Computational Experiments on Graph Width Metrics

by

John Fouhy

A thesis
submitted to the Victoria University of Wellington
in fulfilment of the
requirements for the degree of
Master of Science
in Computer Science.

Victoria University of Wellington
2003

Abstract

Many real-life problems can be modelled using graphs. However, the solutions to these problems are often $\mathcal{NP}$-complete. In 1986, Robertson and Seymour introduced the notion of a tree-decomposition of a graph, with an associated width called the treewidth, as a formal characterisation of how "tree-like" a graph is. Subsequently, Arnborg and Proskurowski developed techniques for using tree-decompositions to construct dynamic programming algorithms for solving problems such as DOMINATING SET, VERTEX COVER and INDEPENDENT SET.

Unfortunately, the problem of finding a minimum width tree decomposition for a graph is $\mathcal{NP}$-complete in general. Moreover, such dynamic programming algorithms are exponential in w, the width of the decomposition. As such, there is an interest in developing heuristics to estimate a graph's treewidth. But for any such heuristic to be useful, it must be on the one hand efficient, and on the other hand accurate.

In this thesis, we discuss upper bound heuristics for general graphs based on graph triangulations and on vertex separators. We also discuss a treewidth lower bound based on the theory of online graphs.

We then experimentally investigate the running time and accuracy of the algorithms on two classes of random graph.

This work grew out of a paper by Koster, Bodlaender and van Hoesel ([43]), and some of these results will be published in an updated version of this paper.

Acknowledgments

First of all, I would like to thank my supervisor, Professor Rod Downey, for introducing me to the fascinating field of parameterised complexity, and for guiding me and encouraging me through the course of this thesis.

I would also like to thank Professor Geoff Whittle for inspiring my interest in graph theory in the beginning, and for suggesting I talk to Professor Downey when I was considering whether to start a Masters. Thanks also to the other staff in the computer science and mathematics departments here — too many to name individually — for being excellent lecturers and generally nice people.

I think also Professor Hans Bodlaender and his colleagues, for having done so much in this field, and for believing me — eventually.

I would like to thank my fellow graduate students here, for being good friends, and the source of many strange and enjoyably esoteric conversations. I particularly mention my office mates, Simon and Azat, for putting up my foibles.

In the wider world, the ultimate frisbee community of New Zealand — particularly Frenzy, my team — provided some much needed balance in my life.

Finally, I must, of course, mention my family: Cathy and Claire, my sisters, and my parents Mike and Elaine, who supported me (financially and otherwise) through university, whichever course I wished to take.

Contents

Chapter 1

Introduction

This story begins in 1979. For 1979 was the year that Garey and Johnson ([29]) began to populate the space of $\mathcal{NP}$-completeproblems. Since that time, thousands of problems — many of practical interest or importance — have been found to be $\mathcal{NP}$-complete. Each of these problems is solvable, but each is as hard as any other, and none has a known solution that is better than exponential time.

This is bad news for programmers, for exponential algorithms run very slowly. As an example of just how slowly, consider a hypothetical machine capable of 10^8 operations per second, and a problem of size $n = 50$. Running times for three polynomial time algorithms and three exponential time algorithms are given in table 1. The differences are obvious. (a table like this can be found in almost any book on algorithms; I have taken mine from [25])

For a while, this was all we knew: No matter how much we wanted to solve them, $\mathcal{NP}$-completeproblems were intractable for any realistic problem sizes, and hence were (for all intents and purposes) insoluble. The best we could do is attempt to find a polynomial-time approximation scheme. Recently, however, Downey and Fellows have introduced a new idea: the notion of parameterised complexity ([25]).

The basic insight is that complexity analysis thus far had dealt with

Polynomial		Exponential	
Complexity	Running time	Complexity	Running time
n^2	25 μsec	1.5^n	6 min
n^3	1 ms	2^n	130 days
n^5	3 sex	3^n	230E6 years

Table 1.1: Example running times for algorithms with the given complexities, where $n = 50$ and 10^8 operations are executed every second.

the completely general case. But this means considering a whole range of pathological (or just plain unpleasant) cases — cases which may very rarely arise in *practical applications*. As such, we seek to place a bound on some other aspect of the problem — moving, in a sense, to a *two-dimensional* view of complexity.

This is best illustrated with an example. The poster-child for parameterised complexity is the VERTEX COVER problem from graph theory:

VERTEX COVER

Input A graph $G = (V, E)$

Output A set $S \subseteq V$ of minimum size such that for all $e = v_1, v_2 \in E$ either $v_1 \in S$ or $v_2 \in S$ (or both).

This problem is $\mathcal{NP}$-complete([29]) and thus (probably) not solvable in polynomial time. The best known general algorithms are exponential in n. However, in practice, we might have a situation where we *know* that the size of S is small — even though n itself might be large. It turns out that, in these circumstances, the VERTEX COVER problem can be solved in time $O(kn + 1.2852^k k^2)$ ([21]. ie: The parameterised problem PARAMETERISED VERTEX COVER can be solved for arbitrary n, provided k is less than 400 or so.

PARAMETERISED VERTEX COVER

Input	A graph $G = (V, E)$
Parameter	Positive integer k
Output	A set $S \subseteq V$ of size $\leq k$ such that for all $e = v_1, v_2 \in E$ either $v_1 \in S$ or $v_2 \in S$ (or both), if it exists. Return error, otherwise.

The general technique for solving PARAMETERISED VERTEX COVER involves constructing search tree, the height of which can be bounded by k. The details are interesting, but not directly relevant, so I will omit them. The reader is referred to Downey and Fellows' monograph [25] for discussion of a straightforward $O(2^k n)$ algorithm, or to Chen *et al* [21] for recent developments.

Chapter 2

Treewidth

2.1 Introduction

Many $\mathcal{NP}$-hard problems on graphs can be solved fairly easily, in polynomial time, when the input graphs are restricted to trees. Unfortunately, however, the simplicity that makes trees easy to deal with also means that the class of trees is too small to be useful in a great many circumstances. This has lead to the development, originally by Robertson and Seymour, of the idea of a *tree decomposition* of a graph, and an associated width for the decomposition, known as the *treewidth*: Informally, this is a measure of how "tree-like" a graph is. A graph that is very "tree-like" has a certain underlying structure which we may be able to exploit (if we know about it).

There are a number of characterisations of treewidth. I will describe two here, one based on the notion of partial k-trees and another based the idea of tree-decompositions. There are others (for example graphs of bounded treewidth can be elegantly described using parse trees for tree automata).

2.2 Treewidth and k-trees

Consider a technique for constructing a tree: You begin with two vertices connected by a single edge — the smallest possible tree, if we discount the case of a single isolated vertex (or the empty tree). Then, "grow" the tree by repeatedly adding a new vertex and connecting it to a single vertex in the existing graph.

We can generalise this process as follows:

Definition 2.2.1 (k-trees). A k-tree is a graph defined inductively as follows:

1. A $(k+1)$-clique is a k-tree.
2. If G is a k-tree and v is an isolated vertex, a new k-tree can be constructed by connecting v to exactly the vertices of a k-clique in G.

In the case of $k = 1$, this clearly fits the description of (ordinary) trees I gave in the previous paragraph.

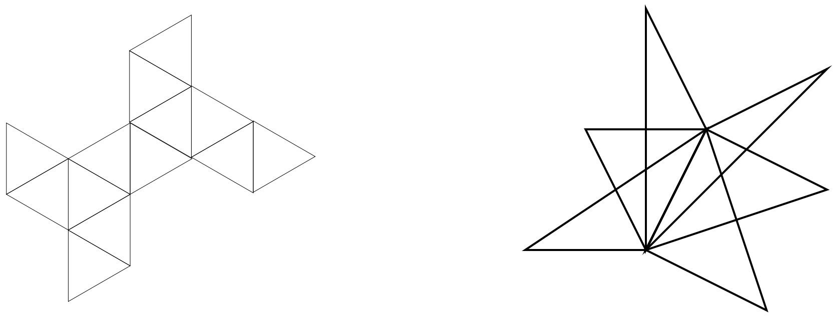
One way to visualise this process is to consider k-simplices in $(k+1)$-space. For example, a 2-tree can be thought of as being built out of (2-dimensional) triangles in 3-dimensional space. Adding a new vertex corresponds to adding a new triangle, by connecting it to an existing one along an edge (ie: along a 1-simplex). A 3-tree can be visualised as built out of tetrahedra in 4-dimensional space.

Definition 2.2.2 (Partial k-trees). Let T be a k-tree. Then G is a partial k-tree if $G \subseteq T$.

Obviously, every partial k-tree is a partial $(k+n)$-tree, for $n \geq 0$. So we can define treewidth as:

Definition 2.2.3 (Treewidth). The treewidth of a graph G is k if k is the least integer such that G is a partial k-tree.

See, for instance, [41] for a more rigorous presentation.



(a) (b)

Figure 2.1: (a) A simple 2-tree. (b) A "star" 2-tree, demonstrating how the graph is made of 2-dimensional triangles in 3-dimensional space.

2.3 Tree decompositions

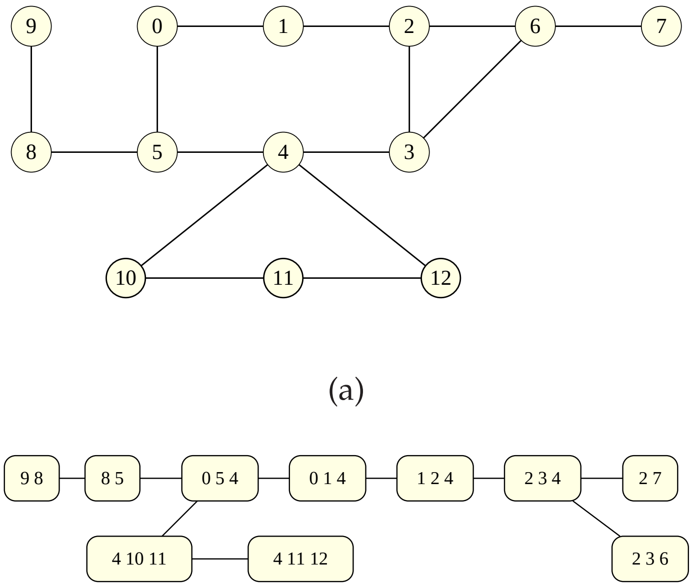
A more well-known way we can look at treewidth is in terms of *tree decompositions*, an idea due originally to Robertson and Seymour [49]. A tree decomposition of a graph G is a tree T such that every node of T is associated with some subset of $V(G)$. The structure of the tree should represent the "tree-like-ness" of the graph.

More formally:

Definition 2.3.1 (Tree-decompositions [49]). Let $G = (V, E)$ be a graph. A tree decomposition of G is a pair $(T, \mathcal{X})$, where $T = (I, F)$ is a tree and $\mathcal{X} = \{X_i : i \in I\}$ is a family of subsets of V, one for each node of T, such that:

1. $\bigcup_{i \in I} X_i = V$,
2. For every edge $\{v, w\} \in E$, there is an $i \in I$ with $v \in X_i$ and $w \in X_i$, and
3. For all $i, j, k \in I$, if j is on the path from i to k in T, then $X_i \cap X_k \subseteq X_j$.

Let T_v be the subgraph of T induced by $\{i \in I : v \in X_i\}$. Then an alternative to the third axiom is the statement that, for all v, T_v is a tree (ie: is connected).



(a)

(b)

Figure 2.2: (a) An undirected graph. (b) A tree decomposition of (a). Notice how one vertex (in this case, vertex 4) must be present in all the tree-nodes corresponding to the central cycle.

The *width* of a given decomposition is then one less than the size of the largest tree node, ie: $width = max\{|X_i| \ : \ i \in I\} - 1$

Definition 2.3.2 (Treewidth 2). The treewidth of a graph G is the minimum width over all possible tree decompositions of G.

A proof of the equivalence of this definition can be found, for example, in [41]. However, part of the proof can be seen in Figure 2.3 — namely, that tree decompositions of width k can be easily constructed from k-trees.

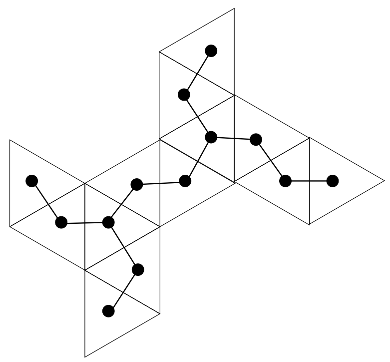


Figure 2.3: 2-tree showing underlying tree structure. In the tree decomposition, each node of the tree is associated with the three graph nodes surrounding it.

2.4 Value of tree decompositions

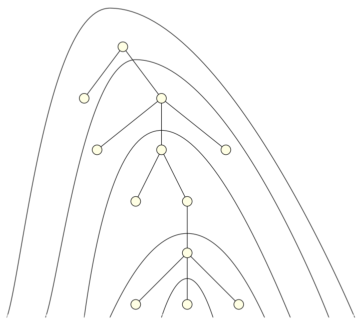
Earlier, we asserted that, as trees can admit easy solutions to hard problems, so can low tree-width graphs (ie: graphs which are very "tree-like"). In 1989, Arnborg and Proskurowski introduced a generic method involving the the algorithmic technique known as *dynamic programming* ([6]). See also Bodlaender ([14]) and Niedermeier ([46]).

Dynamic programming is an algorithm design strategy. In an nutshell, it involves solving the desired problem on multiple smaller sub-instances, and then combining (some of) those solutions into a solution for the whole problem. One of the primary issues in constructing a dynamic programming algorithm, then, is determining how to divide up the problem instance. In the case of graphs, we obviously want to work on subgraphs of the original graph. But how do we choose these subgraphs?

2.4.1 Dynamic Programming and Trees

Consider the situation with trees. In particular, consider the situation with *rooted trees*.

Definition 2.4.1 (Rooted trees). A rooted tree is a tree with one particular vertex identified as the *root*. In a rooted tree, we can talk about a vertex being the *parent* or *child* of another vertex, by considering their distance from the root: if {u, v} is an edge in a tree T with root r, and u lies on the simple path from r to v (ie: u is closer to r), then u is the parent of v and v is {the — a} child of u. Note that the root has no parent. If a vertex u has no children, then it is called a *leaf*.



A tree and some of its subtrees.

Rooted trees have a natural division into smaller graphs. We can examine subtrees consisting of a single vertex, toegether with all vertices "descended" from that vertex (ie: the vertex's children, its children's children, et cetera, down to the leaves).

We can then look to find the solution for a particular subtree by considering the solutions to each of its children's subtrees.

For example, consider the problem INDEPENDENT SET:

INDEPENDENT SET

Input	A graph $G = (V, E)$
Output	A set $S \subseteq V$ of minimum size such that for all $u, v \in S$, u is not adjacent to v.

We can solve this problem on trees as follows:

With each vertex v, we associate the subtree consisting of all its descendents. We also associate two integers in_v, out_v with v. These integers represent the size of the maximum independent set in the subtree that contains v, and the size of the maximum independent set in the subtree that does not contain v (if v is a leaf, then these integers are 1 and 0 respectively).

Now, let u be a non-leaf vertex with children $v_1, \ldots, v_k$. If u is part of the independent set, then none of its children can be, and so we set $in_u := \sum_{i=1}^{k} out_{v_i} + 1$. Otherwise, u is not part of the independent set, and so each of its children may or may not be, whichever is most beneficial. Thus, $out_u := \sum_{i=1}^{k} \max(in_{v_i}, out_{v_i})$.

The procedure terminates when we reach the root, with the greater of in_{root} and out_{root} being the size of the largest independent set. Moreover, it is not hard to see that we can then reverse the procedure to find out which vertices are present: Simply travel down from the root to the leaves, checking at each step whether in_v or out_v contributed to the final sum.

2.4.2 Dynamic Programming and Tree Decompositions

Why does dynamic programming work well with trees? One reason is that, in a tree, every vertex (apart from the leaves) is a separator — that is, its removal will separate the graph. To see the effect of this, consider a tree and three vertices u, v, w in the tree, such that v is on the simple path from u to w. The idea is that, in some sense, w cannot "affect" u, except in terms of its effect on v.

This is one of the problems facing us when we consider how to divide up a graph into subgraphs for dynamic programming: if we aren't careful, the subgraphs we choose may not be independent of each other, and so solutions on those sub problems could be very hard, or even impossible, to combine effectively. We need to formalise what it means for subgraphs to be separate.

Definition 2.4.2. Let $G = (V, E)$ be a graph. Then $S \subset V$ is an *a-b separator* if there exist vertices $a, b \in V$ such that a and b are in different components of $G \setminus S$.

We also say that $S \subset V$ is a *separator* or a *separating vertex set* if, for some $a, b \in V$, S is an a-b separator. Furthermore, if no vertex can be removed from S and have it still be separating, it is called a *minimal separator* or *minimal separating vertex set*.

Proposition 2.4.1 (Tree decomposition nodes are graph separators). *Let $G = (V, E)$ be a connected graph and $T = (I, F)$ be a tree decomposition for G with vertex bags $X_i \subseteq V : i \in I$. Then, for any i, $G \setminus X_i$ is disconnected, provided there exist graph vertices on either "side" of i, but not in X_i. In other words, let j, k, l be tree nodes such that:*

1. *$T \setminus k$ is disconnected, with j and l in different components.*
2. *There exist $u \in X_j$, $v \in X_l$ such that $u \notin X_k$ and $v \notin X_k$.*

Then $G \setminus X_k$ is disconnected.

Proof. Remember that we write T_u for the subtree of T induced by $\{i \in I : u \in X_i\}$, and remember that, for all vertices u, T_u *is* a tree (ie: is connected).

Now, let $u = w_1, w_2, \ldots, w_{s-1}, w_s = v$ be a path from u to v in G. We can then construct a path from j to k in T as follows: Start at j and explore T_u until you meet T_{w_2}. This must happen somewhere, as $\{u, w_2\}$ is an edge, and so by condition (2) of the tree decomposition definition, there is some $q \in I$ with $\{u, w_2\} \subseteq X_q$. This q must be in both T_u and T_{w_2}. Now follow T_{w_2} until you reach T_{w_3}, and so on. Eventually you will come to T_v, which you can then traverse to find l.

This is a path in T from j to l, and so k must be on this path. But that means that, for some vertex w_i, $k \in T_{w_i}$, and hence $w_i \in X_k$.

Thus every path from u to v must include a vertex in X_k, and so we conclude that $G \setminus X_k$ is disconnected. □

In this way, subgraphs induced by some X_i are isolated from the rest of the graph by their neighbours in the tree decomposition. And so, analogously with the cast of trees, we can design a dynamic programming algorithm based around subgraphs derived from the tree decomposition. The idea is to start at the leaves of the decomposition — these induce subgraphs that are, in some sense, on the "outskirts" of the graph — and work your way towards the root. For each vertex bag X_i, we construct a table of possible solutions to the problem. The table for each interior node is created from the tables of its children (and only those tables). The solution (if there is one) will then be found in the root table. The size of these tables will typically be exponential in the treewidth, leading to an algorithmic complexity of $O(k^w f(n))$, for a graph with n nodes and a tree decomposition of width w, where $k > 1$ is a (hopefully small) real number.

This technique, of computing tables of partial solutions, is originally due to Arnborg and Proskurowski ([6]). See also [14] and [46].

All this gets us nowhere, however, without some way to determine the treewidth of a graph, and find a tree decomposition.

In [12], based on work by Bodlaender and Kloks in [18], Bodlaender gives a parameterized linear-time algorithm which will determine, for a given k, whether or not the input graph has treewidth bounded by k, and if so, finds a tree decomposition with width at most k. Unfortunately, this algorithm has a high constant, namely 2^{ck^3}, where $c \approx 32$. This makes it impractical for general use.

The alternative is to look for heuristics which can be implemented efficiently. We begin this search in the next chapter.

Chapter 3

Triangulated Graphs

3.1 Introduction

The first class of heuristics we will examine is based on the notion of *triangulated* or *chordal* graphs.

Definition 3.1.1 (Triangulated graphs). A graph is called triangulated or chordal if every cycle of length at least four contains a chord (ie: an edge between two vertices not consecutive in the cycle).

Why look at triangulated graphs? Because of this result in [30]:

Definition 3.1.2 (Intersection graph). Let $\mathcal{S}$ be a family of sets. Then the intersection graph of $\mathcal{S}$ is a graph G where each vertex of G represents an element of $\mathcal{S}$ and two vertices are joined by an edge if and only if their corresponding sets have non-empty intersection.

Theorem 1 (Gavril, 1974). *A graph is triangulated if and only if it can be constructed as the intersection graph of subtrees of a tree.*

The idea of an intersection graph of subtrees of a tree is similar to the concept of a tree decomposition. Consider a graph G with tree decomposition T. The subtrees of T are those trees T_v for each vertex $v \in V(G)$ and their intersection graph is a graph H, where $V(H) = V(G)$ and $\{u, v\} \in E(H) \iff T_u \cap T_v \neq \emptyset$. Thus, Gavril's theorem is equivalent to stating that a graph is chordal if and only if it has a tree decomposition, with the additional constraint that $T_u \cap T_v \neq \emptyset$ only if u and v are adjacent.

One consequence of this is that any graph can be *triangulated* (ie: have edges added until it becomes a triangulated graph) without increasing its treewidth. To do this, take a tree decomposition of minimum width, and then add edges to the graph until all vertices that share a tree node are adjacent. Clearly, the treewidth has not increased (because the decomposition has not changed, and is still valid), and by Gavril's theorem, the graph is now triangulated.

Furthermore, in such a tree decomposition, every node corresponds to a clique in the graph. Thus, the treewidth of a triangulated graph is one less than the maximum clique size, and so to find the treewidth of a triangulated graph, we simply need to find all the maximal cliques.

3.1.1 Maximal Cliques and Perfect Elimination Schemes

In 1965, Fulkerson and Gross ([28]) proved that a graph is triangulated if and only if it has a *perfect elimination scheme*.

Definition 3.1.3 (Perfect Elimination Scheme). A *perfect elimination scheme* is an ordering σ on vertices of a graph: $\sigma :: \{1 \ldots |V(G)|\} \mapsto V(G)$ This ordering is such that, for every v, $\{v\} \cup M(v)$ induces a clique, where $M(v)$ is the set of all neighbours of v occurring later than v — ie:
$M(v) = \{u \in N(v) : \sigma^{-1}(u) > \sigma^{-1}(v)\}$.

Moreover, such cliques are exactly the maximal cliques of the graph.

Proof. Let $S \subseteq V(G)$ be a maximal clique, and σ be a perfect elimination scheme. Then, of all the vertices in S, some $u \in S$ must be ordered earliest. Clearly, $S \subseteq \{u\} \cup M(u)$. To see that $\{u\} \cup M(u) \subseteq S$, suppose we have some $v \in \{u\} \cup M(u) \setminus S$. Either v is adjacent to all the vertices in S, in which case S is not a maximal clique, or v is not adjacent to all the vertices in S, in which case $\{u\} \cup M(u)$ is not a clique. $\square$

This last fact — the knowledge that, with a perfect elimination scheme, we can find the maximal cliques of the graph — means that the ordering will easily give us the treewidth: simply take each simplicial vertex in turn, count how many neighbours it has, and then eliminate it (delete it from the graph). The largest number seen in the process is the treewidth.

Indeed, we can even go further and use the maximal cliques to find a minimum weight tree decomposition. Gavril gives one technique for doing this, but more recently in 1988, Shibata [51] and Jensen [39] independently discovered a better technique:

Definition 3.1.4 (Junction graph). Let G be a graph. Then the Junction graph of G is a pair $(H, \mathcal{C})$, where $\mathcal{C}$ is the set of maximal cliques of G and $H = (I, F)$ is a weighted graph, with one node for every element if $\mathcal{C}$. Two nodes i, j of H are adjacent if and only if their corresponding cliques have nonempty intsersection. Each edge has a weight equal to the cardinality of the intersection. ie:

- $ij \in F$ iff $\mathcal{C}_i \cap \mathcal{C}_j \neq \emptyset$
- $\text{weight}(ij) = |\mathcal{C}_i \cap \mathcal{C}_j|$

Theorem 2 (Shibata, Jensen). *If G is connected, then a minimum width tree decomposition for G is simply a maximum weight spanning tree in the Junction graph of G.*

(if G is disconnected, simply find the decompositions for the components of G and then join them together arbitrarily)

A perfect elimination ordering gives us the maximal cliques of the graph. Constructing the junction graph, thus, is relatively easy. The minimum / maximum spanning tree problem has been widely studied. Two well-known algorithms, *Prim's algorithm* and *Kruskal's algorithm*, can be implemented to solve the problem in time $\mathcal{O}(m \log n)$.

Jensen and Jensen give a simple proof of this in [40], based around Prim's algorithm.

It should be clear that any graph may enlarged into a triangulated graph by the addition of edges (as a trivial example, the graph could be made into a single large clique, which is triangulated). It should also be clear that the addition of edges will not cause the treewidth to *decrease*. Thus, triangulating a graph will give us an upper bound on the treewidth of the graph.

3.2 Triangulation

But how to triangulate the graph?

Suppose we have an ordering on the vertices of a graph, G. If it is not a perfect elimination scheme, there must be some vertex v such that $M(v)$ does not form a clique. But then we can simply add edges to make this true: Go through the elimination ordering one vertex at a time, and, whenever we come across a vertex which is not simplicial, add edges to make it simplicial. Then eliminate the vertex, keeping track of all the edges added.

This leads to Algorithm 3.1, sometimes called the Elimination Game.

Definition 3.2.1 (Fill-In). In Algorithm 3.1, the set of edges $\bar{E} \setminus E$ is known as the *fill-in*.

The issue that arises, then, is that of how to construct the ordering to be used in the Algorithm 3.1. Certainly, not any ordering will do — there will be many that will produce very bad triangulations.

There are two questions that arise here. The first is: How can we efficiently implement this algorithm? And the second is: How can we find orderings that lead to good triangulations?

We will address the first of these questions here, and the second in the next chapter.

Algorithm 3.1 Triangulation of a graph

Require: Graph $G = (V, E)$, vertex ordering $\sigma = [v_1 \dots v_n]$
Ensure: Triangulated graph $\bar{G} = (V, \bar{E})$, where $E \subseteq \bar{E}$.

  for all $v \in V$ **do**
    $n(v) := N(V)$
  end for
  $\bar{E} := E$
  $tw := 0$
  for $i = 1$ to n **do**
    if $\mid n(v_i) \mid > tw$ **then**
      $tw := \mid n(v_i) \mid$
    end if
    for all $u, w \in n(v_i)$ such that $uw \notin \bar{E}$ **do**
      $\bar{E} := \bar{E} \cup \{uw\}$
      $n(u) := n(u) \cup \{w\}$
      $n(w) := n(w) \cup \{u\}$
    end for
    for all $u \in n(v_i)$ **do**
      $n(u) := n(u) \setminus \{v_i\}$
    end for
  end for

3.2.1 Implementation Issues

In [53], Tarjan and Yannakakis give an efficient algorithm to compute fill-in, due originally to Greg Whitten. This algorithm is based around the idea of a follower of a vertex: that is, a neighbouring vertex ordered later by σ.

Definition 3.2.2. Let G be a graph, σ an ordering of the vertices of G, and $\bar{G}$ the triangulation of G using σ. Then for any vertex v, the *follower* of v is a vertex $follower(v)$ such that:

1. v is adjacent to $follower(v)$ in $\bar{G}$.
2. $\sigma^{-1}(follower(v)) > \sigma^{-1}(v)$.
3. No neighbour of v in $\bar{G}$ is ordered between v and $follower(v)$ by σ.

In other words: $follower(v)$ is the least later-ordered vertex adjacent to v in the triangulated graph.

We define a function $f^i(v)$ for $i \geq 0$ by $f^0(v) = v$ and $f^{i+1}(v) = follower(f^i(v))$.

Tarjan and Yannakakis then prove the following theorem:

Theorem 3 (Tarjan and Yannakakis, 1984). *An edge vw with $\sigma^{-1}(v) < \sigma^{-1}(w)$ is in $\bar{E}$ if and only if there is a vertex u such that $uw \in E$ and $f^i(u) = v$ for some $i \geq 0$.*

In other words: v is adjacent to w in the triangulated graph if and only if v is a "transitive follower" of some neighbour of w in the original graph.

This theorem leads to a fill-in algorithm: We work through the vertices in order from 1 to n, and keep track of the followers of vertices. Here, one of three situations will apply:

1. v has not yet been processed and $follower(v)$ is undefined.

2. v has been processed but v's follower has not yet been found: We set $follower(v) = v$ (so we can test for it later).

3. v has been processed and v's follower has been found: $follower(v)$ is set to the appropriate vertex.

Now, whenever we process a vertex w, we examine its earlier-ordered neighbours: If the neighbour has a follower, then the neighbour plays the role of vertex u in Theorem 3, with $follower(u) = v$, and so we can add an edge between w and v. Moreover, the same applies to any follower of v, and so on.

To help with the specifics, we keep track of an additional number for each vertex, called the index. When $v = \sigma(i)$ is processed, $index(v)$ is set to i. Subsequently, it is updated whenever we process higher-ordered vertices adjacent to v. Thus, $index(v)$ is always the maximum number in the set $\{\sigma(v)\} \cup \{\sigma(w) : vw \in \bar{E}$ and w has been processed $\}$. The $index$ array allows us to avoid, as much as possible, repeated processing of the same edges in $\bar{E}$. It also helps us avoid a nasty infinite loop that would otherwise be present..

Algorithm 3.2 runs in $O(n + m')$ time. The only problem with the algorithm is that it doesn't convieniently give us the treewidth. However, we can find that easily — and also find the maximal cliques, if desired — by taking the triangulated graph and using on it a variant of Algorithm 3.1, where we simply strip off simplicial vertices and record the clique sizes (Algorithm 3.3).

Algorithm 3.2 Fill-in Computation

Require: Graph $G = (V, E)$, vertex ordering $\sigma = [v_1 \dots v_n]$
Ensure: Triangulated graph $\bar{G} = (V, \bar{E})$, where $E \subseteq \bar{E}$.

```
for i = 1 to n do
  w := σ(i)
  follower(w) := w
  index(w) := i
  for all vw ∈ E such that σ^{-1}(v) < i do {For each earlier-ordered neighbour}
    x := v
    while index(x) < i do
      index(x) := i
      Ē := Ē ∪ {xw} {This may be an edge that was already in E}
      x := follower(x)
    end while
    if follower(x) = x then {If we haven't found the follower for x, then it must be w}
      follower(x) := w
    end if
  end for
end for
```

Algorithm 3.3 Treewidth test for triangulated graphs

Require: Triangulated Graph $G = (V, E)$, perfect elimination ordering $\sigma = [v_1 \dots v_n]$

Ensure: Treewidth tw of G.

```
for all v ∈ V do
    n(v) := N(V)
end for
Ē := E
tw := 0
for i = 1 to n do
    if | n(v_i) |> tw then
        tw :=| n(v_i) |
    end if
    for all u ∈ n(v_i) do
        n(u) := n(u) \ {v_i}
    end for
end for
```

Chapter 4

Lexicographic Breadth-First Search

4.1 Breadth-First Search

In the previous chapter, we asked the question: How can we find perfect elimination orderings? The first good answer to this question is due to Rose, Tarjan and Leuker in [50].

Rose *et al* decided to look at techniques based around Breadth-First Search (BFS). The basic idea of BFS is that one vertex is chosen as the "start" vertex, and then subsequent vertices are explored based on how close they are to the start node (or "root"). ie: if the minimum distance (in number of edges) from the root to a vertex u is less than the minimum distance from the root to a vertex v then u will be explored before v. If the distances are equal then the tie may be broken arbitrarily.

BFS can partition vertices of a graph into "shells" of common distance from the root. For example, if r is the root and i an integer, then we might denote $shell_i \stackrel{\text{def}}{=} \{u \;:\;$ the minimum distance from r to $u = i\}$. This defines a partial ordering on vertices where $u \prec_{BFS} v \iff \exists i \geq 0, j \geq 1$ such that $u \in shell_i$ and $v \in shell_{i+j}$. Now, suppose we have three vertices u, v, w, such that $\{u, v\}$ and $\{v, w\}$ are edges, and, for some

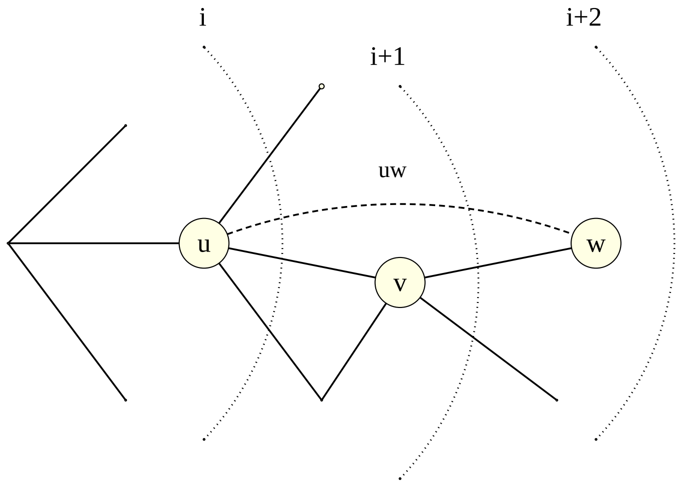


Figure 4.1: In this diagram, the vertices u, v and w have been placed into separate shells by the Breadth First Search algorithm. Thus: (1) the edge uw can not exist, as otherwise w and v would be the same distance from the root, and thus in the same shell; consequently, (2) v cannot be ordered earlier than both u and w, as then $v \cup M(v) \supseteq \{u, v, w\}$ would not be a clique.

i, $u \in shell_i$, $v \in shell_{i+1}$ and $w \in shell_{i+2}$. Rose, et al made the observation that, since there is necessarily no edge connecting u and w, in any perfect elimination ordering, at least one of u, w must be ordered before v (else $\{v\} \cup M(v)$ would not be a clique). This suggests there may be a perfect ordering consistent with the partial ordering imposed by BFS. See Figure 4.1 for an example. Indeed, consider the following two propositions:

Proposition 4.1.1 (Fulkerson and Gross). *Let G be a triangulated graph. Any simplicial vertex can start a perfect elimination scheme for G.*

Proposition 4.1.2 (Dirac). *Every triangulated, non-complete graph has two non-adjacent simplicial vertices.*

These two propositions imply that we can arbitrarily choose the vertex to take last place in the elimination ordering. Why? Suppose we choose a vertex v^* to be last. Perhaps v^* is simplicial; perhaps it is not. Either way, if the graph is not complete, Proposition 4.1.2 implies that there is a vertex $u \neq v^*$ which is simplicial, and Proposition 4.1.1 implies that that vertex can be chosen as first in the ordering. If the graph is complete and Proposition 4.1.2 does not hold, then, of course, every vertex is simplicial. Thus, at every step except the last, we can choose not to choose v^*, making it last in the ordering.

The upshot of this is that (still working on the assumption that the graph is triangulated) we can choose the root of our BFS to be the last vertex in the ordering. Now consider what this means: The root is ordered last, and thus is ordered later than every vertex in $shell_1$. Therefore, every vertex in $shell_2$ must be ordered *earlier* than every vertex in shell 1 (by Rose, et al's observation, above). Then repeat: every vertex in $shell_1$ is ordered later than every vertex in $shell_2$, so every vertex in $shell_2$ must be ordered later than every vertex in $shell_3$. This shows that, in a triangulated graph, a perfect elimination ordering must exist such that for all vertices u, v, if $u \prec_{BFS} v$ then $\sigma^{-1}(v) < \sigma^{-1}(u)$.

4.2 Lexicographic Labelling

The remaining issue to address is that of the ordering within the shells. One possibility is to apply the BFS algorithm again within each shell, but Rose et al propose a faster technique. The ordering is constructed backwards, with the root of the BFS in last place. With each vertex is associated a label, which is a list of integers, ordered from largest to smallest. Whenever a vertex is added to the ordering, say in position i, all unordered adjacent vertices are labelled with its number, i. The next vertex is chosen as the unordered vertex with the lexicographically-largest label.

Definition 4.2.1 (Lexicographic comparison). Let $L_1 = [p_1, p_2, \ldots, p_n]$ and $L_2 = [q_1, q_2, \ldots, q_m]$ be two lists ordered decreasingly. Then we define $L_1 > L_2$ if for some j, $p_i = q_i$ for $i = 1 \ldots (j-1)$ and $p_j > q_j$, or else $p_i = q_i$ for $i = 1 \ldots m$ and $m < n$. $L_1 = L_2$ if $m = n$ and $p_i = q_i$ for $i = 1 \ldots n$.

It should be clear that selecting vertices in the order of largest lexicographic label will be consistent with breadth-first search: Say there are n vertices, and we select a vertex to be ordered last (equivalent to the root of the BFS). Then all vertices in $shell_1$ will be labelled with n. As it is not possible for any other vertex to get n in its label (or any integer greater than n), any vertex in $shell_1$ will be lexicographically larger than any vertex in $shell_{m>1}$. Similarly for the other shells. Within a shell, it can likewise be seen that vertices will be selected in a deterministic breadth-first fashion.

This technique — of labelling adjacent vertices — will generate an ordering. If the graph is chordal already, then the ordering will be a perfect elimination ordering. And, as described earlier, a perfect elimination ordering will give us the maximal cliques for the graph, which in turn give us the treewidth. This algorithm is presented here as Algorithm 4.1 — it is called LEXP because it is a lexicographic (BFS) algorithm for finding Perfect elimination orderings.

Algorithm 4.1 LEXP BFS

Require: Graph $G = (V, E)$, vertex v^*

Ensure: Ordering $\sigma : 1 \ldots |V| \mapsto V$ such that G is triangulated $\iff$ σ is a Perfect Elimination Ordering.

```
for all v ∈ V do
  label(v) := ∅
end for
S := V
for i :=| V | to 1 do
  if i =| V | then
    u := v*
  else
    u := argmax_{v∈S} label(v)
  end if
  σ(i) := u
  S := S \ {u}
  for w ∈ N(u) ∩ S do
    label(w) := label(w) ∪ i
  end for
end for
```

4.2.1 Implementation Issues

The primary implementation issue here is how to deal with the labels — specifically, how can we arrange our data so that we can find the largest label as quickly as possible?

We can observe that the actual *values* of the labels are not important: we are only interested in the relative values of vertices. Moreover, once two vertices have been distinguished, they can never be undistinguished. As such, we only need to keep track of the ranking:

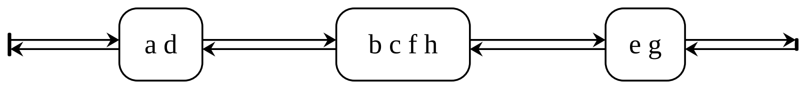
We have a list of bags, $B_1 \ldots B_k$, with the property that, for all i, the vertices in B_i have a larger label than the vertices in B_{i+1}. A vertex with largest label can then be found by selecting one from B_1. For the update step, we have to create a new bag and move the affected vertices into it. That is, if u is being ordered, then, for each bag B_i,

1. Find all the unordered vertices $v_1 \ldots v_r \in B_i$ adjacent to u.
2. Delete these vertices from B_i ($B_i := B_i \setminus v_1 \ldots v_r$).
3. Create a new bag B^* and insert it between B_{i-1} and B_i in the queue.
4. Insert the vertices into B^* ($B^* := \{v_1, \ldots, v_r\}$).

Thus, we can implement the algorithm using a doubly-linked list of bags, together with a map linking each vertex to its corresponding bag.

In Algorithm 4.2, $Top()$ refers to the first bag in the list, $Bagof(v)$ refers to the bag containing vertex v, $Choose(B)$ will return a single vertex in bag B (which must be nonempty), $New(B)$ indicates whether or not bag B is newly created this iteration (used in the update step), $NewBag()$ will create a new bag and $Next(B)$ and $Prev(B)$ refer to the next and previous bags relative to B in the list, respectively.

Over its running time, the Algorithm 4.2 will implicitly examine every edge twice (for each endpoint, it has to find the neighbours). It will also examine any isolated vertex. Thus, it has running time $O(n + e)$.



Before	
Vertex	Label
a	{ 12 10 }
d	{ 12 10 }
b	{ 11 10 8 }
c	{ 11 10 8 }
f	{ 11 10 8 }
h	{ 11 10 8 }
e	{ 9 }
g	{ 9 }

After	
Vertex	Label
a	{ 12 10 }
d	{ 12 10 }
b	{ 11 10 8 7 }
f	{ 11 10 8 7 }
c	{ 11 10 8 }
h	{ 11 10 8 }
g	{ 9 7 }
e	{ 9 }

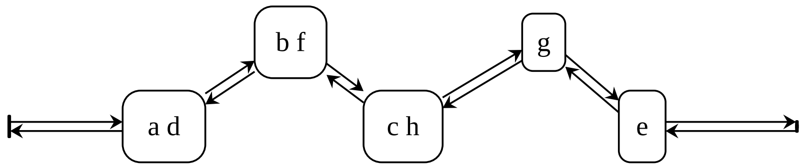


Figure 4.2: Demonstrating the equivalence between a list of bags and the actual labels. These pictures are of the structure immediately before and after a vertex v receives position 7 in the ordering. v is adjacent to b, f and g. Notice that none of the remaining numbers 6 – 1 could, for example, cause g to receive a label greater than or equal to that of c or h.

Algorithm 4.2 LEXP Implementation

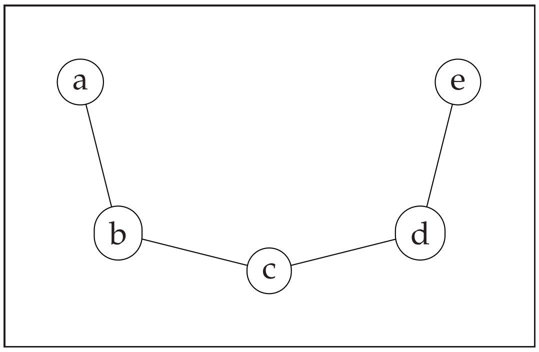
Require: Graph $G = (V, E)$, vertex v^*

Ensure: Ordering $\sigma : 1 \ldots |V| \mapsto V$ such that G is triangulated $\iff$ σ is a Perfect Elimination Ordering.

```
for all v ∈ V do {Initialise the list}
  Top() := Top() ∪ {v}
  Bagof(v) := Top()
end for
New(Top()) := 0
S := V
for i := |V| to 1 do
  if i = |V| then
    u := v*
  else
    u := Choose(Top())
    Top() := Top() \ {u}
  end if
  σ(i) := u
  S := S \ {u}
  Update labels using Algorithm 4.3.
  while Top() = ∅ do {Remove empty bags}
    Top() := Next(Top())
  end while
end for
```

Algorithm 4.3 Update labels — LEXP implementation subroutine

Require: Variables and data structures from Algorithm 4.2.
Ensure: Updated data structures.
  for all $w \in N(u) \cap S$ **do** {Find the vertices to be labelled}
    $B := Bagof(w)$
    if $B = Top()$ or $New(Prev(B)) = 0$ **then** {if we need to create a new bag}
      $B' := NewBag()$
      if $B = Top()$ **then**
        $Top() := B'$ {The new bag goes at the front}
      else
        $Next(Prev(B)) := B'$ {Link in the new bag}
      end if
      $Next(B') := B$
    else $\{New(Prev(B)) = 1$ — that is, we have already created a new bag before this one$\}$
      $B' := Prev(B)$
    end if
    $B := B \setminus \{u\}$
    $B' := B' \cup \{u\}$
    $Bagof(u) := B'$
  end for



Vertex	Label
a	10 9
b	9
c	9
d	9
e	10

Figure 4.3: A graph fragment together with labels. Vertex a will be ordered later than vertex e (ie: selected sooner), which will be ordered later than vertices b, c and d. Whatever the final elimination ordering is, the fill-in will include the edge ae. As such, a will be a neighbour of e when e comes to be processed by Algorithm 3.1.

4.3 Triangulating a Graph

In the case where the graph is not chordal, we can generate an ordering to be used with Algorithm 3.1. When using a LEXP ordering for triangulating a graph, however, a problem arises: there may be fill-in edges. As such, the sets of neighbours of a vertex before and after application of Algorithm 3.1 may be different.

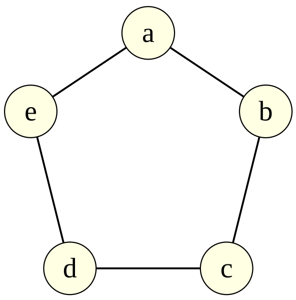
See Figure 4.3.

This was formalised by Rose, Tarjan and Leuker, in [50], as:

Theorem 4 (Rose, Tarjan and Leuker, 1976). *Let $G = (V, E)$ be a graph and $\sigma : 1 \ldots |V| \mapsto V$ an ordering on the vertices of G. Let $\bar{G} = (V, \bar{E})$ be the graph produced by running Algorithm 3.1 on G and σ. Then $\{u, v\}$ is an edge of $\bar{E}$ if and only if there exists a chain $[u, x_1, x_2, \ldots, x_k, v]$ in G such that $\sigma^{-1}(x_i) < \min(\sigma^{-1}(u), \sigma^{-1}(v))$ for $i \in 1 \ldots k$.*

This describes a possible trap which, for example, Koster, Bodlaender and van Hoesel fell into in [43]: There is a temptation to modify the algorithm as Algorithm 4.4 (see lines 13 - 17). The trap is to think that the procedure may be simplified by adding the fill-in edges at the same time as the ordering is constructed.

Tragically, this fails to work when there are fill-in edges. Consider the following counterexample:



If run with a in the last place, Algorithm 4.4 will produce one of the following orderings: $[d, c, e, b, a]$ or $[c, d, b, e, a]$, depending on an arbitrary choice between two vertices with equal labels. The key point is that it will all add *only one* edge — from c to e or from d to b — and this is not sufficient to triangulate the graph.

Once aware of this trap, it is easy to avoid: simply resist any temptation to take that "short-cut". The deeper problem, however, remains: Because fill-in may cause a vertex to gain neighbours, the LEXP algorithm is, potentially, labouring under *incorrect assumptions* (that is, assumptions about the neighbourhoods of vertices).

The solution is hinted at in Figure 4.3: If there is a path between two vertices u, v such that *all* the vertices on that path will be ordered before (chosen later than) *both* endpoints, then those u and v should be considered adjacent.

This leads to Algorithm 4.5. We modify the label update step to update the labels of *all* vertices which can be reached along a path of lower-labelled vertices. Note that, because of the properties of lexicographic labelling, if a vertex a receives a larger label than a vertex b at any stage of

Algorithm 4.4 Incorrect LEXP BFS modification

Require: Graph $G = (V, E)$, vertex v^*

Ensure: triangulated graph $G_T = (V, E \cup E_T)$, if G is triangulated, then $E_T = \emptyset$

$E_T := \emptyset$
for all $v \in V$ **do**
  $\text{label}(v) := \emptyset$
end for
$S := V$
for $i :=\mid V \mid$ to 1 **do**
  if $i = \mid V \mid$ **then**
    $u := v^*$
  else
    $u := \text{argmax}_{v \in S}\text{label}(v)$
  end if
  $\sigma(i) := u$
  for all $j, k \in \{i+1, \ldots, \mid V \mid\}, j \neq k$ **do**
    if $\{\sigma(i), \sigma(j)\} \in E \wedge \{\sigma(i), \sigma(k)\} \in E \wedge \{\sigma(j), \sigma(k)\} \notin E \cup E_T$ **then**
      $E_T := E_T \cup \{\sigma(j), \sigma(k)\}$
    end if
  end for
  $S := S \setminus \{u\}$
  for $w \in N(u) \cap S$ **do**
    $\text{label}(w) := \text{label}(w) \cup i$
  end for
end for

the algorithm, a will thereafter *always* have a larger label than b. As such, there is no danger of making these assumptions about the final state of the ordering before the ordering is fully determined.

Tarjan *et al* show that Algorithm 4.5 is guaranteed to produce *minimal* triangulations (ie: if H is a graph produced by Algorithm 4.5 and e is an edge of H then $H \setminus e$ is not triangulated). As such, the is known in this thesis as LEXM (ie: a Minimal version of LEXP). See [50] for details.

Algorithm 4.5 LEXM BFS

Require: Graph $G = (V, E)$, vertex v^*
Ensure: triangulated graph $G_T = (V, E \cup E_T)$, if G is triangulated, then $E_T = \emptyset$

```
E_T := ∅
for all v ∈ V do
  label(v) := ∅
end for
S := V
for i :=| V | to 1 do
  if i =| V | then
    u := v*
  else
    u := argmax_{v∈S} label(v)
  end if
  σ(i) := u
  S := S \ {u}
  for all w ∈ S : ∃ path [u = v_1, ..., v_{k+1}] in G with v_j ∈ S and label(v_j) <
  label(w) for j = 2, 3, ..., k do
    label(w) := label(w) ∪ i
  end for
end for
```

4.3.1 Implementation Issues

[50] describe an efficient implementation:

We can use a variation of a shortest path algorithm to search for the update paths: The cost of a path from u to w is the maximum label occurring on that path, excluding the endpoints. We then just have to compare the cost of the path with the label on the vertex to determine whether it is an update candidate.

The difficulty that arises, however, is that we (a) need to be able to compare the labels of two vertices, and (b) still need to be able to efficiently locate the vertex with largest label. In implementing LEXP, we introduced a function $Bagof(v)$ which would return the bag in our list containing v. We might think that, if we could somehow compare these two bags (according to their position in the list), we could compare the vertices, and still find the largest label vertex in constant time.

One way we could achieve this is by numbering the bags: We create a new function which associates an integer, $Num(B)$, with every bag B. If there are k bags in the list, then $Num(Top()) = k$ and, otherwise, $Num(B) = Num(Prev(B)) - 1$.

To find our paths, we will introduce a new data structure. This will be an array $reach$ of sets, such that $reach(i)$ will contain those vertices which can be reached by a path with label $\leq i$. We will use this to explore all paths with max label equivalent to bag 1, then all paths with max label equivalent to bag 2, and so on. Or, equivalently, we build a spanning tree of max weight 1, then a spanning tree of max weight 2, et cetera.

Algorithm 4.6 Implementation details for LEXM

Require: Graph $G = (V, E)$, vertex v^*

Ensure: Ordering $\sigma : 1 \ldots |V| \mapsto V$ such that G is triangulated $\iff$ σ is a Perfect Elimination Ordering.

```
for all v ∈ V do {Initialise the list}
  Top() := Top() ∪ {v}
  Bagof(v) := Top()
end for
New(Top()) := 0; Num(Top()) := 1; max := 1; S := V
for i := |V| to 1 do
  if i = |V| then
    u := v*
  else
    u := Choose(Top())
    Top() := Top() \ {u}
  end if
  σ(i) := u
  S := S \ {u}
  Found := Find() {Find vertices to be updated, using Algorithm 4.7.}
  Process Found using Algorithm 4.8.
  while Top() = ∅ do {Remove empty bags}
    Top() := Next(Top()); max := max − 1
  end while
  B := Top() {Renumber bags}
  for j := max to 1 do
    Num(B) := j
    B := Next(B)
  end for
end for
```

Algorithm 4.7 Find — LEXM vertex-finding subroutine

Require: LEXM Implementation variables and data structures.
Ensure: Set $Found \subseteq S$ of vertices to be updated.
  Initialise $reach(i)$ to $\emptyset$ for all i.
  $Found := \emptyset$
  Set $unreached(w) := true$ for all w.
  for all $w \in N(u)$ with $unreached(w) = \mathbf{true}$ **do** {Process u's neighbours first}
    $Found := Found \cup \{w\}$ {Label neighbours}
    $r := Num(Bagof(w))$
    $reach(r) := reach(r) \cup \{w\}$ {We can also reach the neighbours}
    $unreached(w) :=$ **false**
  end for
  for $j := 1$ to max **do**
    while $reach(j) \neq \emptyset$ **do** {Build spanning tree of max weight j}
      $w := Choose(reach(j))$
      $reach(j) := reach(j) \setminus \{w\}$ {w is on a lowest-weight path}
      for all $z \in N(w) \cap S$ **do** {Examine its neighbours}
        $unreached(z) :=$ **false** {Mark z reached}
        if $Num(Bagof(z)) > j$ **then** {If z has a higher label than anything on the path to it}
          $Found := Found \cup \{z\}$ {Then remember it}
          $reach(Num(Bagof(z))) := reach(Num(Bagof(z))) \cup \{z\}$ {We can get to z on a path of max weight $label(z)$}
        else {z's label is $\leq$ some label on the lowest-label path to here}
          $reach(j) := reach(j) \cup \{z\}$
        end if
      end for
    end while
  end for
  Output $Found$

Algorithm 4.8 LEXM vertex relabelling subroutine

Require: LEXM Implementation variables and data structures.
Ensure: Updated data structures.

```
for all w ∈ Found do {Process the vertices found}
  B := Bagof(w)
  if B = Top() or New(Prev(B)) = 0 then {if we need to create a new bag}
    B′ := NewBag(); max := max + 1
    if B = Top() then
      Top() := B′ {The new bag goes at the front}
    else
      Next(Prev(B)) := B′ {Link in the new bag}
    end if
    Next(B′) := B
  else {we have already created a new bag before B}
    B′ := Prev(B)
  end if
  B := B \ {u}
  B′ := B′ ∪ {u}
  Bagof(u) := B′
end for
```

Chapter 5

Maximum Cardinality Search

5.1 Searching through a k-tree

The algorithms we have looked at have been based on breadth-first search. What happens if we choose instead to look at depth-first search? Let us cast our minds back to the characterisation of treewidth in terms of k-trees. We wish to describe a way of traversing a k-tree depth-first. Clearly, we have to respect the structure of the k-tree. Simply treating it as an ordinary graph would most likely produce nothing of use. Instead, we would like to search in a manner similar to that shown in Figure 2.3, on page 8.

So how can we do this? Consider a k-tree, and find a $(k + 1)$-clique — this will be our starting point, and we explore these clique vertices in an arbitrary order. Now, recall the creation of k-trees: New vertices are added by connecting them to a k-clique in the existing tree. So these are the vertices we would like to consider for exploring next: vertices adjacent to a k-clique amongst vertices which we have already explored.

We can find these vertices simply by keeping, for each unexplored vertex, a count of the number of explored vertices it is adjacent to. When this count reaches k, the vertex is eligible to be explored. Will this work? On the one hand, an unexplored vertex adjacent to a k-clique of explored vertices will obviously have a count of at least k. And on the other hand, nothing else will. Consider: Suppose we have found a "problem" — a vertex u which is adjacent to k already-explored vertices not forming a clique. Then find the two closest (to each other) nonadjacent vertices amongst this clique, and the shortest path connecting them: call these v, w and P. Then P, together with uv and wu, forms a cycle of length at least 4 with no chords. But all k-trees are triangulated. Hence this cannot happen.

Now, we can use this technique to traverse a k-tree, and produce an ordering on the number of vertices. Find the last vertex explored. It will be adjacent to a k-clique of previously explored vertices, and not adjacent to any unexplored vertices (because it was last). Thus, it is simplicial. Remove it, and we will still have a k-tree, with a vertex ordering produced by this technique, so find the last vertex ...

In other words, this is another way of producing a perfect elimination ordering for a k-tree.

5.2 Maximum Cardinality Search

This is the idea Tarjan and Yannakakis look at in [53], in the context of a new way of searching for triangulations. As in the lexicographic breadth first searches, and for the same reason, we build the ordering backwards. But we choose the next vertex as simply the unordered vertex adjacent to the most ordered vertices.

I give it here as Algorithm 5.1.

We need to go to a bit more effort, however, to show that Maximum Cardinality Search (MCS) does, in fact, correctly find a perfect elimination ordering for a triangulated graph.

What can we determine about MCS? First, every ordering generated by MCS will have the following property:

Proposition 5.2.1. *Let $G = (V, E)$ be a graph, and $\sigma = [v_1, v_2, ..., v_n]$ be an ordering on V generated by Algorithm 5.1. Let $u, v, w \in V$ be any vertices with:*

- $\sigma^{-1}(u) < \sigma^{-1}(v) < \sigma^{-1}(w)$
- $uw \in E$
- $vw \notin E$

Then there exists a vertex x such that $\sigma^{-1}(v) < \sigma^{-1}(x)$, $vx \in E$ and $ux \notin E$.

Proof. Since $\sigma(u) < \sigma(v)$, v must be adjacent to at least as many later-numbered vertices as u. But w is adjacent to u and not v — it contributes $+1$ to the count of u and nothing to the count of v. As such, a vertex x must exist to "counteract" the effects of w. □

To show that MCS will correctly generate a perfect elimination ordering for a chordal graph, all we have to do is show that the first vertex in any MCS ordering is simplicial. After that, an easy induction finishes the job. I will follow the presentation of Fedor V. Formin [27]:

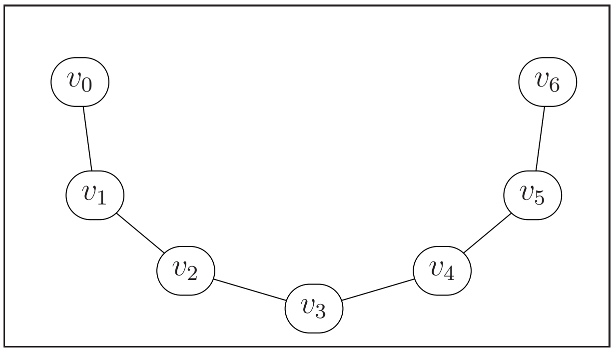
Definition 5.2.1 (Ditch). A *ditch* is an unchorded path of length at at least 3 (ie: a path where non-consecutive vertices are not adjacent) such that the end-points are ordered later by MCS than all the interior points.

For example, if $[v_0, v_1, \dots, v_k]$ is a ditch, then $\min(\sigma^{-1}(v_0), \sigma^{-1}(v_k)) > \max\{\sigma^{-1}(v_i) : 1 \leq i \leq (k-1)\}$.

We intend to show that no ditches exist. This will then imply that $\sigma(1)$ is simplicial. To do this, we will try to establish a contradiction:

Let $D = [v_0, v_1, \dots, v_k]$ be a ditch, such that $\min(\sigma^{-1}(v_0), \sigma^{-1}(v_k))$ is maximal over all possible ditches (Figure 5.1). Let $\sigma^{-1}(v_0) > \sigma^{-1}(v_k)$ (since we are dealing with *undirected* graphs, we can freely choose the "direction" of the ditch to make this true). Now,

- $\sigma^{-1}(v_1) < \sigma^{-1}(v_k) < \sigma^{-1}(v_0)$
- $v_1 v_0 \in E$
- $v_k v0 \notin E$ (as the path is chordless)



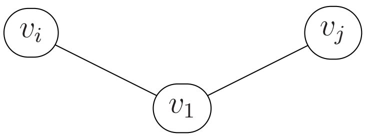
v	$\sigma^{-1}(v)$
v_0	12
v_1	6
v_2	5
v_3	3
v_4	7
v_5	9
v_6	10

Figure 5.1: A simple ditch (a chordless path with endpoints ordered after the middle vertices) together with positions in σ.

As such, by Proposition 5.2.1, there exists a vertex x such that $\sigma^{-1}(v_k) < \sigma^{-1}(x)$, $v_k x \in E$ and $v_1 x \notin E$ (see Figure 5.2).

Now, x may be adjacent to other vertices, but it may not be adjacent to v_1 (as above), nor may it be adjacent to v_0 (as that would create a chordless cycle, and no such cycles exist, by assumption that the graph is chordal). As such, there exists a $j \in 2 \ldots k$ adjacent to x such that $[v_0, v_1, \ldots, v_j, x]$ is a ditch. But $\sigma^{-1}(v_0) > \sigma^{-1}(v_k)$ and $\sigma^{-1}(x) > \sigma^{-1}(v_k)$, contradicting the choice of D that $\min(\sigma^{-1}(v_0), \sigma^{-1}(v_k))$ was maximal over all ditches. $\square$

To show that MCS will produce a perfect elimination ordering is now straightforward. Consider an MCS ordering $\sigma = [v_1, \ldots, v_n]$. The question we must ask is: is v_1 simplicial? If it is not, then there must exist two vertices adjacent to v_1, v_i and v_j which are not adjacent to each other. But then $[v_i, v_1, v_j]$ would be a ditch — and no ditches exist. $\square$

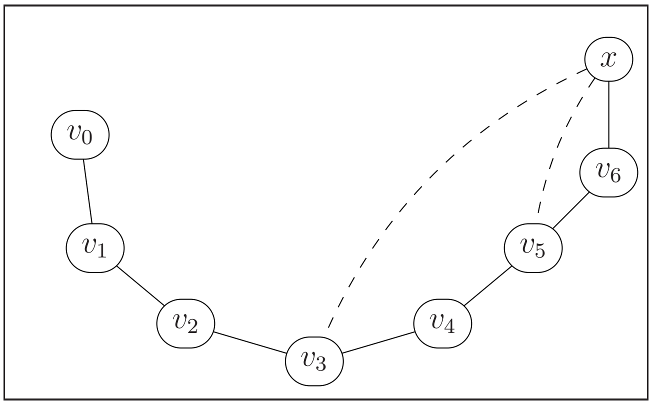


Algorithm 5.1 Minimum Cardinality Search

Require: Graph $G = (V, E)$, vertex v^*

Ensure: Ordering $\sigma : 1 \ldots |V| \mapsto V$ such that σ is a Perfect Elimination Ordering $\iff G$ is triangulated.

```
for all v ∈ V do
  counter(v) := 0
end for
S := V
for i :=| V | to 1 do
  if i =| V | then
    u := v*
  else
    u := argmax_{v∈S} counter(v)
  end if
  σ(i) := u
  S := S \ {u}
  for w ∈ N(u) ∩ S do
    counter(w) := counter(w) + 1
  end for
end for
```



v	$\sigma^{-1}(v)$
v_0	12
v_1	6
v_2	5
v_3	3
v_4	7
v_5	9
v_6	10
x	11

Figure 5.2: The ditch from Figure 5.1, showing the vertex x, which must exist by Proposition 5.2.1.

5.2.1 Implementation Issues

Again, the implementation issues centre around efficiently locating the vertex with largest count. In this case, we can best achieve this by keeping an array of sets of vertices, set, such that $set(i)$ is the set of all vertices v for which $counter(v) = i$.

Thus, Algorithm 5.2:

Algorithm 5.2 Minimum Cardinality Search Implementation

Require: Graph $G = (V, E)$, vertex v^*

Ensure: Ordering $\sigma : 1 \ldots |V| \mapsto V$ such that σ is a Perfect Elimination Ordering $\iff$ G is triangulated.

```
for all v ∈ V do
  counter(v) := 0
end for
Set(0) := V
S := V
max := 0
for i :=| V | to 1 do
  if i =| V | then
    u := v*
  else
    u := Choose(Set(max))
  end if
  σ(i) := u
  S := S \ {u}
  for w ∈ N(u) ∩ S do
    Set(counter(w)) := Set(counter(w)) \ {w}
    counter(w) := counter(w) + 1
    Set(counter(w)) := Set(counter(w)) ∪ {w}
    if max < counter(w) then
      max := counter(w)
    end if
  end for
  while Set(max) = ∅ do {Set(max) might now be empty}
    max := max − 1
  end while
end for
```

5.3 Minimal Triangulations with Maximum Cardinality Search

In [7], Berry, Blair and Heggernes present another heuristic. They note that, as LEXP can be changed to LEXM by labelling not just neighbours, but all vertices reachable along lower-ordered paths, so can MCS be modified to produce an algorithm they call MCSM (see Algorithm 5.3).

Algorithm 5.3 MCSM

Require: Graph $G = (V, E)$, vertex v^*

Ensure: triangulated graph $G_T = (V, E \cup E_T)$, if G is triangulated, then $E_T = \emptyset$

$E_T := \emptyset$
for all $v \in V$ **do**
  $\text{counter}(v) := 0$
end for
$S := V$
for $i :=\mid V \mid$ to 1 **do**
  if $i =\mid V \mid$ **then**
    $u := v^*$
  else
    $u := \text{argmax}_{v \in S}\text{counter}(v)$
  end if
  $\sigma(i) := u$
  $S := S \setminus \{u\}$
  for all $w \in S : \exists$ path $[u = v_1, \ldots, v_{k+1} = w]$ in G with $\text{counter}(v_j) < \text{counter}(w)$ for $j = 2, 3, \ldots, k$ **do**
    $\text{counter}(w) := \text{counter}(w) + 1$
    $E_T := E_T \cup \{uw\}$
  end for
end for

The rationale for this modification is the same as for LEXM: the need to compensate for future fill-in edges when considering which vertices are adjacent.

To see why it works, we have to look at the counter update step. As in Figure 4.3, we search for paths through vertices which will be processed later than the path's endpoints. The difference is that the lexicographic nature of LEXM means that once two vertices have been given labels, they will always retain the same relative order: It is not possible for the smaller-labelled vertex to subsequently acquire a larger label. With Maximum Cardinality Search, this is no longer true. So we must demonstrate that the "path property" illustrated in Figure 4.3 still holds — namely, that if we find a path $[u = v_1, v_2, v_3, \ldots, v_k, v_{k+1} = v]$ with $\mathrm{counter}(v_i) < \mathrm{counter}(u)$ and $\mathrm{counter}(v_i) < \mathrm{counter}(v)$ for $i = 2 \ldots k$, then $\sigma^{-1}(v_i) < \sigma^{-1}(u)$ and $\sigma^{-1}(v_i) < \sigma^{-1}(v)$ for $i = 2 \ldots k$.

To do this, we will stop the algorithm immediately before numbering a vertex, and check that, in any such path, the midpoints do not "rise above" the endpoints.

Let the vertex about to be numbered be u (in other words, at this stage, $\mathrm{counter}(u)$ is maximal). Let $[v, x_1, x_2, \ldots, x_k, w]$ be a path with $\mathrm{counter}(x_i) < \mathrm{counter}(v)$ and $\mathrm{counter}(x_i) < \mathrm{counter}(w)$ for all $i = 1 \ldots k$.

Now, we are about to number u, and increase the count for vertices according to Algorithm 5.3. The crucial thing we need to check is that after we do this, it is still the case that $\mathrm{counter}(x_i) < \min(\mathrm{counter}(v), \mathrm{counter}(w))$ for all $i = 1 \ldots k$. In other words, if any x_i has its count increased, the count needs to be increased for v and w as well.

But we can see that this will happen: if there is a path from u through lower-count vertices to x_i then there will likewise be a path through lower-count vertices to v and w — simply go from u to x_i, and thence to v or w (see Figure 5.3).

Berry *et al* go on to prove that adding edges to E_T at that stage in the algorithm triangulates the graph, producing, in fact, the same triangula-

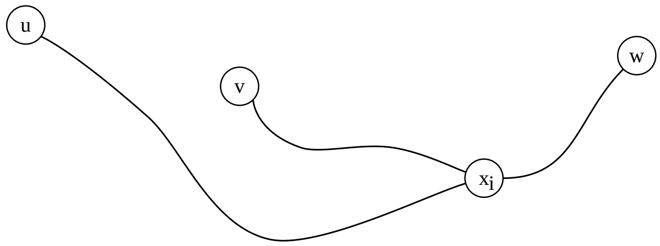


Figure 5.3: In this diagram, the arcs represent paths and the height of the vertices represents the count. The midpoints of the $v - w$ path will not be raised above the endpoints by the ordering of any vertex u.

tion as would be produced by Algorithm 3.1 with the ordering MCSM provides.

Note that this has the interesting implication that LEXM could be modified to avoid the pitfall described in the previous chapter. An example of this modification is shown in Algorithm 5.4.

Algorithm 5.4 LEXM BFS

Require: Graph $G = (V, E)$, vertex v^*

Ensure: triangulated graph $G_T = (V, E \cup E_T)$, if G is triangulated, then $E_T = \emptyset$

```
E_T := ∅
for all v ∈ V do
  label(v) := ∅
end for
S := V
for i :=| V | to 1 do
  if i =| V | then
    u := v*
  else
    u := argmax_{v∈S} label(v)
  end if
  σ(i) := u
  S := S \ {u}
  for all w ∈ S : ∃ path [u = v_1, ..., v_{k+1} = w] in G with v_j ∈ S and
  label(v_j) < label(w) for j ∈ 2...k do
    label(w) := label(w) ∪ i
    E_T := E_T ∪ {uw}
  end for
end for
```

We can prove this by following the technique of Berry *et al*:

Proof. Let G be a graph, H be the result of running Algorithm 5.4 on G, σ be the ordering produced, and $\bar{G}$ be the graph resulting from running Algorithm 3.1 on G with σ. To show that $H = \bar{G}$, we need to show that $uw \in H$ if and only if $\exists$ path $P = [u = v_1, \ldots, v_{k+1} = w]$ in G with $\sigma^{-1}(v_i) < \operatorname{argmin}(\sigma^{-1}(u), \sigma^{-1}(w))$ for $i = 2, 3, \ldots, k$. We can then appeal to Theorem 4 on page 31 (which says that such a path exists if and only if $uw \in \bar{E}$).

Suppose, without loss of generality, that $\sigma^{-1}(w) < \sigma^{-1}(u)$ (ie: u is processed before w). In one direction, it is simple: Say uw is added by Algorithm 5.4. Then a path $\{u = v_1, \ldots, v_{k+1} = w\}$ exists in G with $\operatorname{label}(v_i) < \operatorname{label}(w) \leq \operatorname{label}(u)$ for $i \in 2 \ldots k$. But we already observed that, if $\operatorname{label}(x) < \operatorname{label}(y)$, then $\sigma^{-1}(x) < \sigma^{-1}(y)$, as required.

Otherwise, suppose we have such a path P. We need to show that uw will be added by Algorithm 5.4.

We know $\operatorname{label}(u) > \operatorname{argmax}\{\operatorname{label}(v_2), \ldots, \operatorname{label}(v_k)\}$ and $\operatorname{label}(u) \geq \operatorname{label}(w)$.

We have to show that $\operatorname{label}(w) > \operatorname{argmax}\{\operatorname{label}(v_2), \ldots, \operatorname{label}(v_k)\}$. We know that $\sigma^{-1}(w) > \max\{\sigma^{-1}(v_2), \ldots, \sigma(v_k)\}$, which means that we have $\operatorname{label}(w) \geq \operatorname{argmax}\{\operatorname{label}(v_2), \ldots, \operatorname{label}(v_k)\}$. So suppose equality: for some $j \in 2 \ldots k$, $\operatorname{label}(w) = \operatorname{label}(v_j)$. But then when u is ordered, v_j will have its label increased and w will not, so that $\operatorname{label}(v_j) > \operatorname{label}(w)$. But this means that $\sigma^{-1}(v_j) > \sigma^{-1}(w)$, which cannot happen. $\square$

This is just an aside, however — our main focus is on the quality of the triangulation produced.

Chapter 6

Separating Vertex Sets

6.1 Separators

As we remarked earlier, bags in a tree decomposition form graph separators. This observation leads to the idea, due to [42], that we can *find* tree decompositions by searching for graph separators. More specifically, given a tree decomposition, we can try to improve it by using separators to split up large nodes.

We start with a most straight-forward observation: If there is one bag in a tree decomposition which is larger than all the others, then removing that bag (and replacing it with smaller bags) will reduce the width. If there are multiple largest bags, we need to remove them all, individually.

In order to do this, we find such a maximum size node (perhaps by storing the tree nodes in a priority queue), and consider the subgraph induced by the vertices associated with it. Say we call the node X and the subgraph H. If we can find a separating vertex set S in H, we can replace X by a "star" subtree, where the central node corresponds to S and each of the outer nodes corresponds to one component of $H \setminus S$. In particular, since we want to make the nodes as small as possible, we want to find a minimum separator. See Figure 6.1 for an example.

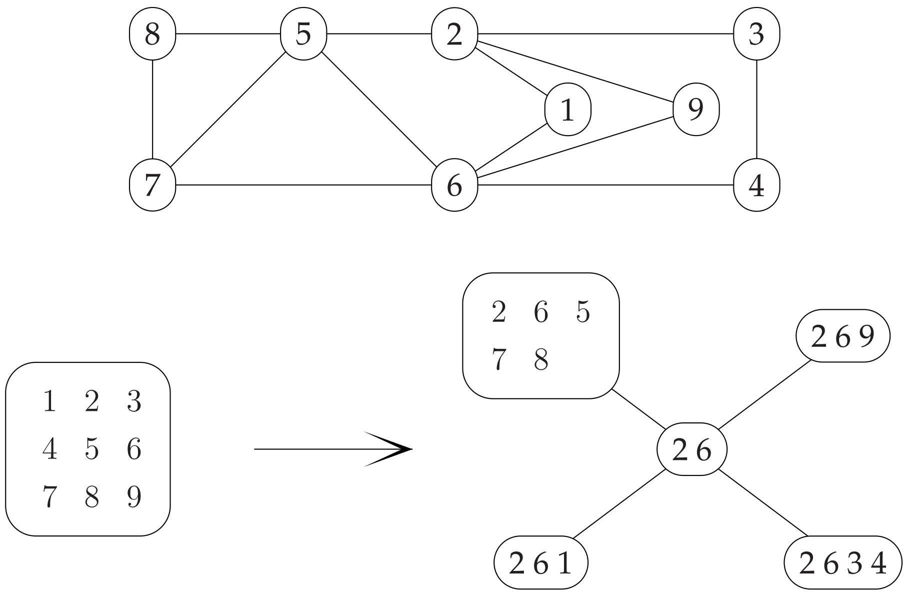


Figure 6.1: In this simple example of tree decomposition improvement, the vertices $\{2, 6\}$ constitute a minimum separating vertex set in the underlying graph. As such, they form the central node in the improvement, and are also present in all the new nodes created. Each of the four surrounding nodes corresponds to a different component of $G \setminus \{2, 6\}$.

In general, however, an interior node of the tree will be expanded, and the new nodes created may need to be connected to the rest of the tree — see Figure 6.2.

To give a precise description: Let $T = (I, F)$ be a tree and $(X) = \{X_1, X_2, \dots, X_{|I|}\}$ be the associated vertex bags. Let $i \in I$ be such that $|X_i|$ is maximal. We form $H := G[X_i]$, together with the additional edges $\bigcup_{j \in N(i)} C(X_i \cap X_j)$, where $C(X)$ denotes the complete graph with vertex set X (the reason for these additional edges will be seen soon). If H is complete then it has no separator, and so the decomposition cannot be improved. Otherwise, compute a minimal separator S in H and find the set of components $\mathcal{C} = \{C_1, \dots, C_m\}$ of $H \setminus S$. We create new tree nodes $i_0, i_1, \dots, i_m$, where $m = |\mathcal{C}| \geq 2$, and vertex bags $X_{i_0}, X_{i_1}, \dots, X_{i_m}$, where $X_{i_0} = S$ and $X_{i_k} = S \cup C_k$ for $k = 1 \dots m$.

Let $\bar{T}$ be the new tree we are creating. We form $\bar{T}$ as follows: First, remove i from T. Now, for each i_k, $k = 1 \dots m$, connect i_k to i_0. Finally, for each $j \in N_T(i)$, if there is a $k \in 1 \dots m$ such that $X_j \cap C_k \neq \emptyset$, connect j to i_k. Otherwise, connect j to i_0.

Now, there will be at most one such set, and this is the reason for the extra edges we are adding. Find a neighbour $j \in N(i)$ and consider any two vertices $u, v \in X_j$. We need to make sure that u and v do not get separated — and so we make them adjacent. Then uv will be an edge of H, even if it is not an edge of G, and so either u and v will occur in the same component of $H \setminus S$, or one or both of u, v are in S. Either way, however, X_j cannot have non-empty intersection with two different components of $H \setminus S$.

Now, if X_i was the only bag of largest size, then the width will have decreased: For any bag X_{i_k}, there must be at least one vertex $u \in H$ such that $u \notin X_{i_k}$, by the simple definition of S as a separator. Likewise, we must have $S \subsetneq X_i$. As such, the size of the largest bag amongst $i_0, \dots, i_m$ must be smaller than $|X_i|$.

It remains to show that T' is a valid tree decomposition.

Proof. We need to check the three tree decomposition properties:

1. The first requirement is that $\bigcup_{j \in I} X_j = V$. This is obvious: We have neither gained nor lost any vertices.

2. The second requirement is that for all $uv \in E$ there is some X_j with $u, v \in X_j$. So let uv be an edge of G. If $u, v \in X_{j \neq i}$ then there is nothing to do (since we are only changing X_i). Now, consider what happens if $u, v \in X_i$. Since uv is an edge, u and v cannot be separated by S. Thus, either $u, v \in S$ and hence $u, v \in X_i$ or u, v are in the same component of $H \setminus S$, in which case $u, v \in X_{i_j}$ for some j.

3. Finally, we have to show that, for every $v \in V$, T'_v is connected (remember that $T_v = T[\{j \in I : v \in X_j\}]$ and T'_v is defined analogously). If $v \notin X_i$ then $\bar{T}_v$ = T_v and hence is connected. Otherwise, $v \in X_i$. If $v \in S$ then for $j \in 0 \ldots m$ $v \in X_{i_j}$ and thus, by construction, $\bar{T}_v$ is connected. Else, for some j, $v \in X_{i_j}$. In this case, every neighbour of i in T containing v will be adjacent to i_j. And hence, $\bar{T}_v$ must still be connected.

$\square$

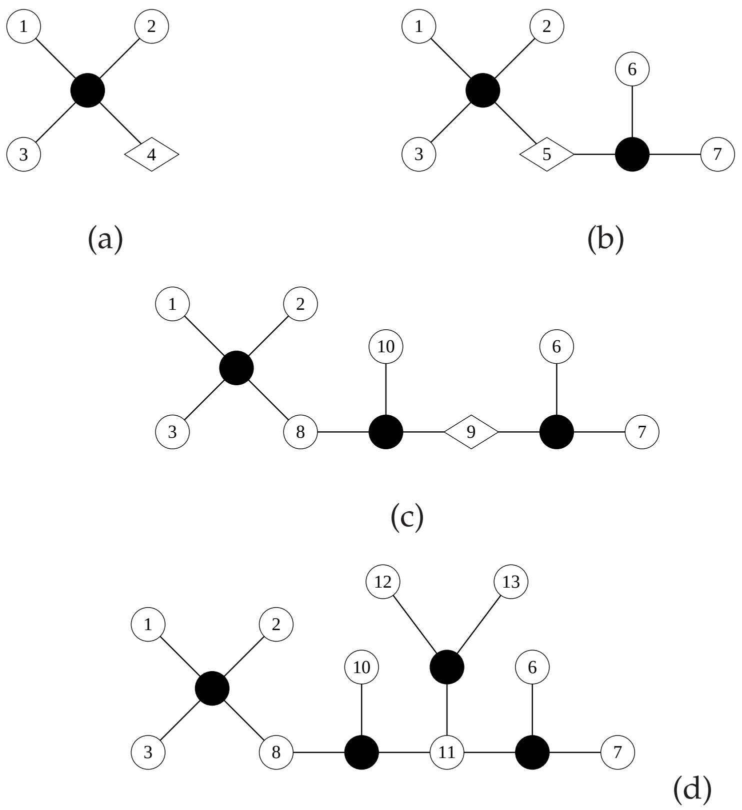


Figure 6.2: Four stages of a tree decomposition improvement. In each stage, the solid nodes represent vertex sets used to separate the graph at earlier intervals. These nodes are indivisible. The diamond nodes are the next ones to be chosen for separation. In (a), the graph induced by the vertices in node (4) is separated into three components. Those three components are associated with tree nodes (5), (6) and (7). Node (5) shares some vertices with the separating vertex set from (a), and so is connected to it. Nodes (6) and (7), necessarily, do not. In going from (b) to (c), the vertices associated with new nodes (8) and (9) both have nonempty intersection with other components of the graph. Finally, when node (9) in (c) is expanded, one of the resulting components is adjacent to both minimum separating vertex sets that bordered on (9).

Algorithm 6.1 Minimum Separating Vertex Set heuristic

Require: Initial tree decomposition, $(T, \mathcal{X})$, where $T = (I, F)$

Ensure: Modified tree decomposition, $(\bar{T}, \bar{\mathcal{X}}$, with $\mathrm{tw}(\bar{T}, \bar{\mathcal{X}}) \leq \mathrm{tw}(T, \mathcal{X})$

  while $\exists\ i \in I :\mid X_i \mid = \max_{j \in I} \mid X_j \mid$ and $H = (X_i, E(H))$ not complete, where $E(H) = \bigcup_{j \in N(i)} Q(X_i \cap X_j) \cup E_G[X_i]$ **do** {While we can find a separator}

    $N_{old}(i) := N(i)$ {Record the old neighbours of i...}

    $F := F \setminus \{ij : j \in N_{old}(i)\}$ {...and disconnect i from T.}

    let $S \subset X_i$ be a minimum separating vertex set in H

    let q be the number of components of $H[X_i \setminus S]$

    $I := I \cup \{i_1, \ldots, i_q\}$

    $F := F \cup \{i_j : j \in 1 \ldots q\}$

    for $p = 1$ to q **do**

      let $V_p \subset X_i$ by the set of vertices in component p of $H\left[X_i \setminus S\right]$

      $X_{i_p} := V_p \cup S$

    end for

    $X_i := S$

    for $j \in N_{old}(i)$ **do**

      if $\exists\ p \in \{1, \ldots, q\}$ with $X_j \cap V_p \neq \emptyset$ **then** {If the old neighbour was "adjacent" to one of the components ...}

        $F := F \cup \{i_p, j\}$ {... then join it to the appropriate new tree node.}

      else

        $F := F \cup \{i, j\}$ {Else, join it to the MSVS tree node.}

      end if

    end for

  end while

6.2 Implementation Issues

The most important implementation issue is finding minimum separating vertex sets.

Theorem 5 (Menger, 1927 [45]). *Let G be an undirected graph, and let u and v be nonadjacent vertices in G. Then, the maximum number of pairwise-internally-disjoint (u, v)-paths in G equals the minimum number of vertices from $V(G) \setminus \{u, v\}$ whose deletion separates u and v.*

So we can certainly find the cardinality of a minimum u, v-separator. But more than that: this should be suggestive of the famous max-flow-min-cut theorem, except that we are looking for a kind of vertex cut. And, indeed, we can approach the problem from a network flow perspective.

Let G be a graph and x, y be vertices. We wish to construct a capacitated network N such that, in a maximum flow, edges with nonzero flow will correspond exactly to vertices on disjoint paths from x to y. Then, edges in a minimum cut in N will correspond to vertices in a minimum x, y-separator in G.

We do this as follows: For every vertex u in G, we create two vertices u', u'' in N, together with an edge from u' to u'' of capacity 1. For every (undirected) edge uv in G, we create two edges $u''v'$ and $v''u'$ in N, each with capacity ∞. See Figure 6.3. In this construction, the only way "in" to a vertex pair $\{u', u''\}$ is at u', and the only way out is from u''. As such, any directed path in N between two vertices not adjacent must pass through one or more $u'u''$ edges — and these have capacity 1. Therefore, a minimum x, y-separator can be found by finding a minimum x'', y'-cut in N, and then forming the set $\{v \ : \ v'v'' \text{ is in the cut}\}$.

Network flow algorithms have been widely studied, and the problem can now be solved effectively on large graphs. See, for instance, [4] for an experimental study, or [31] for more recent developments.

Now, because we are dealing with a directed graph, we can't use algorithms such as that described in [55] which solve the all-pairs minimum

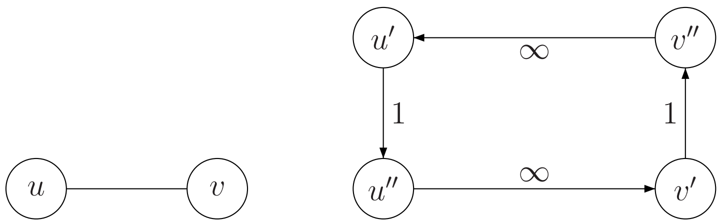


Figure 6.3: MSVS network creation.

cut problem on undirected graphs. As such, in order to find the minimum cut in the whole network, we need to solve multiple x'', y' flow problems. Or, to look at it another way, in order to find the minimum separator in G, we need to find the minimum a, b-separator for many pairs of vertices a, b. To be a bit less vague: Let u be a vertex in G, and find the closed neighbourhood $N_G[u]$. If S is a MSVS in G, then certainly $N_G[u] \not\subseteq S$ (as the neighbours of u on their own form a separator). As such, if v is a minimum degree vertex in G, we can find the MSVS by finding an x, y-separator (and hence solving one flow problem) for every $x \in N_G[v]$ and every $y \in \{G \setminus N_G[x]\}$.

Chapter 7

Online Graphs

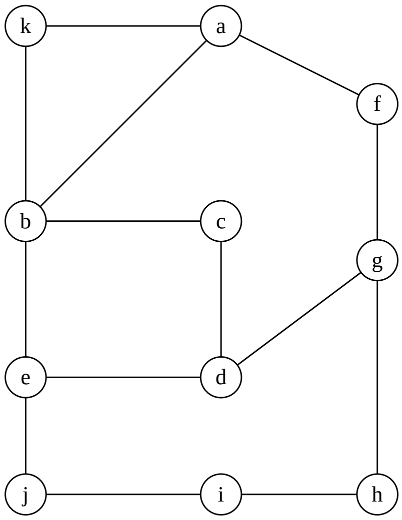
7.1 Online Problems

Suppose you are thinking of taking up skiing. You investigate the local ski slope, and discover that skis can be hired at a cost of $30 per day, or purchased for $300. What do you do? Clearly, if you are going to go skiing at least ten times in your life, it is in your best interests to purchase the skis now[1]. On the other hand, if you buy the skis and then decide you don't enjoy skiing, you will have wasted your money.

This is an example of an *online problem*: You need to make a decision now, yet you do not know what will happen in the future, and consequently you cannot be sure if your decision will be the right one. Such problems occur in the real world; for example, cache replacement algorithms need to decide which page to remove from the memory cache without knowing the future sequence of page requests.

The reader is referred to [19] or [3] for more information.

[1] In a suitably simplified financial world, that is.



Vertex	Neighbours
k	
b	k
a	b k
g	
f	a g
e	b
c	b
d	c e g
h	g
i	h
j	e i

Figure 7.1: Example online presentation of a graph.

7.2 Inductive Graphs

In a typical online presentation of a graph, the vertices will be presented in some order $v_1, v_2, \ldots, v_n$, such that when v_i is seen, we are informed which of the vertices in $v_1, \ldots, v_{i-1}$ it is adjacent to. See Figure 7.1 for an example.

In an online algorithm, we will have to make some decision about a vertex when it is given to us. Our ability to decide correctly will depend in part on how accurate our knowledge of that vertex is when it is given to us. If we know the entire local neighbourhood of a vertex when we see it, we may be able to make a good decision. On the other hand, if a vertex has many neighbours which we do not yet know about, we may choose badly.

We can formalise this notion as follows:

Definition 7.2.1 (d-inductive graphs). A graph G is said to be d-inductive if there exists some ordering $v_1, v_2, \ldots, v_n$ of the vertices of G such that, for all $i \in 1 \ldots n$, v_i is adjacent to at most d vertices in $v_{i+1}, \ldots, v_n$.

Note that this definition is independent of any *particular* online presentation. For example, in Figure 7.1, we give a 3-inductive presentation. However, it is not difficult to find a 2-inductive ordering, and the reader should be easily convinced that there is no 1-inductive ordering possible. Thus, the graph is 2-inductive.

7.3 Connection to Treewidth

Recall Section 2.2, where we discussed the notion of k-trees. Recall, in particular, the *inductive* creation of a k-tree. With a little thought, it should be clear that k-trees are k-inductive: Suppose we create a k-tree by starting with a $(k+1)$-clique $\{v_1, \ldots, v_{k+1}\}$, and then adding $n-(k+1)$ vertices $\{v_{k+2}, \ldots, v_n\}$, as described in Section 2.2. If we simply reverse this vertex ordering, we will arrive at a k-inductive vertex ordering.

Furthermore, it should also be clear that removing edges cannot increase the inductivity of a graph. As such, all partial k-trees are (at most) k-inductive.

Now, this immediately suggests an approach to finding a lower bound for a graph's treewidth: if we can show that a graph is d-inductive, then we will know that the graph's treewidth is at least d (because if the treewidth were less than d then the graph would be less than d-inductive).

We can easily determine the inductivity of a graph, as follows: Let G be a graph, and consider which vertex we are going to present first in the online ordering. Say we call this vertex u. Then clearly, the degree of u will be a lower bound for the inductivity of G. Moreover, once u has been selected, it plays no further part in determining the inductivity of a graph. Thus, $inductivity(G) = max\{degree(u), inductivity(G \setminus \{u\}\}$. This leads to Algorithm 7.1.

Bodlaender *et al* describe this algorithm in [43], where they refer to it as the Maximum Minimum Degree heuristic. They note that it can be implemented in time $O(n \log(\log(n)) + m)$ using a Van Embe Boas tree ([54]).

Algorithm 7.1 Determine the inductivity of a graph

Require: Graph $G = (V, E)$
Ensure: Inductivity of G
  for all $v \in V$ **do**
    $n(v) := N(v)$ {Record the neighbours of each vertex}
  end for
  $S := V$ {S will contain the vertices we have not yet processed}
  $ind := 0$ {ind will hold the maximum inductivity seen thus far}
  while $S \neq \emptyset$ **do**
    $u := \text{argmin}_{v \in S} |n(v)|$
    $S := S \setminus \{u\}$
    if $|n(u)| > ind$ **then**
      $ind := |n(u)|$
    end if
    for all $v \in n(u)$ **do**
      $n(v) := n(v) \setminus u$
    end for
  end while
  Output ind.

However, a straightforward implementation using a library priority queue will need only $O(n\log(n)+m)$ time, which is not significantly slower.

7.4 Colouring Inductive Graphs

Consider the problem of colouring a graph in an online fashion: We have to decide what colour to assign each vertex as we see them. The most natural way to do this is an algorithm known as First-Fit. Essentially, in First-Fit, we assign each vertex the smallest colour not already assigned to one of its (known) neighbours. For example, to First-Fit colour the graph from Figure 7.1, using the presentation given, we would arrive at:

Vertex	k	b	a	g	f	e	c	d	h	i	j
Colour	0	1	2	0	1	0	0	1	1	0	1

In this case, we have produced an optimal colouring. However, this is by no means guaranteed.

We are interested in this because this gives us another way of estimating the inductivity of a graph: Irani proved in [38] that, if a d-inductive graph is coloured online (even if the online ordering used is not itself d-inductive), the number of colours used satisfy $c \leq \log_{\frac{d+1}{d}}(dn)+d+1$. This means that, given values for c and n, we can numerically solve the above inequality, and hence obtain a lower bound for d (which is itself a lower bound for the treewidth k).

This bound has the advantage of being even faster to compute than the previous lower bound mentioned, running in $O(n+m)$. However, it has the disadvantage of being an extremely poor bound in practice, as will be seen later.

Algorithm 7.2 First-Fit Colouring

Require: Graph $G = (V, E)$

Ensure: Estimate of chromatic number for G

```
for all v ∈ V do
   n(v) := N(v) {Record the neighbours of each vertex}
end for
max := 0 {max will hold the maximum colour used thus far}
for all v ∈ V do {Process vertices of G in an arbitrary order}
   C(v) := min(N \ {C(u) : u ∈ n(v)}) {Find smallest available colour}
   if max < C(v) then
      max := C(v)
   end if
   for all u ∈ n(v) do
      n(u) := n(u) \ {v}
   end for
end for
Output max.
```

Chapter 8

Testing

8.1 Preliminaries and tools

All testing was done on a Sun Ultra80-450 — a 4x 450MHz UltraSPARC II — with 2048MB of RAM. However, no advantage was taken of the multiple processors (apart from the reduced interference from other programs). Algorithms were implemented using C++ and version 4.1 of the LEDA graph library.

Curiously, the LEXM and MCSM algorithms produced the same treewidth estimates for all graphs I ran them on. This is somewhat puzzling (they typically produce different vertex orderings, and certainly take different amounts of time to run). It may perhaps be explained by the fact that they are both searching for minimal triangulations, and by the similarity of my implementations of the label update step for each (in terms of searching for the paths through lower-labelled vertices). Furthermore, when tiebreaking is necessary in MCSM (as it often is), I choose (by default, not design) the earliest vertex added to the highest rank — which will result in breadth-first tendencies.

I have used `gnuplot 3.7p3` as my data analysis tool (augmented by much use of `awk` and shell scripts). When used to fit a function to data, `gnuplot` reports the asymptotic standard error for each parameter, as an indication of how closely the function fits the data. In cases where I have fit a straight line and given its slope (primarily the log plots in sections 8.2.2 and 8.2.3), I have also given the error in the slope, as an indication of how neat the fit is. See the `gnuplot` manual for more details.

I used the graph generator functions of `LEDA 4.1` to generate graphs. The algorithm `LEDA` uses can be roughly described as:

> Create n vertices.
> **while** $|E| < m$ **do**
>   **repeat**
>   Pick two vertices u and v
>   **until** $uv \notin E$
>   Add uv to E
> **end while**

8.2 Basic Time Complexity

The first tests I wished to do were basic time experiments, to verify the theoretical time bounds. To do this, I generated a number of random graphs, with a given number of vertices and edges, then ran the algorithms on the graphs and record the running times.

The graph is generated as follows: I create n vertices. I then randomly select two vertices. If the two vertices are different and nonadjacent, I create an edge between them. This process is repeated until m edges have been added.

Now, graph generation is controlled by two parameters, but these parameters are not independent. An undirected graph with n vertices cannot have less than $n-1$ edges and still be connected, and it cannot have more than $n(n-1)$ edges and still be simple.

As such, there are four kinds of experiment I might do:

1. The number of vertices is held constant and the number of edges is

varied from n to $n(n-1)$.

2. The number of edges is held constant, and the number of vertices is varied from $\lceil (1+\sqrt{1+8m})/2 \rceil$ to m. (the reader can check that a complete graph with m edges will have $(1+\sqrt{1+8m})$ nodes)

3. The edge density is held constant — ie: for a fixed $d \in (0,1)$, we vary n and set $m := d\frac{n(n-1)}{2}$. In this case, the number of edges is a quadratic function of n.

4. The number of edges is a linear function of n.

Time results follow. These results were all pretty "good", in that there was little spread. We are looking at mean values, but the trends still would have been clear had we simply plotted all the data.

8.2.1 Theoretical Complexities

The theoretical complexities of using the algorithms to find a single vertex ordering are as follows:

Algorithm	Time complexity	Reference
Fill-In	$O(n+m')$	[53]
LEXP	$O(n+m)$	[50]
LEXM	$O(nm)$	[50]
MCS	$O(n+m)$	[53]
MCSM	$O(nm)$	[7]

Every run of Algorithm 4.1 is followed by a run of Algorithm 3.1. Furthermore, we need to run the algorithm once with each vertex in last place. As such, this gives an overall complexity of $O(n^2+nm')$ in the case of LEXP and MCS and $O(n^2m)$ for LEXM and MCSM.

I am unsure as to the theoretical complexity of MSVS. But we will see what the experiments show.

8.2.2 Constant edge density

First, I consider graphs with a constant edge density. That is, for density $d \in [0, 1]$, $m = d\frac{n(n-1)}{2}$.

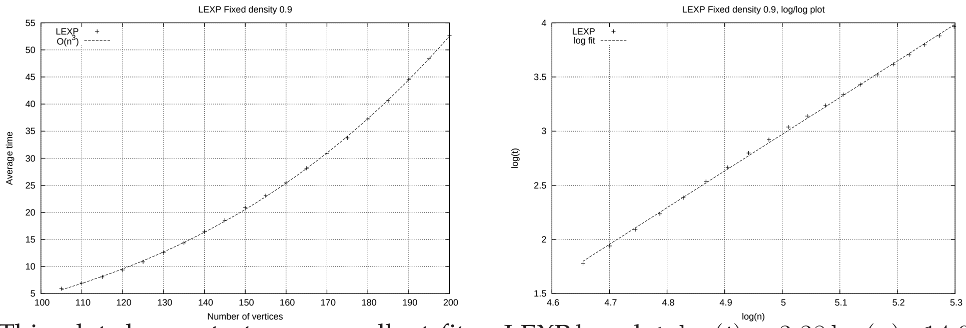
If $m' \geq d\frac{n(n-1)}{2}$ then $O(n^2 + nm')$ reduces to $O(n^3)$ and $O(n^2m)$ reduces to $O(n^4)$.

In what follows, I show the actual data gathered, plus a log-log plot.

(as a reminder, if $\log(y) = a\log(x) + b$ then $y = e^b x^a$ — that is, if the log plot turns out to be linear, the slope will give us the exponent and the "height" of the line will give us the multiplicative constant. Note, of course, that we do not particularly expect the exponent to be an integer: if there are lower-order terms, it will not be)

I present results for two different densities: $d = 0.9$ and $d = 0.5$. I did consider other densities; the results were similar.

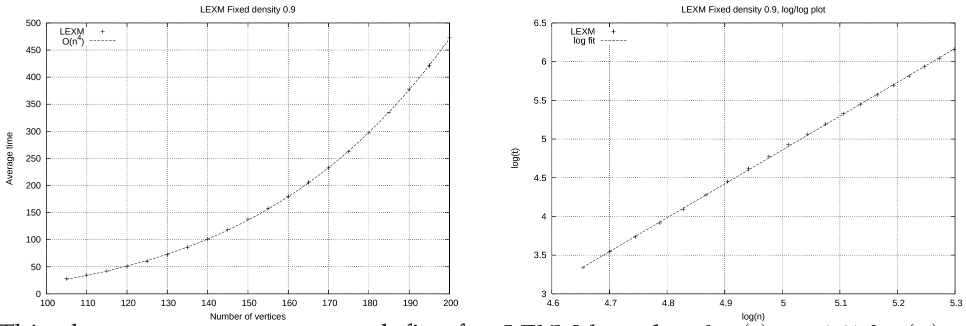
I also compare the running times of individual algorithms on the two different graph densities for, to see what effect density has. The expected effect here is that the triangulation algorithms will be faster on the lower-density graphs, as the theoretical complexities all depend on m. We will see more on the effects of varying density later.



This plot demonstrates an excellent fit between LEXP data and the theoretical complexity of $O(n^2 + nm)$ = $O(n^3)$.

LEXP log plot: $\log(t) = 3.38\log(n) - 14.0$ with error 0.6%.

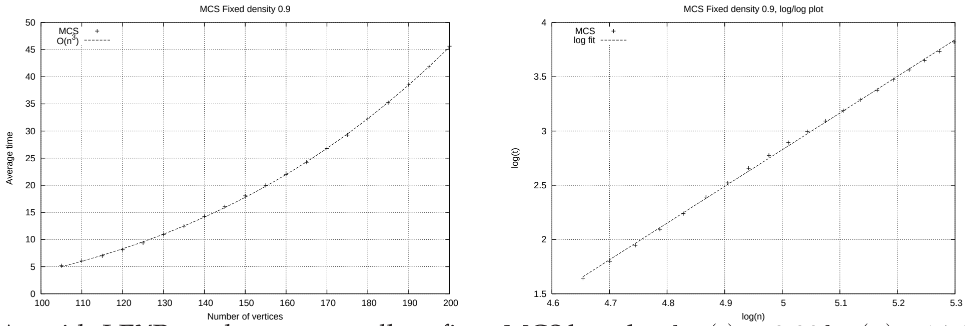
Figure 8.1: LEXP: Fixed edge density, density = 0.9



This demonstrates a very good fit of LEXM data with its theoretical complexity of $O(n^2m)$ = $O(n^4)$.

LEXM log plot: $\log(t) = 4.37\log(n) - 17.0$ with error 0.3%.

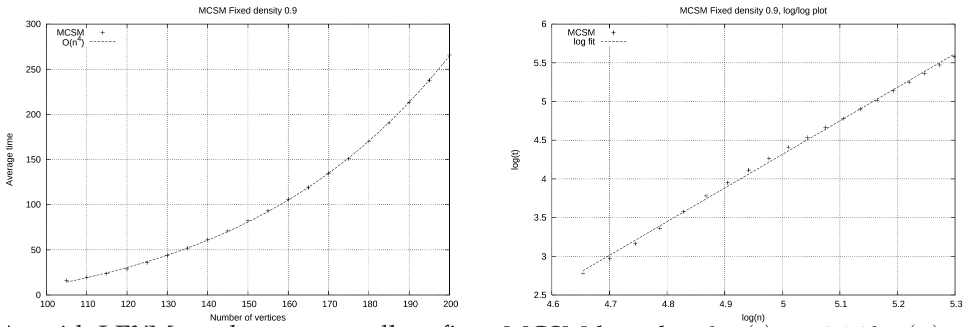
Figure 8.2: LEXM: Fixed edge density, density = 0.9



As with LEXP, we have an excellent fit between the experimental data and the theoretical $O(n^3)$ behaviour.

MCS log plot: $\log(t) = 3.38\log(n) - 14.1$ with error 0.6%.

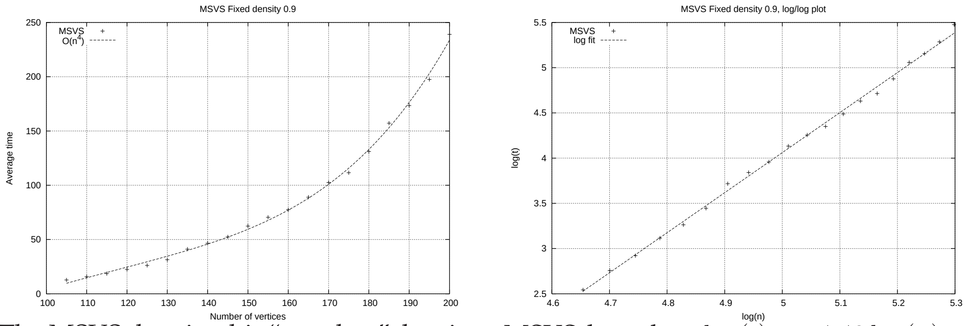
Figure 8.3: MCS: Fixed edge density, density = 0.9



As with LEXM, we have an excellent fit between the experimental data and the theoretical $O(n^4)$ behaviour. It is worth noting that `gnuplot` was unable to fit successfully an $O(n^3)$ curve to this data, or to the LEXM data.

MCSM log plot: $\log(t) = 4.34\log(n) - 17.4$ with error 0.9%.

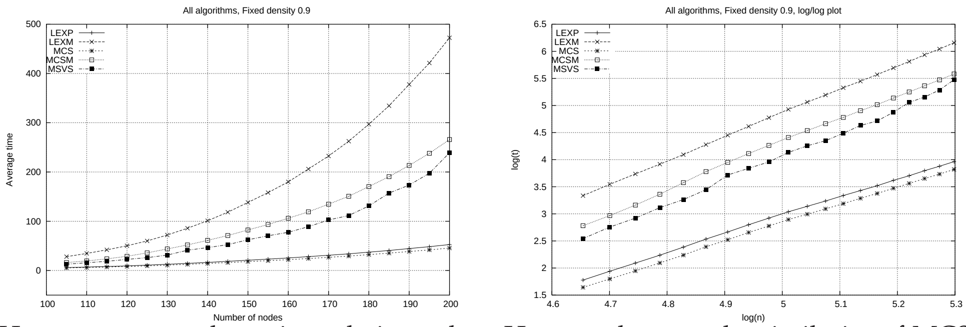
Figure 8.4: MCSM: Fixed edge density, density = 0.9



The MSVS data is a bit "rougher", but it fits a $O(n^4)$ curve fairly well. The choice of the curve exponent was based on the log plot, plus the fact that a cubic curve did not fit particularly well.

MSVS log plot: $\log(t) = 4.43\log(n) - 18.1$ with error 1.1%.

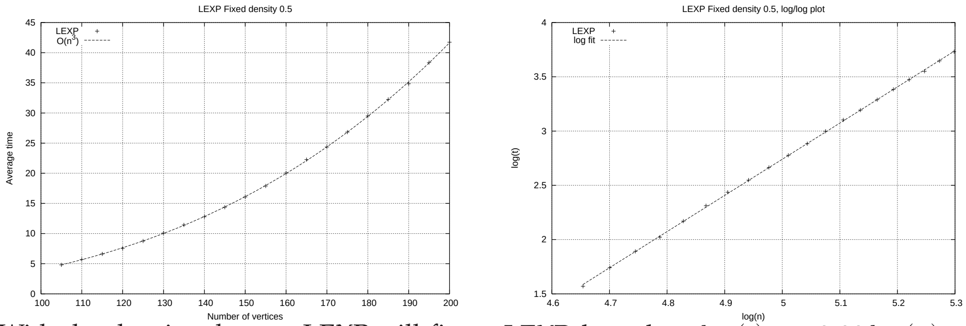
Figure 8.5: MSVS: Fixed edge density, density = 0.9



Here, we see the triangulation algorithms' relative performances, with LEXM $>$ MCSM $>$ MSVS $>$ LEXP $>$ MCS. We also see that MSVS, in this case of quite dense graphs, is performing better than both the minimal triangulation heuristics.

Here can be seen the similarity of MCS and LEXP, with their almost-identical log slopes and similar log intercepts (the net effect being that LEXP has a larger multiplicative constant than MCS). The other three algorithms are likewise similar, although the gaps are larger. Using the guide lines, we can also see that LEXP and MCS diverge slightly from LEXM and MCSM.

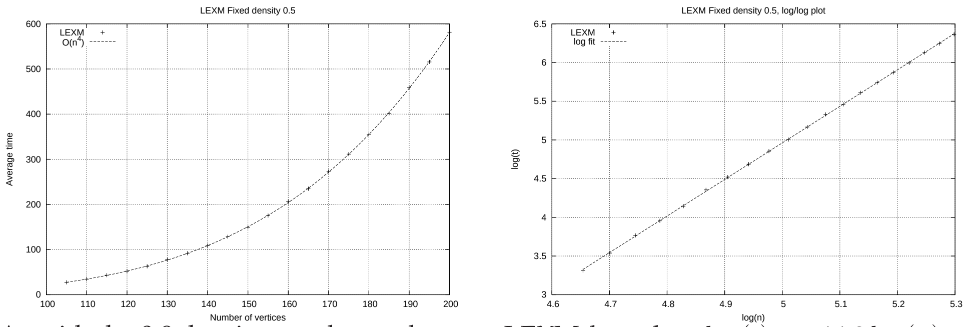
Figure 8.6: All algorithms: Fixed edge density, density = 0.9



With the density change, LEXP still fits a cubic curve.

LEXP log plot: $\log(t) = 3.33 \log(n) - 13.9$ with 0.3% error. This slope is in fact slightly less than the higher-density graphs, perhaps due to the lower average vertex degree (hence the lower number of neighbours that must be checked when each vertex is processed).

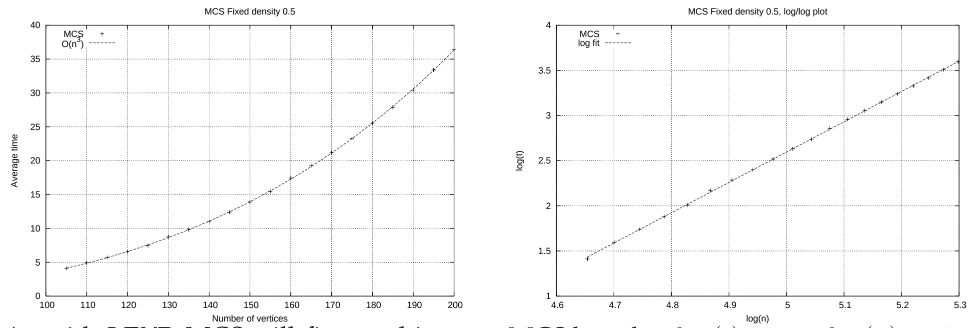
Figure 8.7: LEXP: Fixed edge density, density = 0.5



As with the 0.9 density graphs, we have a good fit for the predicted $O(n^4)$ behaviour (also, I note that `gnuplot` was unable to produce a good fit for a cubic curve).

LEXM log plot: $\log(t) = 4.72 \log(n) - 18.7$ with 0.2% error. This slope is somewhat steeper than the 0.9 density graphs (4.72 versus 4.37). A possible explanation for is is that, as the distance between nodes increases, LEXM has longer paths to search through in its update step.

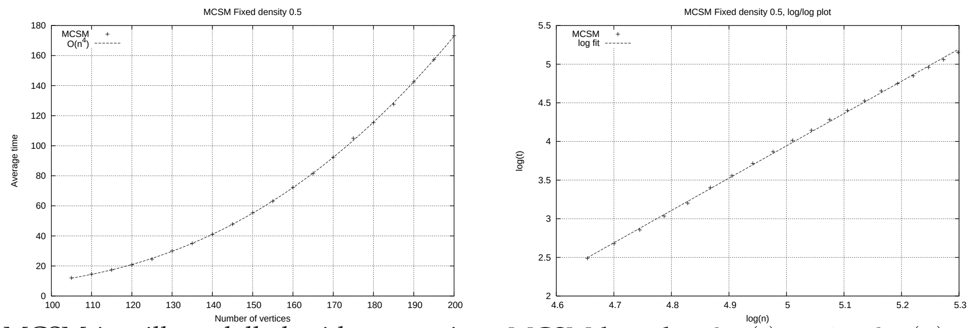
Figure 8.8: LEXM: Fixed edge density, density = 0.5



As with LEXP, MCS still fits a cubic — she has neither gained nor lost weight..

MCS log plot: $\log(t) = 3.36 \log(n) - 14.2$ with error 0.3%. As with LEXP, this slope is very slightly decreased from the density 0.9 graph — so slightly, however, that one is reluctant to read anything into it.

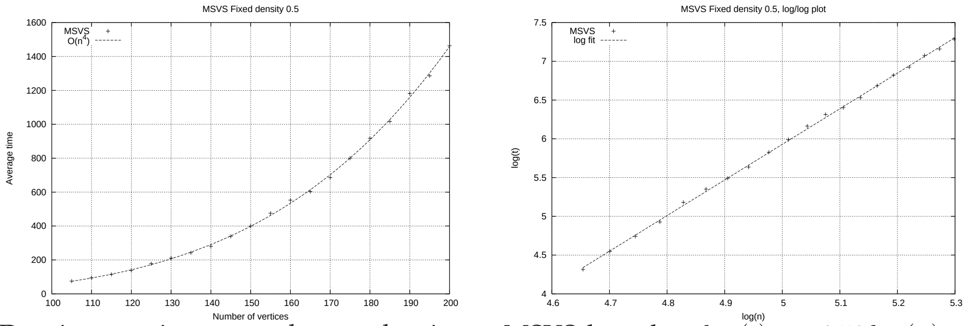
Figure 8.9: MCS: Fixed edge density, density = 0.5



MCSM is still modelled with a quartic but not a cubic curve.

MCSM log plot: $\log(t) = 4.18 \log(n) - 17.0$ with error 0.6%. Unlike LEXM, this slope is in fact fractionally decreased.

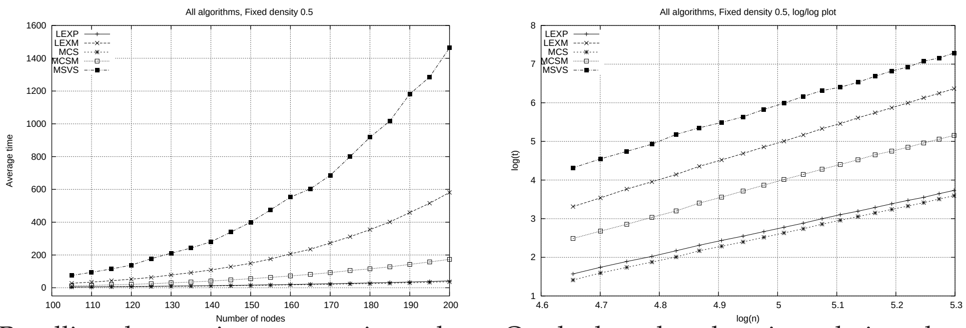
Figure 8.10: MCSM: Fixed edge density, density = 0.5



Despite moving to a lower density, MSVS is still exhibiting quartic performance.

MSVS log plot: $\log(t) = 4.59\log(n) - 17.0$. The slope has increased from the higher density graphs and, furthermore, we note that the intercept (and hence the multiplicative constant of the non-logarithmic curve) has also increased. Later plots will, in fact, show that MSVS behaviour is very dependent on graph edge density.

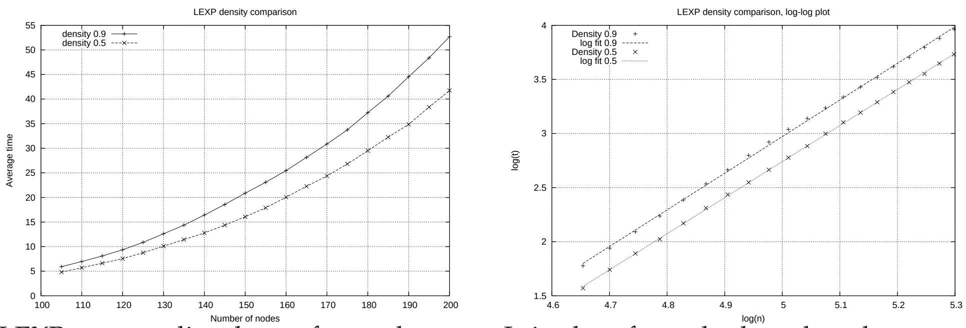
Figure 8.11: MSVS: Fixed edge density, density = 0.5



Recalling the previous comparison, the difference is very plain: MSVS, which (on graphs of the same number of vertices) had performed better than both LEXM and MCSM, is now very much the slowest algorithm of the five.

On the log plot, the triangulation algorithms are all roughly as they were with density 0.9. The MSVS line, however, has is significantly higher.

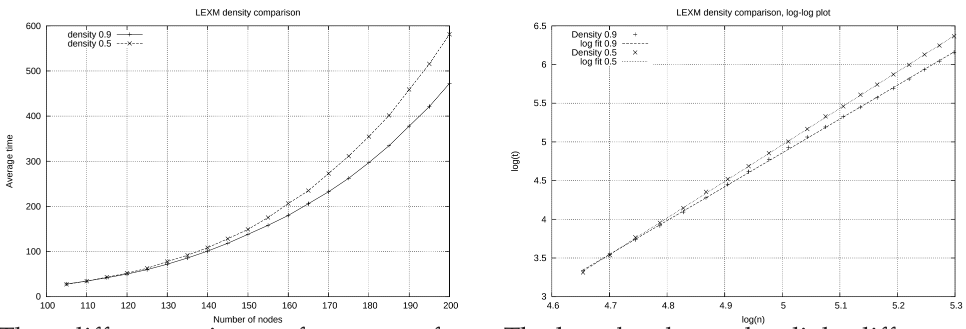
Figure 8.12: All algorithms: Fixed edge density, density = 0.5



LEXP, as predicted, performs better with lower density. The update step of LEXP is concerned only with neighbours, as is the triangulation algorithm. Therefore, this is unsurprising.

It is clear from the log plots, however, that the performance difference is a matter primarily (or only) of a constant factor, and there is no change in the growth rate.

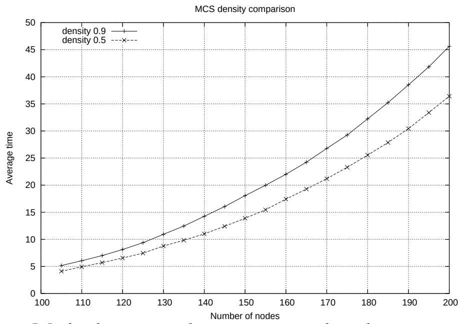
Figure 8.13: LEXP: Fixed edge density, density comparison

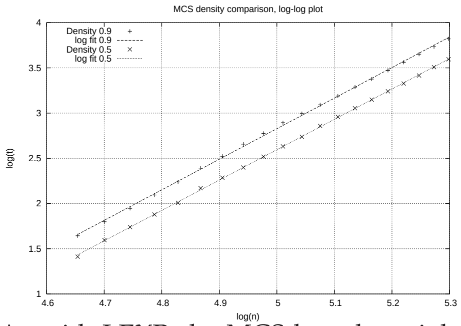


The difference in performance for LEXM as the density changes is clearly noticeable. Furthermore, LEXM performs worse, not better, on the lower density graphs. I speculate that this may be related to the search step of its label update phase: decreasing density, the paths to be explored become longer.

The log plot shows the slight difference in slope that accounts for the performance difference. Although it is worth noting that the greater slope line will (perforce) have a smaller intersection with the vertical axis, and hence a lower multiplicative constant.

Figure 8.14: LEXM: Fixed edge density, density comparison

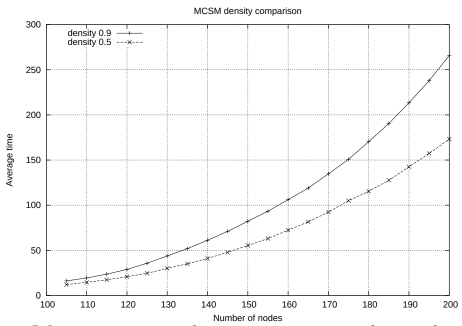


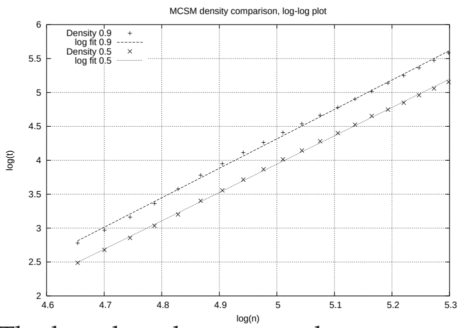


MCS behaves almost exactly the same as LEXP, save for a slightly lower running time. I will not repeat the discussion.

As with LEXP, the MCS log plot might be a set of railway tracks. A reader would need extremely sharp eyes indeed to be able to spot the difference of 0.02 between the slopes of these two lines...

Figure 8.15: MCS: Fixed edge density, density comparison

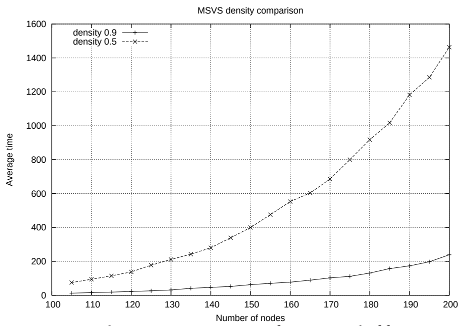




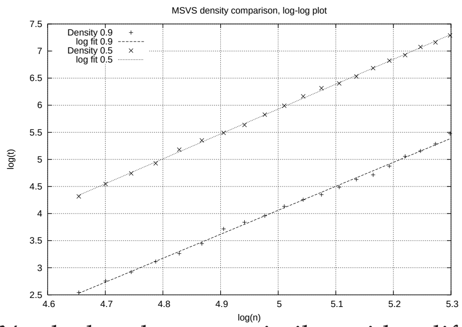
MCSM, somewhat surprisingly, also performs better on the lower density graphs. I had expected that, like LEXM, the lower density would slow it. However, this is evidently not so.

The log plots show, not only a constant factor difference, but also a very slight divergence — although I am reluctant to read too much into a slope difference of 0.16.

Figure 8.16: MCSM: Fixed edge density, density comparison



Here is the most significant difference by far. MSVS clearly performs significantly worse on graphs of "medium" density, compared with graphs of high density.



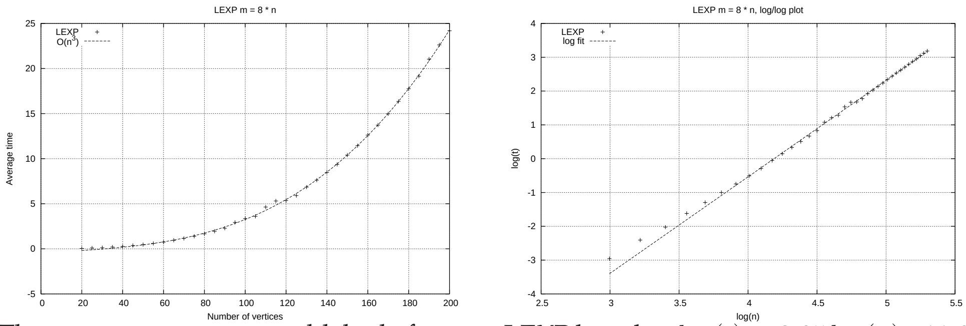
Yet the log slopes are similar, with a difference of only 0.16 — not really significant.

Figure 8.17: MSVS: Fixed edge density, density comparison

8.2.3 Sparse graphs

If we consider graphs in which $|E|$ is a fixed multiple of $|V|$, then we should expect quadratic time performance from LEXP and MCS, as $O(n^2 + nm')$ will reduce to approximately $O(n^2)$ (although there may be a difference as we may have $m' > m$).

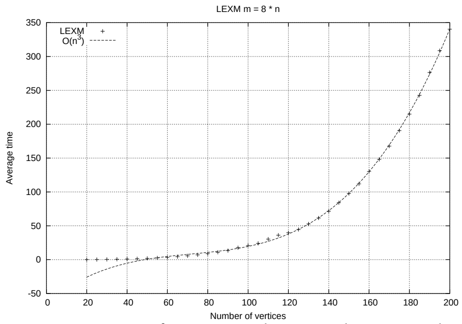
Similarly, we should expect $O(n^3)$ time from LEXM and MCSM.

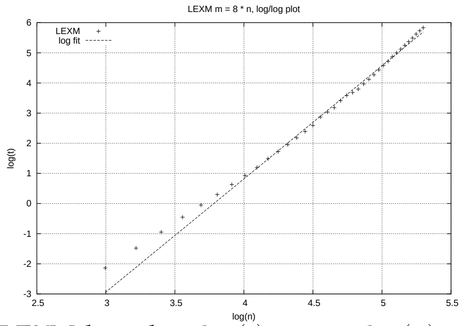


Theory suggests we would look for a quadratic curve here. However, the log slope is close to three, and a quadratic fit was extremely unconvincing. As such, I have fit a cubic curve here.

LEXP log plot: $\log(t) = 2.85\log(n) - 11.9$ with error 0.5%. Compare a slope of 3.33 for the constant density 0.5 data.

Figure 8.18: LEXP: m linear in n, $m = 8n$

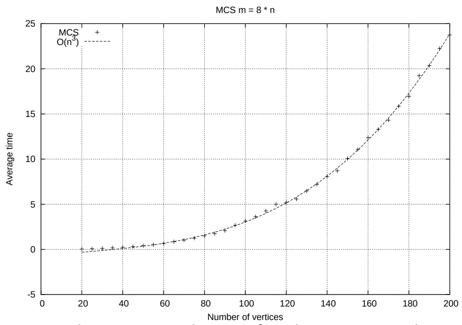


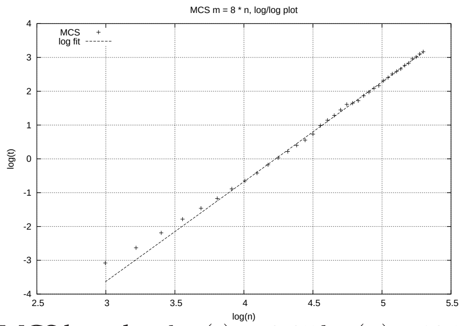


LEXM is performing better here, relative to the fixed density plots — the log/log plot demonstrates that. A cubic curve clearly does not, however, fit the data perfectly.

LEXM log plot: $\log(t) = 3.75\log(n) - 14.2$ with error 0.9%. Compare this slope with 4.72 for the constant density 0.5 data.

Figure 8.19: LEXM: m linear in n, $m = 8n$

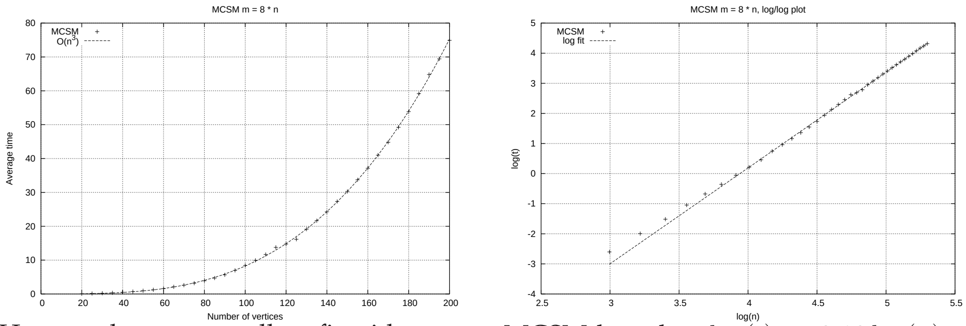




As with LEXP, I have fit this as a cubic.

MCS log plot: $\log(t) = 2.95\log(n) - 12.5$ with error 0.5%. Compare constant density 0.5 slope of 3.36.

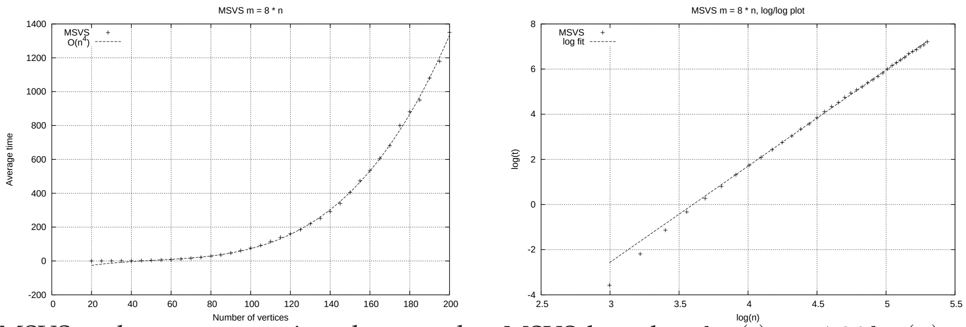
Figure 8.20: MCS: m linear in n, $m = 8n$



Here, we have an excellent fit with a cubic curve. Of all the triangulation algorithms, MCSM agrees most convincingly with the theoretical prediction, in this case $O(n^3)$.

MCSM log plot: $\log(t) = 3.18 \log(n) - 12.5$ with error 0.3%. Compare with a slope of 4.18 for constant density 0.5 — in other words, MCSM has achieved the theoretical prediction of a speedup by a factor of $1/n$.

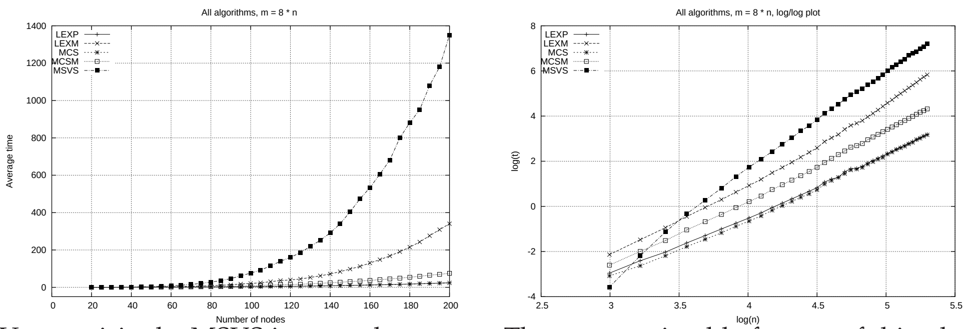
Figure 8.21: MCSM: m linear in n, $m = 8n$



MSVS makes no concessions here, and retains its $O(n^4)$ behaviour. However, because the edge density decreases to the tune of $\frac{16}{n-1}$, which rapidly becomes quite low, it avoids the bad behaviour seen with 30-40% density.

MSVS log plot: $\log(t) = 4.26 \log(n) - 15.3$ with error 0.3%. Compare slope of 4.59 for constant density 0.5 graphs.

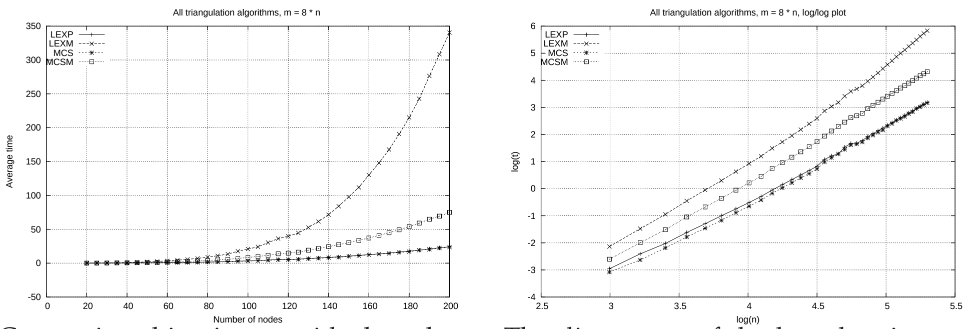
Figure 8.22: MSVS: m linear in n, $m = 8n$



Unsurprisingly, MSVS is worst here, as it is not predicted to improve, whereas the triangulation algorithms are.

The most noticeable feature of this plot is that MSVS initially performs quite well. This is linked to the fact that, for low n, the edge density is actually quite high (85% for $n = 20$), and MSVS performs well on graphs of high density (see Figure 8.38).

Figure 8.23: All algorithms: m linear in n, $m = 8n$



Comparing this picture with the other one, we can see that LEXP, MCS and MCSM have all retained their relative positions, more or less, but LEXM has worsened in comparative performance.

The divergence of the log plots is easy to spot: LEXP/MCS are diverging from MCSM, and so is LEXM.

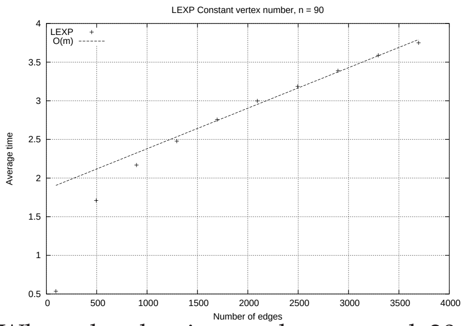
Figure 8.24: All triangulation algorithms: m linear in n, $m = 8n$

8.2.4 Variable edge density

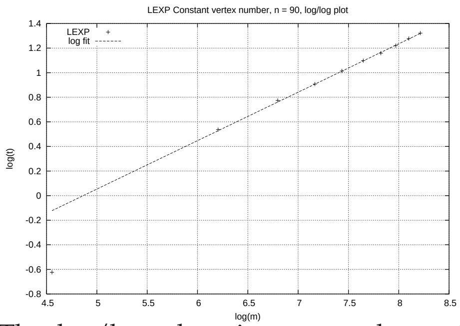
If we hold the number of vertices fixed and vary the number of edges, we might expect linear performance in all the triangulation algorithms (as all theoretical complexities are linear in m). However, this is also equivalent to plotting versus varying edge density and, as such, surprises may be in store…

Note 1: These plots extend as "far right" as is possible, for once the edge density reaches 1, the graph is complete, and thus (a) it is triangulated, and (b) the treewidth is known anyway…

Note 2: In fitting curves to the data, I have discounted the first three points, where the density is very low.

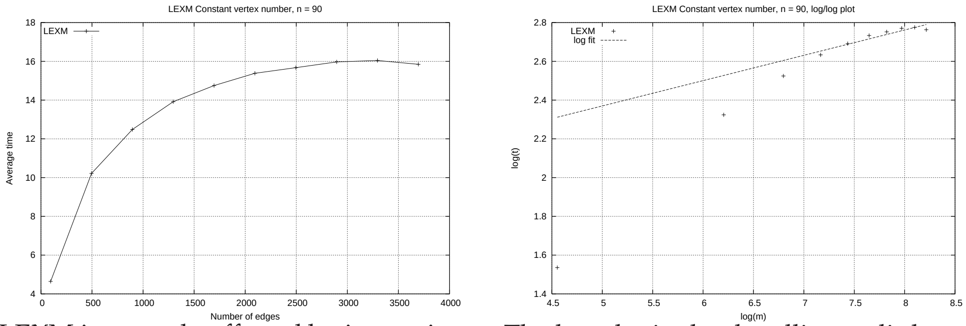


When the density reaches around 30-40%, LEXP exhibits approximately linear behaviour.



The log/log plot gives us a slope of 0.39, with a 0.8% error (discounting the first three points, although only the first point is drastically out of line). This suggests plotting an $O(\sqrt{n})$ curve.

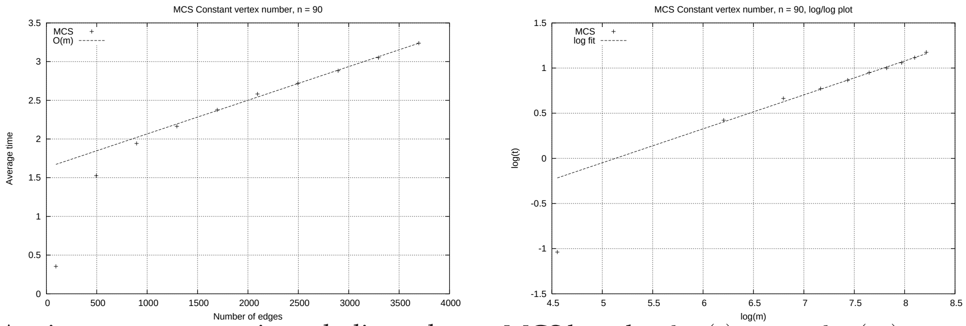
Figure 8.25: LEXP: n fixed, m variable, $n = 90$



LEXM is strongly affected by increasing $|E|$ when the density is low. However, as the density rises the time is less, even negative. It is not clear to me what function models this curve — especially in light of Figure 8.33.

The log plot is clearly telling us little.

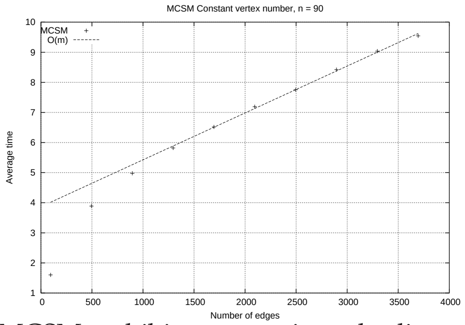
Figure 8.26: LEXM: n fixed, m variable, $n = 90$



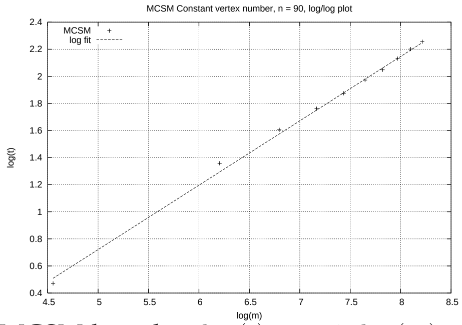
Again, we see approximately linear behaviour, once the graphs have reached a density of around 30%.

MCS log plot: $\log(t) = 0.38\log(m) - 1.93$ with error 2.3%.

Figure 8.27: MCS: n fixed, m variable, $n = 90$

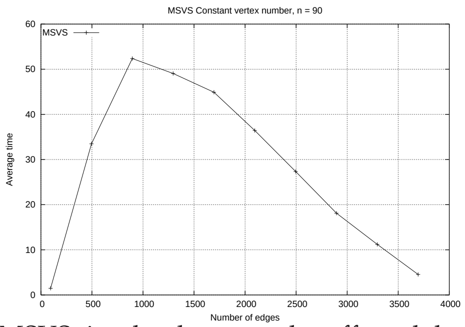


MCSM exhibits approximately linear behaviour as density increases beyond 30%.

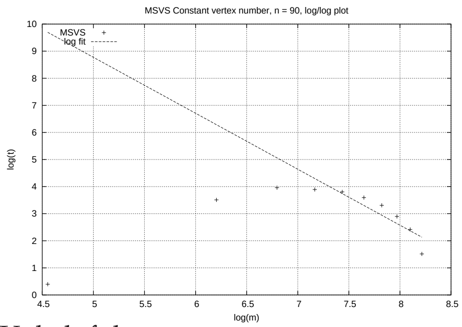


MCSM log plot: $\log(t) = 0.48\log(m) - 1.66$ with error 1.78%.

Figure 8.28: MCSM: n fixed, m variable, $n = 90$

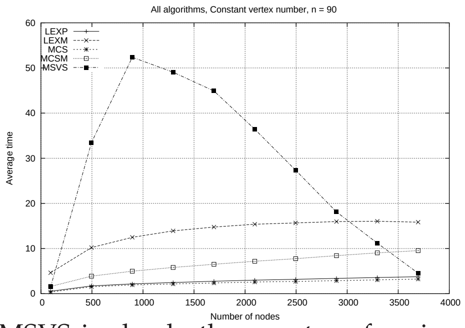


MSVS is clearly strongly affected by edge density. It peaks at around 25% density, and then declines rapidly. This could be modelled with, say, an $O(-n^4)$ curve, but I don't believe that would be meaningful.

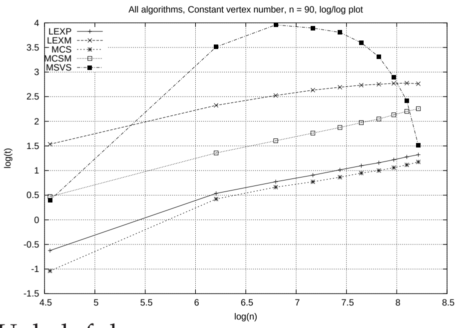


Unhelpful.

Figure 8.29: MSVS: n fixed, m variable, $n = 90$

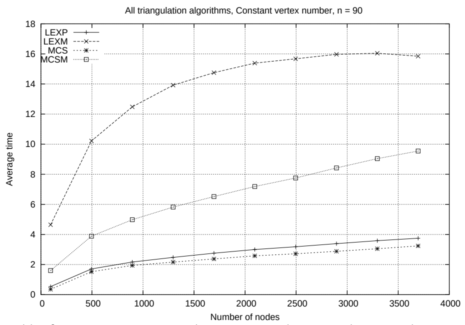


MSVS is clearly the worst performing, apart from on dense graphs.

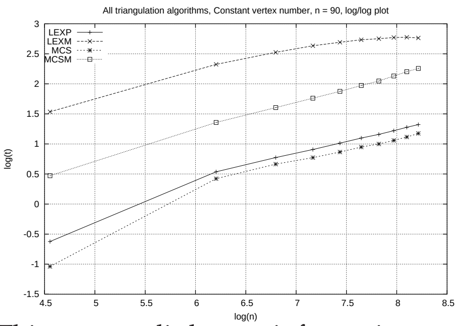


Unhelpful.

Figure 8.30: All algorithms: n fixed, m variable, $n = 90$

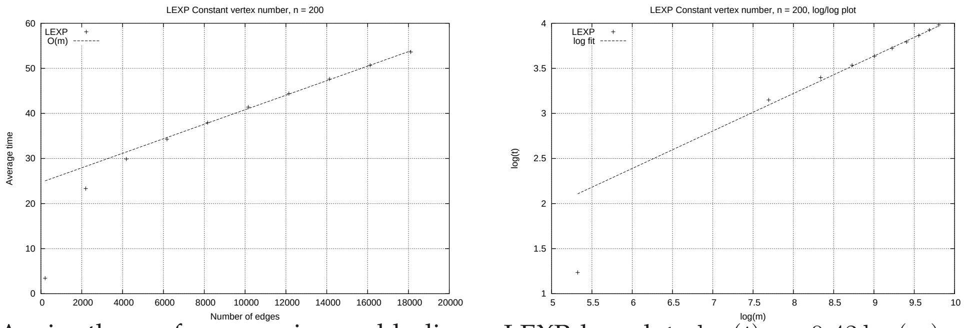


All four triangulation algorithms have the same basic pattern, increasing steeply until around 15% density, before mostly levelling off.



This conveys little new information.

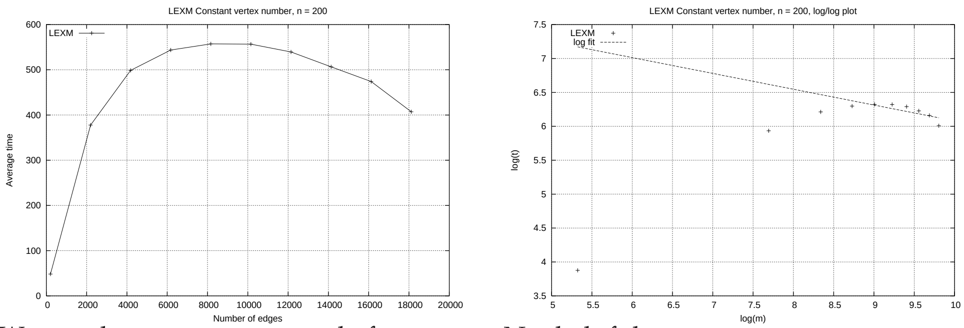
Figure 8.31: All triangulation algorithms: n fixed, m variable, $n = 90$



Again, the performance is roughly linear in $|E|$ beyond 30-40%.

LEXP log plot: $\log(t) = 0.42\log(m) - 0.10$ with 2.0% error. This is a similar slope to the previous case.

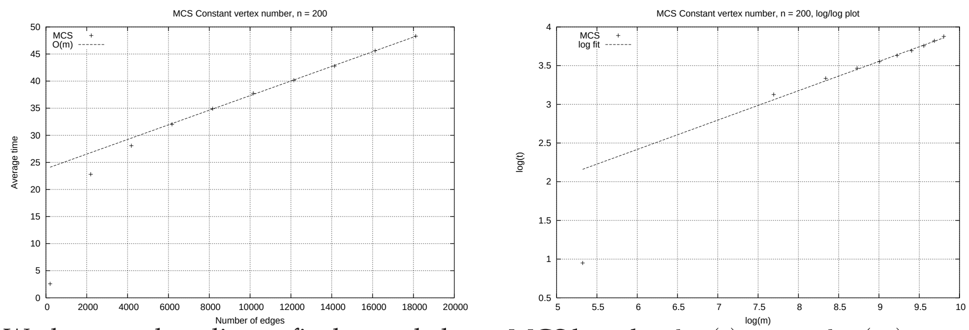
Figure 8.32: LEXP: n fixed, m variable, $n = 90$



We see the same pattern as before, except accentuated. It is clear that LEXM performs better as the density increases (and, hence, as the average distance between nodes decreases).

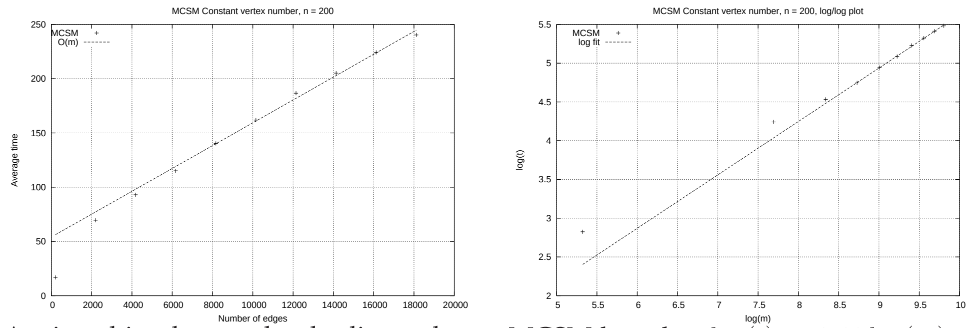
Not helpful.

Figure 8.33: LEXM: n fixed, m variable, $n = 90$



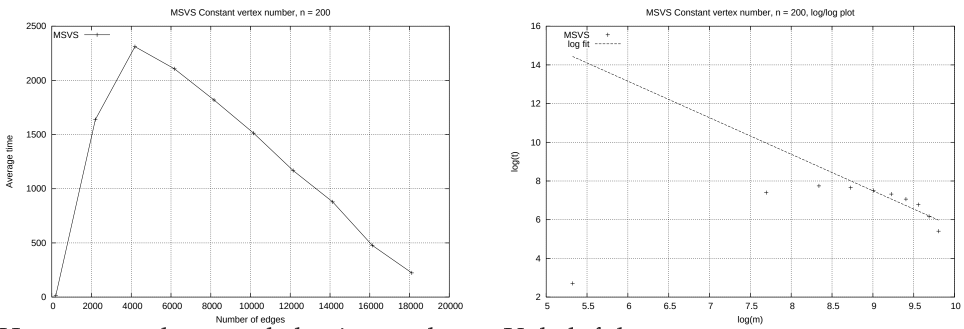
We have a close linear fit, beyond the first few data.	MCS log plot: $\log(t) = 0.38\log(m) + 0.14$ with 3.1% error.

Figure 8.34: MCS: n fixed, m variable, $n = 90$



Again, this shows closely linear behaviour — perhaps the most so of any.	MCSM log plot: $\log(t) = 0.69\log(m) - 1.27$ with 1.0% error.

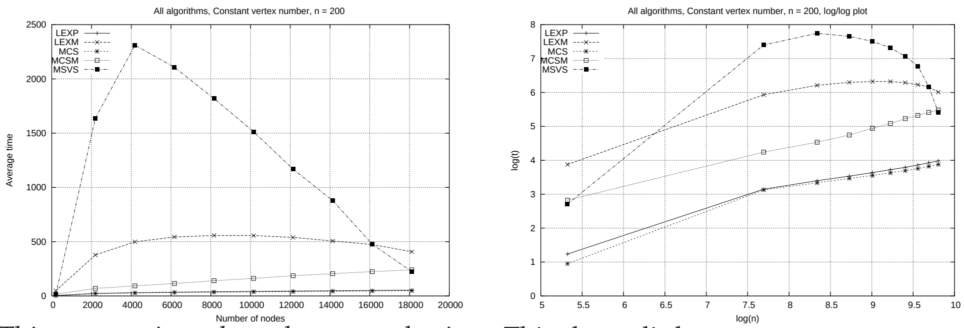
Figure 8.35: MCSM: n fixed, m variable, $n = 90$



Here we see the same behaviour as before, peaking at around 20-25%.

Unhelpful.

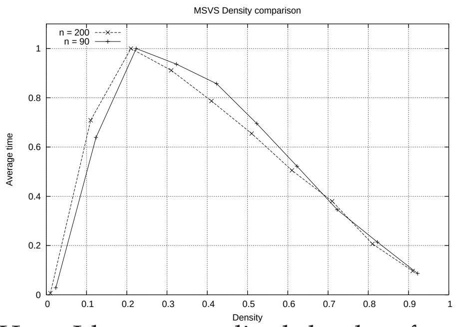
Figure 8.36: MSVS: n fixed, m variable, $n = 90$



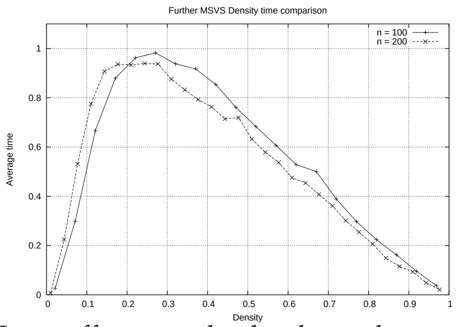
This comparison has the same basic shape as the previous comparison of all algorithms. However, the difference between MSVS and the others is greater (in relative and in absolute terms).

This shows little new.

Figure 8.37: All algorithms: n fixed, m variable, $n = 90$



Here, I have normalised the data from Figures 8.29 and 8.36. The x scale now represents edge density, and the y scale represents nothing in particular. However, this does allow to compare the behaviour of the algorithm on the two sizes in a way which makes sense — and in a way which makes the similarity of shape obvious.



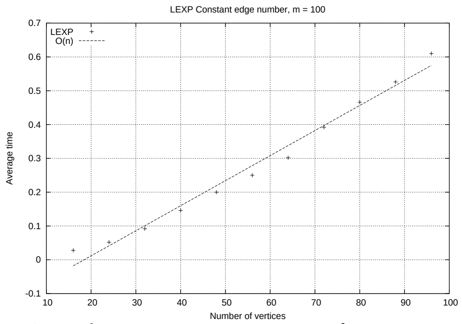
In an effort to make the shape clearer, I reconducted the experiment with more steps. It seems that, for the larger graphs, the time increases more steeply initially. However, both curves start to decrease at the same point — at an edge density of around 0.27.

Figure 8.38: Further MSVS edge density experiments.

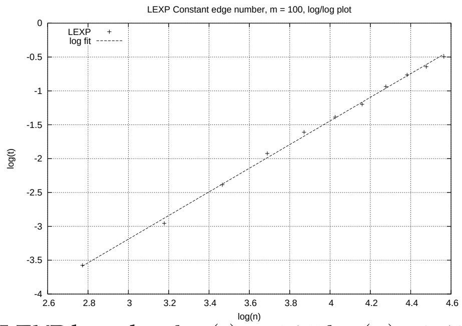
8.2.5 Varying n for constant m

The last experiment to do is to hold the number of vertices constant and vary the number of edges. The theoretical predictions here are for $O(n)$ for LEXP and MCS and $O(n^2)$ for LEXM and MCSM.

These plots start at complete graphs and stop when $n = m$ (ie: the number of vertices that a tree with m edges would have — although there is no guarantee that the graphs are, in fact, trees, and it is more likely that they consist of small connected components and many isolated vertices).

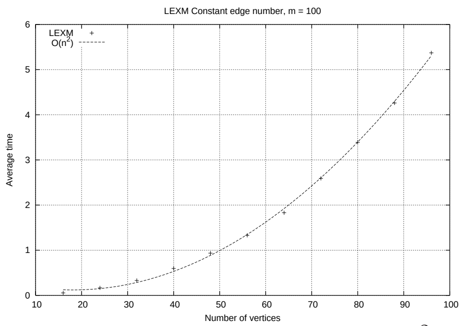


LEXP does not give a perfect curve, however the general trend is clear.

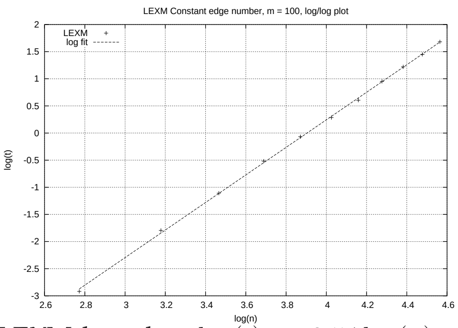


LEXP log plot: $\log(t) = 1.75\log(n) - 8.42$ with 1.4% error.

Figure 8.39: LEXP: m fixed, n variable, $m = 100$

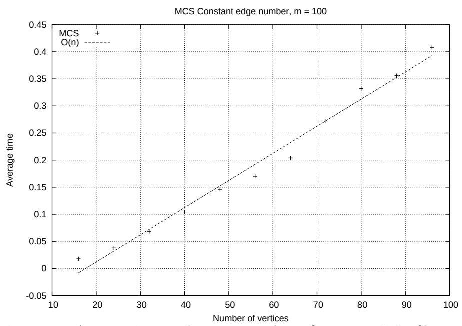


Theoretical complexity of $O(n^2m)$ should yield $O(n^2)$ when m is held constant. And, indeed, this is what we achieve.

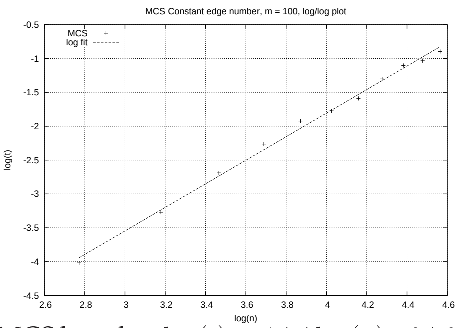


LEXM log plot: $\log(t) = 2.54\log(n) - 9.91$ with error 0.7%.

Figure 8.40: LEXM: m fixed, n variable, $m = 100$

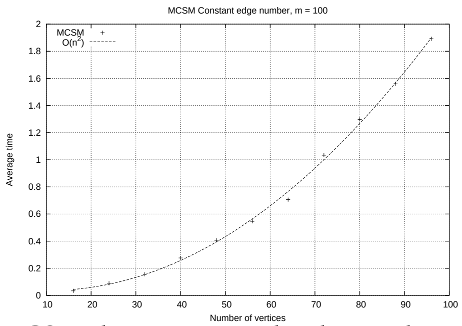


As with LEXP, the results for MCS fluctuate (perhaps due to the low times involved).

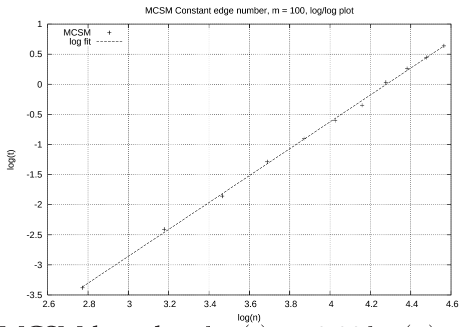


MCS log plot: $\log(t) = 1.74\log(n) - 8.76$, with error 2.2%.

Figure 8.41: MCS: m fixed, n variable, $m = 100$

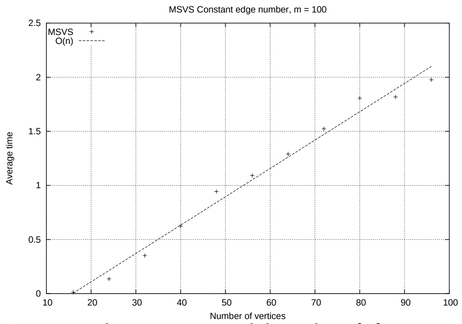


MCSM demonstrates closely quadratic behaviour.

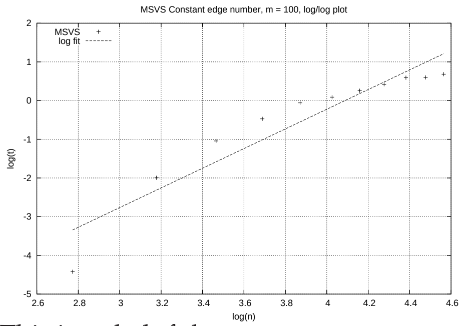


MCSM log plot: $\log(t) = 2.23 \log(n) - 9.56$ with error 1.0%.

Figure 8.42: MCSM: m fixed, n variable, $m = 100$

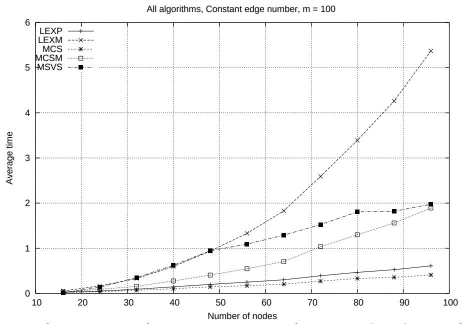


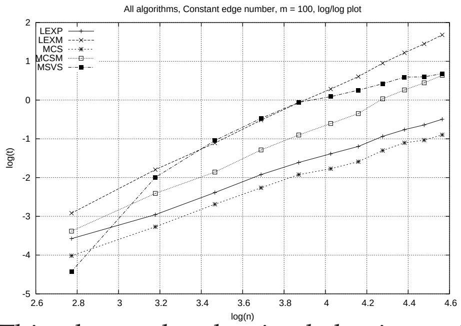
We can do a reasonable job of fitting a linear function to this data. However, the log plot doesn't convey any useful information, so a conclusion "MSVS == Linear" must be held in doubt.



This is unhelpful.

Figure 8.43: MSVS: m fixed, n variable, $m = 100$

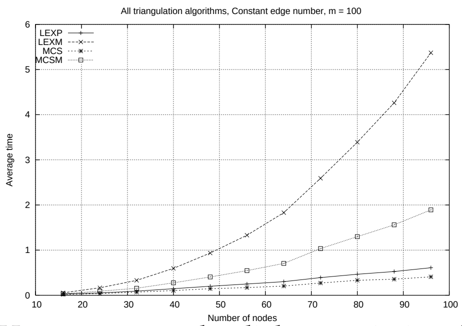


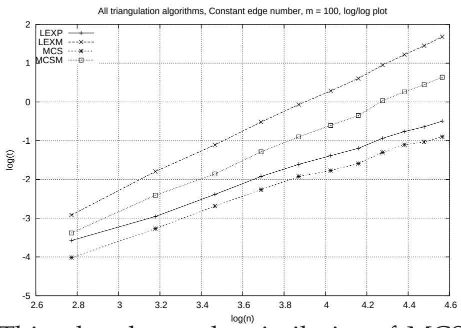


Looking at this, we see that MSVS and LEXM are initially equally poor, but then MSVS tails off, whereas LEXM continues to increase in time at the same rate. In accounting for this, we should recall that MSVS takes less time for graphs of very low density. In this graph, edge density is decreasing as the square of the number of vertices, and so by $n = 50$ already the edge density is below 10%.

This shows the density behaviour of MSVS in a slightly different light: Initially, the graphs are very edge-dense, with the density decreasing as n increases. Thus MSVS' time initially increases sharply, relative to the other algorithms, before falling off. Other than this, there is little to be said here.

Figure 8.44: All algorithms: m fixed, n variable, $m = 100$

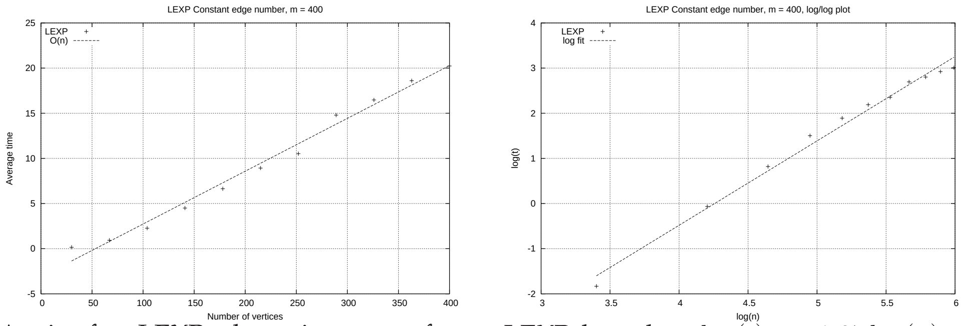




Here we can see the slight superiority of MCS over LEXP, and the definite superiority (time-wise) of MCSM over LEXM.

This plot shows the similarity of MCS and LEXP. It also shows the slightly different growth rates, as the four log curves gradually diverge with increasing n.

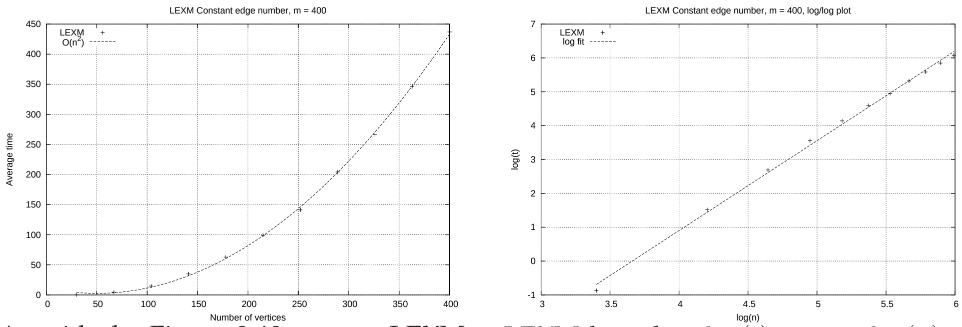
Figure 8.45: All triangulation algorithms: m fixed, n variable, $m = 100$



Again for LEXP, there is no perfect curve. However, a straight line seems a reasonable approximation — there is no clear upwards curve.

LEXP log plot: $\log(t) = 1.87\log(n) - 7.96$, but the error has climbed to 3.4%.

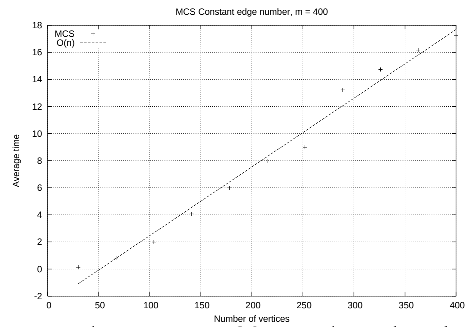
Figure 8.46: LEXP: m fixed, n variable, $m = 400$



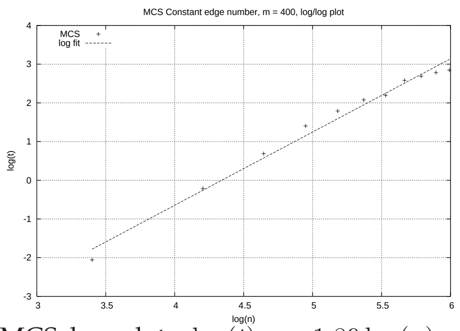
As with the Figure 8.40, we see LEXM closely fitting a quadratic curve.

LEXM log plot: $\log(t) = 2.66\log(n) - 9.72$ with error 1.6%. The appearance of the graph suggests that, if the first point were ignored, the log slope would be lesser (closer to the theoretical value of 2).

Figure 8.47: LEXM: m fixed, n variable, $m = 400$

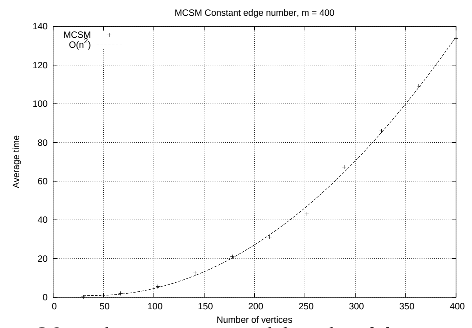


As with LEXP, MCS can best be described as "approximately" linear.

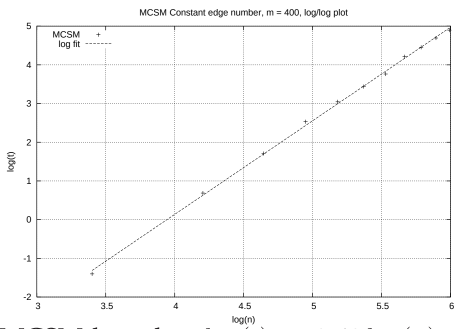


MCS log plot: $\log(t) = 1.89\log(n) - 8.22$ but it is clearly a ragged line, and `gnuplot` gives an error of 4.0% for the slope.

Figure 8.48: MCS: m fixed, n variable, $m = 400$

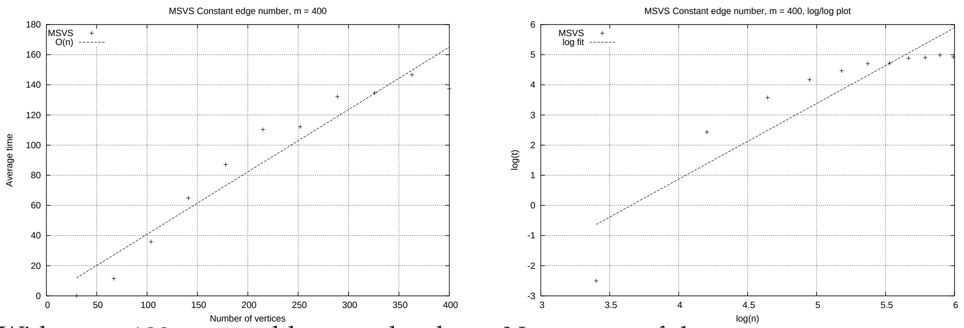


MCSM does a reasonable job of fitting a quadratic curve.



MCSM log plot: $\log(t) = 2.42\log(n) - 9.54$ with error 1.0%.

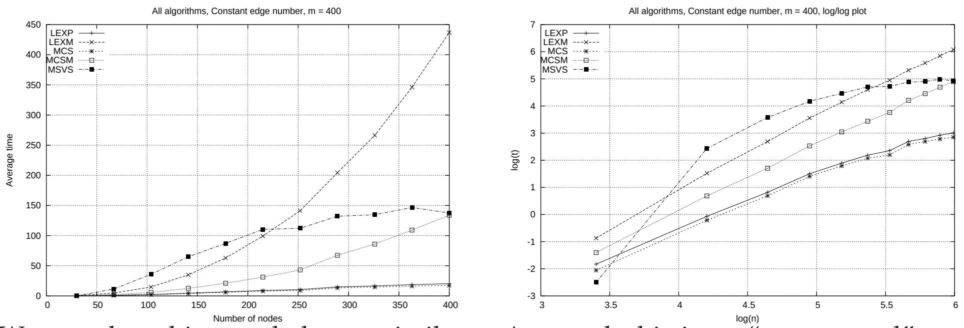
Figure 8.49: MCSM: m fixed, n variable, $m = 400$



With $m = 100$, we could pretend to be looking at $O(n)$ behaviour. However, this is clearly no longer reasonable. Although we might wave our hands and say that the growth is "approximately" linear initially, tailing off from around $n = 250$. This tailing-off could be explained by observing that, by $n = 250$, the edge density has dropped to 1%, and the graphs will be already very tree-like.

Not very useful.

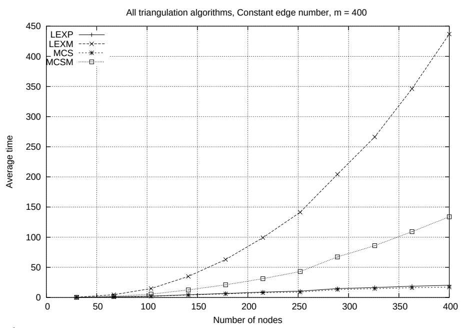
Figure 8.50: MSVS: m fixed, n variable, $m = 400$



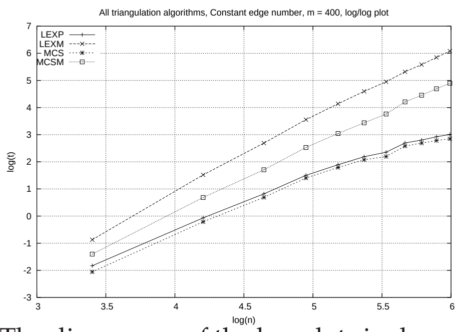
We see that this graph has a similar layout to the previous comparison of all five algorithms. The differences between triangulation algorithms are accentuated, and MSVS takes longer to "cross below" LEXM.

As noted, this is an "accentuated" version of Figure 8.44. The divergence of the triangulation algorithm log plots is clearer here (except for LEXP and MCS, which shadow each other closely).

Figure 8.51: All algorithms: m fixed, n variable, $m = 400$



skip



The divergence of the log plots is clearer here (except LEXP and MCS, which shadow each other quite closely).

Figure 8.52: All triangulation algorithms: m fixed, n variable, $m = 400$

8.2.6 Summary of random graph time complexity

Constant edge density These experiments matched the theoretical time complexities well, as particularly seen in the slopes of the log plots. Specifically, LEXP and MCS showed $O(n^3)$ behaviour and LEXM and MCSM showed $O(n^4)$ behaviour. MSVS also showed $O(n^4)$ behaviour.

Linear number of edges Here, we expected to see a speedup from $O(n^4)$ to $O(n^3)$ or $O(n^3)$ to $O(n^2)$, as appropriate. Or, equivalently, a reduction by 1 in the log slope. LEXM and MCSM both achieved this, while LEXP (decrease of 0.5) and MCS (decrease of 0.3) did not. MSVS remained $O(n^4)$.

Varying edge density LEXM behaved somewhat strangely, showing initially bad behaviour with increasing density, then, eventually, improving (taking less time). On the other hand, LEXP, MCS and MCSM showed behaviour which could be closely modelled with either a linear function or a square root function. This might be related to the fact that the horizontal axis takes linear steps on a scale from 1 to $O(n^2)$. And, indeed, if we plot on a scale of 1 to $O(\sqrt{n^2})$, we get more obviously linear functions. However, a straight line still models the original data quite well, and we are unable to plot "further" to check which is the true curve, due to the fact that there is no point in considering graphs with more than $n(n-1)/2$ edges.

MSVS provided perhaps the most interesting results of this section. It showed significant dependence on edge density, but a function modelling this dependence was not easily-identifiable. The amount of time it took increased sharply to around 25%-30% density, then decreased in a fashion seemingly linear with increasing density. This behaviour was repeated very closely on different-sized graphs.

Varying number of vertices We predicted $O(n)$ behaviour for LEXP and

MCS, and $O(n^2)$ behaviour for LEXM and MCSM. The latter two algorithms matched the prediction closely, but the former two were more "ragged". However, the claim that they behave linearly is plausible.

MSVS could be described as "very approximately linear", but it is again proving difficult to analyse.

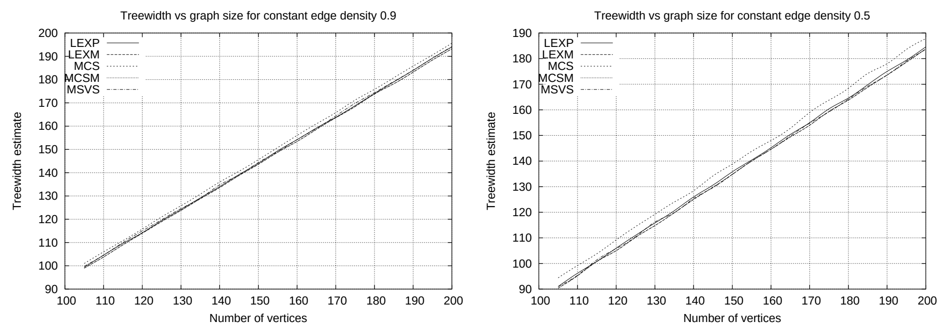
8.3 Estimate testing

These running time results were all very elegant. We do, however, feel obliged to give some small consideration also to the accuracy of the algorithms.

8.3.1 Random graphs

The first context we can do this in is that of the same experiments conducted previously — random graphs generated in some systematic fashion. This is imperfect, as the true treewidth of the graphs is not known. However, we know that none of the heuristics can *under*estimate. As such, the lowest answer must be the best. Moreover, the results may give us some indication of how treewidth changes with changing graph sizes.

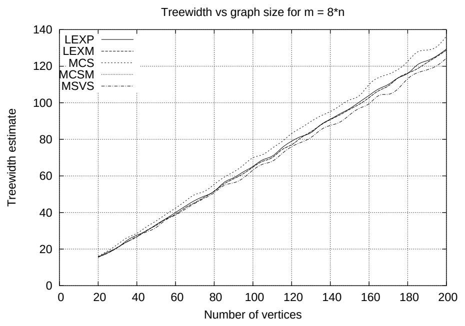
As with the time results, this data was "nice". There was little variation from one run to another (for a given set of parameters).



Treewidth estimate for graphs of various sizes with a fixed edge density. In the density 0.9 plot on the left, the slope ranges as high as 1.00 (MCS) and as low as 0.99 (all others). In the density 0.5 plot, the slope is approximately 0.1 less.

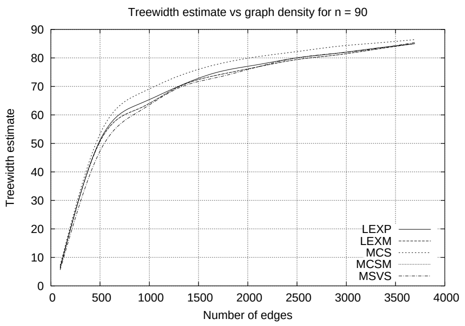
Note that the maximum possible treewidth for a graph with n vertices and edge density d is $\lfloor(1+\sqrt{1+4dn(n-1)})/2\rfloor$ — this is the largest possible clique the graph can contain. If this (effectively) linear function is plotted, it occurs slightly below the 0.9 data and slightly above the 0.5 data. There is an average difference of approximately 9 between the two data sets.

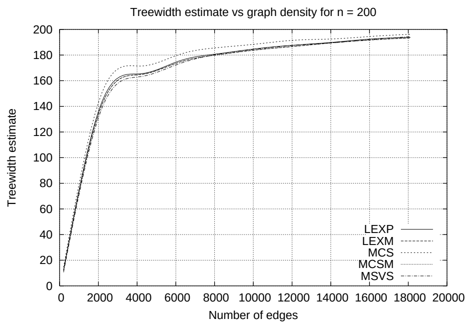
Figure 8.53: Treewidth estimates for increasing graph size with constant edge density.



This is similar to Figure 8.53. However, here the slopes are 0.67 for MCS, 0.61 for MSVS, and 0.63 otherwise. It is interesting to note that, although the edge density is decreasing here with increasing n, the estimates are still increasing linearly with graph size.

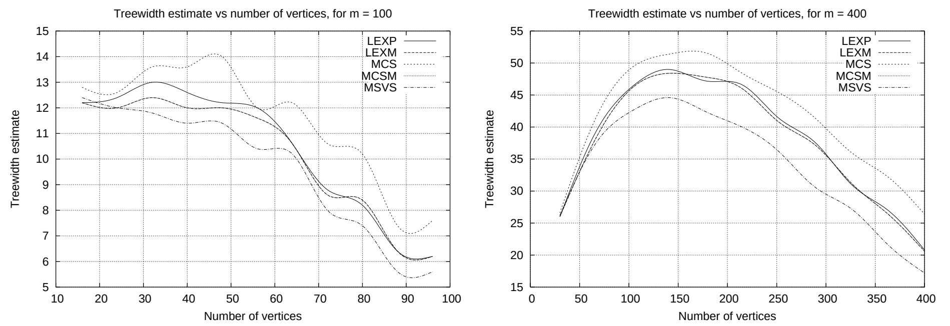
Figure 8.54: Treewidth estimates for increasing graph size with $m = 8n$.





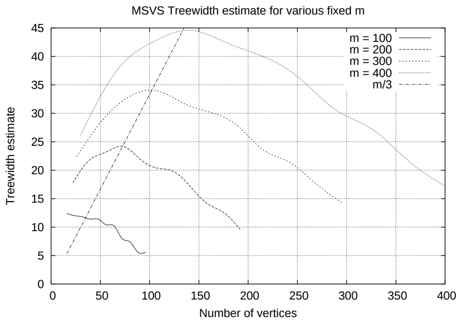
If we discount the first three points, the graph with $n = 90$ has a slope of approximately 0.0060, and the graph with $n = 200$ has a slope of approximately 0.0015. These suggest, once a graph has reached a certain minimum edge density (of around 30%), the treewidth is about as bad as it is going to get.

Figure 8.55: Treewidth estimates for increasing graph edge density.



The first plot shows only that the treewidth decreases with edge density — which is unsurprising — and that MCS provides the worst estimate and MSVS the best. The plot on larger graphs, however, is more interesting: it shows the treewidth estimate initially worsening with increasing n, although the real treewidth is certainly not increasing, and cannot be above 28 in any case (because a complete graph with 29 vertices has more than 400 edges). MSVS has the best performance and MCS has the worst. See also Figure 8.57.

Figure 8.56: Treewidth estimates for fixed number of edges but varying number of vertices.



This plot shows MSVS estimates of treewidth versus n for various fixed m. The curves are all different because of the different numbers and graph sizes involved, but the similarity of shape in the larger three is obvious. I have also plotted the line $m/3$, showing that the turning point occurs at approximately $m = 3n$. It is worth recalling that, while the estimate is decreasing to the right of this line, it is likely that the true treewidth is also decreasing.

Figure 8.57: MSVS treewidth estimates for various fixed m.

8.3.2 Summary of random graph performance

In all these experiments, all five heuristics produced extremely similar results, with only Figure 8.56 showing any clear differences. MCS always provided the worst estimate (if only by a very slight margin), and MSVS the best.

Constant edge density Here, we saw the treewidth estimates increasing perfectly linearly. This, together with the similarity of the different algorithms' results, suggests that treewidth on dense random graphs is $O(n)$ See also Figures 8.66 and 8.67.

Sparse graphs The edge density is lower than the previous case, and moreover is decreasing as $16/(n-1)$. As such, the treewidth is lower, and increases linearly with graph size but at a lesser rate. Furthermore, the results for constant m indicate poor performance on sparse graphs. As such, the true width may conceivably be (much) lower.

Varying m for constant n: Treewidth versus density In these examples, the treewidth estimates had reached 80–90% of their maximum possible value by edge density 30%. The true treewidth may be lower, but my belief is that the heuristics are quite accurate on dense graphs.

Varying n for constant m Putting a bound on the number of edges places a bound on the treewidth: a clique containing m edges will have $\lfloor (1+\sqrt{1+8m})/2 \rfloor$ vertices, so the treewidth cannot exceed this number. With this information as a guide, we see that the algorithms perform poorly on larger, less dense graphs. Focusing on MSVS in particular, we see that the estimate increases until $n = m/3$. I cannot say just how bad the estimate is at this point, but a 3-tree with m edges has $m/3+2$ vertices, and so the true answer could be as low as 3.

8.3.3 Random partial k-trees

The preceding section's results were somewhat inadequate — they allowed us to the evaluate comparative performance of the heuristics, but (in most cases) provided no insight into the absolute accuracy. To solve this problem, we can recall the characterisation of graphs of treewidth k as *partial k-trees*. If I can generate a k-tree and then randomly remove some fraction of its edges, I can create graphs as large as I like, for which I know that the true treewidth is at most k (it may, conceivably, be less).

We can generate random k-trees as follows:

1. Make a $(k+1)$-clique.
2. Pick a random vertex.
3. Find a k-clique containing the vertex.
4. Add a new vertex and connect it to all vertices in this k-clique.
5. Repeat until we have the desired number of vertices.

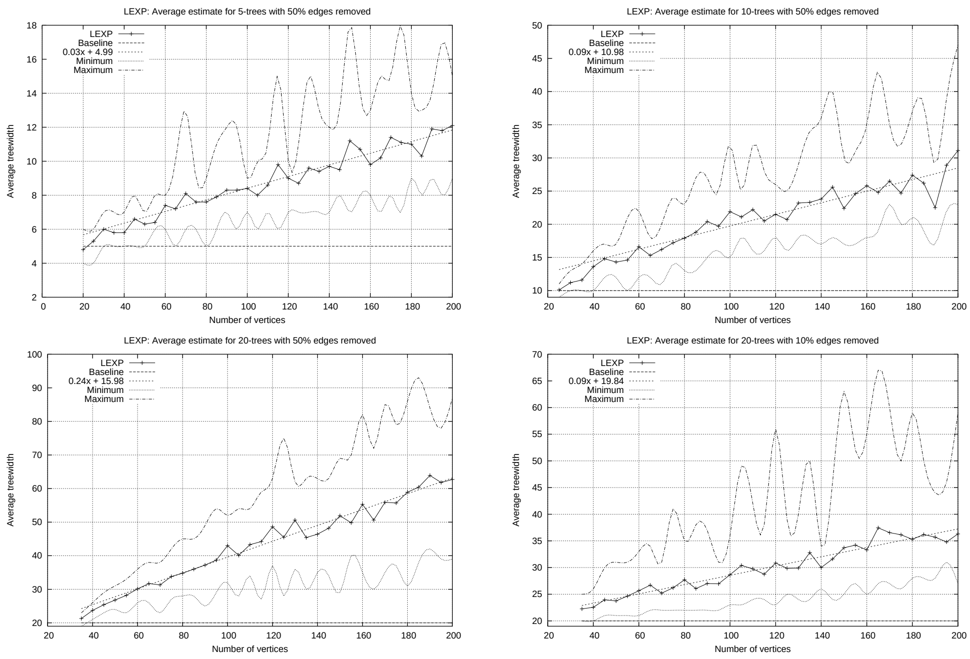
(note that the **Find** step should be done fairly / randomly)

Now, a k-tree with n vertices has $\frac{(k+1)k}{2} + k(n-k-1)$ edges. In what follows, therefore, I will be considering graphs where m is a constant fraction of this number, as opposed to the usual notion of edge density. Notice that this means that m will increase as a linear function of n.

Specifically, I consider trees of 50% "k-tree density" and treewidths 5, 10 and 20. I chose not to test higher width graphs; most algorithmic uses for treewidth are unpleasantly exponential in k. I also examine treewidth 20 graphs with only 10% of the edges removed. Finally, I plot performance against a varying fraction of edges removed, for a fixed number of vertices.

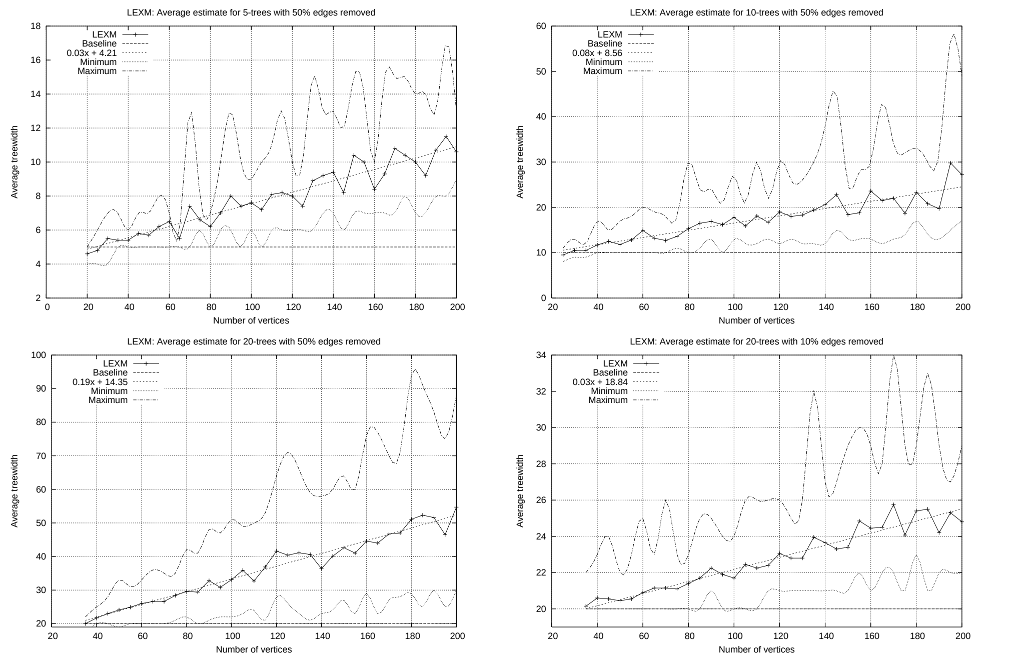
It is important to be aware that this technique will generate different graphs from the purely random graphs of the previous section. For example, the family of 20-trees is sparse (with $m = 20 * n - 210$), but the minimum degree of every vertex will be 20, whereas in purely random graphs one would expect some vertices to be isolated, or nearly so.

In contrast to the purely random graphs, this data is much "rougher". In particular, LEXP, LEXM and MCSM showed significant variation. For example, on 50% "k-tree density" graphs, with 200 vertices and a tree-width of 20, LEXM reported an average treewidth of 54.7, but with a minimum of 30 and a maximum of 88. To indicate this variation, I have plotted minimum- and maximum-estimate curves also.



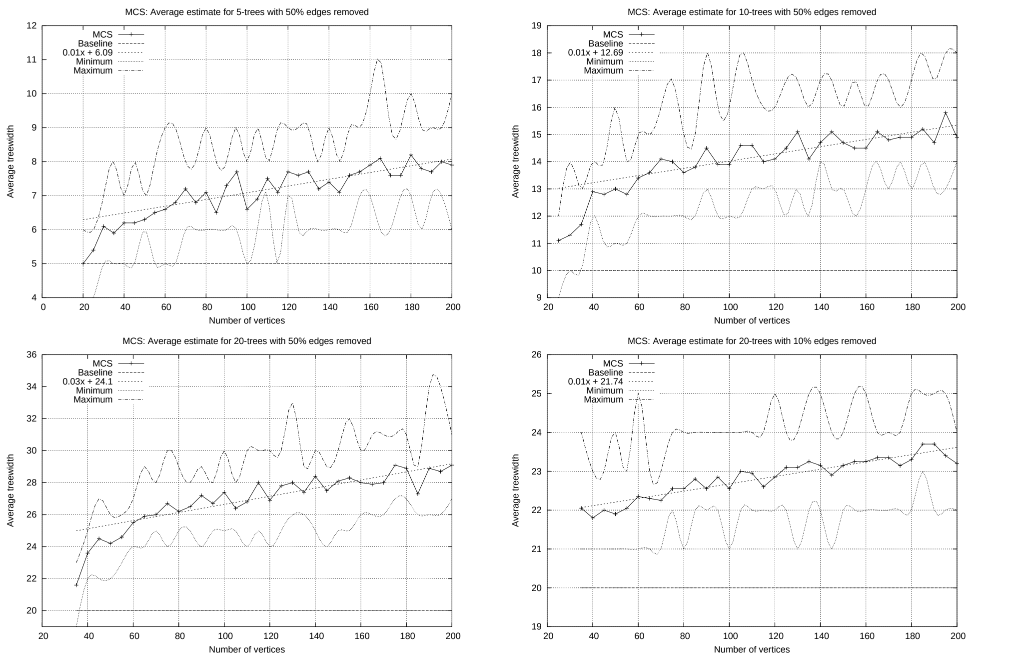
LEXP shows a definite upwards trend in all examples. In particular, for 20-trees with 50the average reaches almost treble the real upper bound. It is clear that performance is comparatively better for denser graphs, which are more likely to be triangulated or almost triangulated.

Figure 8.58: LEXP estimate accuracy in various circumstances.



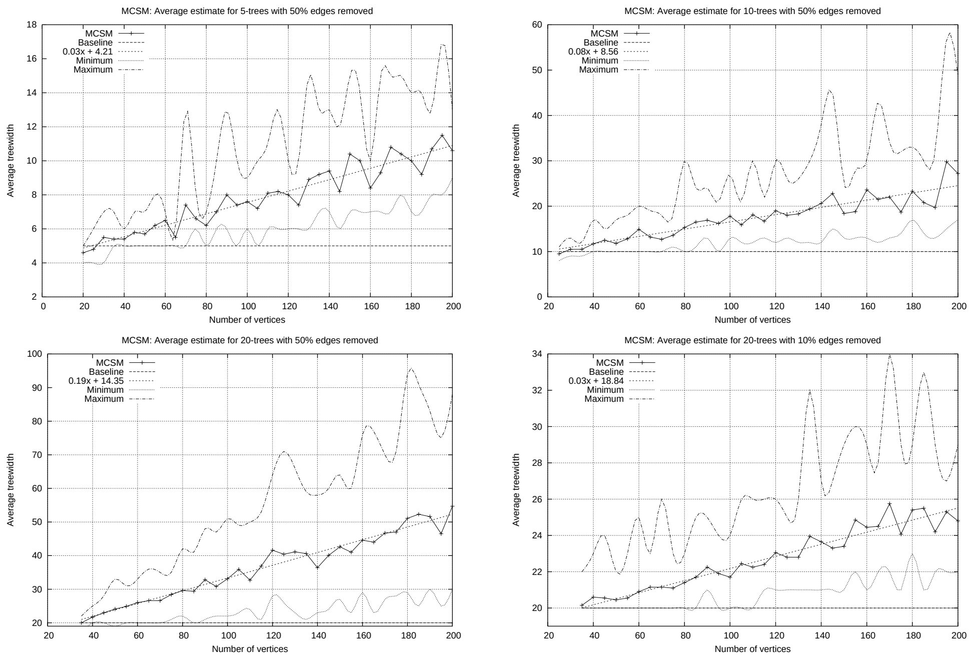
LEXM performs better than LEXP, particularly on the denser graphs. However, the error has climbed to 150% by n = 200 in the case of 20-trees with 50% edge removal.

Figure 8.59: LEXM estimate accuracy in various circumstances.



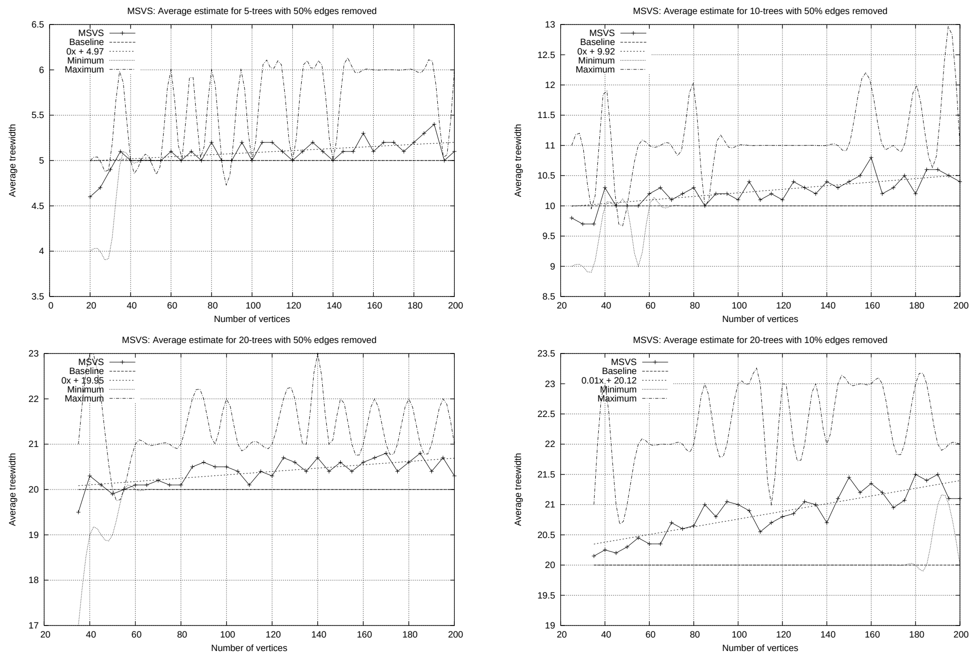
MCS is the "flattest" of the triangulation algorithms. In other words, it appears the least affected by graph size.

Figure 8.60: MCS estimate accuracy in various circumstances.



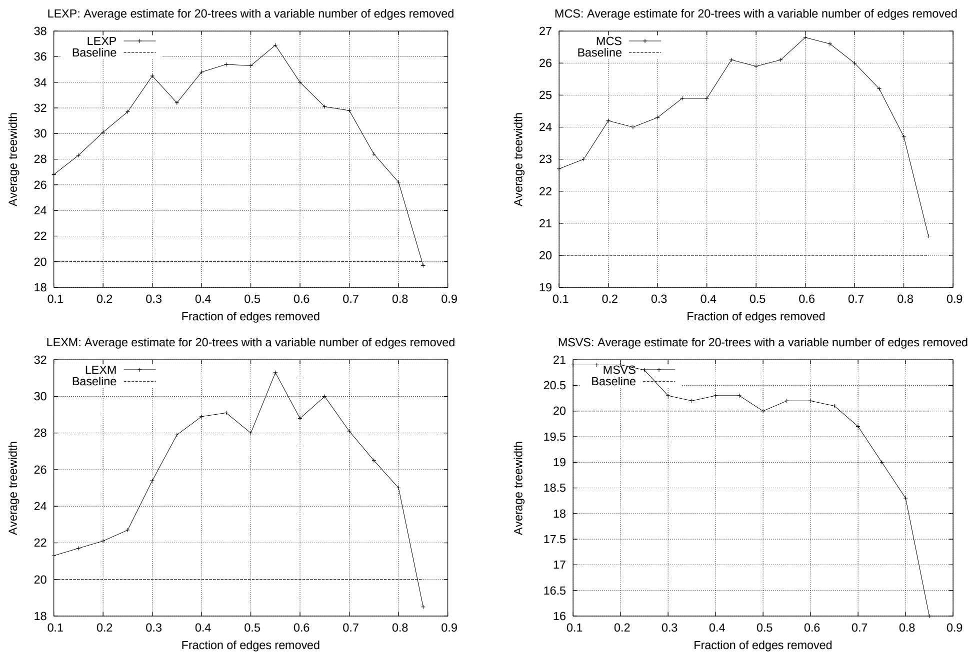
MCSM (for me) behaves exactly as LEXM, so there is no need to comment further.

Figure 8.61: MCSM estimate accuracy in various circumstances.



MSVS is clearly the best, coming very close to the real answer. Interestingly, it performs worse on the denser graphs, whereas the triangulation algorithms performed better. The word "worse", however, is only applied relative to the other plots here; it is still obviously the best.

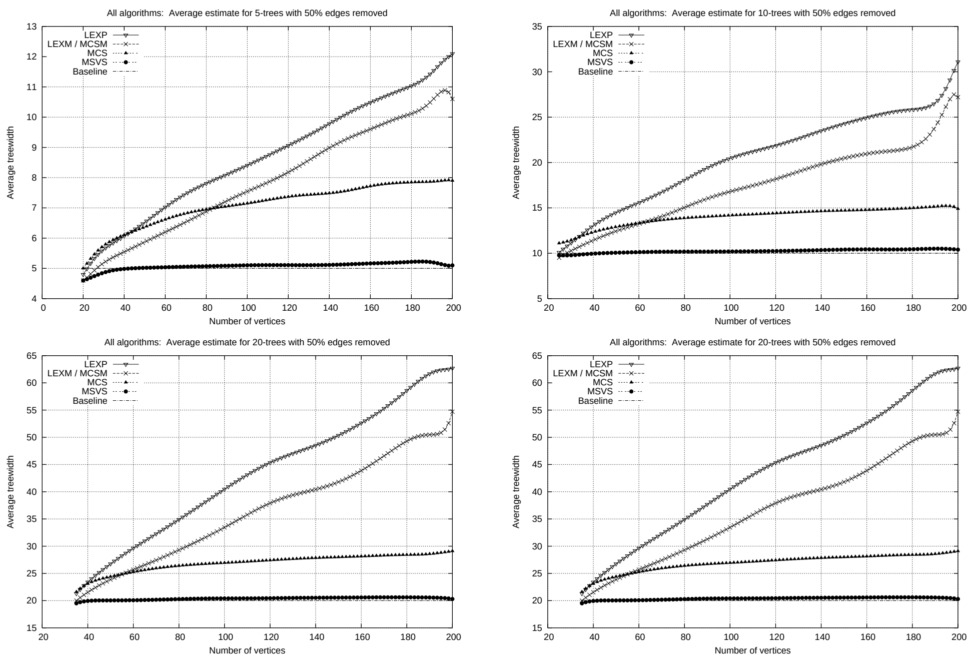
Figure 8.62: MSVS estimate accuracy in various circumstances.



Despite the fact that the real treewidth is certainly not increasing, and may very well be decreasing, all four triangulation heuristics produced worse behaviour as the graph edge density decreased. This is perhaps not particularly surprising, given that the heuristics have proven *perfect* performance for triangulated graphs, such as k-trees (in that they are guaranteed to find a perfect elimination ordering).

MSVS, on the other hand, seems to perform better with decreasing density — although this is hard to be certain of, as the real treewidth may be decreasing (for example, is an estimate of 20 at 0.5 “on the money”, or is it 1 above the true treewidth, as the average was at 0.1?).

Figure 8.63: Plotting algorithm accuracy for fixed n / k, but changing m



Here, we demonstrate the relative behaviours, to allow direct comparison.

Figure 8.64: Performance comparison for all algorithms on k-trees

8.3.4 Summary of random partial k-tree results

These graphs showed much more variation in results. LEXP performed the most poorly of all, such that by the time n had climbed to 200, its results were at best a factor of 2 of the true answer, and at worst, a factor of 4.

At the other end of the scale, MSVS performed extremely well. It only showed a very slight worsening trend as the graph size increased, and always came very close to, or achieved, the true answer.

MCS was the best of the triangulation heuristics. Its estimates were worsening with increasing graph size, but at a much slower rate than the other three.

LEXM and MCSM showed surprising results. It is not clear from the figures, but there were some graphs for which they performed well, achieving results similar to or better than MCS, and other graphs for which they performed extremely poorly. Thus, the average is not the mode; rather, it is (around) half way between the two likely outcomes.

How do we explain this, in light of the fact that LEXM and MCSM produce minimal triangulations and MCS doesn't, and in light of the fact that MCS performed the worst (although only slightly) on random graphs?

I speculate that it is because of the type of graphs being produced. Recall from Chapter 5 that we characterised MCS as a depth-first search through a k-tree. This suggests that MCS is well-suited to dealing with the type of graph we are generating here.

But what of MCSM? Why is it performing poorly, then? In searching for paths to reachable vertices, it may be denying its depth-first nature. Rather than only updating the counts of local vertices, which will support depth-first behaviour, it may update the counts of vertices seen in other parts of the graph. This will lead to arbitrary tie-breaking, which may in turn lead to the breadth-first behaviour that produced poor results in LEXM and LEXP.

This section also compared performance for 20-trees with differing fractions of edges removed. The triangulation heuristics performed better on the denser graphs, which is unsurprising as such denser graphs will require smaller fill-in. On the other hand, MSVS performed slightly worse on the denser graphs, perhaps because small separating vertex sets are easier to find when there are fewer edges.

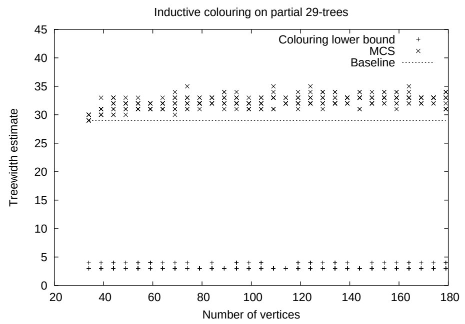
8.4 Inductive lower bounds

In Chapter 7, we discussed two lower bounds based on the idea of inductive graphs.

The idea behind them is that the inductivity of a graph is a lower bound to its treewidth. We can either calculate the inductivity directly, or estimate it based on a graph colouring.

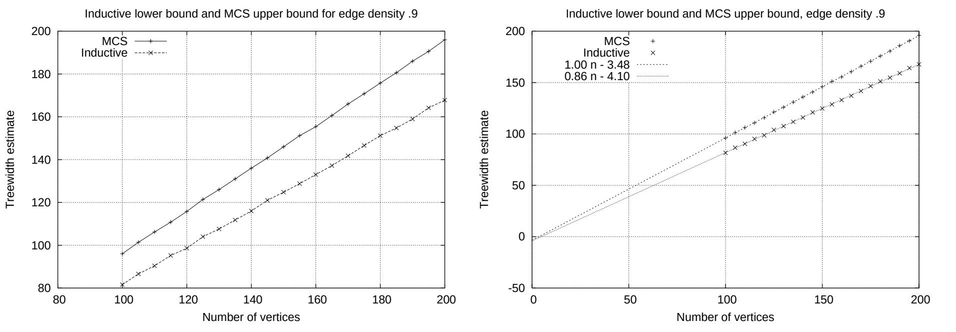
8.4.1 Inductive lower bound results

We will first deal with the idea of using a graph colouring as a lower bound. We stated that this colouring bound is very inexact. As evidence, we present Figure 8.65. This figure is typical of results collected, so we won't present others. Furthermore, the algorithm did fulfil its promise of being fast. In fact, it was too fast to yield any interesting pictures...



This experiment was conducted on 29-trees with only 10% of their edges removed. When considering partial k-trees, we expect th

Figure 8.65: Colouring lower bound versus MCS upper bound for treewidth 29 graphs.

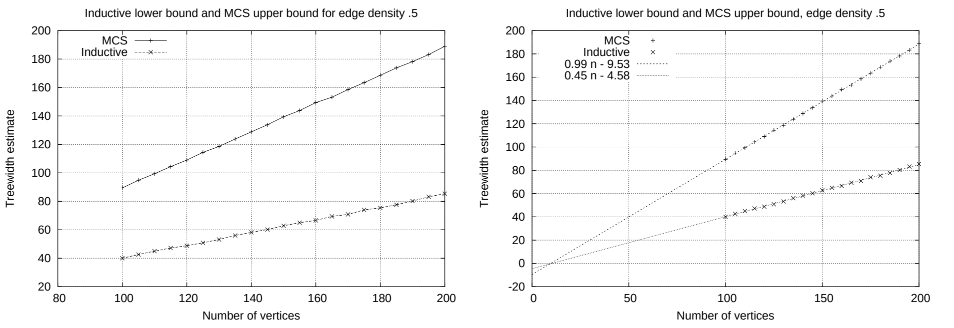


This shows that out intuition in Section 8.3.2 was largely correct: the treewidth is certainly growing as $O(n)$ on these graphs. It is impossible to say what the precise growth rate is, however we can pinpoint it as being between $0.85n$ and n.

Figure 8.66: Inductive lower bound and MCS upper bound versus graph size on random graphs.

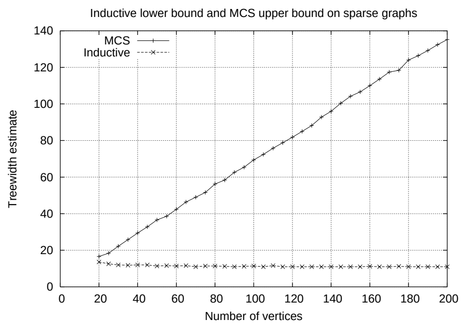
Our other option was to determine the inductivity directly, which was not too difficult to do. This yields a more accurate lower bound (rather than determining a lower bound for a lower bound), and thus allows us to conduct some meaningful experiments.

The first experiments I conduct are on random graphs, which we can then compare with the results from Figures 8.53 and 8.54. Thus, Figure 8.66 compares the inductive bound with MCS on random graphs with edge density 0.9; Figure 8.67 compares with MCS on random graphs with edge density 0.5; and Figure 8.68 compares with MCS on sparse graphs, with $m = 8n$.



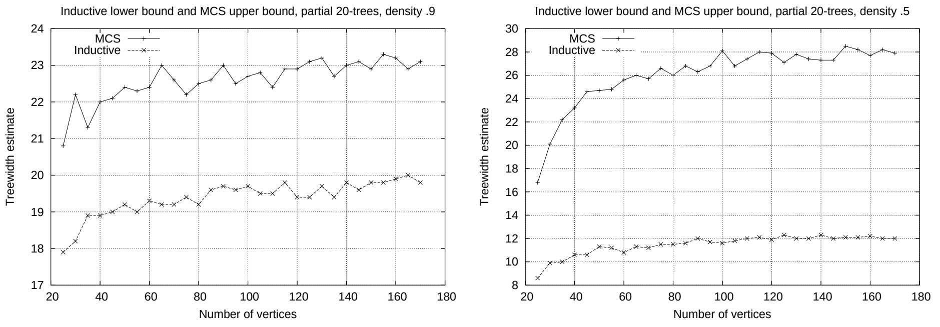
Clearly, compared with the previous figure, the gap between upper and lower bound has increased. My intuition here is that is represents a worsening of the lower bound, however I cannot prove this.

Figure 8.67: Inductive lower bound and MCS upper bound versus graph size on random graphs.



What do we make of this? The algorithm reports a lower bound of 11, irrespective of graph size. Again, my intuition is that this represents a failure of Algorithm 7.1 to produce a good lower bound. I will discuss my reason for this intuition in Section 8.4.2.

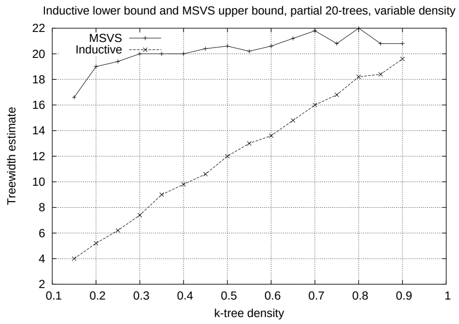
Figure 8.68: Inductive lower bound and MCS upper bound versus graph size on random graphs.



On graphs which are very close to being k-trees, such as those considered in the left-hand plot, the lower bound provided is clearly very good, coming close to (or even occasionally achieving) the known upper bound. This is perhaps not surprising, as with few edges removed, there is a good chance that a $(k+1)$-clique may remain. The presence of such a clique would force any online ordering to be at least k-inductive.
On the other hand, when half the edges of the k-tree are removed, the inductivity of the graphs drops. However, it is not clear that the treewidth has dropped with it. Indeed, as seen in Figure 8.62, even the best algorithm here, MSVS, did not generally produce estimates below k (in this case, below 20).

Figure 8.69: Inductive lower bound and MCS upper bound versus graph size on random graphs.

Finally, we examine the performance of Algorithm 7.1 on partial k-trees. Figure 8.69 shows its performance on partial 20-trees at two different "k-tree densities", and Figure 8.70 demonstrates the effect of varying the fraction of edges removed.



Here, we compare the lower bound with MSVS (as the best of the upper bounds), on k-trees with varying fractions of their total edges removed. Assuming MSVS is fairly accurate (which seems likely), it clearly gives a poorer lower bound the fewer edges the graph has.

Figure 8.70: Inductive lower bound and MCS upper bound versus graph size on random graphs.

8.4.2 Inductive lower bound summary

Random graphs We conducted three different types of experiment on random graphs. The first of these, graphs with an edge density of 90%, showed that Algorithm 7.1 produced an accurate lower bound on the treewidth of very dense graphs. We can know that the bound was close to the true answer because it was close to an upper bound provided by MCS.

When we dropped the edge density to 50%, the difference between the bounds increased to over a factor of 2. Furthermore, when we changed to sparse graphs, the inductive bound stayed fixed (at 11), irrespective of the graph size, whereas the MCS estimate continued to increase. My intuition for random graphs is that, as the graph size increases, it is more likely that the graph will include a small high-treewidth "clump", and as such, the overall treewidth will increase. This suggests to me that Algorithm 7.1 provides a poor lower bound on sparse graphs.

Partial k-trees On graphs which were nearly k-trees, the inductive lower bound was very close to the true answer. However, as the number of edges decreased, the bound decreased, and yet the MSVS estimate did not decrease by as much, suggesting that the lower bound is getting worse. It is unsurprising that the bound should decrease, as with fewer edges, it will clearly be easier to find a low-inductivity online ordering.

Chapter 9

Conclusions and future work

In this thesis, we described five techniques for finding upper bounds on the treewidth of a graph, and two for finding lower bounds. Of the upper bounds, four were based on the notion of triangulations and the last was based on using vertex separators to refine tree decompositions. The lower bounds we considered were based on the theory of online graphs, and in particular, the definition of d-inductive graphs.

We discussed their origins and motivations, and provided thoughts on how to implement them efficiently.

In this chapter, we give a summary of our experimental results, and we discuss some possible related research topics.

9.1 Summary of results

We showed that the running times of the triangulation heuristics were as expected on dense graphs, but only MCSM had the expected speedup when we shifted our focus to sparse graphs (although LEXM came close). We showed that MSVS had $O(n^4)$ behaviour on dense and sparse graphs, both. Finally, we considered the effects of changing density. We expected the triangulation heuristics to be faster on graphs with fewer edges, and this was true for all but LEXM. Furthermore, MSVS was slowest on graphs with an edge density of 0.25–0.30, taking less time as graph density increased thereafter.

We didn't test the time complexity of the lower bound algorithms, as they are (comparatively) very fast.

We then moved to evaluating the estimates produced. On random graphs, we found that the upper bound heuristics all produced very similar results. We found that, on dense random graphs, the treewidth increased linearly with the graph size. By comparing with the best lower bound we had, we found that on very dense graphs (90% edge density), the upper bounds were at most 18% above the true answer. However, when the edge density dropped to 50%, the difference between upper and lower bounds was more than a factor of two.

On sparse random graphs, the upper bound heuristics were still very similar, and the estimates still increased linearly with graph size, although now only half as fast as with dense graphs. However, our lower bounds did not increase with graph size at all. Thus, we cannot determine how treewidth varies on sparse random graphs.

Finally, we considered partial k-trees, by which we mean graphs generated by first creating a k-tree, and then removing each edge with a given probability. On this class of graphs, we found that MSVS performed extremely well, and that MCS also performed well. However, the other three algorithms performed poorly, with estimates worsening to a factor of 2 or more of the true answer when the graph size increased only to 200 vertices. We speculated that MCS performed well due to its nature suiting the generation style of these graphs.

We considered using first-fit graph colouring as a lower bound, but found that even on dense k-trees, it produced very poor lower bounds. We then looked finding the inductivity of the graph as a lower bound, and found that on partial k-trees with only 10% of edges removed, it came very close, but when we increased the number of edges removed to 50%, it reported a bound of only 60% of the true value. These results were unsurprising, as it would report k when run on a k-tree, and as the inductivity will certainly go down as a graph becomes less connected.

9.2 Future work

9.2.1 LB-Triangulation

In [44], Lekker and Boland characterised triangulated graphs in a new way:

Definition 9.2.1 (LB-simplicial). A vertex v is LB-simplicial if every minimal separator contained in the neighbourhood of v is a clique.

Theorem 6 (Lekkerkerker and Boland [44]). *A graph is triangulated if and only if every vertex is LB-simplicial.*

In [10], Berry *et al* observe that this characterisation suggests the triangulation algorithm described in Algorithm 9.1. There are two issues with this algorithm. The first concern is that LB-simpliciality is more difficult to test for than regular simpliciality. However, in [35], Heggernes and Villanger describe an efficient implementation of this algorithm.

Algorithm 9.1 LB-Triang

Require: Graph $G = (V, E)$
Ensure: Triangulated graph $\bar{G} = (V, \bar{E})$, where $E \subseteq \bar{E}$.
  $\bar{E} := E$
  for all $v \in V$ **do**
    Add edges to $\bar{E}$ to make v LB-simplicial in $\bar{G}$.
  end for

The second issue is that the algorithm will produce a minimal triangulation from any ordering given. In fact, it can be shown that a graph triangulation is minimal if and only if it can be produced using Algorithm 9.1

and some vertex ordering. However, this is a double-edged sword. While on the one hand, it gives us a nice characterisation of minimal triangulations, on the other hand there is (potentially) a large gulf between minim*al* and mini*mum* triangulations. As such, it would be interesting to investigate if there were a way to use LB-Triang to produce useful tree decompositions.

9.2.2 Pathwidth

If a tree decomposition T of a graph G is also a path, then T is (also) called a *path decomposition*. The minimum width of all path decompositions of G is called the *pathwidth* of G, and is a measure of how "path-like" a graph is.

What is the relationship between treewidth and pathwidth? Clearly, for any graph, the treewidth is bounded above by the pathwidth. Moreover, there is no upper bound on the pathwidth of trees, and thus the treewidth cannot provide any useful bound on the pathwidth.

However, suppose we create random graphs of bounded pathwidth. What are the tree decompositions likely to look like?

9.2.3 Persistence

We described treewidth and pathwidth before as measures of how "tree-like" or "path-like" a graph is. But these do not necessarily tell the whole story. This is particularly easy to see in the case of path decompositions. For example, consider Figure 9.1: This is a graph where the minimal path decomposition (it is also the minimal tree decomposition) does not accurately represent the structure of the graph. That is, if one were to look at the decomposition, one might expect the corresponding graph to resemble something like the graph in (c).

The fact that it does not is reflected in the presence of node 0 in every bag of the decomposition. We say that this graph is a graph with high

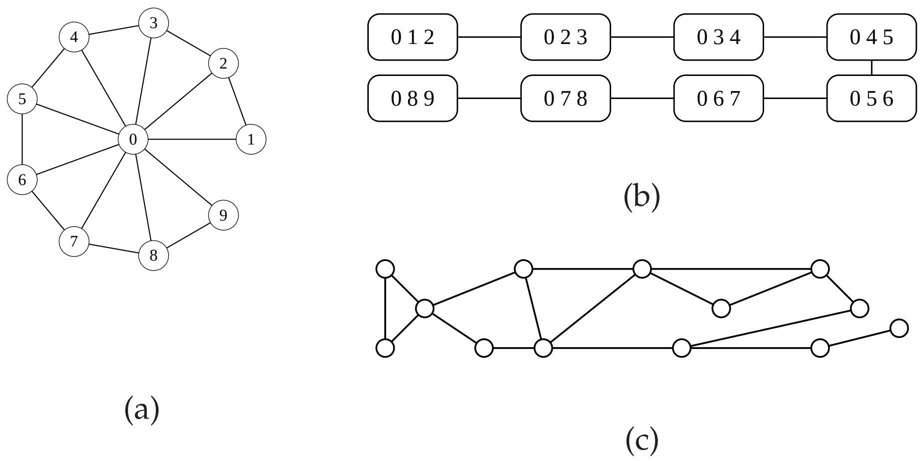


Figure 9.1: (a) An un-path-like graph. (b) A tree / path decomposition. (c) A more obviously path-like graph.

persistence.

Definition 9.2.2 (Persistence). Let G be a graph and T be a tree decomposition for G. Then the persistence of T is $\max\{|T_v| : v \in V\}$.

We can also talk about the (minimum) persistence of a graph, by considering the minimum persistence over all possible tree decompositions of the graph (of minimum treewidth).

Graphs of high persistence, such as the one shown in Figure 9.1, are, in some sense, "unnatural" or pathological. In real-world examples, a single data point would be unlikely to have an influence throughout an entire data structure or model.

As such, it would be interesting to know how the persistence of graphs affects our ability to find the treewidth or pathwidth.

In particular:

1. If we know a graph has a $\{$tree,path$\}$-decomposition of width w and persistence p, can we efficiently find the decomposition?

2. If a graph's size and treewidth are held constant but the minimum

persistence is varied, will the accuracy or running times of the heuristics be affected?

Answering these questions may require more sophisticated graph generation techniques than we have used thus far. One possible approach would be to generate a tree decomposition, and then add edges between vertices sharing a bag. This technique would allow control over the maximum persistence and treewidth, however it would not be obviously simple to control the graph's size, or to ensure that all valid graphs are equally likely. A more promising technique may centre around the parse tree characterisation of graphs of bounded treewidth.

Bibliography

[1] ALBERS, S. Online algorithms: A study of graph-theoretic concepts.

[2] ALBERS, S. Competitive online algorithms. BRICS Lecture Series LS-96-2, BRICS, Department of Computer Science, University of Aarhus, September 1996.

[3] ALBERS, S. Online algorithms: A survey. To appear in Mathematical Programming, 2003. Invited paper in ISMP 2003., 2003.

[4] ANDERSON, R. J., AND SETUBAL, J. C. Goldberg's algorithm for the maximum flow in perspective: a computational study. In *Network Flows and Matching: First DIMACS Implementation Challenge* (1993), D. S. Johnson and C. C. McGeoch, Eds., AMS, pp. 1–18.

[5] ARNBORG, S. Efficient algorithms for combinatorial problems on graphs with bounded decomposability (a survey). *BIT*, 25 (1985), 2–33.

[6] ARNBORG, S., AND PROSKUROWSKI, A. Linear time algorithms for np-hard problems restricted to partial k-trees. *Discrete Applied Mathematics 23*, 1 (1989), 11–24.

[7] BERRY, A., BLAIR, J. R. S., AND HEGGERNES, P. Maximum cardinality search for computing minimal triangulations. In *Graph-Theoretic Concepts in Computer Science* (Cesky Krumlov, Czech Republic, June

2002), L. Kucera, Ed., Lecture Notes in Computer Science, Springer-Verlag.

[8] BERRY, A., AND BORDAT, J.-P. Triangulated and weakly triangulated graphs: Simpliciality in vertices and edges. In *6th International Conference on Graph Theory, Luminy, France.* (2000).

[9] BERRY, A., BORDAT, J.-P., AND HEGGERNES, P. Recognizing weakly triangulated graphs by edge separability. *Nordic Journal of Computing 7*, 3 (2000), 164 – 177.

[10] BERRY, A., BORDAT, J.-P., HEGGERNES, P., SIMONET, G., AND VILLANGER, Y. A wide-range algorithm for minimal triangulation from an arbitrary ordering. Reports in Informatics 243, University of Bergen, 2003.

[11] BERRY, A., BORDAT, J.-P., HEGGERNES, P., SIMONET, G., AND VILLANGER, Y. A wide-range algorithm for minimal triangulation from an arbitrary ordering. Reports in Informatics 243, University of Bergen, Norway, 2003.

[12] BODLAENDER, H. L. A linear time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on Computing 25* (1996), 1305–1317.

[13] BODLAENDER, H. L. A partial k-arboretum of graphs with bounded treewidth. *Theoretical Computer Science 209* (1998), 1–45.

[14] BODLAENDER, H. L. Treewidth: Algorithmic techniques and results. In *Proceedings 22nd International Symposium on Mathematical Foundations of Computer Science* (1998), vol. 1295 of *Lecture Notes in Computer Science*.

[15] BODLAENDER, H. L. A note on domino treewidth. *Discrete Mathematics and Theroetical Computer Science 3*, 4 (1999), 141–150.

[16] BODLAENDER, H. L., AND ENGELFRIET, J. Domino treewidth. Tech. rep., Utrecht university, 1994.

[17] BODLAENDER, H. L., GILBERT, J. R., HAFSTEINSSON, H., AND KLOKS, T. Approximating treewidth, pathwidth, frontsize, and shortest elimination tree. *Journal of Algorithms 18* (1995), 238–255.

[18] BODLAENDER, H. L., AND KLOKS, T. Efficient and constructive algorithms for the pathwidth and treewidth of graphs. *Journal of Algorithms 21* (1996), 358–402.

[19] BORODIN, A., AND EL-YANIV, R. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.

[20] BRASSARD, G., AND BRATLEY, P. *Fundamentals of Algorithmics*. Prentice Hall, Englewood Cliffs, New Jersey 07632, 1996.

[21] CHEN, J., KANJ, I. A., AND JIA, W. Vertex cover: Further observations and further improvements. *Journal of Algorithms*, 41 (2001), 280–301.

[22] DE FLUITER, B. L. E. *Algorithms for Graphs of Small Treewidth*. PhD thesis, Department of Computer Science, Utrecht University, 1997.

[23] DIESTEL, R. *Graph Theory*, 2 ed. Springer-Verlag, 2000.

[24] DOWNEY, R. Parameterized complexity for the skeptic. In *Proceedings of the 18th IEEE Conference on Computational Complexity* (July 2003).

[25] DOWNEY, R., AND FELLOWS, M. *Parameterized Complexity*. Monographs in Computer Science. Springer, 1997.

[26] EPPSTEIN, D. Diameter and treewidth in minor-closed graph families. *Algorithmica 27*, 3–4 (2000), 275–291.

[27] FORMIN, F. V. Lecture notes. http://www.upb.de/cs/ag-madh/vorl/GraphAlgorithmsI02/.

[28] FULKERSON, D. R., AND GROSS, O. A. Incidence matrices and interval graphs. *Pacific Journal of Mathematics 15* (1965), 835–855.

[29] GAREY, M., AND JOHNSON, D. *Computers and Intractability: A Guide to the Theory of NP-completeness*. W.H. Freeman, San Francisco, 1979.

[30] GAVRIL, F. The intersection graphs of subtrees of trees are exactly the chordal graphs. *Journal of Combinatorial Theory, series B 16* (1974), 47–56.

[31] GOLDBERG, A. V. Recent developments in maximum flow algorithms. Tech. rep., NEC Research Institute, Inc., 1998.

[32] GROHE, M. Local tree-width, excluded minors and approximation algorithms. *to appear, Combinatorica* (2001).

[33] GROHE, M. Parameterized complexity for the database theorist. *SIGMOD Record 31*, 4 (2002).

[34] HAJIAHAYI, M. Algorithms for graphs of (locally) bounded treewidth. Master's thesis, University of Waterloo, 2001.

[35] HEGGERNES, P., AND VILLANGER, Y. Efficient implementation of a minimal triangulation algorithm. In *European Symposium on Algorithms* (2002), R. Mohring and R. Raman, Eds.

[36] HICKS, I. V. *Branch Decompositions and their Applications*. PhD thesis, Houston, Texas, 2000.

[37] HUARD, J. Chordal graphs: Their testing and their role, 1994.

[38] IRANI, S. Coloring inductive graphs on-line. *Algorithmica 11*, 1 (January 1994), 53–72.

[39] JENSEN, F. V. Junction trees and decomposable hypergraphs. Tech. rep., Judex Datasystems, 1988.

[40] JENSEN, F. V., AND JENSEN, F. Optimal junction trees. In *Proceedings of the Tenth Conference on Uncertainty in Artificial Intelligence* (San Francisco, 1994), R. L. Mantares and D. Poole, Eds., Morgan Kaufmann, pp. 360–366.

[41] KLOKS, T. *Treewidth Computations and Approximations*. Lecture Notes in Computer Science. Springer-Verlag, 1994.

[42] KOSTER, A. M. C. A. *Frequency Assignment — Models and Algorithms*. PhD thesis, Universiteit Maastricht, Maastricht, The Netherlands, 1999.

[43] KOSTER, A. M. C. A., BODLAENDER, H. L., AND VAN HOESEL, S. P. M. Treewidth: Computational experiments. Tech. rep., Konrad-Zuse-Zentrum für Informationstechnik Berlin, 2001.

[44] LEKKERKERKER, C. G., AND BOLAND, J. C. Representation of a finite graph by a set of intervals on the real line. *Fundamenta Mathematicae 51* (1962), 45 – 64.

[45] MENGER, K. Zur allgemeinen kurventheorie. *Fundamenta Mathematicae*, 10 (1927), 95–115.

[46] NIEDERMEIER, R. Invitation to fixed-parameter algorithms, 2002.

[47] NIEDERMEIER, R., AND ROSSMANITH, P. Upper bounds for vertex cover further improved. *Lecture Notes in Computer Science 1563* (1999), 561–579.

[48] RIGGINS, K. D. On characterising graphs with branchwidth at most four. Master's thesis, Houston, Texas, 2001.

[49] ROBERTSON, N., AND SEYMOUR, P. D. Graph minors ii: Algorithmic aspects of tree-width. *Journal of Algorithms*, 7 (1986), 309–322.

[50] ROSE, D. J., TARJAN, R. E., AND LEUKER, G. S. Algorithmic aspects of vertex elimination on graphs. *SIAM Journal of Computing 5*, 2 (1976), 266–283.

[51] SHIBATA, Y. On the tree representation of chordal graphs. *Journal of Graph Theory 12*, 3 (1988), 421–428.

[52] SIMON, K. A note on lexicographic breadth first search for chordal graphs. *Information Processing Letters 54* (1995), 249–251.

[53] TARJAN, R. E., AND YANNAKAKIS, M. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM Journal of Computing 13*, 3 (1984).

[54] VAN EMBE BOAS, P. Preserving order in a forest in less than logarithmic time. In *Proceedings of the 16th Annual Symbosium on the Foundations of Computer Science* (1975), pp. 75–84.

[55] WAGNER, F., AND STOER, M. A simple min-cut algorithm. *Journal of the ACM*, 44 (1997), 585–591.