2005

Connected Domination in Graphs

Gayathri Mahalingam
University of South Florida

Connected Domination in Graphs

by

Gayathri Mahalingam

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Master of Science
Department of Mathematics
College of Arts and Sciences
University of South Florida

Major Professor: Stephen Suen, Ph.D.
Natasha Jonoska, Ph.D.
Gregory McColm, Ph.D.

Date of Approval:
June 21, 2005

Keywords:
Connected Dominating set, Algorithms, Breadth First Search, Random graphs,
Local optimization.

Dedication

To My Mom

Acknowledgements

First and foremost, I would like to thank my advisor Dr. Stephen Suen. He is a great advisor anyone could have. This thesis wouldn't have been possible without his constant help and valuable suggestions. I am very glad to express my gratitude at this moment to him.

I would also like to thank the other committee members, Dr. Natasha Jonoska, and Dr.Gregory McColm who monitored my work and took effort in reading and providing me with valuable comments on earlier versions of this thesis.

I take great pleasure in thanking all my friends who had helped in many ways. My special thanks goes to Nancy, Denise, Beverly, Mary Ann, Aya and Frances, who work in the Math office.

I am grateful to my brother, sister-in-law, my mother, granny, my uncle and aunt. A very special thanks goes to my sister who has always been my friend, philosopher and guide. She has always been a boost to my mental strength and ability.

Table of Contents

List of Tables

List of Figures

Connected Domination in Graphs

Gayathri Mahalingam

ABSTRACT

A connected dominating set D is a set of vertices of a graph $G = (V, E)$ such that every vertex in $V - D$ is adjacent to at least one vertex in D and the subgraph $\langle D \rangle$ induced by the set D is connected. The connected domination number $\gamma_c(G)$ is the minimum of the cardinalities of the connected dominating sets of G. The problem of finding a minimum connected dominating set D is known to be NP-hard. Many polynomial time algorithms that achieve some approximation factors have been provided earlier in finding a minimum connected dominating set. In this work, we present a survey on known properties of graph domination as well as some approximation algorithms. We implemented some of these algorithms and tested them with random graphs and compared their performance in finding a minimum connected dominating set D. We present the breadth first search algorithm as a heuristic for finding a connected dominating set whose cardinality is hopefully close to that of a minimum connected dominating set. The algorithm finds a spanning tree T of the graph $G = (V, E)$ using breadth first search, and picks up the non-leaf nodes as the connected dominating set D. There are graphs for which the Breadth first search heuristic does not work so well. We implemented some local optimization procedures that would improve the performance of the breadth first search heuristic in finding the minimum connected dominating set D.

Chapter 1

Introduction

1.1 Background of the Dominating Set

The "Five Queens" problem can be said to be the origin of the study of the dominating sets in graphs. The problem of determining the minimum number of queens that can be placed on a chess board, so that all the squares are either attacked by a queen or are occupied by a queen is called the five queen problem or the dominating queen problem. It was shown in 1850's, that five is the minimum number of queens that can dominate all of the squares of a chess board. The dominating queen problem can be stated in general as the domination of vertices of a graph.

A set $D \subseteq V$ of vertices in a graph $G = (V, E)$ is called a dominating set if every vertex $v \in V$ is either in the set D, or is adjacent to a vertex in D. The minimum of the cardinalities of the dominating sets is the domination number of the graph G, denoted by $\gamma(G)$.

Claude Berge in his book [2] defined for the first time the concept of the domination number of a graph. He called this number the "co-efficient of external stability". The term "dominating set" and "domination number" was first used by Ore in his book [17]. In 1977, Cockayne and Hedetniemi[6], published a survey of the few results known at that time about dominating sets in graphs. In that survey paper, Cockayne and Hedetniemi were the first to use the notation $\gamma(G)$ for the domination number of a graph, which subsequently became the accepted notation.

The problem of finding dominating sets in a graph G is applied in a variety of situations. The concept of domination is mainly used in network problems like, com-

puter communication networks, in which a computer network is modeled by a graph $G = (V, E)$, for which the vertices represent the computers and the edges represent direct communication links between pairs of computers. Each processor passes information to other processors connected to it. Thus the information is collected from all the processors. This is done by passing the information from each processor to one of the small set of collecting processors. The collecting processors form the dominating set, and the problem is to find a small set of processors which are connected to all other processors.

Another application worth mentioning is the ad hoc networks. Ad hoc networks are communication systems with no fixed infrastructure. These networks are used in applications such as mobile commerce, search and rescue, and military battlefields. In these networks, the information is passed between hosts in the network. The information is collected at selected hosts in the network called "virtual backbone" of the network. The problem of finding a minimum size backbone in ad hoc networks can be reduced to the problem of finding a minimum connected dominating set in a connected graph G.

Finding a dominating set with minimum cardinality, for an arbitrary graph was shown to be NP-complete. Garey and Johnson [9] in their book on NP-completeness, showed that finding a minimal dominating set is NP-complete. This is denoted as the DOMINATING SET problem. Therefore, no known polynomial time algorithm exists for determining the domination number of an arbitrary graph. If we expect to be able to find a polynomial time algorithm to compute the domination number of a graph, then we would have to restrict the instances to classes of graph instead of arbitrary graphs. The DOMINATING SET problem remains NP-complete even when instances are restricted to certain classes of graphs. Harary and Haynes defined the *conditional domination number*, denoted by $\gamma(G : P)$ as the smallest cardinality of a dominating set $D \subseteq V$, such that the subgraph $\langle D \rangle$ induced by the set D satisfies the property P. Some of the properties have been listed here.

- P1. $\langle D \rangle$ has no isolated vertices.

- P2. $\langle D \rangle$ is connected.

Not every graph satisfies these properties. A dominating set D, and hence the graph G must not have isolated vertices in order to satisfy the property $P1$. It shows that the graph G must be connected in order to have a dominating set, that satisfies property $P2$. Property $P1$ leads to a new domination called *total (open) domination*. A dominating set D is a total dominating set, if $V = N(D)$. The minimum of the cardinalities of the total dominating set gives the total domination number, denoted as $\gamma_t(G)$.

Sampathkumar and Walikar [20] defined the dominating set D to be a *connected dominating set*, if the induced subgraph $\langle D \rangle$ is connected (property $P2$). Since, a connected dominating set includes at least one vertex from each component of G, there exists a connected dominating set if and only if G is connected. The minimum of the cardinalities of the connected dominating sets is the connected domination number, denoted by $\gamma_c(G)$. It is obvious that $\gamma(G) \leq \gamma_c(G)$. Garey and Johnson [9] showed that the problem of finding a minimum connected dominating set is NP-complete. Pfaff, Laskar, and Hedetniemi [18] showed that CONNECTED DOMINATING SET is NP-complete for bipartite graphs.

As the problem is NP-complete, many upper and lower bounds were computed for the connected domination number for an arbitrary graph. For example, many bounds were computed by Sampathkumar and Walikar [20]. Kleitman and West [15] studied connected graphs that have spanning trees with many leaves. The connected domination number of a spanning tree of a graph G is the number of non-leaf nodes in the tree. Hence, finding the minimum connected dominating set D is equivalent to find a spanning tree of G with maximum number of leaves. The results of Kleitman and West [15] give several bounds for $\gamma_c(G)$. In [5], Caro, West and Yuster gave an upper bound which is an improvement of the result of Kleitman and West, and is asymptotically sharp.

In Chapter 2, we present various known bounds on the domination number as well as the connected domination number of a undirected simple graph G. The graphs

for which the upper and the lower bounds are attained are also discussed there. The bounds are given in terms of the order of the graph, the maximum and the minimum degree of the graph.

In Chapter 3, we present some earlier known approximation algorithms to find the connected dominating set D in an undirected simple graph G. The NP-completeness of the DOMINATING SET problem is also discussed in section 3.1. We present a proof as stated in [13] to show that the dominating set problem is NP-complete. In section 3.3, some special graphs for which a connected dominating set can be found in polynomial time are discussed.

In Chapter 4, we present our results. We present in Section 4.1 the Breadth First Search (BFS) as a heuristic in finding the connected dominating set of an undirected simple graph G. The connected dominating set of the graph G is the set of non-leaf vertices in the BFS tree. In Section 4.2, two local optimization procedures are given that, when implemented on a tree (here it is BFS tree with an arbitrary vertex as the root), will improve the number of vertices in the connected dominating set of the graph G. The first optimization procedure is the "Internal-opt", in which we aim to turn the internal vertices of the tree T to leaves in order to gain more leaves in the tree T. The second procedure is the "Leaf-opt" in which we pick the leaves of the BFS tree and turn it into an internal vertex to gain more leaves in the tree T. Finally, in Section 4.3, we tabulate the experimental results obtained by implementing the Breadth First Search together with the local optimization procedures on random graphs and random regular graphs. A comparison of our results with the results of the earlier know algorithms given by Guha and Khuller[11] and by Duckworth and Mans[7] is also presented in this section.

Chapter 2

Bounds on the Domination number

In the study of subsets of a given type in a graph, it is natural to find either a smallest or a largest such set in a graph. For instance, in an ad hoc network, a set of computers called clients are connected to a set of processors called backbone servers that collect the data together from the clients and transfers the same to the other clients in the network. It is desirable to find the minimum number of backbone servers needed to connect all the clients in the network. Since these subset problems are NP-complete, it is natural to find reasonable upper and lower bounds for these numbers. In this chapter, we discuss the bounds for the domination number $\gamma(G)$ and the connected domination number $\gamma_c(G)$.

2.1 Elementary Properties

A dominating set D is a *minimal dominating set* if no proper subset $D' \subsetneq D$ is a dominating set. A dominating set in the worst case can have all the vertices of the graph. It is also required that there must be at least one vertex in the dominating set D. Hence we have

$$1 \leq \gamma(G) \leq n$$

where n is the number of vertices in the graph G. The lower bound is achieved if and only if the graph G has a vertex of degree $n-1$. For example, a complete graph K_n with n vertices has a domination number 1. The upper bound is achieved if and only if $G = \overline{K_n}$, that is, G is a set of isolated vertices. For a graph with no isolated vertices, the upper bound was much improved in an earlier result which was due to

Ore. The bound is given in theorem 2.1.2.

Theorem 2.1.1 [17] *A dominating set D of a connected graph $G = (V, E)$ is a minimal dominating set if and only if for each vertex in $u \in D$, one of two conditions holds:*

1. *u is not adjacent to any vertex in D.*
2. *there exists a vertex $v \in V - D$ for which $N(v) \cap D = \{u\}$.*

Proof: Since a minimal dominating set includes at least one vertex from every component of G, it is clear that the vertex u is either a component of $\langle D \rangle$, that is, u is an isolated vertex in D. Or, since D is a minimal dominating set, there exists a vertex $v \notin D$, such that v is adjacent to only u in D. Thus, $N(v) \cap D = \{u\}$. Conversely, if the above two conditions hold and assume that D is not a minimal dominating set, then there exists a vertex $u \in D$ such that $D - \{u\}$ is a dominating set. Therefore, u is adjacent to at least one vertex in D, and hence condition 1 of theorem 2.1.1 does not hold. Also, every vertex in $V - D$ is adjacent to at least one vertex in $D - \{u\}$. That is, condition 2 of theorem 2.1.1 does not hold for u, thus contradicting the assumptions. Hence D is a minimal dominating set. □

A direct consequence of condition 2 of theorem 2.1.1 is the following theorem given by Ore.

Theorem 2.1.2 [17]

For any graph G without isolated vertices, $\gamma(G) \leq n/2$, where n is the number of vertices in the graph G.

Proof: For a minimal dominating set D, we shall show that $V - D$ is also a dominating set.

(1) By condition 1 of theorem 2.1.1, if a vertex $u \in D$ is not adjacent to any vertex in D then, u must be adjacent to a vertex in $V - D$, since G has no isolated vertices. This implies that u is dominated by a vertex in $V - D$.

(2) If u is a vertex in D that satisfies condition 2 of theorem 2.1.1 then it is obvious that u is dominated by a vertex in $V - D$.

Since every vertex $u \in D$ must satisfy (1) and (2) (by theorem 2.1.1) for a graph G with no isolated vertices, it is seen that the vertices in D are dominated by vertices in $V - D$. Hence, $V - D$ is also a dominating set. This shows that either $|D|$ or $|V - D|$ is at most $n/2$. Hence, the domination number of G is at most $n/2$.

The upper bound is achieved if every component of G is a 4-cycle or G is a special kind of corona graph. The corona of two graphs G_1 and G_2, as defined by Frucht and Harary [8], is the graph $G = G_1 \circ G_2$ formed from one copy of G_1 and $|V(G_1)|$ copies of G_2 where the ith vertex of G_1 is adjacent to every vertex in the ith copy of G_2. The following theorem illustrates this result.

Theorem 2.1.3 *[13] If G is a graph with n vertices, where n is even, and G has no isolated vertices, then $\gamma(G) = n/2$ if and only if the components of G are the cycle C_4 or the corona $H \circ K_1$ for any connected graph H.*

The corona $H \circ K_1$, is the graph in which there is a pendant edge $\{v, v'\}$ between each vertex $v \in V(H)$ and a new vertex v'. Hence, $H \circ K_1$ has an even order and achieves the upper bound $\gamma(G) = n/2$.

The connected domination number of a complete graph is given by $\gamma_c(K_n) = 1$. Since a connected dominating set is necessarily a dominating set, Sampathkumar and Walikar[20] proved the following result.

Theorem 2.1.4 *If G is a connected graph, then,*

$$\gamma(G) \leq \gamma_c(G) \leq 3\gamma(G) - 2.$$

Proof: Let D be a dominating set, and $|D| = \gamma(G)$ be the domination number. Let m be the number of components of the subgraph $\langle D \rangle$ induced by the dominating set D. It is clear that $\gamma(G) \geq m$. We shall show that there exists two components (say C_i and C_j, where $i \neq j$) of $\langle D \rangle$ such that the length of a shortest path between C_i and C_j is at most 3 in G. Suppose for the purpose of contradiction the shortest path

between any pair of disjoint components has length at least 4. Now, let P be the shortest path between the components of $\langle D \rangle$. In other words, let P be the shortest of all the shortest paths between any two distinct components of $\langle D \rangle$. Then there exists a vertex v in the path P such that v is at a distance of at least two from the end points of P. Since D is a dominating set, the vertex v must be at a distance of at most one from a component. This gives us that v lies on a path P' between two components such that P' is shorter than P, thus contradicting the assumption on P. Thus, there exists two components C_i and C_j with a path of length at most 3. Adding the vertices in the path to D decreases the number of components in $\langle D \rangle$ by one. This procedure can be repeated until there is only one component in D, thus resulting in a connected dominating set. Note that, at most $2(m-1)$ vertices are added to D to form a connected dominating set. Hence, the connected domination number

$$\begin{aligned} \gamma_c(G) &\leq |D| + 2(m-1) \\ &\leq \gamma(G) + 2(\gamma(G) - 1) \\ &\leq 3\gamma(G) - 2. \square \end{aligned}$$

Theorem 2.1.5 *Let G be a connected graph with n vertices and a maximum degree $\triangle(G)$, then,*

$$\lceil \tfrac{n}{\triangle(G)+1} \rceil \leq \gamma(G) \leq \gamma_c(G) \leq n - \triangle(G).$$

Proof: Every vertex in the graph G can dominate at most $\triangle(G)$ vertices and itself. Hence, $\gamma(G) \geq \frac{n}{\triangle(G)+1}$.

To prove the upper bound, let v be a vertex with maximum degree $\triangle(G)$ in G. Form a spanning tree T of G such that every neighbor of v in G is also a neighbor of v in T. This will result in a tree T with $N(v)$ branches in it and hence with at least $\triangle(G)$ leaves. Hence, the connected domination number is at most $n - \triangle(G)$. $\square$

The lower bound of the above theorem is achieved if $N[u] \cap N[v] = \emptyset$ for $u, v \in D$, the dominating set of the graph G and, $|N(v)| = \triangle(G)$ for all $v \in D$. The graphs for which the upper bound is attained is given in the next theorem.

Theorem 2.1.6 [*14*] *For any tree T with n vertices and maximum vertex degree $\triangle(T)$,*

$$\gamma_c(T) = n - \triangle(T)$$

if and only if T is a spider (a tree in which there is at most one vertex with degree not less than 3).

Proof: Let v be a vertex with maximum degree $\triangle(T)$ in a tree T. If T is a spider with v as the root, then we see that the tree T has exactly $\delta(T)$ branches from v (since vertices in each of these branches has a degree less than 3 and T is a tree). Thus the number of leaves in the tree T is exactly $\delta(T)$. Hence the connected domination number $\gamma_c(T) = n - \triangle(T)$.

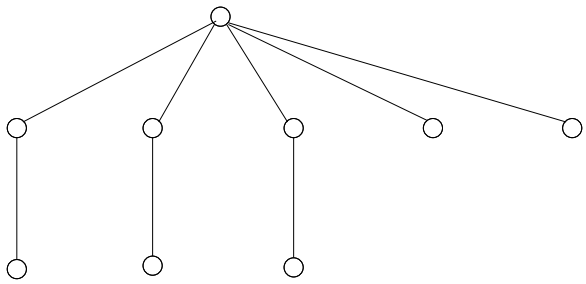

Figure 2.1: A spider

Conversely, if T is not a spider, then there exists a vertex other than v with degree not less than 3 in T. Therefore, the tree T has a branch with more than one leaf in it. This shows that the tree T has more than $\triangle(T)$ leaves. Hence, if T is not a spider then $\gamma_c(T) < n - \triangle(T)$. □

2.2 Bounds in terms of Order and Minimum degree

The bounds in this section are grouped as those bounds which are optimal when $\delta(G)$ is small, where $\delta(G)$ is the minimum vertex degree of a graph G, and the bounds which are nearly optimal, when $\delta(G)$ is large.

West and Kleitman [15] proved some bounds on $\gamma_c(G)$ which were proven to be nearly optimal when k is small where, $\delta(G) \geq k$ for some integer k. They gave an algorithmic proof that finds a spanning tree with many leaves in a connected simple graph G. For a cyclic graph C_n with n vertices we can guarantee only two leaves for the spanning tree T. West and Kleitman [15] considered graphs in which every vertex have degree at least k, that is $\delta(G) \geq k$.

Let $\mathbf{G}_{n,k}$ denote the collection of connected simple graphs with n vertices and a minimum degree at least k. Let $l(n,k)$ denote the maximum m such that every graph in $\mathbf{G}_{n,k}$ has a tree with at least m leaves. Notice that every tree has at least two leaves, and hence $l(n,k) \geq 2$. Since every spanning tree of C_n, a cycle with n vertices, has exactly 2 leaves, We see that $\gamma(C_n) = 2$ and hence $l(n,2) = 2$.

Theorem 2.2.1 [*15*] *For any connected graph G with $k \geq 3$, there is at least one spanning tree that satisfies,*

$$l(n,k) \leq n - 3\lfloor n/(k+1) \rfloor + 2$$

where $\delta(G) \geq k$.

For $k \geq 3$, a simple construction yields a $G \in \mathbf{G}_{n,k}$ in which every spanning tree has a maximum of $n - 3\lfloor n/(k+1) \rfloor + 2$ leaves for an arbitrary k. The bound in theorem 2.2.1 is achieved when $k \leq 4$ and $k+1$ divides n. The construction as explained in [15] is as follows.

Proof: A graph $G_{n,k} \in \mathbf{G}_{n,k}$ for which every spanning tree with a maximum of $n - 3\lfloor n/(k+1) \rfloor + 2$ leaves can be constructed. Let $m = \lfloor n/(k+1) \rfloor$, and let $r = n - m(k+1)$. The vertex set of G is partitioned into sets $R_0, R_1, \ldots, R_{m-1}$ where $|R_0| = k + 1 + r$, and $|R_i| = k + 1$ for $i \neq 0$. For every i, pick two vertices, $x_i, y_i \in R_i$ and add edges between all pairs of vertices in R_i other than $\{x_i, y_i\}$. Now, let $Z = \{\{x_i,\ y_{(i+1) \bmod \mathrm{m}}\} : 0 \leq i \leq m\}$. The edges in Z are added to $G_{n,k}$. The edges in Z connect two sets R_i and $R_{(i+1) \bmod \mathrm{m}}$, and any two edges in Z form an edge cut (see appendix A for definition of an edge cut). Let $W = \{x_i,\ y_i,\ 0 \leq i \leq m\}$. It suffices to show that any spanning tree T of $G_{n,k}$ has at most $n - 3m + 2$ leaves.

Since any pair of edges in Z form an edge cut, at most one edge of Z is not in T. If $\{x_j,\ y_{j+1}\} \notin T$, for some j, then T must have a path from x_i to y_i in R_i, for every i, in order to remain connected. This is because $|R_i| \geq 3$. Hence R_i, for all i, must have a non-leaf vertex other than x_i and $y_i \in R_i$. Also, every vertex of the W must be a non-leaf except perhaps $x_j,\ y_{j+1}$. Thus, the tree T has at least $3m - 2$ internal vertices, and hence at most $n - 3m + 2$ leaves.

If T includes all the edges in Z, then there is no $x_i y_i$-path in R_i for one i (say j) in T. Hence, there must be at least $3(m-1)$ non-leaves in $V(G) - R_j$ and since $k \geq 2$, either x_i or y_j must be a non leaf. Therefore, connected the domination number $\gamma_c(G)$ is at most $3\lfloor n/(k+1) \rfloor - 2$. Hence, the maximum number of leaves in the spanning tree is,

$$l(n,k) \leq n - 3(m-1) - 1 = n - 3\lfloor n/(k+1) \rfloor + 2. \ \square$$

Corollary 2.2.2 *[15] For any connected graph G with $k \geq 3$,*

$$\gamma_c(G) \geq 3\lfloor n/(k+1) \rfloor - 2.$$

West and Kleitman[15] provided some algorithms to determine $l(n,k)$. The algorithm in [15] can be used to construct a tree with at least $n/4 + 2$ leaves in any $G \in \mathbf{G}_{n,3}$. The authors also extended their approach and presented an algorithm to construct a tree with at least $(2n+8)/5$ leaves in any $G \in \mathbf{G}_{n,4}$.

Theorem 2.2.3 [15] *Every connected graph G with $k \geq 3$ has a spanning tree with at least $n/4 + 2$ leaves.*

The proof of the theorem 2.2.3 is an algorithmic approach that constructs a spanning tree with the desired number of leaves. For $k = 4$, the optimal bound $l(n, 4) \geq \frac{2}{5}n + \frac{8}{5}$ was proved in Griggs and Wu [10] and in Kleitman and West [15]. Griggs and Wu [10] also proved that $l(n, 5) \geq \frac{3}{6}n + 2$.

The following bounds in this section are nearly optimal when $\delta(G)$ is large, and as $n \to \infty$. The first bound discussed here is given by Alon and Spencer [1]. We also present the probabilistic proof given by Alon and Spencer [1].

Theorem 2.2.4 [1] *If G is a graph with no isolated vertices, then*

$$\gamma(G) \leq \frac{n(1+\ln(\delta(G)+1))}{\delta(G)+1}.$$

Proof: Let $p = \ln(\delta(G) + 1)/(\delta(G) + 1)$. The dominating set D is constructed as follows. We construct a set S such that every vertex in S is selected independently with the probability p. Then, the expected value of the cardinality of S is np. Let B be the set of vertices that are not dominated by any of the vertices in S, that is, $B = V - N[S]$. A vertex v is in B if and only if v and its neighbors are not in S, that is, if and only if $N[v] \not\subseteq D$. Hence the probability that $v \in B$ is $(1 - p)^{(1+deg(v))}$. Since $e^{-p} \geq 1 - p$, and $deg(v) \geq \delta(G)$, it is clear that the probability that v is in B is at most $e^{-p(1+\delta(G))}$. Therefore the expected value of $|B|$ is at most $ne^{-p(1+\delta(G))}$. It is clear that $D = S \cup B$ is a dominating set, and the expected size of D is at most

$$n(p + e^{-p(1+\delta(G))}) = n(1 + \ln(\delta(G) + 1))/(\delta(G) + 1).$$

Since the average cardinality of D is at most $n(1 + \ln(\delta(G) + 1))/(\delta(G) + 1)$, there must be a particular set S with at most this cardinality. □

A non-probabilistic, algorithmic proof of this theorem involves a greedy approach by choosing the vertices for the dominating set one by one, where in each step a vertex

that covers (dominates) the maximum number of yet uncovered (un-dominated) vertices is picked. Let r denote the number of vertices that is not a neighbor of any vertex v chosen so far in the dominating set. Then the cardinality of the sets that includes the uncovered vertex, say u, and its neighbors $N(u)$ is at least $r(\delta(G)+1)$. Hence, there is a vertex v which is adjacent to at least $r(\delta(G)+1)/n$ vertices. The number of uncovered vertices when v is added to the dominating set is given by $r(1-\frac{\delta(G)+1}{n})$. At each step, the number of uncovered vertices decreases by a factor of $r(1-\frac{\delta(G)+1}{n})$, and after $\frac{n}{\delta(G)+1}\ln(\delta(G)+1)$ steps, there will be at most $n/(\delta(G)+1)$ uncovered vertices which when added to the dominating set will give the bound of theorem 2.2.4.

The probabilistic arguments used in the proof of theorem 2.2.4 are further studied by Caro, West and Yuster [5] for connected dominating set. They obtain the following result for connected domination number, which is asymptotically sharp.

Theorem 2.2.5 [5]
For any connected graph G, with n vertices and minimum degree $\delta(G)$,

$$\gamma_c(G) = (1 + o_{\delta(G)}(1))n\frac{\ln(\delta(G)+1)}{\delta(G)+1}.$$

Thus, $\gamma_c(G)$ behaves essentially like $\gamma(G)$ when $\delta(G)$ is sufficiently large.

Bounds on the connected domination number in terms of other graph-theoretic parameters have been studied by different authors. These parameters include the diameter of the graph, denoted by $diam(G)$, girth of the graph, denoted by $g(G)$, etc.

Chapter 3

Algorithms and their Complexities

In this chapter, we discuss some approximation algorithms and their complexities in finding a connected dominating set. We also discuss the best and worst cases for these algorithms. The polynomial time algorithm for a restricted set of instances for a particular application have also been discussed here. We also present some original heuristics and their results in finding a connected dominating set.

3.1 NP-completeness of the Domination Problem

In this section, we consider problems involved in computing $\gamma(G)$ and in finding dominating sets of minimum cardinalities. For any graph $G = (V, E)$ with n vertices, it is clear that the connected domination number lies in the range $1 \leq \gamma(G) \leq n$. Hence, there are only finite number of possible minimum cardinality dominating sets of G.

The simplest procedure would enumerate the 2^n subsets of $V(G)$ and determine whether the enumerated subset $D \subseteq V$ is a connected dominating set and if so, output the cardinality of D as $\gamma(G)$ and halt. Such an algorithm is easy to construct but requires $O(2^n)$ steps in the worst case. That is, it has an exponential time complexity in the order of the graph G.

We are interested to know if there exist an algorithm that finds the value of $\gamma(G)$ for an arbitrary graph G and runs in polynomial time. To date, no one has constructed a domination algorithm that has better than exponential time complexity for arbitrary graphs. Furthermore, the NP-completeness of the dominating set suggests that it is

not likely that a polynomial time algorithm can be constructed.

Garey and Johnson [9] mentioned that the domination problem is NP-complete for arbitrary graphs. The basic complexity questions concerning the decision problem for the domination number takes the following form:

DOMINATING SET
INSTANCE: A graph $G = (V, E)$ and a positive integer k.
QUESTION: Does G have a dominating set of size $\leq k$?.
David Johnson showed that the dominating set is NP-complete. His proof is as follows. (Also see [13] for the proof.)

Theorem 3.1.1 [9] *DOMINATING SET is NP-complete.*

Proof: There are two steps. The first step is to prove that the DOMINATING SET problem resides in the class of NP. This involves an easy verification of a "yes" instance of DOMINATING SET in polynomial time, that is, for a graph $G = (V, E)$, a positive integer k and an arbitrary set $D \subseteq V$ with $|S| \leq k$, it is easy to verify in polynomial time whether D is a dominating set or not. The second step of an NP-completeness proof is to select a known NP-complete problem, and define a transformation from this problem to the DOMINATING SET problem. We use the well known 3-SAT problem here. This problem can be stated in the following form.

3-SAT
INSTANCE: A set $U = \{u_1, u_2, ..., u_n\}$ of variables, and a set $C = \{C_1, C_2, ..., C_m\}$ of 3-element sets, called clauses, where each clause C_i contains three distinct occurrences of either a variable u_i or its compliment u_i'. For example, the clause C_1 in Figure 3.1 is $C_1 = \{u_1, u_2, u_3'\}$.
QUESTION: Does the set C have a satisfying truth assignment, that is, an assignment of True and False to the variables in U such that at least one variable in each clause in C is assigned the value True?
Given an instance C of 3-SAT, we can construct an instance $G(C)$ of DOMINAT-

ING SET as follows. Construct a triangle for each value of u_i with vertices labeled $u_i \vee u_i' \vee v_i$ where $1 \leq i \leq n$. Create a single vertex C_j for each clause $C_j = u_i \vee u_k \vee u_l$ and add edges $\{u_i, C_j\}, \{u_k, C_j\}$ and $\{u_l, C_j\}$.

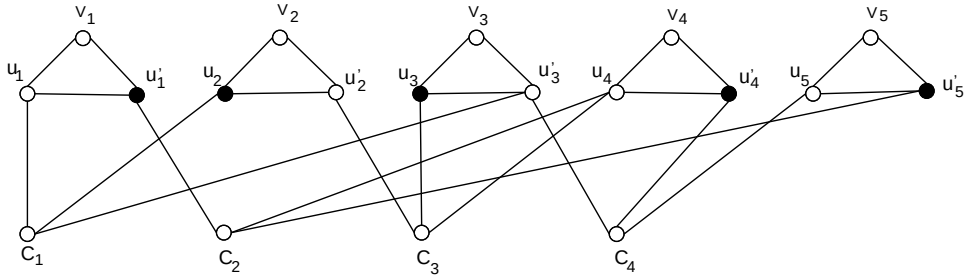

Figure 3.1: Reduction from 3-SAT to DOMINATING SET.

From Figure 3.1, we see that, $C_1 = u_1 \vee u_2 \vee u_3'$, $C_2 = u_1' \vee u_4 \vee u_5'$, $C_3 = u_2' \vee u_3 \vee u_4$, and $C_4 = u_3' \vee u_4' \vee u_5$.

We need to show that C is a "yes" instance of the 3-SAT problem if and only if $G(C)$ is a "yes" instance of the DOMINATING SET, for $k = n$. In other words C has a satisfying truth assignment if and only if the graph $G(C)$ has a dominating set of cardinality at most n.

Let C have a satisfying truth assignment. For example, in Figure 3.1, we see that u_1 =False, u_2 =True, u_3 =True, u_4 =False, u_5 =False. The set D of vertices in $G(C)$ is created such that all the vertices in the set D has a truth assignment. For example, if u_i =True, then $u_i \in D$, and if u_i =False, then $u_i' \in D$. The set D is the dominating set of $G(C)$ since,

- every triangle has exactly one vertex in D, and hence all the vertices in each triangle is either in the set D or is dominated by a vertex in D.
- each clause C_i is dominated by at least one vertex in D. Since, by assumption each clause C_i has a variable with the truth assignment, and those variables are exactly in the set D.

Therefore, the set D is a dominating of $G(C)$ with cardinality n.

Conversely, suppose that $G(C)$ has a dominating set D of cardinality $\leq n$. We need to prove that C has a satisfying truth assignment. Considering the vertices of the form v_i, each of these vertices must be either in the dominating set D or be dominated by a vertex in D, since each triangle must have at least a vertex in D, therefore the cardinality of the set D is at least n, that is , $|D| \geq n$. But, every triangle has exactly one vertex in D. Thus, D contains no clause vertex C_j. Since D is a dominating set by assumption, each clause C_j must be dominated by at least a vertex in D. Hence we can assign a truth assignment for C as follows: each variable u_i is assigned a value True if $u_i \in D$, otherwise u_i is assigned False. It follows clearly that this is a satisfying truth assignment for C.

We must also show that the construction explained by the Figure 3.1, for creating an instance of DOMINATING SET from an instance of 3-SAT, can be carried out in polynomial time. The length of an instance of 3-SAT is given by m sets each of size three plus n variables, that is, $O(3m + n)$. The graph $G(C)$ has $3n + m$ vertices and $3n + 3m$ edges. Hence, the cardinality of $G(C)$ is at most a constant times the cardinality of C, and thus $G(C)$ can be constructed in polynomial time from an instance of 3-SAT problem. □

The DOMINATING SET problem is thus NP-complete for arbitrary graphs. The DOMINATING SET problem remains NP-complete even when instances are restricted to certain classes of graphs. For example, the DOMINATING SET problem is NP-complete for bipartite graphs. Pfaff, Laskar, and Hedetniemi[18] showed that the problem of finding a connected dominating set is NP-complete for bipartite graphs.

3.2 Approximation algorithms

Finding a minimum connected dominating set for an undirected graph was shown to be NP-complete. Hence, many approximation algorithms that run in polynomial time were proposed to find a "near-optimal" solution to the DOMINATING SET problem. In this section, we present some of the approximation algorithms that have been proposed earlier.

Two approximation algorithms for finding the connected dominating set of a graph G was proposed by Guha and Khuller [11]. The first algorithm is a modified greedy algorithm that achieves a ratio $|CDS|/|DS_{OPT}|$ which is approximately $2(1+H(\triangle(G)))$, where H is the harmonic function. The algorithm is explained as follows.

Initially, all vertices are marked as white. As the first step, we select a vertex with the maximum number of white neighbors as a dominating vertex, and mark its neighbors as grey. Iteratively, the grey vertices are *scanned*. The process of *scanning* involves adding the grey vertex to the connected dominating set and coloring all its neighbors to grey (dominated). At each step of the scanning process, we either select the grey vertex or the grey vertex and its white neighbor, whichever dominates more white neighbors. This is called the *look ahead procedure*. At the end, we get a spanning tree T, and the connected dominating set includes the vertices in the tree T that are not leaves.

Theorem 3.2.1 [*11*]
The modified greedy algorithm finds a connected dominating set (CDS) of cardinality,

$$|CDS| \leq 2(1 + H(\triangle(G))).|DS_{OPT}|$$

where,

- DS_{OPT} *is a (not connected) minimum dominating set.*

- *H is the Harmonic function with the property, $H(n) \approx \log(n) + \gamma$, where γ is the Euler function and $\gamma \approx 0.57$.*

The second algorithm proposed by Guha and Khuller [11] achieves an approximation factor of $3 + \ln(\triangle(G))$. There are two phases in this algorithm. In the fist phase, we iteratively select the node with the maximum number of white neighbors as a dominating node. The first phase terminates when there are no white nodes in the graph G. In the second phase, the black nodes, that is, the nodes in the dominating set are connected together by coloring some grey nodes (dominated) black. The set of black nodes gives the connected dominating set of the graph G.

Theorem 3.2.2 [*11*] *The connected dominating set found by the algorithm has a cardinality of at most* $(3 + \ln(\triangle(G))).|CDS_{OPT}|$. *where,* CDS_{OPT} *is the optimal connected dominating set of the graph* G.

Algorithms for Regular graphs

Duckworth and Mans proposed a simple, but efficient randomized greedy algorithm for finding a small connected dominating set of a random regular graph. The model used in [7] to generate a random regular graph was first described by Bollobás. (Also see [3]).

For a d-regular graph on n vertices, dn points are taken in n buckets labeled $1 \ldots n$ with d points in each bucket. Then a disjoint pairing of these dn points is chosen uniformly at random. The buckets are the vertices of the randomly generated graph and each pair represents an edge whose end-points are given by the buckets of the points in the pair. This process is called the pairing process. The algorithm is combined with the pairing process that uniformly at random generates the graph as described above.

The graph being generated is called the *evolving graph*, and a vertex is said to be *saturated* if it has a degree d in the evolving graph. Let $V_i = V_i(t)$ be the set of vertices of degree i of the evolving graph (graph being generated) at time t, and $Y_i = Y_i(t)$ denote $|V_i|$.

The first step of the algorithm involves picking a vertex u from V_0. From this point of the algorithm, the evolving graph has at least one non-saturated vertex whose degree is strictly greater than zero. In the first step, we select a vertex uniformly at random from V_0 and expose all its edges and add u to the dominating set D.

The rest of the algorithm is divided into two stages. In the first stage, we select a vertex, v, uniformly at random from V_1 and then expose one of the edges incident with v to a vertex w (say). If $v \in V_1$, then we add v to D (since v dominates w) and expose the remaining edges of v. If $w \notin V_1$, then the algorithm proceeds without adding v to D.

In the second stage of the algorithm, we denote k to be the current minimum degree of all the vertices that have non-zero degree. If $k = 1$, then we select a vertex u (say), uniformly at random from V_1 and expose its remaining edges. If $k \neq 1$, we select a vertex u from V_k uniformly at random and expose an edge incident with u to v. If $v \in V_1$, then u is added to D, and all the remaining edges incident with u are exposed. Otherwise, the operation terminates without increasing the size of D. The reason behind exposing $k-1$ edges incident with u is to increase the minimum degree of the vertices that have non-zero degree.

A vertex u is added to D if and only if one or more neighbors of u, along the exposed edges have degree 1. this ensures that D is a dominating set of G. Also, since each vertex, u, chosen for possible addition to D, is selected uniformly at random from those vertices of a particular non-zero degree. This ensures that D is connected.

3.3 Algorithms for Special Graphs

The DOMINATING SET problem remains NP-complete when instances are restricted to graphs in most of the classes, while for relatively very few classes we are able to compute $\gamma_c(G)$ in polynomial time.

Paths and Cycles

The connected dominating set of any path includes all the vertices in the path other than the leaves in the path. The connected domination number $\gamma_c(P)$ of any path P with n vertices is $n-2$. An algorithm that runs in polynomial time to find the domination number $\gamma(P)$ of a path P is explained as follows.

We start from the vertex v adjacent to one of the end vertices in the path, and include it in the dominating set D. At each step, we add the vertex which is at a distance of 3 from the vertex added previously to the dominating set, until all the vertices are dominated in the path or in the dominating set D.

The connected domination number of a cycle is simply two vertices less than the number of vertices in the cycle. Since the spanning tree of a cycle is a path, finding a minimum dominating set for any cycle is similar to finding a dominating set of a path. Hence, if G is a path of length n then $\gamma(G) = \lceil \frac{n}{3} \rceil$.

Trees

It is trivial that the connected domination number $\gamma_c(T)$ for any tree T, is simply the number of vertices which remain when all of the end vertices of T are deleted. Many linear algorithms were constructed to find the domination related parameters of a tree T. Mitchell, Cockayne, and Hedetniemi [16] presented a linear algorithm to compute $\gamma(T)$ for an arbitrary un-weighted tree T.

Interval Graphs

It is known that the DOMINATING SET is NP-complete for bi-partite graphs, whereas for interval graphs the DOMINATING SET problem can be solved in polynomial time.

Definition 3.3.1 (Interval Graphs)
A graph $G = (V, E)$ is an interval graph if there is a one-to-one correspondence between its vertices and the intervals on the real line, such that there is an edge between two vertices if and only if their corresponding intervals have a non empty intersection.

We can see that the cycle graphs are not subgraphs of the interval graphs. It was shown in [4] that the problem of determining whether a graph is an interval graph can be done in $o(n)$ time. Also, there exists algorithms that run in polynomial time to find the total domination number $\gamma_t(G)$, and the connected domination number $\gamma_c(G)$. We present here an algorithm as explained by Haynes, Hedetniemi, and Slater [13] to find a minimum connected dominating set.

The collection of intervals of an interval graph $G = (V, E)$ is called the interval model of G and is denoted by I. The coordinates of the intervals are obtained by labeling the end points from left to right by $1, 2, \ldots, 2n$, where n is the number of vertices. The intervals are thus labeled from 1 to n in increasing order of their right end points.

The interval graphs are said to have a *boxicity* of 1. A graph has *boxicity* 1, if it has an intersection model consisting of boxes in 1-dimensional space. For interval graphs the DOMINATING SET problem can be solved in polynomial time. A graph has boxicity 2 if it has an intersection model consisting of boxes in 2-dimensional space. The problem of finding a dominating set for graphs with boxicity 2 is NP-complete.

The interval graph is constructed as follows. Let $V_i = \{1, 2, \ldots, i\}$ and $G_i = \langle V_i \rangle$ be the subgraph of G, induced by the vertices labeled from $1, 2, \ldots, i$. G_i is obtained by adding the vertex i to G_{i-1} and joining it to the vertices in G_{i-1} whose interval intersect with the interval of i. This gives us the "left degree" of the vertex i. Hence

an interval graph can be represented by specifying the left degree of every vertex i in the graph G. This leads to a characterization of this class of graphs.

Theorem 3.3.2 *[19] A graph $G = (V, E)$ with n vertices is an interval graph if and only if there is a labeling from $1, 2, \ldots, n$ such that for $i < j < k$, $\{i,k\} \in E$ implies $\{j, k\} \in E$.*

The following algorithm finds a minimum connected dominating set of an interval graph. The idea behind the algorithm is as follows.

Let $CD(i)$ denote the connected dominating set of G_i, which includes vertex i, and let $LowNbr(i)$ denote the least vertex to which i is adjacent to. Let $MinCD(i)$ denote the $CD(i)$ with minimum cardinality. If $LowNbr(i) = 1$, then vertex i dominates all of the vertices $1, 2, \ldots, i-1$ in G_i and hence $MinCD(i) = \{i\}$. If $LowNbr(i) > 1$, then there must be another vertex other than i in $CD(i)$, which is adjacent to i in G_i. Let j be the maximum vertex in $CD(i) - \{i\}$ such that $LowNbr(j) < LowNbr(i)$. Then, any vertex in G_j which is adjacent to vertex i must also be adjacent to vertex j and hence $CD(i) - \{i\}$ is $CD(j)$.

We conclude this section by presenting the algorithm given in [13] that computes a minimum connected dominating set in a connected interval graph.

For each vertex i, the set $L(i)$ is defined as follows. $L(i) = \{MaxLow(i), \ldots, i\}$, where $MaxLow(i) = \{LowNbr(s) : LowNbr(i) \leq s \leq i\}$.

1. **for** $i = 1$ **to** n **do**

   if $LowNbr(i) = 1$ **then**

   $MinCD(i) = \{i\}$;

   else if $LowNbr(i) > 1$ **then**

   $MinCD(i) = min\{\{i\} \cup MinCD(j) : j < i,\ j$ is adjacent to i, and $LowNbr(j) < LowNbr(i)\}$;

fi;

fi;

od;

2. $MinCD(G) = min\{MinCD(i) : i \in L(n)\}$.

Chapter 4

Breadth First Search

4.1 Breadth First Search as a Heuristic

In this section, we propose the Breadth Fist Search (BFS) as a heuristic in finding a connected dominating set of a random graph. The idea behind BFS is to find a spanning tree T of the graph G, and the connected dominating set will be those non-leaf vertices of the spanning tree T.

Breadth First Search

The BFS constructs a breadth-first tree, initially containing only one vertex called *root*. To keep track of the progress, breadth-first search marks each vertex as *visited* or *unvisited*. The algorithm is given as follows.

1. Initially, all vertices are marked *unvisited*.
2. Choose the *root* as the starting vertex.
3. Mark the *root* as *visited*.
4. Add the *root* at the end of the *queue*, and add it to the tree T.
5. Choose a vertex v from the front of the *queue*, and visit all *unvisited* neighbors of v.
6. For each unvisited neighbor u of v,

   mark u as *visited*.

add u to the end of the *queue*.

add the vertex u, and the edge $\{v, u\}$ to the tree T.

7. Repeat steps 5 and 6 until the *queue* is not empty.

8. The tree T thus obtained is the BFS tree.

The Pseudo code is given in appendix C.

Why breadth first search is good to try?. The BFS expands the tree between the visited and the unvisited vertices uniformly across the breadth of the tree. That is, the algorithm visits all vertices at a distance k from the "root" before visiting any vertex at a distance of $k+1$. The random graph, denoted by $G_{n,p}$ with n vertices and an edge probability p, is generated such that there exists an edge between two vertices independently of other possible edges, with probability p. The graph is generated as follows. For every pair of vertices, a random number between 0 and 1 is generated. There exits an edge between a pair of vertices if the random number generated for that pair is at most the edge probability p. The connected domination number obtained by the BFS with an arbitrary vertex as the root is denoted as $\gamma_{BFS}(G)$. For any connected graph $G_{n,p}$, the following conjecture claims that when n is sufficiently large, then $\gamma_{BFS}(G) \approx \frac{n \ln(d)}{d}$, where d is the average vertex degree of $G_{n,p}$. The argument that we present here uses approximation by solutions of systems of ordinary differential equations, similar to the method used for analyzing random greedy algorithms by Wormald [21].

Conjecture:

Assume that $\frac{np}{\ln(n)} \to \infty$ as $n \to \infty$. Then with probability tending to 1, as $n \to \infty$,

$$\gamma_{BFS}(G_{n,p}) \approx \frac{n \ln(d)}{d}.$$

where, $d = d_n = np = np_n$, and p is the edge probability of the graph $G_{n,p}$.

A heuristic argument for the Conjecture

Let $y = y(x)$, be the number of vertices in the BFS tree, when BFS examines the neighbors of a vertex. Let x be the number of vertices that have been queued and dequeued in the BFS search. Then, the expected number of offsprings in the BFS tree is $(n - y)p$.

At each step of the iteration, we add an expected number of $(n-y)p$ edges to the BFS tree. Hence, the change in the number of vertices in the BFS tree is given by, $\triangle y = (n - y)p$. Since, the change in y is small, we have that,

$$\frac{dy}{dx} = (n - y)p$$

Solving the differential equation, we get,

$$\int \frac{dy}{n - y} = \int p dx,$$

which gives that,

$$-\ln(n - y) = px + c$$

where, since $y(0) = 0$,

$$c = -\ln(n).$$

Therefore,

$$-\ln(n - y) = px - \ln(n), \tag{4.1.1}$$

giving that,

$$px = \ln \frac{n}{n - y}. \tag{4.1.2}$$

Let $(n - y)p = A$, for some large number A. Then,

$$y = n - \frac{An}{d}.$$

where $p = \frac{d}{n}$. Using this in (4.1.2), we get

$$px = \ln \frac{n}{An/d} = \ln(d/A).$$

Therefore,

$$x = \frac{1}{p} \ln \frac{d}{A} = \frac{n}{d}(\ln(\frac{d}{A})) \tag{4.1.3}$$

Notice that the number of internal vertices in the BFS tree is at most x. Thus from equation (4.1.3), when $x = \frac{n}{d} \ln(d/A)$, the number of vertices in the BFS tree is $y = n - \frac{An}{d}$, and the number of internal vertices is at most

$$x = \tfrac{n}{d} \ln(d/A).$$

Since there are An/d vertices not in the BFS tree (so far), and each of these vertices will create at most one new internal vertex when BFS is completed, the number of internal vertices in the final BFS tree is at most

$$x + \frac{An}{d} = \frac{n}{d}(\ln(d) - \ln(A)) + \frac{An}{d}$$

$$= \frac{n}{d}(\ln(d) - \ln(A) + A).$$

Hence, $\gamma_{BFS}(G_{n,p}) \approx \frac{n}{d} \ln(d)$.

Running Time

Let G be a graph with n vertices and m edges. It is well known that BFS traversal of G takes a running time of $O(n + m)$. At each step of the algorithm, we pick the unvisited vertices that are adjacent to a vertex which is visited before. These unvisited vertices are now visited, and are added to the queue. Hence, every vertex is added to the queue exactly once, and takes a running time of $O(1)$. Therefore, the total time it takes to enqueue and dequeue all the vertices is $O(n)$. Since a vertex is dequeued from the queue, its adjacency list is scanned at most once. Scanning the adjacency

list of all the vertices in the queue takes a running time of $O(m)$. Summing up, we get the total running time of BFS to be $O(n+m)$. □

The following example shows that the BFS may not work well for some graphs.

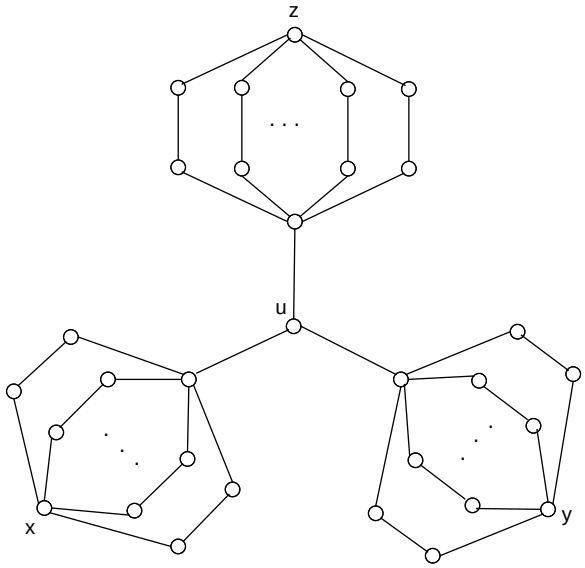

Figure 4.1: Example to show that BFS does not work well with *any* vertex as root.

Let x, y, and z be vertices of degree d. The graph in Figure 4.1 has a solution of size 13, by picking the vertices in the paths from x to u, y to u, and from z to u as the connected dominating set D. Picking any vertex from the graph in the Figure 4.1 as a root will give at least $2(d+2)+1+4$ vertices ($d+2$ from two branches of the vertex u and at least 4 vertices from the third branch) in the connected dominating set D. Thus the performance ratio (from any root) is at least, $\frac{\gamma_{BFS}(G)}{\gamma_c(G)} \geq \frac{2d+9}{13}$ which can be very bad when d is very big.

In view of this bad example, it is desirable to devise some local optimization procedures to the BFS tree so that the connected domination number of the graph G can be improved. In the following section, we discuss these optimization procedures. These local optimization procedures however do not guarantee to increase the number of leaves in the BFS tree for all the graphs G.

4.2 Local Optimization Procedures

In this section, we present two local optimization procedures that aim to increase the number of leaves in the BFS tree, and hence improve the connected domination number $\gamma_c(G)$. The first procedure "Internal-opt" aims to turn the internal vertices of the tree T into a leaf. Whereas, the second procedure "Leaf-opt" aims to gain more leaves in the tree by turning the leaves in the tree T into an internal vertex. The pseudo codes for these algorithms are given in appendix C.

Internal-Opt Procedure

Let T denote the spanning tree of G, with n vertices, and i internal nodes. At each step of the procedure, we pick an internal node and *optimize* it by trying to turn it into a leaf, thus increasing the number of leaves in the tree by 1. *Optimizing* an internal node v involves *joining* of all the branches of v, and removing the edge between v and its neighbors in the tree T until v becomes a leaf. This makes v a leaf at the end.

1. Initially, an internal node v is made the "root" of the tree T. This gives at most $deg_T(v)$ branches in the tree, that is, at most $deg_T(v)$ subtrees, where, $deg_T(v)$ is the degree of the vertex v in the tree T. Initially, there are no *failures* for the vertex v.

2. All these subtrees are joined together by a *joining* process. The *joining* process involves adding a non-tree edge between $u \in N(v)$ or one of the descendants of u with a vertex in another subtree of the tree T with the root v. The *joining* process has the following restrictions in it. The restrictions are designed so that no new internal vertices are created in the joining process.

   (a) Two internal nodes from different subtrees can be joined with a non-tree edge.

(b) If exactly one of the nodes being joined is a leaf, then the leaf must be a neighbor to the vertex v in the tree.

(c) If both the vertices to be joined are leaves, then do nothing.

3. When two vertices in two different subtrees of the tree T are *joined* we get a cycle in the tree, and hence we *remove* an edge between v and one of its neighbors in these two subtrees.

   There are two cases of *removing* an edge between the vertex v, and its neighbor.

   (a) If two internal vertices from different subtrees are joined during the *joining* process, the edge between the vertex v, and any one of its two offsprings in those two subtrees is removed.

   (b) In the case of 2b of the *joining* process, the edge between the vertex v and its neighbor which is a leaf is removed. The joining process for the current branch is continued.

4. If there is a branch that cannot be joined to any of the other branches, then there is a *failure*.

5. Repeat steps 2, 3, and 4 until $deg_T(v) = 1$, and there is not more than one *failure*. If two or more subtrees cannot be joined together then, the vertex v cannot be turned into a leaf. Hence it remains as an internal vertex in the tree T, and the changes made to the tree are discarded.

6. The whole procedure is repeated until all the internal vertices in the tree have been considered.

Case 2b and case 2c of the *joining* process ensures that no leaf in the tree is turned into an internal vertex during Internal-opt procedure.

The *joining* and *removing* processes ensures that no leaf is turned into an internal vertex during the *optimization* of an internal vertex in the tree T, and hence we

do not gain any internal vertices in the tree. At the end, we update the tree T with the current tree if the current tree has more leaves in it. It is noted that if there are no *failures*, then there is always a gain of one leaf to the tree T.

Theorem 4.2.1 *Every internal vertex in T needs only be examined exactly once in Internal-Opt.*

Proof: If there is a failure in optimizing an internal vertex v in the tree T, then it is clear that at least two of the offsprings of the vertex v cannot be removed. Since no leaf is turned into an internal vertex in the process of *joining* and *removing* an edge from the tree T, we gain no internal vertex with the optimization procedure. Therefore, if an internal vertex cannot be turned into a leaf, then it remains as an internal vertex in the tree T. Hence, it suffices to optimize each of the internal vertices of the tree T exactly once. □

Running time

The worst case running time of the Internal-Opt procedure is $O(n^2\triangle^2)$. At each iteration of the optimization procedure, an internal vertex v in T is picked. This incur at most $O(n)$ iterations. Edges between v and its offsprings in T can be removed, and this requires at most $O(\triangle)$ iterations. For each such offspring, either the offspring or one of its descendants is *joined* with a vertex in another branch. To find that neighbor, we perform a breadth first search with the offspring as the "root", and each such BFS requires $O(n)$ steps. The process of joining two branches requires BFS to go through the vertices of one branch, and for each such vertex, look for a neighbor in another branch. Hence, it takes at most $O(n\triangle)$ steps. The joining process requires a total of $O(n + n\triangle)$ iterations. Therefore, the total running time required is at most $O(n^2\triangle^2)$.

The following example shows that the internal optimization does not work so well for some graphs.

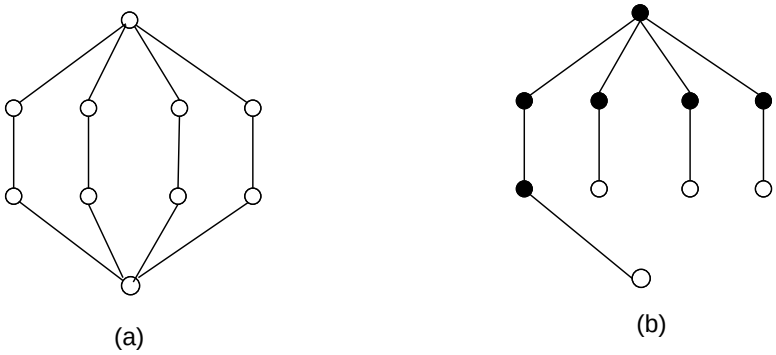

Figure 4.2: Example to show that Internal Optimization fails. (a) Random Graph. (b) BFS tree.

The internal optimization implemented on the BFS tree in figure 4.2(b) does not improve the connected domination number. The BFS tree in figure 4.2(b) shows that any internal vertex in the tree can be turned into a leaf, only by adding an edge between two leaves in the tree, which cannot be done according to the procedure. Hence, we implement another local optimization procedure called the "leaf-opt" to the tree obtained from the internal-opt procedure.

Leaf-Opt Procedure

The second local optimization procedure is called the "leaf-opt". The idea behind this procedure is to turn a leaf of the BFS tree into an internal vertex, so that we gain some more leaves in the BFS tree. In other words, we sacrifice a leaf in the BFS tree in order to gain more leaves in the tree.

Let T denote the current spanning tree of G, with n vertices and l leaves. A leaf x in T is *expanded* if an edge is added to T, from x to any one of its neighbors not in T. When a leaf is *expanded*, the tree T has a cycle in it, and hence we *de-cycle* it. The process of *de-cycling* involves removing an edge in the cycle between two vertices, where at least one of the vertices has a degree two in the tree T. The procedure is explained as follows.

At each step of the procedure, we pick a leaf $x \in T$, and expand it. Now there is a cycle in the tree T, and hence we *de-cycle* the tree T. In the process of de-cycling,

if there are no vertices with degree two, the non-tree edge added during expansion is removed. That is, nothing is done. A leaf is expanded until all its neighbors have been considered in the expansion. The tree T is updated with the current tree if and only if the number of leaves in T is less than that in the current tree. The above procedure is applied to all the leaves in the tree. At the end, we get a spanning tree T with hopefully more leaves, and the non-leaf nodes of T form a connected dominating set.

Since the tree T has at least two leaves in it, the leaf optimization procedure can run indefinitely. To avoid such a situation, we implement the following *stopping rule* on the tree T. The leaf-opt is terminated if it comes across a (current) tree for which the expansion of all its leaves result in no gain in the number of leaves.

It can be easily checked that the leaf-opt performed on the graphs in Figure 4.2 and Figure 4.1 produces a minimum connected dominating set.

Theorem 4.2.2 *The Leaf optimization procedure finds a connected dominating set D in finite number of steps with the stopping rule implemented in it.*

Proof: By definition of the stopping rule, the tree T is updated with the current tree, if and only if there is an overall gain of leaves, that is, if and only if the number of leaves in the tree T increases. Since the graph is finite, and the number of leaves in a tree is bounded from above, the optimization procedure must eventually stop. □

Running time

A simple implementation appears to give a worst case running time of $O(n^3\triangle)$. At each iteration, we choose a leaf for optimization. It is clear that we may have $O(n)$ iterations, since there are $O(n)$ leaves in the tree. In each iteration, every non-tree edge of the leaf is added to the tree T. Hence at each iteration at most $O(\triangle(G))$ edges is added to the tree T. For each edge added, we find a cycle and an edge to be removed from the cycle. At most $O(n)$ steps are used in finding the edge to be removed from the cycle, since there are $O(n)$ edges in the tree. The optimization procedure is repeated until there is no gain of leaves. The tree T is updated at most

n times since each time T is updated, there is a gain in the number of leaves. Hence, the running time is bounded above by $O(n^3\triangle)$.

4.3 Experimental Results

In this section, we discuss the model that we use to generate a random regular graph and a random graph. We implemented the BFS and the local optimization procedures using the programming language C from MS Visual Studio 6 on a computer that has a Pentium IV processor with a processing speed of 2GHz and a memory of 512MB. The results obtained by implementing the Breadth First Search, and the local optimization procedures on random graphs and random regular graphs will discussed here. We also compare our bounds for $\gamma_c(G)$ with other known results and algorithms.

Generation of regular and random graphs

Regular Graph generation

The regular graph is generated uniformly at random by the following procedure.

For a d-regular graph on n vertices, first take dn points in an array of size dn. The array is arranged in ascending order of the dn points, and the first d points correspond to the first vertex, and so on. Use this array to generate a random permutation of the points. Place an edge between the points in the i_{th} and the $(i+1)_{st}$ array, where $i = 1, 3, 5, \ldots, dn-1$. Group the points and the edges corresponding to every vertex to form a $d-$regular graph. We remove the multiple edges and loops in the graph, which gives us a "near" regular graph.

A BFS is performed on the random regular graph, to ensure that the graph is connected. If the graph is not connected, then the whole process is repeated until a connected random regular graph is obtained.

Random Graph Generation

Given the number of vertices n, and the edge probability p of the graph G, a random simple, undirected graph is generated as follows.

At each step of the algorithm, we chose a pair of vertices, and determine whether there is an edge between that pair of vertices. The presence of edge between each pair of vertices is determined by comparing the edge probability p, and a random number generated. The algorithm uses the rand() function of the language C as the random number generator with an initial seed from the user to trigger the random number generator and to generate the random number between 0 and 1. The algorithm proceeds until every pair of vertices have been considered in the graph. The random number generator uses a *seed* to initialize the random number generating function to avoid the generation of the same set of random numbers every time. Thus we can ensure that the random numbers are generated uniformly at random. A random number is generated for every pair of vertices in the graph G.

In the first stage of the algorithm, a pair of vertices, u and v, is chosen from the vertex set $V(G)$ of the graph G. For each pair of vertices $\{u, v\}$, where $v \in V(G) - \{u\}$ a random number between 0 and 1 is generated. There exists an edge between u and v in the graph if and only if the random number generated for the pair u and v is at most the edge probability p of the graph G. This operation is repeated until all pairs of vertices have been considered.

In the second stage of the algorithm, a breadth first search is performed on the random generated graph to ensure that the graph is connected. If the graph is connected, then we retain the adjacency list of the graph. The above algorithm is repeated until a connected simple graph is obtained.

Breadth-First Search and Local Optimization

1. Given a randomly generated graph G as input, we arbitrarily choose a vertex as the root and perform the BFS on the input graph. The BFS tree thus obtained is retained as an input to the local optimization procedure, and the number of internal vertices in the BFS tree gives the $\gamma_{BFS}(G)$.

2. The $\gamma_{BFS}(G)$ can be further improved by applying the local optimization procedures to the BFS tree. A question arises about which procedure to be im-

plemented first. We implement the Internal-opt first since it aims to reduce $\gamma_{BFS}(G)$ without sacrificing any leaves from the BFS tree, while the Leaf-opt aims to reduce $\gamma_{BFS}(G)$ by sacrificing one or more leaves in the BFS tree.

3. The Internal-opt is performed on the BFS tree by choosing an internal vertex from the BFS tree. The tree is updated only if there is a gain of leaves in the tree.

4. The Leaf-opt procedure takes the tree obtained from the Internal-opt as its input. The output of the leaf-opt procedure is a spanning tree with possibly more number of leaves than the spanning tree obtained from the Internal-opt procedure.

5. A BFS is performed on the spanning tree obtained from the optimization procedures. This ensures that the spanning tree produced by the optimization procedures are connected and has the same number of leaves as given by the leaf-opt procedure.

Implementation Results

The following table lists the bounds on the connected domination number $\gamma_c(G)$ for random d-regular graphs.

d	Experimental bound	Duckworth and Mans bound	$\frac{n(\ln(\delta(G)+1))}{\delta(G)+1}$
3	0.542592n	0.5854n	0.3466n
4	0.413667n	0.4565n	0.3219n
5	0.342275n	0.3860n	0.2986n
10	0.212892n	0.2397n	0.2180n
20	0.132000n	0.1493n	0.1450n
30	0.097333n	0.1121n	0.1108n
40	0.078308n	0.0910n	0.0906n
50	0.066183n	0.0771n	0.0771n

Table 4.1: The Bounds on $\gamma_c(G)$ for a random $d-$regular graph G.

The second column, Experimental bound in Table 4.1 summarize our results on the upper bounds on $\gamma_c(G)$ for a random d-regular graph G with n vertices. These results were obtained by taking the average of the bounds obtained for graphs with vertices ranging from $500-2000$ with an increment of 500. For each d we also include the bounds given in [7] in third column, and the value of $\frac{\ln(\delta(G)+1)n}{\delta(G)+1}$ in the fourth column of the table 4.1. We use d for $\delta(G)$. The bounds given by Duckworth and Mans [7] were proven to be asymptotically almost sure upper bounds on the connected domination number of random regular graphs.

Comparing our bounds with the bounds in [7] and [1], we see that the our bounds are similar to the bounds in [7] and [1]. This shows that BFS together with the local optimization procedures work well in finding a connected dominating set.

The following table lists the experimental bounds and the earlier known bounds on the connected domination number $\gamma_c(G)$ of a random graph G.

Edge Probability p	n	Experimental Bound	$\frac{n(\ln(\delta(G)+1))}{\delta(G)+1}$	Guha and Khuller's Bound
$\frac{(\ln(n)+6)}{n}$	500	0.178000n	0.271012n	0.248400n
	1000	0.170400n	0.261181n	0.239000n
	1500	0.169733n	0.255790n	0.239200n
	2000	0.162200n	0.252113n	0.232800n
$\frac{\ln(n)^2}{n}$	500	0.077600n	0.118102n	0.103600n
	1000	0.067600n	0.100292n	0.089400n
	1500	0.063067n	0.091733n	0.082133n
	2000	0.058400n	0.086326n	0.078000n
$\frac{ln(n)^3}{n}$	500	0.015600n	0.026906n	0.019200n
	1000	0.013000n	0.020570n	0.017000n
	1500	0.011867n	0.017778n	0.015067n
	2000	0.010600n	0.016102n	0.013100n

Table 4.2: Bounds on the $\gamma_c(G)$ of a random graph G.

The first column in table 4.2 gives the various edge probabilities p used in the generation of a random graph $G_{n,p}$ with n vertices. The third column lists our bounds

obtained by implementing the BFS together with the local optimization procedures. We have also provided the Alon and Spencer bound [1] in comparison with our bound. We also implemented the algorithm given by Guha and Khuller [11], which uses a *look ahead procedure* in finding a connected dominating set of a random graph $G_{n,p}$. The experimental bounds obtained from the implementation of this algorithm is listed in the last column of the table 4.2. For each value of n we generated 5 graphs and performed the BFS and the local optimization procedures.

Edge probability	n	$\gamma_{BFS}(G)$	$\gamma_{IOPT}(G)$	$\gamma_{LOPT}(G)$	$\frac{\ln(d)}{d}$
$\frac{(\ln(n)+6)}{n}$	2000	455	327	326	383
$\frac{\ln(n)^2}{n}$	2000	154	115	114	140
$\frac{ln(n)^3}{n}$	2000	27	21	21	27

Table 4.3: The connected domination numbers of the optimization procedures.

We use γ_{IOPT} and γ_{LOPT} respectively to denote the connected domination number obtained from the Internal-opt and the Leaf-opt procedures. In the table 4.3, we list a few of the values of the connected domination number obtained from the BFS and the local optimization procedures for a random graph $G_{n,p}$ for various edge probabilities. Comparing the value of $\gamma_{BFS}(G)$ and the value of $\ln(d)/d$ from the table 4.3, we see that the experimental results are similar to the theoretical bound when the graph is not too sparse. (See the rows 3 and 4 of the table 4.3.) The experimental results show that the Internal-opt works well in improving the connected domination number. From the table 4.3, we see that $\gamma_{LOPT}(G)$ is similar to γ_{IOPT}.

Conclusion

We have presented the Breadth First Search (BFS) as a heuristic in finding the connected dominating set D of a simple undirected connected graph G. We have made a heuristic argument showing that BFS gives a connected dominating set whose cardinality is similar to the Alon-Spencer bound. It remains to prove the conjecture and hence show that the BFS gives a bound similar to the bound given by Alon and Spencer [1].

There are graphs for which the BFS gives a bad performance ratio in finding a minimum connected dominating set (see Figure 4.1). Two local optimization procedures that aim to increase the number of leaves in the BFS tree, and hence the connected domination number, have been proposed in this work. Providing a theoretical proof to show that these local optimization procedures will significantly improve the result of BFS needs to be investigated.

The experimental results suggest that the BFS and the local optimization procedures can work well at least for random graphs $G_{n,p}$ and random regular graphs. It was also shown that there are graphs for which the Internal-opt procedure does not give any improvement in the number of leaves in the BFS tree. Although the BFS and the local optimization procedures are polynomial time algorithms, they do not guarantee an optimal solution for the bound on $\gamma_c(G)$. Further investigation is needed to see how well the local optimization procedures perform in the worst case.

Appendix A

Definitions and Notations

For a graph $G = (V, E)$, we let n and m denote the order and size of G, respectively, that is, the number of vertices is $n = |V|$ and the number of edges is $m = |E|$. we use $\langle D \rangle$ to denote the subgraph induced by the set S.

Definition A.0.1 ($N(v)$ **and** $N[v]$)
The open neighborhood of $v \in V(G)$ is the set of vertices adjacent to v, $N(v) = \{w|\{v, w\} \in E(G)\}$, and the closed neighborhood of v is $N[v] = N(v) \cup \{v\}$.

Definition A.0.2 ($N(S)$ **and** $N[S]$)
For a set $S \subseteq V(G)$, $N(S) = \cup_{s \in S} N(s)$ and $N[S] = \cup_{s \in S} N[s]$.

Definition A.0.3 $deg(v)$
The degree of a vertex $v \in G$ is the number vertices adjacent to the vertex v in G. It is denoted by $deg(v) = |N(v)|$.

Definition A.0.4 ($\delta(G)$ **and** $\triangle(G)$)
The minimum degree of G is $\delta(G) = min\{deg(v) : v \in V(G)\}$ and the maximum degree of G is $\triangle(G) = max\{deg(v) : v \in V(G)\}$.

Definition A.0.5 (Induced Subgraph)
An induced subgraph is a subset, S, of the vertices of a graph G together with any edges of G, whose endpoints are both in this subset. That is,

$$\langle S\rangle = (S, E_S), E_S = \{\{a, b\} : a, b \in S, \{a, b\} \in E(G)\}$$

Definition A.0.6 (Diameter of a graph)

The diameter of a connected graph G is the least integer D such that for all vertices u and v in G we have $d(u, v) \leq D$, where $d(u, v)$ denotes the *distance* from u to v in G, that is, the length of the shortest path between u and v.

Definition A.0.7 (Edge Cut)

An *edge cut* for two vertices u and v is the set of edges whose removal from the graph G disconnects u and v. An edge cut for the whole graph G is the set of edges whose removal renders the graph disconnected.

Definition A.0.8 (Dominating set)

A set of vertices D is a *Dominating set* of a graph $G = (V, E)$, is every vertex in $V - D$ is adjacent to at least one vertex in D. The domination number $\gamma(G)$ of a graph G is the minimum of the cardinalities of the dominating sets of the graph G.

Definition A.0.9 (Total Dominating set)

A set D is a total dominating set, also called an open dominating set, if for every vertex $u \in V$ there exists a vertex $v \in D$, such that u is adjacent to v. The total(open) domination number of a graph G is $\gamma_t(G) = \min\{|D| : D \subseteq V(G) \text{ and } V = N(D)\}$.

Definition A.0.10 (Connected Dominating set)

A set $D \subseteq V$ of vertices in any connected graph G is called the *Connected Dominating set* of the graph G if the subgraph $\langle D\rangle$ induced by the set of vertices D is connected. The minimum of the cardinalities of the connected dominating sets of G is called the connected domination number $\gamma_c(G)$ of the graph G.

Since a dominating set must contain at least one vertex from every component of G, there exists a connected dominating set for a graph G if and only if G is connected.

Appendix B

Tabulation of Experimental results

The table B.1 and table B.2 illustrates the elaborate results obtained by implementing the optimization procedures. The connected domination number obtained by the BFS together with the local optimization procedures is denoted as $\gamma_{BFSL}(G)$. The results indicate that the bounds on $\gamma_B FSL(G)$, do not vary much from the bound listed in table 4.1. The table B.1 and the table **??**, lists the degree d of the random regular graph, number of vertices n, average of 5 $\gamma_{BFSL}(G)$ for given n, the average bound for each n.

d	n	$\gamma_{BFSL}(G)$					Average $\frac{\gamma_{BFSL}(G)}{n}$
3	500	274	270	271	270	269	0.5416
	1000	539	547	538	551	541	0.5432
	1500	817	809	817	821	815	0.5438
	2000	1081	1084	1082	1084	1086	0.5417
4	500	212	205	206	210	208	0.4164
	1000	410	416	414	403	416	0.4118
	1500	621	621	621	617	627	0.4142
	2000	820	822	827	818	835	0.4122
5	500	174	171	165	178	169	0.3436
	1000	340	344	345	340	343	0.3424
	1500	505	515	509	517	516	0.3416
	2000	687	690	681	685	680	0.3423

Table B.1: The connected domination number $\gamma_{BFSL}(G)$ for a $d-$regular graph G.

d	n	$\gamma_{BFSL}(G)$					Average $\frac{\gamma_{BFSL}(G)}{n}$
10	500	108	106	105	111	105	0.2140
	1000	211	213	211	211	207	0.2106
	1500	322	321	326	321	314	0.2138
	2000	435	419	425	428	424	0.2131
20	500	67	68	68	66	66	0.1340
	1000	138	131	135	130	132	0.1332
	1500	192	193	198	193	193	0.1292
	2000	266	260	259	266	265	0.1316
30	500	49	49	48	48	51	0.0980
	1000	97	98	100	93	92	0.0960
	1500	143	145	150	143	149	0.0973
	2000	191	198	199	195	197	0.0980
40	500	38	39	40	42	40	0.0796
	1000	73	81	75	75	80	0.0768
	1500	121	120	114	120	117	0.0789
	2000	159	158	156	152	154	0.0779
50	500	32	32	34	33	36	0.0668
	1000	67	67	64	66	66	0.066
	1500	98	99	96	100	100	0.0657
	2000	132	131	136	128	135	0.0662

Table B.2: The connected domination number $\gamma_{BFSL}(G)$ for a $d-$regular graph G.

Edge Probability	n	$\gamma_{BFSL}(G)$					Average $\frac{\gamma_{BFSL}(G)}{n}$
$\frac{(\ln(n)+6)}{n}$	500	89	92	87	91	86	0.17800
	1000	173	168	168	173	170	0.17040
	1500	252	251	254	259	257	0.16973
	2000	328	326	329	318	321	0.16220
$\frac{(\ln(n)^2)}{n}$	500	40	39	38	37	40	0.07760
	1000	64	71	67	68	68	0.06760
	1500	99	93	88	94	99	0.06306
	2000	116	118	120	116	114	0.05840
$\frac{\ln(n)^3}{n}$	500	7	8	8	7	9	0.01600
	1000	13	13	14	13	12	0.01300
	1500	17	17	18	19	18	0.01186
	2000	22	21	22	20	21	0.01060

Table B.3: The connected domination number $\gamma_{BFSL}(G)$ for a random graph $G_{n,p}$

The table B.3 lists the bounds obtained by implementing the BFS together with local optimization procedures on a random graph $G_{n,p}$ for different edge probabilities.

Appendix C

Pseudo Codes

C.1 Breadth First Search

BFS(G, root)

1. **for** each vertex u in $V(G)$ **do**

   mark u a "unvisited"

   parent[u]=u

   end for

2. mark *root* as "visited"

3. Queue = $\{root\}$

4. **While** (Queue is not empty) **do**

   $u \leftarrow$ (Queue)

   for each $v \in N(u)$ **do**

   mark v as "visited"

   parent[v]=u

   Add v to the end of the Queue

   end for

   end while

5. return {parent[ ]}

C.2 Internal-Opt Procedure

The following pseudo code explains the Internal-Opt procedure for a given tree T.

INTERNAL-OPTIMIZE()

1. **For** all $v \in V(T)$ **do**

    If degree($v \geq 2$) **then**

        Make v as "root"

        $INTERNAL\_OPT(root)$

        If (failure ≤ 2) **then**

            update the tree structure.

        end if;

    end if;

  end for;

$INTERNAL\_OPT(root)$

1. **for** $u \in N(root)$ **do**

    $Mark\_Descendants(u)$

    If (degree(u)=1) **then**

        $Join\_Leaf(u)$

    else

        $BFSOPT(u)$

    end if;

  end for;

2. return;

$Join\_Leaf(u)$

1. **for** $w \in N(u)$ **do**

    If $\{u, w\} \in E(G)$ **then**

        Add $\{u, w\}$ to T

        Remove the edge $\{root, u\}$ from T

        return;

    end if;

  end for;

$\underline{Mark\_Descendants(u)}$

1. Mark u as a *descendant*

2. Add u to the *Queue*

3. **while** *Queue* is not empty **do**

    $x \leftarrow Queue$

    for $y \in N(x)$ **do**

        If y is not a *descendant* **then**

            Mark y as *descendant*

            Add y to *Queue*

        end if;

    end for;

  end while;

4. return;

$\underline{BFSOPT(u)}$

1. Add u to *Queue*

2. **while** *Queue* is not empty **do**

$x \leftarrow Queue$

If $y \in N(x)$ **then**

If $\{x, y\} \in E(G)$ and y is not a *descendant* **then**

Add the edge$\{x, y\}$ to T

If y is a leaf **then**

Remove the edge $\{root, y\}$ from T

else

Remove the edge $\{root, u\}$ from T

return;

end if;

end if;

end while;

C.3 Leaf-Opt Procedure

The following pseudo code explains the Leaf-Opt procedure.

LEAF-OPTIMIZE()

1. **do**

    for $v \in T$ **do**

        copy the tree structure to a temporary structure

        If degree(v) = 1 **then**

            $LEAF\_OPT(v)$

            update the tree structure

        end if;

    end for;

  while the updated tree has more leaves than T

$LEAF\_OPT(v)$

1. **for** $u \in N(v)$ **do**

    If $\{u, v\} \in G$ **then**

        Add the edge $\{u, v\}$ to T

        Remove an edge from the cycle formed in T

        return;

    end if;

  end for;

Appendix D

Source Codes

D.1 Random Graph Generator

```
/*****************************************************************
                        GNP.h
       This program generates a random graph G with n
       vertices and an edge probability p.
*****************************************************************/

#include<stdio.h> #include<stdlib.h> #include<time.h> #define LIMIT
2001 double RANDOM_GEN(); void RG(int , float); void
WriteGraph(void); void OneStep(int);
 void bfs(int);
int
n,seed,graph[LIMIT][LIMIT],deg,queue[LIMIT],
visited[LIMIT],tail=1,head=1;
float p; int parent[LIMIT],internal[LIMIT]; int
black[LIMIT],gray[LIMIT]; FILE *f;

/*****************************************************************
        This routine uses the inbuilt rand() function
```

```
        to generate a random number between 0 and 1.
*****************************************************************/

double RANDOM_GEN() {
    double r;
    r = ( (double)rand() / (double)(RAND_MAX+1) );
    return(r);
}

/*****************************************************************
        This routine gets the inputs (n, p and seed)
        from the user and initializes all the variable used
        in the program. Finally, it calls the procedure RG
        which generates the random graph G.
*****************************************************************/

void GenerateGraph() {
    int i,j;
    /*printf("MAXIMUM NUMBER OF VERTICES IS %d.\n",LIMIT);
    printf("PLEASE ENTER THE NUMBER OF VERTICES IN YOUR GRAPH = ");
    scanf("%d",&n);

    printf("ENTER THE EDGE PROBABILITY = ");
    scanf("%f", &p);

    printf("PLEASE ENTER THE SEED TO GENERATE THE RANDOM NUMBER = ");
    scanf("%d",&seed);*/

    //printf("p = %f\n",p);
    for(i=1;i<=n;i++)
```

```
    {
        parent[i]=0;
        internal[i]=0;
        black[i]=0;
        gray[i]=0;
        visited[i]=0;
        queue[i]=0;
        for(j=0;j<=n;j++) graph[i][j]=0;
    }
    head=1; tail=1;
    RG(n,p);
    return;
}

/*****************************************************************
        This routine writes the adjacency list and all the other
        details about the graphs in the file "randomgraph.txt".
*****************************************************************/

void WriteGraph() {
    int n1, deg, i, j;
    f = fopen("randomgraph.txt","w");
    n1=graph[0][0];

    fprintf(f,"seed = %d\n\n",seed);
    fprintf(f,"p = %f\n\n",p);
    fprintf(f,"number of vertices = %d\n\n",n);

    for(i=1;i<=n1;i++)
    {
```

```
        deg = graph[i][0];
        fprintf(f,"Vertex %4d has degree %4d and is
            adjacent to ",i,deg);
        for(j=1;j<=deg;j++) fprintf(f,"%4d",graph[i][j]);
        fprintf(f,"\n");
    }
    fclose(f);
    return;
}

/*****************************************************************
        This routine is a part of the BFS which visits the
        unvisited neighbors of the vertex picked from the queue.
*****************************************************************/


void OneStep(int k) {
    int i;
    int deg = graph[k][0];
    for(i=1;i<=deg; i++)
    {
        if (visited[graph[k][i]]!=1)
        {
            tail=tail+1;
            queue[tail]=graph[k][i];
            visited[graph[k][i]]=1;
            parent[graph[k][i]]=k;
        }
    }
    return;
```

```
}

/*************************************************************
        This routine is the BFS which is performed on the
        graph to check whether the graph is connected or not.
*************************************************************/


void bfs(int root) {
    int currentv;
    queue[head]=root;
    visited[root]=1;
    parent[root]=0;
    while(head<=tail)
    {
        currentv=queue[head];
        head=head+1;
        OneStep(currentv);
    }
    return;
}

/*************************************************************
        This routine generates the random graph. The procedure
        adds the edge between two vertices if the probability
        of that edge generated using the RANDOM_GEN() function
        is at least the edge probabilty p. It calls the BFS()
        to check whether the graph is connected or not.
*************************************************************/
```

```
void RG(int d, float prob) {
    double y;
    int i,j;
    graph[0][0]=d;  //the number of vertices is stored here
    for(i=1;i<=d;i++) graph[i][0]=0;
    //the degree of vertex i is stored here

    for(i=1;i<d;i++)
        for(j=i+1;j<=d;j++)
        {
            y=RANDOM_GEN();
            if (prob >= y)
            {
                graph[i][0]=graph[i][0]+1;
                graph[j][0]=graph[j][0]+1;
                graph[i][graph[i][0]]=j;
                graph[j][graph[j][0]]=i;
            }
        }
        WriteGraph();
        bfs(1);

    //Check if graph is connected

    if(tail==n)
    {
        printf("The graph is connected.\n");
        f = fopen("randomgraph.txt","a");
        if(tail==n) fprintf(f,"The graph is connected.\n");
```

```
        fclose(f);
    }
    else
    {
        printf("The graph is not connected.\n");
        GenerateGraph();
    }

    return;
}
/***************   END OF THE PROGRAM   ********************/
```

D.2 Regular Graph Generator

```
/*****************************************************************
                Regulargraph_Generator.h
             RANDOM d-REGULAR GRAPH GENERATOR
 This program generates a random d-regular graph with n vertices.
 *****************************************************************/

#include<stdio.h> #include<stdlib.h> #include<time.h> #define LIMIT
2001
int RANDOM_GEN(int); void RegularGen(void); void
RandomPermute(void); void formgraph(void); int loop(void); int
multipleedge(void); void copygraph(void); void bfstest(int); void
OneStepbfs(int);

int i,dots[LIMIT*LIMIT],graph[LIMIT][LIMIT],flags=1,tail=1,head=1;
int queue[LIMIT],visited[LIMIT],parent[LIMIT],internal[LIMIT]; FILE
```

```
*f;

/*************************************************************
    This routine is part of the BFS which is performed to
    find out whether the graph is connected or not.
    This procedure adds all the unvisited vertices to the queue
    and mark them as visited.
*************************************************************/
void OneStepbfs(int k) {
    int i;
    int deg = graph[k][0];
    for(i=1;i<=deg; i++)
    {
        if (visited[graph[k][i]]!=1)
        {
            tail=tail+1;
            queue[tail]=graph[k][i];
            visited[graph[k][i]]=1;
            parent[graph[k][i]]=k;
        }
    }
    return;
}

/*******************************************************
    This procedure is the BFS with an arbitrary root
    to check whether the generated regular graph is
    connected or not.
*******************************************************/
```

```
void bfstest(int root) {
    int currentv;
    queue[head]=root;
    visited[root]=1;
    parent[root]=0;
    while(head<=tail)
    {
        currentv=queue[head];
        head=head+1;
        OneStepbfs(currentv);
    }
    return;
}

/*************************************************
    This procedure writes the graphs structure in
    a file called "regular.txt".
*************************************************/

void copygraph() {
    int deg, i, j;

    f = fopen("regular.txt","w");
    fprintf(f,"seed = %d\n",seed);
    fprintf(f,"number of vertices = %d\n",n);

    for(i=1;i<=n;i++)
    {
        deg = graph[i][0];
        fprintf(f,"Vertex %4d has degree %4d and
```

```
            is adjacent to ",i,deg);
        for(j=1;j<=deg;j++)
            fprintf(f,"%4d",graph[i][j]);
        fprintf(f,"\n");
    }
    fclose(f);
    printf("The graph is in the file regular.txt\n");
    return;
}

/*********************************************************
    This routine generates a random number between
    1 and u using the inbuilt random number generator
    function rand().
*********************************************************/

int RANDOM_GEN(int u) {
    int r;
    r = (int)((double)rand() / (((double)RAND_MAX+1)/(double)u)) + 1;
    //r = rand() / (RAND_MAX / n + 1);
    //printf("%d\n",r);
    return(r);
}

/*********************************************************************
    This routine checks for multiple edges in the graph
    generated. If there is a multiple edge, then we simply
    remove that edge from that tree.
*********************************************************************/
```

```
int multipleedge() {
    int i,j,l,p;
    flags=0;
    for(i=1;i<=n;i++)
        for(j=1;j<=graph[i][0];j++)
            for(l=j+1;l<=graph[i][0];l++)
                if(graph[i][j]==graph[i][l])
                {
                    for(p=l+1;p<=graph[i][0];p++)
                        graph[i][p-1]=graph[i][p];
                    graph[i][0]-=1;l=l-1;
                }
    //printf("There are no multiple edges\n");
    return(0);
}

/*******************************************************
    This routine checks for loops formed in the graph
    generated. If there are any loops in the graph,
    then the loop is removed by simply remvoing that
    edge.
*******************************************************/

int loop() {
    int i,j,p;
    for(i=1;i<=n;i++)
        for(j=1;j<=graph[i][0];j++)
            if(graph[i][j]==i)
            {
                for(p=j+1;p<=graph[i][0];p++)
```

```
                    graph[i][p-1]=graph[i][p];
                graph[i][0]-=1;j=j-1;
            }
    //printf("There are no loops in the graph\n");
    return(0);
}

/*****************************************************
    This routine groups all the "dots" corresponding
    to every vertex and, add the edges corresponding
    to these dots. Thus, the graph is formed.
*****************************************************/

void formgraph() {
   int i,j,temp1,temp2;
   graph[0][0]=n;
   for(i=1;i<=n;i++) for(j=0;j<=n;j++) graph[i][j]=0;
   for(i=1;i<=reg*n;i+=2)
   {
       temp1 = dots[i];temp2 = dots[i+1];
       if(temp1%reg==0) temp1=temp1/reg; else temp1=(temp1/reg)+1;
       if(temp2%reg==0) temp2=temp2/reg; else temp2=(temp2/reg)+1;

       graph[temp1][0]+= 1;
       if(temp2!=0) graph[temp1][graph[temp1][0]] = temp2;
       if(temp1!=temp2)
       {
           graph[temp2][0]+= 1;
           if(temp1!=0) graph[temp2][graph[temp2][0]] = temp1;
       }
```

```
   }
   //printf("The multigraph is formed now\n");
   return;
}

/************************************************
    This routine does a random permutation of the
    "dots" in the array "dots". The random number
    generated by the RANDOM_GEN procedure is used
    here to swap two dots in the array
    index from 1 to count-1 and count.
************************************************/

void RandomPermute() {
    int i,dot,count,temp;
    for(i=1;i<=reg*n;i++)
            dots[i]=i;
        count=reg*n;
        while(count>1)
    {
        dot=RANDOM_GEN(count);
        temp=dots[dot];
        dots[dot]=dots[count];
        dots[count]=temp;
        count-=1;
    }
    return;
}
```

```
/*******************************************
    This is the main routine which calls the
    other routines. This routine includes
    accessing the user input, generating
    the graph and testing whether the graph
    is connected or not.
*******************************************/

void RegularGen() {
    /*printf("Enter the degree of the regular graph = ");
    scanf("%d",&reg);
    printf("Enter the number of vertices of the graph = ");
    scanf("%d",&n);
    /*printf("Enter the seed for generation = ");
    scanf("%d",&seed);*/
    flags=1;

    while(flags==1)
    {
        RandomPermute();
        formgraph();
    //  copygraph();
        flags=loop();
        if(flags!=1) flags=multipleedge();
    }
    copygraph();
    for(i=1;i<=n;i++)
    {
        visited[i]=0;
        queue[i]=0;
```

```
    } tail=1; head=1;
        bfstest(1);

    //Check if graph is connected

    if(tail==n)
    {
        printf("The graph is connected.\n");
        f = fopen("regular.txt","a");
        if(tail==n) fprintf(f,"The graph is connected.\n");
        fclose(f);
    }
    else
    {
        printf("The graph is not connected.\n");
        RegularGen();
    }

    return;
}

/**** END OF PROGRAM ****/
```

D.3 Breadth First Search

```
/*************************************************
                    BFS.h
    This program perform the Breadth First Search
    on the given graph and finds a spanning tree
    of the graph.
```

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

#include<stdio.h> #include<math.h> #include<stdlib.h> #define LIMIT
2001 int tree[LIMIT][LIMIT]; int root; void BFS(int); void
WriteBFS(int,int); void ENQUEUE(int); int value;

/************************************************************
    This routine is a part of the BFS which visits the
    unvisited neighbors of the vertex picked from the queue.
************************************************************/

void ENQUEUE(int k) {
    int i;
    int deg = graph[k][0];
    for(i=1;i<=deg; i++)
    {
        if (visited[graph[k][i]]!=1)
        {
            tail=tail+1;
            queue[tail]=graph[k][i];
            visited[graph[k][i]]=1;
            parent[graph[k][i]]=k;
        //  printf("parent of %d is %d\n",graph[k][i], k);
        }
    }
    return;
}

/************************************************
   This routine creates the adjacency list for the
```

```
   tree and copies that to a text file.
**************************************************/

void WriteBFS(int root, int value) {
    int i,leaf,j;
    if(tail<=head)
    {
        for(i=1;i<=n;i++)
        {
            internal[parent[i]]=1;
            for(j=0;j<=n;j++) tree[i][j]=0;
        }
        for(i=1;i<=n;i++)
        for(j=1;j<=n;j++)
        {
          if(parent[i]==j)
            {
                tree[i][0]=tree[i][0]+1;
                tree[j][0]=tree[j][0]+1;
                tree[i][j]=1;
                tree[j][i]=1;
            }
        }
        leaf=0;

        for(i=1;i<=n;i++)
            if (internal[i]==0) leaf=leaf+1;
        if (graph[root][0]==1) leaf=leaf+1;

        printf("The number of leaves in BFS tree
```

```
            with root %d is %d.\n",root,leaf);
        printf("Connected domination number
            is at most %d.\n",n-leaf);

        if(value==0)
            f = fopen("regular.txt","a");
        else
            f = fopen("randomgraph.txt","a");

        fprintf(f,"\nThe number of leaves in BFS
            tree with root %d is %d.\n",root,leaf);
        fprintf(f,"Connected domination number
            is at most %d.\n",n-leaf);

        fprintf(f,"The BFS tree is\n\n");
    for(i=1;i<=n;i++)
    {

        fprintf(f,"Vertex %4d has degree %4d and
            is adjacent to ",i,tree[i][0]);
        for(j=1;j<=n;j++) if(tree[i][j]==1) fprintf(f,"%4d",j);
        fprintf(f,"\n");
    }
//  for(i=1;i<=n;i++) fprintf(f,"%4d,%4d\n",i,parent[i]);
    fprintf(f,"\n");
        fclose(f);
        return;
    }
}
```

```
/*********************************************************
  This routine is the main routine for the BFS which is
  called from the main program with a "root". This routine
  also initializes the tree structure.
 *********************************************************/

void BFS(int root) {
    int currentv,i;
    tail=1;head=1;
    queue[head]=root;
    for(i=1;i<=n;i++)
    {
        internal[i]=0;
        parent[i]=0;
        visited[i]=0;
    }
    visited[root]=1;
    parent[root]=0;

    while(head<=tail)
    {
        currentv=queue[head];
        head=head+1;
        ENQUEUE(currentv);
    }
    WriteBFS(root,value);
    return;
} /*****  END OF PROGRAM  ****/
```

D.4 Internal-Opt procedure

```
/*****************************************************************
                    INTERNAL_OPTMIZATION.h
 This program aims to turn the internal vertices of the spanning
 tree into a leaf in order to gain more leaves in the tree.
*****************************************************************/

#include<stdio.h> #include<stdlib.h>


int  bfsopt(int); void Find_Descendants(int); void
Find_Neighbour(int); void INTERNAL_OPT(int); void
Mark_Descendants(int); void Internal_Optimize(void); int
Join_Leaf(int); void update_tree(void);
void write_the_graph(void);
void update_parent(int); void Find_Parent(int);

int descendant[LIMIT]; int visit[LIMIT]; int
queues[LIMIT],found,failure,front, back; int
temp_tree[LIMIT][LIMIT], temp_parent[LIMIT],
temp_internal[LIMIT];
int vertex1, leaf,flags;


/*********************************************************
 This routine updates the old tree structure with the new
 tree structure, if the new tree has more leaves in it than
 the old tree.
*********************************************************/
```

```
void update_tree() {
    int i,j;
    int oldleaf=0; int newleaf=0;
    for(i=1;i<=n;i++)
    {
        if(internal[i]==0) oldleaf=oldleaf+1;
        if(temp_internal[i]==0) newleaf=newleaf+1;
    }
        if(oldleaf<newleaf)
        {
            for(i=1;i<=n;i++)
            {
                internal[i]=temp_internal[i];
                for(j=0;j<=n;j++)
                tree[i][j]=temp_tree[i][j];
            }
        }
    //printf("tree is updated\n");
        return;
}


/*****************************************************************
 This routine writes the updated tree structure to the file
 "randomgraph.txt" or "regulargraph.txt" depending on the graph
 that we use in the procedure.
*****************************************************************/

void write_the_graph() {
```

```
int i,j;leaf=0;
for(i=1;i<=n;i++)
    if (internal[i]==0) leaf=leaf+1;

if (graph[root][0]==1) leaf=leaf+1;

printf("The number of leaves in the Internal-optimized
    tree with root %d is %d.\n",root,leaf);
printf("Connected domination number is
    at most %d.\n",n-leaf);

if(value==0)
      f = fopen("regular.txt","a");
else
      f = fopen("randomgraph.txt","a");

fprintf(f,"\nThe number of leaves in the Internal-optimized
    tree with root %d is %d.\n",root,leaf);
fprintf(f,"Connected domination number is
    at most %d.\n\n",n-leaf);

/*for(i=1;i<=n;i++)
{
    fprintf(f,"Vertex %4d has degree %4d and
        is adjacent to ",i,tree[i][0]);
    for(j=1;j<=n;j++)
        if(tree[i][j]==1) fprintf(f,"%4d",j);
    fprintf(f,"\n");
}*/
```

```
    /*for(i=1;i<=n;i++)
        fprintf(f,"%4d,%4d\n",i,parent[i]);*/
    fprintf(f,"\n");
    fclose(f);
    return;
}

/************************************************************
  This routine puts all the vertices in the descendant list
  from the bfsopt() routine, in a queue
************************************************************/


void Find_Neighbour(int k) {
    int i;

    for(i=1;i<=n; i++)
    {
        if (visit[i]!=1 && temp_tree[k][i]!=0)
        {
            tail=tail+1;
            queue[tail]=i;
            visit[i]=1;
        }
    }
    return;
}
```

```
/*********************************************************
 This routine finds all the descendants for the vertex r.
*********************************************************/

void Find_Descendants(int r) {
    int i;
    for(i=1;i<=n;i++)
    {
     if(temp_tree[r][i]==1 && descendant[i]!=1)
     {
       back+=1;
       queues[back]=i;
       descendant[i]=1;
       //printf("%d<-- ",i);
     }
    }
    return;
}

/*********************************************************
 The routine Mark_Descendants() along with the routine
 Find_descendants()is the BFS performed on that branch
 of the tree. This is to find all the descendants of
 the vertex k which is a neighbor to the root of the
 spanning tree.
*********************************************************/
```

```
void Mark_Descendants(int k) {
    int currentv;
    front=1;back=1;
    descendant[k]=1;
    queues[front]=k;
    //printf("descendants of %d are ",k);
    while(front<=back)
    {
      currentv=queues[front];
      front+=1;
      Find_Descendants(currentv);
    }
    //printf("\n");
    return;
}



/*******************************************************
  This routine is part of the routine update_parent().
 *******************************************************/

void Find_Parent(int s) {
    int i;
    for(i=1;i<=n;i++)
    {
     if(tree[s][i]==1 && visit[i]!=1)
     {
       back+=1;
       queues[back]=i;
```

```
        parent[i]=s;
        visit[i]=1;
        //printf("%d<-- ",i);
     }
    }
    return;
}

/*************************************************************
 This routine updates the parent structure of the tree at the
        end of the program.
 *************************************************************/

void update_parent(int k) {
    int currentv;
    front=1;back=1;
    visit[k]=1;
    parent[k]=0;
    queues[front]=k;
    //printf("descendants of %d are ",k);
    while(front<=back)
    {
      currentv=queues[front];
      front+=1;
      Find_Parent(currentv);
    }
    //printf("\n");
    return;
}
```

```
/*****************************************************************
 This routine performs a BFS to look for an internal vertex in
 the descendant list which has a non-tree neighbor in a different
 branch. For every vertex in the queue we add a non-tree edge if
 and only if the vertex is an internal vertex.
*****************************************************************/

int bfsopt(int root1) {
    int current,i,temp,found1;
    tail=1; head=1;
    queue[head] = root1; visit[root1] = 1;
    found=0;
    flags=0;

    while(head<=tail)
    {
      current = queue[head];
      if(temp_tree[current][0]>1)
      {
        i=1;
        while(descendant[graph[current][i]]==1
            && i<=graph[current][0]) i++;

        if(i<=graph[current][0])
        {
          temp = graph[current][i];
          if(temp_tree[temp][0]>1)
          {
        //  printf("the edge between %d and %d
```

```
                is removed\n",vertex1, root1);
            temp_tree[vertex1][root1]=0; temp_tree[root1][vertex1]=0;
            temp_tree[root1][0]-=1; temp_tree[vertex1][0]-=1;
            temp_tree[current][temp]=1;  temp_tree[temp][current]=1;
            temp_tree[current][0]+=1; temp_tree[temp][0]+=1;

            found=1;
            return(found);
          }
          else if(temp_tree[vertex1][temp]==1)
          {
            temp_tree[vertex1][temp]=0; temp_tree[temp][vertex1]=0;
            temp_tree[vertex1][0]-=1; temp_tree[temp][0]-=1;
            descendant[temp]=1;
            temp_tree[current][temp]=1;  temp_tree[temp][current]=1;
            temp_tree[current][0]+=1; temp_tree[temp][0]+=1;
          }

        //printf("vertex %d is connected to %d\n",current,temp);
        }
      }
      Find_Neighbour(current);
      head = head + 1;
   }
   return(found);
}

/*****************************************************************
 This routine joins the leaf which is a neighbor to the root
 and one of its non-tree neighbor which is not in the descendant
```

```
 list. The edge between the root and the leaf is removed here.
*****************************************************************/

int Join_Leaf(int leaf) {
   int j,temp; found=0;
   for(j=1;j<=graph[leaf][0];j++)
   {
       temp = graph[leaf][j];
       if(descendant[temp]!=1 && temp_tree[temp][0]>1)
       {
         temp_tree[leaf][temp]=1; temp_tree[temp][leaf]=1;
         temp_tree[vertex1][leaf]=0; temp_tree[leaf][vertex1]=0;
         temp_tree[temp][0]+=1; temp_tree[vertex1][0]-=1;
         found=1;

       //printf("%d is connected to %d in the process
            of turning %d to leaf\n",leaf,temp,vertex1);
         return(found);
       }
   }
// printf("could not connect %d to any of it's neighbors
    and hence the failure is %d\n",leaf,failure+1);
   failure = failure + 2;
   return(found);
}



/*****************************************************************
 The INTERNAL_OPT routine takes two kinds of neighbors of the root.
```

```
 1. a leaf 2. an internal vertex. For both these cases, the
 descendant list is scanned find a vertex which has a neighbor not
 in the descendant list.
**********************************************************************/

void INTERNAL_OPT(int vertex) {
    int i,j,k;
    vertex1=vertex;  failure=0;

    for(i=1;i<=n;i++)
    {
       if(failure<2 && temp_tree[vertex1][0]!=1)
       {
         found=0;

    /**** THE CASE WHEN THE NEIGHBOR IS A LEAF ****/

         if(temp_tree[vertex][i]==1)
         {
          for(k=1;k<=n;k++)
          {
            descendant[k]=0; visit[k]=0;
          } //Initialization is done here
          descendant[vertex]=1; visit[vertex]=1;

          if(temp_tree[i][0]==1)
          {
            descendant[i]=1;
            found = Join_Leaf(i);
            if(found==0)
```

```
            {
          //  printf("The vertex %d cannot be turned into
                 a leaf\n",vertex1);
              return;
            }
          }

    /**** THE CASE WHEN THE NEIGHBOR IS AN INTERNAL VERTEX ****/

          else if(temp_tree[vertex][i]==1)
          {
            for(j=1;j<=n;j++) if(j!=vertex) descendant[j]=0;
            Mark_Descendants(i);
            if(found!=1 && failure<2)
            {
            //printf("%d is an internal vertex to %d\n",i,vertex);
              found = bfsopt(i);
              if(found!=1) failure = failure + 1;
            }
          }
        }
      }
    }
    return;
}


/************************************************************
  This is the main routine which calls other procedures.
  This routine picks all the internal vertices in the tree
```

```
  and passes it to the INTERNAL_OPT procedure.
*************************************************************/

void Internal_Optimize() {
  int i,j,r,k;

  for(i=1;i<=n;i++)
  {


    /*    WE PICK THE VERTEX i WITH DEGREE AT LEAST 2  */

      if(tree[i][0]>=2)
      {

     /* THE TREE STRUCTURE IS COPIED TO A TEMPORARY ARRAY */
        for(k=1;k<=n;k++)
        {
          for(j=0;j<=n;j++)  temp_tree[k][j]=tree[k][j];
          temp_internal[k]=internal[k];
          temp_parent[k]=parent[k];
        }

        INTERNAL_OPT(i);
            //THE INTERNAL_OPT PROCEDURE IS PERFORMED FOR VERTEX i

   /*  IF THE NUMBER OF FAILURES IS LESS THAN 2 THEN THE
         VERTEX i HAS BEEN TURNED INTO A LEAF */

       if(failure<2)
```

```
        {
        // printf("vertex %d has been turned into a leaf\n",i);
           for(j=1;j<=n;j++)
           {
             if(temp_tree[j][0]>1)
                 temp_internal[j]=1;
             else
                 temp_internal[j]=0;
           }

           update_tree();
        }
     }
   }

   /* WE PICK THE FIRST INTERNAL VERTEX IN THE TREE AND PERFORM THE
      BFS TO GET THE COMPLETE PARENT STRUCTURE OF THE TREE. */

   r=1;
   while(tree[r][0]==1 && r<=n) r++;
   for(i=1;i<=n;i++) {visit[i]=0; }
   update_parent(r);
   root=r;
   write_the_graph();
   return;
}
/*************  END OF PROGRAM **************/
```

D.5 Leaf-Opt procedure

```
/*****************************************************
                LEAF-OPTIMIZATION.h
    This procedure picks the leaves in the spanning
    tree of G and aim to turn them into internal
    vertices to yield a spanning tree with more
    leaves in it.
*****************************************************/

#include<stdio.h> #include<stdlib.h> void LEAF_OPT(int); void
findcycle(int, int); void Remove_Cycle(int,int); void
writegraphs(void); void COPY_TREE(); void updategraph(int,int);

void update(); void finalupdate(); void Leaf_Optimize(void);

void InternalOptimize(void);

int flag,oldleaf,newleaf,oleaf=0; int
cycle[LIMIT],incycle[LIMIT],length,neighbour,checked[LIMIT];

int gain,next[LIMIT];

/*************************************************************
  This routine updates the old tree structure with the new
  tree structure, if the new tree has more leaves in it than
  the old tree.
*************************************************************/

void update() {
```

```
    int i,j;
    oldleaf=0;newleaf=0;
    for(i=1;i<=n;i++)
    {
        if(internal[i]==0) oldleaf=oldleaf+1;
        if(temp_internal[i]==0) newleaf=newleaf+1;
    }
        if(oldleaf<newleaf)
        {
            for(i=1;i<=n;i++)
            {
                parent[i]=temp_parent[i];
                internal[i]=temp_internal[i];
                for(j=0;j<=n;j++)
                    tree[i][j]=temp_tree[i][j];
            }
        }
        //printf("updated\n");
        return;
}

/*******************************************************
 This routine copies the tree structure to a temporary
 structure to which the LEAF-OPT is performed.
*******************************************************/

void COPY_TREE() {
    int i,j;
    for(i=1;i<=n;i++)
    {
```

```
        temp_parent[i]=parent[i];
        temp_internal[i]=internal[i];
        for(j=0;j<=n;j++)
            temp_tree[i][j]=tree[i][j];
    }
//  gain=-1;
    return;
}

/*****************************************************
    This routine updates the temporary tree structure
    and the parent structure as the LEAF-OPT makes
    changes by adding a non-tree edge.
*****************************************************/

void updategraph(int p,int q) {
    int v1, v2,i;
    temp_tree[p][0]-=1; temp_tree[q][0]-=1;
    temp_tree[p][q]=0; temp_tree[q][p]=0;
    temp_tree[vertex1][neighbour]=1;
    temp_tree[neighbour][vertex1]=1;
    for(i=1;i<=n;i++)
    if (temp_tree[i][0]==1)
    {temp_internal[i]=0;} else {temp_internal[i]=1;}
    if(temp_parent[p]==q)
    {
        v1=vertex1; v2=neighbour;
        while(v1!=q)
        {
            temp_parent[v1]=v2;
```

```
            v2=v1;
            v1=next[v1];
        }
    }
    else
    {
        v1=q; v2=next[q];
        while(v1!=vertex1)
        {
            temp_parent[v1]=v2;
            v1=v2;
            v2=next[v1];
        }
    }
}

/*************************************************************
    This routine writes the updated tree structure
    to the file "randomgraph.txt" or "regulargraph.txt"
    depending on the graph that we use in the procedure.
*************************************************************/

void writegraphs() {

  int i,j;
  leaf=0;
  for(i=1;i<=n;i++)
    if (internal[i]==0) leaf=leaf+1;

  if (graph[root][0]==1) leaf=leaf+1;
```

```
  printf("The number of leaves in the leaf-optimized
    tree with root %d is %d.\n",root,leaf);
  printf("Connected domination number is at
    most %d.\n",n-leaf);

  if(value==0)
      f = fopen("regular.txt","a");
  else
      f = fopen("randomgraph.txt","a");

  fprintf(f,"\nThe number of leaves in the leaf-optimized
    tree with root %d is %d.\n",root,leaf);
  fprintf(f,"Connected domination number is at
    most %d.\n\n",n-leaf);

  for(i=1;i<=n;i++)
    {

        fprintf(f,"Vertex %4d has degree %4d and is
            adjacent to ",i,tree[i][0]);
        for(j=1;j<=n;j++)
        if(tree[i][j]==1) fprintf(f,"%4d",j);
        fprintf(f,"\n");
    }
  /*for(i=1;i<=n;i++) fprintf(f,"%4d,%4d\n",i,parent[i]);*/
  fprintf(f,"\n");
  fclose(f);
  return;
}
```

```
/*************************************************************
  This routine finds an edge in the cycle formed by adding a
  non-tree edge to the tree. The value of "flag" is the
  number of end-vertices with degree 2 of that edge which is
  to be removed from the cycle.
*************************************************************/

void Remove_Cycle(int l,int r) {
    int v1=l,v2=next[l],vv1,vv2;
    int flag1=0;
    flag=0;
    while(flag<2 && v1!=r)
    {
        flag=0;
        if(temp_tree[v1][0]==2) flag = flag+1;
        if(temp_tree[v2][0]==2) flag = flag+1;
        if (flag>0) {vv1=v1;vv2=v2; if(flag>flag1) flag1=flag;}
        v1=v2;v2=next[v2];
    }

    /**** If there are no edges in the cycle with none of
    end-vertices having degree 2 then, we remove the
    non-tree edge added before. ****/

    if( (flag1==2) || ((flag1==1) && (temp_tree[r][0]>2)) )
    updategraph(vv1,vv2);
    else {temp_tree[l][0]-=1; temp_tree[r][0]-=1;}
    return;
}
```

```
/*****************************************************
    This routine is to find the vertices that are in
    the cycle formed. we track the vertices in the
    cycle using the parent structure of the tree.
*****************************************************/

void findcycle(int s, int r) {
     //int temp,temp1,temp2;
     int v1,v2;  v1=s;

     while (v1!=root && v1==next[v1])
     {
        next[v1]=temp_parent[v1];
        v1=temp_parent[v1];//printf("finding leaf-1\n");
     }
     v1=r; v2=s;
     while ((v1==next[v1]) && (v1!=root))
     {
        next[v1]=v2;
        v2=v1;
        v1=temp_parent[v1];//printf("%4d ",v1);
     }
     next[v1]=v2;
     return;
}

/*****************************************************
    This routine picks the leaf "vertex" and performs
    the LEAF-OPT on it. This routine adds the non-tree
    edge and calls the other procedures that finds the
```

```
    cycle and the edge to be removed from the cycle.
*****************************************************/

void LEAF_OPT(int vertex) {
    int i,j,temp=0;
    vertex1=vertex;
    gain = -1;
     for(j=1;j<=graph[vertex][0];j++)
     {
        if (temp_tree[vertex][graph[vertex][j]]==0)
        {
            neighbour=graph[vertex][j];

            temp_tree[vertex][0]+=1;
            temp_tree[neighbour][0]+=1;

            for(i=1;i<=n;i++) next[i]=i;
            findcycle(vertex,neighbour);
            Remove_Cycle(vertex, neighbour);
        }
    }
    return;
}

/*****************************************************
    This routine repeats the LEAF-OPT procedure
    for all the leaves in the tree until the
    updated tree does not give any more improvement
    with the number of leaves in the tree.
*****************************************************/
```

```
void Leaf_Optimize() {
    int i;
    do
    {
      for(i=1;i<=n;i++)
      {
          COPY_TREE();
          if(temp_internal[i]==0)
          {
            LEAF_OPT(i);
            update();
          }
      }
    }
    while(oldleaf<newleaf);
    writegraphs();
    return;
}

/************   END OF PROGRAM   ***************/
```

D.6 Checking the Output

```
/*********************************************************
               Connectedness.h
  This routine checks whether the graph obtained after
  implementing the BFS and the local optimization
  procedures is connected or not and whether it is a
```

```
  tree or not.
 ******************************************************/
#include <stdio.h> void Connectedness(void);

void Connectedness() {
    int i,j, Nedges, QQ[2500], first, last,
        vx, vtd[2000], Nvtd, deg0, Nintl;

    Nedges=0;
    for(i=1;i<=n;i++)
        for(j=1;j<=n;j++)
            if (tree[i][j]==1) Nedges=Nedges+1;
    //printf("Number of edges is %d.\n",Nedges/2);
    if(Nedges/2!=n-1)
        printf("# of edges is not equal to # of
            vertices minus 1 %d, %d\n",Nedges,n-1);

    for(i=1;i<=n;i++) vtd[i]=0;
    first=1; last=1; QQ[1]=1; vtd[i]=1;

    while(first<=last)
    {
        vx=QQ[1];
        for(i=1;i<=n;i++)
            if((tree[vx][i]==1) && (vtd[i]==0))
            {
                last=last+1;
                QQ[last]=i;
                vtd[i]=1;
            }
```

```
        first=first+1;
    }

    Nvtd=0;
    for(i=1;i<=n;i++) if(vtd[i]=1) Nvtd=Nvtd+1;

    if(Nvtd!=n)
        printf("The graph is not connected\n");

    /*******************************************************
      If the graph is connected and # edges = # vertices-1
      then the graph is a tree.
     *******************************************************/

}

/****** END OF PROGRAM ******/
```

D.7 The Main Programs

Random Graph

```
/*************************************************************
                Random.Output.cpp
        BFS and LOCAL OPTIMIZATION on a RANDOM GRAPH G
  This program generates a random graph G with n vertices
  and an edge probability p. The BFS and the LOCAL-
  OPTIMIZATION procedures are performed on the graph
  generated and the output is stored in a file called
  "Random.Output.txt".
```

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

#include <stdio.h> #include <math.h> #include "GNP.h"

#include "bfs.h" #include "Internal-Optimization.h"

#include "Leaf-Optimization.h" #include "Heuristic.h"

#include "Connectedness.h" FILE *d;

void main() {
    int i,sum;
    float average,finalbound,bound;

    /*********************************************************
       The number of vertices are in the range 500, 1000,
       1500 and, 2000. The graph is generated with various
       edge probabilities. Here the probability we see is
       p= ((ln(n)^3)/n. The random number generator is
       initialized with a seed(1211) here.
     *********************************************************/

        bound = 0;
        n=500;
        seed=1211;
        srand(seed);

        while(n<=2000)
        {
            int k=1;
```

```
        sum=0;average=0;
        d = fopen("Random.Output.txt","a");

     /** FIVE GRAPHS ARE GENERATED FOR A GIVEN
              NUMBER OF VERTICES **/

        while(k<=5)
        {

/*** THE EDGE PROBABILITY IS CALCULATED HERE ***/

             p = (log((double)n)*log((double)n)
                     *log((double)n))/(double)n;

             fprintf(d,"\nNumber of Vertices = %d\n",n);
             fprintf(d,"Edge Probability = %f\n",p);
             value=2;

/*** THIS ROUTINE GENERATES A RANDOM GRAPH G ***/

             GenerateGraph();

/*** THIS ROUTINE PERFORMS A BFS ON G WITH ROOT 1 ***/

             BFS(1);

             leaf=0;
             for(i=1;i<=n;i++) if(tree[i][0]==1) leaf =leaf+1;
             fprintf(d,"The CDN of BFS tree is = %d\n",n-leaf);
```

```
        /*** THIS ROUTINE PERFORMS THE INTERNAL-OPT ***/
        Internal_Optimize();

        leaf=0;
        for(i=1;i<=n;i++) if(tree[i][0]==1) leaf =leaf+1;
        fprintf(d,"The CDN of after Internal-opt
        tree is = %d\n",n-leaf);


        /*** THIS ROUTINE PERFORMS THE LEAF-OPT ***/
        Leaf_Optimize();

        leaf=0;
        for(i=1;i<=n;i++) if(tree[i][0]==1) leaf =leaf+1;
        fprintf(d,"The CDN of after Leaf-opt tree
        is = %d\n",n-leaf);
        Connectedness();

/*****************************************************
   THE FOLLOWING LINES CALCULATE THE AVERAGE OF THE
        CONNECTED DOMINATION NUMBERS OBTAINED.
 *****************************************************/

        sum = sum + (n-leaf);
        average = average + (double)(n-leaf)/(double)n;
        fprintf(d,"The average is = %f\n",
            (double)(n-leaf)/(double)n);

        k++;
   }
```

```
            finalbound = (double)average/5;
            fprintf(d,"The Upper Bound is = %fn\n\n",finalbound);
            n = n + 500;
            fclose(d);
        }

} /***** END OF PROGRAM *****/
```

Regular graph

```
/*******************************************
            Regular.Output.cpp
   BFS and LOCAL OPTIMIZATION performed on a
          RANDOM d-REGULAR GRAPH
  This program generates a random d-regular
  graph G and performs the BFS and the LOCAL-
  OPTIMIZATION procedures on it.
 *******************************************/

#include <stdio.h> #include "RegularGraph_Generator.h"

#include "bfs.h" #include "Internal-Optimization.h"

#include "Leaf-Optimization.h" #include "Connectedness.h"
FILE *d;

int n, seed, reg; void main() {
    int i,sum, k;
    double average,finalbound,bound;

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

```
  The program shows the degree d of the graph in the range from
  10-50 with an increment by 10. The number of vertices range
  from 500-2000 with an increment of 500. A seed 1211 is given
  to generate the random numbers for the random permutation
  in generating the graph.
*****************************************************************/

   reg=10;
   seed=1211;
   srand(seed);

   while(reg<=50)
   {
       bound = 0;
       d = fopen("Regular.Output.txt","a");
       fprintf(d,"Degree = %d\n\n",reg);

       n=500;
       while(n<=2000)
       {
         k=1; sum=0;average=0;

   /** FIVE GRAPHS ARE GENERATED FOR THE GIVEN NUMBER
       OF VERTICES **/
         while(k<=5)
         {
          fprintf(d,"Number of Vertices = %d\n",n);
   /**** THIS ROUTINE GENERATES A RANDOM d-REGULAR GRAPH ****/
          RegularGen();
   /**** THIS ROUTINE PERFORMS A BFS ON G WITH ROOT 1 *******/
```

```
        BFS(1);
        leaf=0;
        for(i=1;i<=n;i++) if(tree[i][0]==1) leaf =leaf+1;
        fprintf(d,"The CDN of BFS tree is = %d\n",n-leaf);

 /**** THIS ROUTINE PERFORMS THE INTERNAL-OPT ****/
        Internal_Optimize();
        leaf=0;
        for(i=1;i<=n;i++) if(tree[i][0]==1) leaf =leaf+1;
        fprintf(d,"The CDN of after Internal-opt
         tree is = %d\n",n-leaf);

 /**** THIS ROUTINE PERFORMS THE LEAF-OPT ****/
        Leaf_Optimize();
        leaf=0;
        for(i=1;i<=n;i++) if(tree[i][0]==1) leaf =leaf+1;
        fprintf(d,"The CDN of after Leaf-opt
         tree is = %d\n",n-leaf);

/*******************************************************
  THE FOLLOWING LINES CALCULATE THE AVERAGE OF THE
       CONNECTED DOMINATION NUMBERS OBTAINED.
 *******************************************************/

        sum = sum + (n-leaf);
        k++;
      }
      average = (double)sum/5;
      bound = bound + (double)average/n;
      n = n + 500;
```

```
        }
        finalbound = bound/4;
        fprintf(d,"The Upper Bound for degree %d is =
         %fn\n\n",reg,finalbound);
        reg = reg + 10;
        fclose(d);
    }
}

/****** END OF PROGRAM ******/
```

References

[1] N. Alon and J. Spencer, The Probabilistic Method, Wiley Interscience, New York, 1992.

[2] C. Berge, The Theory of Graphs and its applications, Methuen and Co, London, 1962.

[3] Bollabás, B.: Random Graphs. Academic Press, 1985.

[4] K. Booth and G. Lueker, Testing for the Consecutive Ones Property, Interval Graphs, and Graph Planarity Using PQ-Tree Algorithms, J. of Computer and System Sciences, vol. 13, pp. 335-379, 1976.

[5] Y. Caro, D. B. West, and R. Yuster, Connected Domination and Spanning Trees with Many Leaves, SIAM J. Discr. Math. 13, 2000, 202-211.

[6] E. J. Cockayne and S. T. Hedetniemi, Towards a theory of domination in graphs, John Wiley and Sons, NY, 7, 1977, 247-261.

[7] W. Duckworth and Bernard Mans, On the Connected Domination Number of Random Regular Graphs, Springer-Verlag, Berlin Heidelberg, 2002.

[8] Frucht and Harary, On the corona of two graphs, Aequationes Math., 4, 1970, 322-324.

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

[10] J. R. Griggs, and M. Wu, Spanning trees in graphs with minimum degree 4 or 5, Disc. Math. 104, 1992, 167-183.

[11] S.Guha and S.Khuller, Approximation algorithms for connected dominating sets, Algorithmica, 20(4), Page 374-387, Apr.1998.

[12] T. W. Haynes, S. T. Hedetniemi, and Peter J. Slater, Domination in Graphs, Advanced topics, Marcel Dekker Inc., New York, 1998.

[13] Teresa W. Haynes, S. T. Hedetniemi, Peter J. Slater, Fundamentals of Domination in Graphs, Marcel Dekker, New York, 1998.

[14] S. T. Hedetniemi and Renu Laskar, Connected domination in Graphs, In B. Bollobás, editor, Graph Theory and Combinatorics, Academic Press, London, 1984, 209-218.

[15] D. J. Kleitman and D. B. West, *Spanning trees with many leaves*, SIAM J. Disc. Math. 4(1991), 99-106.

[16] S. L. Mitchell, E. J. Cockayne, and S. T. Hedetniemi, Linear algorithms on recursive representations of trees. J. Comput. System Sci., 18(1), 76-85, 1979.

[17] O. Ore, Theory of Graphs, Amer. Math. Soc. Colloq. Pub., Providence, RI 38 1962.

[18] J.Pfaff, R.Laskar, and S.T.Hedetniemi. NP-completeness of total and connected domination, and irredundance for bipartite graphs. Technical Report 428, Dept. Mathematical Sciences, Clemenson Univ.,1983

[19] G. Ramalingam and C. Pandu Rangan. A unified approach to domination problems in interval graphs. inform. Process. Lett., 27:271-274, 1988.

[20] E. Sampathkumar and H. B. Walikar, The connected domination of a graph. Math. Phys. Sci. 13 (1979), 607-613.

[21] Wormald, N. C.: The Differential Equation Method for Random Graph Processes and Greedy Algorithms. In: Lectures on Approximation and Randomized Algorithms, PWN (1999), 73-155.

About the Author

Gayathri Mahalingam completed her Bachelors in Computer Science and Engineering in University of Madras, India.