In compliance with the
Canadian Privacy Legislation
some supporting forms
may have been removed from
this dissertation.

While these forms may be included
in the document page count,
their removal does not represent
any loss of content from the dissertation.

Parallel Fixed-Parameter Tractability of Max-Leaf Spanning Tree Calculation

By

Xinhong Zhang

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of
Master of Science
Information and System Science

Ottawa-Carleton Institute for Computer Science
School of Computer Science
Carleton University
Ottawa, Ontario

September 24, 2003

© Copyright
2003, Xinhong Zhang

National Library
of Canada

Bibliothèque nationale
du Canada

Acquisitions and
Bibliographic Services

Acquisitions et
services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

ISBN: 0-612-88791-X

Our file Notre référence

ISBN: 0-612-88791-X

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

Canada

The undersigned hereby recommend to
the Faculty of Graduate Studies and Research
acceptance of the thesis,

**Parallel Fixed-Parameter Tractability of Max-Leaf Spanning Tree
Calculation**

submitted by

Xinhong Zhang

Dr. Douglas Howe
(Director, School of Computer Science)

Dr. Frank Dehne
(Thesis Supervisor)

Carleton University
September 24, 2003

Abstract

This thesis considers the sequential coordinatized kernels and catalytic reductions fixed-parameter tractability (FPT) algorithm introduced by Fellows *et al* for calculating the max-leaf spanning tree of a connected graph [7]. The parallelizations of Fellows *et al*'s sequential algorithm are studied. Both the sequential algorithm and one of the parallel algorithms are implemented. This thesis presents the findings in the experiments, which are asymptotic linear speedup or superlinear speedup of the parallel algorithm over the sequential algorithm. Superlinear speedup indicates the possible improvement of Fellows *et al*'s approach by combining it with the sequential simulation of the parallel algorithm.

Acknowledgements

Firstly, I wish to express my sincere gratitude to my supervisor, Dr. Frank Dehne, for his trust, patient guidance and inspiration, which was always there when I needed it.

I would like to thank Peter Taillon for his practical advice and thoughtful comments.

I also want to thank Dr. Andrew Rau-Chaplin for providing me their wonderful lab equipment.

Thanks to Guo Hua for offering me the nice test instances. Thanks to Kenji Imasaki for being my labmate and giving me lots help. Thanks to the many other friends who have helped me, one way or another.

Also thanks to my defense committee: Dr. Doron Nussbaum from School of Computer Science, Dr. Daniel Panario from School of Mathematics and Statistics, and Dr. Sivarama Dandamudi from School of Computer Science for spending their precious time reading my thesis, for their corrections and comments.

Lastly, thanks to my parents and my husband who have constantly given me encouragement, support and love. To them I dedicate this thesis.

Contents

Abstract	iii
Acknowledgements	iv
List of Tables	viii
List of Figures	ix
1 Introduction	1
1.1 Motivation	1
1.2 Overview of the Main Results	3
1.3 Organization of the Thesis	3
2 Review on Fixed-Parameter Tractability	5
2.1 Preliminaries	5
2.2 Computational Complexity	7
2.2.1 Is $P = NP$?	7
2.2.2 NP-completeness	7
2.3 Parameterized Complexity	8
2.3.1 Fixed-Parameter Tractability	9
2.3.2 Kernelization and Bounded Tree Search	10

3	State of the Art in Max-Leaf Spanning Tree Calculation	14
3.1	Existence Proofs on FPT Algorithms	14
3.1.1	Nonconstructive Proof with Well-Partial-Order Theory	15
3.1.2	Linear Time Minor Tests	16
3.2	Coordinatized Kernels and Catalytic Reductions	19
3.3	Max-leaf Spanning Tree Algorithms with Heuristics	20
3.3.1	The Power of Local Optimization	21
3.3.2	Greedy Approximation Algorithm	22
4	Algorithms and Implementation	25
4.1	Graph Data Structure	25
4.2	Graph Algorithms and Graph Library	29
4.2.1	Breadth-First Search	29
4.2.2	Depth-First Search	30
4.2.3	Bridge	33
4.2.4	Cut-Vertex	34
4.3	Sequential FPT Algorithm	34
4.3.1	Kernelization	35
4.3.2	Bounded Tree Search	36
4.3.3	Restarting Technique	40
4.4	Parallel FPT Algorithm	40
4.4.1	Catalyst Parallel Algorithm	41
4.4.2	In-branch Parallel Algorithm	41
5	Experimental Results	44
5.1	Random Graph Generator	44
5.2	k -Leaf Spanning Tree Tester	47
5.3	Experimental Setup and Methodology	50

5.4	Data Sets	51
5.5	Performance Results: Sequential Experiments	52
5.6	Performance Results: Parallel Experiments	52
Bibliography		73

List of Tables

4.1	Class LabelList Public Interface	28
4.2	Graph Library	29
4.3	Major Functions of Sequential FPT Implementation	35
5.1	Data Sets	51
5.2	Measured Sequential Wall Clock Times in Seconds	55
5.3	Measured Parallel Wall Clock Times in Seconds on 32 Processors	57
5.4	Measured Parallel Wall Clock Times in Seconds on 3, 5, 9 and 17 Processors	58
5.5	Measured Relative Speedups on 3, 5, 9 and 17 Processors	59

List of Figures

2.1	Relationships among P, NP and NPC	8
2.2	An Example of Vertex Cover	12
3.1	$W_4 \leq_m Q_3$	15
3.2	$GR_{2 \times 5}$ and CC_5	17
3.3	Graphs that are a minor of a $2 \times k$ grid and a minor of a circus graph	18
3.4	$K_{1,5}$	19
3.5	Examples of Local-change	21
3.6	Example of a Maximally Leafy Forest	23
4.1	Adjacency List and Adjacency Matrix Representation of an Undirected Graph	27
4.2	DFS Path of Restarting Process	40
4.3	Catalyst Parallelization	42
4.4	In-branch Parallelization	43
5.1	A Random Graph Generated by P0	46
5.2	Test Procedure of the Existence of a Spanning Tree on a Graph G	49
5.3	Sequential Wall Clock Time in Seconds for G^* , INET.* and TINs	53
5.4	Sequential Wall Clock Time in Seconds for G^* , RG.* and Protein Sequences	54
5.5	Parallel Wall Clock Time in Seconds on 32 Processors for G^* , INET.* and TINs	56

5.6	Parallel Wall Clock Times and Speedups for G0 on 3, 5, 9 and 17 Processors	60
5.7	Parallel Wall Clock Times and Speedups for G2 on 3, 5, 9 and 17 Processors	61
5.8	Parallel Wall Clock Times and Speedups for INET.500 on 3, 5, 9, 17 Processors	62
5.9	Parallel Wall Clock Times and Speedups for INET.600 on 3, 5, 9, 17 Processors	63
5.10	Parallel Wall Clock Times and Speedups for INET.700 on 3, 5, 9, 17 Processors	64
5.11	Parallel Wall Clock Times and Speedups for ATIN on 3, 5, 9 and 17 Processors	65
5.12	Parallel Wall Clock Times and Speedups for JAPAN on 3, 5, 9, 17 Processors	66
5.13	Superlinear Speedup Analysis	68
5.14	Average Relative Speedup for G*	70
5.15	Average Relative Speedup for INET.*	71
5.16	Average Relative Speedup for TINs	72

Chapter 1

Introduction

This chapter describes motivation and overview of the main results, and organization of this thesis.

1.1 Motivation

A Graph (defined in Section 2.1) is an abstract concept. It does not indicate what vertices or edges represent. Vertices and edges could be cities with connecting roads, or computers connected by a LAN (Local Area Network) or WAN (Wide Area Network). By leaving out the detailed representation we can construct a theory that is reusable, that can help us solve lots of different kinds of problems. Many problems arising in computer science can be viewed as a problem of, given a graph, finding a spanning tree (defined in Section 2.1) that satisfies a specified property.

The *max-leaf spanning tree problem* is defined as follows: given a connected graph G , find a spanning tree of G with maximum number of leaves. The max-leaf spanning

tree problem has been proved to be NP-complete [9], MAX SNP-complete [8] and Fixed-Parameter Tractable [4]. Fixed-Parameter Tractability (FPT) redefines the max-leaf spanning tree problem as a *k-Leaf spanning tree problem*: given a connected graph G and a positive integer k , find out whether G possesses a spanning tree in which k or more vertices are of degree one. In this thesis, the terms of max-leaf spanning tree and k -leaf spanning tree are used interchangeably.

The max-leaf spanning tree problem has been extensively studied [1, 4, 6, 7, 10, 12, 14, 15, 20]. A few of the previous work focused on finding spanning trees with many leaves in graphs whose vertices are of minimum degree d , where d is at least 3. For such graphs, previous studies have provided approaches to obtain good lower bounds to the number of leaves in a spanning tree [10, 12, 20]. The existence of k -leaf spanning tree was nonconstructively proved by Fellows and Langston [6] as well as Bodlaender [1]. Fellows *et al* designed a FPT approach, which can be applied to calculate the k -leaf spanning tree in $O(n+(k+1)(14.23)^k)$ [7]. Lu and Ravi contributed their max-leaf spanning tree algorithm based on heuristics [14, 15].

The max-leaf spanning tree problem finds its applications in communication networks, circuit layouts and other graph-theoretic problems. For example, Dijkstra studied the self-stabilization problem of distributed processors and proposed a solution based on mutual exclusion [3]. Kranakis *et al* studied the *VC-dimension* [21] for set systems induced by *connected sets* [11] and showed that for a given graph the maximum size of a *shattered set* [13] for the connectedness property differs by at most one from the number of leaves in a max-leaf spanning tree [13].

1.2 Overview of the Main Results

- Studied in detail and implemented Fellows *et al*'s sequential coordinatized kernels and catalytic reductions FPT algorithm for max-leaf spanning tree [7]. This is the first implementation of Fellows *et al*'s sequential FPT algorithm.
- Introduced the restarting method (explained in Chapter 4) into the implementation.
- Studied and implemented the facilities: one is designed to generate random graphs, and the other is used to verify the solutions produced by Fellows *et al*'s algorithm (its solution is just a set of leaves and not the spanning tree).
- Studied parallelization of Fellows *et al*'s FPT algorithm and implemented the parallelization.
- Tested both the sequential and the parallel algorithms extensively.
- Observed superlinear speedup, which indicates a possible improvement of Fellows *et al*'s sequential FPT algorithm by combining it with the sequential simulation of our parallel algorithm.

1.3 Organization of the Thesis

The rest of the thesis is organized as follows: Chapter 2 first introduces the graph related preliminaries. Then it describes the classical computational complexity including three classes of problems: P, NP, and NPC (NP-Complete Problems). Finally, Chapter 2 describes the parameterized complexity, which includes the FPT and its ad hoc methods: kernelization and bounded search tree.

Chapter 3 presents the related work on the max-leaf spanning tree problem. Two non-constructive proofs on the existence of max-leaf spanning tree, coordinatized kernels and catalytic reductions FPT algorithm, and approximate algorithms are covered.

Chapter 4 provides relevant details on the sequential and the parallel FPT algorithms of k -leaf spanning tree and their implementation. It begins with the design decision for the graph data structure. It is followed by the introduction of several graph algorithms used by the FPT algorithm or mentioned in this thesis. They are graph search algorithms: breadth-first search (BFS) and depth-first search (DFS), and graph connectivity algorithms: bridge and cut-vertex. Three of the above mentioned algorithms and several graph operations have been implemented. A graph library has been formed for software reuse purpose based on these implementations. Furthermore, Chapter 4 describes the sequential FPT algorithm in detail and the restarting technique. It ends with the explanation of the parallel FPT algorithms of k -leaf spanning tree: catalyst parallelization and in-branch parallelization.

Chapter 5 first presents two facilities: the random graph generator and the k -leaf spanning tree tester. Then it describes the system environment and methodology of experiment. Furthermore, it describes the data sets used for the experiments. At the end, it shows some experimental results, observations and analysis.

Chapter 2

Review on Fixed-Parameter Tractability

This chapter concentrates on the introduction to the background of FPT technique. It starts from preliminaries where terminologies and notations are introduced. Then it reviews computational complexity theory, where it refers to three classes of problems: P, NP, and NPC (NP-Complete Problems) and shows the current understanding of the relationship among them. It is followed by the review of parameter complexity theory and its essence, namely FPT, as well as the ad hoc FPT approaches.

2.1 Preliminaries

In this section, some graph related terminologies and notations are introduced.

A graph $G = (V, E)$, consists of a set V of *vertices* (or *nodes*) and a set E of *edges*. An edge uv is a link joining two vertices u and v , u and v are then incident with edge uv . u and v are *neighbors* to each other, they are also called to be *adjacent*. Similarly, two distinct edges are *adjacent* if they have a vertex in common. The *degree* of a vertex is the number of edges incident with it. A *walk* is a ‘way’ of getting from one vertex to another,

and consists of a sequence of edges. A *path* is a walk without a vertex appearing more than once. A *cycle* is a path which begins and ends at same vertex. A *bridge* (or *cut-edge*) is an edge, the removal of which causes the disconnection of a graph. A *cut-vertex* is a vertex, the removal of which causes the disconnection of a graph.

A *simple graph* is the graph without loops (edges joining a vertex to itself) and multiple edges (more than one edge joining a pair of vertices). Graphs discussed in this thesis are simple ones. A *subgraph* of graph $G = (V, E)$ is a graph, whose vertices belong to V and whose edges belong to E . A *connected graph* is in one piece, any two vertices are connected by a path. A *planar graph* is a graph that can be drawn in the plane without crossings - no two edges intersect geometrically except at a vertex to which both are incident.

Subgraphs are obtained by *deleting edges and vertices*. Let e be an edge of a graph G , we denote by $G - e$ the graph obtained from G by deleting the edge e . Similarly, let v be a vertex of G , we denote by $G - v$ the graph obtained from G by deleting the vertex v together with the edges incident with v . *Contracting edge* means removing an edge and identifying its ends, v and w , such that the resulting vertex is incident with those edges (other than edge vw) that were originally incident with v or w .

A *tree* is a connected graph without cycles. A particular vertex on a tree is specified as *root*. On a tree T rooted at vertex r , if vertex u is on a path from r to v , then u is an ancestor of v and v is a descendant of u . In addition, if uv is an edge of T , then u is the parent of v and v is a child of u . A vertex of T with no children is called a *leaf*, or *external vertex*, while all the other vertices are referred to as *internal*.

A *spanning tree* of a graph G is a tree that connects all the nodes of G . It partitions the edge set E into two sets: tree edge and non-tree edge sets. A *binary tree* is a tree in which all nodes have at most two children. The children of a node in a binary tree are distinguished as the left child or right child of their parent node.

2.2 Computational Complexity

Computational complexity theory is part of the theory of computation. It deals with the resources required during the computation to solve a given problem. The most common resources are time (the steps taken to solve a problem) and space (the memory taken to solve a problem).

2.2.1 Is $P = NP$?

In computational complexity, the class P (Polynomial-time) consists of all the Yes/No decision problems that can be solved on a deterministic sequential machine in an amount of time that is polynomial in the size of the input; the class NP (Non-deterministic Polynomial-time) consists of all the decision problems whose solutions can be verified in polynomial time by given the right information, or equivalently, whose solutions can be found in polynomial time on a non-deterministic machine.

The biggest open question in theoretical computer science concerns the relationship between the classes P and NP : Is $P = NP$? In essence, the $P = NP$ question asks: if solutions to a Yes/No problem can be verified quickly, can the answers also be computed quickly? Most people think that the answer is probably “No”; some people believe the question may be undecidable from the currently accepted axioms.

2.2.2 NP-completeness

To attack the $P = NP$ question, the concept of NPC is very useful. NPC problems are the “toughest” problems in NP in the sense that they are the ones most likely not to be in P . If a single NPC problem could be shown in P , then $P = NP$. Unfortunately, many important problems have been shown in NPC and not a single fast algorithm for any of them is known.

Figure 2.1: Relationships among P, NP and NPC

A decision problem L is *NP-complete* if it is in NP and if every other problem in NP is reducible to it. Here “Reducible” means that for every NP problem L' , there is a polynomial time algorithm which transforms instances of L' into instances of L , such that the two instances have the same truth values. As a consequence, if we had a polynomial time algorithm for L , we would solve all NP problems in polynomial time.

In mathematical terms, a problem L is NP-complete, if

1. $L \in \text{NP}$, and
2. $L' \leq_p L$ for every $L' \in \text{NP}$.

Figure 2.1 shows how most theoretical computer scientists view the relationships among P, NP, and NPC. Both P and NPC are wholly contained within NP, and $P \cap \text{NPC} = \emptyset$.

2.3 Parameterized Complexity

Compared to computational complexity theory which deals with decision problems, parameterized complexity theory examines decision problems with parameters as its name implies. It tries to distinguish problems that have good behavior when parameterized from those where parameterizing the problems bring little help. In parameterized complexity,

the focus is not on whether a problem is “hard” (the theory assumes that most interesting problems are intractable when considered classically), but on the question: What makes the problem computationally difficult?

In classical computational complexity, a decision problem is specified by two parts of information:

1. The input to the problem.
2. The question to be answer.

In parameterized complexity, there are three items of a problem specification:

1. The main part of the input.
2. The aspects of the input that constitute the parameter.
3. The question.

Practical computer scientists have frequently complained about the classical complexity that problems often turn out to be much easier in practice than one would expect from theoretical analyses. Parameterization fundamentally enriches the classical complexity analysis by which theory engages practice in the specification of computational problems. The parameter has provided a systematic way of including distributional information in complexity analysis.

2.3.1 Fixed-Parameter Tractability

In the same way that the notion of polynomial time is the essence of computational complexity, the essence of parameterized complexity is the notion of FPT. We may regard a

problem as a combination of “good” and “bad” complexity behavior, where the main part of the problem input contributes to the overall complexity in a good way (polynomially), but some aspects of the input (the parameter) may contribute to the problem complexity in a bad way (non-deterministic polynomially). FPT confines the parameter to some small but still useful range in order to solve the “slice” of the problem in the natural world.

Let Σ be a finite set of symbols and let problem L be a parameterized problem such that $L \subseteq \Sigma^{*1} \times \Sigma^*$. Problem L is *Fixed-Parameter Tractable*, if there exists an algorithm that decides, given an input $(x, y) \in \Sigma^* \times \Sigma^*$, whether $(x, y) \in L$ in time $f(k)n^\alpha$, where $|x|^2 = n$, $|y| = k$ is the parameter, α is a constant independent of n and k , and f is an arbitrary function. The main part x of the input to the problem contributes to the overall complexity polynomially as n^α , where the size n of the problem is frequently quite large. The parameter part of the input contributes to the problem complexity the arbitrary function $f(k)$, where the parameter k is hopefully small. The connection between the two contributions is specified to be multiplicative: $f(k)n^\alpha$. One of the fundamental properties of FPT is that the definition is unchanged if the connection between the two contributions becomes additive: $f(k) + n^\alpha$.

2.3.2 Kernelization and Bounded Tree Search

Two fundamental methods for solving FPT problems are Kernelization and Bounded Tree Search (BTS).

Kernelization is the reduction of a problem instance I with input (x, k) to an “equivalent” instance I' with input (x', k') in polynomial time, where the size of I' is bounded by some function of the parameter k . The instance I' is then exhaustively analyzed, probably by BTS, and a solution for I' can be lifted to a solution for I , in the case where a solution

¹ Σ^* denotes all strings made up of symbols from Σ .

² $||$ means cardinality.

exists. This technique will often lead to an additive rather than a multiplicative $f(k)$ exponential factor. A problem is Fixed-Parameter Tractable if and only if it is kernelizable [5].

Compared to kernelization, BTS is easier to apply. Many combinatorial problems can be solved by BTS algorithms that can be decomposed into two distinct parts:

- Within the algorithm we compute some searching space which is often an exponential-sized search tree (perhaps inefficiently).
- On each branch of the search tree, we execute some relatively efficient algorithm, which is often based on depth-first search (see Section 4.2.2).

The exponential worst case complexity of BTS algorithms comes from problem instances where complete tree traversal is needed. The critical observation is that for many parameterized problems, the size of the search tree only depends on the parameter. Therefore, for a fixed k , the search space becomes constant size, then BTS is efficient for each fixed k .

We illustrate kernelization and BTS methods by the example of the vertex cover problem. The vertex cover problem is known to be NP-complete, and fixed-parameter tractable in the context of parameterized complexity. A vertex cover of a graph $G = (V, E)$ is a collection of vertices V' of G such that for all edges uv of G either $u \in V'$ or $v \in V'$. Figure 2.2 shows a connected graph, where $\{3,4\}$ is a vertex cover. The vertex cover problem takes graph G and parameter k as input. It asks if G has a vertex cover of size $\leq k$.

From observation, for a graph G , any vertex of degree greater than k must belong to every k -element vertex cover of G . Then the following kernelization procedure can be applied:

1. Locate all vertices in G of degree greater than k ; let p equal the number of such

Figure 2.2: An Example of Vertex Cover

vertices. If $p > k$, there is no k -vertex cover. Otherwise, let $k' = k - p$.

2. Delete all p vertices found in step 1 and the edges incident to them. If the resulting graph G' has more than $k'(k + 1)$ vertices, there is no k -vertex cover.
3. If G' has no k' -vertex cover, there is no k -vertex cover. Otherwise, all vertices in any k' -vertex cover of G' plus all p vertices from step 1 constitutes a k -vertex cover of G .

Through BTS, a binary search tree of height k is constructed as follows. Label the root of the search tree with the empty set $\emptyset$ and the graph G . Randomly choose an edge $uv \in E$. In any vertex cover V' of G , it must have either $u \in V'$ or $v \in V'$, so create two children of the root node corresponding to these two possibilities. Thus, the first child is labelled with u and $G - u$, and the second child is labelled with v and $G - v$. The set of vertices labelling a node represents a “possible” subset of a vertex cover, and the graph labelling the node represents what remains to be explored in G . In general, for a node labelled with the set of vertices S and the subgraph H of G , randomly choose an edge $uv \in E(H)$ and create the two child nodes labelled, respectively, $S \cup u$ and $H - u$, and $S \cup v$ and $H - v$. If a node at height at most k in the search tree is created with the label of a graph having

no edges, then a vertex cover of cardinality at most k has been found. There is no need to explore the search tree beyond height k .

Chapter 3

State of the Art in Max-Leaf Spanning Tree Calculation

In Section 3.1, two proofs on the existence of FPT algorithms for k -leaf spanning tree are discussed. Fellows and Langston exploited minor-closure to ensure a low-degree polynomial running time [6]. Bodlaender expedited the time complexity by applying a minor test method [1]. The focus of Section 3.2 is on coordinatized kernels and catalytic reductions FPT algorithms designed by Fellows *et al.* In Section 3.3, two heuristic algorithms are presented. Lu and Ravi contributed a series of approximation algorithms for the max-leaf spanning tree problem based on the technique of local optimization [14]. Furthermore, Lu and Ravi gave another greedy approximation algorithm with a better time complexity [15].

3.1 Existence Proofs on FPT Algorithms

Existence proof is a proof which shows the existence of a solution without providing a specific example or an algorithm for producing an example. It is also called *nonconstructive*

Figure 3.1: $W_4 \leq_m Q_3$.

proof. Both of the following nonconstructive proofs apply Robertson and Seymour's *graph minor theorems* (Robertson - Seymour theorems) [17, 18].

3.1.1 Nonconstructive Proof with Well-Partial-Order Theory

Fellows and Langston nonconstructively proved the existence of k -leaf spanning tree in $O(n^2)$ time complexity [15]. Their method applied Robertson - Seymour theorems on the well-partial-ordering of graphs under minor order [17, 18]. Fellows and Langston first observe that max-leaf spanning tree is fixed-parameter tractable [15].

A graph H is less than or equal to a graph G in *minor order*, written $H \leq_m G$, if and only if a graph isomorphic to H can be obtained from G by a series of these two operations: taking a subgraph, and contracting an edge. For example, the construction depicted in Figure 3.1 shows that W_4 is a minor of Q_3 . A family F of graphs is said to be *closed* under minor order if G is in F and $H \leq_m G$ can together imply that H is in F .

Theorem 1 (see[18]) *Graphs are well-partially-ordered¹ by $\leq_m$.*

Theorem 2 (see[17]) *For every fixed graph H , the problem that takes a graph G as input and determines whether $H \leq_m G$ is solvable in polynomial time.*

¹A partially-ordered set $(X, \leq)$ is well-partially-ordered if (1) any subset of X has finitely many minimal elements and (2) X contains no infinite descending chain $x_1 \geq x_2 \geq x_3 \geq \dots$ of distinct elements.

Theorems 1 and 2 guarantee the existence of a polynomial-time decision algorithm for any minor closed family of graphs, but do not provide any details of what that algorithm might be. The interesting feature of Theorems 1 and 2 is the low degree of the polynomials bounding the time complexity of the decision algorithms. Let n be the number of vertices in G , the time spent to recognize F is $O(n^3)$. If F excludes a planar graph, then F has bounded tree-width and the time complexity decreases to $O(n^2)$ [19].

For the k -leaf spanning tree problem (does graph G have a k -leaf spanning tree?), consider a set of no-instances of G as the family F of graphs, none of whose connected components has a k -leaf spanning tree. Then F is closed under minor order. For an input graph, only its connectedness and non-membership in F need to be tested, so that the existence of k -leaf spanning tree can be decided in $O(n^2)$ time.

3.1.2 Linear Time Minor Tests

Bodlaender further extended the results of Fellows and Langston. He presented a linear time algorithm to test if a graph contains another graph as a minor [1]. As a corollary it follows that for fixed k , one can test in linear time whether a given graph $G = (V, E)$, contains a k -leaf spanning tree.

The $n \times m$ grid graph is the graph $GR_{n \times m} = (V_{n \times m}, E_{n \times m})$, with $V_{n \times m} = \{(i, j) : 0 \leq i \leq n - 1, 0 \leq j \leq m - 1\}$ and $E_{n \times m} = \{((i_1, j_1), (i_2, j_2)) : (i_1, j_1), (i_2, j_2) \in V_{n \times m}, \text{ and } (|i_1 - i_2| = 1 \wedge j_1 = j_2) \text{ or } (i_1 = i_2 \wedge |j_1 - j_2| = 1)\}$. The l th circus graph is the graph $CC_l = (\{x\} \cup \{y_i : 1 \leq i \leq l\} \cup \{z_i : 1 \leq i \leq l\}), \{(x, y_i) : 1 \leq i \leq l\} \cup \{(y_i, z_i) : 1 \leq i \leq l\} \cup \{(z_i, z_{i+1}) : 1 \leq i \leq l - 1\})$. The 2×5 grid graph and the fifth circus graph are shown, as examples, in Figure 3.2.

Lemma 1 *Let H be a minor of the $2 \times k$ grid graph, and a minor of the l th circus graph. Then there exists an algorithm, that given a graph G , decides in $O(n)$ time whether G*

Figure 3.2: $GR_{2 \times 5}$ and CC_5

contains H as a minor.

$K_{1,k}$ is like a “ k -star” graph, Figure 3.4 shows an example of $k_{1,5}$. If G contains a k -leaf spanning tree, then $K_{1,k}$ graph can be obtained by removing all non-tree edges from G and contracting over every tree edge between two non-leaf vertices, such that G contains the graph $K_{1,k}$ as a minor. Suppose that G is connected and contains $K_{1,k}$ as a minor, one can obtain $K_{1,k}$ from G by first choosing a spanning tree of G and then applying a number of contractions. This spanning tree must have at least k leaves. This leads the lemma as follows.

Lemma 2 G contains a spanning tree with $\geq k$ vertices of degree 1, if and only if G is connected and it contains the graph $K_{1,k}$ as a minor.

From observation, the class of graphs that are both a minor of a $2 \times k$ grid graph and a minor of a circus graph simultaneously includes the following graphs:

1. all paths (Figure 3.3 (a))
2. all cycles (Figure 3.3 (b))

Figure 3.3: Graphs that are a minor of a $2 \times k$ grid and a minor of a circus graph

3. all graphs that consist of two cycles having exactly one edge in common (Figure 3.3 (c))
4. all caterpillars with hairs of length 1 (Figure 3.3 (d))
5. all graphs that consist of a number of paths and cycles which have exactly one vertex in common (Figure 3.3 (e))
6. all subgraphs of the above graphs

The graph $K_{1,k}$ is a subgraph of Figure 3.3 (d) or (e) and is a minor of $GR_{2 \times k}$ and a minor of CC_k . By Lemma 1, $O(n)$ time is required to decide whether G contains $K_{1,k}$ as a minor. According to Lemma 2, if G contains the graph $K_{1,k}$ as a minor, G has a k -leaf spanning tree. This leads to Theorem 3.

Figure 3.4: $K_{1,5}$

Theorem 3 *For fixed k , the existence of a k -Leaf Spanning Tree can be decided in $O(n)$ time.*

3.2 Coordinatized Kernels and Catalytic Reductions

Fellows *et al* presented an FPT approach: coordinatized kernels and catalytic reductions, and the solution to the max-leaf spanning tree problem by using this approach [7]. They slightly changed the problem to a *catalytic max-leaf spanning tree problem*. The inputs of the problem are a graph $G = (V, E)$ with a distinguished *catalytic vertex* (an internal node of a spanning tree) $t \in V$ and a parameter k . The question asks if G has a spanning tree T with at least k leaves, such that t is an internal vertex of T . This approach combines kernelization (coordinatized kernel) with BTS (catalytic reduction). The running time for a graph with n nodes is $O(n + (k+1)(14.23)^k)$ [7].

Fellows *et al* concluded the following reduction rules for kernelization. If graph G has two adjacent vertices u and v of degree 2, neither of which is the catalytic vertex t , then contract uv to obtain G' and let $k' = k$ if uv is a bridge, or delete the edge uv to obtain G' and let $k' = k$ if uv is not a bridge. After kernelization is performed, the size of the original problem is bounded by $5.75k$ [7].

The catalytic reduction follows after coordinatized kernel. Considering a neighbor u of t in G , (G, k) with catalytic vertex t is a yes-instance if and only if one of the following two branch instances is a yes-instance. The first branch is developed on the assumption that u is an internal vertex of a k -leaf spanning tree T for which t is internal. The second branch is developed on the assumption that u is a leaf of T where t is internal. Through first branching, we have such an instance (G', k) that G' is obtained from G by contracting the edge between t and u . The resulting combined vertex is the catalytic vertex for G' . Through second branching, we have $(G', k - 1)$, where G' is obtained by deleting u . Since the value of parameter k in the second branching has been decreased by 1, re-kernelization can be performed to output the graph with size of at most $5.75(k-1)$. The re-kernelization part is the key to improve the efficiency of the algorithm.

A catalytic vertex can be any one in a set of $k + 1$ vertices which are randomly selected from G . Because if (G, k) is a yes-instance for the k -leaf spanning tree problem, at least one of those $k+1$ vertices can be an internal vertex of the existing k -leaf spanning tree. Therefore, the $k + 1$ branches, which contain BTS with $k+1$ catalytic vertices, must be explored until a yes-instance has been found. If none of $k+1$ branches finds a yes-instance, no k -leaf spanning tree exists.

3.3 Max-leaf Spanning Tree Algorithms with Heuristics

Approximation algorithm is another approach to solve NP-complete problem. It returns near-optimal solution which may be good enough in practice. *Approximation ratio* ρ is used to evaluate the performance of an approximation algorithm. An algorithm for a problem has an *approximation ratio* of ρ if the cost of the solution produced by the approximation algorithm is within a factor of ρ of the cost of an optimal solution.

Figure 3.5: Examples of Local-change

3.3.1 The Power of Local Optimization

Lu and Ravi proposed the local-improvement heuristics that perform local changes to increase the number of leaves in the resulting spanning tree [14]. It is a natural notion to do such a local change, namely, swapping a tree edge for a non-tree edge. The algorithm performs such local changes that increase the number of leaves until no improvement is possible. The resulting locally optimal solution becomes the approximate solution.

The algorithm starts with an arbitrary spanning tree T of a given graph G . Let $e = (u, v)$ be a non-tree edge, and f be an edge in the unique tree path of T connecting u and v . Making e a tree edge and f a non-tree edge is denoted as $(e; f)$ with respect to T . A local-change procedure involving k tree edges is called as k -changes. Figure 3.5 shows the examples of 1-changes and 2-changes by swapping 1 edge and 2 edges per local-change respectively. In Figure 3.5 (a), the tree edges of T are shown in dark lines and the non-tree edges are dotted. The change $(e; f)$ is a 1-change on T , which results T_1 with one more leaf as shown in Figure 3.5 (b). T_1 does not allow any 1-change but 2-changes $(e_1, e_2; f_1, f_2)$ to give T_2 one more leaf as shown in Figure 3.5 (c). The 1-change and 2-changes have approximation ratios of 5 and 3 respectively. A k -changes on a graph with n nodes requires time $O(n^{3k+1})$.

By varying the limit on the number of tree edges participating in a single local-change

procedure, a series of approximation algorithms are derived. Although increasing the limit can yield a better solution, but it corresponds to allowing more local optimization steps. Therefore, the bigger the limit, the higher the time complexity.

3.3.2 Greedy Approximation Algorithm

Lu and Ravi designed a greedy algorithm with time complexity $O((m + n)\alpha(m, n))^2$, where m is the number of edges and n is the number of vertices of the graph [15]. This algorithm has approximation ratio of 3.

Let V_i denote the set of nodes of degree i in G . Let $\bar{V}_i(G)$ denote the set of nodes that have degree at least i in G . Clearly, $\bar{V}_0(G) = V(G)$, and the leaves of G are the nodes in $V_1(G)$. A subtree T of graph G is *leafy* if $\bar{V}_3(T)$ is not empty, and every node in $V_2(T)$ is adjacent in T to exactly two nodes in $\bar{V}_3(T)$. A forest F of G is *leafy* if F is composed of disjoint leafy subtrees of G . F is called as *maximally leafy* if F is not a subgraph of any other leafy forest of G .

A maximally leafy forest has the following properties, which are illustrated by the example in Figure 3.6. In the figure, the dark edges of the graph G are the edges in the maximally leafy forest F , and the gray ones are the remaining edges in G . F is composed of three leafy subtrees T_1 , T_2 , and T_3 .

- Let ω be a node in $\bar{V}_3(T_i)$, then ω cannot be adjacent in G to any node not in T_i . In Figure 3.6, nodes x_1 and x_2 are two examples of ω . Suppose x_1 were adjacent to a node such as x_5 , then F would not be maximal since the edge (x_1, x_5) could be added to F .
- Let ω be a node in T_i . Let ω_1 and ω_2 be two distinct nodes adjacent to ω in G . If ω_1

² $\alpha(m, n)$ is a very slowly growing function.

Figure 3.6: Example of a Maximally Leafy Forest

is not in F , then ω_2 must be in T_i . In Figure 3.6, nodes x_3 is an example of ω . If x_3 had two neighbors not in F , both these edges could be added to F contradicting its maximality.

- Let ω be a node not in F . If ω is adjacent to two distinct nodes not in F , then the degree of ω in G is two. In Figure 3.6, nodes x_5 and x_6 are two examples of ω . If the degree of say x_5 were greater than two, then x_5 and its three neighbors not in F could be added as an additional star in F contradicting its maximality again.

This approximation algorithm for max-leaf spanning tree has two phases. The first phase is invoking Algorithm 1 to obtain a maximally leafy forest F of graph G . The second phase is adding edges to F to make it a spanning tree of G . The second phase can be done by shrinking each leafy subtree in F into a single vertex, then finding a spanning tree for the corresponding shrunk graph.

Algorithm 1 *MaximallyLeafyForest*(G) – Find a maximally leafy forest F for G

```

1:  $F \leftarrow \emptyset$ 
2: for every node  $v \in G$  do
3:    $S(v) \leftarrow v$ 
4:    $d(v) \leftarrow 0$ 
5: end for
6: for every node  $v \in G$  do
7:    $S' \leftarrow \emptyset$ 
8:    $d' \leftarrow 0$ 
9:   for every node  $u$  which is adjacent to  $v$  in  $G$  do
10:    if  $u \notin S(v)$  and  $S(u) \not\subseteq S'$  then
11:       $d' \leftarrow d' + 1$ 
12:       $S' \leftarrow S(u)$ 
13:    end if
14:  end for
15:  if  $d(v) + d' \geq 3$  then
16:    for every  $S(u) \in S'$  do
17:       $F \leftarrow uv$ 
18:       $S(v) \cup S(u)$ 
19:       $d(u) \leftarrow d(u) + 1$ 
20:       $d(v) \leftarrow d(v) + 1$ 
21:    end for
22:  end if
23: end for

```

Chapter 4

Algorithms and Implementation

The focus of this chapter is to provide relevant details on sequential and parallel FPT algorithm for k -leaf spanning tree calculation and their implementation. Section 4.1 describes how to represent a graph and the implementation of graphs. Section 4.2 introduces several graph algorithms and the graph library. This chapter is completed with the description of the sequential and parallel FPT algorithm based on coordinatized kernels and catalytic reductions [7].

4.1 Graph Data Structure

There are two standard ways to represent a graph $G = (V, E)$: as a collection of adjacency lists or as an adjacency matrix. The adjacency list representation is usually preferred, because it provides a compact way to represent *sparse* graphs, where $|E|$ is much less than $|V|^2$. An adjacency matrix may be preferred, however, when the graph is *dense*, where $|E|$ is close to $|V|^2$, or when we need to be able to tell quickly if there is an edge connecting two given vertices.

For the *adjacency matrix representation* of G , we assume that the vertices are numbered $1, 2, \dots, |V|$ in some arbitrary manner. Then the adjacency matrix representation of G consists of a $|V| \times |V|$ matrix A_{ij} such that $A_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E, \\ 0 & \text{otherwise.} \end{cases}$ Figure 4.1 (c) is the adjacency matrix of the undirected graph G as shown in Figure 4.1 (a). The adjacency matrix of a graph requires $\Theta(V^2)$ space complexity, independent of the number of edges in the graph.

The *adjacency list representation* of G consists of an array A of $|V|$ lists, one for each vertex in V . For each $u \in V$, the adjacency list $A[u]$ contains all the vertices v such that there is an edge $(u, v) \in E$. That is, $A[u]$ consists of all the vertices adjacent to u in G . The vertices in each adjacency list are typically stored in an arbitrary order. Figure 4.1 (b) is an adjacency list representation of the undirected graph as shown in Figure 4.1 (a).

In our implementation, we choose the adjacency list to represent the graphs. G is represented by an array A of $|V|$ lists. For each $u \in V$, the pointer $A[u]$ points to an object **labelList** which stores all vertices adjacent to u in G (u 's neighbors). The interface **LabelList** is outlined in Table 4.1. All neighbors of u are linked together as a linked list. Every node in the linked list is represented by a *struct* of its labelling and a pointer to its next node.

Figure 4.1: Adjacency List and Adjacency Matrix Representation of an Undirected Graph

<i>Function</i>	<i>Description</i>
<i>GetDegree</i>	Return the degree of this vertex.
<i>GetLabel</i>	Return the label of first node.
<i>GetHeadP</i>	Return the pointer of the first node.
<i>GetTailP</i>	Return the pointer of the last node.
<i>GetCurrentP</i>	Return the pointer of the current node.
<i>GetCurrentLabel</i>	Return the label of the current node.
<i>GetInternalFlag</i>	See Section 4.3 for the internal flag.
<i>SetInternalFlag</i>	Set internal flag to either 1 or 0.
<i>SetCurrentP</i>	Set the pointer to a specific position.
<i>IsEmpty</i>	Return true if this vertex has no neighbor.
<i>InsertLabel</i>	Add a new neighbor
<i>RemoveFront</i>	Remove the first node.
<i>RemoveCurrent</i>	Remove the current pointed node.
<i>ChangeCurrentLabel</i>	Change the current pointed node by a new one.
<i>AdvanceCurrentP</i>	Move the current pointer a position forward.

Table 4.1: Class LabelList Public Interface

4.2 Graph Algorithms and Graph Library

The purpose of a library is to provide a means of performing common tasks in some domain of application that the programming language does not directly address [16]. The graph library offers a functional method of enriching the existing programming languages with new instructions, implemented as procedure calls. The graph library contains the graph operations and graph algorithms, which are listed in Table 4.2. This section first introduces breadth-first search (BFS) and depth-first search (DFS), which are graph search algorithms and commonly presented in introductory to algorithm textbooks (e.g., [2]). It is followed by the description of two graph connectivity algorithms: bridge and cut-vertex.

<i>Procedures</i>	<i>Description</i>
<i>BFS</i>	Search a graph and produce a spanning tree. (See Algorithm 2)
<i>Bridge</i>	Inspect if an edge is a bridge. (See Algorithm 5)
<i>CutVertex</i>	Inspect if a vertex is a cut-vertex. (See Algorithm 6)
<i>Contract</i>	Contract over an edge.
<i>Delete</i>	Delete a vertex and its incident edges.
<i>DeleteEdge</i>	Delete an edge.

Table 4.2: Graph Library

4.2.1 Breadth-First Search

Breadth-First Search (*BFS*) is one of the algorithms for searching a graph. Given a graph G and a distinguished *source* vertex s , *BFS* systematically explores the edges of G to “discover” every vertex that is reachable from s . It computes the distance (smallest number of edges) from s to each reachable vertex. It also produces a “breadth-first tree” which is a

spanning tree of G rooted at s .

BFS colors each vertex white, gray, or black. All vertices start out white and may later become gray and then black. A vertex is *discovered* the first time it is encountered during the search, at which time it becomes non-white. Black vertices are adjacent to discovered vertices only. Gray vertices may have some undiscovered vertices adjacent.

BFS constructs a BFS tree, initially containing only its root, which is the source vertex s . Whenever a white vertex v is discovered in the course of scanning the adjacency list of an already discovered vertex u , the vertex v and the edges (u, v) are added to the tree. The algorithm is shown in Algorithm 2. It uses a FIFO (first in first out) queue (Q) to manage the set of gray vertices. Denote the predecessor or parent of a vertex u by $p[u]$, denote the distance of u from s by $d[u]$, denote the color of u by $c[u]$, denote the vertex set of G by $V[G]$. $\text{Enqueue}(Q, s)$ and $\text{Enqueue}(Q, v)$ put s and v into Q as the tail elements respectively, and $\text{Dequeue}(Q)$ moves the head element out of Q .

The time complexity of *BFS* algorithm is $O(|V| + |E|)$ because $O(|E|)$ time is spent in total scanning adjacency lists and the initialization overhead is $O(|V|)$.

In order to work with *Bridge* and *Cut-Vertex* algorithms, *BFS* is *extended* by adding the calculation of the number of vertices of the “breadth-first tree”. In the case of the *Bridge* algorithm, the implementation neglects the existence of the tested edge while scanning the adjacency lists of the already discovered vertices.

4.2.2 Depth-First Search

Depth-First Search (*DFS*) is another graph search algorithm, as its name implies, to search deeper in the graph whenever possible. In *DFS*, edges are explored out of the most recently discovered vertex v that still has unexplored edges leaving it. When all of v ’s edges have

Algorithm 2 $BFS(G, s)$ – Breadth-First Search of G with Source s

```

1: for each vertex  $u \in V[G] - s$  do
2:    $c[u] \leftarrow \text{WHITE}$ 
3:    $d[u] \leftarrow +\infty$ 
4:    $p[u] \leftarrow \text{NIL}$ 
5: end for
6:  $c[s] \leftarrow \text{GRAY}$ 
7:  $d[s] \leftarrow 0$ 
8:  $p[s] \leftarrow \text{NIL}$ 
9:  $\text{ENQUEUE}(Q, s)$ 
10: while  $Q \neq \emptyset$  do
11:    $u \leftarrow \text{DEQUEUE}(Q)$ 
12:   for each  $v \in u$ 's adjacency list do
13:     if  $c[v]$  is WHITE then
14:        $c[v] \leftarrow \text{GRAY}$ 
15:        $d[v] \leftarrow d[u] + 1$ 
16:        $p[v] \leftarrow u$ 
17:        $\text{Enqueue}(Q, v)$ 
18:     end if
19:   end for
20:    $\text{Dequeue}(Q)$ 
21:    $c[u] \leftarrow \text{BLACK}$ 
22: end while

```

been explored, the search backtracks to explore edges leaving the vertex where v was discovered. This process continues until we have discovered all the vertices that are reachable from the original source vertex. If any undiscovered vertices remain, then one of them is selected as a new source and the search is repeated from that source. This entire process is repeated until all vertices are discovered. *DFS* forms a depth-first forest composed of several depth-first trees rooted at source vertices.

The algorithm is shown in Algorithm 3 as *DFS* and Algorithm 4 as *DFS-Visit*. As *BFS*, when a vertex v is discovered during a scan of the adjacency list of an already discovered vertex u , *DFS* indicates v 's predecessor or parent $p[v]$ as u . Also like *BFS*, *DFS* colors vertex white initially, then gray when the vertex is discovered during the search, finally black when its adjacent list has been examined completely. *DFS* timestamps each vertex as well as. Each vertex v has two timestamps: the first timestamp $d[v]$ records when v is first discovered and grayed, and the second one $f[v]$ records when the search finishes examining v 's adjacency list and blackens v .

The overall time complexity of *DFS* is $\Theta(|V| + |E|)$, as *DFS* costs $\Theta(|V|)$ excluding *DFS-Visit* and *DFS-Visit* takes $\Theta(|E|)$ time in examining the whole adjacent lists.

Algorithm 3 *DFS*(G) – Depth-First Search of G

```

1: for each vertex  $u \in V[G]$  do
2:    $c[u] \leftarrow \text{WHITE}$ 
3:    $p[u] \leftarrow \text{NIL}$ 
4: end for
5:  $\text{time} \leftarrow 0$ 
6: for each vertex  $u \in V[G]$  do
7:   if  $c[u] = \text{WHITE}$  then
8:     DFS-Visit( $u$ )
9:   end if
10: end for
```

Algorithm 4 *DFS – Visit(v)* - Procedure invoked by *DFS(G)*

```

1:  $c[u] \leftarrow \text{GRAY}$ 
2:  $\text{time} \leftarrow \text{time} + 1$ 
3:  $d[u] \leftarrow \text{time}$ 
4: for each  $v \in u$ 's adjacency list do
5:   if  $c[v] = \text{WHITE}$  then
6:      $p[v] \leftarrow u$ 
7:     DFS-Visit( $v$ )
8:   end if
9: end for
10:  $c[u] \leftarrow \text{BLACK}$ 
11:  $f[u] \leftarrow \text{time} \leftarrow \text{time} + 1$ 

```

4.2.3 Bridge

Bridge is an algorithm to test if an edge is a bridge edge of a given graph G . It is based on the following idea: According to the definition of bridge, removing a bridge disconnects the graph into two parts, hence this bridge is the only connection between those two parts. If we use the *extended BFS* searching G , then some of the vertices (those in the part of G not including s) are not able to be reachable from s , so whether or not the edge is a bridge can be decided by comparing the number of vertices of the “breadth-first tree” with the one of G . The algorithm is shown in Algorithm 5.

Algorithm 5 *Bridge(G, e)* – Examine if e is a bridge edge of G .

```

1: Invoke the extendedBFS and get vertex number of the “breadth-first tree”, named as  $|TV|$ 
2: if  $|TV| = |V|$  then
3:   RETURN False
4: else
5:   RETURN True
6: end if

```

4.2.4 Cut-Vertex

Cut-Vertex can determine if a vertex v is a cut vertex of a given graph G . It takes a similar idea as Bridge algorithm. We first obtain G' by removing v from G , then call *extended BFS* on G' . As removing cut-vertex disconnects G into two parts, if v is a cut vertex, then the number of vertices of the “breadth-first tree” must be less than the one of G' minus 1. The algorithm is shown in Algorithm 6.

Algorithm 6 *CutVertex*(G, v) – Examine if v is a cut-vertex of G .

```

1: Generate the graph  $G'$  by removing  $v$  from  $G$ 
2: Invoke the extendedBFS on  $G'$  and obtain the vertex number of the “breadth-first
   tree”, named as  $|TV|$ 
3: if  $|TV| = |V| - 1$  then
4:   RETURN False
5: else
6:   RETURN True
7: end if
```

4.3 Sequential FPT Algorithm

In the implementation of *FBranch*, instead of storing the complete graph at every search tree node, we only store the modification information for greater time and space efficiency. With the modification information, the graph can be restored later. The functions *MakeUncontractDataList*, *MakeUndeleteDataList*, and *MakeUndeleteEdgeDataList* are used to keep the modification information, and the functions *Uncontract*, *Undelete*, *UndeleteEdge* and *Unkernelize* perform the restoration. Table 4.3 lists the major functions of the sequential FPT implementation.

<i>Function</i>	<i>Description</i>
<i>Kernelize</i>	Determine the existence of k -leaf spanning tree and encapsulate the reduction rules. (See Algorithm 7)
<i>FBranch</i>	Explore one of the $k+1$ branches. (See Algorithm 8)
<i>Unkernelize</i>	Undo kernelization.
<i>Uncontract</i>	Undo contraction.
<i>Undelete</i>	Undo vertex deletion.
<i>UndeleteEdge</i>	Undo edge deletion.
<i>MakeUncontractDataList</i>	Store the data required by “Uncontract”.
<i>MakeUndeleteEdgeDataList</i>	Store the data required by “UndeleteEdge”.
<i>MakeUndeleteDataList</i>	Store the data required by “Undelete”.

Table 4.3: Major Functions of Sequential FPT Implementation

4.3.1 Kernelization

The *Kernelize* algorithm determines the possibility of existence of a k -leaf spanning tree by *three conditions*:

1. If $n > 4(k+2)(k+1)$, then k -leaf spanning tree exists [4];
2. If $k \leq 0$ and $n \geq 1$, then k -leaf spanning tree exists;
3. If $k > 0$ and $n \leq 1$, then k -leaf spanning tree with a specified internal node (catalytic vertex) does not exist.

It also encapsulates three *reduction rules* to reduce the problem size:

1. If bridge edge uv has both its two incident nodes u and v of degree 2, then contract uv ;

2. If non-bridge edge uv has both its two incident nodes u and v of degree 2, then remove uv ;
3. If vertex v of degree 1 get its internal node flag marked 0 (0 marking external node; 1 marking internal node), then remove v .

The *Kernelize* algorithm is shown in Algorithm 7, which takes a graph G , the number of leaves k and a catalytic vertex t as input values, and outputs 1, 2, or 3 according to certain conditions. The three conditions will be checked twice in Line 1 - 9 and in line 28 - 36, because the kernelization operations may cause the conditions changed. If condition 1 ($n > 4(k + 2)(k + 1)$) satisfies, according to [4], G has a k spanning tree T . Obviously, if condition 2 ($k \leq 0$ and $n \geq 1$) is met, such a T exists; else if condition 3 ($k > 0$ and $n \leq 1$) is found to be the case, thus T with catalytic vertex t as internal node does not exist. The **for** loop of line 10 - 20 explores all edges belong to G . If reduction rule 1 is the case (bridge edge uv has both its two incident nodes u and v of degree 2), uv will be contracted, otherwise, reduction rule 2 is established, uv will be deleted. Line 13 and 16 keep the data required by undoing procedures in the lists. The next **for** loop of line 21 - 27 examines all vertices in G to see if reduction rule 3 (vertex v of degree 1 has internal node flag 0) is applied, and if v is not a catalytic vertex, if so, v can be a leaf node and will be deleted. Line 23 sets the internal node flag of v 's neighbor to 1 (marking an internal node). Line 24 stores a list of data required by undoing deletion.

4.3.2 Bounded Tree Search

If (G, k) is a yes-instance for k -leaf spanning tree, then at least one of $k+1$ vertices of G can be assumed as a catalytic vertex. Therefore the bounded search tree has at most $k+1$ "branches", which are binary search subtrees starting with $k+1$ catalytic vertices. The

Algorithm 7 *Kernelize*(G, k, t) – Encapsulates reduction rules and determines the existence.

```

1: if  $n > 4(k + 2)(k + 1)$  then
2:   return 1
3: end if
4: if  $k \leq 0$  and  $n \geq 1$  then
5:   return 1
6: end if
7: if  $k > 0$  and  $n \leq 1$  then
8:   return 3
9: end if
10: for all  $uv \in E$  do
11:   if both of  $u$  and  $v$  of degree 2 then
12:     if  $uv$  is a bridge then
13:       make a list of data required by uncontract  $uv$ 
14:       contract  $uv$ 
15:     else
16:       make a list of data required by undeleteEdge  $uv$ 
17:       delete  $uv$ 
18:     end if
19:   end if
20: end for
21: for all  $v \in V$  do
22:   if  $v \neq t$  and  $d(v) = 1$  and  $f(v) = 0$  then
23:      $f(v\text{'s neighbor}) = 1$ 
24:     make a list of data required by undelete  $v$ 
25:     delete  $v$ 
26:   end if
27: end for
28: if  $n > 4(k + 2)(k + 1)$  then
29:   return 1
30: end if
31: if  $k \leq 0$  and  $n \geq 1$  then
32:   return 1
33: end if
34: if  $k > 0$  and  $n \leq 1$  then
35:   return 3
36: end if
37: return 2

```

FBranch algorithm shown in Algorithm 8 constructs the binary search subtree in a depth-first manner, using an *extended branching rule*, which is based on the branching rule introduced in Section 3.2. It takes a graph G , the number of leaves k and catalytic vertex t as input values, and outputs either true or false. It begins with the invocation to *kernelize*, which checks whether three conditions match, reduces the problem size if possible, and returns an integer 1, 2, or 3 corresponding to three cases. Line 3 of **case 1** indicates G has a k -leaf spanning tree. Line 33 of **case 3** infers G has not such a k -leaf spanning tree with t as one of internal nodes. Line 4 - 32 of **case 2** cannot tell the existence of k -leaf spanning tree immediately.

In **case 2**, it establishes a combinatorial search subtree in a depth-first manner, based on the extended branching rule and labelled as follows. Label the root of the search subtree with the empty set $\emptyset$ and the graph G . Choose a neighbor of t , named v . If v is not a cut vertex, then it can be either leaf node or internal node, so the left and the right child of the root will be created corresponding to these two possibilities respectively. If v is a cut vertex, then it must be internal node, so only the right child will be created. The left child will be labelled with $\{v\}$ and $G - v$ (delete v). The right child will be labelled with $\emptyset$ and $G \setminus tv$ (contract tv and name the new vertex t). The set of vertices labelling a node represents a “possible” subset of leaves of a potential k -leaf spanning tree, and the graph labelling the node represents what remains to be explored in G . In general, for a node labelled with the set of vertices S and the subgraph H of G , choose a neighbor u of the current catalytic vertex t . If u is not a cut vertex, the two child nodes will be created with label, respectively, $S \cup \{u\}$ and $H - u$, and S and $H \setminus tu$. If u is a cut vertex, only the right child will be created with label of S and $H \setminus tu$. The search path goes in a depth-first manner. Starting from the root, *FBranch* expands a left child and keeps expanding the left children or the only right child until one of the three conditions is met. Then it backtracks its search path, for every left node passed, it expands its right sibling.

Algorithm 8 $FBranch(G, k, t)$ - One of the $k + 1$ branches of bounded search subtree

```

1: run kernelize algorithm and obtain an integer  $x$  with value 1, 2 or 3 as output
2: switch ( $x$ )
3: case 1: return True
4: case 2: select  $v \in t$ 's neighbors
5: if  $v$  is not a cut-vertex and  $f(v) = 0$  then
6:   make a list of data required by undelete  $v$ 
7:   delete  $v$ 
8:   if  $FBranch(G', k', t)$  then
9:     return true
10:  else
11:    undelete  $v$ 
12:    make a list of data required by uncontract  $tv$ 
13:    contract  $tv$ 
14:    if  $FBranch(G'', k, t)$  then
15:      return True
16:    else
17:      uncontract  $tv$ 
18:      unkernelization
19:      return False
20:    end if
21:  end if
22: else
23:   contract edge  $tv$ 
24:   if  $FBranch(G''', k, t)$  then
25:     return true
26:   else
27:     uncontract  $tv$ 
28:     unkernelization
29:     return False
30:   end if
31: end if
32: unkernelization
33: case 3: return False

```

Figure 4.2: DFS Path of Restarting Process

4.3.3 Restarting Technique

For each branch of the $k+1$ branches, *FBranch* takes $O(2^k)$. As k grows, the running time becomes impractical. We associate a timer with *FBranch*, so it will restart the search from another catalytic vertex if it times out. The timer starts from 30 seconds. The timer doubles its time setting for the next run until the program finds a solution, or until the running time is out of reasonable range. Figure 4.2 shows the DFS path using the restarting technique, where the black node is the catalytic vertex and the gray one is the node where a solution is discovered. A graph may have multiple k -leaf spanning trees, some of which are easier to be found than the others. The restarting technique has been found to be effective and efficient in practice. By applying restarting technique, the k -leaf spanning tree problem with a much larger k value can be resolved within a reasonable time.

4.4 Parallel FPT Algorithm

Two methods can be applied to parallelize the sequential algorithm. One is catalyst parallelization, where parallel operations start after the catalytic vertices have been chosen. Figure 4.3 illustrates the catalyst parallelization. The other approach uses in-branch parallelization, where for every catalytic vertex, the parallelism begins after $O(\log_2 P)$ sequential steps. Figure 4.4 illustrates the in-branch parallelization. As the latter one has a synchronization delay before it begins searching from the next catalytic vertex, we select

the first approach to implement.

4.4.1 Catalyst Parallel Algorithm

Assuming p processors, processor P_0 randomly chooses $p-1$ different catalytic vertices and assigns them to the processor $P_1, P_2, \dots, P_{p-1}$.

Every processor $P_i, 1 \leq i \leq p-1$, begins from its own catalytic vertex t and computes its unique search subtree in a depth-first fashion, using the reduction rules introduced in Section 4.3.1 and the extended branching rule in 4.3.2, until one of the three conditions in 4.3.1 is applied. If condition 1 ($n > 4(k+2)(k+1)$) or condition 2 ($k \leq 0$ and $n \geq 1$) is found to be the case, a k -leaf spanning tree exists. If condition 3 ($k > 0$ and $n \leq 1$) is satisfied, then the k -leaf spanning tree with catalyst t as internal node does not exist, so the processor in question will request another catalytic vertex from P_0 .

Once processor P_0 receives a request from a processor P_i , it randomly selects a new catalytic vertex and sends it to the requesting processor. If $k+1$ different vertices have already been chosen as catalytic vertices and G does not have a k -leaf spanning tree with those catalytic vertices as internal nodes, then as explained in Section 3.2, G does not have k -leaf spanning tree.

4.4.2 In-branch Parallel Algorithm

From a unique catalytic vertex, t , we sequentially expand the search path in breadth-first fashion, applying the extended branching rule, until there are exactly $p-1$ leaves $\gamma_1 \dots \gamma_p$ of the search tree.

Every processor, $P_i, 1 \leq i \leq p-1$, computes a unique path to γ_i . Let (G_i, S_i) , be subgraph and the leaf set associated with γ_i . Processor P_i , with input (G_i, S_i) , now searches

Figure 4.3: Catalyst Parallelization

Figure 4.4: In-branch Parallelization

the subtree rooted at γ_i in depth-first manner, using the reduction rules and the extended branching rule, until one of the three conditions are satisfied. If condition 1 ($n > 4(k + 2)(k + 1)$) or condition 2 ($k \leq 0$ and $n \geq 1$) is the case, then a k -leaf spanning tree exists and P_i notifies all other processors to halt; otherwise the G_i does not contain a k_i -leaf spanning tree with catalytic vertex t as internal node.

If none of the processors finds its spanning tree, the above process will be repeated from an unexplored catalytic vertex, t' . If $k + 1$ different vertices have been chosen as catalytic vertex and no k -leaf spanning tree with t' as internal node is found, then as explained in Section 3.2, G does not have k -leaf spanning tree. A synchronization delay is incurred as all processors have to complete their search in one “branch” before they together begin the next search from a new catalytic vertex.

Chapter 5

Experimental Results

This chapter first introduces two facilities, one is designed to generate random graphs, and the other is used to verify the solutions. It then explain the experimental setup and methodology, and the data sets applied in the experiments. The chapter ends with the performance results of the sequential and parallel experiment as well as our observation and analysis.

5.1 Random Graph Generator

Generating the test instances for NPC combinatorial problem is nontrivial. The Random Graph Generator, referred to as **p0**, generates a connected random graph with a known k -leaf spanning tree. k is close to the maximum of the number of leaves. The basic idea is to first create a tree and then add edges between pairs of tree nodes. The edge density of the graph decides how close k is to the maximum. The higher the density, the less close k is to the maximum.

The command to generate a random graph is: `p0 [-d P] [-e Q] [-f U] [-g V]`

The options in the command are:

- -d P: the maximum number of children an internal node of the spanning tree has. Default is 6.
- -e Q: the minimum number of children an internal node of the spanning tree has. Default is 2.
- -f U: the maximum degree of a leaf node in the “wrapping” graph is $U+1$. Default is 3.
- -g V: the minimum degree of a leaf node in the “wrapping” graph is $V+1$. Default is 1.

After we execute the above command line, **p0** prompts the input of the output file name, the value of k , and the edge number of the graph. The output of **p0** follows the format below:

```

Number of vertices

Number of edges

Number of leaves

Vertex ID Degree Adjacent vertices

...

```

Figure 5.1 shows a random graph with 16 vertices, 22 edges and a spanning tree of 10 leaves. This graph is generated by the default values.

The adjacency matrix is used as the data structure. There are two major steps to generate an instance. First, a random tree with a given number of leaves is created. Then, a certain number of edges are added to join pairs of vertices of the tree. The number of the edges

```

16
22
10
1 deg=3 8 14 15
2 deg=3 5 11 12
3 deg=5 4 6 9 11 12
4 deg=5 3 7 11 13 14
5 deg=4 2 10 12 16
6 deg=2 3 10
7 deg=1 4
8 deg=3 1 10 15
9 deg=1 3
10 deg=4 5 6 8 13
11 deg=3 2 3 4
12 deg=3 2 3 5
13 deg=2 4 10
14 deg=2 1 4
15 deg=2 1 8
16 deg=1 5

```

Figure 5.1: A Random Graph Generated by **P0**

to be added to the tree is equal to the given number of edges of the graph deleted by the number of tree edges.

5.2 k -Leaf Spanning Tree Tester

The sequential or parallel FPT algorithm writes a set of leaf nodes into an output file, if it finds a yes-instance. We designed a tool referred to as **p2** to check the existence of the spanning tree with this set of nodes as leaves. The test algorithm is shown in Algorithm 9.

This algorithm takes a graph G , a set S of vertices and the catalytic vertex t as input, and returns either true or false. The following describes how Algorithm 9 works. Line 3 paints the catalytic vertex t gray. The **for** loop of lines 4 - 7 discovers all t 's neighbor nodes, paints them gray and pushes them into the queue Q . The **while** loop of lines 8 - 16 iterates as long as Q is not empty. In the while loop, Line 9 determines u is the head element of Q and removes it from Q . If u is not in S , then it is an internal node and the algorithm discovers all its neighbor nodes by executing the **for** loop of lines 11 -14. The **for** loop of lines 17 - 19 examines the color of all nodes in G . If all of them are in gray, then we have constructed a spanning tree of G with the vertices in S as leaves.

Figure 5.2 illustrates the test process of the existence of a spanning tree on a sample graph G . Figure 5.2 (a) is the adjacency list representation of G , and Figure 5.2 (b) are the catalytic vertex t (9) and the set S of leaf nodes (6, 2, 3, 1, 4, 5). Figure 5.2 (c), (d), and (e) show the constructing procedure of the spanning tree, where vertices are shown shaded as they are discovered in the test procedure. The spanning tree is built gradually according to the neighborhood relationship as shown in Figure 5.2 (a). Tree edges are shown shaded as they are produced by the test procedure. The queue Q is checked for emptiness at the beginning of each iteration of while loop of lines 8 - 16.

Algorithm 9 *Test*

```

1: Input:  $(G, S, t)$ 
2: Output: (True or False)
3:  $t \leftarrow \text{GRAY}$ 
4: for every vertex  $v \in t$ 's adjacency list do
5:    $c[v] \leftarrow \text{GRAY}$ 
6:    $Q \leftarrow v$ 
7: end for
8: while  $Q \neq \emptyset$  do
9:    $u \leftarrow \text{DEQUEUE}(Q)$ 
10:  if  $u \notin S$  then
11:    for every vertex  $v \in u$ 's adjacency list do
12:       $c[v] \leftarrow \text{GRAY}$ 
13:       $Q \leftarrow v$ 
14:    end for
15:  end if
16: end while
17: for every vertex  $v \in G$  do
18:   check  $c[v]$ 
19: end for
20: if all vertices in GRAY then
21:   RETURN True
22: else
23:   RETURN False
24: end if

```

Figure 5.2: Test Procedure of the Existence of a Spanning Tree on a Graph G

5.3 Experimental Setup and Methodology

Our experimental platform is a 32 board, dual processor Linux cluster. In total, there are 64 Xeon 1.8 GHz chips (2 per board), 32 GB of distributed memory (1 GB per board), and 2.56 TB of distributed external memory (two 40 GB IDE disks per board). There is a 100 Mb interconnect between the processors.

We first implemented in C++ the sequential FPT algorithm described in Section 4.3. This sequential code is referred to as **p1**. Then, we implemented our parallel FPT approach described in Section 4.4, using C++ and MPI communication library, by adding the pertinent C++ and MPI code to **p1**. The parallel code is referred to as **p4**.

We executed **p1** on a single processor of our parallel machine and measured the sequential wall clock times, then executed **p4** on 32 processors of our parallel machine and measured the parallel wall clock times, and finally executed **p4** on 3, 5, 9 and 17 processors of our parallel machine and measured the relative speedup with respect to the parallel wall clock time. The parallel wall clock time is the wall clock time between the beginning of the first processor and the end of the last processor. Both sequential time and parallel time contain the time spent to read the input graph from a file and write the solution into an output file.

In a multi-user environment, simultaneous execution of user codes would result in very poor system utilization as multiple users compete for scarce resources. Moreover, analysis of test results would be nearly impossible since the running time characteristics would change dramatically over time. **p1** and **p4** are not directly run from the command line. Instead, the code is submitted through a batch submission infrastructure that guarantees fair, efficient use of resources. A queuing system known as OpenPBS (Portable Batch System) monitors all submission requests and takes care of the running applications and returning the results to users.

5.4 Data Sets

The test data for our experiments include the graphs generated by **p0**, the Internet topologies generated by Inet-3.0 [22] and triangular irregular networks (TIN). To also explore general graph classes, we also tested the sequential FPT algorithm on the random graphs and the protein sequence graphs used for the vertex cover problem. Those graphs are in various format, so we implemented two converters in PERL to convert those formats to the format accepted by our sequential and parallel program. Table 5.1 lists the graphs used in our experiments.

Graph	$ V $	$ E $	k	Type
G0	705	2000	545	Generated by p0
G2	700	1500	507	Generated by p0
INET.500	500	783	416	Created by Inet-3.0
INET.600	600	920	488	Created by Inet-3.0
INET.700	700	1081	573	Created by Inet-3.0
ATIN	552	1563	299	TIN
JAPAN.1500	1500	4353	833	TIN
G.315	73	314	60	Random Graph
G.346	88	395	71	Random Graph
G.410	120	575	97	Random Graph
RG.20	50	225	42	Random Graph
RG.21	100	2370	98	Random Graph
WW	425	40182	417	Protein Sequence
Somatostatin	559	33652	549	Protein Sequence

Table 5.1: Data Sets

5.5 Performance Results: Sequential Experiments

For every the above mentioned graph, we have run **p1** five times on a single processor of our parallel machine and measured the sequential wall clock times. We set the timer for the restart to 30 seconds in order to finish the experiment in a reasonable time. Table 5.2 lists the sequential experimental results. Five runs conducted for each graph result time in seconds from *time.1* to *time.5* respectively. In the same graph category, the more vertices and edges a graph has, the higher k value can be calculated in an acceptable time. Graph RG.21, WW and Somatostation are high edge density, their k -leaf spanning trees can be calculated in a very short time even instantly. Figure 5.3 and 5.4 compare the five sequential experimental results for each graphs. The five consecutive vertical bars are the five experimental results from *time.1* to *time.5* respectively. We have observed that the five running times vary a lot for most graphs, so the catalyst parallel approach will be effective and might yield superlinear speedup.

5.6 Performance Results: Parallel Experiments

Taking G*, INET.* and TINs as input graphs (as they have more than 500 vertices and their sequential running times are long enough to be improved), we have run **p4** five times for every graphs on 32 processors of our parallel machine and measures the parallel wall clock times. Table 5.3 lists the experimental results. Five experiments conducted for each graph result time in seconds from *time.1* to *time.5* respectively. We have noticed that our catalyst parallel approach is able to solve those problem instances of size $|V| \geq 500$ in at most 6 minutes. Figure 5.5 compares the five parallel experimental results on 32 processors for every graphs. Compare Figure 5.5 and Figure 5.3, the vertical bars shown for the same graph may have the different trend. For instance, in Figure 5.3, G0's fourth

Figure 5.3: Sequential Wall Clock Time in Seconds for G*, INET.* and TINs

Figure 5.4: Sequential Wall Clock Time in Seconds for G.*, RG.* and Protein Sequences

Graph	$ V $	$ E $	k	<i>time.1</i>	<i>time.2</i>	<i>time.3</i>	<i>time.4</i>	<i>time.5</i>
G0	705	2000	545	3812	5402	7020	930	9870
G2	700	1500	507	1560	4111	2353	2498	2762
INET.500	500	783	416	1711	4471	3931	1501	4411
INET.600	600	920	488	3480	3210	6960	5850	3420
INET.700	700	1081	573	1170	2340	3990	4050	1950
ATIN	552	1563	299	1627	8190	2370	907	330
JAPAN	1500	4353	833	2602	3764	1844	2711	2184
G.315	73	314	60	1472	1830	1592	152	62
G.346	88	395	71	990	627	1080	151	690
G.410	120	575	97	493	1122	72	1752	1182
RG.20	50	225	42	180	371	12	691	31
RG.21	100	2370	98	35	35	0	120	6
WW	425	40182	417	627	20	168	168	200
Somatostation	559	33652	549	935	170	166	196	330

Table 5.2: Measured Sequential Wall Clock Times in Seconds

experiment takes the least time and fifth one takes the most time, but in Figure 5.5, G0's second experiment takes the least time and fourth one takes the most time. Because if the sequential running time is rather small, its counterpart parallel time has a pricy parallel overhead.

We also executed **p4** on 3, 5, 9 or 17 processors of our parallel machine and measured their parallel wall clock times. We also take G*, INET.* and TINs as input graphs. Table 5.4 lists the parallel experimental results and the corresponding sequential experimental results. The parallel experiment applies the same "random seed" as its corresponding sequential experiment, so we can measure the relative speedup (sequential time / parallel time) accordingly. The relative speedups are given in Table 5.5.

Figure 5.5: Parallel Wall Clock Time in Seconds on 32 Processors for G*, INET.* and TINs

Graph	$ V $	$ E $	k	$time.1$	$time.2$	$time.3$	$time.4$	$time.5$
G0	705	2000	545	301	2	121	481	391
G2	700	1500	507	18	197	63	77	257
INET.500	500	783	416	273	273	33	213	333
INET.600	600	920	488	1	90	30	60	90
INET.700	700	1081	573	150	0	0	150	1
ATIN	552	1563	299	14	121	121	194	44
JAPAN	1500	4353	833	538	352	538	291	80

Table 5.3: Measured Parallel Wall Clock Times in Seconds on 32 Processors

Graph	Number of Processors	<i>time.1</i>	<i>time.2</i>	<i>time.3</i>	<i>time.4</i>	<i>time.5</i>
G0	1 <i>p</i>	3812	5402	7020	930	9870
$ V = 705$	3 <i>p</i>	1890	2702	3510	450	4920
$ E = 2000$	5 <i>p</i>	930	1350	1740	210	2460
$k = 545$	9 <i>p</i>	451	663	870	91	1230
	17 <i>p</i>	212	333	422	32	601
G2	1 <i>p</i>	1560	4111	2353	2498	2762
$ V = 700$	3 <i>p</i>	780	2040	1178	1238	1383
$ E = 1500$	5 <i>p</i>	390	1020	578	608	691
$k = 507$	9 <i>p</i>	181	510	283	321	332
	17 <i>p</i>	91	241	141	163	155
INET.500	1 <i>p</i>	1711	4471	3931	1501	4411
$ V = 500$	3 <i>p</i>	844	2221	1951	751	2191
$ E = 783$	5 <i>p</i>	425	1111	961	361	1081
$k = 416$	9 <i>p</i>	211	542	482	181	541
	17 <i>p</i>	97	274	244	93	274
INET.600	1 <i>p</i>	3480	3210	6960	5850	3420
$ V = 600$	3 <i>p</i>	1740	1590	3480	2910	1710
$ E = 920$	5 <i>p</i>	870	780	1740	1440	840
$k = 488$	9 <i>p</i>	420	390	870	720	420
	17 <i>p</i>	210	180	420	360	210
INET.700	1 <i>p</i>	1170	2340	3990	4050	1950
$ V = 700$	3 <i>p</i>	570	1170	1980	2010	960
$ E = 1081$	5 <i>p</i>	270	570	990	990	480
$k = 573$	9 <i>p</i>	120	270	480	480	240
	17 <i>p</i>	60	120	240	240	210
ATIN	1 <i>p</i>	1627	8190	2370	907	841
$ V = 552$	3 <i>p</i>	817	4080	1170	460	421
$ E = 1563$	5 <i>p</i>	397	2040	570	216	211
$k = 299$	9 <i>p</i>	186	1020	270	96	91
	17 <i>p</i>	104	510	121	43	32
JAPAN	1 <i>p</i>	2602	3764	1844	2711	2184
$ V = 1500$	3 <i>p</i>	1209	1780	879	1299	1059
$ E = 4353$	5 <i>p</i>	609	876	446	641	516
$k = 833$	9 <i>p</i>	306	430	216	325	262
	17 <i>p</i>	156	231	107	171	135

Table 5.4: Measured Parallel Wall Clock Times in Seconds on 3, 5, 9 and 17 Processors

Graph	Number of Processors	1st run speedup	2nd run speedup	3rd run speedup	4th run speedup	5th run speedup	Average speedup
G0	3 p	2.0	2.0	2.0	2.1	2.0	2.0
$ V = 705$	5 p	4.1	4.0	4.0	4.4	4.0	4.1
$ E = 2000$	9 p	8.5	8.1	8.1	10.2	8.0	8.6
$k = 545$	17 p	18.0	16.2	16.6	29.1	16.4	19.3
G2	3 p	2.0	2.0	2.0	2.0	2.0	2.0
$ V = 700$	5 p	4.0	4.0	4.1	4.1	4.0	4.0
$ E = 1500$	9 p	8.6	8.1	8.3	7.8	8.3	8.2
$k = 507$	17 p	17.1	17.1	16.7	15.3	17.8	16.8
INET.500	3 p	2.0	2.0	2.0	2.0	2.0	2.0
$ V = 500$	5 p	4.0	4.0	4.1	4.2	4.1	4.1
$ E = 783$	9 p	8.1	8.2	8.2	8.3	8.2	8.2
$k = 416$	17 p	17.6	16.3	16.1	16.1	16.1	16.5
INET.600	3 p	2.0	2.0	2.0	2.0	2.0	2.0
$ V = 600$	5 p	4.0	4.1	4.0	4.1	4.1	4.0
$ E = 920$	9 p	8.3	8.2	8.0	8.1	8.1	8.2
$k = 488$	17 p	16.6	17.8	16.6	16.3	16.3	16.7
INET.700	3 p	2.1	2.0	2.0	2.0	2.0	2.0
$ V = 700$	5 p	4.3	4.1	4.0	4.1	4.1	4.1
$ E = 1081$	9 p	9.8	8.7	8.3	8.4	8.1	8.7
$k = 573$	17 p	19.5	19.5	16.6	16.9	16.3	17.8
ATIN	3 p	2.0	2.0	2.0	2.0	2.0	2.0
$ V = 552$	5 p	4.1	4.0	4.2	4.2	4.0	4.1
$ E = 1563$	9 p	8.7	8.0	8.8	9.4	9.2	8.8
$k = 299$	17 p	15.6	16.1	19.6	21.1	26.3	19.7
JAPAN	3 p	2.2	2.1	2.1	2.1	2.1	2.1
$ V = 1500$	5 p	4.3	4.3	4.1	4.2	4.2	4.2
$ E = 4353$	9 p	8.5	8.8	8.5	8.3	8.3	8.5
$k = 833$	17 p	16.7	16.3	17.2	16.3	16.2	16.5

Table 5.5: Measured Relative Speedups on 3, 5, 9 and 17 Processors

Figure 5.6: Parallel Wall Clock Times and Speedups for G0 on 3, 5, 9 and 17 Processors

Figure 5.7: Parallel Wall Clock Times and Speedups for G2 on 3, 5, 9 and 17 Processors

Figure 5.8: Parallel Wall Clock Times and Speedups for INET.500 on 3, 5, 9, 17 Processors

Figure 5.9: Parallel Wall Clock Times and Speedups for INET.600 on 3, 5, 9, 17 Processors

Figure 5.10: Parallel Wall Clock Times and Speedups for INET.700 on 3, 5, 9, 17 Processors

Figure 5.11: Parallel Wall Clock Times and Speedups for ATIN on 3, 5, 9 and 17 Processors

Figure 5.12: Parallel Wall Clock Times and Speedups for JAPAN on 3, 5, 9, 17 Processors

For the parallel execution on 3, 5, 9 and 17 processors, we used the following graph instances for testing: G0, G2, INET.500, INET.600, INET.700, ATIN and JAPAN. Figure 5.6 to Figure 5.12 are the plots for each graph respectively. The plots are done according to the data listed in Table 5.4 and 5.5. In each figure, a pair of plots are shown for every test environment (namely 3, 5, 9 or 17 processors), one plot represents the parallel wall clock time in seconds, the other plot shows the relative speedup. Five tests are performed in each test environment for each graph. We make the following observations regarding those figures:

- For every graph, the vertical bars shown for the parallel times on 3, 5, 9 and 17 processors display a similar trend, but it is not the case for the relative speedups on 3, 5, 9 and 17 processors.
- The vertical bars of the parallel times fluctuate significantly. For the relative speedups, the vertical bars for 3, 5, and 9 processors do not show much fluctuation in most cases but for 17 processors, significant fluctuation occurs.
- The run taking the least parallel time may obtain the biggest speedup among five runs, as its counterpart sequential time is not the least, which we have observed while we compare the vertical bar figures of the sequential wall clock times with the ones of the parallel wall clock times on 32 processors.

Figures 5.14, 5.15 and 5.16 show the average speedups based on the data listed in Table 5.5. The straight line labeled “linear” represents linear speedup (speedup = the number of processors p). The linear speedup is known as the best possible case. From those figures, we have observed either the asymptotic linear speedup or even superlinear speedup (speedup greater than p). We also noticed that as the number of processors increases, so does the speedup.

Figure 5.13: Superlinear Speedup Analysis

Let x be the number of catalytic vertices explored, y be the time spent on a particular catalytic vertex which finds the solution, p be the number of the processors (processor P_1 to $P_p - 1$ are responsible for parallel calculation) and t be the restarting time. The sequential time is $t(x - 1) + y$ and the parallel time is $t(\lceil \frac{x}{p-1} \rceil - 1) + y$, therefore the speedup is $\frac{t(x-1)+y}{t(\lceil \frac{x}{p-1} \rceil - 1) + y}$. When the remainder of x divided by $p-1$ is close to $p-1$, such that $\frac{t(x-1)+y}{t(\lceil \frac{x}{p-1} \rceil - 1) + y} > p$, the superlinear speedup occurs. The more the number of processors, the more the chance this happens. Examples of this are given in Figure 5.13, where the circles represent catalytic vertices and the gray ones of them are the ones with solutions. In Figure 5.13 (a), x is 11, in Figure 5.13 (b), x is 5, and for both cases shown in (a) and (b), p is 5, t is 30 seconds and y is 20 seconds, then only the case shown in Figure 5.13 (a) has superlinear speedup.

The superlinear speedup indicates that the sequential FPT approach can be improved by the sequential simulation of the parallel algorithm. In the parallel simulation on one processor, p catalytic vertices are chosen for p virtual processors. The virtual processors start searching from those catalytic vertices. Each virtual processor executes the same number of search steps in their turn. This process runs repetitively until a solution is found. If the time for each searching steps are small enough to be ignored, the virtual parallel time

is equal to the real parallel time on p processors multiplying p . Assuming speedup greater than p then the virtual parallel time is less than the sequential time.

Figure 5.14: Average Relative Speedup for G*

Figure 5.15: Average Relative Speedup for INET.*

Figure 5.16: Average Relative Speedup for TINs

Bibliography

- [1] H. L. Bodlaender. On linear time minor tests with depth-first search. *J. Algorithms*, 14:1–23, 1993.
- [2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001.
- [3] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, November 1974.
- [4] R. G. Downey and M. R. Fellows. *Parameterized Complexity*. Springer, 1999.
- [5] R. G. Downey, M. R. Fellows, and U. Stege. Parameterized complexity: A framework for systematically confronting computational intractability. In *Contemporary Trends in Discrete Mathematics: From DIMACS and DIMATIA to the Future*, F. Roberts, J. Kratochvil, J. Nešetřil (Eds.), *AMS-DIMACS Proceedings Series*, 49(AMS):49–99, 1999.
- [6] M. R. Fellows and M. A. Langston. On well-partial-order theory and its application to combinatorial problems of VLSI design. *SIAM Journal on Discrete Mathematics*, 5(1):117–126, February 1992.

- [7] M. R. Fellows, C. McCartin, F. A. Rosamond, and U. Stege. Coordinatized kernels and catalytic reductions: An improved FPT algorithm for max leaf spanning tree and other problems. *Lecture Notes in Computer Science*, 1974:240–251, 2000.
- [8] G. Galbiati, F. Maffioli, and A. Morzenti. A short note on the approximability of the maximum leaves spanning tree problem. *Information Processing Letters*, 52:45–49, 1994.
- [9] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to NP-Completeness*. W.H. Freeman and Company, San Francisco, California, 1979.
- [10] J. R. Griggs, D. J. Kleitman, and A. Shastri. Spanning trees with many leaves in cubic graphs. *Journal of Graph Theory*, 13:669–695, 1989.
- [11] D. Haussler and E. Welzl. Epsilon-nets and simplex range queries. *Discrete and Computational Geometry*, 2:127–151, 1987.
- [12] D. J. Kleitman and D. B. West. Spanning trees with many leaves. *SIAM Journal on Discrete Mathematics*, 4(1):99–106, February 1991.
- [13] E. Kranakis, D. Krizanc, B. Ruf, J. Urrutia, and G. Woeginger. The VC-dimension of set systems defined by graphs. *DAMATH: Discrete Applied Mathematics and Combinatorial Operations Research and Computer Science*, 77, 1997.
- [14] H.-I. Lu and R. Ravi. The power of local optimizations: Approximation algorithms for maximum-leaf spanning tree. Technical Report CS-96-05, Brown University - Department of Computer Science, January 1996. <ftp://ftp.cs.brown.edu/pub/techreports/96/cs96-05.ps.Z>.
- [15] H.-I. Lu and R. Ravi. Approximating maximum leaf spanning trees in almost linear time. *Journal of Algorithms*, 29(1):132–141, October 1998.

- [16] B. Meyer. Principles of package design. *Communications of the ACM*, 25(7):419–428, 1982.
- [17] N. Robertson and P. D. Seymour. Graph minors XIII. The disjoint paths problem.
- [18] N. Robertson and P. D. Seymour. Graph minors XVI, wagner’s conjecture.
- [19] N. Robertson and P. D. Seymour. Graph minors v: excluding a planar graph. *J. Combinatorial Theory, Series B*, 41:92–114, 1986.
- [20] J. A. Storer. Constructing full spanning trees for cubic graphs. *Information Processing Letters*, 13(1):8–11, October 1981.
- [21] V. N. Vapnik and A. Y. Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probab. and its Applications*, 16(2):264–280, 1971.
- [22] J. Winick and S. Jamin. Inet-3.0: Internet topology generator. Technical report, University of Michigan. <http://topology.eecs.umich.edu/inet/>.