Generating tree-like graphs

Elisabeth Moldeklev
Institutt for Informatikk
Universitetet i Bergen

Norway

November 20, 2007

Contents

1 Introduction **4**
1.1 Graphs and graph algorithms in the real world 4
1.2 The web graph and search engines 6
1.3 Graph algorithm design techniques 9
1.4 Tree-like graphs 12

2 k-trees, k-branches and k-graphs **17**
2.1 k-trees and chordal graphs 17
2.2 k-branches and k-graphs 20

3 An algorithm generating k-graphs **23**
3.1 k-trees 23
3.2 Overview 27
3.3 Overview of the algorithms of the program 27
3.4 The main algorithm 28
3.5 Valid minimal separators 31
3.6 The random integer p 31
3.7 Adding new nodes to the graph 31
3.8 Help algorithms 34

4 The Implementation **36**
4.1 LEDA 36
4.2 The program 42
4.3 Overview of the user interface 42
4.4 Node attributes 47
4.5 Constructing the graph and error messages 50

5 Summary and conclusion **59**

6 Bibliography **61**

Abstract

Many graph algorithms are NP-hard for general graphs. But if we know that the graph is a tree, or tree-like, then many such NP-hard problems can be solved efficiently. To measure how close to being a tree a graph is we can use either the parameter treewidth or the closely related parameter branchwidth. The edge-maximal graphs of treewidth k are the k-trees. The algorithm for generating k-trees is trivial and well known. The k-branches are the edge-maximal graphs of branchwidth k. Since the branchwidth often is smaller than the treewidth, dynamic programming algorithms can often solve problems faster when using the branch-decomposition instead of the tree-decomposition. Until the results of [12] nobody knew how to generate k-branches. In this thesis we implement a simplified version of the fairly complicated algorithm from [12]. The result is a user-friendly program that interactively generates edge-maximal graphs of branchwidth k.

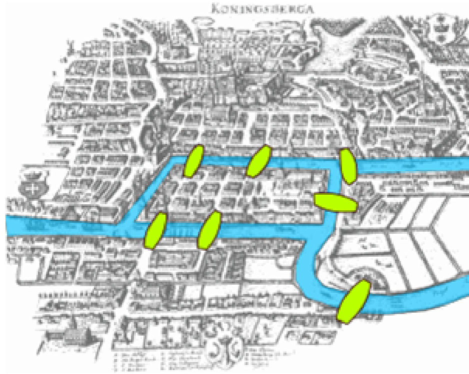
1 Introduction

In this chapter we first discuss graphs and graph algorithms in the real world, and give several short examples of this. We then briefly describe how a program called a crawler is used to discover parts of the Web graph. Since there are more than 11 billion web pages search engines are needed to find pages related to different topics. Google is one of the most popular search engines today. The algorithm it uses to rank the web pages is called PageRank and is described in chapter 1.2. We then list several algorithm design techniques and give examples for each of them.

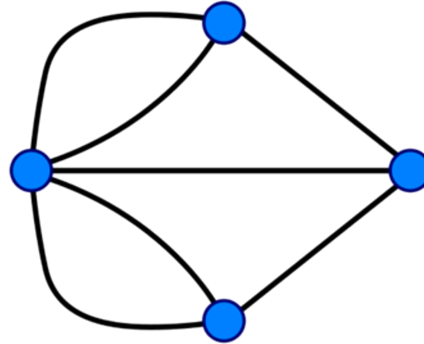
1.1 Graphs and graph algorithms in the real world

Graphs are often used to make facts clearer and more understandable. They visually represent relationships among data elements, and are used to solve problems in many fields. Graphs with weights assigned to edges can be used to solve problems such as finding the shortest path between two cities in a transportation network. To solve this problem, and other graph problems, we use graph algorithms. In this case one could use Dijkstra's shortest path algorithm. We can also represent airline schedules as a graph. The vertices would correspond to airports, and the edges would correspond to flights. If we store the time, cost and airline for each flight, Dijkstra's algorithm could again be used to find the cheapest route from one city to another. We can also use graphs to schedule exams, to represent various relationships between people, or to show which computers are connected via a communications network.

The history of graph theory starts with the seven bridges of Köningsberg problem that was solved by Euler. Köningsberg, Prussia (now Kaliningrad, Russia) was divided into four sections by the Pregel River. These four regions were connected by seven bridges (see Figure 1(a)). The question is whether it is possible to walk with a route that crosses each bridge exactly once. In 1736, Leonhard Euler proved that this was not possible. He formulated the problem in terms of graph theory. Each of the four sections of land was replaced by a vertex, and each bridge with an edge (see Figure 1(b)).



(a) Map of Köningsberg



(b) Graph representation

Examples of graph problems in the real world:

- The problem of determining whether a message can be sent between two computers using intermediate links can be studied with a graph model. Are two cities connected by railroad?

- Topological ordering is the problem of ordering n tasks where some tasks can be completed only after others have been finished. Suppose that a project is made up of 30 different tasks. Some tasks can only be started after others have been completed. In what order could the tasks be started? In what order could one choose computer science courses such that all prerequisites were satisfied?

- Suppose you have a lecture room and many people request to use the room for periods of time. A request starts at a time s and ends at time f. What is the maximum number of requests that can be accepted, so that no two requests overlap in time?

- Data elements that are stored in the cache are quicker to access than the elements that are on the hard disk. Since the cache can only store a limited number of data elements, one has to decide which elements that should be evicted from the cache when a new element is requested. In the 1960s, Les Belady showed that one should evict the item that is needed the farthest into the future.

- Shortest path between two points is a problem that arises frequently. What is the shortest path between two cities? What combinations of

flights has the smallest total flight time between two cities? What is the cheapest fare between two cities?

- Suppose we have a set of locations, and we want to build a communication network on top of them. Where should we place the links so that the network is connected and the total cost is as small as possible?

- Consider a computer network where the edges are links that can carry packets and the nodes are switches. The number of packets on each edge cannot exceed the capacity of the edge. What is the maximum number of packets that can flow through such a network?

- Graph coloring is the problem of coloring a map such that two adjacent regions never have the same color. Graph coloring has a variety of applications, e.g. scheduling final exams. How can the final exams at a university be scheduled so that no student has two exams at the same time?

- Suppose you have n friends, and some pairs of them don't get along. How many can you invite while avoiding fights?

- In which order can a salesman visit n cities exactly once and return to his starting point so that he travels the minimum total distance?

1.2 The web graph and search engines

Graphs can also be used to study the structure of the World Wide Web. The World Wide Web can be modeled as a directed graph. Each web page is represented by a vertex and if there is a hyperlink on page x pointing to page y, then the graph will have a directed edge from x to y. Each vertex is therefore a URL (Unique Resource Locator), and the outgoing edges of a vertex are the hypertext links contained in the corresponding page. The Web graph changes on an almost continual basis. New pages are added every day and others are removed. In July 2000, it was estimated that the web graph contained about 2.1 billions vertices and 15 billions edges ([2]). Because of the size of the Web graph it is impossible to get the whole graph. A method called crawl is used to discover parts of the Web graph. This method is also used by search engines for locating web pages. A crawler consists of a computer program which browses the World Wide Web in a methodical manner.

It starts at a predefined set of web pages and follows the links from each of the pages, and iterates the process from the newly discovered pages.

Since there are today more than 11 billion ([3]) web pages, one of the challenges is finding what pages you want. Search engines are a popular way to find web pages relating to a particular topic. Google is a popular search engine which uses graph theory to decide which web pages are most relevant. Google assigns a number called PageRank to every web page. Then when you do a search, it returns the pages related to the searched topic with the highest PageRank. PageRank is a link analysis algorithm that was developed at Stanford University by Larry Page and later Sergey Brin as part of a research project on a new kind of search engine. PageRank is one of several factors used by Google to determine best search results. It evaluates two things: how many links there are to a web page from other pages, and the quality of the linking sites. Links from pages with low credibility are worth less than links from pages with higher credibility. PageRank assigns a weight to each page of the World Wide Web. A link to a page counts as a vote of support. The PageRank of a page is defined recursively and depends on the number of incoming links to the page and the PageRank of the incoming links. A page that is linked to by many pages with high PageRank receives a high rank itself.

A web page is generally more important if many other web pages link to it. If a search engine only considered link popularity the rankings could easily be manipulated by pages which are only created to deceive search engines and which don't have any significance within the web. The incoming links do therefore not count equally in PageRank. The rank of a page is given by the rank of those pages which link to it. Their rank again is given by the rank of pages which link to them. Hence, the PageRank of a document is always determined recursively by the PageRank of other documents. ([18] and [19])

$$PR(A) = (1 - d) + d(PR(T_1)/C(T_1) + ... + PR(T_n)/C(T_n))$$

where
$PR(A)$ is the PageRank of page A.
$PR(T_i)$ is the PageRank of pages T_i which link to page A.
$C(T_i)$ is the number of outgoing links on page T_i. The more outgoing links a page T has, the less will the pages that T link to benefit from such a link.

d is a damping factor which can be set between 0 and 1. d is often set to 0.85.

Consider an example:
Assume the web only contains three pages A, B and C, where A links to both B and C, B links to C, and C links to A (see Figure 1). Let d be set to 0.85. We then get the following equations:
$PR(A) = 0.15 + 0.85PR(C)$
$PR(B) = 0.15 + 0.85(PR(A)/2)$
$PR(C) = 0.15 + 0.85(PR(A)/2 + PR(B))$
when we solve these equations we get the following PageRank values for the single pages:
$PR(A) = 1.1634$
$PR(B) = 0.6444$
$PR(C) = 1.1922$
We also see that the sum of all pages' PageRank is equal to the total amount of pages, $PR(A) + PR(B) + PR(C) = 3$. C receives higher PageRank than A since C is the only page that B links to. B has the lowest PageRank since it only has one incoming link from A, and since A has two outgoing links, B only receives half of $A's$ rank.

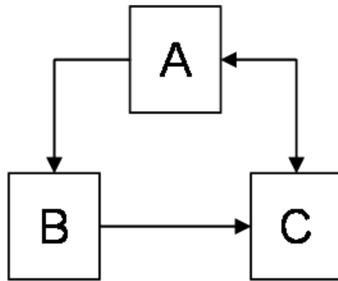


Figure 1: The web containing three web pages

Because of the size of the actual web, the PageRank values are computed iteratively. So each page is assigned an initial starting value, and the PageRanks of all pages are then calculated in several computations, until converging.

The Google Toolbar displays the PageRank as an integer between 0 and 10. For this display the PageRank has to be scaled. It is generally assumed that the scaling is not linear but logarithmic. The PageRank of a web page can

be checked at `http://www.checkpagerank.com/`

1.3 Graph algorithm design techniques

There exists various algorithm design techniques that are used for solving different problems. Let us briefly describe brute force, greedy algorithms, divide-and-conquer, dynamic programming and primal/dual techniques.

Brute force (also called exhaustive search) is the simplest of the design strategies. A brute force algorithm solves a problem in a straightforward and simple way. Such an algorithm can often end up doing far more work to solve a problem than a more clever algorithm might do. A brute force algorithm usually searches through all the possibilities and checks whether any of the possibilities satisfy the problem statement. It is simple to implement, and will always find the solution if it exists. The weakness is efficiency. When the problem size increases, the number of candidate solutions have a tendency to grow very quickly. Brute force search is therefore typically used when the problem size is limited.

Greedy algorithms make the choice that looks best at the moment, and never changes the choices made. They often fail to find the optimal solution since early decisions often prevent the algorithm from finding the best overall solution later. However, they are easy to implement and often give good approximations to the optimum. One example of a greedy algorithm is Dijkstra's algorithm for finding the shortest path between two vertices.

Divide-and-conquer algorithms divide a problem into several (often only two) subproblems. The subproblems are then solved recursively. The solutions for the subproblems are then combined to get a solution to the original problem. Mergesort is a typical example where the divide-and-conquer strategy is used. Mergesort sorts a list of numbers by first dividing the list into two equal parts, sorting each half separately by recursion, and then merging the two sorted halves.

The divide-and-conquer strategy is efficient when the subproblems don't overlap. However, when the subproblems overlap, the recursion does redundant work. In this case we use another strategy called dynamic programming.

Dynamic programming is nonrecursive, and instead of starting on the top and working it's way down (such as divide-and-conquer algorithms do) they start on the bottom and work their way up. The main idea is to solve several smaller (overlapping) subproblems, and record the solutions in a table so that each subproblem is only solved once. The last entry in the table will in the end contain the overall solution to the problem.

Primality/duality as used in network flow is yet another algorithm design technique. A flow network is a directed graph containing a single source node and an single sink node, and where each edge has a certain capacity. The primality/duality is reflected in the fact that a maximum flow is equal to a minimum cut, and this is used to design an efficient algorithm. This technique is used for finding the maximum flow in a network, and also for other problems, like the bipartite matching problem. A bipartite graph is an undirected graph whose node set can be partitioned into two parts with the property that every edge has one end in each part. A matching is a subset of the edges such that each node appears in at most one edge in the matching. The bipartite matching problem is that of finding the largest matching in a given graph.

Before giving some examples of problems that can and cannot be solved in polynomial time, we will give a definition of polynomial time, the complexity classes **P** and **NP**, and define **NP−complete** problems.

Definition 1. *An algorithm is polynomial if its running time is bounded by a polynomial in the size of the input.*

Definition 2. **P** *is the complexity class containing decision problems (problems with a yes-or-no answer) which can be solved in polynomial time.*

Definition 3. *A verifier for a problem A is an algorithm V that takes as input two arguments, a string w which is an input string to the decision problem, and a certificate string c. The algorithm V accepts $<w, c>$ for some c if and only if $w \in A$.*

The clique problem is to determine whether a graph G contains a k-clique. The input string w would contain G and k. The certificate given to the verifier would in this case consist of the nodes in the clique. The verifier

would then test if c is a set of k nodes in G and whether G contains all edges connecting nodes in c. If so, the verifier would return YES, otherwise NO.

Definition 4. **NP** *is the class of languages that have polynomial time verifiers.*

Definition 5. *A problem is NP-hard if all problems in NP are polynomial time reducible to it, even though it may not be in* **NP** *itself.*

Definition 6. *A language B is* **NP−complete** *if it satisfies two conditions:*

1. *B is in* **NP**, *and*
2. *Every A in* **NP** *is polynomial time reducible to B.*

Reducible means that for every problem A, there is a polynomial time algorithm which transforms $a \in A$ into instances $b \in B$, such that the answer to b is YES if and only if the answer to a is YES. So if we have a polynomial time algorithm for any of these **NP−complete** problems then we could solve all problems in **NP** in polynomial time.

Let us give some examples of problems that can be solved in polynomial time.

The shortest path problem can be solved in polynomial time. As mentioned earlier we can use Dijkstra's greedy algorithm. The algorithm maintains a set S of nodes that we have already determined the shortest path to from the start node s. In each iteration it adds the node that has one edge in S and where the value $min_{e=(u,v):u \in S} d(u) + l_e$ is minimized, where $d(u)$ is the value of the shortest path from the s to node u, and l_e is the weight of the edge e. There are $n-1$ iterations (one node v is added in each iteration). In each iteration one has to consider each node $v \notin S$, and go through all the edges between S and v to determine the minimum $min_{e=(u,v):u \in S} d(u) + l_e$. The running time would with this implementation be $O(mn)$. With a different data structure, we could reduce the running time to $O(mlogn)$ ([6]).

Computing a topological ordering in a directed acyclic graph, i.e. a total order on nodes such that all edges go from lower to higher numbered nodes, can also be done in polynomial time. Since we know that in every directed acyclic graph, there is a node with no incoming edges, the algorithm recursively finds this node v, orders it first, deletes it, and calls the algorithm with

$G - \{v\}$ as input.

We can construct a greedy algorithm that solves the scheduling problem that also runs in polynomial time. In each iteration this algorithm chooses the request that has the smallest finishing time, adds it to a set S, deletes it and all the other requests that are not compatible with this one. The algorithm terminates when there are no more requests left, and returns the set S.

We describe two problems that cannot be solved in polynomial time, namely k-coloring and the traveling salesman problem. Both of these problems are **NP−complete**, i.e. the problem is in **NP** and every other problem in **NP** is polynomial time reducible to it.

The problem of coloring a graph so that no two adjacent vertices are assigned the same color is an example of a problem that cannot be solved in polynomial time. Given a graph G and a bound k, does G have a k-coloring?

Traveling salesman is another **NP−complete** problem. In what order can n vertices be traversed so that the start node equals the end node, and each node (except the start node) is traversed exactly once. The goal is to find a traversal with the minimum total distance. This traversal is also called a Hamilton circuit after the famous Irish scientist William Rowan Hamilton.

1.4 Tree-like graphs

Several graph algorithms are NP-hard for general graphs. If we know certain properties about the graph, e.g. that the graph is a tree, then many NP-hard problems can be solved efficiently. Maximum-size independent set is an almost trivial example. An independent set is a set of vertices in a graph where no two vertices are adjacent. A maximum independent set is a largest independent set for a given graph. An algorithm that solves the maximum-size independent set on trees would include an arbitrary node v of degree 0 or 1, and delete both v and its neighbour. It would continue recursively until the graph is empty.

Many graph problems that are NP-hard can also be solved efficiently even when the graph has cycles. We use a parameter, called treewidth, to measure how close to being a tree the graph is. The treewidth says how many nodes

must be removed from the graph to split the graph into disconnected pieces. This allows us to implement dynamic programming algorithms. Another parameter which is closely related to treewidth is branchwidth. These two parameters are related by the inequalities: $branchwidth(G) \leq treewidth(G) + 1 \leq \lfloor 3/2\ branchwidth(G) \rfloor$. ([13])

Before we can define the treewidth we need to define a tree decomposition, whose width measures how close the graph is to being a tree.

Definition 7. *Given a graph $G = (V, E)$ a tree decomposition of G is a pair (X, T) where $X = \{X_1, \ldots, X_n\}$ is a collection of subsets of V, and T is a tree such that: ([6])*

- *Every node of G belongs to at least some bag X_i*
- *For every edge $e = (u, v)$ there is a bag X_i that contains both u and v.*
- *If vertex v is in both X_i and X_j, then all nodes X_w of the tree between X_i and X_j contain v.*

The width of a tree decomposition is the size of the largest set X_i minus one. The treewidth, $tw(G)$ of a graph G is the minimum width among all possible tree decompositions of G ([6]). Note that a tree has treewidth one.

Graphs of bounded treewidth often appear in practice. For example, it has been shown that GOTO-free C programs have control-flow graphs of treewidth 6. ([16])

Several problems that are NP-hard on general graphs can be solved efficiently for graphs that have treewidth (or branchwidth) bounded by a constant. The crucial property is that any bag X_i is a separator of the graph G. One example of this is maximum independent set, which can be solved in polynomial time with dynamic programming.

Let T be the tree decomposition of the graph G of width w. We call the nodes of T: $1, 2, ..., n$, and the bags of G: $X_1, ..., X_n$. We root the tree T at a bag node r, and let T_i denote the subtree rooted at i. Let G_i be the subgraph of G induced by the nodes in all bags associated with nodes of T_i. w_u is the weight of a node u, and $w(U)$ is the total weight of nodes in a subset U of

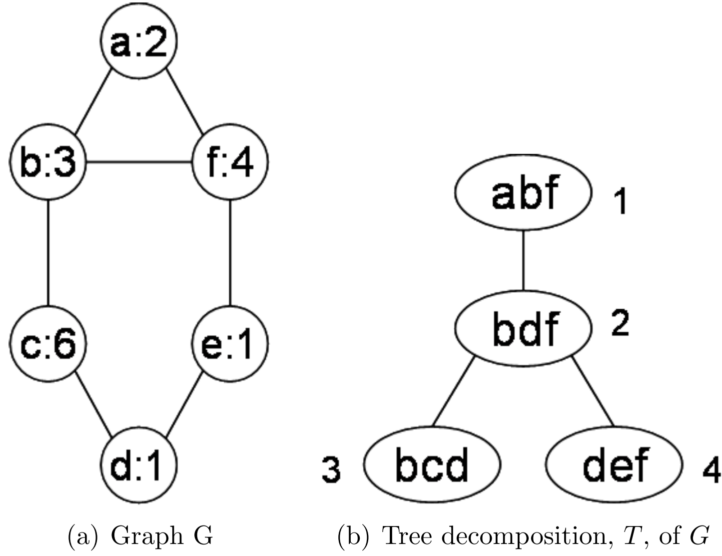
V.

The optimal independent set intersects the bag X_i in some subset U, but we do not know which set U is. We therefore go through all possibilities of this set U and calculate the maximum weight, $f_i(U)$, for all independent sets in the subgraph G_i which has intersection U with X_i. If the subset U is not an independent set we let $f_i(U) = -\infty$. There may be a total of 2^{w+1} subsets of i, so for each piece of the tree we go through all the subsets in a brute force manner.

We traverse the tree from the leaves and up. For an internal bag node i, we go through the different possibilities for what to do with the nodes in i. Since once this decision is fixed, the problems for the different subgraphs induced by nodes in subtrees below i become independent.

When i is a leaf, $f_i(U)$ is equal to $w(U)$ for each independent set $U \subseteq i$. When i has children i_1, ..., i_d the value $f_i(U)$ is given by the recurrence: ([6] page 583)
$f_i(U) = w(U) + \sum_{j=1}^{d} max\{f_{i_j}(U_j) - w(U_j \cap U) :$
$\quad U_j \cap i = U \cap i_j$ and $U_j \subseteq i_j$ is independent.$\}$
We will end this section with a rather trivial example of the maximal independent set problem.

Let $G = (V, E)$ be the graph in Figure 2(a) with the tree decomposition T (Figure 2(b)). Note that a has weight 2, b weight 3, c weight 6, d and e weight 1 and f weight 4.



(a) Graph G (b) Tree decomposition, T, of G

The values for the two leaves 3 and 4 are as follows:

(a) Values for leaf 4 of T. Name of array: $A4$

	4	$f_4(U)$
0	$\{d\}$	1
1	$\{e\}$	1
2	$\{f\}$	4
3	$\{de\}$	$-\infty$
4	$\{df\}$	5
5	$\{ef\}$	$-\infty$
6	$\{def\}$	$-\infty$
7	$\{\}$	0

(b) Values for leaf 3 of T. Name of array: $A3$

	3	$f_3(U)$
0	$\{b\}$	3
1	$\{c\}$	6
2	$\{d\}$	1
3	$\{bc\}$	$-\infty$
4	$\{bd\}$	4
5	$\{cd\}$	$-\infty$
6	$\{bcd\}$	$-\infty$
7	$\{\}$	0

Table 1: Values for the two leaves of T

Note that the values used to calculate $f_2(U)$ are taken from Table $A3$ and Table $A4$. The values for the internal node 2 is given by the above recurrence and are as follows:

	2	$w(U)$	max for child 3	max for child 4	$f_2(U)$
0	$\{b\}$	3	0	$A4[1]$	4
1	$\{d\}$	1	0	0	1
2	$\{f\}$	4	$A3[1]$	0	10
3	$\{bd\}$	4	0	0	4
4	$\{bf\}$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
5	$\{df\}$	5	0	0	5
6	$\{bdf\}$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
7	$\{\}$	0	$A3[1]$	$A4[1]$	7

Table 2: Values for the internal node 2 of T. Name of array: $A2$

The values for the root node 1 is given by the same recurrence and are as follows:

	1	$w(U)$	max for child 2	$f_1(U)$
0	$\{a\}$	2	$\max\{A2[1], A2[7]\}$	9
1	$\{b\}$	3	$A2[1]$	4
2	$\{f\}$	4	$\max\{A2[1], A2[2]-4\}$	10
3	$\{ab\}$	$-\infty$	$-\infty$	$-\infty$
4	$\{af\}$	$-\infty$	$-\infty$	$-\infty$
5	$\{bf\}$	$-\infty$	$-\infty$	$-\infty$
6	$\{abf\}$	$-\infty$	$-\infty$	$-\infty$
7	$\{\}$	0	$\max\{A2[1], A2[7]\}$	7

Table 3: Values for the root node 1 of T

The maximal independent set for the graph G therefore has weight 10 and consists of the vertices: c and f.

The remaining chapters of this thesis are organized as follows. In chapter 2 we give an overview of k-trees, k-branches and k-graphs, where k-trees are the edge-maximal graphs of treewidth k, and k-branches are the edge-maximal graphs of branchwidth k. The k-graphs are a superclass of k-branches. The main part of this thesis is an interactive program that generates k-graphs. Chapter 3 describes the different algorithms of the program, while chapter 4 discusses the implementation of the program. Chapter 4 also give a short overview of the C++ class library for data types and algorithms called LEDA.

2 k-trees, k-branches and k-graphs

For a class C of graphs, we say that graph $G \in C$ is edge-maximal if adding any edge to G will result in a graph not belonging to C. In this section we introduce k-trees, the edge-maximal graphs of treewidth k, and k-branches, the edge-maximal graphs of branchwidth k. The k-graphs are a superclass of k-branches which are more easy to handle algorithmically.

2.1 k-trees and chordal graphs

We consider simple undirected and connected graphs $G = (V, E)$ with vertex set and edge set.

Given a graph $G = (V, E)$ and a subset $U \subseteq V$, the subgraph of G induced by U is the graph $G' = (U, D)$, where $(u, v) \in D$ iff $u, v \in U$ and $(u, v) \in E$.

A clique is a set of vertices that induce a complete subgraph of G. A maximal clique is a clique which is not a subset of any other clique.

A set of vertices $S \subset V$ is a separator if the subgraph of G induced by $V - S$ is disconnected. The set S is a uv-separator if u and v are in different connected components of $G[V - S]$. A uv-separator S is minimal if no subset of S separates u and v.

Definition 8. *k-trees are defined recursively:*

- *The complete graph on k vertices is a k-tree.*
- *A k-tree G with $n + 1$ vertices ($n \geq k$) can be constructed from a k-tree H with n vertices by adding a vertex adjacent to all vertices of a k-clique of H.*

Theorem 1. *A graph G has treewidth at most k iff G is a subgraph of a k-tree. ([8])*

A chord is an edge joining two nonconsecutive vertices of a cycle.

A graph is chordal if every cycle of length > 3 has a chord.

Theorem 2. *A graph G is chordal if and only if every minimal separator of G is a clique. see [4]*

Proof. Adapted from [4]
$\Rightarrow$: Let $G = (V, E)$ be chordal and let S be a minimal separator of G. Let x and y be any two vertices in S. Let a and b be the vertices for which S is a minimal ab-separator, and let A and B be the connected components of $G[V - S]$ containing respectively a and b. There must exist a path between x and y through vertices belonging to A. Let p_1 be the shortest such path, and let p_2 be the shortest such path in B. p_1 and p_2 joined together create a cycle of length at least 4. Since G is chordal, this cycle must have a chord. Since no edges exist between vertices in A and vertices in B, the edge (x, y) must be present and a chord of the cycle. (Figure 2)
$\Leftarrow$: Let G be a graph where each minimal separator is a clique. Assume that G is not chordal and let $w, x, y, z_1, ..., z_k, w$ be a chordless cycle of length at least 4 in G ($k \geq 1$). Any minimal wy-separator of G must contain x and at least one z_i for $1 \geq i \geq k$. Since all minimal separators are cliques, the edge (x, z_i) must belong to G contradicting that the cycle is chordless. (Figure 3) $\square$

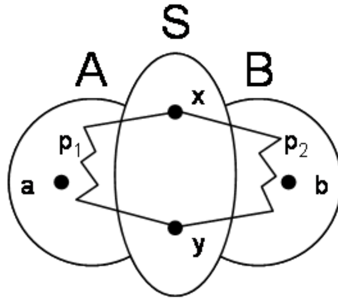


Figure 2: P_1 joined together with P_2 create a cycle of length at least 4. Since G is chordal, this cycle must have a chord, namely xy.

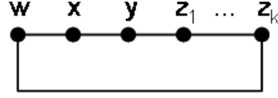


Figure 3: Every minimal wy-separator must contain x and at least one z_i. Since all minimal separators are cliques, (x, z_i) must belong to G

Lemma 1. *k-trees are chordal. see [4]*

Definition 9. *A vertex v is said to be simplicial if all its neighbours are adjacent to each other.*

Note that any set of connected simplicial vertices form a clique.

Lemma 2. *A chordal graph is either complete or has at least two nonadjacent simplicial vertices. (see [4])*

Proof. The proof has been adapted from [4].
Let G be a chordal graph which is not complete. The proof is by induction on the number of vertices n. The base case is when $n = 2$ and G has two isolated vertices that are both simplicial.
Let $n > 2$ and assume that the lemma holds for all such graphs with less than n vertices. Let a and b be two nonadjacent vertices of G, let S be a minimal ab-separator, let $G[A]$ be the connected components of $G[V - S]$ that contains a, and $G[B]$ be the connected components that contain b. $G[A \cup S]$ is a chordal graph with less than n vertices. So the lemma is valid and $G[A \cup S]$ is either complete (every vertex of A is simplicial) or has at least two nonadjacent simplicial vertices one of which must belong to A since S is a clique. G has therefore at least one simplicial vertex that belongs to A. With the same argument, G has another simplicial vertex in B, and the proof is complete. □

As shown in Theorem 1 it is well-known that the subgraphs of k-trees are exactly the graphs of treewidth k. In other words, the k-trees are the edge-maximal graphs of treewidth k. Apart from the definition of k-trees given in Definition 8 k-trees can also be characterized by the following 3 conditions:

Theorem 3. *A graph G is a k-tree iff ([13])*

1. *G is chordal*

2. *Every minimal separator of G has size k*

3. *Every maximal clique of G has size $k+1$*

Paul and Telle ([13]) studied also the edge-maximal notion for branchwidth.

2.2 k-branches and k-graphs

As mentioned earlier, there is an alternative parameter measuring how tree-like a graph is, that is similar to treewidth, called branchwidth. The two parameters are related by the inequality $branchwidth(G) \leq treewidth(G) + 1 \leq \lfloor 3/2\ branchwidth(G) \rfloor$. For many graphs branchwidth is smaller than treewidth and this means that the dynamic programming algorithms to solve optimization problems are faster when using the so-called branch-decompositions [1]. Let us start with the standard definition of branchwidth that however we will not be using in this thesis.

Let G be a graph with node set $V(G)$ and edge set $E(G)$. Let T be a tree having $|E(G)|$ leaves in which every nonleaf node has degree 3. Let μ be a bijection from the edges of G to the leaves of T. The pair (T, μ) is called a branch decomposition of G. Removing an edge of T, e, partitions the edges of G into two subsets A_e and B_e. The middle set of e, denoted by m$id(e)$, is the set $V(A_e) \cap V(B_e)$. The width of a branch decomposition (T, μ) is the maximum cardinality of the middle sets over all edges in T. The branchwidth of G, denoted by $bw(G)$, is the minimum width over all branch decompositions of G.

Recently Paul and Telle gave an alternative characterization of branchwidth.

Definition 10. *A k-troika (A, B, C) of a set X are 3 subsets of X such that $|A| \leq k, |B| \leq k, |C| \leq k$, and $A \cup B = A \cup C = B \cup C = X$. (A, B, C) respects $S \subseteq X$ if $S \subseteq A$ or $S \subseteq B$ or $S \subseteq C$.*

For example, Let $A = \{0, 1, 2\}$, $B = \{1, 2, 3\}$, $C = \{0, 1, 3\}$. (A, B, C) is a 3-troika of $X = \{0, 1, 2, 3\}$. X does not have a 2-troika.

Theorem 4. *A graph G has branchwidth at most k iff G is a subgraph of a chordal graph H and every maximal clique X of H has a k-troika respecting the minimal separators of H contained in X. (see [13])*

Definition 11. *A graph is a k-branch if it has branchwidth k and is edge-maximal, i.e. adding any edge will increase its branchwidth.*

The k-branches can be characterized by five conditions.

Theorem 5. *A graph G is a k-branch iff ([13])*

1. *G is chordal*
2. *Every minimal separator of G has size k*
3. *Every maximal clique of G has a k-troika respecting minimal separators*
4. *G has at least $\lfloor 3(k-1)/2 \rfloor + 1$ vertices.*
5. *The maxclique-minsep tree-decomposition of G has no mergeable subtree.*

The first two are common with Theorem 3 for k-trees. They are chordal and their minimal separators have size k only. The third condition can also be compared to the third condition for k-trees, namely that every maximal clique of G has size $k+1$. The fourth condition is a size constraint. There is also a fifth condition, namely "The maxclique-minsep tree-decomposition of G has no mergeable subtree" which we will not consider in this thesis, as it is quite technical and does not have a main practical importance. Thus, the main graphs that we study are those that satisfy the first 4 conditions of k-branches. We call these the k-graphs.

Definition 12. *G is a k-graph if and only if:*

1. *G is chordal*
2. *Every minimal separator of G has size k*
3. *Every maximal clique of G has a k-troika respecting minimal separators*
4. *G has at least $\lfloor 3(k-1)/2 \rfloor + 1$ vertices.*

Lemma 3. *The maximal number of nodes that a maximal clique X can contain and still have a k-troika respecting its minimal separators is $k + \lfloor k/2 \rfloor$.*

Proof. Let $|X| = k + z$. A k-troika (A, B, C) of the set X can, without loss of generality, be built as follows. A will contain the first k nodes. Since the union of A and B should equal X, B has to contain the z last nodes, and $k - z$ of the nodes that A already contains (assuming $z \leq k$). To ensure that $A \cup C = B \cup C = X$, C has to contain the nodes that are in X, but not in the intersection of A and B. Since the size of each set has to be less than or equal to k:

$$\begin{aligned} |X| - |A \cap B| &\leq k \\ k + z - (k - z) &\leq k \\ 2z &\leq k \\ z &\leq \lfloor k/2 \rfloor \end{aligned}$$

□

Note that any k-branch is a k-graph and also that a k-graph has branchwidth k.

3 An algorithm generating k-graphs

The main part of this thesis is an implementation of an interactive algorithm to generate k-graphs. The need for such an algorithm arose because of the recent interest in branchwidth. As we show in section 3.1 the interactive algorithm to generate edge-maximal graphs of treewidth k is trivial, but until the results of [12] nobody knew how to do this for k-branches. The interactive algorithm given in [12] is very complicated. The need for a simpler algorithm was clear, and the decision taken in this thesis was to drop condition 5 of Theorem 5 which turned out to make it much easier to generate the resulting superclass of k-branches called k-graphs. Still, the algorithm is non-trivial and much more complicated than the one generating k-trees.

3.1 k-trees

Let us first describe the similar but almost trivial algorithm (see Algorithm 1) to generate k-trees based on Definition 8. The algorithm starts by constructing a (k+1)-clique. In each iteration of the for loop it selects a random k-clique, and constructs a new (k+1)-clique by adding a new node adjacent to all nodes of the selected k-clique.

Algorithm 1: construct_k-trees(graph G, int k, int n)

input : empty graph G, positive integer k, positive integer n
output: A k-graph G consisting of n cliques

Construct a (k+1)-clique
for $i = 2$ **to** n **do**
  Choose a random k-clique C in G
  Construct a new (k+1)-clique by adding a new node adjacent to all nodes of C
end

Figure 4 shows a 2-tree consisting of 8 nodes while Figure 5 shows the same graph drawn in a different manner. The k-graph algorithm (see Algorithm 2) constructs k-graphs in a similar way to how Algorithm 1 constructs k-trees. Figure 6 shows how the program draws a 3-graph. As in Figure 5 maximal cliques are represented by grey rectangles and minimal separators are represented by green ellipses.

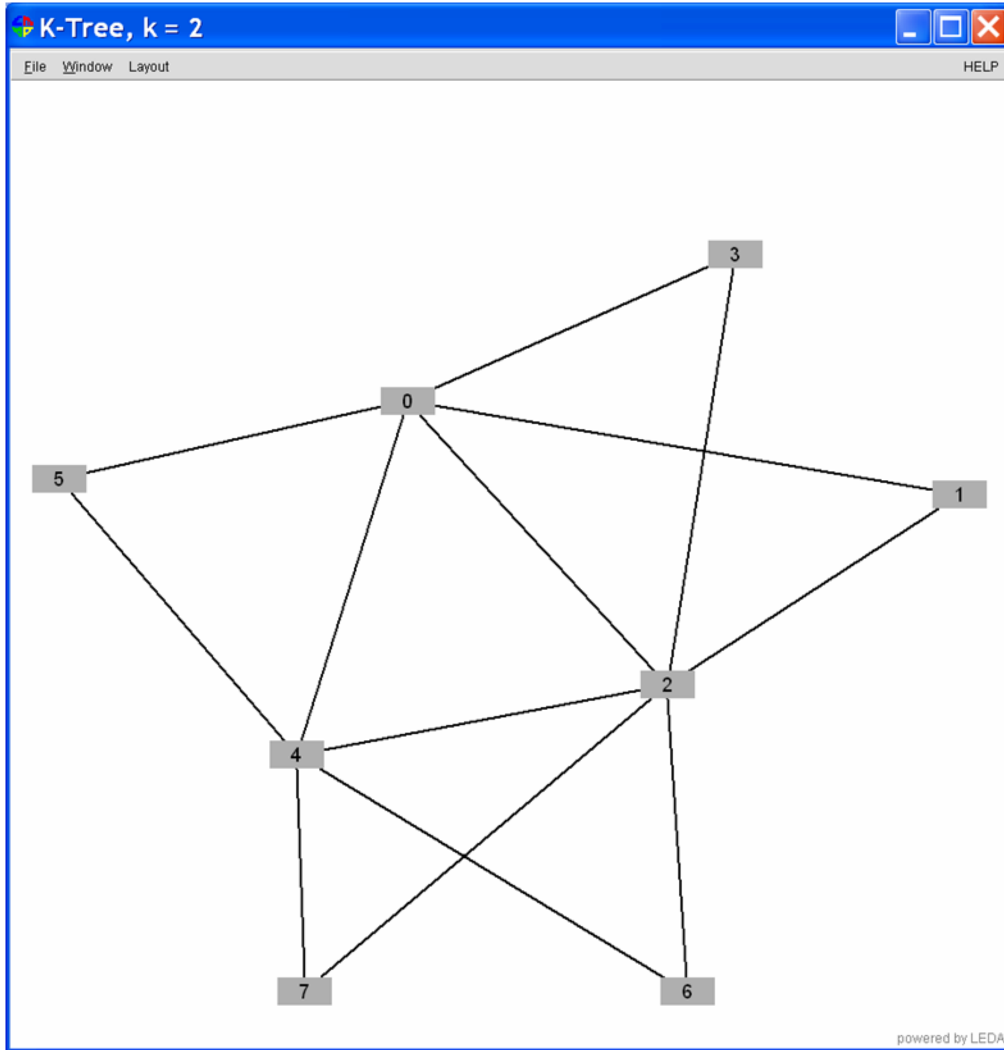


Figure 4: 2-tree with 8 nodes. Note that $\{0, 1, 2\}$ and $\{0, 2, 4\}$ are maximal cliques sharing the minimal separator $\{0, 2\}$

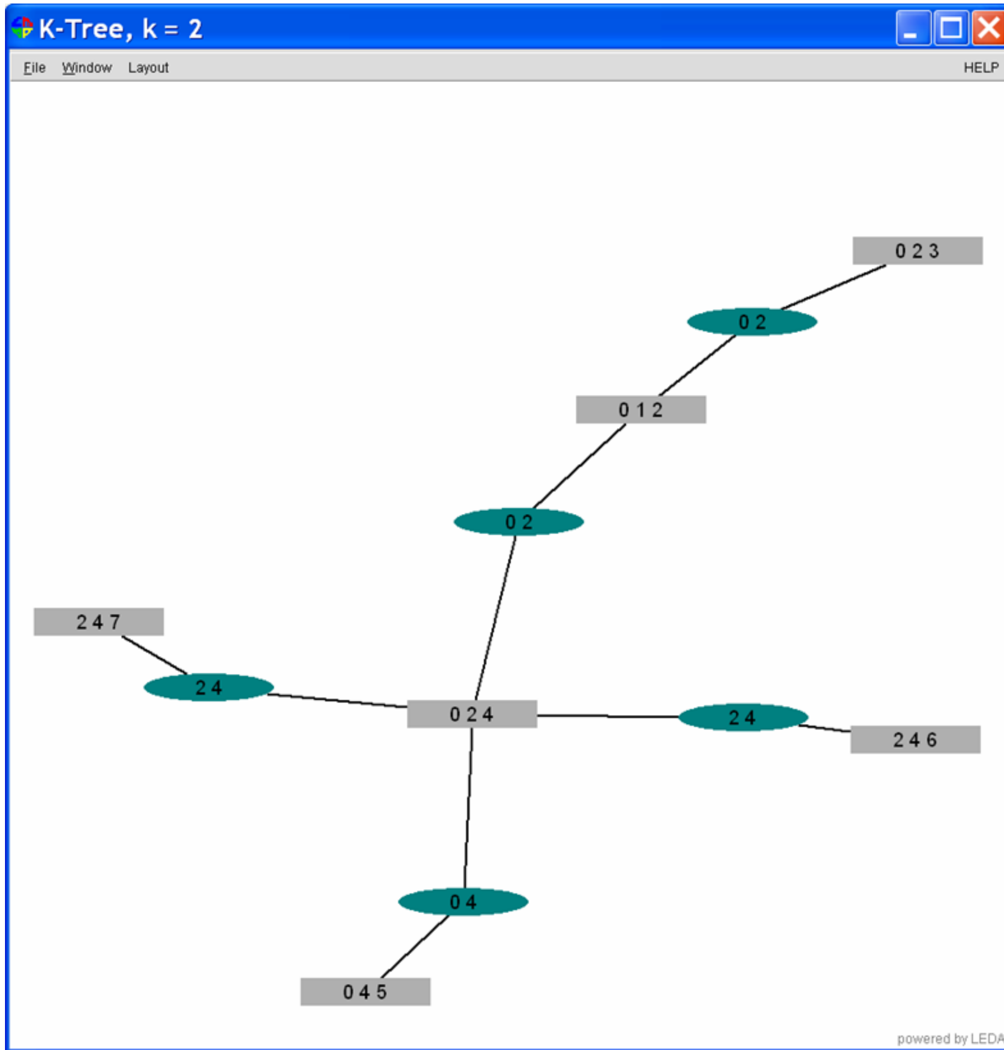


Figure 5: Same graph as in Figure 4. Maximal cliques are represented by grey rectangles and minimal separators are represented by green ellipses.

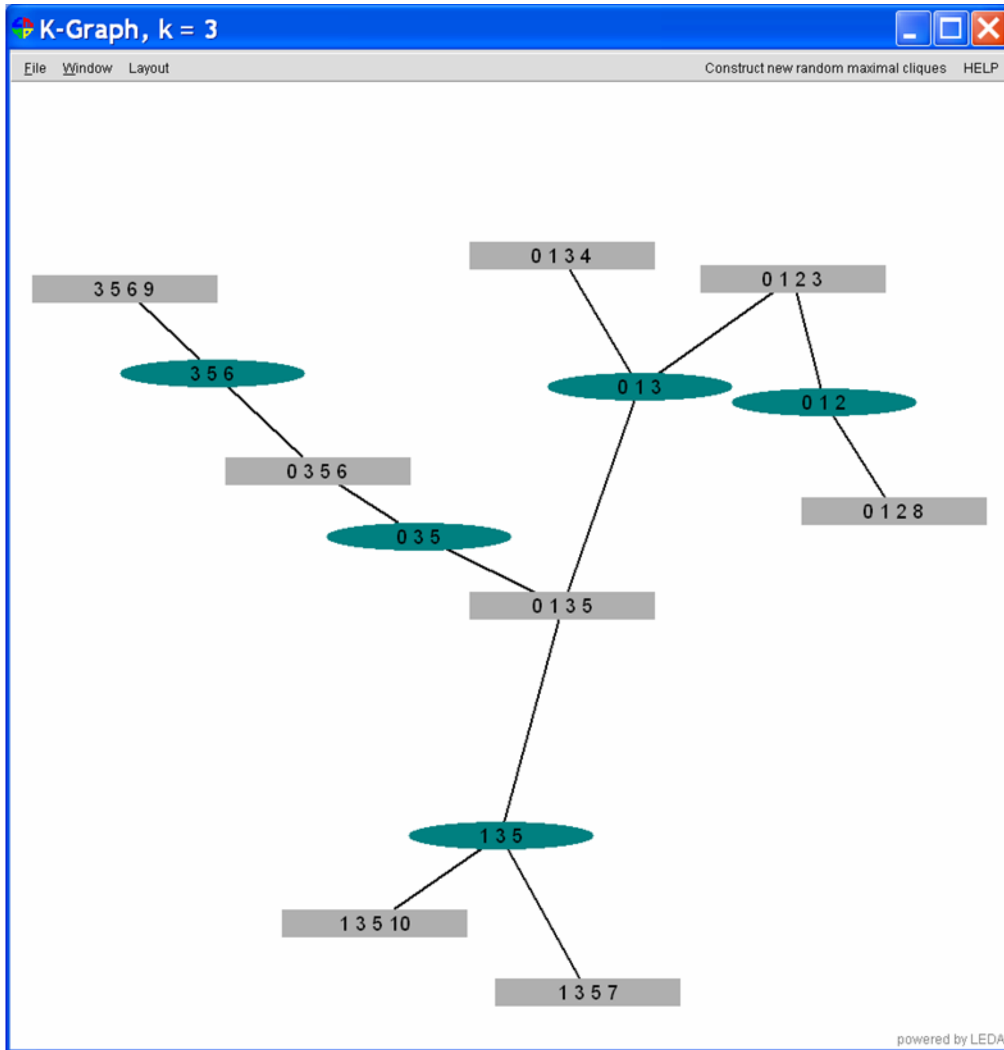


Figure 6: 3-graph with 8 maximal clique nodes

Note that the main algorithm for interactively generating k-graphs is given on separate pages, and that the actual implementation of these algorithms are discussed in the next section. When discussing the algorithm in detail we will use the line numbers listed on these separate pages. We will now describe the two main algorithms of the program behind the implementation, and briefly three help algorithms.

3.2 Overview

The program starts by asking the user how many nodes the graph should contain. The user can thereby choose if she would like to start with an empty graph and add one and one node, or if she would like the program to generate a random k-graph. After the random k-graph has been generated the user can click on nodes of the graph and construct new neighbours. The program can also at any point of the execution generate random cliques.

The program can therefore be used as a interactive way of learning about k-graphs. If the user constructs nodes one at the time she will be informed whether or not the new node is valid. If it is not valid she will receive an error message that explains why this node cannot be constructed. There is also a "help" button that the user can click on which explains how to create nodes and which conditions valid k-graphs must satisfy.

3.3 Overview of the algorithms of the program

Algorithm 2 will be called if the user chooses to start with a graph containing more than one node. The algorithm receives a graph G consisting of at least one node as input. It then constructs a graph that satisfies the three first conditions of Lemma 12. When the number of nodes exceeds $\lfloor 3(k-1)/2 \rfloor$ the fourth condition of Lemma 12 is also satisfied and the graph is a valid k-graph. If the user chooses to start with n nodes, where $n > 0$, the help algorithm *construct_x* is called before Algorithm 2, so that the graph given as input consists of a single node. Algorithm 2 is also called when the user clicks on the "Construct new random maximal cliques" button. The algorithm adds as many new maximal cliques as the user choose (with an upper limit of 100).

Algorithm 3 is called every time the user clicks on a node of the graph. If the node was a maximal clique, both a minimal separator and a maximal clique

is added to the graph. If she clicked on a minimal separator only a maximal clique is added.

Both Algorithm 2 and Algorithm 3 uses the help algorithms *construct_sep* and *construct_y*. *construct_x* is only used to construct the first node of the graph. *construct_sep* is used to construct all the minimal separators of the graph, while *construct_y* constructs all the maximal cliques of the graph (except the first).

3.4 The main algorithm

Algorithm 2 constructs a random graph G that satisfies the 4 conditions of Definition 12. In Line 2 it enters a for loop where a new maximal clique is constructed in each iteration, until n maximal cliques has been added to G. In each iteration a random maximal clique x is chosen and either both a new maximal clique and a new minimal separator are constructed, or only a new maximal clique is constructed adjacent to an already existing minimal separator. This depends on the random variable p.

Line 6 adds a random minimal separator and a new maximal clique containing between 1 and $k/2$ new nodes to the graph. Only the first node of the graph can have degree 0.

When the maximal clique node has degree 1 (Line 9) the algorithm will either only construct a new maximal clique adjacent to the already existing minimal separator, or construct both a new maximal clique and minimal separator. If a new minimal separator is to be created it has to satisfy the third condition of Lemma 12. The set given to *construct_sep* will therefore consist of the nodes that are in the maximal clique, but not in the already existing minimal separator (Line 14).

Line 17 is similar to the case when the degree of x was 1. The set given to *construct_sep* has to contain the nodes that are in the maximal clique x minus the nodes that are in the intersection of the two minimal separators, u and v, adjacent to x (Line 20).

If the maximal clique already has degree 3 (Line 25), then a new minimal

separator cannot be created. A new maximal clique will then be added next to one of the already existing minimal separators.

The last if statement (Line 30) checks whether the graph is a valid k-graph or not, i.e. if it contains $\lfloor 3(k-1)/2 \rfloor + 1$ or more nodes. If it does a message (see Figure20) is shown informing the user that the graph is now a valid k-graph. This message is only displayed once in each run of the program.

Algorithm 2: ConstructGraph(graph G, int k, int n)

```
   input : k-graph G consisting of at least one node, positive integer k,
           positive integer n
   output: A k-graph G consisting of n new maximal cliques
   begin
2      for i = 1 to n do
           Choose a random maximal clique x in G
           Let p be a random integer between 0 and 2
           switch deg(x) do
6              case deg(x) = 0  /* adds new maxclique and minsep */
                   construct_sep(G, x, <empty set>)
                   construct_y(G, sep, random[1, k/2])
9              case deg(x) = 1
                   Let u denote the neighbour of x (minimal separator).
                   if p = 0 then                /* adds new max clique */
                       construct_y(G, u, random[1, k/2])
                   else          /* adds new maxclique and minsep */
14                     construct_sep(G, x, <nodes in x − nodes in u>)
                       construct_y(G, sep, random[1, k/2])
                   end
17             case deg(x) = 2
                   Let u and v denote the two neighbours of x.
                   if p = 0 then /* adds new maxclique and minsep */
20                     construct_sep(G, x, <nodes in x − [u ∩ v] >)
                       construct_y(G, sep, random[1, k/2])
                   else                         /* adds new max clique */
                       construct_y(G, random[u, v], random[1, k/2])
                   end
25             case deg(x) = 3                  /* adds new max clique */
                   Let u, v and w denote the three neighbours of x.
                   construct_y(G, random[u, v, w], random[1, k/2])
           end
       end
30     if G consists of ⌊3(k − 1)/2⌋ + 1 or more nodes then
           Display a panel informing the user that the graph is now a
           valid k-graph.
       Display graph
   end
```

3.5 Valid minimal separators

Algorithm 2 will only construct valid maximal cliques and minimal separators. Maximal cliques will always contain the k nodes that are in the adjacent minimal separator, and between 1 and $k/2$ new nodes (see Lemma 3). When the maximal clique has one neighbour, the new minimal separator will always contain the nodes that are in x, but not in the already existing minimal separator (Line 14). When the degree of the maximal clique is two, the new minimal separator will always contain the nodes that are in x minus the nodes that are in the intersection of the two neighbours of x (Line 20). The remaining elements are chosen at random. Since the conditions of Lemma 12 are satisfied, all minimal separators that are constructed are valid.

3.6 The random integer p

In each iteration of the for loop (Line 2) a new maximal clique is constructed. An already existing maximal clique node x is chosen in G. The probability of constructing both a minimal separator and a maximal clique depends on the degree of the maximal clique, x, i.e. the number of existing minimal separators.

- if $deg(x) = 0$ then 1
- if $deg(x) = 1$ then $\frac{2}{3}$
- if $deg(x) = 2$ then $\frac{1}{3}$
- if $deg(x) = 3$ then 0

3.7 Adding new nodes to the graph

The user can construct new nodes by left clicking on an already existing node in the graph. Algorithm 3 is called every time the user clicks on a node in the graph. If the user has clicked on a maximal clique the algorithm will construct both a new minimal separator and a new maximal clique, but if

the user clicked on a minimal separator (Line 1) only a new maximal clique will be constructed (by the algorithm *construct_y*). The size of the maximal clique must be between $k+1$ and $k+k/2$, to ensure that condition three of Lemma 12 is satisfied.

If the maximal clique x already has three neighbours (Line 4), the user is asked whether she would like to construct a new maximal clique adjacent to one of the three existing minimal separators (see Figure 24).

In line 6 the user is asked to choose k of the nodes from the maximal clique x. The minimal separator will consist of these k nodes. The program displays all the nodes from the maximal clique x as boolean boxes (see Figure 21). Line 7 checks whether the user chooses exactly k nodes. If she chose too many or too few nodes, error massages are displayed (see Figure 23 and Figure 22) informing the user that the second condition of Lemma 12 has not been satisfied.

Line 9 checks whether or not the minimal separator already exists. If it does she is asked if she would like to construct a new maximal clique adjacent to the already existing minimal separator (see Figure 25).

Line 11 ensures that condition three of Lemma 12 is satisfied. It checks whether this new minimal separator will form a valid k-troika with the maximal clique's other minimal separators. This is done by checking that the union between the new minimal separator and each of the already existing ones equals the nodes of the maximal separator. The program can generate three different error messages (see Figure 26, Figure 27 and Figure 28) depending on the degree of the maximal clique and whether the union of new minimal separator and on or both of the existing minimal separators did not equal the maximal clique.

In Line 13 the minimal separator is created. The user is then asked how many nodes the new maximal clique should consist of (see Figure 31 and 32). According to Lemma 3 the maximal number of new nodes a maximal clique can contain is $\lfloor k/2 \rfloor$. In Line 15 a new maximal clique node is created consisting of between $k+1$ and $k+k/2$ nodes. If either of the two if statements, Line 4 or Line 9, evaluates to true then only a new maximal clique is constructed.

Instead of selecting the nodes oneself, the user can click on a "random" button. In this case the program will automatically construct a valid minimal separator consisting of k nodes, and a valid maximal clique of random size.

The last if statement (Line 16) checks whether the graph is a valid k-graph or not, i.e. if it contains $\lfloor 3(k-1)/2 \rfloor + 1$ or more nodes. If it does a message is shown informing the user that the graph is now a valid k-graph. This message is only shown once in each run of the program, so if it has already been displayed, the if statement will be ignored.

Algorithm 3: Construct_new_minimal_separator

input : graph G, positive integer k, node x
output: A new maximal clique and a new minimal separator

```
1  if x is a minimal separator then
       User chooses number of new nodes the maximal clique should
       contain, between 1 and k/2
       return construct_y (G, x, number of nodes)
4  if degree of maximal clique x is three then
       Ask the user if she would like to construct a new maximal clique
       adjacent to one of the already existing neighbours of x.
6  User chooses k of the nodes from x which will be the new minimal
   separator, sep.
7  if the user either chose more or less than k nodes then
       return error message
9  if sep already exists then
       Ask the user if she wants to construct a new maximal clique
       adjacent to sep.
11 if union(sep, w) ≠ x, ∀ w neighbour of x then
       return error message
   /* A minimal separator and a maximal clique is added     */
13 construct_sep (G, x, <nodes chosen by user>)
   User chooses number of new nodes the maximal clique should contain,
   between 1 and k/2
15 construct_y (G, number of nodes)
16 if G consists of ⌊3(k − 1)/2⌋ + 1 or more nodes then
       Display a panel informing the user that the graph is now a valid
       k-graph.
   Display graph
```

3.8 Help algorithms

All nodes are constructed by either of the three help algorithms *construct_x*, *construct_sep* and *construct_y*.

Algorithm 4 is used to construct the first maximal clique node of the graph. It is given the size of the node as input.

All minimal separators are generated by Algorithm 5. The algorithm takes a maximal clique node x and a set of nodes (integers) as input. To satisfy condition two of Lemma 12 the size of the minimal separator has to be k. If the size of the set is less than k, random nodes are added from the maximal clique node x. Since elements never appear more than once in sets the same node will never be added twice to the minimal separator. Before returning the minimal separator, the algorithm constructs an edge between the maximal clique node x and the new minimal separator node *sep*.

Algorithm 6 is used to generate all the maximal cliques of the graph, except the first one. It receives the minimal separator node *sep* and an integer n as input. The new maximal clique node will contain the nodes that are in the minimal separator and n new ones. An edge between the minimal separator and the new maximal clique is also constructed.

Algorithm 4: construct_x(graph G, int n)

```
begin
    Add new maximal clique node x of size n to G
end
```

Algorithm 5: construct_sep(graph G, node x, set S)

```
begin
    while size of S is less than k do
        Add random node from x to S
    end
    Add new minimal separator node sep containing the nodes in S to
    G
    Add edge (x, sep)
    return sep
end
```

Algorithm 6: construct_y(graph G, node *sep*, int n)

```
begin
    Add new maximal clique node y consisting of the k nodes in sep
    and n new nodes to G
    Add edge (y, sep)
end
```

4 The Implementation

In this section we will talk about the graph drawing package LEDA that was used for the implementation. We also describe the user interface and the different messages that are given to the user. Note that the program code is given as a separate file.

4.1 LEDA

LEDA stands for Library of Efficient Data Types and Algorithms. It is a C++ class library for data types and algorithms. LEDA has been used in such diverse areas as code optimization, robot motion planning, traffic scheduling, machine learning and computational biology [9]. Some of the data types that were used in the program are *random source*, *node_map*, *list*, *set*, *graph* and *GraphWin*.

The stream of integers generated by a random source is only pseudo-random. It is generated by a seed that can either be set by the user or it can be generated by the internal clock. By using the same seed, one can create the same random sequence because the algorithm that creates the random numbers always creates the next number from the previous one. Using the same seed can be very useful during debugging since the program in that case always takes the same decisions, but since this program was not meant to be deterministic the seed is created by the internal clock. The graphs that are generated will therefore vary from time to time, even though the k value and the number of maximal cliques are the same. The built in data type *random_source* is used by the program to:

- Decide whether only a new maximal clique or both a new maximal clique and minimal separator should be constructed.
- Generate a random point on the screen.
- Decide the number of new nodes in a maximal clique.
- Decide which nodes should be included in the new minimal separator from the maximal clique.

The data type *node_map* $< E >$ can be used to associate additional information of type E with the nodes of the graph. The program uses a *node_map*

with the data type $list < int >$. The data type *node_map* is dynamic, so when a new node is added to the graph it is automatically added to the *node_map*. Each list includes the nodes (represented as integers) that the maximal clique or minimal separator contains. The initialization of a *node_map* is in constant time, and access is in expected constant time [21]. Node maps use hashing to associate information to nodes.

In the program we originally wanted to use the data type *set* to store the different nodes that were in each maximal clique and minimal separator. Since the data type *set* has methods that calculate the union and intersection between two sets, and also the difference between two sets, it would have been ideal to be able to use this data set. However, the only way to get a random element from a set is by using the predefined method *set.choose*() which is meant to return a random element of the set. The only problem is that this method always returns the first element of the set. The *construct_sep* method (see Algorithm 5) takes as input a maximal clique x and a set that contains the nodes that the minimal separator has to contain. If the size of this set is less than k, random nodes are added from the maximal clique x. Since the method *set.choose*() always gives back the first element the code below would hang:

Algorithm 7: example()

Let N denote the set that contains the nodes that must be in the minimal separator.
Let C denote the set that contains the nodes that the maximal clique x contains.
while *size of* N *is less than* k **do**
  N.insert(*C.choose*())
end

The data type *list* was therefore used to store the nodes contained in each maximal clique and minimal separator. Lists are generally not sorted, but since the lowest numbers always were included first, all lists in the program are sorted. Some places in the program the data type *set* is also used despite the problems with the method *set.choose*(). The method *construct_sep* receives a set as input. It could alternatively have also received a list, but ensuring that the list contained k distinct and sorted elements would have been harder.

Graphs and their data types are very central to LEDA. A graph $G = (V, E)$ consists of a set of nodes V and a set of pairs of nodes E, called edges. The data type *graph* is the basic data type for representing graphs in LEDA. They are implemented by doubly linked lists. The space requirement for graphs is $O(n + m)$, where n is the number of nodes and m is the number of edges. This data type is used in the program to represent the k-graph constructed by the user. The built in method *G.undirected*() is used since the graph should be undirected. Methods such as *G.degree*(*node v*), *G.choose_node*, $G.first_adj_edge(node\ v)$, $G.last_adj_edge(node\ v)$, $G.opposite(node\ v, edge\ e)$, $forall_nodes$ and $forall_adj_nodes(node_v)$ were used frequently in the program. The method *G.choose_node* is used in Algorithm 2 when a random maximal clique is selected. $G.first_adj_edge(node\ v)$, $G.last_adj_edge(node\ v)$ and $G.opposite(node\ v, edge\ e)$ return the first adjacent edge to v , the last adjacent edge to v, and the node u opposite to v such that $e = (u, v)$. The two last methods are used to iterate through either all the nodes in the graph or only the neighbours of v.

The data type $GraphWin$ combines the *graph* data type and the *window* data type. A GraphWin object is both a window, a graph, and a (two dimensional) representation of this graph in this window. The class GraphWin was used to visualize the k-graphs. Built in methods such as *save_gw* was used to save the graph in .gw format. When a *.gw* file is opened the build in method *read_gw* was used to read in the graph. The program also uses a boolean variable to determine whether or not the graph has been changed since last save operation. LEDA has a built in function, *gw.unsaved_changes*(), that reports whether or not the graph has been changed since last save operation, but in the program this function did not work properly. The only explanation why this method did not work properly would be that the method *save_gw* is called by a pointer of type $GraphWin$ instead of the object.

LEDA has several built in algorithms for solving various problems. The only algorithm used by the program is *spring embedding*, but several of the problems from the list in chapter 1.1 can be solved by such built in algorithms. The $TOPSORT$ algorithm finds a topological ordering if the input graph has one, the $NT\ DIJKSTRA_T$ method computes a shortest path from a source node to a sink node and returns its length, and the $NT\ MAX_FLOW_T$ computes a maximal flow in the input graph and returns its value.

The only algorithm the program that generates k-graphs uses is *spring embedding*. It models the nodes of a graph as points in the plane that repulse each other, and it models each edge as a spring between the endpoints of the edge. In each iteration the force acting on any node is computed as the sum of repulsive forces (from all other nodes) and attractive forces (from incident edges) and the node is moved accordingly [9]. The algorithm does not guarantee that edges do not cross or nodes overlap. Figure 7 and Figure 8 shows the same graph before and after the spring embedder algorithm was used.

Methods such as increase/decrease node size and construct random maximal cliques have a maximal limit of 100. The user can click the button again and construct 100 new random maximal cliques, but to ensure that the user do not give too high numbers a maximal value was added in.

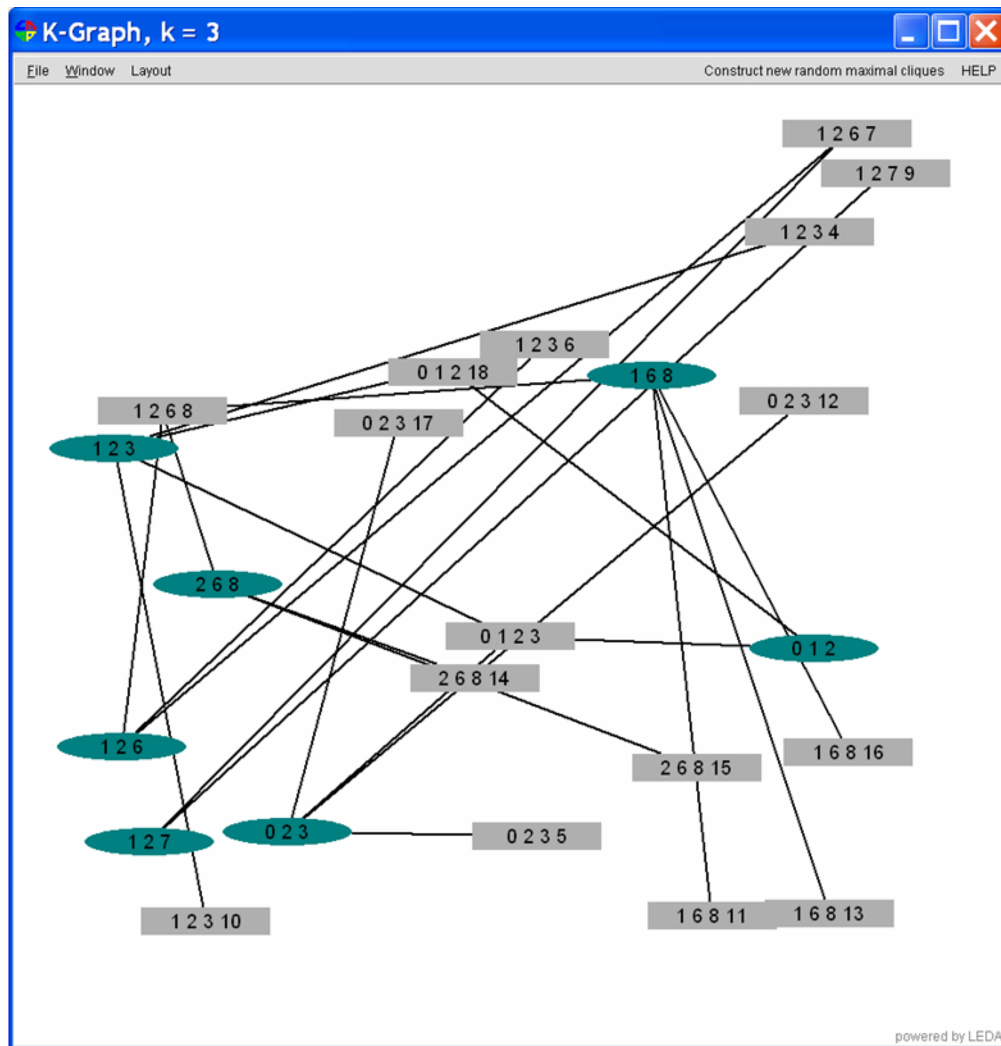


Figure 7: Before the spring embedding algorithm was called

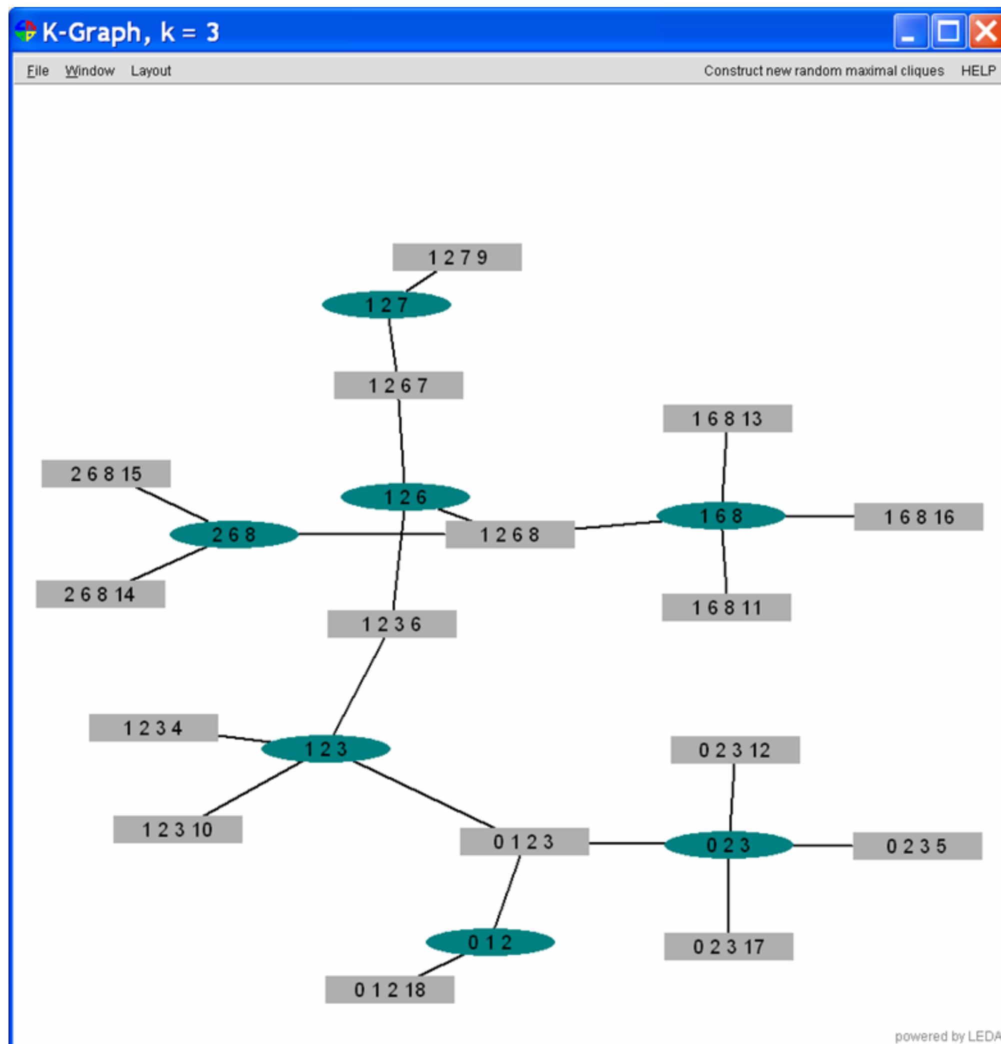


Figure 8: After the spring embedding algorithm was called.

4.2 The program

The program generates k-graphs (graphs that satisfies the four conditions of Lemma 12) either automatically or manually. The user can choose to start with an empty graph and add one and one node and will then receive error messages if either of the four conditions have not been satisfied, or she can ask the program to generate a random k-graph.

4.3 Overview of the user interface

Figure 9 shows the upper part of the program. It has three sub-menus, namely file, window and layout, and two buttons "Construct new random maximal cliques" and "Help". Figure 10 shows the three sub-menus expanded.

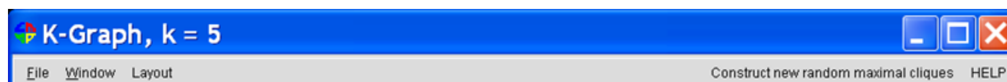
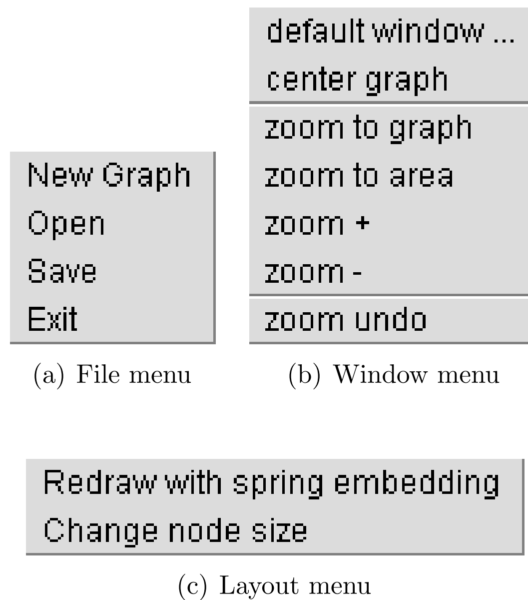


Figure 9: The upper part of the program

The file menu offers I/O operations for graphs. It contains the following operations:

- *New Graph* : Deletes the old graph, asks the user for a new value of k, and the number of maximal cliques the new graph should contain. (Figure 11)
- *Open* : Opens a file in gw format. The program assumes that the graph in the file has the right format, i.e. that the four conditions of Lemma 12 have been satisfied. If the file cannot be opened, an error message is produced (Figure 12).
- *Save* : Saves the graph in gw format.
- *Exit* : If the graph has been altered since last save operation the user will be asked if she wants to save the file before exiting the program (Figure 13).



(a) File menu

(b) Window menu

(c) Layout menu

Figure 10: The contents of the file, window and layout menu

Figure 11: The user is asked if she wants to start on a new graph

The window menu has zoom operations that allows the user to change the user space of the drawing window. (This menu is the default window menu that was already implemented in LEDA)

Figure 12: Error message when a file cannot be loaded

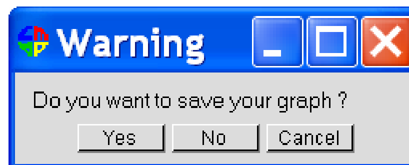


Figure 13: Warning message that is shown if the user wants to exit the program and the graph has been altered since last save operation

The layout menu allows the user to change the design of the graph. It contains the following operations:

- *Redraw with spring embedding* : The graph is redrawn with spring embedding. The idea of spring embedding is to simulate a graph as a system of mass particles. The nodes are the mass particles and the edges are springs between the particles. The algorithm tries to minimize the energy of this physical system. (Even though the graph is planar edges may cross, and edges may also cross nodes.)

- *Change node size* : Asks the user how many pixels she wants to increase or decrease all the nodes in the graph with. The maximum value is 100, while the minimum value is -100.

The "Construct new random maximal cliques" button asks the user how many new maximal cliques she wants to add to the graph (Figure 14). The maximum number of new cliques is 100 (she can however click the button several times).

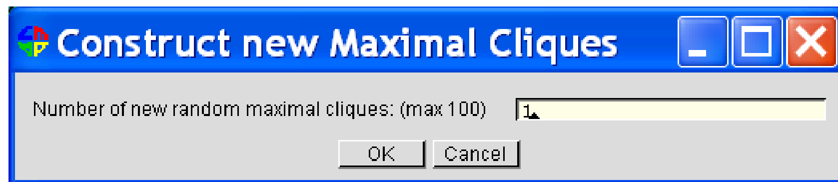


Figure 14: User is asked how many new maximal cliques she wants to add to the graph

The "HELP" button produces a panel that explains the different buttons of the program and how to construct a valid graph (Figure 15).

If the user clicks on the "X" in the upper right corner of any panel (for instance the help panel, error messages, and so on.) the whole program will be terminated due to the way LEDA has been implemented.

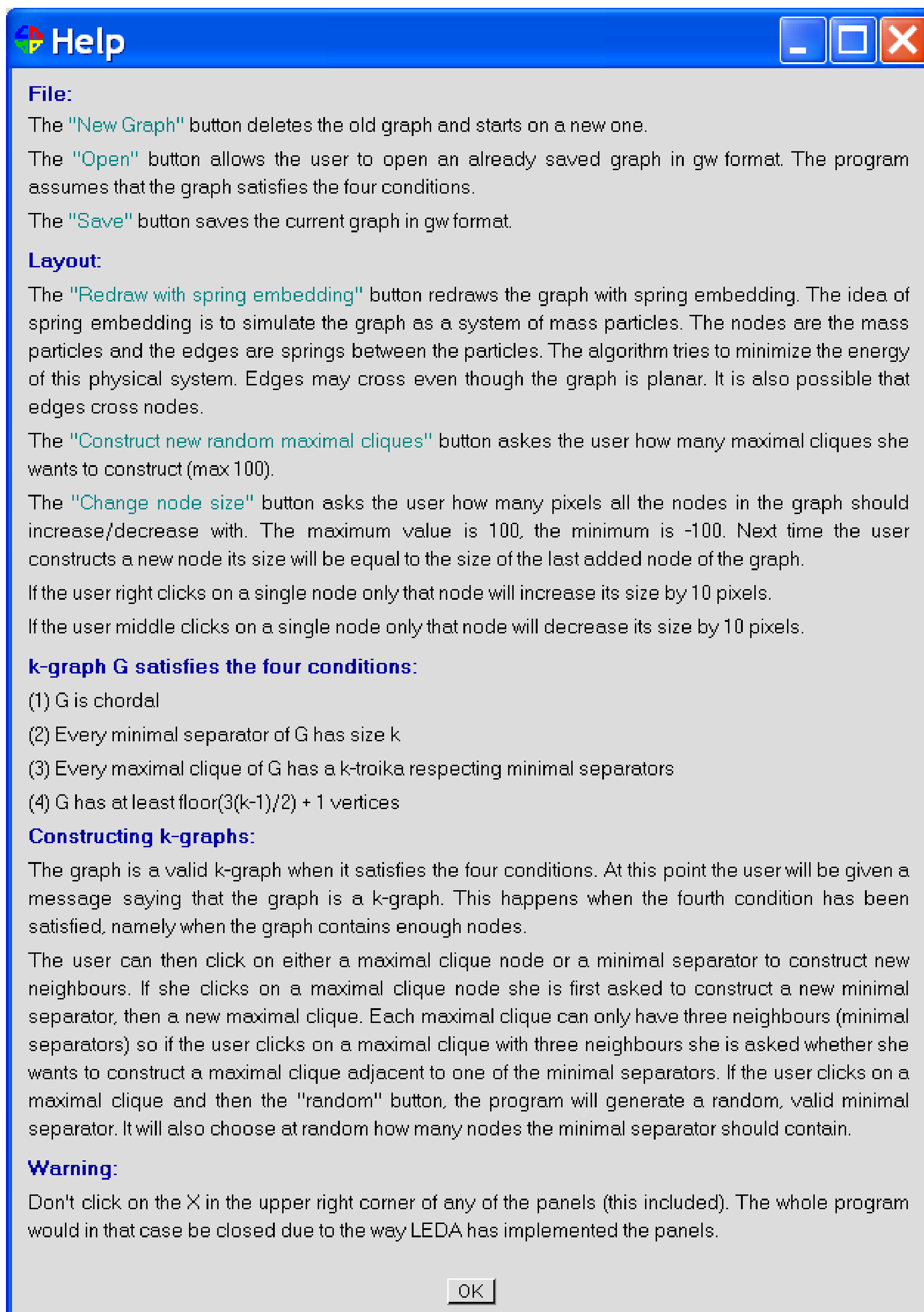


Figure 15: Help panel

4.4 Node attributes

The user can change the size of either a single node or all the nodes in the graph simultaneously (Figure 16). If the user right clicks on a node it will increase its size by 10 pixels, if she middle clicks it will decrease by 10 pixels. Next time the user constructs a new node its size will be equal to the size of the last added node of the graph.

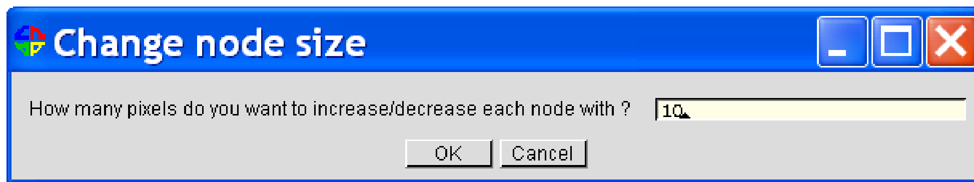


Figure 16: User is asked how many pixels the node should increase or decrease with

The maximal clique nodes are grey, while the minimal separators are green. Both maximal cliques and minimal separators can be moved by dragging. While the user drags a maximal clique node, all its adjacent edges will change color to grey (Figure 17). If she drags a minimal separator, all its adjacent edges will change color to green (Figure 18).

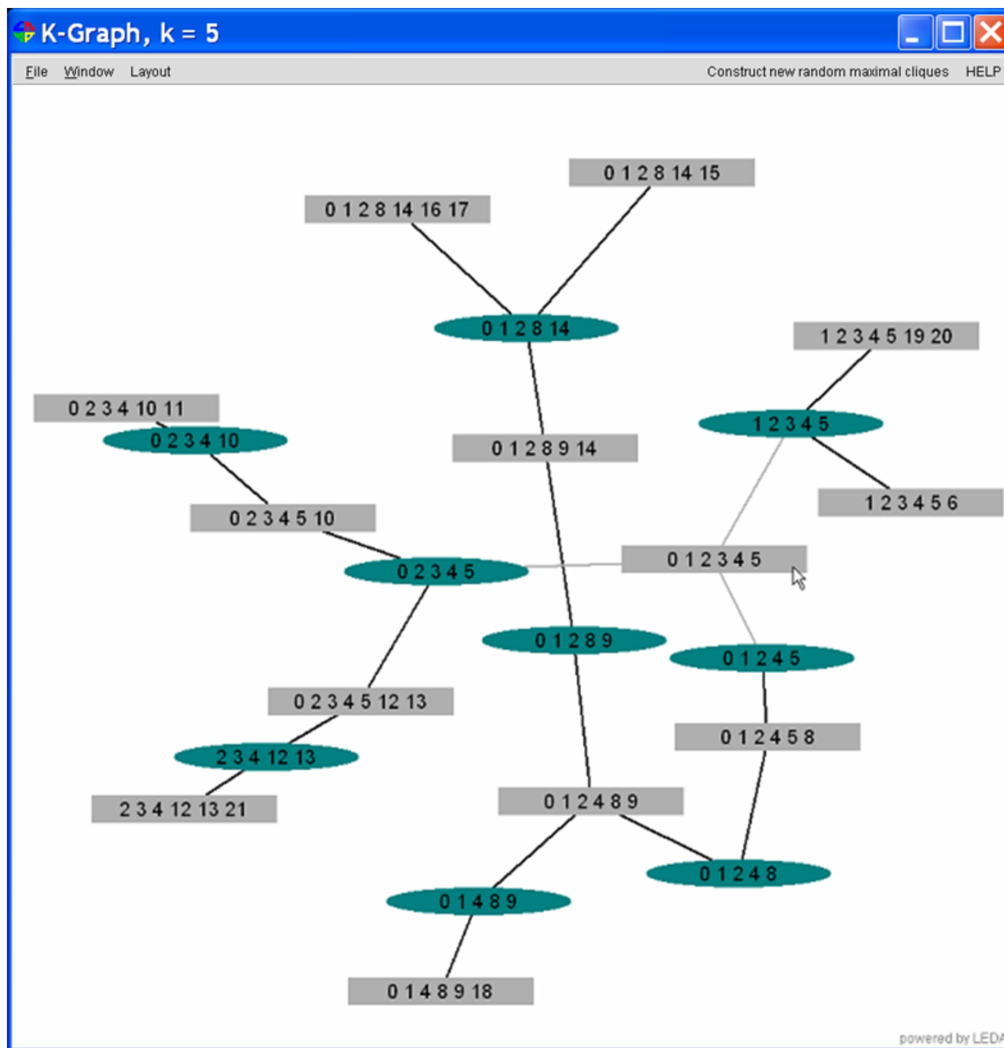


Figure 17: User drags a maximal clique node and its adjacent edges change color

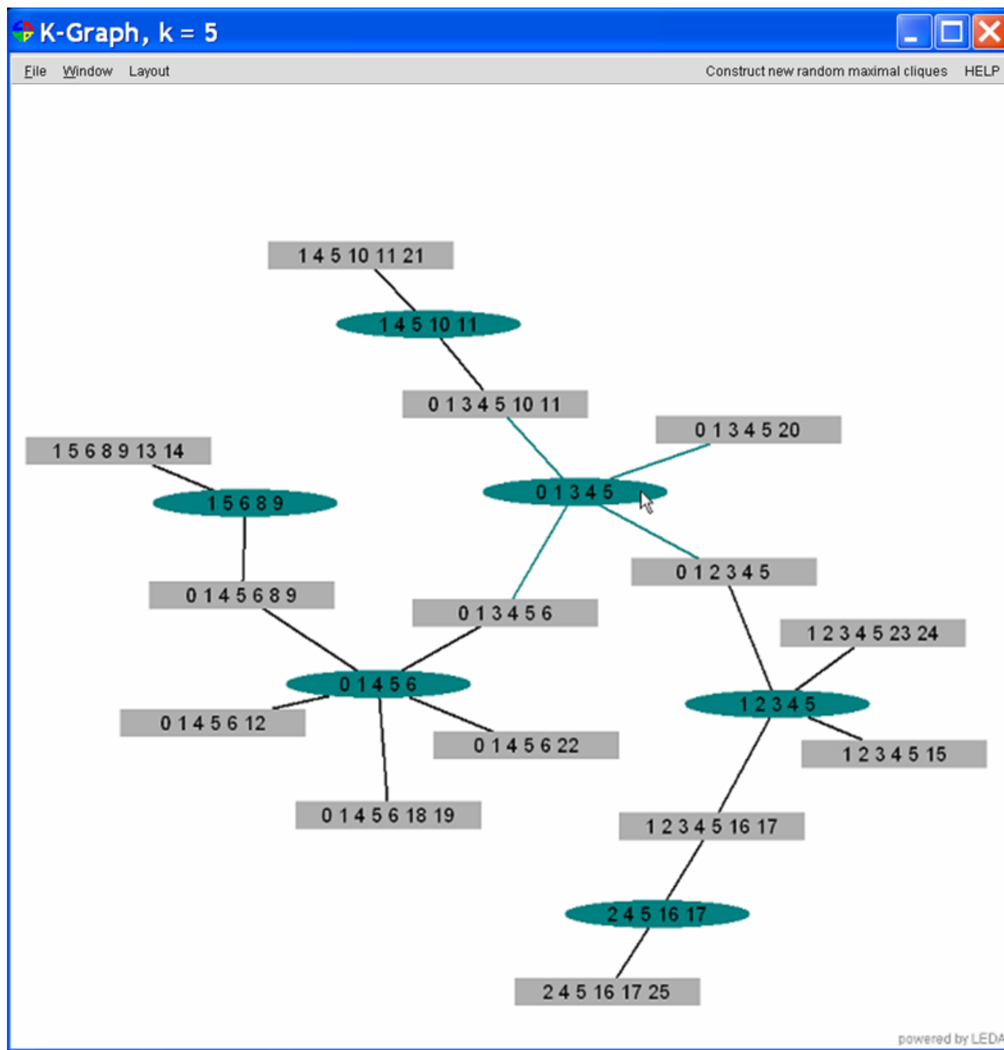


Figure 18: User drags a minimal separator node and its adjacent edges change color

4.5 Constructing the graph and error messages

The program starts by asking the user how many maximal cliques she wants to start with and the value of k (Figure 19).

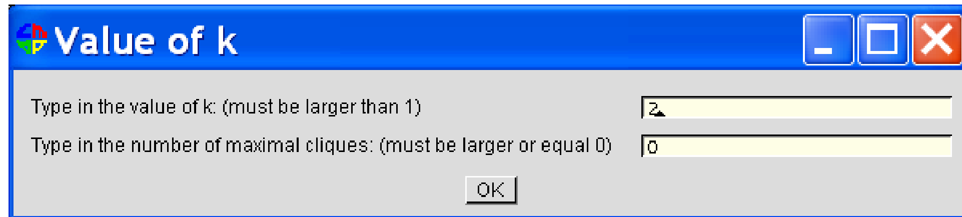


Figure 19: User is asked how many maximal cliques she wants to start with and the value of k

When the graph consists of $\lfloor 3(k-1)/2 \rfloor + 1$ or more nodes the fourth condition of Lemma 12 has been satisfied and a message informing the user that the graph is now a valid k-graph is produced (Figure 20).

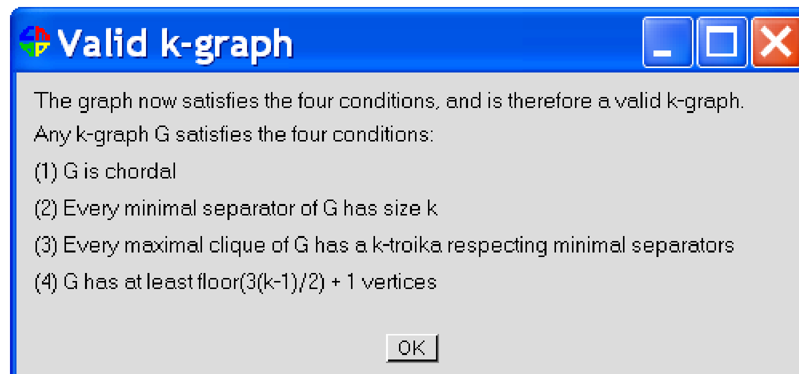


Figure 20: The user has created a valid k-graph

If the user clicks on a maximal clique node x she has to choose k nodes from x that the new minimal separator should consist of (Figure 21). All the nodes from the maximal clique x are displayed as boolean boxes.

Figure 21: User is asked to choose k nodes

If the user chooses too few or too many nodes error massages are displayed (Figure 23 and Figure 22) informing the user that the second condition of Lemma 12 has not been satisfied.

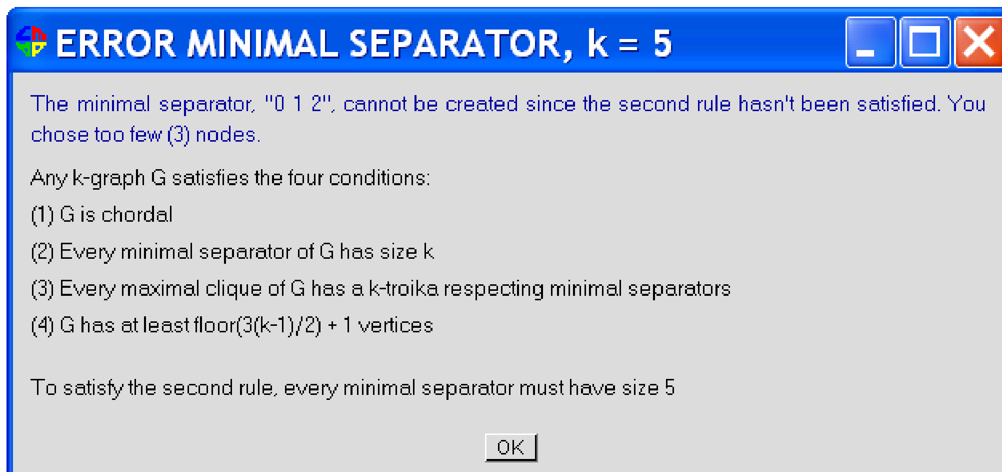


Figure 22: User chooses too few nodes.

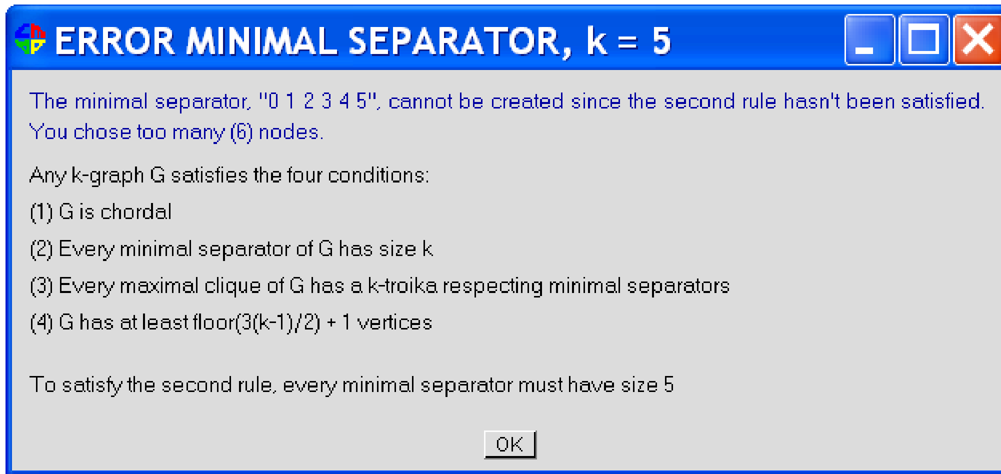


Figure 23: User chooses too many nodes.

If the minimal separator already has three neighbours she is asked whether she would like to construct a new maximal clique adjacent to one of the three minimal separators (Figure 24).

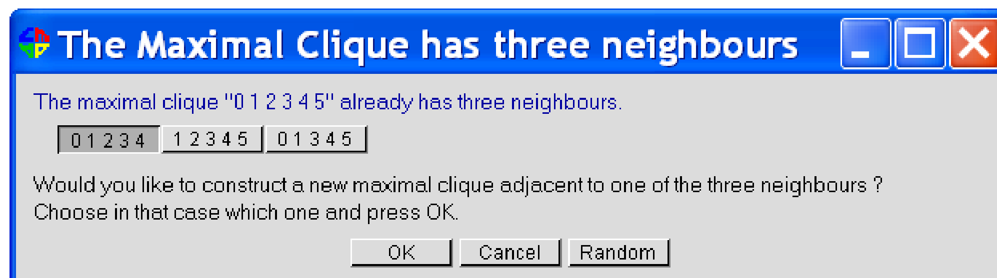


Figure 24: The maximal clique "0 1 2 3 4 5" already has three neighbours (minimal separators). The user is asked whether she would like to construct a new maximal clique adjacent to one of the already existing minimal separators.

If the minimal separator already exists the user is asked if she would like to construct a new maximal clique adjacent to the minimal separator she chose (Figure 25).

Figure 25: User is asked whether she would like to construct a new maximal clique adjacent to an already existing minimal separator.

If the union of the new minimal separator, chosen by the user, and one or several of the already existing ones, does not equal the maximal clique, three different error messages can be produced. If the degree of the maximal clique is one an error message is created (Figure 26). If the union between one of the already existing minimal separators and the new minimal separator is unequal to the maximal separator, a second error message (Figure 27) is shown. If the union between the new minimal separators and two of the already existing minimal separators is unequal to the maximal separator, a slightly different error message (Figure 28) is produced.

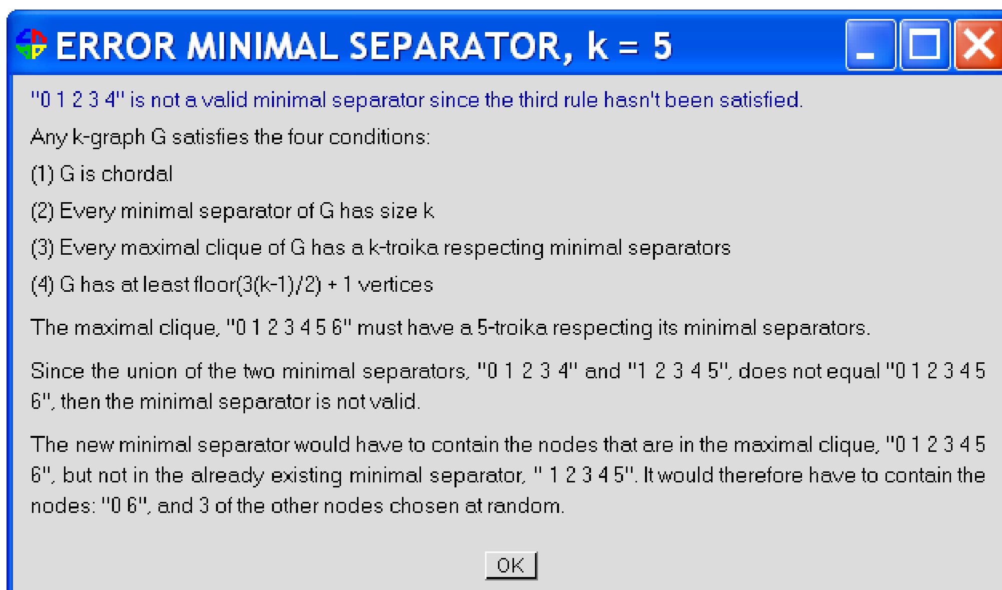


Figure 26: The maximal clique has only one minimal separator. The minimal separator selected by the user cannot be created since the third condition of Lemma 12 has not been satisfied.

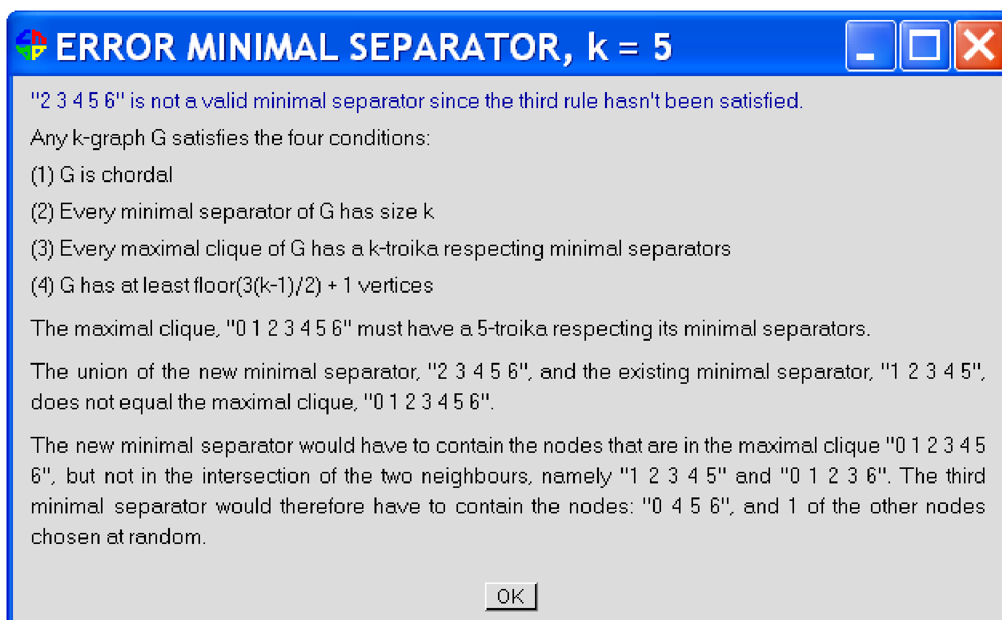


Figure 27: The maximal clique has two minimal separators. The union of the proposed minimal separator and one of the already existing minimal separators does not equal the maximal clique, and can therefore not be created.

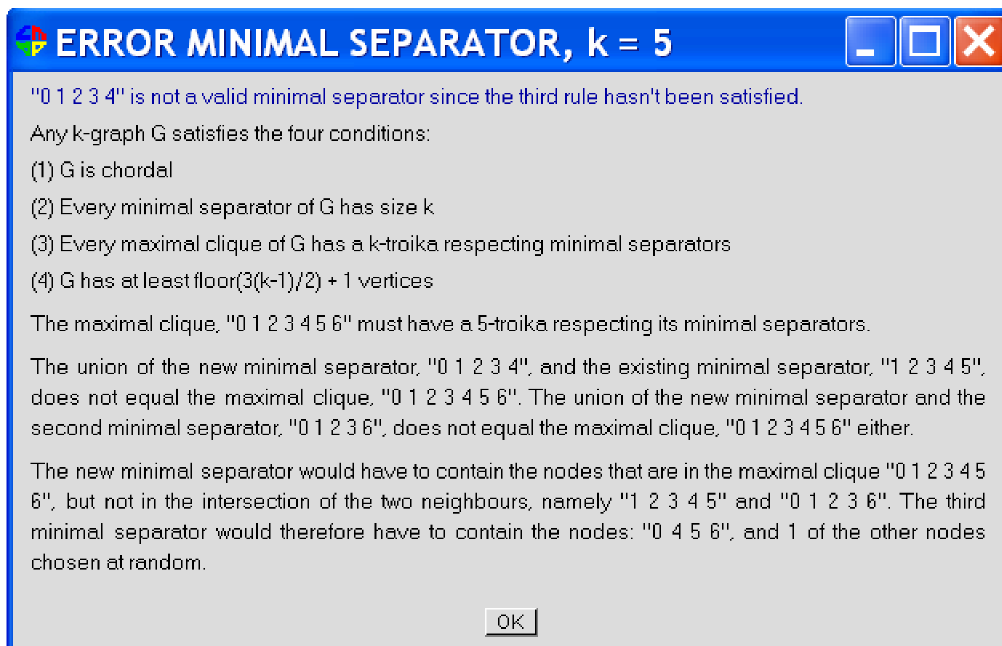


Figure 28: The maximal clique has two minimal separators. The union of the proposed minimal separator and both of the already existing minimal separators does not equal the maximal clique, and can therefore not be created.

If the user has created a valid minimal separator or maximal clique she is asked where she wants to place it on the screen. (Figure 29 and Figure 30)

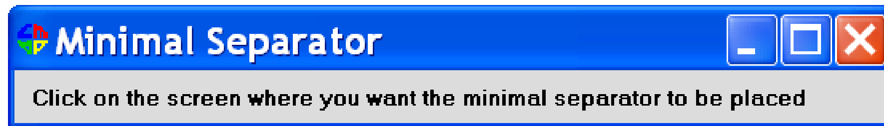


Figure 29: User is asked where she wants to place the new minimal separator

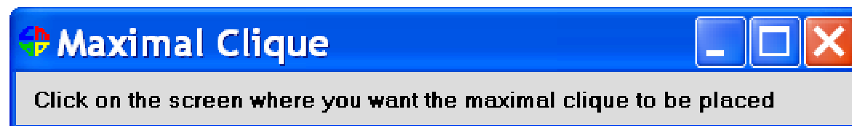


Figure 30: User is asked where she wants to place the new maximal clique

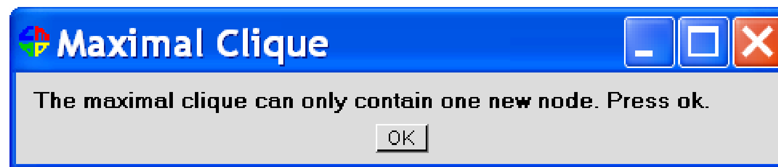


Figure 31: User can only add one new node to the maximal clique

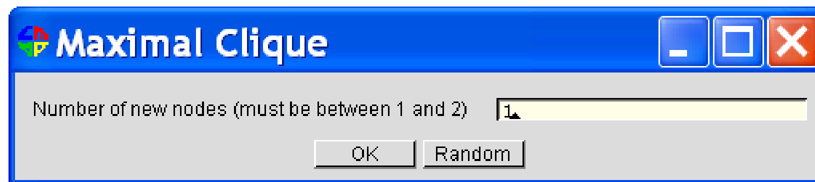


Figure 32: User is asked how many nodes the new maximal clique should consist of

Figure 31 and 32 asks the user how many new nodes she wants to include in the maximal clique. If $\lfloor k/2 \rfloor = 1$ then the user can only add one extra node to the maximal clique so Figure 31 is displayed, otherwise Figure 32 is displayed.

5 Summary and conclusion

We have seen several examples of graphs and graph algorithms in the real world. For example the shortest path problem, scheduling problems and graph coloring. We have also described certain algorithms more thoroughly. One example of this is the maximum independent set. While listing up different algorithm design techniques we have seen that dynamic programming can be used to solve certain graph problems, where the treewidth or branchwidth is bounded, faster. Since the branchwidth often is smaller than the treewidth for graphs, dynamic programming based on branch-decomposition rather than tree-decomposition can be more efficient. In the paper [12] it is demonstrated how one can generate k-branches. In this master thesis we implement a program that generates the superclass of k-branches called k-graphs.

There are certain things in the implementation that could have been done differently. As mentioned in chapter 4.1 we would have preferred to use the data type set to store the nodes that were included in the different maximal cliques, but due to problems with the method *set.choose*() this was impossible. This led to some extra work since methods such as *intersection*, $diff$ and *union* had to be implemented. The *intersection* method receives two lists as input and return the intersection of the two lists as a set. The $diff$ methods also receives two lists as input and returns the elements of the first list minus the elements in the second list. The *union* method receives three lists as input and returns true if the elements in the first set equals the union of the two last sets.

We would also have preferred to use a graph drawing method that ensured that the graph, if it was a tree, could be drawn without edges crossing. However LEDA had no such methods to offer, and we decided that the spring embedding algorithm was the best to use.

Another thing that should be done differently was the size of the nodes in the program. Unfortunately we found no method implemented in LEDA that would change the node size automatically. Instead the user has to right or middle click on a node to change its size. Alternatively she can change the size of all the nodes in the graph with the button "change node size".

One could also have used the data type *array* instead of the data type *list* for storing the different nodes contained in the maximal cliques and minimal separators. We decided to use the data type *list* since elements (nodes) often were added to the list, and the insertion time for lists is $O(1)$. However, since we also need to access the elements by position rather frequently it could have been better to use an *array*.

We hope that this program can be used in pedagogical settings and as an aid to researchers in the field of tree-like graphs. The edge-maximal graphs of branchwidth k, called k-branches, are a little-known graph class. This is in contrast to the similar notion for treewidth, the k-trees, which are well known by every graph algorithms researcher due to the very easy algorithm to generate them. By experimenting with our program for generating the k-graphs we hope that researchers can gain a better intuition also for k-branches.

6 Bibliography

References

[1] Frederic Dorn and Jan Arne Telle. Two birds with one stone: the best of branchwidth and treewidth with one algorithm.

[2] Jean-Loup Guillaume and Matthieu Latapy. The Web Graph: an Overview. http://jlguillaume.free.fr/www/publis.php?lang=eng

[3] A. Gulli and A. Signorini. The Indexable Web is More than 11.5 billion pages. 2005.

[4] Pinar Heggernes. Treewidth, partial k-trees and chordal graphs. Delpensum INF334 - Institutt for informatikk, September 15, 2005.

[5] Gunnar Horneland. Implementasjon av grafalgorithmar i LEDA. Institutt for informatikk, Universitetet i Bergen. 27. August 2004.

[6] Jon Kleinberg and Éva Tardos. Algorithm Design, Addison-Wesley, 2005.

[7] Leslie Lamport. A Document Preparation System, Latex User's Guide And Reference Manual, Second Edition. Addison-Wesley Publishing Company, 1994.

[8] Jan Van Leeuwen. Handbook of Theoretical Computer Science, Volume A, Algorithms and Complexity. The MIT Press.

[9] K. Mehlhorn and S. Näher. LEDA, a platform for combinatorial and geometric computing. Cambridge University Press, 2000.

[10] Ola A. Mæhle. Analyzing the Treewidth of Java Programs. Dept. of Informatics, University of Bergen, Norway. February 7, 2002.

[11] Øyvind Neuman. Algorithmic solutions for drawing a portion of the Internet World Wide Web. Institutt for Informatikk, Universitetet i Bergen, Norway. August 2004.

[12] Christophe Paul, Andrzej Proskurowski and Jan Arne Telle. Generation of graphs with bounded branchwidth.

[13] Christophe Paul and Jan Arne Telle. Edge-maximal graphs of branch-width k: the k-branches. Preprint submitted to Elsevier Science, 15 December 2006.

[14] Kenneth H. Rosen. Discrete Mathematics and Its Applications, Fifth Edition. McGraw-Hill 2003. Pages: 537-618.

[15] Michael Sipser. Introduction to the Theory of Computation, Second Edition, International Edition. Thomson Course Technology 2006. Pages: 260-280.

[16] M. Thorup, All Structured Programs have Small Tree-Width and Good Register Allocation, Information and Computation, Volume 142, Number 2. May 1, 1998.

[17] Algorithm2e.sty package: http://www.lirmm.fr/~fiorio/AlgorithmSty/algorithm2e.pdf

[18] A Survey of Google's PageRank: http://pr.efactory.de/.

[19] Google Page Rank - Whitepaper: http://www.ianrogers.net/google-page-rank/.

[20] Latex Tutorial: http://www.tug.org.in/tutorials.html

[21] Leda Guide: http://www.algorithmic-solutions.info/leda_guide/

[22] Leda Manual: http://www-ma2.upc.es/wood/software/LEDA-4.1-manual/Contents.html

[23] Leda Tutorial: http://www.leda-tutorial.org/