Cand. Scient. Thesis
in
Informatics

Recognizing weakly chordal graphs

by

Lars Severin Skeide

November, 2002

Department of Informatics
University of Bergen
Norway

Preface

The world consists of challenges and solutions, not problems and limitations. An algorithm is a set of instructions for solving a challenge. A computation is the process of applying an algorithm to a question to obtain an answer. Often an algorithm is implemented on a computer using a programming language, and the computation is performed by the computer.

Algorithms are a field in computer science, where in addition to find solutions to challenges, we want the computations to perform as quickly as possible.

Graph theory is a field in discrete mathematics. One way to think of a graph is as a bunch of dots connected by lines. Surprisingly to many, a mathematical graph is not a comparison chart, nor a diagram with an x- and y-axis, nor a squiggly line on a stock report. For example, every city with an airport can be represented by a dot, and lines connect pair of dots corresponding to airlines between cities. An algorithm can then use the resulting structure to find out how to get from one airport to another. Because mathematicians stopped talking to regular people long ago[6], the dots in a graph are called vertices, and the lines connecting the dots are called edges.

This thesis is in the field of graph algorithms. Our focus is to recognize the class of weakly chordal graphs, which are graphs with a special given structure. We will consider sequential algorithms to be performed by a computer with one processor, and parallel algorithms performed by a computer containing multiple processors. To better enjoy this reading, some knowledge about graphs and algorithms will be an advantage.

Motivation

Chordal graphs are a class of graphs which, among other things, is important for solving sparse linear systems of equations. Weakly chordal graphs are a superset of chordal graphs. Recently, Berry, Bordat and Heggernes[5] established a strong structural relationship between the two classes of graphs, leading to a new recognition algorithm for weakly chordal graphs.

We will study and implement this resulting algorithm. First we will do a sequential implementation whose experimental results will be compared to

the proven time complexity. Second we will do a simple but in practice effective parallel implementation and measure the effect of the parallelization.

The algorithm has never been implemented before, neither sequentially nor in parallel, and even if some details are given in the paper of Berry, Bordat and Heggernes[5], we will have to handle several challenges. Among them are computations of minimal separators yielding a huge separator list, and the computation of the connected components of complement graphs.

Chapter 1 Preliminaries briefly covers several topics essential to the thesis. We introduce basic graph terminology, asymptotic time and space complexity, C++ and its Standard Template Library, and parallel computing.

Chapter 2 Chordal vs. weakly chordal graphs gives a further motivation of the thesis by giving a theoretical background of chordal and weakly chordal graphs. After the definitions of the two classes of graphs and some general properties, we see how to recognize chordal graphs before we finally present the recognition algorithm for weakly chordal graphs introduced by Berry, Bordat and Heggernes[5].

Chapter 3 Sequential implementation then explores how to make an efficient implementation of the recognition algorithm for weakly chordal graphs. Both the data structures and the implementation on a sequential computer are investigated in detail.

Chapter 4 Parallel implementation first introduces the concepts of load balancing and termination detection, before developing a simple but efficient parallel implementation of the studied algorithm.

Chapter 5 Performance results introduces the graphs which the tests will be run on, both graphs that we have generated and graphs available on the Web. Then experimental results for both the sequential and parallel implementation are presented.

Chapter 6 Concluding remarks gives an overview of the work and results of the thesis, ending with possible future work.

Acknowledgments

First I would like to thank my excellent supervisor associate professor Pinar Heggernes for her teaching, support, and guidance for two years. Her patience has been tremendous, especially when my focus has been in other directions.

I would also like to thank Yngve Villanger for his help in generating graphs, Oddvar Christiansen for his help in the use of MatLab, and Jill Iren Berge for reading this thesis and correcting many writing errors.

Finally, thanks to all my friends for five interesting years at the University of Bergen. You have given me support and help in turning my mind away from my studies. It has been a great pleasure.

Bergen, November 2002

Lars S. Skeide

Contents

Preface **3**

1 Preliminaries **11**
- 1.1 Basic graph terminology . . . 11
- 1.2 Sequential time and space complexity . . . 13
- 1.3 C++ and the Standard Template Library . . . 14
  - 1.3.1 Abstract container types . . . 14
- 1.4 Parallel computing . . . 17
  - 1.4.1 Measure of performance . . . 17
  - 1.4.2 Overhead . . . 18
  - 1.4.3 Parallel programming models . . . 19
  - 1.4.4 IBM p690 Regatta Turbo system . . . 20

2 Chordal vs. weakly chordal graphs **21**
- 2.1 Chordal and weakly chordal graphs are perfect . . . 22
- 2.2 The relationship between chordal and weakly chordal graphs 23
- 2.3 Recognizing chordal graphs . . . 24
- 2.4 Recognizing weakly chordal graphs . . . 26

3 Sequential implementation **29**
- 3.1 How to make the algorithm to run in $O(m^2)$ . . . 29
  - 3.1.1 A direct approach will fail . . . 29
  - 3.1.2 Solution: Do not re-compute the co-connected components . . . 30
- 3.2 Data structures . . . 31
  - 3.2.1 The graph . . . 33
  - 3.2.2 The neighborhood of an edge . . . 34
  - 3.2.3 The labeled vertices . . . 35
  - 3.2.4 The separators . . . 35
  - 3.2.5 The co-connected components . . . 36
  - 3.2.6 Total space complexity . . . 36
- 3.3 Algorithms . . . 37
  - 3.3.1 Computing $N(ab)$ and the labeling . . . 37

3.3.2 Computing the minimal separators 37
3.3.3 Sorting the separator list 44
3.3.4 Identify original separators 46
3.3.5 Computing the co-connected components 46
3.3.6 Checking the labeling in the co-connected components of a separator 52
3.3.7 Total time complexity 52

4 Parallel implementation 54
4.1 Load balancing 55
4.1.1 Dynamic load balancing of the best known sequential algorithm 56
4.1.2 Simple static load balancing 57
4.2 Termination detection 57
4.2.1 The possibility for superlinear speedup 58
4.3 Final parallel implementation 59
4.3.1 The load balancing in detail 62
4.3.2 The termination detection in detail 63

5 Performance results 67
5.1 Test graphs 67
5.1.1 Elimination game 68
5.1.2 Matrix market 70
5.2 Sequential experimental results 72
5.3 Parallel experimental results 73

6 Concluding remarks 80
6.1 Overview of our work 80
6.2 Future work 81

Bibliography 82

List of Figures

1.1 A graph, a complement graph, and subgraphs 12

2.1 Schematic overview of chordal vs. weakly chordal graphs . . . 21
2.2 Examples of chordal and weakly chordal graphs 22
2.3 C5 isomorphic to its complement 23
2.4 C6 and its complement 24
2.5 Example of a weakly chordal graph 28

3.1 Graph representation 34
3.2 Computation of minimal separators 39
3.3 Data structures when computing the minimal separators . . . 43
3.4 Computation of co-connected components of a graph G . . . 51

4.1 Load balancing 55
4.2 The possibility for superlinear speedup 62

5.1 Verifying weak chordality of chordal graphs with 5000 vertices and increasing number of edges. 74
5.2 Verifying weak chordality of chordal graphs with 10000 vertices and increasing number of edges. 74
5.3 Square and linear approximation to the run time ($n = 5000$) . 75
5.4 Square and linear approximation to the run time ($n = 10000$) 75
5.5 Speedup for increasing number of processors when recognizing graph S3DKQ4M2 for weak chordality 77
5.6 Speedup for increasing number of processors when recognizing graph BCSSTK13 for weak chordality 77
5.7 Speedup for increasing number of processors when recognizing graphs with 10000 vertices for weak chordality 79

List of Algorithms

2.1 Chordal graph recognition 1 25
2.2 Chordal graph recognition 2 26
2.3 Weakly chordal graph recognition 28
3.1 LB-simpliciality of an edge 30
3.2 $O(m^2)$ time weakly chordal graph recognition 32
3.3 Computing $N(ab)$ and the labeling 38
3.4 General depth-first search 40
3.5 Computing the minimal separators 42
3.6 Sorting the global separator list 45
3.7 Comparing two separators 46
3.8 Computing the co-connected components of a separator . . . 48
3.9 Computing the co-connected components of a separator in $O(m)$ time 50
3.10 Checking the labeling in the co-connected components of a separator 53
4.1 Parallel weakly chordal graph recognition with static load balancing 57
4.2 Parallel weakly chordal graph recognition and termination detection 59
4.3 Final parallel weakly chordal graph recognition 60
4.4 LB-simplicial(my_E) 61
4.5 Detailed load balancing 64
5.1 Elimination game 68
5.2 Tree generator 69
5.3 Weakly chordal graph generator 71

Chapter 1

Preliminaries

In this first chapter we will consider some topics essential for the understanding of the thesis. We start out with basic graph terminology, followed by a short introduction to asymptotic time and space complexity. Then we take at look at C++ and its Standard Template Library, before ending with an outline over parallel computing.

1.1 Basic graph terminology

A ***graph*** is a set of points with lines connecting some of the points. The points are called ***vertices***[1], and the lines are called ***edges***. More formally, a graph G is a pair (V, E), where V is the set of vertices and E is the set of edges. We denote the size of V by n, and the size of E by m, thus $|V| = n$ and $|E| = m$. All graphs in this work are undirected, and an edge $e \in E$ is then an unordered pair (u, v), where $u, v \in V$ and $u \neq v$. We will often denote the edge (u, v) simply by uv when there is no ambiguity. Moreover, we will regard the notations $v \in V$ and $e \in E$, knowing v is a vertex and e is an edge, equivalent with $v \in G$ and $e \in G$. Figure 1.1(a) is a pictorial representation of a graph $G = (V, E)$, where $V = \{1, 2, 3, 4, 5\}$ and $E = \{(1, 2), (1, 5), (2, 3), (3, 5), (4, 5)\}$.

The ***complement of a graph*** G is denoted $\overline{G}$ and contains the same vertices as G, but precisely those edges which are not in G, $(a, b) \in \overline{G} \Leftrightarrow (a, b) \notin G$. The graph $\overline{G}$ in Figure 1.1(b) is the complement of the graph in (a).

A ***subgraph*** G' of $G = (V, E)$ is a graph that contains some of the vertices, and some of the edges of G. An ***induced subgraph*** G' contains a subset of V as vertices, and *all* the edges between these vertices that are present in G. $G(A)$ denotes the subgraph induced by a vertex set $A \in V$, and $\overline{G}(A)$ denotes the subgraph induced by A in the complement of G. This

[1] Also called nodes

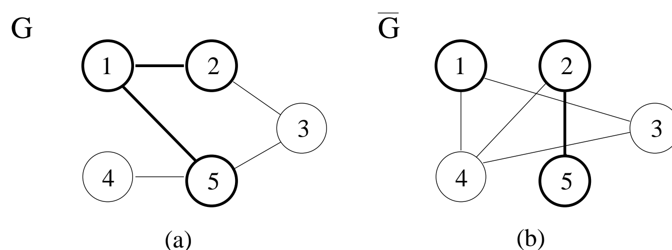

Figure 1.1: The graph $\overline{G}$ in (b) is the complement of the graph G in (a). The subgraphs $G(A)$ and $\overline{G}(A)$, $A = \{1, 2, 5\}$, are shown darker.

can be seen in Figure 1.1, where $A = \{1, 2, 5\} \in V$, and the vertices and edges belonging to the subgraphs are shown darker.

A vertex u is ***neighbor*** of or ***adjacent*** to another vertex v if there is an edge joining them, $(u, v) \in E$, and we will then say that u ***sees*** v. The ***neighborhood*** of a vertex x, $N(x)$, is the set of all vertices y that x sees, $N(x) = \{y \neq x \mid xy \in E\}$. The ***degree*** of the vertex is the number of edges adjacent to it, or equivalent, the size of the neighborhood, $|N(x)|$. The neighborhood of a set of vertices A is the set of all vertices seen from the vertices in A, $N(A) = \cup_{x \in A} N(x) - A$. For a set of vertices A, a ***confluence point*** is a vertex of A that sees all the vertices in $N(A)$.

In order to have analogous definitions for edges, we regard an edge ab as a set of vertices $\{a, b\}$. Then we can let $N(ab)$ denote the neighborhood of an edge ab. The edge sees a vertex x if either a or b sees x.

A ***path*** in a graph is a sequence of vertices connected by edges. The ***length*** of the path is the number of edges in it. If a path starts and ends in the same vertex, the path is a ***cycle*** denoted Ck, where k is the length of the cycle. When there are no cycles in a graph, the graph is ***acyclic***. A ***chord*** in a cycle is an edge between two non-consecutive vertices in the cycle. An induced chordless cycle on five or more vertices in a graph is called a ***hole***, and the complement of a hole is an ***antihole***. A connected acyclic graph is a ***tree***.

When all vertices in a graph are pairwise adjacent, the graph is ***complete***. The number of edges m is then $(n^2 - n)/2$. A ***clique*** in a graph is a complete subgraph. When m is much smaller than $(n^2 - n)/2$,[2] the graph is said to be ***sparse***, otherwise it is ***dense***.

A graph is ***connected*** if every vertex u can be reached by any other vertex v through a path, and ***disconnected*** otherwise. Any disconnected graph G can decomposed into maximal connected subgraphs (not a subgraph of any other connected subgraph), each of which is a ***connected component*** of G. The connected components of the complement graph $\overline{G}$ are denoted

[2] More precise $m = O(n)$

as the ***co-connected components*** of G.

For $X \subseteq V$, $\mathscr{C}(X)$ denotes the set of connected components of $G(V-X)$. $S \subset V$ is called a ***separator*** if $|\mathscr{C}(S)| \geq 2$, an ***ab-separator*** if a and b are in different connected components of $\mathscr{C}(S)$, a ***minimal ab-separator*** if S is an ab-separator and no proper subset of S is an ab-separator, and a ***minimal separator*** if there is some pair $\{a, b\}$ such that S is a minimal ab-separator. A component C of $\mathscr{C}(S)$ is called ***full*** if $N(C) = S$.

Two graphs G and H are ***isomorphic*** if there is a one-one correspondence between the vertices of G and those of H such that the number of edges joining any two vertices of G is equal to the number of edges joining the corresponding vertices of H. The graphs shown in Figure 2.3 in Chapter 2 are isomorphic.

1.2 Sequential time and space complexity

When working with an algorithm it is of great interest to know how much resources we need to execute the algorithm, both theoretical and practical when implemented on a computer. To achieve the best performance in practice when we run our implementations, we use computational complexity theory as a tool to analyze algorithms. Computational complexity theory is an investigation of the time, space[3], or other resources required for solving computational problems[21].

In our analysis we focus on the time and space complexity, expressed as a function of the size of the input to the algorithm. For instance, if the input is a graph, which often is our case, the size of the input may depend on the number of vertices, number of edges, or a combination of these.

Because the exact running time or amount of space often is a complex expression, we use the familiar ***big-0 notation*** to establish an ***asymptotic*** upper bound for the computational complexity. We do so by only considering the highest order term of the expressions, disregarding both the coefficient of that term and any lower order terms, because the highest order term dominates the other terms on large inputs.

For instance, if an algorithm takes a graph of n vertices and m edges as input, we say that the algorithm has a time complexity of $O(n^2 + m)$ if the exact theoretical running time is never *more* than a constant times $n^2 + m$ for large enough n and m. Similarly, using the same example, the time complexity is $\Omega(n^2 + m)$ if the time required is never *less* than a constant times $n^2 + m$ in the worst case of input. If the algorithm is both $O(n^2 + m)$ and $\Omega(n^2 + m)$ we say that the time complexity is $\Theta(n^2 + m)$.

[3] Also called memory

1.3 C++ and the Standard Template Library

To implement our algorithms on a computer, we need a programming language. We will use ***C++***, which is a newer version of ***C***, a powerful and efficient language developed at AT&T's Bell Labs in the early 1970s[15].

Although C++ is an object-based and object-oriented high level programming language, it supports several fundamentally different programming paradigms. Our implementations in this thesis can be viewed as mainly being in the procedural paradigm, where a problem is directly modeled by a collection of algorithms. However, algorithms are only one of the two primary aspects when programming. We also have to consider the matter of the collection of data, the data structures, against which the algorithms are run to provide the solutions.

For the data structures we will extensively use the Standard Template Library, STL, which is a part of the C++ standard library. The STL is actually an important tool in the generic programming paradigm, using parameterized types or template classes. A class template is a predescription for creating a class in which one or more types or values are parameterized.

Many of the basic algorithms and data structures are contained in the STL, but decoupled from each other. It is more accurate to think of the STL as a library of generic algorithms, which also contain data structures for the algorithms to operate on, but we will rather use the STL as a container class library, which our own algorithms operates on. Our use of the generic programming paradigm through the STL is therefore limited to the data structures.

1.3.1 Abstract container types

Containers are objects which contain and manage other objects and provide iterators that allow the contained objects (elements) to be addressed. All of the STL's predefined container classes are models of `Sequence` or `Associative Container`, and all of them are templates that can be instantiated to contain any type of object.

Sequences

A sequence container holds an ordered collection of elements of a single type. The two primary sequence containers are the `vector` and `list` types.

Many of our considerations about data structures concerns the question about whether to use list or a vector. Originally the question has been whether to use a list or an array. We will see that both arguments and conclusions in the list versus vector discussion differ from an original list versus array discussion. Usually, the main rule has been to use an array if we at compile-time know the number of elements to be stored, and otherwise use a list. However, an array in the object-oriented C++ is not a

first-class citizen of the language. It is inherited from the C language and reflects the separation of data and the algorithms that operate on that data that are characteristic of the procedural paradigm. As mentioned we will use the data structures from the generic programming paradigm. Instead of array we therefore use vector, which is is an abstraction of an array, and in addition to fulfill the needs of an array provides several useful member functions. For instance, the vector class supports operations such as assignment of one vector to another, comparison of two vectors for equality, and questions about its size.

Now, the criteria for choosing between a list and a vector is mainly concerned with the insertion characteristics and subsequent access requirements of the elements, due to the fact that *both a list and a vector grows dynamically*. We will now take a closer look into some of the criteria:

- If we require random access to the elements, a vector is the clear choice over a list. A vector represents a contiguous area of memory in which all elements are stored consecutively. Random access is then very efficient because each access is a fixed offset from the beginning of the vector, and therefore a constant time operation. For a list however, random access to an element requires traversal of the intervening elements, which in worst case is traversing all the elements in the list. In addition, there is the space overhead of the two pointers per element.

- If we need to insert and delete elements other than at the end, a list is the clear choice over a vector. A list represents noncontiguous memory, where each element is doubly linked through a pair of pointers that address the elements to the front and back. Insertion and deletion of elements at any position is therefore efficient, since only pointers must be reassigned and no element need to be moved by copying. The operation is therefore performed in constant time.

- If we only need to insert and remove elements at the end, a vector may be the best choice. For a vector to grow dynamically, it must allocate enough memory to all the elements, copy the old elements into the new memory, and deallocate the old memory, before it can add the new elements. Therefore it may seem that a list may be a better choice than a vector. In practice, however, for each time the vector allocates more memory, it allocates more than it needs. How much it allocates is C++ implementation-defined. In our version it starts allocating for one element no matter the size of the element, and doubles the capacity each time extra memory is needed. In this sense it is important to distinguish between a vector's size and its capacity. The size, retrieved by invoking its `size()` operation, is the actual number of elements in the vector. On the other hand, capacity, retrieved by its `capacity()` operation, is the total number of elements for which there is allocated

memory for. That is, the number of elements a vector can contain before it needs to regrow itself.

This way of dynamically growing is much more efficient than the way a list is growing, *as long as the objects are small and simple.*

One last comment is the possibility through the `reserve()` operation to allocate an upper bound for how much memory we need. A drawback may be that we have a serious overhead, occupying needed memory when dealing with a huge graph. Also, in [18], Lippman and Lajoie question the use of of the `reserve()` operation, because tests seem to show that adjusting the capacity cause the performance to degrade, as long as the objects inserted are small and simple. If the objects are large or complex, we will store them indirectly by pointers. Therefore we will not adjust a vector's capacity through the `reserve()` operation.

Associative containers

An associative container supports efficient query to the presence and retrieval of an element. The two primary associative container types are the `map` and the `set`. A map is a key/value pair: the key is used for lookup, and the value contains the data we wish to use. A set contains a single key and supports the efficient query of whether it is present. Both the map and the set can contain only a single occurrence of each key.

When inserting elements into an associative container, the container places them in their ordered positions. The obvious reason for this, is to look up element faster in $O(\log n)$ time instead of $O(n)$ time. In fact, all lookup, insertion, and deletion operations perform in $O(\log n)$ time.

Iterators

An iterator is a generalization of a pointer; it is an object that points to another object. Iterators provide a general method of successively accessing each element within any of the sequential or associative container types. If an iterator points to some object in a range of objects, and the iterator is incremented, then it will point to the next object in that range. Iterators are actually an interface between the algorithms and the data structures, and they make it possible to decouple those two primary aspects of programming when using the STL.

More concrete, each container type provide both a `begin()` and an `end()` member function, returning an iterator, where `begin()` addresses the first element of the container, and `end()` addresses 1 past the last element. If `iter` is an iterator into any container type, then `iter++` advances the iterator to address the next element, and `*iter` returns the value of the element.

For clarity, elements of a vector `V` can be accessed in three different ways. Through iterating the range `[V.begin(), V.end())` or `[V.rbegin(), V.rend())`, which both is equal but in reverse order, or if n is an integer, `V[n]` returns the nth element. `V[n]` is actually a shorthand for `*(V.begin()+n)`.

1.4 Parallel computing

Since there seems to exist a constant desire to always solve larger problems, there exists a continual demand for greater computational speed. For the conventional serial computer however, there is a fundamental physical limitation imposed by the speed of light, which makes further improvements in the speed of such machines definitely.

The solution to this challenge has been parallel computing, where computational problems are solved by several processors that are able to work cooperatively.

There are both scientific and commercial examples of applications that demand greater computational speed, and where parallel computing is in use. Common examples from science are numerical simulations of complex systems such as weather, climate and chemical reactions. A good commercial example is the entertainment industry, with virtual reality and video servers serving thousands of simultaneous requests for real-time video.

The study of parallel algorithms and parallel computing are due to these described above and also other trends become of increase interest. In addition is the algorithm we will study parallel in nature, like many tasks in the real world; If there is a major job to be done, it is often better with several workers working on it than just one. We will therefore also do a parallel implementation of the studied algorithm.

1.4.1 Measure of performance

As described in section 1.2, the execution time of a sequential algorithm is usually evaluated in relation to the size of its input. For a parallel algorithm however, the execution time depends not only on input size, but also on the architecture of the parallel computer and the number of processors. Because we will not focus on parallel architectures, we will mainly consider practical test results.

To measure the performance and describe the qualities of a parallel implementation, we introduce the concepts serial and parallel run time, cost, cost-optimal, speedup, efficiency, scalability, concurrency and locality.

The ***serial run time*** of a program is the elapsed time between the beginning and the end of its execution on a sequential computer[14], which wee denote T_s. The ***parallel run time***, T_p, is then the time elapsed from the parallel computation starts to the moment that the last processor finishes execution. If the number of processors is p, then the total time spent in

solving a problem summed over all processors is pT_p, which we denote as the ***cost*** of the parallel solution. The solution is said to be ***cost-optimal*** if the cost on a parallel computer is proportional to the execution time of the fastest-known sequential algorithm on a single processor.

When evaluating a parallel program, we are interested in how much performance gain is achieved by parallelizing a sequential implementation. Both speedup and efficiency is used for this.

We define the ***speedup*** S as the ratio of the serial run time of the best sequential algorithm for solving a problem to the time taken by the parallel algorithm to solve the same problem on p processors identical to the one used by the sequential algorithm[14], that is, T_s/T_p. In other words, the speedup says how many times faster we can solve a problem using p processors instead of one. Under perfect conditions, the maximal speedup should therefore theoretically be p ($\frac{T_s}{T_s/p}$), which we call *linear speedup*.

However, a speedup greater than p is sometimes observed in practice, which we later will see also can happen with our parallel implementation. Then we have *superlinear speedup*. Actually, this is formally a contradiction, because speedup by definition is computed with respect to the best sequential algorithm. In practice, superlinear speedup may be caused by extra memory in a parallel computer, or working with a search algorithm[22].

Efficiency E is a measure of the fraction of time for which a processor is usefully employed; it is defined as the ratio of speedup to the number of processors[14], $E = \frac{S}{p}$, $0 < E \leq 1$. The efficiency equals 1 when we have linear speedup, so with superlinear speedup it can actually exceed 1.

The ***scalability*** of a parallel algorithm is its ability to achieve performance proportional to the number of processors. For instance, the scalability is very good if the implementation has linear speedup for increasingly number of processors.

Concurrency refers to the ability to perform many actions simultaneously, and is essential for a program to run efficiently on many processors.

Locality means a high ratio of local memory access, so there is little need for interprocessor communication.

1.4.2 Overhead

Linear speedup is achieved when it is possible to divide the work to be done equally on p processors, $\frac{T_s}{T_s/p} = p$. Then we have maximal concurrency and locality, which corresponds to perfect conditions. The total time pT_p then equals T_s.

Such conditions are rare, due to a variety of overheads associated with parallelism. We mention three factors of overhead which limit the speedup.

Load imbalance In many parallel computations it is difficult to predict the size of the subtasks assigned to various processors. If different pro-

cessors have different work loads, some processors may be idle during part of the time that others are working on the problem. This includes the time when only one processor is active on inherently serial parts of the computation.

Interprocessor communication Communication time for sending messages among processors. The time spent on transferring data between processors is often the most significant source of overhead.

Extra computation Computation in the parallel implementation not appearing in the sequential version. For instance recomputing constants locally, or calculate load balancing.

1.4.3 Parallel programming models

We will now describe the two main programming models, together with two well known corresponding specifications, one among them we will use in a parallel implementation.

Shared memory programming (SMP) and OpenMP

In the shared memory programming model, programmers view their programs as a collection of processes accessing a central pool of shared variables, which they read and write asynchronously.

OpenMP[3] is a widely used standard specification containing compiler directives, library functions, and environment variables for shared programming parallelism. Parallel execution is achieved by executing loops or sections of code in parallel. For this to be possible, the order of the loops or sections should not influence the answer.

When all the working processors need access to the same data, this model has a clear advantage, since only one copy is necessary. It is also quite simple to parallelize a sequential program using SMP, in order to minimize the development time.

Message passing programming (MPP) and MPI

Message passing is probably the most widely used parallel programming model today. Programmers view their programs as a collection of processes with private local variables and the ability to explicitly send and receive data between processes by passing messages.

The Message Passing Interface or MPI was released in 1994[1] and extended in 1997[2]. MPI is a specification of a standard set of library functions useful to programmers writing portable message-passing programs in Fortran, C or C++. It is designed for high performance and scalability, and has succeeded in becoming widely used.

The essentials when a MPI program is executed on a parallel computer can be described in three steps[19]:

1. The user issues a directive to the operating system which has the effect of placing a copy of the executable program on each processor.

2. Each processor begins execution of its copy of the executable.

3. Different processes can execute different statements by branching within the program. Typically the branching will be based on process ranks.

1.4.4 IBM p690 Regatta Turbo system

The "IBM p690 Regatta Turbo system" is the supercomputer where we have done our implementation and run our tests on. The supercomputer consists of three 32-way eSeries p690 Regatta SMP nodes, with a total of 96 Power4 processors (1.3 GHz) and has a total of 192 Gigabyte memory. When the Regatta system came alive in January 2002, it had the most powerful CPU's of the supercomputers in Norway.

The Regatta system is a shared-address-space computer, which is naturally suited to the described shared memory programming model. However, it can also be programmed using the message passing programming model, by modeling a message-passing computer without loss of performance.

Chapter 2

Chordal vs. weakly chordal graphs

The class of chordal graphs and the class of weakly chordal graphs have several similarities. Recently, as mentioned in the preface, Berry, Bordat and Heggernes[5] also established a strong structural relationship between chordal and weakly chordal graphs where they applied a variant of Lekkerkerker and Boland's recognition algorithm for chordal graphs to the class of weakly chordal graphs. This yield a new characterization of the class, and the motivation and goal of this thesis is to study practical implementations of this characterization.

After the definitions of the two mentioned classes of graphs, we go through some of the similarities of the two classes in addition to some of the recognition algorithms for chordal graphs. We end the chapter by presenting the recognition algorithm for weakly chordal graphs which will be further studied and implemented in the next chapters. An overview of this chapter is outlined in Figure 2.1.

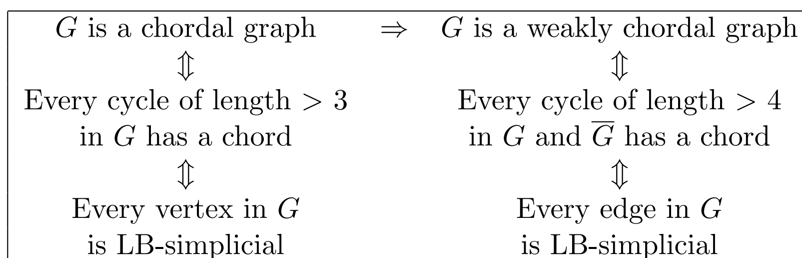

Figure 2.1: Schematic overview of chordal vs. weakly chordal graphs (some of the properties)

Definition 2.1 *A graph $G = (V, E)$ is* chordal[1] *iff every cycle of length $>$ 3 in G has a chord.*

Definition 2.2 *A graph $G = (V, E)$ is* weakly chordal[2] *iff every cycle of length >4 in G and the complement graph $\overline{G}$ has a chord.*

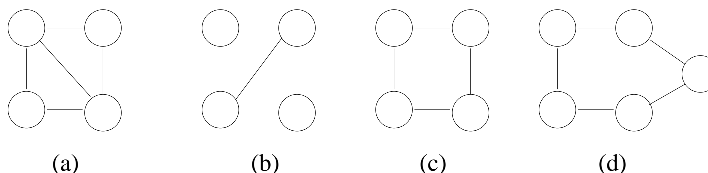

Figure 2.2: The graph in (b) is the complement of the graph in (a), which is both chordal and weakly chordal. In (c), the graph is weakly chordal but not chordal. For the graph in (d), it is neither chordal nor weakly chordal.

In Figure 2.2(a), the longest chordless cycle is of length 3, and the graph is therefore chordal. In addition, the complement of the graph seen in Figure 2.2(b), has no cycles at all, and the definition of a weakly choral graph is fulfilled. The graph in Figure 2.2(a) is therefore also weakly chordal. Later, in Section 2.2, we will show that this is a general result, i.e. the class of weakly chordal graphs is a superset of the class of chordal graphs.

The graph in Figure 2.2(c) is weakly chordal but not chordal. The longest cycle it contains is of length 4, excluding it from being a chordal graph. The complement of the graph does not have any cycles, and the graph is therefore weakly chordal.

In Figure 2.2(d), the graph is neither chordal nor weakly chordal since it contains a chordless cycle of length 5.

2.1 Chordal and weakly chordal graphs are perfect

Weakly chordal graphs were introduced by Hayward[9] in 1985 under the motivation that they were perfect, just as the class of chordal graphs.

Definition 2.3 *A graph $G = (V, E)$ is perfect if $\omega(G(X)) = \chi(G(X))$ for all $X \subseteq V$.*

The *clique number* of a graph G is the size of the largest clique in G, and is denoted by $\omega(G)$. The *chromatic number* of a graph G, is the smallest number c, where all the vertices in G are colored in one of the c colors, but no adjacent vertices have the same color, and is denoted by $\chi(G)$.

[1] Also called triangulated
[2] Also called weakly triangulated

In other words, the definition says that a graph G is perfect if, for each induced subgraph X of G, the chromatic number of X equals the size of the largest clique of X.

Perfect graphs are of interest partly because a number of optimization problems can be formulated as coloring problems on perfect graphs.

2.2 The relationship between chordal and weakly chordal graphs

Hayward shows in his original paper[9] that chordal graphs are weakly chordal. In this section we state this result as a theorem, yielding that the class of weakly chordal graphs is a superset of the class of chordal graphs.

Theorem 2.4 [9] *A graph $G = (V, E)$ is chordal $\Rightarrow$ G is weakly chordal.*

Before we give a proof, we make two observations.

Observation 2.5 *A chordless cycle of length 5 is isomorphic to its complement.*

The observation is illustrated in Figure 2.3.

Observation 2.6 *The complement of every chordless cycle of length $\geq$6 has a chordless cycle of length 4*

This observation is illustrated in Figure 2.4. Figure 2.4(b) shows some of the edges of the complement graph of the chordless cycle of length 6 in Figure 2.4(a). We see that any subgraph induced by the endpoints of two non-consecutive edges of the cycle of length 6 will make a chordless cycle of length 4 in the complement graph. A chord in such a cycle will not exist because it will be a part of the cycle in (a). The same is true for all chordless cycles with length >6.

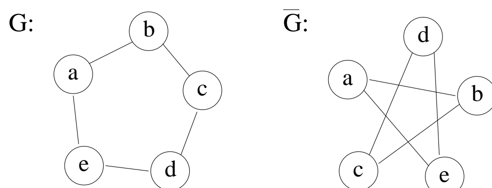

Figure 2.3: G and $\overline{G}$ are isomorphic

Hayward[9] briefly sketches a proof of Theorem 2.4, which we now give more detailed.

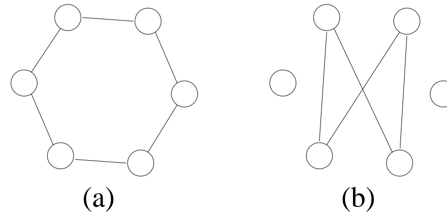

Figure 2.4: The complement of every chordless cycle of length ≥6 has a chordless cycle of length 4.

Proof. We use the definition of weakly chordal graphs on a given chordal graph G. That is, neither G nor $\overline{G}$ can contain a cycle of length >4 without a chord.

It is obvious that G does not have any chordless cycle of length >4, according to the definition of chordal graphs.

Since G does not have any chordless cycle of length 5, $\overline{G}$ cannot have any chordless cycle of the same length, according to Observation 2.5. Neither does G contain any chordless cycle of length 4, and according to Observation 2.6, $\overline{G}$ cannot have any chordless cycle of length ≥6. The longest chordless cycle of $\overline{G}$ is then at most of length 4.

The chordal graph G is therefore also weakly chordal.

□

In addition is it clear that $\overline{G}$ is also weakly chordal, since we have seen that neither $\overline{G}$ nor $\overline{\overline{G}} = G$ contain a chordless cycle of length ≥4.

2.3 Recognizing chordal graphs

There are several different ways to characterize chordal graphs, and thus several corresponding ways to recognize them. We will here mention two ways of recognizing chordal graphs. One uses the notion of a simplicial vertex, and the other the definition of an LB-simplicial vertex.

Definition 2.7 *A* vertex *is called* simplicial *if its neighborhood induces a clique.*

This definition gives the basis for one characterization of chordal graphs:

Characterization 2.8 [7] *Any non-complete chordal graph has at least two non-adjacent simplicial vertices.*

Since all induced subgraphs of a chordal graph also are chordal, we can now give our first algorithm that recognizes chordal graphs. This is given in Algorithm 2.1 [8].

Lekkerkerker and Boland[16] gave another characterization:

Algorithm 2.1 Chordal graph recognition 1

$\rightarrow$ A connected graph $G = (V, E)$
$\leftarrow$ An answer to the question: "Is G chordal?"
$G' = G$
while $\exists$ a simplicial vertex in G' **do**
  Find a simplicial vertex $u \in G'$;
  Remove u and all edges uv, $v \in G'$, from G';
end while
if $G = \emptyset$ **then**
  return (G is chordal);
else
  return (G is not chordal);
end if

Characterization 2.9 [16] *A graph is chordal iff for every vertex x, all the minimal separators included in $N(x)$ are cliques.*

To simplify the notation, Berry, Bordat, and Heggernes[5] introduced the definition of a LB-simplicial vertex (the abbreviation LB refers to Lekkerkerker and Boland).

Definition 2.10 [5] *A* vertex *is* LB-simplicial *if all the minimal separators included in its neighborhood are cliques.*

We then reformulate Characterization 2.9 to a theorem. We also give a proof of this theorem which uses our notation, much simpler than the original proof.

Theorem 2.11 [16] *A graph $G = (V, E)$ is chordal iff every vertex of V is LB-simplicial.*

Proof. $\Rightarrow$: Let $G = (V, E)$ be chordal and let $a \in V$ be any vertex of G. Let s be any minimal separator of G included in the neighborhood of a, separating a from a vertex $b \in V$. We let A and B denote the connected components of $\mathscr{C}(s)$ containing respectively a and b. Further, let x and y be any two vertices of s. We will show that (x, y) must be an edge of G, such that s is a clique and a therefore is LB-simplicial.

First we observe that there must exist a path between x and y through vertices belonging to A. Let p_1 be a shortest such path. In the same way we have a shortest path p_2 between x and y through vertices of B. Joined together, p_1 and p_2 make a cycle of length ≥ 4. Since G is chordal, this cycle must have a chord. Since no edges exist between A and B, the edge (x, y) must be present and is a chord in the cycle. s is then a clique and a is LB-simplicial.

$\Leftarrow$: Let $G = (V, E)$ be a graph where all vertices are LB-simplicial. Assume that G is not chordal, and let $z_1, z_2, \dots, z_k, z_1$ be a chordless cycle where $k \geq 4$. Consider the vertex z_1, and let C_1 be the component of $\mathscr{C}(\{z_1\} \cup N(z_1))$ containing the vertices $z_3, \dots, z_{k-1}$. The neighborhood of C_1, $N(C_1)$, is a minimal separator in the neighborhood of z_1, $N(z_1)$, such that $N(C_1) \subseteq N(z_1)$. Since $\{z_2, z_k\} \subseteq N(C_1)$ and z_1 is LB-simplicial, the edge (z_2, z_k) has to exist, contradicting that the given cycle is chordless.
□

We can now form an algorithm based on the characterization of Lekkerkerker and Boland, which is shown i Algorithm 2.2.

Algorithm 2.2 Chordal graph recognition 2

→ A connected graph $G = (V, E)$
← An answer to the question: "Is G chordal?"
for all $v \in$ G **do**
  if v is not an LB-simplicial vertex **then**
    return (G is not chordal);
  end if
end for
return (G is chordal);

For clarity we give a last characterization of chordal graphs:

Characterization 2.12 [7] *A graph G is chordal iff every minimal separator of G is a clique.*

2.4 Recognizing weakly chordal graphs

Like chordal graphs, there are several different ways to characterize and recognize weakly chordal graphs. We will concentrate on recognizing weakly chordal graphs by the definition of a *LB-simplicial edge*, which is the characterization we will use for practical implementation.

We start with two observations done by Hayward[10] and Kratsch[13]:

Observation 2.13 [10] *Chordal graphs can be generated by repeatedly adding a* vertex *which is not the middle vertex of a chordless path on 3 vertices, while weakly chordal graphs can be generated by repeatedly adding an* edge *which is not the middle edge of a chordless path on 4 vertices.*

Observation 2.14 [13] *In a chordal graph $G = (V, E)$, for every minimal separator s, every component C of $G(V - s)$ contains a* confluence point*, while in a weakly chordal graph G, for every minimal separator s, every full component C contains either a confluence point or a* confluence edge[3].

[3] An edge e such that $N(C) \subseteq N(e)$

These observations show that an *edge* in a weakly chordal graph plays a role similar to a *vertex* in a chordal graph.

Now we define the notion of an s-saturating edge, which is a stronger version of a confluence edge.

Definition 2.15 [11] *Given a set s of vertices, an edge e of $G(V-s)$ is said to be* s-saturating *if, for each connected component s_j of $\overline{G}(s)$, at least one endpoint of e sees all vertices of s_j.*

As Kratsch (Obs. 2.14), Hayward shows that in each full component of a minimal separator in a weakly chordal graph, there is either a confluence point or an s-saturating edge.

Further, in their work for a new characterization for weakly chordal graphs, Berry, Bordat, and Heggernes[5] define the notion of a LB-simplicial edge based on the role such an edge plays in a weakly chordal graph.

Definition 2.16 [5] *An* edge e *is* LB-simplicial *if for each minimal separator s included in the neighborhood of e, e is s-saturating.*

Now we present their theorem yielding a new characterization of weakly chordal graphs:

Theorem 2.17 [5] *A graph $G = (V, E)$ is weakly chordal iff every edge of E is LB-simplicial.*

To prove this theorem, they introduce two lemmas in addition to apply one theorem from the paper where Hayward introduced weakly chordal graphs.

Lemma 2.18 [5] *In a given graph G, an edge that belongs to a hole cannot be LB-simplicial.*

Lemma 2.19 [5] *In a given graph G, each antihole contains an edge that is not LB-simplicial.*

Theorem 2.20 [9] *Let G be a weakly chordal graph, and let s be a minimal separator of G such that $\overline{G}(s)$ is connected. Then in each full component C of $\mathcal{C}(s)$, there is a vertex that sees all the vertices of s.*

Based on these results, we can give their proof of Theorem 2.17:

Proof.[5] $\Leftarrow$: Let G be a graph in which every edge is LB-simplicial. Then by Lemma 2.18 G cannot contain a hole, and by Lemma 2.19 G cannot contain an antihole. Thus G must be weakly chordal.

$\Rightarrow$: Let G be a weakly chordal graph, and suppose some edge ab fails to be LB-simplicial. Let $s = N(C)$ be a minimal separator contained in the neighborhood of ab for which ab fails to be s-saturating, let s_1 be a connected

component of $\overline{G}(s)$ such that neither a nor b sees all the vertices of s_1, and consider the subgraph G' induced by $C \cup s_1 \cup ab$. As any subgraph of a weakly chordal graph is itself weakly chordal, G' must be weakly chordal. s_1 is a minimal separator of G', with 2 full components, {a,b} and C, and $\overline{G'}(s_1)$ is connected. Neither a nor b sees all the vertices of s_1, which contradicts Theorem 2.20.
□

Now we form Algorithm 2.3 based on Theorem 2.17, and give an example of an LB-simplicial edge.

Algorithm 2.3 Weakly chordal graph recognition

→ A connected graph $G = (V, E)$
← An answer to the question: "Is G weakly chordal?"
for all $e \in$ E **do**
  if e is not an LB-simplicial edge **then**
    return (G is not weakly chordal);
  end if
end for
return (G is weakly chordal);

Example 2.1 [5] We will demonstrate an LB-simplicial edge in Figure 2.5, using a graph from Hayward's original paper[9]. The graph is weakly chordal and isomorphic to its complement. We will test if the edge from 2 to 8 is LB-simplicial. The neighborhood of the edge is $N(2, 8) = \{4, 5, 6, 7\}$. The only minimal separator included in the neighborhood is $\{4, 5, 7\}$. Connected components of $\overline{G}(\{4, 5, 7\})$ are $\{4\}$ and $\{5, 7\}$. Vertex 8 sees both vertices in $\{5, 7\}$, and vertex 2 sees 4. The edge from 2 to 8 is then $\{4, 5, 7\}$-saturating and thus LB-simplicial.

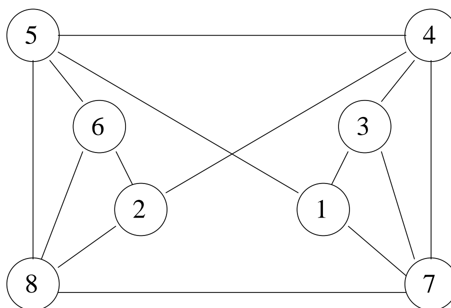

Figure 2.5: A weakly chordal graph isomorphic to its complement.

Chapter 3

Sequential implementation

This chapter will go in detail through our implementation of Algorithm 2.3, Weakly chordal graph recognition, and describe an $O(m^2)$ time sequential implementation.

First we sketch how we can make the algorithm to run in $O(m^2)$ time, then we present the main data structures, before we finally give detailed algorithms for each step of the implementation.

3.1 How to make the algorithm to run in $O(m^2)$

As seen in Algorithm 2.3 in Chapter 2, we can check if each edge e is LB-simplicial to decide whether a graph is weakly chordal. According to Definition 2.16, we then have to check if e is s-saturating for each minimal separator s in the neighborhood of e. For each s we therefore have to compute the connected components of $\overline{G}(s)$ (the co-connected components of $G(s)$), and for each such component check that at least one endpoint of e sees all the vertices in the component. Algorithm 3.1 decides whether an edge in a graph is LB-simplicial.

3.1.1 A direct approach will fail

For Algorithm 2.3 to have an overall time complexity of $O(m^2)$, Algorithm 3.1 has to have a time complexity of $O(m)$. The computation of the neighborhood and its minimal separators can, as we later will see, be done in $O(n)$ and $O(m)$ time. However, an important issue is the number of minimal separators each of which will be processed. In [4] it is shown that this number is at most $n + m$ for a weakly chordal graph, which we will call the *original* separators. But we may, as we will see in Section 3.3.2, encounter the same separator many times. For each edge we actually may encounter at most n separators. However, since each vertex x is adjacent to a distinct edge for each separator x belongs to, the sum of the number of vertices in these

Algorithm 3.1 LB-simpliciality of an edge

```
→ A connected graph G = (V, E) and an edge e ∈ E
← An answer to the question: "Is e LB-simplicial?"

Compute all minimal separators s in N(e);
for all minimal separators s ∈ N(e) do
   Compute the co-connected components of s;
   for all co-connected components ccc of s do
      if no endpoint of e sees all the vertices of ccc then
         return (e is not LB-simplicial);
      end if
   end for
end for
return (e is LB-simplicial);
```

separators will be less than m for each edge. Therefore, to process all the co-connected components for all the minimal separators in the neighborhood of an edge take at most m time. To compute the co-connected components of only one of the minimal separators however, takes $O(m)$ time[1], which gives a time complexity of $O(mn)$ for each edge. Thus Algorithm 3.1 does not obtain a time complexity of $O(m)$.

3.1.2 Solution: Do not re-compute the co-connected components

We now know that while checking a weakly chordal graph we may encounter a total of mn minimal separators if we allow multiple copies, but only $n+m$ otherwise. We also know that we have enough time to compute and process mn separators due to the fact that the sum of the vertices in all the separators is $m \times O(m) = O(m^2)$, but that we do not have the time to compute the co-connected components for all the separators, which also would have resulted in re-computing the co-connected components of a separator with multiple copies. However, we can afford to compute the co-connected components of the $n+m$ original minimal separators, which yields a complexity of $O(mn + m^2) = O(m^2)$, $m \geq n$. If we can gather all the copies of the same separator and also know from which of the vertices of an edge a given vertex in a separator is seen, it is enough to only compute the co-connected components of the $n+m$ minimal separators in a weakly chordal graph once.

Label the vertices of the minimal separators

We solve these challenges by first labeling the vertices when computing the neighborhood of an edge. A vertex x in the neighborhood of an edge e from

[1]We will show how to do this in Subsection 3.3.5

vertex a to b is labeled 1 if it is seen by a, 2 if it is seen by b, and 3 if it is seen by both a and b. In a co-connected component containing several vertices, combinations of 1 and 3, and 2 and 3 is allowed. Combinations of 1 and 2 is not allowed since it corresponds to that one endpoint of an edge does not see all the vertices of the component. If such a combination exists, we can conclude that the graph is not weakly chordal.

It should also be mentioned that computing the co-connected components of a graph is not straight forward if one wants do to it efficiently. We will come back to this issue while discussing the algorithm in more detail.

Compute and sort all the minimal separators

The second part of the solution to the mentioned challenges is to compute a list of all the minimal separators, a total of at most mn, and then sort the list using a linear sorting algorithm. All copies of each separator will now appear consecutively in the sorted list. We can then decide the first copy to be the original and compute the co-connected components only for the original.

The final algorithm

By labeling all the vertices in the neighborhood of an edge, computing a global separator list, computing the co-connected components of only the original separators and in the end checking the labeling of each co-connected component for each separator, we can find if each edge is LB-simplicial in $O(m^2)$ time. This is done in Algorithm 3.2 [5]. The time complexity will be clear as we will go through each step of the algorithm and explain everything in detail in the coming sections.

3.2 Data structures

Although the presented algorithm for recognizing weakly chordal graphs is quite simple to follow and understand, it demands well designed and compact data structures to satisfy the $O(m^2)$ time bound. Both the graph itself, the neighborhood of an edge with labeled vertices, the minimal separators with labeled vertices, the list of minimal separators, the co-connected components, and the set of co-connected components must all have efficient representations for our purpose when checking graphs for weakly chordality. In addition we need extra data structures to compute the minimal separators, to sort the separators, and compute the co-connected components, which will be described in the corresponding subsections in Section 3.3.

Algorithm 3.2 $O(m^2)$ time weakly chordal graph recognition

$\rightarrow$ A connected graph $G = (V, E)$
$\leftarrow$ An answer to the question: "Is G weakly chordal?"

for each edge $ab \in E$ **do**
  Go through the neighbors of a and b simultaneously and form $N(ab)$ as a sorted list of vertices x with labels $l(x) = 1$ if x sees only a, $l(x) = 2$ if x sees only b, and $l(x) = 3$ if x sees both a and b;
  Compute the minimal separators $s \subseteq N(ab)$ and insert them in the global separator list S;
end for
Sort the global separator list;
$original = S[0]$;
Compute the set of connected components CCC of $\overline{G}(S[0])$;
$separators = 1$;
for each minimal separator $s \in S$ **do**
  if $s \neq original$ **then**
    $original = s$;
    $separators{+}{+}$;
    if $separators > n + m$ **then**
      return (G is not weakly chordal);
    else
      Compute the set of connected components CCC of $\overline{G}(s)$;
    end if
  end if
  for each connected component $ccc \in CCC$ **do**
    if $\exists\{x, y\} \in ccc$ with $l(x) = 1$ and $l(y) = 2$ in this copy of s **then**
      return (G is not weakly chordal);
    end if
  end for
end for
return (G is weakly chordal);

3.2.1 The graph

There are two well known standard ways to represent a graph $G = (V, E)$. As an adjacency-matrix or as a collection of adjacency-lists. We will not use the standard definition of any of them.

Using an adjacency-matrix, we have the ability to quickly tell if there is an edge connecting two given vertices. However we will need n^2 space, which is useless with our huge graphs.

Adjacency-list provides a compact way to represent sparse graphs and a space of $2m$ is needed for an undirected graph. The adjacency-list representation consists of an array of n lists, one for each vertex in V. For a given vertex $v \in V$, element number v in the array contains pointers to all the vertices adjacent to v in G. To tell whether there is an edge connecting two vertices or not, we usually have to perform a linear search. In addition, the adjacency list can easily behave dynamically.

However, our graph is final, and we can therefore store it statically. We will use a variant of an adjacency-list in that we place all the lists consecutively in a vector, using another vector to tell where the different lists begin. In this way we only need pointers to the start of the lists and not to every element in each list. We will also store the lists internally sorted, in that we can apply binary search to decide if there is an edge connecting two vertices.

As earlier mentioned, a vector can also grow dynamically, but as far as the representation of the graph is concerned, we use it in a static manner.

Figure 3.1 shows how we represent the graph from Example 2.1. Each of the two vectors contain integer values:

```
vector<int> adj;
vector<int> lstart;
```

The vector *adj* contains all the adjacency-lists, while the vector *lstart* contains one index pointer into *adj* for each vertex in the graph, illustrated by arrows. The adjacency-list for a vertex i therefore starts at *lstart*(i) and ends at *lstart*$(i+1)-1$ in *adj*. Then we can find the neighbors of vertex i in the segment *adj*$[lstart(i) : lstart(i+1)-1]$.

Because we number the vertices from 1 to n, we start the index at one instead of zero in vector *lstart*. For simpliciality we also do this with *adj*. In our undirected graph, *adj* then has a length of $2m + 1$. Vector *lstart* has a length of $n + 2$ because it contains one extra element at the end, pointing to the element beyond the last in *adj*. This last element is necessary to see where the last list in *adj* ends. Overall, the representation of our graph has a space complexity of $\Theta(m + n) = \Theta(m)$, $m \geq n$.

All the graphs in our work will be connected, but due to the completeness of the representation we note that vertices without any neighbors can be represented in *lstart* by -1.

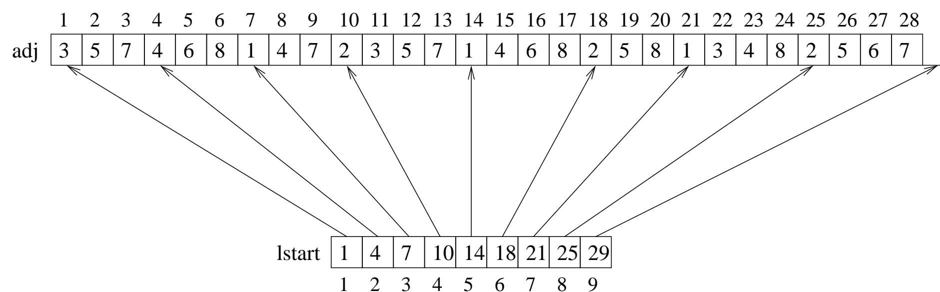

Figure 3.1: Representation of the graph from Figure 2.5

3.2.2 The neighborhood of an edge

To be able to compute the minimal separators in the neighborhood of an edge ab, it is necessary to first compute the neighborhood $N(ab)$. As we will see later, we do this by merging the two sorted adjacency-lists of the vertices a and b. Due to the fact that a and b may have common adjacent vertices, we do not know the final size of the neighborhood of the edge, even if we do know the number of neighbors to a and b. We therefore need a dynamic data structure.

In addition we want the neighborhood of an edge to be sorted, but as long as the adjacency-lists are sorted, we only need to add elements to the end of our data structure. According to the criteria in Subsection 1.3.1, the vector data structure is therefore our clear choice as long as the stored objects are small and simple, which is the case with integers. We will represent vertices in the neighborhood of an edge by a vector of type integer and name it by capital N:

```
vector<int> N;
```

During the computation of the neighborhood the vertices will be labeled. Although there may be a space overhead, we will store the labeling in a vector of size $n + 1$, to secure a fast random access when we later compute the minimal separators. We denote the vector l, for label:

```
vector<int> l;
```

Only the elements corresponding to a vertex in the neighborhood of the present edge are in use.

It is only necessary to store the neighborhood and the labels until the minimal separators are computed. The needed space for the neighborhood is therefore $n - 2$, since an edge can be adjacent to all other vertices, and $\Theta(n)$ for the labels. This yields a space complexity of $\Theta(n)$.

3.2.3 The labeled vertices

For an efficient representation of a labeled vertex in a separator we use the *pair* class template, which is part of the C++ standard library. This class allows us to associate two integer values within a single object, corresponding to a labeled vertex.

An alternative could be to use vectors of size 2, but a vector is a much larger and more complex class than the pair class. This would affect the performance when instantiating objects, and when inserting the labeled vertices into separators.

The individual elements of a pair can be accessed using the member access notations `first` and `second`. For example:

```
pair<int, int> lv( 1, 3);
```

represents the labeled vertex 1, seen by both vertices in an edge, and is therefore labeled 3. We access the vertex by `lv.first` and the labeling of the vertex by `lv.second`.

3.2.4 The separators

The minimal separators in the neighborhood of an edge is of course subsets of the neighborhood of the edge. However, when computing the separators, we do not discover the vertices in an ordered sequence. Because we need them ordered when we will later sort the global separator list, we have to add the vertices to their right position in the separator. At this point a list may seem to be the obvious choice to represent a single separator.

On the other hand, we need random access to the elements of a separator, especially when sorting the list of separators.

In addition will we in Subsection 3.3.2 see that inserting the vertices directly at their ordered place will ruin the time limit. This is solved by only adding vertices to the end of a separator. Therefore vector turns out to be the best choice.

Before the final data structure of a separator is decided, we will consider how to store the global list of separators. Here it is favorable to have random access to the separators, in addition to that we only need to insert them at the end. The clear choice is therefore a vector. However, a separator represented by a vector, is a too large and complex object to directly be inserted into another vector. As mentioned in Subsection 1.3.1, we solve this by storing the objects indirectly by pointers, leading to a very efficient insertion.

To reduce the notational complexity in the description of the data structures, we use the mechanism `typedef` in C++. With `typedef` we can introduce a synonym for an existing data type to improve the readability of our definitions of complex template declarations.

Now, we represent a single separator by a vector of type pair, and give

it the synonym "separator". A single separator is in the algorithms named by the lower-case s:

```
typedef vector<pair<int, int> > separator;
separator* s;
```

The star indicates that s is a pointer to an object, not the object itself.

The global list of separators is then represented by a vector of pointers to single separators. We denote it by the capital letter S:

```
vector<separator*> S;
```

As observed in Subsection 3.1.2, the sum of the number of vertices in the separators is less than m for each edge. Including the labeling, we then need at most $2m^2$ space, obtaining a space complexity of $O(m^2)$.

3.2.5 The co-connected components

For the co-connected components and the set of co-connected components in a minimal separator, we will use data structures quite similar to the minimal separators and the list of minimal separators.

Both the co-connected components and the set of them will have their elements in sorted order inserted to the back, with no needs for the properties of a list. Therefore we will also here use vectors.

Opposite a minimal separator, we do not need the vertices to be labeled. For a co-connected component we then use a vector of type `int`, with the synonym "co_connected_component". We denote a single co-connected component by the lower-cases ccc:

```
typedef vector<int> co_connected_component;
co_connected_component* ccc;
```

As with the minimal separators, we let the set of co-connected components be a vector of pointers to single co-connected components, which we name by the capital letters CCC:

```
vector<co_connected_component*> CCC;
```

For each minimal separator, each vertex can only be in one co-connected component and we do not need to store more than one set at the time. Therefore is the space needed the same as for a minimal separator, namely $O(n)$.

3.2.6 Total space complexity

Now we can aggregate the space complexity for the examined main data structures. The graph representation needed $\Theta(m + n)$ space, the neighborhood of an edge together with its label needed $\Theta(n)$, all the computed minimal separators take $O(m^2)$, while the set of co-connected components will need $O(n)$ space. Totally we then need a space complexity of $O(m^2)$, as long as $m \geq n$.

We have mentioned that we will need some extra data structures to compute the minimal separators, to sort them, and to compute the co-connected components. However, the structures will not ruin the overall space complexity, which we will see in the corresponding subsections of Section 3.3.

3.3 Algorithms

We are now ready to give detailed algorithms of each step of Algorithm 3.2.

3.3.1 Computing $N(ab)$ and the labeling

The first we have to do is to compute the neighborhood $N(ab)$ of each edge $e = ab$, along with the labeling of each vertex in the neighborhood.

As we can see in Algorithm 3.3, we do this by merging the two sorted adjacency-lists of a and b. To know where in the lists we are, we hold one counter for each list. As long as both lists have more vertices, we continue to increment the counters, depending on which list we added a vertex to the neighborhood from.

First we check if the current vertex in a's adjacency-list is less than or equal to the current vertex in b's adjacency-list. If so, we add the vertex in a's list to N, unless it is the other endpoint b of e. If the vertex in a's list is the same as the vertex in b's list, the vertex is seen by both a and b, and is labeled 3. Otherwise it is only seen by a, and is labeled 1.

If current vertex in b's list is less than the current vertex in a's list, we will add the vertex in b's adjacency-list, unless it is the vertex a in e, and label it 2.

After adding a vertex to the list of neighbors, we increment the counters pointing into the lists according to where we added a vertex from. If the vertex is seen by both endpoints of the edge, we increment both counters.

When one of the adjacency-lists has no vertices left, we add the rest of the other list's vertices to the list of neighbors with correct labeling, in addition to check that the opposite vertex of the edge is not added.

Complexity

Both a and b can have n neighbors, so we need $2n$ time go through both adjacency-lists, yielding a time complexity of $O(n)$.

3.3.2 Computing the minimal separators

Second we explain in detail the computation of the minimal separators, which will be stored in the global separator list S.

In general, generating minimal separators can be done by computing the neighborhoods of the connected components resulting from the removal of

Algorithm 3.3 Computing $N(ab)$ and the labeling

$\rightarrow$ A connected graph $G = (V, E)$ and an edge $e = ab \in E$. G is represented with the two vectors `lstart` and `adj`.
$\leftarrow$ A sorted vector N containing the vertices in the neighborhood of e. Each vertex $x \in N$ has label $l(x) = 1$ if x sees only a, $l(x) = 2$ if x sees only b, and $l(x) = 3$ if x sees both a and b.

$counter_a$ = lstart[a];
$counter_b$ = lstart[b];
while both adjacency-lists of a and b have more vertices **do**
  if adj[$counter_a$] $\leq$ adj[$counter_b$] **then**
    $x = adj[counter_a]$;
    if $x \neq b$ **then**
      if x is seen by both a and b **then**
        $l(x) = 3$;
      else
        x is seen by only a
        $l(x) = 1$;
      end if
      Add x to the end of N;
    end if
    if x is seen by both a and b **then**
      $counter_b{+}{+}$;
    end if
    $counter_a{+}{+}$;
  else
    if $x \neq a$ **then**
      $l(x) = 2$;
      Add x to the end of N;
    end if
    $counter_b{+}{+}$;
  end if
end while
if a's adjacency-list has more vertices **then**
  for all remaining vertices x **do**
    if $x \neq b$ **then**
      $l(x) = 1$;
      Add x to the end of N;
    end if
  end for
end if
if b's adjacency-list has more vertices **then**
  for all remaining vertices x **do**
    if $x \neq a$ **then**
      $l(x) = 1$;
      Add x to the end of N;
    end if
  end for
end if

certain vertex sets. In [5], a formal description of all the minimal separators included in the neighborhood of a set of vertices inducing a connected subgraph is given as a theorem:

Theorem 3.1 [5] *Let $K \subset V$ be a set of vertices inducing a connected subgraph of a graph $G = (V, E)$. The set of minimal separators of G included in $N(K)$ is exactly $\{N(C) \mid C \in \mathscr{C}(K \cup N(K))\}$.*

For our special case when computing the minimal separators in the neighborhood of an edge, we let K be the vertices of the given edge and the neighborhood of that edge.

As mentioned in Subsection 3.1.1, we may encounter the same separator many times. This can be seen in Figure 3.2, where we find the minimal separators in the neighborhood e, which are $\{s_1\}$ and $\{s_2\}$. However, separator $\{s_1\}$ is discovered 5 times and $\{s_2\}$ 3 times, due to the fact that all $\{c_1\}...\{c_8\}$ are connected components when we remove $e \cup N(e)$ from the graph. In a special case, we only have one minimal separator, but compute it $n - 3$ times, which gives $O(n)$ number separators in the neighborhood of one edge.

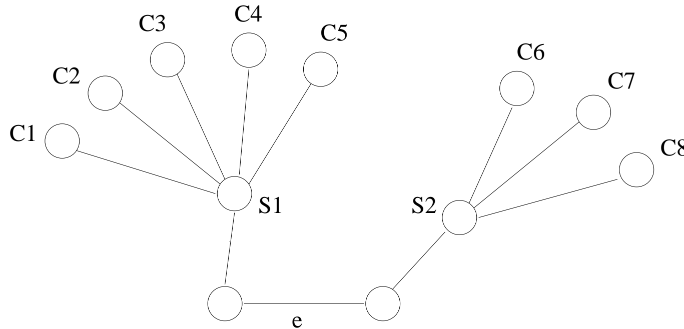

Figure 3.2: Computation of minimal separators

To be able to compute the connected components C of $\mathscr{C}(e \cup N(e))$ as well as their neighborhoods, for a given edge e in a graph, we will use a variant of a depth-first graph search (DFS). Generally a DFS searches from a source vertex recursively deeper into the graph whenever possible, until all the vertices that are reachable from the source vertex have been discovered. If any undiscovered vertices remain, one of them is selected as a new source to continue the search from. The process is repeated until all vertices are discovered.

During the search the vertices are colored to indicate their state. Initially each vertex is white. When a vertex is discovered in the search it is grayed, and when the search finishes, it will finally be blackened. This means the adjacency-list has been examined completely.

In a general DFS, it is usual to record the predecessor of each vertex to form the predecessor subgraph. For a DFS of a non-connected graph, such subgraph forms a depth-first forest composed of several depth-first trees, corresponding to the connected components of the graph. For connected graphs it is a tree. Thus it is a good tool for computing the connected components.

Beside creating a depth-first forest, it is common that each vertex has a time stamp when it is discovered (grayed), and when the search finishes examining its neighbors (blackened).

For our purpose, we do not explicitly need the predecessor subgraph nor the time stamps. Algorithm 3.4 is the general DFS which our variant of DFS is based on. The coloring of the vertices is stored in the vector `color` of length $n + 1$.

Algorithm 3.4 General depth-first search

$\rightarrow$ A graph $G = (V, E)$.

for each vertex $u \in V$ **do**
  $color[u] = white$;
end for
for each vertex $u \in V$ **do**
  if $color[u] = white$ **then**
    Let u be a source vertex
    Call DFS_visit(u);
  end if
end for

DFS_visit(u)
$color[u] = gray$;
for all $v \in N(u)$ **do**
  if $color[v] = white$ **then**
    Call DFS_visit(v);
  end if
end for
$color[u] = black$;

In our variant of DFS, we remove a given edge e and its neighborhood from the graph. However, we let the removed vertices exist in the `color` vector, but give the neighborhood the color *red* and the vertices in the edge *black*. Coloring the vertices of the edge black is actually not necessary, since they are surrounded by red vertices, but we do so to emphasize that they are not to be discovered since they do not belong to the graph at this moment.

Now, for each source vertex we discover, we continue the search in a new depth-first tree, which is a connected component in the graph. The vertices

in the neighborhood of the component in the original graph is according to Theorem 3.1 a minimal separator. These vertices are colored *red*. For each time we then discover a red vertex, we want to add it to a separator. However, we want the separators to be sorted, and to insert the vertices in their correct places may take $O(n)$ time for each inserted vertex. This ruins the overall time complexity of $O(m)$ for each edge.

To solve this challenge we use a vector to store all the minimal separators a given vertex will exist in. We name the vector `v_in_separators`. Even though we will only use the elements x, $x \in N(e)$, we let the size equal $n+1$ to ensure direct access to the elements. When we discover a given red vertex v which we want to be in a separator s, we add a pointer to s in `v_in_separators` at element v. Since vertex v may exist in several separators, and also be discovered to be in a specific separator several times, element v has to contain a structure for which the pointers to the separators will be added to. We let this structure be a vector, and let then `v_in_separators` be a vector of pointers, due to earlier mentioned reasons about the vector class' complexity and size. That is, `v_in_separators` is a vector of pointers to vectors of pointers to separators:

```
vector<vector<separator* >* > v_in_separators;
```

For each time we are about to continue the search from a new source vertex, we create a new separator and add it to the global separator list. When we then discover a red vertex, we add a pointer to this separator to the vertex' list in `v_in_separators`.

Now, after we have completed the DFS, adding which separators each vertex v belongs to, we can traverse each element in vector `v_in_separators`, and add the element v to the separators we find. Actually, we only need to traverse the neighborhood which is sorted, avoiding the overhead of traversing empty elements of the `v_in_separators` vector. The elements of the separators will now be added in sorted order, and we are able to exclude repeated elements. Our version of DFS when computing the minimal separators are shown in Algorithm 3.5.

Figure 3.3 shows both a graph and the corresponding data structure when computing minimal separators in the neighborhood of an edge. In (a) we see the graph, where the letters W, R, and B indicate the initial colors white, red, and black when starting the computation of the minimal separators in the neighborhood of $e = (5, 6)$. The vertices of edge e is colored black, the vertices in its neighborhood red, and the rest of the vertices, belonging to the connected components $\mathscr{C}(e \cup N(e))$, white.

Figure 3.3(b) shows the data structure when the DFS is finished, while (c) shows the resulting global separator list S, containing the minimal separators in the neighborhood of e.

Algorithm 3.5 Computing the minimal separators

$\rightarrow$ A connected graph $G = (V, E)$, an edge $e = ab \in E$, the neighborhood $N(e)$ to the edge, and the labeling l to the neighborhood.
$\leftarrow$ A global list S of minimal separators.

```
for each vertex u ∈ V do
  color[u] = white;
end for
color[a] = color[b] = black;
for each vertex u ∈ N(e) do
  color[u] = red;
end for
for each vertex u ∈ V do
  if color[u] = white then
    Create new s;
    Add s to the global separator list S;
    Call DFS_visit_MinSep(u);
  end if
end for
for each vertex u ∈ N(e) do
  for each separator s u belongs to do
    Get the labeling from l;
    Add u to s;
  end for
end for

DFS_visit_MinSep(u)
color[u] = gray;
for all v ∈ N(u) do
  if color[v] = white then
    Call DFS_visit_MinSep(u);
  else if color[v] = red then
    Add s to the list of separators v belongs to;
  end if
end for
color[u] = black;
```

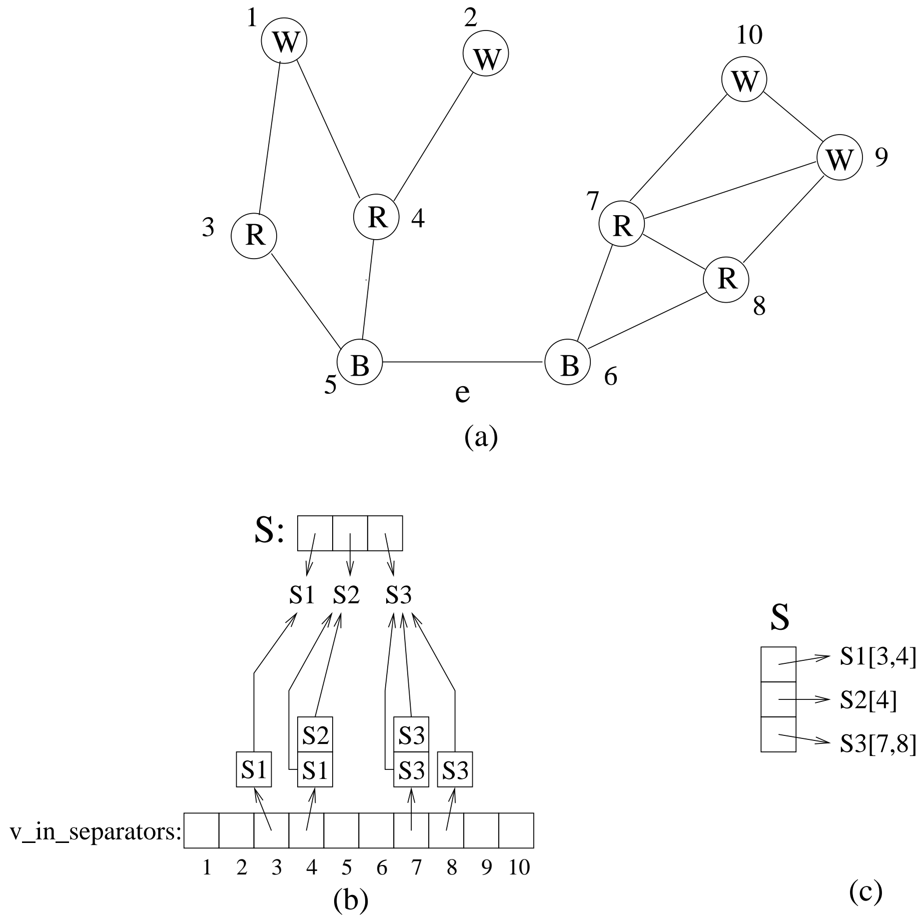

Figure 3.3: Data structures when computing the minimal separators

Complexity

When running the DFS we traverse only those edges incident to initially white vertices, after initializing the `color` vector. This takes $O(n+m)$ time. The number of vertices in the separators for one edge is at most m, as mentioned in Subsection 3.1.1. Traversing all the elements in the separators for one edge therefore takes $O(n+m)$ time, yielding a total time complexity of $O(n+m) = O(m)$, $m \geq n$, which is within our time limit.

The extra space needed for the computation is $\Theta(n+m) = \Theta(m)$, $m \geq n$, for the `color` and `v_in_separators` structures. This is within the limit of the previously found space complexity.

To reduce the potential overhead of space regarding the `v_in_separators` structure, it could be an issue to use the associative container map. However, both insertion and lookup in a map is performed in $O(\log n)$ time, which

ruins the time complexity.

3.3.3 Sorting the separator list

After the minimal separators have been computed, we are ready to sort the global separator list. As mentioned several times, this list may contain mn separators, but the total number of elements is $O(m^2)$, which also is our overall time limit.

To obtain this time complexity we apply radix sort, a linear time sorting algorithm. We regard each separator s as a number with $|s|$ digits d, where $1 \leq |s| \leq k \leq n$, and $1 \leq d \leq n$. Generally, given all elements with k digits, radix sort applies bucket sort k times. The sorting is done from right to left, starting from the least significant digit. That is, first we sort the separators according to their last vertex, then the second last vertex, and so on until the first vertex.

Initially, as we can see in Algorithm 3.6, we keep the separators in a global queue GQ. Starting at the front, we insert each separator in one of n buckets, determined by the least significant digit (last vertex). Thereafter we scan the buckets in order and add all the elements to GQ. This is repeated for all the digits from right to left, a total of k times, which is the size of the largest separator.

For the algorithm to work correctly it is essential that elements that are put in the same bucket remain in the same order. This is achieved by using a queue for each bucket.

Since all the separators do not have size k, it is necessary to remove those of less size before k steps have been done. When we discover a separator from GQ, all of whose elements have been traversed, we can add it to the end of our global separator list S, and it will be added to its correct place. When finished with k steps, we add the remaining separators of GQ, all of size k, to the end of S, and S is now in sorted order.

Mark that the separators initially are in our separator list S, we first have to put them into GQ. At the same time we also find k, the size of the largest separator. Strictly, we could have done this at an earlier point, but as long as it does not obtain extra time nor space complexity, we do it here for clarity.

Complexity

The only thing actually done in the algorithm is to move pointers to separators. First, all separators are added to GQ, which takes $O(mn)$ time. Second, for each element in the separators, the pointer of its separator is moved from GQ to another queue, before moving it back. The total time complexity is therefore $O(mn + m^2) = O(m^2)$, since we assume connected graphs and thus $m \geq n$.

Algorithm 3.6 Sorting the global separator list

```
→ The global separator list S
← S in sorted order

for all s ∈ S do
  Insert s into the global queue GQ;
  if |s| > k then
    k = |s|;
  end if
end for
Empty S;
for i = 1 to n do
  Create queue Q[i];
end for

for i = 0 to k do
  while GQ is not empty do
    Pop s from GQ;
    j = s[|s| − i]; (the i'th last index of s)
    if j < 0 then
      Add s to the end of the global list S;
    else
      d = s[j];
      Insert s into Q[d];
    end if
  end while
  for i = 1 to n do
    Insert Q[i] into GQ;
  end for
end for
while GQ is not empty do
  Pop s from GQ;
  Add s to the end of the global list S;
end while
```

3.3.4 Identify original separators

When we have sorted the global separator list we are ready to compute the co-connected components and check the labeling in them for each separator in the list. As earlier explained, we have only time to compute the components for the $O(n+m)$ original separators. We therefore have to identify them when processing the sorted list.

This is done by comparing the present original separator with the next separator in the list. The algorithm is straight forward, the size of the original and the size of the next separator are checked for equality. If they are, each of the vertices must be compared, as seen in Algorithm 3.7. Of course, the label of each of the vertices may be different.

Algorithm 3.7 Comparing two separators

$\rightarrow$ Two separators s_1 and s_2
$\leftarrow$ An answer to the question: "Do s_1 and s_2 contain exactly the same vertices?"

if $|s_1| \neq |s_2|$ **then**
  return (s_1 and s_2 are unequal);
else
  for $i = 0$ **to** $|s_1|$ **do**
    if $s_1[i] \neq s_2[i]$ **then**
      return (s_1 and s_2 are unequal);
    end if
  end for
  return (s_1 and s_2 contain the same vertices);
end if

Complexity

Though the comparison between two separators may take $O(n)$ time, which is the maximum size of a separator, the total time complexity for identifying the originals is the total number of vertices in the global separator list, namely $O(m^2)$.

3.3.5 Computing the co-connected components

Now we take a closer look at the computation of the co-connected components in an original minimal separator. Since a separator is an induced subgraph of a graph with, as seen in Subsection 3.3.2, $O(n)$ vertices, it is the same problem as finding the connected components of the complement of a graph G. One straight forward way to do this, is to first compute the complement of the graph, $\overline{G}$, and then perform a depth-first search, where the connected components will be the depth-first trees in the depth-first forest.

However, this will take $O(\overline{m})$ time, where $\overline{m}$ is the number of edges in $\overline{G}$. As long as we do not know whether $\overline{m} \leq m$, this is not a satisfactory way to do it. We therefore have to do it in a better way, using only the edges of G and not those of $\overline{G}$, to be sure to achieve $O(m)$ time, which is the limit since we may have a total of $O(m+n)$ different original minimal separators.

In fact, the time complexity of computing the co-connected components in a minimal separator was an issue the authors of [5] had to review to be sure that the overall time limit were satisfied.

High level description

We will now describe a method that considers only edges of G and thus runs in $O(m)$ time. The method is based on the following observation:

Observation 3.2 *If we mark all neighbors of a vertex v in a graph G, at least all unmarked vertices must be in the same connected component as v in the complement graph $\overline{G}$.*

The observation is correct due to the fact that there clearly have to be edges in $\overline{G}$ to all the unmarked vertices from v.

We start the method by assigning each vertex a label initialized to 0. A vertex v_1 are chosen arbitrarily to be in the first co-connected component. All its neighbors are then marked by incrementing their labels to 1. We know that all vertices still having a label 0, must be in the same co-connected component as v_1. We therefore choose a new vertex v_2 with label 0, add it to the same co-connected component as v_1, and increment the label to all vertices in the neighborhood of v_2.

Now, if v_1 and v_2 both see a given vertex v in the original graph, that vertex has label 2. More important, every vertex labeled less than 2 must be in the same component as v_1 and v_2, since this means that either v_1 or v_2 does not have an edge to v in G, and thus does have an edge to v in $\overline{G}$. We therefore continue to choose a vertex having a label < 2. From the following observation we find when to stop adding vertices to the component:

Observation 3.3 *By repeatedly adding vertices to a co-connected component of a graph, and for each added vertex mark all vertices in its neighborhood, no further vertices can be added when all remaining vertices have been marked by all vertices in the co-connected component.*

The correctness of the observation is clear since all remaining vertices are marked by all vertices in the co-connected component, the remaining vertices must all have an edge in G to every vertex in the co-connected component, and no edge in $\overline{G}$ can exist to any of these vertices.

If we denote the number of vertices in a co-connected component the *size* of the component, we know that all remaining vertices are marked

by all vertices of the component when all their labels equals the size of the component. We therefore continue adding vertices with labels less than the size of the component as long as possible. When no further vertices can be added, we add the computed co-connected component to the set of co-connected components. Then we can start all over again, setting the labels of the remaining vertices to 0 and creating a new empty co-connected component.

We repeatedly create new co-connected components as long as there are remaining vertices.

The described method can be seen in Algorithm 3.8.

Algorithm 3.8 Computing the co-connected components of a separator

---

$\rightarrow$ A connected graph $G = (V, E)$ and a separator s
$\leftarrow$ The set CCC of co-connected components ccc in s

$X = s$;
while $G(X)$ not empty **do**
  Create an empty co-connected component ccc;
  for all remaining $x \in G(X)$ **do**
    $vlabel(x) = 0$;
  end for
  repeat
    Choose a vertex $v \in G(X)$, $vlabel(v) < |ccc|$ or $vlabel(v) = 0$;
    for all neighbors w of v in $G(X)$ **do**
      $vlabel(w) + +$;
    end for
    Add v to ccc and remove it from X;
  until every $x \in G(X)$ has $vlabel(x) = |ccc|$
  Add ccc to CCC;
end while

Detailed description

Now we take a closer look into how we can implement Algorithm 3.8 in $O(m)$ time, which is shown in Algorithm 3.9.

As we can see from Algorithm 3.8, we need in constant time to choose a vertex with label less than the size of the present co-connected component, and also be able to tell when all remaining vertices have a label equal to the size of the co-connected component.

If we for each label have a list containing the vertices having that label, we manage to choose a vertex with the smallest label in constant time. We store the lists in a vector `vlist`, and by starting with the smallest label $minLabel = 0$, we can add vertices to a co-connected component ccc and empty each list, $vlist[minLabel]$, by incrementing $minLabel$ as long as

$minLabel < |ccc|$. When $minLabel = |ccc|$ all remaining vertices clearly have a label that equals $|ccc|$. No further vertices can then be added to the component, and we therefore create a new component.

However, when we have added a vertex u to a component, we want to increment the label of all its remaining neighbors. When analyzing the complexity, it will become clear that we do not have time to search the lists in *vlist* to find the neighbors, so we store in vector `vlabel` of length $n+1$ what label each vertex in the graph has. If the vertex does not belong to a separator or if it already has been added to a component, its corresponding element is set to -1. Then we know which list the element we want to move to the next list belongs to, but we still do not have the time to search the list. We therefore keep a pointer to the elements in the lists, or actually an iterator, which is a generalization of a pointer as explained in Section 1.3.1. The iterators are also stored in a vector called `vpointer` of length $n+1$. Through the iterator into the list given by *vlabel*, we can remove the vertex the iterator is pointing to and add it to the next list in constant time.

Figure 3.4 shows the computation of the co-connected components of a graph G, using Algorithm 3.9. The input separator is then the vertex set V of the graph. Initially, all vertices are put into $vlist[0]$. Then, in three steps, 3 vertices are added to the first co-connected component. All the tree vertices had edges to the two last vertices, so they cannot belong to the same component. When $vlist[minLabel = 0]$ now is empty, *minLabel* is incremented, but we find no vertices in the 1st or 2nd list. At list 3, *minLabel* equals the size of the first co-connected component, and all the vertices of that list are moved to list 0, and are candidates for the next component, which as seen is computed in two steps.

Complexity

Putting all the vertices of a separator into the data structure takes $O(n)$ time.

When computing the components, each edge $ab \in E$ is traversed not more than twice. At most once for $b \in N(a)$, and at most once for $a \in N(b)$. Since separator $s \subseteq V$, not all the edges in the graph is traversed.

In addition, after an edge has been traversed, and a vertex' label is incremented and the vertex is moved to the next list, it may be moved back to $vlist[0]$, but not more than once for each time it is moved up. It will therefore only influence on the constant in the time complexity, which have a total of $O(m+n) = O(m)$, $m \geq n$.

Both *vlabel* and *vpointer* take n space. Also *vlist* takes at least n space since it has place for n lists. However, the content of the lists take also no more than n space, since the number of vertices in a separator is $O(n)$. The total extra space is therefore $O(n)$, which fits nicely in the overall space complexity of $O(m^2)$.

Algorithm 3.9 Computing the co-connected components of a separator in $O(m)$ time

$\rightarrow$ A connected graph $G = (V, E)$ and a separator s
$\leftarrow$ The set CCC of co-connected components ccc in s

for each vertex $v \in s$ **do**
  Add v to $vlist[0]$;
  $vlabel[v] = 0$;
  Let $vpointer[v]$ point to v in $vlist[0]$;
end for
$minLabel = 0$;
while $vlist[0]$ not empty **do**
  Create an empty co-connected component ccc;
  repeat
    while $vlist[minLabel]$ not empty **do**
      Remove a vertex u from the front of $vlist[minLabel]$;
      Add u to ccc;
      $vlabel[u] = -1$;
      for all $v \in N(u)$ **do**
        if $vlabel[v] \neq -1$ **then**
          Move v from $vlist[vlabel[v]]$ to the end of $vlist[vlabel[v] + 1]$;
          $vlabel[v] + +$
          Get the new $vpointer[v]$ from $vlist[vlabel[v]]$;
        end if
      end for
    end while
    $minLabel + +$;
  until $minLabel = |ccc|$
  for each vertex $vlist[minLabel]$ **do**
    Through $vpointer[v]$ move v to $vlist[0]$
    $vlabel[v] = 0$;
    Get the new $vpointer[v]$ from $vlist[0]$;
  end for
  Add ccc to CCC;
  $minLabel = 0$;
end while

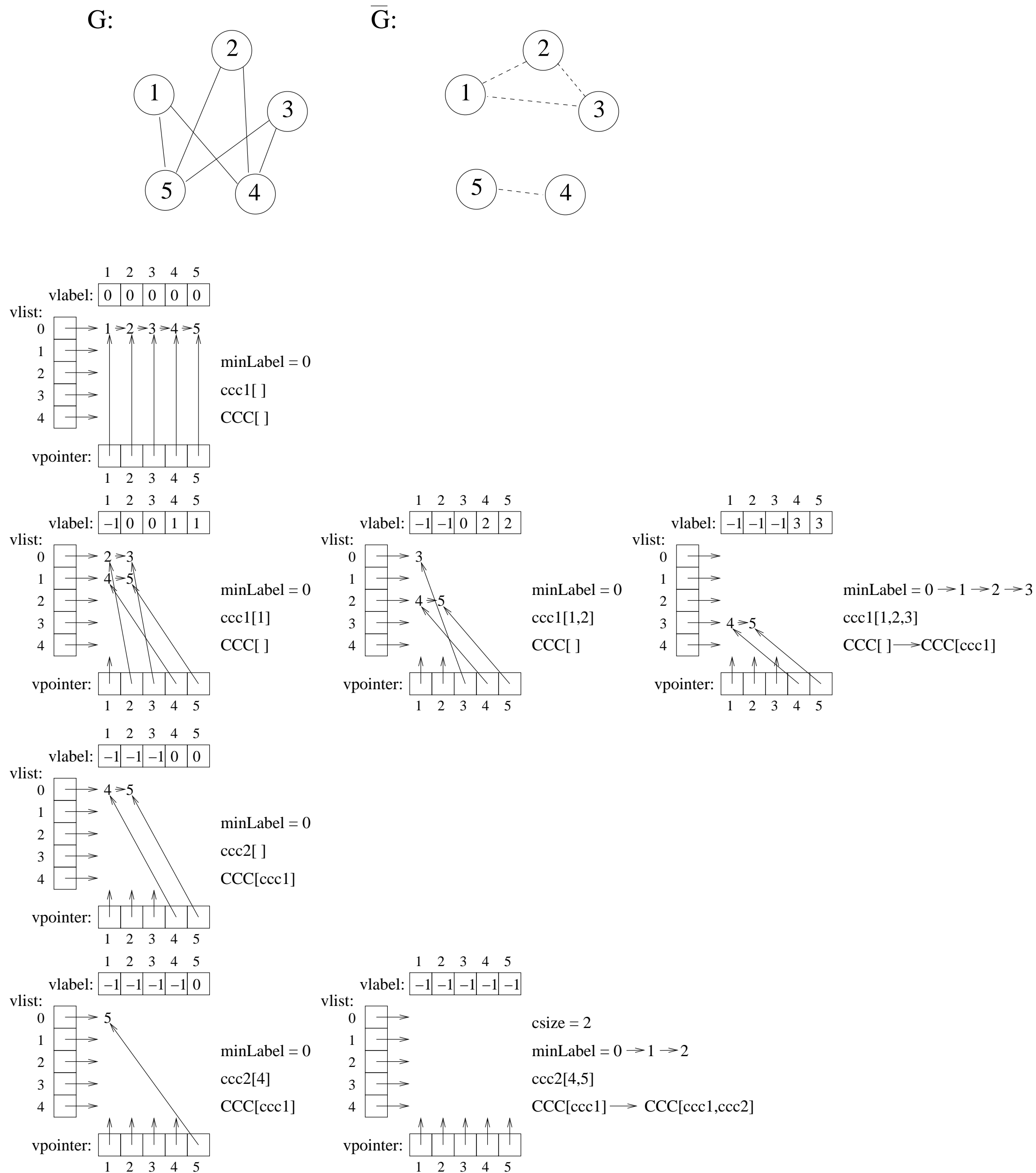

Figure 3.4: Computation of co-connected components of a graph G

3.3.6 Checking the labeling in the co-connected components of a separator

When we have computed a set of co-connected components for an original separator, we have to process each component and check that one endpoint of the edge the separator is in the neighborhood of sees all vertices of the component. As mentioned in Subsection 3.1.2, this corresponds to that the combination of two vertices in a component labeled 1 and 2 is not allowed.

The procedure has to be repeated for all the copies of the original separator.

When checking the labels we must in total process all the vertices of the global separator list of size $O(m^2)$. Thus, the labels of the vertices in each separator have to be accessed in constant time. In the actual C++ implementation we therefore have to first go through the present separator and insert the label of vertex v in element v in vector `l`, which we also used when computing the neighborhood of an edge. The result is that we have to traverse the separator list twice, only influencing the constant in the time complexity. Alternatively we could have stored the labels for each separator in a vector of length n, resulting in a space complexity of $O(mn^2)$, which would ruin the overall space complexity of $O(m^2)$. An associative container map would neither be suitable, since lookup takes $O(\log n)$ time with a separator of length $O(n)$.

Algorithm 3.10 shows how the labeling in a set of co-connected components are checked for one separator. The access of the label of a vertex v of a separator s is done through the notation $s[v].label$ in constant time. We keep a masterlabel *mlabel* which is set to the first 1 or 2 which is discovered. A label 3 is accepted at any time, since it corresponds to the case where both endpoints of an edge sees the vertex. However, if another label not equal to the master label is discovered, we can conclude that at least one edge is not LB-simplicial.

Complexity

As mentioned we have to process all elements in the global separator list when checking the labeling in the co-connected components of each separator, yielding a time complexity of $O(m^2)$.

The restoring of direct access to the labels of a separator in vector `l`, yields no extra space complexity.

3.3.7 Total time complexity

Now we can review Algorithm 3.2 and take a closer look at the time complexity.

The computation of the neighborhood and the minimal separators of an edge takes $O(n)$ and $O(m)$ time. The first **for each** loop which processes

Algorithm 3.10 Checking the labeling in the co-connected components of a separator

$\rightarrow$ A separator s and a set of co-connected components CCC
$\leftarrow$ An answer to the question: "Does there in any component $ccc \in CCC$ exist $\{x, y\}$ where $s[x].label = 1$ and $s[y].label = 2$?"

```
for each ccc ∈ CCC do
  mlabel = 0
  for each v ∈ ccc do
    if s[v].label ≠ 3 then
      if mlabel ≠ 0 then
        if mlabel ≠ s[v].label then
          return (Such {x, y} exists);
        end if
      else
        mlabel = s[v].label
      end if
    end if
  end for
end for
return (Such {x, y} does not exist);
```

each edge then takes a time of $O(m^2 + mn) = O(m^2)$, $m \geq n$.

In the second **for each** loop, processing all the minimal separators, both the identification of the originals, the computation of the $O(n + m)$ co-connected components, and checking the labels take a total of $O(m^2)$ time.

Together with the sorting of the separator list in $O(m^2)$ time, we obtain an aggregated time complexity of $O(m^2)$, as expected.

Chapter 4

Parallel implementation

In this chapter we take a parallel approach to Algorithm 2.3, Weakly chordal graph recognition, and describe a parallel implementation using the Message passing programming (MPP) model mentioned in Subsection 1.4.3.

The MPP model is chosen because it gives the most interesting parallel approach. When implemented by using the MPI, it has succeeded, as earlier mentioned, to achieve high performance and scalability. A drawback may be that every processor needs access to the entire graph, yielding one copy of it for each processor. However, the memory taken by the graph, $O(m)$, is a small part of the total space complexity of $O(m^2)$, as long as the number of processors are much less than the number of edges m, which will be realistic.

Algorithm 2.3 is based on Theorem 2.17, stating that a graph is weakly chordal if and only if every edge is LB-simplicial. It is obvious that if we could have as many processors as edges, that is m processors, we could let each processor check one edge each for LB-simpliciality. Hence, the iterations of the `for`-loop in Algorithm 2.3 is independent of each other and can be processed in parallel, yielding an algorithm parallel in nature.

The parallel time complexity is then the time it takes to check one edge for LB-simpliciality, which is $O(mn)$ as seen in Subsection 3.1.1. We note that the approach gives a cost of $O(m^2n)$. This is not cost-optimal since it is not proportional to $O(m^2)$, which is the time complexity of the best known sequential algorithm. This is due to the fact that we parallelized the sequential direct approach which we know from Section 3.1 will fail to obtain the $O(m^2)$ time complexity.

Another note is that using m processors is unrealistic when working with huge graphs, and from now on we assume that we have p processors, where p is much less than m. An important issue to obtain concurrency yielding high speedup is therefore the load balance of the work between the processors, and the subsequent termination detection to find out when we have a global answer to the question whether a graph is weakly chordal or not.

4.1 Load balancing

For load balancing, the aim is to distribute the computations evenly across the processors in order to obtain the highest possible concurrency and execution speed.

Figure 4.1 illustrates the concept using four processors. In Figure 4.1(a), processor P_1 operates for a longer period and processor P_3 completes its work early. This yields a longer execution time than optimal. For the highest concurrency it would be ideal that part of P_1's work should be given to P_4 to equalize the workload. This is shown in Figure 4.1(b).

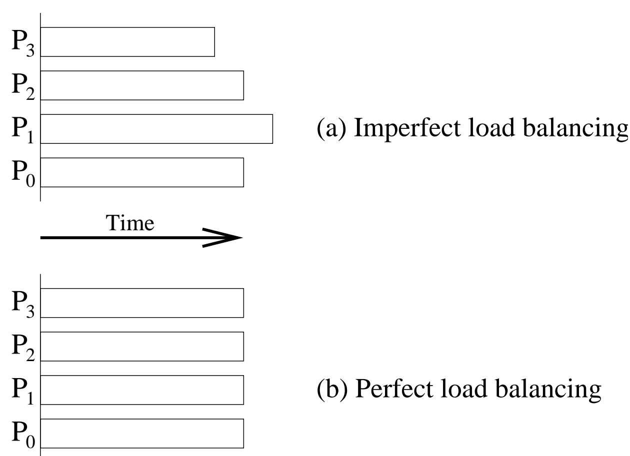

Figure 4.1: Load balancing

The issue of load balancing takes two approaches, static and dynamic.

Static load balancing is usually referred to as the *mapping problem* or *scheduling problem*[14]. When using p processors, we divide the computations in what we hope is p equally large time consuming subtasks and assign each processor one subtask each. If communication is necessary, it should be taken into account that the communicating processors should be close to each other. In newer machines this is not straight forward.

The disadvantage of using static load balancing is the estimation of the size of the subtasks. It can be hard to equally part the computations because there may be an unknown number of steps to reach the solution. In addition, extra computations may be necessary to compute the estimation.

Dynamic load balancing is done in several ways, but all these ways manage to cope with the challenge of dividing the computations. The division of load is dependent upon the execution of the parts as they are

being executed. If one processor finishes working, it helps one other processor that still has more computations to work on.

The disadvantage here is the need for communication. There has to be communication between processors to discover the need to share a workload. If the need is present, transfer of data to be computed is likely to be necessary.

To sum up, static and dynamic load balancing may lead to different types of overhead. In Subsection 1.4.2, three factors was mentioned; load imbalance, interprocessor communication, and extra computation.

Static load balancing may cause load imbalance, missing the goal of evenly distributing the work load. Extra computations may also be needed.

Dynamic load balancing does need interprocessor communication, and the time spent on the transfer of data is often a significant source of overhead.

The choice between static and dynamic load balancing is a consideration of which will lead to least overhead. The next two subsections take a closer look at the approaches of our algorithm.

4.1.1 Dynamic load balancing of the best known sequential algorithm

First we take Algorithm 3.2, $O(m^2)$ time weakly chordal graph recognition, the best known sequential algorithm, as a starting point. Note that also other algorithms have acchieved this bound[12].

In the first `for each`-loop, a simple dynamic load balancing is possible when we for each edge find the neighborhood and all minimal separators. Initially we can distribute the edges equally among p processors, and when one processor finishes, it can ask for more work from other processors. This of course requires the processors to check frequently if there are other processors wanting to help them with their work.

After we have found the neighborhoods and separators, each processor has its own collection of computed separators. To fulfill the sequential algorithm, the co-connected components are only to be computed once for each different separator. All equal separators therefore have to be placed at the same processor. The transfer of all equal separators to one processor may yield a heavy transfer of data.

Now the sorting of the separators can take place, and all copies of a separator will be processed by one processor.

Further, the computation of the co-connected components may take place, together with the checking of the labeling of each separator. Only the co-connected components of the originals will now be computed. If one processor finishes early, it can help another processor having more work to be done, of course with extra interprocessor communication.

4.1.2 Simple static load balancing

Next we combine the direct approach described in the beginning of this chapter and the best sequential algorithm. By distributing the edges evenly among the processors and run the best sequential algorithm on the resulting subgraph on each processor, we manage to do a static parallelization that avoids interprocessor communication, but with $O(p)$ recomputations of each of the original co-connected components.

This parallelization is simple but powerful under the assumption that the time to do extra computation of some co-connected components are much less than the transfer of data when placing all equal separators at one processor, and that the load imbalance resulting from static load balancing is small and less than the time to do interprocessor communication and data transfer to dynamically correct the imbalance. Algorithm 4.1 shows this parallelization. If all processors answer "All my edges are LB-simplicial", the graph G is weakly chordal, otherwise it is not. We assume that the call to Algorithm 3.1 can be executed with a subgraph, despite the fact that the Algorithm may need access to the entire graph.

Algorithm 4.1 Parallel weakly chordal graph recognition with static load balancing

→ A connected graph $G = (V, E)$, p processors $P_i, i = 0, ..., p-1$
← A local answer from all processors to the question: "Are all of "your" edges LB-simplicial?"

for each processor P_i **do**
  Divide E into equal parts and find my_E;
  Call Algorithm 3.1($G' = (V, my_E)$);
  if answer is "G' is weakly chordal" **then**
    return (All my edges are LB-simplicial);
  else
    return (At least one of my edges are not LB-simplicial);
  end if
end for

We will use this static load balancing in our final parallel implementation, but first we consider the matter of termination detection. After all, we are interested in one global answer, and as soon as one processor has discovered one edge which is not LB-simplicial, all processors should end their computations.

4.2 Termination detection

Termination detection is mainly connected to dynamic load balancing to find out when all computations to be done are finished. Special algorithms exist

to detect when to terminate, because a processor may believe it has finished its work, but later it can be delivered more work from another processor.

However, search algorithms do need termination detection both in connection with static and dynamic load balancing, due to the fact that it is not necessary to search through unexplored parts of the search space if the search has succeeded.

Our algorithm takes a similar approach. We check edges for LB-simpliciality, but as soon as we have found one edge that is not LB-simplicial, we can end our computations and conclude that the graph is not weakly chordal. So when p processors each get a number of edges and the question of whether they are LB-simplicial or not, termination and a global answer can be reached in one out of two ways.

1. All processors finish locally, and find that all their edges are LB-simplicial. This yields the global answer that the graph is weakly chordal.

2. One processor finishes locally after finding that one of its edges are not LB-simplicial. All other processors are then told to terminate, and the graph is not weakly chordal, which is the global answer.

Algorithm 4.2 roughly implements this termination detection, using the static load balancing from Algorithm 4.1. We let processor P_0 be a master, and all other processors report to P_0 when returning an answer from Algorithm 3.1. We postpone the details of the communication to the next section, and for the time being we assume that P_0 finishes Algorithm 3.1 as soon as another processor needs to tell that it has an edge which is not LB-simplicial. Likewise we assume that P_0 can just tell another processor to stop working.

4.2.1 The possibility for superlinear speedup

In Subsection 1.4.1 we mentioned that superlinear speedup is possible in parallel search algorithms. Since our algorithm is of similar nature, the same is possible. Figure 4.2 outlines this possibility.

In Figure 4.2(a) 16 minimal separators are found and are about to be processed when checking the separators labeling. One of the separators, S_{13}, has a labeling corresponding to the case where the edge it is in the neighborhood of is not LB-simplicial. When using one sequential processor, that processor will start processing at separator S_1 and finish working when S_{13} is processed.

Linear speedup is achieved if we manage to divide all the work done by the sequential processor equally among several processors. In Figure 4.2(b)

Algorithm 4.2 Parallel weakly chordal graph recognition and termination detection

→ A connected graph $G = (V, E)$, p processors P_i, $i = 0, \ldots, p-1$
← A global answer to the question: "Is G weakly chordal?"

for each processor P_i **do**
  Divide E into equal parts and find my_E;
  Call Algorithm 3.1($G' = (V, my_E)$) and return answer to P_0;
  if I am P_0 **then**
    for all processors P_i **do**
      Receive local answer;
      if local answer is "G' is not weakly chordal" **then**
        Tell all still working processors to stop;
        return (G is not weakly chordal);
      end if
    end for
    return (G is weakly chordal);
  end if
end for

we use 4 processors. Processor P_3 starts its work at S_{13}, immediately discovering that this separator has an irregular labeling. A global answer is then reached and all processors can stop working.

Assuming that the processors use equal time to process a separator, the separators $S_2 - S_4$, $S_6 - S_8$, and $S_{10} - S_{12}$ will not have to be processed. This leads to less work being performed in the parallel than in the sequential approach, opening up for the possibility for superlinear speedup.

4.3 Final parallel implementation

Now we are ready to combine the load balancing and termination detection from the previous sections in a final parallel algorithm for checking a graph for weak chordality. This is done in Algorithm 4.3, which consists of the computation of the static load balancing, a call to Algorithm 4.4, "LB-simplicial(my_E)", which checks the edges in my_E for LB-simpliciality, and one part where processor P_0 controls the termination detection. Algorithm 4.4 is quite similar to the best known sequential algorithm, Algorithm 3.2 "$O(m^2)$ time weakly chordal graph recognition", except that it in addition communicates with processor P_0 to handle termination detection. All its details of data structures and step by step implementation are therefore explained in Chapter 3, Sequential implementation. Only the details of the load balancing in Algorithm 4.3 and the termination detection concerning both Algorithm 4.3 and 4.4 are left to be explained.

Algorithm 4.3 Final parallel weakly chordal graph recognition

```
→ A connected graph G = (V, E), p processors P_i, i = 0, ..., p − 1
← A global answer to the question: "Is G weakly chordal?"

for each processor P_i do
  Divide E into equal parts and find my_E;
  Call LB-simplicial(my_E); (Algorithm 4.4)
  if I am P_0 then
    if NOT "All the edges in my_E are LB-simplicial" then
      for i = 1 to p − 1 do
        Isend((Terminate), P_i, request);
      end for
    else
      for i = 1 to p − 1 do
        Recv(msg = ANY_MSG, ANY_SOURCE);
        if msg is "At least one edge in my_E are not LB-simplicial"
        then
          for i = 1 to p − 1 do
            Isend("Terminate", P_i, request);
          end for
          return (G is not weakly chordal);
        end if
      end for
    end if
    return (G is weakly chordal);
  end if
end for
```

Algorithm 4.4 LB-simplicial(my_E)

$\rightarrow$ A connected graph $G = (V, E)$, a set of edges $my_E \in E$, processor P_0 to send an answer
$\leftarrow$ Nothing if the processor is told to terminate during the process. Otherwise an answer sent to processor P_0 to the question: "Are the edges in my_E LB-simplicial?"

Call Irecv(msg != "All the edges in my_E are LB-simplicial",
    ANY_SOURCE, requestA);
for each edge $ab \in my_E$ **do**
  Call Test(requestA, completed);
  if completed **then**
    Terminate;
  end if
  Form $N(ab)$ with labels;
  Compute the global separator list S of minimal separators;
end for
Sort the global separator list;
$original = S[0]$;
Compute the set of connected components CCC of $\overline{G}(S[0])$;
$separators = 1$;
for each minimal separator $s \in S$ **do**
  Call Test(requestA, completed);
  if completed **then**
    Terminate;
  end if
  if $s \neq original$ **then**
    $original = s$;
    $separators++$;
    if $separators > n + m$ **then**
      Isend((At least one edge in my_E are not LB-simplicial), P_0,
        request);
      Terminate;
    else
      Compute the set of connected components CCC of $\overline{G}(s)$;
    end if
  end if
  for each connected component $ccc \in CCC$ **do**
    if $\exists\{x, y\} \in ccc$ with $l(x) = 1$ and $l(y) = 2$ in this copy of s **then**
      Isend((At least one edge in my_E are not LB-simplicial), P_0,
        request);
      Terminate;
    end if
  end for
end for
Isend((All the edges in my_E are LB-simplicial), P_0, request);

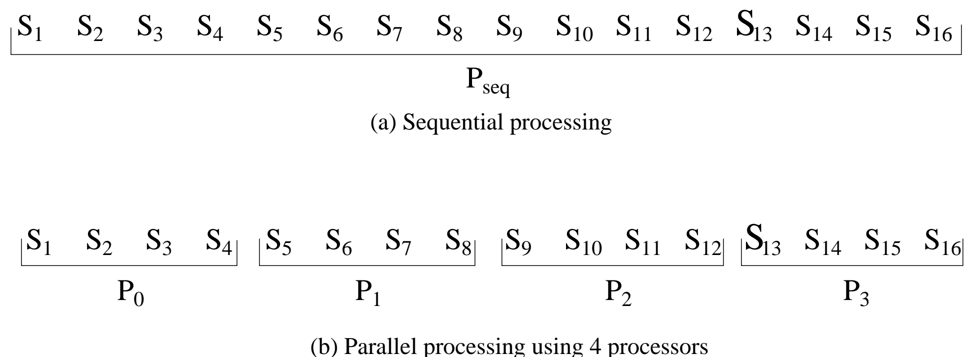

Figure 4.2: Possibility for super linear speedup: Separator 13 has irregular labeling, excluding weak chordality. The sequential processor in (a) discovers this almost in the end. In (b), processor P_3 discovers this immediately, and the processing can end.

4.3.1 The load balancing in detail

The time it takes to verify LB-simpliciality for an edge is mainly determined by the size of the neighborhood of the edge and the number of and the sizes of the minimal separators in the neighborhood. The possibility for great differences in time is therefore present. For example a neighborhood of 1 vertex versus $O(n)$ vertices. However, the processing of $\Theta(n)$ separators of $O(1)$ size, will take approximately the same time as processing $O(1)$ separators of $\Theta(n)$ size.

As a result, the time for verifying an edge for LB-simpliciality is approximately even for all the edges if the size of the neighborhood is even for each edge.

Making this assumption, we can divide E equally among p processors, letting P_0 verify the m/p first edges, P_1 the next m/p edges, and so on. Our implementation follows this approach, which suits well for the distribution of the graph.

Before we take a closer look at the distribution, we mention that when different parts of the graph have different density, it is not possible to detect which edges take long time to process and which take short time. If we had known that, we could distribute the edges unevenly among the processors. Instead it is likely that edges close to each other have almost the same degree. One of several approaches is then to let processor P_0 process the edges $1, m/p+1, 2m/p+1, \ldots, (p-2)m/p+1$, P_1 process the edges $2, m/p+2, 2m/p+2, \ldots, (p-2)m/p+2$, and so on, where $E = \{1, 2, \ldots, m\}$. The distribution of the work load could then be improved.

Now we return to our distribution of the edges of the graph. From Section 3.2, we remember that our graph is represented by two vectors `adj` and

`lstart`, where element i in `lstart` points to the place where the adjacency-list of vertex i starts in `adj`. We give one processor the responsibility for all edges in adjacency-list i, but we keep in mind that one edge shall only belong to one processor. This distribution works well for huge graphs.

The details of the load balancing can be seen in Algorithm 4.5. We go through all edges of the graph, keeping in mind which processor will take responsibility for the edge we are counting at each time.

By *proc_assigned* we keep track of whether a processor is assigned or not, and by *present_proc* we know which processor is about to get its edges. *present_proc* counts from 0 to $p-1$, and as soon as a processor knows both *my_start* and *my_end*, pointing into `lstart`, it can start processing its edges.

We observe that the distribution of the load balancing is sequential in nature as long as all processors are computing the same numbers. However, there is no gain in letting one processor compute the distribution, since the other processors could not start the work, and we would have to communicate to tell the other processors what edges to work on.

Also note that the computation of load balancing is extra computations not needed in the sequential implementation, and therefore pure overhead.

4.3.2 The termination detection in detail

Now we give details of the termination detection of algorithms 4.3 and 4.4. As we did in Section 4.2, we let P_0 be the master processor, which is the destination of all the local answers, and responsible for telling all still working processors whether they ought to terminate.

To handle the termination detection we need to send and receive messages between processor P_0 and the other processors. Communication concerning sending and receiving messages by processors is the basis of message passing programming through `send` and `receive` operations. Generally, the operations have to specify which destination processor to send to or which source processor to receive from. For example, `send(msg, P0)` can be a call to a send routine, sending the message "msg" to processor P_0.

We differ between synchronous and asynchronous message passing.

Synchronous message passing[22]

The term synchronous is used for routines that actually return when the message transfer has been completed. A pair of processes, one with a synchronous send operation and one with a matching synchronous receive operation, will be synchronized, with neither the source processor nor the destination processor being able to proceed until the message has been passed from the source to the destination. Hence, synchronous routines intrinsically perform two actions: They transfer data and they synchronize processors.

Algorithm 4.5 Detailed load balancing

→ A connected graph $G = (V, E)$, p processors P_i, $i = 0, \ldots, p-1$
← Each processor P_i returns $my_E = (my_start, my_end)$

```
for each processor P_i do
  compute_m = ⌊m/p⌋;
  proc_assigned = FALSE;
  present_proc = 0
  for each vertex a ∈ V do
    if proc_assigned is FALSE then
      if I am present_proc then
        my_start = a;
      end if
      proc_assigned = TRUE;
      edge_counter = 0;
      if present_proc is P_{p-1} then
        my_end = n;
        return (my_E = (my_start, my_end));
      end if
    end if
    for each neighbor b to a do
      if a < b then
        edge_counter++;
        if edge_counter has reached compute_m then
          if I am present_proc then
            my_end = a;
            return (my_E = (my_start, my_end));
          end if
          present_proc++;
          proc_assigned = FALSE;
          Break for-loop;
        end if
      end if
    end for
  end for
end for
```

Asynchronous message passing

The term asynchronous is used for communication routines returning immediately, allowing the next statement to execute, whether the communication is completed or not. Later, completion can be detected by other routines. Asynchronous routines provide the ability to overlap communication and computation.

Communication routines

For our parallel implementation we need to use a synchronous receive routine, which we call `Recv(msg = ‘‘What message’’, source)`. In this call, *msg* is the buffer where the receiving message is stored, and *source* is the source processor, for example P_0. We let ANY_SOURCE denote that we can receive from an arbitrary processor.

To be able to control what message to receive, we let the receive routine have the extra syntax `=’’What massage’’`. It is then only possible to receive the indicated message. This is needed, because while processing the "LB-simplicial(my_E)" routine, processor P_0 is only interested in a message leading to abortion. Here we let ANY_MSG denote that we can receive an arbitrary message. In the MPI implementation this can be easily handled with the *tag*-parameter.

In addition we need asynchronous send and receive routines. We name the routines `Isend(msg, destination, request)` and `Irecv(msg= ‘‘What message’’, source, request)`, where I refers to the word *immediate*. The last parameter *request* is used to detect completion of the routines. Through a `Test(request, completed)` routine we can test if a `Isend` or `Irecv` is completed. The request parameter must match. We will for example name our request *requestA*, and `Test(requestA, comp)` will then detect whether a routine with *requestA* is completed. If it is, the parameter *comp* is set to true.

The described message passing routines have corresponding routines in the MPI.

The termination detection step by step

Returning to our algorithms, we see that the call to the "LB-simplicial(my_E)" routine in Algorithm 4.3 is done by all processors. In the routine, the first instruction is a call to the asynchronous `Irecv(msg != ‘‘All the edges in my_E are LB-simplicial’’, ANY_SOURCE, requestA)`. This call initiates the possibility to receive a message telling the present processor to terminate. This receiving operation is completed if another processor finds an edge that is not LB-simplicial. The completion has to happen before the present processor finds an edge which is not LB-simplicial, or finishes processing when all its edges is found to be LB-simplicial. This terminating message is

either received by processor P_0 from an arbitrary processor, or sent from P_0 to any other processor. That is why the source is set to ANY_SOURCE, and the message has to be something else than "All the edges in my_E are LB-simplicial".

The completion of the initial call to the asynchronous receive routine is tested for each edge processed while computing the global separator list, and for each minimal separator when checking the labeling.

As soon as a processor finds out that an edge is not LB-simplicial or that all its edges are LB-simplicial, it will try to tell processor P_0 so. Again we use an asynchronous routine, `Isend`. This is because we want the processor to terminate. If processor P_0 already has received a message that one of the other processors have found an edge not LB-simplicial, it will not receive any more messages from the processors.

After the processors have finished their call to Algorithm 4.4, "LB-simplicial(my_E)", they return to Algorithm 4.3. Only processor P_0 has more work to do by determining when we have a global answer. First it checks if it has discovered an edge not LB-simplicial, or during the computation of "LB-simplicial(my_E)" has received a message that another processor has found so. If it has, it tells all other processors to terminate.

If it has not, it starts to count the number of processors returning an answer. It does so by using a synchronous receive routine $p-1$ number of times. We use a synchronous routine because we know that all other processors have to send P_0 an answer.

This synchronous receive routine is equivalent to an asynchronous routine, immediately followed by a wait routine which will hold until the completion of the routine.

For each answer P_0 receives, it checks if the message is "At least one edge in my_E are not LB-simplicial". If it is, P_0 tells all processors to terminate.

The process of telling the processors to terminate is done by $p-1$ calls to the asynchronous send routine. We use an asynchronous routine because it is likely that some of the processors have already terminated and have not the possibility to receive the message. Processor P_0 must however carry on. After telling all the still working processors to terminate, processor P_0 returns the global answer that the graph is not LB-simplicial.

If P_0 has received $p-1$ answers, and no message is evaluated to "At least one edge in my_E are not LB-simplicial", all edges in the graph are LB-simplicial. Then processor P_0 has only left to return the global answer that the graph is weakly chordal.

Chapter 5

Performance results

Finally we will test our implementations and measure the performance. The questions we want to answer are

- How does the theoretical complexity of the sequential implementation correspond to the test results?
- What gains did the parallel implementation achieve?

To answer the two questions, we need graphs to test on. We will both generate own graphs, and use graphs available one the Web. These graphs are described in the first section of this chapter.

The second and the third section will try answering the two introduced questions, while we in the last section of the chapter draw the conclusion lines of our results.

5.1 Test graphs

The graphs we need to test our implementations can either be weakly chordal or not. To answer the first of our two questions, we need the graphs to be weakly chordal, since the theoretical time complexity is given as an asymptotic upper bound. Because if the graph is weakly chordal, every edge will be checked. If not, the computation may stop quite quickly. Therefore, to answer the first question we want to ***verify*** weak chordality of graphs, rather than recognize them. In Subsection 5.1.1, Elimination game, we show how we generate weakly chordal graphs.

In addition, to give a complete picture, we also need graphs which we do not know whether or not they are weakly chordal, and test them for this purpose. In Subsection 5.1.2 we introduce graphs to be recognized if they are weakly chordal.

A general property of the graphs we test on are their sizes. To compare the test results with big-O notation, we need huge graphs, so that the highest order term of the run time is the significant. This will also demonstrate

whether or not the algorithm is suitable for practical applications. In addition we want the number of edges m to be much greater than the number of processors in the parallel implementation.

5.1.1 Elimination game

Theorem 2.4 states that every chordal graph is weakly chordal. We now use this result to generate chordal, and thus weakly chordal graphs.

From Section 2.3, Recognizing chordal graphs, and Algorithm 2.1, Chordal graph recognition 1, we know that a graph is chordal if we repeatedly can remove a vertex whose neighborhood induces a clique. Such a vertex was called simplicial. We take advantage of this characterization by removing one vertex at a time from a given graph, after *making the vertex simplicial*. We then add the new edges to the original graph. This procedure is called "Elimination game" and can be seen in Algorithm 5.1. It was first introduced by Parter[20] as a simulation of sparse matrix factorization, and later by Fulkerson and Gross[8], who connected it to chordal graphs, and showed that all graphs resulting from this process are chordal.

Algorithm 5.1 Elimination game

$\rightarrow$ A connected graph $G = (V, E)$
$\leftarrow$ A connected weakly chordal graph $G^+ = (V, E^+)$

$G^+ = (V, E^+) = G$;
$G' = (V', E') = G$;
while $G' \neq \emptyset$ **do**
  Choose an arbitrary vertex $u \in G'$;
  Make $N(u)$ into a clique and copy the added edges into E^+;
  Remove u and all edges uv from G';
end while
return (G^+);

Now, the graphs we will generate should contain a given arbitrary number of vertices and edges in addition to be connected. The number of vertices does not make any challenge. For the matters of the edges and connectivity, we start by looking at the connectivity.

To be sure that the graphs we generate are connected, we simply start by making the given number of vertices into a tree. This is shown in Algorithm 5.2, where we let each of the n vertices in the wanted graph initially be a tree containing one vertex. Then we start connecting two an two arbitrary trees until we are left with only one tree. When connecting two trees, it is done by adding an edge between two arbitrary vertices in the trees.

For the matter of the number of wanted edges, which we denote m^*, we know that we want to add edges as long as $m^* > m$, where m is the number

Algorithm 5.2 Tree generator

```
→ G = (V, E = ∅)
← A tree G+ = (V, E+)

i = 0
for each vertex u ∈ V do
  tree_set[i] = u
  i++;
end for
while i > 1 do
  treeA = a random tree_set j, j = 1...i;
  treeB = a random tree_set j, j = 1...i, treeA ≠ treeB;
  a = a random vertex in treeA;
  b = a random vertex in treeB;
  E = E ∪ (ab);
  treeA = treeA ∪ treeB;
  delete treeB;
  i--;
end while
return (G+);
```

of edges currently in the graph. When we start generating the weakly chordal graph after the tree is generated, the optimal number of edges added per vertex is therefore $(m^* - m)/n = (m^* - (n-1))/n$. After i vertices are removed in the elimination game, the number is $(m^* - m)/(n - (i-1))$.

Just to run the elimination game may produce a number of edges far away from m^*. If the number is larger than m^*, there is nothing to do, but if we want more edges, we can add some more.

The way we add more vertices, is by comparing the number of edges we possibly may add to the graph to make a neighborhood a clique, to the optimal number of edges added per remaining vertex. So if the optimal added edges for a vertex v is add^*, we add random edges from the vertex, decrementing add^*, until add^* equals $deg(v)^2 - deg(v)$.

In this way we are not able to generate exactly m^* edges, but it can be a quite good estimate. To further control the number of added vertices, in addition to those added during elimination game, we correct the number by multiplying by a given sensibility *sense*. So we add one more edge if $add^* \geq sense \times (deg(v)^2 - deg(v))$, $0 \leq sense \leq m^*$. If *sense* is set to 0, that indicates that no edges need to be added to make the neighborhood of any vertex a clique, and all add^* edges are added. If it is set to 1, it says that the neighborhood is a clique, while setting *sense* to m^* leads to no edges added, disregarding those added in elimination game.

So if we try to generate a graph containing n vertices and m^* edges,

using a sensibility of 1, we can decrease the sensibility if we see that the number of generated edges are too few, and increase the sensibility if the number are too large. Then we have a heuristic which lets us control the number of edges well, although not perfect due to the randomness when the edges are added.

Another issue is that the degree for the first removed vertices in the elimination game is less than for the last removed. This is because add^* initially may be quite small, since the number of remaining vertices is large. Due to the described sensibility it may be extra small. We do not wish the effect that the degrees of the vertices vary much. To compensate we let more edges be added in the beginning. After some testing, we have come to the conclusion that $add^* = {add^*}^2/(i+1)$, where i is the number of removed vertices, is a good estimate to use. However, we eliminate not the issue, contradicting the assumption in Subsection 4.3.1, saying that the size of the neighborhood is even for each edge. We are therefore not completely satisfied with our generated graphs. The parallel test results may be worse than if the density was equal all over the generated graphs.

Algorithm 5.3 shows the generation of the graphs. First the graph is created with n vertices and no edges, before we ensure connectivity by calling "Tree generator", making the graph into a tree. Then we start the elimination game. Since the tree is randomly generated, we can use the natural ordering from 1 to n when removing one by one vertex. When removing vertex i, we can then regard all vertices $v < i$ as if they do not exist in the graph. Also note that when i is the present removed vertex, $i-1$ is the number of removed vertices.

For each removed vertex i, we calculate how many edges we should add, and add them if suitable, before we make its neighborhood into a clique.

A random edge is only tried added once, because when the graph is starting getting dense, there may be hard to find a non-existing edge. In fact the graph may be complete.

5.1.2 Matrix market

Matrices are useful sources for graphs if they have suitable attributes. We want our graphs to be connected and undirected. Square, symmetric sparse matrices can represent such graphs if they turn out to be connected.

Matrix market, a component of the NIST project on Tools for Evaluation of Mathematical and Statistical Software, at `http://math.nist.gov/MatrixMarket/` provides convenient access to a repository of test data primarily for use in comparative studies of algorithms for numerical linear algebra, featuring nearly 500 sparse matrices from a variety of applications.

Each matrix and matrix set has its own home page which provides details of matrix properties, visualization of matrix structure, and permits downloading of the matrix in one of several text file formats.

Algorithm 5.3 Weakly chordal graph generator

$\rightarrow$ Wanted number of vertices and edges, n and m^*, and sensibility *sense*.
$\leftarrow$ A connected weakly chordal graph $G = (V, E)$, where $|V| = n$ and $|E| \approx m^*$.

```
G = (|V| = n, E = ∅);
G = Tree generator(G); (Call to Algorithm 5.2)
for i = 1, n − 1 do
   add* = (m* − m)/(n − (i − 1));
   if add* > 0 then
      add* = add*^2/i;
   end if
   for j = 0, add* do
      if add* − j ≥ sense × (deg(i)^2 − deg(i)) then
         v = a random vertex k, k = i + 1 . . . n;
         if (iv) ∉ E then
            E = E ∪ (iv);
         end if
      end if
      for all pairs (u, v) ∈ Adj[i], u, v > i do
         if (uv) ∉ E then
            E = E ∪ (uv);
         end if
      end for
   end for
end for
return (G);
```

Among the provided sets, we find the CYLSHELL set resulting from finite element discretization of an octant of a cylindrical shell in the discipline of structural mechanics. The matrices there are all square and symmetric. Wanting huge graphs, the S3DKQ4M2 matrix is the largest with its dimension of 90449×90449 and 2455670 nonzero elements. Since the matrix is symmetric, only half of it is represented. Translating to graph terminology, we have a graph containing 90449 vertices and 2455670 edges.

Another set is the BCSSGRUC1 set from the Harwell-Boeing Sparse Matrix Collection containing standard test matrices arising from problems in linear systems, least squares, and eigenvalue calculations from a wide variety of scientific and engineering disciplines. BCSSTK13 is a matrix from a generalized eigenvalue problem in the mentioned set, having the same attributes as the previous matrix, but the size is much smaller. Its dimension is 2003×2003 and 42943 nonzero elements, including all elements at the diagonal. That is, we have a graph with 2003 vertices and 40940 edges, since we do not want edges starting and ending in the same vertex, corresponding to the elements at the diagonal.

For the purpose of recognizing graphs we will use these graphs, one quite big and one really huge, as examples, both sequentially and in parallel.

5.2 Sequential experimental results

Now we will try to answer the first of the two introduced questions from the beginning of this chapter.

- How does the theoretical complexity of the sequential implementation correspond to the test results?

By using Algorithm 5.3, Weakly chordal graph generator, we have generated two sets of weakly chordal graphs. The first set contains graphs with 5000 vertices and 25000 to 100000 edges, while the second set has graphs with 10000 vertices and 60000 to 130000 edges. The edge number of the graphs differ by 5000 from the closest graph, thus there are 16 graphs in the first set, and 15 graphs in the second set.

While running the sequential algorithm on the graphs, we measure the time consumed for computing the neighborhoods and minimal separators, the sorting of the separator list, the computing of the co-connected components and the checking of the labeling in all separators, and of course the total time consumed. The time t is given in minutes.

In the Figures 5.1 and 5.2 we can see the performance measured for the two sets of graphs. We there see that the computation of neighborhoods and minimal separators take most of the time, although both the sorting of the separator list and the computation of the co-connected components and checking their labeling have the same time complexity. However, returning

to our question, we want to know if the the total time fits into a second degree polynomial, since our proven time complexity is $O(m^2)$.

By applying the method of least squares[17], we can approximate polynomials to our sets of data. If we approximate a second degree curve, for $n = 5000$ we get

$$t = 4.2937 \times 10^{-10} m^2 + 4.9550 \times 10^{-5} m - 0.8223$$

and for $n = 10000$

$$t = -2.5007 \times 10^{-10} m^2 + 2.2289 \times 10^{-4} m - 7.5377$$

We also try to fit linear curves, yielding

$$t = 1.0329 \times 10^{-4} m - 2.2749$$

for $n = 5000$ and

$$t = 1.7523 \times 10^{-4} m - 5.3841$$

for $n = 10000$.

In Figure 5.3 and 5.4 the polynomials are plotted together with the total times. We see that for these graphs we do not get any significantly worse approximation by using a linear approximation. Our curves therefore look more like an $O(m)$ algorithm. At least for these two sets of graphs, we are well within the time complexity.

5.3 Parallel experimental results

Then we turn to the second of the introduced questions.

- What gains did the parallel implementation achieve?

To answer this question, we use both some of the graphs from the generated set of weakly chordal graphs with 10000 vertices and the mentioned graphs from Matrix market.

First we take a look at the graph resulting from the matrix S3DKQ4M2. After almost 88 hours, or equivalently 5258 minutes on a 1.3 GHz processor it is decided not to be weakly chordal. Our serial run time is then $T_s = 5258$ minutes. In Table 5.1 we see the run time, speedup and efficiency for 1, 2, 4, 6,..., 16 processors. In Figure 5.5 the speedup is plotted versus the number of processors. Recall from Section 1.4 that the number of processors is an upper bound on the speedup, denoted linear speedup and shown by the dotted line in the figure, but that superlinear speedup sometimes is observed in practice. We see that for a single processor the speedup of course is one,

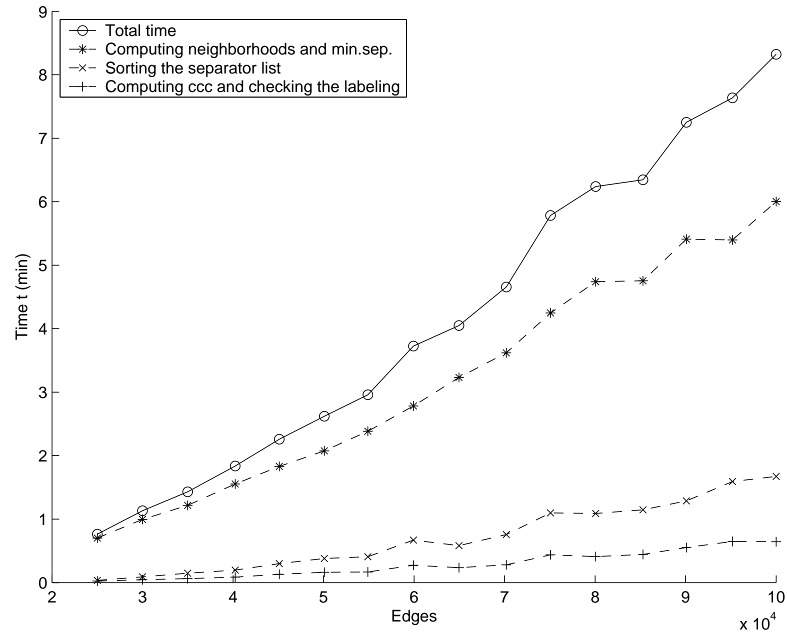

Figure 5.1: Verifying weak chordality of chordal graphs with 5000 vertices and increasing number of edges.

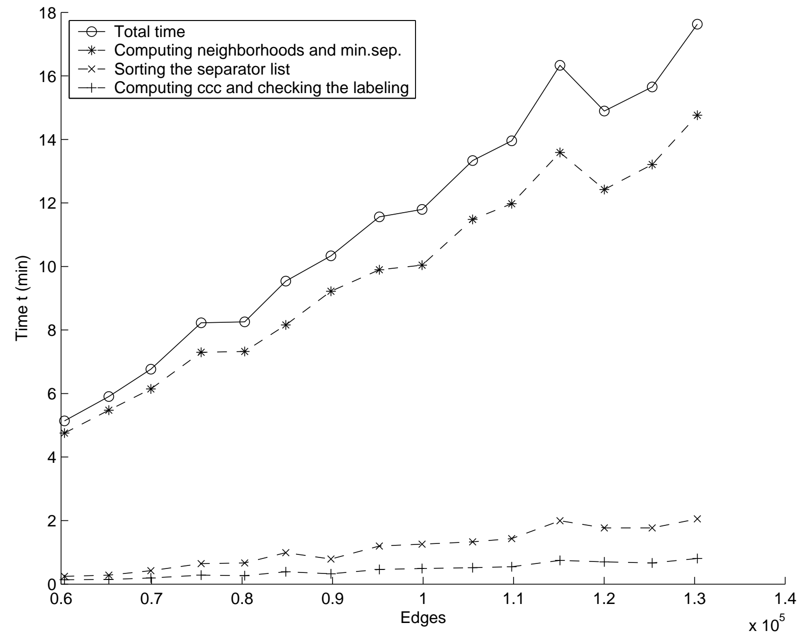

Figure 5.2: Verifying weak chordality of chordal graphs with 10000 vertices and increasing number of edges.

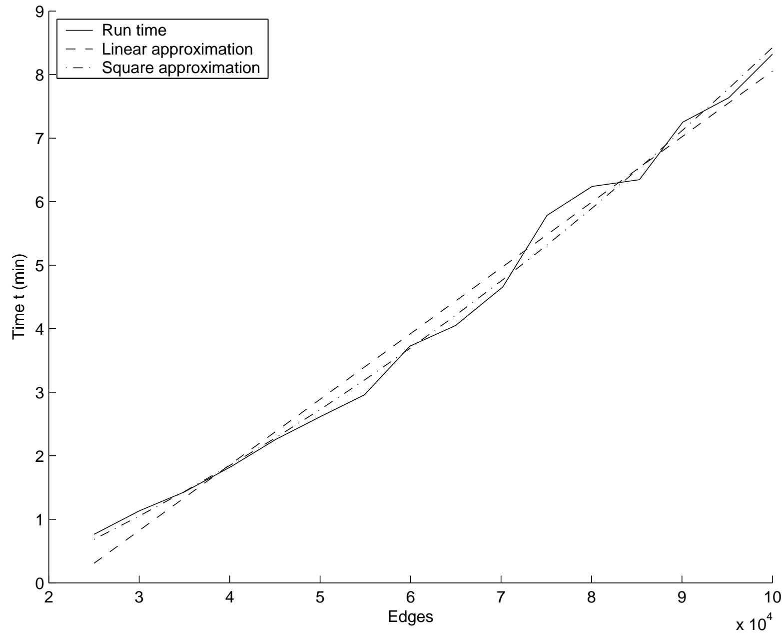

Figure 5.3: Square and linear approximation to the run time ($n = 5000$)

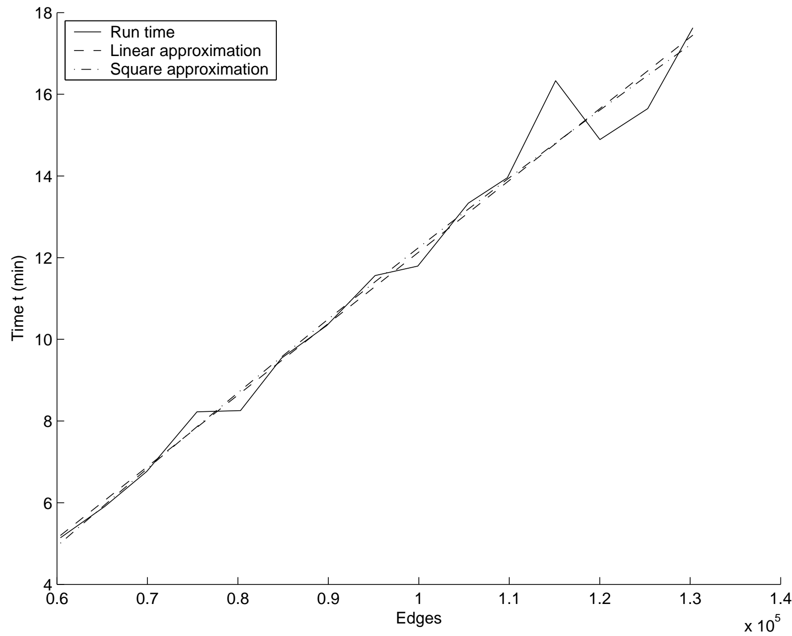

Figure 5.4: Square and linear approximation to the run time ($n = 10000$)

and for two processors it is 2.1, exceeding the number of processors, yielding superlinear speedup. When further processors are added however, speedup is less than the number of processors because of the described sources of overhead in Chapter 4. Though, an efficiency of 0.76 for 16 processors is pretty good.

p	Run time (min)	Speedup	Efficiency
1	5258	1.0	1.00
2	2505	2.1	1.05
4	1563	3.4	0.85
6	1223	4.3	0.72
8	892	5.9	0.74
10	801	6.6	0.66
12	602	8.7	0.73
14	549	9.6	0.69
16	436	12.1	0.76

Table 5.1: Run time, speedup and efficiency for increasing number of processors when recognizing if S3DKQ4M2 is weakly chordal

Next we look at the much smaller graph resulting from the matrix BCSSTK13. After only 1.17 minutes when using one 1.3 GHz processor, it is recognized not to be weakly chordal. Table 5.2 shows run time, speedup and efficiency for 1, 2, 4, 6, 10, and 14 processors. In Figure 5.6 the speedup is plotted versus the number of processors. Again we get a significant speedup, actual linear speedup for all the processors, even slightly exceeding linear speedup for 10 and 14 processors. Because of the short run times, this very good scalability should not be emphasized too much. We notice however, that our implementation works well both for huge and not so huge graphs.

p	Run time (min)	Speedup	Efficiency
1	1.17	1.0	1.00
2	0.61	1.9	0.96
4	0.34	3.4	0.86
6	0.21	5.6	0.93
10	0.11	10.6	1.06
14	0.08	14.6	1.04

Table 5.2: Run time, speedup and efficiency for increasing number of processors when recognizing if BCSSTK13 is weakly chordal

Finally we consider the set of graphs containing 10000 vertices and increasing number of edges. We choose the graphs with 60000, 80000, 100000,

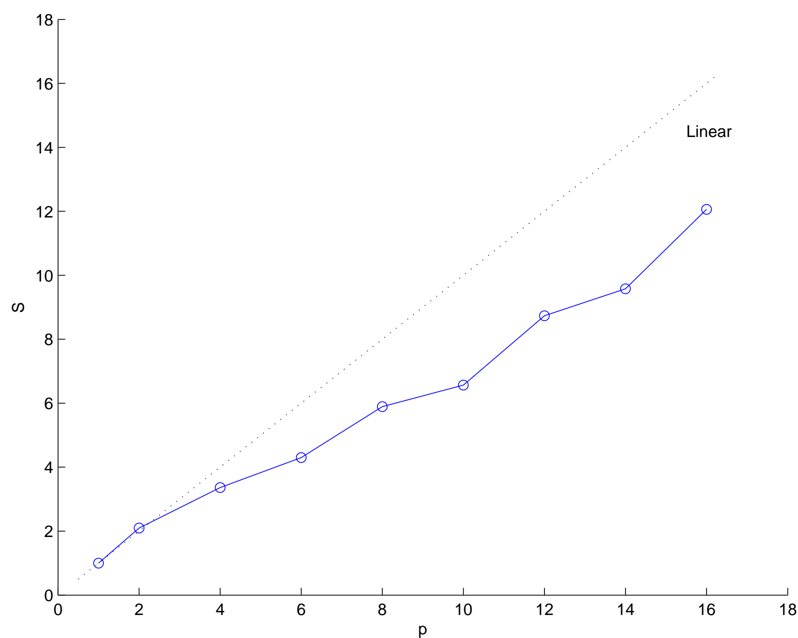

Figure 5.5: Speedup versus the number of processors when recognizing if S3DKQ4M2 is weakly chordal

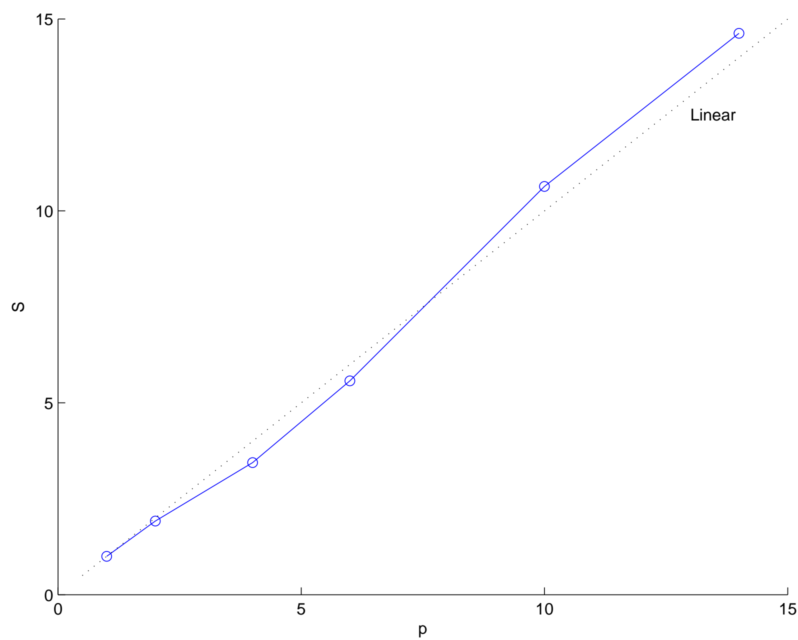

Figure 5.6: Speedup versus the number of processors when recognizing if BCSSTK13 is weakly chordal

and 120000 edges[1] to try to see if the size has any impact on the scalability. Theoretically, increasing the number of processors should decrease the speedup because of more overhead, and increasing the problem size should increase the speedup.

Table 5.3 shows the efficiency of the chosen graphs for 1, 4, 8, and 12 processors, while Figure 5.7 shows the speedup versus the number of processors. We see that the efficiency for all the graphs are quite stable, varying from 0.69 to 0.80, but slightly decreasing for increasing number of vertices. It is not possible to see any clear pattern for the size. As a matter of fact, the biggest graph has worst efficiency, contradicting that in theory, increasing size should improve speedup. However, this is not a general observation when we compare to the other graphs, so no clear pattern can be observed.

m	$p = 1$	$p = 4$	$p = 8$	$p = 12$
60000	1.00	0.80	0.80	0.71
80000	1.00	0.77	0.74	0.77
100000	1.00	0.78	0.70	0.76
120000	1.00	0.73	0.69	0.69

Table 5.3: Efficiency as a function of m and p for graphs with 10000 vertices

As previously mentioned in this chapter, we are not very satisfied with our generated graphs, because of varying density, which may decrease the speedup. Taking this into consideration, the efficiency for these graphs are very well.

[1] The exact number of edges are actually 60378, 80295, 99884, and 120029

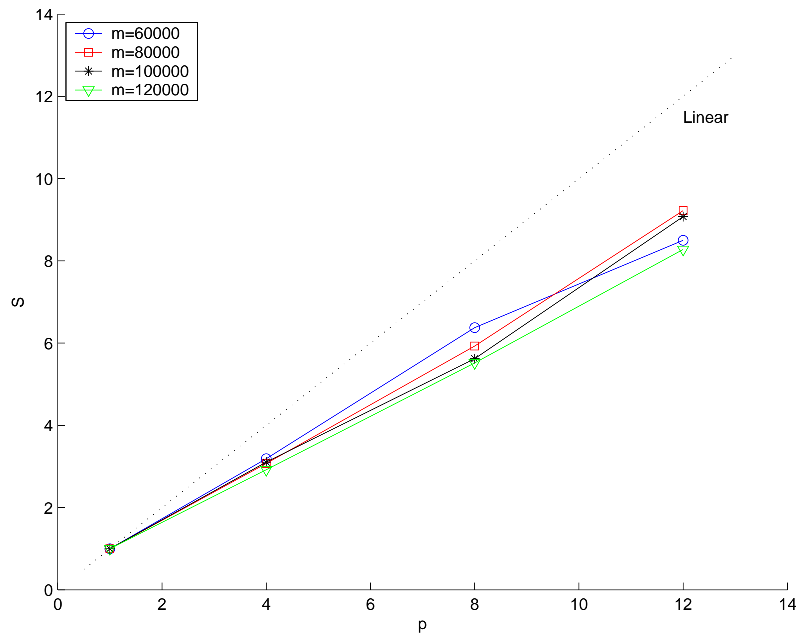

Figure 5.7: Speedup versus the number of processors when recognizing graphs with 10000 vertices and increasing number of edges for weak chordality

Chapter 6

Concluding remarks

This last chapter first gives a brief overview of the work and results in this thesis, before we mention some possible improvements for the future.

6.1 Overview of our work

The goal and motivation for this thesis has been to study practical implementations of the recognition algorithm for weakly chordal graphs by Berry, Bordat and Heggernes[5].

After introducing some topics essential for the thesis in the first chapter, the second chapter gave a more detailed and theoretical introduction to chordal and weakly chordal graphs. Among other things we did a detailed proof of Theorem 2.4, and a proof of Theorem 2.11 much simpler than the original proof. The chapter ended with the introduction to the studied recognition algorithm for weakly chordal graphs.

The third chapter presented the details of a sequential implementation of the algorithm. First we outlined how to obtain the time complexity of $O(m^2)$. Then we explored the data structures yielding a space complexity of $O(m^2)$, and gave details for each step of the algorithm. Of special interest was the computation of minimal separators by using a variant of depth-first graph search, the linear sorting of the global separator list, and the computation of co-connected components which we based on two observations.

In Chapter 4 we described how to parallelize and implement the algorithm with the Message passing programming model. When distributing the work load among several processors, we focused on minimizing overhead in load imbalance, communication and data transfer, and extra computations. After a total consideration of what would result in least overhead, we used a quite simple static load balancing by letting every processor check an equal number of edges for LB-simpliciality. To obtain a global answer as soon as possible we compared our algorithm to search algorithms, and developed a termination detection.

In Chapter 5 we performed both sequential and parallel tests. First we compared sequential test results with the proven worst time complexity of $O(m^2)$. For the graphs we used, the sequential run time was well within the time complexity, and looked more like an $O(m)$ algorithm. Second we measured the effect of the parallel implementation. Despite that we had several sources of overhead using static load balancing and $O(p)$ re-computations of each of the original co-connected components, we managed to obtain very well speedup and scalability for our test graphs for both verifying and recognizing weak chordality.

6.2 Future work

While working with the sequential implementation of our algorithm we have been mainly focused on describing the worst case time complexity. For the tested graphs in Chapter 5 we observed that we were well within this worst time. It would be interesting though, to further analyze if we could tighten the complexity in general or for certain types of graphs, for example sparse graphs, or construct worst case examples that match the given complexity.

In the parallel implementation we chose to use the Message passing programming model because it gave the most interesting parallel approach, and has through MPI succeeded to achieve high performance and scalability, which also our tests have shown. However, using the Shared memory programming model through OpenMP we would have the advantage that all the processors could share the data, instead of having copies of the graphs at all processors. Also the global separator list would be accessible to all processors. If our algorithm had been implemented using OpenMP, we could compare the run time and speedup using the two different models of parallel programming, gaining further knowledge about suitability of the two programming models.

Another interesting test would be to implement the dynamic load balancing described in Subsection 4.1.1, and observe differences in performance.

Last we will also mention that there are probably more advanced methods to generate random chordal graphs, so that they become totally random, and not with increasing degree for the removed vertices in the elimination game.

Bibliography

[1] *MPI: A Message-Passing Interface Standard*, 1994. Version 1.0 http://www.mpi-forum.org/docs/mpi-10.ps.

[2] *MPI-2: Extensions to the Message-Passing Interface*, 1997. Version 2.0 http://www.mpi-forum.org/docs/mpi-20.ps.

[3] *OpenMP C and C++ Application Program Interface*, 2002. Version 2.0 http://www.openmp.org/specs/mp-documents/cspec20.pdf.

[4] Anne Berry and Jean Paul Bordat. Triangulated and weakly triangulated graphs: Simpliciality in vertices and edges, 2000. 6th International Conference on Graph Theorry, Luminy France.

[5] Anne Berry, Jean Paul Bordat, and Pinar Heggernes. Recognizing weakly triangulated graphs by edge separability. In *LNCS 1851, Proceedings of Seventh Scandinavian Workshop on Algorithm Theory*, pages 139–149, 2000.

[6] Michael Bolton. 22 February 2000.

[7] G. A. Dirac. On rigid circuit graphs. *Anh. Math. Sem. Univ. Hamburg*, 25:71–76, 1961.

[8] D.R. Fulkerson and O.A Gross. Incidence matrices and interval graphs. *Pacific Journal of Math.*, 15:835–855, 1965.

[9] R. Hayward. Weakly triangulated graphs. *J. Comb. Theory*, 39:200–208, 1985.

[10] R. Hayward. Generating weakly triangulated graphs. *J. Graph Theory*, 21:67–70, 1996.

[11] R. Hayward. Meyniel weakly triangulated graphs - 1: co-perfect orderability. *Discrete Applied Mathematics*, 73:199–210, 1997.

[12] R. Hayward, J. Spinrad, and R. Sritharan. Weakly chordal graph algorithms via handles. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2000)*, 2000.

[13] D. Kratsch. *The structure of graphs and the design of efficient algorithms.* Habilitation thesis, Friedrich-Schiller Universität, Jena, Germany, 1995.

[14] V. Kumar, Ananth Grama, Anshul Gupta, and Geor Karypsis. *Introduction to Parallel Computing, Design and Analysis of Algorithms.* The Benjamin/Cummings Publishing Company, Inc, 1994.

[15] Kenneth C. Laudon and Jane P. Laudon. *Essentials of Management Information Systems, Managing the Digital Firm.* Prentice-Hall, Inc., 2002.

[16] C. Lekkerkerker and J.C. Boland. Representation of a finite graph by a set of intervals on the real line. *Fund. Math.*, 51:45–64, 1962.

[17] Jostein Lillestøl. *Sannsynlighetsregning og statistikk med anvendelser.* Cappelen Akademisk Forlag, 1997.

[18] S. B. Lippman and J. Lajoie. *C++ Primer.* Addison Wesley Longman, Inc, third edition, 1998.

[19] Peter S. Pacheco. A user's guide to mpi, 1995.

[20] S. Parter. The use of linear graphs in gauss elimination. In *SIAM Rev. 3*, pages 119–130, 1961.

[21] Michael Sipser. *Introduction to the Theory of Computation.* PWS Publishing Company, 1997.

[22] Barry Wilkinson and Michael Allen. *Parallel Programming, Techniques and Applications Using Networked Workstations and Parallel Computers.* Prentice-Hall, Inc., 1999.