University of Bergen

Master Thesis

A polynomial-time solvable case for the NP-hard problem CUTWIDTH

Author:
Simen Lilleeng

Supervisor:
Prof. Pinar Heggernes

A thesis submitted in fulfilment of the requirements for the degree of Master of Science

in the

Algorithms Research Group
Department of Informatics

June 2014

UNIVERSITY OF BERGEN

Abstract

Faculty of Mathematics and Natural Sciences
Department of Informatics

Master of Science

A polynomial-time solvable case for the NP-hard problem CUTWIDTH

by Simen LILLEENG

The CUTWIDTH problem is a notoriously hard problem, and its complexity is open on several interesting graph classes. Motivated by this fact we investigate the problem on superfragile graphs, a graph class on which the complexity of the CUTWIDTH problem is open. We give an algorithm that solves CUTWIDTH on superfragile graphs in $O(n^2)$ time and $O(n)$ space, thus resolving the complexity of the CUTWIDTH problem on superfragile graphs. We also explore the usefulness of the algorithm for cutwidth on threshold graphs by Heggernes, Lokshtanov, Mihai and Papadopoulos [21] as an approximation algorithm for cutwidth on other graph classes. The CUTWIDTH problem is NP-hard for general graphs and a brute force algorithm would require $O(n!)$ time. We give two faster algorithms solving the CUTWIDTH problem: One algorithm applying dynamic programming that runs in $O^*(2^n)$ time and space, and one algorithm that runs in $O^*(4^n)$ time and $O(n \cdot log(n))$ space by applying the divide-and-conquer technique. Finally we take a look at a similar problem called OPTIMAL LINEAR ARRANGEMENT and suggest algorithms for solving the problem on threshold graphs and superfragile graphs in polynomial time.

Acknowledgements

First of all I am very grateful to have Professor Pinar Heggernes as supervisor. You have been supportive, kind and most importantly patient. Thank you Pinar!

I would like to thank the Algorithms group as a whole for being warm and welcoming and for all the Friday lunches and Winter school trips. A special thank-you goes to Martin, Pim and Sigve for all their helpful comments and support.

I am honored to have been in teams with Johan, Magnar, Marco and Tobias in various programming competitions. It has been a pleasure competing together with you guys!

I would also thank the people at the study hall for being kind and pleasant and for all the table tennis matches we have had.

I would especially like to thank Eirik, Erik, Johan and Sebastian, whom I have shared a lot of interesting discussions related and mostly unrelated to this thesis with.

Bergen, June 2014

Simen Lilleeng

Contents

Chapter 1

Introduction

Imagine being a member of a conference organizing committee. You are about to rent a venue for the conference, and being careful with money, you want to pay as little as possible. The cost of the venue is directly related to the capacity of its waiting area. Therefore you want to find the venue with the lowest capacity that can accommodate all the participants. You know that the people attending this conference are of a special kind. They all have exactly two favorite talks which they will attend. Since they are all highly eccentric, none of them share the same preference of the two same talks. Between every talk there is a fifteen-minute break, which the attendants must spend in the waiting area. An attendant will not enter the waiting area before his first favorite talk has ended, and will stay at the waiting area until his second favorite talk starts. Knowing this, how can you find out the lowest capacity required, and in which order should the talks be scheduled?

This can be modeled as a graph, where every talk is a vertex, and every person forms an edge between the vertices corresponding to his two favorite talks. Now, for an integer k, the capacity of the waiting area the problem becomes the following: Find an ordering of the vertices, from left to right, such that for all vertices, the number of edges going between the vertices on its left, including itself, and the vertices on its right is less than or equal to k.

Let us give a concrete example: Adam and Beatrice are responsible for arranging the annual LA (Letters Association) conference on the first five letters in the alphabet, and they have promised every VILP (Very Important Letter Person) a massage chair between the talks. VILPs are as any other conference-goers,

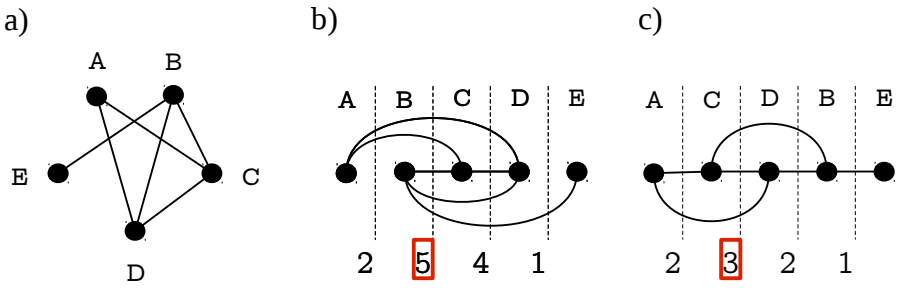

FIGURE 1.1: a) The graph representation of the LA talks. b) Adam's suggestion c) Beatrice's suggestion.

they only attend two talks and none of them share interest in the same pair of talks. There are six VILPs attending the conference in total, their names are hidden for discretion, but their preferences are known: $\{A, C\}$, $\{A, D\}$, $\{B, C\}$, $\{B, D\}$, $\{B, E\}$, $\{C, D\}$. Adam says that they should do like every year, arrange the talks in lexicographical order. Beatrice however, disagrees, she thinks that the talk about letter B should be held second to last. Adam is outraged by this radical suggestion and demands an explanation of this insanity. Beatrice draws a graph modelling the connection between VILPs and the talks, as seen in Figure 1.1. She explains that Adam's suggestion would require five massage chairs while her suggestion only requires three and thus saving LA for a substantial amount of money. Adam's inclination towards order eagers him to object, but he realizes fast that there is no point in arguing against math.

What Beatrice did was solve the CUTWIDTH problem, a problem so hard to solve that it has been shown to belong to a class of problems named NP-*complete* [18]. Solving this problem *efficiently* would be equivalent to solving all problems in *NP* efficiently. Before a precise definition is given, necessary definitions and terminologies will be introduced.

1.1 Definitions and notations

We define a $S_1, S_2, \ldots, S_i$ as a *partition* of a set S if $S_1 \cup S_2 \cup \cdots \cup S_i = S$ and $|S_1| + |S_2| + \cdots + |S_i| = |S|$. A *graph* $G = (V, E)$ consists of a set of vertices V, also denoted by $V(G)$, and a set of edges E, also denoted by $E(G)$. A pair of vertices (u, v) is an edge in G if $(u, v) \in E(G)$. A graph is *simple* if there is

at most one edge between every pair of vertices, and there is no edge between a vertex and itself. In this thesis, all graphs will be simple, unless otherwise stated. Two vertices, u, v, are *neighbors* if $(u, v) \in E$. Equivalently, a vertex u is in the *neighborhood* of vertex v if u and v are neighbors. The *open neighborhood* of vertex v, denoted by $N(v)$, is the set of all vertices u such that u and v are neighbors. The *open neighborhood* of a set $S \subseteq V(G)$, denoted $N(S)$ is the set of all vertices $u \notin S$ such that such that $(u, v) \in V(G)$ for a vertex v in S. The *closed neighborhood* of v, denoted $N[v]$, is the open neighborhood of v plus v itself. Similarly the *closed neighborhood* of a set $S \subseteq V(G)$, denoted $N[S]$ is the open neighborhood of S plus the vertices of S. Given a set S, the *cardinality* of S, denoted $|S|$, is the number of elements in S. The *degree* of a vertex v, denoted $deg(v)$, in graph G is the cardinality of the open neighborhood of v. The *degree* of a set $S \subseteq V(G)$, denoted $deg(S)$, is the number of edges $(u, v) \in E(G)$ such that $u \in S$ and $v \in V(G) \setminus S$. Vertex v is *universal* in G if all vertices in $V(G)$ are in the closed neighborhood of v. A vertex v is *isolated* in G if for all $u \in V(G)$, $(u, v) \notin E(G)$. We define the degree of a vertex v with respect to a set of vertices S, $deg_S(v)$, to be the cardinality of $N(v) \cap S$.

A graph H is a *subgraph* of graph G if $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$. Given a set S of vertices, $G[S]$ denotes the *induced subgraph* of G *induced* by S. $G[S]$ is a graph consisting of the vertices of S and all edges $(u, v) \in E(G)$ such that both $u, v \in S$. For a given graph H, graph G is *H-free* if there does not exist any set $S \subseteq V(G)$ such that H is isomorphic to $G[S]$. A *disjoint union* of two graphs, G_1 and G_2, is a graph G such that $V(G)$ is the disjoint union of $V(G_1)$ and $V(G_2)$, and $E(G)$ is the disjoint union of $E(G_1)$ and $E(G_2)$.

A sequence of vertices $v_1, v_2, \ldots, v_{k-1}, v_k \in V(G)$ is a *path* of length $k-1$ in G if (v_1, v_2), (v_2, v_3), $\ldots$,$(v_{k-1}, v_k) \in E(G)$, and it is a *cycle* of size k if (v_k, v_1) is also in $E(G)$. C_k is the graph consisting only of a cycle of size k. P_k is the graph consisting of k vertices such that $E(P_k) = (v_1, v_2)$, (v_2, v_3), $\ldots$,(v_{k-1}, v_k). Graph G is *connected* if there is a path between every pair of vertices $u, v \in V(G)$. A *clique* in G is a set of vertices in G such that every pair of vertices in the set are neighbors. A *universal clique* C in G is a clique in G such that the closed neighborhood of C is $V(G)$. K_k is the graph consisting only of a clique of size k. A set of vertices $S \subseteq G$ is an *independent set* if $(u, v) \notin E(G)$ for all $v, u \in S$. A set of vertices $S \subset V(G)$ is a *vertex cover* if for all edges $(u, v) \in E(G)$, $u \in S$ or $v \in S$.

A clique K of graph G is *maximal* if there is no $v \in V(G) \setminus K$ such that $K \cup \{v\}$ is also a clique. A maximal clique K in graph G is *maximum* if there is no clique L in G such that $|L| > |K|$.

A *linear ordering* of the vertices of G is a bijective function $\sigma : V(G) \leftrightarrow \{1, 2, \ldots, |V(G)|\}$. A vertex v is to the *left* of vertex u if $\sigma(v) < \sigma(u)$. A vertex v is to the *right* of vertex u if $\sigma(v) > \sigma(u)$. We will sometimes also denote a linear ordering by $\sigma =< v_1, v_2, \ldots, v_n >$, meaning that $\sigma(v_i) = i$ for $1 \leq i \leq n$. The *cut* between two consecutive vertices v_i, v_{i+1} in σ is the set of edges with one endpoint in $\{v_1, \ldots, v_i\}$ and the other endpoint in $\{vi + 1, \ldots, v_n\}$. The size of the cut is the number of edges in it. Given an ordering σ of G, we define V_i to be the set of vertices $\{\sigma^{-1}(1), \sigma^{-1}(2), \ldots, \sigma^{-1}(i)\}$ i.e, the first i vertices in the ordering. We define the size of the cut between two close vertices, u and v where $\sigma(u) + 1 = \sigma(v)$, as $deg(V_{\sigma(u)})$. The *cutwidth* of an ordering σ of G, $cutwidth(\sigma, G)$, is $max_i(deg(V_i))$. A linear ordering σ of G is *optimal* if there is no linear ordering $\sigma*$ of G such that $cutwidth(\sigma*, G) < cutwidth(\sigma, G)$. The *cutwidth* of a graph G, $cutwidth(G)$, is $cutwidth(\sigma, G)$, where σ is an optimal linear ordering of G.

Unless stated otherwise, n refers to $|V(G)|$, the numbers of vertices in the input graph G. By m we mean $|E(G)|$, the number of edges in the input graph G, unless stated otherwise.

1.2 O-notation

How to measure the performance of algorithms? Measuring the time used running on a specific computer can be problematic. First of all, computers are different from each other, and some handle certain tasks better than others. The choice of computers used could be a huge discussion in itself. On what input should the algorithm be run on? Is the implementation optimal? There are many factors that come into play. Thankfully there is a simpler way to measure the performance of algorithms. Instead of actually taking the time spent on a computer, we express the maximum number of constant time operations used on an input of size n as a function $f(n)$. However, some algorithms might require less operations than others on certain inputs. To cope with this, a notation called *big O* is introduced. Given an algorithm A and a function

$f_A(n)$ describing the worst running time on input of size n, we say that $f_A(n)$ is $O(g(n))$ if there is a constant c and an integer k such that $f_A(n) \leq c \cdot g(n)$ for all $n \geq k$. We say that the running time of algorithm A is $O(g(n))$ if $f_A(n)$ is $O(g(n))$. If algorithm A uses at most $3 \cdot n^2 + 13 \cdot n + 100$ operations, we say that algorithm A runs in time $O(n^2)$, as for all $n > 1$, $1000 \cdot n^2$ is larger. An algorithm A is *efficient* if there is a polynomial function $f(n)$ such that the running time of A is $O(f(n))$. We say that $f(n)$ is *linear* if $f(n)$ is $O(n)$.

O-star-notation The O^**-notation* is similar to the big O; instead of just excluding the constant factors, all polynomial factors are also excluded. For example $O(2^n \cdot n^2)$ is $O^*(2^n)$. The O^*-notation becomes useful when comparing exponential functions.

Big Omega and Big Theta There are also two other notations related to the big O. A function $f(n)$ is big Omega of a function $g(n)$, $\Omega(g(n))$, if there exist a constant $c > 0$, such that $f(n) \geq c \cdot g(n)$ for all n larger than some constant. We say that a function $f(n)$ is big Theta of a function $g(n)$, $\Theta(g(n))$, if $f(n)$ is both $O(g(n))$ and $\Omega(g(n))$.

1.3 Complexity classes

Given two problems, A and B, is there any objective way to judge which one is the hardest? If it is possible to transform all instances of A into instances of B *efficiently*, such that a solution to B yields a solution to A, we say that A is *reducible* to B. Such transformations are called *reductions*. By efficiently, we mean that the time used to compute the reduction is polynomial in the input size of A. In this case, it is easy to see that B is at least as hard as A, since any efficient solution to B immediately gives an efficient solution to A. Consider all problems that can be solved by polynomial-time algorithms and put them into a class called P, for polynomial time. Then consider all problems that are *verifiable* in polynomial time, defined as follows. Given an instance of the problem, and a suggestion of a solution, if we can decide in polynomial time whether the suggestion is correct, then we say that the problem is polynomial-time verifiable. The class consisting of all problems verifiable in polynomial

time is called NP. Clearly all problems in P are also in NP, the verifier can just run the algorithm itself and compare the answer. Are all problems in NP also in P? The answer to this question is not known. This question is one of the seven *Millennium Prize Problems* given by Clay Mathematics Institute [1]. An answer to any of these problems grants a prize of one million US dollars. For more information on computational complexity we will refer to the book "Introduction to the Theory of Computation" by Michael Sipser [30].

Space complexity Some algorithms need to remember previously computed values and these values have to be stored somewhere. In the computer world this usually means some storage device as hard disk drives or random-access memory. Similar to time, the amount of storage used by an algorithm is described by some function $f(n)$. As with time, we usually express the space complexity in O-notation and O^*-notation.

NP-Completeness A problem is NP-*hard* if all problems in NP are reducible in polynomial time to this problem. Finding a polynomial time algorithm solving such a problem would thus imply P=NP. A problem is NP-*complete* if it is NP-hard and it is a member of NP. In 1971 Stephen Cook published a paper [11] which proved that any problem in NP could be reduced in polynomial time to the Satisfiability problem, commonly shortened to Sat. The Sat problem is also verifiable in polynomial time, thus it became the first problem to be known as NP-complete. From then on it became enough to give a polynomial reduction from Sat to show that a problem is NP-hard. Once a problem is shown to be NP-hard, the hope for finding a polynomial-time algorithm withers. However, there are ways to cope with NP-hardness, some of which are mentioned below:

Approximation In *optimization* problems one wants to minimize or maximize some parameter of the solution. We denote such maximum or minimum as *optimum* or OPT for short. Sometimes it is deemed too costly to compute an optimum solution, but a solution that is *good enough* is welcome. However, the notion of good enough is not precise, so a precise measure is needed. This is where *approximation* algorithms come into play. An approximation algorithm has a guarantee that it will never be further away from the optimum than

some r, which can be a constant or a function dependent on the input. We say that an algorithm is r-approximate if it always gives solutions that are less than or equal to $r \cdot OPT$ on minimization problems, and more than or equal to $\frac{1}{r} \cdot OPT$ on maximization problems. A classical example on approximation is the 2-approximation-algorithm for the Minimum Vertex Cover problem which asks to find a vertex cover of smallest cardinality. The approximation algorithm picks an edge none of whose endpoints are yet in the vertex cover, and adds both endpoints to the vertex cover. Since any solution must have at least one of the endpoints of an edge, this algorithm will give a solution that is at most twice the size of the optimum on any graph. For more information on approximation we will refer to the book "Complexity and Approximation" [2].

Heuristics *Heuristics* are algorithms that give good and/or fast solutions in practice, but an approximation bound does not exist or is not yet known. Heuristics may perform bad on particular types of input, but in real world applications these situations are not likely to occur or does not even happen at all. An example of this are algorithms that solve problems on road networks (see Figure 1.2). First of all a road network is rather sparse; there are none or few vertices of high degree. Secondly the roads are located in euclidean space, so for finding shortest paths it is usually a good idea to go towards the destination spatially, thus techniques as is seen in the A*-algorithm [20] can reduce the running time in practice.

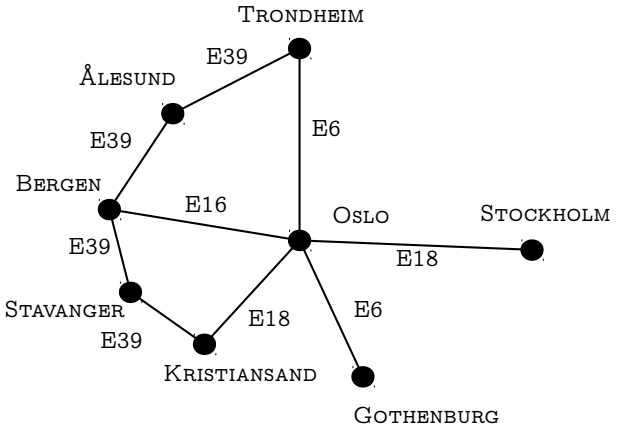

Figure 1.2: Selected highways in Norway.

Restricted Input Instead of an algorithm that works on general input, we can instead limit the allowed input. For example, restricting the input to graphs with certain structural properties. This can be shown useful in the real world as many problems properties that might make the problem easier to solve. For example a graph representation of a river would in most cases be directed and acyclic and one might be able exploit this fact to solve problems related to rivers faster.

Faster exact algorithms If we require the solution of an NP-Hard problem to be general and exact, then there is no polynomial-time algorithm unless P=NP. But there is still room for improvement. A naive algorithm for the Minimum Vertex Cover problem is to try all subset of vertices, which takes $O^*(2^n)$ time. A clever algorithm by Robson [29] has a running time of $O^*(1.18882^n)$. Disregarding polynomial factors the difference in maximum input size is linear. Looking at the O-notation alone and disregarding polynomial factors we can see that the naive algorithm for Minimum Vertex Cover can compute on 30 vertices in around one billion operations, while the faster algorithm could compute around 120 vertices in the same amount of operations. While one could argue that this improvement is small, another interesting look is how many years of hardware development needed to achieve the improvement of the new algorithm using the old one. Let us assume that the average computer today can do one billion operations per second, and an optimistic assumption that the number of operations per second will double every second year. It would take 180 years until we could compute for 120 vertices using the naive algorithm. This is a poor scientific measure because the difference in number of operations depends on the parameters we measure within, in particular the amount of operations currently done in a given timeframe. A better measure would be how many times more vertices the clever algorithm can process in the same timeframe as the naive algorithm. In this case the clever algorithm can process roughly 4 times as many vertices as the naive algorithm in the same timeframe. This can be seen as $\frac{log(2)}{log(1.1882)} \approx 4.02$. An important milestone in the field of exact algorithms is to find an $O^*((2-\epsilon)^n)$ algorithm, where $\epsilon > 0$, an algorithm with running time faster than $O^*(2^n)$. This milestone is not reached on Cutwidth, and it remains unknown whether or not an algorithm faster than $O^*(2^n)$ time exists.

Fixed parameter algorithms For many NP-complete decision problems that ask whether a given graph on n vertices has a structure (like vertex cover) of size k, we can restrict the exponential part of the running time to k and obtain an algorithm that with running time $O(f(k) \cdot poly(n))$, where f is typically an exponential function. In this case we say that the problem is fixed parameter tractable, often shortened to FPT. If k is sufficiently small with regards to f, we clearly get many cases that are solvable in practice. We refer to "Parameterized Complexity" by Downey and Fellows [15] for more depth on this subject.

1.4 Graph classes

It has been shown useful to classify graphs by structural properties. Some problems may be NP-hard in general, but have polynomial-time algorithms on graphs with certain structural properties. In this section a look at some well-known graph classes will be taken. For more depth on this subject we will refer to two classical books on graph classes: "Algorithmic graph theory and perfect graphs" by Golumbic [19] and "Graph Classes: A survey" by Brandstädt, Le, and Spinrad [7].

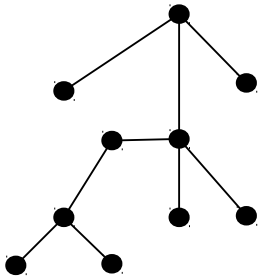

FIGURE 1.3: A tree.

Trees Trees form the class of connected graphs that contain no cycles. A disjoint union of trees is called a *forest*. Trees form an easy class to work with; many problems that are NP-hard in general admit polynomial-time algorithms on trees. A graph is a tree if between any two vertices there is exactly one path. Figure 1.3 is an example of a tree.

Chordal graphs A graph is chordal if it does not have C_k as an induced subgraph for $k > 3$. Clearly, from this definition, the class of chordal graphs is

a superclass of trees. Figure 1.4 shows a graph that is not chordal and a graph that is chordal.

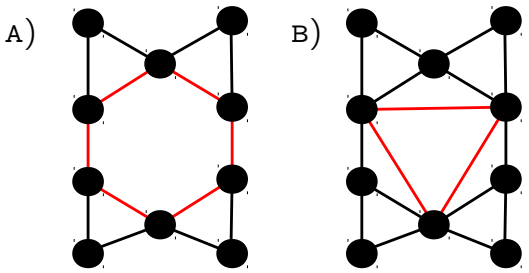

FIGURE 1.4: Left: A graph containing C_6. Right: Chordal graph.

Split graphs The graphs that can be partitioned exactly into one clique and one independent set form the class split graphs. All split graphs are chordal graphs, as split graphs clearly cannot have any cycles of length more than three.

Interval graphs A graph is an interval graph if there is an assignment of its vertices $v_1, v_2, \ldots, v_n$ to intervals $I_1, I_2, \ldots, I_n$ of the real line such that:
1. every vertex v_i is mapped to exactly one continuous interval I_i, and
2. there is an edge between v_i and v_j if and only if $I_i \cap I_j \neq \emptyset$.
Interval graphs do not have C_k as induced subgraphs for $k >$, and therefore all interval graphs are also chordal graphs. An example of an interval representation and its corresponding graph can be seen in Figure 1.5.

Proper interval graphs Proper interval graphs form the class of graphs that have an interval model where no interval is contained in an other interval (see Figure 1.6). We say that an interval $[a, b]$ is *contained* in interval $[c, d]$ if $c < a$ and $b < d$. This class is equivalent to *unit interval graphs* [28], where every interval is of the same length.

Trivially perfect graphs Trivially perfect graphs form the class of all interval graphs that have an interval model with no overlapping intervals. Two intervals $[a, b]$ and $[c, d]$ *overlap* if $a < c \leq b < d$, or $c < a \leq d < b$. Another name for trivially perfect graphs is *nested interval graphs*, as the intervals are

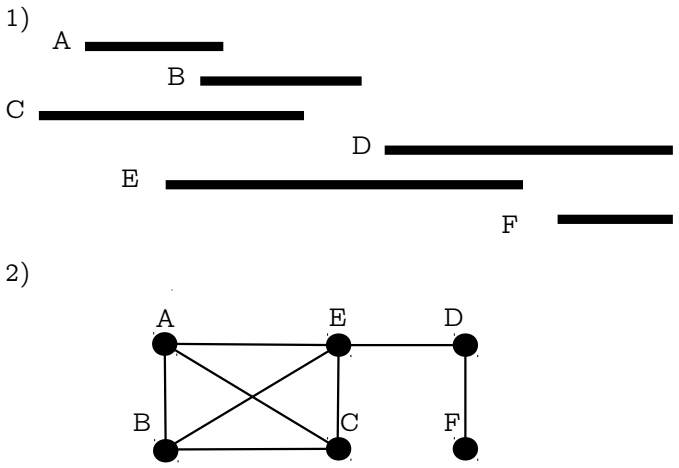

FIGURE 1.5: An Interval representation and its corresponding graph.

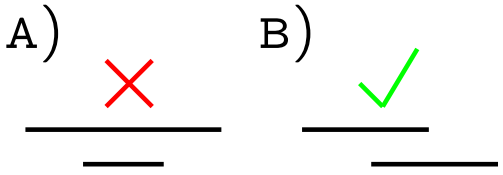

FIGURE 1.6: A) An interval contained in an other interval. B) Overlapping intervals.

"nested" in each other. The disjoint union of trivially perfect graphs is trivially perfect. Adding a universal vertex to a trivially perfect graph creates a new trivially perfect graph. Every connected trivially perfect graph can be represented by a rooted tree, where every path from the root to a leaf represents a maximal clique. We will call this representation a *tree representation*. This can be useful as a tree representation uses $O(n)$ space while the standard graph representation uses $O(n+m)$ space. Figure 1.7 shows a trivially perfect graph, its interval representation and its tree representation.

Threshold graphs Threshold graphs are graphs that can be constructed by repeatedly adding either an isolated vertex or a universal vertex (see Figure 1.8). In other words, there is an order of the vertices $v_1, v_2, \ldots, v_n$ such that v_i is either universal or isolated in the graph induced by $\{v_1, v_2, \ldots, v_i\}$. We call this *the construction order* of a threshold graphs. Threshold graphs form a subset of interval graphs. Given $v_1, v_2, \cdots, v_n$, the construction order of the threshold graph, an interval graph can be constructed as follows: v_j is

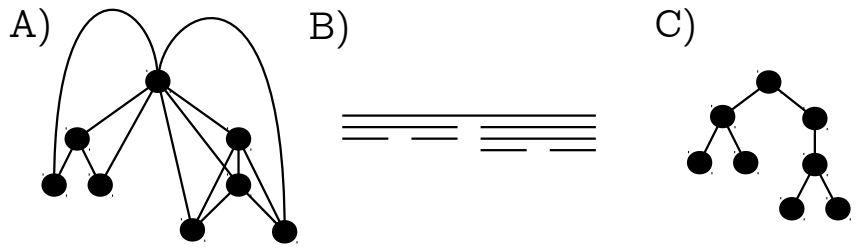

FIGURE 1.7: A) A trivially perfect graph. B) Its interval representation. C) Its tree representation.

assigned the interval $[2 \cdot j - 1, 2 \cdot j]$ if it is isolated or the interval $[1, 2 \cdot j]$ if it is universal. By this construction it is clear that all threshold graphs are trivially perfect graphs. Threshold graphs are also split graphs, the universal vertices form a clique while the isolated vertices form an independent set. The tree representation of connected threshold graphs takes the form of a *caterpillar*. To be more precise, the tree representation has a path $P = v_1, \ldots v_k$, where v_1 is the root, such that $N[P] = V(G)$.

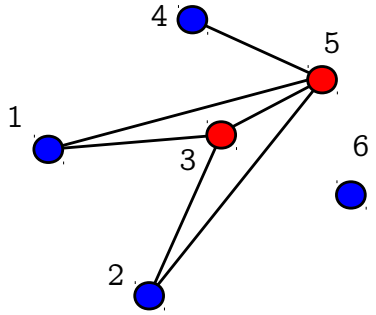

FIGURE 1.8: Example of a threshold graph. Blue vertices are isolated at the time they are added, red vertices are universal, and the order is represented by the numbers.

Superfragile graphs Superfragile graphs form the class of graphs that can be constructed with two operations:

1. Adding a universal clique to a disjoint union of cliques.
2. The disjoint union of superfragile graphs.

Another definition of superfragile graphs is through their forbidden subgraphs: Superfragile graphs are exactly those graphs which contain no *Dart*, C_4 or P_4 as an induced subgraph (see Figure 1.9). Superfragile graphs are also trivially perfect graphs: cliques are clearly trivially perfect graphs, a trivially perfect

graph plus a universal vertex is also trivially perfect, and a disjoint union of trivially perfect graphs is also trivially perfect. In the tree representation of a connected superfragile graph there can be two cases: either the root vertex has degree one and there is at most one vertex with degree larger than two, or the root vertex has degree larger than one and no other vertex has degree larger than two.

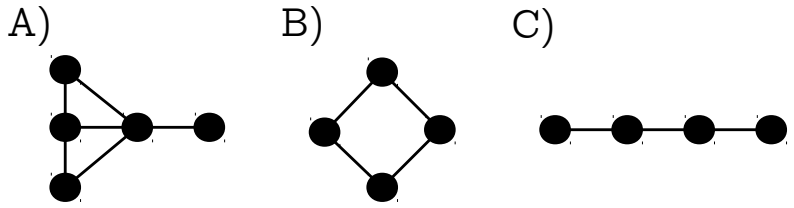

FIGURE 1.9: Forbidden subgraphs of superfragile graphs. A) *Dart*, B) C_4 and C) P_4.

Compact tree representation We want to give a new representation of trivially perfect graphs which can be useful for classifying graphs. Notice that the tree representation of a superfragile graph can only have one vertex with degree more than two. This means that the tree representation of a superfragile graph often consists of several longer paths without any branches. Surely, it must be possible to compress the representation of this path, as all the vertices have the same closed neighborhood. Therefore we will introduce a representation such that no such path of length greater than or equal to 2 exists. We define a *compact tree representation* of a trivially perfect graph G as a rooted tree of *bags* where every bag represents a set of vertices in G of at least size 1 and every vertex is in one and only one bag, and the vertices contained in the bags in a path from the root to a leaf forms a maximal clique. There are no edges between vertex u and v if there is no path from the root to a leaf where both u and v is contained. Looking back to the definition of superfragile graphs, connected superfragile graphs are those graphs that have a compact tree representation of depth 1. Figure 1.10 gives examples of compact tree representations.

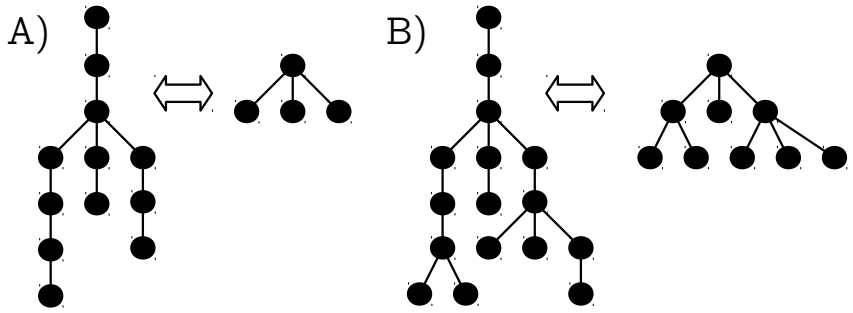

FIGURE 1.10: A) A tree representation of a superfragile graph and a compact tree representation of the same graph, with depth 1. B) A tree representation of a trivially perfect graph and its compact tree representation of depth 2.

1.5 Results on Cutwidth

The main focus of this thesis will be on the CUTWIDTH problem. The problem is defined as follows: given a graph G and an integer k, is there a linear ordering σ of G such that the maximum number of edges crossing a vertical line between any two consecutive vertices in σ is less than or equal to k? CUTWIDTH appeared in 1967 in the paper "Large Scale Integration of MOS Complex Logic: A Layout Method" by Weinberger [32], as a subroutine in a circuit design method. CUTWIDTH has found applications within circuit design [9][32], network reliability [24] and protein engineering [4]. Regarding the graph classes introduced in Section 1.4, CUTWIDTH is known to be NP-complete on chordal graphs [21] through an NP-completeness-proof on *split graphs*. Split graphs are a subclass of chordal graphs, and therefore NP-completeness on them implies NP-completeness on chordal graphs, too. To be more general: if a problem is found NP-hard on a certain graph class, then it is NP-complete on all superclasses of this graph class. Similarly, if a polynomial-time algorithm is found on a certain graph class, the same algorithm holds for any subclass of this graph class, and thus the problem is also in P on the subclasses. Among the classes from Section 1.4, polynomial-time algorithms for CUTWIDTH has been found for trees [33], threshold graphs [21] and proper interval graphs [21]. It can also be solved in polynomial time on bipartite permutation graphs [22]. It remains unknown whether or not CUTWIDTH is NP-hard on interval graphs and trivially perfect graphs. See Figure 1.11 for an illustration of the relations between the aforementioned classes and their memberships.

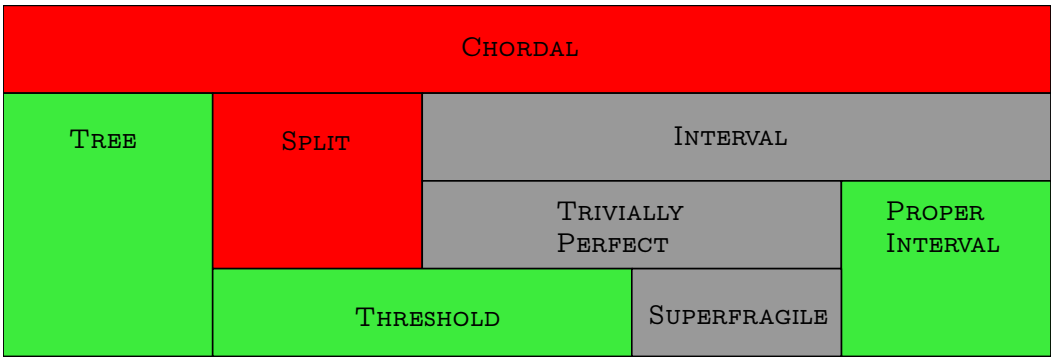

Figure 1.11: Cutwidth on selected graph classes. Red indicates NP-complete, Green indicates membership in P, Grey is unknown. A class is a subclass of another class, if it is below the other and there is a vertical line intersecting both.

Cutwidth was shown to be in FPT in 1988 by Fellows and Langston [16]. An algorithm running in linear time for constant cutwidth was given by Thilikos, Serna and Bodlaender in 2005 [31]. An $O(log^2(n))$ approximation algorithm was presented by Leighton and Rao at FOCS in 1988 [25].

There are not many non-trivial graph classes on which Cutwidth is known to be solvable in polynomial time. As already mentioned, the computational complexity is open on interval graphs, and even on trivially perfect graphs. In fact, at a very recent Dagstuhl seminar [6], it was conjectured to be NP-complete even on a simple subclass of trivially perfect graphs. This motivates our exploration of Cutwidth on superfragile graphs, another subclass of trivially perfect graphs, for which the computational complexity is open.

1.6 Overview of the thesis

In this thesis we will focus on algorithms solving the Cutwidth problem. We will both design exponential-time algorithms for general graphs, and give polynomial-time algorithms solving Cutwidth on superfragile graphs.

Chapter 2: Dynamic programming and Divide-and-Conquer on Cutwidth A naive solution to the Cutwidth problem is to try all possible linear orderings, which gives an algorithm with running time $O^*(n!)$. In Chapter 2 we show a faster algorithm that solves Cutwidth using $O^*(2^n)$ time and $O^*(2^n)$

space using *dynamic programming* techniques. Later in this chapter we will also trade time with space and produce a polynomial-space algorithm with a running time of $O^*(4^n)$.

Chapter 3: A look into the threshold-algorithm The first objective of this thesis was to look at the linear-time algorithm for Cutwidth on threshold graphs [21] and the techniques used by this algorithm. In particular, the algorithm is easily applicable on any type of graph, since it does not rely on the structure of threshold graphs for its execution. For this reason, the authors asked whether or not it could be useful as an approximation algorithm on some graph classes. In Chapter 3 we explore if these techniques are applicable for constructing approximation algorithms and heuristic algorithms on other graph classes.

Chapter 4: Cutwidth on Superfragile graphs In Chapter 4 we will take a look on superfragile graphs, and show a polynomial-time algorithm solving the Cutwidth problem on superfragile graphs. In fact Chapter 4 contains the main scientific contribution of this thesis. As we have mentioned earlier the computational complexity of Cutwidth on superfragile graphs has been open until the results that we present in Chapter 4.

Chapter 5: Conclusion and further research We conclude with a summary of the work presented in earlier. We introduce some new linear ordering problems, and ask open questions related to these problems as well as the Cutwidth problem.

Chapter 2

Dynamic Programming and Divide-and-Conquer on Cutwidth

An easy way to solve the CUTWIDTH problem is to go through every possible linear ordering of the graph and find out which one of them yields the lowest cutwidth. There are a total of $n!$ different orderings and for every one of them the cutwidth must be calculated, which takes $O(n + m)$ time. Thus this naive algorithm has a running time of $O(n! \cdot (n + m))$. In this Chapter we will present two algorithms with faster running time using two different methods, dynamic programming and divide-and-conquer. The existence of algorithms for cutwidth with these running times have been known [3] [23] [5], but developing these on our own and working out all details is a good exercise to get a better understanding of the CUTWIDTH problem and its challenges.

2.1 Dynamic Programming

On a sunny Saturday evening Bob's more eccentric friend Alfred knocks at his door. Alfred is a "Master Wizard" of the local "High Intelligence Club", but he often consults Bob, a programmer, for help. This time Alfred brought a knotted string and a short ruler that is at least twenty times shorter than the string. "I am to find the length between the two outermost knots of this string, but it is impossible, this ruler is way too short," Alfred tells Bob. Bob thinks for a second and says, "I have two ways to find the length, I will show you

in a bit." Alfred is a bit puzzled, Bob did not even pass the entry test for the club and now he claims to have solved one of the seven "Grand problems" that have remained unsolved since the founding of the club. Bob asks Alfred to fetch him pen and paper, and starts measuring from the first knot to the second knot with the short ruler. "I noticed that the space between every knot is no longer than the length of this ruler," Bob says while Alfred handed him a pen and some paper. First Bob writes down: *Knot one to two: 15.3cm*, he then measures the length between the second knot and the third, the ruler shows 5.6cm, *Knot one to three: 15.3cm + 5.6cm = 20.9cm* he writes on the paper. He continues measuring, writing down the length between the previous knot and the current knot plus the previously found length, until he reaches the last knot. The last thing he writes is *Knot one to thirty-seven: 216.3cm*. "Here is your answer, two-hundred and sixteen point three centimeters," Bob tells Alfred. Alfred is curious and asks "How did you manage that?" and adds "You are not even a member of the club," in an arrogant manner. "Ahh, it is just an old programmer trick... dynamic programming we call it," Bob answers while smiling proudly.

Dynamic programming is a method of solving problems on larger instances through the solutions of smaller instances. The idea is that we can remember a solution of previously solved instances and use it later to obtain solutions for larger instances. Dynamic programming usually involves a recursive solution formulation. Then the idea is to find an appropriate order in which the smaller subproblems can be solved, so that their solution are calculated and stored in a table. Then in larger instances, rather than using recursion, we simply look up solutions of smaller instances in the table. But there is a trade off, sometimes we have to store a vast amount of solutions and thus getting a worse space complexity than alternative methods. We will now give an algorithm using dynamic programming to solve Cutwidth on general graphs:

Preliminaries Let G be the input graph, and let k be the input integer. Given a set $S \subseteq V(G)$, we call a linear ordering σ an *S-minimal* ordering if $\sigma(v) \leq |S|$ for all vertices v in S, and the size of the maximum cut between two consecutive vertices of S in σ is minimized. We define the *cut* of a set $S \subseteq V(G)$ in G, $cut(S)$, to be the size of the largest cut between two consecutive vertices of S in an S-minimal ordering. We define the *cutwidth* of a set $S \subseteq V(G)$ in G to

be $max\{deg(S), cut(S)\}$. Clearly the cutwidth of G is equal to the cutwidth of $V(G)$, and thus an $V(G)$-minimal ordering is also an optimal linear ordering of G. Figure 2.1 A shows that an optimal linear ordering is not necessarily S-minimal for every set S.

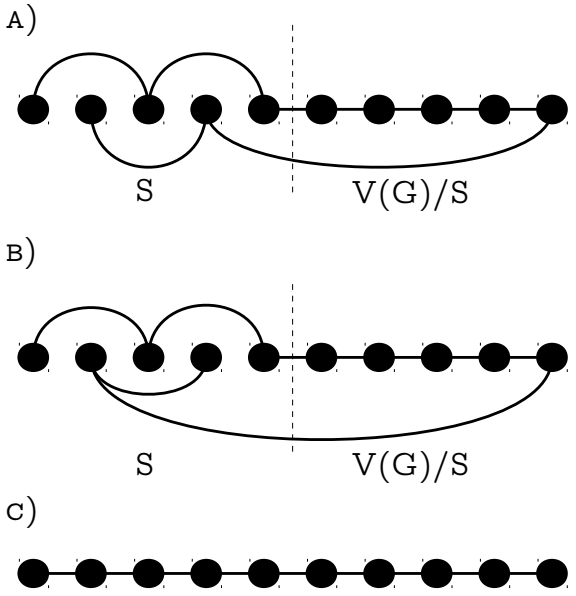

FIGURE 2.1: A) An S-minimal ordering, where the largest cut between two consecutive vertices in S is 2. B) An ordering that is not S-minimal, as the largest cut between two consecutive vertices in S is 3. C) A linear ordering that is $V(G)$-minimal, and thus also optimal.

The Algorithm Let G be the input graph, and let k be the input integer. We attack CUTWIDTH by calculating the cutwidth of a subset $S \subseteq V(G)$. Assume for the moment that the cutwidth of S is the maximum of the degree of S and $min_{S' \subset S} cutwidth(S')$ where $|S'| = |S| - 1$. We will prove this fact when proving the correctness of our algorithm on the next page. We start by initializing the cutwidth of $\emptyset$ to 0. Then find the cutwidth of all subsets of cardinality one by using the result of the set of cardinality zero, then all of cardinality two by using the results from the sets of cardinality one. The algorithm continues until the cutwidth of the set containing all vertices, $V(G)$, has been calculated. If the cutwidth of $V(G)$ is less than or equal to k, the algorithm return YES, if not, the algorithm will return NO. There are in total 2^n subsets S. For each S the algorithm first checks all $S' \subset S$ such that $|S'| = |S| - 1$, for which there

are $O(n)$ of, then calculate the degree of S which takes $O(n+m)$ time. Thus the algorithm runs in $O(2^n \cdot (n+m))$ time. For every $S \subseteq V(G)$ the cutwidth of S has to be stored, thus $O(2^n)$ space is used by the algorithm.

Algorithm 1: CutwidthDP

Input: A graph $G = (V, E)$ and an integer k
Output: YES if G has cutwidth $\leq k$, NO otherwise

```
1 CW[∅] = 0;
2 for i = 1 to |V(G)| do
3 |   for all S ⊆ V(G) such that |S| = i do
4 |   |   CW[S] = max(min_{v∈S}(CW[S \ {v}]), deg(S));
5 if CW[V(G)] ≤ k then
6 |   return YES;
7 else
8 |   return NO;
```

Proof of Correctness The algorithm considers all subsets S of $V(G)$ and computes the cutwidth of S for each such subset. By the definition of cutwidth of of a set, the cutwidth of $S = V(G)$ gives the cutwidth of G at the end. To prove the correctness of the algorithm, we need to prove what we claimed in the description of the algorithm, namely that

$$cutwidth(S) = max \begin{Bmatrix} deg(S) \\ min\{cutwidth(S') \mid S' \subset S, |S'| = |S| - 1\} \end{Bmatrix} \tag{2.1}$$

Let $R = V(G) \setminus S$. Let σ be an ordering in which vertices of S are ordered before vertices of R. Observe that the local ordering of the vertices within R is irrelevant for $cutwidth(S)$. Hence to prove the above claim we only need to prove that $cut(S) = min\{cutwidth(S') | S' \subset S, |S'| = |S| - 1\}$. Assume that a vertex v is the rightmost vertex in an S-minimal ordering ($\sigma(v) = |S|$), under the previously mentioned conditions. An S'-minimal ordering, where $S' = S \setminus \{v\}$, with v added to the end (such that $\sigma(v) = |S|$), will also be an S-minimal ordering (see Figure 2.2). The algorithm does not know which vertex is the rightmost in an S-minimal ordering, so it tries all $|S|$ vertices in S. Since all possibilities are covered, an S-minimal ordering is found, given that we already know the minimal ordering for all $S' \subset S$ with $|S'| = |S| - 1$, and under the assumption that the optimal linear layout orders S before the rest. It

follows that when the algorithm examines $S = V(G)$ it finds the cutwidth of G.

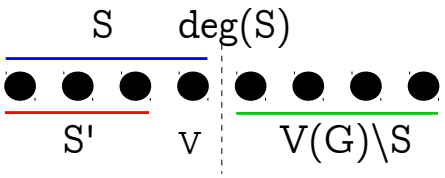

FIGURE 2.2: Given that an S-minimal ordering has v as the rightmost vertex in S, the cutwidth of S is $max(cutwidth(S'), deg(S))$, where $S' = S \setminus \{v\}$.

Even faster algorithm We can modify this algorithm to run even faster. Computing the degree of $S \subseteq V(G)$ in G takes $O(n+m)$ time, but we can make it take $O(n)$ time. If we for every $S \subseteq V(G)$ already know the degree of some $S' \subset S$, $|S'| = |S| - 1$, we can easily compute the degree of S. Let $S' \cup \{v\}$ be equal to S, then $deg(S) = deg(S') - deg_{S'}(v) + deg_{V(G)\setminus S}(v)$. This can be computed in $O(n)$ time as v can have at most $n-1$ neighbors. Storing the degree for every subset of $V(G)$ requires $O(2^n)$ space. Thus we get an $O(2^n \cdot n)$ time algorithm using $O(2^n)$ space.

With all the results of this section, we can now conclude the following:

Theorem 1. *Cutwidth of an arbitrary graph on n vertices can be computed in $O(2^n \cdot n)$ time and $O(2^n)$ space.*

2.2 Divide and Conquer

"You said you had two ways to find the solution. Do you mind showing me the second one?" Alfred asked Bob, humbly for once. "Oh, yes I can... another programmer trick. Divide and Conquer we call it, let me show you." "You see, the length from knot one to knot nineteen plus the length from knot nineteen to knot thirty-seven," Bob explained. Alfred gave it some seconds of thought

and said "Uhm, yeah... that might work." "Alright then, we call this step Divide. Now, that we have divided it into two parts, we have to Conquer these two parts. However, our ruler is not long enough to measure them, do you have any idea what we should do?" Bob asked rhetorically. "Can we divide again?" Alfred answered. "Yes! We find the length between knot one to knot nineteen by adding the length from knot one to knot ten to the length from knot ten to knot nineteen," Bob answered, took a long breath and added "We continue dividing until we reach a trivial problem. Like what is the distance between two consecutive knots, or even: what is the distance between knot i and knot i? After we are done dividing, we start conquering. Elementary!"

Divide and Conquer is a useful method on problems that can be partitioned into independent sub-problems recursively until a simple problem-instance is reached. Divide-and-conquer-algorithms are especially useful within parallel computing as they might often provide a straight-forward parallelization. The divide and conquer method often provides more space-efficient algorithms when compared to dynamic programming. We will now give a divide-and-conquer-algorithm for the CUTWIDTH problem, running in $O(4^n \cdot n^2)$ time and $O(n \cdot log(n))$ space.

Preliminaries Let G be the input graph, and let k be the input integer. Given a partition of $V(G)$ into three sets L, R, S, we call a linear ordering σ an L, R, S-*minimal* ordering if for all $u \in L$, $v \in S$, $w \in R$, we have that $\sigma(u) < \sigma(v) < \sigma(w)$, and the size of the maximum cut between two consecutive vertices of S in σ is minimized under this restriction. We define the *cut* of three sets, $L, R, S \subseteq V(G)$, $L \cap R = L \cap S = R \cap S = \emptyset$, denoted $cut(L, R, S)$ as the maximum of the degree of L, the degree of R (see Figure 2.3), and the size of the largest cut between two consecutive vertices of S in an L, R, S-minimal ordering.

The Algorithm Let G be the input graph, and let k be the input integer. The main algorithm starts with a call $(G, \emptyset, \emptyset, V(G))$ to a recursive method. This method takes as input (G, L, R, S) and it computes $cut(L, R, S)$. In particular, it recursively tries all subsets S' of S, such that $|S'| = \lfloor \frac{|S|}{2} \rfloor$ and the maximum of $cut(L \cup S', R, S \setminus S')$ and $cut(L, R \cup S \setminus S', S')$ is minimized (see Figure 2.4). When the base case is reached, $|S| \leq 1$, the maximum of the degree of $S \cup L$ and

the degree of $S \cup R$ is returned. When the recursive algorithm obtains the value of $cut(\emptyset, \emptyset, V(G))$, the cutwidth of the graph G, it returns the value to the main algorithm. The main algorithm returns YES if this value, the cutwidth of G, is less than or equal to k, and NO otherwise. The algorithm starts with the set $V(G)$, of size n, and branches into two new instances for all subsets of size $\frac{n}{2}$, for which there are $O(2^n)$ of. This gives a total of $O(2^n) \cdot 2 \cdot O(2^{\frac{n}{2}}) \cdot 2 \cdot O(2^{\frac{n}{4}}) \ldots = O(2^n) \cdot \prod_{i=1}^{log(n)} 2 \cdot O(2^{\frac{n}{2^i}}) = O(4^n \cdot n)$ instances. For each of these instances $O(n+m)$ time is spent on calculating the degrees, thus the algorithm has a total running time of $O(4^n \cdot n \cdot (n+m))$. Regarding space, the algorithm uses $O(n)$ space on each level of recursion, and there is maximum $O(log(n))$ levels of recursion, so the algorithm uses $O(n \cdot log(n))$ space.

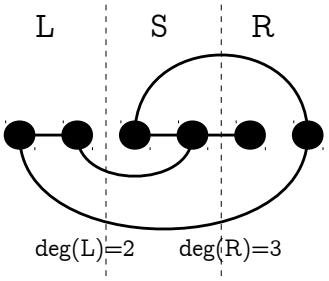

FIGURE 2.3: Illustration of $deg(L)$ and $deg(R)$, where $deg(L)$ is the size of the cut between L and S, and $deg(R)$ is the size of the cut between S and R.

Algorithm 2: DivideAndConquer

Input: A graph $G = (V, E)$ and an integer k
Output: YES if G has cutwidth $\leq k$, NO otherwise

```
1 if Recursive(G, ∅, ∅, V(G)) ≤ k then
2 |   return YES;
3 else
4 |   return NO;
```

Proof of Correctness Let G be the input graph, and let k be the input integer. Given a partition of $V(G)$ into L, R, S, and a linear order σ of G such that for all $u \in L$, $v \in S$, $w \in R$, $\sigma(u) < \sigma(v) < \sigma(w)$, we want to prove that the recursive method correctly computes $cut(L, R, S)$. Observe that the local ordering of vertices within L and within R are irrelevant for $cut(L, R, S)$. Clearly, there is a partition of S into S' and S'' such that $cut(L, R, S) = max(cut(L \cup$

Algorithm 3: Recursive

Input: A graph $G = (V, E)$ and a partition of $V(G)$ into L, R, S.
Output: $cut(L, R, S)$

```
1 cut = 0;
2 if |S| > 1 then
3 |   cut = ∞;
4 |   for all S' ⊆ S such that |S'| = ⌊|S|/2⌋ do
5 |   |   cut = min(cut, max(Recursive(G, L ∪ S', R, S \ S'),
6 |   |       Recursive(G, L, R ∪ S \ S', S')));
7 cut = max(cut, deg(L), deg(R));
8 return cut;
```

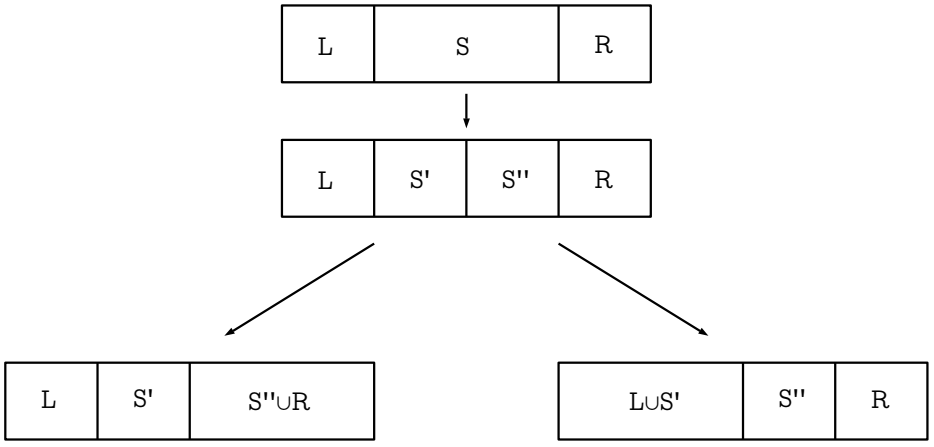

FIGURE 2.4: The recursive call performed by the Divide-and-Conquer algorithm.

$S', R, S''), cut(L, R \cup S'', S'))$. To see this, let σ be an L, R, S-minimal ordering and let i be the position of the largest cut in S. If $|L \cup S'| < i$ then $cut(L, R, S) = cut(L \cup S', R, S'')$ and otherwise $cut(L, R, S) = cut(L, R \cup S'', S')$. Thus by trying all subsets of S, we get the correct $cut(L, R, S)$. Observe that it is enough to check subsets S' with $|S'| = \lfloor \frac{S}{2} \rfloor$, since either $|L \cup S'| < i$ or $|L \cup S'| \geq i$ for these.

By the above discussion, we can conclude the following:

Theorem 2. *Cutwidth of an arbitrary graph on n vertices can be computed in time $O(4^n \cdot n \cdot (n + m))$ and space $O(n \cdot log(n))$.*

Chapter 3

A look into the threshold algorithm

In 2008, Heggernes, Lokshtanov, Mihai and Papadopoulos gave an algorithm for solving Cutwidth on threshold graphs in linear time [21] and thus showing that Cutwidth on threshold graphs is in P. The authors asked whether or not this algorithm, with some modifications, could be used as an approximation algorithm for Cutwidth on other graph classes. The first task of this thesis was to search for an answer to this question. We will later in this chapter give an answer to this question, but let us first take a look at the algorithm for threshold graphs:

3.1 Cutwidth on threshold graphs

Let G be a threshold graph, and let σ be a linear ordering minimizing cutwidth, which we want to compute. This ordering will be created incrementally. Before step i, the first $i-1$ vertices of the resulting ordering will be ready. At step i, the algorithm decides which vertex will be the i'th vertex of the resulting ordering.

Preliminaries We define a set of vertices V_i to be the first i vertices in the computed linear ordering. Given a set of vertices $S \subseteq V(G)$ we define the rank of a vertex v with respect to S, $rank_S(v)$, to be the degree of v with respect to $V(G) \setminus S$ minus the degree of v with respect to S, i.e. $rank_S(v) = deg_{V(G)\setminus S}(v) - deg_S(v)$ (see Figure 3.1).

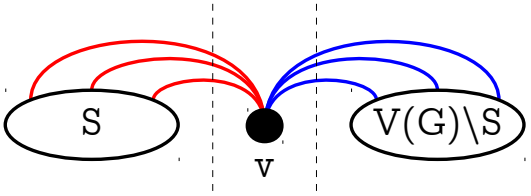

FIGURE 3.1: The rank of v with respect to S is $deg_{V(G)\setminus S}(v) - deg_S(v)$ (number of blue edges minus number of red edges).

The algorithm The algorithm goes through $|V(G)| = n$ steps. At step i the algorithm picks vertex $v \notin V_{i-1}$ such that $rank_{V_{i-1}}(v)$ is minimized. The algorithm then sets $V_i = V_{i-1} \cup \{v\}$ and $\sigma(v) = i$. If several vertices have the same rank with respect to V_{i-1}, then the algorithm picks the one with the largest degree in G. This means that the algorithm picks the vertex v minimizing the degree of $V_{i-1} \cup \{v\}$, and when there is a tie, the vertex v reducing the rank of most vertices with respect to $V_{i-1} \cup \{v\}$ is picked.

Algorithm 4: MinCut

Input: A graph $G = (V, E)$
Output: A linear ordering $\sigma =< v_1, v_2, \ldots, v_n >$ of G

```
1  V_0 = ∅;
2  for i = 1 to n do
3      v_i = an arbitrary vertex in V \ V_{i-1} ;
4      for every vertex v in V \ V_{i-1} do
5          if rank_{V_{i-1}}(v) < rank_{V_{i-1}}(v_i) then
6              v_i = v;
7          else
8              if rank_{V_{i-1}}(v) == rank_{V_{i-1}}(v_i) and deg(v) > deg(v_i) then
9                  v_i = v;
10     V_i = V_{i-1} ∪ {v_i};
11     σ(v_i) = i;
12 return σ;
```

3.2 Threshold algorithm on other graph classes

The threshold algorithm uses the rank and the degree of vertices to choose the next vertex. Rank and degree are defined for all graphs and thus the algorithm is applicable on all graph classes. Note that if a graph is disconnected, then there is always an optimal ordering that keeps the connected components separate. Hence we can without loss of generality consider connected graphs. The threshold algorithm, without modifications, could in fact result in an ordering where the connected components are interleaved. Observe that a threshold graph can only have one component that is not a vertex of degree zero, and thus separating components in a linear ordering makes no difference on the cutwidth. In the algorithm they will obviously get picked first, as their, and only their, rank is zero. Thus, when applying the algorithm on other graph classes, we believe it is fair to only consider connected graphs, as we can trivially reduce a disconnected instance into several connected ones.

Split graphs

An interesting graph class to test the threshold algorithm on is split graphs. Figure 3.2 shows that the threshold algorithm is not optimal. As mentioned, Cutwidth is NP-complete on split graphs, thus a good approximation algorithm would be welcome. However, we are able to construct an example where the algorithm performs poorly. Let $k \geq 1$ be an integer, and let us construct a split graph with a clique of size $3k$. For every vertex in the clique, add $k^2 + 3k - 3$ independent vertices and construct an edge between each of them and the corresponding clique vertex. Furthermore, we partition the vertices of the clique into k sets of size 3. For each of these sets we add k^2 vertices, and connect them with edges to the three vertices in the set. The graph now has an independent set of size $3k \cdot (k^2 + 3k - 3) + k^3$ and a clique of size $3k$, a total of $\Theta(k^3)$ vertices. Observe that we have $\Theta(k^3)$ vertices of rank 1 and $\Theta(k^3)$ vertices of rank 3. We have carefully constructed the graph such that picking all vertices of rank 1 first would leave every vertex in the clique at rank 2. The structure of these graphs is shown in Figure 3.3.

On this graph the threshold algorithm will pick all vertices of rank 1 first, giving a cut of size $\Theta(k^3)$. Continuing the algorithm would pick all the vertices

in the clique and then the rest of the independent set. This will also yield a cut of size $\Theta(k^3)$. Clearly, any labeling would not change the outcome of this, thus the threshold algorithm gives a linear ordering with cutwidth $\Theta(k^3)$. However, there is a linear ordering where the vertices of the clique are ordered by the partitioning given in the construction of the graph. The clique itself yields a highest cut of size $\Theta(k^2)$. By carefully placing each vertex of the independent set close to its neighbors in the linear ordering, they also yield a largest cut with size $\Theta(k^2)$. Thus in an optimal ordering the cutwidth would be $\Theta(k^2)$. The threshold algorithm therefore yields an $\frac{k^3}{k^2} = k$ approximation on these graphs. As n, the number of vertices, is $\Theta(k^3)$, this is clearly $\Theta(\sqrt[3]{n})$ times worse than optimal. Considering the fact that a $log^2(n)$ approximation already exists for cutwidth on general graphs, we can without any doubt claim that the threshold algorithm performs poorly as an approximation algorithm on split graphs.

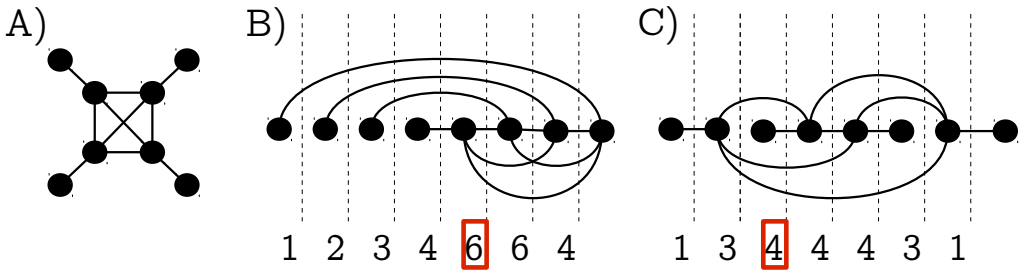

FIGURE 3.2: A) A split graph. B) Ordering given by the threshold algorithm. C) Optimal ordering.

Interval graphs

It is unknown whether or not CUTWIDTH is in P on interval graphs. While applying the threshold algorithm on interval graphs, we discovered that certain vertices made the algorithm do bad choices. In particular, a vertex with low degree adjacent to high degree vertices will often get placed far to the left in the linear ordering compared to its neighbors, thus its edges will be unnecessarily long. We mean that an edge $(u, v) \in E(G)$ is long if $|\sigma(v) - \sigma(u)|$ is large, and thus the edge is included in many cuts. Let us call such vertices *bumps*. The idea of bumps is that they all have low degree such that they all will get picked early by the threshold algorithm, but their neighbors will be picked

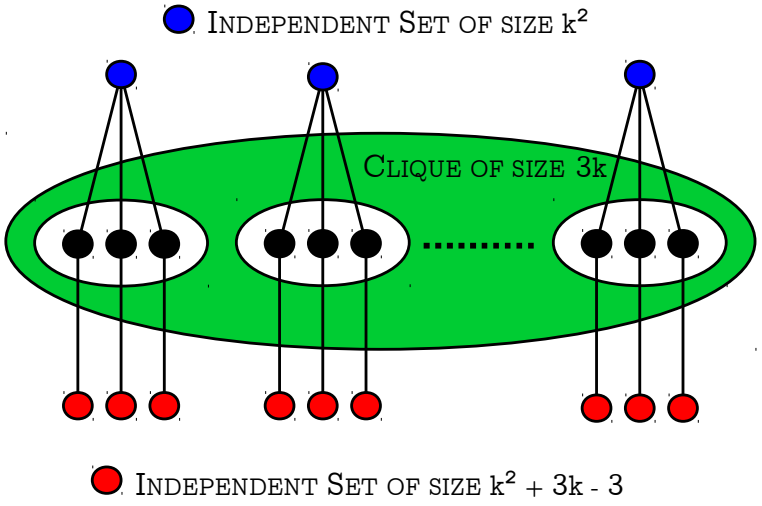

FIGURE 3.3: A graph where the threshold algorithm gives an ordering with cutwidth $\Theta(k^3)$, while an optimal ordering has cutwidth $\Theta(k^2)$.

late, because of their relatively high degree. This is essentially the same idea we used for the split graphs. By constructing a graph with several occurrences of bumps, we can make the algorithm perform arbitrarily bad, achieving a linear ordering with cutwidth which is of factor $\Theta(n)$ times the cutwidth of an optimal linear ordering. The graphs given by Figure 3.4 all yield a cutwidth of 3, while the threshold algorithm gives a linear ordering with cutwidth that is $\Theta(n)$. Because of this we think it is reasonable to discard the threshold algorithm as an approximation algorithm for cutwidth on interval graphs.

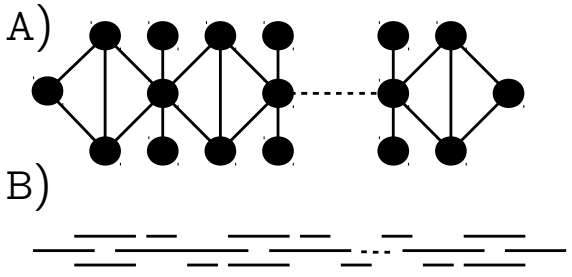

FIGURE 3.4: A) Interval graph where the pattern is repeated along the dashed line. B) Interval representation of the graph.

Unit interval graphs Even though a linear-time algorithm already exists for CUTWIDTH on unit interval graphs, it could be interesting to test the threshold algorithm on them. We observed that bumps made the threshold algorithm

perform poorly on interval graphs. Are we able to do the same for unit interval graphs? Yes. In fact, on the example given by Figure 3.5 the threshold algorithm achieves an ordering with cutwidth $\Theta(n)$, while the optimal ordering yields a constant cutwidth.

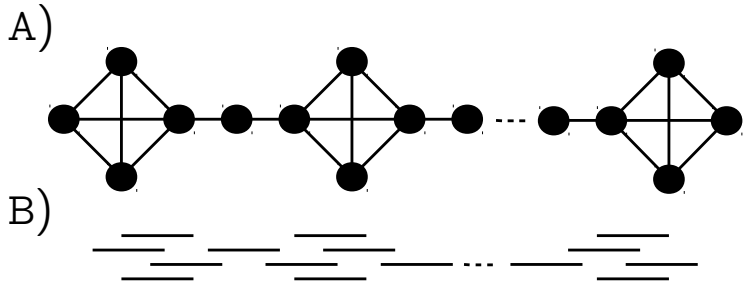

FIGURE 3.5: A) Unit interval graph. B) Interval representation of the graph.

Trivially perfect graphs It is easy to show an example where the threshold algorithm performs badly on disconnected trivially perfect graphs, e.g. Figure 3.6. In order to prove poor performance on connected graphs, we will need a bit more complicated example: We define c as the size of the universal clique in our graph. We will add l disjoint cliques, each of size $l \cdot c$, and consider a linear ordering where all l cliques are to the left of the universal clique. The number of vertices in this graph will therefore be $c \cdot (l^2 + 1)$, and the size of the largest cut in our predefined ordering will be $\Theta(c^2 \cdot l^2)$. Now, we want to remove edges from every disjoint clique such that we obtain bumps. Clearly, removing edges from a graph cannot increase its cutwidth. We will make each former clique contain $\frac{c \cdot l}{4}$ bumps, all with degree $\frac{c \cdot l}{4}$. In total the graph will have $\frac{c \cdot l^2}{4}$ bumps. An example of such a graph is shown in Figure 3.7. How will the threshold algorithm perform? The threshold algorithm will pick all $\frac{c \cdot l^2}{4}$ bumps of degree $\frac{c \cdot l}{4}$, leaving a cut with size $\frac{c^2 \cdot l^3}{16}$ which is $\Theta(c^2 \cdot l^3)$. By setting $c = 1$, we see that we have a graph with $n = \Theta(l^2)$ vertices and a cutwidth of $\Theta(l^2)$, where the threshold algorithm gives a linear ordering of cutwidth $\Omega(l^3)$. By replacing l^2 with n, we see that the threshold algorithm performs $\Omega(\sqrt{n})$ times worse than the optimal on this trivially perfect graph. This is worse than the approximation algorithm already known for general graphs, and thus we can claim that the threshold algorithm is not useful as an approximation algorithm on trivially perfect graphs.

FIGURE 3.6: An interval representation of a disconnected trivially perfect graph where the optimal cutwidth is 4, but the threshold algorithm yields a linear ordering with cutwidth $\Theta(n)$.

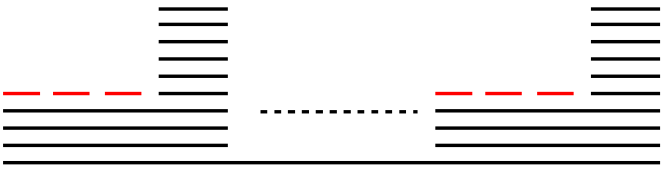

FIGURE 3.7: An example of a trivially perfect graph where l is equal to 12, and c is 1. The red intervals represent bumps.

Conclusion

On the graph classes we have looked at we have shown that the threshold algorithm can give an ordering with cutwidth that is at least $\Omega(n)$ times the optimal. This is especially bad considering the fact that an $O(log^2(n))$ algorithm already exists, solving these instances. We do not exclude the possibility of the threshold algorithm acting as a good approximation algorithm or heuristic on certain types of graphs.

Chapter 4

Cutwidth on Superfragile graphs

As we have seen earlier, the computational complexity of CUTWIDTH is not known on trivially perfect graphs, whereas it can be solved in linear time on their subclass threshold graphs. In this chapter, we will show that CUTWIDTH can be solved in polynomial time on another subclass of trivially perfect graphs, namely superfragile graphs. Superfragile graphs are not related to threshold graphs, other than being a subclass of trivially perfect graphs. The general picture of a superfragile graph, which was defined in Chapter 1, is shown in Figure 4.1. Informally, a connected superfragile graph is basically a star, where we blow up each vertex to a clique. The center clique is universal. Superfragile graphs forms the class of graphs with a compact tree representation of depth 1. In a recent Dagstuhl seminar [6], it was conjectured that CUTWIDTH of trivially perfect graphs that have a compact tree representation of depth 2, like Figure 4.2 B, is NP-hard to compute.

To show our result, we first provide an algorithm that gives an optimal *segregated* (will be defined below) linear ordering of a superfragile graph in polynomial time. We will later show that there always exists an optimal linear ordering that is segregated. We will thus show that our algorithm solves CUTWIDTH on superfragile graphs in polynomial time, proving that CUTWIDTH on superfragile graphs is in P. We will assume that the input graph is connected; clearly the cutwidth of a disconnected graph is the maximum cutwidth of the connected components.

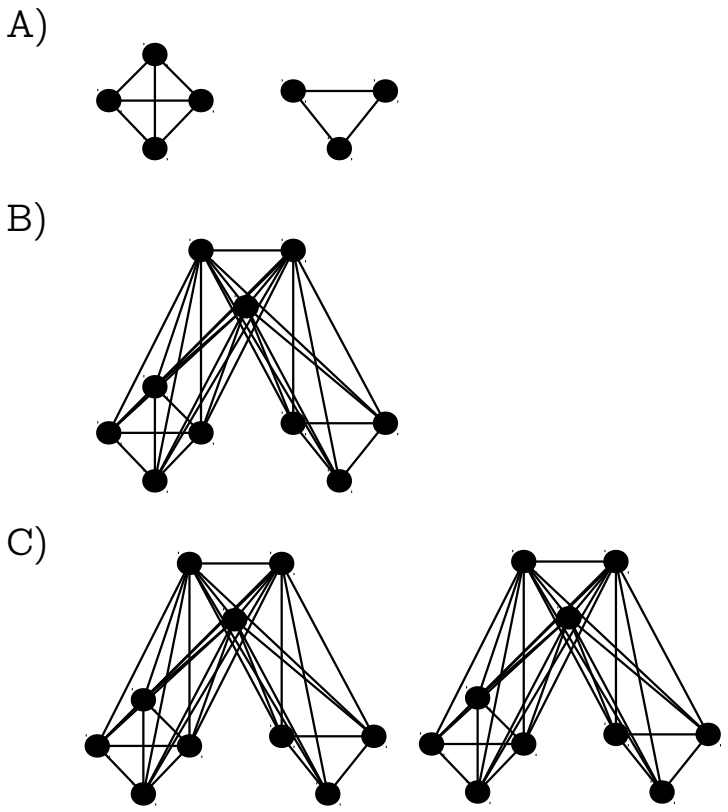

FIGURE 4.1: A) A set of disjoint cliques is superfragile. B) A graph with a universal clique whose removal results in disjoint cliques, is superfragile. C) A disjoint union of two superfragile graphs is also superfragile.

Preliminaries

Let G be a connected superfragile graph with universal clique U and components of $G[V \setminus U]$: $C_1, C_2, \ldots, C_k$, and let σ be a linear ordering of G. Observe first that for any pair of vertices $u, v \in C_i$ (or $u, v \in U$), $N[u] = N[v]$. Thus the vertices of each clique are equivalent with respect to how they are placed in a linear ordering.

We say that two sets of vertices A and B *cross* in σ if $\sigma(v) < \sigma(u) < \sigma(w)$ for any $v, w \in A$ and $u \in B$, or for any $v, w \in B$ and $u \in A$. We say that σ is *segregated* if the universal clique, U, and the disjoint cliques, $C_1, C_2, \ldots, C_k$, in $G[V(G) \setminus U]$ do not *cross* each other in σ. We say that a clique C_i is on the *left side* (of U) in a segregated linear order σ if $max_{v \in C_i}(\sigma(v)) < min_{u \in U}(\sigma(u))$. Similarly we say that a clique C_i is on the *right side* (of U) in a segregated linear order σ if $min_{v \in C_i}(\sigma(v)) > max_{u \in U}(\sigma(u))$.

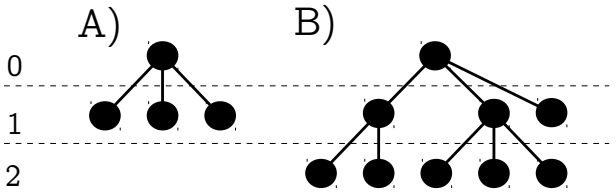

FIGURE 4.2: A compact tree representation of trivially perfect graphs where every vertex in the tree represents a set of vertices, and for every path from the root to a leaf, the vertices contained in the sets in the path forms a maximal clique. A) Superfragile graphs are those graphs that can be represented with depth 1. B) Cutwidth is conjectured to be NP-complete on those graphs with depth 2.

We say that two cliques C_i and C_j are *close* in a segregated linear ordering σ if either $min_{u \in C_i}(\sigma(u)) = max_{v \in C_j}(\sigma(v)) + 1$ or $min_{v \in C_j}(\sigma(v)) = max_{u \in C_i}(\sigma(u)) + 1$ holds. We say that two vertices, u and v, are close in σ if either $\sigma(u) + 1 = \sigma(v)$ or $\sigma(u) = \sigma(v) + 1$. We define a *swap* between two close vertices, u and v, in σ as setting $\sigma(u) = \sigma(v)$ and $\sigma(v) = \sigma(u)$. Let us define a swap between two close cliques, C_i and C_j, in a segregated linear ordering σ, where C_i is to the left of C_j: we get a new segregated ordering where C_i and C_j are close, C_j is to the left of C_i, and all other vertices (outside of C_i and C_j) keep their positions.

A set of vertices S is *gathered* in σ if $max_{v \in S}(\sigma(v)) - min_{u \in S}(\sigma(u)) = |S| - 1$. The left degree of a vertex v in σ on a set of vertices S, $degL^S_\sigma(v)$, is the number of vertices $u \in N(v) \cap S$ such that $\sigma(u) < \sigma(v)$. Similarly the right degree of a vertex v in σ, $degR^S_\sigma(v)$, is the number of vertices $u \in N(v) \cap S$ such that $\sigma(u) > \sigma(v)$. We say that an operation to change a linear ordering σ into $\sigma*$ is *safe* if $cutwidth(\sigma*, G) \leq cutwidth(\sigma, G)$. We say that the largest cut *over* a gathered set of vertices S in σ is $max_i(deg(V_i))$ where $min(\sigma(v \in S)) - 1 \leq i \leq max(\sigma(u \in S))$.

Given a superfragile graph G with universal clique U and a segregated linear ordering σ of G, we say that the *configuration* of σ is a pair of integers (l, r), where $l = degL^{V(G)\setminus U}_\sigma(u)$ and $r = degR^{V(G)\setminus U}_\sigma(u)$, for any $u \in U$. Similarly we say that the configuration of σ for a subset S, $S \subseteq V(G)$, is a pair of integers (l, r), where $l = degL^{S\setminus U}_\sigma(u)$ and $r = degR^{S\setminus U}_\sigma(u)$, for any $u \in U$.

Given a universal clique U and a vertex $u \in U$, we say that a segregated linear ordering σ is *good* if there is no segregated linear ordering $\sigma*$ with the same configuration as σ such that $cutwidth(\sigma*, G) < cutwidth(\sigma, G)$. Two linear orderings σ and $\sigma*$ of G are *twins* if they have the same configuration and if $cutwidth(\sigma*, G) = cutwidth(\sigma, G)$. We say that a set S of segregated linear orderings *covers* (is a *cover* of) G, if for all segregated linear orderings σ of G there is one segregated linear ordering $\sigma*$ in S such that σ and $\sigma*$ have the same configuration and $cutwidth(\sigma*, G) \leq cutwidth(\sigma, G)$. We say that a cover S of G is *sparse* if for every σ in S, there exists no $\sigma*$ of G with the same configuration as σ such that $cutwidth(\sigma*, G) < cutwidth(\sigma, G)$, and there are no σ, $\sigma*$ in S such that σ and $\sigma*$ are twins.

By *adding* a set of vertices $S = \{s_1, s_2, \ldots, s_{|S|}\}$ to a linear ordering σ at position i, we mean that $\sigma(v)$ is set to $\sigma(v) + |S|$ for all v that previously had $\sigma(v) \geq i$, and then $\sigma(s_j)$ is set to be equal $i + j - i$ for all $s_j \in S$.

A segregated linear ordering σ of G is *optimal* if there is no segregated linear ordering $\sigma*$ of G such that $cutwidth(\sigma*, G) < cutwidth(\sigma, G)$.

4.1 Segregated linear orderings

In this section, we will consider only segregated linear orderings. For a given superfragile graph, we will find in polynomial time a segregated ordering of minimum cutwidth, i.e, out of all its segregated orderings, one with smallest cutwidth. We will call such an ordering an *optimal segregated linear ordering*. Out first central result is stated in Lemma 4.1. See Figure 4.3 for an illustration of this statement.

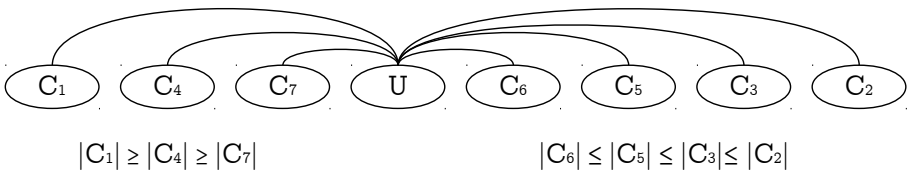

FIGURE 4.3: A segregated linear ordering where the cliques on the left side of U appear in decreasing order by size, and the cliques on the right side of U appear in increasing order by size.

Lemma 4.1. *For every superfragile graph there is an optimal segregated linear ordering where the cliques on the left side appear in decreasing order with respect to their size, and the cliques on the right side appear in increasing order with respect to their size.*

Proof Let G be a superfragile graph with universal clique U, and $C_1, C_2, \ldots, C_k$ as defined above. Let us consider two close cliques C_i and C_j, of size c_i and c_j respectively, on the left side in a segregated linear ordering σ. Let u be the size of the universal clique. We now want to find out whether or not swapping C_i and C_j is safe. To do this, we need to check whether or not a swap could increase the cutwidth. Notice that we only need to look at the largest cut over the clique to the right, as the largest cut over the left clique cannot become smaller by swapping the two cliques.

Case 1) c_i and c_j are both less than or equal to u: If C_i is to the right of C_j, we get a largest cut over C_i of size $c_i \cdot u + c_j \cdot u$. Similarly if C_j is to the right of C_i the largest cut over C_j is $c_j \cdot u + c_i \cdot u$. Since $c_i \cdot u + c_j \cdot u$ is equal to $c_j \cdot u + c_i \cdot u$, it is safe to swap them such that either C_j is to the left of C_i or to the right of C_i. Thus we can safely swap them such that the larger clique is to the left of the smaller clique.

Case 2) c_i is greater than u, c_j is not: If C_i is to the right of C_j, we get a largest cut over C_i of size $(\frac{c_i+u}{2})^2 + c_j \cdot u$. If C_j is to the right of C_i the largest cut over C_j is $c_j \cdot u + c_i \cdot u$. For placing C_i to the left of C_j to be safe, we need that $c_j \cdot u + c_i \cdot u \leq (\frac{c_i+u}{2})^2 + c_j \cdot u$. This is equivalent to $0 \leq (\frac{c_i-u}{2})^2$, an inequality that always holds since c_i and u are integers.

Case 3) c_i is greater than or equal to c_j, c_j is greater than u: If C_i is to the right of C_j, we get a largest cut over C_i of size $\frac{c_i+u}{2}^2 + c_j \cdot u$. If C_j is to the right of C_i the largest cut over C_j is $(\frac{c_j+u}{2})^2 \cdot u + c_i \cdot u$. For placing C_i to the left of C_j to be safe, we require that $(\frac{c_j+u}{2})^2 \cdot u + c_i \cdot u \leq (\frac{c_i+u}{2})^2 + c_j \cdot u$. Since c_i is greater than or equal to c_j, we can set $c_i = c_j + k$, where $k \geq 0$. Substituting this into our equation we get $(\frac{c_j+u}{2})^2 \cdot u + (c_j + k) \cdot u \leq (\frac{c_j+k+u}{2})^2 + c_j \cdot u$, which is equivalent to $\frac{k \cdot u}{2} \leq \frac{k \cdot c_j}{2} + \frac{k^2}{4}$. Since c_j is greater than u, we know that this inequality always holds.

Similarly this holds for cliques on the right side in an ordering. We have now shown that if two close cliques have different size, it is always safe to swap the largest clique furthest away from the universal clique. We can do this until the property of the lemma holds, without increasing the cutwidth of the linear ordering. Thus we can always swap any optimal segregated linear ordering to an optimal segregated ordering where the property from the lemma holds. □

Now we give an algorithm that computes the cutwidth of an optimal segregated linear ordering. The algorithm processes the disjoint cliques in decreasing order with respect to size, placing the cliques at the left or right side close to the universal clique. Observe that, although we know that we have an decreasing-increasing ordering of the cliques, we still need to decide on which side of the universal clique each clique needs to go. Once we know which cliques are on the right and which are on the left, we can apply Lemma 4.1. There are $O(2^n)$ different partitions of the cliques, as a clique can either be to the left of the universal clique or to its right. To cope with this, the algorithm applies dynamic programming, where it only has to consider $O(n)$ partitions. This is a standard way of coping with partition problems of this type, like the knapsack problem [13].

Observation 1. *Given a superfragile graph G with universal clique U and the maximal cliques $C_1, \ldots, C_k$ of $G[V(G) \setminus U]$ and a segregated linear ordering σ, the largest cut over a clique C_i is dependent on the number of vertices to the left of C_i, not which cliques are to its left, if C_i is to the left of U. Similarly the largest cut over a clique C_i is dependent on the number of vertices to the right of C_i, not which cliques are to its right, if C_i is to the right of U.*

Since the next algorithm will be based on building covers, we need to prove two properties of covers:

Lemma 4.2. *Let G be a superfragile graph and S a set of segregated linear orderings that covers G. There exists an optimal segregated linear ordering σ in S.*

Proof Let us consider any segregated linear ordering σ of G. There must be a segregated linear ordering $\sigma*$ in S with the same configuration as σ such that

$cutwidth(\sigma*, G) \leq cutwidth(\sigma, G)$. If no such $\sigma*$ exists then S is clearly not a cover of G. □

Lemma 4.3. *Let G be a superfragile graph with a universal clique U and maximal cliques $C_1, \ldots, C_k$ where $|C_i| \geq |C_j|$ for $i < j$. Given a set of segregated linear orderings S that covers $G[U \cup C_1 \cup \cdots \cup C_{i-1}]$, $1 \leq i \leq k$, a set T covering $G[U \cup C_1 \cup \cdots \cup C_i]$ can be built by placing C_i close to U on the left side and on the right side of U for every ordering in S.*

Proof Consider any segregated ordering σ of $G[U \cup C_1 \cup \cdots \cup C_i]$ with the property of Lemma 4.1 such that C_i is close to U in σ. Let σ' be the the ordering given by σ for $G[U \cup C_1 \cup \cdots \cup C_{i-1}]$. Adding C_i close to the left side and right side of U of σ' would give σ. However, since S is a cover of $G[U \cup C_1 \cup \cdots \cup C_{i-1}]$, there is a $\sigma*$ in S such that $\sigma*$ and σ' have the same configuration and $cutwidth(\sigma*, G[U \cup C_1 \cup \cdots \cup C_{i-1}]) \leq cutwidth(\sigma', G[U \cup C_1 \cup \cdots \cup C_{i-1}])$. By Observation 1 there must be a $\sigma*$ in T such that σ and $\sigma*$ have the same configuration and $cutwidth(\sigma*, G[U \cup C_1 \cup \cdots \cup C_i]) \leq cutwidth(\sigma, G[U \cup C_1 \cup \cdots \cup C_i])$. □

The algorithm The algorithm takes a superfragile graph $G = (V, E)$ as input and identifies the universal clique U and the cliques $C_1, \ldots, C_k$ of G such that $|C_i| \geq |C_j|$ for all $i < j$. Then the algorithm basically tries all segregated linear orderings of $C_1, C_2, \ldots, C_k$ with the property of Lemma 4.1. Notice that there is a bijection between a partition of the cliques into two sets and segregated linear orderings with property of Lemma 4.1. Thus as mentioned earlier there are up to $O(2^n)$ different orderings to consider. Observe that the new cuts appearing when placing a new clique is not affected by which cliques are to its left or right, but only by the number of vertices as remarked in Observation 1. This means that some orderings are unnecessary to look at, as there are other orderings which have the same number of vertices on the left and right side that are as good. Consequently, the algorithm only has to look at $O(n)$ orderings, as we explain next. The algorithm iterates through all cliques $C_1, C_2, \ldots C_k$ in the given order where $|C_1| \geq |C_2| \geq \ldots |C_{k-1}| \geq |C_k|$. At step j the algorithm looks at all the orderings in a sparse cover of cliques $C_1, \ldots, C_{j-1}$ and adds C_j both to the left side and the right side of each ordering, creating

new orderings. At the end of step j the algorithm goes through all the newly created orderings of $C_1, \ldots, C_j$ and removes all orderings that are not good. If there are two orderings with the same configuration and the same cutwidth (twins), the algorithm removes one of them arbitrarily, such that the remaining set of orderings sparsely covers $G[U \cup C_1 \cup \cdots \cup C_j]$. At the end, after step k, the algorithm goes through a set of orderings that covers G and returns the cutwidth of the ordering yielding the lowest cutwidth. To save space, the algorithm does not represent each ordering as an actual ordering, but as its configuration and its cutwidth. This means that the algorithm does not give an optimal segregated linear ordering, but instead returns the cutwidth of one.

Algorithm 5: CutwidthSuperfragile

Input: A superfragile graph $G = (V, E)$
Output: The cutwidth of G

Find U and $C_1, \ldots C_k$ s.t. $|C_i| \geq |C_j|$ for all $i < j$;
$S_0 = (0, 0, 0)$;
for $i = 1$ **to** k **do**
  for *all triples* (x, y, z) *in* S_{i-1} **do**
    $S_i = S_i \cup (x + |C_i|, y, max(z, cCut(|C_i|, |U|, x), uCut(|U|, x + |C_i|, y)))$;
    $S_i = S_i \cup (x, y + |C_i|, max(z, cCut(|C_i|, |U|, y), uCut(|U|, x, y + |C_i|)))$;
  $S_i = cleanUp(S_i)$;
$ans = \infty$;
for *all triples* (x, y, z) *in* S_k **do**
  $ans = min(ans, z)$;
return *ans*;

The method *cCut* takes as input three integers, $|C|$, $|U|$ and x, and returns the size of the largest cut over a clique C of size $|C|$ that is between a universal clique of size $|U|$ and x vertices. If $|C|$ is smaller than $|U|$, then the largest cut over C is the cut that is closest to U, having a cut size of $|C| \cdot |U| + u \cdot x$. If $|C|$ is larger than or equal to $|U|$, the largest cut over C will be the cut that is the i-th closest to U, where i is $\frac{|C|+|U|}{2}$. The size of this cut is $\lfloor \frac{|C|+|U|}{2} \rfloor \cdot \lceil \frac{|C|+|U|}{2} \rceil + |U| \cdot x$. This method uses $O(1)$ time.

The method *cleanUp* takes as input a set of triples of integers S, and returns a *nice* subset S' of S. A subset S' of S is nice if for all triples $(x, y, a) \in S'$ there is no triple $(x, y, b) \in S'$ with $b \neq a$, and for every triple $(x, y, b) \in S$

Algorithm 6: cCut

Input: Three integers: $|C|, |U|, x$
Output: The largest cut over a clique of size $|C|$ between a universal clique of size $|U|$ and x vertices

```
if |C| < |U| then
    return |C| · |U| + |U| · x;
else
    return ⌊(|C|+|U|)/2⌋ · ⌈(|C|+|U|)/2⌉ + |U| · x;
```

there is a triple $(x, y, a) \in S'$ such that $a \leq b$. The naive way to finding this set is by trying all pairs of triples, costing $O(|S|^2)$ time. By using e.g. hashing on the x and y value of the triples this can be efficiently done in $O(|S|)$ time. The purpose of *cleanUp* is to enable us to store only one triple for each pair of integers l, r, where l is the number of vertices on the left side and r is the number of vertices on the right side in an ordering with the smallest cutwidth.

Algorithm 7: cleanUp

Input: A set of triples of integers : S
Output: A *nice* subset of S

```
S' = S;
for all distinct pairs of triples (x, y, z1), (x, y, z2) ∈ S' do
    z = max(z1, z2);
    S' \ {(x, y, z)};
return S';
```

The method *uCut* takes as input the size of the universal clique U, the number of vertices to its left (l), and the number of vertices to its right (r). It returns the largest cut over a universal clique (in a linear segregated ordering) U, when U has l vertices to its left and r vertices to its right. Given a positive integer $i < |U|$, the size of the cut between the i-th vertex and the $i+1$-th vertex of u will be $i \cdot (|U| - i) + i \cdot r + (|U| - i) \cdot l$. Notice that this formula also takes care of the cut to the left of U ($i = 0$) and the cut to the right of U ($i = |U|$), and thus covers all cuts over U. As we want to find the size of the largest cut, we need to find the value of i such that this formula is maximized. The derivative of the formula with respect to i is $|U| - 2 \cdot i - l + r$, thus $i = \frac{|U|-l+r}{2}$ yields the highest cut, as the formula is quadratic and the sign before i^2 is negative. There is however a problem, i could be less than 0 or more than $|U|$, for which the cut is undefined. If i is less than 0 then $l > |U| + r$. In this case the size

of the largest cut over U is $|U| \cdot l$ ($i = 0$) as all other cuts for $0 < i \leq |U|$ must necessarily be lower as our formula is quadratic. Similarly if i is more than $|U|$ then $r > |U| + l$, and the largest cut over U is $|U| \cdot r$ by the previous argument. This calculation takes $O(1)$ time.

Algorithm 8: uCut

Input: Three integers: $|U|, l, r$
Output: The largest cut over a universal clique of size $|U|$ with l vertices to its left and r vertices to its right

```
1 if l − r > |U| then
2 |   return |U| · l;
3 else
4 |   if r − l > |U| then
5 |   |   return |U| · r;
6 |   else
7 |   |   mid = (r+l+|U|)/2 − l;
8 |   |   return mid · (|U| − mid) + mid · r + (|U| − mid) · l;
```

Now that we have explained what the subroutines do, we can go on to explain the main algorithm, $CutwidthSuperfragile$, and prove its correctness.

Proof of correctness Algorithm $CutwidthSuperfragile$ identifies U and $C_1, C_2, \ldots, C_k$ such that $|C_i| \geq |C_j|$ for $i < j$. At step i the algorithm goes through the segregated linear orderings of a sparse cover of $G[U \cup C_1 \cup \cdots \cup C_{i-1}]$ and creates two segregated linear orderings of $G[U \cup C_1 \cup \cdots \cup C_i]$ by placing C_i close to the universal clique on both sides. Placing C_i close to the universal clique is according to Lemma 4.1. As the largest cut over cliques $C_1, \ldots, C_{i-1}$ remains unchanged when placing C_i, we only have to calculate the largest cuts over C_i and U and compare it with the largest cut over $C_1, \ldots, C_{i-1}$, which is already known. These new orderings will form a cover of $G[U \cup C_1 \cup \cdots \cup C_i]$ by Lemma 4.3. The algorithm then goes through all the segregated linear orderings of $G[U \cup C_1 \cup \cdots \cup C_i]$ and removes as few orderings as possible such that all orderings are good and there are no twins. These orderings now form a sparse cover of $G[U \cup C_1 \cup \cdots \cup C_i]$. After step k the algorithm has a set of segregated linear orderings of G that covers G. This set contains an optimal segregated linear ordering by Lemma 4.2, thus the algorithm only has to go through all of the orderings in the set, and return the cutwidth of an optimal segregated linear ordering.

Running time analysis Finding U and $C_1, C_2, \ldots, C_k$ such that $|C_i| \geq |C_j|$ for $i < j$ takes linear time, which is $O(n)$ when given a tree representation, or $O(n+m)$ when given the graph (adjacency list) representation. Then the algorithm goes through a loop with k steps. At step i the algorithm goes through a set of segregated linear orderings that covers $G[U \cup C_1 \cup \cdots \cup C_{i-1}]$, which contains $O(|U| + |C_1| + \cdots + |C_{i-1}|)$ orderings, which is $O(n)$. For each of these orderings, two new orderings are made in $O(1)$ time, making a new set with a total of $O(n)$ orderings. Removing orderings from this set such that the set forms a sparse cover of $G[U \cup C_1 \cup \cdots \cup C_i]$ takes $O(n)$ time. Thus running through this loop takes $O(k \cdot n)$ time. As k is can be $O(n)$, this is $O(n^2)$. In the end the algorithm loops through a set of size $O(n)$, and spends $O(n)$ time on it. Thus this algorithm has running time $O(n^2)$.

When it comes to space, the algorithm only needs to know and store the universal clique U and the cliques $C_1, \ldots, C_k$ and at each step store the previous ordering and the next ordering. All of which is $O(n)$, thus the algorithm uses $O(n)$ space. We can thus conclude the following:

Theorem 4.4. *The cutwidth of an optimal segregated linear ordering of a superfragile graph can be computed in $O(n^2)$ time and $O(n)$ space.*

Finding the actual linear ordering In order to find a linear ordering yielding the lowest possible cutwidth, we need, for every triple (x, y, z) in S_i, to store the choices for $C_1, \ldots, C_i$ which led to this triple. As there can be $O(n)$ triples, and storing the choices takes $O(n)$ space for every triple, this will require $O(n^2)$ space. However, we will next show that we can improve this to $O(n)$ space by using a divide-and-conquer strategy.

Divide and Conquer version

We gave an algorithm that computes the cutwidth of a superfragile graph in $O(n^2)$ time, using $O(n)$ space. However, if we want an actual linear ordering yielding the lowest cutwidth, we have to use $O(n^2)$ space. We will now show an algorithm using the principles of divide-and-conquer to find an optimal linear ordering in $O(n^2)$ time, using $O(n)$ space.

The algorithm The algorithm takes as input a superfragile graph $G = (V, E)$ and then identifies the universal clique U of G and the cliques $C_1, \ldots, C_k$ such that $|C_i| \geq |C_j|$ for $i < j$. The algorithm then iterates through $C_1, \ldots, C_k$ in k steps, doing the same operations as the previous algorithm, Algorithm 5. After these steps the algorithm goes through all the orderings in S_k and identifies the number of vertices on the left side, denoted $endX$ and the number on the right side , denoted $endY$, of the ordering with the least cutwidth. The algorithm now calls the recursive method *recursive* with $(1, k, 0, 0, endX, endY)$ as input. The *recursive* method takes six integers as input $start$, end, $startX$, $startY$, $endX$, and $endY$ as input. This recursive method then iterates through cliques $C_{start}, \ldots C_{end}$ in the same manner as earlier, with one exception: When processing clique $C_{\frac{start+end}{2}}$, the method stores the size of the left and right sets. These two values will then be carried through in the next iterations, such that for each ordering we know how many vertices were on the left side and right side after placing $C_1, \ldots, C_{\frac{start+end}{2}}$. After processing the cliques the method looks at the orderings with $endX$ vertices on the left side, and $endY$ vertices on the right side. For this orderings the number of vertices on the left side and number of vertices on the right side after placing the cliques $C_1, \ldots, C_{\frac{start+end}{2}}$ is known, let us denote these numbers as $midX$ and $midY$ respectively. The method now performs two calls to itself: $(start, \frac{start+end}{2}, startX, startY, midX, midY)$ and $(\frac{start+end}{2} + 1, end, midX, midY, endX, endY)$. When the recursive method receives a call where $start = end$, then there is only one clique to place, namely C_{start}. As the number on the left side before adding the clique is given, $startX$, and the number on the left side after adding the clique is given, $endX$, the method can assess the placement of C_{start}. If $startX < endX$ then clearly C_{start} was placed on the left side, if not, C_{start} was placed on the right side. After storing the choice for C_{start} the method does not call itself further. When the recursive method terminates, the choice of side for all cliques $C_1, \ldots, C_k$ is determined. Finding the actual segregated linear ordering σ is now trivial, as the side each clique is positioned at is known, and the order between the cliques is known. In particular, the algorithm starts with placing all the cliques that belong to the left side in decreasing order, then the universal clique is placed, and at last the cliques that belong to the right side are placed in increasing order. The main algorithm is given as Algorithm 9 and the recursive method is given as Algorithm 10.

The method *ordering* takes a table *choice* and a superfragile graph G as input.

Algorithm 9: DivideAndConquer

Input: A superfragile graph $G = (V, E)$
Output: An optimal segregated linear ordering of G

Find U and $C_1, \dots C_k$ s.t. $|C_i| \geq |C_j|$ for all $i < j$;
$S_0 = (0, 0, 0)$;
for $i = 1$ **to** k **do**
    for *all triple* (x, y, z) *in* S_{i-1} **do**
        $S_i = S_i \cup (x + |C_i|, y, max(z, cCut(|C_i|, |U|, x), uCut(|U|, x + |C_i|, y)))$;
        $S_i = S_i \cup (x, y + |C_i|, max(z, cCut(|C_i|, |U|, y), uCut(|U|, x, y + |C_i|)))$;
    $S_i = cleanUp(S_i)$;
$ans = \infty$;
$endX = 0$;
$endY = 0$;
for *all triples* (x, y, z) *in* S_k **do**
    if $ans > z$ **then**
        $ans = z$;
        $endX = x$;
        $endY = y$;
$choice[]$;
$recursive(1, k, 0, 0, endX, endY)$;
$\sigma = ordering(choice, G)$;
return σ;

It identifies the universal clique U of G and the cliques $C_1, \dots, C_k$ such that $|C_i| \geq |C_j|$ for $i < j$. It then gives a segregated linear ordering σ where the vertices of each clique is placed according to the *choice* table. The *choice* table describes for every clique C_i if clique C_i is to the left or right of the universal clique U. Since the ordering has the property described by Lemma 4.1, the cliques on the left side will be placed in decreasing order with respect to size, and the cliques on the right side will be placed in increasing order with respect to size. This method takes $O(n)$ time.

The method $cleanUp*$ takes as input a set of quintuples of integers S, and returns a *nice* subset S' of S. A subset S' of S is nice if for all quintuples $(x, y, a, l_1, r_1) \in S'$ there is no quintuple $(x, y, b, l_2, r_2) \in S'$, and for every quintuple $(x, y, b, l_2, r_2) \in S$ there exists a triple $(x, y, a, l_1, r_1) \in S'$ such that $a \leq b$. Finding this set can be done in $O(|S|)$ time in a similar way as in the $cleanUp$ method.

Algorithm 10: recursive

Input: Integers: $start, end, startX, startY, endX, endY$
Output: VOID

if $start == end$ **then**
  if $startX < endX$ **then**
    $choice[start] = 1;$
  else
    $choice[start] = 0;$
else
  $S_{start-1} = \{(startX, startY, 0, 0, 0)\};$
  $mid = \lfloor \frac{start+end}{2} \rfloor;$
  for $i = start$ **to** end **do**
    for *all quintuples* (x, y, z, l, r) *in* S_{i-1} **do**
      if $i = mid$ **then**
        $S_i = S_i \cup (x + |C_i|, y, max(z, cCut(|C_i|, |U|, x), uCut(|U|, x + |C_i|, y)), x + |C_i|, y);$
        $S_i = S_i \cup (x, y + |C_i|, max(z, cCut(|C_i|, |U|, y), uCut(|U|, x, y + |C_i|)), x, y + |C_i|);$
      else
        $S_i = S_i \cup (x + |C_i|, y, max(z, cCut(|C_i|, |U|, x), uCut(|U|, x + |C_i|, y)), l, r));$
        $S_i = S_i \cup (x, y + |C_i|, max(z, cCut(|C_i|, |U|, y), uCut(|U|, x, y + |C_i|)), l, r);$
    $S_i = cleanUp^*(S_i)$;
  $midX = 0;$
  $midY = 0;$
  for *all quintuples* (x, y, z, l, r) *in* S_k **do**
    if $x == endX$ *and* $y == endY$ **then**
      $midX = l;$
      $midY = r;$
  $recursive(start, mid, startX, startY, midX, midY);$
  $recursive(mid + 1, end, midX, midY, endX, endY);$

Algorithm 11: ordering

Input: A table *choice* and a superfragile graph G
Output: a segregated linear ordering of G according to choice

Find U and $C_1, \dots C_k$ s.t. $|C_i| \geq |C_j$ for all $i < j$;
$nxt = 1$;
for $i = 1$ *to* k **do**
    if $choice[i] == 1$ **then**
        for *Vertex* $v \in C_i$ **do**
            $\sigma(v) = nxt$;
            $nxt = nxt + 1$;
for *Vertex* $v \in U$ **do**
    $\sigma(v) = nxt$;
    $nxt = nxt + 1$;
for $i = k$ *to* 1 **do**
    if $choice[i] == 0$ **then**
        for *Vertex* $v \in C_i$ **do**
            $\sigma(v) = nxt$;
            $nxt = nxt + 1$;
return σ;

Algorithm 12: cleanUp*

Input: A set of quintuples of integers: S
Output: A "good" set of quintuples

for *all distinct pairs of quintuples* $(x, y, z_1, l_1, r_1), (x, y, z_2, l_2, r_2) \in S$ **do**
    if $z_1 > z_2$ **then**
        $S \setminus \{(x, y, z_1, l_1, r_1)\}$;
    else
        $S \setminus \{(x, y, z_2, l_2, r_2)\}$;
return S;

Proof of correctness Let G be the input graph with a universal clique U and cliques $C_1, \dots, C_k$, such that $|C_i| \geq |C_j|$ for $i < j$. We want to prove that Algorithm 9 produces an optimal segregated linear ordering. The algorithm starts by finding the number of vertices on the left, l, of U and the number of vertices on the right, r, of U in an optimal segregated linear ordering σ, by using the same technique as in the previous algorithm. Clearly, σ in $G[U]$ has 0 vertices on the left and the right side of U. Thus the start and end configuration of an optimal segregated ordering is known. The algorithm then performs a

similar technique as before, but now the configuration of every good segregated linear ordering after placing half the of the cliques is known. In other words, for an optimal segregated linear ordering σ of G, the configuration of σ in $G[U \cup C_1 \cup \cdots \cup C_{\frac{k}{2}}]$ is known. The algorithm then performs two recursive call, one going from the start configuration to the configuration of σ in $G[U \cup C_1 \cup \cdots \cup C_{\frac{k}{2}}]$ storing the configuration of σ after placing $\frac{k}{4}$ cliques. The second call goes from the configuration of σ in $G[U \cup C_1 \cup \cdots \cup C_{\frac{k}{2}}]$ to the configuration of σ in G, finding the configuration of σ after placing $\frac{3 \cdot k}{4}$ cliques. Since the ordering of $C_1, \ldots, C_{\frac{k}{2}}$ in σ does not affect any cut over cliques $C_{\frac{k}{2}+1}, \ldots, C_k$ and vice versa it is enough to only know the configuration σ after placing $C_1, \ldots, C_{\frac{k}{2}}$. The algorithm continues doing these recursive calls until the configuration of σ in $G[U \cup C_1 \cup \cdots \cup C_i]$ is known for all $1 \leq i \leq k$. When all of these configurations are known, the side (with respect to U) of each clique C_i in σ is found by looking at the configuration of σ in $G[U \cup C_1 \cup \cdots \cup C_{i-1}]$ and $G[U \cup C_1 \cup \cdots \cup C_i]$. When the side (with respect to U in σ) of every clique is known, the actual ordering σ is easily found by exploiting Lemma 4.1.

Running time analysis Finding U and $C_1, C_2, \ldots, C_k$ such that $|C_i| \geq |C_j|$ for $i < j$ takes linear time, which is $O(n)$ when given a tree representation, or $O(n + m)$ when given the graph (adjacency list) representation. Then the algorithm goes through a loop with k steps. At step i the algorithm goes through a set of segregated linear orderings that covers $G[U \cup C_1 \cup \cdots \cup C_{i-1}]$, which contains $O(|U| + |C_1| + \cdots + |C_{i-1}|)$ orderings, which is $O(n)$. For each of these orderings, two new orderings are made in $O(1)$ time, making a new set with a total of $O(n)$ orderings. Removing orderings from this set such that the set forms a sparse cover of $G[U \cup C_1 \cup \cdots \cup C_i]$ takes $O(n)$ time. Thus running through this loop takes $O(k \cdot n)$ time. As k is can be $O(n)$, this is $O(n^2)$. The algorithm then goes through all orderings in a set of size $O(n)$, picking the one of smallest cutwidth. Then the recursive function gets called. At the first level of the recursion tree the algorithm spends $O(k \cdot n)$ time, by going through k cliques and for each of them checking up to n orderings. On the second level of the recursion tree, the algorithm has made two calls, both placing $\frac{k}{2}$ cliques, but not the same amount of vertices. However if the first call places a vertices and thus has to go through a orderings, the second call only has to place $n - a$ vertices and go through $n - a$ orderings. Clearly $\frac{k}{2} \cdot a + \frac{k}{2} \cdot (n - a) =$

$\frac{k}{2} \cdot n$. So for every level in the recursion tree the vertices are not spread evenly, but the cliques are. So at level i in the recursion tree the algorithm spends $O(\frac{k}{2^{i-1}}) \cdot O(n)$ time. Since $\sum_{i=0}^{log(k)} O(\frac{k}{2^i})$ is $O(k)$, the algorithm spends $O(k \cdot n)$ time in the recursive method. After the recursion is done, the algorithm calls the method ordering which spends $O(n)$ time. Thus the running time of this algorithm is $O(k \cdot n)$, which is $O(n^2)$.

With the explanations above, we have now proved the following:

Theorem 4.5. *An optimal segregated linear ordering of a superfragile graph can be computed in $O(n^2)$ time and $O(n)$ space.*

4.2 Optimal linear orderings

We have so far shown that we can find an optimal segregated linear ordering in $O(n^2)$ time. Now we want to show that we can change any linear ordering safely into a segregated linear ordering. By safely we mean that the cutwidth of the ordering will not increase.

Observation 2. *When swapping two close vertices u and v, the only cut that changes is the one between them. See Figure 4.4.*

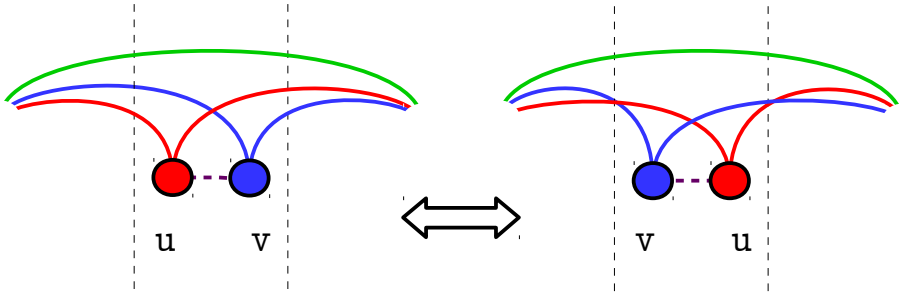

FIGURE 4.4: Swapping two close vertices u and v can only change the cut between them.

Consider two close vertices u and v where $\sigma(u) < \sigma(v)$. Let c be the number of edges with one endpoint to the left of u and one endpoint to the right of v (the green edges in Figure 4.4). Notice that all these edges appear in all the cuts over u and v, even after swapping them. Thus in further proofs, when

comparing these cuts, we will disregard these c edges, as they appear in all of the cuts.

Lemma 4.6. *In every linear ordering of an arbitrary graph G, it is safe to swap two close vertices u and v, where $\sigma(u)+1=\sigma(v)$ if one of the following conditions is satisfied:*

- $degL(v) \leq degR(v)$ *or* $degR(v) \leq degL(v)$, *and* $(u,v) \notin E(G)$.
- $degL(v)+2 \leq degR(v)$ *or* $degR(v)+2 \leq degL(v)$, *and* $(u,v) \in E(G)$.

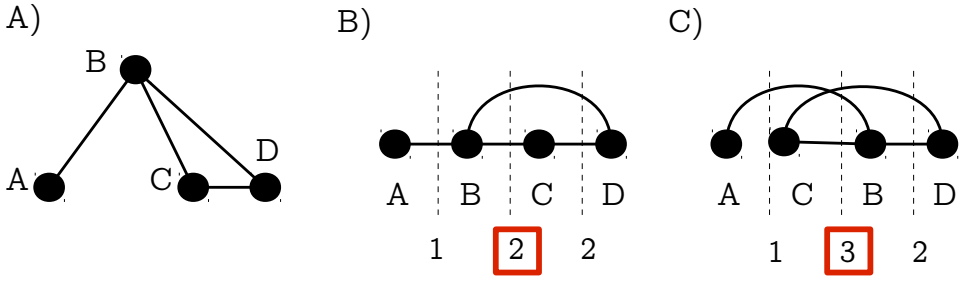

FIGURE 4.5: A) A graph. B) A linear ordering where $degL(B)+1=degR(B)$ with cutwidth 2. C) A linear ordering where B is swapped, yielding a higher cutwidth of 3.

Proof Given a graph G and a linear ordering σ, let u and v be close in the ordering. Let l_u be the left degree of u and let r_u be the right degree of u, and let l_v and r_v be the left and right degree of v respectively. Assuming that $\sigma(u)+1=\sigma(v)$, we want to show that swapping u and v is safe. Notice that the only cut that changes when swapping u and v is the cut between them, see Observation 2. Now we want to find the constraints such that this cut is guaranteed not to be larger than the cutwidth of σ.

Case 1) u and v are not neighbors: We consider the sizes of the two cuts around u and v:
I: l_u+l_v
II: r_u+r_v

If we swap u with v, we get a new cut of size l_u+r_v. We need one of the following inequalities to be satisfied for this swap to be safe:

I: $l_u + r_v \leq l_u + l_v \Rightarrow r_v \leq l_v$
II: $l_u + r_v \leq r_u + r_v \Rightarrow l_u \leq r_u$

From I and II we can see that the swap is safe if either $r_v \leq l_v$ or $l_u \leq r_u$ holds, which is according to the lemma.

Case 2) u and v are neighbors: We consider the two cuts around u and v:
I: $l_u + l_v - 1$
II: $r_u - 1 + r_v$

If we swap u with v, we get a new cut of size $l_u + r_v + 1$. We need one of the following inequalities to be satisfied for this swap to be safe:

I: $l_u + r_v + 1 \leq l_u + l_v - 1 \Rightarrow r_v + 2 \leq l_v$
II: $l_u + r_v + 1 \leq r_u - 1 + r_v \Rightarrow l_u + 2 \leq r_u$

From I and II we can see that the swap is safe if either $r_v + 2 \leq l_v$ or $l_u + 2 \leq r_u$ holds, which is according to the lemma. □

Lemma 4.7. *Given a graph G, there is always an optimal linear ordering σ of $V(G)$ where the universal clique is gathered.*

Proof Given a graph G and a linear ordering σ of $V(G)$, consider any universal vertex u. If $\sigma(u)$ is less than $\frac{V(G)}{2}$, it is clear by Lemma 4.6 that we can swap u safely until $\sigma(u)$ is equal to $\frac{V(G)}{2}$. Similarly, if $\sigma(u)$ is greater than $\frac{V(G)}{2} + 1$, we can safely swap u until $\sigma(u)$ is equal to $\frac{V(G)}{2} + 1$. □

Lemma 4.8. *Given a superfragile graph G, there is always an optimal linear ordering σ of $V(G)$ that is segregated.*

Proof Consider a clique C_i of G in a linear ordering σ where the universal clique U is gathered, this can be assumed by Lemma 4.7. There exists a vertex $v \in C_i \cup U$, such that $degL_\sigma^{C_i \cup U}(v) = degR_\sigma^{C_i \cup U}(v)$ or $degL_\sigma^{C_i \cup U}(v) + 1 = degR_\sigma^{C_i \cup U}(v)$. Clearly for any vertex $w \in C_i$ where $\sigma(w) < \sigma(v)$, we can safely swap w to the right until $\sigma(w) = \sigma(v) - 1$. Similarly we can for any vertex $w \in C_i$ where $\sigma(w) > \sigma(v)$ safely swap w until $\sigma(w) = \sigma(v) + 1$.

If $v \in C_i$, then we clearly have an ordering where C_i is gathered. If $v \in U$, then C_i and U are not necessarily gathered. However, $C_i \cup U$ is gathered. Let L be $degL_{\sigma}^{V(G)\setminus\{C_i \cup U\}}(v)$, the vertices to the left of $C_i \cup U$, and let R be $degR_{\sigma}^{V(G)\setminus\{C_i \cup U\}}(v)$. Let us now consider two close vertices, $u \in U$ and $v \in C_i$. Let $|C_i| - b$ be the number of vertices in C_i to the left of v, and let $b - 1$ be the number of vertices in C_i to the right of v. Let $|U| - a$ be the number of vertices in U to the left of u, and let $a - 1$ be the number of vertices in U to the right of u. We now have two cases:

Case 1) $\sigma(u) < \sigma(v)$:

$degL(v) = |C_i| - b + |U| - a + 1$

$degR(v) = b - 1 + a - 1$

$degL(u) = |C_i| - b + |U| - a + L$

$degR(u) = b + a - 1 + R$

Let us consider the size of the cut between u and v in the ordering. Before the swap it is $degR(u) + degL(v) - 1$ which is equal to $b + a - 1 + R + |C_i| - b + |U| - a + 1 - 1 = |C_i| + R + |U| - 1$. After the swap the size of the cut between u and v is $degR(v) + 1 + degL(u) + 1 - 1$ which is equal to $b - 1 + a - 1 + 1 + |C_i| - b + |U| - a + L + 1 - 1 = |C_i| + |U| + L - 1$. Clearly this swap is safe if the new cut between u and v is smaller than or equal to the old. Thus we require that: $|C_i| + |U| + L - 1 \leq |C_i| + R + |U| - 1 \Rightarrow L \leq R$. Notice that this inequality is independent of the relative positions of u and v. So if $L \leq R$ then we can safely repeatedly swap each vertex of C_i with its left neighbor, such that eventually all vertices of C_i end up to the left of the vertices in U. This will make both C_i and U gathered.

Case 2) $\sigma(u) > \sigma(v)$:

$degL(v) = |C_i| - b + |U| - a$

$degR(v) = b - 1 + a$

$degL(u) = |C_i| - b + 1 + |U| - a + L$

$degR(u) = b - 1 + a - 1 + R$

Let us consider the cut between u and v in the ordering. Before the swap its size is $degR(v) + degL(u) - 1$ which is equal to $b - 1 + a + |C_i| - b + 1 + |U| - a + L - 1 = |C_i| + |U| + L - 1$. After the swap the size of the cut between u and v is $degR(u) + 1 + degL(v) + 1 - 1$ which is equal to $b - 1 + a - 1 + R + 1 + |C_i| - b + |U| - a + 1 - 1 = R + |C_i| + |U| - 1$. Clearly this swap is safe if the new cut between u and v is smaller than or equal to the old. Thus we require that: $R + |C_i| + |U| - 1 \leq |C_i| + |U| + L - 1 \Rightarrow R \leq L$. Notice that this inequality is independent of the relative positions of u and v. So if $R \leq L$ then we can safely repeatedly swap the vertices of C_i with its right neighbor, such that they all end up to the right of the vertices in U. This will make both C_i and U gathered.

If $L \leq R$ then for every vertex of C_i, we can repeatedly swap it with its close vertex to the left, until it is to the left of all vertices in U. If not, then for every vertex of C_i, we can repeatedly swap it with its close vertex to the right, until it is to the right of all vertices in U. Thus making both C_i and U gathered. Clearly, we can make all cliques gathered without increasing the cutwidth and thus making σ segregated. □

From Theorem 4.4 and Lemma 4.8, we can now immediately conclude the following.

Theorem 4.9. *The cutwidth of a superfragile graph on n vertices can be computed in $O(n^2)$ time and $O(n)$ space.*

From Theorem 4.5 and Lemma 4.8 it follows that we can produce a linear ordering corresponding to the cutwidth also within the same running time, as stated in the following theorem.

Theorem 4.10. *An optimal linear ordering of a superfragile graph on n vertices can be computed in $O(n^2)$ time and $O(n)$ space.*

Chapter 5

Conclusion and further research

The initial purpose of this thesis was to explore whether the algorithm for solving Cutwidth on threshold graphs [21] could act as an approximation algorithm on other graph classes. As we have seen in Chapter 3, this algorithm does not seem to be useful in this respect on any of the graph classes that we studied. The work on this thesis quickly developed in a more theoretical direction. First we improved the naive $O^*(n!)$ time solution on general graphs with a standard dynamic programming algorithm solving Cutwidth in $O^*(2^n)$ time and space in Chapter 2. We also traded time with space with a divide-and-conquer algorithm solving Cutwidth in $O^*(4^n)$ time and $O(n \cdot log(n))$ space. These algorithms were well known [3] [23] [5], but developing these algorithms on our own was a good exercise to get a deeper understanding of the Cutwidth problem.

The main result of this thesis is presented in Chapter 4. Motivated by the fact that it is unknown whether or not Cutwidth is in P on interval graphs, we started by looking at interval graphs and its subgraphs. Knowing that Cutwidth is solvable in linear time on proper interval graphs, which have representations where no intervals are nested, we took a look at those interval graphs which have a representation where all intervals are nested, namely trivially perfect graphs. Even on trivially perfect graphs and some of their simpler subclasses, the computational complexity of Cutwidth is open. We concentrated on resolving this on a subclass of trivially perfect graphs, called superfragile graphs. Our main result in this thesis is giving an $O(n^2)$ time algorithm solving Cutwidth on superfragile graphs. Before our result, it was unknown

whether or not CUTWIDTH on superfragile graphs were in P or NP-complete. Thus our algorithm is the first to show that CUTWIDTH on superfragile graphs, in fact, is in P. Parallel to our discovery, CUTWIDTH was conjectured to be NP-complete on those graphs with a compact tree representation of depth 2. This implies that our result might be tight, in the sense that no natural superclass of superfragile graphs admits a polynomial time algorithm for CUTWIDTH.

In this last chapter, we will have a look at graph parameters similar to cutwidth, and we will give some discussions on these related to the work we have done in this thesis. We will also list open problems for further research, both regarding these parameters and regarding cutwidth.

5.1 Linear ordering problems

The CUTWIDTH problem belongs to a family of problems where the question is whether or not a linear ordering of vertices exists such that some condition is satisfied. These problems are often referred to as *layout problems*, and the word layout is often used interchangeably with linear ordering. A survey by Díaz, Petit and Serna in 2002 [14] gives an overview over different layout problems and results on them. Petit later followed up in 2011 with addenda [27] to this survey including more results, many of which where published after the first mentioned survey.

Bandwidth

The BANDWIDTH problem is one of the most well known and well studied layout problems. Given a graph $G = (V, E)$ the problem asks to find a linear ordering σ of G such that $max_{(u,v)\in E(G)}(|\sigma(u) - \sigma(v)|)$ is minimized. In other words the problem asks for an ordering where the length of the longest edge is as small as possible. Surprisingly, the BANDWIDTH problem is NP-hard on trees [17], but is, interestingly, solvable on interval graphs in $O(n \cdot log(n))$ time [26]. One interesting fact is that while most linear ordering problems have an exact $O^*(2^n)$ algorithm [5], the fastest exact algorithm for BANDWIDTH has a running time that is $O^*(4.383^n)$ [12]. The *bandwidth* of a graph G, $bandwidth(G)$ is the lowest integer k such that there exists a linear ordering σ of G such that

$max_{(u,v)\in E(G)}(|\sigma(u) - \sigma(v)|) \leq k$. The bandwidth of a graph gives an upper bound on the cutwidth of the graph: consider any graph G with bandwidth k, the cutwidth of G cannot be more than $\frac{k\cdot(k+1)}{2}$ due to the argument that follows. Given a linear ordering σ of G, a cut between two close vertices can at most contain l edges of length l, $l-1$ edges of length $l-1$, and so on. If there is an ordering where the length of an edge is at most k, then clearly the largest cut can have at most $\sum_{i=1}^{k} i$ edges.

Bandwidth on Threshold graphs Bandwidth is solvable in $O(n \cdot log(n))$ time on interval graphs, and thus also in the same time on threshold graphs. However, it could still be interesting to see if the threshold algorithm for CUTWIDTH by Heggernes, Lokshtanov, Mihai and Papadopoulos [21] solves BANDWIDTH. Experimenting with this idea, we quickly found a counter-example, where the threshold algorithm performed worse than optimal. This example is shown in Figure 5.1.

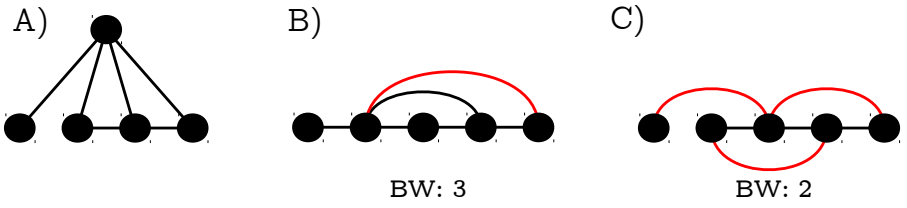

FIGURE 5.1: A) A threshold graph G. B) The ordering given by the threshold algorithm, with a bandwidth of 3 (red edge). C) An optimal ordering, with a bandwidth of 2 (red edges).

Optimal Linear Arrangement

The OPTIMAL LINEAR ARRANGEMENT problem, also known as the MINIMUM LINEAR ARRANGEMENT problem, abbreviated OLA and MINLA respectively, is closely related to both CUTWIDTH and BANDWIDTH. Instead of asking for the linear ordering minimizing the largest cut or longest edge, it asks for the linear ordering minimizing the sum of the lengths of all edges. A more formal definition of OLA: given a graph G the problem asks for the linear ordering σ of G such that $\sum_{(u,v)\in E(G)} |\sigma(u) - \sigma(v)|$ is minimized. This is equivalent to the sum of the size of every cut, $\sum_{i=1}^{V(G)-1} deg(V_i)$, since each edge $(u, v) \in E(G)$

is included $|\sigma(u) - \sigma(v)|$ cuts. For this reason OLA is also referred to as *sum bandwidth* and *sum cutwidth*. We say that $OLA(G)$ is the lowest integer k such that there exists a linear ordering σ of G such that $\sum_{(u,v)\in E(G)} |\sigma(u) - \sigma(v)| \leq k$. Given a graph G with cutwidth k, $OLA(G)$ is at most $(n-1) \cdot k$. This follows from the fact that there are $n-1$ cuts, all of which size is at most k. Similarly, if k is the bandwidth of G, then $OLA(G)$ is at most $m \cdot k$, where m is the number of edges in G, as there are m edges with length at most k. The computational complexity of OLA is opposite to that of Bandwidth, as OLA is polynomial on trees [8] and NP-hard on interval graphs [10].

OLA on Threshold graphs It is unknown whether or not OLA is in P for threshold graphs. We thought of an algorithm where the vertices are processed in the construction order of the input threshold graph. Let G be the input graph and v_i be the i-th vertex in the construction order of G. If v_i is an isolated vertex, v_i placed at the end of the linear ordering, $\sigma(v_i) = i$. If v_i is a dominant vertex, the vertex is placed in the end of the linear ordering, and continuously swapped to the left as long as $\sum_{(u,v)\in E(G[\{v_1,\ldots,v_i\}])} |\sigma(u) - \sigma(v)| \leq k$ is decreased by the operation. The idea is that whenever a dominant vertex v_i is added, it neighbors $v_1, \ldots, v_{i-1}$, thus in order to minimize the sum of its edges, it should be placed toward the middle of vertices $v_1, \ldots, v_{i-1}$. Why not place it exactly in the middle? Well, if there was a dominant vertex v_j, $j < i$, then placing v_i just to the right of v_j might not be optimal, as v_j may have neighbors to the right of v_j, whose distance now are increased by 1. Toying with the idea of this algorithm, we came to the realization that the algorithm for solving Cutwidth on threshold graphs [21] actually gives such an ordering, in linear time. In order to get an indication if such a linear ordering is optimal for OLA, we implemented an exact $O^*(2^n)$ algorithm solving OLA and compared it to the threshold algorithm for Cutwidth. Comparing the output of both algorithms for all threshold graphs with up to 16 vertices, our result was positive: the threshold algorithm for Cutwidth is optimal for all threshold graphs with up to 16 vertices. This is of course no proof, but we believe that the correctness of this algorithm deserves further study. In this regard we will state two open questions:

- Is OLA on threshold graphs in P?

- Does the threshold algorithm for the CUTWIDTH problem [21] also admit an optimal ordering for OLA?

OLA on Superfragile graphs As with threshold graphs, the computational complexity of OLA on superfragile graphs is open. We have made an algorithm which we believe is correct, solving OLA on superfragile graphs in $O(n^2)$ time and $O(n)$. Given a superfragile graph G with universal clique U and cliques $C_1, C_2, \ldots, C_k$, the algorithm yields a linear ordering which is segregated. The idea is that the sum of the lengths of the edges between vertices in each clique is minimized, and a segregated linear ordering will do just that. The sum of the lengths of the edges from vertices in U to vertices in cliques $C_1, C_2, \ldots, C_k$ is however not necessarily minimized. In order to minimize this sum, the cliques $C_1, \ldots, C_k$ are placed such that the number of vertices to the left of U is as close to the number of vertices to the right of U as possible. To find such an ordering, we can apply the idea of the knapsack problem, similar to what we did for CUTWIDTH on superfragile graphs. Our algorithm gives an optimal segregated linear ordering, but we do not know if such an ordering is in fact an optimal linear ordering. As with OLA on threshold graphs we tested this algorithm on all superfragile graphs with up to 20 vertices, by comparing our algorithm to the exact algorithm. The result was positive: our algorithm is optimal for all superfragile graphs with up to 20 vertices. We will thus state two open questions:

- Is OLA on superfragile graphs in P?
- Given a superfragile graph G, is there always an optimal linear ordering σ of $V(G)$ that is segregated?

5.2 Further research on Cutwidth

Throughout this thesis we stated that is unknown whether or not the CUTWIDTH problem is NP-hard on several graph classes. Since most of these questions were left unanswered in the thesis, we find it natural to ask them again:

- Is CUTWIDTH in P or NP-hard for trivially perfect graphs with compact tree representation of depth 2?

- Is Cutwidth in P or NP-hard for trivially perfect graphs?
- Is Cutwidth in P or NP-hard for interval graphs?

Notice that in order to prove that Cutwidth is NP-hard on all these classes, it is enough to show that it is NP-hard for the graphs with a compact tree representation of depth 2, as they are subgraphs of trivially perfect graphs and interval graphs. Similarly, in order to prove that they are all in P, it is enough to show that Cutwidth is polynomial time solvable for interval graphs, as interval graphs is a superclass to both the other classes. Of course, Cutwidth on these classes can be both NP-hard and in P, but only if P=NP.

In Chapter 2 we gave an $O^*(2^n)$ algorithm solving the Cutwidth problem on general graphs. As mentioned earlier, a common question for NP-hard problems has been if there is any algorithm faster than $O^*(2^n)$ and this question remains unanswered for Cutwidth on general graphs. So we pose the question again:

- Is there an algorithm with running time $O^*((2-\epsilon)^n)$, $\epsilon > 0$, solving Cutwidth on general graphs?

Bibliography

[1] Clay Mathematics Institute millennium prize. `http://www.claymath.org/millennium/`. Accessed: 2013-11-10.

[2] Ausiello, G., Crescenzi, P., Gambosi, G., Kann, V., Marchetti-Spaccamela, A., and Protasi, M. *Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability Properties*. No. 1. Springer, 1999.

[3] Bellman, R. Dynamic programming treatment of the travelling salesman problem. *Journal of the ACM (JACM) 9*, 1 (1962), 61–63.

[4] Blin, G., Fertin, G., Hermelin, D., and Vialette, S. Fixed-parameter algorithms for protein similarity search under mrna structure constraints. *Journal of Discrete Algorithms 6*, 4 (2008), 618–626.

[5] Bodlaender, H. L., Fomin, F. V., Koster, A. M., Kratsch, D., and Thilikos, D. M. A note on exact algorithms for vertex ordering problems on graphs. *Theory of Computing Systems 50*, 3 (2012), 420–432.

[6] Bodlaender, H. L., Heggernes, P., and Lokshtanov, D. Graph modification problems (dagstuhl seminar 14071).

[7] Brandstädt, A., Le, V. B., and Spinrad, J. P. *Graph classes: a survey*. No. 3. Siam, 1999.

[8] Chung, F. Labelings of graphs. *Selected topics in graph theory 3* (1988), 151–168.

[9] Chung, F. R., Leighton, F. T., and Rosenberg, A. L. Embedding graphs in books: a layout problem with applications to vlsi design. *SIAM Journal on Algebraic Discrete Methods 8*, 1 (1987), 33–58.

[10] Cohen, J., Fomin, F., Heggernes, P., Kratsch, D., and Kucherov, G. Optimal linear arrangement of interval graphs. In *Mathematical Foundations of Computer Science 2006*. Springer, 2006, pp. 267–279.

[11] Cook, S. A. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing* (1971), ACM, pp. 151–158.

[12] Cygan, M., and Pilipczuk, M. Exact and approximate bandwidth. *Theoretical Computer Science 411*, 40 (2010), 3701–3713.

[13] Dasgupta, S., Papadimitriou, C. H., and Vazirani, U. *Algorithms*. McGraw-Hill, Inc., 2006, ch. Dynamic Programming.

[14] Díaz, J., Petit, J., and Serna, M. A survey of graph layout problems. *ACM Computing Surveys (CSUR) 34*, 3 (2002), 313–356.

[15] Downey, R., and Fellows, M. *Parameterized Complexity*. No. 1. Springer, 1999.

[16] Fellows, M. R., and Langston, M. A. Layout permutation problems and well-partially-ordered sets. In *Proceedings of the fifth MIT conference on Advanced research in VLSI* (1988), MIT Press, pp. 315–327.

[17] Garey, M. R., Graham, R. L., Johnson, D. S., and Knuth, D. E. Complexity results for bandwidth minimization. *SIAM Journal on Applied Mathematics 34*, 3 (1978), 477–495.

[18] Gavril, F. Some NP-complete problems on graphs. In *Proc. 11th Conf. on Information Sciences and Systems* (1977), pp. 91–95.

[19] Golumbic, M. C. *Algorithmic graph theory and perfect graphs*, vol. 57. Elsevier, 2004.

[20] Hart, P. E., Nilsson, N. J., and Raphael, B. A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on 4*, 2 (1968), 100–107.

[21] Heggernes, P., Lokshtanov, D., Mihai, R., and Papadopoulos, C. Cutwidth of split graphs, threshold graphs, and proper interval graphs. In *Graph-Theoretic Concepts in Computer Science* (2008), Springer, pp. 218–229.

[22] Heggernes, P., van't Hof, P., Lokshtanov, D., and Nederlof, J. Computing the cutwidth of bipartite permutation graphs in linear time. In *Graph Theoretic Concepts in Computer Science* (2010), Springer, pp. 75–87.

[23] Held, M., and Karp, R. M. A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial & Applied Mathematics 10*, 1 (1962), 196–210.

[24] Karger, D. R. A randomized fully polynomial time approximation scheme for the all-terminal network reliability problem. *SIAM review 43*, 3 (2001), 499–522.

[25] Leighton, F. T., and Rao, S. An approximate max-flow min-cut theorem for uniform multicommodity flow problems with applications to approximation algorithms. In *FOCS* (1988), pp. 422–431.

[26] Mahesh, R., Rangan, C. P., and Srinivasan, A. On finding the minimum bandwidth of interval graphs. *Information and Computation 95*, 2 (1991), 218–224.

[27] Petit, J. Addenda to the survey of layout problems. *Bulletin of EATCS 3*, 105 (2013).

[28] Roberts, F. S. Indifference graphs. *Proof techniques in graph theory 139* (1969), 146.

[29] Robson, J. M. Finding a maximum independent set in time o (2n/4). Tech. rep., Technical Report 1251-01, LaBRI, Université de Bordeaux I, 2001.

[30] Sipser, M. *Introduction to the Theory of Computation*. Cengage Learning, 2006.

[31] Thilikos, D. M., Serna, M., and Bodlaender, H. L. Cutwidth i: A linear time fixed parameter algorithm. *Journal of Algorithms 56*, 1 (2005), 1–24.

[32] Weinberger, A. Large scale integration of mos complex logic: A layout method. *Solid-State Circuits, IEEE Journal of 2*, 4 (1967), 182–190.

[33] Yannakakis, M. A polynomial algorithm for the min-cut linear arrangement of trees. *Journal of the ACM (JACM) 32*, 4 (1985), 950–988.