In Solving the Dominating Set Problem: Group Theory Approach

Ka Leung Ma

A Thesis
in
The Department
of
Computer Science

Presented in Partial Fulfillment of the Requirements for
the Degree of Doctor of Philosophy at
Concordia University
Montréal, Québec, Canada

May 1998

0-612-40311-4

Abstract

Solving the Dominating Set Problem: A Group Theory Approach

Ka Leung Ma, Ph.D.
Concordia University 1998

This thesis presents a new way to find the dominating set of a graph by introducing the concept of an orbit graph and a weighted dominating set. We showed that the Blokhuis-Lam method for the football pool problem is a special case of assuming that the solution has a non-trivial automorphism group. A general purpose algorithm for solving the dominating set problem is developed by using a coverage test, a wastage test, and the orbit and block structure of the graph's symmetry group to prune the search. The algorithm has been implemented and the program can be used to find all the dominating sets of any undirected, loopless, and labeled graph or all the weighted dominating sets of an orbit graph.

The program analyzed grid graphs and football pool graphs, and was able to find the minimum dominating sets of any $m \times n$ grid graph for $m \leq 10$ and $n \leq 11$. For the football pool problem, we have determined that the automorphism group of n matches is $S_3 \wr S_n$ and the automorphism group of a rook domain graph $\Gamma_{n,q}$ is $S_q \wr S_n$. We also applied the orbit graph transformation to the football pool problem of 5, 6 and 7 matches. By considering only cyclic subgroups up to conjugacy, we generated 107 orbit graphs for the case of 5 matches, 220 orbit graphs for 6 matches and 428 orbit graphs for 7 matches. Using our program, we found dominating sets for 5 matches with size 27. For more matches, we found no better bounds within the limitations of our implementation. However, by using the mixed integer optimizer in the linear programming package CPLEX, we found weighted dominating sets, and hence, dominating sets matching the best known upper bounds. They also have more symmetry than the known solutions. There is hope that some of the unsolved orbit graphs can lead to smaller dominating sets.

Acknowledgments

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.

Throughout this research, Dr. Jean-Marie Bourjolly, Dr. G. Butler, at Concordia University, Montréal, Dr. Gene Cooperman at Northeastern University, Boston, and Dr. G.H.J. van Rees at University of Manitoba have given me a lot of assistance and invaluable advice. I would like to express my deepest thanks to them.

I would like to express my appreciation to the staffs of the Department of Computer Services, CICMA - Centre Interuniversitaire en Calcul Mathématique Algébrique, the Department of Computer Science at Concordia University, and the Centre de Recherche sur les Transports (CRT) group at Université de Montréal, for their permission to use their computer facilities. The version 2.0 of the CPLEX software used in this research belongs to CRT. The version 4.0.8 of the CPLEX software, MAGMA software, the *prof* software and the DECstation 5000 used in this research belong to CICMA. The ISOM package used in this research belongs to the Department of Computer Science at Concordia University.

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

At various stages of this research, a number of my friends like Rev. Thomas Chan, Dr. Wei Ping He, Mr. Leon Lau, Dr. Derek Pao, Dr. Gokul Chander Prabhakar, and many brothers and sisters in my church have given me encouragement, helpful suggestions, and different kinds of support. In this regard, I am ever so grateful to them for all their help.

This research was supported by a scholarship from the Natural Sciences and Engineering Research Council of Canada, a scholarship form FCAR, the Concordia University J. W. McConnell Memorial Fellowship, the CICMA Graduate Scholarship and several research grants from NSERC, FCAR and Loto Quebec.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis is on the dominating set problem. We start with a general purpose algorithm which finds a dominating set by recursively refining the vertex set. We first prune the search by using symmetry. We next develop a method of finding dominating sets by assuming the existence of a non-trivial automorphism group. These algorithms are tested using the grid graph and the football pool problem.

Being a general purpose algorithm, one of its advantages is that we can test it with different kind of graphs and find all their dominating sets. On the other hand, one of its disadvantage is its performance may not be as good as that of the algorithms that are tailored to a specific type of graph or algorithms that only search for the domination number without finding all the dominating sets. There is rarely any efficient general purpose program that finds all dominating sets of any graph. Finding all dominating sets of any graph can help us to understand more about a variety kinds of graphs. This is one of the reasons that motivates us to design one such program. We have developed a coverage test, a wastage test and an isomorph rejection to effectively search for a dominating set of a graph.

In this chapter, we give an introduction, some brief history and a survey of some previous work done in this area. Finally, we describe briefly the organization of this thesis and end with a summary of the research contribution.

1.1 The Dominating Set Problem

Definition 1.1 Graph *A graph $G(V, E)$ is a finite set V of* vertices *and a finite set E of edges such that $E \subseteq V \times V$.*

If the edge set E, when considered as a relation, is symmetric, then the graph $G(V, E)$ is said to be *undirected*; otherwise, it is *directed.* If $(v, v) \notin E$, for all $v \in V$, then G is *loopless.* If the vertices are labeled, typically by integers, but sometimes by letters, then $G(V, E)$ is called a *labeled* graph. In this thesis, the unqualified word "graph" means an undirected, loopless, and labeled graph with no multiple edges.

Definition 1.2 Extended Edge Set *Given a graph $G(V, E)$, the* extended edge set *of E, denoted by E', is $E \cup \{(u, u) : \forall u \in V\}$.*

Let $V = \{v_1, v_2, \ldots, v_n\}$ be the vertex set of a graph $G(V, E)$. Its *adjacency matrix* $M = (a_{ij})$ is defined by

$$a_{ij} = \begin{cases} 1 & \text{if } (v_i, v_j) \in E, and \\ 0 & \text{otherwise.} \end{cases}$$

Definition 1.3 Covering *A vertex v is said to be* covered *by a vertex u if and only if $(u, v) \in E'$. The term* dominated *and* covered *are synonyms.*

We call u a *neighbor* of v, if and only if $(u, v) \in E'$. The *neighborhood*, N_u, of a vertex u, is the subset of vertices adjacent to u. In other words,

$$N_u = \{v \in V \mid (u, v) \in E'\}.$$

We can now define the concept of domination in a graph. In the literature, this is sometimes called *total domination.*

Definition 1.4 Dominating Set *A graph, $G(V, E)$, is said to be covered by a set of vertices D, if and only if $\bigcup_{u \in D} N_u = V$. The set D is called a* dominating set *of G. The total number of vertices in D is called the* domination number *of G.*

The *minimum domination number* $\gamma(G)$ is the cardinality of the smallest D that covers G. The corresponding dominating set is called a *minimum dominating set.* The

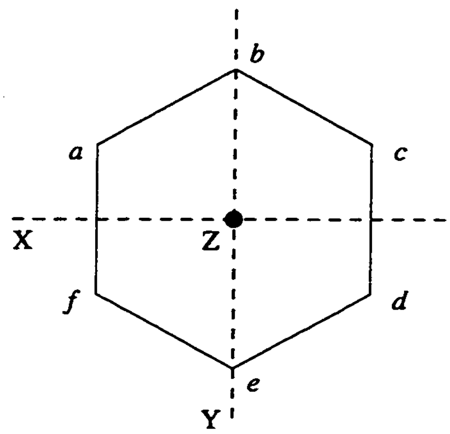


Figure 1: A regular hexagon.

dominating set problem is to find a minimum domination set of a graph. Depending on the graph, its minimum dominating set may not be unique. However, given a graph $G(V, E)$, the following statements are clearly true:

- one of its domination numbers is $|V|$, with V as the dominating set, and
- all its domination numbers are less than or equal to $|V|$.

Figure 1 shows an example of a regular hexagon. The minimum domination number for this graph is 2 and the largest domination number is 6. The three minimum dominating sets are $\{a,\ d\}$, $\{b,\ e\}$, and $\{c,\ f\}$.

1.2 History and Previous Work

The ideas of dominating set can be traced back to the origin of the game of chess in India. Questions concerning the optimum placement of chess pieces on a chessboard were first published in [5, 21]. Two earlier discussions can be found in [7, 55]. A more detailed discussion, which contains a quick review of results and applications concerning dominating sets in graphs, can be found in [20]. For an extensive survey of domination and its related problems, see Hedetniemi and Laskar's article in [34] which included also a comprehensive bibliography. Harary and Hanynes [30] had extended ordinary domination to conditional domination and presented a survey of the literature in which conditions are placed on the dominating set.

In 1965, Vizing [6, 73] established an upper bound for the minimum domination number $\gamma(G)$ of a graph $G(V, E)$. This bound is represented by the inequality $\gamma(G) \leq |V| + 1 - \sqrt{1 + 2|E|}$. During the next thirty years, several new inequalities for $\gamma(G)$ were derived [24, 49, 50, 51, 61]. Unfortunately, for the graphs of interest in this thesis, the upper bounds obtained from those inequalities are still far from their minimum domination numbers.

Some combinatorial problems, like the football pool problem [23, 37, 38, 75], can be solved by transforming the problem into a graph and then finding a dominating set for that graph. The football pool problem is discussed in detail in Chapter 6.

Different algorithms have been developed for special cases, such as for grid graph and for the knight's domination number of a $K \times N$ chessboard [17, 18, 32, 31]. For example, several subclasses of these graphs, such as the $1 \times n$ grid graph can be solved by a polynomial time algorithm. Unfortunately, finding the minimum dominating set of an arbitrary graph is NP-complete [25]. Since it is NP-complete, one approach is to use *approximate algorithms* to reduce the upper bound on the domination number [59]. In the last few years, several people have used *simulated annealing* [2, 1, 3, 38, 69] and obtained new upper bounds for the football pool problem [42, 58, 68, 76]. Another approach to solve the dominating set problem is the development of parallel algorithms. Certain graphs, after restricted to a subclass, can be solved effectively using this approach. Some examples that make use of the parallel algorithm approach are [8, 60, 64].

The importance of solving these combinatorial problems lies in their applications. For example, grid graphs can be used to model a variety of routing problems in street networks and also the interconnections in a multiprocessor VLSI system. For other applications of grid graphs, see [22, 26, 33, 45, 65, 71, 72, 74]. In another application [46], a graph is used to represent a communication network involving cities. A dominating set is a set of cities which, acting as transmitting stations, can transmit messages to every city in the network. Optimizing replication in a group of databases among several sites can also be modeled as a dominating set problem. For general theories concerning graphs and the dominating set problem, see [6, 11, 16, 29, 52, 67].

1.3 Our Previous Related Research

In [47, 48], we presented a partitioning algorithm to obtain all the dominating sets of a graph with a specified domination number t. The algorithm was motivated by [38] in which the authors used a similar method, to prove that 27 bets are required for the football pool problem with 5 matches. The algorithm recursively selects a group of vertices, called a *cell*, and refines it into smaller subcells. After the cell is refined into subcells, the algorithm partitions the domination number of the cell among its subcells. Two tests, namely the *coverage test* and the *wastage test*, are developed to cut down the search tree.

These tests restrict the possible partitions of a cell's domination number among its subcells. The two tests guarantee that those partitions which are not considered will never lead to a solution. Based on this algorithm, a program has been implemented. This program can find all the dominating sets of a given size t for a given graph. Using grid graphs as some of our test cases, the program could find the minimum dominating set of all the $m \times n$ grid graphs with $m, n \leq 8$. Moreover, we used the program to verify that the dominating sets in the knight's tour graphs on the 3×11, 4×6, and 5×12 chessboards, as published in [31], are indeed complete. Since this algorithm is the starting point of this thesis, it is described in detail in Chapter 2.

1.4 Organization of this Thesis

In this thesis, we present an improved partitioning algorithm by combining the coverage test, the wastage test, and *isomorph rejection* to cut down the computation required to obtain all the dominating sets of a graph. In addition to the algorithm, we present a new way to find dominating sets of a graph by introducing the concept of an orbit graph and a weighted dominating set.

The organization of this thesis is as follows. Chapter 2 discusses in detail the partitioning algorithm and its concepts. Chapter 3 describes the implementation details of the algorithm. Chapter 4 gives the results for selected grid graphs and summarizes the performance of this program on grid graphs. Chapter 5 presents the orbit graph and explains how to transform the dominating set problem of a graph into a weighted dominating set problem of its orbit graphs.

Chapter 6 shows the creation of football pool graph for the football pool problem, and reports the results of applying our program to the orbit graphs derived from football pool graphs. It also explains how we use a commercial linear programming package CPLEX to help us find weighted dominating sets on several football pool graphs. We found several solutions that have a non-trivial symmetry group and match the best known upper bounds. The chapter includes a proof that the size of the automorphism group of a rook domain graph $\Gamma_{n,q}$ is $(q!)^n n!$.

Chapter 7 compares the performance of the program using different methods to partition the grid graphs and the football pool graphs. Finally, Chapter 8 summarizes our work and concludes with suggestions for future research.

Appendix A gives a complete list of the orbit graphs of the football pool problem of 6 matches and the three solutions we found for the 7 matches.

Appendix B provides the source code of the program.

1.5 Research Contribution

The following list gives a summary of the research contribution in this thesis.

1. In the process of finding a dominating set, we designed a coverage test and a wastage test to trim the search tree. A rigorous proof of the coverage and wastage tests is presented in this thesis.
2. Isomorph rejection is incorporated into the algorithm.
3. We presented a partitioning method based on an orbit structure of a graph under the action of its automorphism group.
4. We determined the complete automorphism group of a rook domain graph $\Gamma_{n,q}$.

5. We showed that the Blokhuis-Lam method for the football pool problem is a special case of assuming that the solution has a non-trivial automorphism group. By assuming the solution has other automorphism groups of small order, we generated other graphs which may be useful for lowering the upper bounds. A list of such graphs for the football pool problem of 6 matches is given in Appendix A on page 99.

6. We have discovered several solutions matching the best known upper bounds of the football pool problem of 6 and 7 matches. Those solutions for the 7 matches we found have larger symmetry groups from the known ones.

Chapter 2

Partitioning Algorithm

This chapter outlines a partitioning algorithm to solve the dominating set problem. It also gives the necessary definitions and mathematical theories.

2.1 Basic Approach

The basic approach in this algorithm is to recursively refine the vertex set of a graph into subsets. We called a set of vertices a *cell.* The *size* of a cell C, denoted by $|C|$, is the number of vertices in that cell.

Definition 2.1 Refinement of a Cell *A* refinement of a cell C *is a decomposition of the cell into non-empty subcells* $C_1, \ldots, C_n$, *where no two of the subcells have common vertices and the union of all the subcells is* C.

When all the cells are refined into subcells of size 1, no more cells can be further refined, and the refinement is *complete.* Cells of size 1 are called *discrete cells.* In the following definitions, we define the relationship between cells and subcells.

Definition 2.2 Child *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* $C_1, \ldots, C_n$ *the* children *of* C.

Definition 2.3 Sibling *Two cells* C_i *and* C_j *are* siblings *of one another if and only if there exists a cell* C *such that* C *is the parent of both* C_i *and* C_j.

In the dominating set problem, a vertex is either in, or not in the dominating set. We say a vertex is *on*, if it is in the dominating set; otherwise it is said to be *off*.

Definition 2.4 Content *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. Obviously, $\sigma(C)$ satisfies the following condition:

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

Moreover, when a cell is refined, the total content of its subcells is equal to the content of the original cell. This leads to the following rule that relates the contents of a cell and its subcells:

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

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

Definition 2.5 Cell Refinement and Content Partition *There are two kinds of partitions involved in the partitioning algorithm. We call the partitioning of a cell the* cell refinement, *and we call the partitioning of the content the* content partition.

Our method of solving the domination set problem is to recursively refine the cells and partition their contents among their subcells until none of the cells can be refined further. This happens when the cells are all discrete. At this stage, those cells that have a content of one is equivalent to a vertex that is on. Prior to that stage, for each cell whose size is greater than one, we do not know which vertices in that cell are on. We only know the number of vertices in that cell that are on.

Using the above approach, we have developed a program to search for a dominating set of size t for a graph $G(V, E')$, where E' is the extended edge set of the graph $G(V, E)$. We call t the *test size* of $G(V, E)$. In order to find a minimum dominating set, if a dominating set of size t is found, t will be reduced by one and the program will be run again. The minimum domination number will be t for which the algorithm can find a dominating set of size t but not one of size $t - 1$.

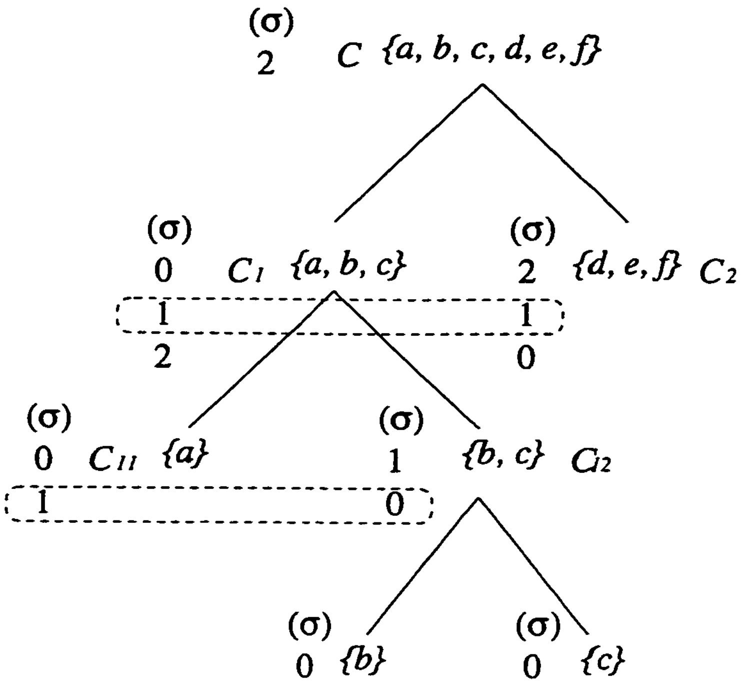


Figure 2: Refinement of a cell of size 6 with content 2.

Figure 2 shows an example of how we refine the cells and partition their contents to obtain a solution for the dominating set problem. The graph is the regular hexagon shown in Figure 1 on page 3. Initially, cell $C = \{a, b, c, d, e, f\}$ has six vertices and a content of two. At each level, a cell is selected and refined into two subcells of roughly equal size. After C is refined into $C_1 = \{a, b, c\}$ and $C_2 = \{d, e, f\}$, there are three ways to partition its content into its subcells. Suppose we select $\sigma(C_1) = 1$, and $\sigma(C_2) = 1$. Next we refine C_1 into $C_{11} = \{a\}$ and $C_{12} = \{b, c\}$. There are two ways to partition C_1's content of 1 to its children. Suppose we select $\sigma(C_{11}) = 1$, and $\sigma(C_{12}) = 0$. Since C_{11} has only one vertex in it and its content is one, this implies a is on. The cell C_{12} can be refined into two subcells. Since $C_{12} = 0$, both vertices b and c are off. The refinement of C_2 will be similar to C_1.

Algorithm 2.1 on page 11 gives an outline of the partitioning algorithm that is used to find the dominating set. The set of predetermined rules to refine a cell will be discussed in Section 2.2.6 on page 33.

In order to speed up the searching for a dominating set, we have to limit the number of possible choices for content partitions. In the next section, we describe

Algorithm 2.1 A Partitioning Algorithm to find the Dominating Set

---

Find_dominating_set ()
{ **if** $\exists$ a cell C with $|C| > 1$;
  { **refine** C into subcells $C_1, \ldots, C_k$ according to a set of predetermined rules;
    for each possible content partition of $\sigma(C)$ into $\sigma(C_1), \ldots, \sigma(C_k)$
      Find_dominating_set ();
  }
  else
    check if those vertices that are on form a dominating set;
}

---

how we make use of tests to reduce the number of content partitions.

2.2 Symmetry Partition

Three tests have been developed to limit the possible choices of content partition. The first two are the *coverage test* and the *wastage test* [47, 48]. They restrict the number of possible partitions of the parent's content among its children. The third test is *isomorph rejection*. It makes use of the orbit and block structures of the graph's automorphism group to reject a content partition if it is isomorphic to one that was generated earlier. The idea of refining a cell and partitioning its content was used in [38], and the coverage test is a generalized version of the method used there. These three tests have been incorporated into an algorithm called the *symmetry partitioning algorithm*. The following sub-sections describe these three tests in detail.

2.2.1 The Coverage Test

We first extend the definition of the neighbor of a vertex to the neighbor of a cell.

Definition 2.6 Neighbor *A cell C_i is said to be a* neighbor *of a cell C_j if there exists an edge (u, v) in E' where $u \in C_i$ and $v \in C_j$. We use $C_i \sim C_j$ to denote C_i is*

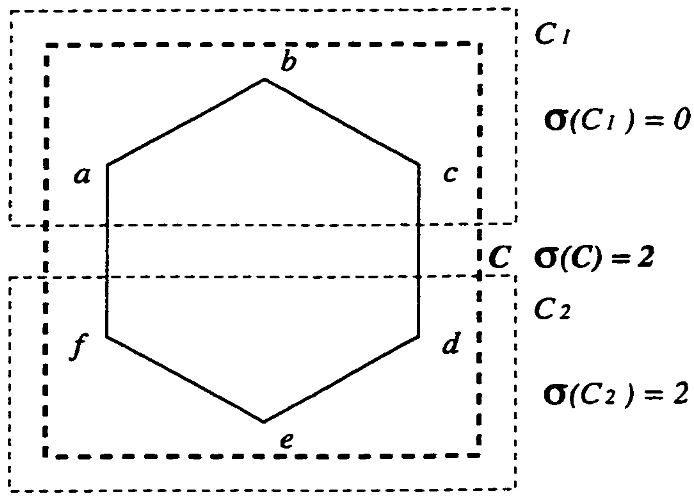


Figure 3: Refinement of a regular hexagon.

a neighbor of C_j and $C_i \not\sim C_j$ to denote C_i is not a neighbor of C_j.

Since we are using the extended edge set E', each cell C_i is a neighbor of itself. The coverage test can be informally stated as follows:

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

Let us consider an example using the regular hexagon in Figure 3. Let $C = \{a,\ b,\ c,\ d,\ e,\ f\}$ and let $\sigma(C) = 2$. Assume that the refinement of C is $C_1 = \{a,\ b,\ c\}$ and $C_2 = \{d,\ e,\ f\}$. Consider the content partition $\sigma(C_1) = 0$ and $\sigma(C_2) = 2$. There are only two vertices in C_2 that have edges connecting to C_1. The maximum number of vertices that can be covered in C_1 by the vertices in C_2 is 2. Moreover, as $\sigma(C_1) = 0$, there is no vertex in C_1 that can contribute to cover any vertex in itself. Since the size of C_1 is 3, C_1 cannot be covered with this content partition. Hence, $\sigma(C_1)$ has to be increased from 0 to 1. By the conservation rule, $\sigma(C_2) = 2 - \sigma(C_1)$, and hence $\sigma(C_2)$ has to be decreased from 2 to 1.

In order to compute the maximum number of vertices of a cell C_j that can be covered by another cell C_i, irrespective of which vertices of C_i are on, we define the upper influence function. Here, N_u is the neighborhood of vertex u.

Definition 2.7 Upper Influence Function *For each ordered pair* $\langle C_i, C_j \rangle$ *of cells, the* upper influence function *of* C_i *on* C_j *is* $U_{C_iC_j}$, *for* $0 \le m \le |C_i|$,

$$U_{C_iC_j}(m) = \max_{B \subseteq C_i, |B|=m} \left(\sum_{u \in B} |N_u \cap C_j| \right).$$

In other words, $U_{C_iC_j}(m)$ gives the upper bound on the number of edges connecting m of the vertices in C_i to vertices in C_j. The maximum coverage occurs when all the edges end on distinct vertices in C_j. For simplicity, we use $U_{ij}(m)$ instead of $U_{C_iC_j}(m)$ when it is clear that the subscripts i and j refer to cells C_i and C_j. Given $\sigma(C_i)$, the maximum number of vertices in C_j that can be covered by the vertices in C_i is $U_{ij}(\sigma(C_i))$. One way to compute $U_{ij}(m)$ is by sorting the vertices u of C_i in descending order according to the value of $|N_u \cap C_j|$. $U_{ij}(m)$ is then the sum of $|N_u \cap C_j|$ for the first m vertices u in this sorted list. This, however, gives only the upper bound on the number of vertices C_j that can be covered by C_i with $\sigma(C_i) = m$, because several edges may end at the same vertex in C_j.

Consider again the hexagon example in Figure 3 on page 12, where $C_1 = \{a, b, c\}$ and $C_2 = \{d, e, f\}$. The list of the vertices in C_1 sorted in descending order according to the number of edges in E' going from C_1 to C_1 is b, a, and c, where the corresponding edge counts are 3, 2 and 2. Thus, $U_{11}(2) = |N_b \cap C_1| + |N_a \cap C_1| = 3 + 2 = 5$. Using these data,

$$U_{11} = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 0 & 3 & 5 & 7 \end{bmatrix},$$

where the first row gives the values of m, and the second row gives the values of $U_{11}(m)$. As for U_{12}, we consider edges from C_1 to C_2. The sorted list of vertices is a, c and b with the corresponding edge counts being 1, 1 and 0. Thus

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

Note that even though $U_{11}(2)$ is 5, the actual number of vertices in C_1 covered by the vertices b and a of C_1 is 3. This is because the end point of the edge (a, a) is already covered by the edge (b, a), and the edge (a, b) also duplicated the coverage of (b, b).

In this example, one can verify that $U_{12} = U_{21}$. In general U_{ij} is not equal to U_{ji}. Consider the following choice of refining the hexagon. Choose $C_2 = \{d,\ e,\ f\}$, $C_3 = \{b\}$, and $C_4 = \{a,\ c\}$, then

$$U_{33} = \begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix},$$

$$U_{34} = \begin{bmatrix} 0 & 1 \\ 0 & 2 \end{bmatrix},$$

$$U_{32} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix},$$

$$U_{43} = \begin{bmatrix} 0 & 1 & 2 \\ 0 & 1 & 2 \end{bmatrix} = U_{42} = U_{44},$$

$$U_{23} = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 0 & 0 & 0 & 0 \end{bmatrix},$$

$$U_{24} = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 0 & 1 & 2 & 2 \end{bmatrix},$$

and

$$U_{22} = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 0 & 3 & 5 & 7 \end{bmatrix}.$$

Notice that the largest m in $U_{34}(m)$ is 1 and the largest m in $U_{43}(m)$ is 2. This is because the largest m in $U_{ij}(m)$ is equal to $|C_i|$. When $C_i \not\sim C_j$, U_{ij} becomes a zero function as in U_{23} and U_{32}.

The informal statement of the coverage test on page 11 can now be stated as:

Theorem 2.1 *For each cell C_j, a necessary condition for it to be covered is*

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

Proof: By definition, $U_{ij}(\sigma(C_i))$ gives the maximum number of vertices in C_j that can be covered by the vertices in C_i with content $\sigma(C_i)$. Hence, $\sum_{C_i \sim C_j} U_{ij}(\sigma(C_i))$ gives the maximum number of vertices that can be covered in C_j. Therefore, in order for C_j to be covered, the sum has to be greater than or equal to the number of vertices in C_j. $\square$

After each cell refinement, the parent's content has to be distributed to its children according to the conservation rule. However, the number of possible content partitions to be examined can be huge. For each child C_k, its content depends on its size, the size of its parent and the conservation rule. Initially, the lower bound of its content is 0 and the upper bound of its content is the minimum of $|C_k|$ and the content of C_k's parent. If we can further improve these two bounds, the number of possible content partitions can be reduced. However, Theorem 2.1 cannot be applied directly, because we do not know exactly the contents of the children. We know only the upper and lower bounds for their contents. We need to restate Theorem 2.1 in a form which takes into account that some contents are not exactly known. For this purpose, we define first the S-slack of a cell.

We use $\overline{\sigma}(C)$ and $\underline{\sigma}(C)$, respectively, to denote the upper bound and lower bound of the content of a cell C. For simplicity in names, we call $\overline{\sigma}(C)$ the *upper content* of C and $\underline{\sigma}(C)$ the *lower content* of C. Obviously, $\underline{\sigma}(C) \leq \sigma(C) \leq \overline{\sigma}(C)$.

Definition 2.8 S-slack *We define S_{kj}, the* S-slack *of a cell C_k relative to a cell C_j, to be*

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

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 restated in the following theorem.

Theorem 2.2 *For each cell C_j and for each neighbor cell C_k of C_j, in order for C_j to be covered, the following inequality must hold:*

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

Proof: In order to cover C_j, the best case occurs when each edge from the neighbors of C_j covers a different vertex in C_j. If all the neighbors except C_k have their 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 obtain the following inequality,

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

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

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

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.

Suppose after a refinement, there are q subcells in total. According to the coverage test in Theorem 2.2, for each fixed j, these q subcells $C_1, \ldots, C_q$ will lead to the following inequalities: $\left[\sum_{C_i \sim C_j} U_{ij}(\overline{\sigma}(C_i))\right] - |C_j| \geq S_{kj}$, for $k = 1$ to q and $C_k \sim C_j$. In the beginning, if $\sigma(C_k)$ is known, both $\underline{\sigma}(C_k)$ and $\overline{\sigma}(C_k)$ are initialized to $\sigma(C_k)$; otherwise, $\underline{\sigma}(C_k)$ is initialized to 0 and $\overline{\sigma}(C_k)$ is initialized to the minimum of $|C_k|$ and the content of C_k's parent. After applying the coverage test, if any of the inequalities are not satisfied, the corresponding S-slack, S_{kj}, will be reduced by increasing $\underline{\sigma}(C_k)$. If this increase leads to a $\underline{\sigma}(C_k)$ which is greater than $\overline{\sigma}(C_k)$, then the vertices in C_j cannot be covered, and further refinement will not lead to any solution. The partitioning algorithm will backtrack. The coverage test is summarized in Algorithm 2.2 on page 24.

2.2.2 The Wastage Test

The notion of *wastage* is to measure the amount of excess coverage of the vertices in a cell. For example, if a cell has m vertices and the total number of vertices that can be covered by its neighbors is n, and if $n \geq m$, then the wastage is $n - m$. If $n < m$, then there is insufficient coverage and the wastage is defined as zero rather than the negative value of $n - m$. Let us first describe the basic idea behind the wastage test.

"Given a cell C, we first measure its maximum amount of wastage. When C is refined into a number of subcells, the sum of the minimum amount of wastage for C's children subcells should not be greater than the maximum amount of wastage of C."

Consider again the hexagon example in Figure 3 on page 12. Once again, $C = \{a,\ b,\ c,\ d,\ e,\ f\}$, and $\sigma(C) = 2$. The maximum amount of wastage in C is equal to the maximum amount of coverage in C minus $|C|$. The maximum amount of coverage in C is $U_{CC}(\sigma(C)) = U_{CC}(2) = 6$, because each vertex in C has degree 3 in the extended edge set. Since $|C| = 6$, the maximum amount of wastage in C is $U_{CC}(\sigma(C)) - |C| = 0$.

Consider the refinement of C into $C_1 = \{a,\ b,\ c\}$ and $C_2 = \{d,\ e,\ f\}$ with the content partition $\sigma(C_1) = 0$ and $\sigma(C_2) = 2$. Let us compute the minimum coverage on C_2. Since $\sigma(C_1) = 0$, C_1's contribution to the coverage of C_2 is 0. As for vertices in C_2, vertex e covers all 3 vertices of C_2. Vertices d and f each covers only 2 vertices of C_2. The minimum coverage of C_2 by 2 vertices of C_2 is then $2 + 2 = 4$, by using vertices d and f. The minimum wastage is then $4 - |C_2| = 4 - 3 = 1$. Since C_2's parent has a maximum wastage of 0, it is impossible for C_2 to have a minimum wastage of 1.

It is instructive to compute also the minimum wastage of C_1. The coverage of C_1 due to vertices in C_1 is 0, because $\sigma(C_1) = 0$. As for vertices d, e, and f in C_2, they cover 1, 0, and 1 vertices in C_1, respectively. The minimum coverage of C_1 by 2 vertices in C_2 is $0 + 1 = 1$. If we had defined wastage simply as coverage minus cell size, then we would have obtained a minimum wastage of $1 - |C_1| = -2$. Adding this wastage to that of C_2 gives $(-2) + 1 = -1$, which would be smaller than the wastage of its parent and would not have led to a contradiction. To avoid this situation, we insist that wastage cannot be a negative quantity, which is the same as insisting that we cannot leave some vertices uncovered in order to pay for the wasted coverage elsewhere.

The maximum amount of wastage of a cell is called the *maximum allowable wastage* of a cell and the minimum amount of wastage of a cell is called the *unavoidable wastage* of a cell. Mathematically, the maximum allowable wastage of a cell C_j, $W(C_j)$, is defined as

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

The summation $\sum_{C_i \sim C_j} U_{ij}(\sigma(C_i))$ represents the *maximum possible coverage* for the cell C_j. Hence after subtracting the size of C_j, we get the maximum allowable wastage of C_j. In order to compute the maximum allowable wastage of each cell, we have to reach a stage where the content of each cell is defined.

To compute the unavoidable wastage of each cell, we need to define another function that states clearly the minimum coverage between one cell and the other.

Definition 2.9 Lower Influence Function *For every ordered pair of cells $\langle C_i, C_j \rangle$, the* lower influence function *of C_i on C_j is $L_{C_i C_j}$, where $0 \le m \le |C_i|$, and*

$$L_{C_i C_j}(m) = \min_{B \subseteq C_i, |B|=m} \left(\sum_{u \in B} |N_u \cap C_j| \right).$$

If there is no ambiguity in identifying the cells C_i and C_j, for simplicity, we denote the lower influence function as $L_{ij}(m)$. Given $\sigma(C_i)$, the minimum number of edges connecting $\sigma(C_i)$ vertices in C_i to C_j is $L_{ij}(\sigma(C_i))$. To compute $L_{ij}(m)$, the vertices u of C_i are first sorted in ascending order according to $|N_u \cap C_j|$. $L_{ij}(m)$ is then obtained by summing over the first m vertices in this sorted list.

Consider again the hexagon example in Figure 3 on page 12, where $C_1 = \{a, b, c\}$ and $C_2 = \{d, e, f\}$. The list of the vertices in C_1 sorted in ascending order according to the number of edges in E' going from C_1 to C_1 is a, c, and b, where the corresponding edge counts are 2, 2 and 3. Thus, $L_{11}(2) = |N_a \cap C_1| + |N_c \cap C_1| = 2+2 = 4$. Using these data,

$$L_{11} = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 0 & 2 & 4 & 7 \end{bmatrix}.$$

As for L_{12}, we consider edges from C_1 to C_2. The sorted list of vertices is b, a and c with the corresponding edge counts being 0, 1 and 1. Thus

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

The following are some properties of the two influence functions. For all i, j :

1. $L_{ij}(m) \leq U_{ij}(m)$ for $m = 0,\ 1, \ldots,\ |C_i|$,
2. $L_{ij}(|C_i|) = U_{ij}(|C_i|)$,
3. $L_{ij}(0) = U_{ij}(0) = 0$, and
4. $C_i \not\to C_j$ if and only if $L_{ij}(m) = U_{ij}(m) = 0$, for $m = 0,\ 1, \ldots,\ |C_i|$.

In our hexagon example, choosing $C_2 = \{d,\ e,\ f\}$, $C_3 = \{b\}$, and $C_4 = \{a,\ c\}$, as $C_2 \not\to C_3$, hence both the lower and upper influence functions U_{23} and L_{23} are zero functions.

The wastage test depends on the relationship between the maximum allowable wastage of a cell C and the unavoidable wastage of C's children. Let us state some of these relationship in a lemma.

Lemma 2.3 *Let C be refined into $C_1, \ldots,\ C_n$ and let B be any cell where $0 \leq m \leq |B|$. Then*

$$U_{BC}(m) \geq \sum_{j=1}^{n} L_{BC_j}(m).$$

Proof: Let $\{v_1, \ldots,\ v_m\}$ be the set of m vertices in B which have the largest number of edges from B to C. Let N_i be the neighborhood of v_i in E'. Then, by definition

$$U_{BC}(m) = \sum_{i=1}^{m} |N_i \cap C|.$$

Since $C_1, \ldots,\ C_n$ is a refinement of C, we also have

$$U_{BC}(m) = \sum_{i=1}^{m}\sum_{j=1}^{n} |N_i \cap C_j| = \sum_{j=1}^{n}\sum_{i=1}^{m} |N_i \cap C_j|. \tag{1}$$

By the definition of $L_{BC_j}(m)$, we have

$$\sum_{i=1}^{m} |N_i \cap C_j| \geq L_{BC_j}(m).$$

Therefore, (1) implies

$$U_{BC}(m) \geq \sum_{j=1}^{n} L_{BC_j}(m). \ \square$$

Let us consider the special case where $B = C$. We start with the following lemma.

Lemma 2.4 *Let C be refined into $C_1, \ldots, C_n$ with contents $\sigma(C_1), \ldots, \sigma(C_n)$, where $\sum_{i=1}^{n} \sigma(C_i) \leq \sigma(C)$. Then*

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

Proof: Let $\{v_1, \ldots, v_{\sigma(C_1)}\}$ be the set of $\sigma(C_1)$ vertices of C_1 which have the largest number of edges from C_1 to C. Similarly, let $\{v_{\sigma(C_1)+1}, v_{\sigma(C_1)+2}, \ldots, v_{\sigma(C_1)+\sigma(C_2)}\}$ be the set of $\sigma(C_2)$ vertices of C_2 with the largest number of edges from C_2 to C; and so on. Let N_j be the neighbors of v_j and let $\tau(i) = \sum_{j=1}^{i} \sigma(C_j)$. With the convention that the vacuous sum $\tau(0) = 0$, we have, for $i = 1, \ldots, n$,

$$U_{C_iC}(\sigma(C_i)) = \sum_{t=1+\tau(i-1)}^{\tau(i)} |N_t \cap C|.$$

By letting $\tau(n) = s$, we have

$$\sum_{i=1}^{n} U_{C_iC}(\sigma(C_i)) = \sum_{i=1}^{n} \sum_{t=1+\tau(i-1)}^{\tau(i)} |N_t \cap C| = \sum_{t=1}^{s} |N_t \cap C|. \tag{2}$$

From the assumption of the Lemma, $s \leq \sigma(C)$. Since U_{CC} is monotonic non-decreasing

$$U_{CC}(\sigma(C)) \geq U_{CC}(s).$$

By the definition of U_{CC},

$$U_{CC}(s) \geq \sum_{t=1}^{s} |N_t \cap C|,$$

for any set of $\{v_1, \ldots, v_s\}$ of vertices. Hence,

$$U_{CC}(\sigma(C)) \geq \sum_{t=1}^{s} |N_t \cap C|.$$

Using (2), we get

$$U_{CC}(\sigma(C)) \geq \sum_{i=1}^{n} U_{C_iC}(\sigma(C_i)). \ \square$$

By applying Lemma 2.3 to the term $U_{C_iC}(\sigma(C_i))$ in Lemma 2.4, we have the following Corollary.

Corollary 2.5

$$U_{CC}(\sigma(C)) \geq \sum_{i=1}^{n}\sum_{j=1}^{n} L_{C_iC_j}(\sigma(C_i)), \; where \; \sum_{i=1}^{n}\sigma(C_i) \leq \sigma(C).$$

Using the lower influence function, the unavoidable wastage can be defined mathematically in the following way:

Definition 2.10 Unavoidable Wastage *The* unavoidable wastage *of a cell C_j, denoted by $\underline{W}(C_j)$, is defined as:*

$$\underline{W}(C_j) = max \begin{cases} 0 \\ \left[\sum_{B\sim C_j} L_{BC_j}(\sigma(B))\right] - |C_j|. \end{cases}$$

The term $\sum_{B\sim C_j} L_{BC_j}(\sigma(B))$ gives the minimum coverage C_j can get from all its neighbors. Hence, if $\left[\sum_{B\sim C_j} L_{BC_j}(\sigma(B))\right] - |C_j|$ is greater than or equal to zero, it will give the unavoidable wastage or excessive coverage of the cell C_j. Since $\left[\sum_{B\sim C_j} L_{BC_j}(\sigma(B))\right] - |C_j|$ may be negative, which means no excessive coverage, we take the maximum between this value and zero.

Let us use $C_j \dashv C$ to denote C_j is a child of C. The following theorem gives the relationship between the maximum allowable wastage and the unavoidable wastage.

Theorem 2.6 *If the contents of the children $C_1, \ldots, C_n$ of C satisfies $\sum_{i=1}^{n}\sigma(C_i) \leq \sigma(C)$, then*

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

Proof: By definition,

$$W(C) = \sum_{B\sim C} U_{BC}(\sigma(B)) - |C|.$$

If $B \neq C$, Lemma 2.3 states

$$U_{BC}(\sigma(B)) \geq \sum_{j=1}^{n} L_{BC_j}(\sigma(B)).$$

If $B = C$, then it is refined into $C_1, \ldots, C_n$ with contents $\sigma(C_1), \ldots, \sigma(C_n)$. Corollary 2.5 implies

$$U_{CC}(\sigma(C)) \geq \sum_{i=1}^{n}\sum_{j=1}^{n} L_{C_iC_j}(\sigma(C_i)).$$

Therefore

$$\sum_{B \sim C} U_{BC}(\sigma(B)) \geq \sum_{j=1}^{n} \sum_{B \sim C_j} L_{BC_j}(\sigma(B)),$$

where the set of cells B on the L.H.S. corresponds to the cells before the refinement of C and the set of cells B on the R.H.S. are for after the refinement. Subtracting $|C|$ from both sides and noting that $\sum_{j=1}^{n} |C_j| = |C|$, we have

$$\sum_{B \sim C} U_{BC}(\sigma(B)) - |C| \geq$$

$$\sum_{j=1}^{n} [\sum_{B \sim C_j} L_{BC_j}(\sigma(B)) - |C_j|].$$

Since $\sum_{B \sim C_j} U_{BC_j}(\sigma(B)) - |C_j|$ measures coverage of C_j, it must be non-negative in order for C_j to be covered. Moreover, $\sum_{B \sim C_j} U_{BC_j}(\sigma(B)) - |C_j| \geq \sum_{B \sim C_j} L_{BC_j}(\sigma(B)) - |C_j|$ and $\sum_{j=1}^{n} [\sum_{B \sim C_j} U_{BC_j}(\sigma(B)) - |C_j|] = \sum_{B \sim C} U_{BC}(\sigma(B)) - |C|$, therefore

$$\sum_{B \sim C} U_{BC}(\sigma(B)) - |C| \geq \sum_{j=1}^{n} \underline{W}(C_j)$$

or

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

The wastage test as stated in Theorem 2.6 is for the case when the contents of the children of C are known. However, it is more efficient to avoid generating the content partitions which will fail the wastage test. As is with the case of the coverage test, we use the wastage test to reduce the gap between the lower and upper bounds for the contents of the children cells.

As an analog to the S-slack, we define a Z-slack.

Definition 2.11 Z-slack *The* Z-slack, Z_{kj}, *of a cell* C_k *relative to a cell* C_j, *is:*

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

We shall reduce the upper content of a cell C_k by computing the unavoidable wastage using the lower content of all the cells, except C_k. For C_k, we use its upper content. Thus, the unavoidable wastage of C_j, which now also depends on k, is denoted by

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

The wastage test condition can now be restated as:

Theorem 2.7 *For each cell C_k,*

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

Proof: We note first that the upper and lower bounds are maintained to satisfy the Conservation Rule. If $C_k \dashv C$, then

$$\overline{\sigma}(C_k) + \sum_{C_i \dashv C,\ i \neq k} \underline{\sigma}(C_i) \leq \sigma(C).$$

If $C_k \not\dashv C$, then

$$\sum_{C_i \dashv C} \underline{\sigma}(C_i) \leq \sigma(C).$$

Therefore, the hypothesis of Theorem 2.6 is satisfied and it implies Theorem 2.7. □

Suppose we are at a stage in the algorithm where a cell has just been refined, and there are q subcells in total. Theorem 2.7 leads to q inequalities, where k varies from 1 to q. 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 $|C_k|$ and $\sigma(C)$, where $C_k \dashv C$. After applying the wastage test, if the inequality $W(C) \geq \sum_{C_j \dashv C} \underline{W_k}'(C_j)$ 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 $\underline{\sigma}(C_k)$, the algorithm would backtrack to its previous stage. Algorithm 2.3 on page 24 shows an algorithm for the wastage test.

2.2.3 Isomorph Rejection

When there are several ways to partition the content of a cell to its subcells, some of these ways may be isomorphic to one another. By making use of the symmetry group

Algorithm 2.2 **The coverage test algorithm**

---

for each C_j
{ **for each** $C_k \sim C_j$
{ *exit* $\leftarrow$ false;
repeat { **if** $\left\lceil \sum_{C_i \sim C_j} U_{ij}(\overline{\sigma}(C_i)) \right\rceil - |C_j| < S_{kj}$ **then**
if $\overline{\sigma}(C_k) > \underline{\sigma}(C_k)$ **then**
increase $\underline{\sigma}(C_k)$ by 1;
else
{ **report** no solution in this content partition;
return;
}
else
exit $\leftarrow$ true;
} **until** *exit*;
}
}

Algorithm 2.3 **The wastage test algorithm**

---

for each k
{ **repeat** { *pass* $\leftarrow$ true;
if $W(C) < \sum_{C_j \dashv C} \underline{W_k}'(C_j)$
then if $\overline{\sigma}(C_k) > \underline{\sigma}(C_k)$
then
{ **decrease** $\overline{\sigma}(C_k)$ by 1;
pass $\leftarrow$ false;
}
else
{ **report** no solution in this content partition;
return;
}
} **until** *pass*;
}

---

of the graph, we can eliminate some of these "redundant" content partitions and hence reduce the search time for a dominating set. Combining this concept with the coverage and wastage tests, we come up with the *symmetry partitioning algorithm.*

We need some group theory terminology to describe the symmetry partitioning algorithm.

Definition 2.12 Group *A* group *is a set* Gp *together with a binary operation* $*$ *called multiplication on* Gp *which satisfies four axioms:*

1. Gp *is closed under* $*$. *In other words,* $\forall g, h \in Gp,\ g * h \in Gp$.
2. *Multiplication is associative. That is* $(g * h) * k = g * (h * k)$ *for any three (not necessary distinct) elements from* Gp.
3. *There is an element* $id \in Gp$*, called the identity element, such that* $g * id = g = id * g$ *for every* g *in* Gp.
4. *Each element* g *of* Gp *has a inverse* g^{-1} *which belongs to the set* Gp *and satisfies* $g^{-1} * g = id = g * g^{-1}$.

The number of elements in a group is called the *order* of a group. We work only with groups of finite order. We usually write $|Gp|$ for the order of the group Gp. A group that contains only the identity element id is called the *identity group.*

The symmetries of the regular hexagon in Figure 1 on page 3 forms a finite group with twelve elements. There are five non-trivial rotations (through $\pi/3$, $2\pi/3$, π, $4\pi/3$, and $5\pi/3$) about Z, the axis perpendicular to the hexagon. In addition, there are three axes of symmetry determined by pairs of opposite corners, e.g. Y, and three others determined by the midpoints of pairs of opposite sides, e.g. X. Including the identity, these are the twelve elements in the group. This group is a special case of a class of group called the *Dihedral Groups.* In our case, the group is D_6. We denote the dihedral group formed by a regular polygon of n equal sizes as D_n.

Definition 2.13 Subgroup *A* subgroup *of a group* Gp *is a subset of* Gp *which forms a group under the same binary operation* $*$.

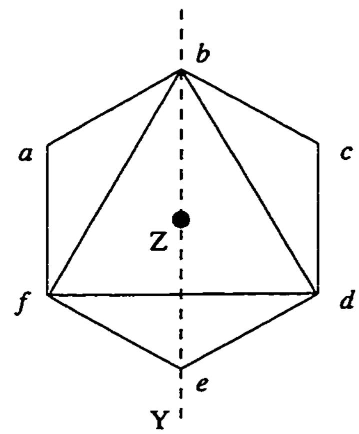


Figure 4: A triangle inside a regular hexagon.

The symmetry group D_3 of a regular triangle is a subgroup of D_6 (Figure 4). It has six symmetries. There are two rotations (through $2\pi/3, 4\pi/3$) about Z. In addition, there are three axes of symmetry determined by the corners and the midpoints of their opposite sides, e.g. Y. Including the identity, these are the six elements in the subgroup.

Definition 2.14 Set of Generators *Let S be a subset of a finite group Gp. The set S generates Gp, denoted by $Gp =< S >$, if every element of Gp can be written as a product $g = s_{i_1} * s_{i_2} * \cdots * s_{i_m}$ of elements in S, for some m. We call S a* set of generators *for Gp.*

Let r be the rotation of $2\pi/3$ about Z in Figure 4 and s be the reflection of π about Y. One can easily show $D_3 = \{id,\ r,\ r^2,\ s,\ rs,\ r^2s\} = < \{r,\ s\} >$.

Definition 2.15 Cyclic Group *If there is an element g in Gp which generates all of Gp, in other word, for which $< \{g\} >= Gp$, we say that Gp is a* cyclic group.

Definition 2.16 Permutation *A* permutation *of a set Ω is a bijection from Ω to itself.*

Definition 2.17 Symmetric Group *The collection of all permutations of Ω forms a group S_Ω under the composition of functions. If $|\Omega| = n$, S_Ω is called the* symmetric group *of degree n.*

When Ω is the set of the first n positive integers, then S_Ω is usually written as S_n. The order of S_n is $n!$.

A permutation can be represented by listing the image of each element. It is symbolized by a $2 \times |\Omega|$ matrix with the upper row containing the elements in Ω and the lower row containing the images of the elements in the upper row. We call this representation the *image form*. Alternatively, a permutation can be specified by listing its cycles. Within a cycle, the smallest element is written in the first position. The entry in position i is the image of the entry in position $i-1$ and the first entry is the image of the last entry in the cycle. Usually, the cycles of length one are omitted. This is called the *cycle form* of the permutation. For example, the image form and its equivalent cycle form of the permutation $1 \rightarrow 2,\ 2 \rightarrow 3,\ 3 \rightarrow 1,\ 4 \rightarrow 4$ are

$$\begin{bmatrix} 1\,2\,3\,4 \\ 2\,3\,1\,4 \end{bmatrix} \text{ and } (1,\ 2,\ 3).$$

2.2.4 Symmetry Groups and Combinational Objects

In this section, we shall establish the relationship between permutations and combinatorial objects. We define the *general search problem* as follows:

Definition 2.18 General Search Problem *Given a collection of sets of candidates $X_1, X_2, \ldots, X_m$, and a boolean predicate function $\mathcal{P}$ defined on the Cartesian product $X_1 \times X_2 \times \cdots \times X_m$, the* general search problem *is to find an m-tuple $(x_1, \ldots x_m)$ such that $\mathcal{P}((x_1, \ldots x_m))$ is true.*

We define a *combinatorial object* as an m-tuple. In this context, a predicate is often called a *property* and we say that a combinatorial object is defined by a property $\mathcal{P}$. We use the letters A and B to denote combinatorial objects. For example, a graph is a special case of a combinatorial object. One possible way to define an undirected, and labeled graphs with k vertices is to treat the adjacency matrix M as a k^2-tuple and to define the property $\mathcal{P}(M)$ as true if and only if M is symmetrical along its diagonal. Another possibility is to consider only the entries in the upper triangle of the adjacency matrix and treat it as a $k(k-1)/2$-tuple with $\mathcal{P}(M)$ as always true.

In order to discuss the isomorphism of combinatorial objects, there must be some notion of permutations acting on them. A permutation $p \in S_{X_1 \times X_2 \times \ldots \times X_m}$, is *property preserving* if $\mathcal{P}(p(x_1, \ldots x_m)) = \mathcal{P}((x_1, \ldots x_m))$ for all m-tuples $(x_1, \ldots x_m)$. A property preserving permutation is also called a *symmetry*. These permutations however, are very large and it is practically impossible to compute with them. Take for example graphs with 4 vertices defined as 6-tuples by using entries in the upper triangles of their adjacency matrices. Each entry is either a 0 or a 1. There are $2^6 = 64$ 6-tuples and the permutations are of degree 64. If possible, we work with permutations defined over a smaller Ω, provided that these smaller permutations induces a well defined action on the m-tuples.

Some common simpler symmetries are those that permute the m subscripts and those that permute the elements within the candidate sets X_i. For graphs, we can choose $\Omega = V$. The permutation p acts by specifying that the vertex labeled i in the graph $G(V, E)$ is to be relabeled as $p(i)$ in the graph $G(p(V), p(E))$. In terms of the adjacency matrix $M = (a_{ij})$, the action of the permutation p is to take a_{ij} to $a_{p(i)p(j)}$. Consider the example of the graphs with 4 vertices again. The permutations of the vertices are of degree 4 only. Moreover, their actions are more natural and related to the problem.

Definition 2.19 Symmetry Group *The group formed by the set of the symmetries is called the* property preserving group *or the* symmetry group *and is denoted by* $\mathcal{S}(\mathcal{P})$. *In other words,*

$$\mathcal{S}(\mathcal{P}) = \{p \in S_\Omega \mid \mathcal{P}(p(A)) = \mathcal{P}(A)\ \forall \textit{ combinatorial objects } A\}.$$

Moreover, two combinatorial objects A and B are said to be *isomorphic* if there exists a permutation $p \in \mathcal{S}(\mathcal{P})$ such that $p(A) = B$.

Definition 2.20 Graph Isomorphism *Two graphs $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$ are said to be* isomorphic *if there exists a bijective mapping g from V_1 to V_2 such that $(u, v) \in E_1$ if and only if $(g(u), g(v)) \in E_2$ for all $u,\ v \in V_1$. The function g is called an* isomorphism *between G_1 and G_2.*

Definition 2.21 Graph Automorphism *An* automorphism *of a graph G is an isomorphism from G to G. The set of all automorphisms forms a group called the* automorphism group.

For example, each element in D_6 is an automorphism of the regular hexagon and D_6 is its automorphism group.

Definition 2.22 Homomorphism *Let Gp and Gp' be two groups with binary operators $\cdot$ and $\odot$ respectively. A function $\varphi : Gp \to Gp'$ is a* homomorphism *if*

$$\varphi(x \cdot y) = \varphi(x) \odot \varphi(y) \quad \forall x, y \in Gp.$$

The group Gp is the *domain* and the group Gp' is the *range.* A homomorphism sends the identity of Gp to the identity of Gp', sends inverses to inverses, and sends each subgroup of Gp to a subgroup of Gp'.

Definition 2.23 Image *The* image, *Im_φ, of the homomorphism is the subgroup of Gp' defined by $Im_\varphi = \{\varphi(g) \mid g \in Gp\}$.*

Definition 2.24 Preimage *Let T be a subset of elements of Im_φ. The* preimage, *$\varphi^{-1}(T)$, of T is the set of all elements of Gp which maps to some elements of T. That is, $\varphi^{-1}(T) = \{g \in Gp \mid \varphi(g) \in T\}$.*

The preimage of a subgroup of Im_φ is a subgroup of Gp.

Definition 2.25 Kernel *The kernel of φ is the preimage of the identity element of Gp'.*

Figure 5 on page 31 shows an example of a homomorphism mapping φ from the automorphism group Gp of the hexagon to Gp', the symmetric group permuting the cells C_1 and C_2, or $S_{\{C_1,\ C_2\}}$. It also shows how each element of Gp is related to the axes in the figure. For example, g_2 is equivalent to a reflection about the axis L and g_8 is equivalent to a rotation of $\pi/3$ clockwise about the axis Z. For each $g \in Gp$, $\varphi(g)$ is the result of g acting on C_1 and C_2, where $g(C_i) = \{g(x) \mid \forall x \in C_i\}$. For example, $g_2(a) = b$, $g_2(c) = f$, and $g_2(e) = d$. Hence $g_2(C_1) = C_2$. Similarly $g_2(C_2) = C_1$.

Hence $\varphi(g_2) = (C_1,\ C_2)$. On the other hand $g_9(C_1) = C_1$ and $g_9(C_2) = C_2$. Hence, $\varphi(g_9) = id$. The kernel of φ is $K_\varphi = \{g_1,\ g_5,\ g_6,\ g_7,\ g_9,\ g_{11}\}$, which contains the elements of Gp that are mapped to identity. The image of φ is $Im_\varphi = \{id,\ (C_1,\ C_2)\}$.

2.2.5 Various Definitions Related to Groups

We will be using the action of a permutation group to refine a cell. In the following section, we are going to show how this can be done.

Definition 2.26 Set Partition *A* set partition *of X, is a decomposition of the set into non-empty subsets, no two of which overlap and whose union is X.*

Definition 2.27 Orbit *Given a group Gp that acts on a set X. Let $x \in X$, the orbit of x is $Gp(x) = \{g(x) \mid g \in Gp\}$.*

An *orbit representative* is an element chosen from an orbit and used as a representative for that orbit. The relation of being in the same orbit is an equivalence relation. The set of orbits is a set partition, called the *orbit partition.*

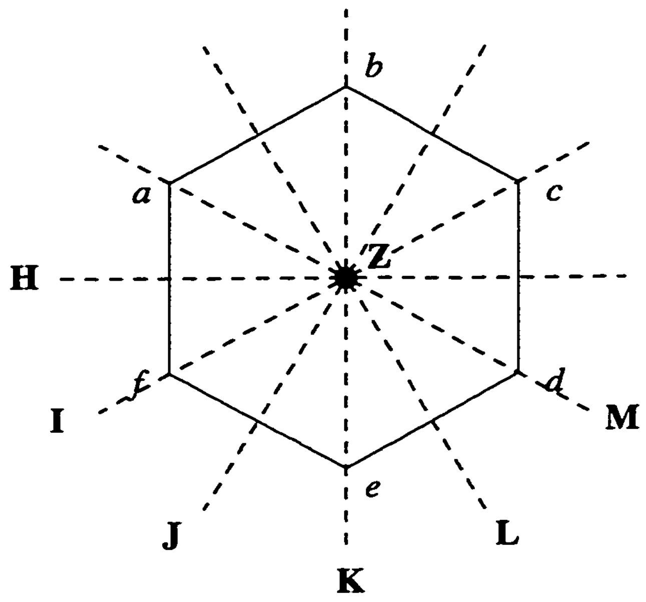
Definition 2.28 Transitivity *A group, Gp, acting on a set X is* transitive *if there is only one orbit. That is, for all $x, y \in X$, $\exists g \in Gp$ such that $g(x) = y$.*

Definition 2.29 Invariant Partition *A set partition $X_1,\ X_2, \ldots,\ X_k$ is* invariant *under the group Gp if the image of each subset X_i of the partition under any element of the group Gp is another subset in the partition. That is, for each i and for all $g \in Gp$, there is a j such that $g(X_i) = X_j$. Such a partition is called an* invariant partition *and those subsets in the invariant partition are called* blocks.

A partition is *discrete* if $|X_i| = 1$ for all i. It is *complete* if $X_1 = X$.

A invariant partition is *trivial* if one of the following holds:

1. the partition is discrete,
2. the partition is complete, or
3. Gp acts intransitively on X and each X_i is a union of one or more orbits.



$C = \{a, b, c, d, e, f\}$

$C_1 = \{a, c, e\}, \quad C_2 = \{b, d, f\}$

	Gp	Gp' : $\varphi(g_i)$	**Symmetries of** Gp
g_1	id	id	the identity symmetry
g_2	$(a, b)(c, f)(d, e)$	(C_1, C_2)	axis **L**
g_3	$(a, f)(b, e)(c, d)$	(C_1, C_2)	axis **H**
g_4	$(a, d)(b, c)(e, f)$	(C_1, C_2)	axis **J**
g_5	$(b, f)(c, e)$	id	axis **M**
g_6	$(a, c)(d, f)$	id	axis **K**
g_7	$(a, e)(b, d)$	id	axis **I**
g_8	(a, b, c, d, e, f)	(C_1, C_2)	rotate $\pi/3$ clockwise about **Z**
g_9	$(a, c, e)(b, d, f)$	id	rotate $2\pi/3$ clockwise about **Z**
g_{10}	$(a, d)(b, e)(c, f)$	(C_1, C_2)	rotate π clockwise about **Z**
g_{11}	$(a, e, c)(b, f, d)$	id	rotate $4\pi/3$ clockwise about **Z**
g_{12}	(a, f, e, d, c, b)	(C_1, C_2)	rotate $5\pi/3$ clockwise about **Z**

Figure 5: An example of a homomorphism φ.

Definition 2.30 Primitivity *A group Gp acting on X is* primitive *if there is no non-trivial partition of X that is invariant under Gp. A group that has a non-trivial invariant partition is called* imprimitive.

Usually, the concept of primitivity is restricted to transitive groups. In this case, all the subsets in an invariant partition have the same size.

As an example, the group Gp acting on $\{1, 2, \ldots, 20\}$ and generated by

$$h_1 = (1,2,3,4,5)(6,10,13,15,9)(7,11,14,8,12)(16,17,18,19,20),$$

$$\text{and } h_2 = (1,2)(3,4)(7,11)(8,10)(9,12)(14,15)(16,17)(18,19)$$

has the following invariant partitions.

1. $X_1 = \{1\},\ X_2 = \{2\}, \ldots,\ X_{20} = \{20\}$,
2. $X_1 = \{1, 2, \ldots, 20\}$,
3. $X_1 = \{1, 2, \ldots, 5\},\ X_2 = \{6, 7, \ldots, 15\},\ X_3 = \{16, 17, \ldots, 20\}$,
4. $X_1 = \{1, 2, \ldots, 15\},\ X_2 = \{16, 17, \ldots, 20\}$, and
5. $X_1 = \{1, 16\},\ X_2 = \{2, 17\},\ X_3 = \{3, 18\},\ X_4 = \{4, 19\},\ X_5 = \{5, 20\}$, $X_6 = \{6, 7, \ldots, 15\}$.

The first is the discrete partition and the second is the complete partition. The third and the fourth are formed by the orbits of Gp. The only non-trivial invariant partition in the above example is the last one. Since a non-trivial partition exists, Gp acts imprimitively on $\{1, 2, \ldots, 20\}$.

Since a cell is a set of vertices, all the definitions we defined in this section can be applied to a cell.

Definition 2.31 Stabilizer *Let Gp acts on X and let $x \in X$. The* stabilizer *of x is $Gp_x = \{g \mid (g \in Gp) \wedge (g(x) = x)\}$.*

Let Gp acts on X and let $X' \subseteq X$. The elements of Gp which fixes X' setwise form a subgroup $Gp_{X'}$ and it is called the *set stabilizer* of X'. In other words, $Gp_{X'} = \{g \mid g(X') = X'\}$. If $Gp_{X'} = Gp$, the we say Gp stabilizes X' or Gp fixes X'. If Gp fixes X', then X' is a union of orbits of Gp.

Definition 2.32 Cell Stabilizer *We define a subgroup that stabilizes a cell setwise as a* cell stabilizer.

Sometimes we want to restrict the action of Gp to a subset X' of X, where Gp fixes X'. First, we define a homomorphism $\varphi : Gp \rightarrow S_{X'}$ by $(\varphi(g))(x) = g(x)$ for all $x \in X'$. The image of Gp under this homomorphism is called the *truncated group of Gp according to X'* and is denoted as $tr_{X'}(Gp)$.

The truncation of Gp to a set X' makes mathematical sense if, when each element in Gp is written in cycle form, the cycles contain either only elements from X' or elements outside X'. For example, in Figure 5 on page 31, it does not make sense to truncate $g_2 = (a,\ b)\ (c,\ f)\ (d,\ e)$ according to cell $C_1 = \{a,\ c,\ e\}$ because the cycle $(a,\ b)$ contain an element a that is in C_1 and an element b that is not in C_1. On the other hand, it is all right to truncate $g_6 = (a,\ c)\ (d,\ f)$ according to C_1 and the result is $g_6' = (a,\ c)$.

The group D_6 is transitive and the orbit is $C = \{a,\ b,\ c,\ d,\ e,\ f\}$. Hence, the two blocks $C_1 = \{a,\ c,\ e\}$ and $C_2 = \{b,\ d,\ f\}$ shown in Figure 5 on page 31 is a non-trivial invariant partition of C under D_6. This implies that D_6 is imprimitive. The stabilizer of the cell C_1 is $\{g_1,\ g_5,\ g_6,\ g_7,\ g_9,\ g_{11}\}$ which is also the kernel of φ. The truncated group of K_φ according to C_1 is $tr_{C_1}(K_\varphi) = \{id,\ (c,\ e),\ (a,\ c),\ (a,\ e),\ (a,\ c,\ e),\ (a,\ e,\ c)\}$.

2.2.6 The Symmetry Partitioning Algorithm

In this section, we present the outline of the algorithm that uses the symmetry group to reduce the search tree. There are two stages of refinement in the algorithm. When symmetry still exists, the algorithm uses the symmetry group to refine a cell and partition its content. We call this partition the *symmetry partition.* When symmetry no longer exists, each refinement refines a cell C into two subcells C_1 and C_2 where $|C_1| = \lceil \frac{|C|}{2} \rceil$ and $|C_2| = \lfloor \frac{|C|}{2} \rfloor$. We call this kind of refinement *refine-into-half* and we call this partition the *even-splitting partition.* In this thesis, we concentrate on the symmetry partition. For more details about the even-splitting partition, see [47, 48].

Symmetry partition uses symmetry to reduce the size of the search. The basic algorithm recursively chooses a cell C, refines it into subcells $C_1, \ldots, C_n$ and partitions C's content into its children subcells. At the moment when the algorithm chooses a next cell to refine, there is a collection of cells, each with a content. The symmetry group at this moment is the subgroup of the automorphism group of the graph which stabilizes setwise the cells and their contents. This subgroup is called the *partition stabilizer.* The size of the search is reduced by reducing the number of content partitions. In order for this to happen, the symmetries must induce an action on the children subcells $C_1, \ldots, C_n$. This requirement explains the structure of the symmetry partition algorithm. It attempts to create a situation where the symmetry group acts transitively on C. If, furthermore, the action is imprimitive, then C is refined into imprimitive blocks. If the action is primitive, a vertex is split off as a discrete cell. By transitivity, the content of this discrete cell can be fixed. In the next few pages, we shall expand on this basic approach.

In the beginning, a graph G, a content size t and an automorphism group Gp of the graph are passed into the algorithm. The algorithm will group all the vertices in the graph into one cell C and the cell refinement will start with that cell. If the automorphism group is not trivial, symmetry partition will be used. In the case of symmetry partition, the group may or may not be transitive. If it is transitive, the group may be primitive or imprimitive. The algorithm will handle each case accordingly.

Non-Transitive Case

If the group is non-transitive, the cell will be refined according to its orbits. The content of C will be partitioned into its subcells. A cell D with a size greater than 1 will be selected from the list of available cells for partition. As Gp stabilizes cell D setwise, the cell D and the group Gp will be used for the next symmetry partition. An outline of how the algorithm handles the non-transitive case is shown in Algorithm 2.4.

Algorithm 2.4 An algorithm to handle the non-transitive case

Handle_non-transitive (Gp, C)
{ **partition** C according to its orbits under Gp;
  apply coverage and wastage tests to develop a lower
    bound and an upper bound for the content of each subcell;
  for each possible content partition of the subcells $C_1, \ldots, C_k$
  { **assign** the content partition to the subcells;
    select a cell D from the list of available cells for partition;
    Symmetry_partition (Gp, D);
  }
}

Transitive and Imprimitive Case

If the group is transitive and imprimitive, the cell will be refined according to the non-trivial invariant partition found. The permutation group Gp', representing the action of Gp on the imprimitive blocks, will be worked out. Each non-isomorphic content partition under the action of the group Gp' will be investigated. The subgroup H' of Gp' that fixes this content partition is evaluated. A cell D is selected from the list of available cells for partition. D and the preimage of H', $\varphi^{-1}(H')$ will be used for the next symmetry partition. The different groups that we used in handling this case are summarized in Figure 6. An outline of how the algorithm handles the transitive and imprimitive case is shown in Algorithm 2.5 on page 36.

Action on: **vertices** **blocks**

$$\begin{tikzcd} Gp \arrow[r] & Gp' \arrow[d, dash, "\textbf{subgroup}"] \\ \varphi^{-1}(H') & H' \arrow[l] \end{tikzcd}$$

Figure 6: The groups generated in the transitive and imprimitive case.

Algorithm 2.5 An algorithm to handle the transitive and imprimitive case

---

```
Handle_transitive_imprimitive (Gp, C)
{  partition C into imprimitive blocks;
   apply coverage and wastage tests to develop a lower
     bound and an upper bound for the content of each subcell;
   compute Gp' the permutation group that Gp acts on the subcells;
   for each non-isomorphic content partition under the group Gp'
   {  assign the content partition to the subcells;
      select a cell D from the list of available cells for partition;
      evaluate the subgroup H' of Gp' that fix this content partition;
      calculate the preimage of H', φ^{-1}(H');
      Symmetry_partition (φ^{-1}(H'), D);
   }
}
```

In Algorithm 2.5, the for-loop is executed for each non-isomorphic content partition. Here, each content partition is treated as an integer vector of size n, where n is the number of imprimitive blocks. The group Gp' permutes the blocks and hence the components of this integer vector. Two content partitions are isomorphic if their respective integer vectors can be permuted from one to the other by the element of Gp'. Among the elements in the orbit of content partition under Gp', we use as orbit representative the one that ranks lexicographically highest. Each of these orbit representatives represents a non-isomorphic content partition.

Referring back to Figure 5 on page 31, where $\sigma(C) = 2$. There are three possible content partitions for C_1 and C_2. They are listed in Table 1 on page 36. Since

	$\sigma(C_1)$	$\sigma(C_2)$
I	2	0
II	1	1
III	0	2

Table 1: The three possible content partitions assigned to cells C_1 and C_2.

Gp' contains the permutation $(C_1,\ C_2)$, content partition I is isomorphic to content partition III. It is sufficient for us to test only content partition I: either each content partition will lead to a solution that is isomorphic to the other, or none of them will lead to any solution. By pruning those isomorphic content partitions, the number of cases to be considered is reduced form 3 to 2, leading to a smaller search tree.

Transitive and Primitive Case

If the group is transitive and primitive, a vertex x will be selected and C will be partitioned into two subcells C_1 and C_2 where $C_1 = \{x\}$, and $C_2 = C - \{x\}$. If $\sigma(C) \neq |C|$ and $\sigma(C) \neq 0$, according to the Conservation Rule on page 9, there are only two possible content partitions, which are $\sigma(C_1) = 1,\ \sigma(C_2) = \sigma(C) - 1$, or $\sigma(C_1) = 0,\ \sigma(C_2) = \sigma(C)$. Consider the first case. Since $\sigma(C) \neq |C|$, eventually there will be a vertex in C_2 that will be refined into a discrete subcell that has a content equal to zero. As the group is transitive, we can relabel that vertex as the vertex in C_1, and obtain the content partition $\sigma(C_1) = 0$, and $\sigma(C_2) = \sigma(C)$. Similarly, if $\sigma(C_1) = 0$, and $\sigma(C_2) = \sigma(C)$, as $\sigma(C) \neq 0$, eventually there will be a vertex in C_2 that will be refined into a discrete subcell that has a content equal to one. Once again because the group is transitive, we can relabel that vertex as the vertex in C_1, we will get the content partition $\sigma(C_1) = 1,\ \sigma(C_2) = \sigma(C) - 1$. Therefore it is sufficient to consider just one of the two possible content partitions.

In order to decide which of the two possibilities to choose, we note that the number of ways to select $\sigma(C_2)$ vertices from a set of size $|C_2|$ is given by the binomial coefficient $\begin{pmatrix} |C_2| \\ \sigma(C_2) \end{pmatrix}$. Since $\sigma(C_2)$ is equal to either $\sigma(C)$ or $\sigma(C) - 1$, and $|C_2| = |C| - 1$, we are comparing $\begin{pmatrix} |C|-1 \\ \sigma(C) \end{pmatrix}$ with $\begin{pmatrix} |C|-1 \\ \sigma(C)-1 \end{pmatrix}$. We note that if $\sigma(C) \leq \frac{|C|}{2}$, then $\begin{pmatrix} |C|-1 \\ \sigma(C)-1 \end{pmatrix} \leq \begin{pmatrix} |C|-1 \\ \sigma(C) \end{pmatrix}$.

Hence, the content of C_1 will be chosen according to the following rule:

$$\sigma(C_1) = \begin{cases} 1 & \text{if } (\sigma(C) \neq 0) \text{ and } ((\sigma(C) \leq \frac{|C|}{2}) \text{ or } (\sigma(C) = |C|)) \\ 0 & \text{otherwise,} \end{cases}$$

Algorithm 2.6 An algorithm to handle the transitive and primitive case

---

Handle_transitive_primitive (Gp, C)
{ **select** a vertex x from C;
  refine C into $C_1 = \{x\}$ and $C_2 = C - \{x\}$;
  if $(\sigma(C) \neq 0)$ and $((\sigma(C) \leq \frac{|C|}{2})$ or $(\sigma(C) = |C|))$
    $\sigma(C_1) \leftarrow 1$;
  else
    $\sigma(C_1) \leftarrow 0$;
  $\sigma(C_2) \leftarrow (\sigma(C) - \sigma(C_1))$;
  select a cell D from the list of available cells for refinement;
  compute the cell stabilizer Gp_{C_1};
  Symmetry_partition (Gp_{C_1}, D);
}

---

and the content of C_2 will be chosen to be $\sigma(C_2) = \sigma(C) - \sigma(C_1)$. That rule makes the content partition bias towards having a larger content in C_2 when the content $\sigma(C)$ is greater than $\frac{|C|}{2}$ and a smaller content in C_2 when $\sigma(C)$ is less than or equal to $\frac{|C|}{2}$.

After the content is partitioned to the subcells, a cell D is selected from the list of available cells to be refined. D and the cell stabilizer Gp_{C_1} will be used for the next symmetry partition. An outline of how the algorithm handles the transitive and primitive case is shown in Algorithm 2.6.

Algorithm 2.7 on page 39 gives an outline of the symmetry partitioning algorithm. Figure 7 on page 40 illustrates how symmetry partition is applied to find a dominating set for the regular hexagon. The levels shown in the figure are the levels of the cell refinements. The size of the group at each level is shown under the $|Gp|$ column. The symbolic name of each cell is written on its left and the content of each cell is written on its right.

In the beginning, the automorphism group Gp of the hexagon graph is of size 12. The elements of the group were shown in Figure 5 on page 31. All vertices are

Algorithm 2.7 The symmetry partitioning algorithm

```
Symmetry_partition (Gp, C)
{  if Gp is not an identity group
      if Gp is not transitive on C
         Handle_non-transitive (Gp, C);
      else
         {
            find the cell stabilizer Gp_C;
            find tr_C(Gp_C), the group
               truncated to act on C;
            if tr_C(Gp_C) acts imprimitively on C
               Handle_transitive_imprimitive (Gp_C, C);
            else
               Handle_transitive_primitive (Gp_C, C);
         }
   else
      Even-splitting_partition (Gp, C);
}
```

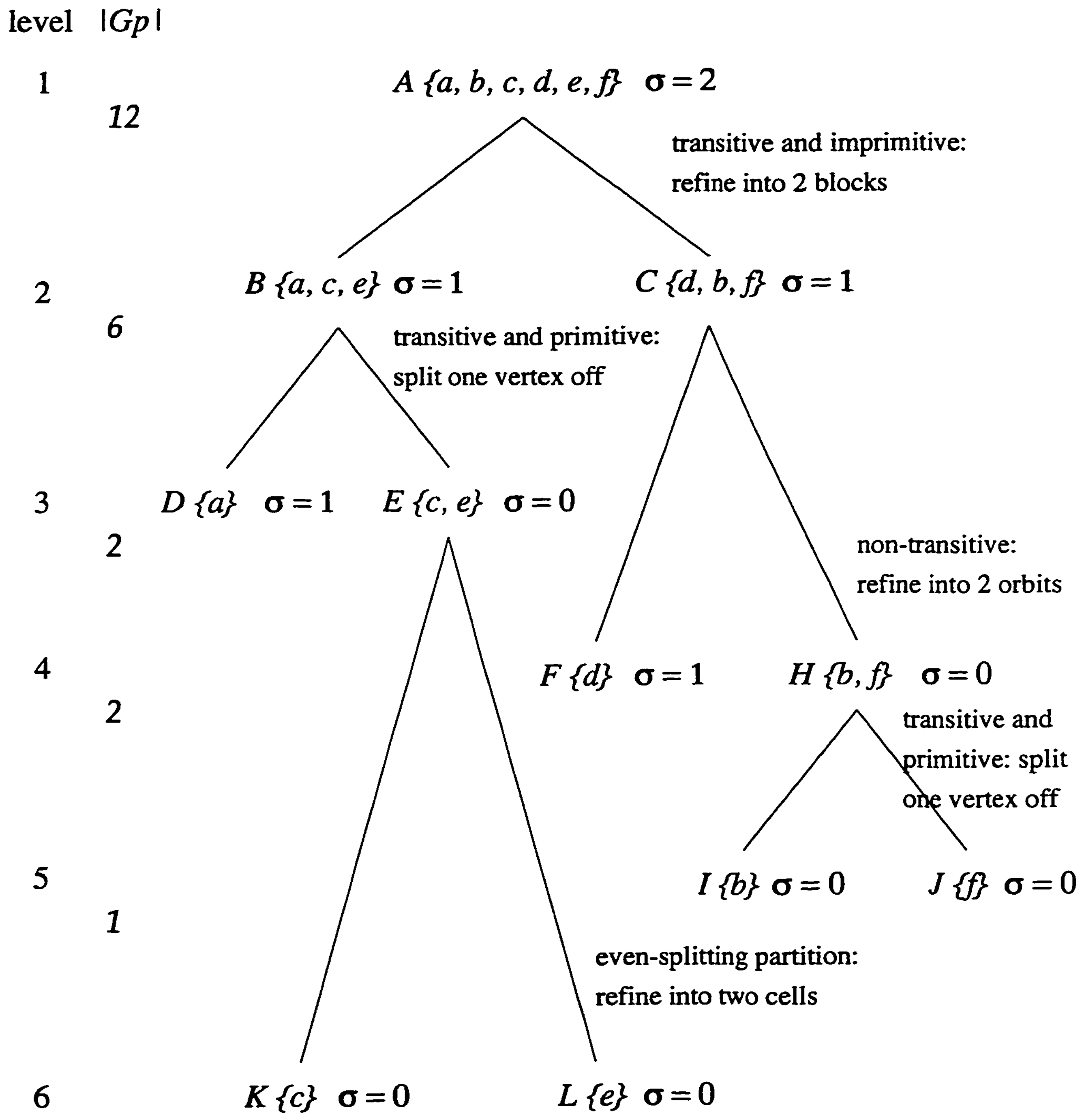


Figure 7: An example in finding the dominating set for the regular hexagon.

grouped into one cell A and hence, there is only one cell in level 1. The content of A is 2. Since the group is transitive and imprimitive, the cell is refined into the two blocks B and C. Among the three possible content partitions, only the case $\sigma(B) = 1$, and $\sigma(C) = 1$ passes the coverage and wastage tests. The content partition $\sigma(B) = 2$, and $\sigma(C) = 0$ fails the coverage test because no two vertices of B can be selected to cover cell B. The three vertices in B are not connected to each other, and no vertices in cell C can be chosen as an element in the dominating set because $\sigma(C) = 0$. Hence, the content of B should be at least 3 in order to cover all the vertices in B. Thus this content partition is rejected by the coverage test. The content partition $\sigma(B) = 0$, and $\sigma(C) = 2$ is rejected similarly. Since the subgroup H' that fixes this content partition is the same as the image of the permutation group of Gp acting on the subcells B and C, that is, $\{id, (B, C)\}$, the preimage of H' is the same as Gp.

In level 2, cell B is selected as the cell to be refined. Since the group acts transitively, the cell stabilizer Gp_B is evaluated. It has a size of 6. Since Gp_B acts primitively on B, the vertex a is selected as cell D, and vertices c and e are placed into cell E. The content of D is set to 1 because $\sigma(B) = 1 < \frac{|B|}{2} = 1.5$. The cell stabilizer of D under Gp_B, $(Gp_B)_D$ is evaluated, and its size is 2.

In level 3, there are three cells C, D, and E. Cell C is selected as the cell to be refined. The group, $(Gp_B)_D$, stabilizes the three cells C, D, and E setwise. It acts non-transitively on C. Therefore, cell C is refined according to the two orbits found. There are two possible content partitions, but only the one with $\sigma(F) = 1$ and $\sigma(H) = 0$ passes the coverage and wastage tests. The content partition $\sigma(F) = 0$ and $\sigma(H) = 1$ is rejected by the coverage test because vertex d cannot be covered. Only vertices c and e have an edge connecting to vertex d. Hence, no other vertices except c and e can cover vertex d. But cell $E = \{c, e\}$ has a content equal to zero. Therefore, the content of cell $F = \{d\}$ has to be 1 in order to cover d. The next cell selected is H. Since refining according orbits does not change the symmetry group, the same group passed from level 2 is passed to the next level of the algorithm. We choose Gp'' to represent this group.

Level 4 contains four cells. D, E, F, and H. Since $\sigma(H) = 0$, we could refine H

into discrete cells and each cell has a content equal to zero. But in order to illustrate how the program deals with transitive groups, we do not choose to refine H into discrete cells. Since the group Gp'' acts transitively on H, the cell stabilizer Gp''_H is obtained and it has a size of 2. Since Gp''_H acts primitively on H, vertex b is selected and refined into cell I, and vertices f is refined into cell J. Since their parent cell H has a content equal to zero, there is only one possible content partition which is $\sigma(I) = 0$ and $\sigma(J) = 0$. The cell stabilizer of I under Gp''_H, $(Gp''_H)_I$ has a size of 1.

In level 5, cell E is selected for refinement. There are five cells in this level and they are D, E, F, I, and J. Since the group has a size of 1, it is the identity group. We cannot reduce the search tree by making use of the symmetry group any more and the even-splitting partition is used. Although $\sigma(E) = 0$, we choose not to refine it into discrete cells and continue with the refine-into-half procedure to illustrate the even-splitting partition. Cell E is refined into two cells of equal size, namely, $K = \{c\}$ and $L = \{e\}$. Since $\sigma(E) = 0$, there is only one possible content partition which is $\sigma(K) = 0$, $\sigma(L) = 0$. After cell E is refined, there are no more cells that can be refined further. Among the six cells, cells D and F have contents equal to one. This implies vertices a and d are on. Since a and d together do cover all the vertices in the graph G, the set $\{a, d\}$ is a valid dominating set. Hence, one solution is found.

Chapter 3

Implementation Details

Based on the algorithms discussed in Chapter 2, a program has been developed to find the dominating set of a graph $G(V, E)$ with a given test size t. In this chapter, we discuss the data structures and assumptions we made to implement the program, and explain why we need those data structures.

Our program to find the dominating set of a graph has been implemented using the C programming language. It was developed using a SUN Sparc 10 workstation. It has also been ported to a Digital 2100 Server Model A500M. The program incorporates a package for isomorphic testing called ISOM [43, 44], developed by C. Lam and L. Thiel. The generators of the automorphism group of the input graph was generated using MAGMA [10, 14], a software package designed and developed in the School of Mathematics and Statistics at University of Sydney, Australia.

Our program takes a graph, a test size t, and the generators of the automorphism group of the graph as input. The program recursively selects a cell to refine. Depending on the preset flags, the program can either be used to find all the dominating sets of size t, or stop once the first dominating set of size t is found. We use the term *level* to denote the recursive level of the cell refinement. In the beginning, the program starts at level 1. We have included all source codes of this program in Appendix B on page 108.

3.1 Data Structure

In this section, we describe the data structures used to store the frequently accessed information. The input graph is represented by two data structures. The first data structure is an adjacency matrix `adj_matrix` implemented using a two dimensional array of integers. The second data structure `neigh_graph` is an array of pointers indexed by the vertices. Each pointer points to the list of neighbors of the corresponding vertex. This data structure helps to reduce the time in searching for the neighbor of a vertex. For a graph of size n, we number the vertices from 1 to n.

All the cells' vertices are recorded in an array of size $|V|$ called `allcells`. Vertices in the same cell are stored in cycle form. The array needs to be updated after each cell refinement. Each cell has a *cell representative* which is the smallest vertex in the cell. It acts as an identifier for that cell. Another array, called `cell_label`, is an integer array of size $|V|$. It maps each vertex to the corresponding cell's representative. Like `allcells`, it allows us to know which vertex belongs to which cell in one computational step and needs updating at each level of cell refinement. All the cells at a given level can be accessed through the data structure `cell_list`: an array of pointers pointing to each cell. The array is indexed by the cell representative and its information is updated at each cell refinement.

Each cell's information is stored in a data structure called `cell_node`. The following is the data structure of `cell_node`.

```
typedef struct cell_node *ptr_to_cell_node;
struct cell_node {
     ptr_to_neigh_info_node row_first_neigh;
     ptr_to_neigh_info_node col_first_neigh;
     ptr_to_cell_node parent;
     ptr_to_cell_node sibling;
     ptr_to_cell_node first_child;
     ptr_to_cell_node prev_cell;
     ptr_to_cell_node next_cell;
```

```
    int representative;
    int ver_before_rep;
    int cell_size;
    int total_children;
    int lower_content;
    int upper_content;
    int actual_content;
    int max_allow_wastage;
};
```

The `row_first_neigh` and `col_first_neigh` are linked to the information of the neighbors of a cell. The neighbor's information is stored in the data structure `neigh_info_node`. Both lists refer to the same set of neighbors, but the `tot_ver_of-_deg_from_row_to_col` field in the `neigh_info_node` linked by `row_first_neigh` and `col_first_neigh` are different. This information is needed by the following four functions: `Coverage_test`, `Wastage_test`, `Update_max_allowable_wastage`, and `Generate_next_level` when working out the influence functions. The field `parent` is a pointer to the parent cell. The field `first_child` is a pointer to the first subcell refined from the current cell. The field `sibling` is a pointer to the next child subcell of its parent. Both of the fields `first_child`, and `sibling` are needed in performing content partition.

To obtain the children of a cell, one must first access the `first_child` pointer of that cell, and then, through the `sibling` pointers of its children, one can reach the rest of its children. The two pointers `next_cell` and `prev_cell` are used to link to all the cells in the same level. They are implemented as a doubly-linked circular list to allow easy insertion and deletion of a cell from one level to another. Nodes are linked in an arbitrary order, but the children of the most recently split cell are linked together. The field `representative` is the cell representative which contains the smallest vertex in a cell. The rest of the vertices in a cell can be found using `allcells`. The field `ver_before_rep` is the vertex before the cell representative in the cycle form stored in the array `allcells`. The field `cell_size` gives the total number of vertices in a cell and `total_children` gives the total number of subcells refined from a cell. It is used to allocate the size of the array in `lower_limit_distribution`, `upper_limit_distribution`, and `actual_content_distribution` during content partition. The fields `lower_content` and `upper_content` give a lower and upper bounds for the content of a cell. They are calculated right after cell refinement, and are determined by the coverage and wastage tests. The `actual_content` is the content assigned to this cell. It is evaluated at the stage of content partition. Finally, `max_allow_wastage` is the maximum allowable wastage of a cell. After the `actual_content` is defined, it is used in the wastage test to check if the total coverage among all the neighbors is at least as large as the `total_vertices` in a cell.

In order to facilitate the calculation of the lower and upper influence functions used by the coverage and wastage tests, the data structure `neigh_info_node` is created. It is referenced through `cell_node` and each `neigh_info_node` contains the information of a neighbor cell of that `cell_node`. The data structure of a `neigh_info_node` is defined below.

```
typedef struct neigh_info_node *ptr_to_neigh_info_node;
struct neigh_info_node {
     int row_cell_num;
     int col_cell_num;
     int *tot_ver_of_deg_from_row_to_col;
     ptr_to_neigh_info_node row_next_neigh;
     ptr_to_neigh_info_node row_prev_neigh;
     ptr_to_neigh_info_node col_next_neigh;
     ptr_to_neigh_info_node col_prev_neigh;
};
```

The `neigh_info_node` is circularly linked in a way to simulate a two-dimensional matrix. It is linked as such to simplify the access and the calculation of the neighbors' influence functions. Both the row and the column entries of the matrix are indexed by the cell representative. There is a dummy `neigh_info_node` linked at the beginning of each row and column of the `neigh_info_node` linked list. It is used to reduce the complication of accessing the `neigh_info_node` and to store other information such as the total number of nodes linked to it in that row or column. They are classified as the entry 0 of the matrix by assuming that there is an imaginary cell with cell representative equal to 0. The dummy `neigh_info_node` also acts as a marker to indicate this is the beginning of the circular linked list. Except when all the vertices are in individual cells, not all the vertices are used as cell representatives, hence the matrix formed by the `neigh_info_nodes` is a sparse matrix most of the time. The fields `row_cell_num`, and `col_cell_num` are used to indicate the position of this node in the sparse matrix. They indicate which cell corresponds to the row and which cell corresponds to the column in the field `tot_ver_of_deg_from_row_to_col`.

The field `tot_ver_of_deg_from_row_to_col`, is a pointer to a node which contains an array indexed from zero to the maximum degree θ of $G(V, E')$. Suppose that the row cell is C_i and the column cell is C_j and suppose we denote the array `tot_ver_of_deg_from_row_to_col` by K_{ij}. Then $K_{ij}(t)$ gives the total number of vertices in C_i which are each connected to t vertices in C_j. In other words,

$$K_{ij}(t) = |\{u \in C_i \mid |N_u \cap C_j| = t\}|,$$

where N_u is the neighborhood of vertex u. Recall that the lower influence function $L_{ij}(m)$ is defined by

$$L_{ij}(m) = \min_{B \subseteq C_i,\ |B|=m} \left(\sum_{u \in B} |N_u \cap C_j|\right).$$

Recall also that $L_{ij}(m)$ can be computed by first sorting the vertices u of C_i in ascending order according to $|N_u \cap C_j|$, and then summing over the first m vertices in this sorted list. Using K_{ij}, $L_{ij}(m)$ can be computed by

$$L_{ij}(m) = \sum_{r=0}^{s-1}(rK_{ij}[r]) + s \times (m - \sum_{r=0}^{s-1} K_{ij}[r]), \tag{3}$$

where s is the minimum value satisfying the inequality $\sum_{r=0}^{s} K_{ij}[r] \geq m$. Here, $\sum_{r=0}^{s-1}(rK_{ij}[r])$ gives the total number of of edges going from the first $\sum_{r=0}^{s-1} K_{ij}[r]$ ascending vertices of C_i into C_j. Finally, $s \times (m - \sum_{r=0}^{s-1} K_{ij}[r])$ gives the total number of edges going from the next $m - \sum_{r=0}^{s-1} K_{ij}[r]$ ascending vertices of C_i into C_j. Together, this gives the total number of edges going from the first m ascending vertices of C_i into C_j, or $L_{ij}(m)$.

Similarly, the upper influence function is

$$U_{ij}(m) = \sum_{r=s+1}^{\theta} (rK_{ij}[r]) + s \times (m - \sum_{r=s+1}^{\theta} K_{ij}[r]), \tag{4}$$

where s is the maximum value satisfying the inequality $\sum_{r=s}^{\theta} K_{ij}[r] \geq m$.

The fields `row_next_neigh` and `row_prev_neigh` link the next or previous `neigh-_info_node` along a row. Similarly, `col_prev_neigh` and `col_next_neigh` link them along a column.

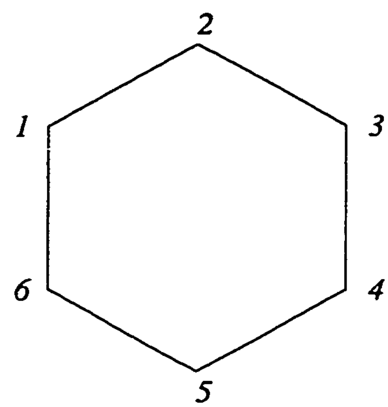
Figure 8 on page 49 shows the adjacency matrix and the neighbor graph of the hexagon. Figure 9 on page 50 shows how all the data structures are related to each other at level 1 and Figure 10 on page 51 shows how they are related to each other at level 2. Both data structures are recorded right after the coverage and wastage tests but before the content partition. Hence, the maximum allowable wastage of the refined subcells in level two of Figure 10 are undefined. We use -1 to indicate entries that are undefined. One can appreciate how the `neigh_info_node` is circularly linked in a way to simulate a two-dimensional matrix. There is a dummy `neigh_info_node` linked at the beginning of each row and column of the `neigh_info_node` linked list. They are classified as entry 0 of the matrix. The various aspects of the program are outlined in the following subsections.

3.2 Neighborhood Assumption

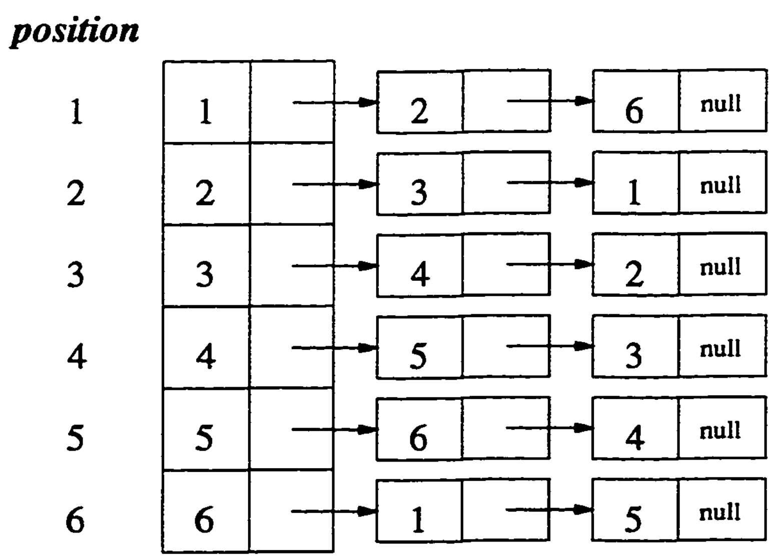
Currently, our program works only on undirected, loopless, and labeled graph with no multiple edges. Throughout the program, if C_i is a neighbor of C_j, the program assumes C_j is also a neighbor of C_i. Using this assumption, the program saved some computational effort when building the `tot_ver_of_deg_from_row_to_col` field in the `neigh_info_node`.

$$
\begin{array}{c c}
\textbf{\textit{vertex}} & \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\
\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} &
\begin{bmatrix}
1 & 1 & 0 & 0 & 0 & 1 \\
1 & 1 & 1 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 0 \\
0 & 0 & 0 & 1 & 1 & 1 \\
1 & 0 & 0 & 0 & 1 & 1
\end{bmatrix}
\end{array}
$$

adj_matrix



hexagon



neigh_graph

Figure 8: The adjacency matrix and the neighbor graph of the regular hexagon.

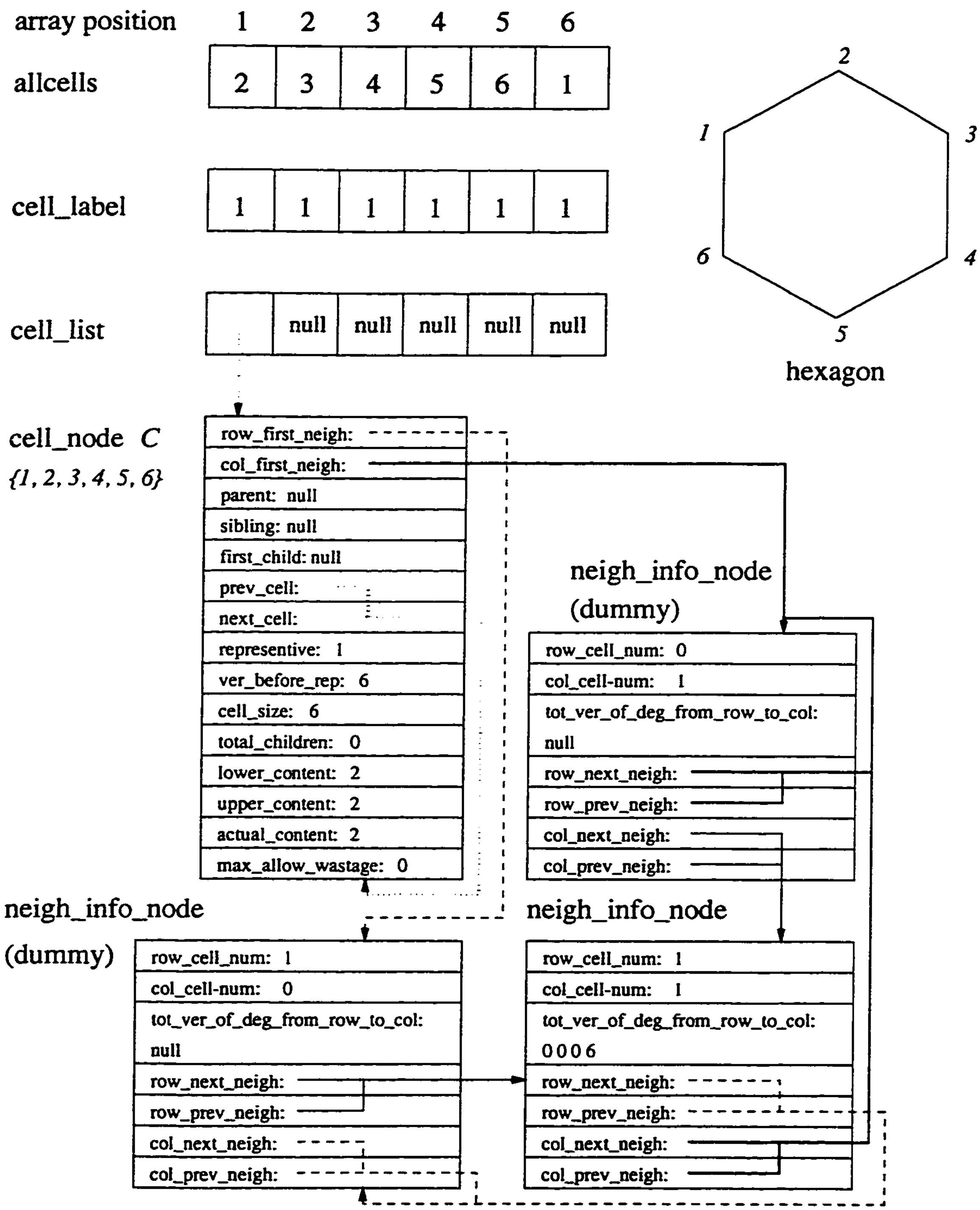


Figure 9: The data structures of the hexagon at its first level.

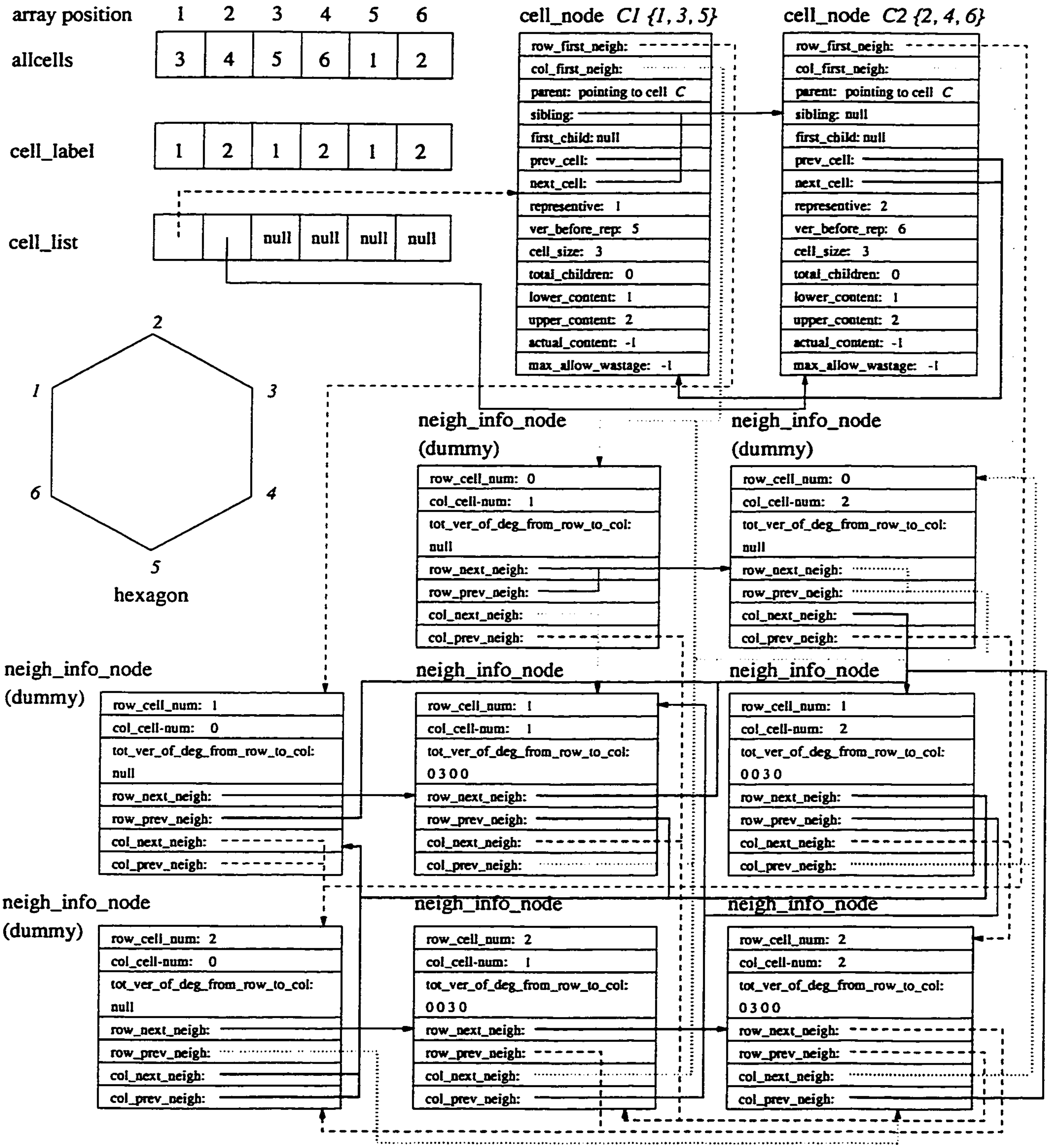


Figure 10: The data structures of the hexagon at level 2.

3.3 Content Partition

The content partition is generated by a non-recursive method. The only recursive part in the program is the cell refinement. The content partition is generated by using each possible content of each cell starting from its `lower_content` to its `upper_content`. If the content partition satisfies the Conservation Rule in Theorem 2.1 on page 9, this content partition is assigned to the subcells and the maximum allowable wastage of each cell will be worked out. If the symmetry group is transitive and imprimitive, isomorph rejection will be applied to the content partition.

3.4 Other Interfaces

The dominating set program uses the ISOM package for many of its group manipulation algorithms. The group storage and generation, isomorph rejection, truncation of a group, and the manipulation of imprimitivity blocks are either directly or indirectly developed from the procedures in the ISOM package. The generators of the input graphs are prepared by MAGMA. We also used MAGMA to generate orbit graphs, and their symmetry groups. Additional details about orbit graphs are described in Chapters 5 and 6.

Chapter 4

Grid Graph

The dominating set program is first tested on the grid graphs. The grid graphs are chosen for two reasons. First, their symmetry group is simple, which makes it easier for debugging. Second, the results for small cases are known, which allows us to verify the correctness of the algorithm.

4.1 Introduction

Definition 4.1 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 either* $i = i'$ *and* $j = j' \pm 1$ *or* $i = i' \pm 1$ *and* $j = j'$.

In other words, two vertices are adjacent if they are consecutive on a row or a column. An example of a 3×3 grid graph is shown in Figure 11 on page 54. In that graph, we let $a = (1,1)$, $b = (1,2)$, $c = (1,3)$, $d = (2,1)$, $e = (2,2)$, $f = (2,3)$, $g = (3,1)$, $h = (3,2)$, $i = (3,3)$.

4.2 Results

In our program, in order to decide whether t is the minimum domination number for a graph G, we have to make sure that t is a domination number of G but $t-1$ is not a domination number of G. The CPU time required to decide whether t is

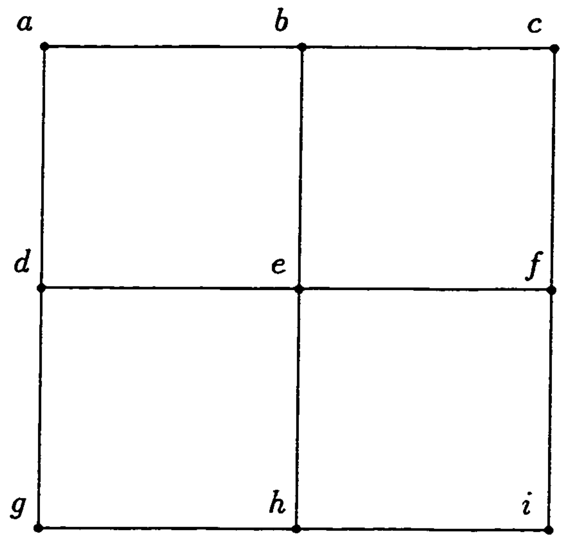


Figure 11: A 3×3 grid graph.

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 $t-1$ and t. In the case of test size t, we can stop once a dominating set of size t is found.

In [47], we have presented some timing results for solving the dominating set problem of several grid graphs. We use the program P_{no_sym} which is based on a partitioning algorithm without making use of the symmetry group of the graph. The program is run on a SUN 4/280 machine. We confirmed that $\gamma_{8\times8} = 16$, $\gamma_{8\times9} > 17$, and $\gamma_{9\times9} > 19$. Some of the best timing results are shown in the Table 2 on page 55. In the table, if the column *"Stop at first solution"* is yes, the program will stop once the first dominating set is found, otherwise, all the dominating sets will be found before the program terminates. The same set of grid graphs plus some bigger grid graphs are tested using the program P_{sym}, which is based on the symmetry partitioning algorithm we presented in this thesis. Some of the tests are run on a SUN SPARC 10 workstation. The rest of the graphs, which require more CPU time, are run on a Digital 2100 Server Model A500M machine. Roughly, when running the same program, P_{no_sym}, with the same set of input, we found that the Digital 2100 machine is about four times faster than the SPARC 10 workstation and a SPARC 10 workstation is about six times as fast as a SUN4/280 machine. The results are

m	n	*Test size*	*Normalized CPU time (s)*	*Number of solutions*	*Stop at first solution*	*Machine*
7	7	11	6.2	0	no	SUN 4/280
7	7	12	56.9	1	yes	SUN4/280
8	8	14	8.1	0	no	SUN4/280
8	8	15	205.2	0	no	SUN4/280
8	8	16	49.5	1	yes	SUN4/280
8	9	17	2412.7	0	no	SUN4/280
9	9	19	7649.5	0	no	SUN4/280

Table 2: Timing results on grid graphs for P_{no_sym}.

shown in the Table 3 on page 56. To simplify the comparison, the CPU time given in the two tables are normalized according to the Digital Alpha Machine.

Comparing Table 2 with Table 3 on page 56, the CPU time taken to run the 8×8 grid graph with a test size 16 requires an explanation. The reason why the program using the symmetry group takes longer to run than the one without using the symmetry group is because both programs stop once the first dominating set is found. Since the two programs use different ways to partition a cell, there is no guarantee which program would find a dominating set first. Moreover, there is overhead spent in group related calculations for the program using the symmetry group before finding the first solution. Fortunately, this kind of situation is rare.

Since P_{sym} makes use of the orbit and block structure of the graph's symmetry group to prune the search, it works best for graph with a large symmetry group. The size of the automorphism group of a $m \times n$ grid graph is 4 when $m \neq n$ and 8 when $m = n$. Despite the small automorphism group, P_{sym} shows a large gain in computational efficiency compared to P_{no_sym} because of the use of better data structures and the redesign of several parts of the program.

In [18], the authors give a list of known minimum domination numbers of some $m \times n$ grid graphs, $\gamma_{m \times n}$. Table 4 on page 56 is taken from [18]. It shows a table of the minimum domination number in $m \times n$ grid graphs where $8 \leq m \leq 12$ and $8 \leq n \leq 12$. Using the new algorithm, we have found the minimum dominating sets

m	*n*	*Test size*	*Normalized CPU time (s)*	*Number of solutions*	*Stop at first solution*	*Machine*
7	7	11	0.08	0	no	SPARC 10
7	7	12	20.0	1	no	SPARC 10
8	8	14	0.2	0	no	SPARC 10
8	8	15	10.2	0	no	SPARC 10
8	8	16	3017.9	1	yes	SPARC 10
8	9	17	20.1	0	no	SPARC 10
8	9	18	324.3	1	yes	SPARC 10
8	10	19	111.4	0	no	Alpha
8	11	21	1421.8	0	no	Alpha
8	12	23	7139.8	0	no	Alpha
9	9	19	198.6	0	no	SPARC 10
9	10	21	212.9	0	no	Alpha
9	11	23	325.9	0	no	Alpha
9	12	25	1508.3	0	no	Alpha
10	10	24	5484.5	0	no	Alpha
10	11	26	38955.6	0	no	Alpha

Table 3: Timing results on grid graphs for P_{sym}.

$m \setminus n$	8	9	10	11	12
8	16	18	20	22	24
9		20	22	24	26
10			24	27	29
11				29	32
12					35

Table 4: The minimum domination numbers of $m \times n$ grid graphs where $8 \leq m \leq 12$ and $8 \leq n \leq 12$.

of all $m \times n$ grid graph for $m \leq 10$ and $n \leq 11$. The sizes of the minimum dominating sets agree with the minimum domination numbers listed in Table 4. Owing to the enormous amount of computation, we are not able to obtain some of these results when using P_{no_sym}.

All the minimum domination numbers for grid graphs computed by our program agree with those in the literature, which is a strong indication of its correctness. In Chapter 7, we shall have more comments on the program's performance.

Chapter 5

Weighted Dominating Set and Orbit Graph

In this chapter, we assume that the graph and its dominating set has a non-trivial automorphism group. This assumed automorphism group partition the graphs into orbits which induce a smaller graph. We shall show that solving a weighted dominating set problem on the smaller graph will lead to a solution of the dominating set problem for the original graph.

5.1 Weighted Dominating Set

We begin by introducing the concept of assigning a weight to a vertex in a graph and by giving the definition of a *weighted dominating set problem.*

As defined in Chapter 2, two graphs, $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$, are isomorphic to each other if there exists a one-to-one onto mapping g from V_1 to V_2 such that $(x, y) \in E_1$ if and only if $(g(x), g(y)) \in E_2$. If $V_1 = V_2$, then g is just a *permutation* of the vertices. A permutation g is a symmetry of a graph $G(V, E)$ if for all $x, y \in V$, $(x, y) \in E$ if and only if $(g(x), g(y)) \in E$. The set of symmetries of G forms a group Gp. Usually, Gp is called an *automorphism group.* However, we use the term *automorphism group* to refer to the subgroup of Gp stabilizing a set of cells. Thus, we choose to call Gp the *symmetry group* to avoid ambiguity.

Suppose each vertex $x \in V$ has a positive weight w_x. The *weighted size* of a set

$S \subseteq V$ is the sum of the weights of all the elements of S.

Definition 5.1 Weighted Dominating Set Problem *Given a graph $G(V, E)$, a test size t, and a set of weights, one at each vertex. The problem of finding whether there exists a dominating set of weighted size t is called the* weighted dominating set problem.

5.2 Orbit Graph

Definition 5.2 Orbit Graph *Given a graph $G(V, E)$ and a symmetry group Gp. Let H be a subgroup of Gp and let H partition V into orbits. Suppose the orbits are $O_1, O_2, \ldots, O_m$. The* orbit graph *of H is $G_{orb}(V_{orb}, E_{orb})$ where the vertex set $V_{orb} = \{1, \ldots, m\}$, the edge set $E_{orb} = \{(i,j) \mid (x,y) \in E$ for some $x \in O_i$ and $y \in O_j\}$. The weight w_i of vertex $i \in V_{orb}$ is $|O_i|$, the size of O_i.*

The orbit graph G_{orb} is *non-trivial* if $|V_{orb}| < |V|$. Referring back to the regular hexagon on page 3, let H be generated by the element $(a,\ c)(d,\ f)$, that is to say, $H =< (a,\ c)(d,\ f) >$. The adjacency matrix of the orbit graph becomes:

w_x	V_{orb}	$\{a,c\}$	$\{d,f\}$	$\{b\}$	$\{e\}$
2	$\{a,c\}$	1	1	1	0
2	$\{d,f\}$	1	1	0	1
1	$\{b\}$	1	0	1	0
1	$\{e\}$	0	1	0	1

.

Figure 12 on page 60 shows this orbit graph. There exists a dominating set of weighted size 2 for the orbit graph and its weighted dominating set is $\{\{b\},\ \{e\}\}$.

When searching for the dominating set of a graph, the size of the graph may be too big for the program to solve in a reasonable amount of time. By transforming the graph into a graph based on its orbits under a subgroup of its symmetry group, a weighted graph of smaller size can be obtained which may be solvable by the program. Moreover, for each subgroup of the symmetry group, we can solve the corresponding orbit graph using a different computer. Hence, we can search for

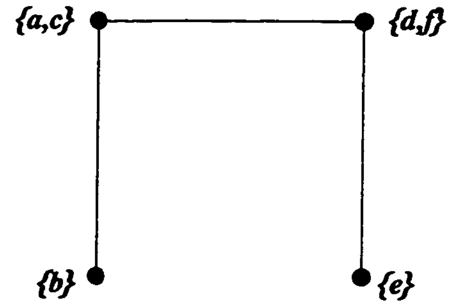


Figure 12: Orbit graph of the regular hexagon under the group $< (a,\ c)(d,\ f) >$.

weighted dominating sets for the orbit graphs in parallel. We now explain why finding a weighted dominating set will lead to a dominating set in the original graph.

Lemma 5.1 *If $(i, j) \in E_{orb}$, then for all $x \in O_i$, there exists $y \in O_j$ such that $(x, y) \in E$.*

Proof: Given $(i, j) \in E_{orb}$, there exists $v \in O_i$ and $z \in O_j$ such that $(v, z) \in E$. Since O_i is an orbit, there exists $h \in H$ such that $x = h(v)$. We can take $y = h(z)$ and $(x, y) \in E$, as required. □

Theorem 5.2 *If the orbit graph of H has a weighted dominating set D_{orb} and weighted size t, then*

$$D = \bigcup_{i \in D_{orb}} O_i$$

is a dominating set with size t in G. Moreover, D is fixed by H setwise.

Proof: It is clear that D has size t. The fact that that D is a dominating set follows from the Lemma 5.1. Since D is a union of orbits of H, it is fixed by H setwise. □

Given a subgroup H of Gp, a subgroup K is a *conjugate* of H under Gp if there exist $g \in Gp$ such that $K = gHg^{-1}$.

Theorem 5.3 *If K is a conjugate of H under Gp, then the orbit graph of K is isomorphic to the orbit graph for H.*

Proof: Suppose $K = gHg^{-1}$ with $g \in Gp$. Let O be an orbit under the action of H. For all $y \in O$ and for all $ghg^{-1} \in K$ where $h \in H$, $ghg^{-1}(g(y)) = gh(g^{-1}g(y)) = g(h(y))$. As y and $h(y)$ are in O and $g(y)$ and $g(h(y))$ are in an orbit under the action of K. Hence, $g(O)$ is an orbit of K. If x is an orbit representative of O, one can choose $g(x)$ as the orbit representative of $g(O)$. This implies g is a one-to-one onto mapping from the orbits of H to the orbits of K. Next, we show that g maps edges to edges as follows. If O_i and O_j are orbits under H and there exists an edge (x, y) with $x \in O_i$ and $y \in O_j$, as Gp is a symmetry group, $(g(x), g(y))$ is an edge with $g(x) \in g(O_i)$ and $g(y) \in g(O_j)$. Hence, g is an isomorphism mapping which maps the orbit graph for H to the one for K. $\square$

Theorem 5.4 *The property of having a (weighted) dominating set of a particular (weighted) size is invariant under isomorphism.*

Proof: Suppose that a graph $G_1(V_1, E_1)$ is isomorphic to a graph $G_2(V_2, E_2)$ under an isomorphism g. Let D be a dominating set of G_1, and let $D' = \{g(v) \mid \forall\ v \in D\}$. By the definition of graph isomorphism, $(u, v) \in E_1 \Rightarrow (g(u), g(v)) \in E_2$. Therefore, D' is a dominating set for $G(V_2, E_2)$. The same proof can be applied to a weighted dominating set. $\square$

Theorems 5.3 and 5.4 together imply that, to generate all the orbit graphs, we only have to consider subgroups of Gp up to conjugacy. There may still be many such subgroups. To reduce this number, we consider only a subset of these subgroups which are complete in the sense that if there exists such a non-trivial automorphism group, this dominating set can be obtained as a weighted dominating set from an orbit graph generated from one of the subgroups in the reduced set. This subset is obtained by observing that if there exists a dominating set with a non-trivial automorphism group, then there exists a non-trivial element g in the automorphism group. The same dominating set is also fixed by $< g >$, the subgroup generated by g. Hence, we can consider first the subset of subgroups with only a single generator. This subset is essentially the lists of elements up to conjugacy.

5.3 Adapting Our Program for Weighted Dominating Set

Since the vertices in an orbit graph may have different weights, the dominating set program described in Chapter 2 does not work directly on orbit graphs. However, with some simple modification, the program can work on orbit graphs.

First, we refine V_{orb} according to weights, with $V_{orb} = C_1 \cup C_2 \cup \ldots \cup C_r$ where vertices in the same subcell have the same weight and those in different subcells have different weight. For explanation purposes, we assume that vertex $i \in C_i$ for $i = 1, \ldots, r$. Thus we can use w_i to denote the weight of a vertex in subcell C_i. If t is the weighted size of the weighted dominating set for G_{orb}, then

$$t = \sum_{i=1}^{r} w_i \sigma(C_i), \tag{5}$$

where $\sigma(C_i)$ is the content of subcell C_i. The initial cell refinements, $C_1, \ldots, C_r$, are done outside the dominating set program. Thus, the input to the dominating set program is the initial cell refinement, the weight of each of the subcell in the refinement and the test size t of the original graph.

Based on equation (5), once the content partition of the first level is defined, we can treat all the vertices in each subcell as having the same weight, and the rest of the program can be used unmodified. If a solution of an orbit graph is found, the actual dominating set can be obtained by mapping back the vertices of the orbit graph to the vertices in the orbits of the original graph as stated in Theorem 5.2 on page 60.

We also have to be careful about the initial symmetry group. It has to be the automorphism group of the weighted orbit graph. In other words, the initial symmetry group is the subset of vertex permutations such that the weights of x and $g(x)$ are the same, and that g is edge preserving.

Before ending this chapter, we would like to remark that although the idea of orbit graph can reduce the size of the original graph, we cannot conclude that the domination number is greater than t if we cannot find a dominating set of size t for all the non-trivial orbit graphs. We can only conclude that there is no non-trivial automorphism for the dominating set of that graph for that test size. To claim that the domination number is greater than t, we have to perform an exhaustive search with test size t.

Chapter 6

Football Pool Problem

In this chapter, we discuss the results of the orbit graphs obtained and the dominating sets found for the football pool problems.

6.1 Introduction

Let Z_q^n denote the set of all n-tuples $(x_1, x_2, \ldots, x_n)$ with $x_i \in Z_q = \{0, 1, \ldots, q-1\}$. These n-tuples are also called *words*. The *hamming distance* $d_H(x,\ y)$ between two words $x, y \in Z_q^n$ is the number of coordinates in which they differ. A *covering code* C is a subset of Z_q^n. The elements of C are called *codewords*. The *covering radius* of C is the smallest integer R such that every $x \in Z_q^n$ is within Hamming distance R from at least one codeword of C. We let $K_q(n, R)$ denote the smallest possible cardinality in any code $C \subseteq Z_q^n$ with covering radius R. Few exact values for $K_q(n, R)$ are known.

Covering radius problems have been intensively studied in recent years [9, 15, 23, 28, 58, 68]. A special case, where $q = 3$ and $R = 1$, is known as the *football pool problem* [66]. Football pool is a gambling game popular in western Europe. Typically, one bets on the outcome of 13 football matches. Each match has three possible outcome: a win, a loss or a draw for the home team. Bets which correctly identify all 13 outcomes win the first prize and bets with exactly 12 correct predictions, the second prize. Because some teams are obviously stronger than some other teams, many bettors may believe that the outcomes of some matches are very likely. In

Matches	*Best known lower and upper bounds*	*Graph size*
1	1	3
2	3	9
3	5	27
4	9	81
5	27	243
6	63-73	729
7	147-186	2187
8	393-486	6561
9	1048-1341	19683
10	2814-3645	59049
11	7767-9477	177147
12	21395-27702	531441

Table 5: The best known upper bounds for the football pool problem of 1 to 12 matches.

order to win at least a second prize, they may want to place bets in a way that regardless of the outcome of the remaining matches, at least one of the bets will have twelve correct results. Technically, the football pool problem on n matches is to find the minimum number of bets required to guarantee at least one of the bets has $n-1$ correct results, regardless of the outcome of the matches.

In [38], Kamps and van Lint proved that the football pool problem of 5 matches has a minimum domination number equal to 27. Table 5 gives a summary of the best known upper bounds on the number of bets of the football pool problem from 1 to 12 matches [23, 37, 38, 57, 75].

6.2 Rook Domain Graph $\Gamma_{n,q}(V, E)$

Let $G(V, E)$ be a graph with vertex set V and edge set E. The *distance* between two vertices is the minimum length of a path between the two vertices. The distance between a vertex and itself is defined to be 0. If the two vertices are in disconnected components of the graph, the distance between them is infinity. In this construction, we only encounter graphs which are connected. Hence, the distance between two vertices in a graph is always finite.

A subset $S \subseteq V$ is a dominating set of *radius* R if every vertex $x \in V$ is at a distance R from at least one of the vertices in S. Given a graph G, a radius R, and a size t, the problem of finding whether there exists a dominating set of radius R and size t is called the *R-dominating set problem.* When R equals 1, it is equivalent to the dominating set problem defined in Chapter 2 on page 8.

We let $\Gamma_{n,q}(V, E)$ denote the *rook domain graph* where $V = Z_q^n$ and $E = \{(x, y) \mid x, y \in Z_q^n,\ d_H(x,\ y) = 1\}$. In this graph, the distance between two vertices is also the Hamming distance between the two vertices as n-tuples. Thus, there is no ambiguity in using $d_H(x,\ y)$ for both distances. When $q = 3$, we call the corresponding graph the *football pool graph* of n matches. Using the rook domain graph $\Gamma_{n,q}$, the problem of finding a covering code of radius R and size t in Z_q^n is the same problem as finding a dominating set of radius R and size t in the graph.

6.3 Symmetry group of $\Gamma_{n,q}(V, E)$

Let us now examine the graph $\Gamma_{n,q}(V, E)$. We shall show that the following operations are symmetries of $\Gamma_{n,q}(V, E)$.

1. Let ρ_i be a permutation of $\{0, \ldots, q-1\}$. Its action on Z_q^n is defined by

$$\rho_i((x_1, \ldots, x_n)) = (x_1, \ldots, x_{i-1}, \rho_i(x_i), x_{i+1}, \ldots, x_n), \text{for } (x_1, \ldots, x_n) \in Z_q^n.$$

The operation ρ_i *relabels* the i-th entries.

2. Let π be a permutation of $\{1, \ldots, n\}$. Its action on Z_q^n is defined by

$$\pi((x_1, \ldots, x_n)) = (x_{\pi(1)}, \ldots, x_{\pi(n)}), \text{for } (x_1, \ldots, x_n) \in Z_q^n.$$

The operation π *permutes* the entries.

It is clear that both ρ_i and π are permutations of the n-tuples in Z_q^n. We need to show that they preserve the edge set E. Let $(x, y) \in E$ with $x = (x_1, \ldots, x_n)$ and

$y = (y_1, \ldots, y_n)$. Since $d_H(x,y) = 1$, there exists a unique index j such that $x_j \neq y_j$. First, we consider ρ_i. If $i \neq j$, $\rho_i(x)$ and $\rho_i(y)$ again differ only in the j-th position. If $i = j$, the only place that $\rho_i(x)$ and $\rho_i(y)$ can differ is in the i-th position. Since ρ_i is a permutation, $\rho_i(x_i) \neq \rho_i(y_i)$ if $x_i \neq y_i$. Therefore, in both case $d_H(\rho_i(x), \rho_i(y)) = 1$ and $(\rho_i(x), \rho_i(y)) \in E$. As for $\pi, \pi(x)$ and $\pi(y)$ differs only in the $\pi(j)$ position, since x and y differs only in the j-th position. Therefore $(\pi(x), \pi(y)) \in E$.

As there are q numbers in $\{0, \ldots, q-1\}$, there are $q!$ permutations to relabel the entries in any position for all the n-tuples in Z_q^n. There are n entries, giving $(q!)^n$ operations of the first type. There are also $n!$ ways to permute the entries. Together, they generate the *wreath product* $S_q \wr S_n$ of order $(q!)^n n!$.

We now show that the symmetry group Gp of $\Gamma_{n,q}$ is exactly $S_q \wr S_n$. The proof is essentially the *sifting* operation of the *Schreier-Sims method* [63, 62]. Instead of using terminology from computational group theory, we shall use terms related to the structure of $\Gamma_{n,q}$.

Let $\bar{0}$ denote the zero word and let $Gp_{\bar{0}}$ denote the subgroup of Gp fixing the vertex $\bar{0}$. Let us establish some simple properties.

Lemma 6.1 *The group $S_q \wr S_n$ acts transitively on the vertices of $\Gamma_{n,q}$.*

Proof: Any vertex of $\Gamma_{n,q}$ can be relabeled as $\bar{0}$. Therefore $S_q \wr S_n$ acts transitively. □

Corollary 6.2 $|Gp| = q^n |Gp_{\bar{0}}|$.

Proof: By LaGrange's theorem, $|Gp| = |Gp_{\bar{0}}| \times [Gp : Gp_{\bar{0}}]$. Since $S_q \wr S_n$ is a subgroup of Gp, Lemma 6.1 implies that Gp also acts transitively on the vertices of $\Gamma_{n,q}$. Since $\Gamma_{n,q}$ has q^n vertices, the index $[Gp : Gp_{\bar{0}}]$ of $Gp_{\bar{0}}$ in Gp, which is equal to the size of the orbit of $\bar{0}$ under Gp, is q^n. □

The neighborhood of $\bar{0}$ is $N_{\bar{0}}$. Its size is $n(q-1)$. Let us give each of the vertex in $N_{\bar{0}}$ a name. For $i \neq 0$, let $ie_j = (0, \ldots, 0, i, 0, \ldots, 0)$, where i is in the j-th position. An important property of any vertex is its distances from these special vertices.

Proposition 6.3 *If* $x = (x_1, \ldots, x_n) \in Z_q^n$, *then*

$$d(x, ie_j) = \begin{cases} d(x, \bar{0}) + 1 & \text{if } x_j = 0, \\ d(x, \bar{0}) - 1 & \text{if } x_j = i, \text{ and} \\ d(x, \bar{0}) & \text{otherwise.} \end{cases}$$

Proof: The Proposition follows from the definition of a Hamming distance. □

If $g \in Gp_{\bar{0}}$, as $Gp_{\bar{0}}$ preserves edges, it takes neighbors to neighbors. Our next step is to show that g is completely determined by its action on the neighbors of $\bar{0}$.

Lemma 6.4 *Suppose* $g \in Gp$, $g(\bar{0}) = \bar{0}$, *and* $g((ie_j)) = i'e_{j'}$. *If* $g((x_1, \ldots, x_n)) = (y_1, \ldots, y_n)$, *and* $x_j = i$, *then* $y_{j'} = i'$.

Proof: Let $x = (x_1, \ldots, x_n)$ and $y = (y_1, \ldots, y_n)$. Since $g \in Gp$ and fixes $\bar{0}$, $d(x, \bar{0}) = d(y, \bar{0})$. Moreover, $d(x, ie_j) = d(y, i'e_{j'})$. If $x_j = i$, then, by Proposition 6.3,

$$d(x, ie_j) = d(x, \bar{0}) - 1 = d(y, \bar{0}) - 1.$$

But, $d(y, i'e_{j'}) = d(y, \bar{0}) - 1$ only if $y_{j'} = i'$. □

We now consider the action of g on the neighbors of $\bar{0}$. The next lemma states that this action is imprimitive with a block system $B_j = \{ie_j | 1 \le i < q\}$ for $1 \le j \le n$.

Lemma 6.5 *If* $g \in Gp_{\bar{0}}$ *and* $g(1e_j) = i'e_{j'}$, *then for* $2 \le i < q$, $g(ie_j) = ke_{j'}$ *for some* k.

Proof: If $2 \le i < q$, then $d(1e_j, ie_j) = 1$. Therefore, $d(i'e_{j'}, g(ie_j)) = 1$. Hence $g(ie_j)$ must also have its non-zero entry in the j'-th position; otherwise, the distance will be two. □

Lemma 6.6 *The size of the subgroup of* Gp *stabilizing* $\bar{0}$ *is at most* $((q-1)!)^n n!$.

Proof: The largest imprimitive group with a block system $B_j = \{ie_j \mid 1 \le i < q\}$ for $1 \le j \le n$ is $S_{q-1} \wr S_n$, whose size is $((q-1)!)^n n!$. □

Theorem 6.7 $Gp = S_q \wr S_n$.

Proof: Since $S_q \wr S_n$ is a subgroup of Gp, $|Gp|$ is at least $(q!)^n n!$. By Corollary 6.2 and Lemma 6.6, it is at most $(q!)^n n!$. □

A special class of symmetry is the *addition* operation. Take any $v \in Z_q^n$, the operation of adding v, modulo q, to all the elements of Z_q^n is a relabeling symmetry operation. Another special class of symmetry is the *multiplication* operation. Take any number x, the operation of multiplying x, modulo q, to all the elements of Z_q^n is another relabeling symmetry operation.

6.4 Generation of the Blokhuis-Lam Theorem

In [9], Blokhuis and Lam presented a construction method that is actually a special case of solving the weighted dominating set problem on the orbit graphs derived by assuming automorphism groups. We first describe briefly the Blokhuis-Lam Theorem. Let $A = (I; M)$ be an $r \times n$ matrix where I is the $r \times r$ identity matrix and M is an $r \times (n-r)$ matrix with entries from Z_q. A set $\mathcal{S} \subseteq Z_q^r$ is said to *R-cover* Z_q^r using A if every $s \in Z_q^r$ can be represented as a sum of one element of $\mathcal{S}$ and a Z_q-linear combination of at most R of the columns of A. A set $\mathcal{S}$ with this property is said to be an *R-covering.* The following theorem, was proved in [15, 70].

Theorem 6.8 *If S R-covers Z_q^r using A, then the code*

$$\mathcal{C} = \{c \in Z_q^n | Ac \in \mathcal{S}\},$$

having $|\mathcal{S}|q^{n-r}$ elements, covers Z_q^n with radius at most R.

The advantage of this construction method is that it replaces the search for a covering code in Z_q^n by one in the smaller Z_q^r. For example, if one wants to find a covering code of radius 1 with 72 elements in Z_3^6, then we can either try to find an 1-covering $\mathcal{S}$ with 24 elements in Z_3^5 or one with 8 elements in Z_3^4. This example illustrates two problems with using this method.

1. The size of the covering code constructed is always divisible by q. There is no reason why $K_q(n, R)$ has to be divisible by q.

2. It is not clear how to construct all the non-equivalent matrices A that are used in the construction.

We generalized the above construction by replacing the role of an R-covering with an assumed automorphism group. The covering code constructed is a union of orbits and its size need not be divisible by q, which addresses the first problem. We also establish that one need only consider automorphism groups up to conjugacy, which addresses the second problem. First, we show that the Blokhuis-Lam method is a special case of using an orbit graph.

Let $c_1, \ldots, c_{n-r}$ be the columns of M. These columns are extended to n-tuples by appending systematically $(n-r-1)$ 0's and one $(q-1)$. More precisely, c_i' is obtained from c_i by appending $(i-1)$ 0's, then $(q-1)$, followed by another $(n-r-i)$ 0's. Note that the c_i''s are chosen so that $Ac_i' = c_{i1} + \ldots + c_{ir} + (q-1)c_{i1} + \ldots + (q-1)c_{ir} = q(c_{i1} + \ldots + c_{ir}) = 0$. Hence, if x is in the code $\mathcal{C}$ constructed by Theorem 6.8, so is $x + c_i'$ for any i. In other words, c_i' stabilizes the code $\mathcal{C}$ setwise. So, the group H generated by using the c_i''s as addition operators is contained in the automorphism group of $\mathcal{C}$. The group H is isomorphic to a direct product of $n-r$ copies of the cyclic group with q elements C_q. The order of H is q^{n-r} and all its orbits are of full length q^{n-r}. Thus, the orbit graph has q^r vertices and all their weights are equal to q^{n-r}. The weighted dominating set problem on the orbit graph for H must have a weighted size divisible by q^{n-r}. By factoring out q^{n-r} from both the weights and the weighted size, the weighted dominating set problem on the orbit graph simplifies to just the dominating set problem on the same graph.

6.5 Orbit Graphs of $\Gamma_{n,3}(V, E)$

The most interesting rook domain graphs are the football pool graphs. Let us see how, by using Theorem 5.2 on page 60, the weighted dominating set of an orbit graph can be used to find solutions to the football pool problem. As an example, we consider the

Vertex in orbit graph	*Orbit in* $\Gamma_{4,3}$					
1	0000	1120	2210			
2	0001	1121	2211	0002	2212	1122
3	0010	1100	2220	0020	2200	1110
4	0011	1101	2221	0022	2202	1112
5	0012	1102	2222	0021	2201	1111
6	0100	1220	2010	0200	2110	1020
7	0101	1221	2011	0202	2112	1022
8	0102	1222	2012	0201	2111	1021
9	0110	1200	2020	0220	2100	1010
10	0111	1201	2021	0222	2102	1012
11	0112	1202	2022	0221	2101	1011
12	0120	1210	2000	0210	2120	1000
13	0121	1211	2001	0212	2122	1002
14	0122	1212	2002	0211	2121	1001

Table 6: The list of vertices in G_{orb} and their corresponding orbits in $\Gamma_{4,3}$.

football pool problem on 4 matches. It is well known that $K_3(4,1) = 9$. Consider the subgroup $H =< \sigma_1, \sigma_2 >$, where σ_1 is the symmetry obtained by adding $(1,1,2,0)$ modulo 3 to every vertex $(x_1,\ldots,x_4) \in Z_3^4$, and σ_2 is the symmetry obtained by multiplying $(x_1,\ldots,x_4)$ by 2 modulo 3. The subgroup H partitions the vertices of $\Gamma_{4,3}$ into 14 orbits, one of size 3 and thirteen of size 6. Table 6 shows the 14 orbits. The orbit graph of H, G_{orb}, has 14 vertices, one with weight 3 and thirteen with weight 6. Its adjacency matrix is given in Table 7 on page 72. To find a dominating set of size 9 in $\Gamma_{4,3}$, we need to find in the orbit graph a dominating set using the vertex of weight 3 and another vertex of weight 6. Indeed, such a dominating set exists, namely vertices 1 and 10 of G_{orb}, or the two orbits containing the vertex $(0,0,0,0)$ and the vertex $(0,1,1,1)$.

Each additional match in the football pool problem will lead to a football pool graph that has three times more vertices than the previous one. The program that uses the symmetry partitioning algorithm can solve the football pool problem up to 4 matches without transforming the football pool graph into orbit graphs. For higher

v	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	1	1	1	0	0	1	0	0	0	0	0	1	0	0
2	1	1	0	1	1	0	1	1	0	0	0	0	1	1
3	1	0	1	1	1	1	0	0	1	0	0	1	0	0
4	0	1	1	1	1	0	1	0	0	1	1	0	0	1
5	0	1	1	1	1	0	0	1	0	1	1	0	1	0
6	1	0	1	0	0	1	1	1	1	0	0	1	0	0
7	0	1	0	1	0	1	1	1	0	1	1	0	1	0
8	0	1	0	0	1	1	1	1	0	1	1	0	0	1
9	0	0	1	0	0	1	0	0	1	1	1	1	0	0
10	0	0	0	1	1	0	1	1	1	1	1	0	1	1
11	0	0	0	1	1	0	1	1	1	1	1	0	1	1
12	1	0	1	0	0	1	0	0	1	0	0	1	1	1
13	0	1	0	0	1	0	1	0	0	1	1	1	1	1
14	0	1	0	1	0	0	0	1	0	1	1	1	1	1

Table 7: The adjacency matrix of orbit graph G_{orb} for $\Gamma_{4,3}$.

number of matches, we try to find the dominating sets by generating the orbit graphs of the football pool graphs. Based on Theorem 5.3 on page 60, when constructing the orbit graphs, we only need to generate subgroups H of Gp up to conjugacy. A larger H would mean a smaller orbit graph and one whose weighted dominating set problem is easier to solve. However, a good covering code may not have such a large automorphism group. We decided to consider first all cyclic subgroups H of the football pool graphs. In other words, we find a representative h from each conjugacy class of elements of Gp. For each element h, we generate $H = \langle h \rangle$, and then its orbit graph.

We have generated such orbit graphs for the football pool problem of 5, 6 and 7 matches using MAGMA. $\Gamma_{5,3}$ has 243 vertices. The elements of its symmetry group are partitioned into 107 conjugacy classes. The number of vertices in the resulting orbit graphs ranges from 9 to 162. For $\Gamma_{6,3}$, it has 729 vertices. The elements of its symmetry group are partitioned into 220 conjugacy classes. The number of vertices in the resulting orbit graphs ranges from 24 vertices to 486 vertices. For $\Gamma_{7,3}$, it has 2187 vertices. Its symmetry group has 428 conjugacy classes of elements. The orbit

Number of matches	*Number of orbit graphs*	*Biggest orbit graphs*	*Orbit graphs with* <100 *vertices*
5	107	162	96
6	220	486	84
7	428	1458	29

Table 8: The results of the orbit graphs for the football pool problem of 5, 6 and 7 matches.

graphs have sizes ranging from 33 to 1458 vertices.

The orbit graphs for the football pool problem of 5 to 7 matches is summarized in the following two tables. Table 8 gives a summary of number of orbit graphs, the number of vertices in the largest orbit graph, and the number of orbit graphs that has fewer than 100 vertices. Orbit graphs with fewer than 100 vertices are singled out because there are small enough for the symmetry partitioning algorithm to do an exhaustive search. Table 9 on page 74 gives a summary of the orbit graphs for the football pool problem of 6 matches grouped by the order of the subgroups generated from the representatives of the conjugacy classes.

Since the smallest unsolved case is one with 6 matches, its orbit graphs may be useful in constructing a better upper bound. Therefore, we will provide a description of each of the 220 orbit graphs in Appendix A.

Table 5 on page 65 gave a summary of the best known upper bounds on the number of bets of the football pool problem from 1 to 12 matches. We attempt to match or better these values using our orbit graphs. For 5 matches, our program was able to find a solution with no difficulty. We have found 48 solutions in one of the orbit graphs. In [41], Kolev proves that there is one non-isomorphic solution to the football pool problem of 5 matches. As we cannot finish searching for a dominating set for the football pool problem of 5 matches without breaking down the football pool graph into orbit graphs, we cannot verify his result using our program.

For 6 and 7 matches, we choose to try only those graphs with fewer than 100 vertices because of resources limitations. Not all the orbit graphs can have a weighted

Subgroup order	*Number of orbit graph*	*Minimum number of vertices*	*Maximum number of vertices*
2	15	365	486
3	11	243	297
4	19	158	243
5	1	153	153
6	89	122	198
8	5	93	99
9	6	81	81
10	3	77	102
12	45	62	99
15	3	51	51
18	14	42	54
24	4	32	33
30	2	26	34
36	3	24	27

Table 9: The orbit graphs of the football pool problem of 6 matches grouped by the order of the subgroup generated from the representative of the conjugacy class.

dominating set that satisfies the weighted size. In order to satisfy the weighted size, equation (5) on page 62 has to be satisfied. For example, if each orbit in the orbit graph has a weight equal to three, its weighted size has to be a multiple of three. For 8 matches, we could generate only one orbit graph that had a size less than 100. It had vertices with weights 9, 18 and 90. We found no weighted dominating set using the best known upper bound as test size for this orbit graph. So far we have not found any better upper bounds for 6, 7, or 8 matches. However, as described in the next section, we applied our weighted dominating set and orbit graph theories on CPLEX and found several solutions which matches the best known upper bounds for the football pool problem of 6 and 7 matches, and they will be described in some detail later in this chapter.

6.6 Using CPLEX

In order to handle larger orbit graphs, we may use *simulated annealing* or *tabu search* which had produced practically all published upper bounds [23, 37, 38, 57, 75]. However, this is left as future work to be done. In order to demonstrate the usefulness of the orbit graph approach, we treat the weighted dominating set problem as an integer programming optimization problem and use the commercial linear programming package CPLEX Mixed Integer Optimizer to solve it. We first show how we construct the inequalities and the objective function in CPLEX to solve the weighted dominating set problem of an orbit graph.

Given an orbit graph $G_{orb}(V_{orb}, E_{orb})$, let w_i be the weight of the vertex v_i in the orbit graph and let the test size be t. For each vertex v_i in the vertex set of G_{orb}, we define a variable x_i such that

$$x_i = \begin{cases} 0 & \text{if } v_i \text{ is off} \\ 1 & \text{if } v_i \text{ is on.} \end{cases}$$

The following equations are the function and inequalities input to CPLEX:

Objective function (to be minimized):

$$\sum_{i=1}^{|V_{orb}|} w_i x_i. \tag{6}$$

For each $v_i \in V_{orb}$, we set up the following set of inequalities:

$$\forall v_j \in V_{orb},\ (\sum_{(v_i,v_j)\in E_{orb}} x_j) + x_i \geq 1. \tag{7}$$

An inequality to limit the dominating set with size less than or equal to t:

$$\sum_{i=1}^{|V_{orb}|} w_i x_i \leq t. \tag{8}$$

Function (6) is the objective we would like to minimize. It leads us to the minimum domination number, if CPLEX can complete the search. The set of inequalities in (7) is the constraints that would guarantee the vertices in the dominating set cover V_{orb}. Each inequality is related to a vertex v_i in V_{orb}, and it constrains the vertex v_i to be covered. Finally, inequality (8) limits the dominating set to have a size less than or equal to t.

Using CPLEX for the orbit graphs of 6 matches, we found a dominating set of size 73 from one of the orbit graphs and several dominating sets of size 77, 78 and higher. For the football pool problem of 7 matches, we found several dominating sets of size 186 with large automorphism groups.

6.7 Dominating Sets for $\Gamma_{6,3}(V, E)$ and $\Gamma_{7,3}(V, E)$

In this section, we present some of the dominating sets we found for $\Gamma_{6,3}(V, E)$ and $\Gamma_{7,3}(V, E)$. Table 10 on page 78 gives the dominating set of size 73 using the orbit graph 212 of the football pool graph of 6 matches. The orbit graph is obtained by assuming a cyclic automorphism group of size 4 generated by the element $3_0 4_0 2_0 1_0 5_3 6_0$. See Appendix A for an explanation of the compact form in which the element is presented. The weighted dominating set contains 207 vertices, 162 of weight 4, 36 of weight 2 and 9 of weight 1. The automorphism group of the weighted dominating set of the orbit graph has a size of 2 and the automorphism group of the dominating set of the original graph is 8. Tables 11, 12 and 13 on pages 79 to 81 give the dominating sets obtained from the orbit graphs 209, 255 and 313 of the football pool graph of 7 matches. The generators for these three orbit graphs are given in Table 21 of Appendix A on page 107. The automorphism groups of the weighted dominating sets of the orbit graphs of 209, 255 and 313 have a size of 4, 2, and 4 respectively. The respective automorphism groups of the dominating sets of the original graph are 48, 72, and 48. We used a colored graph isomorphism program to verify that the three dominating sets are non-isomorphic.

We note that in [76], Wille used an approximation method called simulated annealing to find a dominating set of size 73 for the football pool problem of 6 matches. Since the dominating set of 6 matches found by Wille, has a trivial automorphism group, it is impossible for our program to find his solution as a weighted dominating set in any of the orbit graphs. In [56], Östergård also find a dominating set of size 73. The automorphism group of the dominating set has a size 8. It is different from the one we obtained.

In [68], van Laarhoven et al. make use of simulated annealing and the Blokhuis-Lam theorem to find two dominating sets of size 186 for the football problem of 7 matches. The automorphism groups of the two dominating sets are 6 and 4. Compare with the automorphism groups of the three dominating sets we obtained, our solutions have more symmetry. Most people tend to like solution with more symmetry. Symmetry solution tends not to be random and further investigation may reveal

Orbit Graph Vertex	*Representative in Original Graph*	*Orbit Size*
1	000001	1
2	000002	1
12	000120	4
26	001102	2
29	001112	2
32	001122	2
45	002210	2
49	002221	2
53	010102	4
64	010211	4
78	011200	4
82	011211	4
96	012100	4
105	012200	4
109	012211	4
110	012212	4
122	020222	4
135	021210	4
142	022101	4
160	111101	1
161	111102	1
177	112210	2
181	112221	2
188	121212	4
203	222202	1
207	222220	2

Table 10: The dominating set of size 73 obtained from orbit graph 212 of the football pool graph of 6 matches.

Orbit Graph Vertex	*Representative in Original Graph*	*Orbit Size*
18	0000200	12
23	0000212	12
34	0001021	6
35	0001022	6
48	0001210	12
67	0002111	12
85	0010021	6
86	0010022	6
93	0010210	12
105	0011200	12
156	0022100	6
157	0101111	12
158	0101112	12
163	0100001	3
164	0100002	3
165	0100010	3
175	0102001	3
176	0102002	3
177	0102010	3
187	0111001	3
188	0111002	3
189	0111010	3
196	0112121	12
203	0120122	12
205	0121001	3
206	0121002	3
207	0121010	3

Table 11: The dominating set of size 186 obtained from orbit graph 209 of the football pool graph of 7 matches.

Orbit Graph Vertex	*Representative in Original Graph*	*Orbit Size*
10	0000101	6
11	0000102	12
22	0001011	12
36	0001211	3
45	0002100	12
46	0002101	6
59	0010012	12
71	0010200	12
81	0011010	12
97	0011210	12
117	0012120	12
125	0012222	12
130	0020021	6
155	0022211	6
161	0100010	12
173	0101200	12
183	0110201	12
185	0110211	3
202	0200021	6
206	0221211	3
207	0222211	3

Table 12: The dominating set of size 186 obtained from orbit graph 255 of the football pool graph of 7 matches.

Orbit Graph Vertex	*Representative in Original Graph*	*Orbit Size*
27	0001000	6
38	0001102	3
39	0001122	6
40	0001111	3
41	0001120	6
43	0002001	6
47	0002012	6
53	0010002	12
55	0010011	12
78	0011000	6
92	0011120	6
122	0021102	3
123	0021110	6
124	0021111	3
125	0021120	6
145	0101001	12
153	0101100	12
177	0110120	12
179	0110122	12
185	0111012	12
192	0111210	12
196	0111221	12
203	0121012	12

Table 13: The dominating set of size 186 obtained from orbit graph 313 of the football pool graph of 7 matches.

the pattern underlying it. Furthermore, a solution with more symmetry may provide more information for analysis, and it is also easier to be stored either in printed form or on the hard disk. Moreover, it is also easier to describe to punters how the betting scheme works.

Chapter 7

Performance

In this chapter, we compare the performance of two programs, one with pruning due to symmetry and one without.

7.1 Symmetry versus No Symmetry

In this section, we compared the performance of the program P_{sym}, which uses symmetry, with program P_{no_sym}, which does not use symmetry. The program P_{no_sym} uses only the even-splitting partitioning algorithm to find a dominating set. For more details about the even-splitting partition algorithm, see Section 2.2.6 on page 33. We have tested both P_{no_sym} and P_{sym} with the 7×7 grid graph $grid_{7\times 7}$ using a test size 9, the football pool graph of 4 matches f_4 using test sizes 11 and 12, and an orbit graph with 69 vertices for the football pool graph of 5 matches $f_{5_{orb69}}$ using test sizes 25, 26, 27 and 28. The results are summarized in Tables 14, 15, and 16 on page 84. In each table, the *group size* is the size of the automorphism group of the graph. The *total time* is the CPU time taken to finish running the program. The *total nodes* is the total number of nodes in the search tree. The *time per node* is obtained by dividing the *total time* by the *total nodes.* Finally, the *time ratio* is the *total time* of P_{no_sym} divided by the *total time* of P_{sym} and the *nodes ratio* is the is the *total nodes* of P_{no_sym} divided by the *total nodes* of P_{sym}.

Theoretically, the size of the search tree in P_{no_sym} can be reduced by a factor

		P_{no_sym} without symmetry			P_{sym} with symmetry				
Test size	*Group size*	*Total time*	*Total nodes*	*Time per node*	*Total time*	*Total nodes*	*Time per node*	*Time ratio*	*Nodes ratio*
11	8	1.68	2.35E3	7.09E-4	0.117	29	2.87E-3	14	81
12	8	60.3	7.46E4	8.08E-4	13.1	1.89E4	1.62E-4	4.6	4.0

Table 14: Performance of P_{no_sym} and P_{sym} for the 7×7 grid graph.

		P_{no_sym} without symmetry			P_{sym} with symmetry				
Test size	*Group size*	*Total time*	*Total nodes*	*Time per node*	*Total time*	*Total nodes*	*Time per node*	*Time ratio*	*Nodes ratio*
9	31104	6.88E4	2.02E7	3.41E-3	1.40	231	5.56E-3	4.9E4	8.7E4

Table 15: Performance of P_{no_sym} and P_{sym} for the football pool graph of 4 matches.

		P_{no_sym} without symmetry			P_{sym} with symmetry				
Test size	*Group size*	*Total time*	*Total nodes*	*Time per node*	*Total time*	*Total nodes*	*Time per node*	*Time ratio*	*Nodes ratio*
25	48	6.20E2	1.98E5	3.13E-3	2.52	988	2.48E-3	246	201
26	48	8.99E2	2.42E5	3.72E-3	15.2	5.55E3	2.72E-3	59	44
27	48	2.01E3	5.29E5	3.80E-3	47.2	1.53E4	3.09E-3	43	35
28	48	7.77E3	1.99E6	3.91E-3	1.87E2	5.65E4	3.30E-3	42	35

Table 16: Performance of P_{no_sym} and P_{sym} for an orbit graph with 69 vertices derived from the football pool graph of 5 matches.

Content partition number	*Cell* B_1	*Cell* B_2
i	1	1
ii	2	0
iii	0	2

Table 17: The content partitions of the 3×3 grid graph using program P_{no_sym} with test size 2.

at most equal to the size of the automorphism group of the input graph. Since the program does not necessarily make use of the whole group to reduce the search tree, the reduction may not be optimum. In general, if the size of the automorphism group is large enough, the time needed to process a node in the search tree of P_{sym} would be greater than in P_{no_sym} because of the group related calculations involved. One may be surprised to see that in several cases, the size of the search tree in P_{sym} is reduced by a factor greater than the size of the automorphism group of the input graph. The reason lies in how the two programs refine a graph. The program P_{no_sym} which uses the even-splitting algorithm will refine each cell arbitrary into two subcells with size which differs by at most one. On the other hand, the program P_{sym} often refines the graph into orbits according to the symmetry group. As the following example shows, this tends to lead to a more restricted content partition, and hence, a smaller search tree.

Consider the 3×3 grid graph in Figure 11 on page 54. Suppose the test size is equal to 2. Program P_{no_sym} will refine the graph into two cells, namely, $B_1 = \{a, b, c, d\}$ and $B_2 = \{e, f, h, i, j\}$. Table 17 shows all the content partitions of these two cells using test size 2. Among these three content partitions, only case (i) passes the coverage and wastage tests. In case (iii), cell B_1 does not have enough coverage, and in case (ii), cell B_2 does not have enough coverage. Figure 13 on page 86 shows the cell refinement for the 3×3 grid graph with a test size of 2 using program P_{no_sym}. The levels shown in the figure are the levels of cell refinement. Only the content partitions that pass the coverage and the wastage tests are shown in the figure. When the cell B_1

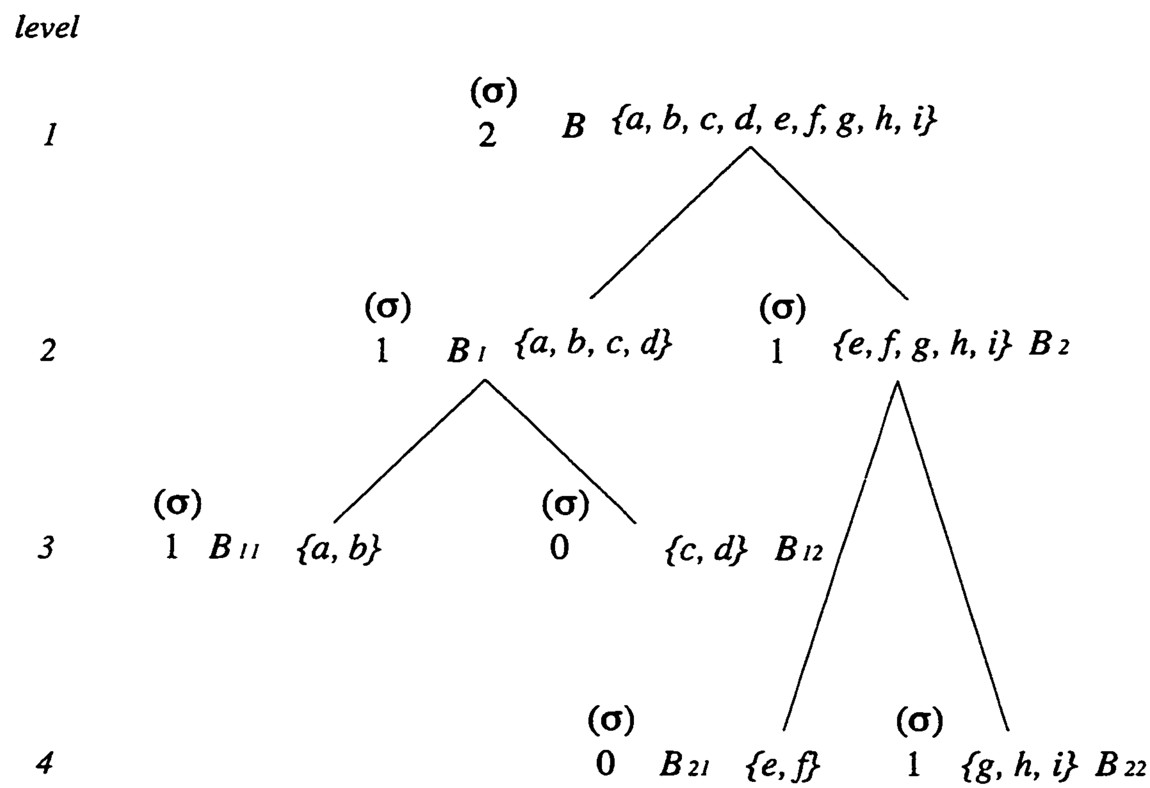


Figure 13: Refinement of the 3 × 3 grid graph with a test size 2 using program P_{no_sym}.

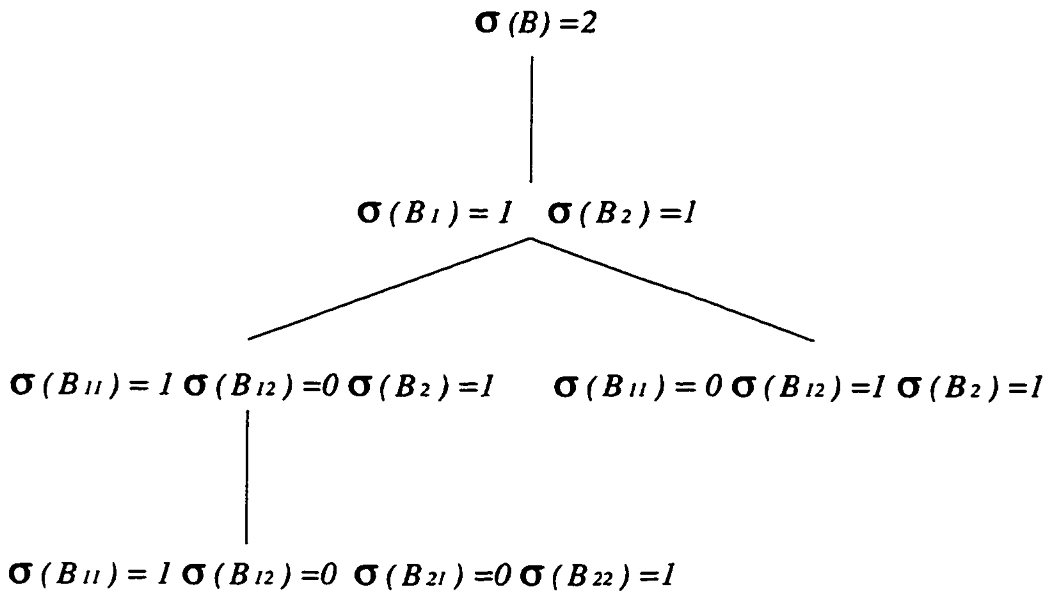


Figure 14: Search tree of the 3 $\times$ 3 grid graph with a test size 2 using program P_{no_sym}.

is refined into two cells $B_{11} = \{a,\ b\}$ and $B_{12} = \{c,\ d\}$, there are two possible content partitions. Choosing the content partition $\sigma(B_{11}) = 1$, $\sigma(B_{12}) = 0$, we then refine B_2 into $B_{21} = \{e,\ f\}$ and $B_{22} = \{g,\ h,\ i\}$. The only successful content partition is $\sigma(B_{21}) = 0$, $\sigma(B_{22}) = 1$. The program stops at this level because no more cells refined can have a content partition that will pass the wastage and coverage tests. When the program backtrack to its previous level, the content partition $\sigma(B_{11}) = 0$, $\sigma(B_{12}) = 1$ is used. Once again we refine B_2 into $B_{21} = \{e,\ f\}$ and $B_{22} = \{g,\ h,\ i\}$ but this time none of the two content partitions are feasible. Both have failed the coverage and wastage tests. Hence, the number of successful nodes in the search tree is 5. Figure 14 shows the search tree for the 3 $\times$ 3 grid graph with a test size of 2 using program P_{no_sym}. Only successful content partitions are shown in the figure.

Program P_{sym} will refine the graph into three cells according to its orbits. The three cells are $A_1 = \{a,\ c,\ g,\ i\}$, $A_2 = \{b,\ d,\ f,\ h\}$, and $A_3 = \{e\}$. Table 18 shows all the content partitions for a test size of 2. Among these five content partitions, none passes the coverage test, because at least one of the cells does not have enough coverage. Cell A_1 does not have enough coverage by partitions (i), (ii), (iii) and (iv). Cell A_2 does not have enough coverage by partitions (i) and (v). Cell A_3 does not

Content partition number	*Cell* A_1	*Cell* A_2	*Cell* A_3
i	1	1	0
ii	1	0	1
iii	0	1	1
iv	2	0	0
v	0	2	0

Table 18: The content partitions of the 3×3 grid graph using program P_{sym} with test size 2.

have enough coverage by partition (iv). Hence, the program will stop once all these 5 content partitions were tried. The number of successful nodes in the search tree is 1. This example shows that a good refinement of the graph can lead to a significant reduction in the search tree, especially when the test size is close to the minimum domination number.

Chapter 8

Conclusions

8.1 Summary of Work Done

In this thesis, we presented a partitioning algorithm to obtain all the dominating sets of a graph. It makes use of the coverage test, the wastage test, and the graph's symmetry group to cut down the search tree. If symmetry exists in the graph, isomorphic branches of the search tree can be pruned leading to an efficient algorithm. Based on this algorithm, a program P_{sym} has been implemented. The program P_{sym} provides us with a powerful tool to analyze graphs. Its behaviour can be varied by changing some input parameters. For example, one can investigate the effect on the estimated running time by changing the manner in which cells are refined.

The program was applied to grid graphs and football pool graphs. We were able to find the minimum dominating sets of any $m \times n$ grid graph for $m \leq 10$ and $n \leq 11$. Our results in the grid graphs and football pool graphs agree with the work done by other people and this strongly indicates the correctness of our program. For the football pool problem on n matches, we have determined that its automorphism group is $S_3 \wr S_n$.

By assuming an automorphism group on the dominating set, we can partition the graph according to its orbits under the group. By mapping the original graph to a graph with orbits as its nodes, we obtain a new graph of a smaller size. By assigning a weight on each vertex in the orbit graph, finding a dominating set is equivalent to finding a weighted dominating set in the orbit graph. This gives us a powerful tool to break down a graph into orbit graphs and to increase the chance of finding a better dominating set. Furthermore, by passing each orbit graph into a different computer, we can solve the weighted dominating set problem in parallel. This new approach to solve the dominating set problem has allowed us to match the known domination numbers for the football pool problem of 5, 6, and 7 matches. Our solutions for 7 matches have larger symmetries than the known ones. We also generated a set of orbit graphs for the football pool problem of 5, 6, and 7 matches by considering only cyclic subgroups up to conjugacy. Based on the results we obtained from the football graph of 6 and 7 matches, there is hope that some of the unsolved orbit graphs can lead to smaller dominating sets.

8.2 Future Work

Since the program we developed is a general purpose program, programs that are specialized to a specific type of graph may perform better than our program. Fortunately, there are still lots of rooms for improvement. The program may be improved in several ways. Based on several profiles we have done using *prof*, if one can optimize the `Get_UIF` procedure and several procedures that have group related operations (for example, the procedure that does group intersection), the program may be able to run faster. Much research can be work on reducing the time of those heavily used operations. The profile also shows that in some cases the number of unsuccessful content partitions are much higher than the successful ones and the total number of content partitions is big. If we can reject those cases without doing the coverage and wastage tests, we should be able to speed up the program.

Usually, the program can only handle orbit graphs with less than 100 vertices for a football pool graph, one can make use of the non-cyclic subgroups of the automorphism group of a graph to obtain more orbit graphs within that size. Because approximation methods such as simulated annealing, when used with certain construction method [9, 15, 37, 42, 68, 70], has proved useful in constructing many good covering codes, we may as well try to incorporate approximation methods with the idea of orbit graph to improve the upper bounds of the football pool problem. We can also limit the search to a certain portion of the search tree at each level of the graph partition. This will allow the program to work on large graphs. Another approach is to modify the program so that it can randomly select a portion of the nodes at each level of the search tree and run this program a certain number of times to see if we can obtain a dominating set.

Other than the above methods, another way is to recursively generate orbit graphs from orbit graphs. This allows one to work on large orbit graphs. Finally, we can try to develop a parallel algorithm and combine all the above approaches to solve the dominating set problem.

Bibliography

[1] E.H.L. Aarts, and J. Korst. *Simulated Annealing and Boltzmann Machines.* John Wiley and Sons, Chichester, 1990.

[2] E.H.L. Aarts, and van Laarhoven. *Simulated Annealing: An Introduction.* Statistica Neerlandica, 43, 1988.

[3] E.H.L. Aarts, and van Laarhoven. *Statistical Cooling: A General Approach to Combinatorial Optimization Problems.* Philips Journal of Research, 40, pp. 193–226, 1985.

[4] M.A. Armstrong. *Groups and Symmetry.* Undergraduate Texts in Mathematics, Springer-Verlag, 1988.

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

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

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

[8] A.A. Bertossi, and S. Moretti. *Parallel Algorithms on Circular-Arc Graphs.* Information Processing Letters, 33, no. 6, pp. 275–281, 1990.

[9] A. Blokhuis, and C.W.H. Lam. *More Coverings by Rook Domains.* J. Comb. Theory, Series A, vol. 36, pp, 240–244, 1984.

[10] W. Bosma, and J. Cannon. *Handbook of MAGMA Functions.* School of Mathematics and Statistics, University of Sydney, Australia, 1995.

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

[12] G. Butler. *Fundamental Algorithms for Permutation Groups.* Lecture Notes in Computer Science, 559, Springer-Verlag, 1991.

[13] G. Butler, C.W.H. Lam, and K.L. Ma. *Constructing Covering Codes via Automorphisms.* Computer Science Department, Concordia University, Canada, 1996 (to appear).

[14] J. Cannon, and C. Playoust. *An Introduction to MAGMA.* School of Mathematics and Statistics, University of Sydney, Australia, 1993.

[15] W.A. Carnielli. *Hyper-Rook Domain Inequalities.* Stud. Appl. Math., vol. 82, pp. 59–69, 1990.

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

[17] T.Y. Chang, and W.E. Clark. *The Domination Numbers of the* $5 \times n$ *and* $6 \times n$ *Grid Graph.* Journal of Graph Theory, 17, no.1, pp. 81–107, 1993.

[18] T.Y. Chang, W.E. Clark, and E.O. Hare. *Domination Numbers of Complete Grid Graphs, I.* Ars Combinatoria, vol. 38, pp. 97–111, 1994.

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

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

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

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

[23] H. Fernandes, and E. Rechtschaffen. *The Football Pool Problem for 7 and 8 Matches.* Journal of Combinatorial Theory, Series A, 35, pp. 109–114, 1983.

[24] P. Flach, and L. Volkmann. *Estimations for the Domination Number of a Graph.* Discrete Mathematics, 80, no. 2, pp. 145–151, 1990.

[25] M.R. Garey, and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, New York, 1979.

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

[27] H. O. Hämäläinen, I. S. Honkala, M. K. Kaikkonen and S. N. Litsyn. *Bounds for Binary Multiple Covering Codes.* Designs, Codes and Cryptography, vol. 3, pp. 251–275, 1993.

[28] H. O. Hämäläinen, I. S. Honkala, S. N. Litsyn, and P. Östergård. *Football Pools — A Game for Mathematicians.* American Mathematical Monthly, 102, vol. 7, pp. 579–588, 1995.

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

[30] F. Harary, and T.W. Haynes. *Conditional Graph Theory IV: Dominating Sets.* Utilitas Mathematica 48, pp. 179–192, 1995.

[31] E.O. Hare, and S.T. Hedetniemi. *A Linear Algorithm for Computing the Knight's Domination Number of a $K \times N$ Chessboard.* Congressus Numerantium 59, pp. 115–130, 1987.

[32] E.O. Hare, S.T. Hedetniemi, and W.R. Hare. *Algorithms for Computing the Domination Number of $k \times n$ Complete grid graphs.* Congressus Numerantium, 55, pp. 81–92, 1986.

[33] 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, 1981.

[34] S.T. Hedetniemi, and R.C. Laskar. *Topics on Domination.* Discrete Mathematics, 86, 1990.

[35] I. Honkala. *On Lengthening of Covering Codes.* In "A collection of contributions in honor of Jack van Lint", Ed. P. J. Cameron and H. C. A. van Tilborg, North-Holland, Amsterdam, pp. 291–295, 1992.

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

[37] 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.

[38] 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.

[39] S. Kirkpatrick, C.D. Gelatt Jr, and M.P. Vecchi. *Optimization by Simulated Annealing.* Science, 220, pp. 671–680, 1985.

[40] D.E. Knuth. *Estimating the Efficiency of Backtrack Programs.* Mathematics of Computations, 29, pp.121-136, 1975.

[41] E. Kolev. *Codes of GF(3) of Length 5, 27 Codewords, and Covering Radius 1.* Journal of Combinatorial Designs, vol. 1, no. 4, pp. 256–276, 1993.

[42] K.U. Koschnick. *A New Upper Bound for the Football Pool Problem for Nine Matches.* Journal of Combinatorial Theory, Series A, 62, no. 1, pp. 162–167, 1993.

[43] C.W.H. Lam. *Computational Combinatorics - A Maturing Experimental Approach.* Concordia University, Canada, 1992.

[44] C.W.H. Lam and L.H. Thiel. *Backtrack Search with Isomorph Rejection and Consistency Check.* J. of Symbolic Computation, 7, pp. 473–485, 1989.

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

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

[47] K.L. Ma *Partition Algorithm for the Dominating Set Problem.* Master Thesis, Department of Computer Science, Concordia University, Canada, 1990.

[48] K.L. Ma, and C.W.H. Lam. *Partition Algorithm for the Dominating Set Problem.* Congressus Numerantium, 81, pp. 69–80, 1991.

[49] D. Marcu. *An Upper Bound for the Domination Number of a Graph.* Mathematica Scandinavica, 59, no. 1, pp. 41–44, 1986.

[50] D. Marcu. *Another Upper Bound for the Domination Number of a Graph.* Revista Colombiana de Mathematicas, 20, no. 1-2, pp. 51–55, 1986.

[51] D. Marcu. *Note on the Domination Number of a Graph.* Demonstratio Mathematica, 26, no. 1, pp. 103–105, 1993.

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

[53] B.D. McKay. *nauty User's Guide.* The Australian National University, Dept. of Comp. Sc., Technical Report, ver. 1.5.

[54] M. Morley, and G.H.J. van Rees. *Lottery Schemes and Covers.* Utilitas Mathematica 37, pp. 159–166, 1990.

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

[56] P.R.J. Östergård. *A Combinatorial Proof for the Football Pool Problem for Six Matches.* Journal of Combinatorial Theory, Series A, 76, pp. 160–163, 1996.

[57] P.R.J. Östergård. *Constructing Covering Codes by Tabu Search.* Journal of Combinatorial Designs, vol. 5, pp. 71–80, 1997.

[58] P.R.J. Östergård. *New Upper Bounds for the Football Pool Problem for 11 and 12 matches.* Journal of Combinatorial Theory, Series A, 67, no. 2, pp. 161–168, 1994.

[59] A.K. Parekh. *Analysis of a Greedy Heuristic for Finding Small Dominating Sets in Graphs.* Information Processing Letters, 39, no. 5, pp. 237–240, 1991.

[60] A.S. Rao, and C.P. Rangan. *Optimal Parallel Algorithms on Circular-Arc Graphs.* Information Processing Letters, 33, no. 3, pp. 147–156, 1989.

[61] L.A. Sanchis. *Maximum Number of Edges in Connected Graphs with a given Domination Number.* Discrete Mathematics, 87, no. 1, pp. 65–72, 1991.

[62] C.C. Sims. *Computation with Permutation Groups.* In S.R. Petrick, editor, Proc. 2nd Symp. on Symbolic and Alg. Manipulation, Los Angeles, pp. 23–28, 1971.

[63] C.C. Sims. *Computational Methods in the Study of Permutation Groups.* In J. Leech, editor, Computational Problems in Abstract Algebra, Pergamon, Elmsford, N.Y., pp. 169–183, 1970.

[64] H.G. Singh, and R.P. Pargas. *A Parallel Implementation for the Domination Number of a Grid Graph.* Congressus Numerantium, 59, pp. 297–311, 1987.

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

[66] O. Taussky and J. Todd. *Some Discrete Variable Computations.* Proc. Symp. Appl. Math., 10, pp. 204–205, 1960.

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

[68] P.J.M. van Laarhoven, E.H.L. Aarts, J.H. van Lint, and L.T. Wille. *New Upper Bounds for the Football Pool Problem for 6, 7, and 8 Matches.* Journal of Combinatorial Theory, Series A, 52, pp. 304–312, 1989.

[69] P.J.M. van Laarhoven, and E.H.L. Aarts. *Simulated Annealing: Theory and Applications.* Kluwer Academic Publishers, Dordrecht, 1989.

[70] J.H. van Lint Jr. *Covering Radius Problems.* M. Sc. Thesis, Eindhoven University of Technology, The Netherlands, June 1988.

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

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

[73] V.G. Vizing. *An Estimate of the External Stability Number of a Graph.* Dokl. Akad. Nauk., SSSR, 164, pp. 729–731, 1965.

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

[75] E.W. Weber. *On the Football Pool Problem for 6 Matches: A New Upper Bound.* Journal of Combinatorial Theory, Series A, 35, pp. 106–108, 1983.

[76] L.T. Wille. *The Football Pool Problem for 6 Matches: A New Upper Bound Obtained by Simulated Annealing.* Journal of Combinatorial Theory, Series A, 45, pp. 171–177, 1987.

[77] *Using the CPLEX Callable Library and CPLEX Mixed Integer Library. Including the CPLEX Linear Optimizer and CPLEX Mixed Integer Optimizer.* Incline Village, Nevada, CPLEX Optimization Inc., Version 4.0, 1995.

Appendix A

Orbit Graphs for the Football Pool Problem

A.1 6 Matches

In this appendix, we present in a table the 220 orbit graphs generated by the cyclic subgroups of the automorphism group of the football pool graph of 6 matches. We also present the 3 orbit graphs on the 7 matches for which there exist weighted dominating sets of size 186, matching the best known upper bound. We first explain how elements in the automorphism group of $\Gamma_{6,3}(V, E)$ can be represented in a compact form. As shown in Section 6.3 on page 66, there are two types of permutations, ρ_i which relabels the i-th entry of a n-tuples in Z_q^n, and π which permutes the entries.

Since $q = 3$, the relabeling operations on $Z_3 = \{0, 1, 2\}$ forms the group S_3 and is generated by $x = (1, 2)$ and $y = (0, 1, 2)$. Every element of S_3 can be represented as $x^i y^j$, with $i \leq 1$ and $j \leq 2$. Thus, we can associate a unique type with each element of S_3 by mapping $x^i y^j$ into the integer $i \times 3 + j$. In other words, we are treating the exponents $[ij]$ as an number in base 3.

These six types of relabeling operations are summarized in Table 19 on page 100. The ρ_i's are listed in cycle form.

The compact form of an element $g \in S_3 \wr S_n$ is obtained by expanding the usual image form of a permutation. In this compact form, the i-th entry contains two

Type	ρ_i
0	identity
1	(0, 1, 2)
2	(0, 2, 1)
3	(1, 2)
4	(0, 1)
5	(0, 2)

Table 19: The six possible types of relabeling operations.

components, $\pi[i]$ and ρ_i (written as subscript). Its action on an n-tuple $(x_1 x_2 \ldots x_n)$ is to take x_i to the $\pi[i]$-th position and then relabel x_i according to ρ_i. For example, the permutation part of the element $g = [5_0\ 6_3\ 3_0\ 1_2\ 2_4\ 4_2]$ takes (0 1 2 0 1 2) to (0 1 2 2 0 1), which is then relabeled to (2 0 2 1 0 2).

In Table 20 on page 101, and Table 21 on page 107, the column *Generator* gives the generator of the assumed automorphism group in our compact form. Since we are assuming a cyclic group, there is only one generator. The column *Weight Distribution* gives the weight distribution of the vertices in the orbit graph. For each weight, its exponent is the total number of vertices with that weight. The column *Weighted Dom. Set Size* gives the best weighted dominating set size we found for the corresponding orbit graph. If the size has the symbol "$\leq$" before it, then the value may not be optimal. Finally, the column *Isomorphic to Graph* indicates if this orbit graph is isomorphic to another orbit graph in that table.

A.2 7 Matches

Of the 428 orbit graphs for $\Gamma_{7,3}(V, E)$ we tested, we are able to find three weighted dominating set of size 186, matching the best known bound. Information on these three orbit graphs are presented in Table 21 on page 107.

Table 20: The result of solving orbit graphs number 1 to 220 from the football pool graph of 6 matches using CPLEX.

Graph	*Generator*	*Weight Distribution*	*Weighted Dom. Set Size*	*Isomorphic To Graph*
1	$5_0 6_3 3_0 1_5 2_4 4_5$	$5^{144}1^{9}$	≤ 99	
2	$2_4 4_2 3_1 5_5 6_5 1_1$	$15^{48}3^{3}$	93	
3	$2_2 4_4 3_0 5_1 6_1 1_0$	$15^{48}3^{3}$	81	
4	$2_2 4_4 3_5 5_1 6_1 1_0$	$15^{48}3^{3}$	78	
5	$5_2 6_0 3_5 1_3 2_5 4_2$	$30^{24}6^{1}3^{1}$	93	
6	$5_1 6_4 3_2 1_0 2_3 4_0$	$30^{16}15^{16}6^{1}3^{1}$	81	
7	$6_4 1_1 3_0 2_1 4_3 5_4$	$10^{72}2^{3}1^{3}$	83	
8	$6_5 1_4 3_3 2_2 4_1 5_1$	$10^{48}5^{48}2^{3}1^{3}$	81	
9	$6_4 1_1 3_3 2_1 4_3 5_4$	$10^{72}2^{4}1^{1}$	77	
10	$1_0 2_0 3_0 4_1 5_5 6_1$	3^{243}	≤ 81	
11	$2_3 3_3 1_0 4_0 5_0 6_0$	$3^{216}1^{81}$	≤ 95	
12	$2_3 3_3 1_0 5_3 6_3 4_0$	$3^{240}1^{9}$	≤ 99	
13	$1_0 2_0 3_0 5_3 6_3 4_1$	9^{81}	90	
14	$1_1 2_5 3_1 5_3 6_3 4_1$	9^{81}	90	
15	$1_5 2_1 3_0 5_3 6_3 4_1$	9^{81}	90	
16	$1_0 2_0 3_1 5_3 6_3 4_1$	9^{81}	90	
17	$2_3 3_3 1_0 5_3 6_3 4_1$	9^{81}	90	
18	$1_1 2_5 3_1 4_1 5_5 6_1$	3^{243}	≤ 96	
19	$2_3 3_3 1_1 5_3 6_3 4_1$	9^{81}	≤ 90	
20	$1_0 2_0 3_0 4_0 5_0 6_1$	3^{243}	≤ 81	
21	$1_0 2_0 3_0 4_5 5_0 6_1$	3^{243}	≤ 81	20
22	$1_1 2_5 3_1 4_0 5_0 6_1$	3^{243}	≤ 102	
23	$1_1 2_5 3_1 4_5 5_1 6_0$	3^{243}	≤ 81	
24	$2_3 3_3 1_0 4_1 5_5 6_1$	3^{243}	≤ 96	
25	$2_3 3_3 1_0 4_0 5_0 6_1$	3^{243}	≤ 99	
26	$2_3 3_3 1_0 4_5 5_0 6_1$	3^{243}	≤ 108	
27	$1_2 2_0 3_2 4_5 5_1 6_5$	$6^{108}3^{27}$	≤ 81	
28	$1_0 2_3 3_0 4_5 5_1 6_5$	$6^{81}3^{81}$	≤ 81	
29	$1_2 2_3 3_2 4_5 5_1 6_5$	$6^{117}3^{9}$	≤ 81	
30	$1_2 2_3 3_2 5_2 4_3 6_5$	$6^{120}3^{3}$	≤ 81	
31	$3_0 2_0 1_2 4_1 5_5 6_1$	$12^{54}3^{27}$	81	
32	$3_0 2_3 1_2 4_1 5_5 6_1$	$12^{54}6^{9}3^{9}$	81	
33	$3_2 2_3 1_0 5_3 4_2 6_1$	$12^{54}6^{12}3^{3}$	81	
34	$1_0 2_0 3_0 5_2 4_3 6_5$	$6^{81}3^{81}$	≤ 81	
35	$1_0 2_3 3_0 5_2 4_3 6_5$	$6^{108}3^{27}$	≤ 84	

Graph	*Generator*	*Weight Distribution*	*Weighted Dom. Set Size*	*Isomorphic To Graph*
36	$3_2 2_0 1_0 5_3 4_2 6_1$	$12^{54} 6^9 3^9$	≤ 81	
37	$1_2 2_0 3_2 5_2 4_3 6_5$	$6^{117} 3^9$	≤ 81	
38	$3_0 2_0 1_0 4_5 5_1 6_5$	$6^{81} 3^{81}$	≤ 81	
39	$3_0 2_3 1_0 4_5 5_1 6_5$	$6^{108} 3^{27}$	≤ 81	
40	$3_0 2_0 1_0 5_2 4_3 6_5$	$6^{108} 3^{27}$	≤ 81	
41	$3_0 2_3 1_0 5_2 4_3 6_5$	$6^{117} 3^9$	≤ 81	
42	$3_0 1_3 2_3 4_0 5_4 6_4$	$6^{96} 3^{24} 2^{36} 1^9$	≤ 81	
43	$3_3 1_0 2_0 4_0 5_0 6_0$	$6^{108} 2^{27} 1^{27}$	≤ 81	
44	$3_0 1_3 2_3 4_4 5_0 6_0$	$6^{72} 3^{72} 2^{27} 1^{27}$	≤ 81	
45	$3_3 1_0 2_0 4_4 5_0 6_0$	$6^{108} 2^{36} 1^9$	≤ 81	
46	$3_3 1_0 2_0 4_0 5_4 6_4$	$6^{108} 2^{39} 1^3$	≤ 85	
47	$3_0 1_3 2_3 4_4 5_4 6_4$	$6^{104} 3^8 2^{39} 1^3$	≤ 87	
48	$3_3 1_0 2_0 4_4 5_4 6_4$	$6^{108} 2^{40} 1^1$	≤ 86	
49	$2_3 3_3 1_0 4_0 6_0 5_4$	$12^{48} 4^{18} 3^{24} 1^9$	81	
50	$2_0 3_0 1_3 4_0 6_0 5_4$	$12^{48} 6^{12} 4^{18} 2^3 1^3$	80	
51	$2_3 3_3 1_0 4_4 6_0 5_4$	$12^{48} 6^8 4^{18} 3^8 2^3 1^3$	80	
52	$2_0 3_0 1_3 4_4 6_0 5_4$	$12^{48} 6^{12} 4^{18} 2^4 1^1$	81	
53	$3_0 1_3 2_3 4_0 6_0 5_0$	$6^{72} 3^{72} 2^{27} 1^{27}$	≤ 102	
54	$3_3 1_0 2_0 4_0 6_0 5_0$	$6^{108} 2^{36} 1^9$	≤ 81	
55	$3_0 1_3 2_3 4_4 6_0 5_0$	$6^{96} 3^{24} 2^{36} 1^9$	≤ 79	
56	$3_3 1_0 2_0 4_4 6_0 5_0$	$6^{108} 2^{39} 1^3$	≤ 87	
57	$3_2 1_5 2_1 6_4 4_1 5_5$	$6^{120} 2^4 1^1$	≤ 77	
58	$3_2 1_5 2_1 6_0 4_3 5_3$	$6^{116} 3^8 2^3 1^3$	≤ 80	
59	$5_2 6_4 4_1 2_0 3_0 1_3$	$12^{60} 4^2 1^1$	77	
60	$6_1 4_4 5_2 3_5 1_2 2_4$	$6^{116} 3^8 2^3 1^3$	≤ 84	
61	$1_0 2_3 3_3 6_1 4_2 5_4$	$18^{36} 9^9$	117	
62	$1_2 2_0 3_0 6_1 4_2 5_4$	$18^{27} 9^{27}$	108	
63	$1_2 2_3 3_3 6_1 4_2 5_4$	$18^{39} 9^3$	108	
64	$1_0 3_3 2_0 5_4 6_2 4_0$	$36^{18} 9^9$	126	
65	$1_2 3_3 2_0 5_4 6_2 4_0$	$36^{18} 18^3 9^3$	117	
66	$1_0 3_3 2_3 6_1 4_2 5_4$	$18^{27} 9^{27}$	108	
67	$1_2 3_3 2_3 6_1 4_2 5_4$	$18^{36} 9^9$	81	
68	$1_5 3_2 2_4 6_1 4_2 5_4$	$18^{27} 9^{27}$	81	
69	$1_1 2_5 3_3 6_1 4_2 5_4$	$18^{27} 9^{27}$	81	
70	$2_4 1_3 3_3 6_1 4_2 5_4$	$18^{36} 9^9$	81	

Graph	*Generator*	*Weight Distribution*	*Weighted Dom. Set Size*	*Isomorphic To Graph*
71	$2_4 1_3 3_0 6_1 4_2 5_4$	$18^{27}9^{27}$	81	
72	$1_3 2_4 3_5 6_1 4_2 5_4$	$18^{36}9^{9}$	90	
73	$2_2 1_2 3_5 6_1 4_2 5_4$	$18^{27}9^{27}$	81	
74	$2_2 1_1 3_1 5_4 6_2 4_0$	$36^{18}9^{9}$	117	
75	$1_0 2_4 3_5 6_1 4_2 5_4$	$18^{27}9^{27}$	81	
76	$3_4 1_1 2_5 6_1 4_2 5_4$	$18^{39}9^{3}$	90	
77	$4_5 2_1 3_5 1_5 6_2 5_4$	$6^{108}3^{27}$	≤ 87	
78	$1_5 3_4 2_3 4_5 5_1 6_5$	$6^{81}3^{81}$	≤ 90	
79	$4_5 3_4 2_3 1_5 6_2 5_4$	$6^{117}3^{9}$	≤ 81	
80	$6_1 2_5 3_1 5_2 1_4 4_1$	$12^{54}6^{9}3^{9}$	≤ 81	
81	$5_2 3_3 2_4 6_1 4_4 1_1$	$12^{54}6^{12}3^{3}$	81	
82	$4_0 5_0 6_0 2_3 3_3 1_1$	$18^{39}9^{3}$	81	
83	$1_0 2_3 3_2 4_2 5_3 6_5$	$6^{120}3^{3}$	81	
84	$1_0 2_0 3_0 4_2 5_3 6_5$	$6^{108}3^{27}$	≤ 87	
85	$1_3 2_0 3_0 4_0 5_0 6_5$	$6^{81}3^{81}$	81	
86	$1_3 2_3 3_2 4_2 5_3 6_5$	$6^{121}3^{1}$	81	
87	$1_3 2_0 3_0 4_2 5_3 6_5$	$6^{117}3^{9}$	≤ 90	
88	$1_0 3_4 2_5 5_4 4_1 6_1$	$12^{60}3^{3}$	81	
89	$1_3 3_4 2_5 5_4 4_1 6_1$	$12^{60}6^{1}3^{1}$	81	
90	$1_0 3_4 2_5 4_0 5_0 6_1$	$12^{54}3^{27}$	≤ 111	
91	$1_3 3_4 2_5 4_0 5_0 6_1$	$12^{54}6^{9}3^{9}$	≤ 105	
92	$1_0 3_1 2_4 4_2 5_3 6_1$	$12^{54}6^{12}3^{3}$	99	
93	$1_3 3_1 2_4 4_2 5_3 6_1$	$12^{54}6^{13}3^{1}$	90	
94	$1_3 4_1 5_4 3_2 2_3 6_5$	$24^{30}6^{1}3^{1}$	81	
95	$1_0 4_1 5_4 3_2 2_3 6_5$	$24^{30}3^{3}$	81	
96	$1_0 3_4 2_4 5_5 4_4 6_1$	$12^{54}6^{9}3^{9}$	87	
97	$1_3 3_4 2_4 5_5 4_4 6_1$	$12^{54}6^{12}3^{3}$	96	
98	$1_0 3_4 2_4 4_0 5_0 6_5$	$6^{81}3^{81}$	81	
99	$1_3 3_4 2_4 4_0 5_0 6_5$	$6^{108}3^{27}$	81	
100	$1_0 3_4 2_4 5_4 4_4 6_5$	$6^{108}3^{27}$	81	
101	$1_3 4_4 5_4 3_2 2_3 6_1$	$12^{54}6^{12}3^{3}$	93	
102	$1_0 4_4 5_4 3_2 2_3 6_1$	$12^{54}6^{9}3^{9}$	≤ 99	
103	$1_3 3_4 2_4 5_4 4_4 6_5$	$6^{117}3^{9}$	81	
104	$1_0 3_1 2_5 4_2 5_3 6_5$	$6^{117}3^{9}$	81	
105	$1_3 3_1 2_5 4_2 5_3 6_5$	$6^{120}3^{3}$	81	

Graph	*Generator*	*Weight Distribution*	*Weighted Dom. Set Size*	*Isomorphic To Graph*
106	$1_3 2_4 3_2 4_1 5_2 6_5$	$6^{120}3^3$	81	83
107	$1_3 2_0 3_2 4_1 5_0 6_5$	$6^{108}3^{27}$	≤ 87	84
108	$1_0 2_0 3_0 6_2 5_0 4_4$	$6^{81}3^{81}$	81	85
109	$1_3 2_4 3_2 6_2 5_2 4_4$	$6^{121}3^1$	81	
110	$1_3 2_0 3_2 6_2 5_0 4_4$	$6^{117}3^9$	≤ 90	87
111	$3_4 5_5 1_5 4_5 2_3 6_1$	$12^{60}3^3$	81	88
112	$1_0 5_3 3_0 4_5 2_1 6_1$	$12^{54}3^{27}$	≤ 111	90
113	$3_1 5_3 1_4 6_4 2_1 4_2$	$12^{60}6^1 3^1$	81	89
114	$1_0 5_3 3_0 6_4 2_1 4_2$	$12^{54}6^9 3^9$	≤ 105	91
115	$1_3 5_5 3_2 4_5 2_3 6_1$	$12^{54}6^{12}3^3$	99	92
116	$1_3 5_5 3_2 6_4 2_3 4_2$	$12^{54}6^{13}3^1$	90	93
117	$5_4 1_1 2_1 4_1 3_0 6_5$	$24^{30}3^3$	81	95
118	$2_2 3_3 5_2 6_2 1_5 4_4$	$24^{30}6^1 3^1$	81	94
119	$3_4 5_3 1_5 4_5 2_3 6_1$	$12^{54}6^9 3^9$	87	96
120	$1_3 5_3 3_0 4_5 2_1 6_1$	$12^{54}6^9 3^9$	≤ 105	91
121	$3_1 5_3 1_4 6_4 2_3 4_2$	$12^{54}6^{12}3^3$	96	97
122	$1_3 5_3 3_0 6_4 2_1 4_2$	$12^{54}6^{12}3^3$	99	92
123	$1_3 2_0 3_0 4_1 5_0 6_5$	$6^{81}3^{81}$	81	85
124	$1_0 5_5 3_0 4_1 2_1 6_5$	$6^{81}3^{81}$	81	98
125	$1_3 2_0 3_0 6_2 5_0 4_4$	$6^{108}3^{27}$	≤ 87	84
126	$1_0 5_5 3_0 6_2 2_1 4_4$	$6^{108}3^{27}$	81	99
127	$3_1 5_3 1_5 4_1 2_3 6_5$	$6^{108}3^{27}$	81	100
128	$5_4 1_2 2_1 4_5 3_0 6_1$	$12^{54}6^9 3^9$	≤ 99	102
129	$2_2 3_5 5_0 6_4 1_4 4_2$	$12^{54}6^{12}3^3$	93	101
130	$3_1 5_3 1_5 6_2 2_3 4_4$	$6^{117}3^9$	81	103
131	$1_0 2_4 3_2 4_1 5_2 6_5$	$6^{117}3^9$	≤ 90	87
132	$1_3 5_3 3_2 4_1 2_3 6_5$	$6^{117}3^9$	81	104
133	$1_0 2_4 3_2 6_2 5_2 4_4$	$6^{120}3^3$	81	83
134	$1_3 5_3 3_2 6_2 2_3 4_4$	$6^{120}3^3$	81	105
135	$1_3 5_5 3_0 4_1 2_1 6_5$	$6^{108}3^{27}$	81	99
136	$1_3 5_5 3_0 6_2 2_1 4_4$	$6^{117}3^9$	81	104
137	$2_2 1_3 6_5 4_0 5_0 3_5$	$6^{108}3^{27}$	≤ 81	
138	$2_2 1_3 6_5 4_2 5_4 3_5$	$6^{120}3^3$	≤ 81	
139	$2_2 1_3 3_5 4_0 5_0 6_5$	$6^{81}3^{81}$	≤ 81	
140	$2_2 1_3 6_5 5_1 4_5 3_5$	$6^{117}3^9$	≤ 81	

Graph	*Generator*	*Weight Distribution*	*Weighted Dom. Set Size*	*Isomorphic To Graph*
141	$3_5 6_3 2_4 4_0 5_0 1_0$	$12^{54} 6^9 3^9$	90	
142	$1_1 2_5 3_1 5_3 4_5 6_1$	$12^{54} 3^{27}$	≤ 99	
143	$6_5 3_3 1_0 4_2 5_4 2_4$	$12^{54} 6^{13} 3^1$	81	
144	$2_3 1_2 6_1 5_1 4_3 3_1$	$12^{54} 6^{12} 3^3$	84	
145	$6_5 3_3 1_0 5_1 4_3 2_4$	$12^{60} 6^1 3^1$	81	
146	$6_5 3_3 1_0 4_0 5_4 2_4$	$12^{54} 6^{12} 3^3$	87	
147	$6_5 3_3 1_0 5_1 4_5 2_4$	$12^{54} 6^{12} 3^3$	81	
148	$1_5 2_1 3_5 4_2 5_4 6_5$	$6^{108} 3^{27}$	≤ 84	
149	$2_3 1_2 3_1 5_3 4_5 6_1$	$12^{54} 6^9 3^9$	≤ 84	
150	$1_5 2_1 3_5 4_2 5_0 6_5$	$6^{81} 3^{81}$	≤ 81	
151	$1_5 2_1 3_3 5_1 4_5 6_5$	$6^{81} 3^{81}$	≤ 84	
152	$2_2 1_3 3_5 4_2 5_4 6_5$	$6^{117} 3^9$	≤ 81	
153	$2_2 1_3 6_5 4_2 5_0 3_5$	$6^{117} 3^9$	≤ 84	
154	$2_2 1_3 3_5 4_2 5_0 6_5$	$6^{108} 3^{27}$	≤ 81	
155	$2_2 1_3 3_5 5_1 4_5 6_5$	$6^{108} 3^{27}$	≤ 87	
156	$4_4 3_4 2_3 1_2 5_5 6_0$	$6^{108} 3^{27}$	≤ 81	
157	$1_5 2_1 3_5 4_1 5_5 6_2$	$6^{81} 3^{81}$	≤ 84	
158	$4_4 3_4 2_3 1_2 5_5 6_2$	$6^{117} 3^9$	≤ 81	
159	$3_5 1_2 4_3 2_0 5_1 6_2$	$12^{54} 6^{12} 3^3$	78	
160	$2_3 4_1 1_0 3_2 5_1 6_0$	$12^{54} 6^9 3^9$	≤ 84	
161	$1_5 3_4 2_3 4_1 5_5 6_0$	$6^{81} 3^{81}$	≤ 81	
162	$4_4 2_1 3_5 1_2 5_5 6_2$	$6^{108} 3^{27}$	≤ 96	
163	$3_3 1_0 2_0 5_4 4_2 6_5$	$6^{120} 3^3$	≤ 81	
164	$3_0 1_3 2_3 5_4 4_2 6_5$	$6^{81} 3^{81}$	≤ 81	
165	$3_3 1_0 2_0 4_5 5_1 6_5$	$6^{117} 3^9$	≤ 84	
166	$3_0 1_3 2_3 4_4 5_4 6_5$	$6^{108} 3^{27}$	≤ 81	
167	$3_3 1_0 2_0 4_0 5_0 6_5$	$6^{117} 3^9$	≤ 78	
168	$3_3 1_0 2_0 4_4 5_4 6_5$	$6^{121} 3^1$	≤ 81	
169	$3_0 1_3 2_3 4_0 5_4 6_5$	$6^{81} 3^{81}$	≤ 81	
170	$3_3 1_0 2_0 4_0 5_4 6_5$	$6^{120} 3^3$	≤ 81	
171	$2_3 3_3 1_0 5_0 4_4 6_1$	$12^{54} 3^{27}$	81	
172	$2_0 3_0 1_3 5_4 4_0 6_1$	$12^{54} 6^{13} 3^1$	81	
173	$3_0 1_3 2_3 5_4 4_4 6_5$	$6^{81} 3^{81}$	≤ 81	
174	$3_3 1_0 2_0 5_4 4_4 6_5$	$6^{120} 3^3$	≤ 78	
175	$3_3 1_0 2_0 6_2 5_0 4_4$	$6^{120} 3^3$	≤ 81	

Graph	Generator	Weight Distribution	Weighted Dom. Set Size	Isomorphic To Graph
176	$3_01_32_3c_15_46_5$	$6^{81}3^{81}$	≤ 81	
177	$3_31_02_06_25_44_4$	$6^{121}3^{1}$	≤ 81	
178	$3_31_02_04_15_06_5$	$6^{117}3^{9}$	≤ 84	
179	$3_01_32_36_25_44_4$	$6^{108}3^{27}$	≤ 87	
180	$3_01_32_36_25_04_4$	$6^{81}3^{81}$	≤ 84	
181	$3_31_02_04_15_46_5$	$6^{120}3^{3}$	≤ 81	
182	$1_32_33_34_35_06_0$	$2^{360}1^{9}$	≤ 100	
183	$1_02_03_04_05_36_3$	$2^{324}1^{81}$	≤ 94	
184	$1_32_33_34_35_36_3$	$2^{364}1^{1}$	≤ 93	
185	$1_02_03_04_05_36_0$	$2^{243}1^{243}$	≤ 93	
186	$1_32_33_34_35_36_0$	$2^{363}1^{3}$	≤ 105	
187	$2_01_03_04_05_36_3$	$2^{351}1^{27}$	≤ 95	
188	$2_01_34_03_35_06_0$	$4^{180}1^{9}$	≤ 87	
189	$2_01_34_03_35_36_3$	$4^{180}2^{4}1^{1}$	≤ 94	
190	$1_02_03_04_06_05_3$	$4^{162}1^{81}$	≤ 88	
191	$1_32_33_34_36_05_3$	$4^{162}2^{40}1^{1}$	≤ 99	
192	$2_31_03_34_35_06_0$	$4^{162}2^{36}1^{9}$	≤ 84	
193	$2_01_33_04_06_05_0$	$4^{162}2^{27}1^{27}$	≤ 84	
194	$2_31_03_34_35_36_0$	$4^{162}2^{39}1^{3}$	≤ 96	
195	$2_01_34_03_35_36_0$	$4^{180}2^{3}1^{3}$	≤ 91	
196	$2_01_34_03_36_05_3$	$4^{182}1^{1}$	≤ 81	
197	$3_04_02_01_35_06_0$	$8^{90}1^{9}$	81	
198	$3_04_02_01_35_36_3$	$8^{90}2^{4}1^{1}$	81	
199	$3_04_02_01_35_36_0$	$8^{90}2^{3}1^{3}$	81	
200	$3_04_02_01_36_05_3$	$8^{90}4^{2}1^{1}$	≤ 81	
201	$2_31_33_34_36_05_3$	$4^{162}2^{39}1^{3}$	≤ 89	
202	$1_32_03_04_06_05_3$	$4^{162}2^{27}1^{27}$	≤ 84	
203	$2_01_04_03_06_05_3$	$4^{162}2^{36}1^{9}$	≤ 104	
204	$2_01_03_34_06_05_3$	$4^{162}2^{36}1^{9}$	≤ 110	
205	$3_04_02_01_36_05_0$	$8^{90}2^{3}1^{3}$	≤ 81	
206	$2_01_34_03_36_05_0$	$4^{180}2^{3}1^{3}$	≤ 86	
207	$1_02_03_04_06_05_0$	$2^{243}1^{243}$	≤ 105	
208	$1_32_33_34_36_05_0$	$2^{363}1^{3}$	≤ 101	
209	$2_01_04_03_05_06_0$	$2^{324}1^{81}$	≤ 96	
210	$3_04_02_01_05_36_3$	$4^{162}2^{39}1^{3}$	≤ 90	

Graph	Generator	Weight Distribution	Weighted Dom. Set Size	Isomorphic To Graph
211	$3_0 4_0 2_0 1_0 5_0 6_0$	$4^{162} 2^{27} 1^{27}$	≤ 82	
212	$3_0 4_0 2_0 1_0 5_3 6_0$	$4^{162} 2^{36} 1^{9}$	≤ 73	
213	$3_0 4_0 2_0 1_0 6_0 5_0$	$4^{164} 2^{36} 1^{9}$	≤ 85	
214	$2_0 1_0 4_0 3_0 5_3 6_3$	$2^{360} 1^{9}$	≤ 109	
215	$3_0 4_0 2_0 1_0 6_0 5_3$	$4^{180} 2^{3} 1^{3}$	≤ 94	
216	$1_3 2_0 3_0 4_0 5_3 6_3$	$2^{351} 1^{27}$	≤ 102	
217	$2_0 1_0 3_0 4_0 5_3 6_0$	$2^{324} 1^{81}$	≤ 101	
218	$2_0 1_0 4_0 3_0 5_3 6_0$	$2^{351} 1^{27}$	≤ 138	
219	$2_0 1_0 3_3 4_0 5_3 6_3$	$2^{360} 1^{9}$	≤ 89	
220	$2_0 1_0 4_0 3_0 6_0 5_0$	$2^{351} 1^{27}$	≤ 94	

Graph	Generator	Weight Distribution	Weighted Dom. Set Size	Isomorphic To Graph
209	$6_1 1_0 3_1 4_5 2_5 5_1 7_0$	$12^{162} 6^{27} 3^{27}$	≤ 186	
255	$6_5 3_1 2_1 4_5 1_5 7_0 5_0$	$12^{162} 6^{36} 3^{9}$	≤ 186	
313	$4_5 5_3 3_1 2_0 1_2 7_4 6_4$	$12^{162} 6^{36} 3^{9}$	≤ 186	

Table 21: The orbit graphs that have a dominating set of size 186 in the football pool graph of 7 matches using CPLEX.

Appendix B

Program Source Code

In this Appendix, we have included all the source code in the program P_{sym}. The file `domset.c` contains the main procedure to start the program. It also lists out the input requirement. The file `basicdom.c` contains the basic operations such as initialization of all the global variables, refinement of a cell by half, partition the content, and the coverage and wastage tests. The file `groupdom.c` contains the group related operations such as the symmetry partition, and the isomorph rejection. The file `memmdom.c` contains the memory management operations such as the dynamic allocation of a cell node. All the definitions of the data structures and the prototypes of the procedures are declared in the corresponding header files.

```
#include <stdio.h>
#include "basicdom.h"
#include "groupdom.h"

/*********************************************************************/

/* This file contains the main procedure to start searching for a
   dominating set.
*/

/*********************************************************************/

/* standard input for C is stdin,
   standard output for C is stdout,
   007 in ASCII is the beep sound */

/* Notice
      i. internally vertex are numbered from 1 to total_vertices

     ii. Input format:
           0) Total number of sets of input graph.

           0.1) A list of numbers which refer to the i th input graph
                that is going to skip. The list must end with a -1.

           Repeat data item 1 to 9 stated below for each set of graph

           1) The input graph's number. This is used to identify the
              graph.
           2) Input vertex started numbered from zero or one?
              0 for starting form zero
              1 for starting form one
           3) Any debug?
              N.B. The total debug entries are 40
              0 for no debug
              1 for debug
                3.1) 0 for user to turn on certain debug variables
                       3.1.1) the input format
                              0
                                 3.1.1.1) the value of the whole debug list,
                                          starting from position one
                                          1 to turn it on
                                          0 to turn it off
                              1  the entry of the debug list which will
                                 be set to true, a -1 is required to
                                 at the end of the list
                     1 for user to turn on all the debug variables
           4) type of graph,
              0 for non-direct graph
              1 for direct graph
           5) total number of vertices
           6) graph input format,
              0 for adjacent matrix
              1 for neighbor vertices for each vertices
           7) the graph in the above format; the input format for neighbor
              vertices for a graph of n vertices is the following:
                all the neighbor of the first vertices must be listed first
                and end with a -1 and follow by all the neighbors of the
                second vertices and so on, the vertices itself is a neighbor
                of itself.
                for non-direct graph, one must at least include all neighbors
                with label greater than that vertex, vertex with no neighbor
                can leave with a -1. N.B. The vertex itself do not require
                to be included.
           8) automated cell partition,
              0 for user input the initial cell partition
                8.1) the total number of subcells
                8.2) the vertices of each subcell, end with a -1
                8.3) the weight of each subcell
                8.4) the total content of the original graph
                     (not the weighted graph)
                8.5) flag for automated content partition,
                       0 for no automated content partition
                         8.5.1) each content of the subcell
```



```
                       1 for automated content partition
                8.6) number of generator for the automorphism of the
                     graph with this partition
              1 for automated cell partition
                8.1) total number of contents allowed
                8.2) number of generator for the automorphism of the graph
           9) the format of the generator
              0 image form
                9.1) each generator in its image form
              1 cyclic form
                9.1) each generator in its cyclic form with
                         a -1 to end each generator
                         e.g. (1, 2) (3, 4) -1
          10) any number (at least three) of '~' characters,
              with 2 blank character at the end. This string is used
              to separate one set of data to another


    iii. below is the corresponding procedure for each position in the
         debug list:
         position    procedure
         0           Set all the debuglist to true
         1           Generate_influ_func
         2           Generate_next_level
         3           Partition_cell
         4           Coverage_test
         5           Wastage_test
         6           Update_max_allowable_wastage
         7           Start_gen
         8           Handle_reg_content_part
         9           Create_and_partition_other_cell_and_level
         10          Initiate_val
         11          Create_first_level
         12          Create_group_by_generator
         13          Create_color_wreath_product_group
         14          Generate_partition_stabilizer
         15          Transform_cell_to_permvect
         16          Invert_mapping
         17          Initialize_a_cell
         18          Transform_orbits_to_cells
         19          Transform_blocks_to_cells
         20          Split_rep
         21          Isomorphic_test
         22          Get_all_orbits
         23          Select_cell_with_nontrivial_gp
         24          Next_level
         25          Identity
         26          Symmetry_partition
         27          Compare
         28          Regular_partition
         29          Generate_cell_stabilizer
         30          Create_cwpg_for_cell_stab
         31          Get_completed_orbit_cells
         32          Handle_transitive_content_part
         33          Create_cwpg_for_domset
         34          Coverage_and_wastage_test
         35          Gen_weight_content
         36          Start_content_gen
         37
         38
         39
         40

     iv. To run this program, one should have the following four files stored
         in the same sub-directory: domset.h, domset.c, partition_func.c,
         and makefile. To compile them, just type make. To run it
         type domset input_file_name output_file_name. stdin means
         input from the screen and stdout means output to the screen.

*/

/* Comments:

    1) This version of the program only work for undirected,
```

```
      loopless, and labeled graph with no multiple edges.
*/

/**********************************************************************/
/* FUNCTIONS */
/**********************************************************************/

/* This procedure will handle all the inputs required to create the
   first level and the automorphism group given by a set of generators. */
/* size effect: automated_cell_partition is updated here (fscanf)
                skip_this_graph will also set to true if the graph's
                size is out of range */

int Initializations (voidplist)

{
   Init_time_val ();
   Initiate_basicdom_val();
   if (skip_this_graph)
     return;
   Initiate_groupdom_val();
   Init_first_cell ();
   fflush(out);
   /* fprintf (out, "First cell created.\n"); */
   /* read in the automated_cell_partition */
   fscanf (in, "%d", &automated_cell_partition);
   Create_first_level();
} /* Initializations  */

/**********************************************************************/

/* To test if the cumulative content is too big */

/* input parameters:
   1) pos: the content of cell 'pos' to be generated
   2) cum_content: the cumulative content
   3) max_leftovers: an array contain the maximum possible content size
                     at each position
*/

int Acceptable (int pos, int cum_content, int *max_leftovers)

{
   int leftover_content;

   leftover_content = org_total_content - cum_content;
/*
   fprintf (out, "%d %d %d %d %d\n", leftover_content, org_total_content,
            cum_content, pos, max_leftovers[pos]);
*/
   if (leftover_content <= max_leftovers[pos])
     return 1;
   else
     return 0;
}

/**********************************************************************/

/* This procedure will set the content partition of a list of subcells. */

/* input parameters:
   1) parent_ptr: the parent of the subcells
   2) content_dis: an array of integer that contain the contents

   side effect: finish_computation may be changed here
*/

void Set_the_content_partition (ptr_to_cell_node parent_ptr,
                                int *content_dis)
{
   ptr_to_cell_node cell_ptr;
   int sum_content = 0, i, temp;

   cell_ptr = parent_ptr->first_child;
   i = 0;
```



```
   while (cell_ptr != NULL)
   {
      /* update the first cell */
      cell_ptr->upper_content = cell_ptr->lower_content
                              = cell_ptr->actual_content
                              = content_dis[i];
      sum_content += content_dis[i];
      cell_ptr = cell_ptr->sibling;
      i++;
   }
   /* fprintf(out,"\n"); */
   parent_ptr->upper_content = parent_ptr->lower_content
                             = parent_ptr->actual_content
                             = total_content
                             = sum_content;

   if (total_content > total_vertices)
   {
      fprintf (out, "\nnumber of contents are greater than the ");
      fprintf (out, "total number of vertices, program halt.\n");
      Myexit(1);
   }
   /* work out the maximum allowable wastage of the parent */
   /* since the first cell only has one neighbor */
   temp = Get_UIF
          (parent_ptr->row_first_neigh->row_next_neigh->
             tot_ver_of_deg_from_row_to_col,
           total_content);

   if ((parent_ptr->max_allow_wastage =
        temp - parent_ptr->cell_size) < 0)
   {
      fprintf (out, "\nProgram halt due to maximum allowable ");
      fprintf (out, "wastage of the first cell < 0.\n");
      finish_computation = TRUE;
      Skip_this_graph();
   }

} /* Set_the_content_partition */

/**********************************************************************/

/* Start_content_gen recursively generate all the possible content
   combination of a list of cells given the weight of the cells and
   its total content. After the content was generated, the procedure
   will try to find the dominating set. */

/* input parameters:
   1) pos: the content of cell 'pos' to be generated
   2) cum_content: the cumulative content
   3) total_cells: the total number of cells
   4) allowable_csizes: an array contain the allowable size of each cell
                        in order to generate a feasible content partition
   5) cell_weights: an array contain the weight of each cell
   6) max_leftovers: an array contain the maximum possible content size
                     at each position
   7) content_dis: to store the actual content distribution of the
                   cells
   8) parent_cell: the parent cell of the input subcells
*/

Start_content_gen (int pos,
                   int cum_content,
                   int total_cells,
                   int *allowable_csizes,
                   int *cell_weights,
                   int *max_leftovers,
                   int *content_dis,
                   ptr_to_cell_node parent_cell)
{
   int j, content_val, reset = 0, tot_content, temp_content;

   if (automated_content_partition)
   {
      /* fprintf (out, "position = %d.\n", pos); */
```

```
      for (content_val = 0;
           content_val <= allowable_csizes[pos];
           content_val++, reset = 0)
      {
         /* fprintf (out, "content_val = %d.\n", content_val); */
         if (Acceptable (pos, cum_content, max_leftovers))
         {
            content_dis[pos] = content_val;
            cum_content += (content_val*cell_weights[pos]);
            /* fprintf (out, "cum_content = %d.\n", cum_content); */
            reset = 1;
            if ((cum_content == org_total_content) && (pos == (total_cells - 1)))
            {
               tot_content = 0;
               if (debuglist[36])
               {
                  fprintf (out, "\nThe following is a good set of contents:\n");
                  for (j = 0; j < total_cells; j++)
                  {
                     fprintf (out, "%d ", content_dis[j]);
                     tot_content += content_dis[j];
                  }
                  fprintf (out, "\n");
               }
               Set_the_content_partition (parent_cell, content_dis);

               if (finish_computation)
                 return;
               if (debuglist[36])
               {
                  fprintf (out, "\nHere is the parent cell at second level.\n");
                  Print_cell (parent_cell);
                  Print_curr_status ();
               }
               if (Coverage_and_wastage_test(parent_cell))
               {
                  /* fprintf (out, "Start looking for a dominating set.\n"); */
                  level_visited[1]++;
                  Start_gen (1, parent_cell, auto_gp, NULL, NULL,
                             NULL, START_STATUS, NULL);
               }
            }
            else
              if (pos < total_cells - 1)
                Start_content_gen (pos + 1, cum_content,
                   total_cells, allowable_csizes, cell_weights,
                   max_leftovers, content_dis, parent_cell);
         }
         if (reset)
           cum_content -= content_val*cell_weights[pos];
      }
      /*fprintf (out, "exit: cum_content = %d pos = %d \n", cum_content, pos);*/
   }
   else /* user input content partition of each cell */
   {
      tot_content = temp_content = 0;
      for (j = 0; j < total_cells; j++)
      {
         fscanf (in, "%d", &(content_dis[j]));
         tot_content += content_dis[j];
         temp_content += content_dis[j] * cell_weights[j];
      }

      if (temp_content != org_total_content)
      {
         fprintf (out, "\nUser input content not equal to ");
         fprintf (out, "total content.\n");
         Myexit(1);
      }

      if (debuglist[36])
      {
         fprintf (out, "\nThe following is a good set of contents:\n");
         for (j = 0; j < total_cells; j++)
         {
```



```
            fprintf (out, "%d ", content_dis[j]);
         }
         fprintf (out, "\n");
      }
      Set_the_content_partition (parent_cell, content_dis);

      if (finish_computation)
        return;
      if (debuglist[36])
      {
         fprintf (out, "\nHere is the parent cell at second level.\n");
         Print_cell (parent_cell);
         Print_curr_status ();
      }
      if (Coverage_and_wastage_test(parent_cell))
      {
         /* fprintf (out, "Start looking for a dominating set.\n"); */
         level_visited[1]++;
         auto_gp = Create_group_by_generator ();
         /* fprintf (out, "Group created.\n"); */
         auto_gp = Generate_partition_stabilizer ();

         Start_gen (1, parent_cell, auto_gp, NULL, NULL,
                    NULL, START_STATUS, NULL);
      }
   }
} /* Start_content_gen */

/*********************************************************************/

/* To create the content for the input partitions based on the
   weight of each subcell. The total content of the original
   graph would be read. */

/* input parameters:
   1) parent_cell: the parent cell
   2) cell_weights: an array contains the weight of each cell
*/

void Create_content_and_find_domset (ptr_to_cell_node parent_cell,
                                     int *cell_weights)
{
  int i, max_size;
  /* recording the weight content distribution among the cells */
  int *content_distribution = NULL;
  int *allowable_csizes = NULL;
  int *upper_limits = NULL;
  int *max_leftovers = NULL;
  int *cell_sizes = NULL;
  ptr_to_cell_node curr_cell;

  content_distribution = (int *) calloc (4*(total_first_part_subcells+1),
                                          sizeof(int));
  allowable_csizes = &(content_distribution[total_first_part_subcells+1]);
  upper_limits = &(content_distribution[2*(total_first_part_subcells+1)]);
  max_leftovers = &(content_distribution[3*(total_first_part_subcells+1)]);
  cell_sizes = (int *) calloc (total_first_part_subcells+1,
                               sizeof(int));

  for (i = 0; i < 4*(total_first_part_subcells+1); i++)
    content_distribution[i] = 0;

  curr_cell = parent_cell->first_child;
  i = 0;
  while (curr_cell != NULL)
  {
     cell_sizes[i] = curr_cell->cell_size;
     curr_cell = curr_cell->sibling;
     i++;
  }

  for (i = 0; i < total_first_part_subcells; i++)
  {
     max_size = org_total_content/cell_weights[i];
     if (max_size > cell_sizes[i])
```

```
      allowable_csizes[i] = cell_sizes[i];
    else
      allowable_csizes[i] = max_size;
  }


  for (i = total_first_part_subcells - 1; i >= 0; i--)
  {
     upper_limits[i] = cell_sizes[i] * cell_weights[i];
     /* fprintf (out, "%d, %d\n", cell_sizes[i], upper_limits[i]);*/
  }
  max_leftovers[total_first_part_subcells - 1] =
        upper_limits[total_first_part_subcells - 1];
  for (i = (total_first_part_subcells - 2); i >= 0; i--)
  {
     max_leftovers[i] = max_leftovers[i+1] + upper_limits[i];
  }

  if (debuglist[35])
  {
     fprintf (out, "The allowable size of each cell is: \n");
     for (i = 0; i < total_first_part_subcells; i++)
       fprintf (out, "%d ", allowable_csizes[i]);
     fprintf(out, "\n");

     fprintf (out, "The upper limits is: \n");
     for (i = 0; i < total_first_part_subcells; i++)
       fprintf (out, "%d ", upper_limits[i]);
     fprintf(out, "\n");

     fprintf (out, "The maximum leftovers is: \n");
     for (i = 0; i < total_first_part_subcells; i++)
       fprintf (out, "%d ", max_leftovers[i]);
     fprintf(out, "\n");
  }

  Start_content_gen (0, 0, total_first_part_subcells, allowable_csizes,
                     cell_weights, max_leftovers, content_distribution,
                     parent_cell);
  free ((void *) cell_sizes);
  cell_sizes = NULL;
  free ((void *) content_distribution);
  content_distribution = NULL;

} /* Create_content_and_find_domset */


/*************************************************************************/

/* Partition the first cell and create other levels.*/
/* side effects: automated_content_partition is updated here (fscanf),
                 the automorphism group given by a set of generators
                 will be created */

/* debug list value = 9 */

void Search_for_dominating_set (voidplist)

{
   ptr_to_cell_node parent_cell;
   int *cell_weights, i;
   enum part_type tag;

   parent_cell = dummy_cell->next_cell;
   if (automated_cell_partition)
   {
       Initiate_groupdom_val();
       /* fprintf (out, "Automated cell partition.\n"); */
       auto_gp = Create_group_by_generator ();
       /* fprintf (out, "Group created.\n"); */
       if (START_STATUS == 1)
         tag = REGULAR;
       else tag = SYMMETRY;
       Next_level (tag, start_level, parent_cell, auto_gp);
   }
```



```
   else
   {
      /* N.B. the auto group have to be the automorphism of the
              cell's partition */
      fprintf (out, "\nUser input the first cell's partition.\n");
      Input_first_cell_partition(parent_cell);

      /* read in the subcells' weight */
      cell_weights = (int *) calloc (total_first_part_subcells,
                                     sizeof(int));
      for (i = 0; i < total_first_part_subcells; i++)
        fscanf (in, "%d", &(cell_weights[i]));

      Print_input_partition_and_weight (parent_cell, cell_weights);

      Create_influ_func_info (parent_cell, parent_cell->first_child);
      Handle_new_level (parent_cell);
      fscanf (in, "%d", &org_total_content);
      Initiate_groupdom_val();

      fscanf (in, "%d", &automated_content_partition);
      if (automated_content_partition)
      {
         auto_gp = Create_group_by_generator ();
         /* fprintf (out, "Group created.\n"); */
         auto_gp = Generate_partition_stabilizer();
      }


      Create_content_and_find_domset (parent_cell,
                                      cell_weights);
      free ((void *) cell_weights);
      cell_weights = NULL;
   }
   disposepg (auto_gp);
   auto_gp = NULL;
} /* Search_for_dominating_set */

/*************************************************************************/

/* Print out the important constants in all the .h files */

void Print_constants (voidplist)

{
   fprintf (out, "\nN_SOLUTION_ONLY is %d.\n", N_SOLUTION_ONLY);
   fprintf (out, "PRINT_LEVEL is %d.\n", PRINT_LEVEL);
   fprintf (out, "CUT_MULTIPLE is %d.\n", CUT_MULTIPLE);
   fprintf (out, "DIVISION_FACTOR is %d.\n", DIVISION_FACTOR);
   fprintf (out, "SKIP_CHAR is %c.\n", SKIP_CHAR);
   fprintf (out, "START_STATUS is %d.\n", START_STATUS);
   fprintf (out, "MAX_CHILDREN is %d.\n", MAX_CHILDREN);
   fprintf (out, "CONSIS_CHECK is %d.\n", CONSIS_CHECK);
   fprintf (out, "COMBINE_OPTION is %d.\n", COMBINE_OPTION);
   fprintf (out, "MIN_GRAPH_SIZE is %d.\n", MIN_GRAPH_SIZE);
   fprintf (out, "MAX_GRAPH_SIZE is %d.\n", MAX_GRAPH_SIZE);
   fprintf (out, "DISCRETION is %d.\n", DISCRETION);
   fprintf (out, "STOP_LEVEL is %d.\n", STOP_LEVEL);
   fprintf (out, "\n");
   fflush(out);
} /* Print_constants */

/*************************************************************************/

/* Initialize and set the array skip_graph */
/* Input parameter:
   1) size: the size of skip_graph, total number of input graph

   Output parameter:
   1) the skip_graph array
*/

char *Set_and_initial_skip_graph (int size)
{
   int i, sg;
```

```
    char *tskip_graph;

    tskip_graph = (char *) calloc (size + 1, sizeof (char));
    for (i = 0; i <= size; i++)
      tskip_graph[i] = 0;
    /* read in the graph to be skipped */
    fscanf (in, "%d", &sg);
    while (sg != -1)
    {
       tskip_graph[sg] = 1;
       fscanf (in, "%d", &sg);
    }
    return (tskip_graph);
} /* Set_and_initial_skip_graph */

/**************************************************************************/

/* MAIN program */

void main (int argc, char *argv[])

{

   /* used for the function random */
   long state[32] = {
     3,
     0x9a319039, 0x32d9c024, 0x9b663182, 0x5da1f342,
     0x7449e56b, 0xbeb1dbb0, 0xab5c5918, 0x946554fd,
     0x8c2e680f, 0xeb3d799f, 0xb11ee0b7, 0x2d436b86,
     0xda672e2a, 0x1588ca88, 0xe369735d, 0x904f35f7,
     0xd7158fd6, 0x6fa6f051, 0x616e6b96, 0xac94efdc,
     0xde3b81e0, 0xdf0a6fb5, 0xf103bc02, 0x48f340fb,
     0x36413f93, 0xc622c298, 0xf5a42ab8, 0x8a88d77b,
     0xf5ad9d0e, 0x8999220b, 0x27fb47b9
   };

   int statesize = 128; /* the size in byte of array state */

  int temp, totalrun, i, graph_num;
  char *skip_graph;
  unsigned seed;


  if (argc < 3)
  {
     printf ("\nTo run this program, it need both the input and ");
     printf ("output file name.\n");
  }
  else
    {
       if (argv[1] && (strcmp(argv[1], "stdin") == 0))
       {
          in = stdin;
          if (argv[2] && (strcmp(argv[2], "stdout") == 0))
            out = stdout;
          else
            out = fopen(argv[2], "w");
       }
       else
         if ((in = fopen(argv[1], "r")) != NULL)
           if (argv[2] && (strcmp(argv[2], "stdout") == 0))
             out = stdout;
           else
             out = fopen(argv[2], "w");
         else
         {
            printf ("\nI could not open the file \"%s\".\n", argv[1]);
            /* Calculate_and_print_time_used (); */
            exit(10);
         }

       Print_constants ();

       /* set up the state in random */
       seed = 1;
```



```
       initstate(seed, (char *) state, statesize);
       setstate(state);

       fscanf (in,"%d", &totalrun);
       skip_graph = Set_and_initial_skip_graph (totalrun);

       for (i = 1; i <= totalrun; i++)
       {
           if (skip_graph[i])
           {
              fscanf (in, "%d", &graph_num);
              fprintf (out, "\nThe %d th graph ", i);
              fprintf (out, "with graph number %d is skipped.\n",
                             graph_num);
              goto exit_this_graph;
           }

          /* fprintf (out, "\nThis is the %d input graph.\n", i); */
          Initializations();
          if (skip_this_graph) /* total_vertices of the input graph is
                                  out of range */
            skip_this_graph = FALSE;
          else
          {
             fprintf (out, "\nThe space count after initialization.\n");
             spacecount(out);
             /* Print_graphs(2); */
             if (!finish_computation)
               Search_for_dominating_set();
             fprintf (out, "\n\nThere are %d set of solutions in total.\n",
                           total_solutions);
             Calculate_and_print_time_used ();
             Print_level_visited ();
             Reinitiate_basicdom_val ();
             Reinitiate_memmdom_val ();
             Reinitiate_groupdom_val ();
             fprintf (out, "\nThe space count after finished running ");
             fprintf (out, "this graph.\n");
             spacecount(out);
          }
          exit_this_graph: ;
          fprintf (out, "\n------------------------------------------\n");
          fflush (out);
          Goto_next_graph();
       }
       printf ("\nDONE\n");
       fclose (in);
       fclose (out);
    }
} /* main */

/**************************************************************************/
```

```
#ifndef BASICDOM

#define BASICDOM 1

#if  defined(apollo) || defined(__GNUC__)
#define HALFANSI
#define voidplist void
#else
#ifdef mips
/*   #define SYSTYPE_BSD43 */
#define HALFANSI
#define void char
#define voidplist
#else
#define voidplist void
#endif
#endif

#include "groupdcl.h"
#include "blockdcl.h"
#include <stdio.h>

/*#define fprintf  fflush(out);fprintf*/

extern void free(void *ptr);

/*********************************************************************/
/* GLOBAL CONSTANTS */
/*********************************************************************/

/* for the status in the procedure Start_gen */
#define REGULAR_PARTITION                      1
#define ORBIT_PARTITION                        2
#define TRANSITIVE_IMPRIMITIVE_PARTITION  3

/* used in the beginning of the program in domset
   to decide whether the search will start from
   symmetry partition or regular partition
   1 for regular partition
   2 for symmetry partition */
#define START_STATUS                       ORBIT_PARTITION

/* different ways to refine a cell according to its orbits */
#define NOT_COMBINE                  1 /* split each orbit into a cell */
#define COMBINE_REST_BY_DISTANCE  2 /* split the biggest orbit
                                        into a cell, combine other orbits
                                        into one cell if they have the same
                                        distance from the biggest orbit */
#define COMBINE_REST_TO_ONE          3 /* split the biggest orbit into a
                                        cell, combine other orbits into
                                        one cell */

#define COMBINE_OPTION               COMBINE_REST_BY_DISTANCE

#define FALSE           0
#define OFF             0
#define TRUE            1
#define ON              1
#define ALL_SOLUTION    0
#define ALL_LEVEL       0

#define HERTZ           60.0
#define DEBUGLISTSIZE   40 /* size of the debuglist */
#define N_SOLUTION_ONLY ALL_SOLUTION  /* 0 means find all solutions,
                                         other value means the program
                                         is forced to stop searching when
                                         finding that number of
                                         solutions */
#define PRINT_LEVEL ALL_LEVEL  /* print the array level_visited
                                  whenever this PRINT_LEVEL
                                  in level_visited is changed,
                                  0 means not print at
                                  any level */

#define CUT_MULTIPLE 1      /* use to control the number of times
```



```
                                   level_visited was flushed out,
                                   1 means it flush out level_visited
                                   whenever there is a change in
                                   level_visited[PRINT_LEVEL] */
#define NEIGH_NODE_SIZE 16  /* the maximum number of neighbor each
                                   neighbor node can represent */
#define NEW_LINE 40         /* the maximum number of integers one
                                   can print in a line. */
#define DIVISION_FACTOR 2   /* In regular partition, the parent is
                                   partitioned to DIVISION_FACTOR subcells. */
#define SKIP_CHAR '~'       /* a character used to indicate the
                                   next set of data when more than one
                                   graph is input to the program */
#define MAX_CHILDREN 3      /* The maximum number of subcells allowed
                                   for each cell partition. Old value that
                                   has problem
                                   after running long.dat for two hours is 7. */
#define CONSIS_CHECK   OFF  /* 1 to turn on the consistence checking */
#define MIN_GRAPH_SIZE 1    /* the minimum graph size */
#define MAX_GRAPH_SIZE 9999 /* the maximum graph size */
#define DISCRETION     OFF  /* 0 is no partition to discrete cell,
                                   this is used to avoid calling
                                   Generate_partition_stabilizer */
#define STOP_LEVEL     OFF  /* the level to stop, 0 means get to the end */


/*********************************************************************/
/* GLOBAL TYPES */
/*********************************************************************/

/* Data Structure to store the neighbor information of all the
   cells at each level. */

/* A sparse matrix of neigh_info_node is formed and each cell in the
   level are both the index of the row and the column.

/*********************************************************************/

/* neigh_info_node contain the information of a neighbor cell. It links
   to other neigh_info_node.

   row_cell_num, col_cell_num: They are used to indicate the
   position of this node in the sparse matrix. They are referenced
   to find out which cell node

   The cell are indexed by the cell_representative.

   tot_ver_of_deg_from_row_to_col: A pointer to a to a node which
   contain the dynamic array with
   max_degree + 1 entries. Entry x means the number of vertices with
   degree x from the corresponding row cell to the corresponding column
   cell.

   row_next_neigh, col_next_neigh, row_prev_next_neigh, col_prev_next_neigh:
   They are used to link to the next or previous neighbor cell along the row
   or column of the sparse matrix. Notice that the nodes are circular
   linked.
*/

typedef struct neigh_info_node *ptr_to_neigh_info_node;

struct neigh_info_node {

     int row_cell_num;
     int col_cell_num;
     int *tot_ver_of_deg_from_row_to_col;

     ptr_to_neigh_info_node row_next_neigh;
     ptr_to_neigh_info_node row_prev_neigh;
     ptr_to_neigh_info_node col_next_neigh;
     ptr_to_neigh_info_node col_prev_neigh;
};

/*********************************************************************/
```

```
/* Structure cell_node represents a cell and a cell
   is a collection of vertices.
   A cell will represent a partition of the graph. */

/* row_first_neigh, col_first_neigh:
   Information of the neighbors of this cell,
   both list refer to the same set of neighbors but
   the tot_ver_of_deg_from_row_to_col in the neigh_info_node
   is different. These information is needed in Coverage_test,
   Wastage_test, Update_max_allowable_wastage, and Generate_next_level
   in working out the influence functions.

   first_child, sibling are need in content partition. First,
   the first_child is accessed and then through the sibling,
   one can obtained all the cells which are most recently split.

   parent: The cell that it split from.

   first_child: The first cell that split from it.

   sibling: The other cells that split from its parent.

   next_cell, prev_cell: Used to link to all the cells in the same level.
   It is implemented as a doublely circular linked list.
   It is doubly linked to allow easy insertion and deletion
   when the cell is stacked. Node are linked in arbitrary order, but
   with the children of the most recently split parent forming a
   contiguous sublist.

   Knowing cell_representative and total_vertices one can find
   all the vertices in the cell easily.

   representative: the vertex that represent this cell should be
   the smallest in the cell. This variable is not necessary but
   it allows fast access.

   ver_before_rep: The vertex before the cell representative in the
   circular linked list. It is used to improve the speed when
   re-join two cells.

   cell_size: Total number of vertices in the cell.

   total_children: The total number of subcells split from
   this cell. It is used to allocate the size of the array
   lower_limit_distribution, upper_limit_distribution, and
   actual_content_distribution, which is used to store the content
   of each children.

   lower_content, upper_content: They are calculated when the
   cell split from its parent. They are calculated by the
   coverage and wastage test. They are used to limit the range
   of content partitioned to each most recently partitioned
   subcell.

   actual_content: The content assigned to this cell.

   max_allow_wastage: After the actual_content is defined,
   the total coverage among all the neighbor must be bigger than
   the total_vertices in this cell. The max_allow_wastage is used
   for this test.
*/

typedef struct cell_node *ptr_to_cell_node;

struct cell_node {

     ptr_to_neigh_info_node row_first_neigh;
     ptr_to_neigh_info_node col_first_neigh;

     ptr_to_cell_node parent;
     ptr_to_cell_node sibling;
     ptr_to_cell_node first_child;
     ptr_to_cell_node prev_cell;
     ptr_to_cell_node next_cell;
```



```
     int representative;
     int ver_before_rep;
     int cell_size;
     int total_children;
     int lower_content;
     int upper_content;
     int actual_content;
     int max_allow_wastage;
};

/*******************************************************************/

/* Part of the data structure of the neighbor_graph. */
/* neighbor_graph is an array of next_neigh_node indexed by the
   vertices of the graph. Each entry of the array contains a list
   of vertices which is the neighbor of the index, each next_neigh_node
   contains a maximum of NEIGH_NODE_SIZE neighbors. */

typedef struct next_neigh_node *ptr_to_next_neigh_node;

/* a link list */
struct next_neigh_node {
           int neighbor_name[NEIGH_NODE_SIZE];
           struct next_neigh_node *next;
};

/*******************************************************************/

typedef struct general_node *ptr_to_general_node;

/* a general double link list which store integer */

struct general_node {
   int size;
   int value;
   ptr_to_general_node next;
   ptr_to_general_node last;
};

/*******************************************************************/

/* to seperate the regular partition and symmetric partition */

enum part_type {REGULAR, SYMMETRY};

/*******************************************************************/
/* GLOBAL VARIABLES */
/*******************************************************************/

/* The following is an array of integer use to turn on or off the debug
   mode in a procedure. */

extern int debuglist[];

/*******************************************************************/

/* Array one to total_vertices of pointer to cells. It is a global
   array and stored all the cells at current level. */

extern ptr_to_cell_node *cell_list;

/*******************************************************************/

/* dummy_cell links all the cells in the current level. It is a
   circular link list. */
/* The total_children field indicates the total number of cells in the
   list. */

extern ptr_to_cell_node dummy_cell;

/*******************************************************************/

/* Representation of the cells. */

/* allcells:  It contains the distribution of the vertices in all the
```

```
   cells. It is a global array with size equal to the total_vertices
   + 1. It needs to be updated at each level of the recursion of the
   program. Vertices in the same cell are stored in cyclic from.

   cell_label: It is a global array of size max_level. It maps
   each vertex to the corresponding cell_representative. It need to be
   updated at each level of recursion.
*/

extern int *allcells;
extern int *cell_label;

/*******************************************************************/

/* Data Structure of the Graph */

/* Since the data structure of the graph will not be partitioned,
   it may be helpful to have more than one data structure to represent the
   same graph in order to facilitate different applications. The first
   data structure will be using an adjacent matrix. It will be
   implemented on a two dimension char array.  */

extern char **adj_matrix;

/*******************************************************************/

/* A second representation of the graph, a dynamic array of total_vertices
   entries of pointer to a list of next_neighbor_node. */

extern struct next_neigh_node **neigh_graph;

/*******************************************************************/

/* An array to store all the tot_ver_of_deg_from_row_to_col
   entries when working out each new neighbor information node. */

extern int ***all_tot_ver_deg_r_c;

/*******************************************************************/

/* vertex_degree is an array which contains the degree of each vertex. */

extern int *vertex_degree;

/*******************************************************************/

extern int stop_level, /* the level to stop */
      given_level,    /* the number of level given */
      total_vertices, /* the total number of vertices of the graph */
      org_total_content, /* used when user input the cell partition of a
                            weighted graph, this represents the total content
                           expected in the original non-weighted graph */
      total_content, /* the total number of contents allowed */
      total_solutions, /* record the total number of solutions */
      total_first_part_subcells, /* the total_number of
                            subcells partitioned from the first cell */
      max_level, /* the maximum number of levels the recursion can go */
      max_neigh_stack, /* the maximum size of the neighbor stack */
      max_degree, /* the maximum outgoing degree among all the
                           vertices */
      start_level, /* the level the program will start with */

      /* boolean variables */
      input_vertex_start_from_1,    /* true if the input vertex starts
                                       from 0 */
      finish_computation, /* to exit all the recursions */
      skip_this_graph,   /* to skip the current graph */
      direct,                      /* true, when the graph is direct */
      automated_cell_partition,    /* false, when user specified the
                                      initial cell partition */
      automated_content_partition, /* false, when user specified the
                                      initial content partition */
      global_level; /* the level the program is working */

/* The following variables are used to control the static variable.
```



```
   When this program is run for more than one graph, those static
   variables are needed to be reset. */

extern int reset_connected_to_cell,
     reset_s_slack,
     reset_unavoidable_wastage,
     reset_old_max_allowable_wastage,
     reset_all_vertices_set,
     reset_cell_counts;


/*******************************************************************/

extern long  start_time, finish_time, elapsed_time;
extern struct tms *sbuffer, *fbuffer, *ebuffer;

/*******************************************************************/

extern FILE *in, *out;

/*******************************************************************/

extern unsigned *level_visited, /* an array to keep record the number of
                                   times each level was visited */
                       /* The following four variables are arrays
                          to keep track the situation when the number
                          of orbit in a cell is more than one. */
         *bfcwtest,  /* before coverage and wastage test */
         *afcwtest,  /* after coverage and wastage test */
         *bfmawtest, /* before maximum allowable wastage test */
         *afmawtest; /* after maximum allowable wastage test */

/*******************************************************************/

/* used for doing estimation
   N.B. this program is designed to do estimation on one graph */

extern double *total_alpha;
extern double totalvisit;
extern double totaltime;
extern int totalruns;

/*******************************************************************/
/* FUNCTIONS */
/*******************************************************************/

/* The following is a list of procedures for printing results. */

/*******************************************************************/

/* Print an integer array with size n. */

/* input parameters
   n: the size of the array,
   array_ptr: the pointer to the beginning of the array
*/

extern void Print_int_array (int n, int *array_ptr);

/*******************************************************************/

/* Print the status of allcells, cell_label, all the neighbor
   information of each cell.
*/

extern void Print_curr_status (voidplist);

/*******************************************************************/

/* Print_graphs will try to print the two graphs out. */

/* Input parameters;
   graph_type: used to indicated the type to be printed.
               1 = adjacency matrix
               2 = neighbor graph
```

```
             3 = both
*/

extern void Print_graphs (int graph_type);

/*********************************************************************/

/* Print the vertices of a cell. */

/* input parameters:
   1) cellrep: the cell representative
*/

extern void Print_cell_vertices (int cellrep);

/*********************************************************************/

/* Print each input cell's vertices and the weight of this cell. */

/* input parameters:
   1) parent: the parent pointer
   2) cweights: the weight of the input cells
*/

extern void Print_input_partition_and_weight (ptr_to_cell_node parent,
                                              int *cweight);

/*********************************************************************/

/* Print_children will print all the children of a cell. */

/* input parameters:
   1) parent_cell: the parent
*/

extern void Print_sibling (ptr_to_cell_node parent_cell);

/*********************************************************************/

/* The following is a list of utility procedures. */

/*********************************************************************/

/* Quicksort will sort an array, a, with n elements starting from
   position 0 to position n-1.
   Array will be sorted in ascending order. */

void Quicksort(int a[], int n);

/*********************************************************************/

/* To check if all the cells in this level is consistent, the invariant
   checked will include:

   1) check if cell_list and the cells in dummy list agree
   1) no repeat cell contain in the neighbor cell list of a cell
   2) the neighbor node of a cell is actually its neighbor
   3) the fields in a cell node is correct
   5) check if allcells and cell_label agree
*/

void Consistence_check (voidplist);

/*********************************************************************/

/* This procedure will skip all the related data of this graph by
   reading all the input data until a SKIP_CHAR is found. */

extern void Skip_this_graph (voidplist);

/*********************************************************************/

/* Go to the start of next input character by reading all the SKIP_CHAR
   before the next set of input character, there should be a series of
   '~' and ending with 2 blank character. */
```



```
extern void Goto_next_graph (voidplist);

/*********************************************************************/

/* Initialize an integer array from position i to j with value equal to x.
   intarray must be allocated.

   Input parameters:
   1) intarray: the pointer to the integer array
   2) i: start position
   3) j: end position
   4) x: the value to be initialized

   Side effect:
   intarray was initialized.
*/

extern void Init_int_array (int *intarray, int i, int j, int x);

/*********************************************************************/

/* Initiate all the timing variable. */

extern void Init_time_val (voidplist);

/*********************************************************************/

/* Initiate the first cell, allcells and cell_label. */

/* debug list value = 9 */

extern void Init_first_cell (voidplist);

/*********************************************************************/

/* To handle the content partition generated like
   Update_the_children_contents, Update_ax_allowable_wastage,
   Select_next_partition_cell, Coverage_test, Wastage_test,
   check if that partition lead to a solution.
*/

/* debuglist = 8 */

extern void Handle_reg_content_part (int current_level,
                                     ptr_to_cell_node parent_ptr,
                                     ptr_to_perm_gp gptr,
                                     int array_vol,
                                     int *content_distribution);

/*********************************************************************/

/* Create_first_level will create the first level,
   variable automated_cell_partition need to be initialized
   before calling this procedure. */
/* side effect: automated_content_partition is updated here */

/* debug list value = 11 */

extern void Create_first_level (voidplist);

/*********************************************************************/

/* Select_next_partition_cell will try to select the next cell for doing
   a partition of cell, if such a cell exist, the address of that cell
   will be returned, otherwise, a NULL pointer will be returned
   it will select the cell with total vertices > 1, with minimum
   max_allow_wastage. In case of tide, it select the first cell with
   more vertices

   Output parameter
   the cell selected to be partition
*/

extern ptr_to_cell_node Select_next_partition_cell (voidplist);
```

```
/*********************************************************************/

/* Initiate basicdom variables will initiate all the variables
   in basicdom.c and read in the input data.

   side effects: memory allocation and fscanf is used here */

/* global variables initialize or allocated here are:
   input_vertex_start_from_1,
   any_debug, debuglist,
   direct,
   total_vertices,
   max_level,
   level_visited,
   adj_matrix,
   neigh_graph,
*/

/* debug list value = 10 */

extern void Initiate_basicdom_val (voidplist);

/*********************************************************************/

/* User input the first cell partitions, user is respondable for a
   correct set of cell partition.

   Input parameter:
   1) parent_ptr: pointer to parent cell

*/

extern void Input_first_cell_partition (ptr_to_cell_node parent_ptr);

/*********************************************************************/

/* This procedure will reinitialize all the variables required to make
   the program run for another set of user input partition cell and
   content given the same graph. */

void Reinitiate_basicdom_val (voidplist);

/*********************************************************************/

#endif
```

```
#include <stdio.h>
#include "memmdom.h"
#include "basicdom.h"
#include "groupdom.h"
#include <sys/types.h>
#include <sys/times.h>

/*********************************************************************/

/* In this file, it contains all the basic operations required to
   solve the dominating set problem. The program
   initiates the global variables and creates the
   first cell and level. All the group operations are stored in
   the file groupdom.c.
*/

/*********************************************************************/
/* GLOBAL TYPES */
/*********************************************************************/

typedef struct tms *ptr_to_tms;

/*********************************************************************/
/* GLOBAL VARIABLES */
/*********************************************************************/

/* We make use of global variable to save time in
   parameter passing. */

/* The following is an array of integer.
   It is used to turn on or off the debug
   mode in a procedure. */

int debuglist[DEBUGLISTSIZE+1];

/*********************************************************************/

/* Array one to total_vertices of pointer to cells. It is a global
   array which stores all the cells at the current level. */

ptr_to_cell_node *cell_list;

/*********************************************************************/

/* dummy_cell link all the cells in the current level. It is a
   circular link list. */
/* The total_children field indicates the total number of cells in the
   list. */

ptr_to_cell_node dummy_cell;

/*********************************************************************/

/* Representation of the cells. */

/* allcells:  It contains the distribution of the vertices in all the
   cells. It is a global array with size equal to total_vertices
   + 1. It needs to be updated at each level of the recursion in the
   program. Vertices in the same cell are packed together. There is
   no need to sort the vertices.

   cell_label: It is a global array of size max_level. It maps
   each vertex to the corresponding cell_representative. It need to be
   updated at each level of recursion.
*/

int *allcells;
int *cell_label;

/*********************************************************************/

/* Data Structure of the Graph. */

/* Since the graph will not be partitioned,
   we decided to have more than one data structure to represent the
```



```
   same graph in order to facilitate different applications. The first
   data structure is an adjacent matrix. It is
   implemented on a two dimension char array.  */

char **adj_matrix;

/*********************************************************************/

/* Second representation of the graph, a dynamic array of total_vertices
   entries of pointer to a list of next_neighbor_node. */

ptr_to_next_neigh_node *neigh_graph;

/*********************************************************************/

/* An array to store all the tot_ver_of_deg_from_row_to_col
   entries when working out each new neighbor information node. */

int ***all_tot_ver_deg_r_c;

/*********************************************************************/

/* vertex_degree is an array which contain the degree of each vertex. */

int *vertex_degree;

/*********************************************************************/

int given_level,       /* the number of level given */
    total_vertices,    /* the total number of vertices of the graph */
    org_total_content, /* used when user input the cell partition of a
                          weighted graph, this represents the total content
                          expected in the original (non-weighted) graph */
    total_content,     /* the total number of contents allowed */
    total_solutions = 0, /* the total number of solutions */
    total_first_part_subcells = 0, /* the total_number of
                            subcells partitioned from the first cell */
    max_level,         /* the maximum number of levels
                          the recursion could go */
    max_degree = 0,    /* the maximum outgoing degree among all the
                          vertices */
    start_level,       /* the level the program start */

    /* boolean variables */
    input_vertex_start_from_1,    /* true if the input vertex starts
                                     from 0 */
    finish_computation = FALSE,   /* to exit all the recursions */
    skip_this_graph = FALSE,      /* to skip the current graph */
    direct,                       /* true, when the graph is direct */
    automated_cell_partition,     /* false, when user specified the initial
                                     cell partition */
    automated_content_partition,  /* false, when user specified the initial
                                     content partition */
    global_level = 0;             /* the level the program is working */

/* The following variables are used to control the static variable.
   When this program is run for more than one graph, those static
   variables have to be reset to FALSE. */

int reset_connected_to_cell = FALSE,
    reset_s_slack = FALSE,
    reset_unavoidable_wastage = FALSE,
    reset_old_max_allowable_wastage = FALSE,
    reset_all_vertices_set = FALSE,
    reset_cell_counts = FALSE;

/*********************************************************************/

long start_time, finish_time, elapsed_time;
ptr_to_tms sbuffer, fbuffer, ebuffer;

/*********************************************************************/

FILE *in, *out;
```

```
/***********************************************************************/

unsigned *level_visited, /* An array to keep track of the number of
                            times each level was visited. */
                         /* The following four variables are arrays
                            to keep track of the situation when the number
                            of orbit in a cell is bigger than one. */
         *total_content_part, /* The total number of content partitions
                                 (including those that do not
                                 pass the coverage and wastage test)
                                 generated at each level */
         *bfcwtest,           /* before coverage and wastage test */
         *afcwtest,           /* after coverage and wastage test */
         *bfmawtest,          /* before maximum allowable wastage test */
         *afmawtest;          /* after maximum allowable wastage test */

/***********************************************************************/
/* FUNCTIONS */
/***********************************************************************/

/* The following is a list of procedures used to print out results. */

/***********************************************************************/

/* Print_graphs prints the two graphs. */
/* This procedure is only used for debug. */

/* Input parameters:
   graph_type: used to indicated the type to be printed.
               1 = adjacence matrix
               2 = neighbor graph
               3 = both
*/

void Print_graphs (int graph_type)
{
  int i, j;
  ptr_to_next_neigh_node tempptr;

  /* adj_matrix */
  if ((graph_type == 1) || (graph_type == 3))
  {
     fprintf (out, "\nThe following is the adjacent matrix:\n");

     for (i = 1; i <= total_vertices; i++)
     {
       for (j = 1; j <= total_vertices; j++)
       {
          fprintf (out,  "%u ", adj_matrix[i][j]);
       } /* for j-loop */
       fprintf (out,  "\n" );
     } /* for i-loop */
  }

  /* neighbor_graph */
  if ((graph_type == 2) || (graph_type == 3))
  {
     fprintf (out, "\n");
     fprintf (out, "The following is the neighbor graph:\n");

     for (i = 1; i <= total_vertices; i++)
     {
        fprintf (out, "%d: ", i);
        tempptr = neigh_graph[i];
        while (tempptr != NULL)
        {
           for (j = 0; j < NEIGH_NODE_SIZE; j++)
             if (tempptr->neighbor_name[j] == -1)
               break;
             else
             {
               fprintf (out, "%d ", tempptr->neighbor_name[j]);
             }
           tempptr = tempptr->next;
        } /* while */
```



```
        fprintf (out, "\n");
     } /* for i-loop */
  }
} /* Print_graphs */

/***********************************************************************/

/* Print all the vertices in a cell. */

/* Input parameter:
   cellrep: the cell's representative.
*/

void Print_cell_vertices(int cellrep)
{
   int cpos;

   cpos = cellrep;

   fprintf (out, "Cell vertices: ");
   do {
     fprintf (out, "%d ", cpos);
     cpos = allcells[cpos];
   } while (cpos != cellrep);
   fprintf (out, "\n");
} /* Print_cell_vertices */

/***********************************************************************/

/* Print_cell prints all the entry of a cell,
   this procedure is used for debug. */

/* Input:
   1) cell_ptr: the cell to be printed.
*/

void Print_cell (ptr_to_cell_node cell_ptr)

{
  int short_form = 0;

  if (cell_ptr == NULL)
  {
    fprintf (out, "NULL\n");
  }
  else
  {
     fprintf (out, "\n");
     fprintf (out, "cell_node = %x\n", cell_ptr);

     if (!short_form)
     {
        fprintf (out, "row_first_neigh's address = %x\n",
                 cell_ptr->row_first_neigh);
        fprintf (out, "col_first_neigh's address = %x\n",
                 cell_ptr->col_first_neigh);

        fprintf (out, "parent's address = %x\n", cell_ptr->parent);
        fprintf (out, "sibling's address = %x\n", cell_ptr->sibling);
        fprintf (out, "first_child's address = %x\n", cell_ptr->first_child);
        fprintf (out, "prev_cell's address = %x\n", cell_ptr->prev_cell);
        fprintf (out, "next_cell's address = %x\n", cell_ptr->next_cell);

        fprintf (out, "cell representative = %d\n", cell_ptr->representative);
        fprintf (out, "vertex before the cell representative = %d\n",
                       cell_ptr->ver_before_rep);
        fprintf (out, "cell_size = %d\n", cell_ptr->cell_size);
        fprintf (out, "total_children = %d\n", cell_ptr->total_children);
        fprintf (out, "lower_content = %d ", cell_ptr->lower_content);
        fprintf (out, "upper_content = %d ", cell_ptr->upper_content);
        fprintf (out, "actual_content = %d\n", cell_ptr->actual_content);
        fprintf (out, "max_allow_wastage = %d\n", cell_ptr->max_allow_wastage);
     }
     else
     {
```

```
      fprintf (out, "lower_content = %d ", cell_ptr->lower_content);
      fprintf (out, "upper_content = %d ", cell_ptr->upper_content);
      fprintf (out, "actual_content = %d\n", cell_ptr->actual_content);
      fprintf (out, "max_allow_wastage = %d\n", cell_ptr->max_allow_wastage);
    }
    if (cell_ptr->cell_size >0)
      Print_cell_vertices(cell_ptr->representative);
  }
  fprintf (out, "\n");
} /* Print_cell */

/**********************************************************************/

/* Print each input cell's vertices and weight. */

/* input parameters:
   1) parent: the parent pointer
   2) cweights: the weight of the input cells
*/

void Print_input_partition_and_weight (ptr_to_cell_node parent,
                                       int *cweight)

{
   int i = 0;
   ptr_to_cell_node cell_ptr;

   cell_ptr = parent->first_child;

   while (cell_ptr != NULL)
   {
      fprintf (out, "The %d cell's weight is %d.\n", i+1, cweight[i]);
      Print_cell_vertices (cell_ptr->representative);
      cell_ptr = cell_ptr->sibling;
      i++;
   }
} /* Print_input_partition_and_weight */

/**********************************************************************/

/* Print_children prints all the children of a cell. */

/* input parameters:
   1) parent_cell: the parent
*/

void Print_children(ptr_to_cell_node parent_cell)

{
  ptr_to_cell_node ptr;

  fprintf (out, "\nThe following is a list of children\n");
  ptr = parent_cell->first_child;
  while (ptr != NULL)
  {
     Print_cell(ptr);
     ptr = ptr->sibling;
  } /* while */
} /* Print_sibling */

/**********************************************************************/

/* Print an integer array with size n. */

/* input parameters
   n: the size of the array,
   array_ptr: the pointer that points to the beginning of the array.
*/

void Print_int_array (int n, int *array_ptr)

{
   int i;

   for (i = 0; i < n; i++)
```



```
   {
     fprintf (out, "%d ", array_ptr[i]);
     if ((i % NEW_LINE) == 0)
     {
       fprintf (out, "\n");
     }
   }
   fprintf (out, "\n");
} /* Print_int_array */

/**********************************************************************/

/* Print the array allcells and cell_label. */

void Print_allcells_and_cell_label (voidplist)

{
   fprintf (out, "allcells:\n");
   Print_int_array (max_level, allcells);

   fprintf (out, "\ncell_label:\n");
   Print_int_array (max_level, cell_label);
} /* Print_allcells_and_cell_label */

/**********************************************************************/

/* Work and print out all the time used. */

void Calculate_and_print_time_used (voidplist)

{
  finish_time = times(fbuffer);

  /* user time */
  ebuffer->tms_utime = fbuffer->tms_utime - sbuffer->tms_utime;
  /* system time */
  ebuffer->tms_stime = fbuffer->tms_stime - sbuffer->tms_stime;

  fprintf (out, "\nProcess's CPU time is %ld HZ ",
           ebuffer->tms_utime);
  fprintf (out, "or %f seconds.\n", ebuffer->tms_utime/HERTZ);
  fprintf (out, "System's CPU time ");
  fprintf (out, "is %ld HZ or %f seconds.\n",
           ebuffer->tms_stime, ebuffer->tms_stime/HERTZ);
  fprintf (out, "Sum of the two times are ");
  fprintf (out, "%f seconds.\n",
           (ebuffer->tms_utime + ebuffer->tms_stime)/HERTZ);

  fflush (out);
} /* Calculate_and_print_time_used */

/**********************************************************************/

/* To print out the information in a neigh_info_node.

   Input parameter:
   1) neigh_info: pointing to the neigh_info_node that is to be printed
*/

void Print_neigh_info (ptr_to_neigh_info_node neigh_info)
{
  if (neigh_info == NULL)
  {
    fprintf (out, "NULL\n");
  }
  else
  {
     fprintf (out, "\n");
     fprintf (out, "neigh_info_node = %x\n", neigh_info);
     fprintf (out, "row_cell_num  = %d\n", neigh_info->row_cell_num);
     fprintf (out, "col_cell_num  = %d\n", neigh_info->col_cell_num);

     fprintf (out, "The array's address = %x\n",
                   neigh_info->tot_ver_of_deg_from_row_to_col);
     if (neigh_info->tot_ver_of_deg_from_row_to_col != NULL)
```

```
    {
       fprintf (out, "The tot_ver_of_deg_from_row_to_col array:\n");
       Print_int_array (max_degree + 1,
                        neigh_info->tot_ver_of_deg_from_row_to_col);
    }

    fprintf (out, "row_next_neigh's address = %x\n",
                  neigh_info->row_next_neigh);
    fprintf (out, "row_prev_neigh's address = %x\n",
                  neigh_info->row_prev_neigh);
    fprintf (out, "col_next_neigh's address = %x\n",
                  neigh_info->col_next_neigh);
    fprintf (out, "col_prev_neigh's address = %x\n",
                  neigh_info->col_prev_neigh);
  }
} /* Print_neigh_info */

/*********************************************************************/

/* To print out the list of neighbor's information of the master cell.
   It traverse along the row_next_neigh of the cell's row_first_neigh.

   Input parameter:
   1) master_cell: the cell's neighbor information to be printed
*/

void Print_all_row_neigh_info (ptr_to_cell_node master_cell)
{
   ptr_to_neigh_info_node curr_neigh_info, dummy_neigh;

   fprintf (out, "\nThe following are all the row neighbor nodes");
   fprintf (out, " of the master cell %x.\n", master_cell);
   curr_neigh_info =  dummy_neigh =  master_cell->row_first_neigh;

   Print_neigh_info (curr_neigh_info);
   curr_neigh_info = curr_neigh_info->row_next_neigh;

   while (curr_neigh_info != dummy_neigh)
   {
      Print_neigh_info (curr_neigh_info);
      curr_neigh_info = curr_neigh_info->row_next_neigh;
   }
} /* Print_all_row_neigh_info */

/*********************************************************************/

/* To print out the list of neighbor's information of the master cell.
   It traverse along the col_next_neigh of the cell's col_first_neigh.

   Input parameter:
   1) master_cell: the cell's neighbor information to be printed
*/

void Print_all_col_neigh_info (ptr_to_cell_node master_cell)
{
   ptr_to_neigh_info_node curr_neigh_info, dummy_neigh;

   fprintf (out, "\nThe following are all the column neighbor nodes");
   fprintf (out, " of the master cell %x.\n", master_cell);
   curr_neigh_info = dummy_neigh = master_cell->col_first_neigh;

   Print_neigh_info (curr_neigh_info);
   curr_neigh_info = curr_neigh_info->col_next_neigh;

   while (curr_neigh_info != dummy_neigh)
   {
      Print_neigh_info (curr_neigh_info);
      curr_neigh_info = curr_neigh_info->col_next_neigh;
   }
} /* Print_all_col_neigh_info */

/*********************************************************************/

/* Print the 3-D array all_tot_ver_deg_r_c. It has size
   max_level * max_level * max_degree */
```



```
void Print_all_tot_ver_deg_r_c(voidplist)
{
   int i, j, k;

   fprintf(out, "\nThe following is the all_tot_ver_deg_r_c array with");
   fprintf(out, " size = %d x %d x %d.\n", total_vertices,
           max_level - 1, max_degree + 1);
   for (i = 1; i <= total_vertices; i++)
   {
      for (j = 1; j <= total_vertices; j++)
      {
         for (k = 0; k <= max_degree; k++)
         {
            fprintf(out, "%d ", all_tot_ver_deg_r_c[i][j][k]);
         }
         fprintf(out, "  ");
      }
      fprintf(out, "\n");
   }
} /* Print_all_tot_ver_deg_r_c */

/*********************************************************************/

/* Print out all the available cells at this level. */

void Print_available_cells (voidplist)
{
   ptr_to_cell_node tempcell;

   /* Print out all the result generated */
   fprintf (out, "\nThis is the dummy cell.\n");
   Print_cell (dummy_cell);
   for (tempcell = dummy_cell->next_cell;
        tempcell != dummy_cell;
        tempcell = tempcell->next_cell)
   {
      fprintf (out, "\nPrinting information of cell %d.\n",
                    tempcell->representative);
      Print_cell (tempcell);
/*
      Print_all_row_neigh_info (tempcell);
      Print_all_col_neigh_info (tempcell);
*/
   }
} /* Print_available_cells */

/*********************************************************************/

/* Print out the information in the cell_list */

void Print_cell_list (voidplist)

{
   int i;

   fprintf (out, "\nThis is the cell_list array.\n");
   for (i = 0; i <= total_vertices; i++)
   {
      fprintf (out, "\n%d cell:\n", i);
      if (cell_list[i] != NULL)
      {
         Print_cell (cell_list[i]);
      }
      else
      {
         fprintf (out, "NULL\n");
      }
   }

} /* Print_cell_list */

/*********************************************************************/

/* Print the status of allcells, cell_label, all the neighbor
```

```
   information of each cell, and all the stacks.
*/

void Print_curr_status (voidplist)
{
   Print_allcells_and_cell_label();
   Print_available_cells();
/*
   Print_all_tot_ver_deg_r_c ();
   Print_cell_list ();
*/
} /* Print_curr_status */

/********************************************************************/

/* Print out the content_part and array level_visited. */
/* N.B. If the i th entry of level_visited = 0, all the entries
        below it will also = 0, hence it is not necessary to print
        them out. */
/* Since level 0 will always equal to 0 so it is not going to
   be printed. */

void Print_level_visited (voidplist)

{
   int i;
   unsigned total_succ_lev = 0, total_level = 0;

   fprintf (out, "\nThe number of times each level was visited.\n");
   for (i=1; (i < max_level) && (level_visited[i] > 0); i++)
   {
      fprintf (out, "%d: %u, %u\n", i, total_content_part[i-1], level_visited[i]);
      total_level += total_content_part[i-1];
      total_succ_lev += level_visited[i];
   } /* for i-loop */
   fprintf (out, "\n");
   fprintf (out, "The content partition generated at all levels: ");
   fprintf (out, "%u\n", total_level);
   fprintf (out, "The successful total levels visited are: ");
   fprintf (out, "%u\n", total_succ_lev);

/*
   fprintf (out, "\nThe following is a print out of each level of\n");
   fprintf (out, "procedure at Handle_non_transitivity, the number of\n");
   fprintf (out, "calls before and after Coverage_and_wastage_test,\n");
   fprintf (out, "these tests are done before the content of the parent\n");
   fprintf (out, "is partitioned into its children.\n");
   fprintf (out, "After the content is partitioned, at procedure\n");
   fprintf (out, "Handle_orb_content_part, the number of calls\n");
   fprintf (out, "before and after Update_max_allowable_wastage\n");
   fprintf (out, "are also printed.\n");

   fprintf (out, "\n level  bf c w  af c w    bf maw  af maw\n");
   for (i=1; i < max_level; i++)
   {
      fprintf (out, "%6d %7u %7u   %7u %7u\n",
        i, bfcwtest[i], afcwtest[i], bfmawtest[i], afmawtest[i]);
   }
   fprintf (out, "\n");
*/
} /* Print_level_visited */

/********************************************************************/

/* Print out the input information. */

void Print_input_info (voidplist)
{
  if (input_vertex_start_from_1)
  {
     fprintf (out, "\nInput vertices starts from one.\n");
  }
  else
  {
     fprintf (out, "\nInput verteices starts from zero.\n");
```



```
  }
  fprintf (out, "\nDebug list:\n");
  Print_int_array (DEBUGLISTSIZE + 1, debuglist);

  if (direct)
  {
     fprintf (out, "\nDirect Graph.\n");
  }
  else
  {
     fprintf (out, "\nIndirect Graph.\n");
  }

  fprintf (out, "\nThe total number of vertices in this graph are %d.\n",
                 total_vertices);

  fprintf (out,  "cell_calloc_size = %d\n", cell_calloc_size);
  fprintf (out,  "neigh_calloc_size = %d\n", neigh_calloc_size);
  fprintf (out,  "tvdrc_calloc_size = %d\n", tvdrc_calloc_size);
  fprintf (out,  "max_level = %d\n", max_level);
  Print_graphs(3);
  fprintf (out,  "max_degree = %d\n", max_degree);
  Print_curr_status ();
} /* Print_input_info */

/********************************************************************/

/* The following is a list of utility procedures. */

/********************************************************************/

/* Quicksort will sort an array, a, with n elements starting from
   position 0 to position n-1.
   The program is
   modified from R. Nigel Horspool, programming in the
   Berkely UNIX environment, 1986, p.66. The original
   program contains some bugs.  The sorted
   array is in ascending order. */

void Quicksort(int a[], int n)

{
  int i, lpos, rpos, pivot, tmp;

  /* handle case of n <= 2 */

  if (n <= 2)
    {
      if ((n == 2) && (a[0] > a[1]))
        {
          tmp = a[0];
          a[0] = a[1];
          a[1] = tmp;
        }
      return;
    }
  pivot = a[0];
  lpos = 1; /* the above if condition has already guaranteed n > 2 */
  rpos = n-1;

  while (lpos < rpos)
    {
      while (a[rpos] >= pivot)
        {
          if (--rpos == lpos) goto EXIT1;
        }

      /* the second condition is to avoid the possibility of
         running out of bound in the case like 2 2 2 2 */

      while ((a[lpos] <= pivot) && (lpos < (n-1)))
        {
          if (++lpos == rpos) goto EXIT1;
        }
```

```
      tmp         = a[lpos];
      a[lpos++] = a[rpos];
      a[rpos--] = tmp;
    }

EXIT1:
  if (a[lpos] > pivot)
    {
      a[0] = a[--lpos];
      a[lpos] = pivot;
    }
  else
    {
      a[0] = a[lpos];
      a[lpos] = pivot;
    }
  Quicksort(a, lpos);
  Quicksort(a + lpos + 1, n - lpos - 1);
} /* Quicksort */

/*******************************************************************/

/* To force the program to exist with a core dump. */

void Gen_core (voidplist)
{
   ptr_to_cell_node tcell = NULL;

   fprintf (out, "%d\n", tcell->cell_size);
} /* Gen_core */

/*******************************************************************/

/* To force the program to exit. */

/* Input parameter:
   exitvalue: 0 no core dump
              1 with core dump
*/

void Myexit(int exitvalue)
{
   fprintf (out, "\nThe level before calling exit is: %d",
            global_level);
   Print_level_visited();
   Calculate_and_print_time_used();
   fflush (out);
   printf ("\007");
   if (exitvalue)
   {
     printf ("\nProgram exited abnormally with core dumped.\n");
/*
     Gen_core();
*/
   }
   else
   {
     printf ("\nProgram exited abnormally with no core dumped.\n");
     exit (100);
   }
} /* Myexit*/

/*******************************************************************/

/* Check the consistence of each neigh_info_node of a cell. */

/* Input parameters:
   1) wcell: the cell_node that we are working on.
*/

void Check_neighbor_nodes (ptr_to_cell_node wcell)
{
   /* to make sure there is no cell duplicated in the neighbor list */
   static *cell_counts = NULL;
```



```
   int i;
   ptr_to_neigh_info_node tneigh, dummy_neigh;
   ptr_to_cell_node tcell;

   if ((cell_counts == NULL) || reset_cell_counts)
   {
      if (cell_counts != NULL)
      {
         reset_cell_counts = FALSE;
         free ((void *) cell_counts);
         cell_counts = NULL;
      }
      cell_counts = (int *) calloc (max_level, sizeof(int));
      for (i = 0; i < max_level; i++)
        cell_counts[i] = 0;
   }

} /* Check_neighbor_nodes */

/*******************************************************************/

/* Check the consistence of each cell in this level. */

void Check_cells (voidplist)
{
   int i;

   /* for each cells in this level check its neighbor info node */
   for (i = 1; i <= total_vertices; i++)
   {
      if (cell_list[i] != NULL)
      {
         Check_neighbor_nodes(cell_list[i]);
      }
   }

} /* Check_cells */

/*******************************************************************/

/* To check if all the cells in this level is consistent, the invariant
   check includes:

   1) check if cell_list and the cells in dummy list agree,
   1) no repeat cell contain in the neighbor cell list of a cell,
   2) the neighbor node of a cell is actually its neighbor,
   3) the fields in a cell node is correct,
   5) check if allcells and cell_label agree.
*/

void Consistence_check (voidplist)

{
   int i, count = 0;
   ptr_to_cell_node tcell;

   /* check if the cells in dummy list agree with the cells in cell_list */
   tcell = dummy_cell->next_cell;
   while (tcell != dummy_cell)
   {
      count++;
      if (tcell != cell_list[tcell->representative])
      {
         fprintf (out, "\nThe cell in dummy_cell list does not match ");
         fprintf (out, "the cell in array cell_list.\n");
         fprintf (out, "The cell in dummy_list:");
         Print_cell(tcell);
         fprintf (out, "\nThe cell in cell_list:");
         Print_cell(cell_list[tcell->representative]);
         Myexit(1);
      }
      tcell = tcell->next_cell;
   }

   /* check if the total number of cells linked by dummy_cell agree */
```

```c
  if (count != dummy_cell->total_children)
  {
     fprintf (out, "\nThe total cell linked by dummy cell %d \n",
              count);
     fprintf (out, "do not agree with the value stored in dummy_cell %d\n.",
              dummy_cell->total_children);
     Myexit(1);
  }

  /* check if the total number of cells in dummy_cell
     is the same as the total number of cells in cell_list */
  count = 0;
  for (i = 1; i < max_level; i++)
  {
     if (cell_list[i] != NULL)
       count++;
  }
  if (count != dummy_cell->total_children)
  {
     fprintf (out, "\nThe total cell linked by dummy cell %d \n",
              dummy_cell->total_children);
     fprintf (out, "do not agree with the number of cells in cell_list%d.\n",
              count);
     Myexit(1);
  }

  Check_cells();

} /* Consistence_check */

/*******************************************************************/

/* This procedure will skip all the related data of this graph by
   reading all the input data until a SKIP_CHAR is found. */

void Skip_this_graph (voidplist)

{
   char ch;

   fscanf (in, "%c", &ch);
   while (ch != SKIP_CHAR)
     fscanf (in, "%c", &ch);
} /* Skip_this_graph */

/*******************************************************************/

/* Go to the start of next input character by reading all the SKIP_CHAR
   before the next set of input character, there should be a series of
   '-' and ending with 2 blank character. */

void Goto_next_graph (voidplist)

{
  char ch;

  fscanf (in, "%c", &ch);
  while (ch != SKIP_CHAR)
    fscanf (in, "%c", &ch);
  while (ch == SKIP_CHAR)
    fscanf (in, "%c", &ch);
} /* Goto_next_graph */

/*******************************************************************/

/* Initialize an integer array from position i to j with value x.
   intarray must be allocated.

   Input parameters:
   1) intarray: the pointer to the integer array
   2) i: start position
   3) j: end position
   4) x: the value to be initialize

   Side effect:
```



```c
     intarray was initialized.
*/

void Init_int_array (int *intarray, int i, int j, int x)

{
  int index;

  for (index = i; index <= j; index++)
    intarray[index] = x;
} /* Init_int_array */

/*******************************************************************/

/* Copy orig_array to des_array from position i to j. */

/* Input parameters:
   1) orig_array: the origin array
   2) des_array: the destination array
   3) i: start position
   4) j: end position
*/

void Copy_int_array (int *orig_array, int *des_array, int i, int j)

{
  int index;

  for (index = i; index <= j; index++)
    des_array[index] = orig_array[index];
} /* Copy_int_array */

/*******************************************************************/

/* Read in the values of the debuglist
   debuglist[0] != 0 means the whole debuglist is true
   otherwise input the debuglist according to the input format
     0 means input each entry of debuglist by 0 (false) or 1 (true)
     1 means input all the positions of the debuglist which required
       to be set to 1 (true) and end the list by -1 */

void Init_debuglist (voidplist)

{
  int i, inputformat, pos;

  fscanf (in, "%d", &(debuglist[0]));
  if (debuglist[0])
    for (i = 1; i <= DEBUGLISTSIZE; i++)
      debuglist[i] = TRUE;
  else
  {
     fscanf (in, "%d", &inputformat);
     if (inputformat == 0)
       for (i = 1; i <= DEBUGLISTSIZE; i++)
         fscanf (in, "%d", &(debuglist[i]));
     else
     {
       fscanf (in, "%d", &pos);
       while (pos != -1)
       {
         debuglist[pos] = TRUE;
         fscanf (in, "%d", &pos);
       }
     }
  }

   fprintf (out, "Here is the debuglist:\n");
   Print_int_array (DEBUGLISTSIZE + 1, debuglist);

} /* Init_debuglist */

/*******************************************************************/

/* Initiate all the timing variables. */
```

```
void Init_time_val (voidplist)

{
  sbuffer = (ptr_to_tms) malloc (sizeof (struct tms));
  fbuffer = (ptr_to_tms) malloc (sizeof (struct tms));
  ebuffer = (ptr_to_tms) malloc (sizeof (struct tms));

  start_time = times(sbuffer);
} /* Init_time_val */

/*********************************************************************/

/* To set up a new next_neigh_node. */

/* Input parameter:
   vertex: the vertex to be worked on (master)

   Output parameter:
   The procedure returns the first non-full next_neigh_node
   in the neighbor list of vertex.

   Side effect:
   The new node created is added to the list pointed by
   neigh_graph[vertex].
*/

ptr_to_next_neigh_node Get_and_init_next_neigh_node (int vertex)

{
   ptr_to_next_neigh_node tempptr;
   int i;

   tempptr = (ptr_to_next_neigh_node)
      malloc(sizeof(struct next_neigh_node));
   for (i = 0; i < NEIGH_NODE_SIZE; i++)
      tempptr->neighbor_name[i] = -1;
   tempptr->next = neigh_graph[vertex];
   neigh_graph[vertex] = tempptr;
   return (tempptr);

} /* Get_and_init_next_neigh_node */

/*********************************************************************/

/* To check if a vertex 'neigh' is in the neighbor list of 'vert'. */

/* Input parameters:
   1) vert: the vertex to be worked on (master)
   2) neigh: the neighbor vertex of vert

   Output parameters:
   The procedure returns 0 if neigh is in the neighbor list of vert and
                         1 if neigh is not in the neighbor list of vert.

*/

int Not_in_neigh_list (int vert, int neigh)

{
   ptr_to_next_neigh_node tempptr;
   int i, no_match = 1;

   tempptr = neigh_graph[vert];
   while (tempptr != NULL)
   {
      for (i = 0; i < NEIGH_NODE_SIZE; i++)
      {
         if (tempptr->neighbor_name[i] == neigh)
         {
            no_match = 0;
            goto exit_Not_in_neigh_list_1;
         }
         else if (tempptr->neighbor_name[i] == -1)
           {
```



```
              break;
           }
        }
      tempptr = tempptr->next;
   }

   exit_Not_in_neigh_list_1: return (no_match);

} /* Not_in_neigh_list */

/*********************************************************************/

/* Return the first empty position in the neighbor list; if the
   next_neigh_node is full, a new next_neigh_node will be created. */

/* Input parameter:
   1) pos: the vertex which is working on (master)

   Output parameter:
   The procedure return the first empty position in the pos's neighbor list.
*/

int Get_first_empty_position (int pos)

{
   int i, empty_pos;
   ptr_to_next_neigh_node tempptr;

   if ((neigh_graph[pos] == NULL) ||
       (neigh_graph[pos]->neighbor_name[NEIGH_NODE_SIZE-1] != -1))
   {
      Get_and_init_next_neigh_node (pos);
      empty_pos = 0;
   }
   else
   {
      i = 0;
      tempptr = neigh_graph[pos];
      while (tempptr->neighbor_name[i] != -1)
         i++;
      empty_pos = i;
   }
   return (empty_pos);
} /* Get_first_empty_position */

/*********************************************************************/

/* Add a new vertex to the neighbor list of vert. */

/* Input parameters:
   1) vert: the vertex to be worked on (master)
   2) neigh: the neighbor vertex of vert
*/

void Add_vertex_to_neigh_list (int vert, int neigh)
{
  int pos;

  pos = Get_first_empty_position(vert);
  neigh_graph[vert]->neighbor_name[pos] = neigh;
} /* Add_vertex_to_neigh_list */

/*********************************************************************/

/* Initiate neigh_graph, adj_matrix, vertex_degree
   of a non-direct graph given the neighbor
   vertices of each vertex. */

/* Here it make sure that if v is a neighbor of u, u is also a neighbor of
   v. If v is a neighbor of u, it will check the neighbor list of v, if u
   is not there, the procedure will add u in the list. For each vertex u,
   it will add to the neighbor list of u. */

void Init_non_direct_neigh_graph (voidplist)
{
```

```
  int i, j, temp;
  ptr_to_next_neigh_node tempptr;

  for (i = 1; i <= total_vertices; i++)
  {
      j = Get_first_empty_position(i);
      /* initiate neigh_graph */
      /* N.B. neighbor is not in the order as input */
      tempptr = neigh_graph[i];
      /* i is a neighbor of i, this should be the first vertex
         added to the list. */
      adj_matrix[i][i] = ((char) 1);
      tempptr->neighbor_name[j] = i;
      vertex_degree[i]++;

      fscanf (in, "%d", &temp);
      while ( temp != -1 )
      {
        if (!input_vertex_start_from_1)
          temp = temp + 1;
        if (adj_matrix[i][temp] != ((char) 1))
        {
           adj_matrix[i][temp] = (char) 1;
           j++;
           if (NEIGH_NODE_SIZE == j)
           {
              j = 0;
              tempptr = Get_and_init_next_neigh_node(i);
           }
           tempptr->neighbor_name[j] = temp;
           vertex_degree[i]++;

           /* update the neighbor list of temp */
           /* the neighbor list of temp should not contain i, otherwise,
              adj_matrix[i,temp] should be equal to 1 */
           adj_matrix[temp][i] = ((char) 1);
           Add_vertex_to_neigh_list(temp, i);
           vertex_degree[temp]++;
        }
        fscanf (in, "%d", &temp);
      }
  }
} /* Init_non_direct_neigh_graph */

/*********************************************************************/

/* Initiate neigh_graph, vertex_degree, adj_matrix
   given that the input is all the neighbor of each
   vertex, starting from vertex 1 in a lexicographical order;
   for non-direct graph, only neighbors with order greater than itself
   can be input, otherwise, the degree of each vertex will be
   calculated wrong; after each
   vertex is finished, a -1 is given; unsigned_size should
   be initiated before calling this procedure. */

/* To include the information of "given x is in the content set, x covers
   by itself", the edge (x,x) is added to all the vertices no matter the
   input graph has this edge or not. In addition, only neighbors with
   label greater than itself will be considered, vertices with smaller
   label will be ignored. */

void Init_neigh_graph_given_neigh_vertex (voidplist)

{
   if (!direct)
     Init_non_direct_neigh_graph();
} /* Init_neigh_graph_given_neigh_vertex */

/*********************************************************************/

/* Used by Init_adj_graph_given_adj_matrix to update each bit of the
   adj_matrix and the neigh_graph.

   Input parameters
   row: the row entry of the adj_matrix,
```



```
   col: the column entry of the adj_matrix,
*/

void Update_adj_matrix_neig_graph_vertex_degree (int row, int col)

{
    ptr_to_next_neigh_node tempptr;

    adj_matrix[row][col] = 1;
    /* initiate neigh_graph */
    Add_vertex_to_neigh_list(row, col);
    vertex_degree[row]++;

} /* Update_adj_matrix_neig_graph_vertex_degree */

/*********************************************************************/

/* Initate adj_matrix, neigh_graph, vertex_degree
   given that the input is in an adjacent matrix form;
   the whole adjacent matrix should be input; unsigned_size should
   be initiated before calling this procedure */

/*
   To include the information of "given x is in the content set, x covers
   by itself", the edge (x,x) is add to all the vertices no matter the
   input graph has this edge or not.
*/

/* N.B. In this procedure, it assume the input adjacency matrix is
        symmetric, i.e. the matrix is correct. */


void Init_neigh_graph_given_adj_matrix (voidplist)

{
  int i, j, temp;
  ptr_to_next_neigh_node tempptr;

  /* for each vertices */
  for (i = 1; i <= total_vertices; i++)
  {
     for (j = 1; j < i; j++)
     {
        fscanf (in, "%d", &temp);
        if (temp == 1)
          Update_adj_matrix_neig_graph_vertex_degree (i,j);
     }
     fscanf (in, "%d", &temp);
     /* add the self-loop to each vertices */
     Update_adj_matrix_neig_graph_vertex_degree (i,i);
     for (j = i+1; j <= total_vertices; j++)
     {
        fscanf (in, "%d", &temp);
        if (temp == 1)
          Update_adj_matrix_neig_graph_vertex_degree (i,j);
     }
  }
} /* Init_neigh_graph_given_adj_matrix */

/*********************************************************************/

/* Initialize allcells and cell_label. */

void Init_allcells_and_cell_label (voidplist)

{
  int i;

  /* REM: max_level - 1 = total_vertices */
  allcells = (int *) calloc (max_level,
             sizeof (int));
  allcells[0] = 0;
  for (i = 1; i < total_vertices; i++)
    allcells[i] = i + 1;
  allcells[total_vertices] = 1;
```

```c
  cell_label = (int *) calloc (max_level,
               sizeof (int));
  Init_int_array (cell_label, 0, max_level-1, 1);
} /* Init_allcells_and_cell_label */

/*********************************************************************/

/* Get the minimum values in an integer array. */

/* Input parameter:
   1) int_arr: the integer array
   2) startpos: the start position
   3) size: the number of entries need to be investigated.
*/

int Get_min_value(int *int_arr, int startpos, int size)

{
   int i, minimum;

   minimum = int_arr[startpos];
   for (i = startpos + 1; i < startpos + size; i++)
     if (minimum > int_arr[i])
       minimum = int_arr[i];
   return minimum;
} /* Get_min_value */

/*********************************************************************/

/* Initiate the first cell. */

/* debug list value = 9 */

void Init_first_cell (voidplist)

{
  int i;
  ptr_to_cell_node fcell;

  fcell = Get_initialized_cell_node();

  /* Create the dummy neighbor nodes. */
  fcell->row_first_neigh = Get_initialized_neigh_info_node ();
  fcell->col_first_neigh = Get_initialized_neigh_info_node ();
  fcell->row_first_neigh->row_cell_num = 1;
  fcell->row_first_neigh->col_cell_num = 0;
  fcell->col_first_neigh->row_cell_num = 0;
  fcell->col_first_neigh->col_cell_num = 1;

  dummy_cell->next_cell->prev_cell = fcell;
  fcell->next_cell = dummy_cell->next_cell;
  dummy_cell->next_cell = fcell;
  fcell->prev_cell = dummy_cell;
  /* one more cells added to the dummy_cell list */
  dummy_cell->total_children++;

  fcell->representative = 1;
  fcell->ver_before_rep = total_vertices;
  fcell->cell_size = total_vertices;
  fcell->lower_content = 0;
  fcell->upper_content = total_vertices;

  cell_list[1] = fcell;

  /* actual_content of the cell is updated later. */
  if (debuglist[9])
  {
     fprintf (out, "\nHere is the first cell:\n");
     Print_cell (fcell);
  }
} /* Init_first_cell */

/*********************************************************************/
```



```c
/* After all_tot_ver_deg_r_c[i][j] was updated, this procedure will
   update all_tot_ver_deg_r_c[j][i].
   k to end_val should be indexed to the same cell.

   Input parameters
   1) k: the first vertex
   2) i: the column cell
   3) j: the row cell
   4) tarr: the array that contains information needed to update
      all_tot_ver_deg_r_c

   Side effect:
   tarr will be updated
*/

void Process_all_tot_ver_deg_j_i (int k, int i, int j,
                                  int *tarr)
{
   int v;

   v = k;
   do
   {
      /* tarr[v] == 0 will also considered. Hence if
         all_tot_ver_deg_r_c[j][i][0] == size of cell j, this
         implies j is not a neighbor of i. */

      all_tot_ver_deg_r_c[j][i][tarr[v]]++;
      v = allcells[v];

   } while (k != v);
} /* Process_all_tot_ver_deg_j_i */

/*********************************************************************/

/*
   This procedure generates all the corresponding
   tot_ver_of_deg_from_row_to_col that need to be created
   when creating the neigh_inf_node of the new level.

   connected_to_cell is used to work out the out going degree of a
   vertex in a newly partitioned cell to another cell.

   tarray is to work out the out going degree of a vertex in cell to
   a newly partitioned cell.

   Here we assumed the parent was stacked but all the neighbor's
   information in the new level is not created.

   N.B. The first_part_cell is also the parent_cell at level 1.

   Input parameters
   1) parent_cell: the parent of the newly partitioned cells
   2) first_part_cell: the first partitioned cell
*/

void Create_influ_func_info (ptr_to_cell_node parent_cell,
                             ptr_to_cell_node first_part_cell)

{
   static int *connected_to_cell = NULL;
   static int *tarray = NULL;

   int i, j, k, v, z, tneigh_ver, cell_rep, parent_cell_num,
       end_val;
   ptr_to_cell_node tcell, newcell;
   ptr_to_neigh_info_node tneigh_info, dummy_col_neigh;
   ptr_to_next_neigh_node tneigh_set;

   dummy_col_neigh = parent_cell->col_first_neigh;
   parent_cell_num = parent_cell->representative;
   if ((connected_to_cell == NULL) || reset_connected_to_cell)
   {
      if (connected_to_cell != NULL)
      {
```

```
        reset_connected_to_cell = FALSE;
        free((void *) connected_to_cell);
        connected_to_cell = NULL;
        free((void *) tarray);
        tarray = NULL;
     }
     connected_to_cell = (int *) calloc (max_level, sizeof(int));
     tarray = (int *) calloc (max_level, sizeof(int));
     connected_to_cell[0] = tarray [0] = 0;
  }

  /* All entry of connected_to_cell and tarray are
     initialized to zero. */
  for (i = 1; i < max_level; i++)
    connected_to_cell[i] = tarray[i] = 0;

  /* The complexity of setting all all_tot_ver_deg_r_c to zero is
     2 * |new cells| * |# of neighbor cells of parent|.
     Since when we create the neigh_info_node, the corresponding
     entries in the all_tot_ver_deg_r_c should be reset to zero,
     here we assume all the entries in the array all_tot_ver_deg_r_c
     is equal to zero. */

  /* Go through each cell Ci */
  tcell = first_part_cell;
  while (tcell != NULL)
  {
     /* for each u in Ci */
     i = cell_rep = tcell->representative;
     do
     {
     /* the above two 'for' loop give size of the parent */
        /* for each tneigh_ver: an neighbor of u */
        tneigh_set = neigh_graph[i];
        while (tneigh_set != NULL)
        {
           for (j = 0; ((j < NEIGH_NODE_SIZE) &&
                       ((tneigh_ver = tneigh_set->neighbor_name[j])
                         != -1));
                j++)
           {
              connected_to_cell[cell_label[tneigh_ver]]++;
              tarray[tneigh_ver]++;
           }
           tneigh_set = tneigh_set->next;
        }
        /* update all_tot_ver_deg_r_c for the vertex v */
        /* instead of checking all cells in the new level,
           the 'for' loop below can be replaced by:
           for each cell D in neighbor of parent (except parent)
           union all newly partitioned cells */

        /* N.B. the first neigh_info_node is a dummy cell */
        tneigh_info = dummy_col_neigh->col_next_neigh;

        while (tneigh_info != dummy_col_neigh)
        /* neighbor of parent except parent itself */
        {
           j = tneigh_info->row_cell_num;
           /* exclude the parent */
           if (j != parent_cell_num)
           {
              all_tot_ver_deg_r_c[cell_rep][j][connected_to_cell[j]]++;
              connected_to_cell[j] = 0;
           }
           tneigh_info = tneigh_info->col_next_neigh;
        }

        /* all newly created cells */
        newcell = first_part_cell;
        while (newcell != NULL)
        {
           j = newcell->representative;
           all_tot_ver_deg_r_c[cell_rep][j][connected_to_cell[j]]++;
           connected_to_cell[j] = 0;
```



```
           newcell = newcell->sibling;
        }
        i = allcells[i];
     } while (i != cell_rep);

     /* At this point we have the information to construct
        all_tot_ver_deg_r_c[cell_rep][x] and
        all_tot_ver_deg_r_c[x][cell_rep]
        for all x and a fix i. */
     /* Update all_tot_ver_deg_r_c[z][cell_rep] because we have
        processed all of Ci. */
     /* The nested 'for' loop below is:
        for each cell D in (neighbors of Ci, but not Ci) do
          for each v in D do.
        N.B. Newly partitioned cells will not be done below,
             otherwise, all_tot_ver_deg_r_c[i][j] where i and
             j are newly partitioned cells will have all its
             entries doubled.
     */
     tneigh_info = dummy_col_neigh->col_next_neigh;

     while (tneigh_info != dummy_col_neigh)
     {
        j = tneigh_info->row_cell_num;

        /* exclude the parent */
        if (j != parent_cell_num)
        {
           k = cell_list[j]->representative;
           Process_all_tot_ver_deg_j_i(k, cell_rep, j, tarray);
        }
        tneigh_info = tneigh_info->col_next_neigh;
     }

     /* The whole tarray has to be re-initialized because
        we have skipped the newly partitioned cells where the
        tarray has also considered those vertices. Hence, it cannot
        be reset to zero in Process_all_tot_ver_deg_j_i. */

     Init_int_array (tarray, 0, max_level-1, 0);

     tcell = tcell->sibling;
  }

} /* Create_influ_func_info */

/*********************************************************************/

/* To work out the   particular size of an
   Upper Influence Function to a cell master.
   Here we assume size is less than or equal to the size of the
   neighbor cell. */

/* Input parameters:
   1) tot_deg_array: the corresponding tot_ver_of_deg_from_row_to_col
   2) content_size: the content of the neighbor cell
*/

int Get_UIF (int *tot_deg_array, int content_size)

{
   int i, tuif = 0;

   if (CONSIS_CHECK)
     Consistence_check();
   for (i = max_degree; i > 0; i--)
   {
      if (tot_deg_array[i] != 0)
        if (tot_deg_array[i] >= content_size)
        {
           tuif += (content_size * i);
           content_size = 0;
           break;
        }
      else
```

```
        {
           tuif += (tot_deg_array[i] * i);
           content_size -= tot_deg_array[i];
        }
    }

    if (tot_deg_array[0] < content_size)
    {
       fprintf (out, "\nProblem! Size of the neighbor cell is less than its ");
       fprintf (out, "content (in Get_UIF), program halt.\n");
       fprintf (out, "total vertices that has zero degree to col cell = ");
       fprintf (out, "%d, content left = %d\n",
                     tot_deg_array[0], content_size);
       fprintf (out, "The rest of the vertices that have edges ");
       fprintf (out, "to col cell:\n");

       for (i = max_degree; i >= 0; i--)
       {
         if (tot_deg_array[i] != 0)
         {
           fprintf (out, "%d: %d ", i, tot_deg_array[i]);
         }
       }
       Myexit(1);
    }

    return (tuif);

} /* Get_UIF */

/*********************************************************************/

/* To work out the a particular size of an
   Lower Influence Function to a cell master.
   Here we assume size is less than or equal to the size of the
   neighbor cell. */

/* Input parameters:
   1) tot_deg_array: the corresponding tot_ver_of_deg_from_row_to_col
   2) size: the content of the neighbor cell
*/

int Get_LIF (int *tot_deg_array, int content_size)

{
    int i, tlif = 0;

    for (i = 0; i <= max_degree; i++)
    {
       if (tot_deg_array[i] != 0)
         if (tot_deg_array[i] >= content_size)
         {
           tlif += (content_size * i);
           content_size = 0;
           break;
         }
         else
         {
           tlif += (tot_deg_array[i] * i);
           content_size -= tot_deg_array[i];
         }
    }

    if (content_size > 0)
    {
       fprintf (out, "\nProblem! Size of the neighbor cell is less than its ");
       fprintf (out, "content (in Get_LIF), program halt.\n");
       Myexit(1);
    }

    return (tlif);

} /* Get_LIF */

/*********************************************************************/
```



```
/* Create_first_level will create the first level.
   Variable automated_cell_partition need to be initialized
   before calling this procedure. */
/* size effect: automated_content_partition, and start_level
                are updated here,
                finish_computation may be changed here,
                global_level is set to 1.*/

/* debug list value = 11 */

void Create_first_level (voidplist)

{
  int i, tUIF;
  ptr_to_neigh_info_node tneigh_info, row_dummy, col_dummy;

  global_level = 1;
  tneigh_info = Get_initialized_neigh_info_node();
  /* N.B.  entry tot_ver_of_deg_from_row_to_col is always
           initialized and allocated. */
  tneigh_info->row_cell_num = 1;
  tneigh_info->col_cell_num = 1;
  tneigh_info->tot_ver_of_deg_from_row_to_col =
     Get_initialized_tvdrc();

  row_dummy = cell_list[1]->row_first_neigh;
  row_dummy->row_next_neigh->row_prev_neigh = tneigh_info;
  tneigh_info->row_next_neigh = row_dummy->row_next_neigh;
  row_dummy->row_next_neigh = tneigh_info;
  tneigh_info->row_prev_neigh = row_dummy;

  col_dummy = cell_list[1]->col_first_neigh;
  col_dummy->col_next_neigh->col_prev_neigh = tneigh_info;
  tneigh_info->col_next_neigh = col_dummy->col_next_neigh;
  col_dummy->col_next_neigh = tneigh_info;
  tneigh_info->col_prev_neigh = col_dummy;

  Create_influ_func_info (cell_list[1], cell_list[1]);
  Copy_int_array (all_tot_ver_deg_r_c[1][1],
                  tneigh_info->tot_ver_of_deg_from_row_to_col,
                  0, max_degree);
  Init_int_array (all_tot_ver_deg_r_c[1][1], 0, max_degree, 0);

  /*
     The maximum allowable wastage of the mastercell was
     not calculated here, it is delayed until the total_content was
     calculated
  */

  /*  Work out the maximum allowable wastage of the mastercell */
  if (automated_cell_partition)
  {
     automated_content_partition = TRUE;
     start_level = 1;

     /* get total number of contents */
     fscanf (in, "%d", &total_content);
     org_total_content = total_content;
     if (total_content > total_vertices)
     {
        fprintf (out, "\nnumber of contents are greater than the ");
        fprintf (out, "total number of vertices, program halt.\n");
        Myexit(1);
     }
     cell_list[1]->lower_content =
     cell_list[1]->upper_content =
     cell_list[1]->actual_content = total_content;
     fprintf (out, "\nThe total number of contents in this graph are %d.\n",
                   total_content);

     tUIF = Get_UIF
            (col_dummy->col_next_neigh->tot_ver_of_deg_from_row_to_col,
             cell_list[1]->actual_content);
     if ((cell_list[1]->max_allow_wastage =
```

```
      tUIF - cell_list[1]->cell_size) < 0)
    {
      fprintf (out, "\nProgram halt due to maximum allowable ");
      fprintf (out, "wastage of the first cell < 0.\n");
      finish_computation = TRUE;
    }
  }

  /* Print out all the result generated */
  /* Print_curr_status(); */

} /* Create_first_level */

/*****************************************************************/

/* Partition_each_vertex_to_one_cell_with_content_x will partition each
   vertex form a cell into a subcell with content equal to x
   upon entry, cell_ptr should be allocated.

   Input variables
   parent_ptr: a pointer to the cell to be partitioned
   x: the content of each new subcell

*/


void Partition_each_vertex_to_one_cell_with_content_x
     (ptr_to_cell_node parent_ptr, int x)

{

  int ver_num, first_ver, curr_ver;
  ptr_to_cell_node cell_ptr, last_sible;

  parent_ptr->first_child = Get_initialized_cell_node();
  cell_ptr = parent_ptr->first_child;
  parent_ptr->prev_cell->next_cell = cell_ptr;
  cell_ptr->prev_cell = parent_ptr->prev_cell;

  first_ver = ver_num = parent_ptr->representative;
  do
  {
     parent_ptr->total_children++; /* one more children */
     curr_ver = ver_num;
     cell_ptr->row_first_neigh = Get_initialized_neigh_info_node();
     cell_ptr->row_first_neigh->row_cell_num = curr_ver;
     cell_ptr->row_first_neigh->col_cell_num = 0;
     cell_ptr->col_first_neigh = Get_initialized_neigh_info_node();
     cell_ptr->col_first_neigh->row_cell_num = 0;
     cell_ptr->col_first_neigh->col_cell_num = curr_ver;

     cell_ptr->parent = parent_ptr;
     cell_ptr->sibling = Get_initialized_cell_node();
     cell_ptr->next_cell = cell_ptr->sibling;
     cell_ptr->sibling->prev_cell = cell_ptr;

     cell_ptr->representative = cell_ptr->ver_before_rep = curr_ver;
     /* cell_ptr->first_vertex_location = i; */
     cell_ptr->cell_size = 1;

     /* N.B. Later the upper content should also depend on the
             actual content. */
     cell_ptr->lower_content = cell_ptr->upper_content =
       cell_ptr->actual_content = x;

     cell_label[curr_ver] = curr_ver;
     cell_list[curr_ver] = cell_ptr;
     last_sible = cell_ptr;
     cell_ptr = cell_ptr->sibling;

     /* the following order is important */
     ver_num = allcells[curr_ver];
     allcells[curr_ver] = curr_ver;

  } while (first_ver != ver_num);
```



```
  last_sible->next_cell = parent_ptr->next_cell;
  parent_ptr->next_cell->prev_cell = last_sible;
  dummy_cell->total_children += (parent_ptr->total_children - 1);

  Free_cell_node(last_sible->sibling);
  last_sible->sibling = NULL;

} /* Partition_each_vertex_to_one_cell_with_content_x */

/*****************************************************************/

/* Partition_cell will partition a cell into subcells, all the
   entries in each subcell will also be initialized;
   N.B. here it assume the cell can be refined, i.e. there must
        have at least 2 vertices in it. */

/* debug list value = 3 */

void Partition_cell (ptr_to_cell_node parent_ptr)

{
  int i, j, normal_csize, csize, crep, vb4rep,
      first_pos, curr_pos, next_pos, last_pos;
  ptr_to_cell_node cell_ptr, last_sible;

  if (parent_ptr->actual_content > 0)
    if (parent_ptr->actual_content != parent_ptr->cell_size)
    {
       parent_ptr->first_child = Get_initialized_cell_node();
       cell_ptr = parent_ptr->first_child;
       parent_ptr->prev_cell->next_cell = cell_ptr;
       cell_ptr->prev_cell = parent_ptr->prev_cell;

       /* The position where the first vertex start partition from
          its parent. */
       next_pos = parent_ptr->representative;
       normal_csize = csize = parent_ptr->cell_size/DIVISION_FACTOR;

       for (i = 1; i <= DIVISION_FACTOR; i++)
       {
          if (i == DIVISION_FACTOR) /* last entry */
            csize = parent_ptr->cell_size -
                    (DIVISION_FACTOR - 1) * normal_csize;

          parent_ptr->total_children++; /* one more children */
          cell_ptr->parent = parent_ptr;
          cell_ptr->sibling = Get_initialized_cell_node();
          cell_ptr->next_cell = cell_ptr->sibling;
          cell_ptr->sibling->prev_cell = cell_ptr;

          /* update allcells, next_pos, ver_before_rep
             and representative. */
          vb4rep = 0;
          crep = first_pos = next_pos;
          for (j = 1; j < csize; j++)
          {
             last_pos = next_pos;
             next_pos = allcells[next_pos];
             if (crep > next_pos)
             {
                vb4rep = last_pos;
                crep = next_pos;
             }
          }
          curr_pos = next_pos;
          next_pos = allcells[next_pos];
          allcells[curr_pos] = first_pos;
          cell_ptr->representative = crep;
          if (vb4rep) /* not the first vertex (!= 0)*/
            cell_ptr->ver_before_rep = vb4rep;
          else
            cell_ptr->ver_before_rep = curr_pos;

          /* create the dummies */
```

```
         cell_ptr->row_first_neigh = Get_initialized_neigh_info_node();
         cell_ptr->row_first_neigh->row_cell_num = crep;
         cell_ptr->row_first_neigh->col_cell_num = 0;
         cell_ptr->col_first_neigh = Get_initialized_neigh_info_node();
         cell_ptr->col_first_neigh->row_cell_num = 0;
         cell_ptr->col_first_neigh->col_cell_num = crep;

         cell_ptr->cell_size = csize;
         cell_ptr->lower_content = 0;
         /* N.B. Later the upper content should also depend on the
                 actual content; upper content should be updated in
                 here because it is also depend on the total_vertices. */
         if (csize <= parent_ptr->actual_content)
           cell_ptr->upper_content = csize;
         else
           cell_ptr->upper_content = parent_ptr->actual_content;

         curr_pos = first_pos;
         /* update cell_label */
         do
         {
            cell_label[curr_pos] = crep;
            curr_pos = allcells[curr_pos];
         } while (curr_pos != first_pos);

         cell_list[crep] = cell_ptr;

         last_sible = cell_ptr;
         cell_ptr = cell_ptr->sibling;
      }
      last_sible->next_cell = parent_ptr->next_cell;
      parent_ptr->next_cell->prev_cell = last_sible;
      dummy_cell->total_children += (parent_ptr->total_children - 1);

     /* free the last extra cell node created */
     /* the free will not work if no partition occur,
        then cell_ptr == last_sible */
     Free_cell_node (last_sible->sibling);
     last_sible->sibling = NULL;
   }
   else
     /* partition to n cells with each contain 1 vertex
        and with content one */
     Partition_each_vertex_to_one_cell_with_content_x
        (parent_ptr, 1);
  else
    /* partition to n cells with each contain 1 vertex
       and with content zero */
    Partition_each_vertex_to_one_cell_with_content_x
       (parent_ptr, 0);
  /* Push_cell_stack (parent_ptr); */

} /* Partition_cell */

/**********************************************************************/

/* To built and connect a specific neigh_info_node. */

/* Input parameter:
   1) row_cell: the row cell number of the neigh_info_node
   2) col_cell: the column cell number of the neigh_info_node

   Side effect:
   The neigh_info_node was built,
   corresponding entries in all_tot_ver_deg_r_c was cleaned.
*/

void Built_neigh_info_node (int row_cell, int col_cell)
{
   ptr_to_neigh_info_node tneigh_info, trdummy, tcdummy;
   int i;

   tneigh_info = Get_initialized_neigh_info_node();
   tneigh_info->row_cell_num = row_cell;
   tneigh_info->col_cell_num = col_cell;
```



```
   tneigh_info->tot_ver_of_deg_from_row_to_col =
      Get_initialized_tvdrc ();
   for (i = 0; i <= max_degree; i++)
   {
      tneigh_info->tot_ver_of_deg_from_row_to_col[i] =
          all_tot_ver_deg_r_c[row_cell][col_cell][i];
      all_tot_ver_deg_r_c[row_cell][col_cell][i] = 0;
   }
   trdummy = cell_list[row_cell]->row_first_neigh;
   trdummy->row_next_neigh->row_prev_neigh = tneigh_info;
   tneigh_info->row_next_neigh = trdummy->row_next_neigh;
   trdummy->row_next_neigh = tneigh_info;
   tneigh_info->row_prev_neigh = trdummy;

   tcdummy = cell_list[col_cell]->col_first_neigh;
   tcdummy->col_next_neigh->col_prev_neigh = tneigh_info;
   tneigh_info->col_next_neigh = tcdummy->col_next_neigh;
   tcdummy->col_next_neigh = tneigh_info;
   tneigh_info->col_prev_neigh = tcdummy;

} /* Built_neigh_info_node */

/**********************************************************************/

/* This procedure create and stack all the neighbor information of
   each newly created cell. */

/* Input parameter:
   parent_cell: pointer to the parent node
*/

void Handle_new_level (ptr_to_cell_node parent_cell)

{
   ptr_to_neigh_info_node pneigh_info, tneigh_info, pdummy;
   ptr_to_cell_node ncell, tcell;
   int parent_cell_num, working_cell_num, neigh_cell_num;

   /* stack the row neighbor list */
   pdummy = parent_cell->row_first_neigh;
   pneigh_info = pdummy->row_next_neigh;
   parent_cell_num = parent_cell->representative;
   if (pneigh_info == pdummy)
   {
      fprintf (out, "There is no row neighbor information to be ");
      fprintf (out, "stacked.\n");
       Myexit(1);
   }
   else
   {
      while (pneigh_info != pdummy)
      {
         if (parent_cell_num != pneigh_info->col_cell_num)
         {
            pneigh_info->col_prev_neigh->col_next_neigh
               = pneigh_info->col_next_neigh;
            pneigh_info->col_next_neigh->col_prev_neigh
               = pneigh_info->col_prev_neigh;
         }
         pneigh_info = pneigh_info->row_next_neigh;
      }
   }

   /* stack the col neighbor list */
   pdummy = parent_cell->col_first_neigh;
   pneigh_info = pdummy->col_next_neigh;
   if (pneigh_info == pdummy)
   {
      fprintf (out, "There is no column neighbor information to be ");
      fprintf (out, "stacked.\n");
       Myexit(1);
   }
   else
   {
      /* Push_neigh_info_stack (pdummy); */
```

```
      while (pneigh_info != pdummy)
      {
         if (pneigh_info->row_cell_num != parent_cell_num)
         {
            pneigh_info->row_prev_neigh->row_next_neigh
               = pneigh_info->row_next_neigh;
            pneigh_info->row_next_neigh->row_prev_neigh
               = pneigh_info->row_prev_neigh;
         }
         pneigh_info = pneigh_info->col_next_neigh;
      }
   }

   /* reset pneigh_info and start to generate the neighbor information
      nodes for all the newly generated cells */

   /* work on the row neighbor information node */
   /* for each newly partitioned cell */
   ncell = parent_cell->first_child;
   while (ncell != NULL)
   {
      working_cell_num = ncell->representative;
      pneigh_info = pdummy->col_next_neigh;
      while (pneigh_info != pdummy)
      {
         neigh_cell_num = pneigh_info->row_cell_num;
         if (parent_cell_num != neigh_cell_num)
         {
            /* Here we assume if A is a neighbor of B then B is also a
               neighbor of A, and if A is not a neighbor of B, B is not
               a neighbor of A either. */
            if (all_tot_ver_deg_r_c[working_cell_num]
                       [neigh_cell_num][0] != ncell->cell_size)
            /* they are neighbor, built the neighbor info. node */
            {
               Built_neigh_info_node(working_cell_num,
                                     neigh_cell_num);
               Built_neigh_info_node(neigh_cell_num,
                                     working_cell_num);
            }
            else
            {
               all_tot_ver_deg_r_c[working_cell_num]
                                 [neigh_cell_num][0] = 0;
               all_tot_ver_deg_r_c[neigh_cell_num]
                                 [working_cell_num][0] = 0;
            }

         }
         pneigh_info = pneigh_info->col_next_neigh;
      }
      tcell = parent_cell->first_child;
      while (tcell != NULL)
      {
         neigh_cell_num = tcell->representative;
         if (all_tot_ver_deg_r_c[working_cell_num]
                  [neigh_cell_num][0] != ncell->cell_size)
         /* they are neighbor, built the neighbor info. node */
         /* N.B. We just built one because
                Built_neigh_info_node(neigh_cell_num, working_cell_num)
                will be covered when we work on cell neigh_cell_num.
         */
           Built_neigh_info_node(working_cell_num,
                                 neigh_cell_num);
         else
           all_tot_ver_deg_r_c[working_cell_num]
                           [neigh_cell_num][0] = 0;

         tcell = tcell->sibling;
      }
      ncell = ncell->sibling;
   }

} /* Handle_new_level */
```



```
/***********************************************************************/

/* Individual Coverage test tries to improve the lower content of a
   particular cell. */

/* Input parameter
   1) tcell: the cell to be tested

   Output parameter
     return False if the Coverage test failed
*/

/* debug list value = 4 */

int Individual_coverage (ptr_to_cell_node tcell)

{
   static int *s_slack = NULL;

   ptr_to_neigh_info_node tneigh_info, dummy_col_neigh;
   ptr_to_cell_node neigh_cell;
   int i, skj, max_coverage = 0, lower_content_updated,
       tuif, neigh_cell_num;

   if ((s_slack == NULL) || reset_s_slack)
   {
      if (s_slack != NULL)
      {
         reset_s_slack = FALSE;
         free ((void *) s_slack);
         s_slack = NULL;
      }
      s_slack = (int *) calloc (max_level, sizeof(int));
   }

   dummy_col_neigh = tcell->col_first_neigh;

   /* --
      \       -
      /   Uij ( o (Ci) ) - size (Cj)
      --
      Ci neighbor
      of Cj */

   i = 0;
   tneigh_info = dummy_col_neigh->col_next_neigh;

   while (tneigh_info != dummy_col_neigh)
   /* neighbor of tcell */
   {
      neigh_cell_num = tneigh_info->row_cell_num;
      neigh_cell = cell_list[neigh_cell_num];
      tuif = Get_UIF
             (tneigh_info->tot_ver_of_deg_from_row_to_col,
              neigh_cell->upper_content);
      max_coverage += tuif;
      s_slack[i] = tuif;

      i++;
      tneigh_info = tneigh_info->col_next_neigh;
   }
   max_coverage -= tcell->cell_size;
   if (debuglist[4])
   {
     fprintf (out, "max_coverage of cell %x = %d\n",
              tcell, max_coverage);
   }
   if (max_coverage < 0)
   /* no way the cell can be covered */
     return (FALSE);
   tneigh_info = dummy_col_neigh->col_next_neigh;

   i = 0;
   while (tneigh_info != dummy_col_neigh)
   /* neighbor of tcell */
```

```
  {
    neigh_cell_num = tneigh_info->row_cell_num;
    if (debuglist[4])
    {
      fprintf (out, "neighbor cell is %d.\n", neigh_cell_num);
    }
    neigh_cell = cell_list[neigh_cell_num];
    lower_content_updated = TRUE;
    do {
      skj = s_slack[i] -
            Get_UIF
            (tneigh_info->tot_ver_of_deg_from_row_to_col,
             neigh_cell->lower_content);
      if (debuglist[4])
      {
        fprintf (out, "skj = %d\n", skj);
      }
      if (max_coverage < skj)
      {
         if (neigh_cell->lower_content >= neigh_cell->upper_content)
         {
            /* The restore of lower_content in below is enough;
               because if this cell is not a newly partitioned
               cell, (and could only be the first one)
               then its content will be predefined, so its
               lower content will be equal to its upper content,
               hence decrease one in its lower content will
               bring it back to its original value, otherwise,
               if this is a newly partitioned cell, it is not
               necessary to restore its value, as this cell will
               be removed, right before the second set of
               contents is generated. N.B this kind of restoring
               method is only good if a set of new cells are
               generated, all their contents will be defined
               before another sets of new cells are generated.
            */
            if (debuglist[4])
            {
               fprintf (out, "\nIndividual coverage failed, the ");
               fprintf (out, "neighbor cell is:\n");
               Print_cell (neigh_cell);
               fprintf (out, "max_coverage = %d, skj = %d.\n",
                        max_coverage, skj);
            }
            return (FALSE);
         }
         else
         {
            neigh_cell->lower_content++;
            if (debuglist[4])
            {
              fprintf (out, "\nNew lower_content is %d\n",
                       neigh_cell->lower_content);
            }
         }
      }
      else
        lower_content_updated = FALSE;
    } while (lower_content_updated);
    tneigh_info = tneigh_info->col_next_neigh;
    i++;
  }

  return (TRUE);

} /* Individual_coverage */

/***************************************************************************/

/* Coverage test will try to improve the lower content of all
   the related cells. Here we assume the parent passed in must
   have children. */

/* Input parameter
   1) parent_cell: the parent of the latest partitioned cells
```



```
   Output parameter
   return False if the Coverage test failed
*/

int Coverage_test (ptr_to_cell_node parent_cell)

{
   ptr_to_neigh_info_node pneigh_info, pdummy;
   ptr_to_cell_node tcell;
   int parent_cell_num, neigh_cell_num, test_result;

   parent_cell_num = parent_cell->representative;
   pdummy = parent_cell->col_first_neigh;
   pneigh_info = pdummy->col_next_neigh;

   /* Go through all the related cells. */
   while (pneigh_info != pdummy)
   {
      neigh_cell_num = pneigh_info->row_cell_num;
      if (parent_cell_num != neigh_cell_num)
      {
         if (!(Individual_coverage (cell_list[neigh_cell_num])))
         {
             if (debuglist[4])
             {
               fprintf (out, "Individual coverage of cell %d failed.\n",
                        neigh_cell_num);
             }
             return (FALSE);
         }
      }
      pneigh_info = pneigh_info->col_next_neigh;
   }
   tcell = parent_cell->first_child;
   while (tcell != NULL)
   {
      neigh_cell_num = tcell->representative;
      if (!(Individual_coverage (cell_list[neigh_cell_num])))
      {
         if (debuglist[4])
         {
           fprintf (out, "Individual coverage of cell %d failed.\n",
                    neigh_cell_num);
         }
         return (FALSE);
      }
      tcell = tcell->sibling;
   }
   return (TRUE);
} /* Coverage_test */

/***************************************************************************/

/* working out:

     --
     \
     /   LIFij ( o (Ci) ) - size (Cj)
     --            -
     Ci neighbor
     of Cj

   input parameter:
   1) master_cell: the Cj
*/

int Get_unavoid_wastage (ptr_to_cell_node master_cell)
{
   ptr_to_neigh_info_node tneigh_info, dummy_col_neigh;
   ptr_to_cell_node neigh_cell;
   int tot_value = 0, tlif, neigh_cell_num;

   dummy_col_neigh = master_cell->col_first_neigh;
```

```
   tneigh_info = dummy_col_neigh->col_next_neigh;

   while (tneigh_info != dummy_col_neigh)
   /* neighbor of master_cell */
   {
      neigh_cell_num = tneigh_info->row_cell_num;
      neigh_cell = cell_list[neigh_cell_num];
      tlif = Get_LIF
             (tneigh_info->tot_ver_of_deg_from_row_to_col,
              neigh_cell->lower_content);
      tot_value += tlif;
      tneigh_info = tneigh_info->col_next_neigh;
   }
   tot_value -= master_cell->cell_size;

   return (tot_value);

} /* Get_unavoid_wastage */

/*******************************************************************/

/* basic idea:

     for each Cj which is a neighbor of parent (but not the parent)
     or it is one of the latest partitioned cell, it works out

     --
     \
     /   LIFij ( o (Ci) ) - size (Cj)
     --                 -
     Ci neighbor
     of Cj

     in a list, and let us call each entry of the list as the
     unavoidable_wastage[j].

     N.B. LIF mean Lower Influence Function.

     Let W'k( Cj) = maximum between
           -
                        0 and
                        unavoidable_wastage[j] - Zkj

                               -
     where Zkj = LIFkj (o(Ck)) - LIFkj(o(Ck)).

     for each latest partitioned cell Ck
       for each neighbor of parent (including the parent) C
         pass = false;
         while (! pass)
                --
           if   \
                /   W'k( Cj) <  W(C)
                --  -       -
                Cj is a child of C

           then pass = true;

           else  -
                 o (Ck) --;

*/

int Wastage_test (ptr_to_cell_node parent_cell,
                  int *wastage_changes)

{
   static int *unavoidable_wastage = NULL;

   ptr_to_neigh_info_node pneigh_info, wneigh_info, pdummy;
   ptr_to_cell_node tcell, working_pcell, tchild;
   int parent_cell_num, neigh_cell_num, upper_content_updated,
       twastage, zkj, j, k, temp;

   /* The unavoidable_wastage do not need to initialize, each time
```



```
      all the necessary entries will be updated to the correct
      value. */
   if ((unavoidable_wastage == NULL) || reset_unavoidable_wastage)
   {
      if (unavoidable_wastage != NULL)
      {
         reset_unavoidable_wastage = FALSE;
         free ((void *) unavoidable_wastage);
         unavoidable_wastage = NULL;
      }
      unavoidable_wastage = (int *) calloc (max_level, sizeof(int));
   }

   *wastage_changes = FALSE;
   parent_cell_num = parent_cell->representative;
   pdummy = parent_cell->col_first_neigh;
   pneigh_info = pdummy->col_next_neigh;

   /* Go through all the related cells. */
   while (pneigh_info != pdummy)
   {
      neigh_cell_num = pneigh_info->row_cell_num;
      if (parent_cell_num != neigh_cell_num)
      {
         unavoidable_wastage[neigh_cell_num] =
           Get_unavoid_wastage (cell_list[neigh_cell_num]);
      }
      pneigh_info = pneigh_info->col_next_neigh;
   }
   tcell = parent_cell->first_child;
   while (tcell != NULL)
   {
      neigh_cell_num = tcell->representative;
      unavoidable_wastage[neigh_cell_num] =
        Get_unavoid_wastage (cell_list[neigh_cell_num]);
      tcell = tcell->sibling;
   }

   /* for each latest partitioned cell Ck */
   tcell = parent_cell->first_child;
   while (tcell != NULL)
   {
     k = tcell->representative;
     pneigh_info = pdummy->col_next_neigh;
     while (pneigh_info != pdummy)
     {
        neigh_cell_num = pneigh_info->row_cell_num;
        if (parent_cell_num != neigh_cell_num)
          working_pcell = cell_list[neigh_cell_num];
        else
          working_pcell = parent_cell;

        upper_content_updated = TRUE;
        while (upper_content_updated)
        {
           twastage = 0;

           tchild = working_pcell->first_child;
           while (tchild != NULL)
           {
              j = tchild->representative;
              wneigh_info = tchild->col_first_neigh->col_next_neigh;
              /* not a neighbor of k or not equal dummy */
              while ((wneigh_info->row_cell_num != k) &&
                     (wneigh_info->row_cell_num != 0))
                wneigh_info = wneigh_info->col_next_neigh;
              if (wneigh_info->row_cell_num == 0)
              /* k is not a neighbor of j */
                zkj = 0;
              else
                zkj = Get_LIF
                      (wneigh_info->tot_ver_of_deg_from_row_to_col,
                       tcell->upper_content) -
                      Get_LIF
                      (wneigh_info->tot_ver_of_deg_from_row_to_col,
```

```
                tcell->lower_content);
            temp = unavoidable_wastage[j] - zkj;
            if (temp > 0)
              twastage += temp;
            tchild = tchild->sibling;
          }

          /* wastage test */
          if (twastage <= working_pcell->max_allow_wastage)
            upper_content_updated = FALSE;
          else
            if (tcell->lower_content >= tcell->upper_content)
            /* refer to the explanation in coverage test */
            {
               if (debuglist[5])
                 Print_cell (tcell);
               return (FALSE);
            }
            else
            {
               tcell->upper_content--;
               *wastage_changes = TRUE;
               if (debuglist[5])
               {
                 fprintf (out, "\nNew upper_content is %d\n",
                          tcell->upper_content);
               }
            }
        }
        pneigh_info = pneigh_info->col_next_neigh;
      }
      tcell = tcell->sibling;
    }

    return (TRUE);

} /* Wastage_test */

/*********************************************************************/

/* This procedure will create a set of actual content partition
   starting form the first entry. Based on the values in the
   array of content partition and the upper bound of each entry of
   the array of content partition, a content partition is worked
   out. */

/* Input parameters
   1) content_array: the array contains the content to be updated;
   2) upper_bound: the upper bound of each entry of the content array;
   3) arr_size: the size of the array;
   4) remain_content: the content left to be assign to the content partition.

   Output parameter
   1) content_array: the updated content array.
*/

void Next_content_partition (int *content_array, int *upper_bound,
                             int arr_size, int remain_content)

{
  int leftover, i;

  /* set up the minimum actual content */
  for (i = 0; i < arr_size; i++)
  {
     leftover = remain_content - upper_bound[i] + content_array[i];
     if (leftover >= 0)
     {
        content_array[i] = upper_bound[i];
        if (leftover == 0)
          break;
     }
     else /* leftover < 0 */
     {
        content_array[i] += remain_content;
```



```
          break;
       }
       remain_content = leftover;
  }
} /* Next_content_partition */

/*********************************************************************/

/* Update the children contents will set the content, the lower and
   upper bound of the children cell. */

/* Input parameter;
   1) parent_ptr: the parent of the latest partitioned cells.
   2) actual_content_distribution: an array contains the content
      of the latest partitioned cells in the same order as the list
      in parent_ptr
*/

void Update_the_children_contents (ptr_to_cell_node parent_ptr,
                                   int *actual_content_distribution)

{
   ptr_to_cell_node child;
   int position = 0;

   child = parent_ptr->first_child;
   while (child != NULL)
   {
      child->lower_content = child->upper_content =
      child->actual_content = actual_content_distribution[position];
      child = child->sibling;
      position++;
   }
} /* Update_the_children_contents */

/*********************************************************************/

/* Partition the content and go to the next recursion. */
/* It works out the maximum allowable wastage from a particular cell. */

/* Input parameter
   1) ncell: the cell to be work on
*/

/* debuglist = 6 */

int Calculate_max_allowable_wastage (ptr_to_cell_node ncell)
{
   ptr_to_neigh_info_node tdummy, tneigh_info;
   int temp = 0;

   tdummy = ncell->col_first_neigh;
   tneigh_info = tdummy->col_next_neigh;
   while (tneigh_info != tdummy)
   {
      temp += Get_UIF
              (tneigh_info->tot_ver_of_deg_from_row_to_col,
               cell_list[tneigh_info->row_cell_num]->upper_content);
      tneigh_info = tneigh_info->col_next_neigh;
   }
   temp -= ncell->cell_size;
   return (temp);
} /* Calculate_max_allowable_wastage */

/*********************************************************************/

/* Restores the max_allowable_wastage of the parent's neighbors. */

/* Input parameters:
   1) parent_cell: the first neighbor to be restored;
   2) oldvalue: a list of array to store the old values;
   3) total_neigh: the total number of neighbor to be restored.
*/

void Restore_pneigh_maw (ptr_to_cell_node parent_cell,
```

```c
                         int *oldvalue,
                         int total_neigh)
{
   ptr_to_neigh_info_node pdummy, pneigh_info;
   ptr_to_cell_node tcell;
   int parent_cell_num, neigh_cell_num, count = 0;

   parent_cell_num = parent_cell->representative;
   pdummy = parent_cell->col_first_neigh;
   pneigh_info = pdummy->col_next_neigh;

   while (total_neigh > count)
   {
      neigh_cell_num = pneigh_info->row_cell_num;
      if (neigh_cell_num != parent_cell_num)
      {
         tcell = cell_list[neigh_cell_num];
         tcell->max_allow_wastage =
           oldvalue[count];

         count++;
      }
      pneigh_info = pneigh_info->col_next_neigh;
   }

} /* Restore_pneigh_maw */

/*********************************************************************/

/* Update all the maximum allowable wastage in each cell
   that belongs to a particular level. */

/* Only neighbor of parents and the new cells are affected. */

/* debug list value = 6 */

int Update_max_allowable_wastage (ptr_to_cell_node parent_cell)
{
   static int *old_max_allowable_wastage = NULL;

   ptr_to_neigh_info_node pneigh_info, pdummy;
   ptr_to_cell_node ncell, tcell;
   int parent_cell_num, neigh_cell_num, temp, count = 0, last_count;
   /* count keeps track of how many cell's max_allowable_wastage was
      touched so far and last_count keeps track the total number of
      neighbor's max_allowable_wastage were touched. */

   if ((old_max_allowable_wastage == NULL) ||
        reset_old_max_allowable_wastage)
   {
      if (old_max_allowable_wastage != NULL)
      {
         reset_old_max_allowable_wastage = FALSE;
         free ((void *) old_max_allowable_wastage);
         old_max_allowable_wastage = NULL;
      }
      old_max_allowable_wastage = (int *) calloc
        (max_level, sizeof(int));
   }

   /* working on the neighbor cells of the parent */
   parent_cell_num = parent_cell->representative;
   pdummy = parent_cell->col_first_neigh;
   pneigh_info = pdummy->col_next_neigh;
   while (pneigh_info != pdummy)
   {
      neigh_cell_num = pneigh_info->row_cell_num;
      if (neigh_cell_num != parent_cell_num)
      {
         ncell = cell_list[neigh_cell_num];
         temp = Calculate_max_allowable_wastage(ncell);

         if ((temp > 50) && (debuglist [6]))
         {
           fprintf (out, "\nWarning, temp > 50, temp = %d !!!!!!!\n", temp);
```



```c
         }

         if (temp < 0)
         {
           if (debuglist [6])
           {
             fprintf (out, "\nPROBLEM: maximum allowable wastage ");
             fprintf (out, "< 0 exists in parent's neighbor.\n");
             fprintf (out, "The cell with maximum allowable wastage ");
             fprintf (out, "= %d is:\n", temp);
             Print_cell (ncell);
             Print_curr_status ();
           }
           /* restore all the maximum allowable wastage of the
              modified cells */
           Restore_pneigh_maw(parent_cell, old_max_allowable_wastage,
                              count);
           return (FALSE);
         }
         else
         {
            old_max_allowable_wastage[count] = ncell->max_allow_wastage;
            ncell->max_allow_wastage = temp;
            count++;
         }
      }

      pneigh_info = pneigh_info->col_next_neigh;
   }

   /* working on the parent's children */
   last_count = count;
   ncell = parent_cell->first_child;
   while (ncell != NULL)
   {
      temp = Calculate_max_allowable_wastage(ncell);
      if (temp < 0)
      {
         if (debuglist [6])
         {
           fprintf (out, "\nPROBLEM: maximum allowable wastage ");
           fprintf (out, "< 0 exists in parent's children.\n");
           fprintf (out, "The cell with maximum allowable wastage ");
           fprintf (out, "= %d is:\n", temp);
           Print_cell (ncell);
         }
         /* Restore all the maximum allowable wastage of the
            modified cells. */
         /* Restore neighbors of parent. */
         Restore_pneigh_maw(parent_cell, old_max_allowable_wastage,
                            last_count);
         /* restore children of parent */
         tcell = parent_cell->first_child;
         while (last_count < count)
         {
            tcell->max_allow_wastage =
              old_max_allowable_wastage[last_count];
            tcell = tcell->sibling;
            last_count++;
         }
         return (FALSE);
      }
      else
      {
         old_max_allowable_wastage[count] = ncell->max_allow_wastage;
         ncell->max_allow_wastage = temp;
         count++;
      }
      ncell = ncell->sibling;
   }
   return (TRUE);

} /* Update_max_allowable_wastage */

/*********************************************************************/
```

```
/* Partition the content. */

void Partition_content (int status,
                        int current_level,
                        ptr_to_cell_node parent_ptr,
                        ptr_to_perm_gp gptr,
                        ptr_to_perm_gp bptr,
                        ptr_to_perm_gp extgp,
                        ptr_to_permvect mapping,
                        int array_vol,
                        int *actual_content_distribution,
                        ptr_to_cell_node next_part_cell)

{
   extern int canonical_rep;

   ptr_to_perm_gp block_auto = NULL;

   total_content_part[current_level]++;

   switch (status)
   {
     case REGULAR_PARTITION:
             /* the upper and lower content limit will also be stacked
                into the prev upper and lower content limit */
             Update_the_children_contents (parent_ptr,
                                           actual_content_distribution);

             /* It is not necessary to restore of the upper
                and lower content of all the subcells. It is because all
                the reference are on the upper and lower content
                distribution arrays. */
             /* There is no need to restore the children content limits
                they are stored locally in Start_gen. */
             if (Update_max_allowable_wastage (parent_ptr))
               Handle_reg_content_part (current_level, parent_ptr, gptr,
                                        array_vol,
                                        actual_content_distribution);
             break;

     case ORBIT_PARTITION:
             Update_the_children_contents (parent_ptr,
                                           actual_content_distribution);
             bfmawtest[current_level]++;
             if (Update_max_allowable_wastage (parent_ptr))
               Handle_orb_content_part (current_level,
                                        parent_ptr, gptr,
                                        array_vol,
                                        actual_content_distribution,
                                        next_part_cell);
             break;

     case TRANSITIVE_IMPRIMITIVE_PARTITION:
             canonical_rep = TRUE;
             block_auto = Isomorphic_test
                          (bptr, actual_content_distribution,
                           bptr->permsize);
             if (canonical_rep)
             /* if it pass the isomorphic test, i.e. not isomorphic
                to the maximum content partition lexicographically */
             {
                Update_the_children_contents (parent_ptr,
                                              actual_content_distribution);

                if (Update_max_allowable_wastage (parent_ptr))
                  Handle_transitive_content_part (current_level,
                                     parent_ptr, gptr, bptr, extgp,
                                     mapping, array_vol,
                                     actual_content_distribution,
                                     block_auto);
             }
             disposepg (block_auto);
             block_auto = NULL;
             break;
```



```
     default: fprintf(out, "Unknown case occur in Start_gen: %d.\n",
                            status);
              Myexit(1);
   }
} /* Partition_content */

/***********************************************************************/

/* Start_gen initializes all the variables in order to
   generate the content for all the cells. */

/* debug list value = 7 */

/* input parameters:
   1) current_level: the latest level created
   2) parent_ptr: the parent of the cells just partitioned
   3) gptr: the pointer to the automorphism group
   4) bptr: the pointer to the automorphism of the block
   5) extgp: the pointer to the intermediate group in the homomorphism
             it is used in the transitive and imprimitive case
   6) mapping: to map back the elements in the truncated group
               to their old value
               in the original graph
   7) status: use to decide what to do after the content was partitioned
        1: regular partition without making use of symmetry group,
           it call the procedure Handle_reg_content_part
        2: Orbit partition.
        3: Transitive partition, imprimitive.
   8) next_part_cell: the pre-selected next cell to be partition
*/

void Start_gen (int current_level,
                ptr_to_cell_node parent_ptr,
                ptr_to_perm_gp gptr,
                ptr_to_perm_gp bptr,
                ptr_to_perm_gp extgp,
                ptr_to_permvect mapping,
                int status,
                ptr_to_cell_node next_part_cell)

{
  /* recording the actual content distribution among the cells */
  int *actual_content_distribution = NULL;
  /* recording the upper limit distribution among the cells */
  int *upper_limit_distribution = NULL;
  /* recording the lower limit distribution among the cells */
  int *lower_limit_distribution = NULL;

  int  i,j,k,
       count,
       array_vol,
       parent_content,
       unuse_content,
       more_content_partition = TRUE,
       decrease_position, increase_position,
       total_lower_content, total_upper_content;
  ptr_to_cell_node tsibling;

  array_vol = parent_ptr->total_children;

  /* Although the following three variables are allocated and deallocated
     each time in this procedure, we cannot make them static because
     the procedure is recursive, otherwise, the value in these variables
     can be changed.  */

  actual_content_distribution = (int *) calloc (3*(array_vol+1), sizeof(int));
  lower_limit_distribution = &(actual_content_distribution[array_vol+1]);
  upper_limit_distribution = &(actual_content_distribution[2*(array_vol+1)]);

  tsibling = parent_ptr->first_child;
  parent_content = parent_ptr->actual_content;

  i = 0;
  while (tsibling != NULL)
  {
```

```
    /* actual_content_distribution is set to be equal to
       lower_limit_distribution */
    upper_limit_distribution[i] = tsibling->upper_content;
    lower_limit_distribution[i] = actual_content_distribution[i]
                                = tsibling->lower_content;
    tsibling = tsibling->sibling;
    i++;
  }

  total_lower_content = lower_limit_distribution[0];
  total_upper_content = upper_limit_distribution[0];
  for (i = 1; i < array_vol; i++)
  {
    total_lower_content += lower_limit_distribution[i];
    total_upper_content += upper_limit_distribution[i];
  }

  /* If the total upper content < given_content or
     total lower content > given content, then there is no valid
     set of content partition, hence any further generation is not
     necessary. Otherwise Next_content_partition will generate
     a wrong partition. */

  if ((total_lower_content > parent_content) ||
      (total_upper_content < parent_content))
  {
    free((void *)actual_content_distribution);
    actual_content_distribution = NULL;
    return;
  }

  /* set up the first content partition, this should always be
     possible  */
  unuse_content = parent_content - total_lower_content;
  Next_content_partition (actual_content_distribution,
                          upper_limit_distribution,
                          array_vol,
                          unuse_content);
  if (debuglist[7])
  {
    fprintf (out, "\nThe first content partition: %d\n", array_vol);
    tsibling = parent_ptr->first_child;
    for (count = 0; count < array_vol; count++)
    {
      fprintf (out, "Cell %x's content is %d.\n", tsibling,
               actual_content_distribution[count]);
      Print_cell (tsibling);
      tsibling = tsibling->sibling;
    }
  }
  Partition_content (status, current_level, parent_ptr,
                     gptr, bptr, extgp, mapping,
                     array_vol, actual_content_distribution,
                     next_part_cell);

  while (more_content_partition)
  {
    decrease_position = increase_position = -1;
    /* find first position that can be decreased */
    for (i = 0; i < array_vol; i++)
    {
      if (actual_content_distribution[i] > lower_limit_distribution[i])
      {
        decrease_position = i;
        break;
      }
    }
    if (decrease_position < 0)
      more_content_partition = FALSE;
    else
    {
      /* find first increasable position, increase it by one and
         work out the unuse content */
      for (j = i+1; j < array_vol; j++)
      {
```



```
        if (actual_content_distribution[j] <
            upper_limit_distribution[j])
        {
          increase_position = j;
          ++actual_content_distribution[increase_position];
          unuse_content = parent_content;
          for (k = 0; k < increase_position; k++)
          {
            actual_content_distribution[k] =
            lower_limit_distribution[k];
            unuse_content -= actual_content_distribution[k];
          }
          for (k = increase_position; k < array_vol; k++)
            unuse_content -= actual_content_distribution[k];
          break;
        }
      }
      if (increase_position >= 0)
      {
        /* Next_content_partition */
        Next_content_partition (actual_content_distribution,
                                upper_limit_distribution,
                                array_vol,
                                unuse_content);
        if (debuglist[7])
        {
          fprintf (out, "Next content partition: %d\n",
                   array_vol);
          tsibling = parent_ptr->first_child;
          for (count = 0; count < array_vol; count++)
          {
             fprintf (out, "Cell %x's content is %d.\n",
                      tsibling,
                      actual_content_distribution[count]);
             tsibling = tsibling->sibling;
          }
          Print_int_array (array_vol, actual_content_distribution);
        }
        Partition_content (status, current_level, parent_ptr,
                           gptr, bptr, extgp, mapping,
                           array_vol, actual_content_distribution,
                           next_part_cell);
      }
      else
        more_content_partition = FALSE;
    }
  } /* while */

  /* release all the local nodes */
  free((void *)actual_content_distribution);
  actual_content_distribution = NULL;

} /* Start_gen */

/*********************************************************************/

/* Restore and backtrack all the cells to previous level for
   backtracking. */

/* Input parameter
   1) parent_ptr: the parent pointer
*/

void Restore_prev_level(ptr_to_cell_node parent_ptr)
{
   ptr_to_cell_node tcell, freecell;
   ptr_to_neigh_info_node tneigh_info, free_neigh_info, dummy_neigh;
   int tneigh_ver, cell_rep, parent_cell_num, fpos, tpos, lpos;

   /* Decompose stage, free rows and columns and reset allcells and
      cell_label. */
   tcell = parent_ptr->first_child;
   fpos = tcell->representative;
   while (tcell != NULL)
   {
```

```
      /* Free the row except the node with row_cell_num equal to
         col_cell_num. */
      dummy_neigh = tcell->row_first_neigh;
      tneigh_info = dummy_neigh->row_next_neigh;
      while (tneigh_info != dummy_neigh)
      {
         free_neigh_info = tneigh_info;
         tneigh_info = tneigh_info->row_next_neigh;

         /* N.B. If we take out the following 'if' statement,
                 the row_next_neigh of the neighbor node
                 where row_cell_num == col_cell_num will
                 be changed, hence the traversal of the
                 row will lead to the free node. The
                 solution are one of the following:
                 1) set up a if statement as below,
                 2) do the column first
                 3) add one more field in the
                    neigh_info_node to link the free node.
         */

         if (free_neigh_info->row_cell_num !=
             free_neigh_info->col_cell_num)
         {
            free_neigh_info->col_prev_neigh->col_next_neigh =
               free_neigh_info->col_next_neigh;
            free_neigh_info->col_next_neigh->col_prev_neigh =
               free_neigh_info->col_prev_neigh;
            Free_neigh_info_node (free_neigh_info);
         }
      }
      Free_neigh_info_node (dummy_neigh);

      /* Free the column. */
      dummy_neigh = tcell->col_first_neigh;
      tneigh_info = dummy_neigh->col_next_neigh;
      while (tneigh_info != dummy_neigh)
      {
         free_neigh_info = tneigh_info;
         tneigh_info = tneigh_info->col_next_neigh;

         if (free_neigh_info->row_cell_num !=
             free_neigh_info->col_cell_num)
         {
            free_neigh_info->row_prev_neigh->row_next_neigh =
               free_neigh_info->row_next_neigh;
            free_neigh_info->row_next_neigh->row_prev_neigh =
               free_neigh_info->row_prev_neigh;
         }
         Free_neigh_info_node (free_neigh_info);
      }
      Free_neigh_info_node (dummy_neigh);

      cell_list[tcell->representative] = NULL;
      freecell = tcell;
      lpos = freecell->ver_before_rep;

      tcell = tcell->sibling;

      /* reset allcells */
      if (tcell != NULL)
        allcells[lpos] = tcell->representative;
      else
        allcells[lpos] = fpos;

      freecell->prev_cell->next_cell = freecell->next_cell;
      freecell->next_cell->prev_cell = freecell->prev_cell;
      Free_cell_node (freecell);
   }

   /* reset cell_label */
   tpos = parent_cell_num = parent_ptr->representative;
   cell_label[tpos] = parent_cell_num;
   lpos = allcells[tpos];
   while (lpos != parent_cell_num)
```



```
   {
     tpos = lpos;
     cell_label[tpos] = parent_cell_num;
     lpos = allcells[tpos];
   }

   /* In the parent_ptr,
      row_first_neigh, col_first_neigh, parent,
      sibling, prev_cell, next_cell, and representative,
      should stay the same.

      ver_before_rep may change due to the different
      ways in splitting the cell (e.g. according to
      the orbits). I.e. the order of cycle form in
      allcells is changed.

      lower_content, upper_content, and upper_content
      will be set as the next content partition in
      Start_gen. The reason that the lower and upper
      content do not need to be initialized is because
      they are stored locally in Start_gen.
      Once the actual_content is determined,
      max_allow_wastage will be calculated, hence,
      these values do not need to be reset.
   */

   parent_ptr->first_child = NULL;
   dummy_cell->total_children -= (parent_ptr->total_children - 1);
   parent_ptr->total_children = 0;
   parent_ptr->ver_before_rep = tpos;

   /* parent_ptr->next_cell = parent_ptr->prev_cell->next_cell; */
   if (parent_ptr->next_cell != parent_ptr->prev_cell->next_cell)
   {
      fprintf (out, "\nparent_ptr->next_cell not equal to ");
      fprintf (out, "parent_ptr->prev_cell->next_cell when restore!!!\n");
       Myexit(1);
   }
   parent_ptr->prev_cell->next_cell = parent_ptr;
   parent_ptr->next_cell->prev_cell = parent_ptr;

   /* Assemble stage. */
   /* parent_ptr = Pop_cell_stack(). */
   /* Instead of pop, we do the following: */

   cell_list[parent_ptr->representative] = parent_ptr;

   /* Reconnect the column. N.B. The column is pushed last. */
   /* dummy_neigh = Pop_neigh_info_node (); */
   /* Instead of pop, we do the following: */
   dummy_neigh = parent_ptr->col_first_neigh;

   tneigh_info = dummy_neigh->col_next_neigh;
   while (tneigh_info != dummy_neigh)
   {
      if (tneigh_info->row_cell_num != parent_cell_num)
      {
         tneigh_info->row_prev_neigh->row_next_neigh
           = tneigh_info;
         tneigh_info->row_next_neigh->row_prev_neigh
           = tneigh_info;
      }
      tneigh_info = tneigh_info->col_next_neigh;
   }

   dummy_neigh = parent_ptr->row_first_neigh;

   tneigh_info = dummy_neigh->row_next_neigh;
   while (tneigh_info != dummy_neigh)
   {
      if (tneigh_info->col_cell_num != parent_cell_num)
      {
         tneigh_info->col_prev_neigh->col_next_neigh
           = tneigh_info;
         tneigh_info->col_next_neigh->col_prev_neigh
```

```
            = tneigh_info;
      }
      tneigh_info = tneigh_info->row_next_neigh;
   }
   if (CONSIS_CHECK)
     Consistence_check();

} /* Restore_prev_level */

/***********************************************************************/

/* Apply the coverage and wastage test together until
   both tests were passed. */

/* Input parameter:
   1) cell_ptr: the cell to be tested
*/

int Coverage_and_wastage_test (ptr_to_cell_node cell_ptr)
{
   int next_c_part = TRUE, wastage_changes;

   /* We only need to check the change_in_wastage
      because the Coverage_test is done before
      Wastage_test.
   */
   if (debuglist[34])
   {
      fprintf (out, "\nThe cells before coverage and wastage rule:\n");
      Print_curr_status ();
      fprintf (out, "\n");
   }

   do
   {
      if (Coverage_test (cell_ptr))
      {
         if (!Wastage_test (cell_ptr, &wastage_changes))
         {
            if (debuglist[9])
            {
              fprintf (out, "upper content too small\n");
            }
            next_c_part = FALSE;
            break;
         }
      }
      else
      {
         if (debuglist[9])
         {
           fprintf (out, "\nlower content too big\n");
         }
         next_c_part = FALSE;
         break;
      }
   } while (wastage_changes);

   if (debuglist[34])
   {
      fprintf (out, "\nThe status after coverage and wastage rule\n");
      Print_curr_status();
      fprintf (out, "\n");
   }

   return (next_c_part);
} /* Coverage_and_wastage_test */

/***********************************************************************/

/* Partition the cell and content without using the
   information of the symmetry group. */

/* input parameters:
   1) current_level: the latest level created
```



```
   2) cell_ptr: pointer to the selected cell which will be partitioned
   3) autogp: the automorphism group to be used
*/

/* debuglist = 28 */

void Regular_partition (int current_level, ptr_to_cell_node cell_ptr,
                        ptr_to_perm_gp autogp)

{
   int j, problem;

   if (cell_ptr == NULL)
   {
      fprintf(out, "\nProgram exit due to NULL pointer was passed");
      fprintf(out, " to Regular_partition.\n");
      Myexit(1);
   }
   else
     if (debuglist[28])
     {
         fprintf(out, "\nCell passed in for Regular Partition are:\n");
         Print_cell(cell_ptr);
     }

   Partition_cell (cell_ptr);
   Create_influ_func_info (cell_ptr, cell_ptr->first_child);
   Handle_new_level (cell_ptr);
   if (!(finish_computation) && (Coverage_and_wastage_test(cell_ptr)))
     Start_gen (current_level, cell_ptr,
                autogp, NULL, NULL, NULL, REGULAR_PARTITION, NULL);
   Restore_prev_level (cell_ptr);

} /* Regular_partition */

/***********************************************************************/

/* Select_next_partition_cell selects the next cell for partition.
   If such a cell exist, the address of that cell
   is returned, otherwise, a NULL pointer is returned.
   It selects the cell with total vertices > 1 and with minimum
   max_allow_wastage. In case of tide, it selects the first cell with
   bigger number of vertices.

   Output parameter
   the cell selected to be partition
*/

ptr_to_cell_node Select_next_partition_cell (voidplist)

{
   ptr_to_cell_node current_cell, best_cell;

   current_cell = dummy_cell->next_cell;

   while ((current_cell != dummy_cell)
          &&
          (current_cell->cell_size <= 1))
     current_cell = current_cell->next_cell;

   if (current_cell != dummy_cell)
   {
      best_cell = current_cell;
      current_cell = current_cell->next_cell;
      while (current_cell != dummy_cell)
      {
         if (current_cell->cell_size > 1)
           if (current_cell->max_allow_wastage <
               best_cell->max_allow_wastage)
             best_cell = current_cell;
           else
             if (current_cell->max_allow_wastage ==
                 best_cell->max_allow_wastage)
               if (current_cell->cell_size >
                   best_cell->cell_size)
```

```
                best_cell = current_cell;

         current_cell = current_cell->next_cell;
      }
      return (best_cell);
   }
   else
     return (NULL);
} /* Select_next_partition_cell */

/*********************************************************************/

/* When the program reach a stage where a compete partition solution
   exist, All_covered will be called to check if the whole graph can
   be covered by this dominating set. */

/* output parameter;

   1) solution: the set of solutions
   2) return 1 if the solution found is actually a dominating set
*/

int All_covered (int *solution)

{
   static int *all_vertices_set = NULL;

   ptr_to_cell_node current_cell;
   int i, j, count = 1, position, total_covered = 0, tneigh_ver;
   ptr_to_next_neigh_node tneigh_set;

   /* assign all_vertices_set to be equal to all zero */
   if ((all_vertices_set == NULL) || reset_all_vertices_set)
   {
      if (all_vertices_set != NULL)
      {
         reset_all_vertices_set = FALSE;
         free ((void *) all_vertices_set);
         all_vertices_set = NULL;
      }
      all_vertices_set = (int *) calloc (max_level, sizeof(int));
   }

   for (i = 1; i < max_level; i++)
     all_vertices_set[i] = 0;

   current_cell = dummy_cell->next_cell;
   while (current_cell != dummy_cell)
   {
      if (current_cell->actual_content == 1)
      {
         i = solution[count] = current_cell->representative;
         count++;

         /* for each tneigh_ver: an neighbor of i */
         tneigh_set = neigh_graph[i];
         while (tneigh_set != NULL)
         {
            for (j = 0; ((j < NEIGH_NODE_SIZE) &&
                        ((tneigh_ver = tneigh_set->neighbor_name[j])
                          != -1));
                 j++)
            {
               if (all_vertices_set[tneigh_ver] == 0)
               {
                  all_vertices_set[tneigh_ver]++;
                  total_covered++;
               }
            }
            tneigh_set = tneigh_set->next;
         }
      }
      current_cell = current_cell->next_cell;
   }
  if (total_covered == total_vertices)
```



```
    return (TRUE);
  else
    return (FALSE);
} /* All_covered */

/*********************************************************************/

/* To check if a solution is found. */

void Check_and_handle_solution (voidplist)

{
   static int *temp_solution = NULL; /* an array to store a temporary
                                        set of solution */
   static int  old_content = 0; /* when the program run more than one
                                   time, it keep the content of the
                                   last run */

   int i;
   double isom_soln;

   if ((temp_solution != NULL) && (old_content != total_content))
   {
      free((void *) temp_solution);
      temp_solution = NULL;
   }
   if ((temp_solution == NULL) || (old_content != total_content))
   {
      old_content = total_content;
      temp_solution = (int *) calloc (total_content + 1, sizeof(int));
   }

   if (All_covered (temp_solution))
   {
      total_solutions++;
      fprintf (out, "\nThe %d set of solution is:\n", total_solutions);
      for (i = 1; i <= total_content; i++)
      {
        fprintf (out, "%d ", temp_solution[i]);
      }
      fprintf (out, "\n");

/*
      isom_soln = Get_isomorphic_size (temp_solution);
      total_isomorphic_solution += isom_soln;
      fprintf (out, "\nThe total number of isomorphic solution of ");
      fprintf (out, "this set of dominating set are: %lf.\n", isom_soln);
      fprintf (out, "\nThe total number of isomorphic solution obtained ");
      fprintf (out, "up to now are: %lf.\n", total_isomorphic_solution);
*/

      if (total_solutions == N_SOLUTION_ONLY)
      {
         fprintf (out, "\nThe search for dominating set stop once ");
         fprintf (out, "the %d th solution is found.\n ", total_solutions);
         Calculate_and_print_time_used();
         /* stop generating any more partition of content */
         fflush (out);
         finish_computation = TRUE;
         /* return is used instead of exit because the program may
            want to automatically restart with another initial
            content partition for the input cell partition */
         return;
/*
         Myexit(1);
*/
      }
      Calculate_and_print_time_used ();
      fflush (out);
   }
} /* Check_and_handle_solution */

/*********************************************************************/

/* To handle the content partition generated like
```

```
   Update_the_children_contents, Update_max_allowable_wastage,
   Select_next_partition_cell, Coverage_test, Wastage_test.
   It check if that partition lead to a solution.
*/

/* debuglist = 8 */

void Handle_reg_content_part (int current_level,
                              ptr_to_cell_node parent_ptr,
                              ptr_to_perm_gp gptr,
                              int array_vol,
                              int *content_distribution)

{
   ptr_to_cell_node selected_cell;

   if (debuglist[8])
   {
      fprintf (out, "the following is a good set of contents:\n");
      Print_int_array (array_vol, content_distribution);
   }
   selected_cell = Select_next_partition_cell();
   if (selected_cell != NULL)
   {
      if (debuglist[8])
      {
         fprintf (out, "the following is the selected cell:\n");
         Print_cell (selected_cell);
      }
      global_level = current_level + 1;
      Next_level(REGULAR, current_level + 1, selected_cell, gptr);
      global_level = current_level;
   }
   else
     Check_and_handle_solution ();

} /* Handle_reg_content_part */

/***************************************************************************/

/* User input the first cell partitions. User is responsible for a
   correct set of cell partition.

   Input parameter:
   1) parent_ptr: pointer to parent cell

*/

void Input_first_cell_partition (ptr_to_cell_node parent_ptr)

{
  int i, j, crep, vb4rep, pos, first_pos, last_pos,
      csize = 0, sum_vertices = 0;
  ptr_to_cell_node cell_ptr, last_sible;

  /* read the total number of subcells */
  fscanf (in, "%d", &total_first_part_subcells);
  fprintf (out, "\nThe total number of subcells in this partition are %d.\n",
                 total_first_part_subcells);
  if ((total_first_part_subcells < 1) ||
      (total_first_part_subcells > total_vertices))
  {
     fprintf (out, "\nNo such partition!\n");
     Myexit(1);
  }

  cell_ptr = Get_initialized_cell_node();
  parent_ptr->first_child = cell_ptr;
  parent_ptr->prev_cell->next_cell = cell_ptr;
  cell_ptr->prev_cell = parent_ptr->prev_cell;

  for (i=0; i < total_first_part_subcells; i++, csize = 0)
  {
     parent_ptr->total_children++; /* one more children */
```



```
     cell_ptr->parent = parent_ptr;
     cell_ptr->sibling = Get_initialized_cell_node();
     cell_ptr->next_cell = cell_ptr->sibling;
     cell_ptr->sibling->prev_cell = cell_ptr;

     vb4rep = 0;
     /* read in the vertices in each subcell */
     fscanf (in, "%d", &pos);
     if (!input_vertex_start_from_1)
       pos++;
     crep = first_pos = pos;
     do
     {
        last_pos = pos;
        fscanf (in, "%d", &pos);
        if (pos != -1)
        {
           if (!input_vertex_start_from_1)
             pos++;
           allcells[last_pos] = pos;
           if (crep > pos)
           {
               vb4rep = last_pos;
               crep = pos;
           }
        }
        else /* this is the last position */
          allcells[last_pos] = first_pos;
        csize++;
     } while (pos != -1);
     cell_ptr->representative = crep;
     if (vb4rep) /* not the first vertex (!= 0)*/
       cell_ptr->ver_before_rep = vb4rep;
     else
       cell_ptr->ver_before_rep = last_pos;

     /* create the dummies */
     cell_ptr->row_first_neigh = Get_initialized_neigh_info_node();
     cell_ptr->row_first_neigh->row_cell_num = crep;
     cell_ptr->row_first_neigh->col_cell_num = 0;
     cell_ptr->col_first_neigh = Get_initialized_neigh_info_node();
     cell_ptr->col_first_neigh->row_cell_num = 0;
     cell_ptr->col_first_neigh->col_cell_num = crep;

     cell_ptr->cell_size = csize;
     cell_ptr->lower_content = 0;

     /* Since the parent's content is not defined, the
        upper_content cannot be defined.
        It is defined in the procedure Get_content_partition. */

     pos = crep;
     /* update cell_label */
     do
     {
        cell_label[pos] = crep;
        pos = allcells[pos];
     } while (pos != crep);

     cell_list[crep] = cell_ptr;

     last_sible = cell_ptr;
     cell_ptr = cell_ptr->sibling;

     sum_vertices += csize;
  }
  last_sible->next_cell = parent_ptr->next_cell;
  parent_ptr->next_cell->prev_cell = last_sible;
  dummy_cell->total_children += (total_first_part_subcells - 1);

  /* Free the last extra cell node created. */
  /* N.B. The free will not work if no partition occur,
     then cell_ptr == last_sible. */
  Free_cell_node (last_sible->sibling);
  last_sible->sibling = NULL;
```

```
  if (sum_vertices != parent_ptr->cell_size)
  {
      fprintf (out, "\nTotal number of vertices in the input ");
      fprintf (out, "subcells are different from the parent cell.\n");
      fprintf (out, "\nTotal vertices in the subcells are %d.\n",
                    sum_vertices);
      fprintf (out, "\nTotal vertices in the parent cell are %d.\n",
                    parent_ptr->cell_size);
      Myexit(1);
  }
} /* Input_first_cell_partition */

/*****************************************************************/

/* Read in each set of content partition given by the user and
   initialize the actual content of the first cell and
   work out the maximum allowable wastage of the first cell.
   Global variable finish_computation is updated here. */

/* Input parameter:
   1) parent_ptr: the parent pointer

   Side effect: finish_computation may be changed here
*/

void Get_content_for_first_partition (ptr_to_cell_node parent_ptr)

{
  ptr_to_cell_node cell_ptr;
  int sum_content = 0, temp;

  if (!automated_content_partition)
  {
      fprintf (out, "\nUser input the first content's partition.\n");
      fprintf(out, "\n\nWorking on the content partition: ");
      cell_ptr = parent_ptr->first_child;
      fprintf(out, "\nContent read: ");
      while (cell_ptr != NULL)
      {
         /* scan the content of the other subcells */
         fscanf (in, "%d", &temp);
         fprintf(out, "%d ", temp);
         /* update the first cell */
         cell_ptr->upper_content = cell_ptr->lower_content
                                 = cell_ptr->actual_content
                                 = temp;
         sum_content += temp;
         cell_ptr = cell_ptr->sibling;
      }
      fprintf(out,"\n");
      parent_ptr->upper_content = parent_ptr->lower_content
                                = parent_ptr->actual_content
                                = total_content
                                = sum_content;
  } /* end if (! automated_content_partition) */
  else
  {
      fprintf (out, "Automated content partition.\n");
      fscanf (in, "%d", &sum_content);
      fprintf(out, "\nContent read: ");
      fprintf(out, "%d\n", sum_content);
      parent_ptr->upper_content = parent_ptr->lower_content
                                = parent_ptr->actual_content
                                = total_content
                                = sum_content;

      /* update the prev_upper_content and upper_content
         of all the children */

      cell_ptr = parent_ptr->first_child;
      while (NULL != cell_ptr)
      {
         if (cell_ptr->cell_size <= sum_content)
           cell_ptr->upper_content = cell_ptr->cell_size;
```



```
         else
           cell_ptr->upper_content = parent_ptr->actual_content;
         cell_ptr = cell_ptr->sibling;
      } /* end while */

   }  /* else */

   if (total_content > total_vertices)
   {
      fprintf (out, "\nnumber of contents are greater than the ");
      fprintf (out, "total number of vertices, program halt.\n");
       Myexit(1);
   }
   /* work out the maximum allowable wastage of the parent */
   /* since the first cell only has one neighbor */
   temp = Get_UIF
          (parent_ptr->row_first_neigh->row_next_neigh->
             tot_ver_of_deg_from_row_to_col,
           total_content);

   if ((parent_ptr->max_allow_wastage =
        temp - parent_ptr->cell_size) < 0)
   {
      fprintf (out, "\nProgram halt due to maximum allowable ");
      fprintf (out, "wastage of the first cell < 0.\n");
      finish_computation = TRUE;
   }
} /* Get_content_for_first_partition */

/*****************************************************************/

/* Find the dominating set from the subcells in the second
   level.
   N.B. Here we assume all the cell are in orbits and the
        auto group is the automorphism group of the orbits. */

/* Input parameter:
   1) cell_ptr: the parent cell of the cells in the second level
*/

void Start_from_second_level (ptr_to_cell_node cell_ptr)
{
   start_level = 2;
   Create_influ_func_info (cell_ptr, cell_ptr->first_child);
   Handle_new_level (cell_ptr);
   if (!(finish_computation) && (Coverage_and_wastage_test(cell_ptr)))
     Start_gen (start_level, cell_ptr,
                auto_gp, NULL, NULL, NULL, ORBIT_PARTITION, NULL);
   Restore_prev_level (cell_ptr);
} /* Start_from_second_level */

/*****************************************************************/

/* Initiate basicdom variables will initiate all the variables
   in basicdom.c and read in the input data.

   side effects: memory allocation and fscanf is used here */

/* global variables initialize or allocated here are;
   total_solutions,
   finish_computation,
   total_isomorphic_solution,
   input_vertex_start_from_1,
   any_debug, debuglist,
   direct,
   total_vertices,
   max_level,
   level_visited,
   adj_matrix,
   neigh_graph,
   vertex_degree,
   input_graph_type,
   max_degree,
   dummy_cell,
   allcells,
```

```
   cell_label,
   cell_list,
   cell_calloc_size,
   neigh_calloc_size,
   tvdrc_calloc_size
   list_of_cells_calloc_size
*/

/* debug list value = 10 */

void Initiate_basicdom_val (voidplist)

{
  int i, j, k, input_graph_type, input_graph_num,
      any_debug, temp;

  /* global variables */
  total_solutions = 0;
  finish_computation = FALSE;

  total_isomorphic_solution = 0;

  /* read in the input graph number */
  fscanf (in, "%d", &input_graph_num);
  fprintf (out, "\nWorking on graph number %d.\n",
           input_graph_num);

  /* read in the the input vertex system */
  fscanf (in, "%d", &input_vertex_start_from_1);

  /* read in the debuglist */
  fscanf (in, "%d", &any_debug);
  if (! any_debug)
    for (i = 0; i <= DEBUGLISTSIZE; i++)
      debuglist[i] = 0;
  else
    Init_debuglist ();

  /* indicate the type of graph */
  fscanf (in, "%d", &direct);

  if (direct)
  {
     fprintf (out, "\nIt is a direct graph.\n");
     fprintf (out, "\nThe current version do not support direct graph.\n");
      Myexit(1);
  }

  /* initiate total_vertices */

  fscanf (in, "%d", &total_vertices);
  fprintf (out, "The total number of vertices in this graph are ");
  fprintf (out, "%d.\n", total_vertices);
  if ((total_vertices > MAX_GRAPH_SIZE) ||
     (total_vertices < MIN_GRAPH_SIZE))
  {
     skip_this_graph = TRUE;
     fprintf (out, "\n");
     if (total_vertices < MIN_GRAPH_SIZE)
       fprintf (out, "The input graph size %d is smaller than %d.\n",
                  total_vertices, MIN_GRAPH_SIZE);
     else
       fprintf (out, "The input graph size %d is bigger than %d.\n",
                  total_vertices, MAX_GRAPH_SIZE);
     fprintf (out, "This graph will be skipped\n");
     return;
  }

  /* initialize cell_calloc_size, neigh_calloc_size,
     tvdrc_calloc_size, list_of_cells_calloc_size */
  /* initialized to max of 100 and total_vertices/4 */

  temp = total_vertices/MEMORY_FACTOR + 1;
  if (temp <= MIN_CALLOC_SIZE)
    cell_calloc_size = neigh_calloc_size =
```



```
      tvdrc_calloc_size = list_of_cells_calloc_size =
      MIN_CALLOC_SIZE;
  else
    cell_calloc_size = neigh_calloc_size =
      tvdrc_calloc_size = list_of_cells_calloc_size = temp;

  /* initiate the max_level and level_visited */
  /* 2 more than total_vertices to make sure one space
     at position zero and one after position total_vertices */

  max_level = total_vertices + 1;

  level_visited = (unsigned *) calloc (max_level, sizeof(unsigned));
  total_content_part = (unsigned *) calloc (max_level, sizeof(unsigned));
  bfcwtest = (unsigned *) calloc (max_level, sizeof(unsigned));
  afcwtest = (unsigned *) calloc (max_level, sizeof(unsigned));
  bfmawtest = (unsigned *) calloc (max_level, sizeof(unsigned));
  afmawtest = (unsigned *) calloc (max_level, sizeof(unsigned));

  for (i=0; i < max_level; i++)
  {
     total_content_part[i] = level_visited[i] = 0;
     bfcwtest[i] = afcwtest[i] = bfmawtest[i] = afmawtest[i] = 0;
  }
  total_content_part[0] = level_visited[0] = 1;

  /* allocate memory to the adj_matrix */
  adj_matrix = (char **) calloc (total_vertices+1, sizeof (char *));
  adj_matrix[0] = NULL;
  for (i = 1; i <= total_vertices; i++)
  {
      adj_matrix[i] = (char *) calloc(total_vertices+1, sizeof (char));
      for (j = 0; j <= total_vertices; j++)
         adj_matrix[i][j] = 0;
  }

  /* allocate memory to the array neigh_graph */
  neigh_graph = (ptr_to_next_neigh_node *) calloc (max_level,
                     sizeof (ptr_to_next_neigh_node));

  /* initiate all the pointer in neigh_graph to NULL*/
  for (i = 0; i < max_level; i++)
    neigh_graph[i] = NULL;

  /* allocate memory to vertex_degree */
  vertex_degree = (int *) calloc (max_level, sizeof (int));

  /* initiate all the entry in vertex_degree to zero */
  Init_int_array (vertex_degree, 0, max_level-1, 0);

  /* check what type of input graph is given, 0 for adjacent matrix,
     1 for neighbor vertices */
  fscanf (in, "%d", &input_graph_type);

  if (input_graph_type)
    Init_neigh_graph_given_neigh_vertex ();
  else
    Init_neigh_graph_given_adj_matrix ();

  /* initialize max_degree */
  for (i = 1; i <= total_vertices; i++)
    if (max_degree < vertex_degree[i])
      max_degree = vertex_degree[i];

  /* create the an array to store all the tot_ver_of_deg_from_row_to_col
     entries when working out each new neighbor information node */
  all_tot_ver_deg_r_c = (int ***) calloc (total_vertices+1, sizeof (int **));
  all_tot_ver_deg_r_c[0] = NULL;
  for (i = 1; i <= total_vertices; i++)
  {
     all_tot_ver_deg_r_c[i] = (int **) calloc (total_vertices+1, sizeof (int *));
     all_tot_ver_deg_r_c[i][0] = NULL;
     for (j =1; j <= total_vertices; j++)
     {
       all_tot_ver_deg_r_c[i][j] = (int *) calloc
```

```
        (max_degree + 1, sizeof (int));
      for (k = 0; k < max_degree + 1; k++)
        all_tot_ver_deg_r_c[i][j][k] = 0;
    }
  }

  /* initialize dummy_cell */
  dummy_cell = Get_initialized_cell_node();

  /* initialize allcells and cell_label */
  Init_allcells_and_cell_label ();

  /* initialize cell_list */
  cell_list = (ptr_to_cell_node *) calloc (max_level,
              sizeof (ptr_to_cell_node));
  for (i = 0; i < max_level; i++)
    cell_list[i] = NULL;

  if (debuglist[10])
    Print_input_info ();

} /* Initiate_basicdom_val */

/*******************************************************************/

/* This procedure will set the bounds in the contents of the input
   partition. It is only called when automated_cell_partition
   is false and automated_content_partition is true. Parent's
   content will be updated here. */

/* Input parameter:
   1) parent_ptr: the parent pointer
*/

void Update_content_of_first_partition (ptr_to_cell_node parent_ptr)

{
   ptr_to_cell_node tchild;
   int pcontent;

   fscanf (in, "%d", &pcontent);
   if (parent_ptr->cell_size < pcontent)
   {
      fprintf (out, "\nThe input content size is too big. Program halt.\n");
      Myexit(1);
   }

   parent_ptr->lower_content = parent_ptr->upper_content =
          parent_ptr->actual_content = pcontent;
   tchild = parent_ptr->first_child;

   while (tchild != NULL)
   {
      if (tchild->cell_size >= pcontent)
        tchild->upper_content = pcontent;
      else
        tchild->upper_content = tchild->cell_size;
      tchild = tchild->sibling;
   }

} /* Update_content_of_first_partition */

/*******************************************************************/

/* Free the first level, it only have four nodes and the
   first cell. */

void Free_first_level (voidplist)

{
   ptr_to_cell_node first_cell;
   ptr_to_neigh_info_node curr_neigh, dummy;

   first_cell = cell_list[1];
   dummy = first_cell->row_first_neigh;
```



```
   curr_neigh = dummy->row_next_neigh;
   Free_neigh_info_node (dummy);
   Free_neigh_info_node (curr_neigh);
   curr_neigh = NULL;
   dummy = first_cell->col_first_neigh;
   Free_neigh_info_node (dummy);
   dummy = NULL;
   Free_cell_node (first_cell);
   first_cell = NULL;
   Free_cell_node (dummy_cell);
   dummy_cell = NULL;
} /* Free_first_level */

/*******************************************************************/

/* This procedure will reinitialize all the variables required to make
   the program run for another set of user input partition cell and
   content given the same graph. */

void Reinitiate_basicdom_val (voidplist)

{
  int i, j, k;
  ptr_to_next_neigh_node neigh_ptr, next_neigh_ptr;

  /* Static variable temp_solution do not need to be reset, the
     procedure had handle all the situations. */

  /* global variables */
  Free_first_level ();

  free ((void *) allcells);
  allcells = NULL;
  free ((void *) cell_label);
  cell_label = NULL;
  free ((void *) level_visited);
  level_visited = NULL;
  free ((void *) bfcwtest);
  bfcwtest = NULL;
  free ((void *) afcwtest);
  afcwtest = NULL;
  free ((void *) bfmawtest);
  bfmawtest = NULL;
  free ((void *) afmawtest);
  afmawtest = NULL;

  for (i = 1; i <= total_vertices; i++)
    free ((void *) adj_matrix[i]);
  free ((void *) adj_matrix);
  adj_matrix = NULL;

  for (i = 1; i <= total_vertices; i++)
  {
     next_neigh_ptr = neigh_graph[i];
     while (next_neigh_ptr != NULL)
     {
        neigh_ptr = next_neigh_ptr;
        next_neigh_ptr = neigh_ptr->next;
        free ((void *) neigh_ptr);
     }
  }
  free ((void *) neigh_graph);
  neigh_graph = NULL;

  free ((void *) vertex_degree);
  vertex_degree = NULL;

  for (i = 1; i <= total_vertices; i++)
  {
     for (j = 1; j <= total_vertices; j++)
       free ((void *) all_tot_ver_deg_r_c[i][j]);
     free ((void *) all_tot_ver_deg_r_c[i]);
  }
  free ((void *) all_tot_ver_deg_r_c);
  all_tot_ver_deg_r_c = NULL;
```

```
  free ((void *) cell_list);
  cell_list = NULL;
  free ((void *) sbuffer);
  sbuffer = NULL;
  free ((void *) fbuffer);
  fbuffer = NULL;
  free ((void *) ebuffer);
  ebuffer = NULL;

  total_solutions = 0;
  total_first_part_subcells = 0;
  max_degree = 0;
  global_level = 0;
  finish_computation = FALSE;

  reset_connected_to_cell =
  reset_s_slack =
  reset_unavoidable_wastage =
  reset_old_max_allowable_wastage =
  reset_all_vertices_set =
  reset_cell_counts = TRUE;

} /* Reinitiate_basicdom_val */

/*************************************************************************/
```

```
#ifndef GROUPDOM

#define GROUPDOM 1

#if  defined(apollo) || defined(__GNUC__)
#define HALFANSI
#define voidplist void
#else
#ifdef mips
/*   #define SYSTYPE_BSD43 */
#define HALFANSI
#define void char
#define voidplist
#else
#define voidplist void
#endif
#endif

/* this header file contain all the procedures related to groups */

/*********************************************************************/
/* GLOBAL TYPES */
/*********************************************************************/

/*********************************************************************/
/* GLOBAL VARIABLES */
/*********************************************************************/

/* global variables relate to isom groupdcl */

/* the partition stabilizer of the latest level
   according to the size and the content of the
   cells */
extern ptr_to_perm_gp auto_gp;  /* the automorphism of the graph */

extern double total_isomorphic_solution;

/*********************************************************************/
/* FUNCTIONS */
/*********************************************************************/

/* To decide whether the next level of cell partition is the
   regular partition (not making use of the group) or
   symmetry partition (making use of the group).
*/

/* Input parameters
   1) tag: the way to partition the cell
   2) lev: current level
   3) cptr: the pointer to the cell to be partitioned
   4) gptr: the pointer to the automorphism group
*/

/* debuglist = 24 */

extern void Next_level (enum part_type tag, int lev,
                        struct cell_node *cptr,
                        ptr_to_perm_gp gptr);

/*********************************************************************/

/* To generate the group. */
/*
   output parameter
   return the group generated by generator, outgp will be allocated in
   this procedure
*/

/* debug list value = 12 */

extern ptr_to_perm_gp Create_group_by_generator (voidplist);

extern ptr_to_perm_gp Generate_partition_stabilizer(voidplist);

/*********************************************************************/
```



```
/* Handle the content partition in the case of not transitive, and the
   cells are partitioned into orbits. */
/* This procedure is used by Start_gen in basicdom. */

/* N.B. disposepg (part_stabilizer) may cause a lot of CPU time. */

/* Input parameters
   1) current_level: the latest level created
   2) parent_ptr: pointer to the cell which was partitioned
   3) gptr: the current automorphism group
   4) array_vol: the total number of subcells partitioned from the
                 parent
   5) content_distribution: the content distribution of the children
                            partitioned from the parent
   6) next_part_cell: if not NULL, it will contain the next cell
                      to be partitioned

*/

extern void Handle_orb_content_part (int current_level,
                                     ptr_to_cell_node parent_ptr,
                                     ptr_to_perm_gp gptr,
                                     int array_vol,
                                     int *content_distribution,
                                     ptr_to_cell_node next_part_cell);

/*********************************************************************/

/* Handle the content partition in the case of transitive and imprimitive,
   and the cells are partitioned according to the block obtained. */
/* This procedure is used by Start_gen in basicdom. */

/* N.B. disposepg (part_stabilizer) may cause a lot of time */

/* Input parameters
   1) current_level: the latest level created
   2) parent_ptr: pointer to the cell which was partitioned
   3) gptr: the current automorphism group
   4) bptr: the block automorphism group
   5) extgp: the intermediate group in the homomorphism
   6) mapping: to map back the elements in the truncated group to their old
               value in the original graph
   7) block_auto: the block automorphism group
   8) array_vol: the total number of subcells partitioned from the
                 parent
   9) content_distribution: the content distribution of the children
                            partitioned from the parent

*/

/* debuglist = 32 */

extern void Handle_transitive_content_part (int current_level,
                               ptr_to_cell_node parent_ptr,
                               ptr_to_perm_gp gptr,
                               ptr_to_perm_gp bptr,
                               ptr_to_perm_gp extgp,
                               ptr_to_permvect mapping,
                               int array_vol,
                               int *content_distribution,
                               ptr_to_perm_gp block_auto);

/*********************************************************************/

/* This procedure will initialize all the variables in this program:
   groupdom.c, the initialization in domset.c should be done first */

extern void Initiate_groupdom_val (voidplist);

/*********************************************************************/

/* This procedure will reinitialize all the variables
   that is required to make
   the program run for another set of user input partition cell and
```

```
   content given the same graph. */

extern void Reinitiate_groupdom_val (voidplist);

/**********************************************************************/

/* Return true if the content partition is not isomorphic to an old
   content partition under the symmetry group
   canonical_rep should be initialized to true before enter to this
   procedure.

   input parameters
   1) b_ptr: pointer to the permutation group of the block
   2) content_dist: the content partition

   output parameters
   1) canonical_rep is changed to FALSE if this content partition
      is isomorphic to an early partition
   2) the automorphism group of the block will be returned
*/

/* debuglist = 21 */

extern ptr_to_perm_gp Isomorphic_test (ptr_to_perm_gp b_ptr,
                                int *content_dist,
                                int total_blks);

/**********************************************************************/

#endif
```

```
#include <stdio.h>
#include "basicdom.h"
#include "memmdom.h"
#include "groupdom.h"

/*
This file contains all the group related operations. It also serve
as an interface between the dominating set program and the isom package.

The dominating set program needs
1) the orbits to partition the graph,
2) the invariant block to partition the graph,
3) the action of the invariant block to partition the content.

symmetry group G
sub-group H
*/

/*********************************************************************/

/* GLOBAL VARIABLES related to isom groupdcl */

ptr_to_perm_gp auto_gp;  /* the automorphism of the graph */

double total_isomorphic_solution = 0;

/* GLOBAL VARIABLES (belongs to this program) */

int canonical_rep;
double initial_group_order;
ptr_to_permvect content_part = NULL, max_part = NULL;
ptr_to_permvect cwpgcycle = NULL; /* a vector to store the
                                     cells with the same size */
ptr_to_permvect cwpgcolor = NULL; /* the color of each cell */
int **colormat = NULL; /* a two dimension array to assign the
                          color to cwptcolor */
int *more_than_one_orbs; /* it is an array of size total_vertices +1,
                            it is used to record which cell (using
                            its representative) will
                            have more than one orbits in it so that
                            one of the cells will be chosen as the
                            next partition cell */
ptr_to_list_of_cells next_part_cell_list = NULL;
   /* cell in this list has highest priority to be partitioned next */

/* The following variables are used to control the static variable.
   When this program is run for more than one graph, those static
   variables needed to be reset. */

int reset_tempvec_ptr = FALSE,
    reset_extended_map = FALSE,
    reset_stabilize_vertex = FALSE,
    reset_torbit = FALSE,
    reset_vert_set = FALSE;

/*********************************************************************/
/* FUNCTIONS */
/*********************************************************************/

/* The following is a list of procedures for printing results. */

/*********************************************************************/

/* To print the order of a group.

   input parameters:
   message: an array of characters (string);
   ingp: the input group;
*/

void Print_group_order (char message[], ptr_to_perm_gp ingp)

{
      double approx_ord;
      long actual_ord;
```



```
      ptr_to_permvect tpvect;

      if (ingp == NULL)
      {
        fprintf(out, "%s is NULL\n", message);
      }
      else
      {
         tpvect = allocatepv(ingp->permsize);
         l_gporder(ingp, tpvect, ingp->permsize);
         l_int_exp_factorize(tpvect, ingp->permsize);
         l_int_exp_print(tpvect, ingp->permsize);
         disposepv(tpvect);
      }

} /* Print_group_order */

/*********************************************************************/

/* To print out the group's information like order, address, permsize
   etc.

   input parameters:
   ingp: the input group;
*/

void Print_group_information (ptr_to_perm_gp ingp)

{
   int short_form = 1;

   if (ingp != NULL)
   {
      fprintf (out, "\nThe group address is %x.\n", ingp);
      Print_group_order("Its order is:", ingp);
      fprintf (out, "\n");
      if (!short_form)
      {
         fprintf (out, "permsize is %hd\n", ingp->permsize);
         fprintf (out, "gp_status is %d\n", ingp->gp_status);
         fprintf (out, "base address is %x\n", ingp->base);
         fprintf (out, "rank address is %x\n", ingp->rank);
         fprintf (out, "orbits address is %x\n", ingp->orbits);
         fprintf (out, "tl address is %x\n", ingp->tl);
         fprintf (out, "point address is %x \n", ingp->point);
         fprintf (out, "tau address is %x\n", ingp->point[1].tau);
         fprintf (out, "tauinv address is %x\n", ingp->point[1].tauinv);
         fprintf (out, "sigma address is %x\n", ingp->point[1].sigma);
      }
   }
   else
   {
      fprintf (out, "\nThe input group is NULL.\n");
   }
   fprintf(out, "\n");
} /* Print_group_information */

/*********************************************************************/


/* This procedure will print a list of cells linked by the structure
   list of cells. */

/* Input parameters:
   1) c_list: the list of cells
*/

void Print_list_of_cells (ptr_to_list_of_cells c_list)

{
  int i = 1;
  ptr_to_list_of_cells list_entry;

  fprintf (out, "\nHere is the list of cells linked the structure ");
  fprintf (out, "list_of_cells.\n");
```

```
  if (c_list == NULL)
  {
    fprintf (out, "NULL list.\n");
  }
  else
  {
     list_entry = c_list;
     while (list_entry != NULL)
     {
        fprintf (out, "%d: ", i);
        fprintf (out, "level = %d ", list_entry->level);
        Print_cell(list_entry->corr_cell);
        list_entry = list_entry->next_entry;
        i++;
     }
  }
} /* Print_list_of_cells */

/*********************************************************************/

/* Print the cells linked by the pointer dummy_cell. */

void Print_dummy_list (voidplist)

{
   ptr_to_cell_node current_cell;
   int vert, count = 0;

   current_cell = dummy_cell->next_cell;

   fprintf (out, "Here is the list of cells linked ");
   fprintf (out, "by the pointer dummy_cell.\n");
   while (current_cell != dummy_cell)
   {
      fprintf (out, "cell_node = %x\n", current_cell);
      current_cell = current_cell->next_cell;
   }
} /* Print_dummy_list */

/*********************************************************************/

/* The following is a list of utility procedures. */

/*********************************************************************/

/* Copy the list in the data structure list_of_cells,
   the duplicated list has the same order as the original list. */

/* Input parameters:
   1) source_ptr: the original list

   Side effect:
   The procedure will return a duplicated list.
*/

ptr_to_list_of_cells Copy_list_of_list_of_cells
    (ptr_to_list_of_cells source_ptr)

{
   ptr_to_list_of_cells temp_ptr, last_pos, next_pos, destination_ptr;

   /* Create the situation that the first node and the last node are
      dummy. They will be free at the end. */
   temp_ptr = source_ptr;
   destination_ptr = last_pos = Get_initialized_list_of_cells();
   last_pos->next_entry = Get_initialized_list_of_cells();
   next_pos = last_pos->next_entry;
   while (temp_ptr != NULL)
   {
      next_pos->corr_cell = temp_ptr->corr_cell;
      next_pos->next_entry = Get_initialized_list_of_cells();
      last_pos = next_pos;
      next_pos = next_pos->next_entry;
      temp_ptr = temp_ptr->next_entry;
   }
```



```
   Free_list_of_cells (last_pos->next_entry);
   last_pos->next_entry = NULL;
   temp_ptr = destination_ptr;
   destination_ptr = destination_ptr->next_entry;
   Free_list_of_cells (temp_ptr);
} /* Copy_list_of_list_of_cells */

/*********************************************************************/

/* Releasing the list in the data structure list_of_cells. */

/* Input parameters:
   1) source_ptr: the list to be free
*/

void Release_list_of_list_of_cells (ptr_to_list_of_cells source_ptr)

{
   ptr_to_list_of_cells temp_ptr, free_ptr;

   temp_ptr = source_ptr;
   while (temp_ptr != NULL)
   {
      free_ptr = temp_ptr;
      temp_ptr = temp_ptr->next_entry;
      Free_list_of_cells (free_ptr);
   }

} /* Release_list_of_list_of_cells */

/*********************************************************************/

/* To calculate the group order.

   input parameters:
   ingp: the input group;
*/

double Get_group_order (ptr_to_perm_gp ingp)

{
      double approx_ord;
      long actual_ord;

      if (ingp == NULL)
      {
        fprintf(out, "%s is NULL\n");
      }
      else
      {
         gporder(ingp, &approx_ord, &actual_ord);
         if (actual_ord != -1)
           return((double) actual_ord);
         else
           return(approx_ord);
      }
} /* Get_group_order */

/*********************************************************************/

/* To generate the group. */
/*
   output parameter
   return the group generated by generator, outgp will be allocated in
   this procedure
*/

/* debug list value = 12 */

ptr_to_perm_gp Create_group_by_generator (voidplist)

{
  ptr_to_perm_gp outgp;
  ptr_to_permmatrix generators;
  int totalgen, i, j, input_format, read_gen = 1,
```

```
      temp, first, new;
   char ch;

   fprintf (out, "\nTotal time used before generated the automorphism group is:\n");
   Calculate_and_print_time_used();

   fscanf (in, "%d", &totalgen);
   generators = allocatepm(totalgen);

   fscanf (in, "%d", &input_format);

   if (input_format == 0)
   {
      /* read each generator in its image form */
      for(i = 1; i <= totalgen; i++)
      {
         generators[i] = allocatepv(total_vertices);
         if (input_vertex_start_from_1)
           readpvect(in, generators[i], total_vertices);
         else
         {
            generators[i][0] = 0;
            for (j = 1; j <= total_vertices; j++)
            {
               fscanf (in, "%d", &temp);
               generators[i][j] = temp + 1;
            } /* for */
         }
      }
   }
   else
   {
      generators[0] = NULL;
      for (i = 1; i <= totalgen; i++)
      {
         generators[i] = allocatepv(total_vertices);
         generators[i][0] = 0;
         for (j = 1; j <= total_vertices; j++)
            generators[i][j] = j;
      }

      /* read each generator in its cyclic form */
      while (read_gen <= totalgen)
      {
         ch = getc(in);
         switch (ch)
         {
            case '(': fscanf (in, "%d", &first);
                      if (!input_vertex_start_from_1)
                        first++;
                      temp = first;
                      break;

            case ')': generators[read_gen][temp] = first;
                      break;

            case ' ': break;

            case '\n': break;

            case '\r': break;

            case ',': break;

            case '-': /* get rid or the -1 */
                      ungetc(ch, in);
                      fscanf (in, "%d", &new);
                      read_gen++;
                      break;

            default : /* a number */
                      ungetc(ch, in);
                      fscanf (in, "%d", &new);
                      if (!input_vertex_start_from_1)
                        new++;
```



```
                      generators[read_gen][temp] = new;
                      temp = new;
         }
      }
   }

   if (debuglist[12])
   {
      fprintf (out, "Total number of generators is %d.\n", totalgen);
      for(i = 1; i <= totalgen; i++)
      {
         fprintf (out, "The %d is:\n", i);
         printpvect (out, generators[i], total_vertices);
      }
   }

   outgp = build_trivial_gp(NULL, total_vertices);


   jerrum(outgp, generators, totalgen);

   initial_group_order = Get_group_order(outgp);

   /* how to print the size and the group generated */
   fprintf (out, "\nThe automorphism group order of this graph is %lf.\n",
            initial_group_order);

   /* disposepm will dispose all its generators */
   disposepm (generators, totalgen);
   generators = NULL;

   fprintf (out, "\nTotal time used after generated the automorphism group is:\n");
   Calculate_and_print_time_used();
   return (outgp);

} /* Create_group_by_generator */

/**********************************************************************/

/* To generate the color wreath product group of the latest level.
   The procedure supplies all the parameters to the isom package to
   obtain the wreath product. Due to the special requirement in
   the domset program, cells with same size but different content
   are not allowed to interchange.

   Input parameters:
   outgp: the preallocated color wreath product group
   total_ver: the total number of vertices amount all the cells

   Output parameters:
   return a pointer to the color wreath product group
   outgp will be allocated in this procedure
*/

/* debug value = 13 */

ptr_to_perm_gp Create_cwpg_for_part_stab (ptr_to_perm_gp outgp,
                                          int total_ver)

{
   extern ptr_to_permvect cwpgcycle;
   extern ptr_to_permvect cwpgcolor;
   extern int **colormat;

   /* ptr_to_perm_gp outgp; */
   ptr_to_cell_node tcell;
   int i, j, colorindex = 1;

   for (i = 0; i <= total_ver; i++)
     cwpgcycle[i] = cwpgcolor[i] = 0;

   for (i = 1; i <= total_ver; i++)
   {
      for (j = 0; j <= total_content; j++)
         colormat[i][j] = 0;
```

```
   }

   /* copy the allcells */

   for (i = 0; i <= total_ver; i++)
     cwpgcycle[i] = allcells[i];

   /* each cell_size and actual_content pair should have a unique
      colorindex. */
   tcell = dummy_cell->next_cell;
   while (tcell != dummy_cell)
   {
       int tempval, rep, npos;

       tempval = colormat[tcell->cell_size] [tcell->actual_content];
       if (0 == tempval)
       {
          tempval = colormat[tcell->cell_size] [tcell->actual_content]
                  = colorindex;
          colorindex++;
       } /* end if */

       /* set up the cwpgcolor of tcell */
       /* here we assume the cell always have at least one vertex */

       npos = rep = tcell->representative;
       do
       {
          cwpgcolor[npos] = tempval;
          npos = cwpgcycle[npos];
       } while (rep != npos);

       tcell = tcell->next_cell;
   }

   outgp = color_wreath_prod_gp(outgp, cwpgcycle, cwpgcolor, total_ver);

   /* think of the total_vertices may not be fixed */
   if (debuglist[13])
   {
      fprintf (out, "\nThe cwpgcycle with size %d is:\n", total_ver);
      fprintf (out, "The cwpgcolor with size %d is:\n", total_ver);
      fprintf(out, "\nThe cwpg for this level is:");
      Print_group_information (outgp);
   }

   return (outgp);

} /* Create_cwpg_for_part_stab */

/*********************************************************************/

/* Generate the partition stabilizer. */

/*   Output parameter:
     return the partition stabilizer
     outgp will be allocated here
*/

/* debuglist is 14 */

ptr_to_perm_gp Generate_partition_stabilizer (voidplist)

{
   static ptr_to_perm_gp cwpgp = NULL;

   ptr_to_perm_gp outgp;

   cwpgp = Create_cwpg_for_part_stab (cwpgp, total_vertices);

   if (debuglist[14])
   {
      fprintf (out, "\nThe auto group %x that intersect with ",
               auto_gp);
      fprintf (out, "the cwpgp %x in Generate_partition_stabilizer is:",
```



```
               cwpgp);
      Print_group_information (auto_gp);
   }

   outgp = gp_inter (auto_gp, cwpgp, NULL);

   if (debuglist[14])
   {
     fprintf (out, "\nThe partition stabilizer generated after ");
     fprintf (out, "intersection:\n");
     Print_group_information(outgp);
   }

   return (outgp);

} /* Generate_partition_stabilizer */

/*********************************************************************/

/* Partition the parent cell into discrete subcells with content
   equal to the parameter content, it makes use of an external
   procedure Partition_each_vertex_to_one_cell_with_content_x

   Input parameters
   1) parent: pointer to the cell to be partitioned
   2) content: the content of each subcell, it should be either
            0 or 1

   Side effect
   1) the parent cell node will has all its children created

*/

void Partition_to_discrete_cell (ptr_to_cell_node parent,
                                 int content)

{
  if ((NULL == parent) || (content > 1) || (content < 0))
  {
     fprintf (out, "Illegal input parameters in procedure ");
     fprintf (out, "Partition_to_discrete_cell.\n");
     fprintf (out, "parent address = %d, content = %d.\n",
                   parent, content);
     Myexit(1);
  } /* if ((NULL ==  parent) || (content > 1) || (cotent < 0)) */
  else
  {
     Partition_each_vertex_to_one_cell_with_content_x (parent, content);
  }

} /* Partition_to_discrete_cell */

/*********************************************************************/

/* Partition the cell into cells with size one during Symmetry_partition. */

/* Input parameters
   1) current_level: the current level
   2) cell_ptr: pointer to the selected cell which will be partitioned
*/

void Sym_discretion (int current_level, ptr_to_cell_node cell_ptr)

{
   ptr_to_perm_gp part_stabilizer = NULL;
   ptr_to_cell_node selected_cell, next_partition_cell;
   int next_partition_cell_level = 0;
   ptr_to_list_of_cells temp_rack;

   if (debuglist[26])
   {
     fprintf (out,
       "\nThe cell passed in has actual_content zero or cell's size.\n");
   }
   if (cell_ptr->actual_content == 0)
```

```
         Partition_to_discrete_cell (cell_ptr, 0);
   else
         Partition_to_discrete_cell (cell_ptr, 1);
   if (cell_ptr->total_children > 1)
   /* a cell node of size 1 should not be passed,
      because it was a selected cell */
   /* the cell is partitioned successfully */
   /* this part should always be true, just for defensive check */
   {
      Create_influ_func_info (cell_ptr, cell_ptr->first_child);
      Handle_new_level (cell_ptr);

      if (Update_max_allowable_wastage (cell_ptr))
      {
         if (next_part_cell_list != NULL)
         {
            selected_cell = next_partition_cell
                          = next_part_cell_list->corr_cell;
            temp_rack = next_part_cell_list;
            next_partition_cell_level = next_part_cell_list->level;
            next_part_cell_list = next_part_cell_list->next_entry;
            Free_list_of_cells (temp_rack);
            temp_rack = NULL;
         }
         else
         {
            next_partition_cell = NULL;
            selected_cell = Select_next_partition_cell ();
         }

         if (debuglist[26])
         {
            fprintf (out,
               "Symmetry partition: partition to discrete cells.\n");
            fprintf (out, "The selected cell is:\n");
            Print_cell(selected_cell);
         }
         if (selected_cell != NULL)
         {
            part_stabilizer = Generate_partition_stabilizer();
            global_level = current_level + 1;
            Next_level(SYMMETRY, current_level + 1,
                       selected_cell, part_stabilizer);
            global_level = current_level;
            disposepg (part_stabilizer);
            part_stabilizer = NULL;
         }
         /* otherwise, complete partition */
         else
           Check_and_handle_solution();
         if (next_partition_cell != NULL)
         /* put it back to the list */
         {
            temp_rack = Get_initialized_list_of_cells();
            temp_rack->level = next_partition_cell_level;
            temp_rack->corr_cell = next_partition_cell;
            temp_rack->next_entry = next_part_cell_list;
            next_part_cell_list = temp_rack;
         }
      }

      /* release of the level and children are needed here if the
         program has a loop to test the graph with different initial
         content */
      Restore_prev_level (cell_ptr);
   }
   else
   {
       fprintf (out, "Program halt due to a cell got partitioned ");
       fprintf (out, "into one subcell.\n");
       Myexit(1);
   }
} /* Sym_discretion */

/*********************************************************************/
```



```
/* Return a list of cells that are required to complete the orbits containing
   original_cell.
   Here the procedure assumes that  if a vertex in a cell is required to
   complete the orbit, all the vertices in that cell will be needed
   to complete that orbit.
*/


/* Input parameters:
   1) original_cell: a pointer to the input cell that needed
                     to be partitioned
   2) extended_vert: A list of vertices that need to complete the orbits
                     containing the original_cell, the list is an array.
                     An entry with a one in it means the vertex is on,
                     otherwise that entry will have a zero in it

   Output parameter:
   1) the program returns the list of cells that are
      required to complete the
      orbits containing the original cell
*/

/* debuglist = 31 */

ptr_to_list_of_cells Get_completed_orbit_cells (
                       ptr_to_cell_node original_cell,
                       ptr_to_permvect extended_vert)

{
   ptr_to_list_of_cells cell_link, temp_cell_link;
   ptr_to_cell_node current_cell;
   int vert, count = 0;

   cell_link = NULL;
   current_cell = dummy_cell->next_cell;

   while (current_cell != dummy_cell)
   {
      if (current_cell != original_cell)
      {
         vert = current_cell->representative;
         if (extended_vert[vert])
         {
            temp_cell_link = Get_initialized_list_of_cells();
            temp_cell_link->corr_cell = current_cell;
            temp_cell_link->next_entry = cell_link;
            cell_link = temp_cell_link;
            count++;
         }
      }
      current_cell = current_cell->next_cell;
   }
   if (debuglist[31])
   {
     /* Print_dummy_list (); */
     fprintf (out, "There are a total of %d cells found to ", count);
     fprintf (out, "complete the orbits.\n");
     Print_list_of_cells (cell_link);
   }
   return (cell_link);

} /* Get_completed_orbit_cells */

/*********************************************************************/

/* Function Get_first_zero_permvect will get the first zero entry in
   the permvect in the given range. -1 will be returned if there
   is no zero entry in the given range. */

/* input parameters:
   1) start_range: the starting position
   2) finish_range: the finishing position
   3) search_array: the array to be search
```

```
   output parameter:
   1) the position found; -1 means no zero entry in the given range
*/

int Get_first_zero_permvect (int start_range, int finish_range,
                             ptr_to_permvect search_array)
{
   int i;

   if (search_array == NULL)
   {
      fprintf (out, "\nAn null pointer is passed into ");
      fprintf (out, "Get_first_zero_permvect, program halt.\n");
      Myexit(1);
   }
   for (i = start_range; i <= finish_range; i++)
   {
      if (!(search_array[i]))
        /* found */
        return (i);
   }
   /* not found */
   return (-1);
} /* Get_first_zero_permvect */


/**********************************************************************/

/* The rest of the vertices in the graph is treated differently and
   a new color is assigned to them before working out the
   color wreath product group. */

/* input parameters:
   1) cycle: the array contain the orbits
   2) color: the array store the color of each vertex
   3) index: the new color going to be assigned
   4) total_ver: the total number of vertices of the graph
*/

void Assign_rest_color (ptr_to_permvect cycle,
                        ptr_to_permvect color,
                        int index, int total_ver)
{

   int fpos, spos, lpos;

   fpos = Get_first_zero_permvect (1, total_ver, color);
   if (fpos != -1)
   {
      lpos = fpos;
      color[fpos] = index;
      spos = Get_first_zero_permvect (fpos+1, total_ver, color);
      while (-1 != spos)
      {
         color[spos] = index;
         cycle[fpos] = spos;
         fpos = spos;
         spos = Get_first_zero_permvect (fpos+1, total_ver, color);
      }
      cycle[fpos] = lpos;
   }
} /* Assign_rest_color */

/**********************************************************************/

/* To create the color wreath product group of a list of cells and
   the cell pointed by c_ptr. The cell pointed by c_ptr has a color
   different from all other cells from the list and cells with the
   same size and content has the same colors and the rest of the
   vertices from the graph form a cell with a different color. */

/* Input parameters:
   1) outgp: the preallocated color wreath product group
   2) c_ptr: the cell to be partitioned
   3) cell_link: the list of cells to complete the orbits
```



```
   4) total_ver: the total number of vertices of all the cells

   Output parameter
   1) the color wreath product group will be returned
*/

/* debuglist = 30 */

ptr_to_perm_gp Create_cwpg_for_cell_stab (ptr_to_perm_gp outgp,
                                          ptr_to_cell_node c_ptr,
                                          ptr_to_list_of_cells cell_link,
                                          int total_ver)
{
   extern ptr_to_permvect cwpgcycle;
   extern ptr_to_permvect cwpgcolor;
   extern int **colormat;

   /* ptr_to_perm_gp outgp; */
   ptr_to_cell_node tcell;
   ptr_to_list_of_cells tlist; /* to go through the list */
   unsigned *cell_vset;
   int i, j, colorindex = 1,
       processed_input_cell = FALSE;

   for (i = 0; i <= total_ver; i++)
     cwpgcycle[i] = cwpgcolor[i] = 0;

   for (i = 1; i <= total_ver; i++)
   {
      for (j = 0; j <= total_content; j++)
         colormat[i][j] = 0;
   }

   tlist = cell_link;
   while ((tlist != NULL) || (!processed_input_cell))
   {
      int tempval, rep, npos;

      if (tlist != NULL)
      {
         tcell = tlist->corr_cell;
         tempval = colormat[tcell->cell_size] [tcell->actual_content];
         if (0 == tempval)
         {
            tempval = colormat[tcell->cell_size] [tcell->actual_content]
                    = colorindex;
            colorindex++;
         } /* end if */
         tlist = tlist->next_entry;
      }
      else /* the input cell is treated differently, a new color is assigned
              to it */
      {
         tempval = colorindex;
         colorindex++;
         tcell = c_ptr;
         processed_input_cell = TRUE;
      }
      /* set up one cell */
      /* here we assume the cell always have at least one vertex */
      npos = rep = tcell->representative;
      do
      {
         cwpgcycle[npos] = allcells[npos];
         cwpgcolor[npos] = tempval;
         npos = allcells[npos];
      } while (rep != npos);
   }

   /* the rest of the vertices in the graph is treated differently and
      a new color is assigned to them */

   Assign_rest_color (cwpgcycle, cwpgcolor, colorindex, total_ver);
   colorindex++;
```

```
   outgp = color_wreath_prod_gp(outgp, cwpgcycle, cwpgcolor, total_ver);

   if (debuglist[30])
   {
      fprintf (out, "\nThe cwpgcycle with size %d is:\n", total_ver);
      printpvect (out, cwpgcycle, total_ver);
      fprintf (out, "The cwpgcolor with size %d is:\n", total_ver);
      printpvect (out, cwpgcolor, total_ver);
      fprintf (out, "\nThe color wreath product group for ");
      fprintf (out, "this level is:\n");
      Print_group_information (outgp);
   }

   /* release the memory of the colormat */

   return (outgp);

} /* Create_cwpg_for_cell_stab */

/*****************************************************************/

/* To generate the cell stabilizer of a cell based on the autogp. */

/* Generate the partition stabilizer. */
/* Input parameters:
   1) c_ptr: the input cell
   2) cell_link: the list of cells to complete the orbits
   3) partstab: the partition stablizer of the latest level

   Output parameter:
   return the cell stabilizer
   outgp will be allocated here
*/

/* debuglist is 29 */

ptr_to_perm_gp Generate_cell_stabilizer (ptr_to_cell_node c_ptr,
                                         ptr_to_list_of_cells cell_link,
                                         ptr_to_perm_gp partstab)

{
   static ptr_to_perm_gp cwpgp = NULL;

   ptr_to_perm_gp outgp = NULL;

   cwpgp = Create_cwpg_for_cell_stab (cwpgp, c_ptr, cell_link,
             total_vertices);
   if (debuglist[29])
   {
      fprintf (out, "\nThe auto group %x that intersect with ",
               auto_gp);
      fprintf (out, "the cwpgp %x in Generate_cell_stabilizer is:",
               cwpgp);
      Print_group_information (auto_gp);
   }

   outgp = gp_inter (partstab, cwpgp, NULL);

   if (debuglist[29])
   {
     fprintf(out, "\nThe color wreath produce group intersects ");
     fprintf(out, "with the partition stablizer (cell stablizer) is:\n");
     Print_group_information (outgp);
   }

   /* No need to disposepg(cwpgp), Create_cwpg_for_cell_stab
      will handle it. */

   return (outgp);

} /* Generate_cell_stabilizer */

/*****************************************************************/

/* To free memory in the list of cells. */
```



```
/* Input parameters:
   1) c_list: the list of cells
*/

void Release_list_of_cells (ptr_to_list_of_cells c_list)

{
  ptr_to_list_of_cells list_entry, next_list_entry;

  list_entry = c_list;

  while (list_entry != NULL)
  {
     next_list_entry = list_entry->next_entry;
     Free_list_of_cells (list_entry);
     list_entry = next_list_entry;
  }

} /* Release_list_of_cells */

/*****************************************************************/

/* If necessary, extend the cell so that it forms a complete orbit.
   Find the cell stablizer of the completed cell.
*/

/* Input parameter:
   1) cell_ptr: pointer to the selected cell which will be partitioned
   2) pvec_ptr: an array that contain all the vertices in cell_ptr, if
               the array is not full, it should end with a zero entry
   3) autogp: the automorphism group to be used

   Output parameters:
   1) The group that is truncated to the vertices of the cell passed in
      using autogp is returned
   2) Due to the use fo truncate_gp, pvec_ptr will be changed accordingly
*/

/* debuglist = 26 */

ptr_to_perm_gp Extend_cell (ptr_to_cell_node cell_ptr,
                            ptr_to_permvect pvec_ptr,
                            ptr_to_perm_gp autogp)

{
   static ptr_to_permvect tempvec_ptr = NULL;
   static ptr_to_permvect final_orb_vert = NULL;

   ptr_to_list_of_cells comp_orb_cell_link;
   ptr_to_perm_gp tgp_ptr = NULL;
   int i, final_orb_size;

   if ((tempvec_ptr == NULL) || reset_tempvec_ptr)
   {
      if (tempvec_ptr != NULL)
      {
         reset_tempvec_ptr = FALSE;
         disposepv (tempvec_ptr);
         tempvec_ptr = NULL;
         disposepv (final_orb_vert);
         final_orb_vert = NULL;
      }
      tempvec_ptr = allocatepv (total_vertices+1);
      final_orb_vert = allocatepv (total_vertices);
   }
   copy_pv(tempvec_ptr, pvec_ptr, total_vertices+1);
   tempvec_ptr[0] = pvec_ptr[0];

   if (autogp->permsize < total_vertices)
   {
      for (i = autogp->permsize + 1; i <= total_vertices; i++)
        final_orb_vert[i] = 0;
   }
```

```
   final_orb_size = complete_orbits (autogp, tempvec_ptr,
                                     final_orb_vert, TRUE);
   if (debuglist[26])
   {
      fprintf (out, "\nComplete orbit size is: %d.\n", final_orb_size);
      fprintf (out, "The completed orbit(s)' vertices are:\n");
      printpvect (out, final_orb_vert, total_vertices);
   }
   if (final_orb_size == cell_ptr->cell_size)
   {
     /* the cell form one or more orbits by itself */
     tgp_ptr = copy_gp(autogp, tgp_ptr);
     if (debuglist[26])
     {
        fprintf (out, "\ntgp_ptr in Extend_cell is allocated by ");
        fprintf (out, "copy_gp from autogp\n");
     }
   }
   else /* the input cell has incomplete orbit */
   {
      if (debuglist[26])
      {
         fprintf(out, "\nThe input cell do not form a complete orbit, ");
         fprintf(out, "its cell stablizer will be generated.\n");
      }
      /* mask the cell's vertices in final_orb_vert */
      i = 1;
      while ((pvec_ptr[i] != 0) && (i <= cell_ptr->cell_size))
      {
         final_orb_vert[pvec_ptr[i]] = 0;
         i++;
      }
      if (debuglist[26])
      {
         fprintf (out, "The completed orbit(s)' vertices - ");
         fprintf (out, "cell_ptr's vertices are:\n");
         printpvect (out, final_orb_vert, total_vertices);
      }
      comp_orb_cell_link = Get_completed_orbit_cells (
                              cell_ptr, final_orb_vert);
      tgp_ptr = Generate_cell_stabilizer(cell_ptr,
                                          comp_orb_cell_link,
                                          autogp);
      Release_list_of_cells (comp_orb_cell_link);
   }
   truncate_gp(tgp_ptr, pvec_ptr, FALSE);

   return (tgp_ptr);
} /* Extend_cell */

/***********************************************************************/

/* Put all the vertices in the cell into an array.

   input parameter
   1) c_ptr: a pointer to the cell to be transform

   output parameter
   1) the program will return the transformed permvect
   2) the allocation of the permvect is done in this procedure
*/

/* debuglist is 15 */

ptr_to_permvect Transform_cell_to_permvect (ptr_to_cell_node c_ptr)

{
  int position, i, rep;
  ptr_to_permvect trunc_set;

  trunc_set = allocatepv (total_vertices+1);
  trunc_set[0] = 0;
  if (c_ptr->cell_size == total_vertices)
  {
     for (i = 1; i <= c_ptr->cell_size; i++)
```



```
        trunc_set[i] = i;
     trunc_set[total_vertices+1] = 0;
     if (debuglist[15])
     {
        fprintf (out, "\nTruncate group is not necessary since cell's ");
        fprintf (out, "size is equal to the graph's size.\n");
     }
  }
  else
  {
     position = rep = c_ptr->representative;
     i = 1;
     do
     {
       trunc_set[i] = position;
       position = allcells[position];
       i++;
     } while (position != rep);
     trunc_set[i] = 0;
  }

  if (debuglist[15])
  {
     fprintf (out, "\nThe premvect transformed from cell is:\n");
     printpvect (out, trunc_set, total_vertices);
  }

  return(trunc_set);

} /* Transform_cell_to_permvect */

/***********************************************************************/

/* To test if the size of a group is equal to one (identity). */

/* Input parameter
   1) ingp: the group to be tested
*/

/* debuglist = 25 */

int Identity (ptr_to_perm_gp ingp)

{
   double approx_ord;
   long actual_ord;

   gporder(ingp, &approx_ord, &actual_ord);
   if (actual_ord == 1)
   {
      if (debuglist[25])
      {
        fprintf (out, "Identity group is found.\n");
        Print_group_information (ingp);
      }
      return (TRUE);
   }
   else
   {
      if (debuglist[25])
      {
         fprintf (out, "\nNon-identity group is found\n");
         Print_group_information (ingp);
      }
      return (FALSE);
   }

} /* Identity */

/***********************************************************************/

/* Select a cell whose truncated group is non-trivial.

   Input parameters
   1) last_selected_cell: the last cell selected which leads to an
```

```
      identity group after truncation
   2) origin_gp: the input automorphism group

   Output parameters
   1) *ptr_result_gp: the truncated group, this should be an identity
   2) the cell which has a nontrivial truncate group, the procedure will
      return NULL if no such cell is found
   3) *ptr_temp_permvec: the newly allocated permvect transformed from the
                        selected cell
*/

/* debuglist = 23 */

ptr_to_cell_node Select_cell_with_nontrivial_gp (
                   ptr_to_cell_node last_selected_cell,
                   ptr_to_perm_gp origin_gp,
                   ptr_to_perm_gp *ptr_result_gp,
                   ptr_to_permvect *ptr_temp_permvec)

{
   ptr_to_cell_node best_cell;
   int not_found = TRUE;

   if (*ptr_result_gp == NULL)
   {
      fprintf (out, "Result_gp must be allocated.\n");
      Myexit(1);
   }
   best_cell = NULL;

   best_cell = dummy_cell->next_cell;
   while ((best_cell != dummy_cell) && (not_found))
   {
      if (!((best_cell->cell_size == 1) ||
            (best_cell->actual_content ==
             best_cell->cell_size) ||
            (best_cell->actual_content == 0) ||
            (best_cell == last_selected_cell)))
      /* test if this is a qualify cell */
      {
         if (*ptr_temp_permvec != NULL)
           disposepv (*ptr_temp_permvec);
         *ptr_temp_permvec = Transform_cell_to_permvect (best_cell);
         /* Quicksort(*ptr_temp_permvec, best_cell->cell_size + 1); */
         disposepg (*ptr_result_gp);
         *ptr_result_gp = Extend_cell (best_cell, *ptr_temp_permvec,
                                       origin_gp);
         if (!Identity (*ptr_result_gp))
         {
            not_found = FALSE;
            break;
         }
      }
      best_cell = best_cell->next_cell;
   }

   if (debuglist[23])
   {
      fprintf (out, "The subcell with non-trivial group selected is:\n");
      if (best_cell == dummy_cell)
      {
        fprintf (out, "dummy cell: ");
      }
      Print_cell (best_cell);
   }

   if (not_found)
     return (NULL);
   else
     return (best_cell);

} /* Select_cell_with_nontrivial_gp */

/*****************************************************************/
```



```
/* To make sure the cell passed for symmetry partition is not identity. */

/* Input parameter;
   1) current_level: the latest level created
   2) autogp: the automorphism group to be used
   3) tgp_ptr: a pointer to the group pointer of the extended cell
   4) ptr_pvec_ptr: a pointer to an array that contain all the
             vertices in cell_ptr, if
             the array is not full, it should end with a zero entry
   5) ptr_cell_ptr: a pointer to the pointer of the  selected cell which
             will be partitioned
*/

/* debuglist = 26 */

int Find_non_identity_cell (int current_level,
                            ptr_to_perm_gp autogp,
                            ptr_to_perm_gp *ptr_tgp_ptr,
                            ptr_to_permvect *ptr_pvec_ptr,
                            ptr_to_cell_node *ptr_cell_ptr)
{
   ptr_to_cell_node selected_cell;
   int cell_is_good = FALSE;

   if (!Identity (*ptr_tgp_ptr))
     cell_is_good = TRUE;
   else /* after the group was truncated according to the input cell,
           the truncated group is trivial */
   {
      if (debuglist[26])
      {
         fprintf (out, "\nSymmetry partition: After truncation, ");
         fprintf (out, "group is trivial, another cell has to be ");
         fprintf (out, "selected.\n");
      }
      /* Search for a cell which do not give a identity group
         after truncation. */
      /* N.B. Here ptr_tgp_ptr should be allocated */
      selected_cell = Select_cell_with_nontrivial_gp
               (*ptr_cell_ptr, autogp, ptr_tgp_ptr, ptr_pvec_ptr);
      if (NULL != selected_cell)
      {
         cell_is_good = TRUE;
         *ptr_cell_ptr = selected_cell;
      }
      /* else cell_is_good should be equal to FALSE, which was initialized
            in the beginning of this procedure */
   }
   return (cell_is_good);
} /* Find_non_identity_cell */

/*****************************************************************/

/* Print the cycle form of an array given the first position. */

/* Input parameters:
   1) vector: the array which is in cycle form
   2) fpos: the starting position
*/

void Print_cycle (ptr_to_permvect vector, int fpos)
{
   int cpos;

   cpos = fpos;
   fprintf (out, "\nThe following vertices from a cycle.\n");
   do {
      fprintf (out, "%d, ", cpos);
      cpos = vector[cpos];
   } while (cpos != fpos);
   fprintf (out, "\n");
} /* Print_cycle */

/*****************************************************************/
```

```c
/*
    This procedure combines all the orbit in the orb_list and
    put it in its cycle form in the array destination. Since, each
    node in the orbit contain only the orbit representative of
    that orbit, to find the whole orbit, it needs the array reference.

    Input parameter
    1) destination: the cycle form of the combined orbit
    2) reference: contain the cycle form of each orbit
    3) orb_list: a link list of the representative of each orbit

    Side effect:
    1) Returns the representative of the combined orbit.
    2) All nodes in orb_list will be free.

    N.B. orb_list cannot be NULL
*/

int Built_new_orbit (ptr_to_permvect destination,
                     ptr_to_permvect reference,
                     ptr_to_general_node orb_list)
{
   ptr_to_general_node temp_ptr, last_ptr;
   int min_rep, fpos, tfpos, cpos, npos;

   /* set up the first orbit */
   destination[0]++;
   temp_ptr = orb_list;
   fpos = tfpos = cpos = min_rep = orb_list->value;
   npos = reference[cpos];

   do {
     while (tfpos != npos)
     {
        destination[cpos] = npos;
        cpos = npos;
        npos = reference[cpos];
     }
     last_ptr = temp_ptr;
     temp_ptr = temp_ptr->next;
     free ((void *) last_ptr);
     if (temp_ptr != NULL)
     {
        cpos = tfpos = destination[cpos] = temp_ptr->value;
        npos = reference[cpos];
        if (min_rep > tfpos)
          min_rep = tfpos;
     }
   } while (temp_ptr != NULL);
   destination[cpos] = fpos;
   return (min_rep);

} /* Built_new_orbit */

/**********************************************************************/

/* This procedure will try to look for the neighbor orbits of an orbit. */

/* Input parameter
   1) selectrep: the selected orbit.
   2) free_orb_list: the list of all the free orbit
   3) worbit: the new set of orbits, it is used to store the
             combined orbit
   4) torbit: the original orbits, it is used for reference
   5) map: the mapping of the current vertex label to the label in
          the original graph

   Side effect:
   1) the orbit of maxorbrep will be added into worbit
   2) the orbit of maxorbrep will be taken away and free from the
      free_orb_list
   3) the neighbor list will be returned
*/

ptr_to_general_node Find_distance_orbits (int selectrep,
```



```c
          ptr_to_general_node free_orb_list,
          ptr_to_permvect worbit,
          ptr_to_permvect torbit,
          ptr_to_permvect map)
{
   ptr_to_general_node temp_orb_ptr, next_orb_ptr, neigh_list = NULL;
   int cpos, norep, sover;

   /* find all the other cells that have a distance one */
   temp_orb_ptr = free_orb_list->next;
   while (temp_orb_ptr != free_orb_list)
   {
      cpos = norep = temp_orb_ptr->value;
      do {
        /* check if this vertices is an neighbor of the
           selected orbit */
        sover = selectrep;
        do {
          if (adj_matrix[map[sover]][map[cpos]])
          /* is an neighbor */
          {
             /* remove the cell and put it to the neighbor list */
             temp_orb_ptr->last->next = temp_orb_ptr->next;
             temp_orb_ptr->next->last = temp_orb_ptr->last;
             next_orb_ptr = temp_orb_ptr->next;
             free_orb_list->value--;
             temp_orb_ptr->next = neigh_list;
             neigh_list = temp_orb_ptr;
             temp_orb_ptr = next_orb_ptr;
             goto try_next_free_orb;
          }
          /* N.B. worbit has to be used */
          sover = worbit[sover];
        } while (sover != selectrep);
        cpos = torbit[cpos];
      } while (cpos != norep);
      temp_orb_ptr = temp_orb_ptr->next;
      try_next_free_orb:;
   }
   return (neigh_list);

} /* Find_distance_orbits */

/**********************************************************************/

/* This procedure will write an orbit to worbit and free it from
   free_orb_list. */

/* Input parameter:
   1) orbrep: the representative of the orbit that will be written to
              worbit
   2) orb_ptr: the orbit that will be free from free_orb_list
   3) free_orb_list: the list of cells that are not written in
                     worbit
   4) worbit: the new set of orbits, it is used to store the
              combined orbit
   5) torbit: the original orbits, it is used for reference

   Side effect:
   worbit and free_orb_list will be changed
*/

void Update_worbit (int orbrep, ptr_to_general_node orb_ptr,
                    ptr_to_general_node free_orb_list,
                    ptr_to_permvect worbit, ptr_to_permvect torbit)
{
   int cpos, npos;

   /* partition the max orbit */
   /* write back the maximum orbit */
   worbit[0]++;
   cpos = orbrep;
   npos = worbit[cpos] = torbit[cpos];
   while (orbrep != npos)
   {
```

```
      cpos = npos;
      npos = worbit[cpos] = torbit[cpos];
   }

   /* remove the biggest cell from the link list */
   orb_ptr->last->next = orb_ptr->next;
   orb_ptr->next->last = orb_ptr->last;
   free_orb_list->value--;
   free ((void *) orb_ptr);

} /* Update_worbit */

/*************************************************************************/

/* Combine orbits according to the distance.

   Input parameter
   1) worbit: the working orbit
   2) map: the mapping of the current vertex label to the label in
           the original graph
   3) osize: the size of the working orbit

   Side effect
   The working orbit will be changed.
   worbit[0] contains the total number of new orbits
   more_than_one_orbs may be changed.
*/

void Combine_by_distance (ptr_to_permvect worbit,
                          ptr_to_permvect map, int osize)
{
   /* static variables */
   static ptr_to_general_node free_orb_list = NULL;
   static ptr_to_permvect torbit;

   int i, maxsize = 0, trep, cpos, npos, tsize, selectrep;
   ptr_to_general_node temp_orb_ptr, max_orb_ptr,
                       neigh_list = NULL;

   if ((NULL == torbit) || reset_torbit)
   {
      if (torbit != NULL)
      {
         reset_torbit = FALSE;
         disposepv (torbit);
         torbit = NULL;
      }
      torbit = allocatepv (total_vertices);
   }

   if (free_orb_list == NULL)
   /* create the header node */
   {
      free_orb_list = (ptr_to_general_node) malloc
                      (sizeof (struct general_node));
      free_orb_list->size = 0;
   }
   free_orb_list->next = free_orb_list->last = free_orb_list;
   free_orb_list->value = 0;

   temp_orb_ptr = max_orb_ptr = NULL;

   /* create a copy of the working orbit */
   copy_pv(torbit, worbit, osize);

   /* Search for the biggest orbit and build the cell list. */
   for (i=1; i<=osize; i++)
   {
      if (worbit[i] != 0) /* find an orbit */
      {
         /* add this orbit to the free_orb_list */
         temp_orb_ptr = (ptr_to_general_node) malloc
                        (sizeof (struct general_node));
         temp_orb_ptr->value = i;
         free_orb_list->next->last = temp_orb_ptr;
```



```
         temp_orb_ptr->next = free_orb_list->next;
         temp_orb_ptr->last = free_orb_list;
         free_orb_list->next = temp_orb_ptr;
         free_orb_list->value++;
         cpos = trep = i;
         npos = worbit[cpos];
         tsize = 1;
         worbit[cpos] = 0;
         while (trep != npos)
         {
            cpos = npos;
            npos = worbit[cpos];
            tsize++;
            worbit[cpos] = 0;
         }
         temp_orb_ptr->size = tsize;
         if (maxsize < tsize)
         {
            max_orb_ptr = temp_orb_ptr;
            maxsize = tsize;
         }
      }
   }

   /* partition the max orbit */
   /* write back the maximum orbit */
   worbit[0] = 0;
   selectrep = max_orb_ptr->value;
   Update_worbit (selectrep, max_orb_ptr, free_orb_list,
                  worbit, torbit);

   /* find all the other cells according to distance */
   while (free_orb_list->value != 0)
   {
      neigh_list = Find_distance_orbits (selectrep,
                                         free_orb_list,
                                         worbit,
                                         torbit, map);
      if (neigh_list != NULL)
      {
         if (neigh_list->next != NULL)
         {
            /* more than one orbit is combined */
            /* NB the next statement cannot put outside the if
                  statement because neigh_list will be changed in
                  Built_new_orbit */
            selectrep = Built_new_orbit (worbit, torbit, neigh_list);
            /* indicate that this cell has more than one orbit */
            more_than_one_orbs[selectrep] = TRUE;
         }
         else
         {
            selectrep = Built_new_orbit (worbit, torbit, neigh_list);
         }
         neigh_list = NULL;
      }
      else /* select another cell as the maximum orbit */
      {
         temp_orb_ptr = free_orb_list->next;
         maxsize = 0;
         max_orb_ptr = NULL;
         while (temp_orb_ptr != free_orb_list)
         {
            if (temp_orb_ptr->size > maxsize)
            {
               maxsize = temp_orb_ptr->size;
               max_orb_ptr = temp_orb_ptr;
            }
            temp_orb_ptr = temp_orb_ptr->next;
         }
         selectrep = max_orb_ptr->value;
         Update_worbit (selectrep, max_orb_ptr, free_orb_list,
                        worbit, torbit);
      }
   }
```

```
   disposepv (torbit);
   torbit = NULL;

} /* Combine_by_distance */

/*********************************************************************/

/* Given an array of orbits in their cycle form, the program will find
   the largest orbit and combine the rest of the orbits into one.
   In case there is more than one largest orbit, it will take the
   first one as the largest.

   Input parameter
   1) worbit: the working orbit
   2) osize: the size of the working orbit

   Side effect
   The working orbit will be changed to the combined orbit.
   The smallest vertex in the combined orbit will be returned.
   This is used to force the next partition cell to be the combined
   orbit.

   N.B. Here we assume we  will choose the smallest vertex
        in a cell to be the cell representative.
*/

void Split_into_two_orbits (ptr_to_permvect worbit, int osize)
{
   ptr_to_permvect torbit;
   int i, maxsize = 0, maxorbrep = 0,
       trep, cpos, npos, tsize, firstzero, lastzero;

  /* create a copy of the working orbit */
  torbit = allocatepv(osize);
  copy_pv(torbit, worbit, osize);

  /* Search for the biggest orbit */
  for (i=1; i<=osize; i++)
  {
     if (worbit[i] != 0) /* find an orbit */
     {
        cpos = trep = i;
        npos = worbit[cpos];
        tsize = 1;
        worbit[cpos] = 0;
        while (trep != npos)
        {
           cpos = npos;
           npos = worbit[cpos];
           tsize++;
           worbit[cpos] = 0;
        }
        if (maxsize < tsize)
        {
           maxsize = tsize;
           maxorbrep = trep;
        }
     }
  }

  /* write back the maximum orbit */
  cpos = maxorbrep;
  npos = worbit[cpos] = torbit[cpos];
  while (maxorbrep != npos)
  {
     cpos = npos;
     npos = worbit[cpos] = torbit[cpos];
  }

  /* combine all other orbits */
  cpos = 1;
  while ((worbit[cpos] != 0) && (cpos <= osize))
    cpos++;
  if (cpos > osize)
```



```
  {
     fprintf (out, "\nOne complete orbit was past into Combine_orbit, ");
     fprintf (out, "program cannot go furter.\n");
     Myexit(1);
  }
  else
  {
     firstzero = lastzero = cpos;
     for (i = firstzero + 1; i <= osize; i++)
     {
        if (worbit[i] == 0)
        {
           worbit[lastzero] = i;
           lastzero = i;
        }
     }
     worbit[lastzero] = firstzero;
  }

  disposepv (torbit);
  torbit = NULL;

  if (MAX_CHILDREN > 2)
    more_than_one_orbs[firstzero] = TRUE;

} /* Split_into_two_orbits */

/*********************************************************************/

/* To sort a cell's circular link list in allcells. */

/* input parameter
   1) in_cell: an array contain the circular list

   side effect
   the cell's circular link list in allcells will be sorted and
   the vertex before rep. in the cell node will be updated accordingly
*/

void Sort_cell_vertices (ptr_to_cell_node in_cell)
{
   static int *vert_set = NULL;
   int i, pos, cellsize, biggest_ver, cellrep;

   if ((NULL == vert_set) || reset_vert_set)
   {
      if (vert_set != NULL)
      {
         reset_vert_set = FALSE;
         free ((void *) vert_set);
         vert_set = NULL;
      }
      vert_set = (int *) calloc (max_level, sizeof (int));
   }

   cellrep = in_cell->representative;
   cellsize = in_cell->cell_size;
   pos = cellrep;
   i = 0;
   do {
     vert_set[i] = pos;
     i++;
     pos = allcells[pos];
   } while (pos != cellrep);
   Quicksort (vert_set, cellsize);
   biggest_ver = vert_set[cellsize-1];
   pos = vert_set[0];
   for (i = 1; i < cellsize; i++)
   {
      allcells[pos] = vert_set[i];
      pos = vert_set[i];
   }
   allcells[pos] = vert_set[0];
   in_cell->ver_before_rep = pos;
```

```
} /* Sort_cell_vertices */

/*********************************************************************/

/* If the total number of orbits is less than or equal to MAX_CHILDREN,
   then the parent cell (parent_ptr) will be partitioned according to
   the orbits it found. Otherwise, the biggest orbit will be partitioned
   out and the rest or the orbits will be combined into one cells
   if Split_into_two_orbits is called. If Combine_by distance is called,
   cells will be partitioned according to the distance from the
   biggest size orbits.
   Here we assume the total number of orbits are stored in
   orb_size[0].

   Input parameter
   1) parent_ptr: pointed to the parent cell node
   2) orb_ptr: pointed to the permvect representing the orbits
   3) orbs_size: the total number of vertices in all the orbits
   4) map: the mapping form the new truncated vertex labeling
         to the original vertex labeling
   5) lev: the current level

   Side effect
   1) the parent cell node will has all its children created
   2) the next_part_cell_list will be updated
*/

/* debuglist is 18 */

void Transform_orbits_to_cells (ptr_to_cell_node parent_ptr,
                                ptr_to_permvect orb_ptr,
                                int orbs_size,
                                ptr_to_permvect map,
                                int lev)

{
  ptr_to_permvect working_orbit;
  int i, vb4rep, curr_pos,
      crep, last_pos, actu_curr_pos, csize,
      next_part_rep = 0;
  ptr_to_cell_node cell_ptr, last_cell;
  ptr_to_list_of_cells temp_rack;

  if (debuglist[18])
  {
     fprintf (out, "Transform_orbits_to_cells:\n");
     fprintf (out, "Parent_ptr is:\n");
     Print_cell (parent_ptr);
     fprintf (out, "Orbit vector of size %d is:\n", orbs_size);
     printpvect (out, orb_ptr, orbs_size);
     fprintf (out, "The mapping are:\n");
     printpvect (out, map, total_vertices);
  }

  /* copy the orb_ptr */
  /* create orbit by orbit with masking */
  working_orbit = allocatepv(orbs_size);
  copy_pv(working_orbit, orb_ptr, orbs_size);

  if (orb_ptr[0] > MAX_CHILDREN)
  {
     switch (COMBINE_OPTION)
     {
        /* split each orbit into a cell */
        case NOT_COMBINE: break;

        /* split the biggest orbit into a cell, combine other orbits
           into one cell if they have the same distance from the biggest
           orbit */
        case COMBINE_REST_BY_DISTANCE: Combine_by_distance
                                        (working_orbit, map, orbs_size);
                                       break;

        /* split the biggest orbit into a cell, combine other orbits into
           one cell */
```



```
        case COMBINE_REST_TO_ONE: Split_into_two_orbits
                                   (working_orbit, orbs_size);
                                  break;

        default: fprintf(out, "Unknown COMBINE_OPTION: %d.\n",
                               COMBINE_OPTION);
                 Myexit(1);
     }
     if (debuglist[18])
     {
        fprintf(out, "\nThe number of orbits found are %d.\n", orb_ptr[0]);
        fprintf(out, "It is greater than %d.\n", MAX_CHILDREN);
        fprintf(out, "The new combined orbits are:\n");
        printpvect (out, working_orbit, orbs_size);
     }
  }

  cell_ptr = Get_initialized_cell_node();

  /* link the last block to the next cell */
  parent_ptr->next_cell->prev_cell = cell_ptr;
  cell_ptr->next_cell = parent_ptr->next_cell;

  for (i=1; i<=orbs_size; i++)
  {
     if (working_orbit[i] != 0)
     /* generate this orbit as a cell */
     {
        parent_ptr->total_children++; /* one more children */

        /* update cell entries */
        cell_ptr->parent = parent_ptr;

        csize = 0;
        vb4rep = 0;
        curr_pos = i;
        crep = map[i];

        if (more_than_one_orbs[i])
        {
           /* put this cell_ptr to the next_part_cell_list */
           temp_rack = Get_initialized_list_of_cells();
           temp_rack->level = lev;
           temp_rack->corr_cell = cell_ptr;
           temp_rack->next_entry = next_part_cell_list;
           next_part_cell_list = temp_rack;
           more_than_one_orbs[i] = FALSE;
        }

        do
        {
           last_pos = curr_pos;
           curr_pos = working_orbit[curr_pos];
           working_orbit[last_pos] = 0;
           allcells[map[last_pos]] =
             actu_curr_pos = map[curr_pos];
           csize++;
           if (crep > actu_curr_pos)
           {
              vb4rep = last_pos;
              crep = actu_curr_pos;
           }
        } while (i != curr_pos);
        cell_ptr->representative = crep;
        if (vb4rep) /* not the first vertex (!= 0)*/
          cell_ptr->ver_before_rep = map[vb4rep];
        else
          cell_ptr->ver_before_rep = map[last_pos];

        /* create the dummies */
        cell_ptr->row_first_neigh = Get_initialized_neigh_info_node();
        cell_ptr->row_first_neigh->row_cell_num = crep;
        cell_ptr->row_first_neigh->col_cell_num = 0;
        cell_ptr->col_first_neigh = Get_initialized_neigh_info_node();
        cell_ptr->col_first_neigh->row_cell_num = 0;
```

```
        cell_ptr->col_first_neigh->col_cell_num = crep;

        cell_ptr->cell_size = csize;
        cell_ptr->lower_content = 0;
        /* N.B. later the upper content should also depend on the
                actual content; upper content should be updated in
                here because it is also depend on the total_vertices */
        if (csize <= parent_ptr->actual_content)
          cell_ptr->upper_content = csize;
        else
          cell_ptr->upper_content = parent_ptr->actual_content;

        curr_pos = crep;
        /* update cell_label */
        do
        {
           cell_label[curr_pos] = crep;
           curr_pos = allcells[curr_pos];
        } while (curr_pos != crep);

        cell_list[crep] = cell_ptr;

        cell_ptr->sibling = parent_ptr->first_child;
        last_cell = parent_ptr->first_child = cell_ptr;
        Sort_cell_vertices (cell_ptr);

        cell_ptr = Get_initialized_cell_node();
        cell_ptr->next_cell = last_cell;
        last_cell->prev_cell = cell_ptr;
     }
  }

  parent_ptr->prev_cell->next_cell = last_cell;
  last_cell->prev_cell = parent_ptr->prev_cell;
  dummy_cell->total_children += (parent_ptr->total_children - 1);
  /* free the extra cell */
  Free_cell_node (cell_ptr);
  cell_ptr = NULL;

  if (debuglist[18])
  {
     fprintf (out, "The cells created based on the orbit:\n");
     Print_children(parent_ptr);
     fprintf (out, "The actual labeling relative to the original ");
     fprintf (out, "cell are:\n");
     printpvect (out, map, orbs_size);
     Print_list_of_cells (next_part_cell_list);
  }
  disposepv (working_orbit);
  working_orbit = NULL;

} /* Transform_orbits_to_cells */

/********************************************************************/

/* Remove those list_of_cells that has the same level number as
   level_num. */

void Remove_list_of_cells (level_num)
{
   ptr_to_list_of_cells last_ptr, curr_ptr;

   while ((next_part_cell_list != NULL) &&
          (next_part_cell_list->level == level_num))
   {
      curr_ptr = next_part_cell_list;
      next_part_cell_list = next_part_cell_list->next_entry;
      Free_list_of_cells (curr_ptr);
   }
   if (next_part_cell_list != NULL)
   {
      last_ptr = next_part_cell_list;
      curr_ptr = last_ptr->next_entry;
      while (curr_ptr != NULL)
      {
```



```
         if (curr_ptr->level == level_num)
         {
            last_ptr->next_entry = curr_ptr->next_entry;
            Free_list_of_cells (curr_ptr);
            curr_ptr = last_ptr->next_entry;
         }
         else
         {
            last_ptr = curr_ptr;
            curr_ptr = curr_ptr->next_entry;
          }
      }
   }
} /* Remove_list_of_cells */

/********************************************************************/

/* Not transitive, partition cell according to their orbits. */

/* Input parameters
   1) current_level: the latest level created
   2) given_ver: the size of the cell pointed by cell_ptr
   3) cell_ptr: pointer to the selected cell which will be partitioned
   4) orbvec: an array that contain all the vertices in the orbits
   5) pvec_ptr: an array that contain all the vertices in cell_ptr, if
            the array is not full, it should end with a zero entry
   6) autogp: the automorphism group to be used
*/

void Handle_non_transitivity (int current_level,
                              int given_ver,
                              ptr_to_cell_node cell_ptr,
                              ptr_to_permvect orbvec,
                              ptr_to_permvect pvec_ptr,
                              ptr_to_perm_gp autogp)
{
   int next_partition_cell_level = 0;
   ptr_to_cell_node next_partition_cell;
   ptr_to_cell_node temp_ptr;
   ptr_to_list_of_cells temp_rack;

   /* partition the orbits into subcells */
   /* next_part_cell_list is updated */
   Transform_orbits_to_cells (cell_ptr, orbvec,
                              given_ver, pvec_ptr, current_level);
   /* take care the next partition cell */
   if (next_part_cell_list != NULL)
   {
      next_partition_cell = next_part_cell_list->corr_cell;
      temp_rack = next_part_cell_list;
      next_partition_cell_level = next_part_cell_list->level;
      next_part_cell_list = next_part_cell_list->next_entry;
      Free_list_of_cells (temp_rack);
      temp_rack = NULL;
   }
   else
     next_partition_cell = NULL;

   if (debuglist[18])
   {
      fprintf(out, "\nThe pre-selected partition cell is %x;\n",
                   next_partition_cell);
   }

   if (cell_ptr->total_children > 1)
   /* this part should always be true because
      the group is not an identity */
   /* the cell is partitioned successfully */
   {
      Create_influ_func_info (cell_ptr, cell_ptr->first_child);
      Handle_new_level (cell_ptr);
      hfcwtest[current_level]++;

      if (!(finish_computation) && (Coverage_and_wastage_test(cell_ptr)))
      {
```

```
        afcwtest[current_level]++;
        /* Is it better to select a cell from the latest_level
           if such is the case, the autogp should map to the appropriate
           preimage */
        Start_gen (current_level,
                   cell_ptr, autogp, NULL, NULL, NULL, ORBIT_PARTITION,
                   next_partition_cell);
      }
      Restore_prev_level (cell_ptr);
    }
    else
    {
      fprintf (out, "Program halt due to a cell with more than one ");
      fprintf (out, "orbits got partitioned into one subcell.\n");
      Myexit(1);
    }
    if (next_partition_cell != NULL)
    /* put it back to the list */
    {
      temp_rack = Get_initialized_list_of_cells();
      temp_rack->level = next_partition_cell_level;
      temp_rack->corr_cell = next_partition_cell;
      temp_rack->next_entry = next_part_cell_list;
      next_part_cell_list = temp_rack;
    }
    Remove_list_of_cells (current_level);
} /* Handle_non_transitivity */

/********************************************************************/

/* Transform each block into a cell. The parent cell (parent_ptr)
   will be partitioned according to the block.

   Input parameter
   1) parent_ptr: pointed to the parent cell node
   2) blk_ptr: pointed to the permvect that representing the blocks
   3) blks_size: the total number of vertices in all the blocks
   4) map: the mapping form the new truncated vertex labeling
           to the original vertex labeling
   5) block_rep_map: entry 0 contains the total number of blocks,
                     the other entry contain the actual representative
                     of each block

   Side effect
   1) the parent cell node will has all its children are created
*/
/* This procedure also
   need the relabel mapping, the size of all the blocks and total
   number of blocks. */

/* debuglist is 19 */

void Transform_blocks_to_cells (ptr_to_cell_node parent_ptr,
                                 ptr_to_permvect blk_ptr,
                                 int blks_size,
                                 ptr_to_permvect map,
                                 ptr_to_permvect block_rep_map)

{
  int i, crep, first_pos, curr_pos, last_pos,
      actu_curr_pos, vb4rep, csize;
  ptr_to_cell_node cell_ptr, last_cell;

  /* create block by block starting from the last block */
  /* each block should have the same size */
  csize = blks_size / block_rep_map[0];
  cell_ptr = Get_initialized_cell_node();

  /* link the last block to the next cell */
  parent_ptr->next_cell->prev_cell = cell_ptr;
  cell_ptr->next_cell = parent_ptr->next_cell;

  for (i = block_rep_map[0]; i >= 1; i--)
  {
    parent_ptr->total_children++; /* one more children */
```



```
    /* update cell entries */
    cell_ptr->parent = parent_ptr;

    /* generate this block as a cell */
    /* update allcells, ver_before_rep and representative. */
    vb4rep = 0;
    first_pos = curr_pos = block_rep_map[i];
    crep = map[first_pos];
    do
    {
      last_pos = curr_pos;
      curr_pos = blk_ptr[curr_pos];
      allcells[map[last_pos]] =
        actu_curr_pos = map[curr_pos];
      if (crep > actu_curr_pos)
      {
        vb4rep = last_pos;
        crep = actu_curr_pos;
      }
    } while (first_pos != curr_pos);

    cell_ptr->representative = crep;
    if (vb4rep) /* not the first vertex (!= 0)*/
      cell_ptr->ver_before_rep = map[vb4rep];
    else
      cell_ptr->ver_before_rep = map[last_pos];

    /* create the dummies */
    cell_ptr->row_first_neigh = Get_initialized_neigh_info_node();
    cell_ptr->row_first_neigh->row_cell_num = crep;
    cell_ptr->row_first_neigh->col_cell_num = 0;
    cell_ptr->col_first_neigh = Get_initialized_neigh_info_node();
    cell_ptr->col_first_neigh->row_cell_num = 0;
    cell_ptr->col_first_neigh->col_cell_num = crep;

    cell_ptr->cell_size = csize;
    cell_ptr->lower_content = 0;
    /* N.B. The upper content also depends on the
            actual content; upper content should be updated in
            here because it also depends on the total_vertices. */
    if (csize <= parent_ptr->actual_content)
      cell_ptr->upper_content = csize;
    else
      cell_ptr->upper_content = parent_ptr->actual_content;

    curr_pos = crep;
    /* update cell_label */
    do
    {
      cell_label[curr_pos] = crep;
      curr_pos = allcells[curr_pos];
    } while (curr_pos != crep);

    cell_list[crep] = cell_ptr;

    cell_ptr->sibling = parent_ptr->first_child;
    last_cell = parent_ptr->first_child = cell_ptr;
    Sort_cell_vertices (cell_ptr);

    cell_ptr = Get_initialized_cell_node();
    cell_ptr->next_cell = last_cell;
    last_cell->prev_cell = cell_ptr;
  }
  parent_ptr->prev_cell->next_cell = last_cell;
  last_cell->prev_cell = parent_ptr->prev_cell;
  dummy_cell->total_children += (parent_ptr->total_children - 1);
  /* free the extra cell */
  Free_cell_node (cell_ptr);
  cell_ptr = NULL;
  if (debuglist[19])
  {
    fprintf (out, "The %d block(s) with size blks_size is/are:\n",
             block_rep_map[0], blks_size);
    printpvect (out, blk_ptr, blks_size);
```

```
    fprintf (out, "The actual labeling relative to the original ");
    fprintf (out, "cell are:\n");
    printpvect (out, map, blks_size);
    fprintf (out, "The cells created based on the blocks are:\n");
    Print_children(parent_ptr);
  }

} /* Transform_blocks_to_cells */

/*********************************************************************/

/* Handle the situation when cell is transitive but not primitive. */

/* Input parameters
   1) effective_vertices: the size pointed by cell_ptr
   2) current_level: the latest level created
   3) total_blocks: the total number of blocks
   4) cell_ptr: pointer to the selected cell which will be partitioned
   5) block_map_form: the rep. of a point in a block
   6) block_link_form: the contents of blocks
   7) pvec_ptr: an array that contains all the vertices in cell_ptr, if
                the array is not full, it should end with a zero entry
   8) autogp: the automorphism group to be used
   9) tgp_ptr: a pointer to the group pointer of the extended cell
*/

void Handle_transitivity_and_imprimitivity (int effective_vertices,
                                            int current_level,
                                            int total_blocks,
                                            ptr_to_cell_node cell_ptr,
                                            ptr_to_permvect block_map_form,
                                            ptr_to_permvect block_link_form,
                                            ptr_to_permvect pvec_ptr,
                                            ptr_to_perm_gp autogp,
                                            ptr_to_perm_gp tgp_ptr)
{
    int problem = FALSE;
    ptr_to_permvect renumbered_bl_m_f = NULL, block_rep = NULL;
    ptr_to_perm_gp block_gp = NULL, extended_gp = NULL;

    renumbered_bl_m_f = allocatepv (effective_vertices);
    block_rep = allocatepv (total_blocks);
    ConvertPart(effective_vertices, block_map_form,
                renumbered_bl_m_f, block_rep);
    if (debuglist[26])
    {
       fprintf(out, "\nAfter ConvertPart, the renumbered blocks' ");
       fprintf(out, "element's representative (renumbered_bl_m_f)  are:\n");
       printpvect(out, renumbered_bl_m_f, effective_vertices);
       fprintf(out, "The block(s)' representatives (block_rep) are:\n");
       printpvect(out, block_rep, total_blocks);
    }
    /* partition blocks into subcells */
    Transform_blocks_to_cells (cell_ptr, block_link_form,
                               effective_vertices,
                               pvec_ptr, block_rep);
    if (cell_ptr->total_children > 1)
    /* This should always be true because it
       is imprimitive with group size > 1. */
    /* The cell is partitioned successfully. */
    {
       Create_influ_func_info (cell_ptr, cell_ptr->first_child);
       Handle_new_level (cell_ptr);

       extended_gp = allocatepg (tgp_ptr->permsize + total_blocks);
       /* Due to the fact that in BlockImage it will use copy_gp
          and resize blockgp if it is not big enough.
          It is better to allocate
          block_gp to have the same size as extended_gp. */

       block_gp = allocatepg (tgp_ptr->permsize + total_blocks);
       BlockImage (tgp_ptr, extended_gp, block_gp,
                   renumbered_bl_m_f, block_rep);
       /* check_branch (block_gp); */
       if (debuglist[26])
```



```
       {
          fprintf(out, "\nAfter BlockImage, the extended group is:\n");
          Print_group_information(extended_gp);
          fprintf(out, "\nThe tgp_ptr group is (domain):\n");
          Print_group_information (tgp_ptr);
          fprintf(out, "\nThe block_gp group is (range):\n");
          Print_group_information (block_gp);
       }

       if (!(finish_computation) && (Coverage_and_wastage_test(cell_ptr)))
       {
          Start_gen (current_level,
                     cell_ptr, autogp, block_gp,
                     extended_gp, pvec_ptr,
                     TRANSITIVE_IMPRIMITIVE_PARTITION, NULL);
       }
       Restore_prev_level (cell_ptr);
       disposepg (block_gp);
       block_gp = NULL;
       disposepg (extended_gp);
       extended_gp = NULL;
    }
    else
    {
       fprintf (out, "Program halt due to a cell which is imprimitive ");
       fprintf (out, "got partitioned into one subcells.\n");
       Myexit(1);
    }
    disposepv (renumbered_bl_m_f);
    renumbered_bl_m_f = NULL;
    disposepv (block_rep);
    block_rep = NULL;
} /* Handle_transitivity_and_imprimitivity */

/*********************************************************************/

/* Split the cell pointed by c_ptr into two cells C1, C2. C1 is a
   cell with only one vertex at position 'pos' and the rest of the
   vertices are in C2. The content of C1 is assigned according to
   the following heuristic:
     content(C1) = 1 if content(C) <= size(C)/2; otherwise
     content(C1) = 0
   this heuristic is bias to large size(C) and small content(C);

   Here we assume pos is the next vertex rep. under the cycle form
   in allcells of the parent cell. Moreover, parent cell must have
   a size greater than one.

   Input parameters
   1) p_ptr: the parent cell to be split

   Side effect:
   1) the parent cell node will have all its children created
   2) the vertex split will be returned

*/

/* debuglist = 20 */

int Split_rep (ptr_to_cell_node parent_ptr)

{
  int i, crep, total_subcells, first_cell_content = 0, pos,
      vbcrep, currpos, fpos, lpos;
  ptr_to_cell_node cell_ptr, last_sible;

  parent_ptr->first_child = Get_initialized_cell_node();
  cell_ptr = parent_ptr->first_child;
  parent_ptr->prev_cell->next_cell = cell_ptr;
  cell_ptr->prev_cell = parent_ptr->prev_cell;
  pos =  parent_ptr->representative;

  for (total_subcells = 1; total_subcells <= 2; total_subcells++)
  {
```

```
parent_ptr->total_children++; /* one more children */
cell_ptr->parent = parent_ptr;
cell_ptr->sibling = Get_initialized_cell_node();
cell_ptr->next_cell = cell_ptr->sibling;
cell_ptr->sibling->prev_cell = cell_ptr;

/* update allcells, cell_label, and representative, cell_size,
   ver_before_rep, actual_content. */
switch (total_subcells)
{
   /* the first cell, split one vertex */
   case 1: cell_ptr->representative =
              cell_ptr->ver_before_rep = crep = pos;
           allcells[parent_ptr->ver_before_rep] = allcells[crep];
           cell_label[crep] = allcells[crep] = crep;
           cell_ptr->cell_size = 1;

           if ((parent_ptr->actual_content != 0) &&
               ((parent_ptr->cell_size == parent_ptr->actual_content) ||
                (parent_ptr->actual_content <=
                  ((parent_ptr->cell_size)/2))))
          {
              first_cell_content = cell_ptr->actual_content
                                 = 1;
           }
           else
              cell_ptr->actual_content = 0;
           break;
   /* split the rest of the vertices into a cell,
      N.B. allcells and cell_label are up-to-date */
   case 2: /* find the new cell representative and the ver_before_rep */
           fpos = lpos = crep = parent_ptr->ver_before_rep;
           currpos = allcells[fpos];
           while (currpos != fpos)
           {
              if (currpos < crep)
              {
                  crep = currpos;
                  vbcrep = lpos;
              }
              lpos = currpos;
              currpos = allcells[currpos];
           }
           if (crep == fpos)
             vbcrep = lpos;
           /* update cell_label */
           currpos = crep;
           do {
              cell_label[currpos] = crep;
              currpos = allcells[currpos];
            } while (currpos != crep);

           cell_ptr->representative = crep;
           cell_ptr->cell_size = parent_ptr->cell_size - 1;
           /* the order of the next 2 lines must be preserved */
           cell_ptr->ver_before_rep = vbcrep;
           cell_ptr->actual_content =
                    parent_ptr->actual_content - first_cell_content;
           break;
}

/* create the dummies */
cell_ptr->row_first_neigh = Get_initialized_neigh_info_node();
cell_ptr->row_first_neigh->row_cell_num = crep;
cell_ptr->row_first_neigh->col_cell_num = 0;
cell_ptr->col_first_neigh = Get_initialized_neigh_info_node();
cell_ptr->col_first_neigh->row_cell_num = 0;
cell_ptr->col_first_neigh->col_cell_num = crep;

/* update the contents */
cell_ptr->lower_content = cell_ptr->upper_content =
  cell_ptr->actual_content;

cell_list[crep] = cell_ptr;
last_sible = cell_ptr;
```



```
      cell_ptr = cell_ptr->sibling;
   }
   last_sible->next_cell = parent_ptr->next_cell;
   parent_ptr->next_cell->prev_cell = last_sible;
   dummy_cell->total_children += (parent_ptr->total_children - 1);

   /* free the last extra cell node created */
   /* the free will not work if no partition occur,
      then cell_ptr == last_sible */
   Free_cell_node (last_sible->sibling);
   last_sible->sibling = NULL;

   if (debuglist[20])
   {
      fprintf(out, "One vertex was split, the cells are:\n");
      Print_children(parent_ptr);
   }
   return (pos);

} /* Split_rep */

/*********************************************************************/

/* Handle the situation when cell is transitive and primitive. */

/* Input parameter:
   1) current_level: the latest level created
   2) cell_ptr: pointer to the selected cell which will be partitioned
   3) autogp: the automorphism group to be used
*/


void Handle_primitivity_and_transitivity (int current_level,
                                          ptr_to_cell_node cell_ptr,
                                          ptr_to_perm_gp autogp)
{
   static ptr_to_permvect stabilize_vertex = NULL;

   ptr_to_perm_gp part_stabilizer = NULL;
   ptr_to_cell_node selected_cell;

   if ((NULL == stabilize_vertex) || reset_stabilize_vertex)
   {
      if (stabilize_vertex != NULL)
      {
         reset_stabilize_vertex = FALSE;
         disposepv (stabilize_vertex);
         stabilize_vertex = NULL;
      }
      stabilize_vertex = allocatepv (total_vertices);
   }
   /* The position next to the cell representative will be selected. */
   /* The cell node pointed by cell_ptr */
   /* always has more than one vertex. */
   /* The vertex selected will be returned from Split_rep. */
   stabilize_vertex[1] = Split_rep (cell_ptr);
   Create_influ_func_info (cell_ptr, cell_ptr->first_child);
   Handle_new_level (cell_ptr);

   if (Update_max_allowable_wastage (cell_ptr))
   {
      selected_cell = Select_next_partition_cell ();
      if (debuglist[26])
      {
          fprintf (out, "Symmetry partition: group is primitive.\n");
          fprintf (out, "The selected cell is:\n");
          Print_cell(selected_cell);
      }

      /* Selected cell should not be empty. */
      /* otherwise, a complete partition is achieved. */
      if (selected_cell != NULL)
      {
         part_stabilizer = copy_gp(autogp, part_stabilizer);
         if (debuglist[26]) {
```

```
            fprintf (out, "\npart_stabilizer in Hpat is allocated by copy_gp ");
            fprintf (out, "from autogp\n");
            Print_group_information (part_stabilizer);
         }

         stabilize_pts(part_stabilizer, stabilize_vertex, 1);
         global_level = current_level + 1;
         Next_level(SYMMETRY, current_level + 1,
                    selected_cell, part_stabilizer);
         global_level = current_level;
         disposepg (part_stabilizer);
         part_stabilizer = NULL;
      }
      else
        Check_and_handle_solution();
   }
   Restore_prev_level (cell_ptr);
} /* Handle_primitivity_and_transitivity */

/*************************************************************************/

/* To get all the orbits from the group gp.
   It call the procedure find_orbit in isom.

   input parameter
   1) gp: the group
   2) vertices_array: either NULL or
                      an 0,1 array with vertices_array[i] equal to
                      1 if the vertices is in this array
   3) size: total size of all the orbits

   output parameter
   1) the procedure returns a vector containing all the orbits
   2) orbit_rep: an allocated array. At
                 orbit_rep[0] it will contain the total
                 number of orbits found and the representative
                 of each orbit will be contained at
                 orbit_rep[1] to orbit_rep[total_orbits].

                 This array is updated only if vertices_array
                 is not NULL.

*/

/* debuglist = 22 */

ptr_to_permvect Get_all_orbits (ptr_to_perm_gp gp,
                                ptr_to_permvect vertices_array,
                                ptr_to_permvect orbit_rep,
                                int size)

{
   ptr_to_permvect orbits;
   int total_orbits = 0, i;

   orbits = allocatepv (total_vertices);

   /* initialize orbits */
   for (i=0; i <= size; i++)
     orbits[i] = 0;

   /* generate the orbits */
   if (vertices_array == NULL)
   {
      for (i=1; i <= size; i++)
        if (0 == orbits[i])
        {
           find_orbit (gp, i, 0, orbits, NULL);
           /* sort_circ_list(orbits, sorted_orbits, i); */
           total_orbits++;
        }
   }
   else
   {
      orbit_rep[0] = 0;
```



```
      for (i=1; i <= size; i++)
        if ((0 == orbits[i]) && (vertices_array[i] != 0))
        {
           orbit_rep[0]++;
           orbit_rep[orbit_rep[0]] = i;
           find_orbit (gp, i, 0, orbits, NULL);
           /* sort_circ_list(orbits, sorted_orbits, i); */
           total_orbits++;
        }
   }
   orbits[0] = total_orbits;

   if (debuglist[22])
   {
      fprintf(out, "\nAll the %d orbits obtained are:\n", total_orbits);
      printpvect(out, orbits, size);
      print_orbits (out, gp, 1);
   }
   return (orbits);

} /* Get_all_orbits */

/*************************************************************************/

/* Handle the transitivity case */
/* Input parameters
   1) effective_vertices: the size pointed by cell_ptr
   2) pvec_ptr: an array that contains all the vertices in cell_ptr, if
                the array is not full, it should end with a zero entry
   3) tgp_ptr: a pointer to the group pointer of the extended cell
   4) current_level: the latest level created
   5) cell_ptr: pointer to the selected cell which will be partitioned
   6) autogp: the automorphism group to be used
*/

void Handle_transitivity (int effective_vertices,
                          ptr_to_permvect pvec_ptr,
                          ptr_to_perm_gp tgp_ptr,
                          int current_level,
                          ptr_to_cell_node cell_ptr,
                          ptr_to_perm_gp autogp)
{
    int total_blocks = 0;
    ptr_to_permvect block_map_form, block_link_form;

    /* using Leon's interface */
    block_map_form = allocatepv(effective_vertices);
    block_link_form = allocatepv(effective_vertices);
    total_blocks = GreedyMaxPart(tgp_ptr, block_map_form,
                                 block_link_form);
    if (debuglist[26])
    {
       fprintf(out, "\nAfter GreedyMaxPart, the total block(s)");
       fprintf(out, " size are %d.\n", effective_vertices);
       fprintf(out, "%d block(s) are found.\n", total_blocks);
       fprintf(out, "The block(s)' elements' representative ");
       fprintf(out, "(block_map_form) are:\n");
       printpvect(out, block_map_form, effective_vertices);
       fprintf(out, "The blocks (block_link_form) are:\n");
       printpvect(out, block_link_form, effective_vertices);
    }
    if (total_blocks < tgp_ptr->permsize) /* imprimitive */
       Handle_transitivity_and_imprimitivity(effective_vertices,
          current_level, total_blocks, cell_ptr, block_map_form,
          block_link_form, pvec_ptr, autogp, tgp_ptr);
    else /* primitive */
      Handle_primitivity_and_transitivity (current_level,
        cell_ptr, autogp);
    disposepv (block_map_form);
    block_map_form = NULL;
    disposepv (block_link_form);
    block_link_form = NULL;
} /* Handle_transitivity */

/*************************************************************************/
```

```
/* To test if the cell under the group autogp is transitive.
   When exit, if the cell is non_transitive,
   the program will return a vector containing all the
   (possibly incomplete) orbits relabelled from 1 to cellsize.
   Otherwise a NULL pointer will
   be returned. The vertices may be relabelled and
   pvec_ptr will map the new label back to the old label.

   Input parameter:
   1) cell_ptr: pointer to the selected cell which will be partitioned
   2) pvec_ptr: an array that contains all the vertices in cell_ptr, if
                the array is not full, it should end with a zero entry
   3) autogp: the automorphism group to be used

   Output parameters:
   1) a vector contains all the (possibly incomplete) orbits
      relabelled from 1 to the size of the cell, orbit will be stored
      in their cyclic form.
   2) pvec_ptr will be the mapping from new label to old label
*/

ptr_to_permvect Test_transitivity (ptr_to_cell_node cell_ptr,
                                   ptr_to_permvect pvec_ptr,
                                   ptr_to_perm_gp autogp)
{
   /* array used to extend the vertices after calling complete_orbits */
   static ptr_to_permvect extended_map = NULL;
   /* array with 1 in the entries where the vertices exist,
      this include the extend ones */
   static ptr_to_permvect orb_vertices = NULL;
   /* an array with 1 in the vertices of the cell */
   static ptr_to_permvect vertices_set = NULL;
   /* the inverse mapping of the vertices in cell when
      sorted in ascending order */
   static ptr_to_permvect inverse_map = NULL;
   /* orbrep contains a list of representative for each orbit */
   static ptr_to_permvect orbrep = NULL;

   int i, cellsize, final_orb_size, rep, cpos, topos, fopos;
   ptr_to_permvect orbvec = NULL, finalorbvect = NULL;

   if ((extended_map == NULL) || reset_extended_map)
   {
      if (extended_map != NULL)
      {
         reset_extended_map = FALSE;
         disposepv (extended_map);
         extended_map = NULL;
         disposepv (orb_vertices);
         orb_vertices = NULL;
         disposepv (vertices_set);
         vertices_set = NULL;
         disposepv (inverse_map);
         inverse_map = NULL;
         disposepv (orbrep);
         orbrep = NULL;
      }
      extended_map = allocatepv (total_vertices+1);
      orb_vertices = allocatepv (total_vertices);
      vertices_set = allocatepv (total_vertices);
      inverse_map = allocatepv (total_vertices);
      orbrep = allocatepv (total_vertices);
   }


   copy_pv(extended_map, pvec_ptr, total_vertices+1);
   extended_map[0] = pvec_ptr[0];

   cellsize = cell_ptr->cell_size;

   if (autogp->permsize < total_vertices)
   {
      for (i = autogp->permsize + 1; i <= total_vertices; i++)
```



```
         orb_vertices[i] = 0;
   }


   final_orb_size = complete_orbits (autogp, extended_map,
                                     orb_vertices, TRUE);

   /* the extended_map returned with the vertices in the cell
      sorted in ascending order followed by the extended vertices used
      to complete the orbits sorted in ascending order. */

   if (debuglist[26])
   {
      fprintf (out, "\nComplete orbit size is: %d.\n", final_orb_size);
      fprintf (out, "The completed orbit(s)' vertices are:\n");
      printpvect (out, orb_vertices, total_vertices);
   }
   orbvec = Get_all_orbits (autogp, orb_vertices, orbrep,
                            total_vertices);
   /* orbvec contain all the completed orbits, orbrep contains a list
      of representative for each orbit */

   if (orbvec[0] > 1) /* non transitive */
   {
      /* initialization of static variables, orbrep do not
         need to be initialized, it will be updated in
         Get_all_orbits, so do orb_vertices */

      for (i = 0; i <= total_vertices; i++)
      {
         vertices_set[i] = inverse_map[i] = 0;
      }
      /* N.B. we only run from 1 to cellsize */
      for (i = 1; i <= cellsize; i++)
      {
         vertices_set[extended_map[i]] = 1;
         inverse_map[extended_map[i]] = i;
         /* copy the sorted permvect to the original permvec */
         pvec_ptr[i] = extended_map[i];
      }

      finalorbvect = allocatepv (total_vertices);
      for (i = 0; i <= total_vertices; i++)
      {
         finalorbvect[i] = 0;
      }

      for (i = 1; i <= orbrep[0]; i++)
      {
         topos = 0;
         rep = orbrep[i];
         cpos = rep;
         do {
           if (vertices_set[cpos])
           {
              if (topos == 0) /* first vertex */
              {
                 fopos = topos = inverse_map[cpos];
              }
              else
              {
                 finalorbvect[topos] = inverse_map[cpos];
                 topos = inverse_map[cpos];
              }
              finalorbvect[topos] = fopos;
           }
           cpos = orbvec[cpos];
         } while (cpos != rep);
      }
      if (debuglist[26])
      {
         /* print the result finalorbitvect and pvec_ptr */
      }
     finalorbvect[0] = orbvec[0];
   }
```

```
   else
   {
      if (debuglist[26])
      {
         /* report non-transitive */
      }
   }
   disposepv (orbvec);
   orbvec = NULL;
   return (finalorbvect);

} /* Test_transitivity */

/*******************************************************************/

/* Include the method of symmetry when partition
   of the contents in order to reduce the search tree.
   enum type part_type is used here
*/

/* Input parameters
   1) current_level: the latest level created
   2) cell_ptr: pointer to the selected cell which will be partitioned
   3) autogp: the automorphism group to be used

   Side effect
   1) Partitioned the input using autogp
*/

/* debuglist = 26 */

void Symmetry_partition (int current_level,
                         ptr_to_cell_node cell_ptr,
                         ptr_to_perm_gp autogp)

{
   /* a global group pointer to represent the group */

   ptr_to_perm_gp tgp_ptr = NULL;
   /* block_gp is used as the group for isomorphic testing
      the set of content; */
   ptr_to_permvect pvec_ptr = NULL, orbvec = NULL;
   ptr_to_cell_node old_cell_ptr;
   int i, effective_vertices;

   if (cell_ptr == NULL)
   {
      fprintf(out, "\nProgram exit due to NULL pointer was passed ");
      fprintf(out, "to Symmetry_partition.\n");
      Myexit(1);
   }
   else
     if (debuglist[26])
     {
        fprintf(out, "\nCell passed in for Symmetry Partition ");
        fprintf(out, "at level %d are:\n", current_level);
        Print_cell(cell_ptr);
     }

   if (((cell_ptr->actual_content == 0) ||
       (cell_ptr->actual_content == cell_ptr->cell_size)) &&
      DISCRETION)
     Sym_discretion (current_level, cell_ptr);
   else
   {
      if (!Identity (autogp))
      {
         if (debuglist[26])
         {
            fprintf(out, "\nThe cell to be transform to permvect:\n");
            Print_cell(cell_ptr);
         }

         pvec_ptr = Transform_cell_to_permvect (cell_ptr);
         /* Quicksort sort position 0 to cell_ptr->cell_size */
```



```
         /* Quicksort(pvec_ptr, cell_ptr->cell_size + 1); */
         orbvec = Test_transitivity (cell_ptr, pvec_ptr, autogp);

         effective_vertices = cell_ptr->cell_size;
         if (orbvec != NULL)
            Handle_non_transitivity(current_level, effective_vertices,
                 cell_ptr, orbvec, pvec_ptr, autogp);
         else
         {
             /* tgp_ptr and effective_vertices was set */
            if (total_vertices != cell_ptr->cell_size)
              tgp_ptr = Extend_cell (cell_ptr, pvec_ptr, autogp);
            else
              tgp_ptr = copy_gp(autogp, tgp_ptr);
            if (debuglist[26])
            {
               fprintf(out, "\nThe premvect extended after calling ");
               fprintf(out, "truncate_gp.\n");
               printpvect(out, pvec_ptr, total_vertices);
               fprintf (out, "\nThe truncated group is:\n");
               Print_group_information (tgp_ptr);
            }
            orbvec = Get_all_orbits (tgp_ptr, NULL, NULL,
                                     effective_vertices);

            old_cell_ptr = cell_ptr;
            if (Find_non_identity_cell(current_level, autogp, &tgp_ptr,
                                       &pvec_ptr, &cell_ptr))
            {
            /* 19/12/96 added the condition (orbvec[0] == 1) */
               if ((old_cell_ptr == cell_ptr) && (orbvec[0] == 1))
                 /* cells does not change */
                 Handle_transitivity (effective_vertices,
                                      pvec_ptr, tgp_ptr,
                                      current_level,
                                      cell_ptr, autogp);
               else
                 if (old_cell_ptr == cell_ptr)
                 {
                    /* no change in cell, the group is not identity */
                    Handle_non_transitivity(current_level, effective_vertices,
                       cell_ptr, orbvec, pvec_ptr, autogp);
                 }
                 else /* cell pass in has identity group, a new cell
                         with non identity group is found */
                 {
                    effective_vertices = cell_ptr->cell_size;
                    disposepv (orbvec);
                    orbvec = Get_all_orbits (tgp_ptr, NULL, NULL,
                                             effective_vertices);
                    if (orbvec[0] != 1)
                      Handle_non_transitivity(current_level, effective_vertices,
                         cell_ptr, orbvec, pvec_ptr, autogp);
                    else /* transitive */
                      Handle_transitivity (effective_vertices,
                                           pvec_ptr, tgp_ptr,
                                           current_level,
                                           cell_ptr, autogp);
                 }
            }
            else
              Regular_partition (current_level, cell_ptr, autogp);
         }
         disposepv (orbvec);
         orbvec = NULL;
         disposepv (pvec_ptr);
         pvec_ptr = NULL;
         disposepg (tgp_ptr);
         tgp_ptr = NULL;
      }
      /* Identity is past to Symmetry_partition. That happen when block or
         orbit partition has used up the group. */
      else
      {
         if (debuglist[26])
```

```
        {
          fprintf (out, "\nIdentity group is passed to ");
          fprintf (out, "Symmetry_partition, switched back to ");
          fprintf (out, "Regular_partition.\n");
        }
        /* In that case cell_ptr will be used for partition, since
           no new level is created, next level is not called. */
        Regular_partition (current_level, cell_ptr, autogp);
      }
    }
} /* Symmetry_partition */

/*********************************************************************/

/* To decide whether the next level of cell partition is the
   regular partition (not making use of the group) or
   symmetry partition (making use of the group).
*/

/* Input parameters
   1) tag: the way to partition the cell
   2) lev: current level
   3) cptr: the pointer to the cell to be partitioned
   4) gptr: the pointer to the automorphism group
*/

/* debuglist = 24 */
void Next_level (enum part_type tag, int lev,
                 ptr_to_cell_node cptr,
                 ptr_to_perm_gp gptr)

{
   if (CONSIS_CHECK)
     Consistence_check();
   level_visited[lev]++;

   if (debuglist[24])
   {
     fprintf (out, "\nLevel %d is visited %d times.\n",
                 lev, level_visited[lev]);
   }
   if ((lev == PRINT_LEVEL) &&
       ((level_visited[lev] %
         CUT_MULTIPLE) == 0))
   {
      Print_level_visited();
      Calculate_and_print_time_used ();
      fflush(out);
   }

   if (lev != STOP_LEVEL)
     switch (tag)
     {
       case SYMMETRY: if (debuglist[24])
                      {
                         fprintf (out, "\nSwitched to ");
                         fprintf (out, "SYMMETRY_PARTITION.\n\n");
                      }
                      Symmetry_partition (lev, cptr, gptr);
                      break;
       case REGULAR:  if (debuglist[24])
                      {
                         fprintf (out, "\nSwitched to ");
                         fprintf (out, "REGULAR_PARTITION.\n\n");
                      }
                      Regular_partition (lev, cptr, gptr);
                      break;
     } /* switch (tag) */
} /* Next_level */

/*********************************************************************/

/* Select one of the children to be the next partition cell.

   Input parameters
```



```
   1) p_ptr: pointer to the parent cell

   Output parameter
   1) a pointer to the next partition cell will be returned, if there is
      no partitionable cell, NULL will be returned
*/

/* debuglist = 21 */

ptr_to_cell_node  Select_subcell_from_parent (ptr_to_cell_node p_ptr)

{
   ptr_to_cell_node current_cell, best_cell;

   best_cell = NULL;
   current_cell = p_ptr->first_child;

   while ((current_cell != NULL)
          &&
          (current_cell->cell_size <= 1))
     current_cell = current_cell->sibling;

   best_cell = current_cell;

   if (current_cell != NULL)
     current_cell = current_cell->sibling;

   while (current_cell != NULL)
     {
        if (current_cell->cell_size > 1)
          if (current_cell->max_allow_wastage <
              best_cell->max_allow_wastage)
            best_cell = current_cell;
          else
            if (current_cell->max_allow_wastage ==
                best_cell->max_allow_wastage)
              if (current_cell->cell_size >
                  best_cell->cell_size)
                best_cell = current_cell;

        current_cell = current_cell->sibling;
     }

   if (debuglist[20])
   {
      fprintf (out, "\nA subcell selected from parent is:\n");
      Print_cell (best_cell);
   }

   return (best_cell);

} /* Select_subcell_from_parent */

/*********************************************************************/

/* A procedure used by the procedure find_certificate in the isom
   package to compare whether a partial permutation will lead to
   a higher lexicographical order.

   Input parameters
   1) choice and index: the parameter choice[1..index]
*/

/* debuglist = 27 */

comprestype Compare(ptr_to_permvect choice, int index)

{
   extern int canonical_rep;
   extern ptr_to_permvect content_part, max_part;

   /* compare max_part to content_part^(choice[1..index]) */

   int temp1, temp2;
```

```
   temp1 = content_part[choice[index]];
   temp2 = max_part[index];

   if (debuglist[27])
   {
      fprintf (out, "Calling compare:\n");
      fprintf (out, "index = %d, choice[index]= %d, ",
                    index, choice[index]);
      fprintf (out, "content_part[choice[index]] (temp1) = %d, ",
                    temp1);
      fprintf (out, "max_part[index] (temp2) = %d.\n", temp2);
   }

   if (temp1 == temp2)
     return indifferent;
   else  /* our lexicographical order is ranked decreasingly, hence
            1 is worse that 3, we rank it this way because the content
            is generated in that way */
     if (temp1 < temp2) return worse;
     else
     {
         canonical_rep = FALSE;
         return isom_exit;
     }
} /* Compare */

/*********************************************************************/

/* Return true if the content partition is not isomorphic to an old
   content partition under the symmetry group.
   canonical_rep should be initialized to true before entering to this
   procedure.

   input parameters
   1) b_ptr: pointer to the permutation group of the block
   2) content_dist: the content partition

   output parameters
   1) canonical_rep is changed to FALSE if this content partition
      is isomorphic to an early partition
   2) the automorphism group of the block will be returned
*/

/* b_ptr may have to make global */

/* debuglist = 21 */

ptr_to_perm_gp Isomorphic_test (ptr_to_perm_gp b_ptr,
                                int *content_dist,
                                int total_blks)

{
   extern int canonical_rep;
   extern int total_vertices;
   extern ptr_to_permvect content_part, max_part;

   ptr_to_perm_gp automorphism_gp = NULL;

   ptr_to_perm_gp return_gp = NULL;
   int i;

   if (debuglist[21])
   {
      fprintf(out, "\nIsomorphic_testing:\n");
      fprintf(out,"\nThe input group is:\n");
      Print_group_information (b_ptr);
      fprintf(out, "The total number of blocks are %d.\n", total_blks);
   }

   /* canonical_rep = TRUE; */

   /* the following two global variables will be pre-allocated to the
      size of the graph. */
   if (NULL == content_part)
   {
```



```
      content_part = allocatepv(total_vertices);
      max_part = allocatepv(total_vertices);
   }

   for (i = 0; i < total_blks; i++)
     content_part[i+1] = (permval) (content_dist[i]);
   content_part[0] = 0;
   if (debuglist[21])
   {
      fprintf(out, "The content_part is initialized as:\n");
      printpvect(out, content_part, total_blks);
   }
   copy_pv (max_part, content_part, total_blks);
   max_part[0] = content_part[0];
   automorphism_gp = build_trivial_gp (automorphism_gp, total_blks);

   if (debuglist[21])
   {
     fprintf (out, "Trying to find certificate.\n");
   }

   find_certificate (b_ptr, automorphism_gp, Compare, 0);

   if (debuglist[21])
   {
      fprintf(out, "The set of content passed for isomorphic testing is:\n");
      Print_int_array (total_blks, content_dist);
      fprintf(out, "The isomorphic test result is %d.\n", canonical_rep);
   }
   if (canonical_rep)
   {
      return_gp = automorphism_gp;
   }
   else
   {
      disposepg (automorphism_gp);
      automorphism_gp = NULL;
   }

   return (return_gp);

} /* Isomorphic_test */

/*********************************************************************/

/* Handle the content partition in the case of not transitive, and the
   cells are partitioned into orbits. */
/* This procedure is used by Start_gen in basicdom. */

/* Input parameters
   1) current_level: the latest level created
   2) parent_ptr: pointer to the cell which was partitioned
   3) gptr: the current automorphism group
   4) array_vol: the total number of subcells partitioned from the
                 parent
   5) content_distribution: the content distribution of the children
                            partitioned from the parent
   6) next_part_cell: if not NULL, it will contain the next cell
                      to be partitioned

*/

void Handle_orb_content_part (int current_level,
                              ptr_to_cell_node parent_ptr,
                              ptr_to_perm_gp gptr,
                              int array_vol,
                              int *content_distribution,
                              ptr_to_cell_node next_part_cell)
{
   ptr_to_cell_node selected_cell, temp_ptr;

   /* Select_subcell_from_parent is better to put here because the
      content is known and a better cell can be selected.
      Selected cell should not be
      empty because the total orbits
```

```
      is greater than one and the
      truncated group is not identity.
      Since the group is not identity
      there must have orbit with size > 1.
   */

      afmawtest[current_level]++;

      /* We do not need to care about
         the cells in next_part_cell_list because procedure
         Handle_non_transitivity has take care of it. This procedure
         is called by Handle_non_transitivity. */
      if (next_part_cell == NULL)
      {
         selected_cell = Select_subcell_from_parent(parent_ptr);
         if (selected_cell == NULL)
           selected_cell = Select_next_partition_cell ();
      }
      else
        selected_cell = next_part_cell;
      /* We should not need to generate a new partition stabilizer
         because in the case of orbit partition, the partition
         stabilizer should be the same as the old one. */

      if (selected_cell != NULL)
      {
         global_level = current_level + 1;
         Next_level(SYMMETRY, current_level + 1,  selected_cell, gptr);
         global_level = current_level;
      }
      else
        Check_and_handle_solution ();
} /* Handle_orb_content_part */

/*********************************************************************/

/* Handle the content partition in the case of transitive and imprimitive.
   The cells are partitioned according to the block obtained. */
/* This procedure is used by Start_gen in basicdom. */

/* Input parameters
   1) current_level: the latest level created
   2) parent_ptr: pointer to the cell which was partitioned
   3) gptr: the current automorphism group
   4) bptr: the block automorphism group
   5) extgp: the intermediate group in the homomorphism
   6) mapping: to map back the elements in the truncated group to their
               old value in the original graph
   7) block_auto: the block automorphism group
   8) array_vol: the total number of subcells partitioned from the
                 parent
   9) content_distribution: the content distribution of the children
                            partitioned from the parent

*/

/* debuglist = 32 */

void Handle_transitive_content_part (int current_level,
                                     ptr_to_cell_node parent_ptr,
                                     ptr_to_perm_gp gptr,
                                     ptr_to_perm_gp bptr,
                                     ptr_to_perm_gp extgp,
                                     ptr_to_permvect mapping,
                                     int array_vol,
                                     int *content_distribution,
                                     ptr_to_perm_gp block_auto)
{
   ptr_to_cell_node selected_cell;
   ptr_to_perm_gp part_stabilizer = NULL, preimage = NULL;

   if (debuglist[32])
   {
      fprintf (out, "\nThe block group in the ");
      fprintf (out, "Handle_transitive_content_part is:\n");
```



```
      Print_group_information(bptr);
   }


   /* blocks of size 1 may exist, hence if all the blocks have
      a size one, Select_subcell_from_parent will return NULL */

   selected_cell = Select_subcell_from_parent (parent_ptr);
   if (NULL == selected_cell)
      selected_cell = Select_next_partition_cell();
   if (NULL == selected_cell)
      Check_and_handle_solution ();
   else
   {
      /* The select_subcell */
      /* should fail because of primitive. */

      preimage = allocatepg(gptr->permsize);

      if (debuglist[32])
      {
         fprintf (out, "Handle transitive content partition ");
         fprintf (out, "(imprimitive).\n");
         fprintf (out, "\nextgp:\n");
         Print_group_information (extgp);
         fprintf (out, "extgp->permsize = %d.\n", extgp->permsize);
         fprintf (out, "\nblock_auto:\n");
         Print_group_information (block_auto);
         fprintf (out, "block_auto->permsize = %d.\n",
                  block_auto->permsize);
         fprintf (out, "Number of blocks (bptr->permsize) = ");
         fprintf (out, "%d, N = %d.\n",
                       bptr->permsize, extgp->permsize - bptr->permsize);

      }

      BlockPreImage (preimage, extgp, block_auto, bptr->permsize,
                     extgp->permsize - bptr->permsize);

      if (debuglist[32])
      {
         fprintf (out, "Finished BlockPreImage.\n");
         fprintf (out, "\npreimage:\n");
         Print_group_information (preimage);
         fprintf (out, "\nextgp:\n");
         Print_group_information (extgp);
         fprintf (out, "extgp->permsize = %d.\n", extgp->permsize);
         fprintf (out, "\nblock_auto:\n");
         Print_group_information (block_auto);
         fprintf (out, "block_auto->permsize = %d.\n",
                  block_auto->permsize);
         fprintf (out, "\nLast partition stabilizer gptr:\n");
         Print_group_information (gptr);
         fprintf (out, "Here is the mapping past in:\n");
         printpvect (out, mapping, total_vertices);
      }

      part_stabilizer = copy_gp(gptr, part_stabilizer);
      replace_sg (part_stabilizer, preimage, mapping);

      if (debuglist[32])
        Print_group_order ("New partition stabilizer", part_stabilizer);

      global_level = current_level + 1;
      Next_level(SYMMETRY, current_level + 1,
         selected_cell, part_stabilizer);
      global_level = current_level;

      disposepg (part_stabilizer);
      part_stabilizer = NULL;
      disposepg (preimage);
      preimage = NULL;
   }
} /* Handle_transitive_content_part */
```

```
/**********************************************************************/

/* This procedure will initialize all the variables in this program:
   groupdom.c, the initialization in domset.c should be done first;
   total_vertices and total_content should be initialized. */

void Initiate_groupdom_val (voidplist)

{
   int i;

   cwpgcycle = allocatepv (total_vertices);
   cwpgcolor = allocatepv (total_vertices);
   /* allocate memory to the two dimension array colormat */
   colormat = (int **) calloc (total_vertices+1, sizeof (int *));
   more_than_one_orbs = (int *) calloc (total_vertices+1, sizeof (int));
   more_than_one_orbs[0] = 0;
   colormat[0] = NULL;
   for (i = 1; i <= total_vertices; i++)
   {
      more_than_one_orbs[i] = 0;
      colormat[i] = (int *) calloc(org_total_content+1, sizeof (int));
   }


} /* Initiate_groupdom_val */

/**********************************************************************/

/* This procedure will reinitialize all the variables
   that is required to make
   the program run for another set of user input partition cell and
   content given the same graph. */

void Reinitiate_groupdom_val (voidplist)

{
   int i;

   disposepv (cwpgcycle);
   cwpgcycle = NULL;
   disposepv (cwpgcolor);
   cwpgcolor = NULL;
   /* release the memory of the colormat */
   for (i = 1; i <= total_vertices; i++)
     free((void *) colormat[i]);
   free((void *) colormat);
   colormat = NULL;
   free ((void *) more_than_one_orbs);
   more_than_one_orbs = NULL;

   if (content_part != NULL)
   {
      disposepv(content_part);
      content_part = NULL;
      disposepv(max_part);
      max_part = NULL;
   }

   reset_tempvec_ptr =
   reset_extended_map =
   reset_stabilize_vertex =
   reset_torbit =
   reset_vert_set = TRUE;

} /* Reinitiate_groupdom_val */

/**********************************************************************/
```

```
#ifndef MEMMDOM

#define MEMMDOM 1

#if  defined(apollo) || defined(__GNUC__)
#define HALFANSI
#define voidplist void
#else
#ifdef mips
/*   #define SYSTYPE_BSD43 */
#define HALFANSI
#define void char
#define voidplist
#else
#define voidplist void
#endif
#endif

#include "basicdom.h"

/*********************************************************************/
/* GLOBAL CONSTANTS opened to other files */
/*********************************************************************/

/* Memory management parameters, these parameter helps to
   choose the pre-allocate size of the cell, neighbor and
   tvdrc_rack node. */

#define MEMORY_FACTOR 4 /* the total_vertices will be divided by
                           this factors to get the pre-allocate size */
#define MIN_CALLOC_SIZE 100 /* this is the minimum pre-allocate size */

/*********************************************************************/
/* GLOBAL TYPES opened to other files */
/*********************************************************************/

typedef struct list_of_cells *ptr_to_list_of_cells;

struct list_of_cells {
       int level;
       ptr_to_cell_node corr_cell;
       ptr_to_list_of_cells next_entry;
};

/*********************************************************************/
/* GLOBAL VARIABLES opened to other files */
/*********************************************************************/

extern int cell_calloc_size, neigh_calloc_size,
           tvdrc_calloc_size, list_of_cells_calloc_size;

/*********************************************************************/
/* GLOBAL FUNCTIONS opened to other files */
/*********************************************************************/

/* Create and initialize a cell_node. */

/* Output parameter
   Return a initialized cell_node. */

/* N.B. Entry next_cell is used to link all the free cells,
        hence this entry will be changed. */

extern ptr_to_cell_node Get_initialized_cell_node (voidplist);

/*********************************************************************/

/* Procedure to release a cell node.

   Input parameter:
   tcell: the cell node to be free
*/

/* N.B. Entry next_cell is used to link all the free nodes,
        hence this entry will be changed. */
```



```
extern void Free_cell_node (ptr_to_cell_node tcell);

/*********************************************************************/

/* Get and initialize a tvdrc.

   Output parameter
   it returns an initialized tvdrc

   N.B. entry value in tvdrc_rack is always allocated.

*/

extern int *Get_initialized_tvdrc (voidplist);

/*********************************************************************/

/* Procedure to release a tvdrc_info_node.

   Input parameter:
   tvdrc_ptr: the tvdrcbor node to be free
*/

/* N.B. Entry row_next_tvdrc is used to link all the free nodes,
        hence this entry will be changed. */

extern void Free_tvdrc (int *tvdrc_ptr);

/*********************************************************************/

/* Get and initialize a neigh_info_node.
   All the entry are initializer to NULL.

   Output parameter
   it returns an initialized neigh_info_node

*/

extern ptr_to_neigh_info_node Get_initialized_neigh_info_node
        (voidplist);

/*********************************************************************/

/* Procedure to release a neigh_info_node.

   Input parameter:
   neigh_ptr: the neighbor node to be free
*/

/* N.B. Entry row_next_neigh is used to link all the free nodes,
        hence this entry will be changed. */

extern void Free_neigh_info_node (ptr_to_neigh_info_node neigh_ptr);

/*********************************************************************/

/* Get and initialize a list_of_cells.

   Output parameter
   it returns an initialized tvdrc

   N.B. Entry value in tvdrc_rack is always allocated.

*/

ptr_to_list_of_cells Get_initialized_list_of_cells(voidplist);

/*********************************************************************/

/* Procedure to release a list_of_cells node.

   Input parameter:
   loc_ptr: the list_of_cells node to be free
*/
```

```
/* N.B. entry next_entry is used to link all the free nodes,
        hence this entry will be changed. */

void Free_list_of_cells (ptr_to_list_of_cells loc_ptr);

/**************************************************************************/

/* This procedure will reinitialize all the variables
   that is required to make
   the program run for another set of user input partition cell and
   content, given the same graph.
*/

void Reinitiate_memmdom_val (voidplist);

/**************************************************************************/

#endif
```

```c
#include <stdio.h>
#include "basicdom.h"
#include "memmdom.h"

/* Memory management procedures are stored here. */

/*******************************************************************/

/* GLOBAL TYPES used locally. */

/* The following structure is used in neigh_info_node to
   store an array of total vertices of degree from row to column.
   it is designed as such to reduce the effort when doing one's own
   memory management.

   value: The array of total vertices of degree from row to column.
   next: A pointer to the next tvdrc_rack.
*/


typedef struct tvdrc_rack *ptr_to_tvdrc_rack;

struct tvdrc_rack {
     int *value;
     ptr_to_tvdrc_rack next;
};


/* all rack is used to keep track all the allocated tvdrc_rack */

typedef struct all_rack *ptr_to_all_rack;

struct all_rack {
     ptr_to_tvdrc_rack rack;
     ptr_to_all_rack next;
};

/* all_tarray  is used to keep track all the allocated tarray */

typedef struct all_tarray *ptr_to_all_tarray;

struct all_tarray {
     int *array;
     ptr_to_all_tarray next;
};


/*******************************************************************/

/* GLOBAL VARIABLES */

/* the empty_neigh_info_node->row_next_neigh is used to link each
   of the available node together */

int cell_calloc_size, neigh_calloc_size, tvdrc_calloc_size,
    list_of_cells_calloc_size;
ptr_to_neigh_info_node empty_neigh_info_node = NULL;
ptr_to_cell_node empty_cell_node = NULL;

/* empty_tvdrc_rack is the list of node with the entry value
   initialized to an array of size max_degree +1;
   rack is the list of node with entry value equal to NULL */

ptr_to_tvdrc_rack empty_tvdrc_rack = NULL, empty_rack = NULL;
ptr_to_all_rack initialized_rack = NULL;
ptr_to_all_tarray initialized_array = NULL;
ptr_to_list_of_cells empty_list_of_cells = NULL;

/*******************************************************************/

/* To print out the information in a tvdrc_node

   Input parameter
   1) tvdrc: pointing to the tvdrc_node that is to be printed
*/
```



```c
void Print_rack (ptr_to_tvdrc_rack tvdrc)

{
   int i;

   if (tvdrc == NULL)
   {
     fprintf (out, "NULL\n");
   }
   else
   {
      fprintf (out, "\n");
      fprintf (out, "tvdrc = %x\n", tvdrc);
      fprintf (out, "The array's address = %x\n", tvdrc->value);
      if (tvdrc->value != NULL)
      {
         fprintf (out, "The array of the value entry:\n");
         Print_int_array (max_degree + 1, tvdrc->value);
      }
      fprintf (out, "The next address = %x\n", tvdrc->next);
   }
} /* Print_rack */

/*******************************************************************/

/* Create and initialize a cell_node. */

/* Output parameter
   Return a initialized cell_node */

/* N.B. Entry next_cell is used to link all the free cells,
        hence this entry will be changed. */
ptr_to_cell_node Get_initialized_cell_node (voidplist)

{
   ptr_to_cell_node tcell;
   int i;

   if (empty_cell_node == NULL)
   /* create a list of free cell node */
   {
     empty_cell_node = (ptr_to_cell_node)
           calloc (cell_calloc_size, sizeof (struct cell_node));

     for (i=0; i < cell_calloc_size - 1; i++)
     {
        tcell = &(empty_cell_node[i]);
        tcell->next_cell = &(empty_cell_node[i+1]);
     }
     tcell = &(empty_cell_node[cell_calloc_size-1]);
     tcell->next_cell = NULL;
   }

   tcell = empty_cell_node;
   empty_cell_node = empty_cell_node->next_cell;

   tcell->row_first_neigh = tcell->col_first_neigh = NULL;

   tcell->parent =
   tcell->sibling =
   tcell->first_child = NULL;

   tcell->prev_cell =
   tcell->next_cell = tcell;

   tcell->representative =
   tcell->ver_before_rep =
   tcell->lower_content =
   tcell->upper_content =
   tcell->actual_content =
   tcell->max_allow_wastage = -1;

   tcell->cell_size  = tcell->total_children = 0;
```

```
   return (tcell);
} /* Get_initialized_cell_node */

/******************************************************************/

/* Procedure to release a cell node

   Input parameter:
   tcell: the cell node to be free
*/

/* N.B. Entry next_cell is used to link all the free nodes,
        hence this entry will be changed. */

void Free_cell_node (ptr_to_cell_node tcell)
{
   tcell->next_cell = empty_cell_node;
   empty_cell_node = tcell;
} /* Free_cell_node */

/******************************************************************/

/* Get and initialize a tvdrc.

   Output parameter
   it return an initialized tvdrc

   N.B. Entry value in tvdrc_rack is always allocated.

*/

int *Get_initialized_tvdrc (voidplist)
{
   /* The total number of neigh_info_node obtained from calloc
      when there is no more free neigh_info_node. */
   extern int tvdrc_calloc_size;

   ptr_to_tvdrc_rack track;
   ptr_to_all_rack tallrack;
   ptr_to_all_tarray tallarray;

   int i, arraysize, pos, *tarray;
   unsigned total_array;

   if (empty_tvdrc_rack == NULL)
   /* create a list of free tvdrc information node */
   {
      arraysize = max_degree + 1;
      total_array = arraysize * tvdrc_calloc_size;
      empty_tvdrc_rack = (ptr_to_tvdrc_rack)
           calloc (tvdrc_calloc_size, sizeof (struct tvdrc_rack));

      tallrack = (ptr_to_all_rack) malloc (sizeof (struct all_rack));
      tallrack->rack = empty_tvdrc_rack;
      tallrack->next = initialized_rack;
      initialized_rack = tallrack;

      tarray = (int *) calloc (total_array, sizeof (int));

      tallarray = (ptr_to_all_tarray) malloc (sizeof (struct all_tarray));
      tallarray->array = tarray;
      tallarray->next = initialized_array;
      initialized_array = tallarray;

      for (i=0, pos=0; i < tvdrc_calloc_size - 1; i++)
      {
         track = &(empty_tvdrc_rack[i]);
         track->value = &(tarray[pos]);
         pos += arraysize;
         track->next = &(empty_tvdrc_rack[i+1]);
      }
      track = &(empty_tvdrc_rack[tvdrc_calloc_size-1]);
      track->value = &(tarray[pos]);
      track->next = NULL;
   }
```



```
   /* get a free node */
   track = empty_tvdrc_rack;
   empty_tvdrc_rack = track->next;

   tarray = track->value;
   /* initialize each entry */
   /* this may not be necessary */
   /* this entry should not be NULL */
   for (i = 0; i <= max_degree ; i++)
     tarray[i] = 0;

   track->value = NULL;
   track->next = empty_rack;
   empty_rack = track;

   return (tarray);
} /* Get_initialized_tvdrc */

/******************************************************************/

/* Procedure to release a tvdrc node.

   Input parameter:
   tvdrc_ptr: the tvdrcbor node to be free
*/

/* N.B. Entry row_next_tvdrc is used to link all the free nodes,
        hence this entry will be changed. */

void Free_tvdrc (int *tvdrc_ptr)
{
   ptr_to_tvdrc_rack track;

   /* get an empty rack, N.B. there are always empty rack
      available since each tvdrc has a rack. */
   track = empty_rack;
   empty_rack = empty_rack->next;
   track->value = tvdrc_ptr;
   track->next = empty_tvdrc_rack;
   empty_tvdrc_rack = track;
} /* Free_tvdrc */

/******************************************************************/

/* Get and initialize a neigh_info_node.
   All the entry are initialized to NULL.

   Output parameter
   it return an initialized neigh_info_node

*/

ptr_to_neigh_info_node Get_initialized_neigh_info_node (voidplist)
{
   /* The total number of neigh_info_node obtained from calloc
      when there is no more free neigh_info_node.*/
   extern int neigh_calloc_size;

   ptr_to_neigh_info_node tneigh;
   int i;

   if (empty_neigh_info_node == NULL)
   /* create a list of free neighbor information node */
   {
     empty_neigh_info_node = (ptr_to_neigh_info_node)
           calloc (neigh_calloc_size, sizeof (struct neigh_info_node));

     for (i=0; i < neigh_calloc_size - 1; i++)
     {
        tneigh = &(empty_neigh_info_node[i]);
        tneigh->tot_ver_of_deg_from_row_to_col = NULL;
        tneigh->row_next_neigh = &(empty_neigh_info_node[i+1]);
     }
     tneigh = &(empty_neigh_info_node[neigh_calloc_size-1]);
```

```
      tneigh->tot_ver_of_deg_from_row_to_col = NULL;
      tneigh->row_next_neigh = NULL;
    }

   /* get a free node */

   tneigh = empty_neigh_info_node;
   empty_neigh_info_node = tneigh->row_next_neigh;

   /* initialize each entry */
   tneigh->row_cell_num =
   tneigh->col_cell_num = -1;

   tneigh->tot_ver_of_deg_from_row_to_col = NULL;

   tneigh->row_next_neigh =
   tneigh->row_prev_neigh =
   tneigh->col_next_neigh =
   tneigh->col_prev_neigh = tneigh;

   return(tneigh);
} /* Get_initialized_neigh_info_node */

/*********************************************************************/

/* Procedure to release a neigh_info_node.

   Input parameter:
   neigh_ptr: the neighbor node to be free
*/

/* N.B. Entry row_next_neigh is used to link all the free nodes,
       hence this entry will be changed. */

void Free_neigh_info_node (ptr_to_neigh_info_node neigh_ptr)
{
   neigh_ptr->row_next_neigh = empty_neigh_info_node;
   if (neigh_ptr->tot_ver_of_deg_from_row_to_col != NULL)
     Free_tvdrc (neigh_ptr->tot_ver_of_deg_from_row_to_col);
   empty_neigh_info_node = neigh_ptr;
} /* Free_neigh_info_node */

/*********************************************************************/

/* Get and initialize a list_of_cells.

   Output parameter
   it return an initialized tvdrc

   N.B. Entry value in tvdrc_rack is always allocated.

*/

ptr_to_list_of_cells Get_initialized_list_of_cells(voidplist)
{
   /* The total number of neigh_info_node obtained from calloc
      when there is no more free neigh_info_node. */
   extern int list_of_cells_calloc_size;

   ptr_to_list_of_cells tloc;
   int i;

   if (empty_list_of_cells == NULL)
   /* create a list of free list_of_cells information node */
   {
     empty_list_of_cells = (ptr_to_list_of_cells)
           calloc (list_of_cells_calloc_size,
                   sizeof (struct list_of_cells));

     for (i=0; i < list_of_cells_calloc_size - 1; i++)
     {
        tloc = &(empty_list_of_cells[i]);
        tloc->next_entry = &(empty_list_of_cells[i+1]);
     }
     tloc = &(empty_list_of_cells[list_of_cells_calloc_size-1]);
```



```
      tloc->next_entry = NULL;
    }

   tloc = empty_list_of_cells;
   empty_list_of_cells = empty_list_of_cells->next_entry;

   tloc->level = 0;
   tloc->corr_cell = NULL;
   tloc->next_entry = NULL;

   return (tloc);

} /* Get_initialized_list_of_cells */

/*********************************************************************/

/* Procedure to release a list_of_cells node.

   Input parameter:
   loc_ptr: the list_of_cells node to be free
*/

/* N.B. Entry next_entry is used to link all the free nodes,
       hence this entry will be changed. */

void Free_list_of_cells (ptr_to_list_of_cells loc_ptr)
{
   loc_ptr->next_entry = empty_list_of_cells;
   empty_list_of_cells = loc_ptr;
} /* Free_list_of_cells */

/*********************************************************************/

/* This procedure will reinitalize all the variables required to make
   the program run for another set of user input partition cell and
   content given the same graph.
*/

void Reinitiate_memmdom_val (voidplist)

{
   ptr_to_tvdrc_rack track;
   ptr_to_all_rack tallrack, next_tallrack;
   ptr_to_all_tarray tallarray, next_tallarray;

   /* empty_neigh_info_node, empty_cell_node,
      empty_list_of_cells do not need to be reset */

   if (empty_tvdrc_rack != NULL)
   {
       next_tallrack = initialized_rack;
       while (next_tallrack != NULL)
       {
          tallrack = next_tallrack;
          next_tallrack = next_tallrack->next;
          free ((void *) tallrack);
          tallrack = NULL;
       }

       next_tallarray = initialized_array;
       while (next_tallarray != NULL)
       {
          tallarray = next_tallarray;
          next_tallarray = next_tallarray->next;
          free ((void *) tallarray);
          tallarray = NULL;
       }

       empty_tvdrc_rack = NULL;
   }
   initialized_rack = NULL;
   initialized_array = NULL;
   empty_rack = NULL;

} /* Reinitiate_memmdom_val */
```

```
      tneigh->tot_ver_of_deg_from_row_to_col = NULL;
      tneigh->row_next_neigh = NULL;
    }

    /* get a free node */

    tneigh = empty_neigh_info_node;
    empty_neigh_info_node = tneigh->row_next_neigh;

    /* initialize each entry */
    tneigh->row_cell_num =
    tneigh->col_cell_num = -1;

    tneigh->tot_ver_of_deg_from_row_to_col = NULL;

    tneigh->row_next_neigh =
    tneigh->row_prev_neigh =
    tneigh->col_next_neigh =
    tneigh->col_prev_neigh = tneigh;

    return(tneigh);
} /* Get_initialized_neigh_info_node */

/*******************************************************************/

/* Procedure to release a neigh_info_node.

    Input parameter:
    neigh_ptr: the neighbor node to be free
*/

/* N.B. Entry row_next_neigh is used to link all the free nodes,
        hence this entry will be changed. */

void Free_neigh_info_node (ptr_to_neigh_info_node neigh_ptr)
{
    neigh_ptr->row_next_neigh = empty_neigh_info_node;
    if (neigh_ptr->tot_ver_of_deg_from_row_to_col != NULL)
      Free_tvdrc (neigh_ptr->tot_ver_of_deg_from_row_to_col);
    empty_neigh_info_node = neigh_ptr;
} /* Free_neigh_info_node */

/*******************************************************************/

/* Get and initialize a list_of_cells.

    Output parameter
    it return an initialized tvdrc

    N.B. Entry value in tvdrc_rack is always allocated.

*/

ptr_to_list_of_cells Get_initialized_list_of_cells(voidplist)
{
    /* The total number of neigh_info_node obtained from calloc
       when there is no more free neigh_info_node. */
    extern int list_of_cells_calloc_size;

    ptr_to_list_of_cells tloc;
    int i;

    if (empty_list_of_cells == NULL)
    /* create a list of free list_of_cells information node */
    {
      empty_list_of_cells = (ptr_to_list_of_cells)
            calloc (list_of_cells_calloc_size,
                    sizeof (struct list_of_cells));

      for (i=0; i < list_of_cells_calloc_size - 1; i++)
      {
         tloc = &(empty_list_of_cells[i]);
         tloc->next_entry = &(empty_list_of_cells[i+1]);
      }
      tloc = &(empty_list_of_cells[list_of_cells_calloc_size-1]);
```



```
      tloc->next_entry = NULL;
    }

    tloc = empty_list_of_cells;
    empty_list_of_cells = empty_list_of_cells->next_entry;

    tloc->level = 0;
    tloc->corr_cell = NULL;
    tloc->next_entry = NULL;

    return (tloc);

} /* Get_initialized_list_of_cells */

/*******************************************************************/

/* Procedure to release a list_of_cells node.

    Input parameter:
    loc_ptr: the list_of_cells node to be free
*/

/* N.B. Entry next_entry is used to link all the free nodes,
        hence this entry will be changed. */

void Free_list_of_cells (ptr_to_list_of_cells loc_ptr)
{
    loc_ptr->next_entry = empty_list_of_cells;
    empty_list_of_cells = loc_ptr;
} /* Free_list_of_cells */

/*******************************************************************/

/* This procedure will reinitalize all the variables required to make
    the program run for another set of user input partition cell and
    content given the same graph.
*/

void Reinitiate_memmdom_val (voidplist)

{
    ptr_to_tvdrc_rack track;
    ptr_to_all_rack tallrack, next_tallrack;
    ptr_to_all_tarray tallarray, next_tallarray;

    /* empty_neigh_info_node, empty_cell_node,
       empty_list_of_cells do not need to be reset */

    if (empty_tvdrc_rack != NULL)
    {
        next_tallrack = initialized_rack;
        while (next_tallrack != NULL)
        {
           tallrack = next_tallrack;
           next_tallrack = next_tallrack->next;
           free ((void *) tallrack);
           tallrack = NULL;
        }

        next_tallarray = initialized_array;
        while (next_tallarray != NULL)
        {
           tallarray = next_tallarray;
           next_tallarray = next_tallarray->next;
           free ((void *) tallarray);
           tallarray = NULL;
        }

        empty_tvdrc_rack = NULL;
    }
    initialized_rack = NULL;
    initialized_array = NULL;
    empty_rack = NULL;

} /* Reinitiate_memmdom_val */
```

```
/***************************************************************************/
```