Sabbir, Tarikul Alam Khan

2011

Topology sensitive algorithms for large scale uncapacitated covering problem

Department of Mathematics and Computer Science

https://hdl.handle.net/10133/3235

Downloaded from OPUS, University of Lethbridge Research Repository
TOPOLOGY SENSITIVE ALGORITHMS FOR LARGE SCALE UNCAPACITATED COVERING PROBLEM

Tarikul Alam Khan Sabbir
Bachelor of Science, Islamic University of Technology, 2006

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

© Tarikul Alam Khan Sabbir, 2011
TOPOLOGY SENSITIVE ALGORITHMS FOR LARGE SCALE UNCAPACITATED COVERING PROBLEM

TARIKUL ALAM KHAN SABBIR

Approved:

Signature          Date

Supervisor:  

Co-Supervisor:  

Committee Member:  

External Examiner:  

Chair, Thesis Examination Committee:  
I dedicate this thesis to my parents.
Abstract

Solving NP-hard facility location problems in wireless network planning is a common scenario. In our research, we study the Covering problem, a well known facility location problem with applications in wireless network deployment. We focus on networks with a sparse structure. First, we analyzed two heuristics of building Tree Decomposition based on vertex separator and perfect elimination order. We extended the vertex separator heuristic to improve its time performance. Second, we propose a dynamic programming algorithm based on the Tree Decomposition to solve the Covering problem optimally on the network. We developed several heuristic techniques to speed up the algorithm. Experiment results show that one variant of the dynamic programming algorithm surpasses the performance of the state of the art mathematical optimization commercial software on several occasions.
Acknowledgments

At first, I would like to thank the Almighty for giving me the strength to finish my thesis. I am forever grateful to my loving family. The continuous word of encouragement from my parents, my brother and my sister-in-law always boosted my confidence and guided me through difficult times.

I would like to take this opportunity to thank all my friends and well-wishers near and abroad who had faith in me even when I did not have faith in myself. I feel lucky to have befriended Chad, Chris, Mecole and Ben, whom I met in Lethbridge. They always made me feel at home in a totally foreign environment. I would also like to thank my fellow grad students Salimur Choudhury, Mohammad Tauhidul Islam, Sangita Bhattacharjee, Kaisar Imam, Shah Mostafa Khaled, Mahmudul Hasan for their continuous support.

I would like to express my deep gratitude for both of my Supervisors. I specially thank Dr. Daya Gaur to give me the opportunity to be his student. His watchful supervision, admirable suggestions and astute guidance throughout the period was essential to complete my thesis. I profoundly thank Dr. Robert Benkoczi for being an outstanding mentor. Without his endless help and continuous reassurance at the most difficult of times, this thesis wouldn’t have been a reality. I am indebted to both of them for their unhindered support and the invaluable knowledge they shared with me.

I would like to thank my M.Sc. supervisory committee member Dr. Saurya Das for his constructive suggestions. I would also like to thank my external Dr. Apurva Mudgal for his valuable suggestions and comments. And I must thank Dr. Shahadat Hossain for his continuous advice and help.

I also would like to thank the School of Graduate Studies and the department of Math and Computer Science. Thank you all for supporting me throughout the years and adding to my strength as I continue on the road ahead.
**Contents**

<table>
<thead>
<tr>
<th>Approval/Signature Page</th>
<th>ii</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dedication</td>
<td>iii</td>
</tr>
<tr>
<td>Abstract</td>
<td>iv</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>v</td>
</tr>
<tr>
<td>Table of Contents</td>
<td>vi</td>
</tr>
<tr>
<td>List of Tables</td>
<td>viii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>ix</td>
</tr>
</tbody>
</table>

1 **Introduction**  
1.1 Facility Location Covering Problem ............................................. 2  
1.2 Motivation for Using Tree Decomposition ........................................... 3  
1.3 Summary of Contributions ................................................................. 6  
1.4 Thesis Outline ................................................................................ 7

2 **Background on Tree Decomposition**  
2.1 Definitions and Notations ................................................................. 9  
2.1.1 Nice Tree Decomposition ................................................................. 10  
2.2 Construction of a Tree Decomposition .................................................. 11  
2.3 Minimum Separating Vertex Set Heuristic ................................................ 12  
2.3.1 Minimum Separating Vertex sets and their Computation .......................... 12  
2.3.2 MSVS Heuristic ....................................................................................... 14  
2.3.3 Random Separator Vertex set Heuristic (RSVS) ...................................... 17  
2.4 Clique Tree Heuristic ................................................................................ 18  
2.4.1 Chordal Graph: Definitions and Characteristics ..................................... 18  
2.4.2 Minimum Degree Heuristic ....................................................................... 20  
2.4.3 Clique Tree Algorithm ........................................................................... 21

3 **Dynamic Programming Algorithm to Solve Covering Problem**  
3.1 Dynamic Programming Approach ............................................................... 26  
3.1.1 Cost function and other Definitions ...................................................... 27  
3.1.2 Leaf Node .................................................................................................. 28  
3.1.3 Introduce node .......................................................................................... 29  
3.1.4 Forget Node .............................................................................................. 30  
3.1.5 Join Node .................................................................................................. 31  
3.1.6 Running Time ............................................................................................ 32  
3.1.7 Proof of Correctness ................................................................................ 33  
3.2 Heuristic Techniques ................................................................................... 35  
3.2.1 Pruning Heuristic ..................................................................................... 35
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2.2 Reductions of cost functions based on the Covering Neighborhood</td>
<td>36</td>
</tr>
<tr>
<td>3.2.3 Pruning Using Branch and Bound</td>
<td>39</td>
</tr>
<tr>
<td>3.3 Hybrid and Parallel Algorithm</td>
<td>43</td>
</tr>
<tr>
<td>3.3.1 A Hybrid algorithm with CPLEX and Dynamic Programming</td>
<td>43</td>
</tr>
<tr>
<td>3.3.2 A Parallel Algorithm for Solving Covering Problem</td>
<td>46</td>
</tr>
<tr>
<td>3.4 Implementation</td>
<td>48</td>
</tr>
<tr>
<td>3.4.1 Dynamic Program</td>
<td>48</td>
</tr>
<tr>
<td>3.4.2 Pruning Heuristic Module</td>
<td>50</td>
</tr>
<tr>
<td>3.4.3 Branch and Bound Technique</td>
<td>51</td>
</tr>
<tr>
<td>3.4.4 Hybrid Algorithm</td>
<td>51</td>
</tr>
<tr>
<td>3.4.5 Parallel Algorithm</td>
<td>52</td>
</tr>
<tr>
<td>3.4.6 Cost Table Reduction Technique</td>
<td>54</td>
</tr>
<tr>
<td>3.4.7 Bounding the Assignment Function</td>
<td>55</td>
</tr>
<tr>
<td>3.4.8 Balancing the Height of the Tree Decomposition</td>
<td>58</td>
</tr>
<tr>
<td>3.4.9 Using a Modified Dijkstra’s algorithm to Compute Shortest Path</td>
<td>60</td>
</tr>
<tr>
<td>4   Experiments</td>
<td>61</td>
</tr>
<tr>
<td>4.1 Tree Decomposition Experiments</td>
<td>61</td>
</tr>
<tr>
<td>4.1.1 Data Sets</td>
<td>62</td>
</tr>
<tr>
<td>4.1.2 Tree Decomposition Experiment Results</td>
<td>63</td>
</tr>
<tr>
<td>4.2 Covering Problem Algorithm Experiments</td>
<td>65</td>
</tr>
<tr>
<td>4.2.1 Data Sets</td>
<td>66</td>
</tr>
<tr>
<td>4.2.2 Experiment Results</td>
<td>67</td>
</tr>
<tr>
<td>4.3 Analysis</td>
<td>81</td>
</tr>
<tr>
<td>5   Conclusion</td>
<td>85</td>
</tr>
<tr>
<td>Bibliography</td>
<td>88</td>
</tr>
</tbody>
</table>
## List of Tables

4.1 Tree Decomposition Experiments on Random Dense Graphs ........................................ 64  
4.2 Tree Decomposition Experiments on Random Sparse Graphs ..................................... 64  
4.3 Runtime Comparison among CPLEX, Hybrid Program and Hybrid with BB(Branch and Bound) for a Covering problem Instance where radius = 10, facility opening cost = 15 and penalty = 20 .................................................. 68  
4.4 Permutations comparison between Hybrid Program and Hybrid with BB(Branch and Bound) for a Covering problem Instance where radius = 10, facility opening cost = 15 and penalty = 20 .................................................. 70  
4.5 Runtime Comparison among CPLEX, Hybrid Program and Hybrid with BB(Branch and Bound) for a Covering problem Instance where radius = 20, facility opening cost = 25 and penalty = 20 .................................................. 72  
4.6 Number of Permutations comparison between Hybrid Program and Hybrid with BB(Branch and Bound) for a Covering problem Instance where radius = 20, facility opening cost = 25 and penalty = 20 .................................................. 73  
4.7 Runtime Comparison among CPLEX, Hybrid Program and Hybrid with BB(Branch and Bound) for a Covering problem Instance where radius = 30, facility opening cost = 13 and penalty = 22 .................................................. 75  
4.8 Number of Permutations comparison between Hybrid Program and Hybrid with BB(Branch and Bound) for a Covering problem Instance where radius = 30, facility opening cost = 13 and penalty = 22 .................................................. 76  
4.9 Runtime Comparison among CPLEX, Hybrid Program and Hybrid with BB(Branch and Bound) for a Covering problem Instance where radius = 40, facility opening cost = 10 and penalty = 17 .................................................. 78  
4.10 Number of Permutations comparison between Hybrid Program and Hybrid with BB(Branch and Bound) for a Covering problem Instance where radius = 40, facility opening cost = 10 and penalty = 17 .................................................. 79
List of Figures

1.1 An Example of a Covering Problem in a Wireless sensor network .................. 1
1.2 An Example of a Covering Problem in a Wireless sensor network ................. 5

2.1 A graph and it’s tree decomposition ............................................... 10
2.2 Nice Tree Decomposition ............................................................. 11
2.3 Transformation of a non directed graph to its auxiliary equivalent .............. 13
2.4 Improvement step of a tree decomposition [16] .................................. 15
2.5 Construction of Tree Decomposition by MSVS .................................. 15
2.6 Step by Step execution of Clique Tree Heuristic ................................ 24

3.1 Different types of Nice Tree Decomposition nodes with their facility allocation . 29
   3.1.1 Leaf Node .............................................................................. 29
   3.1.2 Introduce Node ....................................................................... 29
   3.1.3 Forget Node ........................................................................... 29
   3.1.4 Join Node .............................................................................. 29
3.2 Subtree $G_T$ and $G \setminus G_T$ at node $i$ ....................................... 34
3.3 Tree Decomposition with Bounds ..................................................... 41
3.4 Solving a Tree Decomposition with the Hybrid Algorithm ...................... 44
3.5 Parent and child with common elements ............................................. 56
3.6 distances representation between clients and facilities ........................... 57
3.7 Balancing the Height of a Tree Decomposition .................................... 59

4.1 Runtime Comparison among CPLEX, Hybrid and Hybrid_with_bb for client radius = 10, facility opening cost = 15 penalty = 20 ................................. 71
4.2 Permutation Comparison between Hybrid and Hybrid_with_bb for client radius = 10, facility opening cost = 15 penalty = 20 ........................................ 71
4.3 Runtime Comparison among CPLEX, Hybrid and Hybrid_with_bb for client radius = 20, facility opening cost = 25 penalty = 20 ................................. 74
4.4 Permutation Comparison between Hybrid and Hybrid_with_bb for client radius = 20, facility opening cost = 25 penalty = 20 ........................................ 74
4.5 Runtime Comparison among CPLEX, Hybrid and Hybrid_with_bb for client radius = 30, facility opening cost = 13 penalty = 22 ................................. 77
4.6 Permutation Comparison between Hybrid and Hybrid_with_bb for client radius = 30, facility opening cost = 13 penalty = 22 ........................................ 77
4.7 Runtime Comparison among CPLEX, Hybrid and Hybrid_with_bb for client radius = 40, facility opening cost = 10 penalty = 17 ................................. 80
4.8 Permutation Comparison between Hybrid and Hybrid_with_bb for client radius = 40, facility opening cost = 10 penalty = 17 ........................................ 80
4.9 Client Radius Vs Runtime data of the Hybrid program for the graph with 3330 nodes with subproblem size 2000 .................................................. 82
4.10 Client Radius Vs Total number of permutations of Hybrid Program for the graph with 3330 nodes with subproblem size 2000 ................................. 83
Chapter 1

Introduction

In Wireless Network Planning, many scenarios involves solving NP-hard problems like Dominating Set, Facility Location Covering problem or P-median problem. For example, in wireless sensor networks, often router motes are needed to be placed to collect data from the data collection sensors which is a Facility location Covering problem. The objective in this case is to minimize the total number of routers(thus minimizing total cost) while maximizing the sensor Coverage. In Figure 1.1, a superficial instance of a Covering problem in wireless sensor network is shown. In our research we will solve the Facility Location Covering Problem using a graph theoretical concept called Tree Decompositions.

Figure 1.1: An Example of a Covering Problem in a Wireless sensor network

In their study of graph minors, Robertson and Seymour defined the graph structures path width [26], treewidth [27], and branchwidth [25, 28] with their associated graph structures path decomposition,
tree decomposition and branch decomposition. These notions proved to be useful in many areas of computational complexity theory. Many NP-hard graph problems can be solved in polynomial time for graphs with pathwidth, treewidth or branchwidth bounded by a constant. In this thesis we will employ a dynamic programming algorithm which uses the tree decomposition to solve the Facility location Covering Problem.

In section 1.1 we will describe a formal definition of the Facility Location Covering Problem. In Section 1.2, we will discuss the motivation behind using tree decomposition for our problem. In section 1.3, we will provide a outline of our thesis.

1.1 Facility Location Covering Problem

Following we will describe the client constrained Covering problem as defined by Kolen and Tamir [15].

Minimise \[ \sum_{j=1}^{n} c_{j}y_{j} + \sum_{i=1}^{m} b_{i}z_{i} \]
subject to \[ \sum_{j=1}^{n} a_{ij}y_{j} + z_{i} \geq 1, \quad i = 1, \ldots, m \]
\[ y_{j} \in \{0, 1\}, \quad j = 1, \ldots, n \]
\[ z_{i} \in \{0, 1\}, \quad i = 1, \ldots, m \]

(1.1)

Where, \( m \) and \( n \) are finite index sets and \( a_{ij} \) is an \( |n| \times |m| \) \((0,1)\) matrix. Let, \( G = (V,E) \) be an undirected network with node set \( V = \{v_1, v_2 \ldots v_m\} \) and edge set \( E \). We define \( d(v_i, v_j) \) is the shortest path distance between client \( v_i \) and \( v_j \). We assume that at each node there exists exactly one client. We refer to the client located at node \( v_i \) as client \( i, i = 1, 2, \ldots, m \). If client \( i \) is not served by any facility, then a non-negative penalty cost of \( b_i \) is incurred. We assume that the set of potential sites for the facilities is a subset of nodes of the network. Without loss of generality let this be \( v' = v_1, v_2, \ldots, v_n, n \leq m. \) The non negative setup cost of establishing a facility at \( v_j \) is \( c_{j}, \quad j = 1, 2, \ldots, n \) (we will refer to a facility established at site \( v_j \) as facility \( j, j = 1, 2, \ldots, n \)). The client constrained
covering problem corresponds to the case where we have a region of attraction of radius $r_i$ for client $i$ and we set $a_{ij} = 1$ if and only if $d(v_i, v_j) \leq r_i$, $i = 1, 2, \ldots m$, $j = 1, 2, \ldots n$, which means for every facility opened at vertex $i$, $i = 1, 2, \ldots m$, every client $j$, $j = 1, 2, \ldots n$ is either covered or uncovered, in the later case a penalty is incurred.

We define the LP-relaxation of (1.1) by replacing the integrality constraints by the non negativity constraints on the variables. The relaxation of (1.1) is given below-

$$\begin{align*}
\text{Minimise} & \quad \sum_{j=1}^{n} c_j y_j + \sum_{i=1}^{m} b_i z_i \\
\text{subject to} & \quad \sum_{j=1}^{n} a_{ij} y_j + z_i \geq 1, \quad i = 1, \ldots m \\
& \quad 0 \leq y_j \leq 1, \quad j = 1, \ldots n \\
& \quad 0 \leq z_i \leq 1, \quad i = 1, \ldots m
\end{align*}$$

(1.2)

In our research, we will model a CPLEX (a mathematical problem solver tool) instance according to this linear program to solve subproblems of an input graph.

### 1.2 Motivation for Using Tree Decomposition

In many wireless networks, the graph representation of the flow of data has a decisive connectedness that resembles a tree like pattern. The current methods for solving optimization problems such as integer programming doesn’t take this property into account. In our research we wanted to exploit this property. The idea of using tree decomposition is derived from the fact that many NP-hard problems like Maximum-Weight Independent Set, Graph Coloring problem and Dominating Set problem can be solved polynomially when the input is a tree (a restricted structure which has no cycle). We can easily build a dynamic programming algorithm which exploits the fact that on a tree the computation can be broken down into several subproblems with very limited interaction among them (see figure 1.2). Once we decide whether or not to include a node in the dominating set, the subproblems in each subtree becomes completely separated; we can solve each as though the others did not exists [13].
We don’t encounter such a nice situation in general graphs (or in graphs derived from the wireless networks), where there might not be a node that “breaks the communication” between subproblems in the rest of the graphs. But in such cases we can recursively decompose the input graph by removing small sets of nodes rather removing a single node to break the communication among subproblems in the graph. This means that these graphs have a “tree like” pattern. By utilizing this tree like structure, it is possible to design a dynamic programming algorithm to solve NP-hard problems on general graphs. Not all wireless sensor networks have a tree like structure. In certain applications, we conjecture that the treewidth of sensor net graphs is small. We will use such graphs (graphs are randomly generated to mimic a network structure) to solve the Facility Location Covering problem using a dynamic programming algorithm that exploits the independent subproblem structure of those graphs. But before that we need to decompose a graph into that structure which discerns it’s tree like characteristics. This structure is called a tree decomposition.

To better understand the intuition behind the tree decomposition of a general graph, let us discuss Figure 1.2. The graph $G$ pictured in this figure is decomposable in a tree like way. If $G$ is seen like in the Figure 1.2(a), then its tree like structure might not appear immediately. In Figure 1.2(b), however, we see that $G$ is actually composed of ten interlocking triangles; and seven of the ten triangles have the property that if we delete them, then the remainder of $G$ falls apart into disconnected pieces that recursively have this interlocking-triangle structure. The other three triangles are attached at the extremities, and deleting them is similar to deleting the leaves of a tree.

In Figure 1.2(a), it is apparent that $G$ has many cycles, but in the Figure 1.2(b), it appears as though it does not have any cycles when it is viewed as ten interlocking triangles. Based on this structure, $G$ inherits many of the nice decomposition properties of a tree.

In Figure 1.2(c) a tree representation of the graph is shown where each node corresponds to one of the triangles of the graph in Figure 1.2(b). Two tree nodes are adjacent if the correspondent trian-
Parts (a) and (b) depict the same graph drawn in different ways. Part (b) emphasizes the way in which it is composed of ten interlocking triangles. Part (c) illustrates schematically how these ten triangles "fit together". It also shows triangles containing a vertex forming a subtree (black triangles) [Algorithm Design-Kleinberg, Tardos, 2006].

Figure 1.2: An Example of a Covering Problem in a Wireless sensor network

gles have common graph vertices. If we notice in Figure 1.2(c), the nodes in the tree that shares a common vertex forms a connected subtree within the tree. For example, vertex $a$ is shared by 3 triangles in Figure 1.2(b). In the tree, they are represented by black triangles which forms a connected subtree. So, we can say that graph $G$ has been decomposed into a tree like pattern, and the final output is called a tree decomposition of $G$ [13]. If we compare between a tree (a graph without a cycle) and tree decomposition the difference that is noticeable is that each node in a tree is a single vertex, whereas in a tree decomposition every node is composed of a set of vertices of the input graph. If we delete a single node in a tree then it separates the tree into more than one connected subtree. Similarly in a tree decomposition, if we delete a node then it separates the entire graph into more than one connected components, which as described earlier is one of the key reason for using tree decomposition. The formal definition of a tree decomposition is discussed in Chapter 2, section 2.1.

Another reason to use tree decomposition is the recent computational study results. Though at the beginning, the application of tree decomposition has been considered for theoretical interest only, over the years research studies have shown that Tree Decompositions can be applied in practical purposes too. One of the first attempt is taken by Cook and Seymour [7]. They used branch decom-
positions to obtain close-to-optimal solutions of the traveling salesman problem. Path decompositions are used by Verweij [31] to solve lifting problems of cycle inequalities for the independent set problem. Koster, Van Hoesel and Kolen [17, 19] used tree decompositions to obtain lower bounds and optimal solutions for a special type frequency assignment problems. Currently one of the most efficient algorithm for the inference calculation in probabilistic (or Bayesian) networks builds upon a tree decomposition of a network’s moralized graph [12, 21]. These studies implies that dynamic programming algorithm based on a path/tree/branch decomposition of the graph can be an alternative for integer programming techniques to solve hard optimization problems.

1.3 Summary of Contributions

The primary goal of our research is to investigate whether the use of algorithms that exploits structure of sensor network graphs using tree decomposition are practical or not. We chose a fundamental facility location problem (the Covering problem) that models sensor net deployment. Solving Covering problem with dynamic programming exists for trees but not for tree decompositions. We conclude that dynamic programming on tree decomposition is expensive in terms of memory requirement because of the fact that swapping occurs frequently between main memory and data storage to compensate the lack of main memory. We propose several heuristics meant to reduce the storage which are discussed in Chapter 3. We also propose an algorithm which uses the tree decomposition partially by stopping the recursive step in dynamic programming before the reaching the leaves of the decomposition. The bottom halve of the tree decomposition is solved by an optimization software package called CPLEX.

The secondary goal of our research is to review the practicality of computing tree decomposition. Two types of heuristics are used in practices. The first type is based on finding minimum separator of a graph. The second type is based on a chordal graph characteristic called perfect elimination order which is then used to find a clique tree from a graph. We implemented two heuristics based on these two types. Another heuristic is developed based on the minimum separator heuristic which
randomly selects a separator instead of a minimum separator. After experiments (details in Chapter 4), we conclude that clique tree heuristic is the fastest producing good quality tree decomposition while minimum separator heuristic produces better quality tree decomposition but really expensive in terms of running time. The random separator heuristic performs well in terms of running time with good quality tree decomposition compared to the minimum separator heuristic. As we discuss in the Conclusion chapter, one of the ideas that is conceived from our research is to use a partial tree decomposition rather than using an entire one. In this respect, the separator based heuristic should be useful as the algorithms for the partial tree decomposition should decide on finding a balanced separator as a root so that the components of the graph left without a tree decomposition will be balanced in size.

1.4 Thesis Outline

Solving an optimization problem using tree decomposition of a graph of bounded treewidth is a two step procedure: (i) computation of a (good) tree decomposition, and (ii) application of an algorithm (dynamic programming) based on the tree decomposition. We divided the chapters of this thesis in this respect.

In Chapter 2, we discuss about the formal definition of tree decomposition and the two heuristics to compute a good (may be not optimal) tree decomposition of the input graph. Also nice tree decomposition, another form of tree decomposition is discussed.

In Chapter 3, we discuss about the dynamic programming algorithm that solves the Covering problem on the tree decomposition (computed by the heuristics in Chapter 2) of a graph. We discuss several techniques which helped us to prune the redundant cost functions in order to make the algorithm more efficient. We also discuss a variant of this algorithm in which a part of the tree decomposition is solved by CPLEX, an optimization software package, rather than the dynamic program.
In Chapter 4, we discuss the empirical results and analysis of our algorithms. Several data tables and graph comparisons are made to facilitate the scrutiny. In Chapter 5, we discuss the implications of our research with future directions.
Chapter 2
Background on Tree Decomposition

In this chapter we will discuss the concept and definition of Tree Decomposition and it’s computation. In section 2.1 we discuss the formal definition of Tree Decomposition and Nice Tree Decomposition. As computing Tree Decomposition of a graph is NP-hard, we use heuristics with good practical running time to compute the Tree Decomposition. In section 2.3 and section 2.4 we described in detail the two heuristics, minimum separator heuristics and clique tree heuristics with pseudo-code algorithm.

2.1 Definitions and Notations

We introduce the definitions and notations that will be used throughout this thesis. Let $G = (V, E)$ be an undirected graph with vertex set $V$ and edge set $E$. Let $n = |V|$ and $m = |E|$. The set of adjacent vertices denoted by $N(v) = \{w \in V : vw \in E\}$. Let $\delta(v) = |N(v)|$ be the degree of $v$. A set of vertices $Q \subseteq V$ is called a clique in $G$ if there is an edge between every pair of distinct vertices from $Q$. The cardinality $|Q|$ of $Q$ is the size of the clique [18].

Now, Let us formally define the tree decomposition of an input graph $G$.

**Definition (Robertson and Seymour [27]).** Let $G = (V, E)$ be a graph. A tree decomposition of $G$ is a pair $(T, \chi)$, where $T = (I, F)$ is a tree with node set $I$ and edge set $F$, and $\chi = \{X_i : i \in I\}$ is a family of subsets of $V$, one of each node of $T$, such that

(i) $\bigcup_{i \in I} X_i = V$.

(ii) for every edge $vw \in E$, there is an $i \in I$ with $v \in X_i$ and $w \in X_i$, and

(iii) for all $i, j, k \in I$, if $j$ is on the path from $i$ to $k$ in $T$, then $X_i \cap X_k \subseteq X_j$.

The width of a tree decomposition is $\max_{i \in I} |X_i| - 1$. The treewidth of a graph $G$, denoted by $tw(G)$, is the minimum width over all possible tree decompositions of $G$. 


The third condition of the tree decomposition is equivalent to the condition that for all \( v \in V \), the set of nodes \( \{i \in I : v \in X_i\} \) is a connected subtree of \( T \). Figure 2.1 shows a graph and its tree decomposition [18].

### 2.1.1 Nice Tree Decomposition

A tree decomposition can be easily converted into a nice tree decomposition of the same width with a linear size of \( T \). The resulting tree is rooted and binary [5]. There are four kinds of nodes in a nice tree decomposition -

- **Leaf** nodes \( i \) are leaves of \( T \) and have \( |X_i| = 1 \).

- **Introduce** nodes \( i \) have one child \( j \) with \( X_i = X_j \cup \{v\} \) for some vertex \( v \in V \).

- **Forget** nodes \( i \) have one child \( j \) with \( X_i = X_j - \{v\} \) for some vertex \( v \in V \).

- **Join** nodes \( i \) have two children \( j \) with \( X_i = X_{j_1} = X_{j_2} \).

In our algorithm we used a nice tree decomposition instead of a normal tree decomposition which considerably eases the design of the algorithm which solves the covering problem. Details are described in chapter 3. In figure 2.2 a nice tree decomposition of a normal tree decomposition (of a graph from figure 2.1) is shown.
2.2 Construction of a Tree Decomposition

Finding a tree decomposition with optimal width is NP-hard [4]. In our research we used two heuristics without any theoretical quality guarantee to compute tree decomposition of a graph which approximates the treewidth close to optimality with a reasonable running time. The two heuristics are

(i) Minimum Separating Vertex Set heuristic (MSVS)

(ii) Clique tree heuristic

The reason they are called heuristics because, these algorithms don’t have a solid theoretical guarantee but in practicality they performs well. Following, we describe these two heuristics in detail.
2.3 Minimum Separating Vertex Set Heuristic

This heuristic was developed in the context of solving frequency assignment problems with a tree decomposition approach [16]. It is based on a characteristics of a tree decomposition. Every tree decomposition can be transformed to a tree decomposition in which the vertex set associated to an internal node separates the graph into at least two components, the vertices associated with the node form a separating vertex set. The heuristic therefore searches for separating vertex sets. To find a good tree decomposition, we in fact search for minimum separating vertex sets.

Before we describe the MSVS heuristic, we briefly describe how a separating vertex set of minimum cardinality can be found in a graph.

2.3.1 Minimum Separating Vertex sets and their Computation

Definition (Minimum Separating Vertex set). An st-separating set of a graph $G = (V, E)$ is a set $S \subseteq V \setminus \{s, t\}$ with the property that any path from $s$ to $t$ passes through a vertex of $S$. The minimum separating vertex set of $G$ is given by the st-separating set $S$ with minimum cardinality over all combinations $st \notin E$.

As described by Ahuja, Magnanti and Orlin [1], the st-separating set with minimum cardinality can be found efficiently using Menger’s theorem.

Theorem 1 (Menger [22]). Given a graph $G = (V, E)$ and two distinct non-adjacent vertices $s, t \in V$, the minimum number of vertices in an st-separating set is equal to the maximum number of vertex-disjoint paths connecting $s$ and $t$.

So, we have to calculate the maximum number of vertex-disjoint paths. This problem is solvable in polynomial time by standard network flow techniques. First, $G$ is transformed into a directed graph $D = (V, A)$ in which each edge $vw$ is replaced by two arcs $(v, w)$ and $(w, v)$. Next, we construct an auxiliary directed graph $D'$, with weights on the arcs, by

- replacing each vertex $v$ by two vertices $v'$ and $v''$,
• redirecting each arc with head $v$ to $v'$, and introducing a weight of $\infty$,

• redirecting each arc with tail $v$ to $v''$, and introducing a weight of $\infty$, and

• adding an arc from $v'$ to $v''$ with weight 1.

In the following Figure 2.3, such a transformation from a non directed Grid graph $G$ to its auxiliary directed equivalent $D'$ is shown.

![Figure 2.3: Transformation of a non directed graph to its auxiliary equivalent](image)

Figure (a) of 2.3 corresponds to a non directed Grid Graph $G$ of nine vertices. Figure (b) of 2.3 corresponds to a auxiliary directed graph $D'$ after its construction from the non directed graph $G$ as instructed above. Here each vertex $v$ in Figure (a) is replaced by two vertices $v'$ and $v''$ in Figure (b). Incoming edges ended at vertex $v'$ and outgoing edges originated at $v''$. Each edge from $v'$ to $v''$ has weight 1. All the other edges have $\infty$ as their weight.

The minimum number of vertices in an $s$-$t$-separating set in $G$ is equal to the minimum weight of an $s''$-$t'$ cut in $D'$. So, if we calculate the minimum $s''$-$t'$ cut for every non-adjacent pair $s, t \in V$, we obtain the minimum separating vertex set [16].
In the next section, we describe the MSVS heuristic algorithm that builds a tree decomposition by finding the separator of the input graph.

2.3.2 MSVS Heuristic

The MSVS heuristic is an improvement heuristic. We start with the trivial tree decomposition in which we have one node corresponding to the complete graph. During the process we have a tree decomposition \((T, \chi)\), where \(I\) is the node set and \(F\) is the edge set of the tree \(T\). We select the node \(i \in I\) with \(|\chi_i|\) maximum. This node is replaced by \(m + 1\) nodes \(i_0, \ldots, i_m\) with vertex sets \(\chi_{i_0}, \ldots, \chi_{i_m}\). The nodes \(i_1, \ldots, i_m\) are connected with \(i_0\). Each node \(k \in N(i)\) is connected to exactly one node \(j \in \{i_0 \ldots i_m\}\), such that all conditions of a tree decomposition are satisfied again.

The sets \(\chi_{i_0}, \ldots, \chi_{i_m}\) are defined as follows. We construct a graph \(H = (V(H), E(H))\) where \(V(H)\) consists of set \(\chi_i\) and \(E(H)\) consists of the induced subgraph \(G_{\chi_i}\) and \(\bigcup_{k \in N(i)} C(\chi_i \cap \chi_k)\), which denotes the additional edges, where \(C(X)\) denotes a complete graph on the vertices \(X\). If \(H\) is a complete graph, then \(\chi_{i_0} := \chi_i\) and \(m = 0\), i.e. we do not change the tree decomposition. If \(H\) is not a clique, then we calculate a minimum separating vertex set \(S \subseteq \chi_i\). These actions corresponds to line 1 and 4 of Algorithm 2.1 described later. Let \(Y_{i_1}, \ldots, Y_{i_m}\) be the vertex sets of the \(m \geq 2\) components of \(H_{V(H) \setminus S}\). We define \(\chi_{i_0} := S\), and \(\chi_{i_j} := Y_{i_j} \cup S\) for all \(j \in 1, \ldots, m\) which corresponds to lines 9-13 of Algorithm 2.1. The set \(\chi_k\) has a non-empty intersection with at most one set \(Y_{i_j}\), \(j = 1, \ldots, m\): Let \(v, w \in \chi_i \cap \chi_k\), then \(\{v, w\} \in C(\chi_i \cap \chi_k) \subset E(H)\), which implies that \(v\) and \(w\) cannot be separated by \(S\). So, either \(v, w \in S\) or \(v, w \in Y_{i_j} \cup S\) for only one \(j \in \{1, \ldots, m\}\). Therefore, we connect each neighbor \(k \in N(i)\) with the node \(i_j\), \(j \in \{1, \ldots, m\}\) for which the intersection of \(\chi_k\) and \(\chi_i\) is non-empty, or in case there is none the we connect with \(i_0\) (corresponds to lines 14-18 of Algorithm 2.1). As a consequence, the new construction is a tree again (see Figure 2.3). In Figure 2.4, a step by step construction of the tree decomposition is shown.
In the new tree the conditions for a valid tree decomposition again hold. Since \( \bigcup_{j=0}^{m} X_j = (\bigcup_{j=0}^{m} Y_j) \cup S = X_i \), condition (i) (from Section 2.1) is satisfied. To satisfy condition (ii) we have to prove that for each edge \( \{v, w\} \in E(X_i) \) one of the new vertex sets \( X_{i_0}, \ldots, X_{i_m} \) contains both vertices. If \( v, w \in S \), then this is trivially true. Otherwise, suppose \( v \in Y_{i_j} \) for some \( j \in \{1, \ldots, m\} \). If \( w \in Y_{i_k}, k \neq j \), then \( S \) does not separate \( Y_{i_j} \) and \( Y_{i_k} \); a contradiction. And thus, \( w \in Y_{i_j} \cup S = X_{i_j} \). Condition (iii) states that all nodes in the tree that contain the same vertex \( v \) must form a subtree. We only need to check this for \( v \in X_i \). If \( v \in S \) then \( v \) is contained in all new nodes and the condition is trivially satisfied. Otherwise let \( v \in Y_{i_j} \) for some \( j \in \{1, \ldots, m\} \). By construction, nodes \( k \in N(i) \) and \( i_j \) are connected if \( X_k \) and \( Y_{i_j} \) intersect. Hence, all nodes that contain \( v \) form a subtree again.

Figure 2.5: Construction of Tree Decomposition by MSVS
Note that, if $H$ is not a clique, then there exist vertices $v, w \in X_i$ with $\{v, w\} \neq E(H)$. Thus $S = X_i \setminus \{v, w\}$ separates $H$ in two components; $Y_1 = \{v\}$ and $Y_2 = \{w\}$. So, $\max\{|Y_1 \cup S|, |Y_2 \cup S|\} = |X_i| - 1 < |X_i|$. As a consequence, the width of the tree decomposition may decrease [16].

The width of the resulting tree decomposition may not be optimal. However, as long as the separating vertex sets $S$ form cliques in the original graph, the algorithm provides optimal result, since the optimal tree decomposition will contain a node for every clique that separates the graph in multiple components.

In the following, we describe a pseudo-code implementation of the heuristic.

**Algorithm 2.1 MSVS Heuristic [18]**

**Require:** Initial tree decomposition $(T, \chi)$

**Ensure:** modified tree decomposition $(\bar{T}, \bar{\chi})$ with $tw((\bar{T}, \bar{\chi})) \leq tw(T, \chi)$

1. while $\exists i \in I : |X_i| \equiv \max_{j \in I}|X_j|$ and $H = (X_i, E(H))$ non-complete with $E(H) = \bigcup_{k \in N(i)} C(X_i \cap X_k) \cup E_G[X_i]$ do
2. \hspace{1em} $N_{old}(i) := N(i)$ \hspace{1em} \{store old neighbors of $i$\}
3. \hspace{1em} $F := F \setminus \{ij : j \in N_{old}(i)\}$ \hspace{1em} \{disconnect $i$ from tree\}
4. \hspace{1em} let $S \subset X_i$ be a minimum separating vertex set in $H$
5. \hspace{1em} let $q$ be the number of components of $H[X_i \setminus S]$
6. \hspace{1em} $n := |I|$ \hspace{1em} \{current number of nodes\}
7. \hspace{1em} $I := I \cup \{n + 1, \ldots, n + q\}$ \hspace{1em} \{construct $q$ new nodes\}
8. \hspace{1em} $F := F \cup \{ij : j = n + 1, \ldots, n + q\}$ \hspace{1em} \{connect new nodes with $i$\}
9. \hspace{1em} for $p = 1$ to $q$ do
10. \hspace{2em} let $Y_p \subset X_i$ be the set of vertices in component $p$ of $H[X_i \setminus S]$
11. \hspace{2em} $X_{n+p} := Y_p \cup S$ \hspace{1em} \{define new vertex subsets\}
12. \hspace{1em} end for
13. \hspace{1em} $X_i := S$
14. \hspace{1em} for $j \in N_{old}(i)$ do
15. \hspace{2em} if $\exists P \in \{1, \ldots, q\}$ with $X_j \cap Y_p \neq \emptyset$ then
16. \hspace{3em} $F := F \cup \{n + p, j\}$ \hspace{1em} \{reconnect old neighbors\}
17. \hspace{2em} else
18. \hspace{3em} $F := F \cup \{i, j\}$ \hspace{1em} \{reconnect old neighbors\}
19. \hspace{2em} end if
20. \hspace{1em} end for
21. end while
2.3.3 Random Separator Vertex set Heuristic (RSVS)

As described in Chapter 4, building tree decompositions using the minimum separator vertex set heuristic (MSVS) is quite expensive. This is mainly because finding minimum separator of a graph is time consuming. It is evident from line 4 of algorithm 2.1 that to build the tree decomposition using this heuristic, the program needs to compute minimum separator of a subgraph a number of times which leads to a high running time of the entire program.

In this aspect, we decided to find techniques through which this high running time can be reduced. One such approach was to find a random separator of a set instead of a minimum separator. In this method, instead of considering all non-adjacent vertex pairs of a graph, we consider only a handful of non-adjacent vertex pair. The resultant separator may not be minimum but it greatly reduces running time as the algorithm now deals with only a few non-adjacent vertex pairs. We built another algorithm which is similar to 2.1. The only change is in Line 4 where the new algorithm will find a random separator instead of a minimum one. After experimenting on different graphs (details in Chapter 4), we found the results encouraging. This RSVS heuristic takes much less time than the MSVS approach and most of the time the quality of the tree decomposition is within an acceptable range.

Though we did not use the RSVS heuristic to generate tree decompositions for our research, but it has potential to be used in our future endeavors. As discussed in the section 1.3, we believe that this modified version of the separator heuristic will contribute significantly in developing the algorithm for partial tree decomposition where finding a balanced separator in quick time is crucial.

In the next section, we will begin describing the notations and definitions and later the algorithms for Clique Tree Heuristic.
2.4 Clique Tree Heuristic

This heuristic is based on the characteristics of triangulated graph or chordal graph. In this technique we discern a clique tree from a triangulated graph. We used the technique (for finding clique tree) developed by Habib, McConnell, Paul and Viennot [11]. They proposed an $O(n + m)$ ($n =$ number of vertices, $m =$ number of edges) algorithm to find a clique tree from a chordal graph given an elimination scheme.

2.4.1 Chordal Graph: Definitions and Characteristics

The following definitions and descriptions are based upon the work by Habib, McConnell, Paul, Viennot [11] and Koster, Bodlaender, Hoesel [18].

A graph is called **Chordal** if every cycle of length at least four contains a chord, that is, two non-consecutive vertices on the cycle are adjacent. A chordal graph is also a perfect graph.

Chordal graphs are characterized by the existence of **perfect elimination ordering** of their vertices, which is defined as follows. A **clique** is a set of vertices inducing a complete subgraph. An ordering $x_1, \ldots, x_n$ of vertices is a perfect elimination ordering of a graph $G = (V, E)$ if the neighborhood of each vertex $x_i$ is a clique of the induced subgraph $G_{x_i, \ldots, x_n}$. There exists an arrangement of maximal cliques in an chordal graph such that the maximal cliques containing a given vertex always induces a connected subtree. Our target is to find this clique tree which will also serve as a tree decomposition of the graph. The following properties are described in detail in respect to the target.

**Theorem 2** (Gavril [10]). A graph $G = (V, E)$ is triangulated if and only if $G$ can be constructed as the intersection graph of subtrees of trees, i.e., there exists a tree $T = (I, F)$ such that one can associate a subtree $T_v = (I_v, F_v)$ of $T$ with each vertex $v \in V$, such that $vw \in E$ if and only if $I_v \cap I_w \neq \emptyset$.

To state this definition in a different way, a graph is triangulated if and only if there exists a tree
decomposition with the additional property that \( vw \in E \) if and only if \( I_v \cap I_w \neq \emptyset \). This additional property guarantees that the tree decomposition has minimal width. Let \( X_i := \{ v \in V : i \in I_v \} \). Non-adjacent vertices are not in a common subset \( X_i \). Hence, a maximum cardinality subset \( X_i \) contains the vertices of a maximum cardinality clique \( C \) in \( G \). Since it also holds that in any tree decomposition \( (T, \chi) \) and every maximum clique \( C \), there exists a node \( i \in I \) with \( C \subseteq X_i \), the tree decomposition has minimum width. Moreover, it follows that the treewidth of a triangulated graph equals the maximum clique number minus one, \( tw(G) = \omega(G) - 1 \).

**Lemma 1.** For every graph \( G = (V, E) \), there exists a triangulation of \( G \), \( \bar{G} = (V, E \cup E_t) \), with \( tw(G) = tw(\bar{G}) \)

**Proof.** Let \( G \) be a general graph and \( (T, \chi) \) a tree decomposition of minimum width. We construct a graph \( \bar{G} = (V, \bar{E}) \) by the following rule: \( vw \in \bar{E} \) if and only if \( v, w \in X_i \) for some \( i \in I \). From Theorem 2 it is clear that \( \bar{G} \) is triangulated. From the second condition (from section 2.1) of a tree decomposition, the edge set \( \bar{E} \) can be divided into two parts \( E \) and \( E_t \). So, \( \bar{G} \) is a triangulation of \( G \) by the addition of the triangulation edges \( E_t \). Moreover, the treewidth of \( G \) and \( \bar{G} \) is equal, \( tw(G) = tw(\bar{G}) \).

**Corollary 1.** Finding the treewidth of a graph \( G \) is equivalent to finding a triangulation \( \bar{G} \) of \( G \) with minimum clique size.

Since finding the treewidth of a graph is \( NP \)-hard, also finding a triangulation with minimal clique number is an \( NP \)-hard problem. The maximal clique number (minus one) of any triangulation \( \bar{G} \) of a graph \( G \) provides an upper bound on the treewidth. Moreover, \( \omega(\bar{G}) \) can be computed in polynomial time.

There exists several algorithm which can convert any graph into it’s triangulated (or chordal) equivalent and also provides us the elimination order. We will use an algorithm called the Minimum degree heuristic [2] to triangulate a graph and find it’s elimination order. Then given the elimination scheme we will use the clique tree algorithm [11] to build a clique tree (also a tree decomposition)
of the triangulated graph. Since, the treewidth of a graph and its triangulated equivalent is the same, the tree decomposition of the triangulated graph is also a valid tree decomposition of the original graph.

### 2.4.2 Minimum Degree Heuristic

The Minimum degree algorithm (MD) is widely used as one of the heuristics for computing a triangulation of a graph. The algorithm is based on the algorithm called Elimination Game (EG), developed by S. Parter [24]. EG simulates Gaussian elimination on graphs by repeatedly choosing a vertex and adding edges to make its neighborhood into a clique before removing it. The resulted graph is always a triangulated graph. MD is observed [3] to produce triangulations which are often minimal or close to minimal. The following definitions, notations are based on the work done by Berry, Heggernes and Telle [2].

Given a graph $G = (V, E)$, we denote $n = |V|$ and $m = |E|$. For any subset $S$ of $V$, $G(S)$ denotes the subgraph of $G$ induced by $S$. We define $H = G + \{e\} + \{x\}$ when $H$ is obtained from $G$ by adding edge $e$ and vertex $x$. For any vertex $v$ of $G$, $N_G(v)$ denotes the neighborhood of $v$ in $G$, and $N_G[v]$ denotes the set $N_G(v) \cup \{v\}$. For a given set of vertices $X \subset V$, $N_G(X) = \cup_{v \in X} N_G(v) \setminus X$ and $N_G[X] = \cup_{v \in X} N_G(v) \cup X$. A function $\alpha : V \to \{1, 2, \ldots, n\}$ is called a ordering of the vertices of $G = (V, E)$, and $(G, \alpha)$ will denote a graph $G$, the vertices of which are ordered according to $\alpha$. We will use $\alpha = (v_1, v_2, \ldots, v_n)$, where $\alpha(v_i) = i$.

**Algorithm 2.2 Elimination Game [2]**

**Require:** A graph $G = (V, E)$, and an ordering $\alpha$ of the vertices in $G$

**Ensure:** A triangulation $G_\alpha^+$ of $G$

1. $G^1_\alpha \leftarrow G$
2. $G^+ \leftarrow G$
3. **for** $k = 1$ to $n$ **do**
4. \hspace{1em} Let $F$ be the set of edges necessary to saturate $N_G^k(v_k)$ in $G^k_\alpha$
5. \hspace{1em} $G^{k+1}_\alpha \leftarrow G^k_\alpha + F - \{v_k\}$
6. \hspace{1em} $G^+_\alpha \leftarrow G^+_\alpha + F$
7. **end for**
In the algorithmic description of EG, $G_k^\alpha(\{v_k, \ldots, v_n\} \setminus N_{G_k^\alpha}[v_k])$ is the section graph at step $k$. For any distinct $i, j \in [k, n]$, edge $v_iv_j$ is present in $G_k^\alpha$ iff there is a path in $G$ between $v_i$ and $v_j$ (the path may have only one edge) all intermediate vertices of which are numbered $< k$ (i.e. do not belong to $G_k^\alpha$). Similarly, the edges added during an execution of $EG$ are well defined as $v_iv_j$ is an edge of $G_k^\alpha$ iff $v_iv_j$ is an edge of $G$ or there is a path in $G$ between $v_i$ and $v_j$, all intermediate vertices of which have a number which is strictly smaller than $\min\{i, j\}$.

The Minimum Degree Heuristic is based on EG: it takes as input an unordered graph $G$, and computes and ordering $\alpha$ along with the corresponding triangulation $G_k^\alpha$, by choosing at each step a vertex of minimum degree in $G_k^\alpha$ and numbering it as $v_k$.

In the next section we will describe the clique tree algorithm which utilizes the triangulated graph and the elimination order computed by this MD heuristic.

### 2.4.3 Clique Tree Algorithm

While working with lexicographic breadth first search (Lex-BFS) and partition refinement Habib, McConnell, Paul and Viennot [11] came up with a simple but efficient algorithm to find the maximal cliques from a triangulated graph. The following definitions, notations, algorithms and proofs are based on the narrative by the mentioned authors.

Before we describe the algorithm, we define below $RN(x)$, the right neighborhood of $x$. Given a graph $G$ and an elimination order $\pi$, we define $RN(x)$ to be the neighbors to the right of $x$, namely, the set, $\{y: y \in N(x) \text{ and } \pi(y) > \pi(x)\}$. $G$ is triangulated if and only if there exists a perfect elimination order [9]. An algorithm can be devised to recognize triangulated graphs. Given the elimination order $\pi$, this algorithm checks whether this ordering is a perfect elimination ordering or not. The algorithmic description is given below-
Algorithm 2.3 Chordality test [11]

Require: a graph $G = (V, E)$, and an ordering $\pi$ of vertices

Ensure: TRUE if $\pi$ is a perfect elimination ordering

1: for each vertex $x$ do
2: let $RN(x)$ be its neighbors to the right
3: let $parent(x)$ be the leftmost member of $RN(x)$ in $\pi$
4: end for
5: Let $T$ be the tree defined by the parent pointers
6: for each vertex $x$ in $T$ in postorder do
7: check that $(RN(x) \setminus parent(x)) \subseteq RN(parent(x))$
8: end for
9: if no check failed then
10: return TRUE
11: end if

For the correctness, note that if $\pi$ is a perfect elimination ordering, then $\{x\} \cup RN(x)$ is a clique, where $x$ is its leftmost member and $parent(x)$ is its next leftmost member. The check obviously cannot fail. If it is not a perfect elimination ordering, then for some $x$, $x \cup RN(x)$ is not a clique. Without loss of generality, let $x$ be the rightmost vertex in $\pi$ with this property. By our choice of $x$, $parent(x)$ fails to have as a neighbor some vertex to its right that is a neighbor of $x$, so the check fails.

In the second for loop of Algorithm 2.3, it is checked whether $(RN(x) \setminus parent(x)) \subseteq RN(parent(x))$ or not. If the order $\pi$ is indeed a perfect elimination order then this condition is true. We will prove this in the following. For each vertex $x$, $RN(x)$ is a subset of the ancestors of $x$ in $T$. This is true for the root. Suppose it is true for any vertex at depth $k$, and assume that $x$ is at depth $k + 1$. The parent of $x$ is the earliest member of $RN(x)$ in $\pi$. Since $RN(x)$ is a clique, $RN(x) \setminus parent(x)$ is a subset of $RN(parent(x))$. By the inductive hypothesis, $RN(x) \setminus parent(x)$ is a subset of the ancestors of $parent(x)$.

We will now describe the algorithm for finding the clique tree, that is, to arrange the maximal cliques into a tree such that for each vertex, the subtree induced by the cliques that contain $x$ are connected.
**Algorithm 2.4 Clique Tree [11]**

**Require:** $G$ is a triangulated graph, and $\pi$ is a perfect elimination ordering

**Ensure:** A clique tree $\tau$ of $G$

Let $T$ be defined as in Algorithm 2.3

Let $r$ be the root of $T$

for each vertex $x$ in $T$ except the root, in preorder do

if $(RN(x) \setminus parent(x)) \neq RN(parent(x))$ then

create a new clique $C = \{x\} \cup \{RN(x)\}$

$C(x) \leftarrow C$

$parent(C) \leftarrow C(parent(x))$

else

$C(parent(x)) \leftarrow C(parent(x))\cup \{x\}$

$C(x) \leftarrow C(parent(x))$

end if

end for

It can be easily proven that for the same clique, the property $(RN(x) \setminus parent(x)) = RN(parent(x))$ is true. Finally, we show that after each vertex is processed, the parent relation is a clique tree on the subgraph induced by the set of processed vertices. To do this, we show that for an arbitrary processed vertex $y$, the cliques containing $y$ induce a connected subtree of this tree. As a base case, it is true just after $y$ is processed, since it is contained in only one clique of the tree. Suppose it is true just before some subsequent vertex $x$ is processed. If no new clique containing $y$ is created, it continues to be true. So assume that processing $x$ creates a new clique $C$ and $y$ is contained in $C$. It suffices to show that the parent of $C$ is a pre-existing clique that contains $y$. For each processed vertex $z$, $C(z)$ contains $\{z\} \cup RN(z)$. In particular, $C(parent(x))$ contains $\{parent(x) \cup RN(parent(x))\}$. Since $\{parent(x) \cup RN(parent(x))\}$ contains $RN(x)$, $C(parent(x))$ contains $y$. The parent of the new clique is a pre-existing clique containing $y$. It follows that the tree is a clique tree or a tree decomposition for $G$ after all vertices are processed.

In Figure 2.4, a step by step construction of clique tree from an input graph is shown.
Elimination Order:
k < j < i < c < d < f < e < g < h

Min degree Heuristic

Keep adding vertex \( x \in V \), in the same tree decomposition node if for \( x \) and parent(parent(x)), \( RN(x) \setminus \text{parent}(x) = RN(\text{parent}(x)) \). It means the vertices inside the node are members of the same clique.

For vertices that are not in the same clique (in this case, for \( f \), \( RN(x) \setminus \text{parent}(x) \neq RN(\text{parent}(x)) \)), build a new node with the vertex set = \{x\} \cup \{RN(x)\} (in this case the new node vertex set = \{f\} \cup \{e\}). Then for the next vertex check whether it belongs to the newly formed clique or not. If it is, then add in the same node like before (\( d \) is added in the new node).

Figure 2.6: Step by Step execution of Clique Tree Heuristic
We have implemented both heuristics and experiments show (chapter 4) that clique tree heuristic is much faster than the minimum separator heuristic. Based on the results we decided to use the clique tree heuristic to compute the tree decomposition on which we will apply our dynamic programming algorithm. Details are described in the next chapter.
Chapter 3

Dynamic Programming Algorithm to Solve Covering Problem

Algorithms for covering problem on trees already exists [14]. In our research we extend dynamic programming on tree decompositions to build an algorithm to solve covering problem optimally on general graphs where the complexity is exponential in treewidth. In this chapter we describe the algorithm and various heuristic techniques to speed up the algorithm.

In section 3.1 we discuss a bottom up dynamic programming algorithm with its cost function definition. Then we describe the cost functions of four types of nodes of a special tree decomposition called nice tree decomposition. In section 3.2 we discuss techniques that reduce the number of entries in the cost tables in the dynamic program and speed up the running time. In section 3.3 we discuss a parallel algorithm which solves subproblems on different machines and a hybrid algorithm based on the dynamic programming algorithm where part of the tree decomposition is solved by an optimization software package called CPLEX. In the next section (3.4), we discuss brief implementation details of the algorithms and some implementation tricks which helped reduce the total run time.

3.1 Dynamic Programming Approach

As described in section 1.1, in a client constrained covering problem given \( n \) clients in a graph where each client has a radius \( r \), the objective is to open facilities to cover these clients at a minimized cost. The two types of cost incurred here are the facility opening cost and the penalty cost for uncovered clients. Any arrangement of client facility allocation will yield a solution value. The optimal solution will have the lowest cost value.

Before we describe the dynamic program to solve the Covering problem, we discuss the notations and definitions required for the algorithm.
3.1.1 Cost function and other Definitions

Given a input graph $G = (V,E)$ and it’s tree decomposition is a pair $(T, \chi)$, (where $T = (I,F)$ is a tree with node set $I$ and edge set $F$, and $\chi = \{X_i : i \in I\}$ is a family of subsets of $V$), for each tree decomposition node $i$, let us define $X_i$ as the set $\{x_1 \ldots x_k\}$ of graph vertices corresponding to the node. We define $T_i$, a subtree rooted at node $i$ and $G_{T_i}$, a subgraph induced by the union of the vertices included in the nodes of $T_i$. Also, let $n = |V|$ and $m = |E|$ throughout the following sections.

**Definition (Assignment Function).** Let $f$ be a function $f : X_i \to V$, that maps every vertex in $X_i$ to a vertex of the graph. For a vertex $x \in X_i$, $f(x)$ is the opened facility that is closest to $x$. This function will define a restricted covering problem on the subgraph induced by $G_{T_i}$. We also denote the entire facility allocation for a node $X_i$, by $f_X$.

We say that a client $x \in X_i$, is covered by a facility $f(x)$, if the shortest path distance between $x$ and $f(x)$, defined as $d(x, f(x))$ is less than or equal to the client radius $r$. When a facility $f(x)$ is opened, then an opening cost $c_f(x)$ is incurred. Likewise, when a client, $x \in X_i$, is not covered by any opened facility, then a penalty cost $b_x$ is incurred. The following cost function definition is similar to those defined by Arie Tamir in his study of P-median problem on trees [29].

**Definition (Recursive Cost Function).** We define the cost function as $\Phi(X_i, f(x))$, that denotes the cost value for the covering sub-problem defined on the subgraph $G_{T_i}$ with the constraint that $f(x)$ is the closest open facility to $x$, for all $x \in X_i$. The cost value for this function is equal to the sum of the opening cost of the facilities opened inside or outside the subgraph $G_{T_i}$ and the sum of the penalties paid for any uncovered client in the subgraph $G_{T_i}$. The optimal value of the sub-problem defined on the subgraph $G_{T_i}$ can be found by taking the minimum cost functions over all possible allocation functions $f(x)$.

As we are employing a nice tree decomposition, there are only four types of nodes present in the tree decomposition. In the following subsections, for each types of node, a set of recursive cost functions are defined. As this is a leaves to root dynamic program, the solution propagates from leaves to root. Each node has a cost table associated with it which contains a cost function entry for
every possible facility configuration to serve the client set contained in the node. After the program ends, the root contains the optimal solution.

### 3.1.2 Leaf Node

Let $i$, where $i \in I$ be a leaf node. According to the nice tree decomposition properties, $X_i$ contains a single vertex $x$, that is, $X_i = \{x\}$. Two cases are possible for this type of node.

**Case 1** If the single vertex $x$ is covered by a facility $f(x)$ then the facility opening cost will be added to the cost function. So, for $d(x,f(x)) \leq r$, the recursive cost function for leaf node becomes

$$
\Phi(\{x\}, f(x)) = c_{f(x)}
$$

**Case 2** If the client $x$ is not covered by the assigned facility $f(x)$ then a penalty must be paid and as such a penalty cost is added to the cost function. The cost function becomes

$$
\Phi(\{x\}, f(x)) = c_{f(x)} + b_x
$$

Except leaf node, every other node’s cost functions depends on its child node’s cost function. For a parent node $i$ and its child node $j$, Let us define another assignment function $g$, $g : X_j \Rightarrow V$, that maps every vertex in $X_j$ to a vertex of the graph. It is similar to the assignment function $f$. Given the assignment functions $f$ and $g$ for the parent node $i$ and child node $j$, Let $G_f$ be the set of facility allocations functions at the child node $X_j$ that have the same image as functions $f$ on the set $X_i \cap X_j$. Formally,

$$
G_f = \{ g : X_j \rightarrow V \mid g(x) = f(x) \text{ for all } x \in X_i \}
$$

The following figure shows four pieces of the nice tree decomposition from figure 2.4, where each piece describes a different nice tree decomposition node. It also shows the relation between facility assignment functions $f$ and $g$ at the parent and child node.
3.1.1: Leaf Node

Leaf Node Vertex Set
\[ X_i = \{ k \} \]
facility allocation
\[ f(k) = u, u \in V \]

3.1.2: Introduce Node

Child \( Y_j \)

\[ Y_j = \{ d, e \} \]
facility allocation
\[ g(d) = u, u \in V \]
\[ g(e) = v, v \in V \]
\[ g(f) = w, w \in V \]

Parent \( X_i \)

\[ X_i = \{ d, f \} \]
Introduce vertex
\[ \text{facility allocation} \]
\[ f(d) = g(d) \]
\[ f(e) = g(e) \]
\[ f(f) = w, w \in V \]

3.1.3: Forget Node

Child \( Y_j \)

\[ Y_j = \{ d, f \} \]
facility allocation
\[ g(d) = u, u \in V \]
\[ g(e) = v, v \in V \]
\[ g(f) = w, w \in V \]

Parent \( X_i \)

\[ X_i = \{ d, e \} \]
Forgotten vertices
\[ = \{ f \} \]
facility allocation
\[ f(e) = g(e) \]
\[ f(f) = w, w \in V \]

3.1.4: Join Node

Child \( Y_j \)

\[ Y_j = Y_1 = Y_2 = \{ d, f \} \]
facility allocation for
\[ g_1 = g_2 \]
\[ g(d) = u, u \in V \]
\[ g(e) = v, v \in V \]
\[ g(f) = w, w \in V \]

Parent \( X_i \)

\[ X_i = \{ d, e \} \]
facility allocation
\[ f(d) = g_1 \]
\[ f(e) = g_2 \]
\[ f(f) = g_3 \]

Figure 3.1: Different types of Nice Tree Decomposition nodes with their facility allocation

3.1.3 Introduce node

Let \( i \), where \( i \in I \), be an introduce node. According to the definition of Introduce Nodes, if the child node is \( j \), where \( j \in I \) and \( X_j = (x_2 \ldots x_k) \), then \( X_i = X_j \cup \{ x_1 \} \), where \( x_1 \) is the extra or introduced vertex in node \( i \). The facility serving \( x_1 \) is \( f(x_1) \), which can be any vertex in the graph, including \( x_1 \). The other vertices \((x_2 \ldots x_k)\) are serviced by the facility set \{\( f(x_2) \ldots f(x_k) \)\} whose costs are already computed at the child node \( j \). Two cases are possible based on the criteria whether \( f(x_1) \) is already
opened or about to be opened.

**Case 1** The facility \( f(x_1) \) is not already opened, which means \( f(x_1) \notin \{g(x_2) \ldots g(x_k)\} \). Suppose \( x_1 \) is covered by \( f(x_1) \), thus \( d(x_1,f(x_1)) \leq r \). The value of the cost function is the addition of the opening cost of facility \( f(x_1) \) and the cost of the subproblem (with the facility configuration \( \{g(x_2) \ldots g(x_k)\} \) defined on the subgraph \( G(T_{X_j}) \). The cost function becomes

\[
\Phi(X_i, (f)) = \Phi(X_j, (g)) + c_{f(x_1)}
\]

If \( x_1 \) is not covered by \( f(x_1) \) that is \( d(x_1,f(x_1)) > r \), then a penalty cost must be paid. The cost function becomes

\[
\Phi(X_i, (f)) = \Phi(X_j, (g)) + c_{f(x_1)} + b_{x_1}
\]

**Case 2** The facility \( f(x_1) \) is already opened at the child node \( X_j \) which is covering \( x_1 \), which means, \( f(x_1) \in \{g(x_2) \ldots g(x_k)\} \) and \( d(x_1,f(x_1)) \leq r \), the cost function for node \( X_i \) is the same as the cost for it’s child \( X_j \) as the facility \( f(x_1) \) has already been opened at the child node. The cost function becomes

\[
\Phi(X_i, (f)) = \Phi(X_j, (g))
\]

Again, if \( x_1 \) is not covered by \( f(x_1) \) that is \( d(x_1,f(x_1)) > r \), then a penalty cost must be paid. The cost function becomes

\[
\Phi(X_i, (f)) = \Phi(X_j, (g)) + b_{x_1}
\]

### 3.1.4 Forget Node

Let \( i \), where \( i \in I \), be a Forget node. According to the definition of **Forget Nodes**, if the child node is \( j \) and \( X_j = \{x_1, \ldots, x_k\} \) then \( X_i = X_j \setminus \{x_1\} \), where \( x_1 \) is the forget vertex in Node \( i \).
Given the facility allocation \( f : X_i \Rightarrow V \) for forget node \( i \) and \( g : X_j \Rightarrow V \) for the forget node’s child \( j \), computing the cost function at \( i \) requires to find the minimum cost function from child node \( j \) over all cost functions \( \{ \Phi(X_j, g) : g \in G_f \} \). The cost function for forget node becomes:

\[
\Phi(X_i, (f)) = \min \{ \Phi(X_j, (g)) : g \in G_f \}
\]

### 3.1.5 Join Node

Let \( i \), where \( i \in I \), be a Join node. If node \( i \) has \( t \) children, \( j_1, j_2 \ldots j_t \), then according to the definition of Join Nodes, the nodes \( i, j_1, j_2, \ldots, j_t \) have the same vertex set, that is \( X_i = X_{j_1} = X_{j_2} = \ldots = X_{j_t} = (x_1 \ldots x_k) \). Because of this reason, the facility allocation for join node \( i \) and for all it’s children, \( j_1, j_2 \ldots j_t \) will be the same, that is \( f_{X_i} = g_{X_{j_1}} = g_{X_{j_2}} = \ldots = g_{X_{j_t}} \). The cost function value at node \( i \) will be equal to the sum of the cost values of the subproblem defined on the subgraphs \( G_{T_{j_1}}, G_{T_{j_2}} \ldots G_{T_{j_t}} \).

As all the children has the same vertex set and the same \( f \), the covering cost which includes opening costs and penalties for client vertices in set \( X_i = X_{j_1} = X_{j_2} = \ldots = X_{j_t} \), will be repeated in the final cost value. So, the repeated cost is deducted \( t - 1 \) times, where \( t \) is the number of children for a Join node. The recursive cost function becomes

\[
\Phi(X_i, (f)) = \Phi((X_{j_1}, (g)) + \Phi((X_{j_2}, (g)) + \ldots + \Phi((X_{j_t}, (g)) - \sum_{1}^{t-1} \left\{ \sum_{x \in X_i} \kappa(x,f) \right\}
\]

Here, \( \kappa(x,f) \) is a function which computes the covering cost for a pair \((x, f(x))\), where \( x \in X_i \) and \( f(x) \in f_{X_i} = \{f(x_1) \ldots f(x_k)\} \), the facility assignment vector for node \( i \) and also for it’s children \( j_1, j_2, \ldots, j_t \). The function \( \kappa(x,f) \) can be defined as

\[
\kappa(x,f) = \sum_{x \in X} p(x,f(x)) + \sum_{y \in f(x)} C_y
\]
Where $\rho(x, f(x))$ can also be defined as a function

$$
\rho(x, f(x)) = \begin{cases} 
0, & \text{if } d(x, f(x)) \leq r_x \\
 b_x, & \text{if } d(x, f(x)) > r_x.
\end{cases}
$$

This function computes the penalty cost for a pair $(x, f(x))$.

### 3.1.6 Running Time

For each tree decomposition node the cardinality of the vertex set is at most $k + 1$, where $k$ is the treewidth. For a tree node $i$, the number of possible cost functions is $n^{k+1}$, where $n$ is the number of vertices of the original graph. In the worst case, to compute for a parent node $i$, we need to generate $n^{k+1}$ possible cost functions $\Phi(X_i, f)$ at the child node $j$, one for each $n^{k+1}$ assignment function $f$, as defined in Section 3.1.1. Each of the cost function entry can be accessed in a constant time from the cost table. If we look into the cost function equation from the earlier sections then we can determine a upper bound on the running time for processing cost functions for different types of nodes. In case of join node, if it has $t$ number children then to process each cost function, we need to access $t$ number of cost function, one for each child. So, the running time for a join node cost function is bounded by $t$. For forget node, to process a cost function, we need to access at most $n$ number of cost functions from the child node, where $n$ is the number of vertices in the input graph. So, the running time to process a forget node cost function is bounded by $n$. For introduce and leaf node, it takes constant time to each of their cost functions. So, the running time to process a single cost function is no more than $n$. As there are $n^{k+1}$ number of possible cost functions per node, for a single node the running time is bounded by $n^{k+2}$. In a tree decomposition there can be at most $O(n)$ number of tree nodes. So, in the worst case the total running time is $n^{k+3}$. However, with amortized analysis we can improve this running time a bit. For join and forget node, it is described earlier that their running time is bounded by $t$ and $n$. These bounds directly depends on the usage of cost function of their children. In case of a child of a join node, it is evident that each cost function of the child node is used once at the join node. So, in total each cost function generated at a child of join
node is used twice, once at the node that it is generated and next it is used at the join node. Same holds true for introduce and forget node’s child as well. So, we can say that the usage for each cost function generated any node is no more than 2. Now, let us reanalyze the running time. For each node, the total number of operations for the generated cost functions is \(2n^{k+1}\). Again, as there are \(O(n)\) number of nodes, the total number of operations becomes \(2n^{k+2}\). In this term, the previous bounds \(t\) and \(n\) disappears as they are replaced by a constant. Using the big-oh notation, the running time becomes \(O(n^{k+2})\).

### 3.1.7 Proof of Correctness

The dynamic program that we discussed earlier has a optimal substructure. Let us define the following notations for the proof. Given an input graph \(G = (V, E)\) and it’s tree decomposition \((T, \chi)\), let \(OPT\) be the set of all possible solutions to the Covering problem on this graph. Let \(Y_{OPT}\) be one of the optimal solution from the set \(OPT\) which comprises a list of open facilities and the optimal cost value. Let \(f_{Y_{OPT}}, f_{Y_{OPT}} \subseteq Y_{OPT}\) be the set of facilities that covers \(G_T\) when we restrict the optimal solution \(Y_{OPT}\) to the subproblem \(G_T\). The restricted solution will only contain the opened facilities from \(Y_{OPT}\) that cover clients in the subgraph under consideration. The cost value computation for the restricted solution remains the same, that is the sum of opening cost and the sum of penalties for the uncovered clients. For any node \(i\) and it’s client set \(X_i\), let \(G_{T_i}\) be the subgraph defined by the subtree \(T_i\) rooted at node \(i\). We will prove the following-

**Theorem 3.** Given an optimal solution \(Y_{OPT}\), for any node \(i\), let \(C(Y_{OPT}X_i)\) be the cost value produced by the solution provided by \(Y_{OPT}\) if it is restricted to the subproblem \(G_{T_i}\). Then the cost function \(\Phi(X_i, (f_{Y_{OPT}}))\) at node \(i\) will have the cost value \(C(Y_{OPT}X_i)\) and will be optimal to the subproblem defined by \(G_{T_i}\) with respect to the entire problem.

**Proof.**

In figure 3.2 an arbitrary node \(i\) with client set \(X_i\) of a tree decomposition is shown. For simplicity
Figure 3.2: Subtree $G_{T_i}$ and $G \setminus G_{T_i}$ at node $i$

the subtree $T_i$ of node $i$ and the rest of tree $T \setminus T_i$ is depicted as a subgraph $G_{T_i}$ and $(G \setminus G_{T_i})$ defined by them. Now, Let us restrict the solution $Y_{OPT}$ to the subproblem defined by the subgraph $G_{T_i}$.

When we restrict $Y_{OPT}$ to the subproblem defined by $G_{T_i}$, we define the restricted solution as $Y_{OPT_{G_{T_i}}}$, the cost value as $C(Y_{OPT_X_i})$ and the facility allocation given by the restricted solution as $f_{Y_{OPT}}$. For all cost functions at node $i$ defined by $\Phi(X_i, (f))$, one of the cost functions will have the same facility configuration as the $f_{Y_{OPT}}$. We define this cost function as $\Phi(X_i, (f_{Y_{OPT}}))$. We claim that the cost value generated by the cost function $\Phi(X_i, (f_{Y_{OPT}}))$ is equivalent $C(Y_{OPT_X_i})$, that is this cost function is optimal for set $X_i$, as well as for the subproblem defined by $G_{T_i}$.

We will use the proof by contradiction to prove our claim. As described earlier, the optimal solution defined by the cost function $\Phi(X_i, (f_{Y_{OPT}}))$. Some other solution in $G_{T_i}$ that produces a suboptimal cost value at node $i$ may yield a better result than $f_{Y_{OPT}}$ by covering more clients in subgraph $(G \setminus G_{T_i})$ thus reducing the overall cost. This means that at least one facility $p$ from $G_{T_i}$ not in the set $f_{Y_{OPT}}$ covers at least one client $z$ from $(G \setminus G_{T_i})$. The shortest path between $p$ and $z$ must go through one of the clients in $X_i$ as according to the definition of tree decomposition, $X_i$ is a separator that connects $G_{T_i}$ and $(G \setminus G_{T_i})$. Again, by definition, the facilities in $f_{Y_{OPT}}$ are closest to the clients in $X_i$ as $f_{Y_{OPT}}$ is optimal for node $i$. So, any other facility configuration other than $f_{Y_{OPT}}$ cannot have more covered clients in $(G \setminus G_{T_i})$. So, $f_{Y_{OPT}}$ is optimal for node $i$ as well as for the subproblem $G_{T_i}$ and
will eventually form the optimal solution $Y_{OPT}$ in the end.

It is proven that the optimal solution to the whole problem contains within it the optimal solution solution to the subproblem.

3.2 Heuristic Techniques

The brute force dynamic programming approach which is discussed in the earlier section solves the Covering problem optimally but its running time is exponential when the treewidth is unbounded. As shown by our experiments on series parallel graphs\(^1\) of small size. The experiment results showed a high running time because of the high memory consumption of the algorithm. More details on the experiments are further discussed in the next chapter.

To counter this problem, we developed some heuristic techniques which greatly reduced the number of entries in the cost table. We describe these techniques below.

3.2.1 Pruning Heuristic

In the brute force approach, suppose for a tree node $i$, $X_i$ contains $p$ number of vertices. As mentioned earlier in Section 3.1.6, $p$ can be at most $k + 1$, where $k$ is the treewidth. Now, for a given client set $X_i = \{x_1, x_2, \ldots x_p\}$, the facility allocation is $f_{X_i} = \{f(x_1), f(x_2) \ldots f(x_p)\}$. The number of cost functions is equal to the number of possible facility assignment functions over set $X_i$. Computing the cost functions reduces to enumerating the permutations with repetitions of choosing $p$ elements from $n$. The number of such permutations are $n^p$. Thus each cost table has $n^p$ entries which is very high in practice. In order to compute whether a client is being covered by a facility or not, the dynamic program computes all pair shortest path distances of the input graph $G$. The

\(^1\)treewidth not more than 2
heuristic utilizes this already computed information.

According to the definition of assignment function \( f \), defined in Section 3.1.1, for a vertex \( x \in X_i \), \( f(x) \) is the opened facility that is closest to \( x \). For a valid cost function, the client-facility mapping obeys this relation. Our heuristic checks whether every possible facility configuration abide by the definition of assignment function \( f \) or not. For a given facility configuration \( f_{X_i} \), the heuristic checks whether there exits a client pair \( (x_a, x_b) \in X_i \) for which \( d(x_a, f(x_b)) < d(x_a, f(x_a)) \), where \( f(x_a), f(x_b) \in f_{X_i} \) and \( a \neq b \). Then it simply discards that facility configuration and moves on to process the next. Thus for each node it checks for \( n^p \) possible facility configuration, where \( p \) is at most \( k + 1 \), mentioned in Section 3.1.6 and discards the entry of the cost function into the cost table when the facility configuration for that cost function violates the definition of \( f \).

Through this technique, a great number of invalid entries are discarded and the size of the cost table reduces significantly. We ran a series of experiment which confirms this claim. The experiment results will be further discussed in detail in the next chapter. In the next section, we will discuss another heuristic technique that will reduce the number of cost functions at a node based on the Covering neighborhood of the node that remains at a proximity of the Covering radius.

### 3.2.2 Reductions of cost functions based on the Covering Neighborhood

Given a facility allocation and a client set, if for a client the closest facility is situated outside the covering radius, then no other facility will cover that client unless the client itself becomes a facility. So, for a client a potential facility to serve that client would be one of the members of it’s covering neighborhood. The covering neighborhood for client \( x \) can be defined as below-

\[
CN(x) = \{ v: d(x,v) < R_x, R_x = \text{Covering Radius of client } x \}
\]

Similarly, for any node \( i \), the covering neighborhood for the node would be the union of covering neighborhood of the clients in \( X_i \). Covering neighborhood of a node \( i \) can be defined as-
\[ CN(i) = \{ \bigcup_{k \in \text{size}(X_i)} CN(x_k) \} \]

For every facility that is situated outside the clients covering radius, a penalty is paid for the uncovered client. We can represent the set of the facility situated outside the covering radius of the client by a symbol \( \varepsilon \). Then for any node \( i \), we modify the set \( CN(i) \) by adding the \( \varepsilon \) symbol in to the set-

\[ CN(i) = \{ \{ \varepsilon \} \cup CN(i) \} \]

For any node \( i \), if we restrict the facility configuration permutation to the modified set \( CN(i) \), then the final permutation set will encapsulate all the necessary scenarios that will likely to produce the optimal solution. The number of permutations is directly related to the client’s covering radius. The bigger the covering radius, the larger is the set of permutations.

In this context, to reflect the changes on the number of facility configurations, we need to modify the assignment function. For any node \( i \), instead of \( f : X_i \rightarrow V \), the assignment function will be redefined as \( f : X_i \rightarrow CN(i) \).

We also need to modify the cost functions for different types of nice tree decomposition nodes to capture the inclusion of \( \varepsilon \) symbol.

**Leaf Node**

Continuing from the section 3.1.2, we need to add another case for the Leaf Node cost function definition.

**Case 3** When \( f(x) = \varepsilon \), then only a penalty cost need to be paid as \( f(x) \) represents the set of facilities situated outside the client’s covering radius. The cost function definition becomes-

\[ \Phi(\{x\}, \varepsilon) = b_x \]
**Introduce Node**

Continuing from section 3.1.3, we need to add another case for the introduce node, where we set $f(x_1) = \varepsilon$ to represent the set of facility situated outside the covering radius of the client.

**Case 3** If $f(x_1) = \varepsilon$, then we are considering that $x_1$ is uncovered. So, only a penalty cost will occur in this case (without opening any facility).

$$\Phi(X_i, (\varepsilon)) = \Phi(X_j, (g)) + b_{x_1}$$

**Forget Node**

As we only need to find the minimum cost function from the child node, we don’t need to modify the cost function definition in this case.

**Join Node**

In case of join node, we also don’t need to modify the cost function definition. We only need to modify the $\kappa(x, f)$ function which computes the covering cost for a pair $(x, f(x))$. This function is used to compute the repetition cost for the join node’s children. The modified definition is given below-

$$\kappa(x, f) = \sum_{x \in X} \rho(x, f(x)) + \sum_{y \in f(X)} \phi(y)$$

Only the second part of the function $\kappa(x, f)$ is modified where $\phi(y)$ is another function which can be defined as-

$$\phi(y) = \begin{cases} 
C_y, & \text{if } y \text{ is a facility covering } x \\
0, & \text{if } y = \varepsilon
\end{cases}$$

By using this technique, the number of cost functions per cost table can be greatly reduced if the client covering radius is small. In practice this technique in coalition with the heuristic improves
the running time significantly.

### 3.2.3 Pruning Using Branch and Bound

The Branch and Bound technique is a well known method for solving various optimization problems, especially in discrete and combinatorial optimization. This method was first proposed by A. H. Land and A. G. Doig [20]. This approach is based on the principle that the feasible solution space can be partitioned into smaller subsets of solutions. Then each of these subsets can be evaluated until the best possible solution is found. This technique is often affiliated with the non integer solution where the integer constraints are relaxed to get a solution.

Considering the maximization problems (in our context) suppose the goal is to find the maximization of a function $f(x)$ of variables $(x_1 \ldots x_n)$ over a region of feasible solution, $S$. The first step of the Branch and Bound procedure is splitting or branching, where given $S$ (the feasible solution space), this procedure will generate two or more subproblems $S_1$, $S_2$ from $S$ generally by adding new constraints. A subproblem hence corresponds to a subspace of the original solution space. This recursive process defines a search tree. The solution of a problem is described as a search through a search tree, in which the root node corresponds to the original problem to be solved, and other nodes corresponds to subproblem which satisfy the same constraints as the root and additionally a number of others. The next step in the Branch and Bound technique is called the bounding process where to each node in the tree a bounding function associates a real number called bound for the node. For leaves the bound equals the value of the corresponding solution, whereas for internal nodes the value is a lower bound for the value of any solution in the subspace corresponding to the node.

In order for the Branch and Bound technique to work, a feasible solution to the entire problem needs to be computed beforehand. For any integer program, a feasible solution can be produced by relaxing it’s integral constraints and solve the linear program. In other cases, the feasible solution can be produced by some advanced heuristic. This value will be used as the current best solution.
and will be called the incumbent. In each iteration of a Branch and Bound algorithm, a node is selected for exploration from the pool of unexplored nodes. Two or more children of the node are constructed through the addition of constraints to the subproblem of the node. In this way the subspace is divided into smaller subspaces. For each of these the bound for the node is calculated. In case the bound is the value of an optimal solution, the value is compared to the incumbent, and the best solution and its value are kept. If the bound is no better than the incumbent, the subproblem is discarded or pruned, since no feasible solution of the subproblem can be better than the incumbent. In case no feasible solutions to the subproblem exits, the subproblem is also discarded. Otherwise that subproblem node is added to pool of unexplored nodes.

We will apply a modified version of this Branch and Bound Technique to prune the redundant cost functions in our algorithm. A similar branch and bound approach was used in the study of protein chain lattice fitting problem by Thomas Dallas as part of his M.Sc thesis [30]. As we discussed earlier, Branch and Bound Techniques has two steps, Splitting or Branching and Bounding. As we used a Tree Decomposition, we do not have to use the Branching step as the subproblems are already defined by the Tree Decomposition nodes. The general steps for our Branch and Bound Technique are-

- Find a feasible solution (an upper Bound) of the problem.
- Compute a lower bound for each cost function for every node of the Tree Decomposition.
- Compare the lower bound and the upper bound. If the lower bound > upper bound, then discard the cost function.

In Figure 3.3, a symbolic representation of a Tree Decomposition is shown. The root node of the Tree Decomposition and another arbitrary node \( i \) at the middle is present in this figure. The node vertex set \( X_i \) of node \( i \) contains two vertices \( x_1 \) and \( x_2 \). The subtree rooted at node \( i \) is \( T_i \). The subproblem defined by the subtree between the root node and node \( i \) is denoted by \( G_s \). Vertices \( a_1, a_2 \) and \( a_3 \) are part of \( G_s \). The subproblem defined by the subtree at the root node denotes the entire
We chose CPLEX, a software tool designed to solve integer programming problems (more on section 3.3.1), to compute the feasible solution $F$ for the problem. CPLEX solves the relaxed version of the Covering Problem as described in section 1.1. This solution value $F$ will provide an upper bound for the Covering problem on the input graph.

As discussed earlier, for each Tree Decomposition, if the treewidth is $k$, then there can be $n^{k+1}$ cost functions in the worst case. For each such cost function, we will compute a lower bound for the whole problem. For example, in Figure 3.3 at node $i$, there can be $n^2$ number of cost functions, as $|X_i| = 2$. Each such cost function will produce a cost value $C$ (through dynamic programming computation) that represents the solution value of the Covering problem restricted to the subtree $T_i$ rooted at $i$. To compute the lower bound, we need to have a feasible solution $S$ of Covering Problem on the subproblem defined by $G_s$. Once $S$ is computed, for each cost function (with cost value $C$)
at node $i$, the lower bound will be, $S + C$. Because $S$ represents the feasible solution for $G_s$ and $C$ represents the cost value for the subgraph $G_{T_i}$ and $G_s + G_{T_i} = G$. If for this specific cost function the lower bound > upper bound, then the cost function is discarded, as it will never produce an optimal solution.

We use CPLEX like before, to solve the relaxation of the Covering problem restricted on the subgraph $G_s$ to compute $S$. But in order to compute the correct lower bound, we need to remove the overlapping cost that may be included in $C$, the cost function value for any cost function at node $i$. In Figure 3.3, at node $i$ with $X_i = (x_1, x_2)$, suppose a generated cost function permutation is $(b_1, b_2)$. That is the at $b_1$ and $b_2$ facilities are opened and they are serving (or paying the penalty if the clients are uncovered) the clients $x_1, x_2$. These facilities $b_1, b_2$ may very well cover clients that are outside the subgraph $G_{T_i}$. Suppose $a_1, a_2$ and $a_3$ are such clients, who are part of the subgraph $G_s$ but covered by the facilities $b_1$ and $b_2$. For this cost function $(b_1, b_2)$, the solution value $C$ of the Covering problem restricted at $G_{T_i}$, will include the clients $a_1, a_2$ and $a_3$. While computing $S$, if these clients $a_1, a_2$ and $a_3$ are still part of the problem model defined by the subgraph $G_s$, then the solution may open facilities in any one these clients, which will be an extra cost as all these clients are already covered by the solution $C$. So, to remove such overlapping extra cost from $S$, for any node $i$, we will compute the set of clients that are outside the subtree $T_i$ covered by the node vertex set $X_i$. Because the vertices in a node vertex set are undoubtedly the closest vertex to the clients present outside the subtree defined by that node (described in 3.1.7). So, the clients that are uncovered by the node vertex set $X_i$ are also uncovered by any of the vertices inside $T_i$. Hence, for any vertices (facilities) present in the cost function permutation, the set of covered clients (by those facilities), will be a subset of the set of clients covered by the node vertex set. Removal of the set of covered clients(covered by the node vertex set) from the subgraph $G_s$, while computing $S$, the produced lower bound will not be as tight as it could have been if the set of clients were covered by the facilities present in the cost function permutation. But in that case we have to compute $S$ for each cost function at node $i$. In our way, we need to compute $S$ only once per node.
The Branch and Bound technique prunes more cost functions than the pruning heuristic in most cases. But the computation of the bounds for each node is costly. More is discussed on the implementation section of this chapter.

3.3 Hybrid and Parallel Algorithm

We incorporated all the techniques with the dynamic programming algorithm described in the earlier section and wrote a program to run experiments. It performed really well comparing with the program with no heuristic techniques. We then used CPLEX, a software tool developed by IBM to check the correctness of our program. But we found that CPLEX is very efficient when it solves the Covering problem on the same data set and is much quicker than our algorithm. To reduce the difference between the running times, we then designed a Hybrid algorithm and a Parallel programming algorithm which uses the CPLEX and CONDOR GRID to facilitate the execution of the algorithm. We will discuss these algorithms in the following sections.

3.3.1 A Hybrid algorithm with CPLEX and Dynamic Programming

IBM ILOG CPLEX Optimization Studio (often informally referred to simply as CPLEX) is an optimization software package developed by IBM. The IBM ILOG CPLEX Optimizer solves integer programming problems, very large linear programming problems using either primal or dual variants of the simplex method or the barrier interior point method, convex quadratic programming problems, and convex quadratically constrained problems.

We used CPLEX mainly to check the correctness of our algorithm. But we found out that for our data set (more discussed in 4.2.1) CPLEX runs very fast. We incorporated all the techniques (heuristics and implementation techniques) in our algorithm, it did improve the running time considerably from last time, but still it’s quite slow when compared to CPLEX. We then decided to incorporate the services of the CPLEX module in our algorithm, thus developing a Hybrid algorithm which has
both dynamic programming and CPLEX.

The reason the dynamic program is slow is because for each Tree Decomposition node it has to process a large number of cost functions in case of large graphs. The idea is to use CPLEX to solve a bottom chunk of the Tree Decomposition so that the dynamic program does not have to get stuck with a huge number of cost functions. The size of the chunk or subproblem will be supplied as command line parameter to the algorithm.

In Figure 3.4 the solution steps of a symbolic Tree Decomposition by the Hybrid algorithm is shown. The intuition behind this design is that, if the dynamic program has less number of nodes to work with then this hybrid algorithm will be faster than CPLEX in case of large graph instances.

Figure 3.4: Solving a Tree Decomposition with the Hybrid Algorithm
The CPLEX part of this algorithm has two steps—

- Node selection
- Subproblem Solution

In the Node selection step, the nodes with the desired subproblem size is selected to be solved by CPLEX. As mentioned earlier, this subproblem size is controlled by a parameter called *target subproblem size*. Given a target subproblem size, the program traverses the Tree Decomposition the same way (bottom up traversal) it traverses for the dynamic programming algorithm. It also keeps track of the subproblem size defined by the subtree rooted at the node it is visiting. While traversing on a branch the most recent node that has a subproblem size less than or equal to target subproblem size is selected for CPLEX. In Figure 3.6, on the leftmost branch, node $a$ is the last node or the most recent node that has a subproblem size less than or equal to target subproblem size. The parent node $i$ (the next node visited by the traversal module) has a subproblem size greater than the target subproblem size. So, only node $a$ is selected to be processed by the CPLEX module, not node $i$. Similarly, node $b$, $c$ and $d$ is selected to be processed by the CPLEX module.

The next step is the solution of the Subproblem defined by the subtree rooted at the selected node. Though CPLEX can solve the subproblem optimally in a short time, a cost table with cost function values still needs to be generated because without it the dynamic program cannot resume its operation at the upper level nodes. In this respect, initially the CPLEX instance is modeled after the Covering Problem integer program (as described in 1.1) defined by the subproblem at the selected node. Then for every cost function, variables and constraints are deleted from the CPLEX instance for the set of clients that are already covered by the facilities in the cost function. CPLEX then solves the downsized integer program. The cost function value will include the opening (or penalty) costs of the facilities present in the cost function and the cost value produced by CPLEX which denotes the subproblem solution value at that node. Thus at the end the selected node will have a cost table with cost function values that are produced by CPLEX.
This is a simple enough algorithm, but in practice we had the most success with this algorithm. The experiment results and evaluation are described in detail in chapter 4.

### 3.3.2 A Parallel Algorithm for Solving Covering Problem

One of the unique properties of Tree Decomposition is its ability to divide a problem into smaller independent subproblems. At each Tree Decomposition node an independent subproblem can be defined by the set of vertices in the subtree rooted at that node. The simplest parallelism that can be introduced in this respect is to solve each of these subproblems on a separate machine and then consolidate the result at the root node to get the solution. As the problems are independent, the allocated machines does not need to interact with each other. They solve the assigned problem and report the solution back to the source. Our algorithm uses the same idea. The execution of our algorithm depends heavily on the CONDOR GRID installed in our CS network which ensures a high performance in terms of job scheduling, efficient utilization of available resources.

Condor is a open source high-throughput software framework for distributed parallelization of computationally intensive task. As an HTC (High-throughput computing) system Condor is very robust and reliable when running a task using many computing resources. This high throughput computing also ensures an efficient execution of a task over a long time with relatively small overhead. Condor accepts an implementation of an algorithm (a program) as a "job" and assigns this job to a remote machine for execution from its pool of available machines. It allows a user to submit multiple jobs at the same time. Condor offers a huge advantage by efficiently using the idle machines in a network. Condor handles the submission and scheduling of jobs, connection to remote machines, remote system calls and reporting the output back to source machine where the job was submitted. Our algorithm utilizes these services of Condor to parallelize the solution of subproblems.

This algorithm follows from the Hybrid algorithm discussed earlier. In Hybrid algorithm, we solved a chunk of subproblems using CPLEX then the rest of the Tree Decomposition nodes are processed
by the Dynamic program. We will follow the same idea but instead of solving the subproblems with CPLEX in one machine, each of the subproblems will be solved in a separate machine. The output results are reported back to the source and the source machine will then start the dynamic program on the rest of the Tree Decomposition as it has the subproblems solutions available from the remote machines. So, basically there are three steps for this algorithm:

- Subproblem writing
- Subproblem Solving
- Accumulation of the solutions

In the subproblem writing step, the subproblems that are to be solved by CPLEX are written in text files. The next step, the subproblem solving step submits each subproblem file as a job to the Condor Grid. The Condor then assigns a suitable machine from it’s pool of available machines to solve the subproblem. The remote machines solves the problem and writes the solution information into a file. After the source machine has all the solution files, the Accumulation of the solutions step begins where the algorithm resumes the dynamic program for the Tree Decomposition nodes for whom the subproblem sizes are greater than the target subproblem size (as described in 3.3.1). The implementation details are discussed in the implementation section of this chapter.

When compared to the Hybrid Algorithm, the running time for this technique is almost similar to that of the Hybrid. This is because CPLEX is very fast to solve Covering problem even on a large instance on a single machine. Though we hoped that dividing the workload among several machines would outweigh solving the Covering problem by CPLEX on a single machine however in reality the performance improvement was negligible overall. For this reason we used the Hybrid Algorithm for further experimentation. But we conjecture that this parallel algorithm will be effective for problems with larger integrality gap that are difficult to solve by CPLEX.
3.4 Implementation

As mentioned earlier in chapter 2, for implementation we used Python 2.7 as a programming language. In the following sections, we will discuss the major data structures of the dynamic program and then we will briefly describe the steps our program follows to compute the optimal solution. In the later sections we will discuss several techniques that are used to accelerate the program.

3.4.1 Dynamic Program

**Data Structures** In order to represent each node in the tree decomposition, we wrote a class `Node_with_table` with different data attributes. As every node is associated with a cost table, we simply added the cost table as a data attribute in the class. Each cost table is a dictionary, a hash table type data structure of Python. For a tree decomposition node $i$ and its vertex set $X_i$, this dictionary contains the entries where the key is the facility allocation $f$ and the value is the cost function $\Phi(X_i,f)$. We used a two dimensional list, an array type data structure of Python, to represent the pair shortest path weight matrix generated by the Floyd Warshall algorithm.

**Brief Description** First, our dynamic program traverses the tree bottom up recursively. Starting from the root node, the recursive function traverses the child list and appends the child nodes into a stack. Then, the function keeps calling itself with the top node of the stack as the root for the next iteration. If it reached a target node (leaf node, or a node whose children has already been traversed), it deletes that node from the stack and the function returns. Whenever each of the recursive call is terminated, that means either a leaf node or a node (top node of the stack) whose children has already been traversed is reached at that point. So, through the recursive call terminations, the module keeps moving upward through the tree decomposition towards the root.

Now, before the function return statement, the program invokes another module that generates one or more nice tree decomposition nodes till it reaches up to the parent node. If the target node $i$ is a
original tree decomposition leaf node with the cardinality of the vertex set greater than 1, then this module generates a nice tree decomposition leaf node $j$ with a single vertex chosen randomly from the original leaf node’s vertex set. Then, it treats the original tree decomposition leaf node just as a regular introduce node where the introduce vertex set is $= \{X_i \setminus X_j\}$, where $i$ is the original tree decomposition leaf node and $j$ is the newly generated nice tree decomposition leaf node.

If the target node $i$ is a node whose child $j$ has been traversed, then the module generates a nice tree decomposition forget node where the forgotten vertices are $= \{X_j \setminus X_i\}$. Then the module treats the already present node $i$ as an introduce node, where the introduce vertex set is $= \{X_i \setminus X_j\}$. Whenever the module generates a nice tree decomposition nodes (or treat like one), invokes the module for processing that specific node. For example, if the node is a introduce node, the program will invoke the process_introduce_node module. Then the node processing module invokes the permutation module to enumerate all possible facility allocation functions for the client set in the node. Then, for each facility allocation function, it computes the cost value according to the cost function definitions of that node. After computing the cost value, it inserts the cost entry in the cost table, where it stores the facility allocation function as a key and the cost value as value in the dictionary.

As the traversal module is working its way upwards and reaches the root, the root node’s cost table then has all possible solutions for the Covering problem. We take the solution with minimum cost value. The solution is partial in the sense that only the value of the optimal objective is known, and the facility allocation function for the vertices of the root node is known. However, the information about facilities covering the other vertices of the graph needs to be recovered from the cost functions at tree decomposition nodes below the root.

In this respect, the program keeps another table called cost_backtrack_table(a dictionary) for each node. While computing each possible cost function for a client set in a node, the program accesses the child node’s cost function for that specific facility allocation. This cost_backtrack_table stores the child node’s cost function keyed by the current node’s cost function for each possible facility
allocation at a node. We developed a solution backtrack module where given an optimal solution at the root (contains cost value and cost function for the root node), it recursively accesses the node’s cost_backtrack_table, uses the solution cost function as the key and finds the child node’s cost function which was used to compute current node’s cost function. The module keeps adding the facility allocation found from the child node’s cost function in a list. It keeps calling itself until it reaches a leaf node. At the end it returns with the opened facility list it has discovered from the nodes below the root.

But this approach to recover the full solution proved really costly as each cost_backtrack_table occupies a considerable amount of memory which leads to a memory exhaustion. To counter this problem we found a simple yet efficient way to discover the solution. At each cost table against a cost function the program will store a pair of values where the first value is the cost function value and second value is a list of integer which denotes the indices (of the vertex set of the input graph) of all opened facilities for that cost function. At a leaf node this set of opened facilities will contain only one vertex (leaf node vertex). At a parent node, the set of opened facilities indices will contain the set union of the facilities that are opened by the cost function and set of opened facilities from the child node for that respective cost function. In this way at the root the optimal solution cost tuple will contain all the facilities that are opened for this solution branch. We just have to retrieve the facilities from the vertex set by the integer indices.

3.4.2 Pruning Heuristic Module

Given a client set and it’s assigned permuted facility set, for each client this module checks the distances between the client and each facility in the permuted facility set. If it finds a facility that is closer to the client than the already assigned one, then the module discards the current facility allocation function, and as a result, no cost function will be computed for it.
3.4.3 Branch and Bound Technique

There is a preprocessing step for executing the dynamic programming algorithm equipped with the Branch and Bound Technique. We need to compute a upper bound (feasible solution) of the entire problem defined the input graph and a lower bound for every cost function generated at each node. We used the CPLEX module to compute both bounds. Though CPLEX is very fast to compute the bounds, given a large graph instance the cumulative time to compute these bounds are quite high. So, for a faster execution of our program, we precomputed these bounds (solving relaxation of subproblems) for each node and write them into a file using a sorted list. Whenever the program is processing a introduce node or a join node (because only for these nodes we include the pruning by Branch and Bound, for leaf and forget node this technique does not apply) rather than computing the relaxation of the subproblem defined by the subtree in between the given node and the root, it reads the corresponding solution value from the file. It resumes normal execution after that.

In case of a introduce or a join node, after computing the lower bound for the cost function by adding the cost function objective value with the retrieved subproblem (defined by the Tree Decomposition without the subtree rooted at the given node) solution value, the program checks whether the lower bound is greater than the upper bound (feasible solution). If greater then the cost function is discarded or pruned else the program inserts the cost function into the cost table with it’s solution value.

3.4.4 Hybrid Algorithm

To implement the Hybrid Algorithm we developed two new modules-

- modified traversal module
- a subproblem solver module with CPLEX

We introduced a new parameter called target subproblem size. This parameter is used for the selection of nodes for which the subproblems defined by their subtree will be solved by CPLEX. If the
subproblem size of a node is less than or equal to the target subproblem size, then the subproblem will be solved by CPLEX, else it will be solved by the dynamic program.

We modified the traversal module described in 3.4.1 to support the algorithm requirement. As it traverses the tree recursively using a bottom up approach, it also keeps track of the size of the subtree at each node. Because this is a bottom up traversal, if the first node that has a subtree (subproblem) whose size is greater than the target subproblem size (described in 3.2.3), then it’s child must be the last node whose subtree (subproblem) size is less than or equal to target subproblem size. To solve the subproblem defined by the subtree rooted at this selected node, it calls a subproblem solver module with CPLEX. For nodes that has the subtree size is greater than the target subtree size, the program resumes the dynamic programming operation as discussed in 3.4.1

We used the CPLEX 12.1 PYTHON API to implement the subproblem solver module. Given the subproblem definition, this module fills up a CPLEX instance with necessary variable and constraints to imitate the integer program for Covering problem restricted to this subproblem. For each cost function permutation, a set of clients that are covered by the facilities opened by the cost function is computed. The variables and constraints corresponding to those covered clients are deleted from the CPLEX model. Then this reduced model is solved optimally by CPLEX. The cost function value contains this curtailed subproblem solution value and the facility opening cost (or penalty) from the facilities present in the cost function. At the end of this module, the cost table is returned.

3.4.5 Parallel Algorithm

To implement the parallel programming algorithm, we developed three separate programs:

- **Subproblem Writer**
- **Subproblem Solver**
- **Accumulator**
The program structure for the Subproblem Writer is similar to that of Hybrid Algorithm program. It traverses the tree using the modified traversal algorithm described earlier. It uses the target subproblem size parameter the same way Hybrid algorithm does to select the subproblems to be solved by CPLEX module. Once the subproblems are selected, it writes their definition in text files. Then for each subproblem files it generates a MakeFlow rule which is then added into a Makeflow script. MakeFlow is a workflow engine for executing large complex workflows on clusters, clouds and grids. It accepts a specification of a large amount of work to be performed, and can be made to submit these jobs to some already installed grid system. As we have a CONDOR GRID installed in our system, the MakeFlow tool is used to generate specifications for a series of jobs and the submission of the Makeflow script (script_name.makeflow) to Condor. Each MakeFlow rule specifies a target file, a set of source files needed to create it and a command that generates the target file from the source files. In our case, in each rule, the target file is a cost table associated with the current subproblem, the set of source files are all the external python modules imported to run the program and the command is python runtime command which instructs the python interpreter to execute the Subproblem Solver program with the given subproblem file name. At the end, this program will generate a MakeFlow script called "Coverage.makeflow" which will have a rule for each subproblem that are to be solved by CPLEX.

When this Coverage.makeflow script is submitted to Condor grid, for each rule the program Subproblem Solver is called with a subproblem file name. This program reads the subproblem definitions from the subproblem file (given the subproblem file name) and calls the subproblem solver module with CPLEX (as described in 3.4.4) to solve the subproblem. This program returns the cost table containing the cost function values associated with this subproblem. Once the cost table is returned by this program, Condor automatically returns this cost table back to the source machine, where it is stored in a text file.

The Accumulator program is activated once all the necessary subproblem cost tables are returned by Condor. Because once the cost tables are available, this program can start the dynamic program
for those nodes that has subtree (subproblem) size is greater than the target subproblem size. It then follows the dynamic program and finds the solution from the root.

In practice the brute force approach is slow even for small graphs. During Implementation we developed some tricks and techniques to speed up the algorithm. In the following sections, we will discuss these techniques.

3.4.6 Cost Table Reduction Technique

The brute force approach has proven to be very expensive as it has a very high storage space (memory) requirements. This is mainly because of the exponential size of the cost tables associated with each tree decomposition node. Whilst the pruning heuristic reduces a significant number of entries from the tables, for larger graphs the memory overhead for these cost tables are still high. In this respect we developed a technique which requires only a few number of cost tables for the entire dynamic program.

As mentioned earlier in Section 3.4.1, the bottom up traversal module keeps traversing a node’s children recursively until it reaches a leaf node or a node whose children has already been visited. Then it moves upward through recursive call terminations. So depending on the child list of a node, the module moves from branch to branch of the tree decomposition. In this technique, rather than keeping a separate cost table for each node, we keep a pair of cost tables (parent_node_cost_table and child_node_cost_table) for a child node to parent node transition. As described in Section 3.4.1, there can be more than one nice tree decomposition nodes in a child to parent transition. So, we reuse this pair of cost table for each child-parent pair. Once a node is being processed, initially it’s cost function entries are store in the parent_node_cost_table. After the node is processed, it empties the cost table parent_node_cost_table after it has been copied into child_node_cost_table as now the processed node becomes a child while it’s parent node is being processed. At the end of the transition (for the final child-parent pair), the module keeps the parent_node’s_cost_table into
a `cost_tables` dictionary. It empties the child_node’s_cost_table as it is no longer required. Then the traversal module moves onto a new node. If the new node is a leaf node, then the program executes the steps described earlier for the transition from the leaf node to introduce node. After the transition is finished, it keeps the cost table of the parent node (in this case the introduce node) into the `cost_tables` dictionary. If the node is any other node except a leaf node, then the node’s child has already been processed and it’s cost table must be stored in the `cost_tables` dictionary. It copies that specific cost table into the child_node’s_cost_table and then deletes it from the `cost_tables` dictionary. At this point the parent_node’s_cost_table is blank. The module repeats the same steps including reusing these two cost tables for this new transition.

### 3.4.7 Bounding the Assignment Function

As we have described in section 3.2.2, for a node \( i \), the modified assignment function \( f, f : X_i \rightarrow CN(i) \), where \( CN(i) \) is the covering neighborhood of node \( i \), now enumerates all vertices of the set \( CN(i) \), thus reducing a great number of redundant cost functions. Even then, it is possible to have may cost functions that are generated through the process described in 3.2.2 only to be discarded by the pruning heuristic. In this technique, we will discard some of those cost functions before they are generated by the assignment function \( f \).

This technique follows from the Pruning Heuristic described in Section 3.2.1 which checks the distances between the clients and the facilities (assigned by the \( f \) function) and discards the facility configuration if the facility assigned to a client is not closest to it. Using the triangle inequality we can extend this distance checking technique to assign a upper and lower bound on the assignment function \( f \). This way, vertices that are outside the upper and lower bound will not be visited while processing an introduce node.

In the nice tree decomposition, the client set at the child node is a subset of the parent node. As we
are using a bottom up dynamic program, when the program processes the parent node, the subset which is equal to the child node’s client set is already assigned some facilities whose cost function entries can be found on the child node’s cost table. Currently while generating the facility configurations (by the assignment function $f$), the program generates all possible configurations and then applies the Pruning Heuristic to get rid of the invalid configurations. In this technique we will use the already allocated clients to apply bounds on the non-allocated clients.

We will describe the bounding technique in the context of figure 3.6 where a parent node and a child node is shown, where the parent node $i$ is an introduce node and the child node $j$ is a leaf node. The client set of node $j$ is $X_j = \{x_1\}$ and the client set of node $i$ is $X_i = \{x_1, x_2\}$. So the common element between node $i$ and node $j$ is $x_1$. The dynamic program will process the leaf node before the parent node and will assign a facility for every valid permutations. Let $f(x_1)$ represent the facility assigned to $x_1$.

While processing the parent node $i$, we will use the allocation of client $x_1$ to apply a upper and lower bound on the assignment function $f$ for client $x_2$. In the below figure, a symbolic representation of the distances between clients $\{x_1, x_2\}$ and their assigned facilities $\{f(x_1), f(x_2)\}$. here $f(x_2)$ represents the set of potential facilities for client $x_2$. 

Figure 3.5: Parent and child with common elements
The client $x_1$ is already served by $f(x_1)$. The upper bound that will be applied to $f(x_2)$ follows directly from the Pruning technique. The distance $d(x_2, f(x_2))$ must be smaller than the distance $d(x_2, f(x_1))$. Otherwise $x_2$ can be served by $f(x_1)$. So the upper bound on $f(x_2)$ can be defined as:

$$d(x_2, f(x_2)) \leq d(x_2, f(x_1))$$

Now, using the triangle inequality, we can derive the following relation from the triangle formed by $x_2, x_1$ and $f(x_2)$ from the figure 3.6-

$$d(x_2, f(x_2)) + d(x_2, x_1) \geq d(f(x_2), x_1)$$

Again, in order to become a valid configuration the distance $d(f(x_2), x_1)$ must be greater than the distance $d(x_1, f(x_1))$. So, we can rewrite the previous relation and define the lower bound on $f(x_2)$ as below -

$$d(x_2, f(x_2)) + d(x_2, x_1) \geq d(x_1, f(x_1)),$$

$$d(x_2, f(x_2)) \geq d(x_1, f(x_1)) - d(x_2, x_1)$$

So facility assignment function $f(x_2)$ will permute from the set of vertices for which the distance between them and the client $x_2$ falls between the upper and lower bound. The rest of the vertices will be discarded.
These bounds can be extended for client sets with any number of clients. For example, for any node \( k \), if the client set is \( X_k = \{x_1, x_2, x_3\} \), the fixed facility allocation for \( x_2 \) and \( x_3 \) is \( f(x_2) \) and \( f(x_3) \), then the upper bound on \( f(x_1) \), the potential facility for \( x_1 \) is -

\[
d(x_1, f(x_1)) \leq \min_{e>1} d(x_1, f(x_e))
\]

Similarly using the same triangle inequality technique described above the lower bound on \( f(x_1) \) is,

\[
d(x_1, f(x_1)) \geq \max_{e>1} \{d(x_e, f(x_e)) - d(x_e, x_1)\}
\]

Earlier, for a tree node \( i \) where \( Xi \) contains \( p \) number of vertices, the program used to generate \( n^p \) number of facility configurations where \( n \) denotes the number of vertices in the graph. Now with this bounding technique the program is generating \( l^p \) entries for some number \( l \leq n \), which depends on the upper and lower bound at that node. The program still uses the pruning technique to validate the permutations, as both these bounds are loose bounds. In practice, the combination of these techniques produces a much better result than before.

### 3.4.8 Balancing the Height of the Tree Decomposition

The Tree Decompositions produced by the Tree Decomposition heuristics are not height balanced. As a result sometimes the root of the Tree Decomposition can be located in the longest branch of the Tree Decomposition. As we described in 3.4.3, in a branch the number of cost tables is depended on the height of that branch from leaf to root. The longer is the height, the more number of cost tables are kept in the memory. As there can be numerous branches spawned from the root node, we can’t chose an arbitrary root because it might reduce the length or height of a particular branch but might increase the height of some other branch spawned from it.

One way to find such a root is to find the center of a Tree Decomposition and set it as root. For a
center $a$ of a Tree Decomposition, the following relation holds-

$$\min_{x \in I} \max_{y \in I} d(x, y)$$

Where $d(x, y)$ is shortest distance between $x$ and $y$, and $I$ is the node set of the Tree Decomposition. There is a simple algorithm for finding the center of a tree proposed by Murdasov [23]. We followed the same idea in case of a Tree Decomposition. The steps of our algorithm is given below-

- Select an arbitrary leaf node $i$ from the Tree Decomposition.
- Find the most remote node from $i$ using a depth first search. Suppose the remote node is $j$.
- Find a node $p$ that is furthest from $j$ using a depth first search (this operation will yield the diameter of the Tree Decomposition).
- The node $t$ for which the equality

$$d(j, t) = d(t, p)$$

holds is the center of the Tree Decomposition. Make this node $t$ as the new root.

Figure 3.7: Balancing the Height of a Tree Decomposition

A Tree Decomposition of a graph drawn in Figure 2.1 is shown in the left hand side. In the right hand side, another Tree Decomposition of the same graph is shown with it’s center as the new root.
In practice this technique reduced the number of cost tables that are kept in the memory at a given time.

### 3.4.9 Using a Modified Dijkstra’s algorithm to Compute Shortest Path

As discussed in 3.4.1, the shortest path matrix (a two dimensional list) is computed with the Floyd-Warshall algorithm which has a complexity $O(n^3)$. For large graph instances, the computation for the shortest path matrix using this algorithm is very costly in terms of running time. We decided to run modified version of dijkstra’s algorithm (complexity $O(n^2)$) for every vertex of the graph to fill the shortest path matrix.

The modification follows from the improvement discussed in 3.2.2 where the set of potential facilities for a client is restricted to the covering neighborhood of that client. We stopped the execution of the dijkstra’s algorithm when the current distance estimate between the source vertex and any other vertex is greater than the client radius. So, for a client, only the shortest path between the facilities inside the client radius and the client are computed through this modification. Also, to make it faster we used a min heap as a data structure for the queue in the algorithm.

Using this modified dijkstra’s algorithm helped us as it reduced the total running time greatly compared to the Floyd-Warshall algorithm. It does not have any effect on the dynamic program as the dynamic program run time is computed by subtracting the total running time from the shortest path running time. But it made the execution of the program really fast.

In the next Chapter we will present the empirical data of several experiments for Tree Decomposition computation and for solving the Covering problem using our algorithms. We will analyze the data and will discuss the findings of our research.
Chapter 4

Experiments

In this chapter we present the experimental results of the algorithms to solve the standard Covering problem. In this research, we used only randomly generated graphs for our experiments because it is expensive to gather and interpret the wireless network data and generate graph representation from them. We plan to use the real time wireless data for experiments in future. In section 4.1, we discuss the experiment results of the construction of Tree Decomposition by the heuristics described in Chapter 2. In the next section (4.2), we will present the experiment results of three algorithms which solves the standard Covering problem. In section 4.3, we analyze the results and find inference from the evaluation.

All the algorithms are implemented using Python 2.7. We used Python because it is easy to program and very quick to implement any algorithm in it. All the experiments presented in this chapter were conducted on 3.00 GHz Pentium(R) 4, 64 bit processor with 1 GB RAM in the Linux CentOs environment.

4.1 Tree Decomposition Experiments

For Tree Decomposition experiments, we will use the following three algorithms that are described in Chapter 2 -

- Minimum Separator Vertex Set Heuristic (MSVS)
- Clique Tree Heuristic (CLQT)
- Random Separator Vertex Set Heuristic (RSVS)
**4.1.1 Data Sets**

For these experiments, we used unit disk graphs to randomly generate dense graphs and series parallel graphs to randomly generate sparse graphs. The procedure for generating these graphs are discussed below.

**Unit Disk Graph** In geometric graph theory, a unit disk graph is an intersection graph of a set of unit circles in the Euclidean plane; each vertex corresponds to a circle, and an edge is placed between two vertices when the corresponding circles intersect [6]. The steps for generating random unit disk graph is given below-

- We specified the total number of vertices \(|V|\) of the graph.
- For each vertex we randomly generate a \((x, y)\) co-ordinate and assign it to that vertex.
- For every pair of vertex we compute the Euclidean distance \(d\) between them, and then check whether \(d < T\), where \(T\) is the threshold value that we defined earlier. If \(d < T\), then we insert an edge between the vertices if not then we move to the next pair of vertices.

**Series Parallel Graph** In graph theory, a *series parallel graph* is a graph of two distinguished vertices called source \((s)\) and sink \((t)\) which denotes two terminals (end points) of the graph and formed recursively using two composition(series composition or parallel composition) operation. A two terminal series parallel graph \(G\) with terminals \(s\) and \(t\) can be produced by a sequence of the following operations:

1. Create a new graph, consisting of a single edge directed from \(s\) to \(t\).

2. Given two two-terminal series parallel graphs \(X\) and \(Y\), with terminals \(s_X, t_X, s_Y, \) and \(t_Y\), form a new graph \(G = P(X, Y)\) by identifying \(s = s_X = s_Y\) and \(t = t_X = t_Y\). This is known as the *parallel composition* of \(X\) and \(Y\).

3. Given two two-terminal series parallel graphs \(X\) and \(Y\), with terminals \(s_X, t_X, s_Y, \) and \(t_Y\), form a new graph \(G = S(X, Y)\) by identifying \(s = s_X, t_X = s_Y\), and \(t = t_Y\). This is known as the *series* composition of \(X\) and \(Y\).
This definition and the steps of composition follows from the work of David Eppstein [8] in his study of recognition of series parallel graphs.

Series-parallel graphs are a useful class of graphs. They are fairly simple to generate and allows easy proofs for many results. In particular, series-parallel graphs are a fertile testing ground for various conjectures. Also one of the main reason to use series parallel graphs in our experiments is that the treewidth for this class of graph is no more than 2, which proved to be an excellent test ground for our algorithms. Because it is expensive to run algorithms on a tree decomposition of higher treewidth.

Below we discuss the steps of generating random series-parallel graphs:

- We generate a series of random even number upto a given limit.

- For each even number $e$ we generate a two-terminal series-parallel subgraph where the number $e$ will define the number of nodes of the subgraph. In order to generate it, we will create a graph with two vertices and only one edge connecting them. We recursively add this graph to itself using either a series connection or a parallel connection until the the number $e$ is reached. We do this using a coin toss procedure. We generate a random number $r$, between 0 and 1. If $r > 0.5$, then we add the edge using series connection and if $r \leq 0.5$, we add the edge in parallel connection.

- After all the series-parallel subgraph/components are generated, we connect them either through series connection or parallel connection using the same coin toss procedure described earlier.

### 4.1.2 Tree Decomposition Experiment Results

In this section we present the experimental data of computing Tree Decomposition by the three algorithm mentioned above. Table 4.1 contains the empirical data of experiments done on the randomly generated dense graphs (unit disk graphs) and Table 4.2 contains data of experiments
done on randomly generated sparse graphs (series parallel graphs). The first and second column of the tables denotes the number of nodes and edges of the graph. The third, fourth and fifth column represents the average running time (in sec) of the algorithms Clique Tree Heuristic (CLQT), Minimum Separator Vertex Set Heuristic (MSVS), Random Separator Vertex Set Heuristic (RSVS). The sixth, seventh and eighth column of the tables represent the treewidth data of algorithm CLQT, MSVS and RSVS.

Table 4.1: Tree Decomposition Experiments on Random Dense Graphs

<table>
<thead>
<tr>
<th>Vertices</th>
<th>Edges</th>
<th>Runtime (Sec)</th>
<th>TreeWidth</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>CLQT</td>
<td>MSVS</td>
</tr>
<tr>
<td>50</td>
<td>396</td>
<td>0.085</td>
<td>32.34</td>
</tr>
<tr>
<td>75</td>
<td>836</td>
<td>0.272</td>
<td>546.65</td>
</tr>
<tr>
<td>100</td>
<td>1450</td>
<td>0.610</td>
<td>2591.72</td>
</tr>
<tr>
<td>150</td>
<td>3562</td>
<td>2.66</td>
<td>27826</td>
</tr>
<tr>
<td>200</td>
<td>2416</td>
<td>6.44</td>
<td>35973</td>
</tr>
<tr>
<td>250</td>
<td>2770</td>
<td>7.98</td>
<td>88416</td>
</tr>
</tbody>
</table>

It is evident after observing the data in the above tables that the Clique Tree Heuristic (CLQT) is significantly faster than the other two algorithms. The Minimum Separator Vertex set heuristic (MSVS) has the largest running time. This is because finding a minimum separator is really expensive. The Random Separator Heuristic (RSVS) fares better in terms of running time than MSVS. Because in RSVS, instead of finding separator for every pair of vertices, we randomly select a few pairs of vertices and then find the minimal separator for each of those selected pairs.

Table 4.2: Tree Decomposition Experiments on Random Sparse Graphs

<table>
<thead>
<tr>
<th>Vertices</th>
<th>Edges</th>
<th>Runtime (Sec)</th>
<th>TreeWidth</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>CLQT</td>
<td>MSVS</td>
</tr>
<tr>
<td>171</td>
<td>192</td>
<td>0.288</td>
<td>2081.24</td>
</tr>
<tr>
<td>264</td>
<td>303</td>
<td>1.018</td>
<td>4523.07</td>
</tr>
<tr>
<td>359</td>
<td>418</td>
<td>2.569</td>
<td>21134.39</td>
</tr>
<tr>
<td>435</td>
<td>495</td>
<td>4.491</td>
<td>61220.85</td>
</tr>
<tr>
<td>532</td>
<td>614</td>
<td>8.45</td>
<td>116251.83</td>
</tr>
<tr>
<td>612</td>
<td>699</td>
<td>12.77</td>
<td>47030.16</td>
</tr>
</tbody>
</table>

It is evident after observing the data in the above tables that the Clique Tree Heuristic (CLQT) is significantly faster than the other two algorithms. The Minimum Separator Vertex set heuristic (MSVS) has the largest running time. This is because finding a minimum separator is really expensive. The Random Separator Heuristic (RSVS) fares better in terms of running time than MSVS. Because in RSVS, instead of finding separator for every pair of vertices, we randomly select a few pairs of vertices and then find the minimal separator for each of those selected pairs.
Though the CLQT has the least running time, the quality of the Tree Decomposition (Treewidth) is not minimal all the time. MSVS algorithm has the ability to produce the best (among these three algorithms) quality Tree Decomposition almost at every run. RSVS algorithm is produces a decent quality Tree Decomposition, some times even better than CLQT algorithm. After evaluating these results, we decided to chose CLQT for our primary algorithm to generate Tree Decomposition for our dynamic program. Though sometimes the Treewidth is not minimal in the Tree Decomposition produced by CLQT, but after a few runs of CLQT on the same graph, eventually we were able to get the desired Tree Decomposition with the desired Treewidth (in case of series parallel graphs, the desired treewidth is 2).

4.2 Covering Problem Algorithm Experiments

We followed the trial and error approach while developing the dynamic program algorithm to solve the Covering Problem. At the initial phase of the program (where at each node the program was generating \( n^{k+1} \) cost functions) even for a 200 node graph, the program took more than a week to solve the problem. This is mainly because of the generation of huge amount of cost functions that filled up the memory very quickly. So we started developing techniques that reduced the number of cost functions. These includes the pruning heuristic, bounding the assignment functions, branch and bound technique and several implementation tricks as discussed in Chapter 3. We employed all of them into a single program and then began experiment with larger graphs. This time it solved the problem much quicker than the initial program. We used CPLEX to solve the problem on the same graphs to ensure the correctness of our program. But we found out that CPLEX solves this problem on the same graphs very quickly. At that point we decided to incorporate the power of CPLEX in our program and developed a Hybrid algorithm which employs dynamic program as well as CPLEX. With CPLEX we solved subproblems defined by the bottom part of the Tree Decomposition. We defined the size of the subproblems that will be solved by CPLEX, the rest of the Tree nodes are solved by the Dynamic program. We developed this Hybrid algorithm with the intuition that, once
CPLEX solves the subproblems defined by the bottom parts (larger parts) of the Tree Decomposition, then for the rest of the tree nodes the dynamic Program will outweigh CPLEX which will result a total runtime that will beat CPLEX.

For our experiments, we ran the following three algorithms on the random graphs—

- CPLEX
- Hybrid Program (Hybrid)
- Hybrid Program with Branch and Bound (Hybrid_with_bb)

The Hybrid Program (Hybrid) and Hybrid Program with Branch and Bound (Hybrid_with_bb) are two versions of the same program. The first one (Hybrid) contains all the techniques and the implementation tricks discussed in Chapter 3 except the Branch and Bound technique. The later one (Hybrid_with_bb) contains all the techniques including the Branch and Bound. The reason we developed these two versions of the program is to show the performance difference between the program without branch and bound and the program that employed the branch and bound. We used CPLEX to solve the same graphs to ensure correctness and also for the runtime comparison among these three algorithms.

### 4.2.1 Data Sets

In our experiments, we decided to use the randomly generated series-parallel graphs (as described in section 4.1.1). It is expensive in terms of running time to run our algorithm on a Tree Decomposition with high Treewidth. As a series-parallel graph can have treewidth at most two, it ensures a quick solution time by our algorithms. Also it loosely mimics (by sparsity and connectivity) a wireless network environment.

Though our initial target was to evaluate the performance of our algorithms on sparse graphs (so that they can be employed later on wireless networks to solve the covering problem), we tried to run
our algorithms on dense graphs as well. But our algorithms are not feasible for Tree Decomposition with high treewidth. So for this thesis we kept our focus on running algorithms on series-parallel graphs.

4.2.2 Experiment Results

In the following, we present the empirical data for four Covering Problem instance on different series-parallel graphs. We designed four Covering problem instances for our experiment. The parameters for the instances of the Covering Problem is given below-

- **1st Instance:** client radius = 10, facility opening cost = 15 and penalty = 20.
- **2nd Instance:** client radius = 20, facility opening cost = 25 and penalty = 20.
- **3rd Instance:** client radius = 30, facility opening cost = 13 and penalty = 22.
- **4th Instance:** client radius = 40, facility opening cost = 10 and penalty = 17.

In these instances the client radius sets up a covering range for the clients. This covering range determines an average size of the neighborhood set of a vertex. We started with a small radius in the first instance then gradually increased it in the other instances to check the performance of our algorithms with a large neighborhood set. As for the different facility opening cost and penalty, we at first ran experiments with several random opening and penalty cost on smaller graphs. From those experiments we chose four sets of opening and penalty costs to be included in our instances.

Table 4.3 contains runtime data for solving the 1st instance described above by CPLEX, Hybrid and Hybrid_with_bb on six series-parallel graphs of treewidth 2. The first four columns contains the graph data (number of vertices |V| and number of edges |E|) and Tree Decomposition data (number of Tree Decomposition Nodes |N| and number of Tree Decomposition Edges |E|). The fifth column Runtime for CPLEX(s) includes the average runtime (in seconds) data for CPLEX on each graphs. This runtime data is computed after subtracting the runtime to compute the shortest path.
Table 4.3: Runtime Comparison among CPLEX, Hybrid Program and Hybrid with BB (Branch and Bound) for a Covering problem Instance where radius = 10, facility opening cost = 15 and penalty = 20

<table>
<thead>
<tr>
<th>Graph</th>
<th>Tree D</th>
<th>Runtime for CPLEX(s)</th>
<th>Subprob Size</th>
<th>Num of Dyn Node</th>
<th>Runtime for Hybrid(s)</th>
<th>Runtime for Hybrid with BB(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>516</td>
<td>582</td>
<td>514</td>
<td>513</td>
<td>0.421</td>
<td>9.986</td>
<td>8.983</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>150</td>
<td>4.90</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>200</td>
<td>2.512</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>250</td>
<td>1.74</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>300</td>
<td>1.006</td>
</tr>
<tr>
<td>1149</td>
<td>1323</td>
<td>1146</td>
<td>1145</td>
<td>0.864</td>
<td>200</td>
<td>36.794</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>400</td>
<td>19.68</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>600</td>
<td>3.042</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>700</td>
<td>2.777</td>
</tr>
<tr>
<td>2004</td>
<td>2283</td>
<td>2002</td>
<td>2001</td>
<td>1.901</td>
<td>500</td>
<td>46.18</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>800</td>
<td>26.53</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1000</td>
<td>14.451</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1200</td>
<td>3.357</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1500</td>
<td>2.655</td>
</tr>
<tr>
<td>3330</td>
<td>3810</td>
<td>3328</td>
<td>3327</td>
<td>4.657</td>
<td>1000</td>
<td>129.44</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1200</td>
<td>92.741</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1500</td>
<td>40.605</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2000</td>
<td>4.796</td>
</tr>
<tr>
<td>5254</td>
<td>6008</td>
<td>5252</td>
<td>5251</td>
<td>14.138</td>
<td>2000</td>
<td>241.095</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2500</td>
<td>65.44</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2800</td>
<td>14.16</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3000</td>
<td>14.38</td>
</tr>
<tr>
<td>8752</td>
<td>10004</td>
<td>8749</td>
<td>8748</td>
<td>38.657</td>
<td>2500</td>
<td>387.061</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3500</td>
<td>320.251</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5000</td>
<td>322.156</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6500</td>
<td>35.192</td>
</tr>
</tbody>
</table>

matrix from the total running time. The sixth column Subproblem Size contains the sizes of various subproblems. Each subproblem size indicates a number which defines the size of the subproblems that are to be solved by CPLEX in the Hybrid and Hybrid_with_bb algorithm. For each graphs we experimented with different sizes. The seventh column Num of Dyn Node contains the data of number of Tree Decomposition nodes processed by the dynamic program while running the Hybrid and Hybrid_with_bb program. The eighth and ninth column Runtime for Hybrid(s) and Runtime for Hybrid_with_bb(s) contains the average runtime(in seconds) for the algorithms Hybrid and Hybrid_with_bb. These runtimes are computed after subtracting the runtime to compute the shortest path matrix from the total running time. Table 4.5, 4.7, 4.9 contains similar data for 2nd, 3rd and 4th instances.
Table 4.4 contains the permutation data for solving the 1st instance of the Covering Problem for the algorithm Hybrid and Hybrid_with_bb. The first four columns contain the graph data (number of vertices $|V|$ and number of edges $|E|$) and Tree Decomposition data (number of Tree Decomposition Nodes $|N|$ and number of Tree Decomposition Edges $|E|$). The fifth column Subproblem Size contains the same data as the column Subproblem Size in Table 4.3. The sixth, seventh and eighth column are the subcolumns of Perm data for Hybrid which gathers the permutation data for the Hybrid algorithm. The sixth column Total Perm indicates the total number of permutations (cost functions) generated for that specific subproblem size. The seventh column After Pruning denotes the number of permutations or cost functions processed by the algorithm that is the number of permutations that are not pruned. The eighth column Total pruned perm is the number of permutations pruned by the pruning Heuristic in the Hybrid algorithm. Columns nine to thirteen are the subcolumns of Perm data for Hybrid_bb which describes the permutation data for the algorithm Hybrid_with_bb. The ninth and tenth column Total Perm and After pruning contains similar data as in for the Hybrid program. The eleventh column Pruned by Pruning Heuristic contains the number of permutations pruned by the pruning heuristic alone. The next column Pruned by BB contains the number of permutations pruned by the Branch and Bound technique. The Total pruned perm column contains the sum of the earlier two columns. Table 4.6, 4.8 and 4.10 contains similar data for 2nd, 3rd and 4th instances of the Covering Problem.

Figure 4.1 shows the runtime comparison among CPLEX, Hybrid and Hybrid_with_bb. In case of Hybrid and Hybrid_with_bb, for each graph we took the best case running time among all the experiments for different subproblem sizes and plotted them against the number of vertices. The best case for Hybrid and Hybrid_with_bb algorithm always occurs when the dynamic program part needs to solve only one Tree Decomposition node (the root). Figure 4.3, 4.5 and 4.7 shows similar comparisons for different instances (2nd, 3rd and 4th) of the Covering problem. Each curves in these graphs are polynomial in nature as for series parallel graph the runtime $O(n^{k+2})$ becomes polynomial as $k$ (treewidth) is no more than 2.
Figure 4.2 shows the number of permutations comparison between Hybrid and Hybrid_with_bb. For each graph we took the sum of the Total pruned perm for that graph and plotted these number against the number of vertices. Figure 4.4, 4.6, 4.8 shows similar permutations comparison between Hybrid and Hybrid_with_bb for other instances (2nd, 3rd and 4th) of the Covering Problem.

In the following, the rest of the data tables along with their comparison graphs are shown sequentially. We will discuss the evaluation of these data in the analysis section.

Table 4.4: Permutations comparison between Hybrid Program and Hybrid with BB (Branch and Bound) for a Covering problem Instance where radius = 10, facility opening cost = 15 and penalty = 20

<table>
<thead>
<tr>
<th>Graph</th>
<th>Tree D</th>
<th>Subprob Size</th>
<th>Total Perm</th>
<th>Perm data for Hybrid</th>
<th>Total Perm</th>
<th>Perm data for Hybrid_BB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>After pruned</td>
<td>Pruned perm</td>
<td>After pruned</td>
<td>Pruned perm</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>516</td>
<td>582</td>
<td>514 513</td>
<td>3774</td>
<td>3705</td>
<td>69</td>
<td>1688</td>
</tr>
<tr>
<td>1149</td>
<td>1323</td>
<td>1146 1145</td>
<td>5263</td>
<td>3974</td>
<td>3705</td>
<td>1688</td>
</tr>
<tr>
<td>2004</td>
<td>2283</td>
<td>2002 2001</td>
<td>3624</td>
<td>3597</td>
<td>120</td>
<td>2064</td>
</tr>
<tr>
<td>3330</td>
<td>3810</td>
<td>3328 3327</td>
<td>7668</td>
<td>7631</td>
<td>398</td>
<td>17346</td>
</tr>
<tr>
<td>5254</td>
<td>6008</td>
<td>5252 5251</td>
<td>3531</td>
<td>3484</td>
<td>97</td>
<td>1509</td>
</tr>
<tr>
<td>8752</td>
<td>10004</td>
<td>8749 8748</td>
<td>1516</td>
<td>1469</td>
<td>234</td>
<td>5196</td>
</tr>
</tbody>
</table>
Figure 4.1: Runtime Comparison among CPLEX, Hybrid and Hybrid_with_bb for client radius = 10, facility opening cost =15 penalty =20

Figure 4.2: Permutation Comparison between Hybrid and Hybrid_with_bb for client radius = 10, facility opening cost =15 penalty =20
Table 4.5: Runtime Comparison among CPLEX, Hybrid Program and Hybrid with BB (Branch and Bound) for a Covering problem Instance where radius = 20, facility opening cost = 25 and penalty = 20

<table>
<thead>
<tr>
<th>Graph</th>
<th>Tree D</th>
<th>Runtime for CPLEX(s)</th>
<th>Subprob Size</th>
<th>Num of Dyn node</th>
<th>Runtime for Hybrid(s)</th>
<th>Runtime for Hybrid with BB(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>75</td>
<td>70</td>
<td>15.337</td>
<td>14.247</td>
</tr>
<tr>
<td>516</td>
<td>582</td>
<td>514</td>
<td>513</td>
<td>0.427</td>
<td>100</td>
<td>93.648</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>150</td>
<td>33</td>
<td>7.061</td>
<td>6.737</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>200</td>
<td>13</td>
<td>3.873</td>
<td>3.815</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>250</td>
<td>4</td>
<td>2.85</td>
<td>2.689</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>300</td>
<td>1</td>
<td>1.405</td>
<td>1.465</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>200</td>
<td>104</td>
<td>53.265</td>
<td>48.393</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>600</td>
<td>3</td>
<td>6.138</td>
<td>4.595</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>700</td>
<td>1</td>
<td>3.854</td>
<td>3.102</td>
</tr>
<tr>
<td>1149</td>
<td>1323</td>
<td>1146</td>
<td>1145</td>
<td>0.862</td>
<td>200</td>
<td>55.95</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>500</td>
<td>206</td>
<td>57.785</td>
<td>50.731</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>800</td>
<td>108</td>
<td>35.211</td>
<td>31.027</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1000</td>
<td>52</td>
<td>19.35</td>
<td>17.306</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1200</td>
<td>4</td>
<td>4.087</td>
<td>3.891</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1500</td>
<td>1</td>
<td>3.25</td>
<td>3.01</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1000</td>
<td>397</td>
<td>183.601</td>
<td>155.026</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1200</td>
<td>281</td>
<td>131.72</td>
<td>110.917</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1500</td>
<td>112</td>
<td>54.611</td>
<td>45.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2000</td>
<td>1</td>
<td>5.109</td>
<td>5.526</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2000</td>
<td>376</td>
<td>127.307</td>
<td>221.443</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2500</td>
<td>76</td>
<td>72.2</td>
<td>58.791</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2800</td>
<td>1</td>
<td>12.4</td>
<td>11.533</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3000</td>
<td>1247</td>
<td>12.4</td>
<td>11.631</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2500</td>
<td>264</td>
<td>492.921</td>
<td>416.445</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3000</td>
<td>199</td>
<td>382.823</td>
<td>331.911</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3500</td>
<td>199</td>
<td>380.48</td>
<td>343.832</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>5000</td>
<td>199</td>
<td>380.48</td>
<td>343.832</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>6500</td>
<td>1</td>
<td>35.92</td>
<td>38.6</td>
</tr>
<tr>
<td>Graph</td>
<td>Tree D</td>
<td>Subprob Size</td>
<td>Perm data for Hybrid(s)</td>
<td>Perm data for Hybrid_BB(s)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------</td>
<td>--------</td>
<td>--------------</td>
<td>-------------------------</td>
<td>---------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Total perm</td>
<td>After pruning</td>
<td>Total pruned perm</td>
<td>Total perm</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>516</td>
<td>582</td>
<td>514 513</td>
<td>75</td>
<td>6914</td>
<td>6529</td>
<td>385</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>150</td>
<td>2843</td>
<td>2742</td>
<td>101</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>200</td>
<td>1287</td>
<td>1254</td>
<td>33</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>250</td>
<td>650</td>
<td>635</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>300</td>
<td>145</td>
<td>135</td>
<td>10</td>
</tr>
<tr>
<td>1149</td>
<td>1323</td>
<td>1146 1145</td>
<td>200</td>
<td>19508</td>
<td>18386</td>
<td>1122</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>400</td>
<td>11377</td>
<td>10574</td>
<td>803</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>600</td>
<td>1146</td>
<td>1060</td>
<td>86</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>700</td>
<td>469</td>
<td>415</td>
<td>54</td>
</tr>
<tr>
<td>2004</td>
<td>2283</td>
<td>2002 2001</td>
<td>500</td>
<td>14196</td>
<td>13468</td>
<td>728</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>800</td>
<td>8781</td>
<td>8246</td>
<td>535</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1000</td>
<td>4914</td>
<td>4576</td>
<td>338</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1200</td>
<td>471</td>
<td>437</td>
<td>34</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1500</td>
<td>151</td>
<td>133</td>
<td>18</td>
</tr>
<tr>
<td>3330</td>
<td>3810</td>
<td>3328 3327</td>
<td>1000</td>
<td>31885</td>
<td>29971</td>
<td>1914</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1200</td>
<td>22241</td>
<td>20913</td>
<td>1328</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1500</td>
<td>8285</td>
<td>7752</td>
<td>533</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2000</td>
<td>48</td>
<td>48</td>
<td>0</td>
</tr>
<tr>
<td>5254</td>
<td>6008</td>
<td>5252 5251</td>
<td>2000</td>
<td>25133</td>
<td>23934</td>
<td>1199</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2500</td>
<td>5748</td>
<td>5371</td>
<td>377</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2800</td>
<td>105</td>
<td>105</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3000</td>
<td>105</td>
<td>105</td>
<td>0</td>
</tr>
<tr>
<td>8752</td>
<td>10004</td>
<td>8749 8748</td>
<td>2500</td>
<td>20108</td>
<td>19255</td>
<td>853</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3500</td>
<td>14575</td>
<td>13995</td>
<td>580</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>5000</td>
<td>14553</td>
<td>13973</td>
<td>580</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>6500</td>
<td>107</td>
<td>107</td>
<td>0</td>
</tr>
</tbody>
</table>
Figure 4.3: Runtime Comparison among CPLEX, Hybrid and Hybrid_with_bb for client radius = 20, facility opening cost = 25 penalty = 20

Figure 4.4: Permutation Comparison between Hybrid and Hybrid_with_bb for client radius = 20, facility opening cost = 25 penalty = 20
Table 4.7: Runtime Comparison among CPLEX, Hybrid Program and Hybrid with BB (Branch and Bound) for a Covering problem Instance where radius = 30, facility opening cost = 13 and penalty = 22

<table>
<thead>
<tr>
<th>Graph</th>
<th>Tree D</th>
<th>Runtime for CPLEX(s)</th>
<th>Subprob Size</th>
<th>Num of Dyn node</th>
<th>Runtime for Hybrid(s)</th>
<th>Runtime for Hybrid with BB(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>516</td>
<td>582</td>
<td>514</td>
<td>513</td>
<td>0.437</td>
<td>22.715</td>
<td>21.337</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>11.14</td>
<td>10.624</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6.51</td>
<td>6.28</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4.03</td>
<td>4.356</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.98</td>
<td>2.212</td>
</tr>
<tr>
<td>1149</td>
<td>1323</td>
<td>1146</td>
<td>1145</td>
<td>0.879</td>
<td>200</td>
<td>81.361</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>104</td>
<td>45.356</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>9.828</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>9.203</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7.18</td>
<td>6.958</td>
</tr>
<tr>
<td>2004</td>
<td>2283</td>
<td>2002</td>
<td>2001</td>
<td>1.967</td>
<td>500</td>
<td>92.521</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>108</td>
<td>52.585</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>52</td>
<td>33.286</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>5.74</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5.318</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>150</td>
<td>3.822</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3.645</td>
</tr>
<tr>
<td>3330</td>
<td>3810</td>
<td>3328</td>
<td>3327</td>
<td>4.704</td>
<td>1000</td>
<td>397</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>281</td>
<td>304.47</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>112</td>
<td>214.729</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>200</td>
<td>90.18</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>71.13</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>7.925</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8.164</td>
</tr>
<tr>
<td>5254</td>
<td>6008</td>
<td>5252</td>
<td>5251</td>
<td>13.109</td>
<td>2000</td>
<td>376</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>76</td>
<td>410.311</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>104.99</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>83.929</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>150</td>
<td>16.497</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>15.239</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3000</td>
<td>16.177</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>15.24</td>
</tr>
<tr>
<td>8752</td>
<td>10004</td>
<td>8749</td>
<td>8748</td>
<td>39.761</td>
<td>2500</td>
<td>264</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>264</td>
<td>684.553</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3500</td>
<td>511.696</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5000</td>
<td>512.005</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6500</td>
<td>36.431</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>42.556</td>
</tr>
</tbody>
</table>
Table 4.8: Number of Permutations comparison between Hybrid Program and Hybrid with BB (Branch and Bound) for a Covering problem Instance where radius = 30, facility opening cost = 13 and penalty = 22

<table>
<thead>
<tr>
<th>Graph</th>
<th>Tree D</th>
<th>Subprob Size</th>
<th>Perm data for Hybrid(s)</th>
<th>Perm data for Hybrid_BB(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Total perm</td>
<td>After pruning</td>
</tr>
<tr>
<td>VI</td>
<td>IE</td>
<td>IN</td>
<td>IE</td>
<td></td>
</tr>
<tr>
<td>516</td>
<td>582</td>
<td>514</td>
<td>513</td>
<td>75</td>
</tr>
<tr>
<td>200</td>
<td>2271</td>
<td>2075</td>
<td>197</td>
<td>150</td>
</tr>
<tr>
<td>250</td>
<td>888</td>
<td>830</td>
<td>58</td>
<td>200</td>
</tr>
<tr>
<td>300</td>
<td>225</td>
<td>207</td>
<td>18</td>
<td>230</td>
</tr>
<tr>
<td>1149</td>
<td>1323</td>
<td>1146</td>
<td>1145</td>
<td>200</td>
</tr>
<tr>
<td>2004</td>
<td>2283</td>
<td>2002</td>
<td>2001</td>
<td>400</td>
</tr>
<tr>
<td>3330</td>
<td>3810</td>
<td>3282</td>
<td>3327</td>
<td>600</td>
</tr>
<tr>
<td>5254</td>
<td>6008</td>
<td>5252</td>
<td>5251</td>
<td>800</td>
</tr>
<tr>
<td>8752</td>
<td>10004</td>
<td>8749</td>
<td>8748</td>
<td>1000</td>
</tr>
<tr>
<td>2000</td>
<td>40025</td>
<td>37175</td>
<td>2850</td>
<td>1200</td>
</tr>
<tr>
<td>5000</td>
<td>3330</td>
<td>3282</td>
<td>3327</td>
<td>1500</td>
</tr>
<tr>
<td>7652</td>
<td>8749</td>
<td>8748</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Figure 4.5: Runtime Comparison among CPLEX, Hybrid and Hybrid_with_bb for client radius = 30, facility opening cost = 13 penalty = 22

Figure 4.6: Permutation Comparison between Hybrid and Hybrid_with_bb for client radius = 30, facility opening cost = 13 penalty = 22
Table 4.9: Runtime Comparison among CPLEX, Hybrid Program and Hybrid with BB (Branch and Bound) for a Covering problem Instance where radius = 40, facility opening cost = 10 and penalty = 17

<table>
<thead>
<tr>
<th>Graph</th>
<th>Tree D</th>
<th>Runtime for CPLEX(s)</th>
<th>Subprob Size</th>
<th>Num of Dyn node</th>
<th>Runtime for Hybrid(s)</th>
<th>Runtime for Hybrid with BB(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>516 582 514 513</td>
<td>0.396</td>
<td>75</td>
<td>70</td>
<td>30.931</td>
<td>29.052</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1149 1323 1146 1145</td>
<td>1.188</td>
<td>200</td>
<td>226</td>
<td>108.034</td>
<td>94.75</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2004 2283 2002 2001</td>
<td>2.277</td>
<td>500</td>
<td>206</td>
<td>125.629</td>
<td>127.295</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3330 3810 3328 3327</td>
<td>5.074</td>
<td>1000</td>
<td>397</td>
<td>1571.985</td>
<td>1309.398</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5254 6008 5252 5251</td>
<td>13.551</td>
<td>2000</td>
<td>376</td>
<td>612.082</td>
<td>653.461</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8752 10004 8749 8748</td>
<td>38.332</td>
<td>2500</td>
<td>264</td>
<td>968.473</td>
<td>758.696</td>
<td></td>
</tr>
</tbody>
</table>
Table 4.10: Number of Permutations comparison between Hybrid Program and Hybrid with BB(Branch and Bound) for a Covering problem Instance where radius = 40, facility opening cost = 10 and penalty = 17

<table>
<thead>
<tr>
<th>Graph</th>
<th>Tree D</th>
<th>Subprob Size</th>
<th>Perm data for Hybrid(s)</th>
<th>Perm data for Hybrid_BB(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Total perm</td>
<td>After pruning</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Total</td>
<td>After pruning</td>
</tr>
<tr>
<td>516</td>
<td>582</td>
<td>514</td>
<td>513</td>
<td>75</td>
</tr>
<tr>
<td>516</td>
<td>582</td>
<td>514</td>
<td>513</td>
<td>150</td>
</tr>
<tr>
<td>516</td>
<td>582</td>
<td>514</td>
<td>513</td>
<td>200</td>
</tr>
<tr>
<td>516</td>
<td>582</td>
<td>514</td>
<td>513</td>
<td>250</td>
</tr>
<tr>
<td>516</td>
<td>582</td>
<td>514</td>
<td>513</td>
<td>300</td>
</tr>
<tr>
<td>1149</td>
<td>1323</td>
<td>1146</td>
<td>1145</td>
<td>200</td>
</tr>
<tr>
<td>1149</td>
<td>1323</td>
<td>1146</td>
<td>1145</td>
<td>400</td>
</tr>
<tr>
<td>1149</td>
<td>1323</td>
<td>1146</td>
<td>1145</td>
<td>600</td>
</tr>
<tr>
<td>1149</td>
<td>1323</td>
<td>1146</td>
<td>1145</td>
<td>700</td>
</tr>
<tr>
<td>2004</td>
<td>2283</td>
<td>2002</td>
<td>2001</td>
<td>500</td>
</tr>
<tr>
<td>2004</td>
<td>2283</td>
<td>2002</td>
<td>2001</td>
<td>800</td>
</tr>
<tr>
<td>2004</td>
<td>2283</td>
<td>2002</td>
<td>2001</td>
<td>1000</td>
</tr>
<tr>
<td>2004</td>
<td>2283</td>
<td>2002</td>
<td>2001</td>
<td>1200</td>
</tr>
<tr>
<td>2004</td>
<td>2283</td>
<td>2002</td>
<td>2001</td>
<td>1500</td>
</tr>
<tr>
<td>3330</td>
<td>3810</td>
<td>3328</td>
<td>3327</td>
<td>1000</td>
</tr>
<tr>
<td>3330</td>
<td>3810</td>
<td>3328</td>
<td>3327</td>
<td>1200</td>
</tr>
<tr>
<td>3330</td>
<td>3810</td>
<td>3328</td>
<td>3327</td>
<td>1500</td>
</tr>
<tr>
<td>3330</td>
<td>3810</td>
<td>3328</td>
<td>3327</td>
<td>2000</td>
</tr>
<tr>
<td>5254</td>
<td>6008</td>
<td>5252</td>
<td>5251</td>
<td>2000</td>
</tr>
<tr>
<td>5254</td>
<td>6008</td>
<td>5252</td>
<td>5251</td>
<td>2500</td>
</tr>
<tr>
<td>5254</td>
<td>6008</td>
<td>5252</td>
<td>5251</td>
<td>2800</td>
</tr>
<tr>
<td>5254</td>
<td>6008</td>
<td>5252</td>
<td>5251</td>
<td>3000</td>
</tr>
<tr>
<td>8752</td>
<td>10004</td>
<td>8749</td>
<td>8748</td>
<td>2500</td>
</tr>
<tr>
<td>8752</td>
<td>10004</td>
<td>8749</td>
<td>8748</td>
<td>3500</td>
</tr>
<tr>
<td>8752</td>
<td>10004</td>
<td>8749</td>
<td>8748</td>
<td>5000</td>
</tr>
<tr>
<td>8752</td>
<td>10004</td>
<td>8749</td>
<td>8748</td>
<td>6500</td>
</tr>
</tbody>
</table>
Figure 4.7: Runtime Comparison among CPLEX, Hybrid and Hybrid_with_bb for client radius = 40, facility opening cost = 10 penalty = 17

Figure 4.8: Permutation Comparison between Hybrid and Hybrid_with_bb for client radius = 40, facility opening cost = 10 penalty = 17
4.3 Analysis

In this section we will analyze the results described in the earlier data tables. In the following, we will discuss our findings after observing the data.

- As described in Chapter 3, if the treewidth of the Tree Decomposition is $k$, then for each Tree Decomposition node, there will be $n^{k+1}$ number of cost functions (permutations). During the initial phase of the development, our algorithm used to generate and store a huge number of cost functions (before pruning) in accordance with the theoretical estimation $n^{k+1}$ to solve a Covering Problem instance. After employing the techniques discussed in section 3.2.2 (Reduction of cost functions based on Covering Neighborhood) and in section 3.4.7 (Bounding the Assignment Function), we were able to decrease greatly the number of generated cost functions (permutations) even before pruning. For example, for a series-parallel graph (treewidth $k = 2$) of 516 nodes and 582 edges, according to our theoretical estimation, in the worst case, for a single node, the number of cost functions generated can be $516^3 = 137388096$. In table 4.3 and 4.4, for the same graph of 516 nodes and 582 edges, for a subproblem size of 75 (where the number of nodes processed by the dynamic program is 70) the total number of permutations generated by the Hybrid program is 3774 (for a Covering Problem instance where client radius = 10, facility opening cost = 15 and penalty = 20) which indicates a considerable amount of decrease in the generation of permutations (before pruning). For all the other experiments, the techniques discussed in section 3.2.2 and 3.4.7 greatly reduces the total number of cost functions or permutations (before pruning).

- After scrutinizing the table, we found a relation between the runtime of the programs (both Hybrid and Hybrid_with_bb) and the total number of generated permutations (before pruning) to the client radius. As the client radius increases, so as the total number of generated permutations (as well as the total number of processed permutations) which leads to a longer runtime. This is because as the client radius increases, so as the Covering Neighborhood of a Tree Decomposition node (described in section 3.2.2). As the cost function generator for
each node depends on the size of the Covering Neighborhood of that node, the larger the set of the Covering Neighborhood, the larger is the number of total permutations. As an example, In Figure 4.9, we plotted the data for the series-parallel graph with 3330 nodes and 3810 edges with different client radius (10, 20, 30, 40) against the Hybrid runtime for subproblem size 2000 (from Table 4.1, 4.3, 4.5, 4.7). In Figure 4.10, we plotted the permutation data for subproblem size 1000 of the graph with nodes 3330 and edges 3810 (from Table 4.2, 4.4, 4.6, 4.8). In both the Figures, it is clear that the runtime and the number of permutations for this graph (with their respective size) increases with the client radius. This trend follow for both Hybrid and Hybrid_with_bb program with different graphs with different subproblem sizes.

Figure 4.9: Client Radius Vs Runtime data of the Hybrid program for the graph with 3330 nodes with subproblem size 2000
In case of smaller graphs (graphs with nodes 516, 1149, 2004, 3330) if we compare the running time between CPLEX, Hybrid and Hybrid_with_bb, then CPLEX beats both the program comfortably for all the Covering Problem Instances. But on larger graphs (graphs with nodes 5254, 8752) with smaller client radius (10,20) either Hybrid or Hybrid_with_bb beats CPLEX in terms of running time. This occurs whenever the subproblem size reaches the amount for which the dynamic program part of Hybrid and Hybrid_with_bb needs to process only one Tree Decomposition node (the root of the Tree Decomposition). This confirms that dynamic programming is very expensive when applied to an entire problem space (with respect to Covering Problem), but can be effective if it can be restricted to a subset of the problem space given that the rest was solved by a quick heuristic algorithm. This also leads us to the notion of partial Tree Decomposition and other future research interests which will be discussed in the conclusion chapter.
After studying the number of total pruned permutation for the Hybrid and Hybrid_with_bb program, it is apparent that the program Hybrid_with_bb (equipped with the branch and bound technique) is pruning the generated permutations far more than the Hybrid program with branch and bound technique being the main contributor. The reason the branch and bound technique is so successful than the pruning heuristic is for series-parallel graphs the integrality gap between the integer and relaxed version of the Covering Problem is very small ($\approx 0$). So, the bounds generated by solving the relaxed subproblems in Hybrid_with_bb is very tight. As a result more number of cost functions are pruned by the branch and bound technique. Also, we noticed that the total number of generated permutations differs between the programs Hybrid and Hybrid_with_bb. This is because of the process that we used to mimic Nice Tree Decomposition. For example, given a original Tree Decomposition leaf node $l$ with the size of the vertex set more than one, we pick a random vertex and make it the vertex set for the newly constructed Nice Tree Decomposition leaf node $l'$. Then we compute the set minus between the vertex set of $l$ and $l'$ and add them to the vertex set of $l'$ to generate the vertex set of a newly constructed Introduce Node. This means while executing Hybrid and Hybrid_with_bb, the vertex ordering of the Nice Tree Decomposition nodes does not match. This leads to compute different bounds (as described in 3.4.7) for Hybrid and Hybrid_with_bb for the same node. As a result of these different bounds, the total number of permutations generated by Hybrid and Hybrid_with_bb for the same node will differ.

In this Chapter, we have presented the empirical data of the Tree Decomposition experiments and the Covering Problem experiments. We have compared the performance of different algorithms and mentioned our findings after evaluating the data. In the Conclusion chapter, we will talk about the implication of our research with future directions.
Chapter 5

Conclusion

In this thesis, we developed a dynamic programming algorithm to solve standard Covering Problem on a Tree Decomposition of a graph with the intuition that the unique properties of a Tree Decomposition would facilitate the design of the dynamic program. We developed a bottom up dynamic programming framework which utilizes the Tree Decomposition to build the solution. But for this approach to work, the program needs to generate $n^{k+1}$ ($k =$ Treewidth) number of cost functions for each Tree Decomposition node. This leads to a shortage of memory to solve the Covering problem even on a small graph below 100 nodes.

To counter this problem, we developed few techniques that are added on top of the framework. These techniques considerably reduces the number of cost function per node. Though after employing this technique we solved the memory problem to an extent (the program was able to solve series-parallel graphs with thousands of nodes and edges considerably quickly), but this dynamic programming algorithm proved to be expensive when compared to other tools like CPLEX for solving the Covering Problem. CPLEX takes far less time than our dynamic program to solve Covering Problem on series-parallel graphs.

We then worked on several ideas to even out this difference of running time. One idea we implemented was to use the CONDOR grid (a high throughput system) to develop a parallel algorithm to solve the Covering Problem where each computer will solve a different subproblem and report back the solution back to the parent. But for Covering Problem, this technique was proved to be infeasible because CPLEX was already solving the Covering Problem very fast on series-parallel graphs. The experiment results from the parallel program didn’t show any significant improvement. But we believe that this parallel algorithm will be useful for other more difficult facility location problems.
In our next idea, we decided to incorporate the power of CPLEX in our dynamic programming framework. We call this algorithm the Hybrid algorithm. The CPLEX module was used to solve all the subproblems (of a certain size) at the bottom part of the Tree Decomposition. After experimenting we got some success in this approach. When the client radius is small and the graph instance is quite large, then the Hybrid algorithm beats the CPLEX but not by a large margin. We experimented with two different versions of the Hybrid algorithm, one equipped with the branch and bound technique (Hybrid_with_bb) and the other version (Hybrid) without the branch and bound technique. Hybrid_with_bb fares better than Hybrid in smaller graphs but performs similarly in cases of large graphs. But almost in every case Hybrid_with_bb version prunes more cost functions than the Hybrid version. as the branch and bound technique is very useful on series-parallel graphs (integrality gap $\approx 0$). But the preprocessing step for the Hybrid_with_bb is expensive in terms of running time as the bounds for each Tree Decomposition node are pre-computed and saved in a file.

In cases where the Hybrid version beats CPLEX, the number of node processed by the dynamic programming technique is always one (the root of the Tree Decomposition). This shows that the dynamic program is indeed expensive if it solves the greater part of the Tree Decomposition but can be effective if it can be restricted to solve a few number of nodes (in our case the root). Though it beats CPLEX marginally but given the fact that on series-parallel graphs CPLEX is very effective, this is a success none the less. Moreover this finding will also guide us to our future endeavors.

As solving only the root of Tree Decomposition using Dynamic programming gives the Hybrid program the edge over CPLEX, it is evident that we don’t require a full Tree Decomposition of a graph. Instead a partial Tree Decomposition which will contain a few nodes with equally balanced subproblem size would do the trick. This partial Tree Decomposition will involve finding a small separator for the input graph where the divided components will be balanced. In this respect, we can try to find a centroid or a geometric center of a tree decomposition and make it a separator. Then the divided components will be balanced in size because they are equally distanced from the center. This separator will work as a root of this construct. We can involve CPLEX or other methods
to build solution of the subproblems defined by the components. In this way, after finding a small separator, we can even handle graphs with larger Treewidth.

Our future plan will include finding and developing efficient methods for finding small size separator of a graph. We plan to use our current algorithm to solve covering problem on real world wireless network data to check it’s effectiveness. We can also extend our algorithm for P-median problem (a facility location problem similar to Covering problem) if the experiments are successful. Also we would like to apply the parallel programming algorithm on much harder facility location problems for which we believe the technique will be quite effective.
Bibliography


