DOMINATION IN GRAPH

PREPARED BY

ACHAL AGARWAL[1] **&** **PRASHANT KHANDELWAL**[2]

(2015B4A70436P) (2016B4A70930P)

UNDER THE GUIDANCE OF

DR. RAJIV KUMAR

PROFESSOR, DEPARTMENT OF MATHEMATICS

IN PARTIAL FULFILMENT OF THE COURSE

DESIGN PROJECT (MATH F376)[1] **STUDY PROJECT (MATH F266)**[2]

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI

APRIL, 2019

ACKNOWLEDGEMENT

We would like to thank the Vice Chancellor, Prof. Souvik Bhattacharyya, the Director, Prof. Ashoke Kumar Sarkar and Dr. Bhupendra Kumar Sharma, HoD, Department of Mathematics, who provided me an opportunity to enroll in this Project; so that we can enhance our understanding of Graph Theory.

We are obliged to our instructor Dr. Rajiv Kumar for his invaluable suggestions and help. Working on this project equipped us with indispensable skill of reading research literature in Mathematics which will go a long way. During the course of the project we faced various challenges, but with the help of our instructor we sailed.

ABSTRACT

This report is based on studies conducted by us over a period of two initial months of the Study Project. Firstly, we learned about the concepts of dominating sets, independent sets, connected dominating sets, domination numbers and connected domination numbers. Then we further learned about some major applications of connected dominating sets in real world and how they are tackled as of now.

We then built our own algorithm for constructing Dominating sets which performs better than any other Greedy Iterative Algorithm that we know of.

Further, we also studied an exact algorithm to solve maximum leaf spanning tree problem and hence, the minimum connected dominating set problem.

Lastly, we explore multiple greedy algorithms for computing the Connected Dominating Set of a graph and analyse the results.

Table of Contents

1. Introduction

Domination in graphs has been an extensively researched branch of graph theory. Graph theory is one of the most flourishing branches of modern mathematics and computer applications. The last 30 years have witnessed spectacular growth of Graph theory due to its wide applications to discrete optimization problems, combinatorial problems and classical algebraic problems.

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

If D, in addition, induces a connected subgraph, then it is called a connected dominating set (CDS). The connected domination number of a graph G is the minimum cardinality of a CDS, denoted by $\gamma c(G)$. A CDS that has the size equal to the connected domination number is called a minimum CDS.

Laskar and Pfaff showed the NP-hardness of computing the connected domination number or the minimum CDS. Namely, the following problem is NP-hard.

MIN-CDS: Given a graph G = (V,E), find a CDS with minimum cardinality.

2. NP-Completeness of MIN-CDS

Consider the following problem.

SET COVER: Given a collection C of subsets of a base set X and a positive integer $k \leq |X|$, determine whether C contains a set cover with cardinality at most k, where a set cover is a sub-collection A of C such that every element of X appears in at least one subset in A.

SET-COVER is a well-known NP-complete problem. We construct a reduction from SET-COVER to MIN-CDS as follows. For input collection C and base set X in SET-COVER we first construct a bipartite graph H with n+m vertices labelled by all elements x1,u2,...,xn in X and all subsets S1,S2,...,Sm in C. An edge exists between two vertices a and b if and only if $a \in b$ or $b \in a$. Graph G is obtained from H by adding two new vertices s and t and connecting s to t and every Si for i = 1,2,...,m.

Suppose C has a set cover A of at most size k. Then the vertices with labels in A together with s form a CDS with cardinality at most k+1.

Conversely, suppose G has a CDS C of size k ≤ k + 1. Note that C must contains node s in order to dominating node t or connection t to other vertices in C. Furthermore, we claim that if a∈C for some a∈X, then C−{a} is still a CDS. In fact, to have a path connecting a and s, there must exist A∈C such that a∈A. Thus, a can be dominated by A. Moreover, all vertices dominated by a are also dominated by s. Thus, C−{a}is still a CDS. Now, let us denote by C the CDS obtained from C by deleting t and all elements in X. Then, C contains s and some vertices labeled by subsets A1,A2,...,Ah (h≤k−1) in C. These h subsets A1,A2,...,Ah must cover all elements in X. Therefore, G has a CDS of size at most k+1 if and only if C has a set cover of size at most k.

While MIN-CDS in general is NP-hard, a lot of earlier efforts were made on design of polynomial-time algorithms for special class of graphs, such as series parallel graphs and permutation graphs.

3. Applications

Dominating Graph plays an important role in networks (Computer Communication, Biological and Social). This concept has also been used to solve a vast number of problems, ranging from local balancing to preventing epidemics. The current research is focused on Connected Dominating Sets (CDS) which have more applications than ordinary Dominating sets.

3.1 Virtual Backbone in Wireless Networks

Finding the minimum connected dominating set (MCDS) is a key problem in wireless sensor networks, which is crucial for efficient routing and broadcasting.

Wireless networks play a critical role in many areas, such as environmental monitoring, disaster forecast, etc. A key problem in Wireless Networks is multi-hop communication, because the communication range of a individual sensor is generally limited. In multi-hop communication, any two sensors that are within the communication range of each other are called *neighbors*, which can communicate to each other. Other sensors that are not within the communication range of each other and want to communicate, need intermediate sensors between them to forward their packets.

However, due to the broadcasting nature of the wireless communication, if there is not a specific routing path for packet forwarding, all neighbors are possible to become intermediators for forwarding messages, which causes *message flooding* problem. The key way to avoid flooding is to find a communication backbone, so that the packets are relayed by the backbone sensors to save energy for the other sensors.

The wireless network can be formulated as a disk graph as shown in Fig 3.1.

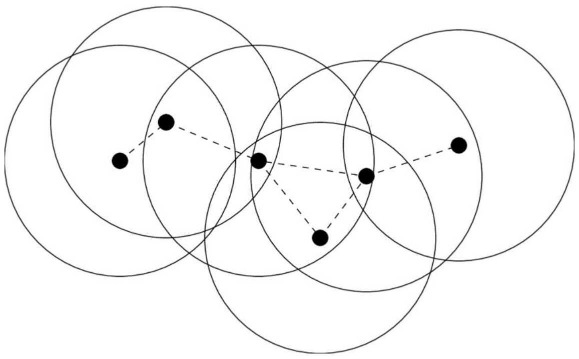


Fig 3.1 WN as Unit disk graph

With this graph, the virtual backbone is a CDS of the graph (Fig 3.2) so that all communications between nodes can be executed through the virtual backbone.

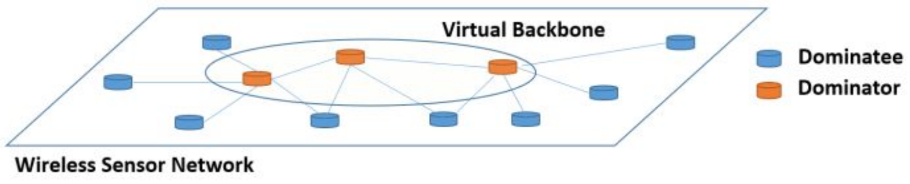


Fig 3.2 CDS based virtual backbone in WN

In fact, the virtual backbone is required to have two properties:

1. Every node not in virtual backbone should be able to directly communicate with (adjacent to) a node in the virtual backbone.

2. All nodes in the virtual backbone should be able to communicate each other within the virtual backbone, i.e. the virtual backbone induces a connected subgraph.

However, forwarding message may run into collision, which introduces retransmissions and increases end-to-end delays. As the number of sensors in the CDS grows, the negative effect of retransmissions increases greatly. Hence, CDS with smaller number of sensors is highly desired, which leads to the problem of finding the CDS with the minimum number of sensors, i.e., the minimum connected dominating set (MCDS) problem. However, it has been proved that the MCDS problem is NP-hard. Therefore, approximation algorithms become the focus of addressing the MCDS problem.

3.2 Converter Placement in Optical Networks

All-optical networks employing wavelength-division multiplexing (WDM) and wavelength routing practice are considered as promising candidates for backbones to meet ever-increasing bandwidth requirements for advanced telecommunication services. In wavelength routed optical networks, communication between end nodes is achieved through all-optical connections called the lightpaths. If network nodes are not equipped with wavelength converters (WCs), each lightpath has to be assigned a unique wavelength over all physical links along the chosen route, which is known as the wavelength continuity constraint.

Due to limited available wavelengths, the wavelength continuity constraint usually results in increased network blocking probability. It has been shown that application of WCs at network nodes has an important role to avoid the wavelength conflicts along the path and consequently reduce the overall network blocking probability. If all network nodes are equipped with WCs, it is referred to as a network with full wavelength conversion.

However, considering that WCs are still highly expensive components, it may not be economical to implement the converters in all nodes. Furthermore, it has been shown that if only some nodes are equipped with wavelength converters, the network blocking probability could be almost to that with full wavelength conversion.

Therefore, a more realistic scenario is a network with sparse wavelength conversion, which means only a sub-set of network nodes have the capability of wavelength conversion.

Hence, a minimization problem is formulated as follows.

CONVERTER PLACEMENT: Given a graph G = (V, E) and color-sets for each edge of G such that for every color all edges in the color form a connected subgraph, find the minimum number of vertices such that placing converters on them would connect some colors into a connected spanning subgraph of G.

CONVERTER PLACEMENT can be reduced to MIN-CDS.
To do so, we construct another graph G' with vertex set V. Two vertices u and v are connected with an edge if and only if they are in the same color of G. Without loss of generality, we may assume that no color covers all vertices because in such a case, no converter is required. Under the assumption, we can show that a vertex subset C is a feasible solution for CONVERTER-PLACEMENT if and only if C is a CDS in G'.
First, suppose C is a feasible solution. Then every vertex x must be adjacent to a converter; otherwise, it cannot communicate with any converter which is also a vertex in G. Moreover, C must induce a connected subgraph in G' since, otherwise, two converters in different connected components cannot communicate each other. Therefore, C is a CDS in G'.

Conversely, if C is a CDS in G', then there is a spanning tree T of G' with all internal vertices in C. Thus, placing converters at all vertices in C would connect all colors appearing in T together, which is clearly covering T. Therefore, C is a feasible solution for CONVERTER PLACEMENT.

4. Connected Domination Number

We have seen that MIN-CDS is NP-complete, i.e. there doesn't exist any polynomial time algorithm for MIN-CDS. Many families of graphs have connected domination number in closed form. Some are given below:

1. Antiprism Graph

An antiprism graph is a graph corresponding to the skeleton of an antiprism. Antiprism graphs are therefore polyhedral and planar. The n-antiprism graph has 2n vertices and 4n edges.

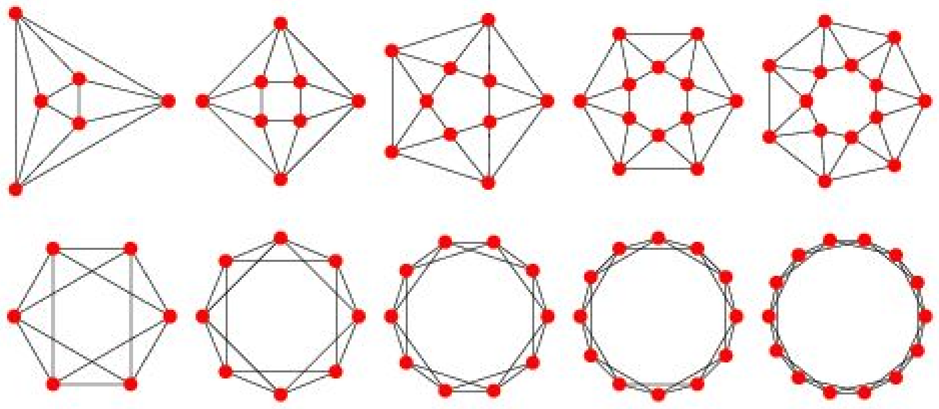


Fig 4.1 Antiprism graphs

The Connected Domination Number of Antiprism graph,

$$\gamma_c(G) = n-1$$

2. Barbell Graph

The n-barbell graph is the simple graph obtained by connecting two copies of a complete graph K_n by a bridge.

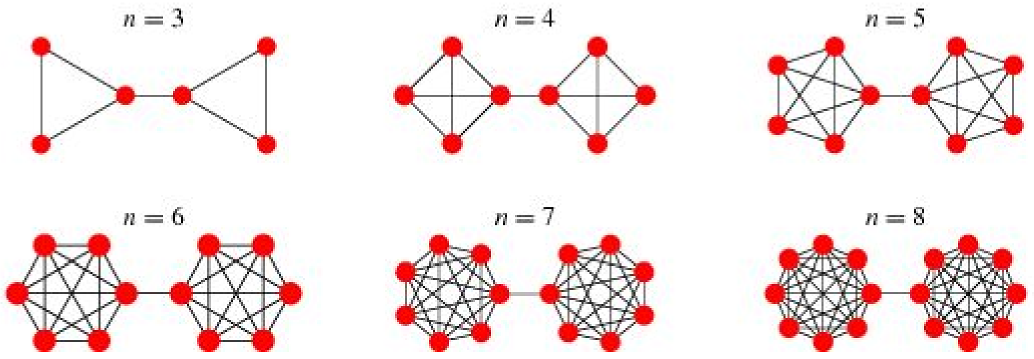


Fig 4.2 Barbell graphs

The Connected Domination Number of Barbell graph,

$$\gamma_c(G) = 2$$

3. Black Bishop Graph ($BB_{n,n}$)

A black bishop graph is a graph formed from possible moves of a bishop chess piece, which may make diagonal moves of any length on a chessboard (or any other board), when starting from a black square on the board.

To form the graph, each chessboard square is considered a vertex, and vertices connected by allowable bishop moves are considered edges.

The connected Domination number of Black Bishop graph,

$$\gamma_c(G) = \begin{cases} 1 & \text{for } n=1,2 \\ n-2 & \text{otherwise} \end{cases}$$

4. Book Graph

The m-book graph is defined as the graph Cartesian product $S_{(m+1)} \times P_2$, where S_m is a star graph and P_2 is the path graph on two nodes.

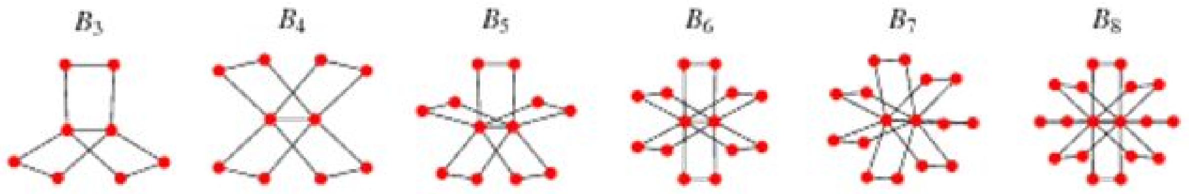


Fig 4.3 Book graph

The Connected Domination Number of Book graph,

$$\gamma_c(G) = 2$$

5. Cocktail Party Graph

The cocktail party graph of order n, also called the hyperoctahedral graph or Roberts graph, is the graph consisting of two rows of paired nodes in which all nodes, but the paired ones are connected with a graph edge.

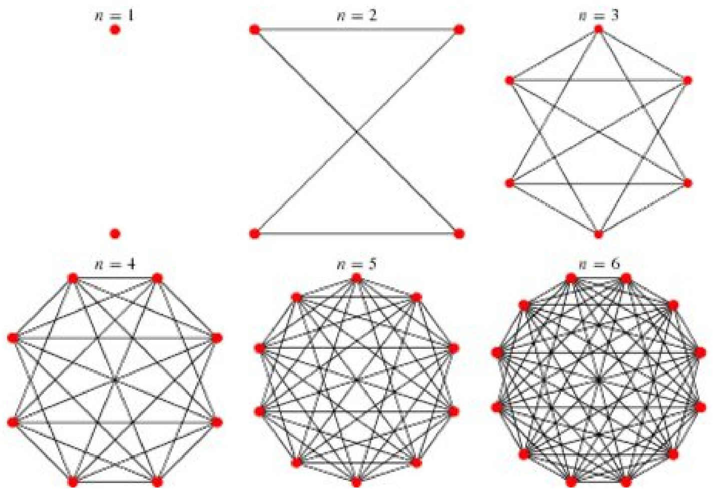


Fig 4.4 Cocktail Party Graph

The Connected Domination Number of Cocktail Party Graph,

$$\gamma_c(G) = 2$$

6. Complete Bipartite Graph ($K_{m,n}$)

A complete bipartite graph, sometimes also called a complete bicolored graph or complete bigraph, is a bipartite graph (i.e., a set of graph vertices decomposed into two disjoint sets such that no two graph vertices within the same set are adjacent) such that every pair of graph vertices in the two sets are adjacent. If there are p and q graph vertices in the two sets, the complete bipartite graph is denoted $K_{p,q}$.

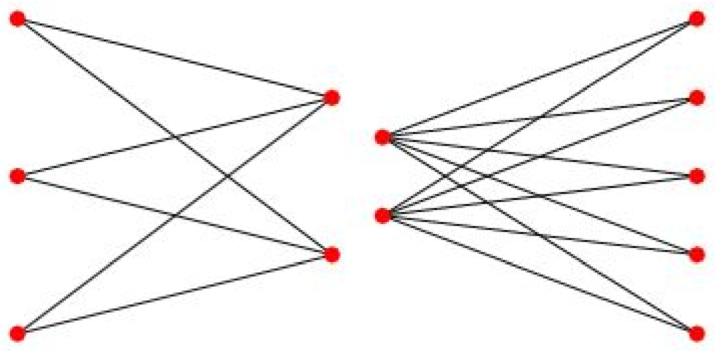


Fig 4.5 Complete Bipartite Graph, $K_{3,2}$ and $K_{2,4}$

The Connected Domination Number of Cocktail Party Graph,

$$\gamma_c(G) = 1$$

7. Complete Graph (K_n)

A complete graph is a graph in which each pair of graph vertices is connected by an edge. The complete graph with n graph vertices is denoted K_n and has = (the triangular numbers) undirected edges, where is a binomial coefficient.

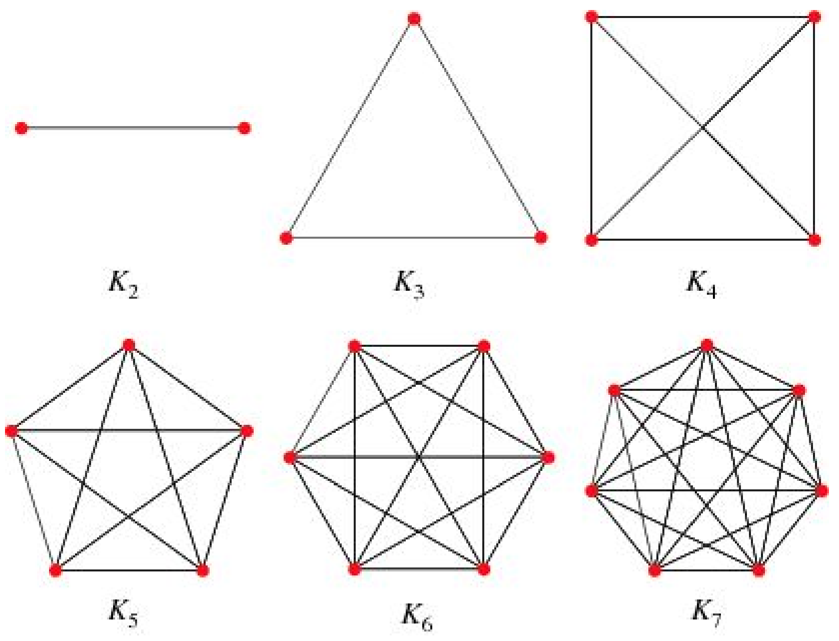


Fig 4.6 Complete Graph

The Connected Domination Number of Complete graph,

$$\gamma_c(G) = 1$$

8. 2n-Crossed Prism Graph

A n-crossed prism graph for positive even n, is a graph obtained by taking two disjoint cycle graphs C_n and adding edges (v_k, v_{2k+1}) and (v_{k+1}, v_{2k}) for k=1, 3, ..., (n-1).

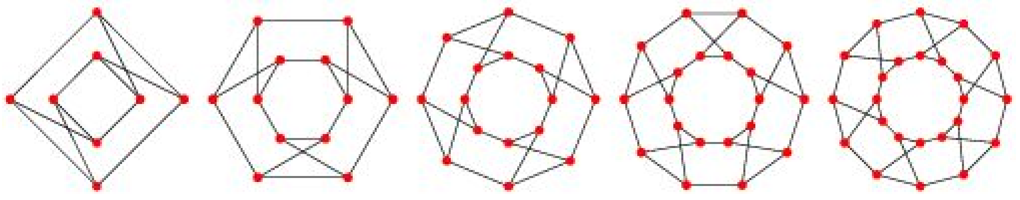


Fig 4.7 Crossed Prism Graph

The Connected Domination Number of Crossed Prism graph,

$$\gamma_c(G) = 2n$$

9. Crown Graph

The n-crown graph for an integer n>=3 is the graph with vertex set $\{x_0,x_1,...,x_{n-1},y_0,y_1,...,y_{n-1}\}$

and edge set $\{(x_i,y_j): 0<= i,j<= n-1, i!=j\}$.

It is therefore equivalent to the complete bipartite graph $K_{n,n}$ with horizontal edges removed.

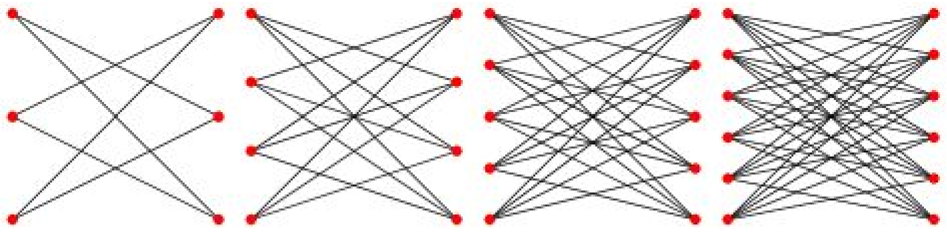


Fig 4.8 Crown Graph

The Connected Domination Number of Crown graph,

$$\gamma_c(G) = 4$$

10. Cycle Graph (C_n)

A cycle graph C_n, sometimes simply known as an n-cycle, is a graph on n nodes containing a single cycle through all nodes.

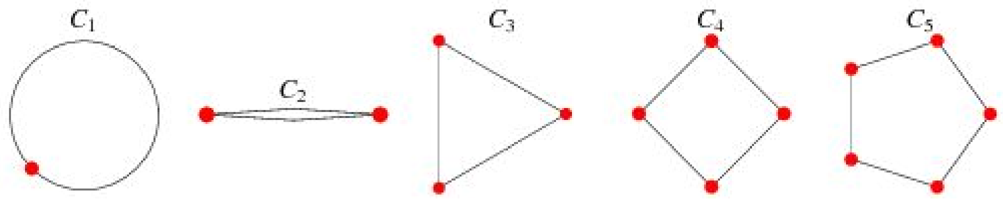


Fig 4.9 Cycle Graph

The Connected Domination Number of Cycle graph,

$$\gamma_c(G) = n-2$$

11. Gear Graph (G_n)

The gear graph, also sometimes known as a bipartite wheel graph, is a wheel graph with a graph vertex added between each pair of adjacent graph vertices of the outer cycle. The gear graph G_n has 2n+1 nodes and 3n edges.

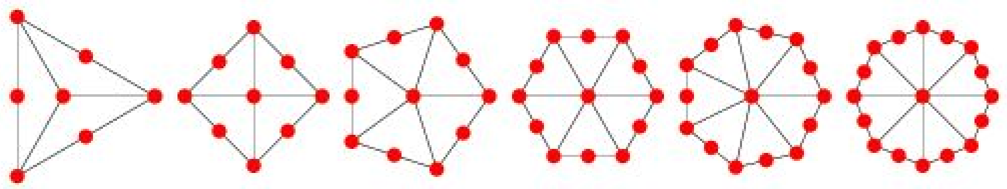


Fig 4.10 Gear Graph

The Connected Domination Number of Gear graph,

$$\gamma_c(G) =$$

12. Helm Graph (H_n)

The Helm graph H_n is the graph obtained from an n-wheel graph by adjoining a pendant edge at each node of the cycle.

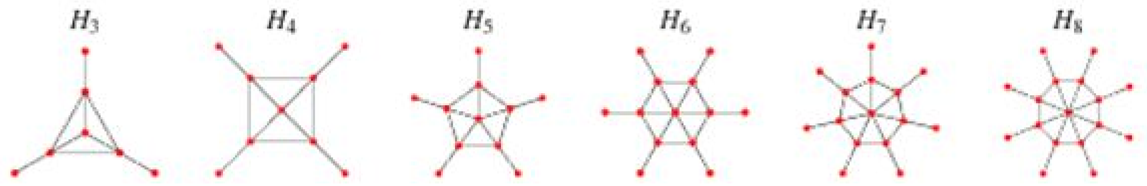


Fig 4.11 Helm Graph

The Connected Domination Number of Helm graph,

$$\gamma_c(G) = n$$

14. Path Graph (P_n)

The path graph P_n is a tree with two nodes of vertex degree 1, and the other n-2 nodes of vertex degree 2. A path graph is therefore a graph that can be drawn so that all of its vertices and edges lie on a single straight line.

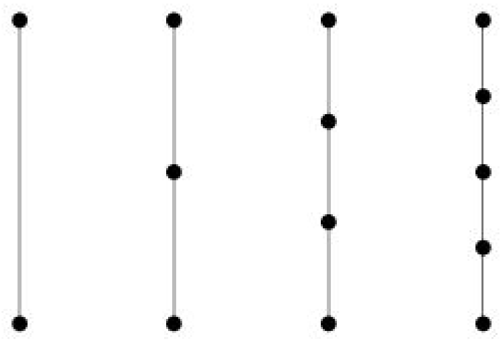


Fig 4.12 Path Graph

The Connected Domination Number of Path graph,

$$\gamma_c(G) = \begin{cases} 1 & \text{for } n=1,2 \\ n-2 & \text{otherwise} \end{cases}$$

15. Star Graph (S_n)

The star graph S_n of order n, sometimes simply known as an "n-star", is a tree on n nodes with one node having vertex degree n-1 and the other n-1 having vertex degree 1.

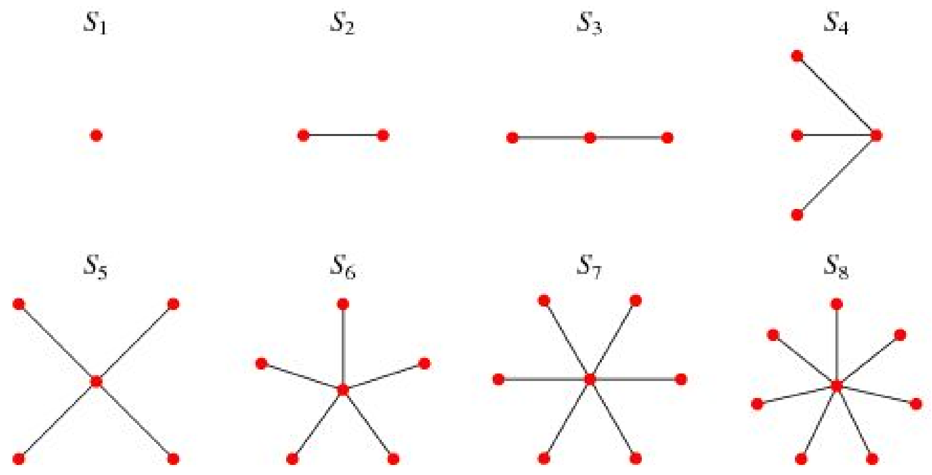


Fig 4.13 Star Graph

The Connected Domination Number of Star graph,

$$\gamma_c(G) = 1$$

5. Maximum Leaf Spanning Tree problem

Given an undirected graph with n vertices, the Maximum Leaf Spanning Tree problem (MLST) is to find a spanning tree with as many leaves as possible.

In the sense of exact algorithms, the Maximum Leaf Spanning Tree problem is equivalent to the Minimum Connected Dominating Set problem (MCDS) as the internal nodes of a spanning tree with k leaves are a connected dominating set of size n-k and vice versa.

5.1 An exact algorithm to MLST

5.1.1 Preliminaries

Let G = (V,E) be a simple, undirected graph. We denote by n, the number of its vertices and by m, the number of its edges. Given a vertex $v \in V$, the set of its neighbors is defined by $N(v) = \{u \in V \mid \{u,v\} \in E\}$. The closed neighborhood of v is $N[v] = \{v\} \cup N(v)$. Given a subset $S \subseteq V$, we define N(S) as the set $\cup_{v \in S}$ $N(v) \setminus S$ and for a $X \subseteq V$, we define $N_X(S) = N(S) \cap X$. We write $H \subseteq G$ if H is a subgraph of G.

Let G = (V,E) be the input graph to the MLST problem and consider a partition $V = \text{Free} \cup \text{FL} \cup \text{BN} \cup \text{LN} \cup \text{IN}$ of the vertex set into the sets of free vertices (Free), floating leaves (FL), branching nodes (BN), leaf nodes (LN), and internal nodes (IN).

Definition 1. Let G= (V,E) be a graph, and let IN, BN, LN, $FL \subseteq V$ be disjoint sets of vertices and $T \subseteq G$ be a tree. We say T extends (IN, BN, LN, FL) iff $IN \subseteq \text{internal}(T)$, $LN \subseteq \text{leaves}(T)$, $BN \subseteq \text{internal}(T) \cup \text{leaves}(T)$, and $FL \cap \text{internal}(T) = \emptyset$.

Definition 2. Let G = (V,E) be a graph, let IN, BN, LN, $FL \subseteq V$ be disjoint sets of vertices, and let $x_1,...,x_l \in V$. By $x_1 \rightarrow X_1$, ..., $xl \rightarrow X_l$, where each X_i is one of IN, BN, LN or FL, we denote the operation of moving each x_i to the respective set X_i, and additionally, if $X_i = IN$, of moving all $y \in N_{Free}(x_i)$ to BN and all $y \in N_{FL}(x_i)$ to LN. The notation is extended to $Y \rightarrow X$, where $Y \subseteq V$, in a straightforward manner. To solve an instance, the algorithm considers subinstances obtained from G, IN, BN, LN, FL by applying a set of operations of the form $x \rightarrow X$. We write $\langle x_{1,1} \rightarrow X_{1,1},...,x_{1,l1} \rightarrow X_{1,l1} \| ... \| x_{k,1} \rightarrow X_{k,1} ...,x_{k,lk} \rightarrow X_{k,lk} \rangle$, to express that the algorithm branches into k subinstances, where in the jth call, $1 \leq j \leq k$, the algorithm considers the subinstance obtained from G, IN, BN, LN, FL by applying the operations

$x_{j,1} \rightarrow X_{j,1}$ to $x_{j,lj} \rightarrow X_{j,lj}$.

For any $v \in V \setminus (IN \cup LN)$, we define its degree d(v) as $d(v) = |N(v) \cap (\text{Free} \cup FL)|$ if $v \in BN$, as $d(v) = |N(v) \cap (\text{Free} \cup FL \cup BN)|$ if $v \in \text{Free}$, and as $d(v) = |N(v) \cap (\text{Free} \cup BN)|$ if $v \in FL$. A vertex $v \in \text{Free} \cup FL$ is unreachable, if there is no path $uv_1 ...v_tv$, where $t \geq 0$, $u \in BN$ and $v_i \in \text{Free}$ for all $1 \leq i \leq t$. We note that if $BN = \emptyset$, then every vertex in $FL \cup \text{Free}$ is unreachable.

5.1.2 The Algorithm

An algorithm for Maximum Leaf Spanning Tree is given in Fig 5.1. The notation ⟨v → IN || v → LN⟩ describes the corresponding branches, e.g., in this case v either becomes an internal node or a leaf (see Definition 2).

```
Algorithm $\mathcal{M}$
Input: A graph $G = (V, E)$, IN, BN, LN, FL $\subseteq V$
Reduce $G$ according to the reduction rules.
if there is some unreachable $v \in$ Free $\cup$ FL then return 0
if $V =$ IN $\cup$ LN then return |LN|
Choose a vertex $v \in$ BN of maximum degree.
if $d(v) \geq 3$ or ($d(v) = 2$ and $N_{FL}(v) \neq \emptyset$) then
    $\langle v \to LN \,||\, v \to IN\rangle$
else if $d(v) = 2$ then
    Let $\{x_1, x_2\} = N_{Free}(v)$ such that $d(x_1) \leq d(x_2)$.
    if $d(x_1) = 2$ then
        Let $\{z\} = N(x_1) \setminus \{v\}$
        if $z \in$ Free then
            $\langle v \to LN \,||\, v \to IN, x_1 \to IN \,||\, v \to IN, x_1 \to LN\rangle$
        else if $z \in$ FL then $\langle v \to IN\rangle$
    else if $(N(x_1) \cap N(x_2)) \setminus FL = \{v\}$ and $\forall z \in (N_{FL}(x_1) \cap N_{FL}(x_2))$,
        $d(z) \geq 3$ then
        $\langle v \to LN \,||\, v \to IN, x_1 \to IN \,||\, v \to IN, x_1 \to LN, x_2 \to IN \,||$
         $v \to IN, x_1 \to LN, x_2 \to LN,$            $N_{Free}(\{x_1, x_2\}) \to FL, N_{BN}(\{x_1, x_2\}) \setminus \{v\} \to LN\rangle$
    else $\langle v \to LN \,||\, v \to IN, x_1 \to IN \,||\, v \to IN, x_1 \to LN, x_2 \to IN\rangle$
else if $d(v) = 1$ then
    Let $P = (v = v_0, v_1, \ldots, v_k)$ be a maximum path such that
        $d(v_i) = 2, 1 \leq i \leq k, v_1, \ldots, v_k \in$ Free.
    Let $z \in N(v_k) \setminus V(P)$.
    if $z \in$ FL and $d(z) = 1$ then $\langle v_0, \ldots, v_k \to IN, z \to LN\rangle$
    else if $z \in$ FL and $d(z) > 1$ then $\langle v_0, \ldots, v_{k-1} \to IN, v_k \to LN\rangle$
    else if $z \in$ BN then $\langle v \to LN\rangle$
    else if $z \in$ Free then $\langle v_0, \ldots, v_k \to IN, z \to IN \,||\, v \to LN\rangle$
```

Fig 5.1 Algorithm M

To compute a solution for Maximum Leaf Spanning Tree of a given graph G = (V,E), algorithm M must be called for each vertex v ∈V with IN={v},BN=N(v) and LN=FL=∅.

6. Two-Stage Greedy Approximation

6.1 Greedy Algorithm GK

Consider a graph G and a subset C of vertices in G.

We divide all vertices in G into three classes with respect to C:

- Black vertices: vertices in C.
- Grey vertices: vertices not in C but dominated by black vertices.
- White vertices: vertices not dominated by black vertices.

Clearly, C is a CDS if and only if there does not exist a white vertex and the subgraph induced by black vertices is connected.

Let p(C) be the number of connected components of G[C], the subgraph of G induced by C, and h(C) the number of white vertices. Let g(C)=p(C)+h(C). Then, C is a CDS if and only if g(C)=1.

Pseudo Code:

```
Greedy Algorithm GK:
input a connected graph $G$.
Set $C \leftarrow \emptyset$;
while there exists a vertex $x$ such that $g(C \cup \{x\}) < g(C)$ do
        choose a vertex $x$ to minimize $g(C \cup \{x\})$ and
        set $C \leftarrow C \cup \{x\}$;
output $C$.
```

6.2 Guha-Khuller Algorithm

Greedy GK algorithm may not output a CDS as shown in Figure 4.2.

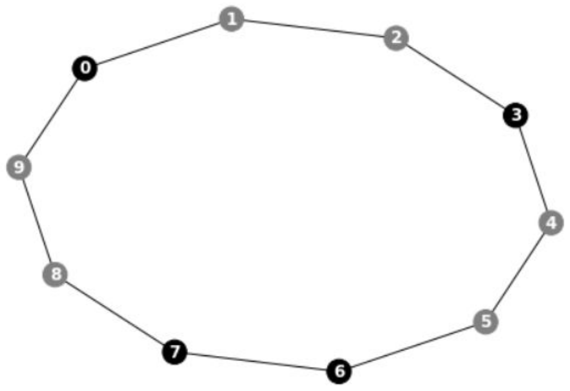


Fig 6.1 Greedy GK on C10

Indeed, even if for vertex x, g(C ∪ {x})=g(C), C may not be a CDS. If a white vertex exists, then let x be a gray vertex adjacent to a white vertex, then we must have g(C ∪ {x}) < g(C). Therefore, for C obtained from Greedy Algorithm GK, no white vertex exists. This means that if output C is not a CDS, then C does not induced a connected subgraph.

In this two-stage greedy approximation, stage 1 is a greedy algorithm computing a dominating set and stage 2 connects this dominating set into a connected one. In the potential function g(C), h(C) is used for issuing that Stage 1 gives a dominating set, and p(C) is used for making the number of black connected components smaller.

Pseudo Code:

```
Guha–Khuller Algorithm:
input a connected graph $G$.
Stage 1
        Employ Greedy Algorithm GK to obtain a dominating set $C$;
Stage 2
        while there are more than one black components do
              find a chain of two gray vertices $x$ and $y$ connecting at least
              two black components and $C \leftarrow C \cup \{x,y\}$;
output $C$.
```

Running Guha Khuller algorithm on output shown in Fig 4.2 gives CDS, as shown in Fig 4.3

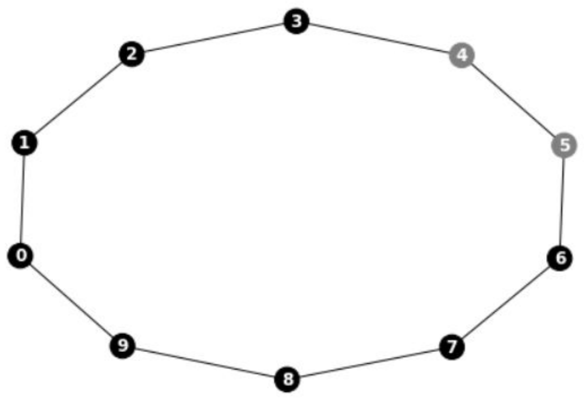


Fig 6.2 Guha Khuller on C10

7. Weakly CDS

Consider a graph G =(V,E). For any vertex subset C, denote by q(C) the number of connected components of the subgraph with vertex set V and the edge set consisting of all edges incident to vertices inC. A dominating set C is called a weakly CDS (WCDS) if q(C)=1.

Chen and Liestman studied the following problem.

MIN-WCDS: Given a graph G, find a WCDS with the minimum cardinality.

7.1 Chen–Liestman Algorithm

Chen Liestman designed a greedy algorithm with potential function q(C).

Pseudo Code:

```
**Chen–Liestman Algorithm**
input graph $G = (V, E)$.
$C \leftarrow \emptyset$;
while $q(C) \geq 2$ do
    choose $u \in V$ to mimimize $q(C \cup \{u\})$
    $C \leftarrow C \cup \{u\}$;
output $C$.
```

Chen–Liestman Algorithm produces an approximation solution within a factor of (1+ln δ) from optimal, where δ is the maximum vertex degree of input graph.

7.2 Wolsey Greedy Algorithm

Let f(A)=|V|−q(A), where V is the vertex set of the graph, the wolsey greedy algorithm solves MIN-WCDS.

Pseudo Code:

```
**Wolsey Greedy Algorithm**
input a monotone increasing submodular function $f : 2^X \to R$;
Initially, set $A \leftarrow \emptyset$;
while $f(A) < f(X)$ do
    choose $x \in X - A$ to maximize $\frac{\Delta_x f(A)}{c(x)}$
    $A \leftarrow A \cup \{x\}$;
output $A$.
```

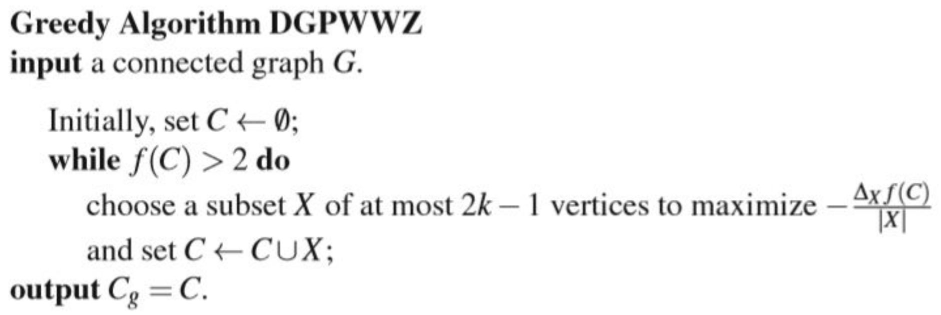
Here, take c(x) to be constant.

8. One-Stage Greedy Approximation

Let q(C) the number of connected components of the subgraph with vertex set V and edge set D(C), where D(C) be the set of all edges incident to vertices in C.

Define f(C) = p(C) + q(C).

Pseudo Code:



```
Greedy Algorithm DGPWWZ
input a connected graph $G$.
    Initially, set $C \leftarrow \emptyset$;
    while $f(C) > 2$ do
        choose a subset $X$ of at most $2k-1$ vertices to maximize $-\frac{\Delta_X f(C)}{|X|}$
        and set $C \leftarrow C \cup X$;
output $C_g = C$.
```

However, choosing the subset X of at most 2k-1 can be computationaly expensive. This involves two steps:

1. Choosing the no. of vertices of X, let say k' which can be at most 2k-1.
2. Choosing the subset X of k' vertices for which the function is minimum.

A good choice of k' can be obtained from sampling. We have used the following procedure to get k':

Take intial range as 1...k...2k-1. Calculate the value of the function at three values,

1. mid value of the range 1… k.
2. min value of the range 1… 2k-1
3. mid value of the range k… 2k-1

Now, run the following steps to get the k'.

- If the function is maximum at first value, then return the corresponding value.
- If function is maximum at second value, then store the corresponding value of k and the maximum value of function and iterate again with the new range as 1...k.
- If function is maximum at third value, then store the corresponding value of k and the maximum value of function and iterate again with the new range as k...2k-1
- Otherwise, return the stored value of k.

9. Experiments

We have run the above four algorithms on different types of graph.

9.1 Complete Graph

	DPGWWZ		Wolsey Greedy		Chen Liestman		Guha Khuller	
no_of_vertices	cds_size	time	cds_size	time	cds_size	time	cds_size	time
100	1	0.113326	1	0.149215	1	0.060982	1	0.381949
200	1	0.242593	1	0.572781	1	0.237488	1	1.572971
300	1	0.355767	1	1.413814	1	0.673587	1	3.696604

Fig 9.1 Sample output on complete graph

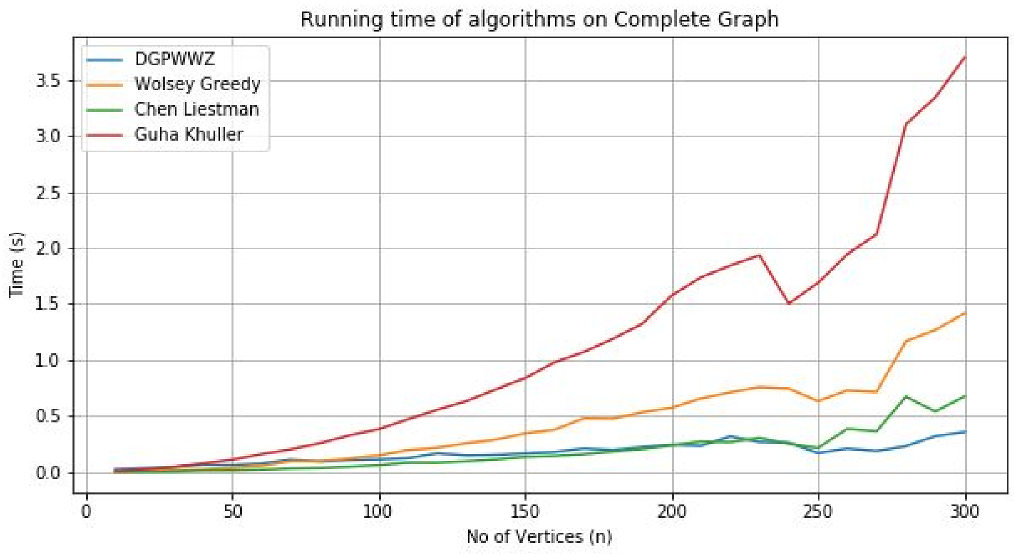


Fig 9.2 Time taken on complete graph on n vertices

9.2 Star Graph

	DPGWWZ		Wolsey Greedy		Chen Liestman		Guha Khuller	
no_of_vertices	cds_size	time	cds_size	time	cds_size	time	cds_size	time
101	1	0.066995	1	0.129926	1	0.065503	1	0.402127
201	1	0.155618	1	0.494703	1	0.230466	1	1.549033
301	1	0.234265	1	1.094192	1	0.55034	1	3.013681

Fig 9.3 Sample output on star graph

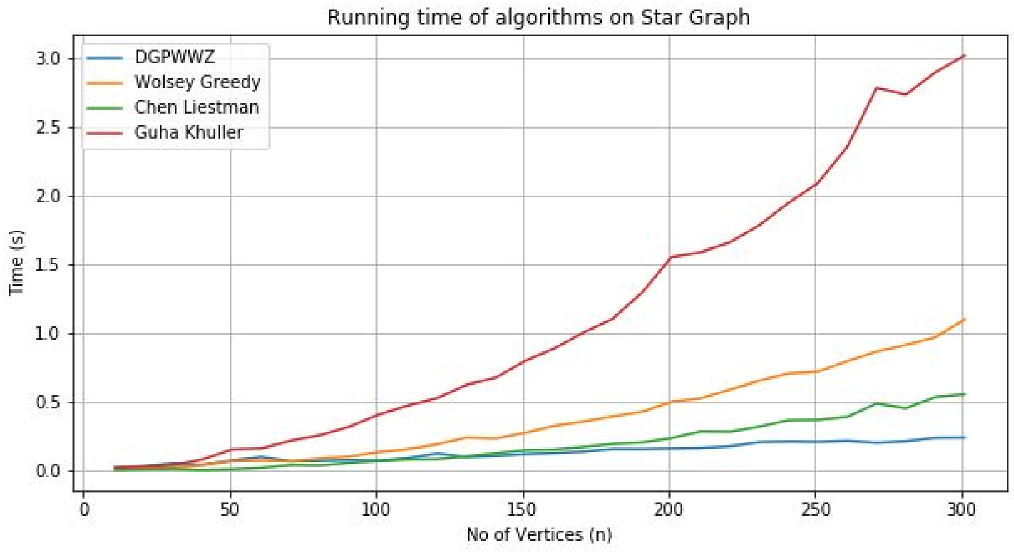


Fig 9.4 Time taken on star graph on n vertices

9.3 Wheel Graph

	DPGWWZ		Wolsey Greedy		Chen Liestman		Guha Khuller	
no_of_vertices	cds_size	time	cds_size	time	cds_size	time	cds_size	time
100	1	0.085014	1	0.121578	1	0.061866	1	0.370764
200	1	0.145492	1	0.51022	1	0.237255	1	1.401913
300	1	0.338904	1	1.279111	1	0.65185	1	2.983832

Fig 9.5 Sample output on wheel graph

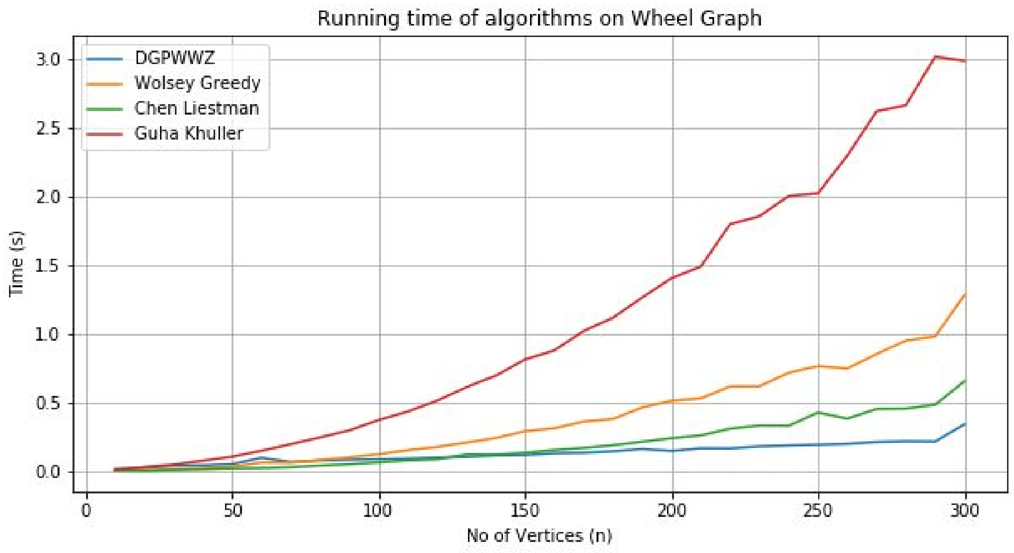


Fig 9.6 Time taken on wheel graph on n vertices

9.4 Path Graph

	DPGWWZ		Wolsey Greedy		Chen Liestman		Guha Khuller	
no_of_vertices	cds_size	time	cds_size	time	cds_size	time	cds_size	time
100	99	0.916313	50	4.831937	50	2.52329	98	10.38066
180	179	2.476072	90	28.03918	90	13.93811	178	51.91247
240	238	4.710075	120	69.35718	120	40.57259	238	141.9399

Fig 9.7 Sample output on path graph

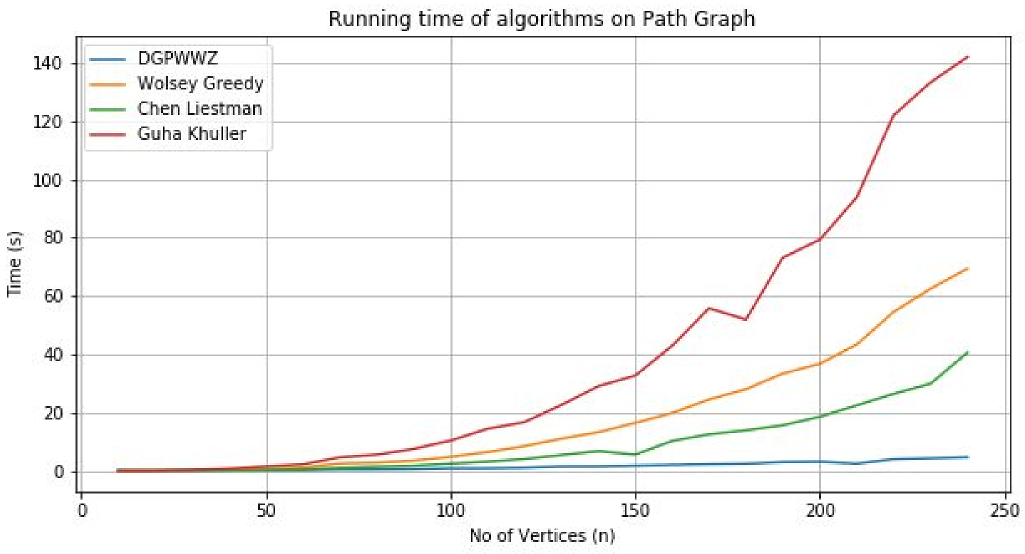


Fig 9.8 Time taken on path graph on n vertices

9.5 Ladder Graph

	DPGWWZ		Wolsey Greedy		Chen Liestman		Guha Khuller	
no_of_vertices	cds_size	time	cds_size	time	cds_size	time	cds_size	time
100	50	0.610201	38	4.036603	38	2.026856	64	11.74933
180	90	1.563004	68	22.62572	68	10.98686	118	59.35712
260	130	3.040276	98	67.83789	98	33.76533	171	192.3167

Fig 9.9 Sample output on ladder graph

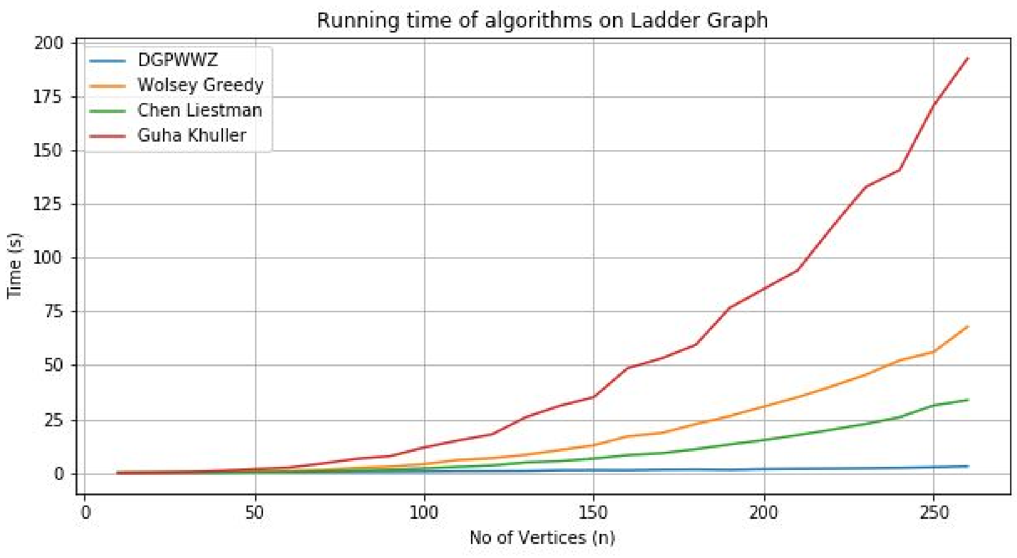
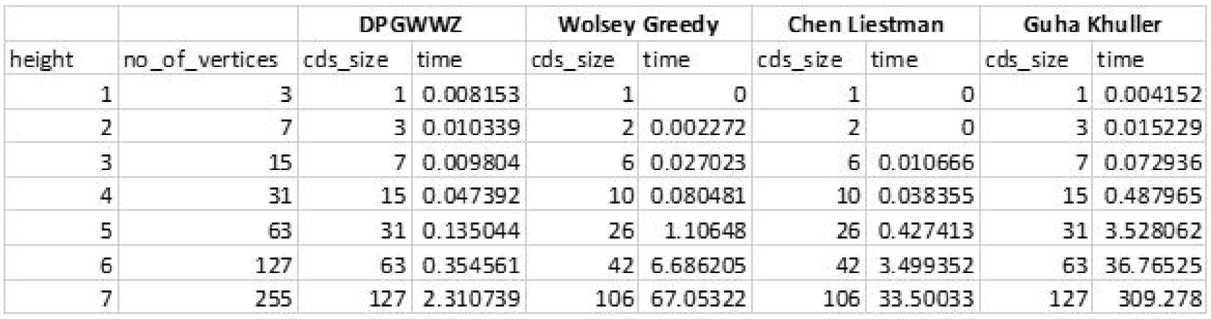


Fig 9.10 Time taken on ladder graph on n vertices

9.6 Balanced Binary Tree



		DPGWWZ		Wolsey Greedy		Chen Liestman		Guha Khuller	
height	no_of_vertices	cds_size	time	cds_size	time	cds_size	time	cds_size	time
1	3	1	0.008153	1	0	1	0	1	0.004152
2	7	3	0.010339	2	0.002272	2	0	3	0.015229
3	15	7	0.009804	6	0.027023	6	0.010666	7	0.072936
4	31	15	0.047392	10	0.080481	10	0.038355	15	0.487965
5	63	31	0.135044	26	1.10648	26	0.427413	31	3.528062
6	127	63	0.354561	42	6.686205	42	3.499352	63	36.76525
7	255	127	2.310739	106	67.05322	106	33.50033	127	309.278

Fig 9.11 Sample output on balanced binary tree

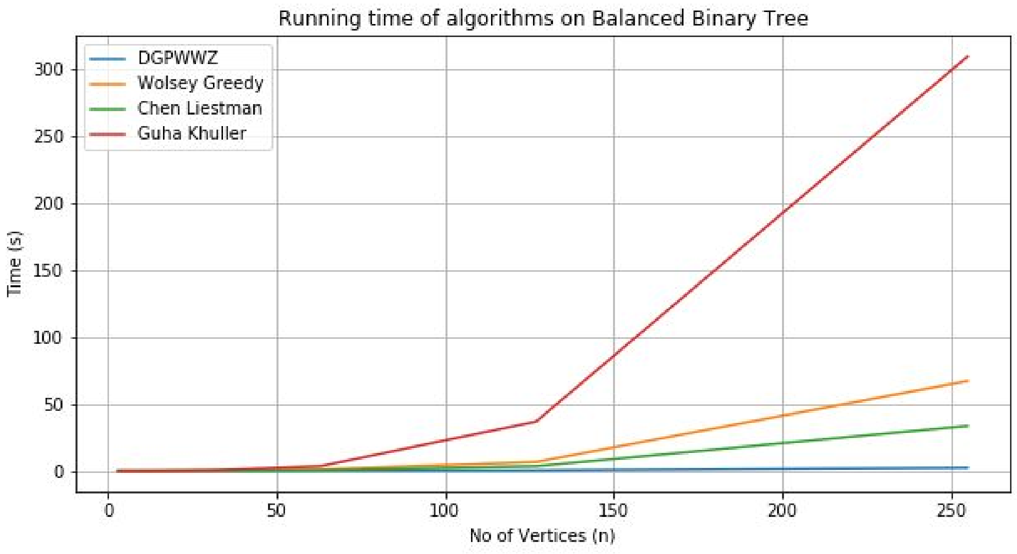
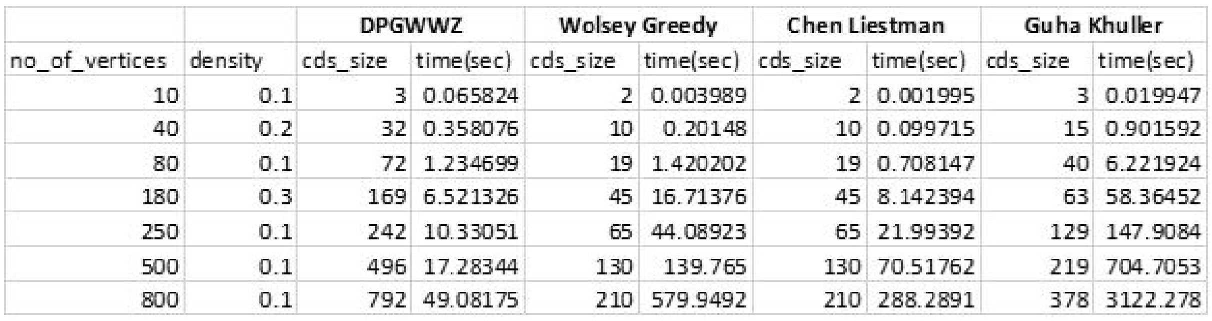


Fig 9.12 Time taken on balanced binary tree on n vertices

9.7 Random Graph

We have used Connected Watts-Strogatz model as the random graph generation model. It produces graphs with small-world properties, including short average path lengths and high clustering.
The graph is generated on the two parameters, i.e. no of vertices and the edge density of the graph.

9.7.1 Results



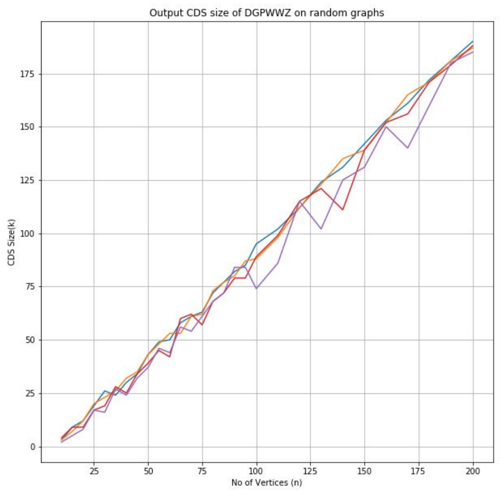
		DPGWWZ		Wolsey Greedy		Chen Liestman		Guha Khuller	
no_of_vertices	density	cds_size	time(sec)	cds_size	time(sec)	cds_size	time(sec)	cds_size	time(sec)
10	0.1	3	0.065824	2	0.003989	2	0.001995	3	0.019947
40	0.2	32	0.358076	10	0.20148	10	0.099715	15	0.901592
80	0.1	72	1.234699	19	1.420202	19	0.708147	40	6.221924
180	0.3	169	6.521326	45	16.71376	45	8.142394	63	58.36452
250	0.1	242	10.33051	65	44.08923	65	21.99392	129	147.9084
500	0.1	496	17.28344	130	139.765	130	70.51762	219	704.7053
800	0.1	792	49.08175	210	579.9492	210	288.2891	378	3122.278

Fig 9.13 Sample output on random graphs

9.7.2 PLOTS

Following are the plots generated from the results:

DGPWWZ Algorithm



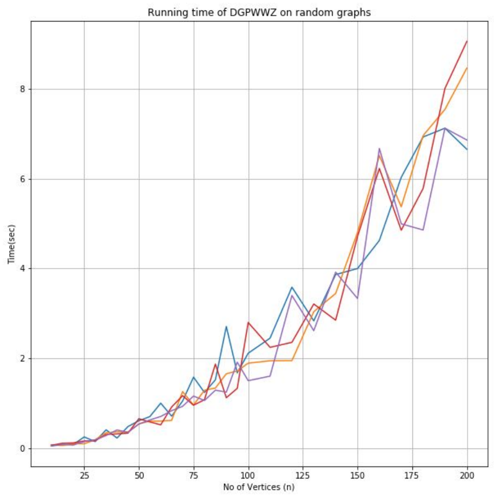
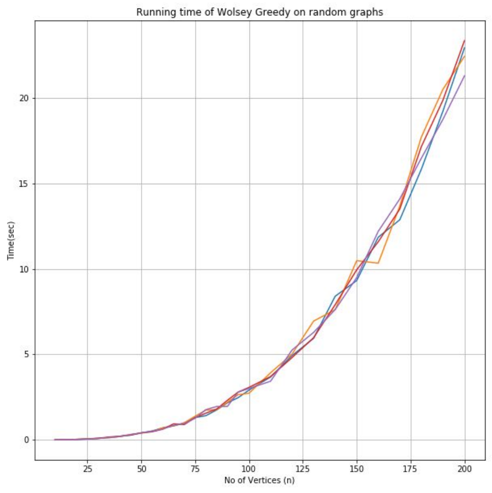


Fig 9.14 CDS size and running time taken by DPGWWZ

WOLSEY GREEDY



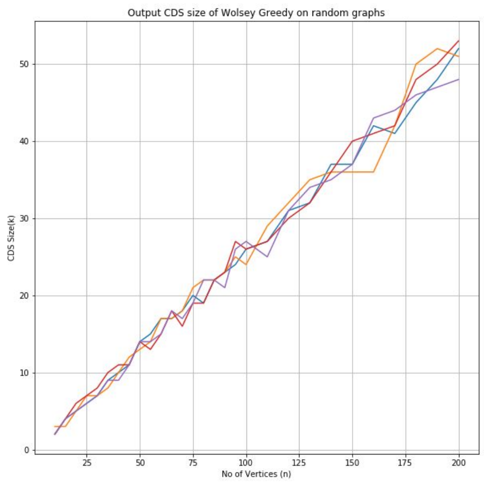
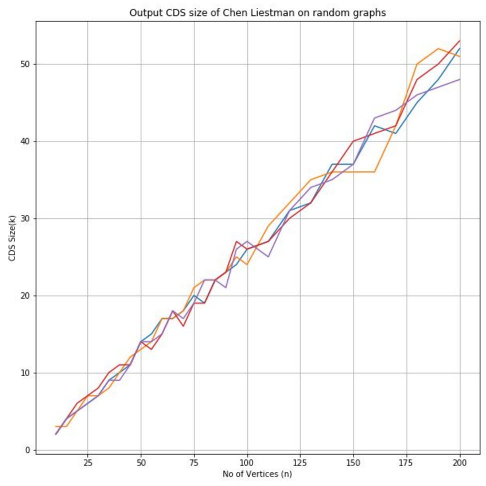


Fig 9.15 CDS size and running time taken by wolsey greedy

CHEN LIESTMAN



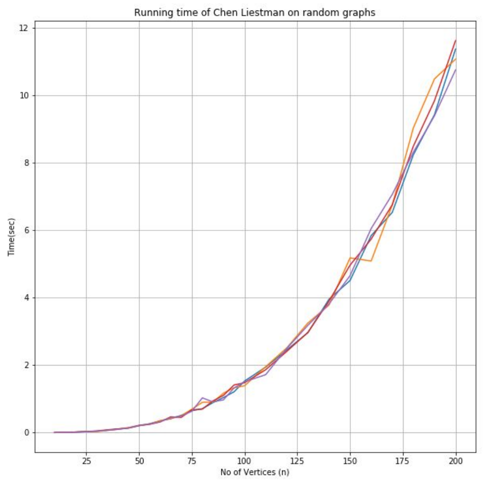
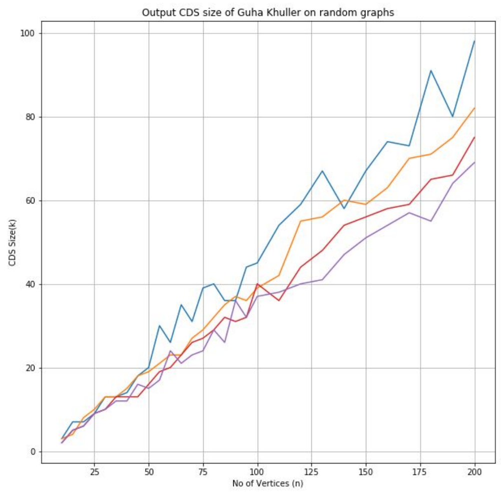


Fig 9.16 CDS size and running time taken by chen liestman

GUHA KHULLER



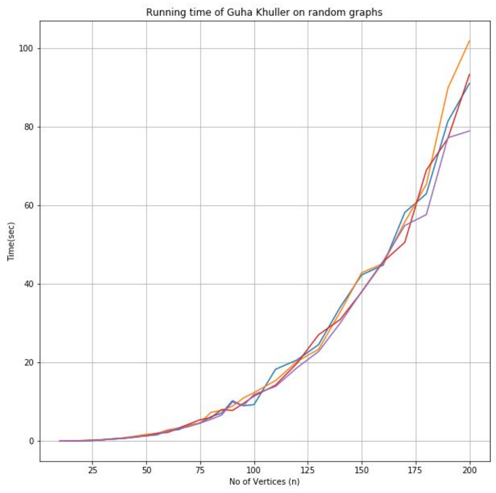


Fig 9.17 CDS size and running time taken by guha khuller

9.7.3 Comparison

Plots are generated to compare various algorithms on the basis of time and CDS size. Plots are shown in Fig 9.18 and Fig 9.19

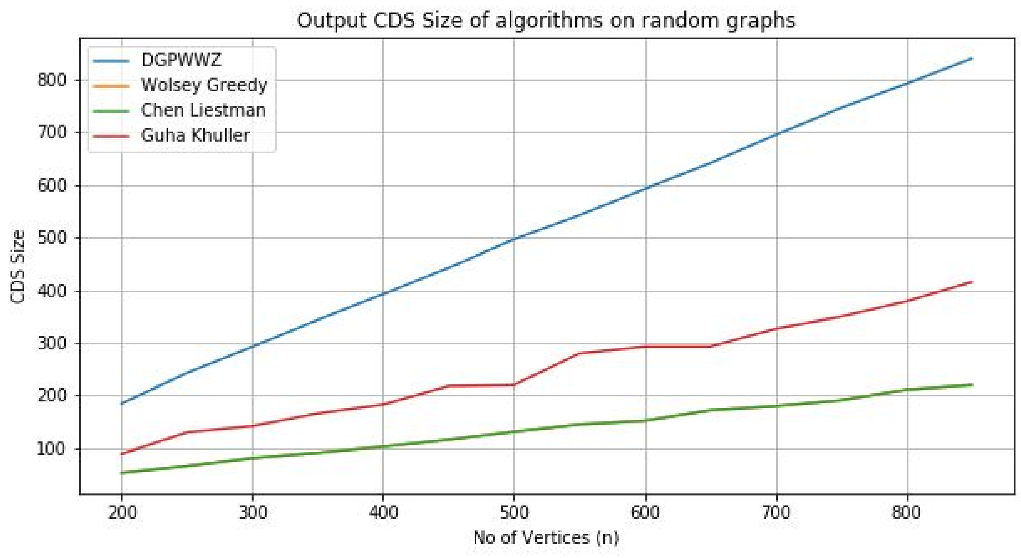


Fig 9.18 CDS size obtained by various algorithms

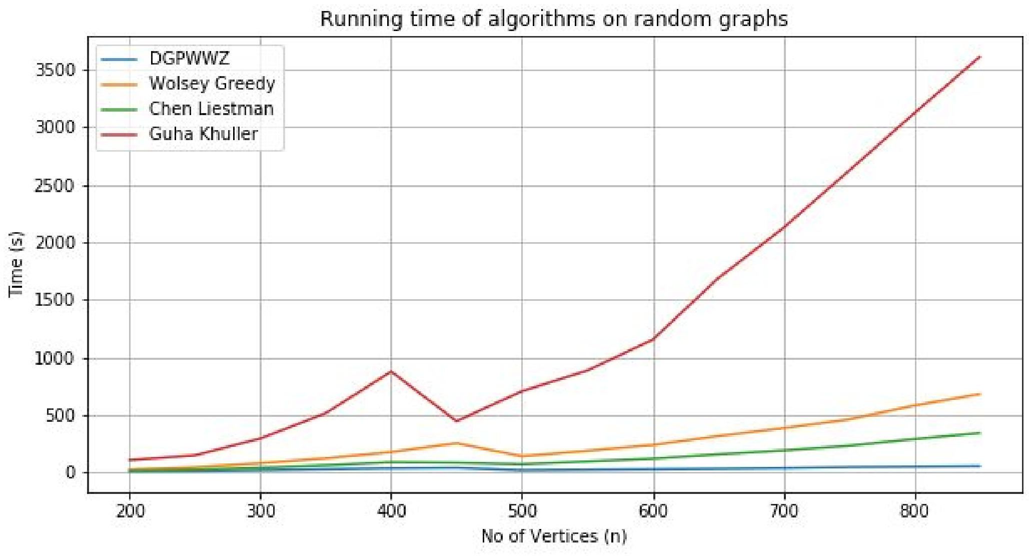


Fig 9.19 Time taken by various algorithms

10. The Penalty Algorithm

10.1 The Algorithm

The Penalty Algorithm is an iterative greedy algorithm which makes use of the definition of penalty as provided in the previous section.

The algorithm works similarly to most iterative greedy algorithms. It initialises an empty set and iteratively chooses a vertex to add into that set based on the highest penalty of all the vertices. It stops when the graph has been dominated and returns the set as the minimal dominating set.

The following algorithm contains an outline of the complete process.

```
Algorithm 1: Iterative Greedy Penalty Algorithm
  input  : Graph in Adjacency List Representation
  output : Minimal Dominating Set
  dominators ← The Dominating Set initialised to NULL;
  undominated ← The Undominated Set initially containing all the vertices in the input
    graph ;
  neighbours ← A mapping of all vertices to their undominated neighbours;
  two-neighbours ← A mapping of all vertices to their neighbours with the highest and
    2nd highest power;
  penalties ← update-penalties(two-neighbours);
  while undominated is not Empty do
      current-dominator ← get-vertex-with-max-penalty(penalties);
      dominators ← add-dominator(current-dominator, dominators);
      undominated ← update-undominated(neighbours[current-dominator],
        undominated);
      neighbours ← adjust-neighbours(current-dominator, neighbours);
      two-neighbours ← adjust-two-neighbours(neighbours, two-neighbours);
      penalties ← update-penalties(two-neighbours);
  end
  return dominators;
```

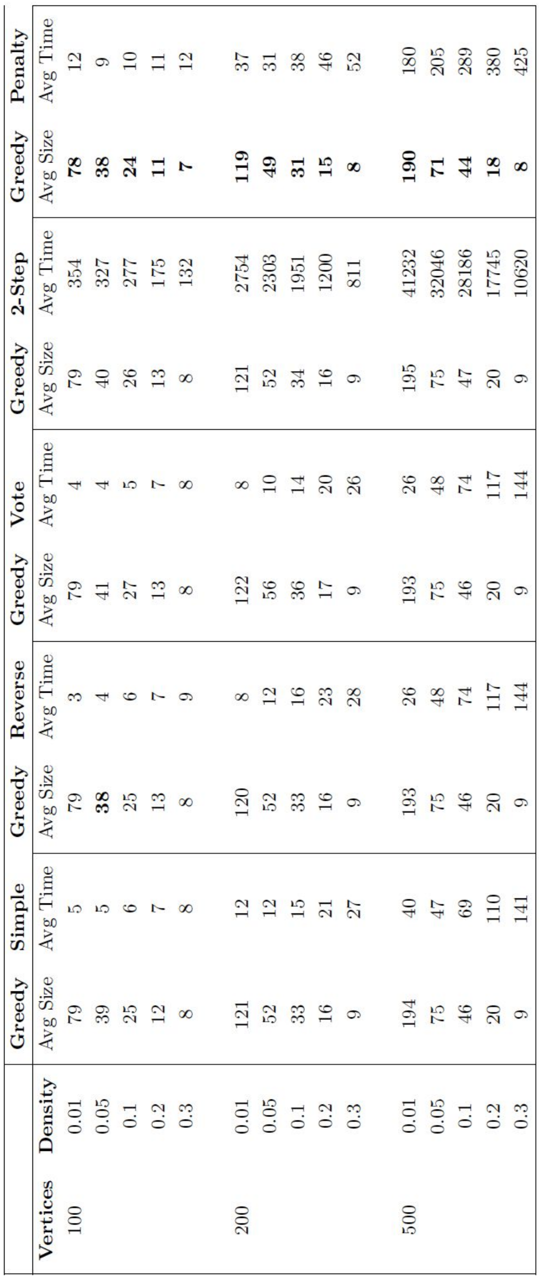
The analysis is for the average time complexity. The outer while loop runs D (Size of the Dominating Set) times. The internal complexity is O(N*d) where N is the number of vertices in the graph and d is the average degree of the vertices in the graph.

The final time complexity is O(N*d*D).

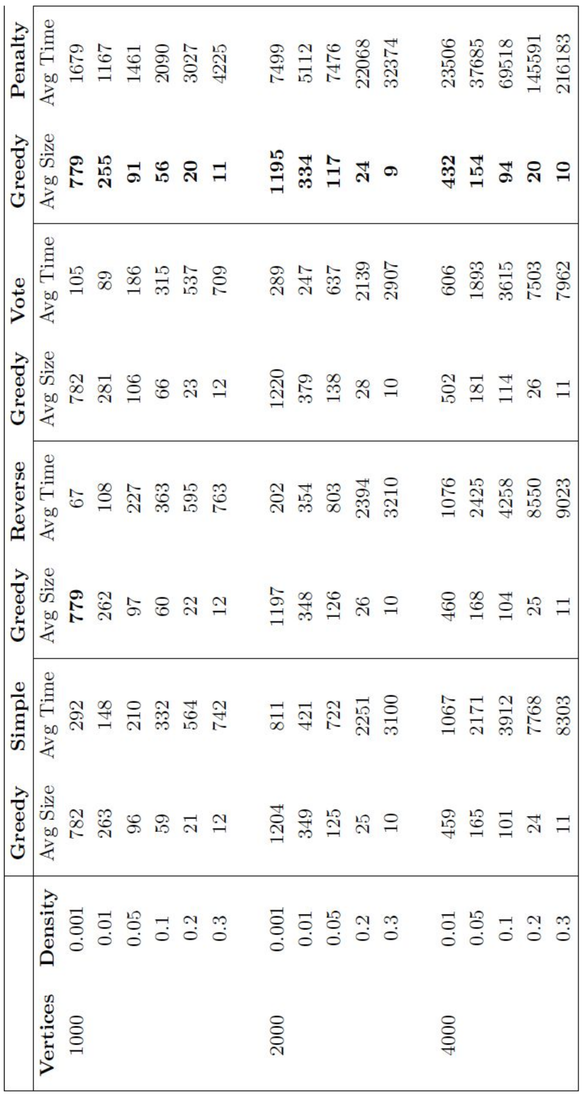
10.2 Results

The following are the results for the simulations of the penalty algorithm against all the other known ones. It resulted in the smallest sets in comparison to all the other algorithms.

The algorithm was tested on random graphs of varying vertices and densities.



		Greedy	**Simple**	**Greedy**	**Reverse**	**Greedy**	**Vote**	**Greedy**	**2-Step**	**Greedy**	**Penalty**
Vertices	**Density**	Avg Size	Avg Time	Avg Size	Avg Time	Avg Size	Avg Time	Avg Size	Avg Time	Avg Size	Avg Time
100	0.01	79	5	79	3	79	4	79	354	**78**	12
	0.05	39	5	**38**	4	41	4	40	327	**38**	9
	0.1	25	6	25	6	27	5	26	277	**24**	10
	0.2	12	7	13	7	13	7	13	175	**11**	11
	0.3	8	8	8	9	8	8	8	132	**7**	12
200	0.01	121	12	120	8	122	8	121	2754	**119**	37
	0.05	52	12	52	12	56	10	52	2303	**49**	31
	0.1	33	15	33	16	36	14	34	1951	**31**	38
	0.2	16	21	16	23	17	20	16	1200	**15**	46
	0.3	9	27	9	28	9	26	9	811	**8**	52
500	0.01	194	40	193	26	193	26	195	41232	**190**	180
	0.05	75	47	75	48	75	48	75	32046	**71**	205
	0.1	46	69	46	74	46	74	47	28186	**44**	289
	0.2	20	110	20	117	20	117	20	17745	**18**	380
	0.3	9	141	9	144	9	144	9	10620	**8**	425



		Greedy	**Simple**	**Greedy**	**Reverse**	**Greedy**	**Vote**	**Greedy**	**Penalty**
Vertices	**Density**	Avg Size	Avg Time	Avg Size	Avg Time	Avg Size	Avg Time	Avg Size	Avg Time
1000	0.001	782	292	**779**	67	782	105	**779**	1679
	0.01	263	148	262	108	281	89	**255**	1167
	0.05	96	210	97	227	106	186	**91**	1461
	0.1	59	332	60	363	66	315	**56**	2090
	0.2	21	564	22	595	23	537	**20**	3027
	0.3	12	742	12	763	12	709	**11**	4225
2000	0.001	1204	811	1197	202	1220	289	**1195**	7499
	0.01	349	421	348	354	379	247	**334**	5112
	0.05	125	722	126	803	138	637	**117**	7476
	0.2	25	2251	26	2394	28	2139	**24**	22068
	0.3	10	3100	10	3210	10	2907	**9**	32374
4000	0.01	459	1067	460	1076	502	606	**432**	23506
	0.05	165	2171	168	2425	181	1893	**154**	37685
	0.1	101	3912	104	4258	114	3615	**94**	69518
	0.2	24	7768	25	8550	26	7503	**20**	145591
	0.3	11	8303	11	9023	11	7962	**10**	216183

Figures 10.1 & 10.2: Simulation Results on Random Graphs

11. Conclusion

Our penalty algorithm has its demerits as it produced larger dominating sets in comparison to the Reverse Algorithm on particular scale free graphs. These cases should be studied to improve the algorithm.

The time complexity of this algorithm is higher than the other greedy algorithms to which we compared it. Research should be done in trying to modify the algorithm to reduce its time complexity but without compromising on the qualitative performance. Moreover, this algorithm should be extended to a distributed environment.

There are many heuristics and some work better in some instances. Research needs to be carried on a single algorithm which incorporates multiple heuristics into a single algorithm that performs better than all the existing algorithms without compromising too much on the execution time.

Research also needs to be carried in the direction of extending our algorithm to the construction of a Connected Dominating Sets by constructing an initial dominating set. Our definition of penalty can also be modified to approximate the Minimum Weighted Dominating Set. Other dominating set problems like Total Dominating Set, Independent Dominating Set and many others can be explored using our proposed algorithm.

We have extended our project to evaluate the existing work in connected dominating sets. This will not only help us to understand the limitations of the existing algorithms but also help us in bridging our penalty heuristic for dominating sets to connected dominating sets.

We have limited our analysis to greedy algorithms only as our greedy penalty algorithm is the primary algorithm we want to analyse and compare.

12. References

1. Chapter 1 of "Connected Dominating Set: Theory and Applications", Volume 77 by Ding-Zhu Du and Peng-Jun Wan.

2. "An exact algorithm for the Maximum Leaf Spanning Tree problem" by Henning Fernaua, Joachim Kneisb, Dieter Kratschc, Alexander Langerb, Mathieu Lied loffd, Daniel Raiblea, and Peter Rossmanith- June 2010.

3. Salim Bouamama, Christian Blum, and Abdellah Boukerram. A population-based iterated greedy algorithm for the minimum weight vertex cover problem. Applied Soft Computing, 12(6):1632–1639, June 2012.

4. S. Raja Balachandar and K.Kannan. A meta-heuristic algorithm for vertex covering problem based on gravity. 2009.

5. Olivier Goldschmidt, Dorit S. Hochbaum, and Gang Yu. A modified greedy heuristic for the set covering problem with improved worst case bound. Information Processing Letters, 48(6):305–310, dec 1993.

6. L. A. Sanchis. Experimental analysis of heuristic algorithms for the dominating set problem. Algorithmica, 33(1):3–18, may 2002.

7. Gang Hu. A proposed algorithm for minimum vertex cover problem and its testing. 10 2016.

8. Alina Campan, T.M. Truta, and M Beckerich. Fast dominating set algorithms for social networks. CEUR Workshop Proceedings, 1353:55–62, 01 2015.

9. Jagadish M and Sridhar Iyer. A method to construct counterexamples for greedy algorithms. In Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education - ITiCSE '12. ACM Press, 2012.

10. Tiago P. Peixoto. The graph-tool python library. figshare, 2014.

11. S. Guha and S. Khuller. Approximation algorithms for connected dominating sets. Algorithmica, 20(4):374–387, Apr 1998.

12. Weisstein, Eric W. "Connected Domination Number." From *MathWorld*--A Wolfram Web Resource. http://mathworld.wolfram.com/ConnectedDominationNumber.html

APPENDIX A

Python Code used for the experiment:

```python
def printGraph(G):
    nx.draw(G, with_labels=True, font_weight='bold')

def Diff(li1, li2):
      return (list(set(li1) - set(li2)))

def Union(lst1, lst2):
    final_list = list(set(lst1 + lst2))
    return final_list

def gray_vertices(G,C):
    dominating = []
    for u in C:
        dominating = Union(dominating,list(G[u]))
    return Diff(dominating,C)

def h(G,C):
    h = 0
    dominating = gray_vertices(G,C)
    dominating = np.union1d(dominating,np.asarray(C))
    for u in list(G.nodes):
        if u not in dominating:
            h = h+1
    return h

def p(G,C):
    H = G.subgraph(C)
    p = nx.number_connected_components(H)
    return p

def g(G,C):
    p_ = p(G,C)
    h_ = h(G,C)
    return p_+h_

def min_g(G,C):
    val = sys.maxsize
    x = None
    for u in list(G.nodes):
        val_ = g(G,C+[u])
        if(val_<val):
            val = val_
            x=u
    return x

def greedy_gk(G):
    C=[]
    while (True):
        exist = False
        for x in G.nodes:
            if g(G,C+[x])<g(G,C):
                exist = True
        if not exist:
            return C
        x = min_g(G,C)
        C = C + [x]

def guha_khuller(G):
    C = greedy_gk(G)
    while p(G,C)>1:
        gray = gray_vertices(G,C)
        components = nx.connected_components(G.subgraph(C))
        a= list(next(components))[0]
        b = list(next(components))[0]
        path = nx.shortest_path(G,source=a,target=b)
        counter = 0
        for x in path:
            if x in gray:
                C = Union(C,[x])
                counter +=1
                if counter == 2:
                    break
    return C

def q(G,C):
    G2 = nx.Graph()
    G2.add_nodes_from(G.nodes)
    G2.add_edges_from(G.edges(C))
    return nx.number_connected_components(G2)

def chen_liestman(G):
    C = []
    V = list(G.nodes)
    while q(G,C)>=2:
        val = sys.maxsize
        v = None
        for u in Diff(V,C):
            res = q(G,Union(C,[u]))
            if(res < val):
                val = res
                v = u
        C = Union(C,[v])
    return C

def marginal(G,f,A,X):
    return f(G,Union(A,X)) - f(G,A)
```

```
def wolsey_greedy(G,f):
    A =[]
    V = list(G.nodes)
    while f(G,A) < f(G,V):
        u = None
        val = 0-sys.maxsize #max negative val
        # considering c(x) to be constant for all vertices
        for x in Diff(V,A):
            res = marginal(G,f,A,[x])
            if res > val:
                    val = res
                    u = x
        A = Union(A,[u])
    return A

def f(G,A):
    return len(G) - q(G,A)

def findsubsets(s, n):
    return list(itertools.islice(itertools.combinations(s, n), 10))

def H(X,f,G,A):
    return -1*(marginal(G,f,A,X)/len(X))

#brute force approach
def chooses_subset(G,f,A,k):
    max_val = 0-sys.maxsize
    max_X = None
    V = list(G.nodes)
    for i in range(1,2*k):
        subsets = findsubsets(V,i)
        for subset in subsets:
            subset = list(subset)
            val = H(subset,f,G,A)
            if val > max_val:
                max_X = subset
                max_val = val
    return max_X

#Helper Function
def calculate(G,f,A,k,V):
    max_val = 0-sys.maxsize
    max_X = []
    subsets = findsubsets(V,k)
    for i in range(len(subsets)):
        subset = list(subsets[i])
        val = H(subset,f,G,A)
        if val > max_val:
            max_X = subset
            max_val = val
    return [max_val,max_X]

#random selection approach
def chooses_subset2(G,f,A,k,V):
    max_val = 0-sys.maxsize
    max_X = []
    min_index = 1
    max_index = 2*k-1
    while(min_index<max_index):
        index = int((min_index + max_index)/2)
        index1 = int((min_index + index)/2)
        index2 = int((max_index + index)/2)

        [val1,X1] = calculate(G,f,A,index1,V)
        [val2,X2] = calculate(G,f,A,index2,V)
        [val,X] = calculate(G,f,A,index,V)

        maxx = max(val1,max(val2,val))
        if maxx == val:
            max_X = X
            max_val = val
            break
        elif maxx == val1:
            max_X = X1
            max_val = val
            min_index = min_index
            max_index = index
        else:
            max_X = X2
            max_val = val
            min_index = index
            max_index = max_index
    return max_X

def dgpwwz(G,f,k):
    C=[]
#     set_trace()
    while f(G,C)>2:
        V = Diff(G.nodes,C)
        X = chooses_subset2(G,f,C,k,V)
        if len(X) == 0:
            break
        C = Union(C,X)
    return C

def f2(G,A):
    return p(G,A)+q(G,A)
```

```python
def get_random_graph(n,p):
    return nx.generators.random_graphs.connected_watts_strogatz_graph(n,5,p)
```

```python
#snippet to run different algorithms on random graphs
result = []
n = 200
while n<=1000:
    density = 0.1
    while density<=0.5:
        G = get_random_graph(n,density)
        start = time.time()
        res = dgpwwz(G,f2,5)
        end = time.time()
        size1 = len(res)
        time1 = end-start

        start = time.time()
        res = wolsey_greedy(G,f)
        end = time.time()
        size2 = len(res)
        time2 = end-start

        start = time.time()
        res = chen_liestman(G)
        end = time.time()
        size3 = len(res)
        time3 = end-start

        start = time.time()
        res = guha_khuller(G)
        end = time.time()
        size4 = len(res)
        time4 = end-start

        result.append([
                n,
                density,
                size1,
                time1,
                size2,
                time2,
                size3,
                time3,
                size4,
                time4,
            ]
        )
        print("Done for n ="+str(n)+" and density ="+str(density))
        density+=0.1
        break
    n+=50
```