Choosing the Efficient Algorithm for Vertex Cover Problem

Thesis submitted in partial fulfillment of the requirements for the award of degree of

Master of Engineering
in
Computer Science & Engineering

By
K.V.R.Kumar
(80732008)

Under the supervision of
Dr. Deepak Garg
Asst. Professor
CSED

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT
THAPAR UNIVERSITY
PATIALA – 147004

JUNE 2009

CERTIFICATE

I hereby certify that the work which is being presented in the thesis report entitled, **"Choosing the Efficient Algorithm for Vertex-cover Problem"**, submitted by me in partial fulfillment of the requirements for the award of degree of Master of Engineering in Computer Science and Engineering submitted in Computer Science and Engineering Department of Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of **Dr. Deepak Garg** and refers other researcher's works which are duly listed in the reference section.

The matter presented in this thesis has not been submitted for the award of any other degree of this or any other university.

S.K.V.R.Kumar.
(K.V.R.Kumar)

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

Deepak Garg
(Dr. Deepak Garg)
Asst. Professor
Computer Science and Engineering Department
Thapar University, Patiala.

Countersigned by:

Seema Bawa
11/06/2009
(Dr.(Mrs.) Seema Bawa)
Professor & Head
Computer Science & Engineering. Department
Thapar University,
Patiala.

R.K.Sharma. 25/6/09.
(Dr. R.K.Sharma)
Dean (Academic Affairs)
Thapar University,
Patiala.

ACKNOWLEDGEMENT

No volume of words is enough to express my gratitude towards my guide ***Dr.Deepak Garg,*** *Asst.Professor, Department of Computer Science & Engineering, Thapar University,Patiala, who has been very concerned and has aided for all the materials essential for the preparation of this thesis report. He has helped me to explore this vast topic in an organized manner and provided me with all the ideas on how to work towards a research-oriented venture.*

I am also thankful to ***Dr.(Mrs.) Seema Bawa,*** *Head of Department, Computer Science & Engineering Department and* ***Mrs.Inderveer channa,*** *P.G. Coordinator, for the motivation and inspiration that triggered me for the thesis work.*

I would also like to thank the staff members and my colleagues who were always there at the need of the hour and provided with all the help and facilities, which I required, for the completion of my thesis work.

Most importantly, I would like to thank my parents and the almightly for showing me the right direction out of the blue, to help me stay calm in the oddest of the times and keep moving even at times when there was no hope.

K V R Kumar
(80732008)

ii

ABSTRACT

The vertex cover (VC) problem belongs to the class of NP-complete graph theoretical problems, which plays a central role in theoretical computer science and it has a numerous real life applications. We are unlikely to find a polynomial-time algorithm for solving vertex-cover problem exactly. Vertex-cover exhibits a coverable–uncoverable phase transition. The relationship of vertex-cover with other NP-complete problems is thoroughly studied in this work. This thesis work also analyzes the various algorithms on minimum vertex cover for standard classes of random graphs. The performance of all algorithms is compared with the complexity and the output solution that of the branch-and-bound problem solver (BB), approximation algorithm, greedy algorithm, simple genetic algorithm (GA), primal-dual based algorithm (PDB) and the alom's algorithm. The results indicate that all algorithms give near optimal solutions. The performance differences of all algorithms on a graph are relatively small to obtain a vertex-cover. For undirected graphs, better performance is achieved by alom's algorithm and for weighted graphs, better performance is achieved by primal-dual based approach. Additionally, alom's algorithm is extended in order to give the all possible vertex-covers of a graph and the algorithm is implemented in c++.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

The vertex cover (VC) problem belongs to the class of NP-complete graph theoretical problems, which plays a central role in theoretical computer science and it has a numerous real life applications. We are unlikely to find a polynomial-time algorithm for solving vertex-cover problem exactly. Vertex-cover exhibits a coverable–uncoverable phase transition. The relationship of vertex-cover with other NP-complete problems is thoroughly studied in this work. This thesis work also analyzes the various algorithms on minimum vertex cover for standard classes of random graphs. The performance of all algorithms is compared with the complexity and the output solution that of the branch-and-bound problem solver (BB), approximation algorithm, greedy algorithm, simple genetic algorithm (GA), primal-dual based algorithm (PDB) and the alom's algorithm. The results indicate that all algorithms give near optimal solutions. The performance differences of all algorithms on a graph are relatively small to obtain a vertex-cover. For undirected graphs, better performance is achieved by alom's algorithm and for weighted graphs, better performance is achieved by primal-dual based approach. Additionally, alom's algorithm is extended in order to give the all possible vertex-covers of a graph and the algorithm is implemented in c++.

CHAPTER 1
INTRODUCTION

Vertex cover problem is a NP-complete problem. If a problem is NP-complete, we are unlikely to find polynomial-time algorithm for solving it exactly, but this does not imply that all hope is lost. There are two approaches to getting around NP-completeness. First if the actual inputs are small, an algorithm with exponential running time may be perfectly satisfactory. Second, it may still possible to find near optimal solutions in polynomial time. In practice near optimality is often good enough. An algorithm that returns near-optimal solutions is called an approximation algorithm.

In computer science, the **vertex cover problem** or **node cover problem** is one of Karp's 21 NP-complete problems. It is often used in complexity theory to prove NP-hardness of more complicated problems.

The classical minimum vertex-cover problem involves graph theory and finite combinatorics and is categorized under the class of NP-complete problems in terms of its computational complexity (Garey & Johnson, 1979) [3]. Minimum vertex cover has attracted researchers and practitioners because of the NP-completeness and because many difficult real-life problems can be formulated as instances of the minimum vertex cover. Examples of the areas where the minimum vertex-cover problem occurs in real world applications are communications, civil and electrical engineering, and bioinformatics. However, only few studies exist that analyzes the performance of evolutionary algorithms.

Definition: Consider a graph G= (V, E) where V and E are accordingly vertex and edges. A vertex cover of an undirected graph is a subset $V' \subseteq V$ such that if (u, v) is an edge of G, Then either $u \in V'$ or $v \in V'$ or both.

The size of a vertex cover is the number of vertices in it. The vertex cover problem is to find a vertex cover of minimum size in a given undirected graph. Such a vertex cover is called an optimal vertex cover. Coreman describes an approximation algorithm with O (E) time for vertex cover problem is given. This algorithm finds the approximate solution. An algorithm for “crown reductions for the Minimum

Weighted Vertex Cover problem" is given [1]. In this algorithm how crown decompositions and strong crown decompositions can be computed in polynomial time are described.

There are two versions of the minimum vertex cover problem: the decision version and the optimization one. In the decision version, the task is to verify for a given graph whether there exists a vertex cover of a specified size. On the other hand, in the optimization version of this problem, the task is to find a vertex cover of minimum size. To illustrate minimum vertex cover, consider the problem of placing guards (Weigt & Hartmann, 2000) in a museum where corridors in the museum correspond to edges and the task is to place a minimum number of guards so that there is at least one guard at the end of each corridor.

Minimum vertex cover is one of the Karp's 21 diverse combinatorial and graph theoretical problems (Karp, 1972), which were proved to be NP-complete. Minimum vertex cover is a special case of the set cover problem (Thomas H. Cormen & Stein, 2001) which takes as input an arbitrary collection of subsets $S = (S_1, S_2, .., S_n)$ of the universal set V, and the task is to find a smallest subset of subsets from S whose union is V.

The minimum vertex cover problem is also closely related to many other hard graph problems and so it interests the researchers in the field of design of optimization and approximation algorithms. For instance, the independent set problem (Karp, 1972; Garey & Johnson, 1979) is similar to the minimum vertex cover problem because a minimum vertex cover defines a maximum independent set and vice versa. Another interesting problem that is closely related to the minimum vertex cover is the edge cover which seeks the smallest set of edges such that each vertex is included in one of the edges.

Recently, the attention of physicists was drawn to the study of NP-complete problems like vertex cover and satisfiability. The reason is that, when studied on suitable random ensembles, these problems exhibit phase transitions in the solvability (Monasson, Zecchina, Kirkpatrick, Selman,& Troyansky, 1999; Weigt & Hartmann, 2000b; Hartmann & Rieger, 2004), which often coincide which peaks in the typical computational complexity or changes of the typical complexity from exponential to polynomial or vice versa.

Suppose that to improve the performance of a computer network, you want to collect statistics on packets being transmitted. Draw a graph where the vertices are computers/routers, and edges are communication links:

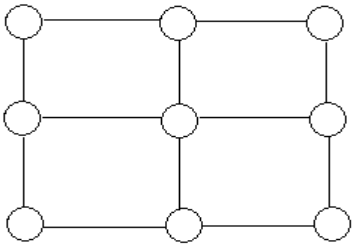


Figure 1.1: graph for computer network

Collecting statistics can slow down the network, and requires installing custom software etc. So, we want to monitor the traffic on as few nodes as possible. We choose as small a set of nodes as possible on which to install the monitoring software, so that each communication link has monitoring software on at least one end,

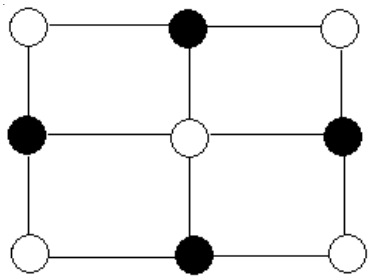


Figure 1.2: vertex-cover solution for computer network

If we install the monitoring software on the black nodes, every communication link has at least one end black. This is the vertex cover.

Example: In the figure1.3, {1, 3, 5, 6} is an example of a vertex cover of size 4. However, it is not a smallest vertex cover since there exist vertex covers of size 3, such as {2, 4, 5} and {1, 2, 4}.

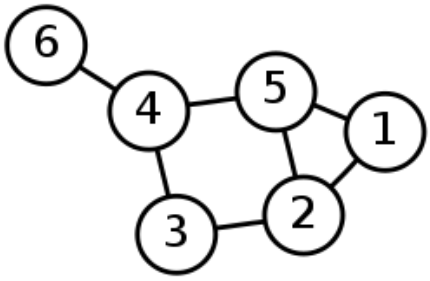


Figure 1.3: Shows different vertex vectors.

CHAPTER 2

VERTEX-COVER RELATIONS

2.1 Clique problem [1]

Def: A clique in an undirected graph G= (V, E) is a subset V' $\subseteq$ V of vertices, each pair of which is connected by an edge in E. In other words, a clique is a complete subgraph of G. The size of a clique is the number of vertices it contains. The Clique problem is the optimization problem of finding a clique of maximum size in a graph. As a decision problem, we ask simply whether a clique of a given size k exists in the graph.

Instance: a graph G= (V, E) and a positive integer k $\leq$ |V|.

Question: is there a clique V' $\subseteq$ V of size $\geq$ k?

2.2 Independent-set problem [1]

Def: An independent set of a graph G= (V, E) is a subset V' $\subseteq$ V of vertices such that each edge in E is incident on at most one vertex in V'. The independent set problem is to find a maximum-size independent set in G.

Instance: a graph G= (V, E) and a positive integer k $\leq$ |V|.

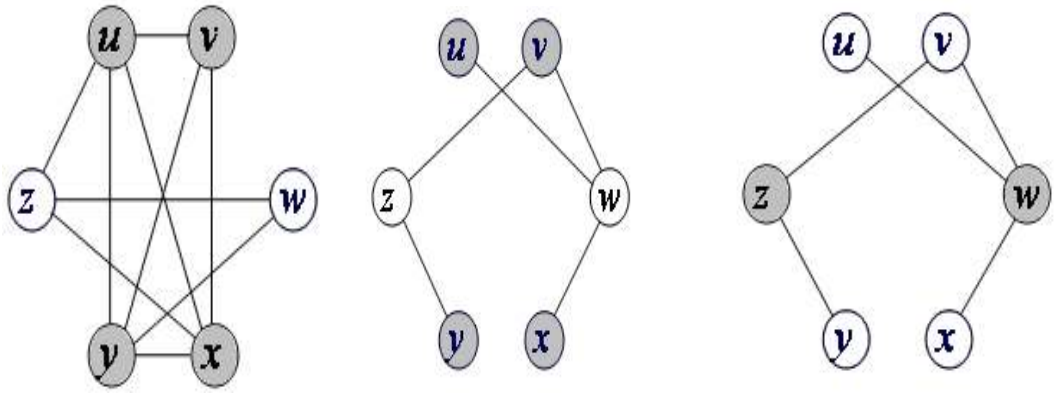
Question: is there an independent set of size $\geq$ k?

The following are equivalent for G= (V, E) and a subset V' of V and $\bar{G}$= (V,$\bar{E}$), where $\bar{E}$= {(u, v): u, v $\in$ V, u$\neq$ v, and (u, v) $\notin$ E}

(a). V' is a clique of G.

(b). V' is an independent of $\bar{G}$

(c). V—V' is a vertex-cover of $\bar{G}$



(a) (b) (c)

Figure 2.1: Relationship of VC with other NP-problems. (a) Clique V' = {u, v, x, y} (b) Independent set V'= {u, v, x, y} (c) Vertex-cover V—V'= {w, z}

2.3 NP-Completeness and reducibility

When analyzing the complexity of algorithms, it is often useful to recast the problem into a decision problem. By doing so, the problem can be thought of as a problem of verifying the membership of a given string in a language, rather than the problem of generating strings in a language. The complexity classes P and NP differ on whether a witness is given along with the string to be verified. P is the class of algorithms which terminate in an amount of time which is O (n) where n is the size of the input to the algorithm, while NP is the class of algorithms which will terminate in an amount of time which is O (n) if given a witness w which corresponds to the solution being verified.

NP-complete problems arise in diverse domains: Boolean logic, graphs, arithmetic, network design, sets and partitions, storage and retrieval, sequencing and scheduling, mathematical programming, games and puzzles and more. We shall use the reduction methodology to provide NP-completeness proofs for a variety of problems drawn from graph theory and set partitioning. There exist some problems which can be used to solve other problems, as long as a way of solving them exists, and a way of converting instances of other problems into instances of the problem with a known solution also exists. When talking about decision problems, a problem A is said to reduce to problem B if there exists an algorithm which takes as input an instance of problem A, and outputs an instance of problem B which is guaranteed to have the

same result as the instance in problem A, i.e. if L_A is the language of problem A, and L_B is the language of problem B, and if there is an algorithm which translates all $l \in L_A$ into $l_B \in L_B$ and which translates all $l' \notin L_A$ into $l'_B \notin L_B$ then problem A reduces to problem B.

Polynomial-time reductions provide a formal means for showing that one problem is at least as hard as another, to within a polynomial-time factor. That is, if $L_1 \leq_p L_2$ i.e a language L_1 is **polynomial-time reducible** to a language L_2, then L_1 is not more than a polynomial factor harder than L_2 , which is why the "less than or equal to" notion for reduction in mnemonic. We can now define the set of NP-complete languages, which are the hardest problems in NP.

For a particular type of computational problems, namely, optimization problems- where one looks for an optimal among all plausible solutions. Some optimization problems are known to be NP-hard, for example, finding a largest size independent set in a graph [Coo71, Kar72], or finding an assignment satisfying the maximum number of clauses in a given 3CNF formula (MAX3SAT)[1].

A proof that some optimization problem is NP-hard, serves as an indication that one should relax the specification. A natural manner by which to do so is to require only an approximate solution one that is not optimal, but is within a small factor $C > 1$ of optimal. Distinct optimization problems may differ significantly with regards to the optimal (closest to 1) factor C_{opt} to within which they can be efficiently approximated. Even optimization problems that are closely related may turn out to be quite distinct with respect to C_{opt}. Let the Maximum Independent Set be the problem of finding, in a given graph G, the largest set of vertices that induces no edges. Let the Minimum Vertex Cover be the problem of finding the complement of this set (i.e. the smallest set of vertices that touch all edges). Clearly, for every graph G, a solution to Minimum Vertex Cover is (the complement of) a solution to Maximum Independent Set. However, the approximation behavior of these two problems is very different, as for Minimum Vertex Cover the value of C_{opt} is at most 2.

One of these problems, and maybe the one that underscores the limitations of known technique for proving hardness of approximation, is Minimum Vertex Cover. Proving hardness for approximating Minimum Vertex Cover translates to obtaining a

reduction of the following form. Begin with some NP-complete language L, and translate 'yes' instances x ∈ L to graphs in which the largest independent set consists of a large fraction (up to half) of the vertices. 'No' instances x ∉ L translate to graphs in which the largest independent set is much smaller. Previous techniques resulted in graphs in which the ratio between the maximal independent set in the 'yes' and 'no' cases is very large.

There are the steps to show the problem is NP-complete

1. Show that the problem is in NP,
2. Reduce an NP-complete problem to it, and
3. Show that the reduction is a polynomial time function.

2.4 Structure of NP-completeness [1]

CIRCUIT-SAT is the first NP-complete problem. All proofs ultimately follow by reduction from the NP-completeness of CIRCUIT-SAT

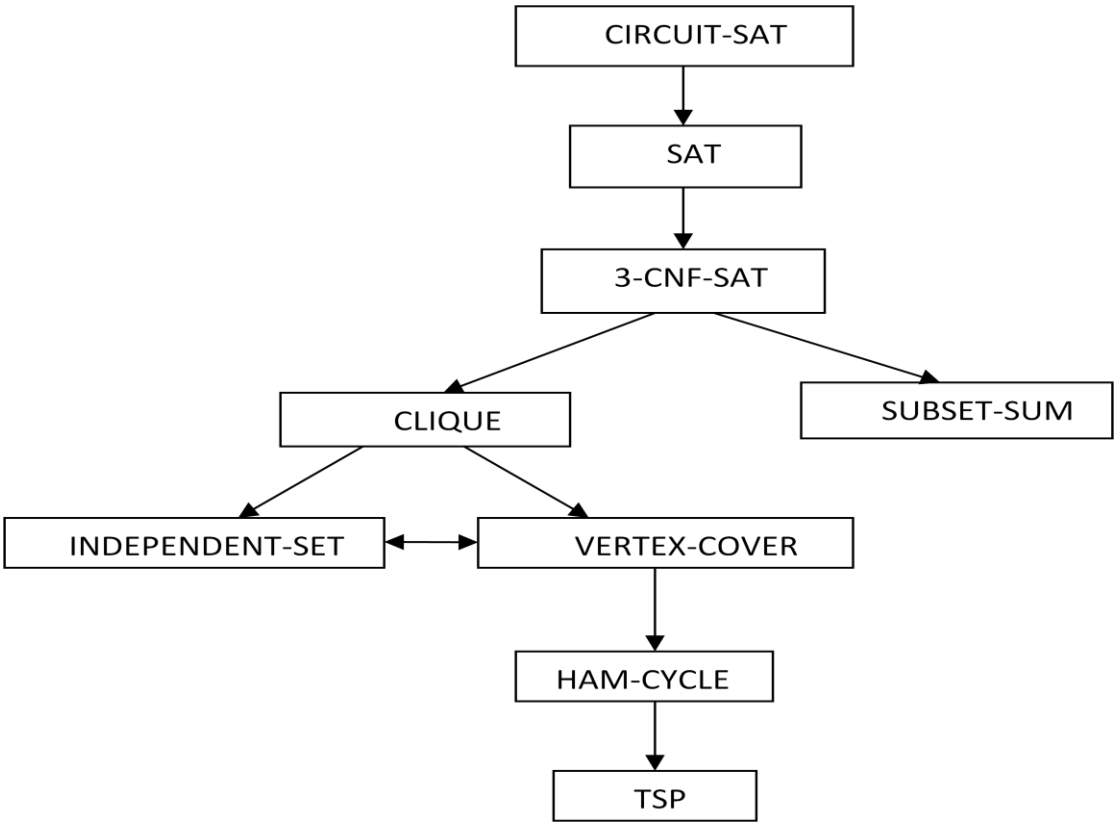


Figure 2.2: Structure of NP-completeness proofs

2.5 Theorem: VERTEX-COVER ∈ NPC [1]

Proof We prove VERTEX-COVER ∈ NP first. Let the certificate to be a set of vertices V' ⊆ V. The verification algorithm checks if the following are true: (1) |V'| = *k*. (2) For every edge (*u*, *v*) ∈ E, either *u* ∈V' or *v* ∈ V'. Obviously, this verification can be done in polynomial time.

Now, we prove VERTEX-COVER ∈ NP-hard by showing CLIQUE $\leq_p$ VERTEX-COVER. Let G (V, E) be the graph for the CLIQUE problem. We construct a new graph $\bar{G}$ for the VERTEX-COVER problem. The construction of G' is easy. It is the complement graph of G. That is $\bar{G}$ (V, $\overline{E}$).

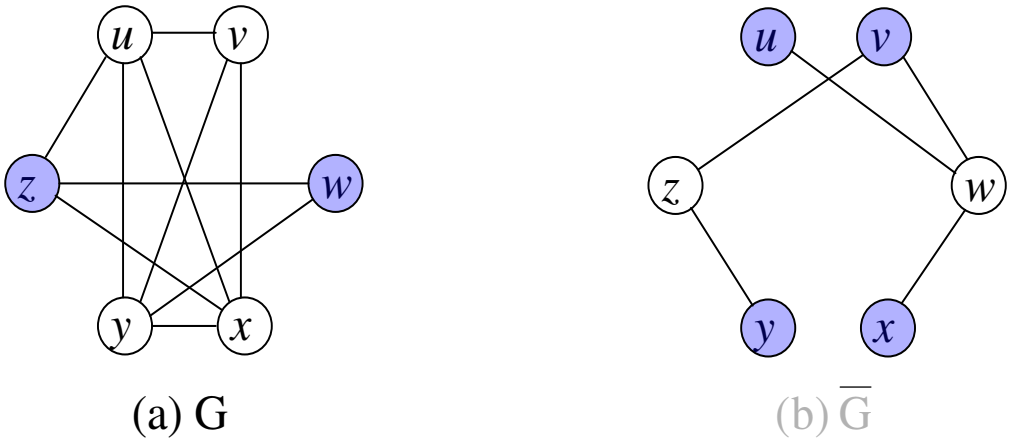


Figure 2.3: Reducing CLIQUE to VERTEX_COVER. (a) An undirected graph G= (V, E) with clique V'= {u, v, x, y}. (b) The graph $\bar{G}$ produced by the reduction algorithm that has vertex cover V– V' = {w, z}.

Let |V| = *n*, *k*' = *n* – *k*.

We shall show that G has a *k*-clique if and only if $\bar{G}$ has a vertex cover with size *k*'.

Suppose G has a *k*-clique V' ⊆ V. We claim that V – V' is a vertex-cover of $\overline{G}$. To see this, look at edge (*u*, *v*) ∈ E'. Obviously, (*u*, *v*) ∉ E. So, either *u* or *v* will not belong to V'. Then, *u* or *v* must belong to V – V'. So, V – V' is a vertex cover of $\bar{G}$ with size |V-V'| = *n* - *k* = *k*'.

Conversely, suppose $\bar{G}$ has a vertex-cover V' $\subseteq$ V, where $|V'| = n - k = k'$. Then, for any $u, v \in V$, if $(u, v) \in E'$, then $u \in V'$ or $v \in V'$ or both. This implies that if $u \notin V'$ and $v \notin V'$, then $(u, v) \notin E'$ or $(u, v) \in E$. Therefore, V – V' is a clique of G with size $|V\text{-}V'| = n - k' = k$.

Thus, we have just proved CLIQUE $\leq_p$ VERTEX-COVER.

So, VERTEX-COVER $\in$ NPC.

2.6 Vertex Cover Heuristics

Many traditional heuristics for vertex cover are known. Perhaps the most obvious is the greedy algorithm, in which the vertex of maximum remaining degree (incident on the maximum number of edges) is repeatedly removed from the graph and added to the cover until all edges are covered. While intuitive, the algorithm actually performs poorly on many classes of graphs and has no fixed performance bounds. An alternative algorithm is focused on finding a maximal matching in the graph: while edges remain, choose an arbitrary edge, add both endpoints to the cover, and remove both vertices from the graph. Because each selected edge must be covered by at least one of its endpoints, the algorithm has a fixed performance bound of 2, i.e., the resulting cover is at most twice the optimum cover. A number of other traditional approaches to VC, typically with performance bounds close to 2, have been reported and are discussed in [4]. A local-ratio approximation algorithm with best known bound of 2- log(log$n$2log$n$) relies on the repeated removal of sub-graphs, specifically small odd cycles or triangles.

Pramanick describes a novel stochastic optimization approach to VC as a "practical method for computing vertex covers for large graphs" [6]. Parallel Dynamic Interaction (PDI) is an inherently parallel optimization methodology that exploits the non-deterministic behaviour of shared-memory multiprocessors as the stochastic input to the algorithm. Individual processors search for covers in subsets of the complete graph. Global covers are formed by dynamic interaction between processors (reminiscent of a commodities trading process), in which the nondeterministic time of completion resolves competition between local solutions. Parallel dynamic interaction displayed very strong empirical performance on two problem classes (described later) when compared against traditional algorithms. Neural network and genetic algorithm

approaches to VC provide improved quality solutions on certain of the benchmark problems used in the PDI study, albeit with significantly increased computation times.

The importance of the encoding of the underlying problem is well-known for EA (Evalutionary Algorithms) optimization approaches. In the case of vertex cover and related problems, the most obvious approach is a direct encoding in which each bit of a binary chromosome of length |V| defines the presence or absence of the corresponding vertex in the cover. Most previously reported genetic algorithms for vertex-cover, independent set, maximum clique and related problems have used this direct encoding style. An obvious drawback of the approach is that the direct encoding allows infeasible solutions. The bit string of all zeros, for example, corresponds to an invalid candidate cover with no vertices.

More importantly, the encoding allows infeasible solutions to be created from existing feasible solutions using typical mutation and recombination operators found in EAs. Previously reported work has typically attacked the problem with penalty functions, in which the fitness of solutions that violate constraints is reduced, or with validation procedures, in which infeasible solutions are corrected to some "nearby" valid solution. Kommu compared two validation techniques with three different penalty methods [12]. After lengthy empirical investigation on the VC problem sets of [15], he concluded that the validation procedure based GAs performed somewhat better than the various penalty methods as well as providing significantly better solutions than traditional or PDI heuristics. Bäck and Khuri used a direct encoding with a graded penalty function in their GA for IS [2]. They tested their algorithm with randomly constructed graphs, as well as a class of scalable regular graphs. Aggarwal and co-authors use a direct encoding along with a domain-specific "optimized crossover" operator in their GA for IS [1]. Their crossover operation incorporates a local search (NP-Hard in general) along with a validation procedure to correct infeasible child solutions.

CHAPTER 3

ALGORITHMS FOR VERTEX-COVER PROBLEM

There are two types of algorithms: incomplete and complete ones. For complete algorithms, it is guaranteed to find the optimum or true solution. Hence the solution space is searched in principle completely. For incomplete algorithms, it is not ensured that the true solution or the global optimum is found. But they are very often sufficient for practical applications. This section outlines all existing algorithms to solve vertex-cover problem. That are (1) Approximation algorithm, (2) Branch-and-Bound algorithm (BB), (3) Greedy algorithm (4) Genetic algorithm (GA), (4) Primal-dual based approach (PDA) and (5) Alom's algorithm.

3.1 Concepts of Approximation algorithm [1]

Many problems of practical significance are NP-complete but are too important to abandon merely because obtaining an optimal solution is intractable. If a problem is NP-complete, we are unlikely to find a polynomial-time algorithm for solving it exactly, but even so, there may be a hope. There are at least three approaches to getting around NP-completeness. First, if the actual inputs are small, an algorithm with exponential running time may be perfectly satisfactory. Second, we may be able to isolate important special cases that are solvable in polynomial time. Third, it may still be possible to find near-optimal solutions in polynomial time (either in the worst case or on average). In practice, near-optimality is often good enough. An algorithm that returns near-optimal solutions is called an **approximation algorithm.**

3.1.1 Performance ratios for approximation algorithms

Suppose that we are working on an optimization problem in which each potential solution has a positive cost, and we wish to find a near-optimal solution. Depending on the problem, an optimal solution may be defined as one with maximum possible cost or one with minimum possible cost; that is the problem may be either maximization or a minimization problem.

We say that an algorithm for a problem has an **approximation ratio** of ρ(n) if, for any input of size n, the cost c of the solution produced by the algorithm is within a factor of ρ(n) of the cost c* of an optimal solution:

$$\text{Max}(c/c^*, c^*/c) \leq \rho(n)$$

We also call an algorithm that achieves an approximation ratio of ρ(n) a **ρ(n)-approximation algorithm.** The definitions of approximation ratio and of ρ (n)-approximation algorithm apply for both minimization and maximization problems.

3.1.2 Existing approximation algorithm of vertex cover problem

```
1       C ← Ø
2       E′ ← E [G]
3       while E′ ≠ Ø
4               do let (u, v) be an arbitrary edge of E′
5                       C ← C U {u, v}
6                       remove every edge in E′ incident on u or v
7       return C
```

3.1.3 Explanation of the approximate vertex cover algorithm

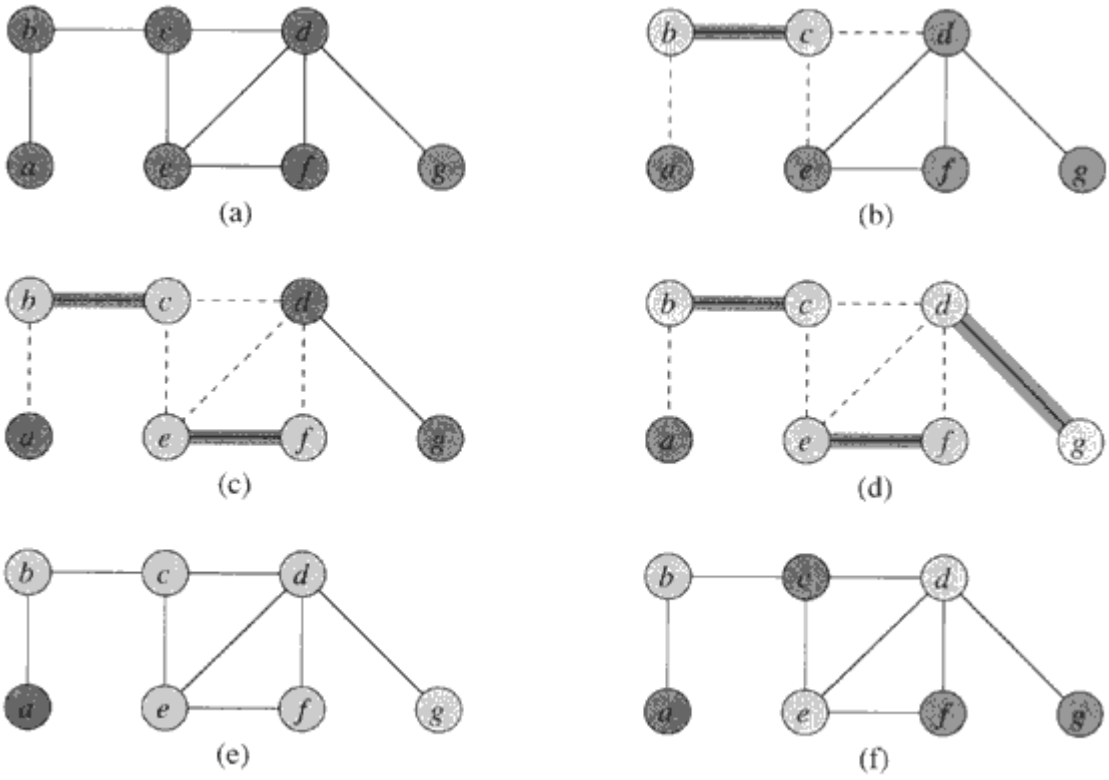


Figure 3.1: Explanation of the approximate vertex cover algorithm

The solution of above graph according to the approximate algorithm is {b, c, d, e, f, g}. This algorithm selects an arbitrary edge and removes the incident edges to it. This process continues until to cover all the vertexes. At first edge (b, c) is chosen from Fig. (b), then (b, a), (c, d) and (c, e) edges are discarded from the graph. Next arbitrary edge (e, f) is chosen and edge (e, d) is discarded. Lastly arbitrary edge (d, g) is chosen. After then no edge remains to be discarded. So the vertex cover set is {b, c, d, e, f, g}.

3.1.4 Complexity Analysis of the approximate vertex cover algorithm

Since the loop in algorithm 3, on lines (3-6) repeatedly picks an edge (u, v) from E′ adds its endpoints u and v to C, and deletes all edges in E′ that are covered by either u or v. The running time of this algorithm is O (E).

3.1.5 The APPROX_VERTEX_COVER has a bound ratio 2

Since this a minimization problem, we are interested in smallest possible c/c^*. Specifically we want to show $c/c^* \leq 2 = p(n)$. In other words, we want to show that APPROX-VERTEXCOVER algorithm returns a vertex-cover that is at most twice the size of an optimal cover.

Proof: Let the set c and c^* be the sets output by APPROX-VERTEX-COVER and OPTIMAL-VERTEX-COVER respectively. Also, let A be the set of selected edges in Fig. 1

by line 4. Because, we have added both vertices, we get $c = 2|A|$ but OPTIMAL-VERTEXCOVER would have added one of two. => $c/c^* \leq p(n) = 2$. Formally, since no two edge in A are covered by the same vertex from c^* (since, once an edge is picked in line 4, all other edges that are incident on its endpoints are deleted from E` in line 6) and we the lower bound: $|c^*| \geq A$---(1). On the size of an OPTIMAL-VERTEX-COVER. In line 4 of figure-1, we picked both end points yielding an upper bound on the size of Vertex-Cover. $|c| \leq 2|A|$ since, upper bound is an exact in this case, we have

$|c| = 2|A|$ ----------(2)

Take $|c|/2 = |A|$ and put it in equation (1)

$|c^*| \geq |c|/2$

$|c^*|/|c| \geq 1/2$

$|c^*|/|c| \leq 2 = p(n)$

3.2 Greedy Technique [1, 3]

This is a simple method used to solve optimization problems. The problems that are solved using the greedy method include finding the best order to execute a certain set of jobs on a computer, finding the shortest path in graph, etc.

To solve an optimization problem, we look for a set of candidates constituting a solution that optimizes (minimizes or maximizes, as the case may be) the value of the objective function. A greedy algorithm proceeds step by step. Initially the set of chosen candidates is empty. Then at each step, we try to add to this set the best remaining candidates, our choice being guided by selection function. The selection function is dependent on the problem at hand. For example, the selection function in the case of minimum weight spanning tree picks an edge of minimum weight from the remaining edges, an object with maximum profit per unit weight out of the remaining objects is chosen for putting in the knapsack in the case of knapsack problem. If the enlarged set of chosen candidates is no longer feasible, we remove the candidate we just added: the candidate we tried and removed is never consider again. However, if the enlarged set is still feasible, then the candidate we just added stays in the set of chosen candidates from then on. Each time we enlarge the set of chosen candidates, we check whether the set now constitutes a solution to the problem.

A popular method to construct successively space of solutions is greedy technique that is based on the evident principle of taking the (local) best choice at each stage of the algorithm in order to find the global optimum of some objective function.
A technique used in solving optimization problems [12]. Typically, we are given a set of n inputs and the goal is to find a subset (or some output) that satisfies some constraints. Any subset (or output) that satisfies these constraints is called a feasible solution. In an optimization problem, we need to find a feasible solution that maximizes or minimizes a given objective function. A feasible solution that does this is called an optimal solution. The greedy technique works in stages, considering one input at a time (typically in some clever order). At each stage, a decision is made depending on whether it is best at this stage. For example, a simple criterion can be whether adding the current input will lead to an infeasible solution or not. Thus, a locally optimal choice is made in the hope that it will lead to a globally optimal solution.

3.2.1 Basic steps to finding efficient greedy algorithms:

- Start by finding a dynamic programming style solution
- Prove that at each step of the recursion, the min/max can be satisfied by a "greedy choice" (greedy substructure)
- Show that only one recursive call needs to be made once the greedy choice is assumed. This is often natural when all the recursive calls are made by the min/max.
- Find the recursive solution using the greedy choice
- Convert to an iterative algorithm if possible

More generally, taking the direct approach:

- Show the problem is reduced to a sub problem via a greedy choice
- Prove there is an optimal solution containing the greedy choice
- Prove that combining the greedy choice with an optimal solution for the remaining sub problem yields an optimal solution

Example:

- For the MST problem: Prim's and Kruskal's algorithms
- For the SSSP problem: Dijkstra's algorithm

Remember, Dijkstra only works for graphs with no negative edge weights.

Usually heuristic algorithms are used for problems that cannot be easily solved.

3.2.2 Greedy Algorithms of vertex-cover problem

Algorithm1:

1. $C \leftarrow \emptyset$
2. **while** $E \neq \emptyset$
3. Pick any edge $e \in E$ and choose an end-point v of e
4. $C \leftarrow C \cup \{v\}$
5. $E \leftarrow E \setminus \{e \in E : v \in e\}$
6. **return** C

3.2.3 Clever greedy algorithm [8]

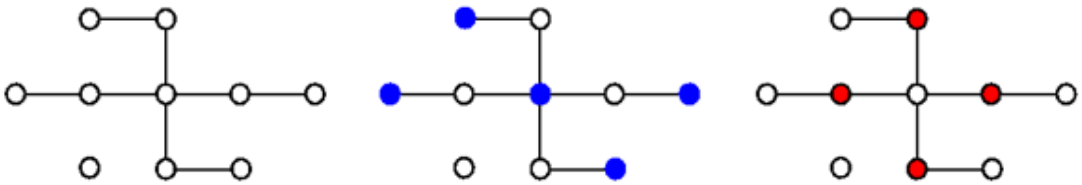
1. $C \leftarrow \emptyset$
2. **while** $E \neq \emptyset$
3. Pick a vertex $v \in V$ of maximum degree in the *current* graph
4. $C \leftarrow C \cup \{v\}$

5. $E \leftarrow E \setminus \{e \in E : v \in e\}$
6. **return** C

3.2.4 Greedy heuristic

Cover as many edges as possible (vertex with the maximum degree) at each stage and then delete the covered edges.

- The greedy heuristic cannot always find an optimal solution!
  - The vertex-cover problem is NP-complete.



(a) A graph instance (b) a vertex cover of size 5 obtained by the greedy algorithm (c) a vertex cover of size 4 optimal solution

Figure 3.2: shows vertex cover by greedy algorithm

3.3 Branch-and-Bound Algorithm [2, 6]

Branch-and-bound (BB) is a general algorithm for finding optimal solutions of various optimization problems, especially in discrete and combinatory optimization. It consists of a systematic enumeration of all candidate solutions, where large subsets of fruitless candidates are discarded, by using upper and lower estimated bounds of the quantity being optimized.

Branch-and-bound is a technique for exploring an implicit directed acyclic graph like the backtracking method. Optimal solutions to some problems like assignment of tasks to workers, etc. can be found using the technique of branch-and-bound. The branch-and-bound (BB) algorithm is a complete algorithm, meaning that it guarantees the exact solution even though the time complexity may increase exponentially with the graph size. As is also supported by the results presented in this paper, the algorithm is often outperformed by stochastic methods, which can often reliably locate the optimum after evaluating only a small portion of the search space.

The branch-and-bound algorithm recursively explores the full configuration space by deciding about the presence or absence of one node in the cover in each step of the recursion and recursively solving the problem for the remaining nodes. The full configuration space can be seen as a tree where each level decides about the presence or absence of one node and for each node there are two possible branches to follow; one corresponds to selecting the node for the cover whereas the other corresponds to ignoring the node. Technically, a covered node and all adjacent edges are removed, while an ignored node remains, but may not be selected in deeper levels of the recursion. The recursion explores the tree and backtracks when there are no more edges to cover or when the bounding condition is met, as described shortly. When backtracking, covered nodes are reinserted into the graph. Subsets of nodes that provide valid vertex covers are identified and the smallest of them is the minimum vertex cover. It is easy to see that in the worst case, the complexity of BB is upper-bounded by the total number of nodes in the recursion tree, which is proportional to 2^n.i.e o (2^n).

The branch-and-bound vertex-cover algorithm goes through different partial vertex covers by marking vertices as either covered or uncovered at each step. It backtracks if it reaches a state, where all the available covering marks xN have been used. The goal is to find a vertex-cover of size at most xN. The algorithm terminates, when it has found a valid vertex-cover or if it has gone through all possible configurations. The process of the algorithm is described by an configuration tree. In the configuration tree a node specifies the current state of all the vertices of the graph.

All vertices are marked as free at the start of the algorithm. The algorithm proceeds by marking a random free vertices as covered if the vertex has free or uncovered neighbors. The size of the largest allowed vertex-cover is xN, where $x \in [0,1]$ and $N=|V|$. If the xN covering marks are not all used, the algorithm can go on with the tree traversal, otherwise it has to backtrack. If the algorithm returns to the node by backtracking, then the vertex is uncovered and the other branch in the configuration tree is taken. If a node's all neighbours are covered, it is first marked as uncovered. A simple bound is used to prune the configuration tree: don't mark a vertex uncovered, if it has uncovered neighbours.

The bound applied in the following algorithm uses the current vertex degree d(i), which is the number of uncovered neighbours at a specific stage of the calculation. By covering a vertex i the total number of uncovered edges is reduced by exactly d(i). If several vertices v_1, v_2, . . , v_k are covered, the number of uncovered edges is at most reduced by $d(v_1) + d(v_2) + ... + d(v_k)$. Assume that at a certain stage within the backtracking tree, there are uncover E edges uncovered and still k vertices to cover. Then a lower bound M for the minimum number of uncovered edges in the subtree is given by $M= Max[0,E-max\ d(v_1)+d(v_2)+. . .+d(v_k)]$.

The algorithm can avoid branching into a subtree if *M* is strictly larger than the number *opt* of uncovered edges in the best solution found so far. For the order, the vertices are selected to be (un-)covered within the algorithm, the following heuristic is applied: the order of the vertices is given by their current degree. Thus, the first descent into the tree is equivalent to the greedy heuristic presented before. Later, it will become clear from the results that this heuristic is indeed not a bad strategy.

The following representation summarizes the algorithm for enumerating all configurations exhibiting a minimum number of uncovered edges. Let G= (V, E) be a graph, k the number of vertices to cover and uncov the number of edges to cover. Initially k= x and uncov= |E|. The variable opt is initialized with opt = |E| and contains the minimum number of uncovered edges found so far. The value of opt is passed via call by reference. At the beginning all vertices $i \in V$ are marked as free. The marks are considered to be passed via call by reference as well (not shown explicitly). Additionally, it is assumed that somewhere a set of (optimum) solutions can be stored.

3.3.1 Algorithm min-cover (G, k, uncov, opt) [7]

begin

if k = 0 then {leaf of tree reached?}

begin

if uncov < opt then {new minimum found?}

begin

opt := uncov;

clear set of stored configurations;

end;

store configuration;

end;

if bound condition is true (see text) then

return;

let i∈ V a vertex marked as free of maximal current degree;

mark i as covered;

k := k − 1;

adjust degrees of all neighbours j of i : d(j) := d(j) − 1;

min-cover(G, k, uncov − d(i), opt) {branch into 'left' subtree};

mark i as uncovered;

k := k + 1;

(re)adjust degrees of all neighbours j of i: d(j):= d(j) + 1;

min-cover (G, k, uncov, opt) {branch into 'right' subtree};

mark i as free;

end

In the actual implementation, the algorithm does not descend further into the tree as well, when no uncovered edges are left. In this case, the vertex covers of the corresponding subtree consist of the vertices covered so far and all possible selections of *k* vertices among all uncovered vertices.

Finally, we note that using the concepts of restarts one can also turn a complete backtracking algorithm into a (possibly) faster incomplete one.. The algorithm must be randomized, for applying restarts. Hence the choice which vertex is treated next is performed in some random way, similar to the generalized heuristic presented above. By applying many restarts, rare events become important: on one hand, the latter may have exponentially smaller search trees, i.e. in this case the algorithm by chance does not need to backtrack as long as usual. On the other hand, events of this type are exponentially rare. Balancing the exponential gain due to the smaller search tree against the exponential loss due to large number of restarts required to find such an event, an optimal backtracking (i.e. running) time per restart can be found.

3.4 Genetic Algorithm [4]

Genetic algorithm is an optimization technique based on the natural evolution. It maintains a population of strings, called chromosomes that encode candidate solutions to a problem. The algorithm selects some parent chromosomes from the population set according to their fitness value, which are calculated using fitness function. The fittest chromosomes have more chances of selection for genetic operations in next generation. Different types of genetic operators are applied to the selected parent chromosomes; possibly according to the probability of operator, and next generation population set is produced. In every generation, a new set of artificial creatures is created using bits and pieces of the fittest chromosomes of the old population.

Although GA is probabilistic, in most cases it produces better population compared to their parent population because selected parents are the fittest among the whole population set, and the worse chromosomes die off in successive generations. This procedure is continued until some user defined termination criteria are satisfied.
In the genetic algorithm (GA), use the same representation of candidate solutions and the same repair operator like local heuristic. GA starts by generating a random population of candidate solutions. At each iteration, a population of promising solutions is first selected. Variation operators are then applied to this selected population to produce new candidate solutions. Specifically, crossover is applied to exchange partial solutions between pairs of solutions and mutation is used to perturb the resulting solutions. Here uniform crossover and bit-flip mutation is used to produce new solutions. The new solutions are substituted into the original population using restricted tournament replacement (RTR) [18]. The run is terminated when the termination criteria are met.

GA applies variation operators inspired by natural evolution and genetics. The fitness function plays an important role in GA because it is used to decide how good a chromosome is. Fitness function is the number of vertices used to cover all the edges of the graph. $M=\sum_{i=1}^{v} Vi$ where $V_i=1$ if $V_i \in V_{cover}$ else 0

In HGA, one offspring is produced from two parent chromosomes. So, in that way best 50% chromosomes will directly go in the next generation using reproduction. All the chromosomes are used to create offspring using heuristic vertex crossover operator (HVX). Because we believe that each chromosome has some important

genes, which may become useful to obtain global optimal solution. Then mutation operator is applied to offspring. Mutation is used to avoid local minima and it should be applied on all the offspring.

3.4.1 How genetic algorithm works

A genetic algorithm is generally started with a randomly generated population of individuals. These individuals are potential solutions of the problem under study. Three genetic operators, namely, *selection, crossover* and *mutation* work on these individuals [20]. A selection method is used to select the individuals according to their values, the selected individuals reproduce the nest generation. The crossover operator recombines the individuals selected for reproduction with a pre-specified probability of crossover. The mutation operator induces changes in the chromosomes by complementing each bit of an individual with a pre-specified probability of mutation.

Let G = (V, E), where G is undirected graph, V is the set of vertices and E denotes the set of edges. VT and ET are vertex table and edge table respectively, v are vertices (genes) of chromosome; P1 and P2 are two parent chromosomes selected for crossover; in our case set of vertices for vertex cover. *V'* is the solution for vertex cover.

3.4.2 Algorithm HVX [10]

begin
V' = { }
create tables VT and ET
VT = (F(v), N(v)), where F(v) is the frequency of the vertex v in P1 and P2,
N(v) is the degree of vertex v in G, for ∀ v ∈P1 and ∀ v ∈ P2

ET = E (x, y) for ∀ E∈ G
while ET <> { } do
select v1 ∈ VT such that N(v1) > N(v) for ∀ v ∈VT. If more than
one vertex has same number of degree then select that vertex, whose
frequency (F(v1)) is high. If still more than one vertex is candidate
for selection then select any vertex randomly. Say v1
ET = ET — {E(x, y) : x = v1 or y = v1)}

$V' = V' — \{v1\}$

end while

return V'

end

3.4.3 Explanation of genetic algorithm

Graph – 1 (Fig.3.3) is used to show how HVX works. Initially chromosomes are generated randomly by selecting vertices one by one such that all the edges are covered. In the successive generations, two chromosomes are selected on the basis of fitness for the HVX. Suppose for Graph -1, two parent chromosomes P1, P2 are as shown below,

P1 = {1, 3, 4, 6} P2 = {1, 2, 3, 4}

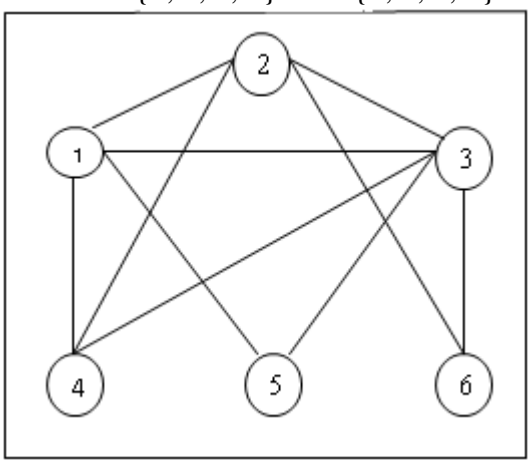


Figure 3.3: Graph-1 with 6 vertices and 9 edges

Now follow the procedure HVX as per below to produce offspring,

1. Create VT (Table 1) and ET (Table 2)

2. Select vertex with highest N (v) from VT. As shown in Table 3, for Run-1, it is vertex 3

3. Remove all the edges from ET, which are connected to vertex 3

4. Add 3 to V'

Repeat this procedure until ET becomes empty and N(v) = 0 for $\forall$ v $\in$VT

Table 3.1: Vertex Table (VT) for Graph-1

Vertex (v)	Frequency of Vertex in Parents F(v)	Number of Edges Connected to the vertex N(v)
1	2	4
2	1	4
3	2	5
4	2	3
5	0	2
6	1	2

Table3. 2: Edge Table (ET) for Graph-1

Vertex					
1	2	3	4	5	
2	1	3	4	6	
3	1	2	4	5	6
4	1	2	3		
5	1	3			
6	2	3			

Table 3.3: How HVX works is shown

Vertex (V)	F(v)	N(v)	Run 1 v1 = 3 V'={3} N_1(v1)	Run 2 v1 = 1 V'= {3,1} N_2(v1)	Run 3 v1 = 2 V' = {3,1,2} N_3(v1)
1	2	4	3	0	0
2	1	4	3	2	0
3	2	5	0	0	0
4	2	3	2	1	0
6	1	2	1	1	0

3.5 Linear Programming Formulation [13]

Linear Programming is one of the most widely used and general techniques for designing algorithms for NP-hard problems.

Def: A linear program is a collection of linear constraints on some real-valued variables with a linear objective function.

Every point in the feasible region is a feasible point, i.e., it is a pair of values that satisfy all the constraints of the program. The feasible region is going to be a polytope which is an n-dimensional volume all faces of which are at. The optimal solution is

either a minimum or a maximum, always occurs at a corner of the polytope (feasible region). The extreme points (corners) are also called basic solutions.

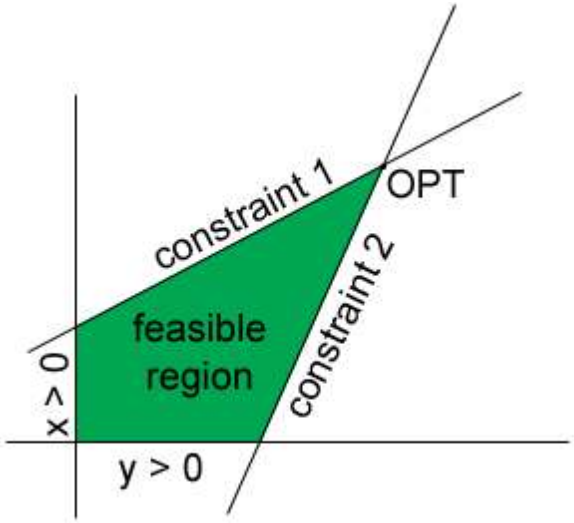


Figure 3.4: The feasible region in linear programming problem.

Linear programs can be solved in polynomial time. If you have n variables and m constraints, then the linear program can be solved in time polynomial in n and m.

One way to solve a linear program is to:

1. Enumerate all extreme points of the polytope.
2. Check the values of the objective at each of the extreme points.
3. Pick the best extreme point.

The good thing about Linear Programs is that they give us a lower bound to any NP-hard problem. If we take the NP-hard problem and reduce it to an Integer Program, relax the Integer Program to a Linear Program and then solve the Linear Program, the optimal solution to the Linear Program is a lower bound to the optimal solution for the Integer Program (and thus the NP-hard problem). This is a very general way to come up with lower bounds. Coming up with a good lower bound is the most important step to coming up with a good approximation algorithm for the problem. So this is a very important technique. Most of the work involved in this technique is coming up with a good reduction to Integer Programming and a good rounding technique to approximate the optimal fractional solution.

A linear program formulation (LP) for the vertex-cover problem can be written as follows.

Given: $G = (V, E)$ with weights $w: V \rightarrow \mathbb{R}^+$

Goal: Find the minimum cost subset of vertices such that every edge is incident on some vertex in that subset.

1. Reducing Vertex Cover to an Integer Program

Let the variables be:

x_v for $v \in V$ where $x_v = 1$ if $v \in V$ C and $x_v = 0$ otherwise.

Let the constraints be:

For all $(u, v) \in E$, $x_u + x_v \geq 1$ (each edge has at least one vertex)

For all $v \in V$, $x_v \in \{0, 1\}$ (each vertex is either in the vertx cover or not)

We want to minimize $\sum_v W_v . X_v$ (the total weight of the cover)

Note that all the constraints except the integrality constraints are linear and the objective function is also linear because the w_v are constants given to us. Thus this is an Integer Linear Program.

2. Relax the Integer Program to a Linear Program

Now relax the integrality constraint $x_v \in \{0, 1\}$ to $x_v \in [0, 1]$ to obtain a Linear Program.

3. Find the optimal fractional solution to the Linear Program

Say x^* is the optimal fractional solution to the Linear Program.

Here is a 2-approximation for the problem of weighted vertex cover. So for this problem:

Given: A graph G (V, E) with weight on vertex v as W_v.

Goal: To find a subset $V' \subseteq V$

Goal: To find a subset $V' \subseteq V$ such that each edge $e \in E$ has an end point in V' and $\sum_{v \in V}$, Wv is minimized.

The Linear Program relaxation for the vertex cover problem can be formulated as:

The variables for this LP will be x_v for each vertex v. So objective function is

$$\text{Min} \sum_v \text{Wv}$$

Subject to the constraints that $x_u + x_v \geq 1 \ \forall (u, v) \in E$

$$x_v \geq 0 \ \forall \in E$$

The Dual for this LP can be written with variables for each edge $e \in E$ as maximizing its objective function:

$$\text{Max} \sum_{e \in E} \text{Yv}$$

Subject to the constraints: $\sum_{v:(u,v) \in E} \text{Yuv}$

$$Y_e \geq 0 \ \forall \in E$$

These constraints in the linear program correspond to finding a matching in the graph G and so the objective function becomes finding a maximum matching in the graph. Hence, this is called the Matching LP.

3.6 Basic Primal-Dual Algorithm [14]

A primal-dual algorithm is an algorithm that starts with a feasible dual solution and an infeasible primal one. Throughout its execution such an algorithm improves the dual-objective function value of the kept dual solution and it reduces the degree of infeasibility of the primal one at the same time. The algorithm terminates as soon as the primal solution is feasible. The final dual solution is used as a lower-bound for the optimum solution value by means of weak duality.

1. Start with x = 0 (variables of primal LP) and y = 0 (variables of dual LP). The conditions that:

• y is feasible for Dual LP.

• Primal Complementary Slackness is satisfied.

are invariants and hence, hold for the algorithm. But the condition that:

• Dual Complementary Slackness is satisfied.

might not hold at the beginning of algorithm. x does not satisfy the primal LP as yet.

2. Raise some of the y_e's, either simultaneously or one-by-one.

3. Whenever a dual constraint becomes tight, freeze values of corresponding y's and raise value of corresponding x.

4. Repeat from Step 2 until all the constraints become tight.

Now let us consider the primal-dual algorithm for vertex cover.

3.6.1 Primal-Dual Algorithm for Vertex Cover

1. Start with x = 0 and y = 0.
2. Pick any edge e for which ye is not frozen yet.
3. Raise the value of y_e until some vertex constraint v goes tight.
4. Freeze all y_e's for edges incident on v. Raise x_v to 1.
5. Repeat until all y_e's are frozen.

3.6.2 Explanation of the primal-dual algorithm

This is an example of how this algorithm works on an instance of the vertex cover problem. We consider the following graph:

Example: Given below is a graph with weights assigned to vertices as shown in the figure and we start with assigning $y_e = 0$ for all edges $e \in E$.

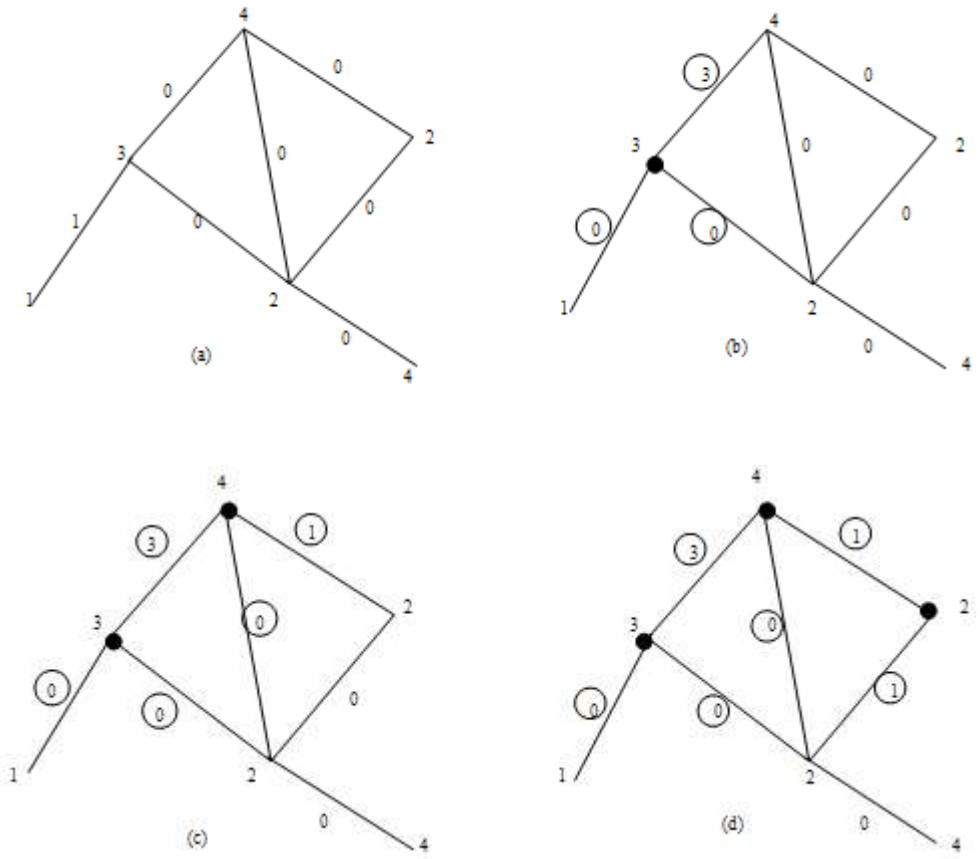


Figure 3.5: Explanation of primal-dual algorithm

So the algorithm proceeds as shown in the figure above. In steps (a)-(d), an edge is picked for which y_e is not frozen and the value of y_e is raised until the corresponding vertex constraint goes tight. All the edges incident on that vertex are then frozen and value of x_v is raised to 1.

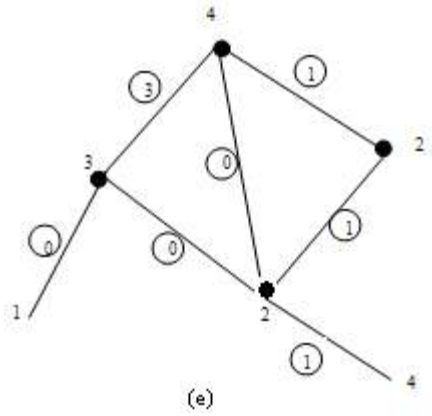


Figure 3.6: The Primal-dual solution

When all the y_e's getting frozen, the algorithm terminates. So the Value of Primal=11 (3+4+2+2) and the Value of Dual=6(3+1+1+1).

Hence ,

$$val_p(x) \leq 2val_d(y)$$

So, primal-dual algorithm is a 2-approximation.

3.6 Alom's algorithm for Vertex Cover Problem [15]

Monjurul alom presented a new algorithm for vertex cover problem that provides the efficient approximate solution that is better than existing approximate algorithm, greedy technique and genetic algorithm. This vertex cover algorithm selects the vertex which has maximum number of edges incident to it. All the edges are discarded incident to that vertex. If more than one vertex have same maximum number of edges, this algorithm select that vertex which have at least one edge that is not covered by other vertices, which has maximum edge. This process is repeated until to cover all the vertices of the graph. This algorithm takes same time as the existing approximate algorithm takes but it provides the solution that is always better than the approximate solution.

1. OPTIMAL_VT_COVER (E, V) {// E is an edge and V is an vertex
2. V′ ←ϕ;
3. E′ ←E [G]
4. While (E′ ≠ ϕ) {
5. M ← Choose vertex which has maximum incident edge;
6. If (More than one vertex have maximum number of edges) then

7. M ← Choose that node which has at least one edge that is not covered by others
Which have maximum number of edges.

8. V′ ← V′ U M;

9. Remove the all incident edges at vertex M;

10. Count incident edge of new graph.}

11. Return V′ }

3.6.1 Explanation of the alom's algorithm

Step1: Counting incident edges of all vertices in Figure 3.7, we see a=1, b=2, c=3, d=4, e=3, f=2, g=1.

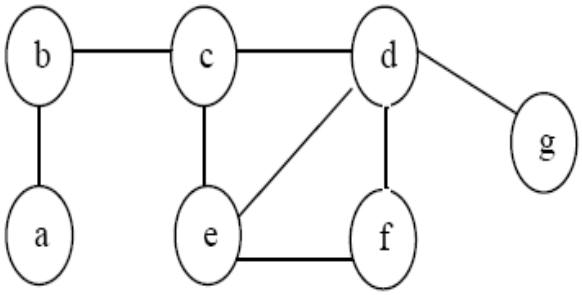


Figure 3.7: Example of undirected graph

Step2: We find d has the maximum edges it is 4. Now discard all the edges incident to d given in Figure-3.8

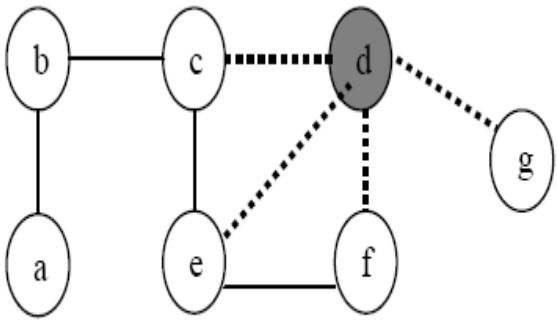


Figure 3.8: Discarding the edges incident to vertex d

Step 3: Again in Figure 3.8, the maximum edges are 2 that is in vertex b, c, and e. But c has two edges that are covered by b and e. Now b and e both have two edges but they have at least one edge that is not covered by other vertices c, which has

maximum edge, this algorithm select either b or e. Here we have chosen b. Remove all the edges incident to b we have only c, e and f exists in Figure 3.9.

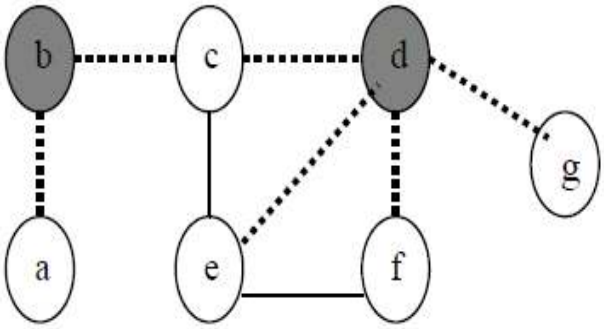


Figure 3.9: Discarding all the edges incident to vertex b

Step 4: Counting edges of c, e and f in Figure 3.8 which represent c=1, e=2, f=1. Now select e and the final graph is given in Figure 3.10.

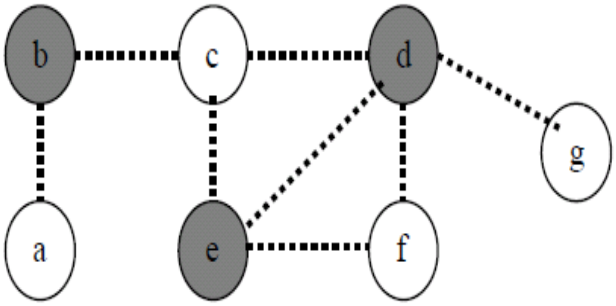


Figure 3.10: Graph representing the optimal vertex covering set

3.6.2 Complexity analysis of Alom's algorithm

Since the number of iterations of the loop is at most E. So time complexity of this Algorithm is O (E), where E is total no.of edges.

CHAPTER 4

PROBLEM STATEMENT

The problem is to find a vertex-cover of minimum size of in a given undirected graph. The problem is NP-Hard. It is not very difficult to find an approximation algorithm for the vertex-cover problem that returns a solution that is near to optimal. The size of the vertex-cover returned by the algorithm is guaranteed to be no more than twice the size of an optimal vertex-cover.

The minimum vertex cover problem is the optimization problem of finding a smallest vertex cover in a given graph.

INSTANCE: Given a graph G

OUTPUT: Smallest number k such that there is a vertex cover S for G of size k.

If the problem is stated as a decision problem, it is called the **vertex cover problem**:
Equivalently, the problem can be stated as a decision problem:

INSTANCE: Graph G and positive integer k.

QUESTION: Is there a vertex cover S for G of size at most k?

Using this strategy we have to find better algorithm which gives nearest solution to optimal.

All existing algorithms for vertex-cover problem will be studied and analyzed and a new improved algorithm will be designed and implemented.

CHAPTER 5
RESULTS AND DISCUSSIONS

5.1 Analysis

This section presents and discusses the analysis of all presented algorithms and complexity as shown below in the table. We illustrate the behavior of all studied algorithms such as approximation algorithm, greedy algorithm, genetic algorithm, alom's algorithm and primal-dual algorithm on the below graph.

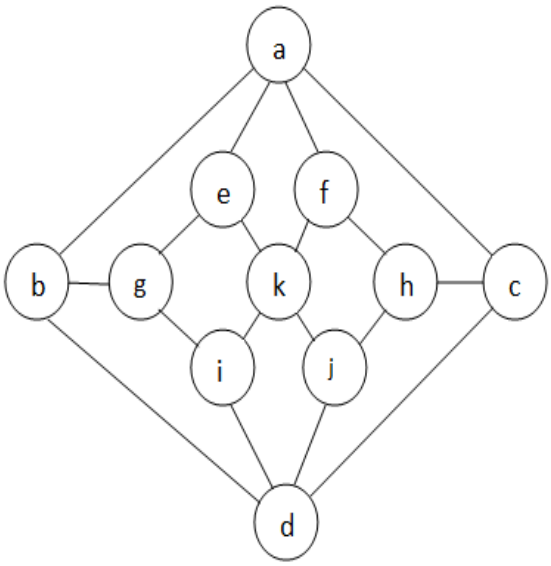


Figure 5.1: Graph (G) with 8 vertices

Table 5.1: Adjacency list of graph (G)

Vertex (v)	Number of edges connected to (v)	connected edges			
a	4	b	c	e	f
b	3	a	d	g	
c	3	a	d	h	
d	4	b	c	i	j
e	3	a	g	k	
f	3	a	h	k	
g	3	b	e	i	
h	3	c	f	j	
i	3	d	g	k	
j	3	d	h	k	
k	4	e	f	i	j

Table 5.2: Shows the comparison of all presented algorithms on graph (G)

Algorithm type	Size of vertex-cover	Solution set	Complexity	Remarks
Branch and bound	5	{a,d,g,h,k}	It grows exponentially fast with problem size for all values of c.	1. BB is complete algorithm that is ensured to find the minimum vertex cover. 2. If no vertex cover of the desired size is found, some covering marks have to be removed and be placed elsewhere, i.e. the algorithm has to backtrack.
Approximation	10 6 6	{a,b,c,d,e,g,h,i,j,k} {b,c,g,h,i,k} {b,c,e,f,i,j}	O(V+E)	1. This gives different solutions but all solutions near to optimal. 2. This is a polynomial-time 2-approximation algorithm means that the solution returned by algorithm is at most twice the size of an optimal.
Greedy Clever greedy	7 5	{a,b,c,g,h,i,k} {a,d,g,h,k}	O(V+E) O (logV)	1. It is easy to find in some situations where this algorithm fails to yield a optimal solution. 2. Greedy algorithm is not a 2-approximation 3. Clever greedy algorithm always gives solutions better than simple greedy.
Genetic	6	{a,c,d,g,h,k}	Time complexity measured by the overall number of candidate solutions examined until the optimum is found.	1. GA is an optimization technique based on the natural evolution. 2. GA fails to obtain consistent results for specific type of regular graphs. 3. For large problems, the growth of the number of evaluations required by GA becomes faster. 4. It gives better results when it is combined in to local optimization technique.

Primal-dual			O(V log V+E)	1. It reduces the degree of infeasibility of the primal one at the same time. 2. The algorithm terminates as soon as the primal solution is feasible. 3. The final dual solution is used as a lower-bound for the optimum solution value by means of weak duality.
Alom's	5	{a,d,g,h,k}	O(E)	1. It gives always optimal solution to the given graph. 2. Complexity is same as with approximation algorithm. 3. For larger graphs, may be this algorithm lost to give an optimal solution.

5.2 Alom's extended algorithm for Vertex Cover Problem

Alom's algorithm is extended in order to give the all possible vertex cover for undirected graph.

The reason behind that for some larger graphs this algorithm may be fails to give exact optimal solution. This algorithm gives all minimal vertex covers and minimum vertex covers.

This paper presents a formal description of the algorithm. Given a simple graph G with |V|= n vertices and |E|= m edges, this algorithm finds every possible minimal vertex cover. This is followed by a small example illustrating the steps of the algorithm.

OPTIMAL-VERTEX-COVER (V, E)

1. For i=1, 2, 3……n in turn
2. $G' = V - \{v_i\}$ and $E - \{e \in E: v_i \in e\}$
3. Apply the Algo1 on G'
4. $VC = V' \cup \{v_i\}$.
5. Return VC

Algo1(V' , E') {

1. $V' \leftarrow \emptyset$
2. $E' \leftarrow E[G']$
3. While $E' \neq \emptyset$ {
4. Count incident edges of all vertices of graph G'
5. Vm ← Choose a vertex which has maximum degree in the current graph;
6. If (More than one vertex have maximum number of edges) then
7. $V_m \leftarrow$ Choose that vertex which has at least one edge that is not covered by others which have maximum number of edges.

   Otherwise choose an arbitrary edge.
8. $V' = V' \cup V_m$.
9. Remove the all incident edges of vertex V_m }
10. Return V' }

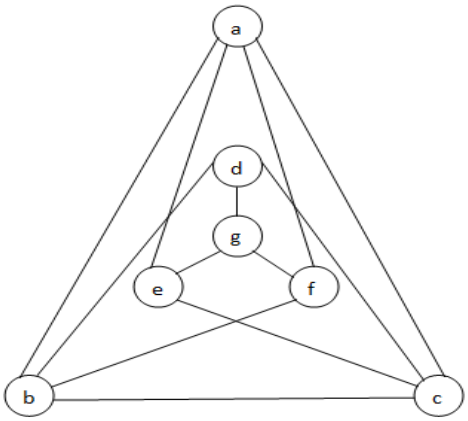


Figure 5.2: Example to show the steps of algorithm

5.3 Explanation of the Alom's extended algorithm

Step1: Remove the vertex a (v_1) and all incident edges of a (v_1) from G.

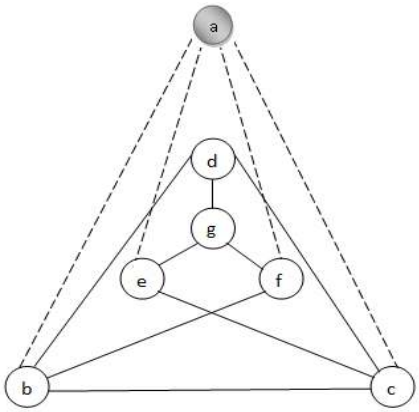


Figure 5.3: Discarding all edges incident to a

Step2: Perform the Algo1 on G' until all edges are removed.

Step3: Counting incident edges of all vertices in Figure-5.3 b=3, c=3, d=3, e=2, f=2, g=3.

Here the vertices b, c, d, g have maximum degree, But d has three edges that is covered by b, c and g. Now b, c and g have three edges but they have at least one edge that is not covered by other vertices, which has maximum edge, this algorithm select b, c or g. Here b has chosen as an arbitrary edge and remove all incident edges to b.

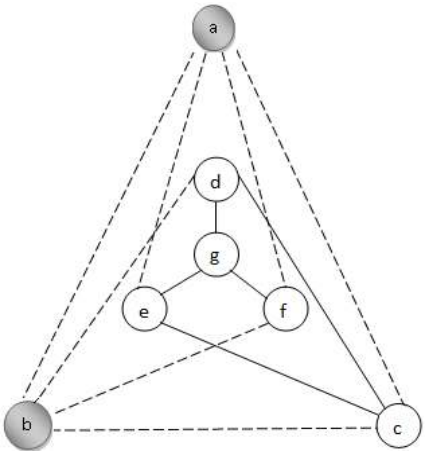


Figure 5.4: Discarding all edges incident to b

Step4: In fig5.4, c=2, d=2, e=2, f=2, g=3 choose g and discard all edges incident to g.

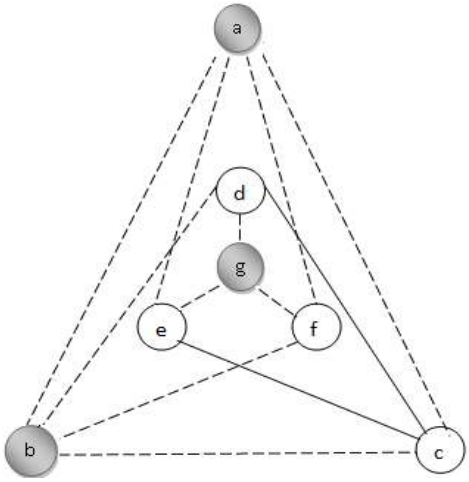


Figure 5.5: Discarding all edges incident to g

Step5: In fig5.5, c=2, d=1, e=1, f=0, now select c and remove all incident edges.

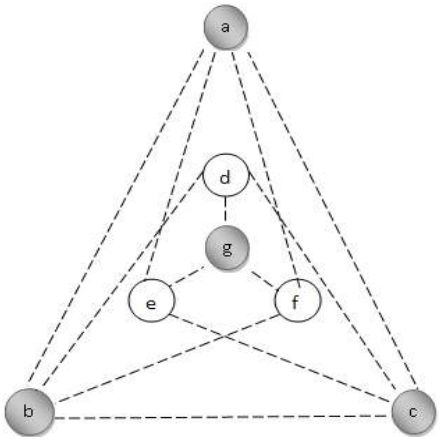


Figure 5.6: Graph representing the optimal vertex covering set

Step6: Return V'= {b, g, c}

Step7: Return VC= {a, b, c, g}

Repeat the same procedure for i=2, 3, 4,5,6,7. Then we obtain

Table 5.4: All possible vertex-covers

vertex	Vertex Name	Vertex cover
V_1	a	{a,b,c,g}
V_2	b	{ a,b,c,g }
V_3	c	{ a,b,c,g }
V_4	d	{ a,b,d,e,f}
V_5	e	{ a,b,d,e,f}
V_6	f	{a,c,d,e,f }
V_7	g	{ a,b,c,g }

In this table, vertices V1, V2, V_3, V_7 produces minimum vertex cover and V_4, V_5, V_6 produces minimal vertex cover.

5.4 Implementation

The efficient algorithm may be applied to any simple graph and will always terminate in polynomial-time. Minimal vertex covers of a certain size will be found by applying this algorithm. Specifically, we prove that every graph with n vertices and maximum vertex degree Δ must have a minimum vertex cover of size at most $n-\lceil n/(\Delta+1) \rceil$ and that the algorithm will always find a vertex cover of at most this size. Furthermore, it is proved that this condition is the best possible in terms of n and Δ by explicitly constructing graphs for which the size of a minimum vertex cover is exactly $n-\lceil n/(\Delta+1) \rceil$. Demonstrate the algorithm with a C++ program style.

Implementation code is given in 43-45 pages.

CHAPTER 6
CONCLUSION

This thesis analyzed performance of the branch-and-bound (BB) algorithm and several evolutionary algorithms on minimum vertex cover for standard classes of random graphs. In addition to branch-and-bound (BB), greedy algorithm, simple genetic algorithm (GA), primal dual algorithm (PDA) and alom's algorithm has been considered. In branch-and-bound technique, we calculated a bound on the possible value of any solution that might lie further on the graph. If the bound shows that any solution must necessarily be worse than the best solution found so far, then we need not go on exploring this part of the graph. The algorithm makes certain choices where to put covering marks. If no vertex-cover of the desired size is found, some covering marks have to be removed and be placed elsewhere, i.e. the algorithm has to backtrack. This is done in a symmetric way allowing investigation of the full configuration space.

Sometimes the approximation algorithm gives larger solution since the algorithm runs by choosing the arbitrary edge of the graph. This is a polynomial-time 2-approximation algorithm means that the solution returned by algorithm is at most twice the size of an optimal, since we do not know what the size of the optimal vertex cover is.

The greedy algorithm gives solutions better than approximation algorithm. The algorithm always makes the choice that looks best at the moment. Clever greedy algorithm always takes the vertex with the highest degree, add it to the cover set, remove it from the graph, and repeats. But the greedy heuristic cannot always find an optimal solution.

The genetic algorithm is slower than one local step of branch-and-bound. This is an optimization technique based on the natural evolution and this is fails to obtain consistent results for specific type of regular graphs. For large problems, the growth of the number of evaluations required by GA becomes faster. The heuristic vertex crossover (HVX) especially for minimum vertex cover problem, which works very well and converges fast to optimal solution. With the help of HVX and LOT (local

optimization technique), we can achieve optimal solution with less number of generation and population size.

The primal dual algorithm is only for weighted graphs and this is a 2-approximation algorithm. It reduces the degree of infeasibility of the primal one at the same time. The algorithm terminates as soon as the primal solution is feasible. The main goal of the algorithm is to find a vertex cover of minimum total cost. The final dual solution is used as a lower bound for the optimum solution value by means of weak duality.
In some cases, Greedy and genetic algorithms outperformed BB, which is not a surprising result because BB is a complete method that guarantees that the global optimum is found.

Finally, the alom's algorithm is the efficient algorithm for the vertex-cover problem because it gives optimal solutions in most cases. In this algorithm also, we have to choose an arbitary edge when the condition is coincide. So, for some larger graphs we may lose the exact optimal solution by alom's algorithm. That's way the alom's algorithm is extended in order to give the all possible solutions i.e all minimum vertex-covers and all minimal vertex-covers. From these all possible solutions we can easily choose the exact optimal solution which we want. But the complexity is more and the extended alom's algorithm is implemented.

ANNEXURE-I

REFERENCES

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. McGraw-Hill, New York, 2nd edition, 2001.

[2] Miroslav Chleb and Janka Chleb, "Crown reductions for the Minimum Weighted Vertex Cover problem" Electronic Colloquium on Computational Complexity, Report No. 101 (2004).

[3] M.R.Grarey, and D.S. Johnson, computers and intractability: A guide to the Theory of NP-Completeness, freeman, (1978).

[4] Dorit S.Hochbaum "Approximation Algorithms for NP-hard problems" (2002).

[5] X. Huang, J. Lai, and S. F. Jennings. Maximum common subgraph: Some upper bound and lower bound results. *BMC Bioinformatics*, 7(Suppl 4):S6, 2006, pp.80-94.

[6] M.Pelikan. Hierarchical Bayesian optimization algorithm: Toward a new generation of evolutionary algorithms. Spr inger-Verlag, 2005, pp.102-160.

[7] Alexandra k hartmann and martin weigt "statistical mechanics of the vertex-cover problem", j.phys. A; Math. Gen. 36(2003) 11069-11093.

[8] K.clarkson, "A modification to the greedy algorithm for the vertex cover problem", IPL, vol 16:23-25,(1983).

[9] R. Arakaki, and L. Lorena, "A Constructive Genetic Algorithm for the Maximal Covering Location Problem", in Proceedings of Metaheuristics International Conference, 2001, pp 13-17.

[10] Ketan Kotecha and Nilesh Gambhava "A hybrid genetic algorithm for Minimum Vertex-cover Problem", vol 2: pp 16-20.

[11] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. Science, 220:671–680, 1983.

[12] R. Luling and B. Monien. Load balancing for distributed branch and bound algorithms. In The 6th International Parallel Processing Symposium, pages 543–549, Los Alamitos, USA, 1992. IEEE Computer Society Press.

[13] R.Bar-Yehuda and S.even. A linear time approximation algorithm for the weighted vertex cover problem. *J. Algorithms*, vol 2:198-203,(1981).

[14] S.khuller, U.vishkin and N.young. A primal-dual parallel approximation technique applied to weighted set and vertex covers. *J. Algorithms*, 17(2):280–289, 1994.

[15] B.M.Monjurul Alom "An Efficient Approximation Algorithm to solve the Vertex cover Problem".

[16] P. Erdos and A. Renyi. On the evolution of random graphs. Publ. Math. Inst. Hung. Acad. Sci., 5:17, 1960.

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

[18] M. Weigt and A. K. Hartmann. The typical-case complexity of a vertex-covering algorithm on finite-connectivity random graphs. *Phys. Rev. Lett.*, 86:1658, 2001.

[19] K.Hartmann and M.Weigt.Phase Transitionsin Combinatorial Optimization Problems. Wiley-VCH, Weinheim, 2005, pp.28-150.

[20] Bollobas. Random Graphs. Cambridge University Press, Cambridge, UK, 2nd edition, 2001.

[21] K. Sastry. Evaluation-relaxation schemes for genetic and evolutionary algorithms. Master's thesis, University of Illinois at Urbana-Champaign, Department of General Engineering, Urbana, IL, 2001. Also IlliGAL Report No. 2002004.

[22] R. Bar-Yehuda and S. Even. A local-ratio theorem for approximating the weighted vertex cover problem. Annals of Discrete Mathematics, 25:27–45, 1985.

[23] M. Mezard, G. Parisi, and R. Zecchina. Analytic and algorithmic solution of random satisfiability problems. *Science*, 297:812, 2002.

[24] Pramanick I. and Kuhl, J. G. "A Practical Method for Computing Vertex Covers for Large Graphs," in *Proc. Intl. Symposium on Circuits and Systems* pp. 1859-1862, 1992.

[25] Weigt, M., & Hartmann, A. K. (2000a). Minimal vertex covers on finite-connectivity random graphs — A hard-sphere lattice-gas picture. Phys. Rev. E, 63, 056127.

Implementation code

```
//program to find minimal vertex covers for a given graph
#include<stdio.h>
#include<iostream.h>
#include<conio.h>
#include<process.h>
#define dim 10
   //enter dimension of edjacency matrix of graph
void vertex_cover(int,int*); //prototype for vertex_cover
int main()
{
int i,j,*p,arr[dim][dim];
clrscr();
cout<<"enter matrix\n";
      //enter input as graph adjacency matrix
for(i=0;i<dim;i++)
  {
  for(j=0;j<dim;j++)
  cin>>arr[i][j];
  cout<<"\n";
  }
p=&arr[0][0];
for(i=0;i<dim;i++)
vertex_cover(i,p);
getch();
return 0;
}//end of main
void vertex_cover(int v,int *q)
       //this function produces several different vertex covers
{
int a[dim][dim],count,max=0,flag,x=0,i,j;
int degree[dim],vertices_maxdegree[dim],vc[dim],l=0,r;
for(i=0;i<dim;i++)
  {
  for(j=0;j<dim;j++)
    {
    a[i][j]=*q;
    q++;
    }
  }
//remove the incident edges of correspanding vertex
for(j=0;j<dim;j++)
  {
  if(a[v][j]==1)
    {
    a[v][j]=0;
    a[j][v]=0;
    }
  }
//until all edges are removed
while(1)
{
int k=0,flag=0;
```

```
//
for(i=0;i<dim;i++)
  {
  count=0;
  for(j=0;j<dim;j++)
    {
    if(a[i][j]==1)
      {
      flag++;
      count++;
      }
    }
   degree[i]=count;
  }
//if all edges are removed then stop
if(flag==0)
break;
//find maximum degree
max=degree[0];
for(i=1;i<dim;i++)
  {
  if(max<degree[i])
  max=degree[i];
  }
//find vertices of maximum degree
for(i=0;i<dim;i++)
  {
  if(max==degree[i])
  vertices_maxdegree[k++]=i;
  }
/*Choose that vertex which has at least one edge that is not covered by others
  which have maximum number of edges.
   Otherwise choose an arbitary edge.*/

for(i=0;i<k;i++)
  {
  x=0;
  for(j=0;j<dim;j++)
    {
    if(a[vertices_maxdegree[i]][j]==1)
      {
      for(r=0;r<k;r++)
        {
        if(j==vertices_maxdegree[r])
        x++;
        }
      }
    }
  if(x<max)
  break;
  }
if(i==k)
i=0;
vc[l++]=vertices_maxdegree[i];
//remove the incident edges of selected vertex
```

```
for(j=0;j<dim;j++)
  {
  if(a[vertices_maxdegree[i]][j]==1)
   {
   a[vertices_maxdegree[i]][j]=0;
   a[j][vertices_maxdegree[i]]=0;
   }
  }
}//end of while
vc[l++]=v;
//it prints the vertex cover
cout<<"\n vertex cover of v="<<v+1;
cout<<" {";
for(i=0;i<l;i++)
  {
  cout<<" "<<vc[i]+1;
  }
cout<<"}";
}//end of vertex_cover
```

ANNEXURE-II

LIST OF PUBLICATIONS

[1] K V R Kumar, Deepak Garg, "Complete Algorithms on Minimum Vertex Cover" CIIT International Journal of Software Engineering and Technology, Issue May-2009 ISSN 0974 – 9748 & Online: ISSN 0974 – 9632.