POINT COVER PROBLEM IN 3D

SHARMIN AKTER
Bachelor of Science, Bangladesh University of Professionals, 2011

A Thesis
Submitted to the School of Graduate Studies
of the University of Lethbridge
in Partial Fulfillment of the
Requirements for the Degree

MASTER OF SCIENCE

Department of Mathematics and Computer Science
University of Lethbridge
LETHBRIDGE, ALBERTA, CANADA

POINT COVER PROBLEM IN 3D

SHARMIN AKTER

Date of Defence: August 21, 2017

Dedication

To my beloved parents and dearest husband.

Abstract

We examine the problem of covering points with minimum number of axis-parallel lines in three dimensional space which is an NP-complete problem. We introduce Lagrangian based algorithms to approximate the point cover problem. We study the Lift-and-Project relaxation of the standard IP to obtain lower bounds. This method is used to strengthen the integrality gap of a problem. Our experimental results show that the Lagrangian relaxation method gives very good lower bounds at reasonable computational cost. We present a hybrid method where the Lift-and-Project LP is solved using the Subgradient Optimisation technique. We propose an approximation algorithm which iteratively uses the Lagrangian relaxation procedure. We also study a Branch-and-Bound method which gives an optimal solution. We use a drop-in accelerator while conducting the simulations on large instances.

Acknowledgments

I would like to express my heartfelt thanks and gratitude to my thesis supervisor Dr. Daya Gaur for his persistent encouragement, continuous advice, guidelines, and inspiration throughout the journey of my MSc program. Without his invaluable insights, incredible supervision, and parent-like guidance this thesis would not see the light of day. Simply saying thanks to him will be inadequate for the lessons which I have learned from him regarding both academic area and real life.

I would sincerely like to thank my committee members Dr. Shahadat Hossain and Dr. Robert Benkoczi for their valuable time and suggestions for improving the quality of my thesis, and for their encouragement and cooperation throughout the journey.

Special thanks goes to the School of Graduate Studies for offering me the SGS Tuition Award. Besides this, I am deeply grateful to my supervisor for the financial assistance.

I would like to show my gratefulness to the Admin Assistant Barbara Hodgson, all the members of the Optimization Research Group of the University of Lethbridge, and my fellow students who cooperated in every possible way and made my journey memorable.

Last but not the least I would like to express my deep gratitude to my parents, my husband and my siblings for standing beside me during my academic and personal odds. The completion of this thesis would not be possible without their continuous support and unwavering encouragement. Especially, I am grateful to my dearest husband, who was my sole inspiration and was by my side through all the tough times. I am truly blessed to have him in my life.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Introduction

In this thesis, we will study the point cover problem where a set P of n points and a set L of axis parallel lines going through the n points are given as input. The goal is to find $C \subseteq L$ with minimum cardinality, such that every point $p \in P$ lies on some line $l \in C$. Here the axis parallel lines cover the points. The point cover problem was first studied by Hassin and Megiddo [13], and it is a special case of the hitting set problem. Some variants of the covering problems are: hitting a finite number of slopes with lines [13], the rectangle stabbing problem [10], and so on.

The problem of covering points by axis parallel lines in 2D can be solved in polynomial time. In this thesis, we study the point cover problem in three dimensional space, which is an NP-complete problem. One real life application of this problem is in medical science, particularly in radiotherapy [13]. Radiotherapy uses high energy X- rays, gamma rays, electron beams, or protons to destroy the affected cells. To apply these rays, radioactive needles are inserted into a certain area of the body. The problem of finding the minimum number of needles required to push into the body for applying these rays can be modeled as a point cover problem. The purpose of this thesis is to design good approximation algorithms for the point cover problem. We also explore some approaches to find good lower bounds in order to strengthen the integrality gap of the standard integer programming formulation.

1.2 Organization of the Thesis

In Chapter 2, we give the definitions used in this thesis. We include a description of the basic concepts as a linear program, the solution strategies for an LP, approximation algorithm design techniques, and some other methods used in this thesis. The Lagrangian relaxation method is the main technique used in the design of the proposed algorithms. We also give a general overview of the process of relaxing the standard integer program using the lift-and-project method in Chapter 2.

In Chapter 3, we formally define the point cover problem along with the integer program, the primal LP, and the dual LP. We show the transformation of the point cover problem into the set cover problem, and the vertex cover problem in hypergraphs. We also discuss the previous research work done in this area.

In Chapter 4, we propose Lagrangian based algorithms to approximate the point cover problem. We study the Lagrangian relaxation using the subgradient optimization technique, the lift-and-project method, and a hybrid approach to compute a lower bound for the instance of point cover problem. We also propose an iterative Lagrangian relaxation algorithm, and a branch-and-bound algorithm to obtain the integral solution of the point cover problem.

In Chapter 5, we describe the experimental setup and the data set, and discuss the results of experimentation. We give a comparative analysis of the results, and the computational time of the upper and lower bounds. We compare the running time of our implementation with and without GPU accelerators.

Finally, we conclude in Chapter 6 with some ideas on how to extend the work in this thesis.

Chapter 2

Preliminaries and Related Concepts

2.1 Introduction

In this chapter, we will discuss the basic concepts and techniques used in this thesis. We will describe the methods which have been used as solution strategies. We will also describe the problems that are related to the point cover problem.

2.2 Important Definitions

The following definitions are taken from the books by Luca Trevisan [28], Vijay V. Vazirani [30], Williamson and Shmoys [31], Thomas H. Cormen [5] and Anany Levitin [3].

Definition 2.1 (Approximation Algorithm). An α-approximation algorithm for an optimization problem is a polynomial time algorithm that gives a solution within α times the optimal value for every input instance. The value of α is also known as the approximation ratio.

Let $OPT(I)$ be the cost of the optimal solution to an instance of an optimization problem, and $ALG(I)$ is the cost of the solution returned by an algorithm ALG. Assume that the algorithm runs in polynomial time. We know that for a minimization problem: $ALG(I) \geq OPT(I)$, and for a maximization problem: $OPT(I) \geq ALG(I)$. For a minimization problem, the algorithm ALG is called an α-approximation algorithm if $\frac{ALG(I)}{OPT(I)} \geq \alpha$ for all I. For a maximization problem, the algorithm ALG is called an α-approximation algorithm if $\frac{OPT(I)}{ALG(I)} \geq \alpha$ for all I.

Definition 2.2 (**The Set Cover Problem**). Given a universal set of elements E and m subsets of those elements $X_1, X_2, \ldots, X_m$, the goal is to find a collection Y of those subsets such that every element of E belongs to some subset in Y and $|Y|$ is minimum possible.

A set cover instance is said to have frequency f if every element in the universal set E belongs to at most f subsets.

Definition 2.3 (**The Vertex Cover Problem**). Given an undirected graph $G = (V, E)$ where V is the set of vertices and E is the set of edges, a vertex cover VC is a subset of vertices such that for every edge $(u, v) \in E$ either u or v is an element of VC. The minimum vertex cover problem is to look for a vertex cover VC where VC contains the fewest vertices possible.

Definition 2.4 (**Matching**). Given an undirected graph $G = (V, E)$, a matching M is a set of edges that is $M \subseteq E$ such that no two edges of M share any vertices. A matching is maximum if its size is the largest of all matchings of G. A matching M is said to be maximal if M is not included in any other matching. A perfect matching is a matching where every vertex of the given graph is connected to exactly one edge of the matching. Perfect matchings are possible only on a graph with an even number of vertices.

Definition 2.5 (**Stack**). A stack is an ordered array of elements of a type. The insertion and deletion operation in a stack obey last-in-first-out or LIFO policy. A stack can be implemented by using an array. In a stack, insertion is done at the end of the array. This end point is called the TOP. The insertion operation and the deletion operation are named as $PUSH$ and POP respectively.

The stack operations are :

$a)$ $PUSH(S,a)$ - Insert an element a into the last index of the stack.

$b)$ $POP()$ - Delete an element from the last index of the stack.

$c)$ $TOP(S)$ - This function returns the position of the last element of the stack.

To check whether the stack is empty or not, check the value of $TOP(S)$. If the value is 0, the stack is empty. If an empty stack is popped, it is said that the stack underflows. If the value of $TOP(S)$ exceeds the maximum data size of the array, then this is an overflow.

Definition 2.6 (**Depth-first Search**)**.** Many graph algorithms require traversing vertices or edges of a graph in a systematic way. One of the simple graph search algorithms is depth-first search.

The algorithm starts at an arbitrary vertex of a graph and marks the vertex as a visited vertex. It then examines the incident edges in order to explore vertices which are not discovered yet. In each iteration, the algorithm tries to discover an unvisited node which is adjacent to the current node and traverses deeper until there is no longer any unexplored vertex. Then the algorithm ”backtracks” to the parent of the current node and traverses through the other branches of that parent node and continues further. This procedure continues until the search reaches a point from where no new vertices can be explored. If there are still some unvisited vertices in the graph, we repeat the steps starting at another unexplored node as a start vertex. Finally the algorithm halts when all the nodes are visited.

2.3 Linear Programming

In 1939, a Russian economist named Leonid Kantorovich [24] was the first to formulate a problem as a linear program.

A linear program is the problem of optimizing (minimizing or maximizing) a linear objective function satisfying a set of linear constraints. An objective function is a linear combination of some real-valued decision variables. A linear function has the following form

$$f = d_1 p_1 + d_2 p_2 + \ldots + d_n p_n \tag{2.1}$$

Here, p_1, p_2, ..., p_n are the decision variables and d_1, d_2, ..., d_n are the constant coefficients for those decision variables respectively.

A constraint is a linear inequality or equality (expressed as a linear combination of the decision variables) that must be satisfied by the decision variables.

$$l_1 p_1 + l_2 p_2 + \ldots + l_n p_n \quad RELOP \quad r \tag{2.2}$$

where *RELOP* $\in \{ \geq, =, \leq \}$

r and l_j where j = 1, 2, ..., n are real numbers.

$$p_j \geq 0, \quad j = 1, 2, \ldots, n \tag{2.3}$$

Equation (2.2) is an example of a technological constraint and Equation (2.3) is an example of a non negativity constraint.

For a maximization problem, the linear program seeks the largest possible value of the objective function subject to the constraints. We can write a maximization linear programming problem in canonical form [27] as follows:

$$\begin{aligned} \text{maximize} \quad & \sum_{j=1}^{n} d_j p_j & & \quad (2.4) \\ \text{subject to} \quad & \sum_{j=1}^{n} l_{ij} p_j \leq r_i & & \forall i \in \{1, 2, \ldots, m\} \\ & p_j \geq 0 & & \forall j \in \{1, 2, \ldots, n\} \end{aligned}$$

Here, n is the number of variables and m is the number of constraints.

We can also write a minimization linear programming problem in canonical form [27] as follows:

$$\begin{aligned} \text{minimize} \quad & \sum_{j=1}^{n} d_j p_j & & \quad (2.5) \\ \text{subject to} \quad & \sum_{j=1}^{n} l_{ij} p_j \geq r_i & & \forall i \in \{1, 2, \ldots, m\} \\ & p_j \geq 0 & & \forall j \in \{1, 2, \ldots, n\} \end{aligned}$$

Here, n is the number of decision variables and m is the number of constraints.

Let $OPT(I)$ be the value of the optimal integral solution of an optimization problem and $LP(I)$ be the value of the optimal LP solution, then

for a minimization problem, Integrality Gap = $\max_{I} \frac{OPT(I)}{LP(I)}$

for a maximization problem, Integrality Gap = $\max_{I} \frac{LP(I)}{OPT(I)}$

To represent a canonical LP into the standard form, we use vector and matrix notation as below.

$$\begin{aligned} \text{maximize} \quad & \boldsymbol{d}^{\mathsf{T}}\boldsymbol{p} \\ \text{subject to} \quad & L\boldsymbol{p} = \boldsymbol{r} \\ & \boldsymbol{p} \geq 0 \end{aligned} \tag{2.6}$$

$$\begin{aligned} \text{minimize} \quad & \boldsymbol{d}^{\mathsf{T}}\boldsymbol{p} \\ \text{subject to} \quad & L\boldsymbol{p} = \boldsymbol{r} \\ & \boldsymbol{p} \geq 0 \end{aligned} \tag{2.7}$$

Equations (2.6) and Equations (2.7) are the standard forms of the maximization and minimization problems respectively. Let n be the number of decision variables, and m be the number of constraints. In equations (2.6) and (2.7), $\boldsymbol{d}$ is a row vector of the constant coefficients of size n, and $\boldsymbol{p}$ is a column vector of the decision variables with size n. L is called the coefficient matrix which is of size m x n, and $\boldsymbol{r}$ is a column vector of size m. A solution is feasible if it satisfies all the constraints. A solution is infeasible if the assignment of the decision variables does not satisfy every constraint. The aim of a linear program is to seek a feasible solution that maximizes (for maximization problems) or minimizes (for minimization problems) the value of the objective function. An instance of an LP either in maximization or minimization canonical form can be transformed into an equivalent standard form of LP by adding slack or subtracting surplus variables respectively.

2.3.1 Solving LP using Geometric Interpretation

When the number of decision variables is two, we can plot the constraints graphically. Each decision variable is represented by one coordinate axis. We plot a constraint as an equality. We find a region which is called a feasible region using the equality. Consider the maximization problem below.

$$\begin{aligned} \text{maximize} \quad & f = 3p_1 + 2p_2 \\ \text{subject to} \quad & p_1 + p_2 \leq 40 \\ & 2p_1 + p_2 \leq 50 \\ & p_1 \leq 20 \\ & p_1 \geq 0, \;\; p_2 \geq 0 \end{aligned} \tag{2.8}$$

We plot the equalities p_1 + p_2 = 40, 2p_1 + p_2 = 50, p_1 = 20, p_1 = 0 and p_2 = 0 as shown in the following figure.

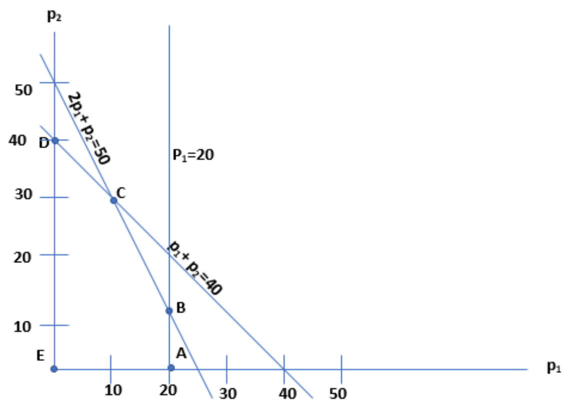

Figure 2.1: Solving LP using Graph

In Figure 2.1, ABCDE is the feasible region which means any point in this region satisfies all the constraints. There are infinitely many points which are the feasible points, but the corner points determine the optimal value for a maximization/minimization problem. The points A, B, C, D, E are the corner points. The value of the objective function for these corner points are given below:

Table 2.1: Solution to the maximization LP (2.8)

Corner Point	Coordinate (p_1, p_2)	Objective Function Value
A	(20, 0)	60
B	(20, 10)	80
C	(10, 30)	90
D	(0, 40)	80
E	(0, 0)	0

Here C is the point which gives the maximum objective function value of 90 for the LP (2.8).

2.3.2 Simplex Method

The simplex method is a popular procedure for solving linear programming problems. This method was invented by Dantzig in 1947 [6]. It is an iterative method. Here we illustrate the simplex method by demonstrating each step using the LP mentioned in (2.8).

The very first step is the conversion of a given canonical LP into an standard LP by introducing slack variables. For each inequality of (2.8) except for the non-negativity constraints, we introduce one slack variable. Let s_1, s_2 and s_3 be the slack variables for the three constraints. The standard LP for (2.8) is given below.

$$\begin{aligned} \text{maximize} \quad & f = 3p_1 + 2p_2 \\ \text{subject to} \quad & s_1 = 40 - p_1 - p_2 \\ & s_2 = 50 - 2p_1 - p_2 \\ & s_3 = 20 - p_1 \\ & p_1 \geq 0,\ \ p_2 \geq 0,\ \ s_1 \geq 0,\ \ s_2 \geq 0,\ \ s_3 \geq 0 \end{aligned} \tag{2.9}$$

The LP in the form of (2.9) is called a dictionary. The variables on the right hand side in the equality constraints are called the nonbasic variables. So the decision variables are the nonbasic variables initially. The basic variables are those on the left hand side of each equality constraint. The set of basic variables is also referred to as a basis. In other words, the objective function and the basic variables are expressed as a linear combination of the nonbasic variables in a dictionary. The basic feasible solution is obtained by setting the nonbasic variables to zero. At first, p_1 and p_2 are set to 0; s_1, s_2, s_3 are 40, 50, 20 respectively. The value of the objective function is 0.

In the next iteration, we create a new dictionary by bringing a nonbasic variable into the basis and for that reason, one of the the basic variables must leave the basis. We move a nonbasic variable into the basis if it increases (for maximization problems) or decreases (for minimization problems) the objective function value, subject to the condition that all variables are non-negative. Either p_1 or p_2 can enter into the basis. Let us bring p_2 into the basis as the entering variable. Next we determine how much the value of p_2 can be increased without violating the non-negativity constraints on the variables s_1, s_2 and s_3. Now we will look at the equations which contain p_2 and seek the best possible bound.

$s_1 = 40 - p_2$

As $s_1 \geq 0$, So $40 - p_2 \geq 0$ which implies $p_2 \leq 40$

$s_2 = 50 - p_2$ implies $p_2 \leq 50$

Here we choose s_1 as the leaving variable. p_2 will enter the basis and s_1 will leave the basis. Incorporating these changes into the basis gives a new dictionary. We perform elementary row operations to determine the dictionary corresponding to the new basis. We want to make p_2 a basic variable and s_1 a nonbasic variable. First we will change the equation involving p_2 and s_1 as follows

$$p_2 = 40 - p_1 - s_1 \tag{2.10}$$

Now we will rewrite the objective function and the rest of the two constraints using equation (2.10).

The new dictionary is given below.

$$\text{maximize } f = 80 + p_1 - 2s_1 \tag{2.11}$$

$$\text{subject to } p_2 = 40 - p_1 - s_1$$

$$s_2 = 10 - p_1 + s_1$$

$$s_3 = 20 - p_1$$

$$p_1 \geq 0,\ p_2 \geq 0,\ s_1 \geq 0,\ s_2 \geq 0,\ s_3 \geq 0$$

The solution is (p_1, p_2, s_1, s_2, s_3) = (0, 40, 0, 10, 20), and the value of the objective function is 80. This step is called a pivot.

In the next step p_1 will be the entering variable. We compute the variable which will leave the basis. We find that s_2 is the leaving variable. The updated dictionary is as follows

$$\begin{aligned} \text{maximize } & f = 90 - s_1 - s_2 \\ \text{subject to } & p_2 = 30 - 2s_1 - s_2 \\ & p_1 = 10 + s_1 - s_2 \\ & s_3 = 10 - s_1 + s_2 \\ p_1 \geq 0, \; & p_2 \geq 0, \; s_1 \geq 0, \; s_2 \geq 0, \; s_3 \geq 0 \end{aligned} \tag{2.12}$$

The solution to this dictionary is (p_1, p_2, s_1, s_2, s_3) = (10, 30, 0, 0, 10), and the value of the objective function is 90. In this dictionary there are no nonbasic variables with a positive coefficient, which indicates that this is the optimal solution for this problem.

Now we describe the simplex algorithm in general form. Let B be the set of basic variables and N be the set of nonbasic variables.

In each iteration where $p_j \in N$ and $d_j > 0$

1. Move a $p_j \rightarrow B$ and some $p_i \in B$ will leave the basis.
2. Compute which $p_i \in B$ will leave basis using the following formula
$$\min \frac{r_i}{l_{ij}} \quad \forall i \in B, \text{ where } l_{ij} \neq 0$$
3. Perform a pivot to update the dictionary.
4. If all new d_js are negative, then stop. Otherwise go to step 1.

2.3.3 The LP-duality

Given a linear program which is called the primal LP, we can obtain the dual of this linear program. If the primal LP is a minimization LP, then the dual LP will be for a maximization problem and vice versa.

In general we can write the dual of the maximization problem presented in (2.4) as below

$$
\begin{aligned}
\text{minimize} \quad & \sum_{i=1}^{m} r_i q_i & & \\
\text{subject to} \quad & \sum_{i=1}^{m} l_{ij} q_i \geq d_j & & \forall j \in \{1,2,....,n\} \\
& q_i \geq 0 & & \forall i \in \{1,2,....,m\}
\end{aligned}
\tag{2.13}
$$

The dual program gives a lower bound on the optimal solution of a primal LP for the minimization problems. For a maximization problem, the dual LP solution gives an upper bound on the optimal value of the primal problem. This relation is known as the weak duality theorem. If the primal and dual have feasible solutions and the objective function values are equal, then we have strong duality. These ideas are collectively known as the LP-duality theorem [29].

Theorem 2.7 (The Weak Duality Theorem [29]). *If (p_1, p_2,.....,p_n) is a feasible solution to the primal (2.4) and (q_1, q_2,.....,q_m) is a feasible solution to the dual (2.13), then*

$$\sum_{j=1}^{n} d_j p_j \;\leq\; \sum_{i=1}^{m} r_i q_i$$

Theorem 2.8 (The Strong Duality Theorem [29]). *If the primal problem has an optimal solution (p_1^*, p_2^*,......, p_n^*), then the dual also has an optimal solution (q_1^*, q_2^*,......, q_m^*) such that*

$$\sum_{j=1}^{n} d_j p_j^* \;=\; \sum_{i=1}^{m} r_i q_i^*$$

There is another theorem known as the complementary slackness theorem used to recover an optimal dual LP solution when an optimal primal LP solution is known. Let us first define the slack variables for the primal LP (2.4) and the dual LP (2.13).

$$s_i = r_i - \sum_{j=1}^{n} l_{ij} p_j \qquad \forall i \in \{1,2,....,m\}$$

$$v_j = d_j - \sum_{i=1}^{m} l_{ij} q_i \qquad \forall j \in \{1,2,....,n\}$$

Theorem 2.9 (The Complementary Slackness Theorem [29])**.** *Suppose that p = (p_1, p_2,.....,p_n) is primal feasible and that q = (q_1, q_2,.....,q_m) is dual feasible. Let (s_1, s_2,.....,s_m) denote the corresponding primal slack variables, and let (v_1, v_2,.....,v_n) denote the corresponding dual slack variables. Then p and q are optimal for their respective problems if and only if*

$$p_j v_j = 0 \qquad \forall j \in \{1,2,....,n\}$$

$$s_i q_i = 0 \qquad \forall i \in \{1,2,....,m\}$$

2.3.4 Other Methods for Solving LP

There exist some other methods to solve a linear program. Khachiyan [20] devised the ellipsoid method in 1979 that can solve linear programs in polynomial time. Another method which is a polynomial time algorithm, and is very efficient in practice for solving linear programs is the interior point method due to Karmarkar [19].

2.4 Integer Programming

A linear program with the constraint that one or more of the decision variables are restricted to take an integer value in the solution is called an integer program.

$$\begin{aligned} \text{maximize} \quad & \boldsymbol{d}^{\mathsf{T}}\boldsymbol{p} \\ \text{subject to} \quad & L\boldsymbol{p} \leq \boldsymbol{r} \\ & \boldsymbol{p} \in \mathbb{Z} \end{aligned} \tag{2.14}$$

$$\begin{aligned} \text{minimize} \quad & \boldsymbol{d}^{\mathsf{T}}\boldsymbol{p} \\ \text{subject to} \quad & L\boldsymbol{p} \geq \boldsymbol{r} \\ & \boldsymbol{p} \in \mathbb{Z} \end{aligned} \tag{2.15}$$

Equations (2.14) and Equations (2.15) are the canonical forms of the IP of the maximization problem and the minimization problem respectively.

2.5 Approximation Algorithm Design Techniques using LP

Two basic techniques [30] for designing approximation algorithms using linear programming are rounding, and the primal-dual schema.

2.5.1 Rounding

General steps to design an approximation algorithm using the rounding technique are given below.

1. Write an integer program for the problem.

2. Relax the integrality constraints and obtain a linear program relaxation from the integer program.

3. Solve the LP optimally in polynomial time.

4. Convert the fractional solution into an integer solution by rounding the fractional variables. Choose any rounding scheme that fits the problem and try to ensure that the rounding scheme does not increase the cost much.
5. Prove a bound on the approximation ratio.

2.5.2 The Primal-Dual Schema

The primal-dual schema is a sophisticated method for designing approximation algorithms which uses the dual solution of the LP-relaxation. In 1955 Kuhn [22] devised the Hungarian method to solve the assignment problem in polynomial time. Now the Hungarian method is known as the primal-dual method. The primal-dual algorithm relies on the complementary slackness theorem.

Generic steps in designing an approximation algorithm using the primal-dual method are given below.

1. Write the LP relaxation of the primal problem and obtain the corresponding dual.
2. Start with a primal infeasible solution $p = 0$ and a dual feasible solution $q = 0$.
3. While some constraint i in the primal LP is not satisfied

   a) Raise the corresponding dual variable q_i to the largest extent possible until some dual constraints become tight. Do this increase without violating the dual feasibility of q.

   b) Find out the dual constraints j which have become tight.

   c) Set the corresponding primal variable p_j to 1.

   d) Repeat steps 3a to 3c until a primal feasible solution is found.
4. Compare the primal solutions and the dual solutions to establish the performance ratio.
5. Prove a bound on the approximation ratio.

For a long list of approximation algorithms based on the primal dual schema, see the book by Vazirani [30].

2.6 The Lagrangian Relaxation Method

Held and Karp [14, 15] were the first to use the Lagrangian relaxation to obtain bounds for the traveling salesman problem. Later, Geoffrion [11] named this technique as the Lagrangian relaxation method.

It is a well-used and efficient approach to find the lower bounds for minimization problems. The steps for calculating a lower bound using the Lagrangian relaxation method are listed below.

1. Formulate the given problem as an integer program.

2. Multiply each hard constraint with a lagrange multiplier and absorb those constraints into the objective function.

3. Solve the relaxed integer program optimally.

2.6.1 Lagrangian Lower Bound Program (*LLBP*)

Let us consider an integer program P of a minimization problem. In the canonical form, we can write P using matrix representation.

$$\begin{aligned} \text{minimize} \quad & \boldsymbol{d}^{\mathsf{T}}\boldsymbol{p} \\ \text{subject to} \quad & L\boldsymbol{p} \geq \boldsymbol{r} \\ & \boldsymbol{p} \in \{0,1\}^n \end{aligned} \tag{2.16}$$

Here, $\boldsymbol{p}$ is the column vector of the decision variables which are nonnegative and binary. $\boldsymbol{d}$ is the row vector of the coefficients of the decision variables. L is the coefficient matrix of the constraints. $\boldsymbol{r}$ is the column vector which contains values for the right hand side of the constraints.

If we want to formulate the Lagrangian relaxation, we need to introduce a new vector which is called the Lagrange multiplier vector λ, where each entry in λ is nonnegative. After multiplying λ with $L\boldsymbol{p} \geq r$ and bringing them into the objective function, the resulting integer program becomes

$$\text{minimize} \quad \boldsymbol{d}^{\intercal}\boldsymbol{p} + \lambda(\boldsymbol{r} - \boldsymbol{L}\boldsymbol{p}) \tag{2.17}$$

$$\text{subject to} \quad \boldsymbol{p} \in \{0,1\}^n$$

The program above for a fixed λ is called the Lagrangian lower bound program (*LLBP*). We can also write the problem (2.16) in summation notation which is shown below.

$$\text{minimize} \quad \sum_{j=1}^{n} d_j p_j \tag{2.18}$$

$$\text{subject to} \quad \sum_{j=1}^{n} l_{ij} p_j \geq r_i \qquad \forall i \in \{1,2,....,m\}$$

$$p_j \in \{0,1\} \quad \forall j \in \{1,2,....,n\}$$

The *LLBP* of (2.18) is

$$\text{minimize} \quad \sum_{j=1}^{n} d_j p_j + \sum_{i=1}^{m} \lambda_i (r_i - \sum_{j=1}^{n} l_{ij} p_j) \tag{2.19}$$

$$\text{subject to} \quad p_j \in \{0,1\} \qquad \forall j \in \{1,2,....,n\}$$

Objective function in (2.19) can be rewritten as

$$\text{minimize} \quad \sum_{j=1}^{n} [d_j - \sum_{i=1}^{m} \lambda_i l_{ij}] p_j + \sum_{i=1}^{m} \lambda_i r_i \tag{2.20}$$

$$\text{subject to} \quad p_j \in \{0,1\} \qquad \forall j \in \{1,2,....,n\}$$

Let z_j = [d_j - $\sum_{i=1}^{m} \lambda_i l_{ij}$], j=1, 2, 3, ..., n. We solve the *LLBP* optimally by setting p_j=1 if $z_j \leq 0$, 0 otherwise.

2.6.2 The Subgradient Optimisation Method

One important consideration to produce a better lower bound is to find the next set of the Lagrange multipliers. The subgradient optimisation method is one of the approaches used for deciding values for the Lagrange multiplier vector.

The subgradient optimisation approach is an iterative method which iteratively adjusts the values of the Lagrange multipliers. The procedure starts with an initial set of multipliers and tries to improve the value of the lower bound by updating the set of multiplies in each iteration.

Below are the steps in the subgradient optimisation technique.

1. Take an initial vector λ for the Lagrange multipliers.
2. Choose the value of the parameter π satisfying $0 < \pi \leq 2$.
3. Initialize Z_{UB} with a feasible solution of the original problem.
4. Solve *LLBP* using the current set of multipliers λ.
5. Compute the subgradient g_i for each relaxed constraint i using the current solution of p_j.

$$g_i = r_i - \sum_{j=1}^{n} l_{ij} p_j$$

6. Define a scalar step size T which is dependent on the difference between the current lower bound Z_{LB} and the specified upper bound Z_{UB}. T also depends on π and the subgradient vector g as

$$T = \pi(Z_{UB} - Z_{LB}) / \sum_{i=1}^{m} (g_i)^2$$

7. Update λ_i by using the following formula.

$$\lambda_i = max(0, \lambda_i + T g_i)$$

8. Go to step (4) and again compute Z_{LB} using the updated Lagrange multiplier vector.

In each step, this procedure calculates a new Z_{LB} using the recent multiplier set, and keeps track of the best solution found so far. For minimization problems, this technique attempts to find the maximum lower bound on the optimal solution, and vice versa for maximization problems.

As this procedure is an iterative method, it requires a termination rule to stop. One idea is limiting the number of iterations. Another way is to reduce the value of π systematically, and stop when π is sufficiently small.

To illustrate the Lagrangian relaxation method and the subgradient optimisation technique, let us consider an example of a point cover problem.

$$\text{minimize} \quad l_1 + l_2 + l_3 + l_4 + l_5 \tag{2.21}$$

$$\begin{aligned} \text{subject to} \quad & l_1 + l_2 + l_3 \geq 1 \\ & l_2 + l_3 + l_5 \geq 1 \\ & l_1 + l_4 + l_5 \geq 1 \\ & l_j \in \{0,1\}, \quad j = 1,2,3,4,5 \end{aligned}$$

Let us choose the first three constraints for relaxation, so there will be three lagrange multipliers λ_1, λ_2, λ_3. If we multiply the three multipliers with three constraints, and bring them into the objective function, the *LLBP* becomes

$$\begin{aligned} \text{minimize} \quad & l_1 + l_2 + l_3 + l_4 + l_5 + \lambda_1(1 - l_1 - l_2 - l_3) + \\ & \lambda_2(1 - l_2 - l_3 - l_5) + \lambda_3(1 - l_1 - l_4 - l_5) \end{aligned} \tag{2.22}$$

$$\text{subject to} \quad l_j \in \{0,1\}, \quad j = 1,2,3,4,5$$

We can rewrite the *LLBP* as below.

$$\text{minimize} \quad (1-\lambda_1-\lambda_3)l_1+(1-\lambda_1-\lambda_2)l_2+(1-\lambda_1-\lambda_2)l_3+(1-\lambda_3)l_4+ \\ (1-\lambda_2-\lambda_3)l_5+\lambda_1+\lambda_2+\lambda_3 \tag{2.23}$$

$$\text{subject to} \quad l_j \in \{0,1\}, \quad j=1,2,3,4,5$$

Here,

z_1 = 1 - λ_1 - λ_3

z_2 = 1 - λ_1 - λ_2

z_3 = 1 - λ_1 - λ_2

z_4 = 1 - λ_3

z_5 = 1 - λ_2- λ_3

l_j will be 1 if $z_j \leq 0$, and l_j will be 0 if z_j >0.

Z_{LB} = $z_1 l_1$ + $z_2 l_2$ +$z_3 l_3$ +$z_4 l_4$ +$z_5 l_5$ + λ_1 + λ_2 + λ_3.

From Table 2.2, we can observe three iterations of the Lagrangian relaxation method with the subgradient optimisation technique where π = 2, Z_{UB} = 3, and initial (λ_1, λ_2, λ_3) = (0.3, 0.2, 0.8)

Table 2.2: Computation using the subgradient optimisation steps

z vector	l vector	Z_{LB}	Z_{MAX}	g vector	T	updated λ
(-0.1,0.5,0.5,0.2,0)	(1,0,0,0,1)	1.2	1.2	(0,0,-1)	3.6	(0.3,0.2,0)
(0.7,0.5,0.5,1.0,0.8)	(0,0,0,0,0)	0.5	1.2	(1,1,1)	1.667	(1.97,1.87,1.67)
(-2.6,-2.8,-2.8,-0.7,-2.5)	(1,1,1,1,1)	-6	1.2	(-2,-2,-2)	1.5	(0,0,0)

2.7 The Lift-and-Project Method

The maximum ratio between the optimal integer solution and the solution of a relaxed LP, over all the instances of the problem is known as the integrality gap. Lovasz and Schrijver [23] introduced one powerful system to strengthen the relaxation so that the integrality gap is reduced. Sherali and Adams [25] devised a different system. All of these relaxation methods are collectively known as relaxation hierarchies or the lift-and-project techniques.

Starting from an LP relaxation, the lift-and-project method tries to reduce the integrality gap using a hierarchy of relaxations. At each level the space and time required to solve the relaxation are increased. Solving the final relaxation may take exponential time, and the intermediate relaxations may take either polynomial or exponential time depending on the input size and the level.

We will describe the lift-and-project method to find a better relaxation of an integer program in general.

Consider the integer program of a minimization problem in matrix form.

$$\text{minimize} \quad \boldsymbol{d}^{\mathsf{T}}\boldsymbol{p} \tag{2.24}$$

$$\text{subject to} \quad L\boldsymbol{p} \geq \boldsymbol{r}$$

$$\boldsymbol{p} \in \{0,1\}^m$$

The first step of the lift-and-project method is to homogenize the inequalities. As the right hand side of the constraints are constants, so we introduce a supplementary variable p_0 in the right hand side of each constraint. The value of p_0 is always 1 in any solution of this LP. So the constraints become

$$L\boldsymbol{p} \geq \boldsymbol{r}.p_0 \tag{2.25}$$

$$\boldsymbol{p} \in \{0,1\}^m, \;\; p_0 = 1$$

We can write the modified constraints (2.25) using the matrix notation as follows

$$\begin{bmatrix} L & -\boldsymbol{r} \end{bmatrix} \begin{bmatrix} \boldsymbol{p} \\ p_0 \end{bmatrix} \geq 0 \tag{2.26}$$

$$\boldsymbol{p} \in \{0,1\}^m, \;\; p_0 = 1$$

$$L'\boldsymbol{p}' \geq 0 \tag{2.27}$$

$$\boldsymbol{p}' \in \{0,1\}^{m+1}, \;\; p_0 = 1$$

The next step is to formulate an equivalent quadratic program. The value of each p_i is either 0 or 1 (in any solution) which implies the following equation.

$$p_i(1-p_i) = 0, \quad \forall i \in \{1,2,\dots,m\} \tag{2.28}$$

$$\text{Or,} \quad p_i(p_0-p_i) = 0, \qquad \forall i \in \{1,2,\dots,m\}, \;\; p_0 = 1$$

The constant 1 is replaced by p_0 to maintain the homogeneous criteria. The next step is to multiply each constraint with p_i and (p_0-p_i), $\forall i \in \{1,2,\dots,m\}$.

$$(L'\boldsymbol{p}')p_i \geq 0, \quad \forall i \in \{1,2,\dots,m\} \tag{2.29}$$

$$(L'\boldsymbol{p}')(p_0-p_i) \geq 0, \qquad \forall i \in \{1,2,\dots,m\}, \;\; p_0 = 1$$

We can write the constraints (2.29) as a summation shown below,

$$\sum_{k=0}^{m} l_{jk}p_ip_k \geq 0, \qquad\qquad \forall i \in \{1,2,\dots,m\}, \;\; \forall j \in \{1,2,\dots,n\} \tag{2.30}$$

$$\sum_{k=0}^{m} l_{jk}(p_0p_k - p_ip_k) \geq 0, \quad \forall i \in \{1,2,\dots,m\}, \;\; \forall j \in \{1,2,\dots,n\}$$

Next, we linearize the quadratic program by replacing each $p_i p_j$ term with a new non-negative variable z_{ij}.

$$z_{ij} = p_i p_j, \quad \forall i \in \{1,2,\ldots,m\}, \ \forall j \in \{1,2,\ldots,n\}$$

The linearized version of the integrality constraint $p_i(p_0 - p_i) = 0, \forall i \in \{1,2,\ldots,m\}$ is

$$z_{0i} = z_{ii}, \quad \forall i \in \{1,2,\ldots,m\}$$

The lifted version of the LP is

$$\begin{aligned}
\text{minimize} \quad & \boldsymbol{d}^{\mathsf{T}}\boldsymbol{p} \\
\text{subject to} \quad & p_i = z_{0i} = z_{ii}, \quad \forall i \in \{1,2,.....,m\} \\
& z_{ij} = z_{ji}, \quad \forall i \in \{0,1,2,.....,m\}, \ \forall j \in \{0,1,2,.....,n\} \\
& \sum_{k=0}^{m} l_{jk} z_{ik} \geq 0 \quad \forall i \in \{1,2,.....,m\}, \ \forall j \in \{1,2,.....,n\} \\
& \sum_{k=0}^{m} l_{jk}(z_{0k} - z_{ik} \geq 0 \quad \forall i \in \{1,2,.....,m\}, \ \forall j \in \{1,2,.....,n\} \\
& p_0 = 1
\end{aligned} \tag{2.31}$$

The lifted LP is solved and only the solution of the variables p_i are used to compute the result of the original LP. This is the level-1 of the lift-and-project hierarchy. If we want to implement some more levels of these hierarchies, we need to project the constraints back onto the original space. We also need to derive new inequalities which contains only p_i variables. This can be done by adding together the 3^{rd} and 4^{th} type of constraints of the lifted LP after multiplying them by a positive scalar. The 1^{st} and 2^{nd} constraints are some properties of the linearized variables z_{ij}. These two constraints help to cancel out all the z_{ij} terms so that the resultant inequality contains only p_i variables. Now in the next level, the initial constraints along with these new constraints will be the set of constraints of the integer program. When the level increases, the LP solution gives a tighter relaxation. Each time an improved fractional solution is found which is closer to the optimal integral solution.

2.8 The Branch-and-Bound Method

The branch-and-bound approach was proposed by Land and Doig in 1960. The method enumerates every possible feasible solutions, while pruning those branches which can not lead to a feasible solution. The enumeration can be visualized as a binary tree.

Below are the general steps for the minimization problems.

1. Solve the LP- relaxation optimally and obtain LB.

2. Compute a feasible integral solution UB. Consider the value of LB and UB at the root node of the binary tree.

3. If the value of each decision variable of the LP solution is integer, return the objective function value and stop.

4. If any one of the decision variables is fractional, take that variable into consideration for branching.

5. Let l_1 be the variable we have chosen for branching. Create two child nodes S1 and S2. In subproblem S1, compute an upper bound LUB and a lower bound LLB using LP by assuming l_1=0. In subproblem S2, compute a lower bound RLB by solving LP and also compute an upper bound RUB by taking l_1=1.

6. Update UB with the minimum value among UB, LUB and RUB.

7. If UB is greater than the value of RLB, this indicates that this node needs to be explored more. Solve S2 and go to step 4.

8. If the same condition is true for LLB, solve S1 and go to step 4.

9. If the value of UB is either equal to LLB or RLB, then do not solve the subproblems.

10. If none of the conditions stated in step 7, step 8 or step 9 is true, then also do not solve the subproblems.

11. Return the value of UB as the optimal integral solution for the given problem.

The steps from 4 to 9 are followed recursively to find the optimal integral solution. Recursive algorithm above can be made iterative using the depth first search.

In this chapter, we discussed the terminologies, the related problems and the solution strategies used in this thesis. In the next chapter, we will describe the definition and the background of the problem which we have chosen to work with.

Chapter 3

Point Cover Problem

3.1 Introduction

In this chapter, we will address the problem of covering points using minimum number of axis parallel lines in three dimensional space. First, we will present the problem definition in Section 3.2. We will discuss reductions from a point cover instance to a set cover instance and to a vertex cover instance in Sections 3.3 and 3.4 respectively. Then we will formulate the problem as an integer program. We will also show the corresponding primal and dual linear program relaxation of this IP. We will discuss the research work done so far on this problem in Section 3.7.

3.2 Problem Definition

Input: A set of n points in three dimensional space.

Output: Minimum number of axis parallel lines to cover all the points.

Let the total number of points be n, and L be the set of all axis parallel lines going through the n points. Each point can have at most 3 axis parallel lines in three dimensional space, so $|L| \leq 3n$.

Let us consider the following example. The points are (2, 3, 2), (2, 3, 3), (4, 3, 3). There are three axis parallel lines for each of these three points. Any one of the x- axis, y- axis or z- axis parallel line can be used to cover a point.

Let, $L_{(x,y,z)}$ be the set of axis parallel lines going through a point having coordinate (x, y, z).

$(*, y, z)$ is the x- axis parallel line through point (x, y, z).

$(x, *, z)$ is the y- axis parallel line through point (x, y, z).

$(x, y, *)$ is the z- axis parallel line through point (x , y, z).

In our example,

$L_{(2,3,2)} = \{(*, 3, 2), (2, *, 2), (2, 3, *)\}$

$L_{(2,3,3)} = \{(*, 3, 3), (2, *, 3), (2, 3, *)\}$

$L_{(4,3,3)} = \{(*, 3, 3), (4, *, 3), (4, 3, *)\}$

Here $|L|$=7, as two lines are common to two points. We want to find the set of axis parallel lines that not only cover the points but also are fewest in number. Points (2, 3, 2) and (2, 3, 3) can be covered by the same z- axis parallel line. There will be one point left to be covered. We can choose any one of the x- axis, y- axis or z- axis parallel line to cover point (4, 3, 3). This gives us one optimal solution.

Choosing a x- axis parallel line to cover points (2, 3, 3) and (4, 3, 3), and any one of the x- axis, y- axis or z- axis parallel lines is another optimal solution for this example.

3.3 Reduction from the Point Cover Problem to the Set Cover Problem

We defined the set cover problem in Section 2.2. Here we will show how to formulate a set cover instance given a point cover instance.

Consider an instance of the point cover problem where $p_1,p_2,.....,p_n$ are the points. $L_{p_1},L_{p_2},......,L_{p_n}$ are the sets of axis parallel lines going through points $p_1,p_2,.....,p_n$ respectively. $L = \{l_1, l_2, ..., l_m\}$ is the set of m axis parallel lines going through the n points.

The universal set U consists of all the n points, U=$\{p_1, p_2,....., p_n\}$. For every line $l_i \in L$, there is a subset S_{l_i} of points that lie on that line. The goal is to find a collection C of $\{S_{l_i} | l_i \in L\}$ such that every element of U belongs to some subset in C and $|C|$ is minimum possible.

We illustrate the reduction on the previous example and find out the minimum number of subsets so that every element of U is covered.

Here,

p_1= (2, 3, 2)

p_2= (2, 3, 3)

p_3= (4, 3, 3)

U= $\{p_1, p_2, p_3\}$

l_1 = (*, 3, 2)

l_2 = (2, *, 2)

l_3 = (2, 3, *)

l_4 = (*, 3, 3)

l_5 = (2, *, 3)

l_6 = (4, *, 3)

l_7 = (4, 3, *)

$S_{l_1} = \{p_1\}$

$S_{l_2} = \{p_1\}$

$S_{l_3} = \{p_1, p_2\}$

$S_{l_4} = \{p_2, p_3\}$

$S_{l_5} = \{p_2\}$

$S_{l_6} = \{p_3\}$

$S_{l_7} = \{p_3\}$

We can choose S_{l_3} and S_{l_4} which gives the minimum sized set cover. It is also optimum to pick S_{l_3} and S_{l_6}, or S_{l_3} and S_{l_7}, or S_{l_4} and S_{l_1}, or S_{l_4} and S_{l_2}. Any one of these combinations gives the minimum cover for this example.

In general, consider an instance of the point cover problem with n points and L axis parallel lines. First, we create an instance of the set cover problem where the universal set U consists of all points. There are $|L|$ number of subsets where each subset S_{l_i} contains the points that lie on line l_i. Let k be the minimum number of subsets that cover all the elements of the universal set U, then there exists k axis parallel lines to cover n points of the given point cover instance. Again, if there are k lines to cover n points of the point cover instance, then there exists k subsets which cover all the elements of U of the set cover instance. This means that we approximate the point cover problem by reducing it to the set cover problem. In general, the set cover problem can be approximated within a factor of $O(\log n)$ and no better [7]. Therefore we need a more direct approximation algorithm for the point cover problem which does not rely on the reduction to the set cover problem.

3.4 Transformation from the Point Cover Problem to the Vertex Cover in Hypergraphs

3.4.1 Construction of a Bipartite Graph from 2D Point Cover Instance

It is proved by Hassin and Megiddo [13] that the point cover problem in two dimensions can be transformed into a vertex cover problem in bipartite graphs. Here we discuss their reduction. Given is a set of points in 2D, and a set of axis parallel lines consisting of vertical and horizontal lines which cover all of the points. To transform this point cover instance into a vertex cover instance, we need to construct a bipartite graph G with a vertex set V and an edge set E. V consists of the potential axis parallel lines going through the points and E contains all the points. We know that each edge of a graph has two endpoints. As each point has one vertical line and one horizontal line, we can think of each line as a vertex, and each point as a edge in the bipartite graph. The graph is bipartite because there is no point that lies on two horizontal (vertical) lines. An example is shown below.

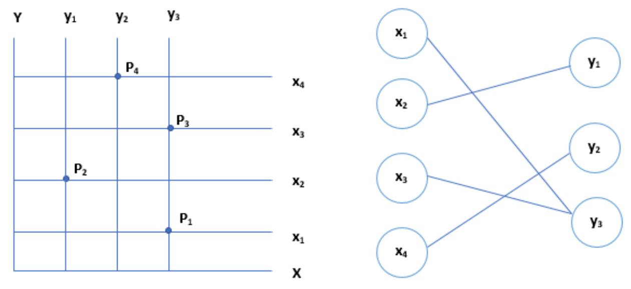

Figure 3.1: Construction of a Bipartite graph from a 2D Point cover instance

A minimum vertex cover is the same as the minimum number of axis-parallel lines required to cover all points in 2D and vice versa. The minimum vertex cover in a bipartite graph can be computed in polynomial time using the König- Egerváry theorem. The maximum matching problem is the dual of the minimum vertex cover. Bipartite matching problem can be solved in polynomial time by formulating it as a maximum flow problem [8]. So the point cover problem in two dimensional space can be solved in polynomial time optimally given the König- Egerváry theorem below.

Theorem 3.1. *The size of the minimum vertex cover is the same as the size of the maximum matching in a bipartite graph.*

3.4.2 Construction of a Tripartite Hypergraph from 3D Point Cover Instance

A hypergraph H=(V, E) consists of a set of vertices V and a set of hyperedges E. A hyperedge is a subset of vertices. A hypergraph is called 3-uniform if each edge has exactly 3 vertices. A 3-uniform hypergraph is called tripartite if the vertex set V can be partitioned into three subsets such that every hyperedge contains exactly one vertex from each of these three subsets.

Given an instance of the 3D point cover problem, we can construct an equivalent tripartite hypergraph by considering each axis parallel line as a vertex and each point as a hyperedge. This idea of reduction is taken from the master's thesis of Jahan [17]. Let us consider an example in Table 3.1 of the point cover problem in 3D.

Table 3.1: An Example of the Point Cover Problem in 3D

Point	x- axis parallel line	y- axis parallel line	z- axis parallel line
p_1 = (2, 3, 2)	x_1 = (*, 3, 2)	y_1 = (2, *, 2)	z_1 = (2, 3, *)
p_2 = (2, 3, 3)	x_2 = (*, 3, 3)	y_2 = (2, *, 3)	z_1 = (2, 3, *)
p_3 = (4, 3, 3)	x_2 = (*, 3, 2)	y_3 = (4, *, 3)	z_2 = (4, 3, *)

The axis parallel lines along the three axes are the three subsets of the vertex set V. The corresponding hypergraph of this table is in Figure 3.2.

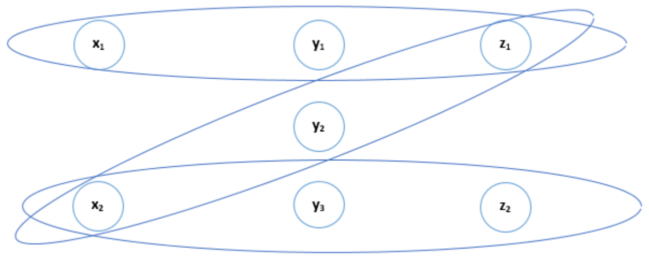

Figure 3.2: Construction of a Tripartite hypergraph from a 3D Point Cover problem instance

Here we can see that the minimum number of vertices required to cover each hyperedge is 2 which is the same as the number of axis parallel lines needed to cover all points.

In general, a point cover instance in d-dimensions can be reduced to a vertex cover instance in d-uniform d-partite hypergraphs. It is known that the vertex cover problem is NP-complete for 3-uniform 3-partite hypergraphs [12]. We can approximate the point cover problem by reducing it to the vertex cover problem in 3- uniform 3-partite hypergraphs.

3.5 Integer Program for Point Cover Problem

Consider a set of n points p_1, p_2, ..., p_n in 3-dimensional space. S_1, S_2, S_3, ..., S_n are the sets of axis parallel lines associated with point p_1, p_2,......,p_n respectively. Let $S = \bigcup_{i=1}^{n} S_i$. The goal is to pick lines from S which cover all of the n points, and the number of selected lines should be as few as possible. A line is said to cover a point if the point lies on that line. An integer programming formulation for the point cover problem is given below.

$$\text{minimize} \sum_{i=1}^{m} l_i \tag{3.1}$$

$$\text{subject to} \sum_{l_i \in S_j} l_i \geq 1, \qquad \forall j \in \{1,2,....,n\}$$

$$l_i \in \{0,1\}, \qquad \forall i \in \{1,2,....,m\}$$

Here, l_1, l_2, l_3,......., l_m are the binary variables which correspond to the lines in S. The value of an axis parallel line l_i will be 1 if that line is picked in the solution and 0 if the line is not selected in the solution.

3.6 Linear Programming Relaxation

3.6.1 Primal LP Formulation

We can transform an integer program into a linear program by relaxing the integrality constraints. In (3.1), $l_i \in \{0,1\}$ ($\forall i \in \{1,2,....,m\}$) is the only integrality constraint. We transform this into $l_i \geq 0$, $\forall i \in \{1,2,....,m\}$. The linear program relaxation for the point cover problem is given below.

$$\begin{aligned} \text{minimize } & \sum_{i=1}^{m} l_i \qquad (3.2) \\ \text{subject to } & \sum_{l_i \in S_j} l_i \geq 1, \qquad \forall j \in \{1,2,....,n\} \\ & l_i \geq 0, \qquad \forall i \in \{1,2,....,m\} \end{aligned}$$

3.6.2 Dual LP Formulation

In Section 2.3, we describe how to formulate a dual program from a primal linear program. For the point cover problem, we minimize the number of axis parallel lines going through n points in the primal program. In the dual program, we maximize the number of points subject to the packing constraints. Let Q_1, Q_2, ..., Q_m be the sets of points which lie on lines l_1, l_2, ..., l_m respectively. Let $Q = \bigcup_{i=1}^{m} Q_i$. The goal is to choose points from Q so that the maximum number of points are picked subject to some constraints.

$$\text{maximize} \sum_{j=1}^{n} p_j \tag{3.3}$$

$$\text{subject to} \sum_{p_j \in Q_i} p_j \leq 1, \qquad \forall i \in \{1,2,....,m\}$$

$$p_j \geq 0, \qquad \forall j \in \{1,2,....,n\}$$

3.7 Previous Research Work

Hassin and Megiddo [13] were the first to study the point cover problem. They described the point cover problem as a special case of the hitting set problem. The hitting set problem is as follows: given a universal set X=$\{1, 2, ..., n\}$, and m subsets where each subset Y_i contains elements from X, the goal is to find a subset S of X that hits each Y_i where i=1, 2, ..., m and $|S|$ is minimum possible. The integer program for the hitting set problem is similar to the set cover problem.

The point cover problem is a special case of the hitting set problem. They gave a greedy algorithm and showed that it does not have a constant factor approximation ratio. The greedy algorithm picks a line which covers the maximum number of points and creates a subproblem by removing the covered points, and repeat until all the points are covered. Johnson [18] and Chvatal [4] proved that the approximation ratio of this greedy algorithm for the set cover problem is $\log n$. This greedy heuristic does not provide a constant performance ratio for the point cover problem even in two dimensional space. Hassin and Megiddo [13] gave an example in 2 dimensional space.

In the example (shown in Figure 3.3), at first n disjoint sets are constructed where each set S_i $(1 \leq i \leq n)$ contains $n!$ points. Each set S_i is divided into $\frac{n!}{i}$ disjoint subsets where each subset consists of i points.

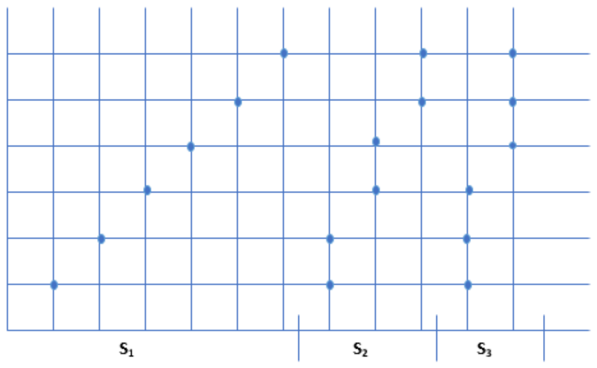

Figure 3.3: Example discussed by Hassin and Megiddo [13]

In Figure 3.3, there are n=3 sets S_1, S_2, S_3 and each set contains 3!=6 points. Then S_1 is divided into $\frac{6}{1}$ = 6 subsets containing 1 point in each, S_2 is partitioned into $\frac{6}{2}$ = 3 subsets each containing 2 points, and S_3 is divided into $\frac{6}{3}$ = 2 subsets consisting of 3 points in each. If we use the greedy algorithm, it can choose either one of the the last two vertical lines as these lines cover the maximum number of points for this instance, or any one of the horizontal lines because each horizontal line also covers the same number of points. This statement is true in each recursive step because in each iteration whether the algorithm chooses a vertical line or a horizontal line in the solution, there exists at least one horizontal and one vertical line which can cover the maximum number of points in that subproblem. If the algorithm chooses the horizontal lines in each subproblem, then the point cover instance is solved optimally. The optimal value is n!=3!=6. But if the algorithm chooses the vertical lines in each iteration, then the solution contains 11 lines. In general, the greedy algorithm uses $n!H(n)$ lines where $H(n)= \sum_{j=1}^{n} \frac{1}{j}$.

There is another approximation algorithm discussed by Hassin and Megiddo [13]. While there is an uncovered point, select both the horizontal and the vertical line going through that point. Remove all the points that lie on both the lines. If the algorithm selects $2k$ lines, then there are k points such that no two lie on a line. Therefore, the algorithm is a 2-approximation algorithm for the point cover problem in 2D. In general, it is a d-approximation algorithm for the point cover problem in d dimensional space.

Another approximation algorithm [16] for the point cover problem is to solve the linear program and round the optimal fractional solution. Each line with fractional value at least $\frac{1}{d}$ is selected to be in the solution. This rounding gives an integral solution and the performance ratio of this algorithm is d.

Gaur and Bhattacharya [9] gave a (d-1) approximation algorithm for the point cover problem in d dimensions which was based on deterministic rounding.

Jahan [17] proposed an iterative rounding algorithm, and a branch-and-bound algorithm based on the LP solution using the dual-simplex method to obtain an integral solution of the point cover problem. She also gave another iterative rounding algorithm based on the primal dual method.

The point cover problem in 2D can be solved in polynomial time which we discussed in Section 3.4.1. But for higher dimensions where $d \geq 3$, the point cover problem is NP-complete. The point cover problem in d dimensions can be reduced to the vertex cover problem in d-uniform d-partite hypergraphs. Lovasz [2] gave a $\frac{d}{2}$ approximation algorithm for d-partite hypergraphs by rounding the optimal solution to the standard LP relaxation. This implies that we can approximate the point cover problem in 3D with approximation ratio $\frac{3}{2}$ by reducing it to the vertex cover problem in 3-uniform 3-partite hypergraphs.

So far we have seen the best performance ratio of the point cover problem is $\frac{d}{2}$ for d dimensions, which implies a $\frac{3}{2}$ approximation algorithm for the point cover problem in 3 dimensional space.

In the next chapter, we will present the algorithms which we have developed and implemented to solve an instance of the point cover problem in 3 dimensional space.

Chapter 4

Lagrangian Approaches to Solve the Point Cover Problem

4.1 Introduction

In Section 3.7 of Chapter 3, we noted an algorithm with an approximation ratio of 2 for covering points with axis parallel lines in three dimensional space. In this chapter, we introduce Lagrangian based algorithms to approximate the point cover problem. In Section 4.2, we present three strategies for computing lower bounds. In Section 4.3, we describe three approaches for computing upper bounds. We test the algorithms empirically.

4.2 Lower Bounds

4.2.1 The Lagrangian Relaxation Method

The first approach that we study to obtain a lower bound is a Lagrangian relaxation of the standard integer program. We use the subgradient optimisation method to solve the Lagrangian relaxation.

Let us revisit the integer program for the point cover problem (4.1) which was formulated in Section 3.5.

$$\begin{aligned} \text{minimize} \quad & \sum_{i=1}^{m} l_i && (4.1) \\ \text{subject to} \quad & \sum_{l_i \in S_j} l_i \geq 1, \qquad \forall j \in \{1,2,....,n\} \\ & l_i \in \{0,1\}, \qquad \forall i \in \{1,2,....,m\} \end{aligned}$$

Here, n is the total number of points. m is the total number of axis parallel lines for n points. l_1, l_2, l_3,......., l_m are the binary variables corresponding to the lines. S_1, S_2, S_3, ..., S_n are the sets of axis parallel lines associated with point 1, 2, ..., n respectively, where S_j is the set of axis parallel lines that pass through point j. The value of an axis parallel line l_i is 1 if that line is selected in the solution and 0 if the line is not selected.

Steps for Computing Lower Bound:

We follow the steps stated below to generate a lower bound using a Lagrangian relaxation.

1. At first, we formulate the Lagrangian lower bound program (*LLBP*) by introducing a Lagrange multiplier vector λ where λ_j is the Lagrange multiplier for constraint j. The *LLBP* for the point cover problem is:

$$\text{minimize} \sum_{i=1}^{m} l_i + \sum_{j=1}^{n} \lambda_j (1 - \sum_{l_i \in S_j} l_i) \tag{4.2}$$

$$\text{subject to} \quad l_i \in \{0,1\}, \qquad \forall i \in \{1,2,....,m\}$$

The general description of the process of formulating a *LLBP* from an integer program is given in Section 2.6 of Chapter 2.

2. We initialize each Lagrange multiplier with a random fractional value between 0 and 1. We also decide values for some user defined parameters. Z_{UB} is an upper bound and we assume Z_{UB} as the total number of points, because to cover n points at most n axis parallel lines are required. We set π to an initial value of 2.

3. We solve the *LLBP* optimally using the current value of Lagrange multipliers using the procedure in Section 2.6 of Chapter 2. After solving the *LLBP*, we find a solution Z_{LB} (a lower bound) and a vector l where each l_i has a value 0 or 1.

4. We compute the subgradient g_j for each constraint j using the following formula

$$g_j = 1 - \sum_{l_i \in S_j} l_i \tag{4.3}$$

5. We calculate the step size T as follows:

$$T = \pi(Z_{UB} - Z_{LB}) / \sum_{j=1}^{n} (g_j)^2 \tag{4.4}$$

6. We update the new Lagrange multipliers using the following equation.

$$\lambda_j = \max(0, \lambda_j + T g_j) \tag{4.5}$$

7. We repeat steps 3 to 6 for 200 iterations. We also reduce the value of π by half if the value of Z_{LB} does not increase for four consecutive iterations.

Algorithm:

Below is our algorithm for computing a lower bound of the point cover problem by solving the Lagrangian relaxation using a subgradient optimisation method.

```
Algorithm 1: Subgradient Method to solve Lagrangian Relaxation          Running Time

Input: A minimization IP for the point cover problem in standard form.
Output: A Lower Bound on the value of the objective function.
1: initialize $\lambda$, $Z_{UB}$, $\pi$, iteration as described.              $O(1)$
2: $Z_{LB\_MAX} \leftarrow -\infty$.                                           $O(1)$
3: while iteration $\leq 200$ do
4:     Compute $l$.                                                            $O(n)$
5:     Calculate $Z_{LB}$ by solving the LLBP.                                 $O(n)$
6:     $Z_{LB\_MAX} \leftarrow max(Z_{LB\_MAX}, Z_{LB})$.                      $O(1)$
7:     Compute subgradient by using equation 4.3.                              $O(n)$
8:     Compute the step size using the formula 4.4.                            $O(1)$
9:     Update lagrange multipliers using equation 4.5.                         $O(n)$
10:    if $Z_{LB\_MAX}$ does not improve for four iterations then              $O(1)$
11:        $\pi \leftarrow \frac{\pi}{2}$.                                     $O(1)$
12:    end
13: end
14: return $Z_{LB\_MAX}$.                                                      $O(1)$
```

We use 200 iterations to terminate the process. It is also possible to use the value of π as a termination rule.

Time Complexity of the Algorithm:

Assuming that the number of points in an input instance is n, then the number of rows in the constraint matrix is also n and the size of the vector l can be at most $3n$. For Algorithm 1, the running time of each step is in the column on the right hand side.

There are 200 iterations of step 3 to step 13. In summary, Algorithm 1 takes $O(n)$ time where n is the total number of points in the input instance.

4.2.2 The Lift-and-Project Method using LP

In the lift-and-project method [23, 25], we lift an integer program with n variables to one with n^2 variables by adding auxiliary variables and linear inequalities. After solving the modified system, we project it back to the lower dimensional space to obtain a solution to the original LP.

Steps for Computing Lower Bound:

We illustrate the steps of the lift-and-project method below. The detailed description of each step is provided in Section 2.7 of Chapter 2.

1. The first step is to homogenize the equations. After homogenizing, all the terms in a constraint have the same degree. We introduce an auxiliary variable l_0 which is always 1. So the integer program becomes

$$\begin{aligned} \text{minimize} \quad & \sum_{i=1}^{m} l_i \qquad (4.6) \\ \text{subject to} \quad & \sum_{l_i \in S_j} l_i \geq l_0, \qquad \forall j \in \{1,2,....,n\} \\ & l_0 = 1,\ l_i \in \{0,1\}, \quad \forall i \in \{1,2,....,m\} \end{aligned}$$

2. The second step is to formulate a quadratic program from the homogenized integer program (4.6). We start by changing the constraint $l_i \in \{0,1\}$ into a quadratic constraint.

$$l_i(1-l_i) = 0, \qquad \forall i \in \{1,2,....,m\} \qquad (4.7)$$

We use the extra variable l_0 to remain inside a homogeneous setting, we can rewrite equation (4.7) as follows

$$l_i(l_0 - l_i) = 0, \qquad \forall i \in \{1, 2, ..., m\} \tag{4.8}$$

$$l_0 l_i - l_i l_i = 0, \qquad \forall i \in \{1, 2, ..., m\}$$

We know that there are at most three axis parallel lines for each point. Let us write the first constraint of the homogeneous version of IP (4.6) for a particular point p, where l_i, l_j and l_k are the three axis parallel lines going through point p as,

$$l_i + l_j + l_k \geq l_0 \tag{4.9}$$

To obtain a quadratic equation of (4.9), we multiply this with l_i and (l_0-l_i).

$$l_i(l_i + l_j + l_k - l_0) \geq 0 \tag{4.10}$$

$$(l_0 - l_i)(l_i + l_j + l_k - l_0) \geq 0$$

We multiply all the constraints with l_i and (l_0-l_i) for all i. If L is the set of all axis parallel lines going through n points, we need to multiply with each l_i in L. So finally we get $2n|L|$ constraints from n constraints.

3. We simulate the quadratic program using a linear program. We linearize the quadratic program by introducing extra nonnegative variables y_{ij} for term $l_i l_j$. The properties of y_{ij} variables which will be included in the constraint set of the linear program are given below.

a) y_{ij} = y_{ji} which is the symmetry of the product.

b) l_i = y_{0i} as y_{0i} = $l_i l_0$ =l_i.

c) y_{0i} = y_{ii} , this property can be derived from equation (4.8).

4. We solve the linearized version of the LP using the dual simplex method in MATLAB.

5. We project the solution of the lifted LP back onto the variables l_i using property c.

Algorithm:

The algorithm for computing lower bound using the lift-and-project method is described below:

```
Algorithm 2: Lift-and-Project method for Computing Lower Bound
Input: A minimization integer program in standard form.
Goal: A lower bound on the value of the optimal solution.
1: Construct a Quadratic Program as shown in Step 2.
2: Linearize the quadratic program as shown in Step 3.
3: Solve the modified problem as an LP using the dual-simplex method.
4: Project the solutions onto the space of $l_i$ variables as shown in Step 3($c$).
```

Time Complexity of the Algorithm:

Assume that the total number of points for an input instance of the point cover problem is n. So the total number of axis parallel lines is at most $3n$ which we denote by L.

The quadratic program can be generated in $O(2nL)$ time. The linearization of the LP can be done in $O(nL)$ time. This indicates that the generation of the constraint matrix can be done in polynomial time in the lift-and-project method.

Now, deciding whether the running time of Algorithm 2 is polynomial or exponential depends on the running time of the dual simplex method which is in step 3. It has been shown by Klee and Minty [21] that the worst case complexity of simplex is exponential. Spielman and Teng [26] observed that the simplex algorithm ”usually” takes polynomial time. In practice, so far we have observed that the simplex method takes linear time to solve an input instance of the point cover problem. Projection of the results back onto lower dimensions takes $O(nL)$ time. Therefore, Algorithm 2 takes $O(nL)$ time in practice.

As the value of L is at most $3n$, we can also say that Algorithm 2 takes $O(n^2)$ time in practice.

4.2.3 Solve the Lift-and-Project LP using the Subgradient Optimisation

This is a hybrid method in which we solve the lift-and-project LP using the subgradient optimisation procedure. We use the algorithm proposed in Section 4.2.1, the only difference is that the constraint matrix is formulated using the lift-and-project method.

Steps for Computing a Lower Bound:

1. We introduce a quadratic program based on the initial integer program. The detailed description is given in Section 4.2.2.

2. We linearize the quadratic program.

3. We add symmetry constraints on the variables.

4. Then we solve this modified linear program using the subgradient method for Lagrangian relaxation proposed in Section 4.2.1.

Time Complexity of the Algorithm:

In this algorithm, we create the constraint matrix by using the lift-and-project method and solve the linear program by using the subgradient optimisation method. In Section 4.2.2, we computed that it takes $O(nL)$ time to generate the linear program, where n is the number of points and L is the number of axis parallel lines in the input instance. In Section 4.2.1, we observed that the subgradient method takes $O(n)$ time as the number of constraints is n. Here, the number of constraints generated by using the lift-and-project method is $2nL$. So the running time of our proposed hybrid algorithm is $O(nL)$. As the value of L is at most $3n$, we can also say that our proposed hybrid algorithm takes $O(n^2)$ time in practice.

4.3 Algorithms for Computing an Upper Bound

In this section, we describe three approaches for computing an upper bound. We named the three approaches as the Iterative Rounding Algorithm 2, the Iterative Lagrangian Relaxation Algorithm, and the Branch-and-Bound technique using the Subgradient Optimisation Method.

4.3.1 The Iterative Rounding Algorithm 2

The scheme of iterative rounding was proposed by Jahan [5] in her master's thesis for the point cover problem. We remove few conditions and change one parameter, and implement the modified version. These modifications give an improved approximation factor in practice.

Steps:

The steps of the algorithm are given below.

1. We initialize a threshold variable α with a value 2, which we use to round the fractional values.

2. We solve the primal LP using the dual simplex method in MATLAB.

3. We retrieve the solution of the dual LP.

4. If none of the variables of the primal solution contains any fractional value, that means it is an integral solution. We return the upper bound and stop the algorithm.

5. If there are n points and the dual solution is at least $\frac{n}{\alpha}$, this indicates that it is possible to get a α- approximate integral solution by covering each point with a unique line.

6. If both of the conditions stated in Step 4 and Step 5 are false, then we find a primal variable with value $\frac{1}{\alpha}$ or more and round it to 1. We go to Step 2 for the next iteration without changing the current solution of the variables which have been rounded to 1.

7. If all of the dual variables have value greater or equal to $\frac{1}{\alpha}$, then the condition stated in step 5 is satisfied, therefore there is a point with dual value $\leq \frac{1}{\alpha}$. The algorithm uses all

the lines passing through this point in the primal solution. We go to Step 2 after choosing the corresponding variables to 1.

Algorithm:

Below is the iterative rounding algorithm 2.

Algorithm 3: Iterative Rounding Algorithm 2.

```
Input: A minimization linear program in standard form.
Goal: Upper bound where the values of all variables are integral.
1: initialize the rounding factor $\alpha$.
2: find the optimal primal LP solution X using the dual simplex method.
3: find the optimal dual LP solution Y using the dual simplex method.
4: if there exists no variables with fractional value in Primal LP then
5:        return X.
6: else if $Y \geq \frac{n}{\alpha}$ then
7:        return the greedy solution of the IP.
8: else if there exists a primal variable with fractional value $\geq \frac{1}{\alpha}$ then
9:        round the first fractional variable to 1.
10:       goto Step 2 with the modified LP.
11: else if there exists a dual variable with value $\leq \frac{1}{\alpha}$ then
12:       pick the 3 lines going through the first such point.
13:       goto Step 2 with the modified LP.
14: end
```

This is a recursive procedure which we call from the main function with the initial LP. When any change is made to any variables, the updated values of the current iteration are passed to the next iteration.

Modifications made to the Algorithm of Jahan [17]:

Jahan [17] used a constant ε with value $0<\varepsilon \leq 1$ and considered the value of α as $\frac{1}{4}$ + $\sqrt{\frac{1}{4}+d(1-\varepsilon)}$. We considered the value of α as 2 which is not dependent on the value of ε. There was a condition after step 7 in Jahan's Algorithm [17]: if the value of the dual LP solution Y is less or equal to $\frac{d}{\varepsilon}$ then return the cheapest cover by exploring all the subsets of lines with size no more than $\frac{d^2}{\varepsilon}$. In our case we did not include this step in the implementation.

Time Complexity of the Algorithm:

In Algorithm 3, most of the operations require linear time and the number of recursive calls is at most $3n$. So the complexity of this algorithm is dependent on the running time of the dual simplex method. For the reasons discussed in Section 4.2.2, we can say that it is possible to compute the upper bound of the point cover problem using the iterative rounding algorithm 2 in polynomial time, $O(n^2)$ is observed in practice.

4.3.2 The Iterative Lagrangian Relaxation Algorithm

We implement another iterative method for computing upper bounds for the point cover problem. This iterative method extends the solution of the variables (obtained by solving the current Lagrangian lower bound program) by solving a recursive problem. We already know that the upper bound generated by using this algorithm is always an integral solution because in the vector generated by solving the Lagrangian relaxation all the variables have a value either 0 or 1.

Steps:

1. At first, we solve the LP using the subgradient optimisation algorithm described in Section 4.2.1. We use the value of the solution vector l, which we get while solving *LLBP*. The method of solving *LLBP* and the output vector l is discussed in Section 2.6.

2. We derive a subproblem by removing those points which are covered by the solution *l* to *LLBP*.

3. Repeat Step 1 and 2 until all of the points are covered. Details are in Algorithm 4.

Algorithm:

Algorithm 4: The Iterative Lagrangian Relaxation Algorithm.

```
Input: A minimization integer program in standard form.
Goal: Integral Solution.
1: $UB \leftarrow 0$.
2: while all points are not covered do
3:     initialize $\lambda$, $Z_{UB}$, $\pi$ and iteration.
4:     $Z_{LB\_MAX} \leftarrow -\infty$.
5:     $final\_l$= vector of all 1s.
6:     while iteration $\leq 200$ do
7:         Compute l given the current lagrange multipliers.
8:         Calculate $Z_{LB}$ given l and the lagrange multipliers.
9:         $Z_{LB\_MAX} \leftarrow max(Z_{LB\_MAX}, Z_{LB})$.
10:        if ($Z_{LB}$==$Z_{LB\_MAX}$) then
11:            $final\_l \leftarrow min(l, final\_l)$
12:        end
13:        if $Z_{LB\_MAX}$ does not improve in consecutive 4 iterations then
14:            return $\pi \leftarrow \frac{\pi}{2}$
15:        end
16:        Compute Subgradient and the Step size.
17:        Update Lagrange multiplier vector.
18:    end
19:    Find the points that are covered.
20:    $UB \leftarrow UB + \textbf{sum}(final\_l)$.
```

```
21:         Create a subproblem by removing the covered points.
22:         goto step 2.
23: end
24: return $UB$.
```

Time Complexity of the Algorithm:

While trying to obtain an upper bound using Algorithm 4, we use the subgradient procedure in each recursive call and determine the points which are covered. In the worst case, we need at most n recursive calls to the subgradient procedure. We show in Section 4.2.1 that the running time of the subgradient algorithm with the Lagrangian relaxation technique is $O(n)$. So, the running time of algorithm 4 is $O(n^2)$.

4.3.3 The Branch-and-Bound Method using the Lagrangian Relaxation Approach

We use an exhaustive search technique which possibly involves solving a large number of linear programs to find an optimal integral solution. To solve the LP, Lagrangian relaxation program with the subgradient optimisation method is used.

Branching Strategy:

Let, l be the solution vector of the variables obtained by solving *LLBP*. l contains all the axis parallel lines going through n points, and the vector l is integral. So, we can not branch at a node based on the fractional output of a decision variable which is the traditional way of branching from a node. First we arrange all the lines in some random order. We choose one line at a time in that order. The root node is at level 1. We explore the root node by creating two subproblems- left subproblem S1 and right subproblem S2, setting the value for the first line as 0 and 1 respectively.

In S1, we have a subproblem where l_1 is set to 0, this means we do not pick l_1 in the solution. We compute an upper bound *LUB*, and a lower bound *LLB* for the left subproblem. In S2, we have a subproblem where l_1 is in the solution. We compute an upper bound *RLB*, and a lower bound *RUB* for the right subproblem considering l_1 as 1.

In general, if we are currently in level i and we need to explore a node, then we set the i-th line as 0 and 1 and two new subproblems are created. At level i, the value of the last (i-1) lines is already set. By observing the path from the root to a node, it is possible to get the values of l_1, l_2, l_3,......l_{i-1}, as shown in Figure 4.1.

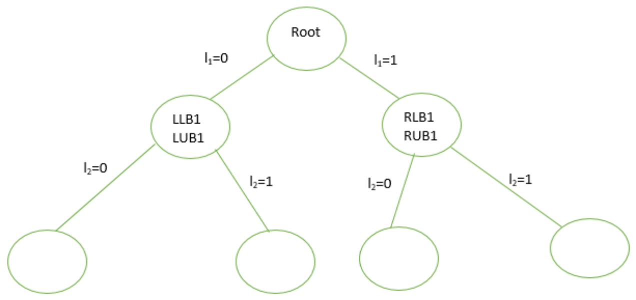

Figure 4.1: Branch and Bound

Enumeration Tree:

At a given level, we go deeper using the idea of DFS which is described in Section 2.6 of Chapter 2. As we perform DFS at a node, we need to keep track of the other nodes which we explore later. One of the way is to use a recursive function, but recursive function needs a stack space which may be an issue when the depth of recursion is large. We use a stack to implement recursion, which is described in Section 2.5.

We want to evaluate a branch where l_i is set to 1 first. In order to evaluate a branch where l_i is set to 1, we need to push the opposite branch onto the stack and then push the desired one. The subproblems will be visited in the right order.

In our implementation, we use a vector named *index*, initially empty. The size of this *index* vector grows by 1 at each level. If the value at the i-th position is 0, that means the corresponding line is not picked in the solution. If it is 1, the line is picked in the solution. We are actually enumerating, but we do not need to explore all the nodes because of our pruning technique.

Algorithm:

Algorithm 5: Branch-and-Bound Algorithm.

Input: A minimization linear program in standard form.

Goal: An Optimal Integral Solution.

```
1: Compute $LB$ using the Lagrangian relaxation algorithm described in 4.2.1.
2: Compute $UB$ using the Iterative lagrangian relaxation algorithm discussed in 4.3.2.
3: initialize index, lc and rc vectors as empty.
4: push the vector index into the stack.
5: while stack is not empty do
6:      pop index vector from stack
7:      $node \leftarrow node + 1$
8:      $lc \leftarrow [index;0]$
9:      $rc \leftarrow [index;1]$
10:     Solve the left subproblem S1 and get LLB and LUB.
11:     Solve the right subproblem S2 and get RLB and RUB.
12:     $UB \leftarrow \mathbf{min}(UB, LUB, RUB)$.
13:     if $UB > LLB$ then
14:         push lc vector into the stack.
15:     end
16:     if $UB > RLB$ then
17:         push rc vector into the stack.
18:     end
19:     if $(UB = RLB) \vee (UB = LLB)$ then
20:         if $UB = RLB$ then
21:             return $UB$
22:         end
```

```
23:           **if** $UB = LLB$ **then**
24:               **return** $UB$
25:           **end**
26:       **end**
27:       **if** *none of the above condition satisfies* **then**
28:           **return** $UB$
29:       **end**
30: **end**
31: **return** $UB$ *and number of nodes.*
```

Time Complexity of the Algorithm:

The branch-and-bound strategy can be visualized as a binary tree where the left child is the case when line l_i is not picked in the solution, and the right child is the case where line l_i is included with value 1 in the solution. The tree has a search space of at most $O(2^L) \approx O(2^{3n})$ nodes because the depth of the tree can be at most L. The work done at each internal node is polynomial. Therefore, Algorithm 5 has a running time exponential in the number of points n.

In this chapter, we described our proposed algorithms for obtaining the upper bound of the point cover problem. We also discussed a few methods which we studied for computing the lower bound. We will present the experimental results and our observation in the next chapter.

Chapter 5

Experiment and Evaluation

5.1 Introduction

In this chapter, we present our experimental results and analyses of the algorithms discussed in Chapter 4. First, we give a brief description of the specifications of the computer used for the experiments. We then mention the process of generating the inputs to the algorithms. Finally, we make some observations and analyze the output of the algorithms. We perform experiments on large instances and observe the approximation ratio of the point cover problem (empirical).

5.2 General Empirical Evaluation

This section will describe the lower bound and the upper bound results of the proposed algorithms.

5.2.1 Experimental Setup

We performed most of the experiments using MATLAB R2016b. For large instances, we used Octave with drop in acceleration using GPU [1]. The platform where MATLAB R2016b is installed has the following specifications:

Processor Model Name: Intel Core i7-5500U CPU

Clock Speed: 2.40GHz

RAM Size: 8.00 GB

System Type: 64-bit Operating System

Environment: Windows 10

5.2.2 Data Set

We generated points in 3D grid randomly. In the initial stage, we experimented with instances on a 10x10x10 grid. Points were selected at random with probability p, p in [0.1,0.2,....1] with skips of 0.1. For each probability, we generated 5 instances. After analyzing the results, we observed that the instances with probability 0.4 are typically ”hard” to solve for all the proposed algorithms. Then we decided to generate 1000 input instances on an 8x8x8 grid where for each input instance, each point is picked with a probability of 0.4.

5.2.3 The Lower Bounds

We used four ways to compute a lower bound of the point cover problem. The simplex method is used to solve the linear program relaxation optimally. We also implemented a LP-based lift-and-project algorithm described in Section 4.2.2. In both the cases, ”linprog” method of MATLAB R2016 is used to solve the LP. LP and the lift-and-project LP are also solved using the subgradient optimisation method to solve a Lagrangian relaxation for both the problems.

The total number of points in each instance varies from 170 to 240. We generated 1000 instances whose probability of selecting a point is 0.4. A few distinct group of instances have the same number of points. For that reason, we plot the results of all four algorithms against the serial number of the instances where the instances are sorted according to the increasing order of the number of points.

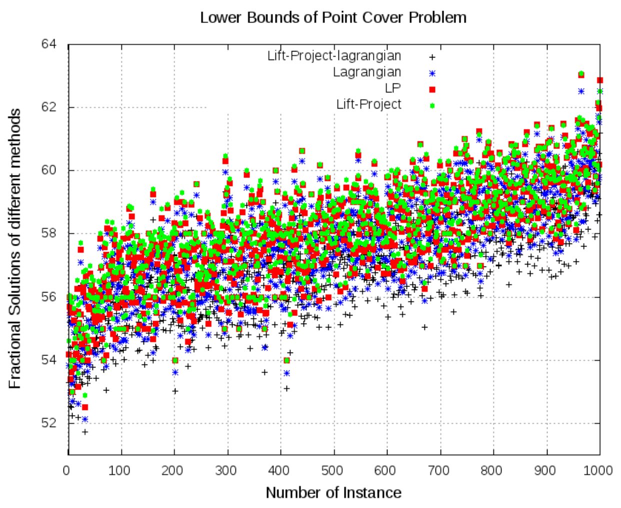

Figure 5.1: Lower Bounds

In Figure 5.1, we plot the instance number along the x-axis, and the fractional solutions of different methods along the y-axis. Based on the figure, we can say that the lift-and-project method is marginally better than the LP. Out of 1000 instances, the lower bound computed using the lift-and-project method is either equal or better than the LP lower bound in 925 cases. We also observe from the graph that when the number of points increases, there is a possibility of not finding better lower bound than the LP lower bound. This occurred because we took a timespan of 600 seconds to solve the lift-and-project LP. If we increase the timespan with the increase of the number of points, the chance of getting better results also increases. On the other hand, the lower bound found by using the Lagrangian relaxation is close to the LP lower bound, but there is a tiny gap between the two. There is no noticeable improvement when the lift-and-project LP is solved using the subgradient optimisation method.

5.2.4 Time to Compute the Lower Bounds

As we already know, the number of variables and constraints increases in the lift-and-project procedure and this modified LP takes greater time and space to solve depending on the input size. For a few instances in our data set, we solved the lift-and-project LP using MATLAB and it takes around 10800 seconds to solve. So we chose a time-limit of 600 seconds and solved the 1000 instances. Only 75 instances generated lower bound worse than the LP lower bound within that timespan. The lift-and-project method solved using the method of Lagrangian relaxation took more time than either the Lagrangian relaxation or the LP, because of the large constraint matrix size and the increase in the number of variables.

We notice that the Lagrangian relaxation solved using the subgradient optimisation method, and the LP take the least time to compute a good lower bound among the four strategies. We graphically plot the ratio of the running time of the subgradient optimization method, and the dual-simplex method used to solve the LP to observe which method is faster. Time is computed in seconds in both cases. We sort the instances according to the increasing order of the number of points.

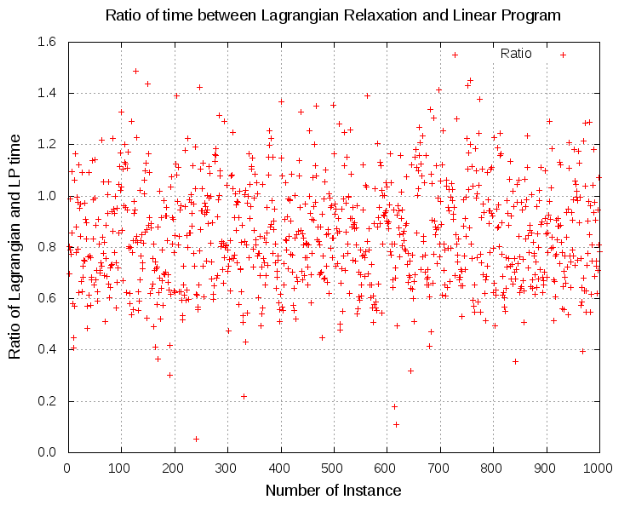

Figure 5.2: Ratio of the time between Lagrangian Relaxation and Linear Program

Based on Figure 5.2 where the ratio between the computational time of the Lagrangian relaxation method and the LP (LP is solved using the dual simplex method) is set along the y-axis, we can say that more than 78% of the instances can be solved faster using the Lagrangian relaxation method than with the dual simplex method.

We observe that it is not clear whether the ratio between the computational time of the Lagrangian relaxation and the LP increases or decreases as a function of the number of points. The reason is that all of the instances we take from the 8x8x8 grid can be solved within less than half of a second using these two methods. See Table A.1 in Appendix A to get an idea of the computational time of the Lagrangian relaxation and the LP, and the changes of the ratio with the increase of the number of points. We present the time of the first 25 instances and the last 25 instances.

To show whether this ratio increases or decreases with the increase of the number of points, we take 100 instances randomly which take more than one second to solve an instance of the point cover problem. We take 5 instances from each Grid 10x10x10, 12x12x12, 15x15x15, 18x18x18, 20x20x20, 22x22x22, 25x25x25, 28x28x28, 30x30x30, 32x32x32, 33x33x33, 34x34x34, 35x35x35, 36x36x36, 37x37x37, 38x38x38, 39x39x39, 40x40x40, 41x41x41, and 42x42x42 where the number of points varies from 386 to 29925. In Table A.2 in Appendix A, we present the computational time to solve these instances using the Lagrangian relaxation method, the computational time to solve the LP using the dual-simplex method, and the ratio between these two computational times.

We graphically plot the ratio of the time of computing lower bound using the Lagrangian relaxation and the LP with the increase of the number of points.

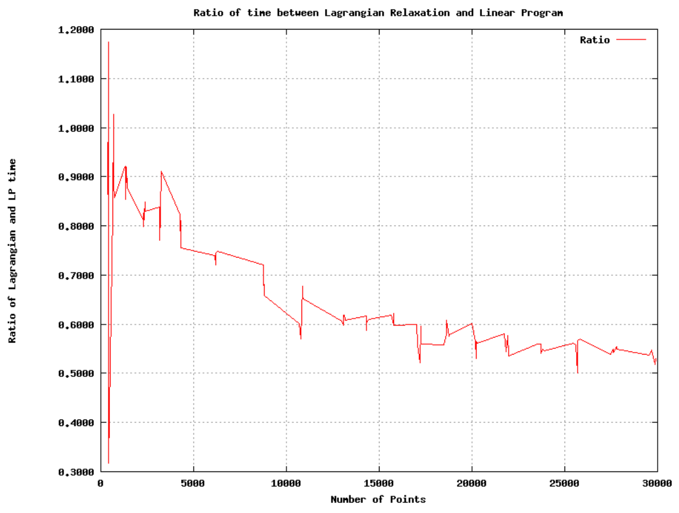

Figure 5.3: Ratio of time between Lagrangian Relaxation and Linear Program

Figure 5.3 shows that the ratio between the computational time of solving an instance using the Lagrangian relaxation method and the dual-simplex method decreases with the increase of the number of points for most of the instances except those small instances which can be solved within a second.

5.2.5 The Upper Bounds

To determine the number of axis-parallel lines required to cover all the points, we used an integer program, the iterative rounding algorithm 2, the iterative lagrangian relaxation algorithm and the branch-and-bound technique using the Lagrangian relaxation. The algorithms are explained in Section 4.3 of Chapter 4. To solve the integer program optimally, we used the ”intlinprog” method of MATLAB 2016b where the dual-simplex algorithm is used as RootLPAlgorithm. Intlinprog seeks the optimal solution among the heuristics method, cut-generation method, and branch-and-bound method.

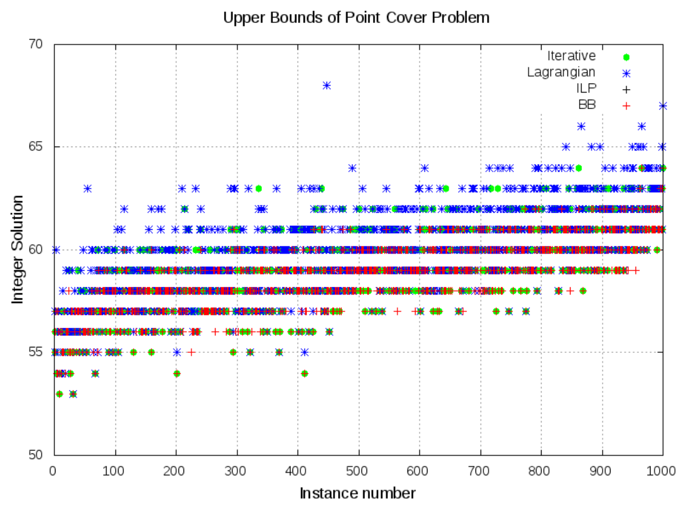

Figure 5.4: Upper Bounds

In Figure 5.4, we graphically represent the upper bound computed by the four algorithms. Here, the instances are sorted in the increasing order of the number of points. Our proposed branch-and-bound algorithm based on the Lagrangian relaxation gives the optimal result as expected. As the IP and the branch-and-bound approach give the same result, so in the figure there is one line for both. It can also be seen that the iterative lagrangian relaxation algorithm gives the weakest upper bound among the four algorithms. The iterative rounding algorithm 2 is much better than the iterative lagrangian relaxation approach because for 663 instances out of 1000 it computes the optimal result.

5.2.6 Time to Compute the Upper Bounds

We notice that the branch-and-bound method using the Lagrangian relaxation approach takes longer time than the rest of the three approaches for every instance. We graphically plot the computational time taken by each method in Figure 5.5. Again we sort the instances according to the increasing order of the number of points. We do not include the plotting time of generating upper bounds using our proposed branch-and-bound algorithm because it takes a relatively large amount of time compared to the other three approaches.

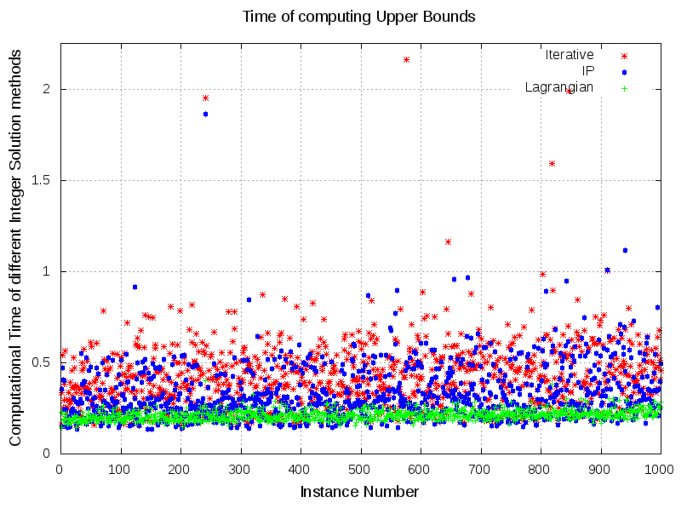

Figure 5.5: Time of Computing Upper Bound

In Figure 5.5, we plot the number of the instance along the x-axis, and the time of different approaches excluding the branch-and-bound method along the y-axis. Here, all times are computed in seconds. We observe that the computation of upper bounds using the iterative lagrangian relaxation algorithm takes the smallest amount of time among the three approaches.

5.2.7 Comparison between Integer Program and Branch-and-Bound

We observe that the integer program and the branch-and-bound technique both give optimal solutions for the point cover problem as expected. We also notice that the computational time for solving an instance using the branch-and-bound technique is much more than the amount of time needed by the IP. Following the same experimental setup described in Section 5.2.1, we use the instances generated with probability 0.4 on different grid sizes starting from 10x10x10 to 100x100x100. While trying to solve an instance on 70x70x70 grid where the total number of points is 137630, our proposed branch-and-bound algorithm computes an optimal solution of value 4900, as the number of axis-parallel lines required to cover those points. On the other hand, IP could not solve this. This inability of the IP to solve an instance optimally repeats for larger instances than this. We can say that the branch-and-bound using the Lagrangian relaxation method can solve some of the large instances optimally which the IP cannot solve.

5.3 Evaluation on Large Instances

In this section, we give a brief description of all the platforms used for evaluating lower bounds and upper bounds by using the Lagrangian relaxation method on large instances.

5.3.1 Experimental Setup

We have used Octave 4.0.2 for conducting all the experiments presented in this section. Octave 4.0.2 was installed on a computer with following specifications:

Processor Model Name: Intel Core i7-4770 CPU

Clock Speed: 3.40GHz

RAM Size: 8.00 GB

System Type: 64-bit Linux Operating System

Environment: CentOS release 6.7

GPU:NVIDIA Corporation GK107GL [Quadro K600]

Accelerators: OpenBLAS, CUDA 8.0 with NVBLAS

5.3.2 Data Set

For performing our simulations on large instances, we generated 25 instances with probability 0.4 on different 3-dimensional grids, varying in size from 10x10x10 to 120x120x120. The minimum number of points in any instance is 400, and the maximum number of points in an instance is 692047.

5.3.3 Ratio of UB and LB using Lagrangian Relaxation as a function of problem size

According to the discussion in Section 5.2.4, we know that among the four approaches for computing the lower bound, the Lagrangian relaxation using the subgradient optimisation technique requires the least computational effort. Therefore, we used the Lagrangian relaxation procedure described in Section 4.3.2 and Section 4.2.1 for computing both the upper bound and lower bound respectively for all the instances of the large data set. The experiments on these large instances were conducted using Octave 4.0.2. We also used one library named NVBLAS to accelerate GNU Octave [1]. NVBLAS can detect all compute-capable GPUs on the system and use them to accelerate basic linear algebra routines. For each instance, we compute the best performance ratio, the worst performance ratio, and the average performance ratio where the performance ratio is defined as UB/LB. In Table 5.1, we report these experimental results along with the range of the number of points for each grid. We can see that the approximation ratio is less or equal to $\frac{3}{2}$ in all the cases.

Table 5.1: Best, Worst and Average Case Ratio of Point Cover Problem

Grid Size	Input Range	Best Ratio	Worst Ratio	Average Ratio
10X10X10	385-440	1.0238	1.1145	1.0607
20X20X20	3103-3287	1.0029	1.2857	1.0794
30X30X30	10612-10943	1.0257	1.2511	1.1226
40X40X40	25472-25838	1.0201	1.2075	1.0964
50X50X50	49417-50359	1.0656	1.2988	1.1204
60X60X60	86009-86775	1.0544	1.1583	1.1001
70X70X70	136653-137901	1.0412	1.2663	1.0893
80X80X80	204271-205355	1.0541	1.1580	1.0907
90X90X90	290932-292520	1.0258	1.1228	1.0833
100X100X100	399038-400876	1.0329	1.1361	1.0805
110X110X110	531458-532836	1.0472	1.1178	1.08418
120X120X120	689632-692047	1.0313	1.0990	1.0663

5.3.4 Computational Time of With and Without GPU Acceleration

In this section, we discuss the importance of using the GPU to speed by the subgradient optimisation method. It is possible to access the computing power of GPU using the BLAS library. So we decide to use NVBLAS along with OpenBLAS. NVBLAS can use all the GPUs on the system to vectorize BLAS routines. So, several matrix calculations become very fast. In the Lagrangian relaxation technique, most of the steps can be vectorized. As a result, use of a GPU accelerates the subgradient optimisation method.

We use 25 instances each on 40x40x40, 50x50x50, 60x60x60, and 70x70x70 grids. The probability of selecting a point is 0.4. We solve these instances with the same initializations, number of iterations for generating solutions for each subproblem, and steps of algorithms using NVBLAS library. We did not use any BLAS accelerator library in the second case. Figure 5.6 shows the time required in both the cases to compute the lower bound and upper bound using the Lagrangian relaxation method solved with the subgradient optimisation method. All the times are repeated in seconds (See Table A.3 in Appendix A for details).

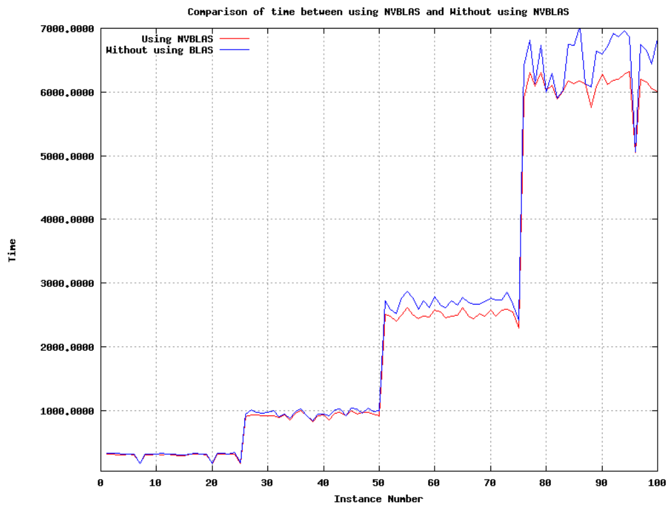

Figure 5.6: Time taken by using NVBLAS and Without using NVBLAS

In Figure 5.6, we plot the number of the instance sorted on the increasing order of the number of points along the x-axis, and the time required to solve the point cover problem with and without NVBLAS along the y-axis. It shows that use of NVBLAS reduces the computational cost. The gap increases with the increase in the input size.

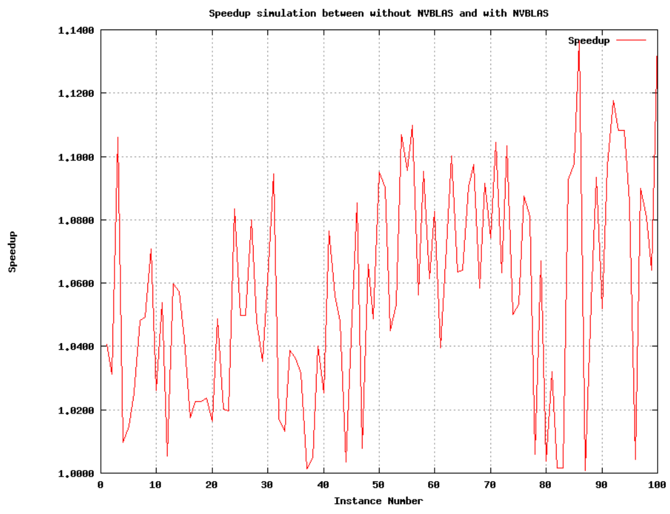

Figure 5.7: Speedup of time by using NVBLAS

In Figure 5.7, we again plot the number of sorted instances along the x-axis and the speedup of all of these instances along the y-axis. We define speedup as the ratio of the time required to solve the point cover problem without using NVBLAS and the time taken by using NVBLAS.

5.4 Approximation Ratio

We compute the approximation ratio for the iterative rounding algorithm 2, the iterative lagrangian relaxation algorithm, and the branch-and-bound method using the Lagrangian relaxation where the LP solution is used as a lower bound on the optimal solution. LP is solved using the dual simplex method.

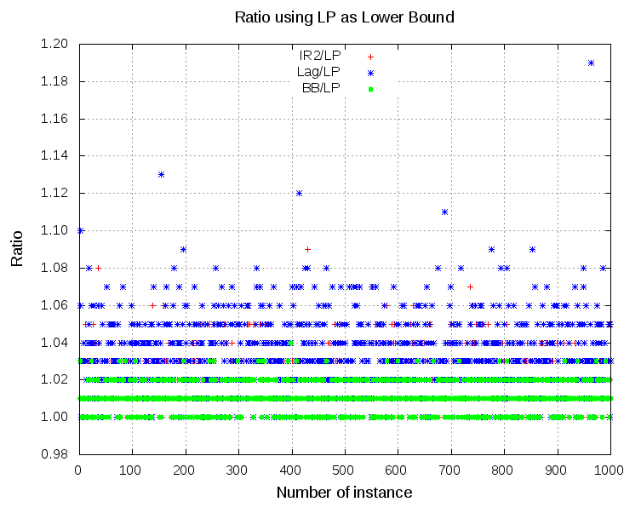

Figure 5.8: Approximation Ratio using LP as Lower Bound

In Figure 5.8, the number of the instance is along the x-axis, and the three performance ratios are along the y-axis. The graph indicates that the ratio in the three cases is no more than $\frac{3}{2}$.

Next we use the solution of the lift-and-project relaxation as the lower bound to compute the ratio. We added some extra inequalities to the LP-relaxation in lift-and-project which tightens the formulation and thereby strengthens the lower bound. We graphically plot in Figure 5.9 the performance ratio between the iterative rounding algorithm2, the iterative lagrangian relaxation algorithm, and the branch-and-bound method where the solution of the lift-and-project method is used as a lower bound on the optimal solution.

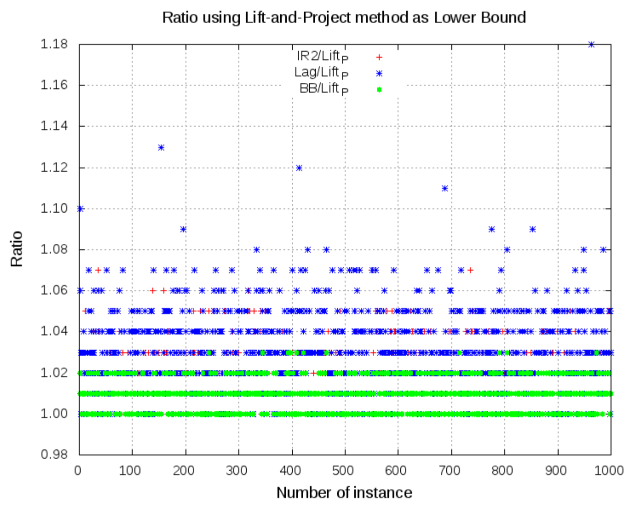

Figure 5.9: Approximation Ratio using Lift-and-Project as Lower Bound

We have also seen that the Lagrangian relaxation can compute very good lower bounds with a reasonable computational time. Figure 5.10 depicts that the subgradient optimisation technique can compute lower bounds that are close to the optimal solution to the LP-relaxation, and our experiments found that there is little or no negative impact on the approximation ratio.

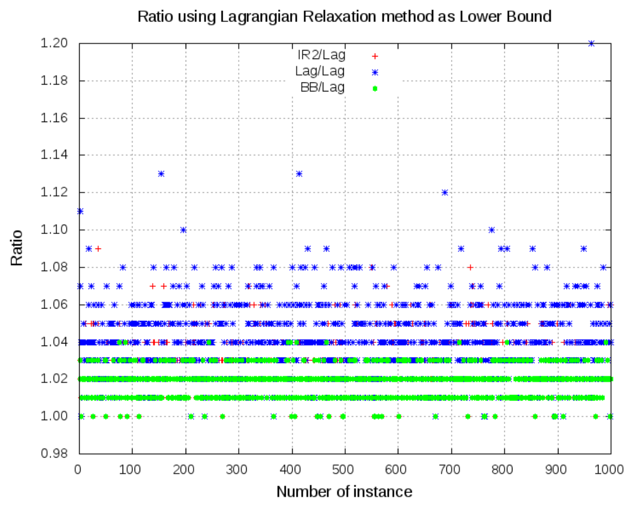

Figure 5.10: Approximation Ratio using Lagrangian Relaxation method as Lower Bound

These findings for approximation ratio are also true for instances generated with probability p, where p is in any value between 0.1 and 1. We test this on instances on a 10x10x10 grid where each point is chosen with probability p. We repeat each experiment 5 times for each value of p. Based on the experimental evaluation, we can say that the approximation ratio for the point cover problem in 3 dimensional space is at most $\frac{3}{2}$.

Chapter 6

Conclusion

6.1 Summary

Given a set of n points in 3D, the goal is to pick the fewest number of lines which are parallel to the axes so that every point lies on one of the lines. This is referred to as the point cover problem in 3D. We studied the approximation ratio for the point cover problem empirically. We have proposed two Lagrangian based algorithms. The iterative Lagrangian relaxation algorithm which gives an integral solution for a point cover instance, has a good performance ratio in practice. The branch-and-bound method gives optimal integral solution. We explored the lift-and-project relaxation which gives marginally better lower bounds than the standard LP. We also computed the lower bound of the point cover instances using the subgradient optimisation method to solve the Lagrangian relaxation problem. The strength of the Lagrangian relaxation approach is that it requires minimum computational time. To solve large instances, we use drop-in accelerators which result in faster computation. Finally we noted that the approximation ratio of the point cover problem in 3D appears to be $\frac{3}{2}$ empirically.

The following tables represent the current state of the point cover problem.

Complexity:

Table 6.1: d-dimensional Point Cover Problem

Dimension	Complexity	Reference
2	Polynomial	[13]
3 or more	NP-complete	[13]

Approximation Ratio for Point Cover Problem in d dimensions:

Table 6.2: Approximation Ratio for d-dimensional Point Cover Problem

Author	Approximation Ratio	Reference
Hassin and Megiddo	d	[13]
Gaur and Bhattacharya	d-1	[9]
Lovasz	$\frac{d}{2}$	[2]

Algorithms for 3D Point Cover Problem:

Table 6.3: Methods and Algorithms explored in this thesis

Method	Purpose	Based On	Running Time
Lagrangian Relaxation	Lower Bound	Subgradient Optimisation approach	$O(n)$
Lift-and-Project	Lower Bound	Lifted LP	$O(n^2)$
Hybrid	Lower Bound	Lift-Project and Lagrangian	$O(n^2)$
Iterative Rounding	Upper Bound	Primal-Dual	$O(n^2)$
Iterative Lagrangian	Upper Bound	Lagrangian	$O(n^2)$
Branch-and-Bound	Upper Bound	Lagrangian	Exponential

6.2 Future Research Work

In this thesis, we implemented the lift-and-project method for level-1 of the hierarchy of relaxations. It is expected that solving the relaxations for larger levels will improve the values of the lower bounds. This improvement may help to obtain better approximation factor.

Rounding schemes based on the solution to the lift and project method need to be explored more.

We examined the approximation ratio experimentally. The theoretical question whether the integrality gap is $\frac{3}{2}$ for the point cover problem in 3D remains open.

The weighted version of the point cover problem need to be explored.

Bibliography

[1] Drop-in acceleration of gnu octave. `https://devblogs.nvidia.com/parallelforall/drop-in-acceleration-gnu-octave/`.

[2] Ron Aharoni, Ron Holzman, and Michael Krivelevich. On a theorem of lovász on covers inr-partite hypergraphs. *Combinatorica*, 16(2):149–174, 1996.

[3] Levitin Anany. Introduction to the design and analysis of algorithms. *Villanova University*, 2003.

[4] Vasek Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of operations research*, 4(3):233–235, 1979.

[5] Thomas H Cormen. *Introduction to algorithms*. MIT press, 2009.

[6] George B Dantzig. Linear programming and extensions. princeton landmarks in mathematics and physics, 1963.

[7] Uriel Feige. A threshold of ln n for approximating set cover. *Journal of the ACM (JACM)*, 45(4):634–652, 1998.

[8] LR Ford and DR Fulkerson. Flows in networks. 1962. *Princeton U. Press, Princeton, NJ*, 1962.

[9] Daya Ram Gaur and Binay Bhattacharya. Covering points by axis parallel lines. In *Proc. 23rd European Workshop on Computational Geometry*, pages 42–45. Citeseer, 2007.

[10] Daya Ram Gaur, Toshihide Ibaraki, and Ramesh Krishnamurti. Constant ratio approximation algorithms for the rectangle stabbing problem and the rectilinear partitioning problem. *Journal of Algorithms*, 43(1):138–152, 2002.

[11] Arthur M Geoffrion. Lagrangean relaxation for integer programming. In *Approaches to integer programming*, pages 82–114. Springer, 1974.

[12] Venkatesan Guruswami and Rishi Saket. On the inapproximability of vertex cover on k-partite k-uniform hypergraphs. *Automata, Languages and Programming*, pages 360–371, 2010.

[13] Refael Hassin and Nimrod Megiddo. Approximation algorithms for hitting objects with straight lines. *Discrete Applied Mathematics*, 30(1):29–42, 1991.

[14] Michael Held and Richard M Karp. The traveling-salesman problem and minimum spanning trees. *Operations Research*, 18(6):1138–1162, 1970.

[15] Michael Held and Richard M Karp. The traveling-salesman problem and minimum spanning trees: Part ii. *Mathematical programming*, 1(1):6–25, 1971.

[16] Dorit S Hochbaum. Approximation algorithms for the set covering and vertex cover problems. *SIAM Journal on computing*, 11(3):555–556, 1982.

[17] Kawsar Jahan. Covering points with axis parallel lines. Master's thesis, UNIVERSITY OF LETHBRIDGE (CANADA), 2015.

[18] David S Johnson. Approximation algorithms for combinatorial problems. *Journal of computer and system sciences*, 9(3):256–278, 1974.

[19] Narendra Karmarkar. A new polynomial-time algorithm for linear programming. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 302–311. ACM, 1984.

[20] Leonid G Khachiyan. A polynomial algorithm in linear programming. In *Doklady Akademiia Nauk SSSR*, volume 244, pages 1093–1096, 1979.

[21] Victor Klee and George J Minty. How good is the simplex algorithm. Technical report, DTIC Document, 1970.

[22] Harold W Kuhn. The hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955.

[23] László Lovász and Alexander Schrijver. Cones of matrices and set-functions and 0–1 optimization. *SIAM Journal on Optimization*, 1(2):166–190, 1991.

[24] Alexander Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, 1998.

[25] Hanif D Sherali and Warren P Adams. A hierarchy of relaxations between the continuous and convex hull representations for zero-one programming problems. *SIAM Journal on Discrete Mathematics*, 3(3):411–430, 1990.

[26] Daniel A Spielman and Shang-Hua Teng. Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time. *Journal of the ACM (JACM)*, 51(3):385–463, 2004.

[27] James K Strayer. *Linear programming and its applications*. Springer Science & Business Media, 2012.

[28] Luca Trevisan. Combinatorial optimization: Exact and approximate algorithms. *Standford University*, 2011.

[29] Robert J Vanderbei. *Linear programming*. Springer, 2015.

[30] Vijay V Vazirani. *Approximation algorithms*. Springer Science & Business Media, 2013.

[31] David P Williamson and David B Shmoys. *The design of approximation algorithms*. Cambridge university press, 2011.

Appendix A

Results of the Experimentation

A.1 Computational Time of the LP and the Lagrangian Relaxation Method

Table A.1: Computational Time of LP and Lagrangian Relaxation Method

Instance No.	No. of Points	LP	Lagrangian	Ratio=Lag/LP
1	171	0.16266152	0.13087585	0.8045901083
2	173	0.18945079	0.13235212	0.6986094912
3	173	0.17557789	0.17349274	0.9881240742
4	173	0.19734759	0.15602836	0.7906271366
5	175	0.219025	0.16949678	0.7738695583
6	175	0.18565539	0.15899286	0.8563869867
7	176	0.14877607	0.16275638	1.0939688083
8	177	0.22814134	0.13256918	0.5810835511
9	178	0.30354368	0.13634082	0.4491637579
10	178	0.33564665	0.13651643	0.4067266275
11	178	0.15356293	0.16305206	1.0617931033
12	179	0.29293917	0.16723816	0.5708972276
13	179	0.15467814	0.18019854	1.1649903471
14	179	0.15412524	0.1354589	0.8788884935
15	179	0.1984628	0.15513363	0.7816761126
16	180	0.15618432	0.15535923	0.9947172034
17	180	0.18990644	0.15046042	0.7922870862
18	180	0.18006437	0.17551337	0.974725705
19	180	0.23036042	0.1439422	0.6248564749
20	181	0.15057237	0.16881697	1.1211683126
21	181	0.17282533	0.16488425	0.9540514113
22	181	0.16117671	0.13575843	0.8422955773
23	181	0.16932416	0.16388654	0.9678863312
24	182	0.20086568	0.13386	0.6664154872
25	182	0.22778285	0.14308507	0.6281643679
976	228	0.23729931	0.17330816	0.7303357098
977	228	0.19275557	0.17402258	0.9028147928
978	228	0.22996561	0.17830311	0.7753468443

Table A.1: Computational Time of LP and Lagrangian Relaxation Method

Instance No.	No. of Points	LP	Lagrangian	Ratio=Lag/LP
979	228	0.193417	0.18915271	0.9779528687
980	228	0.18944882	0.24398352	1.287859803
981	228	0.21168938	0.17957343	0.8482873822
982	229	0.3671016	0.20145636	0.5487754889
983	229	0.20757207	0.1995511	0.9613581442
984	230	0.26999122	0.16637035	0.6162065196
985	230	0.28523524	0.21347671	0.7484233365
986	230	0.20817155	0.16829825	0.8084594172
987	231	0.20141405	0.20967773	1.0410283195
988	231	0.18573401	0.21962448	1.1824677667
989	231	0.17634273	0.16489621	0.935089357
990	233	0.2454459	0.16879561	0.6877100412
991	233	0.18848828	0.18398385	0.9761023338
992	233	0.21294474	0.19302262	0.9064446485
993	234	0.26535969	0.19284359	0.7267252611
994	235	0.23859953	0.17814373	0.7466223006
995	235	0.27033732	0.16775218	0.6205291227
996	236	0.27994845	0.19883411	0.710252584
997	239	0.20230793	0.19106908	0.9444468143
998	239	0.19532611	0.2096068	1.0731120381
999	240	0.27126818	0.21933948	0.8085706182
1000	240	0.24292855	0.19010042	0.7825363466

A.2 Computational Time of the LP and the Lagrangian Relaxation Method with the increase of the number of points

Table A.2: Computational Time of LP and Lagrangian Relaxation Method with the increase of points

Instance No.	No. of Points	Lagrangian	LP	Ratio=Lag/LP
1	386	0.33785115	0.34849009	0.969471327
2	397	0.36587883	0.48452485	0.755129133
3	399	0.44818778	0.43617641	1.027537872
4	401	0.45655998	0.38862784	1.174799983
5	419	0.4869202	1.53173928	0.317887128
6	662	0.54802392	0.65892585	0.831692853
7	696	0.61472334	0.5987096	1.026747091
8	699	0.63203603	0.66105415	0.956103263
9	701	0.56047369	0.63562607	0.881766366
10	730	0.58237113	0.67877617	0.857972268
11	1307	1.21790404	1.32374437	0.920044736

Table A.2: Computational Time of LP and Lagrangian Relaxation Method with the increase of points

Instance No.	No. of Points	Lagrangian	LP	Ratio=Lag/LP
12	1330	1.2789958	1.38746862	0.921819623
13	1333	1.19296563	1.39775932	0.853484297
14	1372	1.2721913	1.3840777	0.919161764
15	1408	1.24658974	1.42124241	0.877112681
16	2284	2.1818524	2.68708642	0.811977011
17	2305	2.21626926	2.68514612	0.82538125
18	2313	2.22260801	2.78444602	0.798222696
19	2372	2.26573804	2.66945028	0.848765777
20	2390	2.28516236	2.75428277	0.829676018
21	3173	3.40820043	4.06832738	0.837739963
22	3188	3.34946899	4.34104039	0.771582084
23	3197	3.55552617	4.3402653	0.819195585
24	3201	3.34298624	4.02689794	0.830164134
25	3247	3.68927966	4.04943802	0.911059668
26	4273	4.89308591	5.94179327	0.823503223
27	4288	4.92618502	6.13081677	0.803512029
28	4304	4.70542031	6.08026098	0.773884596
29	4308	4.88691892	6.02299428	0.811376982
30	4314	4.85709366	6.42787386	0.755629897
31	6159	7.62903308	10.3235803	0.738991015
32	6205	7.54185223	10.47762721	0.71980536
33	6214	7.62991627	10.40719033	0.733138919
34	6214	7.71385361	10.36604889	0.74414598
35	6272	7.65722612	10.23415493	0.748203068
36	8745	12.20573628	16.92797021	0.721039565
37	8761	12.1216682	17.8508989	0.679050857
38	8769	11.98342267	17.63201595	0.679639963
39	8781	12.01163793	17.07492977	0.703466315
40	8801	12.15930336	18.46267997	0.65858821
41	10697	15.82711457	26.3240157	0.601242407
42	10802	15.89251976	27.84254123	0.5707999
43	10863	15.85024558	23.41006809	0.677069606
44	10893	15.79860577	24.04419311	0.657065334
45	10903	15.60908149	23.94577042	0.651851296
46	12993	20.28454365	33.48768504	0.605731439
47	13062	20.51137323	34.32085209	0.597635897
48	13087	20.37272605	32.9641855	0.618026071
49	13118	20.48937879	33.09644051	0.619081039
50	13185	20.69314017	34.03552993	0.607986425
51	14307	23.39270541	37.95632733	0.616305819
52	14319	23.95968278	40.84299628	0.58662892

Table A.2: Computational Time of LP and Lagrangian Relaxation Method with the increase of points

Instance No.	No. of Points	Lagrangian	LP	Ratio=Lag/LP
53	14329	23.59556394	39.13663082	0.60290228
54	14382	24.02108945	39.55267149	0.607319014
55	14431	23.81850566	39.0258058	0.610327069
56	15622	26.50307367	42.7973987	0.619268331
57	15760	26.64119998	44.61708323	0.597107611
58	15770	26.62669498	42.80439034	0.622055232
59	15778	26.73293311	44.62121079	0.599108196
60	15846	26.65286996	44.5885881	0.597750929
61	17024	30.02956462	50.12350168	0.599111467
62	17054	29.87829762	53.62160133	0.557206366
63	17179	30.25664587	58.05551698	0.521167452
64	17236	30.2630611	50.75369013	0.596273119
65	17266	30.18518607	53.89892004	0.560033226
66	18499	33.67063855	60.29505654	0.558431163
67	18612	33.88561442	58.5517242	0.578729574
68	18620	33.89835731	55.67496865	0.608861722
69	18748	34.21632609	59.40765223	0.575958228
70	18821	34.59865747	59.8214035	0.57836586
71	20020	37.9010492	63.01133627	0.601495722
72	20183	38.34088529	67.87788657	0.5648509
73	20223	38.24127519	72.20160146	0.529645803
74	20230	38.36561477	67.97431426	0.564413414
75	20284	38.43031066	68.45081985	0.561429516
76	21737	42.93217383	73.96099839	0.580470447
77	21831	42.58458146	78.16176836	0.544826228
78	21933	43.26736727	75.27733879	0.574772806
79	21935	43.04828777	74.55681742	0.57738902
80	21977	43.14683479	80.52764676	0.535801511
81	23534	48.5892974	86.82735129	0.559608196
82	23700	49.97738643	89.35590175	0.559307057
83	23720	50.15979901	92.92978488	0.539760197
84	23787	49.22913392	89.73833782	0.548585311
85	23908	49.98813903	91.70363383	0.545105324
86	25463	59.22364418	105.4835207	0.561449255
87	25603	57.81003775	103.5810209	0.558114192
88	25679	58.58381535	117.0920523	0.500322731
89	25695	58.03815473	102.4704988	0.566388916
90	25814	58.52383368	102.8077501	0.569255077
91	27473	62.85989792	116.5279916	0.539440327
92	27620	63.28233459	115.3572699	0.548576909
93	27636	63.29936356	116.6784544	0.542511159

Table A.2: Computational Time of LP and Lagrangian Relaxation Method with the increase of points

Instance No.	No. of Points	Lagrangian	LP	Ratio=Lag/LP
94	27800	63.67780197	114.6823171	0.555253884
95	27808	63.55710925	115.8313413	0.548703905
96	29551	70.39958297	131.0772785	0.537084564
97	29660	71.59167096	131.1743301	0.545775007
98	29681	71.14310546	129.7992471	0.548101064
99	29874	71.17153735	137.4333704	0.517862126
100	29925	71.4143264	134.6768748	0.530264209

A.3 Computational Time of With and Without GPU Acceleration

Table A.3: Computational Time of With and Without GPU Acceleration

Instance No.	No. of Points	Time using GPU	Time without GPU
1	25461	318.04425597	330.991467
2	25472	314.314816	324.092417
3	25512	300.13999796	331.95254803
4	25528	309.38550496	312.34453297
5	25546	314.49518991	319.19082499
6	25548	307.60876989	315.36793709
7	25562	168.66989493	176.81585813
8	25566	297.97137594	312.61812711
9	25579	302.40286589	323.84906697
10	25585	312.22351003	320.37393999
11	25591	309.90674806	326.58080602
12	25594	319.43344188	321.15510082
13	25599	301.03564501	319.01747394
14	25609	293.16456294	309.97395897
15	25610	289.21303797	301.48260117
16	25619	318.25178909	323.83185911
17	25624	318.92374086	326.16241002
18	25642	314.53492022	321.60654998
19	25654	305.73132205	312.99264789
20	25659	169.49292111	172.28679681
21	25672	313.45464206	328.70869088
22	25706	318.65650606	325.10213614
23	25743	315.19987702	321.361727
24	25773	321.44861412	348.25937891
25	25838	171.15226507	179.67805219
26	49417	899.76188087	944.642205
27	49717	937.169168	1012.05268502
28	49740	926.48536897	970.76637602

Table A.3: Computational Time of With and Without GPU Acceleration

Instance No.	No. of Points	Time using GPU	Time without GPU
29	49754	923.63528991	956.14299893
30	49872	916.30364299	972.34356093
31	49875	915.61665392	1002.07415414
32	49926	891.53042006	906.86642504
33	49953	935.60552001	947.94741607
34	49967	850.32315207	883.24571681
35	49971	957.57573104	992.07419491
36	49999	998.34050798	1029.80877209
37	50022	914.42648697	915.76118302
38	50069	828.35769081	832.47390485
39	50070	912.23548293	948.79875302
40	50071	928.36034989	951.8848722
41	50085	847.68281198	912.41999316
42	50105	953.19365692	1006.62699699
43	50111	977.56206298	1024.366781
44	50113	918.35353088	921.55054903
45	50149	996.38089609	1039.1573329
46	50184	945.38641906	1026.11563706
47	50218	962.171839	969.58551502
48	50229	973.42926598	1037.68111014
49	50241	940.93364692	986.68950701
50	50359	915.2476449	1002.25708008
51	86009	2503.54798007	2729.35510588
52	86032	2478.11793685	2589.2290771
53	86112	2399.87756395	2527.68258095
54	86203	2502.37859392	2769.5541532
55	86214	2622.03546596	2872.83082819
56	86221	2504.6092031	2779.54746103
57	86231	2448.01519108	2585.71871591
58	86263	2481.92986917	2718.35880208
59	86359	2463.49725199	2614.92249918
60	86406	2572.06392789	2784.81406403
61	86409	2555.0643189	2656.06323695
62	86412	2453.451401	2617.01851797
63	86432	2480.48099804	2729.13641119
64	86448	2496.15551591	2654.80474806
65	86452	2610.06334305	2777.36864495
66	86489	2476.51195383	2700.37539887
67	86495	2436.04026103	2673.46112299
68	86588	2525.09114504	2672.61701393
69	86594	2482.98562908	2710.029145
70	86620	2571.21352696	2762.24724007

Table A.3: Computational Time of With and Without GPU Acceleration

Instance No.	No. of Points	Time using GPU	Time without GPU
71	86620	2482.57678294	2741.89629292
72	86642	2572.98255801	2735.92816496
73	86649	2594.49782491	2862.74506807
74	86775	2546.45425105	2674.06930113
75	87007	2296.59162211	2419.20680904
76	136653	5902.26078892	6418.43068886
77	136862	6305.93570709	6816.59370708
78	136876	6090.13843513	6125.56376815
79	136971	6310.34727216	6732.56026101
80	137007	6001.10991001	6023.22960806
81	137037	6102.83052993	6297.74626994
82	137050	5891.550349	5900.66991091
83	137065	6015.3551662	6025.57502198
84	137081	6172.67253613	6743.78702903
85	137146	6136.72223806	6736.96398187
86	137220	6172.44345784	7014.98454094
87	137237	6129.73195887	6134.55212092
88	137268	5763.63133788	6073.32990718
89	137285	6075.69588804	6642.50709891
90	137290	6278.99620414	6605.55852795
91	137296	6116.40306902	6709.67184997
92	137297	6190.68700194	6919.01981187
93	137308	6197.86958385	6868.31784606
94	137351	6275.41291499	6954.7600131
95	137416	6324.27853703	6864.3508749
96	137526	5043.57970595	5065.14795899
97	137590	6192.6621089	6750.16857195
98	137600	6153.42388296	6650.48366308
99	137630	6056.20077395	6443.56807709
100	137901	6015.29155087	6839.995754