Theory.* Addison-Wesley, U. S. A., 1969.

[14] E. O. Hare, and S. T. Hedetniemi. *A Linear Algorithm for Computing the Knight's Domination Number of a $K \times N$ Chessboard.* Clemson University, Technical Report, 87-MAY-1.

[15] S. M. Hedetniemi, and S. T. Hedetniemi. *A Survey of Gossiping and Broadcasting in Communication Network.* University of Oregon, Technical Report, CIS-TR-81-5.

[16] M. S. Jacobson and L. F. Kinch. *On the Domination of the Products of Graphs II: Trees.* Journal of Graph Theory, vol. 10, 1986, pp. 98–106.

[17] D. S. Johnson. *The NP-completeness Column: An Outgoing Guide.* Journal of Algorithms, 6, 1985, pp. 434–451.

[18] H. J. L. Kamps, and J H. van Lint. *A Covering Problem.* Colloquia mathematica societatis jános bolyai, 4, Combinatorial theory and its applications, Balatonfüred, Hungary, 1969.

[19] H. J. L. Kamps, and J. H. van Lint. *The Football Pool Problem for 5 Matches.* Journal of Combinatorial Theory, vol. 3, no. 4, Belgium, December 1967.

[20] A. L. Liestman. *Fault-tolerant Grid Broadcasting.* University of Illinois, Technical Report, no. UIUCDCS-R-80-1030, 1980.

[21] L. C. Liu. *Introduction to Combinatorial Mathematics.* Ch. 15, p. 235, McGraw-Hill, New York, 1968.

[22] C. W. Marshall. *Applied Graph Theory.* Wiley-Interscience, New York, 1971.

[23] W. McCuaig, and B. Shepherd. *Domination in Graphs with Minimum Degree Two.* Journal of Graph Theory, vol. 13, Dec., 1989, pp. 749–762.

[24] O. Ore. *Theory of Graphs.* America Mathematics Society Colloq. Publ., 38, Providence, 1962.

[25] Q. F. Stout. *Information Spread in Mesh-connected Computers.* SIAM Conf. on the Application of Discrete Mathematics, Troy, N. Y., June, 1981.

[26] A. Tucker. *Applied Combinatorics.* John Wiley and Sons, New York, 1980.

[27] F. L. VanScoy. *Broadcasting a Small Number of Messages in a Square Grid Graph.* Proc. Seventeenth Allerton Conf. on Communication, Control, and Computing, 1979.

[28] F. L. VanScoy. *Parallel Algorithms in Cellular Spaces.* Ph. D. Thesis, University of Virginia, 1976.

[29] T. H. M. Vo. *On the Domination Number of Grid Graphs.* Master Thesis, Concordia University, 1988.