RICE UNIVERSITY

Linear-Time Algorithms for Graphs with Bound Branchwidth

by

William A. Christian, Jr.

Department of Computational and Applied Mathematics

Rice University

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE

Doctor of Philosophy

APPROVED, THESIS COMMITTEE:

[Signatures and names]

Nathaniel Dean, Co-Chair
Associate Professor of Computational and Applied Mathematics

William J. Cook, Co-Chair
Russ and Sammie Chandler Professor of Industrial and Systems Engineering
Georgia Institute of Technology

Robert Bixby
Emeritus Professor of Computational and Applied Mathematics

Alan Cox
Associate Professor of Computer Science

HOUSTON, TEXAS

NOVEMBER 2002
ABSTRACT

Linear-Time Algorithms for Graphs with Bounded Branchwidth

by

William A. Christian, Jr.
Department of Computational and Applied Mathematics
Rice University

We present an algorithmic framework (including a single data structure) that is extended into linear-time algorithms to solve several NP-complete graph problems (i.e., INDEPENDENT SET, MAXIMUM CUT, GRAPH COLORING, HAMILTONIAN CYCLE, and DISJOINT PATHS). The linearity is achieved assuming the provision of a branch decomposition of the instance graph. We then modify the framework to create a multithreaded framework that uses the existing problem-specific extensions without any revision. Computational results for the serial and parallel algorithms are provided. In addition, we present a graphical package called JPAD that can display a graph and branch decomposition, show their relationship to each other, and be extended to run and display the progress and results of algorithms on graphs or on branch decompositions.
Acknowledgments

“The fear of the Lord is the beginning of wisdom; and the knowledge of the holy is understanding.” [Proverbs 9:10] God, I thank you constantly. Here I acknowledge that you created me, you strengthen me, and it is you who provided all the wonderful people and opportunities that came together to make me who I am. And it is you who will make me who I am to become.

Anyone who knows me knows the importance of my family—nuclear and extended. Thank you to my wife, who stands by me in everything that I do and who inspires me to be more. Kenya Christian, I love you and thank you for marrying me. Rachel, Jessica, and Kimberly, in the years to come, I pray that you will experience the joy that you give me when I look into your eyes. And thank you to Myong and William Christian—my mother and father. I cannot think of a better set of parents. Obviously, I could not be here without you. Thank you for being my Mommy and Daddy.

It is customary to thank one’s committee as a matter of respect; however, I cannot thank my committee enough for their standing by me along the way to the completion of this dissertation. The road has not been smooth and I appreciate their standing by me for every twist and turn despite all of our ever-changing circumstances.

Of course, I cannot name everyone so I will not try, but I do want to acknowledge some people who believed in and supported me when others might not have. Thank you to Mrs. Hemler, Mrs. Bonnell, Dr. Treisman, Dr. Hrabowski, Mrs. Baker, Mr. and Mrs. Meyerhoff, Mrs. Daria Lawrence, Ms. Theresa Chatman, and Dr. Tapia for not giving up on me. And thank you to my classmates at Rice and elsewhere, especially Dr. Illya V. Hicks without whom I literally could not have completed this work.
Nobody completes a doctoral degree by himself. There are countless people who contribute in large ways that are obviously helpful and in small ways that seem insignificant at the time. Thank you.
# Contents

Abstract .............................................................. ii
Acknowledgments ..................................................... iii

1 Introduction .......................................................... 1

2 Background ................................................................ 4
    2.1 Graphs ............................................................... 4
    2.2 Branch and Tree Decompositions ................................. 7
    2.3 Logical Structures and Graph Properties ...................... 12
    2.4 Monadic Second-Order Logic .................................. 15
    2.5 Decidability of MS Properties ................................. 18

3 Algorithms Using Branch-Decompositions ....................... 21
    3.1 Decomposition Trees ............................................ 23
    3.2 Algorithmic Framework ........................................ 28
    3.3 Solution Storage ................................................ 30
    3.4 Subroutines ....................................................... 35
    3.5 Linearity of the Framework .................................... 36
    3.6 Algorithms Preview ............................................. 36

4 Vertex Bipartitioning Problems .................................... 38
    4.1 Join Operation .................................................... 38
    4.2 Maximum Independent Set ..................................... 39
    4.3 Maximum Cut .................................................... 43
4.4 Graph Bisection ............................................. 46

5 Graph Coloring: Minimum $k$ Vertex $k$-Partitioning 53
  5.1 Fixed Partitioning ........................................ 54
  5.2 Isomorphic Partitioning ................................... 55
  5.3 Algorithmic Complexity and Computational Results ....... 62

6 Path Partitioning ............................................. 64
  6.1 Solution Storage .......................................... 65
  6.2 Hamiltonian Cycle .......................................... 67
  6.3 The Traveling Salesman Problem ......................... 72
  6.4 Disjoint Paths ............................................. 73

7 Parallel Algorithms ........................................... 77
  7.1 ThreadLock Semaphore .................................... 78
  7.2 Thread Pooling ............................................ 85
  7.3 Possible Improvements .................................... 91

8 JPAD ............................................................. 93
  8.1 File Options ............................................... 94
  8.2 Views ....................................................... 96
  8.3 Features .................................................... 97
  8.4 Algorithms ................................................ 97
  8.5 Branch Decomposition Based Algorithms .................. 99
  8.6 Future Additions .......................................... 100

9 Conclusions .................................................. 102
9.1 Algorithms .................................................... 102
9.2 Parallelization ................................................ 104
9.3 JPAD .............................................................. 104

Bibliography ....................................................... 106
Chapter 1

Introduction

This thesis presents serial and parallel linear-time algorithms to solve several NP-complete problems when given a branch decomposition of the instance graph. We also present a package called JPAD as a tool for studying branch decomposition based algorithms and the relationships between a graph and its branch decompositions.

In recent years, decomposing graphs has received much attention in the study of NP-complete problems. The areas of branch and tree decompositions, introduced by Robertson and Seymour [19, 21], have received particular attention. The reason for our interest is that large classes of problems—namely, those that can be stated in monadic second-order logic—can be solved in polynomial or even linear time when restricted to graphs of bounded branchwidth—a result due in part to Courcelle [11]. Despite some significant differences, the algorithms for such problems have several commonalities and their core procedures are the same.

Ideally, one could develop a general algorithm for solving graph problems using branch decompositions. Advances in this direction include the algorithm for general vertex partitioning using partial $k$-trees by Telle and Proskurowski [28] and the method of translating a problem description into an algorithm by Borie, Parker, and Tovey [9]. These results are limited, however, in that the former algorithm does not address edge separations and the latter is theoretical and not practical.

While we do not claim to have solved all the problems inherent to implementing such algorithms, we present a framework that will allow us to solve general classes of graph problems using branch decompositions. A general solver for graph problems using branch decompositions is valuable because such a framework can serve as a
foundation for building algorithms that solve specific problems. Thus, we point out in advance that the algorithms we present do not take advantage explicitly of the structure of solutions to any specific problem; therefore, it is both possible and probable that these algorithms can be used with improved performance through minor changes for specialization.

This thesis presents serial and parallel algorithms to solve the following problems: Maximum Independent Set, Maximum Cut, Graph Bisection, Graph Coloring, Traveling Salesman Problem, and Disjoint Paths. We choose these six problems because they are classic problems in combinatorial optimization and each algorithm encapsulates significant features not found in the others.

This thesis proceeds as follows.

In Chapter 2 we define graphs and some related concepts as abstract data types and as logical structures using monadic second-order logic. In addition, we present definitions of branch decompositions and tree decompositions and explain how they are related to each other. These concepts set the foundation to examine Courcelle's Theorem which proves that linear-time algorithms exist for a large class of graph problems if a branch or tree decomposition is known for the instance graph.

With Courcelle's Theorem presented, we begin in Chapter 3 to study the algorithm framework for the algorithms that we present in this thesis. We look closer at some properties of branch decompositions that make them more suitable than tree decompositions for dynamic programming. In this work, we use a single data structure for storing vertex and edge partitioning solutions; the interpretation of the values used for vertex partitioning is presented in this chapter and that for edge partitioning in Chapter 6.

Using the framework and data structures explained in Chapter 3, we present in Chapters 4, 5, and 6 the algorithms to solve vertex bipartitioning problems (Maximum
INDEPENDENT SET, MAXIMUM CUT, and GRAPH BISECTION), a minimum \( k \) vertex \( k \)-partitioning problem (GRAPH COLORING), and path partitioning problems (HAMILTONIAN CYCLE and DISJOINT PATHS), respectively. In all of these cases we present the algorithms, study the computational complexity and give computational results.

Using the same algorithmic framework described in Chapter 3 and used in Chapters 4 through 6, we create an algorithm that takes advantage of parallel processors. Two parallel algorithms are presented in Chapter 7. The first algorithm creates threads as they are needed; whereas, the second uses a thread pool to minimize the need to create new threads. Of course, the second algorithm is both theoretically and empirically more efficient. Computational results are provided for both.

Prior to the algorithm development in this thesis, a package we call JPAD: a Java-based Program for Analyzing and Drawing Graphs was developed and implemented. In addition to being useful for viewing simple graphs, JPAD is unique in its ability to display branch decompositions and graphs and to demonstrate the relationships between the two. The primary features and an overview of how to extend JPAD are presented in Chapter 8. Of particular interest is the fact that traditional and branch decomposition based algorithms can be incorporated into the program using some provided wrapper classes.

In the last chapter—Chapter 9—we present a concluding summary of the work of this thesis. We also suggest possible directions for further research.
Chapter 2

Background

This chapter contains definitions and fundamental results. The term "graph" is used at different times in this thesis in reference to either abstract data types or logical structures; the proper interpretation of the term graph can be determined from context. We start with terminology relating to graphs as an abstract data type followed by definitions of branch and tree decompositions. Sections 2.3 and 2.4 present graphs as logical structures and some logical languages for describing graph properties. Finally, in Section 2.5, we show Courcelle's result which uses the logical representation of graphs in its proof. Courcelle's theorem motivates this work.

2.1 Graphs

This section, included for completeness, is a summary of some of the basic definitions of graph theory. Figure 2.1 is a visual representation of a graph with eight vertices and twelve edges. As an abstract data type, a graph \( G \) is a set \( V \) called vertices and a set \( E \) called edges. When it is necessary to avoid ambiguity, \( V \) and \( E \) are often denoted \( V(G) \) and \( E(G) \). Each edge is mapped to or associated with a subset of the set of vertices having two (not necessarily distinct) elements called endpoints. An edge is said to be incident to its endpoints and the endpoints themselves are adjacent to each other. An edge that has both endpoints as the same vertex is a loop. Two distinct edges that are mapped to the same set of vertices are called parallel edges. A simple graph has no parallel edges and no loops. In this thesis, unless stated otherwise, it is assumed that input graphs are simple (i.e., \( E(G) \subseteq \{\{u, v\}|u, v \in V(G)\} \)).
Figure 2.1 A visualization of a graph $G = (V, E)$ with eight vertices and twelve edges. $V = \{0, 1, 2, 3, 4, 5, 6, 7\}$ and

$E = \{\{0, 1\}, \{0, 4\}, \{0, 3\}, \{3, 2\}, \{2, 1\}, \{1, 5\}, \{5, 6\}, \{4, 5\}, \{6, 7\}, \{6, 2\}, \{7, 3\}, \{7, 4\}\}.$

For a particular vertex $v$, we denote the set of all (non-loop) edges incident to $v$ as $\delta(v)$. Then, the degree of $v$ is $|\delta(v)|$. If $X$ is a subset of vertices, then $\delta(X)$ denotes the set of edges with exactly one endpoint in $X$ and $\delta(X)$ is called a cut set.

If $G$ is a directed graph, its edges are ordered pairs or directed edges that we write as $e = (u, v)$. Then, we say $e$ is an edge from its tail, $u$, to its head, $v$. The out-edges, denoted $\delta^+(v)$, of a vertex $v$ are those edges whose tails are $v$; the in-edges, denoted $\delta^-(v)$, are those edges whose heads are $v$. Similarly, if $X$ is a subset of vertices then $\delta^+(X)$ is the set of edges whose tails are in $X$ and heads are in $V \setminus X$ and $\delta^-(X)$ is the set of edges from vertices outside of $X$ to vertices in $X$. That is, $\delta^+(X) = \{(u, v) \in E(G) | u \in X, v \in V \setminus X\}$ and $\delta^-(X) = \{(u, v) \in E(G) | u \in V \setminus X, v \in X\}$.

A subgraph $H$ of $G$ is a graph with the set of vertices $V(H) \subseteq V(G)$ and edges $E(H) \subseteq E(G)$. If $V(H)$ is a proper subset of $V(G)$ and $E(H) = \{\{u, v\} \in E(G) | u, v \in V(H)\}$, then we say that $H$ is the subgraph of $G$ induced by $V(H)$ and we may write $H = G[V(H)]$. Likewise, if $E(H)$ is a proper subset of $E(G)$ and
$V(H) = \{ v | \{u, v\} \in E(H) \}$, then we say that $H$ is the subgraph of $G$ induced by $E(H)$ and again we may write $H = G[E(H)]$.

A trivial example of a simple graph or subgraph is a path. A path between the vertices $v_0$ and $v_n$ in a graph $G$ is a subgraph of $G$ whose vertices and edges can be ordered as an alternating sequence $p = v_0, e_0, v_1, e_1, v_2, \ldots, e_{n-1}, v_n$ of distinct vertices $v_i \in V(G)$ for $i = 0, \ldots, n$ and edges $e_i = \{v_i, v_{i+1}\} \in E(G)$ for $i = 0, \ldots, n - 1$. If $p$ is a directed path, $e_i = (v_i, v_{i+1})$ for $i = 0, \ldots, n - 1$. A cycle is a path where we require all vertices to be distinct except $v_n = v_0$. Note that if $G$ is a simple graph, then it is not necessary to explicitly state the edges and we can refer to the path $p = v_0, v_1, v_2, \ldots, v_n$ or the cycle $c = v_0, v_1, \ldots, v_n, v_0$.

Two vertices in a graph $G$ are connected if a path exists between them. The entire graph is connected if every pair of vertices is connected. A set of vertices $X \subset V$ is a separator of a connected graph $G$ if $G \setminus X \equiv G[V(G) \setminus X]$ is not connected. If $X = \{v\}$ is a separator, $v$ is called a cut vertex.

A tree is a simple, connected graph with no cycles. If the graph is not necessarily connected, it is called a forest. The leaves of a tree $T$, which we shall denote $\lambda(T)$, are those vertices of degree one. The interior vertices of $T$ are those vertices of degree greater than one; every interior vertex is a cut vertex. A rooted tree is a tree $T$ with directed edges and a distinguished vertex $r$ called the root such that there exists a directed path from $r$ to every other vertex in $V(T)$. The children of a vertex $v$ in a rooted tree $T$ are the vertices $w_i \in V(T)$ such that there exist edges $e_i = (v, w_i) \in E(T)$. The descendants of a vertex $v$ are the vertices to which there exist directed paths from $v$. The descendants of $v$ together with $v$ induce the subtree of $T$ that we shall call the subtree of $T$ rooted at $v$ and more often denote as $T_v$. We shall refer to the subtree induced by the vertices other than $v$ and its descendants as $\overline{T_p}$ where $p$ is the parent of $v$. 
2.2 Branch and Tree Decompositions

Johnson [16] presents a survey of various classes of graphs and some results about the complexity of some graph decision problems over these classes. Many graph problems in NP are solvable in polynomial time when the instance includes a graph known to be a tree or forest. Some of the easiest cases, which all become trivial, are: Minimum Spanning Tree, Hamiltonian Cycle, and Graph Coloring. The class of graphs equivalent to trees is too restrictive for most practical problem instances; however, for classes of graphs that closely resemble trees (e.g., “almost trees” with parameter $k$, partial $k$-trees, degree-$k$ graphs), many hard problems still are solvable in polynomial time with respect to the size of the graph.

It stands to reason that if a general graph can be rewritten to resemble a tree, then the computational complexity of certain problems can be reduced. Robertson and Seymour introduced branch decompositions [21] and tree decompositions [20] in their series of papers proving Wagner’s Conjecture. We interpret the corresponding measures—branchwidth and treewidth—as measures of a graph’s relation to trees. Courcelle [12] and Arnborg, Lagergren, and Seese [3] have independently proven results characterizing various classes of graph problems that can be solved in linear time provided the input includes a decomposition. Of the two decompositions, we shall focus mainly on branch decompositions and, after presenting their definitions, we explain briefly why we choose to not use tree decompositions.

Let $G$ be a graph and let $T$ be a tree with $|E(G)|$ leaves and degree three at every interior vertex. If $\tau$ is a bijection from the leaves of $T$ to $E(G)$, then $(T, \tau)$ is a branch decomposition of $G$. Removing an edge, say $e$, of $T$ partitions the tree into what we refer to as the left and right subtrees. Consequently, $G$ is divided into two subgraphs $A_e$ and $B_e$ induced by the edges on the leaves of the left and
Figure 2.2 An optimal branch decomposition of Figure 2.1, which has branchwidth 4. The leaves of the tree are labeled with \( \tau(t) \) and some edges are labeled with their middle sets.

right trees, respectively. The set \( V(A_e) \cap V(B_e) \) is called the middle set of \( e \) and is denoted \( \text{mid}(e) \). Figure 2.2 depicts a branch decomposition of the graph illustrated in Figure 2.1. A graph can have several different branch decompositions. We say that the width of a particular branch decomposition is the largest cardinality (or order) of all of its middle sets. The branchwidth of \( G \), denoted \( \beta(G) \), is the minimum width over all branch decompositions of \( G \). A branch decomposition is said to be optimal if its width is equal to the branchwidth of \( G \). The width of the branch decomposition in Figure 2.2 is four and it is optimal.

A notion related to branch decomposition is tree decomposition. Let \( G \) be a graph and let \( T \) be a tree. The pair \((T, \mathcal{X})\) where \( \mathcal{X} = \{X_t | t \in V(T)\} \) is called a tree decomposition of \( G \) if:

1. \( \bigcup_{v \in V(T)} X_v = V(G) \);
2. for each edge \{v, w\} ∈ E(G), there exists a vertex t ∈ V(T) such that v, w ∈ X_t; and

3. for all u, v, w ∈ V(T), if v is on the path from u to w in T, then \(X_u \cap X_w \subseteq X_v\).

Figure 2.3 represents a tree decomposition of Figure 2.1. The width of a tree decomposition is equal to \(\max_{t \in T}\{|X_t| - 1\}\). As with branch decompositions, the number of possible tree decompositions for a graph G is large; the treewidth of G, which we shall denote \(tw(G)\), is the minimum width over all of its tree decompositions. An optimal tree decomposition is one that realizes this width.

Many researchers have explored the area of treewidth in terms of both building and using the decompositions. (See Bodlaender’s survey papers [7, 8] for overviews of the early research on treewidth.) On the other hand, branch decompositions and their applications have not been investigated nearly as much. Fortunately, many of the theoretical results for tree decompositions hold true for branch decompositions as well. In particular, Courcelle’s and Arnborg, Lagergren, and Seese’s results are applicable. Thus, one can use either decomposition for building algorithms to solve certain classes of NP-hard problems.

Moreover, we have the following bounds for branchwidth and treewidth.

**Theorem 2.1 (Robertson, Seymour [21])** For any graph G

\[
\beta(G) \leq tw(G) + 1 \leq \max(\lfloor(3/2)\beta(G)\rfloor, 2).
\]

A constructive proof of the right-hand inequality yields a linear-time transformation from a branch decomposition to a tree decomposition. We show that proof below and direct the reader to the paper for a proof of the left-hand inequality.

**Proof** [of right-hand inequality of Theorem 2.1] Isolated vertices have no effect on branchwidth or treewidth; assume that there are none. Given an optimal branch decomposition \((T, \tau)\) of G, for each vertex t of T we define a subgraph \(X_t\) as follows:
Figure 2.3  A tree decomposition of Figure 2.1 with width 4.

(i) if $t$ is a leaf of $T$, then $X_t$ is the set consisting of the endpoints of $\tau(t)$; and

(ii) if $t$ is an interior vertex, then $X_t$ is the set of vertices of $G$ each incident to a pair of edges mapped to the endpoints of a leaf-to-leaf path of $T$ that uses $t$.

One can easily verify that this transformation does indeed give a tree decomposition. (Figure 2.3 is an example of the transformation of the branch decomposition illustrated in Figure 2.2.) Let us bound the width of a tree decomposition generated using this method. For each leaf $t$ of $T$, $|V(X_t)| = 2$. For each interior vertex $t$ of $T$, we see that $X(t) = \bigcup\{\text{mid}(e)|e = \{v, t\} \in E(T)\}$. Of course, $t$ is incident to three edges and since the branchwidth of $G$ is $\beta(G)$, there are at most $3\beta(G)$ vertices in $X_t$. Moreover, we know this is too loose a bound because each vertex of $X_t$ is in the middle sets of at least two of the edges. Thus, $2|V(X_t)| \leq 3\beta(G)$ and the width of $(T, \{X_t|t \in V(T)\})$ is less than or equal to $\max([\lfloor (3/2)\beta(G) \rfloor, 2]) - 1$. So, $tw(G) \leq \max(\{(3/2)\beta(G), 2\})$, as stated. $\square$
The left-hand of inequality 2.1 supports the investigation of branch decompositions in and of themselves. That is, we can always expect the branchwidth of a graph to be close to if not smaller than the treewidth. Additionally, the structure and some properties of branch decompositions, which we explain in Section 3.1, are easier at times to exploit in developing algorithms than those of tree decompositions.

Typically, the running times of algorithms using decompositions increase exponentially (or worse) in the width of the decomposition. Thus, it is important to find an optimal or near optimal decomposition, that is, one having width close to the branchwidth of the graph.

Although finding an optimal branch decomposition of an arbitrary graph is an NP-hard problem [24], for planar graphs the problem is polynomial-time solvable.

**Theorem 2.2** (*Robertson, Seymour [22]*) For a planar graph $G$, there exists a polynomial-time algorithm to determine the branchwidth and an optimal branch decomposition of $G$.

Moreover, upper bounds on the branchwidth do exist for several classes of graphs; some bounds are given in the following theorem. The reader can consult, for example, West [29] for definitions of the classes of graphs not defined in this thesis but mentioned in the theorem.

**Theorem 2.3** Let $G$ be a graph with branchwidth $\beta(G)$. Then

- $\beta(G) = 0$ if and only if every component of $G$ has at most one edge; [Robertson, Seymour [21]]
- $\beta(G) \leq 1$ if and only if every component of $G$ has at most one node of degree $\geq 2$; [Robertson, Seymour [21]]
- $\beta(G) \leq 2$ if and only if $G$ has no $K_4$ minor [Robertson, Seymour [21]].
• $\beta(G) \leq 2$ if $G$ is a forest;

• $\beta(G) \leq 2$ if $G$ is a series-parallel graph;

• $[\frac{2}{3}\omega(G)] \leq \beta(G) \leq \omega(G)$ if $G$ is a chordal graph where $\omega(G)$ is the size of the largest clique in $G$;

• $\beta(G) \leq n$ if $G$ is an $n$-grid; and

• $\beta(G) = \lceil \frac{2}{3}\|V(G)\| \rceil$ if $G$ is complete.

From the breadth of graphs presented in Theorem 2.3, we can get a feel for the applicability of algorithms using branch decompositions in real-world and theoretical applications. In general, the best results are achieved over classes of graphs with low branchwidth.

Lastly, we note that Robertson and Seymour [22] provide a polynomial-time algorithm that determines a branch decomposition of a graph $G$ of width at most $3k$ or a subgraph of $G$ with width greater than $k$ provided $\beta(G) \leq k$.

2.3 Logical Structures and Graph Properties

Graphs—including decompositions—and their properties can be described as a finite logical statement. To the same extent, a graph problem, such as HAMILTONIAN CYCLE can be expressed as a logical statement. The logical language on which we focus is monadic second-order logic and some of its extensions. Before we describe those languages, however, it is important to discuss first-order logic from which they are built.

Mathematical logic is traditionally the study of whether statements are satisfied by elements defined by a logical structure. In other words, it asks “given characteristics of a set of elements, is an additional formula true?” The symbol $\models$ denotes the satisfaction relation between a structure and a formula. The statement “$S \models \phi$" is
interpreted “$S$ satisfies $\phi$” or “$\phi$ is true for $S$.” For a particular element $s$ in $S$, the two notions $s \models \phi$ and $\phi(s)$ are equivalent.

Graphs can be considered as logical structures and their properties can therefore be expressed as logical formulas. Any logical formula $\phi$ describes a set $G$ of graphs—namely, the set of graphs that satisfy $\phi$. Thus, as a logical language $\mathcal{L}$ defines a class of logical formulas, $\mathcal{L}$ defines a class of sets of graphs.

The basic logical language is first-order logic (FOL). Barwise [5] presents a rigorous discussion of first-order logic. FOL consists of the logical connectives, $\&$, $\lor$, $\neg$, $\Rightarrow$, and $=$ (meaning ‘and’, ‘or’, ‘not’, ‘implies’, and ‘equals’, respectively), variables (e.g. $x_1, y_1, a, b, z$) and universal ($\forall$) and existential ($\exists$) quantifiers. Note that the familiar connective $\Leftrightarrow$ or ‘if and only if’ is derived intuitively from the $\Rightarrow$ and $\land$ connectives. In addition to these logical symbols, a set $R$ of primitive non-logical symbols must be defined for each area of study to represent relationships specific to elements in the domain of variables.

Using FOL, Courcelle [11] defines the class of simple, directed graphs $S$. The domain of variables $V_G$ refers to the set of vertices of $G$ and the set of relational symbols $R$ consists only of $\text{edg}_G$, a binary relation analogous to the set $E(G)$ defined in Section 2.1. The relation $\text{edg}_G$ is defined as

$$\text{edg}_G(x, y) \iff G \text{ contains an edge from } x \text{ to } y.$$  

For $G \in S$, associate the logical structure $[G] := < V_G, \text{edg}_G >$.

An example of a first-order formula $\phi$ is

$$\forall x \exists y \ [ (\text{edg}(x, y) \lor \text{edg}(y, x)) \land \neg (x = y)] . \quad (2.2)$$

We read the statement as “for every $x$ there exists $y$ such that the graph in question contains an edge from $x$ to $y$ or the graph in question contains an edge from $y$ to $x$
and it is not true that \( x \) equals \( y \)." For a graph \( G \) in \( S \), we have \([G] = \phi\) if and only if \( G \) has no isolated vertices.

Generally, the domain for each variable in a logical statement is understood. For graphs in the class \( S \), there is only one domain; variables can range only over the set of vertices. Certain graph properties, however, cannot be expressed without quantification over edges. This need is addressed in \( D(A) \), the class of (not necessarily simple) directed graphs with edges labeled from a finite alphabet \( A \). For a graph \( G \) in \( D(A) \), let \( V_G \) and \( E_G \) denote its sets of vertices and edges respectively. Also define the following relational structures:

\[
\text{lab}_{aG}(z) \iff z \text{ is an edge in } G \text{ with label } a,
\]

\[
\text{edg}_{G}(z, x, y) \iff z \text{ is an edge in } G \text{ from } x \text{ to } y, \text{ and}
\]

\[
\text{edg}_{aG}(z, x, y) \iff \text{lab}_{aG}(z) \land \text{edg}_{G}(z, x, y).
\]

A graph \( G \) in \( D(A) \) can be represented by three different logical structures:

\[
[G]_1 := \langle V_G, E_G, (\text{edg}_{aG})_{a \in A} \rangle,
\]

\[
[G]_2 := \langle V_G, E_G, (\text{lab}_{aG})_{a \in A}, \text{edg}_{G} \rangle, \text{ or}
\]

\[
[G]_3 := \langle V_G, E_G, (\text{lab}_{aG})_{a \in A}, \text{edg}_{G}, (\text{edg}_{aG})_{a \in A} \rangle.
\]

Obviously, the last representation is redundant since it contains all of the relational structures in the first two.

The property (2.2) can be restated for graphs in \( D(A) \). Assuming \( x \) and \( y \) are vertex variables and \( z \) is an edge variable, let \( \Phi \) be the statement

\[
\forall x \exists y \exists z \left[ \bigvee \{ \text{edg}_{a}(z, x, y) \lor \text{edg}_{a}(z, y, x) \mid a \in A \} \land \neg(x = y) \right]. \quad (2.3)
\]

We read this statement similarly to the property (2.2): "for every \( x \) there exists \( y \) such that there exists a \( z \) such that in the graph in question, for some \( a \in A \), \( z \) is an
edge labeled a from x to y or in the graph in question z is an edge labeled a from y to x and it is not true that x equals y.

Notice that the same graph property, such as \( \Phi \) above, can hold for different graphs. This raises the issue of how to determine if two graphs are the same. One question we might ask is if changing the labels of a graph will nullify its properties.

Two graphs \( G_1 \) and \( G_2 \) in \( \mathbf{D}(A) \) are isomorphic if there exist two bijections: \( V_{G_1} \to V_{G_2} \) and \( E_{G_1} \to E_{G_2} \) that preserve the incidence relation \( \text{edg} \) and the labels \( a \in A \). In addition, if \( G_1 \) and \( G_2 \) are isomorphic, then the structures \( [G_1] \) and \( [G_2] \) are isomorphic also. A graph property is a statement that remains true under isomorphism. Thus, if \( \mathcal{C} \) is a class of graphs, then we say \( [G] \) represents a graph \( G \) belonging to \( \mathcal{C} \) provided that for every graph \( G' \) in \( \mathcal{C} \), \( [G] \) being isomorphic to \( [G'] \) implies that \( G \) is isomorphic to \( G' \).

A graph is unique up to isomorphism. That is, a logical language \( \mathcal{L} \) defines a class of logical formulas or graph properties. This class of formulas defines a set of graphs. We do not distinguish between isomorphic structures of an element in this set because the graph property which distinguishes the element will always be valid. If a structure \( [G] \) has been chosen to represent a graph \( G \) of the considered class \( L \), then \( L \) is \( \mathcal{L} \)-definable if there is a formula \( \phi \) in \( \mathcal{L} \) such that \( L = \{ G \mid [G] \models \phi \} \).

### 2.4 Monadic Second-Order Logic

We now discuss the language monadic second-order logic and its variants which have wider scope than FOL. We are continuing to expand the set of graph problems that can be stated as finite logical expressions.

Monadic second-order logic (MS) builds from first-order logic by adding vertex set and edge set variables and quantification over these sets. These additions increase the
number of decision problems posed for graphs that we can state as logical formulas. The basic (or atomic) formula concerning sets is element inclusion (i.e., \( x \in X \)).

For example, given the structure \([G]_2\) or \([G]_3\) for a graph \( G \) in the class \( \mathbf{D}(A) \), the following formula expresses 2-colorability of \( G \):

\[
\exists X, Y \left[ \forall x \left\{ x \in X \lor x \in Y \right\} \land \{ \neg(x \in X \land x \in Y) \} \right.
\land \forall x \forall y \forall z \left\{ \text{edg}_G(z, x, y) \Rightarrow \neg(x \in X \land y \in X) \land \neg(x \in Y \land y \in Y) \right\} \right].
\]

Clearly, \( x \) and \( y \) are vertex variables, \( X \) and \( Y \) are vertex set variables, and \( z \) is an edge variable. The statement would be read: "There exist two vertex sets \( X \) and \( Y \) such that for every vertex \( x \), \( x \) is in the set \( X \) or \( x \) is in the set \( Y \) and \( x \) is not in both \( X \) and \( Y \) and for every \( x, y, \) and \( z \), \( z \) is an edge in the graph \( G \) from \( x \) to \( y \) implies \( x \) and \( y \) are not both in \( X \) and \( x \) and \( y \) are not both in \( Y \)." In other words, there are two sets of vertices that partition \( V(G) \) with no edges in the graphs that they independently induce in \( G \). If the two sets exist, each set represents a color.

FOL is relatively weak—it can express only "local" properties. The set variables and quantification that MS adds are an important extension of first-order logic. Unfortunately, there are many graph problems that involve calculations over some vertex or edge set or involve determining the maximum or minimum of an evaluation over a similar set. These problems cannot be expressed as MS properties. To handle these types of problems the language of extended monadic second-order logic (EMS), described in the following paragraphs, is introduced in Arnborg, Lagergren, and Seese [3]. Table 2.1 gives examples suggesting the inclusion of the languages discussed thus far (i.e., \( \text{FOL} \subset \text{MS} \subset \text{EMS} \)).

Suppose a given MS formula has free variables \( X_1, X_2, \ldots, X_n \). Also, let there be evaluations \( f_1, f_2, \ldots, f_m \) over the sets that the \( X_i \)'s represent. An evaluation term, denoted \( F(\mid X_1 \mid_1, \ldots, \mid X_n \mid_m, Y_1, \ldots, Y_t) \), is an expression joining the atoms \( \mid X_i \mid_1 \equiv \mid X_i \mid_m \equiv \mid X_i \mid_t \equiv F(\mid X_1 \mid_1, \ldots, \mid X_n \mid_m, Y_1, \ldots, Y_t) \),
\[ \sum_{x \in X_i} f_j(x) \text{ for } 1 \leq i \leq n \text{ and } 1 \leq j \leq m \text{ and the scalar variables } Y_i \text{ for } 1 \leq i \leq t \]  using the arithmetic operators +, −, and ×. An evaluation relation is a propositional formula combining statements of the form \( F = 0 \) or \( F \leq 0 \) where \( F \) is an evaluation term.

A property \( P \) is an extended monadic second-order property (EMS property) over a class \( K \) of structures \( (G, f_1^G, \ldots, f_m^G) \) if there are constants \( n \) and \( t \), an MS formula \( \Phi \) with set variables \( X_1, \ldots, X_n \), and an evaluation relation \( \psi(Y_1, \ldots, Y_{nm+t}) \) such that for each structure in \( K \) the following statement holds:

\[
P(G, f_1^G, \ldots, f_m^G, C_1, \ldots, C_t) \text{ if and only if there are subsets } A_1, \ldots, A_n \text{ of the domain } A \text{ such that } G \models \Phi(A_1, \ldots, A_n) \text{ and}
\[
\psi\left[|A_1|^G_1, \ldots, |A_n|^G_n, C_1, \ldots, C_t\right]
\]

where the \( C_i \) are given in the problem instance.

This allows us to define perhaps the most common form of graph problems. The extended monadic second-order extremum problem can be stated as finding a maxi-
mum (or minimum) of an evaluation term $F([X_1|_1^G], \ldots, [X_n|_m^G, C_1, \ldots, C_t]$ over the class of objects that satisfy $\Phi[X_1, \ldots, X_n]$ and $\psi([X_1|_1^G], \ldots, [X_n|_m^G, C_1, \ldots, C_t]$ for a given MS formula $\Phi$ and evaluation relation $\psi$. A more recognizable statement of this problem is

\[
\begin{align*}
\text{Maximize} & \quad F([X_1|_1^G], \ldots, [X_n|_m^G, C_1, \ldots, C_t]) \\
\text{subject to} & \quad \Phi[X_1, \ldots, X_n] \\
& \quad \psi([X_1|_1^G], \ldots, [X_n|_m^G, C_1, \ldots, C_t]).
\end{align*}
\]

The linear extended monadic second-order extremum problem simply requires that the evaluation term $F$ be linear in the quantities $|X_i|_j$ and that the evaluation relation $\psi$ be identically true.

## 2.5 Decidability of MS Properties

A property or formula $\Phi$ is \textit{decidable} if there exists an algorithm AL that will determine whether $\Phi$ is true. Moreover, a problem instance $\Pi$ is \textit{decidable} if an algorithm AL exists that will produce the correct answer (e.g., ‘yes’, ‘no’, or $n$).

Arguably, Courcelle’s [11, 12, 13] is the most important decidability result in this graph context. We present a sketch of the proof because it gives insight into the framework we use for our algorithms.

**Theorem 2.4 (Courcelle [11])** Let $\phi$ be a monadic second-order logic problem and $\mathcal{K}$ a class of graphs with branchwidth bounded above by $k$. For a graph $G$ in $\mathcal{K}$, it can be determined in polynomial time if $G$ satisfies $\phi$. If $G$ is given with a branch decomposition with width less than or equal to $k$, then a linear time algorithm exists.
Proof [sketch] Let \((T', \tau)\) be a branch decomposition of \(G\). Create a rooted decomposition tree \((T, \tau)\) from \((T', \tau)\). At each node \(t \in T\) we refer to the middle sets of the edges incident to \(t\) as the active middle sets and define \(X(t)\) to be the union of the active middle sets. That is, \(X(t) \equiv \bigcup \{ \text{mid}(e) | e = \{v, t\} \in E(T) \}\). Also for each node \(t \in T\) let \(G_t\) be defined as the subgraph of \(G\) induced by the union of the edges at the leaves of \(T_t\). We call \(G_t\) the active subgraph at \(t\). Label each node \(t \in T\) with MS formulas that characterize \(G_t\). Define a tree automaton \(M = (S, \Sigma, \delta, s, A)\) that executes the labeled decomposition \((T, X, G_i)\) where \(S\) is a set of states, \(\Sigma\) a finite set of labels, \(\delta : S \times S \times \Sigma \rightarrow S\) a state transition function, \(s\) the initial state, and \(A\) the set of accepting states such that if \((T, X, G_i)\) evaluates to a state in \(A\) then \(G\) satisfies \(\phi\).

The size of a branch decomposition tree of \(G\) is linear in the size of the graph. Transforming the branch decomposition into a rooted branch decomposition takes constant time. Determining the middle sets of each edge and defining \(G_t\) each take linear time. A tree automaton processes the labeled decomposition in time that is linear in the size of the tree. Thus, \(\phi(G)\) can be determined in linear time with respect to the size of the input graph \(G\).

Arnborg, Lagergren and Seese [3] show the same result, which they explicitly extend to EMS and EMS extremum problems. First, they show that any graph \(G\) with bounded treewidth can be transformed in polynomial time to a labeled binary tree \(T\). Then, for any MS property \(\Phi\), one can decide \(\Phi(T)\) in linear time.

All that is left is finding a good tree decomposition of \(G\). To this end, we use the proof of Theorem 2.1 that details a linear transformation between branch decompositions and tree decompositions. And, to find the branch decomposition of \(G\), we
may use the algorithm of Robertson and Seymour [22] referred to in Section 2.2. In practice, however, we use the branch decomposition heuristic developed by Hicks [15].

Courcelle’s theorem has spurred a flurry of research activity in the scientific computing community. For example, Scheffler describes a linear-time algorithm for solving the DISJOINT PATHS problem [23] for graphs with small treewidth. Arnborg, Lagergren, and Seese [3] define EMS properties, prove their decidability and offer an extensive list of examples of MS, EMS, EMS extremum and linear EMS extremum problems. Many others have concentrated their efforts on algorithms to build optimal decompositions. Bodlaender [7] provides a survey of research and results in the area of treewidth.
Chapter 3

Algorithms Using Branch-Decompositions

Using a tree-structuring of a graph \(G\) we can recursively validate a logical property starting at the leaves of the tree and continuing to its root. We begin this chapter with a simple example of dynamic programming and extrapolate some general requirements for an efficient dynamic programming algorithm. We then explain some of the properties of branch decompositions and how they relate to the framework we use in our algorithms. Lastly, we describe major factors influencing the computational complexity of the algorithms and how they are addressed in our implementation.

As a subject for dynamic programming, consider the problem \textsc{Maximum Independent Set} in which we wish to find the largest subset of vertices \(\mathcal{I}\) such that no two vertices in \(\mathcal{I}\) are adjacent. The problem is \textsc{NP}-hard for general graphs; however, an instance is linear-time solvable if the graph is known to be a tree or forest.

For every vertex \(v\), let \(a_v\) be the cardinality of a maximum independent set in \(T_v\) including \(v\) and let \(b_v\) be the cardinality of maximum independent set in \(T_v\) excluding \(v\). The size of the maximum independent set for \(T\) is the maximum of \(a_r\) and \(b_r\). For a vertex \(v\), let \(w_i\) for \(i = 1, \ldots, c\) be its children. Noticing that \(a_v = 1 + \sum_{i=1}^{c} b_{w_i}\) and \(b_v = \sum_{i=1}^{c} \max\{a_{w_i}, b_{w_i}\}\), we see that the cardinality of the maximum independent set can be computed in \(O(|V(T)|)\) time by computing the required values in a depth-first post-processing order.

Figure 3.1(a) shows an instance of this problem having each vertex \(v\) labeled with the pair \((a_v, b_v)\). The size of the maximum independent set is the greater of the two values at the root (i.e., nine). By remembering at each node which values at its
Figure 3.1 Determining a maximum independent set of a tree. Tree (a) shows the vertices of the tree labeled with the values \((a_v, b_v)\). In tree (b), the shaded vertices represent a maximum independent set derived from the values in tree (a).

Consider a tree, where the values were used to calculate the values at the node, one can find a solution to the problem in \(O(|V(T)|)\) time. In the example, the value '9' at the root comes from including vertex 0 and excluding vertices 1 and 2. Excluding 1 allows us to include 3 and 4 (note that since \(a_3 = b_3\), 3 could be excluded instead of 4). The inclusion of 3 requires the exclusion of 7 and 8, which allows the inclusion of 12 and 13. A similar construction is carried out for the right subtree \(T_2\) to get the optimal solution that is shown in Figure 3.1(b).

At each vertex \(v\) of the tree, the solution for the subtree \(T_v\) is computed using only the information available from the descendants of \(v\). This algorithm for **MAXIMUM INDEPENDENT SET** is an example of dynamic programming. Similar techniques exist for many other graph problems if the input graph is known to be a tree.
3.1 Decomposition Trees

Let us look at the concept of tree-based dynamic programming more formally. We begin by defining decomposition trees, which represent a formal and ordered approach to finding solutions to a problem on a graph.

A \textit{k-terminal graph} $G = (V, E, X)$ is a graph $G' = (V, E)$ with an ordered list $X = \{v_1, v_2, \ldots, v_t\}$ of distinguished vertices, called terminals, such that $t \leq k$. The terminals are also called sources in, for example, Courcelle [11]. A $k$-terminal graph that has no non-terminal vertices is called a base graph. A $k$-terminal graph composition function $f$ of arity $c$ is written $G = f(g_1, \ldots, g_c)$ where $g_i$ for $1 \leq i \leq c$ and $G$ are distinct $k$-terminal graphs. For any $k$-terminal graph $G$, a decomposition tree is a rooted tree with node properties $f_v$ and $g_v$ such that

(i) $g_v = G$ if $v$ is the root;

(iv) $g_v$ is a base graph if $v$ is a leaf;

(ii) $f_v$ is a graph composition if $v$ is an interior node; and

(iii) $g_v = f_v(g_{w_1}, \ldots, g_{w_c})$ if $v$ is an interior node and $w_1, \ldots, w_c$ are its children.

Notice that we used the term node when referring to the decomposition tree. While in general distinguishing between the use of the terms node and vertex is a matter of preference, we shall refer to vertices as being elements of general graphs and nodes as being elements of decomposition trees.

The maximum independent set example above can be solved using the $k$-terminal graph framework. The particulars of defining the decomposition tree are left to the reader.

Our reason for introducing yet another decomposition is that decomposition trees provide a natural structure for efficient dynamic programming algorithms that can be
applied to branch decompositions. The characteristics that we would like to replicate for branch decompositions are the following:

- the solution for the problem on the entire graph can be derived from information at the root of the decomposition tree;

- the solutions at each leaf are computable; and

- the solutions at each node of the tree can be computed using only information available from its children.

These properties allow us to construct a tree-based dynamic programming algorithm to solve a problem defined for the graph at the root of the decomposition tree.

Tree traversal can be implemented in linear time provided the tree is stored so that the data structure for a node references the node's set of children. Thus, if the subproblems defined at the nodes of the tree can be computed efficiently, then the problem at the root, and consequently, on the entire graph, can be computed efficiently also. In other words, given the graph $g_v$ defined at an arbitrary node $v$, if the solutions for $g_v$ can be computed from the solutions at the children of $v$ in time $O(t(|G|))$, then it takes $O(|V(T)| \cdot t(|G|))$ to compute the solutions at all of the nodes including the root. If $t(\cdot)$ is a polynomial, then the solution can be computed in polynomial time; if $t(\cdot)$ is constant, then the solution can be computed in linear time.

Not surprisingly, a branch decomposition can be transformed into a rooted binary decomposition tree. This transformation is started by adding an artificial node of degree two along an arbitrary edge and designating that node as the root. The $k$-terminal graphs at each node are defined according to the following rules.

- If $v$ is a leaf of $T$, the base graph at $v$ is $\tau(v)$, the edge of $G$ mapped to $l$, and its terminals are the endpoints of $\tau(v)$. 
• If \( v \) is an interior node of \( T \), the \( k \)-terminal graph at \( v \) is the subgraph of \( G \) induced by the edges mapped to the leaves of \( T_v \) with terminals consisting of the middle set at the edge \((p, v)\) where \( p \) is the parent of \( v \).

The composition function is simply to take the union of the two subgraphs identifying vertices with the same index or label in the original subgraphs as the same vertex in the new graph. It is easily explained through the study of rebuilding a graph using its decomposition. As an example, let us consider the recomposition of the graph in Figure 2.1 from a rooted branch-decomposition-based on the branch decomposition in Figure 2.2. The recomposition is depicted in Figure 3.2. Note that the base graphs are as described above and that indeed each \( k \)-terminal graph at any nonleaf node \( v \) (including the root) can be created by the union of the graphs defined at \( v \)'s children.

For the remainder of this thesis, we shall assume that the branch decomposition tree \( T \) is rooted at some artificial node \( r \); however, when referring to interior nodes we do not regard the artificial one.

The transformation described above is similar to the transformation from branch to tree decomposition in the proof of the right-hand side of Theorem 2.1. Why do we not use a tree decomposition as the initial decomposition? We maintain that using a branch decomposition as the initial decomposition presents several properties of which we will take advantage in our algorithms. In particular, in a branch decomposition, we know that each leaf is mapped to an edge in the graph; therefore, all the base graphs are isomorphic to each other. This simplifies the computation of solutions for the base graphs. Also, we know that every nonleaf node of the rooted decomposition has exactly two children which means that the composition function and the join operation that we discuss in subsequent chapters only need to be defined with two arguments. And as a special note, since this structure guarantees that each
Figure 3.2  Recomposition of the graph in Figure 2.1 from the branch decomposition in Figure 2.2.
edge of $G$ can be considered individually, it is a nice framework for problems that involve edge sets or whose solutions depend on vertex adjacency. We should confess that tree decomposition based algorithms can use a transformation of the tree decomposition into a nice tree decomposition[23] which reduces some of the aforementioned complications of algorithm design.

In general, an interior node $v$ has exactly two children which we refer to as $l$ and $r$ corresponding to its out-edges $e_l = (v, l)$ and $e_r = (v, r)$, respectively, and one parent node $p$ corresponding to its in-edge $e_p = (p, v)$. We will refer to the middle sets of the edges out of $v$ as the active middle sets at $v$.

Another property of branch decompositions concerning middle set interactions is the following lemma, which is summarized by the statement “Once a vertex leaves the middle set, we never have to worry about it again”:

**Lemma 3.1** Given a branch decomposition $(T, \tau)$ of a graph $G$, let $v$ be in $V(T)$ and let $T$ be a rooted tree. If vertex $n$ is in the middle set of the edge $(v, l)$ and not in the middle set of $(p, v)$ for a child $l$ and parent $p$ of $v$, then $n$ is not in the middle set of any edge in $E(T_p)$. Moreover, $n$ must be in the middle set of $(v, r)$ where $r$ is the other child of $v$.

**Proof** The validity of the lemma follows directly from the definition of middle set. Given $e_p = (p, v)$, $e_l = (v, l)$, and $e_r = (v, r)$ are edges in $T$ as described in the lemma, let $(a, b)$ be an edge in $E(T_p)$. If a vertex $n$ is in mid($(v, l)$), then by definition $S$, the set of leaves mapped to edges incident to $n$, is split between $T_v$ and $T_l$. If $n$ is not in mid($(p, v)$), then $S$ is in exactly one of $T_p$ or $T_v$. Without loss of generality, let us say $S \in T_v$. Since $T_v$ is a subgraph of $T_a$ (or $T_b$) for any edge $(a, b)$ in $E(T_p)$, $S$ will be completely contained in $V(T_a)$ or $V(T_b)$. Thus, $n$ is not in mid($(a, b)$).
The second statement of the lemma is just as straightforward. Since the leaves \( \lambda(T_v) = \lambda(T_i \cup \lambda(T_r)) \), if \( S \) is contained in \( T_v \), then \( S = S_l \cup S_r \) where \( S_l \subset \lambda(T_i) \) and \( S_r \subset \lambda(T_r) \). We have assumed that \( S \) is not completely contained in \( T_i \). Thus, \( S_l \neq \emptyset \) meaning \( n \in \text{mid}((v,r)) \). \( \square \)

Not having to “worry about” a vertex that has left the middle set is key to reducing the data storage requirement. Simply put, for any given node in the decomposition tree, only the information for the active middle sets at that vertex is important.

3.2 Algorithmic Framework

Before we describe the algorithms that use this rooted decomposition tree and discuss implementations, we must define several notions relating to solutions. Until this point in the thesis, the reader’s intuition has been sufficient. In general, a solution \( s \) on a graph \( G \) is an assignment of values to the vertices of \( G \) that satisfies a given MS property. The interpretation of the assignment is problem dependent. In, for example, \textsc{Independent Set} the assignment represents a partitioning of the vertices into two or more sets (e.g., all vertices with a specific value would be in the same set).

If \( H \) is a subgraph of \( G \) then a partial solution \( s_H \) on \( H \) is an assignment of values to the vertices of \( H \) such that there exists a graph \( G' \) with a subgraph \( H' \) that is isomorphic to \( H \) and a solution on \( G' \) that when projected onto \( H' \) is equivalent to \( s_H \). For the sake of brevity, we often refer to partial solutions simply as solutions when there is no danger of ambiguity. Two partial solutions \( s_1 \) and \( s_2 \) on subgraphs \( g_1 \) and \( g_2 \) of \( G \), respectively are compatible if they can be combined to form a partial solution on the graph \( f(g_1, g_2) \). Note that a partial solution might not correspond to a solution on the entire graph. It is acceptable to keep such solutions because they will be automatically detected when, at some interior node, the partial solution or
an extension of it will be found to not be compatible with any other partial solution. Hence, we can be sure that infeasible solutions will not be present at the root. Aside from this assurance we should note that determining the feasibility or infeasibility of a partial solution from locally available information at the node where it is generated is not possible.

From the sketch of the proof of Theorem 2.4, we have derived the following algorithmic framework.

**Algorithm 1**

INPUT: A graph $G$, a branch decomposition $(T', \tau)$ of $G$

OUTPUT: A set of solutions to $\Phi$ in $G$ if such exist

create a rooted branch decomposition $(T, \tau)$ with root $r$

for each node $t$ of $T$

if $t$ is a leaf

    generate the set $S_t$ of leaf solutions

    store $S_t$ at $t$

else // $t$ is interior

    let $S_1 \equiv$ Solutions stored at left child of $t$

    let $S_2 \equiv$ Solutions stored at right child of $t$

    for each solution $s_1$ of $S_1$

        for each solution $s_2$ of $S_2$

            if $s_1$ and $s_2$ are compatible

                $s_t = \text{JOIN}(s_1, s_2)$

                add $s_t$ to the set $S_t$ of solutions at $t$

$S_r = $ the solutions to $\Phi$ in $G$
And again, from the proof of Theorem 2.4, we know that this framework has the potential to be efficient. Whether or not it actually is efficient depends tremendously on the efficiency of generating leaf solutions, of determining if solutions are compatible, and of joining solutions.

Of the many possibilities, we use the worst-case consumption of space (RAM) and time (number of operations) as our measures of computational complexity because they are straightforward and easiest to compute. The efficiency of the algorithm depends on the efficiency of the data structures used and on the subroutine implementations. In Section 3.5, following the discussions of data structures and subroutines, we prove the linearity of the algorithmic framework and its possible extensions.

3.3 Solution Storage

The basic data structure for storing the partial solutions contains the values assigned to the terminals, the rank, and the indices of the left partial solution and the right partial solution that were joined to generate it. Figure 3.3 depicts this information as a table. Recording the indices is important for the construction phase; it allows traversing back down the tree (from the root to the leaves) to build the solution.

For the problems that we have addressed in this thesis, this data structure is used in three different ways. We discuss two of these ways below because they are common to the vertex partitioning algorithms discussed in both Chapters 4 and 5. The third usage, being specific to the algorithms in Chapter 6, is discussed there.

Fixed Partitioning

A very straightforward solution type is what we call fixed partitioning. At each node of the decomposition tree and for each solution, the values assigned to the terminal vertices are recorded. Thus, a partial solution is characterized or distinguished from
<table>
<thead>
<tr>
<th>rank</th>
<th>1</th>
<th>2</th>
<th>...</th>
<th>m</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$r_1$</td>
<td>$r_2$</td>
<td></td>
<td>$r_m$</td>
</tr>
<tr>
<td>1</td>
<td>$s_{11}$</td>
<td>$s_{12}$</td>
<td>...</td>
<td>$s_{1m}$</td>
</tr>
<tr>
<td>2</td>
<td>$s_{21}$</td>
<td>$s_{22}$</td>
<td>...</td>
<td>$s_{2m}$</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t</td>
<td>$s_{tl}$</td>
<td>$s_{t2}$</td>
<td>...</td>
<td>$s_{tm}$</td>
</tr>
</tbody>
</table>

**Figure 3.3** Storage of partial solutions for a $k$-terminal graph. The value $s_{ij}$ for each terminal $i$ is stored for each solution $j$. ($t \leq \beta(G)$, $m =$ the number of solutions)

another partial solution by the values assigned to the terminals. When using this data structure, every feasible partial solution for the subgraph at a leaf of the composition tree is generated. These solutions are joined with others successively to generate the solutions for the entire graph.

Initially, this data structure seems sufficient for the algorithm implementation. It allows us to quickly store all the necessary information in a minimal amount of space. Partial solutions can be indexed quickly as can values for terminals.

Indeed, fixed partitioning can be efficient when the number of distinct values or solution states is known in advance and it is known to be small. Knowing this quantity or predicting an accurate upper bound for it beforehand is important for the following reasons. If the bound is too low, the solution to the problem may not be generated because there are not enough distinct solution states defined to represent it; the result is an error when the algorithm terminates prematurely answering that no solution exists. If the bound is too high, the algorithm may take an excessively long time to run because too many solutions are being generated and stored.
Isomorphic Partitioning

We are not guaranteed to know the optimal number of solution states before generating the solution; however, we do know that at any node of an optimal decomposition tree, there are at most $\beta(G)$ different states in any stored partial solution. Isomorphic partitioning, an alternative solution storage method, characterizes the solution not by the value at the terminals but by membership to a solution state. That is, all vertices with the same value are in the same state. In this thesis, we use these solution states to refer to set partitions. If every partition or state must observe the same property, then we can assume that those partitions or states are isomorphic—as in the case of Graph Coloring. There are at most $\beta(G)^{\beta(G)}$ solutions or partial solutions at any node of the decomposition tree.

In discussions of computational complexity, we tend to shy away from factorials more than we do exponentials. Indeed, at first glance, this value strikes us as prohibitively large; however, it is likely that $\beta(G)$ is significantly less than $s$, the actual number of possible solution states over the entire graph $G$. This makes generating possibly $\beta(G)^{\beta(G)}$ partial solutions an appealing alternative to generating $s^{\beta(G)}$ partial solutions.

With this characterization of solutions, we no longer need a priori knowledge about the possible values for the terminals. Instead of checking if vertices have been assigned the same values, comparisons are made as to whether or not state sets agree in their members. Checking compatibility in this manner is not as straightforward for isomorphic partitioning as for fixed partitioning; however, it is still true that if two solutions agree in the states or values assigned to each vertex in the intersection of the active middle sets then they are compatible.
For isomorphic partitioning, partial solutions must be rewritten before compatibility can be tested.

**Algorithm 2 (Intersection Ordering)**

\[
\text{COUNT} = 0
\]

for each vertex \( v \) in the middle set

if \( v \) is in the intersection of the active middle sets

if MAP\([s_v]\) is undefined

\[ \text{MAP}[s_v] = \text{COUNT} \]

\[ \text{COUNT} + = 1 \]

\[ s_v = \text{MAP}[s_v] \]

for each vertex \( v \) in the middle set

if \( v \) is not in the intersection of the active middle sets

if MAP\([s_v]\) is undefined

\[ \text{MAP}[s_v] = \text{COUNT} \]

\[ \text{COUNT} + = 1 \]

\[ s_v = \text{MAP}[s_v] \]

Let \( s \) be a partial solution. For the sake of the current explanation, it does not matter if \( s \) is from the left or right set of solutions. Also let \( f(s_v) \) return the lowest index of a vertex in the intersection of the active middle sets whose value in \( s \) is \( s_v \). After the rewriting, our result is a reordering or mapping of the values so that for \( v_1 \) and \( v_2 \) in the intersection of the active middle sets, if \( v_1 < v_2 \) then \( f(s_{v_1}) \leq f(s_{v_2}) \). For two vertices \( v_1 \) and \( v_2 \) not in the intersection for which neither \( s_{v_1} \) nor \( s_{v_2} \) are assigned to vertices in the intersection, we also have \( f(s_{v_1}) \leq f(s_{v_2}) \) whenever \( v_1 \leq v_2 \). Given
this intersection ordering, we can determine compatibility of left and right partial solutions.

**Dominance**

Regardless of the data structure or solution type that we use, we can apply the following technique called *dominance* for reducing the number of solutions that must be stored at a node of the decomposition tree. Recall from Lemma 3.1 that we only need to store information concerning the active middle sets, or terminals, and that the nonterminal nodes are locally unknown. Suppose that there are two different partial solutions having the same values for the terminal nodes and possibly different values for the nonterminal nodes. If we have some method of ranking solutions—for example, by set cardinality or by objective value for monadic second-order extremum problems—then we need store only the solution having the best rank and discard the other agreeing or matching solutions.

The justification for discarding the dominated partial solution is simple: Since the two are locally indistinguishable, they are compatible with exactly the same set of partial solutions and therefore any extensions will be locally indistinguishable partial solutions. Moreover, for extremum problems, the extensions of the partial solution with the highest rank will always be chosen over those of the dominated partial solution. Note that we have assumed that the rank is non-decreasing as a result of the join operation.

The reader may have observed that using isomorphic partitioning our algorithm may generate two partial solutions that are isomorphic, yet dominance as described above only tests for local equivalence. In order to facilitate dominance checking we perform what we call *zero ordering* on each solution before it is stored. Zero ordering would be equivalent to intersection ordering if all the vertices were in the intersection.
3.4 Subroutines

In this section we discuss, in a little more detail, the key operations of our algorithm—the leaf and join operations. These operations are the keys to the efficiency of the overall algorithm.

Generating Leaf Solutions

At any leaf of the decomposition tree, the base graph is isomorphic to the graph \( \{a, b\}, \{\{a, b\}\} \). Generating the leaf solutions for each base graph consists of exactly the same steps. In fact, for each vertex partitioning problem, if we ignore the vertex labels, the set of solutions on one base graph is identical to the set of solutions on any other base graph. Thus, if a leaf operation takes \( t(l) \) steps to complete for some leaf \( l \), then it takes \( O(|E(G)| \cdot t(l)) \) time to solve all of the leaf subproblems.

Joining Partial Solutions

Suppose \( v \) is an interior node or the root of a decomposition tree and that \( v \) has children \( l \) and \( r \). The join operation is the process of using the solutions created at \( l \) and \( r \) to create new partial solutions for the subgraph defined at \( v \).

Note that all of the problems addressed in this thesis involve edge partitions or vertex partitions that satisfy an edge property. Also note that, as mentioned in Section 3.1, the rooted branch decomposition is aptly suited for such problems. This assertion is important when we consider that at the leaf we only generate partial solutions that satisfy the given edge property and, therefore, each partial solution will still observe the property for the subgraph defined at the interior nodes. We can conclude that all solutions at the root (i.e., for the entire graph) will be feasible. This idea is expounded upon in the discussion of each specific algorithm.
3.5 Linearity of the Framework

Since we have explained the leaf and join operations, it is easy to see that linearity of the algorithmic framework (Algorithm 1) is dependent upon the implementation of the join operations.

Let $G = (V, E)$ be an arbitrary graph with branchwidth $\beta$. The number of nodes in the decomposition tree is $2|E| - 2$ and thus linear with respect to $|E|$. Traversing the decomposition tree can be accomplished in time that is $O(T) = O(O(|E|))$ or linear with respect to $|E|$. The total running time of the algorithms will be

$$O(|E| \times \text{leaf operations}) + O(|E| - 2) \times \text{join operations}).$$

Because all the subgraphs defined at leaves are isomorphic to each other and hence invariant, creating the leaf solutions requires constant running time. The only unknown in the calculation is the join operations. Therefore, the total running time for an algorithm that uses the described algorithmic framework is linear if and only if its join operation is constant with respect to the size of the graph $G$.

3.6 Algorithms Preview

In the following chapters, we present several algorithms to solve various types of graph problems. The chapters will follow an outline similar to the following.

Each discussion begins with a statement of the problem in monadic second-order logic. This is an important procedure when we consider the applicability of Theorem 2.4. Recall that the theorem has two assumptions: the boundedness of the branchwidth of the graphs and the expressibility of the problem in MSL. Since we know that for any integer $k$, there will always exist graphs with branchwidth bounded above by $k$, all that remains before confirming the existence of an algorithm that ap-
plies the theorem is to prove that the problem in question is indeed expressible in monadic second-order logic.

A description of the subroutines that are specific to the problem follows the MS statement. This includes characterizations of the solutions and partial solutions. These definitions are then used to determine base solutions and the join operations. As with any algorithm we must prove that the algorithm generates only feasible solutions and we prove that the algorithm has the potential to generate all feasible solutions.

For each of the graph problems addressed, there is a decision problem and an optimization problem; we continue our discussion by explaining how to express the problem as an EMS extremum problem and how to solve both the decision and optimization variants.

Lastly, each problem explanation concludes with computational results. The decompositions are generated using Hicks's[15] algorithm. Except where noted, the serial and parallel algorithms were tested on a four-processor 200 MHz Pentium II with one gigabyte of RAM. The Java Virtual Machine is that distributed from Sun Microsystems in Java distribution 1.4. The graphs are from various libraries of graphs some of which are available for download from the Internet.
Chapter 4

Vertex Bipartitioning Problems

By vertex bipartitioning problems we mean those problems with the goal of separating the set of vertices of a graph into two subsets having a given property. From this class of problems we have implemented **Maximum Independent Set**, **Maximum Cut**, and **Graph Bisection**.

For each of these problems, the join procedure is the same; while the leaf solutions are problem specific.

4.1 Join Operation

If \( v \) is an interior node or root with children \( l \) and \( r \); then let \( S_l \) and \( S_r \) represent both the tables for (and the solutions sets at) \( l \) and \( r \), respectively. Because the sets of vertices \( X \) and \( V(G) \setminus X \) might not have the same property, a solution \( s_l \) from \( S_l \) is compatible with a solutions \( s_r \) from \( S_r \) if for each vertex \( n \) in the intersection of the active middle sets we have \( s_{nl} = s_{nr} \). Compatible solutions are joined by the following algorithm.

**Algorithm 3**

\[
\text{JOIN}( \ s_l, \ s_r \ ) \\
\text{BEGIN} \\
\quad \text{for } v \text{ in union of active middle sets} \\
\quad \quad \text{if } v \text{ is in the left middle set} \\
\quad \quad \quad s_v = s_{vl} \\
\quad \quad \text{else} \\
\text{END}
\]
\[ s_u = s_{ur} \]

END

We now complete the proof of linearity started in Section 3.5. First, observe that each join is \( O(\beta) \) and that there are at most \( 2^\beta \) partial solutions for each child. Thus, there are at most \( 2^\beta \times 2^\beta = 2^{2\beta} \) pairs of \( O(\beta) \) joins or \( O(\beta 2^{2\beta}) \) operations. Since \( \beta \) is not a function of the size of the instance graph, the collection of join operations is constant with respect to the graph.

The property that each edge of the original graph is treated individually is useful in determining bipartitions that satisfy an edge property \( \Phi \). Since at the leaves, we never generate a partial solution that violates \( \Phi \) at the subgraph defined at that leaf, we will never generate a partial solution through the join procedure that violates \( \Phi \) at the locally defined subgraph. At the root, this subgraph represents the entire graph \( G \), so we know that we will never generate an infeasible solution on the graph. It remains to be shown that this join procedure and the problem specific leaf solutions will generate all feasible solutions in \( G \).

### 4.2 Maximum Independent Set

Our first problem is **Maximum Independent Set**—also known as **Maximum Stable Set**. A set \( X \) is an independent set in a graph \( G \) if it satisfies the monadic second-order formula

\[
\Phi_G(X) = \forall x, y, z \left( \neg (x = y) \land (x \in X) \land (y \in X) \right) \Rightarrow \neg \text{edg}_G(z, x, y). \tag{4.1}
\]

Statement 4.1 is read as "for all triplets \( x, y, z \), if \( x \) is not equal to \( y \), \( x \) is an element of \( X \) and \( y \) is an element of \( X \), then there does not exist an edge \( z \) between \( x \) and \( y \)." The objective is to find the largest set \( X \) of vertices in a graph such that
there are no edges in the subgraph induced by $X$. Note that finding a maximum independent set for a graph is equivalent to the **maximum clique** problem for the graph's complement.

This problem is one in which we know beforehand the number of states that are in a solution—a vertex is either in or not in the independent set. Therefore, it is appropriate to use the fixed partitioning data type discussed in Section 3.3. We cannot use the isomorphic partitioning data type from the same section because the two sets do not have the same properties. That is, the set $X$ must be an independent set while the set $V(G) \setminus X$ has no required property.

Recall that the subgraph at every leaf of the decomposition tree is isomorphic to the graph $\langle \{a, b\}, \{\{a, b\}\} \rangle$—a single edge. Since there can be no edges between vertices in the stable set $X$, we know that the set of partial solutions for this subgraph consists of only the solution with only $a$ in $X$, the solution with only $b$ in $X$, and the solution with neither vertex in $X$. These three partial solutions are represented in Figure 4.1.

It is now necessary to show that this set of leaf solutions and the common join procedure will generate all independent sets in $G$. To this end, consider an independent set $s$ in $G$. It is necessarily true that $s$ projected onto any subgraph $H$ of $G$ is

<table>
<thead>
<tr>
<th>Solution</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>rank</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$a$</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$b$</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>left</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>right</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

*Figure 4.1* Table of leaf solutions when solving **maximum independent set**.
independent in $H$. Given an interior node $v$ consider the subgraphs $g_l$ and $g_r$ at the
children of $v$. Let $s_l$ and $s_r$ be the projections of $s$ onto $g_l$ and $g_r$ respectively. The
values assigned to each vertex in the intersection of the active middle sets must, of
course, be equal. Thus, the solutions are compatible and will be joined together to
form a new feasible partial solution that is the projection of $s$ onto $f_v(g_l, g_r)$. The
last necessary observation is that $s$ projected onto each edge mapped to a leaf of $T$
is represented in the set of leaf solutions described above.

If we implement the dominance method described in Section 3.3, we do not actu-
ally generate every feasible solution. However, recall that we will not keep a solution
only if there is a locally identical solution that is quantifiably better.

This concrete measure of how good a solution is is what we call the rank. For
independent sets, the rank we use is the size of the set. Thus solutions with higher
rank dominate locally identical solutions with lower rank. Moreover, by choosing
the solution at the root with the highest rank, we can solve the problem MAXIMUM
INDEPENDENT SET.

Maximum Independent Set

Let $f_1(x) = 1$, $m = n = 1$, $t = 0$, $\psi = \text{true}$, and $F(|X_1|_1) = |X_1|_1$. Then, MAXIMUM
INDEPENDENT SET can be stated as the linear EMS problem

$$
\begin{align*}
\text{Maximize} & \quad |X| \\
\text{subject to} & \quad \Phi_G(X).
\end{align*}
$$

The maximum independent set problem was implemented in Java with the preceding
leaf solutions and join operation. Some sample running times from the implementa-
tion are plotted in Figures 4.2, 4.3, and 4.4. The plots suggest that the algorithm
is linear; and, in fact, we already proved its linearity based on the description of the
subroutines in Section 4.1.
Figure 4.2  Maximum Independent Set, branchwidth = 3

Figure 4.3  Maximum Independent Set, branchwidth = 9
4.3 Maximum Cut

An edge cut $Z$, as defined in Section 2.1, is the set of edges $\delta(X)$ given a set of vertices $X$. The two disjoint sets $X$ and $Y = V(G) \setminus X$, of course, bipartition the set of vertices; however, the only properties of $X$ and $Y$ are that $X \cap Y = \emptyset$ and $X \cup Y = V(G)$. If $X$ and $Y$ satisfy the following MS statement then they bipartition the set of vertices.

$$
\Phi_G(X, Y) = \forall x \left[ \{(x \in X) \lor (x \in Y)\} \land \neg \{(x \in X) \land (x \in Y)\} \right] 
$$

(4.3)

In order to evaluate the edge set $Z$, we must use the following MS property.

$$
\Phi_G(X, Y, Z) = \forall x \left[ \{(x \in X) \lor (x \in Y)\} \land \neg \{(x \in X) \land (x \in Y)\} \right] 
\land \forall x \forall y \forall z \left( \{(x \in X) \land (y \in Y)\} \lor \{(x \in Y) \land (y \in X)\} \right) 
\land \text{edg}_G(z, x, y) \Rightarrow (z \in Z) 
$$

(4.4)
\[ \land \{ [(x \in X) \land (y \in Y)] \lor [(x \in Y) \land (y \in X)] \} \]

From Statement 4.4 (or Statement 4.3) it is obvious that reversing \( X \) and \( Y \) has no effect on the validity of \( \Phi_G \) (i.e., \( \Phi_G(X, Y, Z) = \Phi_G(Y, X, Z) \)). We can use either the fixed partitioning or the isomorphic partitioning storage method. In fact, both methods were implemented and are described below.

**Fixed Partitioning**

For the fixed partitioning method, let the value 0 represent assignment to the set \( X \) and let 1 represent assignment to the set \( Y \). For a single edge, either the endpoints are in the same partition or they are in opposite partitions. Given that there are two partitions, we have the four cases given in Figure 4.5.

<table>
<thead>
<tr>
<th>Solution</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>rank</td>
<td>( w_e )</td>
<td>( w_e )</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( a )</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>( b )</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>left</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>right</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
</tr>
</tbody>
</table>

**Figure 4.5** Table of leaf solutions when solving MAXIMUM CUT using the fixed partitioning storage type.

The proof that all feasible solutions are generated from these leaf solutions and the common join operation is similar to the proof for the independent set algorithm. In short, every feasible partition can be projected onto the subgraph defined at the leaves of the decomposition tree and each projection is equivalent to one of the leaf solutions. This collection of projections and their extensions obtained through joining will at each node be compatible with each other and will be joined to finally create the full solution at the root.
Isomorphic Partitioning

Using the isomorphic partitioning method, if one vertex is assigned the value 0 and another vertex the value 1, then we conclude that the vertices are in different sets. Again, as with the fixed partition method, for a single edge, either both endpoints are in the same set or they are in different sets. Since the sets observe the same property relative to their respective members, it is not necessary to differentiate between the ordered sets $(X,Y)$ and $(Y,X)$. The set of leaf solutions is depicted in Figure 4.6.

<table>
<thead>
<tr>
<th>Solution</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>rank</td>
<td>$w_e$</td>
<td>0</td>
</tr>
<tr>
<td>a</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>left</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>right</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

Figure 4.6 Table of leaf solutions when solving MAXIMUM CUT using the isomorphic partitioning storage type.

As we did with the previous proofs of the algorithms’ completeness, consider an arbitrary partitioning $s$ of the set of vertices. Projecting $s$ onto the middle sets of the children of some node $v$ will preserve the partitioning.

A solution based on isomorphic partitioning goes through two transformations: zero ordering before it is stored and intersection ordering before it is is tested against other solutions for compatibility. Each of these reorderings preserves the vertex partitioning. Thus, the two partial solutions will be found to be compatible and joined together.
Maximum Cut

The problem MAX CUT is to find the cut that maximizes the sum of the cut edges. Let \( f_1(z) \) return \( w_z \) if it exists and let \( m = 1 \), \( n = 3 \), and \( t = 0 \). Define \( \Psi \) to be always true and define

\[
F(|X|_1^G, |Y|_1^G, |Z|_1^G) = |Z|_1^G = \sum_{z \in Z} f_1(z) = \sum_{z \in Z} w_z.
\]

Then, the following linear EMS extremum problem expresses MAXIMUM CUT.

Maximize \( F(|X|_1^G, |Y|_1^G, |Z|_1^G) \)

subject to \( \Phi_G(X, Y, Z) \) \hspace{1cm} (4.5)

The maximum cut problem was implemented in Java with both solution storage types described in this section. Some sample running times from the fixed partitioning implementation are plotted in Figures 4.7, 4.8, and 4.9. Some sample running times from the isomorphic partitioning implementation are plotted in Figures 4.10, 4.11, and 4.12. The plots for both implementations suggest that the algorithms are linear in practice as we have proven in Section 3.5.

It is evident from the aforementioned plots and the sample results displayed in Table 4.3 that the fixed partitioning implementation is faster than the isomorphic partitioning. This is most easily explained by studying the trade-off between implementing intersection ordering and zero ordering for isomorphic partitioning versus processing twice as many solutions with fixed partitioning. Obviously, the former has a harsher effect.

4.4 Graph Bisection

GRAPH BISECTION is similar to MAXIMUM CUT in that the goal is to partition the vertices into two sets; however, with GRAPH BISECTION, we want to minimize the
Figure 4.7  Maximum Cut (Fixed Partitioning), branchwidth = 3

Figure 4.8  Maximum Cut (Fixed Partitioning), branchwidth = 4
Figure 4.9  Maximum Cut (Fixed Partitioning), branchwidth = 10

Figure 4.10  Maximum Cut (Isomorphic Partitioning), branchwidth = 3
Figure 4.11  Maximum Cut (Isomorphic Partitioning), branchwidth = 4

Figure 4.12  Maximum Cut (Isomorphic Partitioning), branchwidth = 10
\[
\begin{array}{|c|c|c|c|c|c|}
\hline
n & m & n + m & \text{Fixed Part.} & \text{Iso. Part.} & \text{filename} \\
\hline
23 & 71 & 94 & 1.6582 & 2.7058 & \text{myciel4.col.edg.hicks} \\
76 & 215 & 291 & 18.3080 & 28.8678 & \text{eil76.tsp.del.hicks} \\
148 & 265 & 413 & 2.6414 & 4.0726 & \text{m8.graph.hicks} \\
100 & 286 & 386 & 20.6364 & 32.4738 & \text{rd100.tsp.del.hicks} \\
101 & 290 & 391 & 18.0068 & 28.3649 & \text{eil101.tsp.del.hicks} \\
74 & 301 & 375 & 2.3102 & 4.3425 & \text{huck.col.edg.hicks} \\
130 & 377 & 507 & 22.7032 & 35.4486 & \text{ch130.tsp.del.hicks} \\
136 & 377 & 513 & 23.6372 & 36.9262 & \text{pr136.tsp.del.hicks} \\
159 & 431 & 590 & 18.9488 & 27.9080 & \text{u159.tsp.del.hicks} \\
150 & 436 & 586 & 27.3992 & 43.5940 & \text{kroB150.tsp.del.hicks} \\
654 & 1806 & 2460 & 83.0490 & 121.9534 & \text{p654.tsp.del.hicks} \\
\hline
\end{array}
\]

\textbf{Table 4.1} Table of running times for Maximum Cut using Fixed and Isomorphic Partitioning, branchwidth = 3, 50 iterations

sum of the cut edges and the partitions must be of equal size. Because of the latter of these requirements, we must state the problem as an EMS property. The MS property \(\Phi_G\) is exactly the same as defined in Statement 4.4. And if we define an evaluation \(f_2^G(x) = 1\) then the evaluation relation is

\[
\Psi(|X|_2^G, |Y|_2^G, |Z|_2^G) = |X|_2^G - |Y|_2^G = 0.
\]

Obviously, given sets \(X, Y,\) and \(Z,\) if \(\Phi_G(X, Y, Z)\) and \(\Psi(|X|_2^G, |Y|_2^G, |Z|_2^G)\) then \(\Phi_G(Y, X, Z)\) and \(\Psi(|Y|_2^G, |X|_2^G, |Z|_2^G)\). Thus, \textbf{GRAPH BISECTION} can be implemented using either the fixed partitioning or the isomorphic partitioning method.

Recall from Section 4.3, that for \textbf{MAXIMUM CUT}, the fixed partitioning method is the faster of the two. Given the similarity between \textbf{MAX CUT} and \textbf{GRAPH BISECTION}, we shall only implement the fixed partitioning method for Graph Bisection.

The set of leaf solutions is given in Figure 4.13. It is the same set as in Figure 4.5 for \textbf{MAX CUT}; however, the size of the partitions are appended to each solution.
<table>
<thead>
<tr>
<th>Solution</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>rank</td>
<td>w</td>
<td>w</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>size of $V_0$</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>size of $V_1$</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>$a$</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$b$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>left</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
</tr>
<tr>
<td>right</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
</tr>
</tbody>
</table>

**Figure 4.13** Table of leaf solutions when solving Graph Bisection using the fixed partitioning storage type.

The process of joining the partitions is as described in Section 4.1. In addition, we must maintain the sizes of the partitions. Recall that for a bipartition $\{V_0, V_1\}$ of $V$, we have $|V_0^{(l)} \cup V_0^{(r)}| = |V_0^{(l)}| + |V_0^{(r)}| - |V_0^{(l)} \cap V_0^{(r)}|$ where $V_0^{(l)} = V_0 \cap \text{mid}((v, l))$ and $V_0^{(r)} = V_0 \cap \text{mid}((v, r))$ and likewise for $V_1$. Therefore, maintaining sizes is accomplished by adding the sizes from the left and right partial solutions and then subtracting the number of vertices in the intersection of the active middle sets that are in the respective partitions.

After any joining we want the size of each partition to be less than or equal to $\frac{1}{2}|V|$. As we are processing interior nodes this ensures that we will not end of building unequal partitions. And at the root, it ensures that the size of the partitions in all the solutions are equivalent (if such a bipartition exists).

**Graph Bisection (Optimization)**

Graph Bisection is defined as an optimization problem. For us to extend the EMS formula for equal sized partitions, we define $f_1^G(z) = w_z$ for $z$ an edge of $G$. Thus,
we have the evaluation term

\[ F(|X|_1^G, |Y|_1^G, |Z|_1^G, |X|_2^G, |Y|_2^G, |Z|_2^G) = -|Z|_2^G = - \sum_{z \in Z} w_z \]

and the EMS extremum problem

Maximize \[ F(|X|_1^G, |Y|_1^G, |Z|_1^G, |X|_2^G, |Y|_2^G, |Z|_2^G) \]
subject to \[ \Phi_G(X, Y, Z) \]
\[ \Psi(|X|_1^G, |Y|_2^G, |Z|_2^G). \] (4.6)

Unfortunately, the algorithm described above for Graph Bisection is not linear. This is specifically because we are keeping track of the sizes of the partitions which is related directly to the size of the graph.

For further consideration, suppose two partial solutions are generated at an interior node and are locally identical in terms of the partition. If the two solutions are not also identical in the sizes of the partitions, then both solutions must be kept. Why? We do not have \textit{a priori} knowledge of how any current partial solution will be extended. So, because the validity or feasibility of the solutions depends on the local solution as well as the sizes of the partitions, there may exist a feasible extension using one set of partitioning sizes but not the other.

Of course, we expect that a linear-time algorithm using branch decompositions does exist; however, the simple augmentation of our \textsc{Maximum Cut} algorithm is not it.
Chapter 5

Graph Coloring: Minimum $k$ Vertex $k$-Partitioning

**Graph Coloring**—also known as **Chromatic Number**—is the problem of determining the smallest number of subsets into which the vertex set can be partitioned so that no two vertices in the same subset are adjacent.

Of course, our first objective is to prove that finding the chromatic number of a graph is expressible in monadic second-order logic. As with previous problems, we will first define the decision problem as a MS property and then show how the optimization problem is expressed as a linear EMS extremum problem.

We prove in Lemma 5.1 that graph $c$-colorability can be defined as an MS property $\phi$. Then, in Lemma 5.2 we prove that there exists a $c$ such that $\phi$ will always have an affirmative solution for any complete graph whose branchwidth is bounded above by a fixed $k$. We have not proven that this result extends to general graphs; however, this will not be an incumbrance to the success of our algorithm: it is based on isomorphic partitioning and does not explicitly use the value $c$.

**Lemma 5.1** Graph $c$-Colorability of a graph $G$ is expressible in monadic second-order logic for any fixed finite $c$.

**Proof** Let us consider the following statement.

$$
\Phi_G(X_1, \ldots, X_c) = \forall x \left\{ \bigvee_{1 \leq i \leq c} (x \in X_i) \land \bigwedge_{1 \leq i < j \leq c} \neg [(x \in X_i) \land (x \in X_j)] \right\} \land \forall x \forall y \forall z \left( \neg (x = y) \land \left\{ \bigvee_{1 \leq i \leq c} [(x \in X_i) \land (y \in X_i)] \right\} \rightarrow \neg \text{edg}_G(z, x, y) \right)$$

Informally, Statement 5.1 is read "there exist $c$ sets such that every $x$ is in exactly one set and for any triplet $x, y, z$ if $x \neq y$ and $x$ and $y$ are in the same set, then there
cannot be an edge \( z \) from \( x \) to \( y \)." This obviously is the definition of a \( c \)-coloring of a graph \( G \).

We now show that if \( k \) is known to be an upper bound for the branchwidth of a graph then there exists a fixed finite \( c \) such that Statement 5.1 can be satisfied.

**Lemma 5.2** Any graph \( G \) with branchwidth bounded above by \( k \) can be colored with no more than \( c = \max(\frac{3}{2}k, 2) \) colors. This bound is the best possible.

**Proof** Without loss of generality, we shall assume that \( \beta(G) = k \). For every vertex induced subgraph \( H \) of \( G \), we know that \( \min_{v \in V(H)} \delta(v) \leq tw(H) \). Thus, a greedy algorithm can be used to color \( G \) with at most \( c = \min_{v \in V(G)} \delta(v) + 1 \leq tw(G) + 1 \) colors. Applying Theorem 2.1 yields the desired result. By Theorem 2.3, this bound is the best possible.

Since we do not know of a fixed \( c \) such that every graph in the class of graphs with branchwidth bounded above by \( k \) is \( c \)-colorable, let us assume the bound in Lemma 5.2. We can use either fixed or isomorphic partitioning because each color or partition satisfies the same internal property—indeed independence. Ultimately, we will use only the latter; a brief justification is presented in the next section. For isomorphic partitioning, the upper bound \( c \) is not truly relevant because the solution states are created as they are needed.

### 5.1 Fixed Partitioning

If we know that a graph is colorable with at most \( c \) colors, then using fixed partitioning, the number of solutions stored at a single node \( v \) is bounded above by \( c^w \) where \( w = |\text{mid}(p, v)| \) and \( p \) is the parent of \( v \). This worst case occurs when the union of
the active middle sets at $v$ is an independent set in $g_v$. If we use the bound $c = \lceil \frac{3}{2}k \rceil$, the upper bound is actually $(\lceil \frac{3}{2}k \rceil)^w$. While this bound is indeed constant in $|G|$, it is too high for even graphs with small branchwidth.

To emphasize how large this number is, consider a graph with branchwidth 6. The upper bound is $(\lceil \frac{3}{2} \beta(G) \rceil)^w$ if no pair of vertices in the middle set are adjacent— for $\beta(G) = 6$, this comes to 531441 unique solutions to generate and store.

Clearly, from the preceding example, fixed partitioning is not a good choice although it does yield a linear algorithm. The alternative that we choose is isomorphic partitioning.

It is an interesting aside that the best case for graph coloring using fixed partitioning is actually the case when $g_v$ is a clique: the upper bound is $w! \left( \frac{144}{w} \right)$. Here, for a graph with branchwidth 6, the bound is 144.

5.2 Isomorphic Partitioning

Using isomorphic partitioning, at any node there are at most $\beta(G)$ states or partitions. So, there can be at most $\beta(G)!$ solutions. For the previous example with $\beta(G) = 6$, we get at most 720 solutions. For graph coloring the potential difference in the number of solutions using fixed versus isomorphic partitioning is sizable.

As noted above, the best case is when $g_v$ is a clique. For isomorphic partitioning, the number of stored solutions is one.

Leaf Solutions

In a feasible graph coloring no adjacent vertices can be assigned the same color. So the set of leaf solutions consists only of solutions with the ends of the edge assigned different colors. Using isomorphic partitioning we know that all such partial solutions are isomorphic; therefore, the set has the single member shown in Figure 5.1. The
leaf solution has rank 2 because two colors are required to color an edge.

Join Operations

When joining (or considering joining) two solutions, we first perform the intersection ordering in Algorithm 2. Then, the test for compatibility is exactly as we described in Section 4.1: the values assigned to vertices in the intersection of the active middle sets must be the same; the vertices outside this intersection are irrelevant. Once a pair of solutions are determined to be compatible, they are joined to form a new set of solutions that must each preserve membership of the color groups. That is, if $s_{v_1} \neq s_{v_2}$ in the one of the original solutions, then the relationship must be true in any extension of the solution. The inverse also must be true.

Recall from the original description of isomorphic partitioning in Section 3.3 that all vertices with the same assigned value can be considered members of the same set. Presently, we shall refer to these sets as color groups. Algorithm 4 determines the members of the color groups. The input for the algorithm is: the right solution $rs$; an array ISECTION indicating whether vertices in the union of the active middle sets come from the left ($-1$), right ($+1$), or both (0) middle sets; and USIZE the size of the intersection. When the algorithm is complete, the array CGROUP will contain the index of the color group to which each vertex from the right middle set belongs.
Algorithm 4 \textit{(findColorGroups)}

\begin{verbatim}
findColorGroups( rs, ISECTION )
    FIRSTOCC = new array of length USIZE
    CGROUP = new array of length USIZE
    for ( RPOS = 0; POS = 0; POS < USIZE; POS ++ )
        if ( ISECTION[POS] > -1 )
            if ( ISECTION[POS] == 0 && FIRSTOCC[rs[RPOS]] == -1 )
                FIRSTOCC[rs[RPOS]] = POS
                RPOS ++
        for ( RPOS = 0, POS = 0; POS < USIZE; POS ++ )
            if ( ISECTION[POS] > -1 )
                if ( FIRSTOCC[rs[RPOS]] == -1 )
                    FIRSTOCC[rs[RPOS]] = POS
                    RPOS ++
            if ( ISECTION[POS] == +1 )
                CGROUP[POS] = FIRSTOCC[rs[RPOS - 1]]
\end{verbatim}

In joining two partial solutions, the objective is to determine every partial solution for which all left and right color groups are preserved. Consider the problem of joining the following two partial solutions.

<table>
<thead>
<tr>
<th>vertex</th>
<th>left</th>
<th>vertex</th>
<th>right</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>a</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td>b</td>
<td>1</td>
</tr>
<tr>
<td>c</td>
<td>2</td>
<td>c</td>
<td>2</td>
</tr>
<tr>
<td>f</td>
<td>3</td>
<td>d</td>
<td>1</td>
</tr>
</tbody>
</table>
The vertices in the intersection (vertices $a$ and $b$) have the same values, so the pair of solutions are compatible. The following are several new partial solutions that are valid extensions of these solutions and preserve the color groups.

<table>
<thead>
<tr>
<th></th>
<th>$s_1$</th>
<th>$s_2$</th>
<th>$s_3$</th>
<th>$s_4$</th>
<th>$s_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>$b$</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>$c$</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>$d$</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>$e$</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>$f$</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>

Notice that vertices $b$ and $d$ are always assigned the same color as required by the right solution, and, for example, that vertices $e$ and $f$ are never assigned the same color as required by the left solution.

When storing these or any solutions that use the isomorphic partitioning method, we must zero order the solutions. After doing so, we see that solutions $s_2$, $s_3$, and $s_5$ are equivalent and therefore redundant. An obvious question is whether these are all the feasible extensions that a join operation should generate and whether an algorithm exists to generate all of the solutions.

Before explaining the join procedure, we must recognize that we can fix the values assigned to the vertices from one partial solution and any variation will occur among the other values. Fixing some values based on one of the solutions is allowed because we know that doing so will in no way disallow the preservation of color groups. In addition, because we are using isomorphic partitioning, the actual number assigned to the vertex is itself arbitrary. Color groups that can be varied are considered free. Without loss of generality, let us assume that the fixed values are from the left solution and that the free color groups are determined by the right solution.
In the example above, notice that fixing the values assigned from the left solution leaves only one free color group from the right. That free color cannot be numbered 0 or 1 because of the aforementioned goal of preserving color groups; however, it can have the values 2 or 3 meaning that it happens to be in the same color group as $e$ or $f$, respectively—color groups do not transcend middle sets. The free color group could also have the value 4 meaning it is not in any of the established color groups. The following set of solutions is generated.

\[
\begin{array}{c|ccc}
 & s_1 & s_2 & s_3 \\
\hline
 a & 0 & 0 & 0 \\
b & 1 & 1 & 1 \\
c & 2 & 3 & 4 \\
d & 1 & 1 & 1 \\
e & 2 & 2 & 2 \\
f & 3 & 3 & 3 \\
\end{array}
\]

Now suppose that the right middle set has another vertex, say $g$, whose assigned value is 3. In this case, the right solution has two free color groups. Again, each group could be assigned the values 2 or 3 but they could also be assigned the colors 4 or 5 if neither free color group is matched with an existing color. Of course, the color groups still cannot be assigned the same value simultaneously. The permutations of these values yields the following set of colors for $c$ and $g$.

\[
\begin{array}{c|cccccccccc}
c & 2 & 2 & 2 & 3 & 3 & 3 & 4 & 4 & 4 & 5 & 5 \\
g & 3 & 4 & 5 & 2 & 4 & 5 & 2 & 3 & 5 & 2 & 3 \\
\end{array}
\]

In the following recursive algorithm for generating the permutations of the free color groups, assume the following values: $intersectHi$ = the highest index assigned to a vertex in the intersection (after intersection ordering); $leftHi$ = the highest
index assigned to a vertex in the left solution; rightHi = the highest index in the right solution; s1 = the left solution; s2 = the right solution; and ISECTION = the intersection array as described in Algorithm 4.

Algorithm 5

for each vertex in the left middle set
\[ s[v] = s_1[v] \]
findColorGroups( s2, ISECTION )
joinColoringSolutions( s, -1 )

FUNCTION joinColoringSolutions( s, cg )
if there is a color group after cg in CGROUP
\[ cg = cg + 1 \]
\[ loBound = isectHi + 1 \]
\[ upBound = leftHi + (rightHi - isectHi) + cg \]
for ( c = loBound, c ≤ upBound, c + + )
if c is a unique unused color
for all vertices v in color group cg
\[ s[v] = c \]
\[ s^* = s \]
joinColoringSolutions( s^*, cg )
else
store solution s

Note that Algorithm 5 does generate some redundant solutions that will be discovered in the zero ordering process. However, we are guaranteed to generate all feasible
extensions and only feasible extensions. In fact, we can calculate the exact number of unique solutions that will be generated.

**Theorem 5.1** Let solution $s_1$ have $l$ free color groups, solution $s_2$ have $r$ free color groups and $S = \text{join}(s_1, s_2)$. Then

$$|S| = \sum_{i=0}^{r-1} \left[ \binom{r}{i} \frac{l!}{(l-(r-i))!} \right] + 1. \quad (5.2)$$

**Proof** Without loss of generality, when counting free colors, we can assume that the intersection of the two middle sets is empty. If this intersection were not empty, the index of the colors simply would be shifted by the highest color assigned to a vertex in the intersection. We can also assume that $l$ is greater or equal to $r$ because if this is not true, then we simply can switch the partial solutions around without consequence. This latter assumption is relevant to ensuring that the calculations are not undefined.

In order to determine how many solutions are created, we must consider the different types of solutions and how many of each is created.

- Solutions for which $c_1, \ldots, c_r$ are contained in the set $\{1, \ldots, l\}$. These are the solutions for which no new color is introduced. There are $\prod_{i=0}^{l-1} (r-i) = \frac{l!}{(l-r)!}$ of them.

- Solutions with exactly one new color introduced. For these solutions, the new color can be in any of the $r$ places and the other $r-1$ places must be chosen from the set $\{1, \ldots, r\}$. There are $r \cdot \frac{l!}{(l-(r-1))!}$ of these solutions.

- Solutions with $k$ new colors introduced where $2 \leq k \leq r$. First, let us observe that the new colors must be in increasing order; otherwise, there will be redundant solutions. Thus, the number of combinations of color groups that can
have new colors is $\binom{r}{k}$. For each of these combinations, there are \( \frac{l!}{(l-(r-k))!} \) solutions. So, there are $\binom{r}{k} \cdot \frac{l!}{(l-(r-k))!}$ solutions with \( k \) new colors.

Recognizing that \( 1 \equiv \binom{r}{0} \) and \( r = \binom{r}{r} \), then we get the total number of solutions as stated in the theorem.

As explained in the proof of Theorem 5.1, redundant solutions are generated in the join procedure when the index of a new color is skipped or when new colors are not assigned to the vertices in increasing order. Since we can pinpoint where the redundant solutions occur, they can be avoided at the cost of increased overhead in the join procedure.

### 5.3 Algorithmic Complexity and Computational Results

When determining the complexity of the algorithm described in this chapter, we note that the input for all the subroutines is measurable as a function of the width of the decomposition and is in no way affected by the size of the graph. For the worst case described in Section 5.1 the number of possible solutions is exponential indeed in the width; however, given a fixed \( k \), it is constant regardless of the size of the graphs.

The optimal graph coloring problem was implemented in Java using the same set of graphs used in Chapter 4. Some sample running times are plotted in Figures 5.2 and 5.3. As expected, these plots suggest that the algorithm is indeed linear with respect to the branchwidth of the graphs.
Figure 5.2  Graph Coloring (Isomorphic Partitioning), branchwidth = 3

Figure 5.3  Graph Coloring (Isomorphic Partitioning), branchwidth = 4
Chapter 6

Path Partitioning

Whereas in the two preceding chapters we examined vertex partitioning problems, we now will focus on edge partitioning. In particular, we will look at two classic problems—HAMILTONIAN CYCLE and DISJOINT PATHS—whose solutions are sets of edges that exhibit a given property.

The Hamiltonian Cycle problem is that of finding a cycle (or closed path) through all the vertices of a graph. Given a set of $d$ distinguished vertex pairs, the disjoint paths problem is that of finding $d$ paths such that they are mutually vertex-disjoint and for each distinguished pair of vertices there is a path that connects the endpoints.

A solution for a path partitioning problem is a subset of the set of edges. For solutions to vertex partitioning problems, we let the value assigned to each vertex represent to which subset or partition of the vertices the vertex in question belongs. Unfortunately, for branch and tree decompositions the only local information that is explicitly available at an interior node of the decomposition is the set of vertices in the active middle sets. We know neither which pairs of vertices are adjacent nor any edges incident to the middle set vertices. Therefore, adapting the previous approach is not an option. That is, we cannot assign values to the edges because without redefining our decomposition tree, we do not know what edges are in the current subgraph.

We now define a usable solution storage method for these path partitioning problems. As stated in Section 3.3, this storage method is not a new data structure but rather a new interpretation of the values assigned in the data structure.
6.1 Solution Storage

Let us suppose that in some partial solution a vertex that is no longer in an active middle set is not interior to a path and (for disjoint paths) is not a distinguished endpoint. Since by Lemma 3.1 we know that the algorithm will never come across that vertex again, we know that that vertex will never be made part of a Hamiltonian cycle or a disjoint path from one distinguished endpoint to another. Thus, it is not necessary to store such a partial solution.

In the following descriptions of the algorithms, let us assume that the stored partial solutions do not have the aforementioned property. Also, let \((T, \tau)\) be a rooted branch decomposition with root \(r\) of the graph \(G\), and let \(s\) be a feasible solution to a path partitioning problem in \(G\). Before we discuss the leaf and join operations, we first introduce a practical approach to storing a partial solution.

Recall that a subtree of the rooted branch decomposition defines a subgraph of \(G\) that has a set of partial solutions. Every partial solution is a set of paths with both ends in the union of the active middle sets or any end not in the active middle sets is a distinguished vertex. Although we would prefer to not introduce new terminology, defining the concept of a reduced partial solution is helpful in describing our stored solutions. For a partial solution \(s\), the reduced partial solution is a graph whose vertex set is the vertices of the active middle set and who edge set is defined such that two vertices are adjacent if and only if there is a path between them in \(s\). (Figure 6.1 shows an example.)

Instead of showing the complete partial solution, we can store the reduced partial solution without losing any information about connectivity. In this respect, we can think of a stored solution as a list describing each vertex's connectivity in the reduced partial solution. We use the following designations:
Figure 6.1 A partial solution and its reduced partial solution. The vertices within the dashed box comprise the active middle set.

- free the vertex is isolated;
- open[w] the vertex is one end of a path, where w is the other end; and
- full the vertex is in the interior of a path.

Figure 6.1 also shows the solution using these vertex labels.

Below, we examine how the (reduced) partial solutions are generated for the two problems. While the set of leaf solutions are the same, for Hamiltonian cycle and disjoint paths, there is a difference in the join operations that occurs in validating the new partial solution. To reduce the sources of confusion, we will study the join operation in the context of the Hamiltonian cycle problem and later explain the differences for the disjoint paths problem.

Leaf Solutions

For the leaf v with \( \tau(v) = \{a, b\} \), we have \( S_v = \{\{a, b\}, \emptyset\} \). That is, each edge of \( G \) is either in \( S \) or not. In our implementation, \( S_v \) is stored as shown in Figure 6.2.
<table>
<thead>
<tr>
<th>Solution</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>rank</td>
<td>w</td>
<td>0</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>FREE</td>
</tr>
<tr>
<td>b</td>
<td>a</td>
<td>FREE</td>
</tr>
<tr>
<td>left</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>right</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 6.2 Table of leaf solutions when solving Hamiltonian Cycle or Disjoint Paths.

6.2 Hamiltonian Cycle

We begin as in previous chapters with a proof that Courcelle’s Theorem can be applied to this problem.

Theorem 6.1 The Hamiltonian Cycle problem is a monadic second-order problem.

Proof Consider the following MS statement.

\[
\Phi_G(W) = \forall v \{ v \in V_G \}
\]

\[
\Rightarrow \{ \exists y \exists z [ y \in V_G \land z \in V_G \land \neg [(v = y) \lor (y = z) \lor (v = z)]]
\]

\[
\Rightarrow [\exists a \exists b (\neg (a = b) \land (a \in W) \land (b \in W)
\]

\[
\land \text{edg}_G(a, v, y) \land \text{edg}_G(b, v, z)]
\]

\[
\land \forall x [ \neg [(x = v) \lor (x = y) \lor (x = z)]
\]

\[
\Rightarrow \neg [\exists c (c \in W \land \text{edg}_G(c, v, x))] \}\}
\]

\[
\land \forall S \{ \forall v (v \in S \Rightarrow v \in V_G) \}
\]

\[
\Rightarrow \{ [\exists v [(v \in V_G) \land \neg (v \in S)]]]
\Rightarrow
\]

\[
[\exists a \exists b \ [\neg (a = b) \land (a \in W) \land (b \in W)]
\]

\[
\land \exists v [(v \in S) \land \exists y (\neg (y \in S) \land \text{edg}_G(a, v, y))]
\]

\[
\land \exists v [(v \in S) \land \exists y (\neg (y \in S) \land \text{edg}_G(b, v, y))] \}\}
\]

(6.1)
The statement in (6.1) is true for a given graph $G$ if there exists a set $W \subseteq E$ such that for every vertex $v$, there exist exactly two edges $a$ and $b$ incident to $v$ in $W$; and if every proper subset $S$ of vertices has at least two edges $a$ and $b$ each having only one end incident to a vertex in $S$. The MS statement is derived from the linear programming relaxation of the traveling salesman problem $\min \{ w^T x : \forall v \in V, \delta(v) = 2; \forall S \subset V, \delta(S) \geq 2, 0 \leq x \leq 1 \}$ where $\delta(S)$ is the number of edges with exactly one end in $S$. By inspection, one can notice that the first block of linear programming constraints and the first condition in (6.1) above correspond to the so-called vertex degree constraints and the second pair to the so-called subtour elimination constraints. Because set inclusion is discrete, the constraint $0 \leq x \leq 1$ is implicitly included in the logical statement. We can conclude that this MS statement is correct given the correctness of the LP relaxation.

Join Operations

For Hamiltonian cycle, joining two reduced partial solutions is a process of sequentially comparing connectivity at each vertex in the intersection of the active middle sets and determining if the join will violate the restriction on subtours or create a vertex whose degree in the solution is greater than two. Table 6.1 shows the set of possible

<table>
<thead>
<tr>
<th></th>
<th>free</th>
<th>open$[w_1]$</th>
<th>full</th>
</tr>
</thead>
<tbody>
<tr>
<td>free</td>
<td>free</td>
<td>open$[w_1]$</td>
<td>full</td>
</tr>
<tr>
<td>open$[w_2]$</td>
<td>open$[w_2]$</td>
<td>incompatible if $w_1 = w_2$</td>
<td>incompatible</td>
</tr>
<tr>
<td>full</td>
<td>full</td>
<td>incompatible</td>
<td>incompatible</td>
</tr>
</tbody>
</table>

Table 6.1 Possible combinations of partial solutions at a single vertex of the active subgraph
combinations of partial solutions if we consider only one vertex of the active subgraph. This same set of possibilities is also depicted in Figure 6.3.

Figure 6.4 shows the result of joining the two indicated reduced partial solutions at some interior node. In this case, the union and the intersection of the active middle sets is $v_1$, $v_2$, $v_3$, and $v_4$. Since the two sets of paths do not join to form a cycle (or vertex with degree greater than two), they are compatible and form the new path.

The vertex degree violation occurs when a vertex is full in one solution and not free in the other; it is easy to identify. However, when joining at a vertex that is neither open in both solutions, it is not immediately obvious whether the solutions create a subtour. Figure 6.5 sheds light on this situation; it is possible that vertex $v_1$ becomes interior to a path from $v_2$ to $v_3$ however $v_2$ and $v_3$ are themselves interior.
to different paths from \( v_1 \) to \( v_4 \). To discover incompatibilities such as this, we must implement Algorithm 6 that extends the path in either direction based on the partial solutions until a cycle is created or the end of the path is reached.

At the start of the extendPath algorithm (Algorithm 6) the solution is only partially defined by the join procedure up to the calling of the algorithm. The algorithm first extends the path in the direction indicated by the right solution \( s_r \), until a collision occurs—the vertex is full in one solution and open in the other, a subtour or cycle is created, or the path ends. Then, if there is a left path specified in \( s_1 \)—the test if (fromEnd! = center) in the algorithm—the path is extended in the other direction. Of course, in this direction, the only possible outcomes are a collision or a natural end to the path. In Algorithm 6, let \( p(v) \) return the position in the union of the vertex \( v \) and processed is an array whose length is equal to the size of the union of the active middle sets and whose elements are initialized to false.

Algorithm 6

```plaintext
extendPath( fromEnd, toEnd, center, s, s_l, s_r, processed )

s[p(center)] = toEnd

newEnd = s_l[p(toEnd)] // continuing path from toEnd

if ( newEnd == FREE )
```
rightEnd = toEnd // path stops at toEnd
s[p(toEnd)] = fromEnd
processed[ p(toEnd) ] = true
else if (newEnd == FULL)
    return COLLISION
else
    if (fromEnd == newEnd)
        if (at root of decomposition tree)
            rightEnd = fromEnd
            s[p(toEnd)] = full, processed[ p(toEnd) ] = true
            s[p(center)] = full, processed[ p(center) ] = true
            s[p(fromEnd)] = full, processed[ p(fromEnd) ] = true
            return SUBTOUR
        else
            s[p(toEnd)] = full
            processed[ p(toEnd) ] = true
            rightEnd = extendPath(fromEnd, newEnd, fromEnd,
                                  s, s_r, s_l, processed)
            if (rightEnd == COLLISION)
                return COLLISION
            else if (rightEnd == SUBTOUR)
                return SUBTOUR
        if (fromEnd != center)
            result = extendPath( rightEnd, fromEnd, rightEnd,
                                  s, s_r, s_l, processed)
            s[p(center)] = full
processed[ p(center) ] = true
if ( result == COLLISION )
    return COLLISION
return rightEnd

Although the extendPath algorithm is longer than the others presented in this thesis, by inspection, we can see that it visits each vertex in the intersection of the active middle sets at most once. Moreover, recall that in the actual join procedure we only use the extendPath algorithm if a vertex is open in both partial solutions. In all other cases, we use the outcome indicated in Table 6.1 and Figure 6.3.

The last step is to validate the solutions before they are stored. A solution that is not eliminated because it does not violate the vertex degree or subtour properties can be eliminated if a vertex that is leaving the active middle sets does not have degree two.

The running time of the join procedure is a function of the width of the decomposition and is constant with respect to the size of the graph. Therefore, the Hamiltonian cycle algorithm described in this chapter is linear in the size of the graph.

6.3 The Traveling Salesman Problem

The optimization version of the Hamiltonian cycle problem is the well-studied Traveling Salesman Problem wherein we must identify a solution that minimizes the sum of the edge weights. To state the TSP as an MS extremum problem we define $f^G_1(z) = w_z$ for $z$ an edge of $G$. Thus, we have the evaluation term

$$F(|W|G) = -|W|G = -\sum z \in W w_z.$$
And with the evaluation relation $\Psi$ identically true we have the linear EMS extremum problem

$$\begin{align*}
\text{Maximize} & \quad F(\|W\|^G_1) \\
\text{subject to} & \quad \Phi_G(W) \\
& \quad \Psi(W).
\end{align*}$$

(6.2)

We implemented the optimization variant of the algorithm to find a Hamiltonian cycle in Java. Random graphs were generated using the edgegen application from the CONCORDE [1] package and decomposed using Hicks's algorithm. Some sample running times are plotted in Figures 6.6, 6.7, and 6.8.

![Graph](image)

**Figure 6.6** Hamiltonian Cycle, branchwidth = 5

### 6.4 Disjoint Paths

Let $d$ be a finite integer and for $i = 1, 2, \ldots, d$, let $D_i$ be the sets of distinguished node pairs. The $d$-DISJOINT PATHS problem can be paraphrased as "for any $i$ between
Figure 6.7  Hamiltonian Cycle, branchwidth = 6

Figure 6.8  Hamiltonian Cycle, branchwidth = 7
and \(d\) it is true that for every \(x\), \(x\) can belong to at most one set \(W_i\); if \(x\) is a distinguished vertex in set \(D_i\) then \(x\) is in partition \(W_i\) and there exists an edge \(a \in W_i\) between \(x\) and \(y\) and no other edge in \(W_i\) incident to \(x\); and if \(x\) is in the partition \(W_i\) but not a distinguished vertex then \(x\) is adjacent to exactly two distinct vertices in \(W_i\)." It can be expressed as the MS property in Statement 6.3.

\[
\Phi_G(W_1, W_2, \ldots, W_d, D_1, D_2, \ldots, D_d) =
\bigwedge_{1 \leq i \leq d} \forall x \left[ (x \in W_i) \Rightarrow \neg \bigvee_{j \neq i} (x \in W_j) \right]
\land (x \in D_i) \Rightarrow (x \in W_i)
\land \exists y \exists a \left( (y \in W_i) \land (a \in W_i) \land \neg (x = y) \right)
\land \exists b \left( \neg \exists z \exists b \left[ (z \in W_i) \land (b \in W_i) \land \neg (x = z) \land \neg (y = z) \land \exists \text{edg}_G(a, x, y) \land \exists \text{edg}_G(b, x, z) \right] \right)
\land \left( (x \in W_i) \land \neg \bigvee_{1 \leq i \leq d} (x \in D_i) \right)
\Rightarrow \exists y \exists a \exists z \exists b \left[ (y \in W_i) \land (a \in W_i) \land (z \in W_i) \land (b \in W_i) \land \neg [(x = y) \lor (x = z) \lor (y = z)] \land \exists \text{edg}_G(a, x, y) \land \text{edg}(b, x, z) \lor \exists w \{ \neg [(w = x) \lor (w = y) \lor (w = z)] \land \exists c [(c \in W_i) \land \text{edg}(c, x, w)] \right]
\end{align}

The constant \(d\) is arbitrary although, of course, in all instances it must be less than or equal to the number of vertices in the instance graph. Thus, we have shown that the \(d\)-disjoint paths problem is an MS problem for any \(d\) and by Courcelle’s theorem, we know that a linear-time branch decomposition based algorithm exists to find a set of \(d\) disjoint paths.
The similarities of finding a set of disjoint paths in a graph and finding a Hamiltonian cycle are that solutions can be stored using the same method and we can use the extendPath algorithm (Algorithm 6) in both cases. **Disjoint Paths** is most significantly different in the solution validation step:

- non-distinguished vertices can leave the middle set if their degree is either zero or two;
- distinguished vertices can only leave the middle set if their degree in the partial solution is exactly one; and
- a distinguished vertex must be connected by a path to the other distinguished vertex of the pair.

These differences are easy to accommodate. For the first two conditions, when validating the solution, we simply test for the appropriate degrees on all exiting vertices. If an exiting distinguished vertex is the end of a path then we determine the vertex at the other end of the path and if it is not the exiting vertex’s partner, we validate that the other end is not another distinguished vertex.

An added benefit of using dynamic programming to solve the disjoint paths problem is that as solutions are created, it takes constant times to calculate the rank of the solution. In this case, we used the total edge weights of the paths as the rank. Consequently, the algorithm determines the minimum weight disjoint paths.

We implemented the **Disjoint Paths** algorithm in Java. Unfortunately, we lack a significant set of test graphs to get empirical data regarding running times. Despite the absence of actual running times, we can be sure that the algorithm to find disjoint paths is linear because it extends the **Hamiltonian Cycle** algorithm and its running time by processes that require constant time.
Chapter 7

Parallel Algorithms

Various aspects of decomposition based algorithms can be parallelized easily and naturally because of the limited and predictable interaction between branches of the tree or between solutions. In particular, in this chapter, we examine a multithreaded procedure for building the sets of solutions at interior nodes. Within concurrent threads, we handle testing compatibility, joining, and storing the result from pairs of solutions. One feature that makes this multithreaded scheme attractive is that it is only a slight modification to the original algorithmic framework. In fact, it is possible to use either the serial or parallel algorithm with absolutely no changes to the problem specific extensions.

Recall that joining a pair of solutions is independent from joining any other pair. Moreover, the join process does not alter the original left or right solution. In theory, one could perform each of the joins in a separate thread or process. However, in practice, if we create more processes than the finite number of processors available, these processes will compete with each other for system resources and possibly worsen rather than improve the efficiency of our algorithm. Instead of creating and running a burdensome number of threads, we create one thread for each processor and join several pairs of solutions per thread.

As in the general framework presented in Chapter 3 (Algorithm 1), let \( S_2 \) be the set of right solutions at some node of the decomposition tree. Also, let \( p \) be the number of processors available. Given the \( |S_2| \) right solutions, we calculate \( spt = \lfloor |S_2|/p \rfloor + 1 \) as the number of right solutions per thread. Then, we separate the set \( S_2 \) into blocks of solutions so that each block except the last has \( spt \) solutions; the last block has
$|S_2| - (p - 1) \times spt$ solutions. With this partitioning of the set of right solutions and a fixed left solution, we create a thread to join the left solution with all the compatible solutions in a right partition. We should note here that if $|S_2|$ is small relative to $p$, it is not practical to partition the set into blocks. In our implementation, we use the simple condition that $|S_2| > 2p$ in order to multithread.

7.1 ThreadLock Semaphore

Of course, since the set of solutions at a parent node is dependent on the solutions at the node’s children, we must wait for all the join threads to finish before leaving the node. In our implementation, the algorithm waits for all the threads for a given left solution to terminate before choosing the next left solution and generating threads to join it with the solutions of the right blocks.

A common method to ensure that an algorithm waits for a group of threads to finish before continuing is the use of a semaphore. A semaphore is simply an object that reports itself busy until all the associated threads have terminated. To create a semaphore, we use an object we call a ThreadLock which contains an integer counter. When a thread is registered with the ThreadLock, the counter is incremented. When the thread is delisted—presumably, when it completes its work, the counter is decremented. The main thread blocks until the ThreadLock’s counter is zero and it reports itself to be no longer busy.

Due to the unpredictability of multithreading (i.e., not knowing exactly when a thread will be started), it is safest to create, register, and start all the join threads before the main thread begins to block. We accomplish this by implementing the following algorithm.

Algorithm 7
let $S_1 \equiv$ Solutions stored at left child of $t$
let $S_2 \equiv$ Solutions stored at right child of $t$
let tl be a ThreadLock
for each solution $s_1$ of $S_1$
  for each block of right solutions $rb_t$ in $S_2$
    threads[ t ] = new SolutionJoiner( $s_1$, $rb_t$, tl )
  for $t$ between 0 and $p$
    start thread threads[ t ]
  while( tl is Busy )
    wait for resume notification

In our implementation of Algorithm 7, a SolutionJoiner—a subclass of the Java Thread object—registers itself with the ThreadLock when instantiated; and, when started, it tests compatibility of $s_1$ and the individual solutions in $rb_t$, calls the join operation—introduced in Algorithm 1 and defined in the previous chapters for the individual graph problems—when solutions are compatible, stores any created solutions, and finally delists itself from the ThreadLock when all the solutions in $rb_t$ have been considered.

Tables 7.1, 7.2, and 7.3 show the running times of this multithreaded implementation compared to that of the single-threaded implementation for the MAXIMUM INDEPENDENT SET, MAX CUT, and GRAPH COLORING problems, respectively. This is the same set of graphs used to test the serial algorithms in the previous chapters. The speed-up is calculated as the running time for the single-threaded implementation divided by that of the multithreaded. For this set of graphs, the average speed-up of the INDEPENDENT SET algorithm for the multithreaded algorithm (using four threads) is 0.4961 and is at most 0.6751. For MAXIMUM CUT the average speed-
<table>
<thead>
<tr>
<th>$\beta$</th>
<th>$n + m$</th>
<th>1 thread run time (sec)</th>
<th>4 threads run time (sec)</th>
<th>speed-up</th>
<th>graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>20</td>
<td>0.1330</td>
<td>0.1970</td>
<td>0.6751</td>
<td>cube3.hicks</td>
</tr>
<tr>
<td>4</td>
<td>29</td>
<td>0.1350</td>
<td>0.2250</td>
<td>0.6000</td>
<td>bmuryt.hicks</td>
</tr>
<tr>
<td>4</td>
<td>82</td>
<td>0.2140</td>
<td>0.3910</td>
<td>0.5473</td>
<td>m4.graph.hicks</td>
</tr>
<tr>
<td>4</td>
<td>126</td>
<td>0.2290</td>
<td>0.3670</td>
<td>0.6240</td>
<td>ma.graph.hicks</td>
</tr>
<tr>
<td>4</td>
<td>168</td>
<td>0.2720</td>
<td>0.4640</td>
<td>0.5862</td>
<td>...trans...output.graph.hicks</td>
</tr>
<tr>
<td>4</td>
<td>236</td>
<td>0.3280</td>
<td>0.6640</td>
<td>0.4940</td>
<td>mf.graph.hicks</td>
</tr>
<tr>
<td>4</td>
<td>255</td>
<td>0.3120</td>
<td>0.6920</td>
<td>0.4509</td>
<td>ml.graph.hicks</td>
</tr>
<tr>
<td>5</td>
<td>31</td>
<td>0.1530</td>
<td>0.2970</td>
<td>0.5151</td>
<td>myciel3.col.edg.hicks</td>
</tr>
<tr>
<td>5</td>
<td>141</td>
<td>0.2510</td>
<td>0.4550</td>
<td>0.5516</td>
<td>tele56.85.edg.hicks</td>
</tr>
<tr>
<td>5</td>
<td>209</td>
<td>0.3150</td>
<td>0.6400</td>
<td>0.4922</td>
<td>md.graph.hicks</td>
</tr>
<tr>
<td>5</td>
<td>345</td>
<td>0.4510</td>
<td>1.0450</td>
<td>0.4316</td>
<td>bico133.hicks</td>
</tr>
<tr>
<td>6</td>
<td>194</td>
<td>0.3210</td>
<td>0.8550</td>
<td>0.3754</td>
<td>mb.graph.hicks</td>
</tr>
<tr>
<td>6</td>
<td>390</td>
<td>0.8440</td>
<td>2.0410</td>
<td>0.4135</td>
<td>pr107.tsp.del.hicks</td>
</tr>
<tr>
<td>6</td>
<td>514</td>
<td>0.4940</td>
<td>1.4430</td>
<td>0.3423</td>
<td>tele226.288.edg.hicks</td>
</tr>
</tbody>
</table>

Table 7.1 Comparison of running times and speed-up for single- and multi-threaded implementations of Maximum Independent Set algorithm

<table>
<thead>
<tr>
<th>$\beta$</th>
<th>$n + m$</th>
<th>1 thread run time (sec)</th>
<th>4 threads run time (sec)</th>
<th>speed-up</th>
<th>graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>20</td>
<td>0.1270</td>
<td>0.2230</td>
<td>0.5695</td>
<td>cube3.hicks</td>
</tr>
<tr>
<td>4</td>
<td>29</td>
<td>0.1560</td>
<td>0.2620</td>
<td>0.5954</td>
<td>bmuryt.hicks</td>
</tr>
<tr>
<td>4</td>
<td>82</td>
<td>0.2210</td>
<td>0.4840</td>
<td>0.4566</td>
<td>m4.graph.hicks</td>
</tr>
<tr>
<td>4</td>
<td>126</td>
<td>0.2180</td>
<td>0.4120</td>
<td>0.5291</td>
<td>ma.graph.hicks</td>
</tr>
<tr>
<td>4</td>
<td>168</td>
<td>0.2540</td>
<td>0.5180</td>
<td>0.4903</td>
<td>...trans...output.graph.hicks</td>
</tr>
<tr>
<td>4</td>
<td>236</td>
<td>0.3550</td>
<td>0.7020</td>
<td>0.5057</td>
<td>mf.graph.hicks</td>
</tr>
<tr>
<td>4</td>
<td>255</td>
<td>0.3400</td>
<td>0.7390</td>
<td>0.4601</td>
<td>ml.graph.hicks</td>
</tr>
<tr>
<td>5</td>
<td>31</td>
<td>0.1630</td>
<td>0.3350</td>
<td>0.4866</td>
<td>myciel3.col.edg.hicks</td>
</tr>
<tr>
<td>5</td>
<td>141</td>
<td>0.2700</td>
<td>0.5150</td>
<td>0.5243</td>
<td>tele56.85.edg.hicks</td>
</tr>
<tr>
<td>5</td>
<td>209</td>
<td>0.3240</td>
<td>0.6860</td>
<td>0.4723</td>
<td>md.graph.hicks</td>
</tr>
<tr>
<td>5</td>
<td>345</td>
<td>0.5330</td>
<td>1.2310</td>
<td>0.4330</td>
<td>bico133.hicks</td>
</tr>
<tr>
<td>6</td>
<td>194</td>
<td>0.3650</td>
<td>0.9670</td>
<td>0.3775</td>
<td>mb.graph.hicks</td>
</tr>
<tr>
<td>6</td>
<td>390</td>
<td>1.1930</td>
<td>3.1130</td>
<td>0.3832</td>
<td>pr107.tsp.del.hicks</td>
</tr>
<tr>
<td>6</td>
<td>514</td>
<td>0.5260</td>
<td>1.6940</td>
<td>0.3105</td>
<td>tele226.288.edg.hicks</td>
</tr>
</tbody>
</table>

Table 7.2 Comparison of running times and speed-up for single- and multi-threaded implementations of Maximum Cut algorithm
<table>
<thead>
<tr>
<th>$\beta$</th>
<th>$n + m$</th>
<th>1 thread run time (sec)</th>
<th>4 threads run time (sec)</th>
<th>speed-up</th>
<th>graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>20</td>
<td>0.1400</td>
<td>0.1550</td>
<td>0.9032</td>
<td>cube3.hicks</td>
</tr>
<tr>
<td>4</td>
<td>29</td>
<td>0.1560</td>
<td>0.2010</td>
<td>0.7761</td>
<td>bmurty.hicks</td>
</tr>
<tr>
<td>4</td>
<td>82</td>
<td>0.2290</td>
<td>0.3850</td>
<td>0.5948</td>
<td>m4.graph.hicks</td>
</tr>
<tr>
<td>4</td>
<td>126</td>
<td>0.2410</td>
<td>0.3170</td>
<td>0.7603</td>
<td>ma.graph.hicks</td>
</tr>
<tr>
<td>4</td>
<td>168</td>
<td>0.2770</td>
<td>0.3920</td>
<td>0.7066</td>
<td><em>.trans</em>.output.graph.hicks</td>
</tr>
<tr>
<td>4</td>
<td>236</td>
<td>0.3120</td>
<td>0.5230</td>
<td>0.5966</td>
<td>mf.graph.hicks</td>
</tr>
<tr>
<td>4</td>
<td>255</td>
<td>0.3400</td>
<td>0.5220</td>
<td>0.6513</td>
<td>m1.graph.hicks</td>
</tr>
<tr>
<td>5</td>
<td>31</td>
<td>0.1880</td>
<td>0.2210</td>
<td>0.8507</td>
<td>myciel3.col.edg.hicks</td>
</tr>
<tr>
<td>5</td>
<td>141</td>
<td>0.2760</td>
<td>0.4530</td>
<td>0.6093</td>
<td>tele56.85.edg.hicks</td>
</tr>
<tr>
<td>5</td>
<td>209</td>
<td>0.3200</td>
<td>0.5750</td>
<td>0.5565</td>
<td>md.graph.hicks</td>
</tr>
<tr>
<td>5</td>
<td>345</td>
<td>1.1060</td>
<td>0.8350</td>
<td>1.3246</td>
<td>bico133.hicks</td>
</tr>
<tr>
<td>6</td>
<td>194</td>
<td>0.6530</td>
<td>1.5130</td>
<td>0.4316</td>
<td>mb.graph.hicks</td>
</tr>
<tr>
<td>6</td>
<td>390</td>
<td>36.3160</td>
<td>6.8390</td>
<td>5.3101</td>
<td>pr107.tsp.det.hicks</td>
</tr>
<tr>
<td>6</td>
<td>514</td>
<td>2.7120</td>
<td>2.2100</td>
<td>1.2271</td>
<td>tele226.288.edg.hicks</td>
</tr>
</tbody>
</table>

**Table 7.3** Comparison of running times and speed-up for single- and simple multi-threaded implementations of GRAPH COLORING algorithm

up is 0.4710 and is at most 0.5954. For GRAPH COLORING the average speed-up is 1.0928 (median is 0.7334) and is at most 5.3101. Tables 7.4, 7.5, and 7.6 show the results for MAXIMUM INDEPENDENT SET, MAXIMUM CUT, and HAMILTONIAN CYCLE, respectively, on large graphs with higher branchwidth. The few occasions where multithreading improved the GRAPH COLORING and HAMILTONIAN CYCLE algorithms are likely due to the join process being very involved and the fact that, for the former, a single join can create several new solutions.

That the multithreaded algorithm is often slower than the single threaded is not surprising. Creating a thread is a relatively expensive process in any language, particularly in Java. It involves instantiating a Thread object with a variable for all the input and copying the input to new variables. Eliminating the instantiations can improve the impact of multithreading.
<table>
<thead>
<tr>
<th>$\beta$</th>
<th>$n + m$</th>
<th>1 thread run time (sec)</th>
<th>4 threads run time (sec)</th>
<th>speed-up</th>
<th>graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>1694</td>
<td>1.5220</td>
<td>7.1250</td>
<td>0.2136</td>
<td>dsj1000.tree</td>
</tr>
<tr>
<td>7</td>
<td>442</td>
<td>0.6220</td>
<td>2.1890</td>
<td>0.2841</td>
<td>si1032.tree</td>
</tr>
<tr>
<td>7</td>
<td>1243</td>
<td>1.1830</td>
<td>5.2170</td>
<td>0.2268</td>
<td>vm1084.tree</td>
</tr>
<tr>
<td>8</td>
<td>1675</td>
<td>2.1950</td>
<td>9.8990</td>
<td>0.2217</td>
<td>pr1002.tree</td>
</tr>
<tr>
<td>8</td>
<td>1909</td>
<td>2.2530</td>
<td>11.0090</td>
<td>0.2047</td>
<td>u1060.tree</td>
</tr>
<tr>
<td>9</td>
<td>1090</td>
<td>2.0660</td>
<td>8.8470</td>
<td>0.2335</td>
<td>rl1323.tree</td>
</tr>
<tr>
<td>9</td>
<td>1226</td>
<td>2.7070</td>
<td>10.2270</td>
<td>0.2647</td>
<td>rl1304.tree</td>
</tr>
<tr>
<td>9</td>
<td>1683</td>
<td>2.4940</td>
<td>10.3670</td>
<td>0.2406</td>
<td>vm1748.tree</td>
</tr>
<tr>
<td>9</td>
<td>1986</td>
<td>7.1920</td>
<td>28.3760</td>
<td>0.2535</td>
<td>rl1889.tree</td>
</tr>
<tr>
<td>9</td>
<td>1999</td>
<td>2.2800</td>
<td>11.0660</td>
<td>0.2060</td>
<td>fl1577.tree</td>
</tr>
<tr>
<td>9</td>
<td>2597</td>
<td>7.8420</td>
<td>30.4770</td>
<td>0.2573</td>
<td>d2103.tree</td>
</tr>
<tr>
<td>9</td>
<td>3450</td>
<td>9.9330</td>
<td>38.3500</td>
<td>0.2590</td>
<td>fl1400.tree</td>
</tr>
<tr>
<td>9</td>
<td>6076</td>
<td>14.4170</td>
<td>56.2880</td>
<td>0.2651</td>
<td>fl3795.tree</td>
</tr>
<tr>
<td>10</td>
<td>1451</td>
<td>9.8240</td>
<td>17.9280</td>
<td>0.5480</td>
<td>dl291.tree</td>
</tr>
<tr>
<td>10</td>
<td>2422</td>
<td>6.6030</td>
<td>20.0460</td>
<td>0.3294</td>
<td>dl655.tree</td>
</tr>
<tr>
<td>11</td>
<td>1838</td>
<td>46.3180</td>
<td>44.2790</td>
<td>1.0450</td>
<td>pcb1173.tree</td>
</tr>
<tr>
<td>13</td>
<td>2315</td>
<td>45.5440</td>
<td>41.0230</td>
<td>1.1102</td>
<td>nrw1379.tree</td>
</tr>
<tr>
<td>13</td>
<td>2786</td>
<td>100.4320</td>
<td>81.3360</td>
<td>1.2348</td>
<td>u1817.tree</td>
</tr>
<tr>
<td>13</td>
<td>3356</td>
<td>75.5350</td>
<td>67.1960</td>
<td>1.1241</td>
<td>u1432.tree</td>
</tr>
<tr>
<td>13</td>
<td>3603</td>
<td>103.4730</td>
<td>116.3990</td>
<td>0.8890</td>
<td>u2152.tree</td>
</tr>
<tr>
<td>15</td>
<td>3953</td>
<td>1511.2050</td>
<td>139.6410</td>
<td>10.8221</td>
<td>pr2392.tree</td>
</tr>
<tr>
<td>15</td>
<td>5094</td>
<td>1123.7150</td>
<td>504.2790</td>
<td>2.2284</td>
<td>pcb3038.tree</td>
</tr>
</tbody>
</table>

Table 7.4 Comparison of running times and speed-up for multithreaded versus single-threaded implementation of Maximum Independent Set algorithm using ThreadLock
<table>
<thead>
<tr>
<th>$\beta$</th>
<th>$n + m$</th>
<th>1 thread run time(sec)</th>
<th>4 threads run time(sec)</th>
<th>speed-up</th>
<th>graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>1694</td>
<td>1.7070</td>
<td>8.6470</td>
<td>0.1974</td>
<td>dsj1000.tree</td>
</tr>
<tr>
<td>7</td>
<td>442</td>
<td>0.6840</td>
<td>2.6750</td>
<td>0.2557</td>
<td>si1032.tree</td>
</tr>
<tr>
<td>7</td>
<td>1243</td>
<td>1.3570</td>
<td>6.3670</td>
<td>0.2131</td>
<td>vm1084.tree</td>
</tr>
<tr>
<td>8</td>
<td>1675</td>
<td>2.5510</td>
<td>11.7790</td>
<td>0.2166</td>
<td>pr1002.tree</td>
</tr>
<tr>
<td>8</td>
<td>1909</td>
<td>2.6130</td>
<td>13.2630</td>
<td>0.1970</td>
<td>u1060.tree</td>
</tr>
<tr>
<td>9</td>
<td>1090</td>
<td>2.3380</td>
<td>10.7280</td>
<td>0.2179</td>
<td>rl1323.tree</td>
</tr>
<tr>
<td>9</td>
<td>1226</td>
<td>3.0080</td>
<td>12.4430</td>
<td>0.2417</td>
<td>rl1304.tree</td>
</tr>
<tr>
<td>9</td>
<td>1663</td>
<td>2.8320</td>
<td>12.1350</td>
<td>0.2334</td>
<td>vm1748.tree</td>
</tr>
<tr>
<td>9</td>
<td>1986</td>
<td>8.7650</td>
<td>32.4940</td>
<td>0.2697</td>
<td>rl1889.tree</td>
</tr>
<tr>
<td>9</td>
<td>1999</td>
<td>2.6170</td>
<td>13.2960</td>
<td>0.1968</td>
<td>fl1577.tree</td>
</tr>
<tr>
<td>9</td>
<td>2597</td>
<td>10.5700</td>
<td>39.3870</td>
<td>0.2684</td>
<td>d2103.tree</td>
</tr>
<tr>
<td>9</td>
<td>3450</td>
<td>14.1130</td>
<td>54.1460</td>
<td>0.2606</td>
<td>fl1400.tree</td>
</tr>
<tr>
<td>9</td>
<td>6076</td>
<td>22.5220</td>
<td>75.8490</td>
<td>0.2969</td>
<td>fl3795.tree</td>
</tr>
<tr>
<td>10</td>
<td>1451</td>
<td>16.9190</td>
<td>22.9630</td>
<td>0.7368</td>
<td>d1291.tree</td>
</tr>
<tr>
<td>10</td>
<td>2422</td>
<td>8.5060</td>
<td>22.8940</td>
<td>0.3715</td>
<td>d1655.tree</td>
</tr>
<tr>
<td>11</td>
<td>1838</td>
<td>58.1340</td>
<td>50.9970</td>
<td>1.1399</td>
<td>pcb1173.tree</td>
</tr>
<tr>
<td>13</td>
<td>2315</td>
<td>47.3070</td>
<td>44.8510</td>
<td>1.0548</td>
<td>nrw1379.tree</td>
</tr>
<tr>
<td>13</td>
<td>2786</td>
<td>246.6680</td>
<td>106.8310</td>
<td>2.3090</td>
<td>u1817.tree</td>
</tr>
<tr>
<td>13</td>
<td>3356</td>
<td>157.0630</td>
<td>84.3230</td>
<td>1.8626</td>
<td>u1432.tree</td>
</tr>
<tr>
<td>13</td>
<td>3603</td>
<td>211.4930</td>
<td>152.4310</td>
<td>1.3875</td>
<td>u2152.tree</td>
</tr>
<tr>
<td>15</td>
<td>3953</td>
<td>2097.0880</td>
<td>174.2520</td>
<td>12.0348</td>
<td>pr2392.tree</td>
</tr>
<tr>
<td>15</td>
<td>5094</td>
<td>1251.4350</td>
<td>553.1680</td>
<td>2.2623</td>
<td>pcb3038.tree</td>
</tr>
</tbody>
</table>

Table 7.5  Comparison of running times and speed-up for multithreaded versus single-threaded implementation of Maximum Cut algorithm using ThreadLock
<table>
<thead>
<tr>
<th>$\beta$</th>
<th>$n + m$</th>
<th>1 thread run time (sec)</th>
<th>4 threads run time (sec)</th>
<th>speed-up</th>
<th>graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>1694</td>
<td>2.6000</td>
<td>4.1600</td>
<td>0.6250</td>
<td>dsj1000.tree</td>
</tr>
<tr>
<td>7</td>
<td>442</td>
<td>1.3300</td>
<td>1.8600</td>
<td>0.7151</td>
<td>si1032.tree</td>
</tr>
<tr>
<td>7</td>
<td>1243</td>
<td>2.6200</td>
<td>4.4800</td>
<td>0.5848</td>
<td>vm1084.tree</td>
</tr>
<tr>
<td>8</td>
<td>1675</td>
<td>4.3900</td>
<td>8.1700</td>
<td>0.5373</td>
<td>pr1002.tree</td>
</tr>
<tr>
<td>8</td>
<td>1909</td>
<td>4.1300</td>
<td>8.0300</td>
<td>0.5143</td>
<td>u1060.tree</td>
</tr>
<tr>
<td>9</td>
<td>1090</td>
<td>3.5100</td>
<td>5.8800</td>
<td>0.5969</td>
<td>rl1323.tree</td>
</tr>
<tr>
<td>9</td>
<td>1226</td>
<td>3.9000</td>
<td>6.5100</td>
<td>0.5991</td>
<td>rl1304.tree</td>
</tr>
<tr>
<td>9</td>
<td>1663</td>
<td>4.7700</td>
<td>7.3500</td>
<td>0.6490</td>
<td>vm1748.tree</td>
</tr>
<tr>
<td>9</td>
<td>1986</td>
<td>63.9400</td>
<td>22.7100</td>
<td>2.8155</td>
<td>rl1889.tree</td>
</tr>
<tr>
<td>9</td>
<td>1999</td>
<td>3.7100</td>
<td>8.1300</td>
<td>0.4563</td>
<td>fl1577.tree</td>
</tr>
<tr>
<td>9</td>
<td>2597</td>
<td>37.8800</td>
<td>33.8899</td>
<td>1.1177</td>
<td>d2103.tree</td>
</tr>
<tr>
<td>9</td>
<td>3450</td>
<td>756.1500</td>
<td>233.7600</td>
<td>3.2347</td>
<td>fl1400.tree</td>
</tr>
<tr>
<td>9</td>
<td>6076</td>
<td>2230.7200</td>
<td>340.7500</td>
<td>6.5465</td>
<td>fl3795.tree</td>
</tr>
<tr>
<td>10</td>
<td>1451</td>
<td>6.5100</td>
<td>9.0800</td>
<td>0.7170</td>
<td>dl291.tree</td>
</tr>
<tr>
<td>10</td>
<td>2422</td>
<td>42.8000</td>
<td>25.8000</td>
<td>1.6589</td>
<td>dl1655.tree</td>
</tr>
<tr>
<td>11</td>
<td>1838</td>
<td>271.7800</td>
<td>71.0800</td>
<td>3.8236</td>
<td>pcb1173.tree</td>
</tr>
<tr>
<td>13</td>
<td>2315</td>
<td>118.4300</td>
<td>54.2600</td>
<td>2.1826</td>
<td>nrw1379.tree</td>
</tr>
<tr>
<td>13</td>
<td>3356</td>
<td>20768.6600</td>
<td>821.7900</td>
<td>25.2725</td>
<td>u1432.tree</td>
</tr>
<tr>
<td>13</td>
<td>2786</td>
<td>1245.0800</td>
<td>182.0800</td>
<td>6.8381</td>
<td>u1817.tree</td>
</tr>
<tr>
<td>13</td>
<td>3603</td>
<td>1844.3400</td>
<td>605.7200</td>
<td>3.0449</td>
<td>u2152.tree</td>
</tr>
</tbody>
</table>

**Table 7.6** Comparison of running times and speed-up for multithreaded versus single-threaded implementation of HAMILTONIAN CYCLE algorithm using ThreadLock
7.2 Thread Pooling

When implementing code for optimization, it is common to reuse variables whenever possible. Likewise, for multithreaded programs, it can be advantageous to reuse threads. This is particularly true if the program has several short-running threads that do not all have to run at the same time.

After a thread dies it cannot be restarted; therefore, instead of reusing the thread, we create a pool of threads that are designed to cycle continuously between sleep and work modes. In this implementation, the main algorithm begins by creating for each processor a SolutionJoinerThread that immediately goes to sleep waiting to be notified of work to be done. It executes the following

\[
\text{while( true )}
\]

\[
\text{while ( no work assigned )}
\]

\[
\text{wait for notification of work}
\]

\[
\text{doWork()}
\]

We then replace the instantiation of the SolutionJoiner in Algorithm 7 with a call to the method resetJoiner which copies the input variables to a sleeping or waiting SolutionJoinerThread object in the thread pool and registers the thread with the ThreadLock. The start of the thread is replaced by a notification to the sleeping SolutionJoinerThread that work has been assigned and to begin working. The method doWork performs all the steps listed for the started SolutionJoiner in Algorithm 7.

Tables 7.7, 7.8 and 7.9 show the running times for the multithreaded framework using thread pools. The set of graphs is the same as used in Tables 7.1 to 7.3 and we compare these running times with the single threaded algorithm and the multithreaded algorithm that does not use thread pools. Although the new multithreaded algorithms are not faster in all cases than the single threaded algorithms; for most
<table>
<thead>
<tr>
<th>$\beta$</th>
<th>$n + m$</th>
<th>run time (sec)</th>
<th>speed-up</th>
<th>w/o pool</th>
<th>w/ pool</th>
<th>graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>20</td>
<td>0.1290</td>
<td>1.0310</td>
<td>1.5271</td>
<td></td>
<td>cube3.hicks</td>
</tr>
<tr>
<td>4</td>
<td>29</td>
<td>0.1550</td>
<td>0.8710</td>
<td>1.4516</td>
<td></td>
<td>bmruty.hicks</td>
</tr>
<tr>
<td>4</td>
<td>82</td>
<td>0.2180</td>
<td>0.9817</td>
<td>1.7936</td>
<td></td>
<td>m4.graph.hicks</td>
</tr>
<tr>
<td>4</td>
<td>126</td>
<td>0.2300</td>
<td>0.9957</td>
<td>1.5957</td>
<td></td>
<td>ma.graph.hicks</td>
</tr>
<tr>
<td>4</td>
<td>168</td>
<td>0.2610</td>
<td>1.0421</td>
<td>1.7778</td>
<td></td>
<td>...trans..output.graph.hicks</td>
</tr>
<tr>
<td>4</td>
<td>236</td>
<td>0.3590</td>
<td>0.9136</td>
<td>1.8496</td>
<td></td>
<td>mf.graph.hicks</td>
</tr>
<tr>
<td>4</td>
<td>255</td>
<td>0.3580</td>
<td>0.8715</td>
<td>1.9330</td>
<td></td>
<td>m1.graph.hicks</td>
</tr>
<tr>
<td>5</td>
<td>31</td>
<td>0.1570</td>
<td>0.9745</td>
<td>1.8917</td>
<td></td>
<td>mcyriel3.col.edg.hicks</td>
</tr>
<tr>
<td>5</td>
<td>141</td>
<td>0.2780</td>
<td>0.9029</td>
<td>1.6367</td>
<td></td>
<td>tele56_85.edg.hicks</td>
</tr>
<tr>
<td>5</td>
<td>209</td>
<td>0.3430</td>
<td>0.9184</td>
<td>1.8659</td>
<td></td>
<td>md.graph.hicks</td>
</tr>
<tr>
<td>5</td>
<td>345</td>
<td>0.3740</td>
<td>1.2059</td>
<td>2.7941</td>
<td></td>
<td>bico133.hicks</td>
</tr>
<tr>
<td>6</td>
<td>194</td>
<td>0.3750</td>
<td>0.8560</td>
<td>2.2800</td>
<td></td>
<td>mb.graph.hicks</td>
</tr>
<tr>
<td>6</td>
<td>390</td>
<td>0.4960</td>
<td>1.7016</td>
<td>4.1149</td>
<td></td>
<td>pr107.tsp.del.hicks</td>
</tr>
<tr>
<td>6</td>
<td>514</td>
<td>0.5180</td>
<td>0.9537</td>
<td>2.7857</td>
<td></td>
<td>tele226_288.edg.hicks</td>
</tr>
</tbody>
</table>

Table 7.7  Comparison of multithreaded implementation of MAXIMUM INDEPENDENT SET algorithm with and without thread pools

<table>
<thead>
<tr>
<th>$\beta$</th>
<th>$n + m$</th>
<th>run time (sec)</th>
<th>speed-up</th>
<th>w/o pool</th>
<th>w/ pool</th>
<th>graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>20</td>
<td>0.1400</td>
<td>0.9071</td>
<td>1.5929</td>
<td></td>
<td>cube3.hicks</td>
</tr>
<tr>
<td>4</td>
<td>29</td>
<td>0.1560</td>
<td>1.0000</td>
<td>1.6795</td>
<td></td>
<td>bmruty.hicks</td>
</tr>
<tr>
<td>4</td>
<td>82</td>
<td>0.2300</td>
<td>0.9609</td>
<td>2.1044</td>
<td></td>
<td>m4.graph.hicks</td>
</tr>
<tr>
<td>4</td>
<td>126</td>
<td>0.2400</td>
<td>0.9083</td>
<td>1.7167</td>
<td></td>
<td>ma.graph.hicks</td>
</tr>
<tr>
<td>4</td>
<td>168</td>
<td>0.2830</td>
<td>0.8975</td>
<td>1.8304</td>
<td></td>
<td>...trans..output.graph.hicks</td>
</tr>
<tr>
<td>4</td>
<td>236</td>
<td>0.3650</td>
<td>0.9726</td>
<td>1.9233</td>
<td></td>
<td>mf.graph.hicks</td>
</tr>
<tr>
<td>4</td>
<td>255</td>
<td>0.3610</td>
<td>0.9418</td>
<td>2.0471</td>
<td></td>
<td>m1.graph.hicks</td>
</tr>
<tr>
<td>5</td>
<td>31</td>
<td>0.1500</td>
<td>1.0867</td>
<td>2.2333</td>
<td></td>
<td>mcyriel3.col.edg.hicks</td>
</tr>
<tr>
<td>5</td>
<td>141</td>
<td>0.2820</td>
<td>0.9574</td>
<td>1.8262</td>
<td></td>
<td>tele56_85.edg.hicks</td>
</tr>
<tr>
<td>5</td>
<td>209</td>
<td>0.3540</td>
<td>0.9153</td>
<td>1.9379</td>
<td></td>
<td>md.graph.hicks</td>
</tr>
<tr>
<td>5</td>
<td>345</td>
<td>0.4190</td>
<td>1.2721</td>
<td>2.9380</td>
<td></td>
<td>bico133.hicks</td>
</tr>
<tr>
<td>6</td>
<td>194</td>
<td>0.3870</td>
<td>0.9431</td>
<td>2.4987</td>
<td></td>
<td>mb.graph.hicks</td>
</tr>
<tr>
<td>6</td>
<td>390</td>
<td>0.6000</td>
<td>1.9883</td>
<td>5.1883</td>
<td></td>
<td>pr107.tsp.del.hicks</td>
</tr>
<tr>
<td>6</td>
<td>514</td>
<td>0.5200</td>
<td>1.0115</td>
<td>3.2577</td>
<td></td>
<td>tele226_288.edg.hicks</td>
</tr>
</tbody>
</table>

Table 7.8  Comparison of multithreaded implementation of MAXIMUM CUT algorithm with and without thread pools
<table>
<thead>
<tr>
<th>$\beta$</th>
<th>$n + m$</th>
<th>run time (sec)</th>
<th>speed-up</th>
<th>w/o pool w/ pool</th>
<th>graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>20</td>
<td>0.1360</td>
<td>1.0294</td>
<td>1.1397</td>
<td>cube3.hicks</td>
</tr>
<tr>
<td>4</td>
<td>29</td>
<td>0.1550</td>
<td>1.0065</td>
<td>1.2968</td>
<td>bmrty.hicks</td>
</tr>
<tr>
<td>4</td>
<td>82</td>
<td>0.2320</td>
<td>0.9871</td>
<td>1.6595</td>
<td>m4.graph.hicks</td>
</tr>
<tr>
<td>4</td>
<td>126</td>
<td>0.2490</td>
<td>0.9679</td>
<td>1.2731</td>
<td>ma.graph.hicks</td>
</tr>
<tr>
<td>4</td>
<td>168</td>
<td>0.2880</td>
<td>0.9618</td>
<td>1.3611</td>
<td>_trans_output.graph.hicks</td>
</tr>
<tr>
<td>4</td>
<td>236</td>
<td>0.3290</td>
<td>0.9483</td>
<td>1.5897</td>
<td>mf.graph.hicks</td>
</tr>
<tr>
<td>4</td>
<td>255</td>
<td>0.3380</td>
<td>1.0059</td>
<td>1.5444</td>
<td>m1.graph.hicks</td>
</tr>
<tr>
<td>5</td>
<td>31</td>
<td>0.1690</td>
<td>1.1124</td>
<td>1.3077</td>
<td>myciel3.col.edg.hicks</td>
</tr>
<tr>
<td>5</td>
<td>141</td>
<td>0.2840</td>
<td>0.9718</td>
<td>1.5951</td>
<td>tele56_85.edg.hicks</td>
</tr>
<tr>
<td>5</td>
<td>209</td>
<td>0.3250</td>
<td>0.9846</td>
<td>1.7692</td>
<td>md.graph.hicks</td>
</tr>
<tr>
<td>5</td>
<td>345</td>
<td>0.4610</td>
<td>2.3991</td>
<td>1.8113</td>
<td>bico133.hicks</td>
</tr>
<tr>
<td>6</td>
<td>194</td>
<td>0.4910</td>
<td>1.3299</td>
<td>3.0815</td>
<td>mb.graph.hicks</td>
</tr>
<tr>
<td>6</td>
<td>390</td>
<td>4.2690</td>
<td>8.5069</td>
<td>1.6020</td>
<td>pr107.tsp.del.hicks</td>
</tr>
<tr>
<td>6</td>
<td>514</td>
<td>0.7400</td>
<td>3.6649</td>
<td>2.9865</td>
<td>tele226_288.edg.hicks</td>
</tr>
</tbody>
</table>

Table 7.9 Comparison of multithreaded and single-threaded implementation of Graph Coloring algorithm using ThreadLock

graphs in the set, the speed-up is at least .9 and as high as 8.5. Again, the Graph Coloring algorithm is the one that is most affected by parallelization. We also note that in every table, the last column of values shows that the multithreaded algorithm that uses a thread pool is faster than the other multithreaded algorithm.

As in the previous section, we show in tables 7.10, 7.11, and 7.12 the results for Maximum Independent Set, Maximum Cut, and Hamiltonian Cycle on the larger graphs with higher branchwidth. In all cases, the multithreaded implementation with thread pooling is faster than the other tow implementations. For the several of the larger graphs, the large speedup is most likely due, in part, to optimizations in the Java Virtual Machine.
<table>
<thead>
<tr>
<th>$\beta$</th>
<th>$n + m$</th>
<th>run time (sec)</th>
<th>speed-up</th>
<th>w/o pool</th>
<th>w/ pool</th>
<th>graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>1694</td>
<td>1.3180</td>
<td>1.1548</td>
<td>5.4059</td>
<td></td>
<td>dsj1000.tree</td>
</tr>
<tr>
<td>7</td>
<td>442</td>
<td>0.5570</td>
<td>1.1167</td>
<td>3.9300</td>
<td></td>
<td>si1032.tree</td>
</tr>
<tr>
<td>7</td>
<td>1243</td>
<td>1.0020</td>
<td>1.1806</td>
<td>5.2066</td>
<td></td>
<td>vm1084.tree</td>
</tr>
<tr>
<td>8</td>
<td>1675</td>
<td>1.4830</td>
<td>1.4801</td>
<td>6.6750</td>
<td></td>
<td>pr1002.tree</td>
</tr>
<tr>
<td>8</td>
<td>1909</td>
<td>1.7280</td>
<td>1.3038</td>
<td>6.3710</td>
<td></td>
<td>u1060.tree</td>
</tr>
<tr>
<td>9</td>
<td>1090</td>
<td>1.3780</td>
<td>1.4993</td>
<td>6.4202</td>
<td></td>
<td>rl1323.tree</td>
</tr>
<tr>
<td>9</td>
<td>1226</td>
<td>1.3460</td>
<td>2.0111</td>
<td>7.5981</td>
<td></td>
<td>rl1304.tree</td>
</tr>
<tr>
<td>9</td>
<td>1663</td>
<td>1.5300</td>
<td>1.6301</td>
<td>6.7758</td>
<td></td>
<td>vm1748.tree</td>
</tr>
<tr>
<td>9</td>
<td>1986</td>
<td>3.4440</td>
<td>2.0883</td>
<td>8.2393</td>
<td></td>
<td>rl1889.tree</td>
</tr>
<tr>
<td>9</td>
<td>1999</td>
<td>1.7600</td>
<td>1.2955</td>
<td>6.2875</td>
<td></td>
<td>fl1577.tree</td>
</tr>
<tr>
<td>9</td>
<td>2597</td>
<td>4.3930</td>
<td>1.7851</td>
<td>6.9376</td>
<td></td>
<td>d2103.tree</td>
</tr>
<tr>
<td>9</td>
<td>3450</td>
<td>5.3650</td>
<td>1.8514</td>
<td>7.1482</td>
<td></td>
<td>fl1400.tree</td>
</tr>
<tr>
<td>9</td>
<td>6076</td>
<td>8.1060</td>
<td>1.7786</td>
<td>6.9440</td>
<td></td>
<td>fl3795.tree</td>
</tr>
<tr>
<td>10</td>
<td>1451</td>
<td>2.9700</td>
<td>3.3077</td>
<td>6.0364</td>
<td></td>
<td>dl291.tree</td>
</tr>
<tr>
<td>10</td>
<td>2422</td>
<td>2.7830</td>
<td>2.3726</td>
<td>7.2030</td>
<td></td>
<td>dl655.tree</td>
</tr>
<tr>
<td>11</td>
<td>1838</td>
<td>4.9480</td>
<td>9.3610</td>
<td>8.9489</td>
<td></td>
<td>pcb1173.tree</td>
</tr>
<tr>
<td>13</td>
<td>2315</td>
<td>4.2680</td>
<td>10.6710</td>
<td>9.6118</td>
<td></td>
<td>nrw1379.tree</td>
</tr>
<tr>
<td>13</td>
<td>2786</td>
<td>9.4640</td>
<td>10.6120</td>
<td>8.5943</td>
<td></td>
<td>u1817.tree</td>
</tr>
<tr>
<td>13</td>
<td>3556</td>
<td>9.0200</td>
<td>8.3742</td>
<td>7.4497</td>
<td></td>
<td>u1432.tree</td>
</tr>
<tr>
<td>13</td>
<td>3603</td>
<td>14.9440</td>
<td>6.9241</td>
<td>7.7890</td>
<td></td>
<td>u2152.tree</td>
</tr>
<tr>
<td>15</td>
<td>3953</td>
<td>12.6760</td>
<td>119.2178</td>
<td>11.0162</td>
<td></td>
<td>pr2392.tree</td>
</tr>
<tr>
<td>15</td>
<td>5094</td>
<td>51.3700</td>
<td>218.749</td>
<td>9.8166</td>
<td></td>
<td>pcb3038.tree</td>
</tr>
</tbody>
</table>

Table 7.10  Comparison of multithreaded implementation of MAXIMUM INDEPENDENT SET algorithm with and without thread pools
<table>
<thead>
<tr>
<th>$\beta$</th>
<th>$n + m$</th>
<th>run time(sec)</th>
<th>speed-up</th>
<th>w/o pool</th>
<th>graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>1694</td>
<td>1.4390</td>
<td>1.1862</td>
<td>6.0090</td>
<td>dsj1000.tree</td>
</tr>
<tr>
<td>7</td>
<td>442</td>
<td>0.5940</td>
<td>1.1515</td>
<td>4.5034</td>
<td>si1032.tree</td>
</tr>
<tr>
<td>7</td>
<td>1243</td>
<td>1.0910</td>
<td>1.2438</td>
<td>5.8359</td>
<td>vm1084.tree</td>
</tr>
<tr>
<td>8</td>
<td>1675</td>
<td>1.5860</td>
<td>1.6084</td>
<td>7.4269</td>
<td>pr1002.tree</td>
</tr>
<tr>
<td>8</td>
<td>1909</td>
<td>1.9910</td>
<td>1.3124</td>
<td>6.6615</td>
<td>u1060.tree</td>
</tr>
<tr>
<td>9</td>
<td>1090</td>
<td>1.4220</td>
<td>1.6442</td>
<td>7.5443</td>
<td>rl1323.tree</td>
</tr>
<tr>
<td>9</td>
<td>1226</td>
<td>1.5500</td>
<td>1.9406</td>
<td>8.0277</td>
<td>rl1304.tree</td>
</tr>
<tr>
<td>9</td>
<td>1663</td>
<td>1.6720</td>
<td>1.6938</td>
<td>7.2578</td>
<td>vm1748.tree</td>
</tr>
<tr>
<td>9</td>
<td>1986</td>
<td>3.7870</td>
<td>2.3145</td>
<td>8.5804</td>
<td>rl1889.tree</td>
</tr>
<tr>
<td>9</td>
<td>1999</td>
<td>1.8350</td>
<td>1.4262</td>
<td>7.2458</td>
<td>fl1577.tree</td>
</tr>
<tr>
<td>9</td>
<td>2597</td>
<td>5.6200</td>
<td>1.8808</td>
<td>7.0084</td>
<td>d2103.tree</td>
</tr>
<tr>
<td>9</td>
<td>3450</td>
<td>6.6310</td>
<td>2.1283</td>
<td>8.1656</td>
<td>fl1400.tree</td>
</tr>
<tr>
<td>9</td>
<td>6076</td>
<td>9.6830</td>
<td>2.3259</td>
<td>7.8332</td>
<td>fl3795.tree</td>
</tr>
<tr>
<td>10</td>
<td>1451</td>
<td>3.6000</td>
<td>4.6997</td>
<td>6.3786</td>
<td>d1291.tree</td>
</tr>
<tr>
<td>10</td>
<td>2422</td>
<td>3.0610</td>
<td>2.7788</td>
<td>7.4793</td>
<td>d1655.tree</td>
</tr>
<tr>
<td>11</td>
<td>1838</td>
<td>5.5550</td>
<td>10.4652</td>
<td>9.1804</td>
<td>pcb1173.tree</td>
</tr>
<tr>
<td>13</td>
<td>2315</td>
<td>4.4330</td>
<td>10.6475</td>
<td>10.0948</td>
<td>nrw1379.tree</td>
</tr>
<tr>
<td>13</td>
<td>2786</td>
<td>11.3450</td>
<td>21.7424</td>
<td>9.4166</td>
<td>u1817.tree</td>
</tr>
<tr>
<td>13</td>
<td>3356</td>
<td>10.7320</td>
<td>14.6350</td>
<td>7.8572</td>
<td>u1432.tree</td>
</tr>
<tr>
<td>13</td>
<td>3603</td>
<td>17.7840</td>
<td>11.8923</td>
<td>8.5712</td>
<td>u2152.tree</td>
</tr>
<tr>
<td>15</td>
<td>3953</td>
<td>15.6060</td>
<td>134.3770</td>
<td>11.1657</td>
<td>pr2392.tree</td>
</tr>
<tr>
<td>15</td>
<td>5094</td>
<td>55.8800</td>
<td>22.3950</td>
<td>9.8992</td>
<td>pcb3038.tree</td>
</tr>
</tbody>
</table>

**Table 7.11** Comparison of multithreaded implementation of MAXIMUM CUT algorithm with and without thread pools
<table>
<thead>
<tr>
<th>$\beta$</th>
<th>$n + m$</th>
<th>run time (sec)</th>
<th>speed-up</th>
<th>$w/o$ pool (\frac{w}{w/pool})</th>
<th>graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>1694</td>
<td>2.5800</td>
<td>1.0078</td>
<td>1.6124</td>
<td>dsj1000.tree</td>
</tr>
<tr>
<td>7</td>
<td>442</td>
<td>1.3100</td>
<td>1.0153</td>
<td>1.4199</td>
<td>si1032.tree</td>
</tr>
<tr>
<td>7</td>
<td>1243</td>
<td>2.3600</td>
<td>1.1102</td>
<td>1.8983</td>
<td>vm1084.tree</td>
</tr>
<tr>
<td>8</td>
<td>1675</td>
<td>3.1900</td>
<td>1.3762</td>
<td>2.5611</td>
<td>pr1002.tree</td>
</tr>
<tr>
<td>8</td>
<td>1909</td>
<td>3.2700</td>
<td>1.2630</td>
<td>2.4557</td>
<td>u1060.tree</td>
</tr>
<tr>
<td>9</td>
<td>1090</td>
<td>2.5600</td>
<td>1.3711</td>
<td>2.2969</td>
<td>rl1323.tree</td>
</tr>
<tr>
<td>9</td>
<td>1226</td>
<td>2.6300</td>
<td>1.4829</td>
<td>2.4753</td>
<td>rl1304.tree</td>
</tr>
<tr>
<td>9</td>
<td>1663</td>
<td>2.9900</td>
<td>1.5953</td>
<td>2.4582</td>
<td>vm1748.tree</td>
</tr>
<tr>
<td>9</td>
<td>1986</td>
<td>6.2100</td>
<td>10.2963</td>
<td>3.6570</td>
<td>rl1889.tree</td>
</tr>
<tr>
<td>9</td>
<td>1999</td>
<td>3.3200</td>
<td>1.1175</td>
<td>2.4488</td>
<td>fl1577.tree</td>
</tr>
<tr>
<td>9</td>
<td>2597</td>
<td>8.1200</td>
<td>4.6650</td>
<td>4.1736</td>
<td>d2103.tree</td>
</tr>
<tr>
<td>9</td>
<td>3450</td>
<td>23.0400</td>
<td>32.8190</td>
<td>10.1458</td>
<td>fl1400.tree</td>
</tr>
<tr>
<td>9</td>
<td>6076</td>
<td>37.7300</td>
<td>59.1232</td>
<td>9.0313</td>
<td>fl3795.tree</td>
</tr>
<tr>
<td>10</td>
<td>1451</td>
<td>3.3000</td>
<td>1.9727</td>
<td>2.7515</td>
<td>d1291.tree</td>
</tr>
<tr>
<td>10</td>
<td>2422</td>
<td>4.9800</td>
<td>8.5944</td>
<td>5.1807</td>
<td>d1655.tree</td>
</tr>
<tr>
<td>11</td>
<td>1838</td>
<td>12.0500</td>
<td>22.5544</td>
<td>5.8988</td>
<td>pch1173.tree</td>
</tr>
<tr>
<td>13</td>
<td>2315</td>
<td>7.6300</td>
<td>15.5216</td>
<td>7.1114</td>
<td>nrw1379.tree</td>
</tr>
<tr>
<td>13</td>
<td>3356</td>
<td>219.1800</td>
<td>94.7562</td>
<td>3.7494</td>
<td>u1432.tree</td>
</tr>
<tr>
<td>13</td>
<td>2786</td>
<td>29.7000</td>
<td>41.9219</td>
<td>6.1306</td>
<td>u1817.tree</td>
</tr>
<tr>
<td>13</td>
<td>3603</td>
<td>102.5500</td>
<td>17.9848</td>
<td>5.9066</td>
<td>u2152.tree</td>
</tr>
</tbody>
</table>

Table 7.12: Comparison of multithreaded implementation of Hamiltonian Cycle algorithm with and without thread pools
7.3 Possible Improvements

Given that the multithreaded algorithms are not consistently faster than the single threaded, there is definite room for improvement. We identify possible modifications to the existing framework and a different parallelization approach that may be effective.

Solution Pair Selection

Recall that we choose a single left solution, partition the right solutions, and each thread joins the left to each of the compatible solutions in a particular block. Consider, however, when there are more left solutions than right solutions. If we allow the algorithm to switch the left and right solution sets, then we can reduce the amount of thread resetting (i.e., calls to resetJoiner) and, consequently, reduce the overhead involved in multithreading.

In another change to the choice of partitions, we could modify the method by which we establish pairings. That is, currently, we currently choose a single left solution and a block of right solutions; however, we do not know the impact of choosing a block of left solutions and a block of right solutions.

These modifications to the multithreaded approach described in this chapter or one of several other possible partitioning schemes can all affect resource usage and running time. A cost whose effect on running time may or may not be measurable is the cost of adding more complexity to the algorithms.

Parallel Tree Traversal

Another approach to parallelizing decomposition tree based algorithms is to parallelize the tree traversal instead of or (given enough processors) in addition to the join
process. Given a node ordering (i.e., a list of the nodes in the order of some traversal method), a parallel implementation can be as simple as processing each node in that order on alternating processors. Unfortunately, this technique is inefficient for breadth-first and depth-first orderings. For breadth-first ordering, if the rooted decomposition tree is neither balanced nor almost balanced, then processors could be left unused while the nodes of a long, thin branch are being processed. For a depth-first ordering, there is the possibility that a processor may be assigned a node or task and be unable to work because that node’s children have not been completed.

In any implementation, care must be taken to minimize the likelihood of a thread waiting for another thread to complete before it can continue.
Chapter 8

JPAD

Much of the design, analysis, and debugging of the algorithms presented in this thesis was assisted by a software we call JPAD. Written in Java[25], JPAD can be used on several different platforms as an extendible tool to visualize, validate, and analyze graphs, and graph algorithms; especially with respect to branch decompositions. What we generally refer to as JPAD is actually the core functionality of JPAD with functionality for analyzing and displaying branch decompositions added to it.

The creation of JPAD was inspired by programs like NETPAD and LINK [6], two packages rich with features for drawing and manipulating data represented as graphs. Rather than a simple recreation of these or other popular graph visualization tools, what we gained through JPAD was establishing a tool to visualize graph decompositions with the level of flexibility and interaction that we needed to further our understanding of branch decompositions. Although some aspects of the program are still at a developmental level, JPAD serves its purpose in helping us to visualize middle set interactions, graph partitioning through branch decompositions, and behavior of algorithms that are based on these decompositions.

JPAD’s features and functions are accessed through its graphical user interface (GUI). Most of the functionality is intuitive—a combination of menu options and mouse operations. We describe some of the basic functionality of JPAD below.
8.1 File Options

To view a graph in JPAD, the user can specify the filename on the command line when starting JPAD or choose to “Open” a file from the “File” menu. Currently, there is not functionality to draw graphs by hand because JPAD is written for the primary purpose of viewing branch decompositions and analyzing algorithms.

As stated above, JPAD is two applications.

If JPAD is started using the branch decomposition classes, the file specified must define a branch decomposition and graph in the file format

\[
\begin{align*}
n_T & \quad m_T & \quad \beta(G) \\
a_0^T & \quad b_0^T & \quad 0 \\
a_1^T & \quad b_1^T & \quad 0 \\
& \quad \vdots \\
a_{m_T-1}^T & \quad b_{m_T-1}^T & \quad 0 \\
n_G & \quad m_G & \\
a_0^G & \quad b_0^G & \quad \tau(e_0^G) \\
a_1^G & \quad b_1^G & \quad \tau(e_1^G) \\
& \quad \vdots \\
a_{m_G-1}^G & \quad b_{m_G-1}^G & \quad \tau(e_{m_G-1}^G)
\end{align*}
\]

where \( n_T \) is the number of vertices in the decomposition tree, \( m_T \) is the number of edges, \( 0 \leq a_i^T, b_i^T \leq m_T - 1 \) are the endpoints of the edge \( e_i^T \in E(T) \). The same values are defined for the original graph with superscript or subscript \( G \) instead of \( T \). The function \( \tau(e_i) \) is the mapping from the edges of \( G \) to the leaves of \( T \) that defines the branch decomposition.
If JPAD is started using the core application or a graph (without its decomposition) is to be viewed, the graph must be an edge file with content in the format

\[
\begin{array}{ccc}
  a_0 & b_0 & w_0 \\
  a_1 & b_1 & w_1 \\
  \vdots \\
  a_{m-1} & b_{m-1} & w_{m-1}
\end{array}
\]

where \( n \) is the number of vertices, \( m \) is the number of edges, \( 0 \leq a_i, b_i \leq n - 1 \) are the endpoints of the edges, and \( w_i \) is the weight of each edge. The second command line argument for the JPAD core can be a node position file with the format

\[
\begin{array}{cc}
  x_0 & y_0 \\
  x_1 & y_1 \\
  \vdots \\
  x_{n-1} & y_{n-1}
\end{array}
\]

where \( n \) is the number of vertices, and for \( i = 0, \ldots, n - 1 \) the pair \((x_i, y_i)\) is a set of coordinates for the vertices.

JPAD provides the option to save a graph using the file format given for standard graphs. By default, Edge files are given the file extension “.edg” and vertex coordinate files the extension “.dat”.

The user can also save a graph as an encapsulated postscript (EPS) file. EPS files can be saved and added to \LaTeXX\ and other documents. Or they can be viewed and printed using an application such as ghostview. A branch decomposition and the original graph can be saved as a single EPS file that shows both graphs in the same image.
8.2 Views

If the user does not specify a set of node positions when a graph is opened, the graph is displayed with random node positions. Two other options are available. JPAD simply can arrange the vertices in a circle or can position the vertices based on the spring embedding algorithm of Kamada and Kawai [17].

We note here that to achieve better performance, the implementation of the spring embedding algorithm differs slightly from that specified by the authors. Stated briefly, the authors suggest repeating the process of choosing a vertex and iteratively finding its optimal position with respect to the other vertices. We have found better performance (i.e., faster convergence) by choosing a vertex and iteratively improving its position until either the locally optimal position is found or a maximum number of iterations is reached. This modification reduces the likelihood of spending several iterations making negligible improvements because the original algorithm can focus too narrowly on a single vertex.

Regardless of how the graph is rendered, the user has options to display labels for the nodes and/or edges. For vertices of general graphs, the choice is whether to display the node number or not. For edges, the user may choose between no label, edge index or edge weight. For branch decompositions, these options are expanded. Nodes can also be displayed so that leaves are labeled with the edges to which they are mapped or with the endpoints of the edges. Edges in the decomposition can be labeled with their corresponding middle sets.

All of these viewing and rendering options help the user to visualize graphs and their relationships to their branch decompositions.
8.3 Features

Included in JPAD are some basic functions of graph rendering programs. Once the graph is displayed, vertices can be repositioned and labels can be shown for the vertices and the edges. However, the feature of JPAD that sets it apart from other graph drawing programs is its ability to display branch decompositions and their relations to the original graph.

The most obvious difference between JPAD and programs like LINK and NETPAD is that JPAD can open a branch decomposition file and display the decomposition tree and the original graph. Clicking on either the decomposition or the graph has the effect of highlighting the relationships.

For example, since each leaf of the decomposition tree is mapped to an edge in the original graph, clicking on a leaf of the decomposition highlights the corresponding edge in the original graph. The reverse is also true; that is, clicking on an edge in the original graph highlights the corresponding leaf of the decomposition tree.

Consider also that each edge of the decomposition tree bipartitions the tree and the graph. Thus, in JPAD clicking on an edge of the decomposition tree causes the nodes of the two parts of the tree to be colored in two different colors and it causes the edges of the graph to be colored in the corresponding way. In addition, the vertices in the middle set of the clicked edge are displayed in the original graph in yet a different color.

8.4 Algorithms

The JPAD function that highlights edges can be easily called from any graph algorithm and is obviously beneficial for analyzing graph algorithms. By taking advantage
of Java's ease in multithreading, JPAD can run algorithms as a separate thread. The preferred method of adding or running an algorithm in JPAD involves three steps.

1. **Implement the algorithm as a subclass of the class graphs.algorithms.Algorithm.**
   Of course, this implementation does not have to be completely rewritten as a subclass of Algorithm; especially for algorithms already implemented in Java or native code, the subclass of Algorithm can be a simple wrapper that passes the appropriate arguments to an existing implementation.

2. **The subclass of Algorithm is then wrapped in an instance of the class jpad.algorithms.JPADAlgorithm.** This class extends the Java Runnable class and has the necessary interfaces to display in JPAD the progress reported by the Algorithm subclass and to display the results of the algorithm when it is finished.

3. **Finally, the instance of JPADAlgorithm is wrapped in a Thread instance and is started.** This step is not essential; however, is allows the user to perform other tasks while the algorithm executes.

Assuming Step 1 has been completed beforehand, the second and third steps can be accomplished in a single, concise line of source code.

Some sample algorithms implemented in JPAD are Kruskal's algorithm for finding a minimum spanning tree and Tarjan's algorithm [26] to determine biconnected components. Additionally, several other features are implemented using the method outlined above.

Because JPAD is designed for analyzing algorithms, a small window that displays a plot of memory usage is a part of the main window. This monitor periodically checks the amount of RAM being used and is helpful in visually determining the rate at which memory usage is growing.
8.5 Branch Decomposition Based Algorithms

In addition to traditional graph algorithms, JPAD can run branch decomposition based algorithms. These too are launched as a separate thread allowing the user to perform other tasks.

Adding a branch decomposition based algorithm to JPAD is similar to the procedure above for adding regular algorithms; however, instead of creating the algorithm as a subclass of Algorithm and then wrapping it in a JPADAlgorithm, it must be a subclass of BWAlgorithm and wrapped in an instance of JPADBWAeslgorithm from the decomp.BWAlgorithm and decomp.gui packages, respectively.

An alternative method of adding BWAlgorithms to JPAD is to specify the algorithm in the configuration file, DecompMenu.conf. When the Java Virtual Machine loads the classes for JPAD, the names of the algorithms and locations of the corresponding class files are read from the configuration file and added to the algorithms menu.

A feature of running decomposition based algorithms in JPAD is that JPAD can trace the tree as the algorithm progresses. That is, when a node is being processed (i.e., generating leaf solutions or joining solutions) the node of the decomposition tree is highlighted as well as the partition of the original graph as described in Section 8.3. If the algorithm produces other output to the screen, say, for debugging, the user can easily view the decomposition and identify the active middle sets that are involved in the current solutions.

To aid in this analysis process, the user can decide whether to view node-by-node progress of the algorithm or to let the algorithm run on its own. If node-by-node progress is selected, it can be turned off during the run. In addition, branch decomposition based algorithms can be stopped by the user. This is particularly
useful when the user is looking for a certain criterion and finds it early on in the processing of a large decomposition tree.

8.6 Future Additions

As with any software, JPAD is continually being reevaluated and improved. Future versions will include some or all of the following improvements.

**Zoom** For particularly large graphs, the ability to “zoom in” to view a local portion of either the graph or the branch decomposition is planned.

**Web Enabling** A tool as powerful and versatile as JPAD should be made available for educational and research purposes. Because it is written in Java, it is possible to make the entire application available to be used on the web as a web Applet. It may be beneficial to make different components—for example, the graph displaying component DecompViewer—available as Applets for researchers and educators to be able to display graphs on their own web sites. These Applets could be similar to other web Applets and plug-ins (e.g., flash, shockwave) that are currently in use.

**Incorporated Branch Decomposition Algorithm** Since it is likely that individuals will have graphs for which they do not know the branchwidth or an optimal branch decomposition, it is planned that we will incorporate into JPAD an algorithm or heuristic to find branch decompositions.

**User-Assisted Branch Decomposition Algorithm** To assist in researching techniques for determining optimal branch decompositions, we will add a facility to create branch decompositions with user-assistance. For example, the user may
open a graph and determine successive bipartitions of the graph that correspond to edges in a branch decomposition. Through this process, a branch decomposition can be created and its properties can be studied.

**Freehand Graph Drawing** Given that JPAD’s primary purpose is the analysis and rendering of branch decompositions and algorithms, it does not allow for graphs to be drawn on-the-fly. We plan to implement this feature as well as allowing for graphs to be generated based on user input (e.g., number of vertices, maximum degree).

With its features and capabilities, JPAD is a unique tool that assists research in graph algorithms. It is currently available for free by contacting the author and will soon be available as a web Applet. In addition, planned improvement will increase its usefulness to researchers.
Chapter 9

Conclusions

The main focus of this thesis has been the implementation of serial and parallel algorithms to solve several classic combinatorial optimization problems. In addition, software for analyzing graphs and graph algorithms was created.

9.1 Algorithms

We implemented algorithms using branch decompositions to solve several different graph problems and then extrapolated a core linear framework for programming these and other algorithms. To extend the framework, one must implement methods to:

- generate solutions at the leaves of the decomposition tree.

- check compatibility of partial solutions (which possibly includes rewriting the solution based on middle set intersections); and

- join compatible solutions (maintaining rank for optimization problems).

The framework itself is linear because it is based on tree traversal; the efficiency of an entire algorithm is based on the aforementioned components.

We have shown that the algorithms in this thesis can be separated into three basic categories:

- vertex $k$-partitioning with fixed $k = 2$ as in Chapter 4;

- vertex $k$-partitioning for minimum $k$ as in Chapter 5; and

- path separations as in Chapter 6.
As expected, the two classes of vertex partitioning problems are similar with the most significant difference in the join operation. And, of course, the path partitioning problems are significantly different.

All the vertex partitioning problems have similar compatibility testing procedures. Two partial solutions are compatible if, in both solutions, each terminal vertex is assigned the same value. These assigned values are interpreted as indices of the partition to which vertices are assigned.

The straightforward solution storage method, fixed partitioning, associates each partition with a fixed index. However, if each partition must satisfy the same property regarding its members, then the partitions are isomorphic. In this case, we can use the isomorphic partitioning storage method to reduce the number of solutions that must be stored. As observed in Section 4.3, there is a performance (i.e., running time) impact of using isomorphic partitioning as opposed to fixed partitioning— isomorphic partitioning was noticeably slower. Thus, isomorphic partitioning should be used when the number of solutions being stored is relatively large. For example, in Chapter 5 we argue that the number of distinct colorings of a subgraph defined at a node of the decomposition tree is bounded above by \(\left(\frac{3}{2}\beta(G)\right)^{\beta(G)}\) compared to \(\beta(G)!\) solutions and we therefore should use isomorphic partitioning.

As stated above, the vertex partitioning problems differ in the join procedures. Using the fixed or isomorphic partitioning storage method for the first class of problems, the subroutine is essentially the same. For the second class of problems in which we determine the minimum number of partitions there is enough difference to warrant a new category of problems—this is because the problem is to determine the minimum \(k\) and it is necessary to introduce new indices systematically.

Finally, for the path partitioning problems, we introduced an algorithm that joins two sets of paths to determine if a valid new set of paths is formed. Both the
**Hamiltonian Cycle** and **Disjoint Paths** algorithms use this routine; they differ primarily in how new solutions are validated.

Despite the variety of graph problems solved in this research, we use a single solution data structure in all of the algorithms. For vertex partitioning problems, in the stored solution, the value assigned to a vertex is interpreted as membership of the vertex to a set of vertices. For the path partitioning problems, the values indicate edge and path connectivity.

### 9.2 Parallelization

Using the established algorithmic framework and data structure, we extended all of the algorithms by adding parallelization or multithreading. In Chapter 7, we explained an effective approach to parallelizing the join process. Indeed, for all the large graphs with high branchwidth, actual speed-up was achieved. We noted that this modification to the core algorithm did not require any modifications to the problem specific algorithms. It was also explained that the use of a thread pool (i.e., reusing existing threads) was necessary to achieve the desired speed-up.

In this chapter, we also suggested other possible methods for determining sets of solutions to consider in a parallel join process and identified a critical issue in implementing parallelizing tree traversal. That is, a possible encumbrance from decomposition trees with unbalanced workloads is having idle processors.

### 9.3 JPAD

The foundation classes for the GUI program JPAD were written to have a tool that could manipulate the graph and branch decomposition data structures. This was done prior to the implementation of the algorithms and we continued its development.
concurrently with the design and improvement of the algorithmic framework. The result is a versatile and unique tool.

The most significant novel features are its ability to display a graph and its branch decomposition, and its ability to indicate relations between branch decomposition edges and middle sets and between branch decomposition nodes and the graph separation including the active middle sets. These features are capitalized upon when a branch decomposition based algorithm is run using JPAD.

JPAD is easily extendible and is freely available for research use. Several planned new features are described in Section 8.6. It is freely available by contacting the author.

Robertson and Seymour introduced tree decompositions and branch decompositions in their work regarding Wagner's conjecture. Courcelle and Arnborg, Lagnergren, and Seese prove that over the class of graphs whose treewidth or branchwidth are bounded above by a fixed $k$ any graph problem that can be expressed in monadic second-order logic can be solved in linear-time with respect to the size of the graphs. This thesis presents several linear-time algorithms to solve NP-complete graph problems and a framework and data structure that can be used as a basis for developing new algorithms using branch decompositions. In addition, we provide a graphical tool that can be used for studying the progress of any graph algorithm or branch decomposition based algorithm.
Bibliography


