CS 762: Graph-theoretic algorithms

Lecture notes of a graduate course

University of Waterloo

Fall 1999, Winter 2002, Winter 2004

Version of September 20, 2005

Contents

Introduction

This document results from graduate courses that I taught at the University of Waterloo in Fall 1999, Winter 2002 and Winter 2004. Students were asked to take lecture notes in LaTeX, which I then compiled and edited for this document.

The topic of graph algorithms is so unbelievably big that it is entirely hopeless to want to hold just one course about them. Even focusing on advanced graph algorithms (whatever "advanced" may mean) doesn't help, because there is still too many algorithms to cover. The focus of this course was therefore even more specialized: I wanted to look at problems which are "normally" difficult (by which I mean a high polynomial running-time, or even NP-complete), but which can be solved faster when focusing on a specialized subclass.

These notes assume that the reader is familiar with the following topics:

- Data structures, in particular lists, stacks, queues, trees, bucket sorting, graph traversals.
- Graph theory and combinatorial optimization, in particular graph definitions, shortest paths, flows, matchings.
- Analysis of algorithms, in particular the O-notation and NP-completeness.

For some of these topics, a brief review is given here (see the appendix), but for more details, the reader is referred to for example [CLRS00] for data structures and analysis of algorithms, and [Gib85] for graph theory and combinatorial optimization.

I would like to thank everyone who put up with me having little time for them (especially my graduate students at the time), because I was always "busy writing lecture notes". Also, many thanks to the students who took lecture notes in the first place: Mohammed Ali Safari, Broňa Brejová, Joe Capka, Eowyn W. Cenek, Hubert Chan, Vlad Ciubotariu, Fred Comeau, Mike Domaratzki, Alastair Farrugia, Andrew Ford, Yashar Ganjali G., Kang Teresa Ge, Andreas Grau, Masud Hasan, Jonathan D. Hay, Michael Laszlo, Josh Lessard, Martin McSweeney, Mike Newman, Mark Petrick, David Pooley, J.P. Pretti, Selina Siu, Matthew Skala, Lubomir Stanchev, Henning Stehr, Mauro Steigleder, Lawrence Tang, Philip Tilker, Tomáš Vinař, Dana Wilkinson, Jinbo Xu, Daming Yao, Philip Yang. Most of your documents have been rewritten probably beyond recognition, but what you gave me was what I started with, and having it helped.

Someone suggested I make a book out of it. I might, one day, but probably not any time soon (university professors are Very Busy People). Whether or not, I would appreciate getting feedback about these lecture notes, be it because you found typos (I have no doubt

that I added at least as many as I removed), or because you feel that some topic/explanation is really badly done and could be improved. My email is `biedl@uwaterloo.ca`.

Enjoy!

Therese Biedl
Waterloo, September 20, 2005

Chapter 1

Introduction

This course is concerned with special graph classes, and problems that become easy/easier on these graphs classes.

To explain what we mean by that, we would have to review various notions, in particular graphs, graph algorithms, asymptotic analysis, and NP-hardness. This will *not* be done in this course; you are assumed to be familiar with them. If you want to review some material, the book by Cormen et al. [CLRS00] is a good source.

1.1 Some graph classes

In the following, we'll give an overview of graph classes that will be covered in this course. We will come back to all these graph classes in later chapters.

- *Interval graphs*: A graph is an interval graph if it can be represented as intersection graph of intervals. More precisely, given a set of intervals, we can define a graph by taking one vertex for every interval, and an edge between two vertices if and only if the two intervals intersect. A graph is an interval graph if it can be obtained this way. See Figure 1.1.

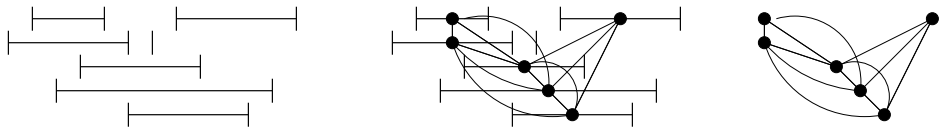


Figure 1.1: A set of intervals and the interval graph defined by it.

  Many problems become polynomial on interval graphs. To name just a few, we will see soon that Clique, Coloring and Independent Set can be solved in linear time on interval graphs. We will also study various related graph classes, for example chordal graphs, comparability graphs and perfect graphs, and show that many of the above problems are easily solvable on these graphs as well.

  The main reference for this part of the course will be the book by Golumbic [Gol80].

- *Trees*: A graph is a tree if it is simple, connected, and has no cycle. See Figure 1.2.

  Just about all graph problems become linear-time solvable on trees. This is quite easy to show for Independent Set, Vertex Cover, and Matching, to name just a few.

  For this reason, generalizations of trees have been studied, the so-called partial k-trees, where most graph problems are still solvable in linear time (if k is a constant) via dynamic programming. In particular, this holds for partial 2-trees (which are known as SP-graphs, and have been studied long before partial k-trees.)

  The main reference for this part of the course will be the article by Bodlaender [Bod93].

- *Planar graphs*: A graph is called a planar graph if it can be drawn (in two-dimensional space) without a crossing. See Figure 1.2. Note that every tree is a planar graph, but the reverse is not true.

  Many results are known about planar graphs, and have inspired much work in graph theory (see for example the lovely book on the 4-color theorem by Aigner [Aig84].) To mention just three algorithmic results here: The Maximum Cut problem and NAE-3SAT become polynomial time solvable on planar graphs. Also, Maximum Flow can be solved more efficiently on planar graphs.

  We will also study various related graph classes, in particular outer-planar graphs, k-outer-planar graphs and SP-graphs (which are planar). The material for this part of the course is assembled from many source, one possible reference is the book by Nishizeki and Chiba [NC88].

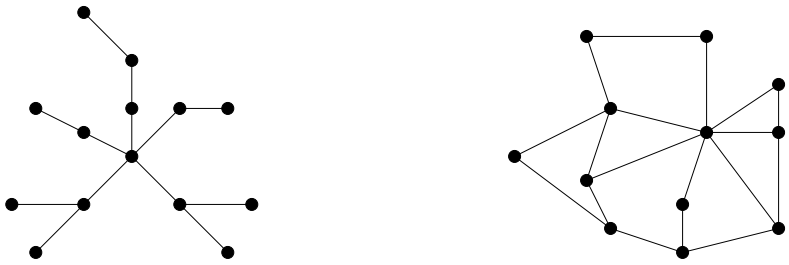


Figure 1.2: A tree and a planar graph that isn't a tree.

Unless explicitly said otherwise, all graphs that are studied in this class are *simple* (no multiple edges, no loops), *connected* (a path between any pair of vertices), and have at least 2 vertices (or whatever small number of vertices is needed to make the graph non-trivial).

Since the objective of this course is algorithms, these assumptions are justified. Most algorithms that work for simple graphs can be generalized to arbitrary graphs by first deleting all multiple edges and all loops.[1] Similarly, if a graph is not connected, then we can run the algorithm for all connected components. Finally, if the graph has just a few vertices, no advanced algorithm is needed.

[1] Obviously there are exceptions, for example edge coloring. So one should be a little careful, but for most problems that we'll see, assuming simplicity poses no difficulty.

Also, normally the given graph is undirected, though we will sometimes impose a direction onto it if this helps for an algorithm. Finally, n will always denote the number of vertices and m will denote the number of edges.

1.2 Typical questions

So the main question in this course is:

> Given a graph problem P and a graph class $\mathcal{C}$, how easy is it to solve P on a graph that belongs to $\mathcal{C}$?

Rather than studying this for each problem (as we did above), we will study this for each graph class. So given a graph class $\mathcal{C}$, typical questions that we ask are the following:

- Which graph problem becomes easier if the graph is in $\mathcal{C}$? (Thus, if the problem was NP-hard, does it become polynomial? If it was polynomial, can we decrease the running time?)
- Which graph problem is still NP-hard on graphs in $\mathcal{C}$?

The motivation for this type of questions is as follows: Assume you have graphs that arise from some application, and you notice that they appear to have some special structure. Then, if you happen to know that the problem you're trying to solve is easier on some graph classes, it would make sense to test whether your graphs actually belong to this class, and if so, run the faster algorithm for it.

This naturally raises a few more questions for a graph class $\mathcal{C}$:

- How easy is it to test whether a given graph G belongs to class $\mathcal{C}$? Is this NP-hard? Polynomial? Linear-time?
- What are equivalent characterizations of graphs in class $\mathcal{C}$?

  Quite often, one characterization of the graph class is easier to convert into an algorithm than another. For example, for *chordal graphs* (which are graphs without an induced 4-cycle; we will see them in Chapter 3), the definition makes it easy to test chordality in $O(n^4)$ time. (There are better algorithms!) On the other hand, an equivalent characterization (a chordal graph is the same as a graph with a perfect elimination ordering) gives rise to a number of linear-time algorithms for chordal graphs.
- If G does not belong to $\mathcal{C}$, how "close" is G to being in $\mathcal{C}$?

  This is motivated by the hope of maybe modifying G into a graph G' in $\mathcal{C}$, solving the problem on G', and obtaining a solution (or at least an approximation thereof) for G from the solution of G'.

  There are various ways of defining "close"; it could by deleting or adding vertices or edges, contracting or subdividing edges, replacing crossings by vertices ....

  Sadly, these problems are just about always NP-hard.

Part I

Interval Graphs And Friends

Chapter 2

Interval graphs

Our first graph class to be studied is interval graphs. These appear fairly naturally as graphs in some applications (see below.)

An interval is a line segment with two definite end points. The ends of the interval can be either open or closed. For the definition of interval graph, we allow both open or closed intervals; as we will see soon that it does not matter.

Definition 2.1 *Given a set of intervals, we define a graph (the* intersection graph of the intervals*) that has a vertex v for every interval I_v, and an edge (v, w) if and only if the two corresponding intervals intersect, i.e., $I_v \cap I_w \neq \emptyset$.*

A graph G is called an interval graph *if it is the intersection graph of some set of intervals.*

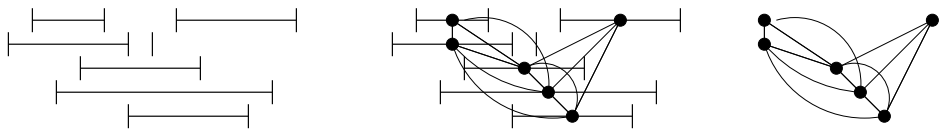


Figure 2.1: A set of intervals and the interval graph defined by it.

Interval graphs have a number of applications. Some examples include:

- **Archaeology:** Here the intervals are the time ranges of each artifact, and the problem is finding an order of the ages of the artifacts. However, there are some doubts as to whether the theoretical results are applicable to the practical setting (see [Spi03] for some discussion on this topic.)

- **Scheduling:** This is a common problem in computer science, in which we are given a number of tasks with different start and processing times, and we are interested in find order of these tasks which have certain properties. Closely related to this is activity selection, which amounts to finding a maximum independent set in an interval graph [CLRS00].

- **Biology:** Here certain sequences of DNA are modeled as intervals, and the problem involves constructing maps of the DNA [WG86] or gene finding [BBD$^+$04].

For a more comprehensive list, or more details on these, see also [Rob76].
Many graphs are not interval graphs:

1. Cycles of length at least 4: To see this, let us examine C_4. Label the vertices in C_4 as in Figure 2.2. Assume that C_4 has an interval representation. Because a is not connected to c, their intervals must be disjoint, say the interval of a is to the left of the one of c. Since b is adjacent to both a and c, it therefore must overlap the whole region between a and c. But the same also holds for d, so the intervals of b and d intersect, a contradiction. See Figure 2.2. The proof is similar for larger cycles.

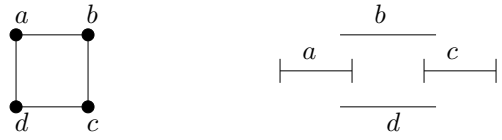


Figure 2.2: C_4 cannot be an interval graph.

2. Any graph that contains an induced $C_k : k \geq 4$. For notice that interval graphs are *closed* under taking induced subgraphs: if we have an interval graph G, we can get an interval representation for any induced subgraph G' of G by taking one of G and deleting all intervals of vertices in G that are not in G'.

3. The tree of Figure 2.3. To see this, note that a, c, e, g is an independent set and hence gives four disjoint intervals. So at least two of c, e, g must be on one side of a. Say the order of intervals is g, e, a. Then we cannot place f to intersect both a and g without it also intersecting e, which is a contradiction.

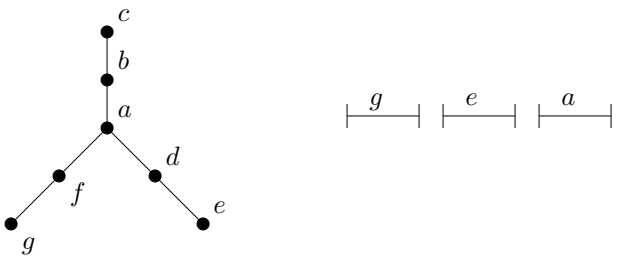


Figure 2.3: A tree that is not an interval graph.

2.1 A standardized representation

Until now, we have not differentiated between interval graphs with open intervals, closed intervals, or some mixture of the two. As we shall see, these all turn out to be equivalent: We can modify any interval representation into one that satisfies certain properties (which in particular implies that openness or closeness of intervals is irrelevant.)

Theorem 2.2 *Let G be an interval graph. Then G has an interval representation such that all endpoints of intervals are distinct integers.*

Proof: The proof can be outlined as follows: take an arbitrary interval representation, and sort the endpoints (breaking ties suitably). See Figure 2.4 for the tie-breaking rule.

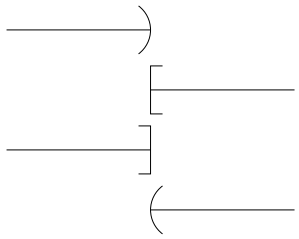


Figure 2.4: Breaking ties for sorting interval endpoints.

The detailed proof has two parts: We need to remove all duplications among the endpoints without changing the intersection graph of the intervals, and we need to assign integers to the endpoints. The second part is easy: once all endpoints are distinct, simply sort them and assign them $\{1, \ldots, 2n\}$ in sorted order.

To assign distinct endpoints, we proceed by induction on the number of coinciding endpoints. If there is none, then we are done. So assume that intervals $I_1, \ldots, I_t$, $t \geq 2$ begin or end at x. We distinguish them into four groups as follows. Say intervals $I_1, \ldots, I_k$ end at x and are open at x, $I_{k+1}, \ldots, I_\ell$ begin at x and are closed at x, $I_{\ell+1}, \ldots, I_s$ end at x and are closed at x, and $I_{s+1}, \ldots, I_t$ begin at x and are open at x.

Let ε be a small number that is smaller than the shortest distance between distinct endpoints. Then assign new endpoints to $I_1, \ldots, I_t$ by replacing endpoint x in I_i by $x + \frac{i}{t}\varepsilon$. Note that all these new endpoints fit in the range between x and the next endpoint of any interval. Hence this transformation does not affect any intersection of intervals other than those that begin or end at x.

Thus, to verify that this still represents the same graph, we need to check for all pairs of vertices v, w with endpoints at x whether their corresponding intervals still do or still do not intersect. There are many cases here:

- Assume the intervals of v and w both end at x. Then in the old and the new representation the intervals both contain the point $x - \varepsilon$, and so edge (v, w) exists in both representations.

- Assume the intervals of v and w both begin at x. Then in the old and the new representation the intervals both contain the point $x + \varepsilon$, and so edge (v, w) exists in both representations.

- Assume the interval of v ends at x, and the interval of w begins at x.

  - If the interval of v is open at x, then it did not intersect the interval of w, hence (v, w) was not an edge. Note that in the new representation, interval of v ends before any beginning interval starts, so likewise (v, w) is not an edge in the new representation.

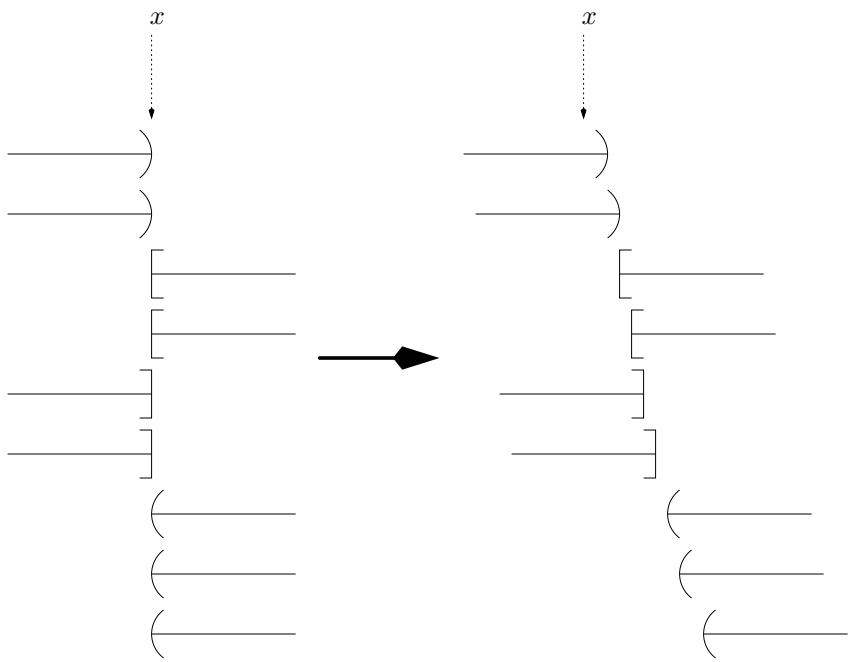


Figure 2.5: Replacing endpoints to remove coinciding endpoints.

- If the interval of w is open at x, then similarly one shows that there is no edge (v, w) in either representation.
- If both intervals are closed at x, then they both contain x, and so (v, w) is an edge. In the new representation, the interval of w will receive a start point that is smaller than the end point of v, and hence there will be an edge (v, w) as well.

□

Without giving details, we note that this standardized representation can be obtained in $O(n \log n)$ time, and often even in $O(n)$ time using bucket sort if the numbers of interval endpoints are small enough.

2.2 Perfect elimination order

Assume we have an interval graph with an interval representation where all endpoints are distinct. This then implies a natural vertex order of the graph: sort the vertices by the left endpoint of the interval. As we will see soon, this vertex order has some special properties. To analyze those clearly, we need some definitions:

Definition 2.3 *Given an undirected graph and a vertex order $\{v_1, \ldots, v_n\}$, we define the following:*

- *The* edge directions implied by the order *is obtained by directing (v_i, v_j) as $v_i \to v_j$ if $i < j$ and $v_j \to v_i$ if $i > j$, i.e., we direct each edge from left to right.*

- *If $v_i \to v_j$ is an edge implied by the order, then v_i is a* predecessor *of v_j and v_j is a* successor *of v_i.*
- *The set of predecessors of a vertex v is denoted $Pred(v)$.*
- *The set of successors of a vertex v is denoted $Succ(v)$.*

Notice that the indegree of a vertex in the implied edge direction is equal to the number of predecessors of that vertex (assuming, as usual, that the graph is simple). Thus, we will use $\text{indeg}(v)$ and $\text{outdeg}(v)$ to denote the number of predecessors and successors of v.

In the context of interval graphs, we can order the vertices by numbering the intervals in order of their leftmost endpoint, reading from left to right. Given some vertex v_i represented by an interval that starts at s_i, $Pred(v_i)$ is the set of all vertices with intervals that start before s_i and do not end before s_i. Therefore, all these vertices contain the point s_i, and hence overlap each other, which means that $Pred(v_i)$ is a clique. Such a vertex order has a special name, which we denote here for future reference.

Definition 2.4 *A* perfect elimination order *(or* p.e.o.*) is a vertex order $v_1, \ldots, v_n$ such that $Pred(v_i)$ is a clique for all $i = 1, \ldots, n$.*

In particular therefore, we have just shown that every interval graph has a perfect elimination order. The reverse does not hold. The graph shown in Figure 1.2 is not an interval graph. But it is easy to see that it has a perfect elimination order. In fact, every tree has a perfect elimination order: If we root the tree arbitrarily and take a pre-order, then in this order every vertex v has $\text{indeg}(v) \leq 1$. Since a set of 0 or 1 vertices is a clique, therefore for every vertex the predecessors form a clique. So a pre-order of a tree is a p.e.o. for the tree.

This naturally raises the question: what are the graphs that have a perfect elimination order? These graphs are called *chordal graphs*, and will be the objective of the next chapter.

Chapter 3

Perfect Elimination Orders

In this chapter, we will study graphs that have a perfect elimination order. We will first study algorithms for graphs that have a perfect elimination order. Then we give a totally different characterization of such graphs, which explains the name chordal graphs for them. Finally, we show how to recognize chordal graphs in linear time.

3.1 Algorithms for graphs with a p.e.o.

Recall that a *perfect elimination order* is an order $v_1, \ldots, v_n$ of the vertices such that $Pred(v_i)$ is a clique for all $i = 1, \ldots, n$. As we will see, this order can be used to solve three otherwise difficult graph problems in linear time: Colouring, Clique and Independent Set.

3.1.1 Colouring

For Colouring, there exists a simple greedy-algorithm that finds a colouring, given a vertex order: scan the vertices in order, and colour each vertex with the smallest colour not used among its predecessors. Figure 3.1 illustrates this process.

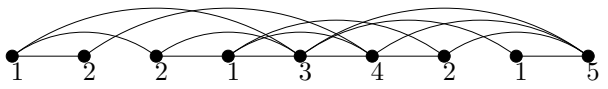


Figure 3.1: A graph with a vertex order and the result of the greedy algorithm for colouring.

Since vertex v has $\text{indeg}(v)$ predecessors, at least one of the colours $1, \ldots, \text{indeg}(v)+1$ is not used among the predecessors. Therefore, the greedy-algorithm will use at most $\max_i \{\text{indeg}(v_i) + 1\}$ colors.

Note that the greedy-algorithm can be applied to any graph and any vertex colouring (we will see this again later, when we find a 6-colouring of planar graphs with it.) However, normally it will not find an optimal colouring.

Theorem 3.1 *The greedy algorithm for Colouring yields the optimal number of colours if the vertex order is a perfect elimination order.*

Proof: We know that our algorithm finds a coloring with at most $\max_i \{\text{indeg}(v_i) + 1\}$ colors. Let i^* be the index where this maximum is achieved. So $\chi(G) \leq \text{indeg}(v_{i^*}) + 1$.

Since we have a perfect elimination order, the predecessors of v_{i^*} form a clique. All those predecessors are adjacent to v_{i^*}, so $v_{i^*} \cup \mathit{Pred}(v_{i^*})$ forms a clique, and $\omega(G) \geq \text{indeg}(v_{i^*}) + 1$.

But we know $\chi(G) \geq \omega(G)$, so $\chi(G) = \omega(G) = \text{indeg}(v_{i^*}) + 1$, which implies that this is an optimal colouring. □

It is not hard to show how to implement the greedy-algorithm in linear time (i.e., $O(m+n)$ time), presuming that the vertex order is given to us. Hence, we can solve Colouring in linear time on a graph with a given perfect elimination order.

3.1.2 Clique

Note that the proof of Theorem 3.1 shows that really, the greedy-algorithm for colouring at the same time also finds the maximum clique, since it finds a clique of size $\chi(G)$, and no clique can be bigger. So we can solve Clique in linear time on a graph with a given perfect elimination order.

3.1.3 Independent Set

For Independent Set, there exists a different greedy algorithm that yet again uses a vertex order: Scan the vertices in order, and for each v_i, add v_i to I if none of its predecessors has been added to I.

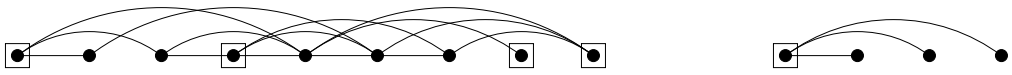


Figure 3.2: Two graphs with a vertex order and the result of the greedy algorithm for Independent Set.

Figure 3.2 shows two graphs and the results of applying the greedy algorithm for Independent Set to them. While it should be clear that the resulting set is indeed an independent set, this is in general not the maximum independent set. In fact, this is not even the maximum independent set for trees, as the second example in Figure 3.2 shows.

However, if we use the perfect elimination order in reverse order, we will obtain a maximum independent set.

Theorem 3.2 *Let $\{v_1, v_2, \ldots, v_n\}$ be a perfect elimination order. Then the greedy algorithm applied with order $v_n, v_{n-1}, \ldots, v_1$ gives a maximum independent set.*

Proof: This proof is given as a homework assignment. □

The greedy-algorithm can easily be implemented in linear time, and hence Independent Set is solvable in linear time on graphs that have a perfect elimination order.

3.2 Chordal graphs

Now we define an apparently different class of graphs, and then show that these are exactly the graphs that have a perfect elimination order.

Definition 3.3 *A graph is a chordal graph if it does not contain an induced k-cycle for $k \geq 4$.*

The name *chordal* comes from the following: Given a k-cycle with $k \geq 4$, a *chord* of the cycle is an edge between two non-consecutive vertices of the cycle. A graph hence is chordal if and only if every k-cycle ($k \geq 4$) in the graph has a chord. Some literature also refers to chordal graphs as *triangulated graphs*; we will not do this here as to not confuse them with triangulated planar graphs which we will meet later. Figure 3.3 shows an example of a chordal graph.

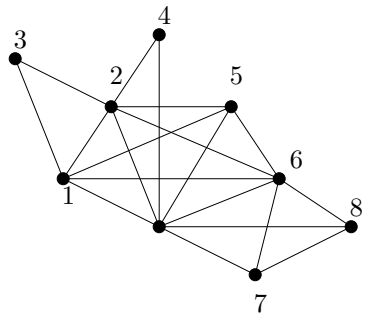


Figure 3.3: A chordal graph

Theorem 3.4 *If G has a perfect elimination order, then G is chordal.*

Proof: Assume C is a k-cycle where $k \geq 4$ and C has no chord. Let v be the vertex in the cycle that appears last of all the cycle vertices in the perfect elimination order. Then v must have at least 2 predecessors, namely, its two neighbours in the cycle. These two neighbours must form a clique since they are predecessors of v in the perfect elimination order. However, there is no edge between them since they are both adjacent to v in a chordless cycle of length at least 4. Contradiction. □

3.2.1 Separators

Now we will prove the other direction. This is much more complicated, and in order to do so, we need the concept of separators. Since this is a concept that we will come back to quite frequently, we will define these first.

Definition 3.5 *A* separator *is a partition $V = S \cup A \cup B$ of the vertices such that there are no edges between A and B.*

Definition 3.6 *Given two non-adjacent vertices a and b, an* (a,b)-separator *is a separator $V = S \cup A \cup B$ such that $a \in A$ and $b \in B$.*

Clearly, such a separator always exists. For example, the set $V - \{a, b\}$ is an (a, b)-separator. Note that it is not really necessary to specify A and B to give an (a, b)-separator. Any set S can be expanded to an (a, b)-separator as long as S does not contain a or b, and S intersects all paths from a to b.

Definition 3.7 *Given two non-adjacent vertices a and b, a* minimal (a,b)-separator *is an (a, b)-separator $V = S \cup A \cup B$ such that no subset of S is an (a, b)-separator.*

The concept of a minimal (a, b)-separator will be crucial for showing that every chordal graph has a perfect elimination order. However, while we are at the topic of separators, we will give two more definitions that will be useful in later chapters.

Definition 3.8 *A* $\frac{2}{3}$-separator *is a separator $V = S \cup A \cup B$ such that $|A| \leq \frac{2}{3}|V|$ and $|B| \leq \frac{2}{3}|V|$.*

The $\frac{2}{3}$-separator is really a special case of the weighted $\frac{2}{3}$-separator defined now:

Definition 3.9 *Let $w : V \to \mathbb{R}$ be a given weight-function (the vertex-weights). Set $w(T) = \sum_{v \in T} w(v)$ for every vertex-set T. Then a* weighted $\frac{2}{3}$-separator *is a separator $V = S \cup A \cup B$ such that $w(A) \leq \frac{2}{3}w(V)$ and $w(B) \leq \frac{2}{3}w(V)$.*

After scaling of the vertex weights, we will often assume that $w(V) = 1$, so that the condition can be written easier as $w(A) \leq \frac{2}{3}$ and $w(B) \leq \frac{2}{3}$.

It is not as obvious, but nevertheless not hard to show, that in the definition of a weighted $\frac{2}{3}$-separator, it again would be suffice to give the set S only. As long as all connected components of the graph induced by $V - S$ have weight at most $\frac{2}{3}w(V)$, it is possible to "piece together" components into sets A and B such that we obtain a weighted $\frac{2}{3}$-separator. The proof of this is left as an exercise.

3.2.2 Simplicial vertices

Before the main step, we need one more definition. A *simplicial vertex* of a graph G is a vertex v such that the neighbours of v form a clique in G. Clearly, if G has a perfect elimination order, then the last vertex in it is simplicial in G. This gives rise to a simple algorithm to find a perfect elimination order if one exists:

Algorithm: Find perfect elimination order.

For $i = n, \ldots, 1$
  Let G_i be the graph induced by $V - \{v_{i+1}, \ldots, v_n\}$.
  Test whether G_i has a simplicial vertex v.
  If no, then stop. G_i (and therefore G) has no perfect elimination order.
  Else, set $v_i = v$.
$v_1, \ldots, v_n$ is a perfect elimination order.

Note that if G is chordal, then after deleting some vertices, the remaining graph is still chordal. So in order to show that every chordal graph has a perfect elimination order, it suffices to show that every chordal has a simplicial vertex; the above algorithm will then yield a perfect elimination order.

3.2.3 Chordal graphs and simplicial vertices

Now we show that every chordal graph has a simplicial vertex. In fact, we show a slightly stronger statement, which is needed for the induction hypothesis.

Theorem 3.10 *Every chordal graph G has a simplicial vertex. If G is not a complete graph, then it has two simplicial vertices that are not adjacent.*

Proof: We proceed by induction n. In the base case, G has just one vertex, which is simplicial in G. So assume now that G has at least two vertices. If G is the complete graph then all vertices are simplicial and we are done, so assume that G is not complete, say a and b are two vertices with $(a, b) \notin E$.

Let S be a minimal (a, b)-separator. For any vertex set T, let G_T be the graph induced by T. Then G_{V-S} has a number of connected components; one contains a (let those vertices be A), one contains b (let those vertices be B), and there may be other connected components. See also Figure 3.4.

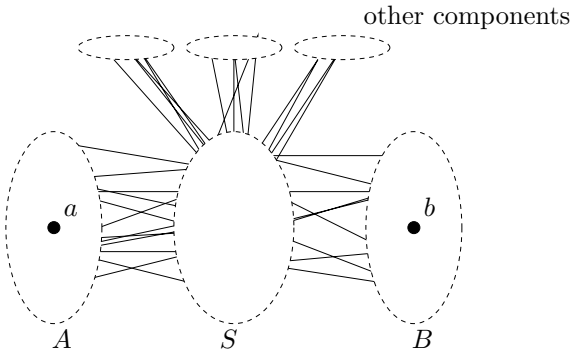


Figure 3.4: The minimal (a, b)-separator.

The theorem will now be proved by finding one simplicial vertex in A and one simplicial vertex in B. Clearly these are not adjacent. We will only do this for A, the other case is similar. We distinguish cases:

Case 1: $G_{A\cup S}$ is complete. Therefore a is a simplicial vertex in $G_{A\cup S}$. But this means that a is simplicial in G, since all neighbours of a (in G) belong to $A \cup S$ and hence to $G_{A\cup S}$.

Case 2: $G_{A\cup S}$ is not complete. Now we apply induction. Since $b \notin A \cup S$, $G_{A\cup S}$ is smaller than G. So we can apply induction, and know that there are two nonadjacent simplicial vertices x,$y \in G_{A\cup S}$.

Now if $x \in A$, then all neighbours of x in G are in $A \cup S$, so x is also simplicial in G and we are done. Likewise we are done if $y \in A$. So the only remaining case is if $x \in S$ and $y \in S$.

We claim that this case cannot happen. Assume for contradiction that $x \in S$ and $y \in S$. Note first that x must have a neighbour a_x in A, for otherwise $S - x$ would also be an (a, b)-separator, contradicting the minimality of S. Likewise, y has a neighbour a_y in A.

Since G_A is connected, there is a path from a_x to a_y using only vertices in A. Thus, there exists a path from x to y for which all intermediate vertices are in A. Among all such paths, let the shortest one be $x - a_1 - a_2 - \cdots - a_k - y$, and note that it has length at least 2 since x and y are not adjacent. Similarly we can find a shortest path from x to y for which all intermediate vertices are in B. See also Figure 3.5.

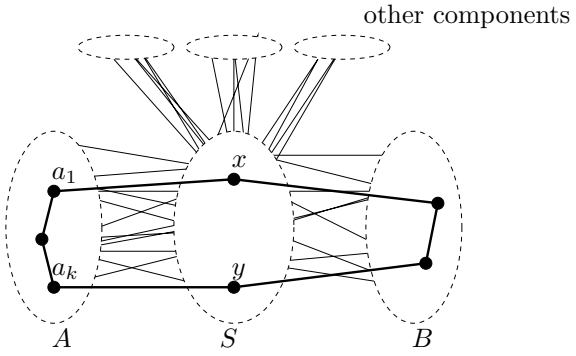


Figure 3.5: A minimum (a, b)-separator must be a clique.

Combining the two paths yields a cycle of length at least 4, which must have a chord since G is chordal. However, there is no chord in the cycle from x or y to either A or B since we chose the shortest paths from x to y in each component. Neither is there an edge from A to B since A and B are two different components. The only other possibility is for there to be a chord between x and y, but x and y are not adjacent. So we have a contradiction, which means that not both x and y can be in S. This finishes the proof. □

Note that the last claim of this proof actually shows that any minimal (a, b)-separator in a chordal graph must be a clique.

3.3 Finding a perfect elimination order

We have already outlined in Section 3.2.2 how to find a perfect elimination order if the graph has one. It is not hard to see that this algorithm can be implemented in polynomial time.

In this section, we will now be concerned with how to find a perfect elimination order very quickly, i.e., in linear time. We will give two different algorithms for this problem, and analyze one in detail.

3.3.1 Maximum Cardinality Search

The first algorithm for finding a perfect elimination order is called Maximum Cardinality Search or MCS, and was presented by Rose and Tarjan in 1975 [RT75]. The algorithm works by repeatedly choosing the vertex for which the maximum number of neighbours has already been chosen, breaking ties arbitrarily. See also Figure 3.6.

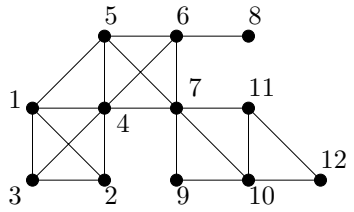


Figure 3.6: An example of Maximum Cardinality Search.

One can show that if the graph is chordal, then the vertex order $v_1, v_2, \ldots, v_n$ found by this algorithm is a perfect elimination order. The proof is omitted.

3.3.2 Lexicographic BFS

A second algorithm for finding a perfect elimination order is called lexicographic BFS or lexBFS, and was presented by Rose, Tarjan and Lueker in 1976 [RTL76]. The algorithm works by assigning labels to vertices and then choosing the next vertex as the one with the label that is lexicographically largest, breaking ties arbitrarily. Here, a label L_a is lexicographically larger than label L_b if L_a would appear after L_b in a phone book. In particular, n is smaller than $n \circ (n-1)$. The smallest possible label is the empty label, which we denote by $\emptyset$.

For all vertices v, let the label $L(v)$ be $L(v) = \emptyset$
for $i = n, \ldots, 1$
  Let v_i be a vertex that has not yet been chosen and that
    has the lexicographically largest label among all unchosen vertices
  For all unchosen neighbours v of v_i, set $L(v) = L(v) \circ i$

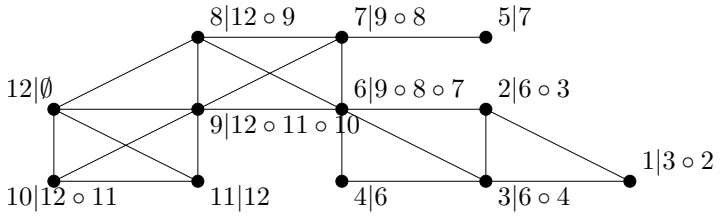


Figure 3.7: An example of lexBFS. A label $i|j$ means that this vertex became v_i, and at the time had label j.

We want to clarify where the name "lexicographic BFS" comes from. In particular, let us consider the tree of exploration, i.e., mark for each vertex from which vertex it first was discovered.

The first vertex to be discovered was v_n. All its neighbours attach "n" to their label. The next vertex to be visited is v_{n-1}, which is one of the neighbours of v_n. All its neighbours attach $n-1$ to their label. Now the next vertex to be visited is one that has the largest label. If there are any vertices left that are neighbours of v_n, then their label starts with

n and hence are largest. So we first visit all other neighbours of v_n before proceeding with neighbours of v_{n-1}. Therefore, this is indeed a BFS. However, the labels influence which of the remaining neighbours of v_n is taken first: if any neighbour of v_n is also adjacent to v_{n-1}, then it is taken before other neighbours of v_n. So lexBFS is a BFS with a special order of the neighbours, which (in essence) ensures that those neighbours that form a clique are visited first.

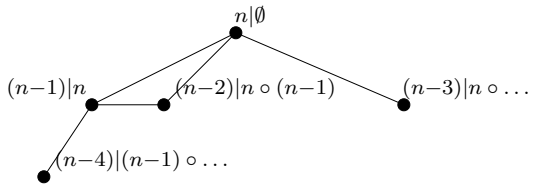


Figure 3.8: Illustration of why lexicographic BFS is called BFS.

Correctness of lexBFS

Now why does lexBFS work, i.e., why does it find a perfect elimination order if the graph is chordal? Note that it suffices to show that the last chosen vertex (i.e., v_1) is simplicial. The same proof then shows that every v_i is simplicial in the graph induced by $v_n, \ldots, v_i$, which means that its predecessors in the order $v_n, \ldots, v_1$ form a clique, so the order is a perfect elimination order. We will only give a sketch of a proof here, for details see [Gol80].

Suppose v_1 is not simplicial, thus there exist two neighbours v_i and v_j of v_1 that do not have an edge (v_i, v_j) between them. Without loss of generality, assume $j > i$, so v_j comes before v_i in the (supposed) perfect elimination order. See Figure 3.9 for an illustration.

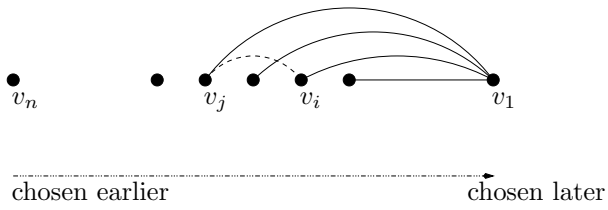


Figure 3.9: There are two neighbours of v_1 without an edge between them.

Note that both v_i and v_j were therefore chosen during lexBFS before we chose v_1. When we chose v_j, we added j to the labels of all its neighbours that were not chosen yet. In particular, we added j to $L(v_1)$, but we did *not* add j to $L(v_i)$, since v_i is not a neighbour of v_j. Thus we know:

$$\begin{aligned} L(v_1) &= \cdots j \cdots \\ L(v_i) &= \cdots \not{j} \cdots \end{aligned}$$

Observe that v_i was chosen by lexBFS before v_1 was chosen. Thus, the label of v_i must have been lexicographically not smaller than the label of v_1. But since $L(v_1)$ contains j and $L(v_i)$ does not, this is possible only if at some point earlier in the label, $L(v_i)$ is larger than

$L(v_1)$. In other words, somewhere earlier there must be an index (say k) that is contained in $L(v_i)$ and not in $L(v_1)$. So we must have

$$\begin{aligned} L(v_1) &= \cdots \not{k} \cdots j \cdots \\ L(v_i) &= \cdots k \cdots \not{j} \cdots \end{aligned}$$

Therefore, there must have been some vertex v_k, with $k > j$, that was incident to v_i but not to v_1. See Figure 3.10.

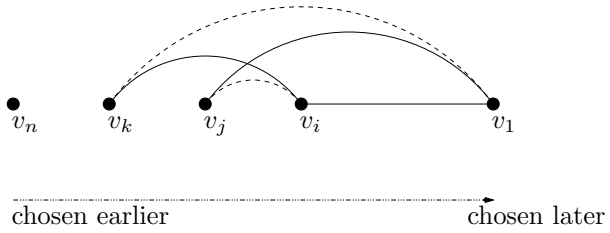


Figure 3.10: There must have been a vertex v_k before v_j that is incident to v_i but not to v_1.

Note that we cannot have edge (v_j, v_k), for if there were such an edge, then G would have a 4-cycle v_1, v_i, v_k, v_j without a chord, which contradicts that G is chordal. So there is no such edge.

Now we repeat this argument. The label of v_i contains k whereas the label of v_j does not contain k. So why was v_j chosen before v_i by the lexBFS? There must have been yet another vertex before v_k that is incident to v_j, but not to v_i. With a lot more arguing (here is where the details are omitted), we can show that this vertex also is not incident to any of v_k and v_1.

And then we repeat the argument again. Why was v_k chosen before v_j? And we get another vertex before v_k. And we repeat the argument again. And again. And again. So we end up is a situation shown in Figure 3.11.

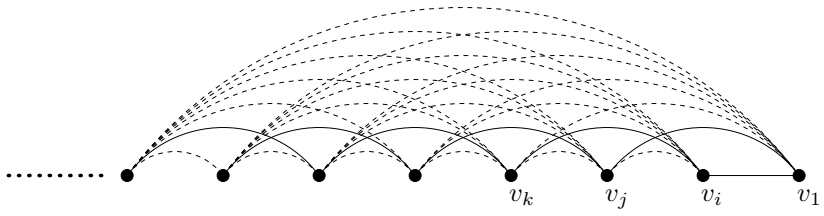


Figure 3.11: We can find more and more vertices, a contradiction.

This argument can be repeated ad infinitum, always adding another vertex that comes earlier in the ordering. But G is a finite graph, so this is a contradiction.

Implementation of lexBFS

To implement lexBFS efficiently, we use a list of buckets Q, i.e., each node in Q refers to another list (a "bucket") S_l. List S_l contains all vertices v with $L(v) = l$. Q is sorted by

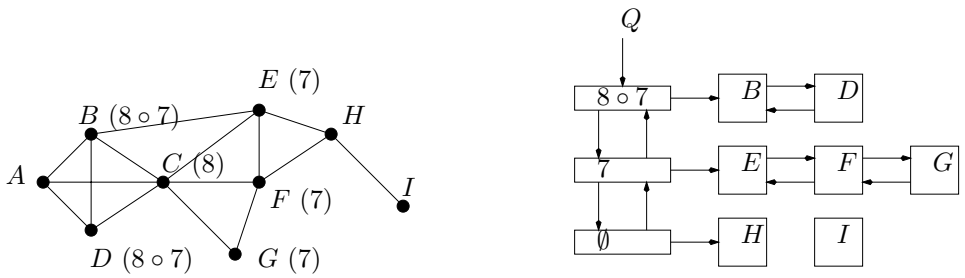


Figure 3.12: The list of buckets structure for a graph where vertex A was chosen first as v_8, and vertex C was chosen next as v_7.

descending lexicographic order (largest label first), and we will ensure that no bucket in Q is ever empty. See Figure 3.12 for an example.

Every vertex knows which bucket contains it (i.e., vertex v has a reference to bucket $S_{L(v)}$) and where it is in this bucket. Furthermore, each bucket knows its place in Q. All lists are doubly-linked for easier insertion and removal.

Initially all vertices have label $\emptyset$. Thus, Q contains just one bucket ($S_\emptyset$) which contains all vertices. Clearly, this can be initialized in $O(n)$ time.

Now to obtain the next vertex, and to update the data structure, we proceed as follows:

1. Let v_i be the first vertex in the first bucket.
   (Since Q is sorted, v_i has the lexicographically largest label.)
2. Delete v_i from its bucket $S_{L(v_i)}$.
3. Delete $S_{L(v_i)}$ if it is now empty.
4. For all neighbors w of v_i
5. If w is still in Q
   (Note that w has not been chosen yet; we need to update its label
   and therefore its place in Q.)
6. Find $S_{L(w)}$ and its place in Q
7. Find the bucket that precedes $S_{L(w)}$ in Q
8. If there is no such bucket, or if this is not $S_{L(w)\circ i}$
9. Create bucket $S_{L(w)\circ i}$ before $S_{L(w)}$ in Q.
10. Remove w from $S_{L(w)}$ and insert it into $S_{L(w)\circ i}$
11. Delete $S_{L(w)}$ if it is now empty
12. Update $L(w) = L(w) \circ i$

Now we analyze the running time of this algorithm. We can get v_i and remove it from Q (Steps 1-3) in $O(1)$ time. Updating Q for a neighbour w (Steps 5-12) takes $O(1)$ time, since we have stored all the necessary reference with each vertex. Thus the total cost per vertex v_i is $O(1 + deg(v_i))$, which yields a running time of $O(n + m)$.

It is not entirely obvious why the space requirement is also linear. Note that the length of the label of each vertex might be $\Omega(n)$, for example if we have a complete graph. But note that we add to the label of a vertex w only if we choose a neighbour v. Hence, the length

of the label of each vertex is proportional to its degree, and the total space for the labels is also $O(n+m)$.

Finally, we should be concerned with the correctness. The crucial claim is that if $S_{L(w)\circ i}$ exists, then it must be located just before $S_{L(w)}$ in Q. To see why this holds, assume that there is some other label between $L(w)$ and $L(w)\circ i$. This label must have the form $L(w)\circ j$ with $j < i$. But at this point in time, i is the smallest number that has been used for labels, so no such label in-between can exist, and hence we can indeed test whether $S_{L(w)\circ i}$ exists (and find the place for it if it does not) in constant time.

3.3.3 Testing a perfect elimination order

Both MCS and lexBFS give a perfect elimination order if the given graph G is chordal. To recognize a chordal graph, we can therefore run one of the above algorithms, and test whether the resulting order is a perfect elimination order. So this yields a new problem: Given a vertex order $v_1, \ldots, v_n$, how can we test efficiently whether this is a perfect elimination order?

A naive approach is to test for every vertex v whether every pair of predecessors of v is adjacent. This takes at least $\Omega(\sum_{v\in V}(\deg(v))^2)$ time, since for each v there are $\theta(\deg(v)^2)$ such pairs, and testing adjacency takes at least constant time (and usually more.)

With a more clever approach, we can reduce this to linear time. The idea is as follows. Assume that vertex v has a number of predecessors. Let u be the last of those predecessors. If we have a perfect elimination order, then the predecessors of v are a clique, and so in particular u must be adjacent to all other predecessors of v. We will test that. But once we have tested that, we need not test for any other edges between the predecessors of v! For if these other predecessors of v are also predecessors of u (recall that u is the last of v's predecessors), then we will test whether they are all adjacent when we test whether all predecessors of u form a clique.

The algorithm to do so uses one more trick: rather than testing immediately whether a given pair of vertices is adjacent, we gather all such pairs into one multi-set *Test* of pairs of vertices and test them all at once in the end.

Input: A graph $G = (V, E)$ and a vertex ordering $v_1, \ldots, v_n$
Output: "TRUE" if and only if $v_1, \ldots, v_n$ is a perfect elimination order

1. for $j = n$ down to 1 do
2. if v_j has predecessors
3. Let u be the last predecessor of v_j.
4. For all $w \in \mathit{Pred}(v_i)$, $w \neq u$
5. Add (u, w) to *Test*.
6. Test whether all vertex-pairs in *Test* are adjacent.

This shifts the burden of the work to the last step. However, this step can be implemented in $O(m + n + |\mathit{Test}|)$ time, or more precisely, for any multiset of k pairs of vertices, we can test in $O(m+n+k)$ time whether all pairs of vertices are adjacent (this is left as an exercise.)

The only remaining question is hence the size of *Test*. But one sees easily that every vertex v adds $\mathrm{indeg}(v) - 1$ pairs to *Test*, so $|\mathit{Test}| \leq m$, and the total running time is $O(m + n)$.

Combining lexBFS with this algorithm to test whether the resulting ordering is indeed a perfect elimination order, we obtain our main result:

Theorem 3.11 *Chordal graphs can be recognized in linear time.*

Chapter 4

Comparability graphs

The study of chordal graphs was motivated by the fact that every interval graph has a perfect elimination order, and the chordal graphs are exactly those graphs that have such an order. For a similar reason we now study comparability graphs. It is easy to see that the complement of every interval graph is a comparability graph, but the class of such graphs is larger than the complement of interval graphs.

4.1 Complements of interval graphs

Assume we have an interval graph G with a given representation by intervals. We now study the complement of G. So assume v and w are two vertices that are not adjacent in G (and hence adjacent in $\overline{G}$.) So the two intervals I_v and I_w of these vertices do not intersect. There are now two possibilities. Either I_v is to the left of I_w, or I_v is to the right of I_w.

This naturally imposes edge directions for $\overline{G}$. For a pair v, w of vertices with $(v, w) \notin E(G)$, direct the edge as $v \to w$ if I_v is to the left of I_w, and as $w \to v$ otherwise. See also Figure 4.1.

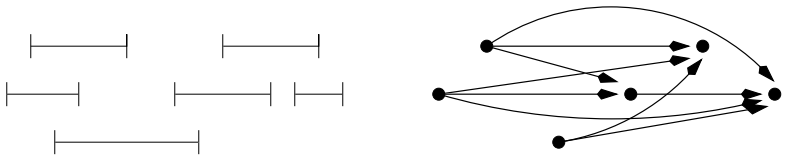


Figure 4.1: A set of intervals, and the edge directions in the complement of the interval graph defined by it.

A few things can be noted about this orientation of the edges:

- The orientation is *acyclic*, i.e., it does not contain a directed cycle. This holds because any edge goes from an interval farther to the left to an interval farther to the right.
- The orientation is *transitive*, i.e., if $u \to v$ is an edge and $v \to w$ is an edge, then $u \to w$ is also an edge. This holds because if I_u is to the left of I_v and I_v is to the left of I_w, then I_u is also to the left of I_w.

Thus, the complement of an interval graph has an edge orientation that is both acyclic and transitive. We introduce a special name for this.

Definition 4.1 *A graph that has an acyclic transitive orientation*[1] *is called a comparability graph.*

Note that a comparability graph may have many different edge orientations that are acyclic and transitive. Note that comparability graphs may also have orientations which aren't acyclic and transitive.

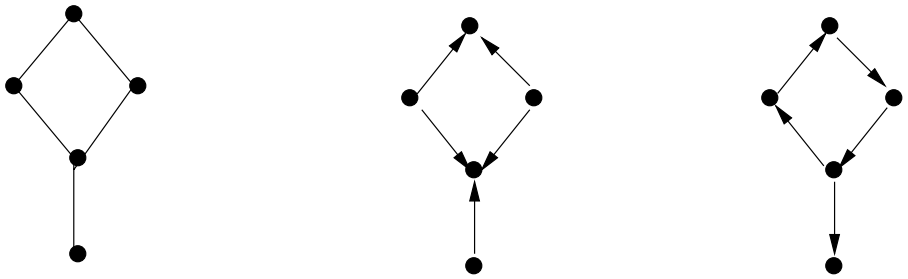


Figure 4.2: A comparability graph, an acyclic transitive orientation, and an orientation that is neither acyclic nor transitive.

4.2 Relationship to other graph classes

As discussed earlier, every complement of an interval graph is a comparability graph. Put into different words, every interval graph is a *co-comparability graph*, where co-comparability graphs are those graphs for which the complement is a comparability graph. More generally, for any graph class X, the graph class co-X is the set of graphs for which the complement belongs to X.

Does the reverse hold? Figure 4.3 shows a graph G that is a comparability graph, but the complement contains an induced 4-cycle, and hence is not an interval graph. So not all co-comparability graphs are interval graphs.

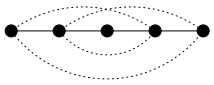


Figure 4.3: A comparability graph G whose complement is not an interval graph and not even a chordal graph.

There is no relationship between chordal graphs and comparability graphs. There are comparability graphs that are not chordal graphs (e.g. C_4 is a comparability graph), and

[1]Some references drop "acyclic", i.e., they define a transitive orientation to mean an acyclic orientation that has all transitive edges. We will try not to do this, though it might occasionally slip through from older lecture notes.

there are chordal graphs that are not comparability graphs. To see the latter, study the graph in Figure 4.4. This graph is chordal (i.e., a, b, c, d, e, f is a perfect elimination order), but not a comparability graph. To prove this, assume for contradiction that it has a transitive orientation. By symmetry, we may assume that (a, b) is oriented $a \to b$. Now we can conclude all other edge directions using transitivity. For example, (a, d) must be oriented $a \to d$, for if it were oriented $d \to a$, then by transitivity we would have to have edge $d \to b$, but there is no such edge. Similarly, since (a, e) does not exist we must have $e \to b$. Since (c, d) does not exist, we must have $a \to c$. Since (c, e) does not exist, we must have $c \to b$. But now we can't orient (c, f)! For if we have $c \to f$ then transitivity implies an edge $a \to f$, which doesn't exist, and if we have $f \to c$ then transitivity implies an edge $f \to b$, which doesn't exist. Contradiction, so the graph is not a comparability graph.

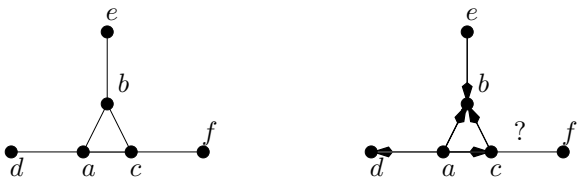


Figure 4.4: A chordal graph that is not a comparability graph.

It is also not hard to show that there is no relationship between chordal graphs and co-comparability graphs. The graph in Figure 4.3 is a comparability graph, but its complement is not a chordal graph since it contains an induced 4-cycle. The complement of the graph in Figure 4.4 is a chordal graph, but its complement, as argued above, is not a comparability graph.

Every bipartite graph is a comparability graph. Assume G is a bipartite graph with vertex partition $V = A \cup B$. Direct every edge from the vertex in A to the vertex in B. This is clearly acyclic, and it also is transitive since there is no directed path of length at least 2. So this is an acyclic transitive orientation. The reverse clearly doesn't hold; for example K_3 is a comparability graph but not bipartite.

4.3 Comparability graph recognition

Golumbic gives a linear algorithm for recognizing comparability graphs [Gol80]; see also [MS89, Gol77]. This will not be reviewed here, and is the topic for a project.

4.4 Problems on comparability graphs

For the following algorithms, we will assume that one specific acyclic transitive orientation has been fixed. A comparability graph with a fixed acyclic transitive orientation is also called a *poset*. The topic of posets is huge, and we will barely scrape the surface of many beautiful results here. (In particular, we will not cover Dilworth's theorem, which is really a pity, but would take too much time.)

4.4.1 Clique and Colouring

For any graph with an acyclic orientation, we can define a layering of the graph recursively as follows: For every vertex w, set

$$L(w) = 1 + \max_{v \to w}\{L(v)\}. \tag{4.1}$$

Thus, if w has no incoming edges, then $L(w) = 1$. If w has incoming edges, then $L(w)$ is one larger than the largest layer among all predecessors of w. In particular therefore, any edge goes from a smaller layer to a layer that is at least one unit larger. See also Figure 4.5.

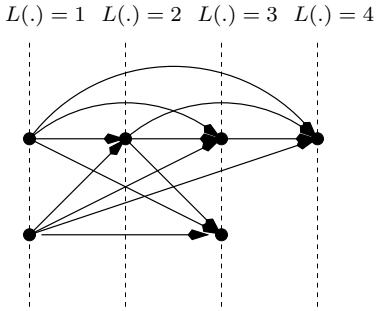


Figure 4.5: A poset, and the layer-numbers.

This layering can be find efficiently as follows. Since the graph is acyclic, we know that it has a *topological order*, i.e., a vertex order $v_1, \ldots, v_n$ such that every edge goes from left to right, i.e., every edge has the form $v_i \to v_j$ for $i < j$. This vertex order can be found in linear time with a directed depth-first search. Given such a vertex order $v_1, \ldots, v_n$, we can compute $L(v_i)$ for each vertex (in this order) using Equation 4.1, since then for all predecessors of v_i the layer is already computed, and all we have to do is to find the maximum layer among the predecessors and add 1 to it. This takes $O(\deg(v_i))$ time per vertex, and linear time overall.

This layering can be defined for any graph with an acyclic orientation (whether it is transitive or not), but if the orientation is also transitive, then we can use it to solve Clique and Colouring efficiently.

Theorem 4.2 *Let G be a comparability graph. Let k be the largest number of a layer in an arbitrary acyclic transitive orientation of G. Then $\omega(G) = \chi(G) = k$.*

Proof: Assign to every vertex v in G the color $L(v)$. Since any edge connects different layers, this is a legal colouring, and $\chi(G) \leq k$.

Now consider a vertex v_k that has $L(v_k) = k$. By definition of $L(.)$, therefore v_k has a predecessor, say v_{k-1}, that has $L(v_{k-1}) = k - 1$. Iterating this argument, we can find a directed path

$$v_1 \to v_2 \to \cdots \to v_k.$$

By transitivity, all edges $v_i \to v_j$ must exist for $1 \leq i < j \leq k$, so these k vertices form a clique and $\omega(G) \geq k$. The result now follows from $\chi(G) \geq \omega(G)$. □

Since the layer-numbers can found in $O(m + n)$ time, we can also find the maximum clique and the best colouring in linear time.

4.4.2 Independent Set

One might be tempted to think that the same layers could also be used to find a maximum independent set. This, however, is not the case. Figure 4.6 shows a poset with an independent set of size 4 for which both layers have size 3. Note in particular that this graph is bipartite, so this also dispels the (commonly believed) myth that in a bipartite graph the maximum independent set must be one of the two sides.

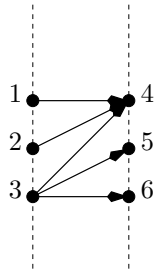


Figure 4.6: A poset for which neither layer gives the maximum independent set.

Another "natural" attempt at finding an independent set would be to use the greedy-algorithm that we saw for chordal graphs in combination with the topological order mentioned above. This "works" for colouring (in fact, the assignment of layers discussed in the previous section can be shown to be the same as the assignment of colours using this vertex order and the greedy-algorithm).

But unfortunately, the greedy-algorithm does not work for independent set, not even for the reverse topological order. The same graph in Figure 4.6 again serves as a counter-example. The vertices in this graph are labeled with the order in which they appear in a topological order. Using this for the greedy-algorithm in forward-direction would give $\{1, 2, 3\}$ as an independent set. Using it in backward-direction would give $\{4, 5, 6\}$ as an independent set. Neither of these is a maximum independent set.

Finding an independent set *can* be done in polynomial time on comparability graphs. In a nutshell, Dilworth's theorem can be used to prove that the size of a maximum independent set is the same as the size of a minimum set of directed paths that cover all vertices. Each such directed path is a clique, so such a set of paths is also called a minimum clique cover. Then one can use maximum flow algorithms to find the minimum clique cover, and recover the maximum independent set from it. Details of this are left as a project.

Chapter 5

Interval Graph Recognition

In this chapter, we return to interval graphs. Now that we have studied both chordal graphs and comparability graphs, we have all the tools needed to develop algorithms that test whether a given graph is an interval graph.

5.1 Interval graphs, chordal graphs, comparability graphs

We showed that every interval graph is a chordal graph, but the reverse is not true. We also showed that every interval graph is a co-comparability graph, but the reverse is not true.

Surprisingly enough, if we combine both properties, then the reverse is true, i.e., we can characterize interval graphs as follows.

Theorem 5.1 *A graph G is an interval graph if and only if G is a chordal graph and $\overline{G}$ is a comparability graph.*

The proof of this theorem is quite complicated, and will be omitted here, see Golumbic's book [Gol80]. Out of interest, we mention that it is usually done via a third equivalent statement, which we will encounter in the next section.

This theorem immediately gives rise to an algorithm to test whether a given graph is an interval graph. We have seen a linear-time algorithm to test whether a graph is chordal, and there is a linear-time algorithm to test whether a graph is a comparability graph, so combining the two gives an algorithm to test whether a graph is an interval graph.

Sadly, the resulting algorithm isn't necessarily linear. Note that the algorithm to test whether a graph is a comparability graph is linear in the size of the input graph, which is the complement graph of G. The complement of G may have a lot more edges than G. Therefore, the only time complexity we can conclude for this algorithm is $O(n^2)$.

5.2 Interval graphs and Maximal Cliques

The next section will be devoted to an interval graph recognition algorithm that achieves true linear-time complexity. Also, it will introduce us to the PQ-tree data structure, which will come in handy again later when we study planarity testing.

We first need yet another equivalent characterization of interval graphs.

Theorem 5.2 *A graph G is an interval graph if and only if the maximal cliques of G can be ordered consecutively, i.e., they can be listed as $C_1, \ldots, C_k$ such that if $v \in C_i$ and $v \in C_j$, then also $v \in C_h$ for all $i < h < j$.*

Proof: Assume G is an interval graph. Pick an interval representation such that all intervals have disjoint endpoints. Recall that if vertices are ordered by the left endpoint of their intervals, then this gives a perfect elimination order, and in particular, every maximal clique has the form $Pred(v) \cup \{v\}$ for some vertex v. So we can assign to each maximal clique C the integer that is the left endpoint of the vertex v for which $C = Pred(v) \cup \{v\}$. Then sort maximal cliques by this integer. See also Figure 5.1.

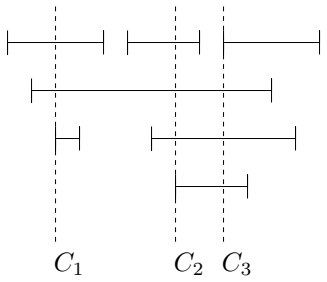


Figure 5.1: Sorting the maximal cliques by the left endpoints of the vertices that define them.

Now if vertex v belongs to both C_i and C_j, $i < j$, then the interval I_v of v intersects both the coordinate of C_i and the one of C_j. Therefore, it also intersects all coordinates of all cliques between C_i and C_j, which means that v belongs to all those cliques as well, so this is a consecutive ordering of the cliques.

The other direction is even easier. If $C_1, \ldots, C_k$ is a consecutive ordering of the maximal cliques, then set

$$I_v = [\min\{i : v \in C_i\}, \max\{j : v \in C_j\}],$$

i.e., the interval for v spans from the smallest to the largest clique that contains v. We need to verify that this is indeed an interval representation for the graph. Assume (v, w) is an edge. Then there exists some maximal clique that contains this edge, say C_h, and therefore $v \in C_h$ and $w \in C_h$. By definition of I_v and I_w, both intervals contain h, and hence they intersect. On the other hand, assume I_v and I_w intersect, say they both contain integer h. Since the clique ordering is consecutive, this implies that both v and w belong to C_h, which means that (v, w) must be an edge. □

The characterization from Theorem 5.2 will lead us to a linear-time interval graph recognition algorithm, which has two parts: Finding all maximum cliques, and testing whether the cliques can be ordered suitably.

5.2.1 Finding Maximal Cliques

In general, the problem of finding all maximal cliques is difficult. In fact, a graph can have an exponential number of maximal cliques (even if it is a comparability graph, see Figure 5.2), so the problem is not even in NP, much less is it polynomial.

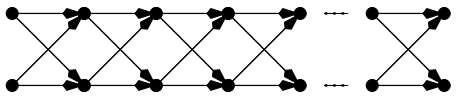


Figure 5.2: A comparability graph (all transitive edges are omitted) which has $2^{n/2}$ directed paths of length $n/2$, which all correspond to maximal cliques.

However, we want to find maximal cliques to test whether a graph is an interval graph. Therefore, we may assume that the graph is a chordal graph (which can be tested in linear time), since if the graph is not chordal, it certainly is not an interval graph. And for chordal graphs, we know that every maximal clique has the form $Pred(v) \cup \{v\}$ (after fixing some perfect elimination order), which shows that there are at most n maximal cliques.

Now, not every set $Pred(v) \cup \{v\}$ has to be a maximal clique. So we need an algorithm that finds all maximal cliques in a chordal graph. This is easy, once we prove the following simple characterization of maximal cliques.

Lemma 5.3 *Let $v_1, \ldots, v_n$ be a perfect elimination order. Then $C = Pred(v_i) \cup \{v_i\}$ is not a maximal clique if and only if there exists a successor v_j of v_i such that v_i is the last predecessor of v_j and $indeg(v_j) = indeg(v_i) + 1$.*

Proof: Assume there exists such a successor v_j. Since v_i is v_j's last predecessor, all predecessors of v_j are either v_i or a predecessor of v_i, so $Pred(v_j) \subseteq Pred(v_i) \cup \{v_i\} = C$. By $\text{indeg}(v_j) = \text{indeg}(v_i) + 1$, equality holds, so v_j is adjacent to all vertices in C, and $C \cup \{v_j\}$ is a bigger clique.

For the other direction, assume C is not maximal. Let j be minimal such that $v_j \notin C$ and $C \cup \{v_j\}$ is a clique. Vertex v_j is adjacent to v_i, but it is not a predecessor, otherwise it would be in C. So v_j is a successor of v_i, which implies $C \subseteq Pred(v_j)$ and $\text{indeg}(v_j) \geq \text{indeg}(v_i) + 1$.

We claim that v_i is the last predecessor of v_j. Assume it is not, so v_j has a predecessor v_k with $i < k < j$. Then v_k is adjacent to all of C, and $C \cup \{v_k\}$ is a clique, contradicting the minimality of j. Therefore, any predecessor of v_j is either v_i or a predecessor of v_i, so $Pred(v_j) \subseteq C$ and $\text{indeg}(v_j) \leq \text{indeg}(v_i) + 1$. This proves the claim. □

So to find the maximal cliques in linear time, we proceed as follows.

For $j = 1, \ldots, n$
  Find all predecessors of v_j.
  Store $\text{indeg}(v_j)$, and which vertex is the last predecessor of v_j.
For $i = 1, \ldots, n$
  Find all successors of v_i.
  For each successor v_j of v_i
    If v_i is the last predecessor of v_j and $\text{indeg}(v_j) = \text{indeg}(v_i) + 1$

Discard $Pred(v_i) \cup \{v_i\}$; it is not a maximum clique.
If $Pred(v_i) \cup \{v_i\}$ has not been discarded, output it as one maximal clique of the graph.

This algorithm takes $O(\deg(v) + 1)$ time per vertex, and hence has linear running time.

5.2.2 PQ-trees

Now we want to bring the maximal cliques into a consecutive order. To do so, we use another data structure, called PQ-trees. This data structure was introduced by Booth and Lueker [BL76], and is good not only for this problem, but also for planarity testing (see Chapter 15), and in general, can be used for testing the consecutive-ones-property for matrices.[1]

A PQ-tree has two types of nodes: P-nodes, and Q-nodes. A *Q-node* specifies that its children can be placed in forward or reverse order, while a *P-node* specifies that its children can be placed in any order. (Note that if a node has two children, it does not matter if it is a P- or a Q-node.) The permutations stored by the entire tree, then, is the set of all allowed permutations of the leaves.

An example of a PQ-tree is given in Figure 5.3. P-nodes are depicted using a circle, whereas Q-notes are depicted by a rectangle. The permutations allowed by the PQ-tree in Figure 5.3 include among others

- $AB \quad CDE \quad F \quad GH \quad IJ$
- $AB \quad EDC \quad F \quad GH \quad IJ$
- $EDC \quad BA \quad F \quad GH \quad IJ$
- $F \quad CDE \quad BA \quad JI \quad HG.$

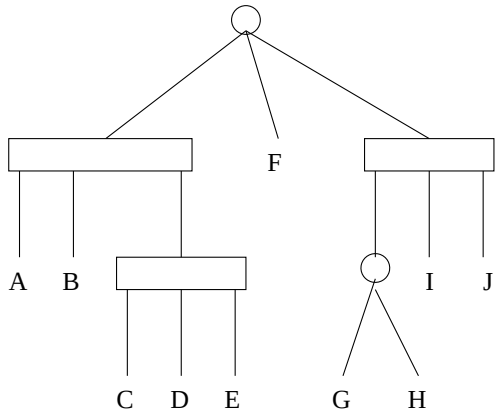


Figure 5.3: Example of a PQ-tree.

In general, PQ-trees serve to solve the following problem:

[1]This means that given a 0/1-matrix, we can permute the columns such that in each row all 1s are consecutive. Ordering the cliques can be phrased as such a problem by using the clique-vertex incidence matrix, but results in an algorithm that is too slow, since the matrix might be an $n \times n$-matrix.

Given: A finite set X and a collection $\mathcal{I}$ of subsets of X.

Want: A permutation $\Pi(X)$ such that for each $I \in \mathcal{I}$, the elements of I are consecutive in Π.

This problem is solved with a PQ-tree in an iterative approach. We start with the PQ-tree allowing all possible permutations (i.e., it consists only of all the leaves connected to a P-node). We then add each constraint in $\mathcal{I}$ one at a time.

To add a constraint $I \in \mathcal{I}$, we try to modify the tree until the elements of I are consecutive. This can be tested in $O(|I|)$ amortized time. If this cannot be done, then there is no possible permutation which satisfies the constraints. If it can be done, then we update the tree to reflect the new constraint. Booth and Lueker showed that this can be done with only a constant number of replacement rules in $O(|I|)$ amortized time. Therefore, in total, adding a constraint can be done in $O(|I|)$ amortized time. In all, determining whether or not a the set X of elements has a permutation in which all consecutiveness constraints $I \in \mathcal{I}$ are satisfied can be done in time and space $O(|X| + \sum_{I \in \mathcal{I}} |I|)$ amortized time. Full details can be found in Booth and Lueker [BL76].

For example, let $X = \{A, B, C, D\}$, and $\mathcal{I} = \{I_1 = \{A, B, C\}, I_2 = \{A, D\}\}$. Figure 5.4 shows the steps in adding the constraints to the PQ-tree. The final tree gives all possible orderings which satisfy the constraints: $\{B, C, A, D\}$, $\{C, B, A, D\}$, $\{D, A, C, B\}$, and $\{D, A, B, C\}$.

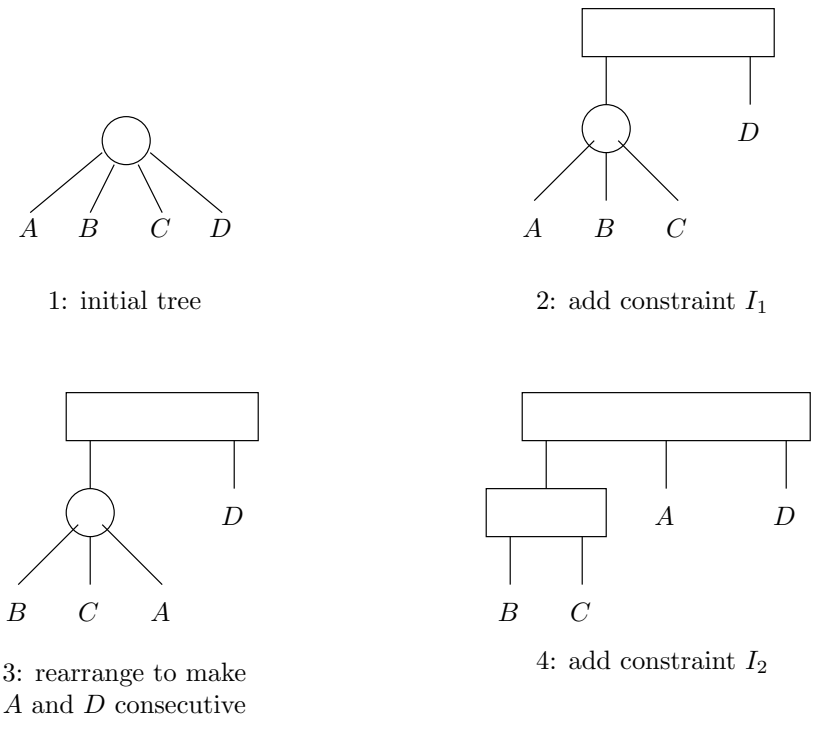


Figure 5.4: Example of adding constraints to a PQ-tree

5.2.3 Recognizing interval graphs with PQ-trees

Recognizing interval graphs with PQ-trees is now a matter of phrasing the problem of ordering cliques such that it fits the description of the problem solved with PQ-trees. This is done as follows. Let X, the leaves of the PQ-tree, be the set of all maximal cliques in a graph. Define $\mathcal{I}$, the consecutiveness constraints, as $\mathcal{I} = \{I_v | v \in V\}$, where I_v is the set of cliques which contain v.

Chordal graphs have at most n maximal cliques, so $|X| \leq n$. Also, $|I_v| \leq \deg(v) + 1$, since every maximal clique containing v has the form $w \cup Pred(w)$ for some vertex w that is either v or a successor of v. Therefore

$$|X| + \sum_{I \in \mathcal{I}} |I| \leq n + \sum_{v \in V} \deg(v) = n + 2m,$$

so the running time of the algorithm to find the order with PQ-trees is $O(n + m)$.

We summarize the interval graph recognition algorithm as follows:

- Test whether the graph is chordal, by running lexBFS. If the graph is not chordal, it is not an interval graph.
- Find all maximal cliques with the algorithm in Section 5.2.1.
- Create a PQ-tree with one element for every maximal clique in the graph.
- For every vertex v, create a set I_v consisting of all indices of maximal cliques that contain v.
- Using the PQ-tree, search for orders such that all elements in I_v are consecutive for all vertices v.
- If no such order exists, G is not an interval graph. Otherwise, reconstruct intervals for the vertices from the order of the maximal cliques as in Theorem 5.2.

Note that after the algorithm is run, if it was successful in adding all constraints, the PQ-tree will store all allowed orderings for the maximal cliques, and in particular, stores all possible interval graph representations of this graph.

5.3 Other algorithms for recognizing interval graphs

The algorithm presented in this lecture was the first linear-time algorithm for recognizing interval graphs. Many other algorithms have been developed since, mostly in the hopes of simplifying the algorithm by avoiding the PQ-tree. Korte and Möhring [KM89] developed an algorithm based on modified PQ-trees. Hsu and Ma [HM91], and Hsu on his own [Hsu93] showed an algorithm which did not use PQ-trees.

Habib, Paul, and Vincent demonstrated an algorithm for recognizing interval graphs in an unpublished paper using a modified lexBFS, and Corneil, Olariu and Stewart [COS98] recognized interval graphs by running lexBFS five times, using different criteria for breaking ties, and starting each subsequent run at the last vertex chosen in the previous run.

Chapter 6

Friends of Interval Graphs

6.1 Perfect graphs

One crucial ingredient in the fast algorithms for clique and coloring in chordal graphs and comparability graphs was that we could prove $\chi(G) = \omega(G)$. This section will introduce a graph class that is defined via this property.

Definition 6.1 *A graph G is* perfect *if $\omega(G) = \chi(G)$, and $\omega(H) = \chi(H)$ for every induced subgraph H of G.*

Note that by definition, every induced subgraph of a perfect graph is again perfect. Also, from what we have seen in previous chapters, we know that interval graphs, chordal graphs, and comparability graphs are perfect. But the class of perfect graphs is much bigger than that, and it is easy to find a perfect graph that is neither an interval graph, nor a chordal graph, nor a comparability graph. (This is left as an exercise.)

Not all graphs are perfect, for example an odd-length cycle C is not perfect since $\omega(C) = 2$ and $\chi(C) = 3$.

Recall that a maximum independent set in a graph G corresponds to a maximum clique in the complement $\overline{G}$. For perfect graphs, a maximum clique corresponds to a minimum colouring. So is there a similar problem that corresponds to independent set? There is, and in fact, we have seen this when we outlined how to find an independent set in a comparability graph.

Definition 6.2 *A* clique cover *of a graph G is a set of cliques such that every vertex in G belongs to one clique. We use $\kappa(G)$ to denote the size of the minimum clique cover.*

It is easy to see that $\kappa(G) = \alpha(\overline{G})$, since each clique in G is an independent set of $\overline{G}$, and a coloring can also be seen as a partition into independent sets. Therefore, if $\overline{G}$ is a perfect graph, then $\alpha(H) = \kappa(H)$ for all induced subgraphs H of G. We call such a graph a co-perfect graph. As it turns out, this does not define a new graph class.

Theorem 6.3 (Perfect graph theorem) *A graph is perfect if and only if its complement is perfect.*

This theorem will not be proved here. Using it, we can see that co-comparability graphs are also perfect. Figure 6.1 shows the relationship between the graph classes which we have seen so far, where arrows depict the subset relationship.

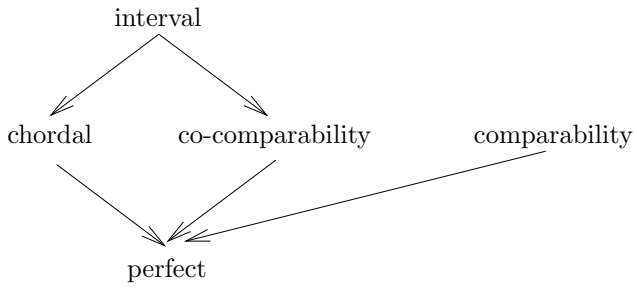


Figure 6.1: Relationship between graph classes.

Recall any odd-length cycle is not perfect. These graphs are *minimally imperfect*, since any induced subgraph of them is perfect. We call an odd-length cycle also an *odd hole*. By the perfect graph theorem, also the complement of an odd hole is not perfect. We call a complement of an odd hole an *odd anti-hole.*

Interestingly, the odd holes and anti-holes seem to be the only minimally imperfect graph, i.e., a graph seems to be perfect if and only if it has no induced odd hole or odd anti-hole. This is called the strong perfect graph conjecture and was posed by Claude Berge in 1960.

Conjecture 1 (Strong Perfect Graph Conjecture) *A graph is perfect if and only if it doesn't have an odd hole or an odd anti-hole as induced subgraph.*

Even though this conjecture was widely believed to be true, nobody had been able to prove it, and a lot of literature has been developed under the assumption that it holds. Finally, in 2002 Chudnovsky and Seymour completed a proof of the Strong Perfect Graph Conjecture, based on earlier work with Robertson and Thomas. (The paper, which is available in a preliminary version from R. Thomas' web-site, has 146 pages and will certainly not be reviewed here. See [CRST03] for an overview.)

Chudnovsky and Seymour also succeeded (apparently independently from the proof of the Strong Perfect Graph Conjecture) in giving a polynomial-time algorithm to test whether a graph is perfect [CS04]. The same result was found independently by Cornuéjols, Liu and Vuskovic [CLV03].

Solving a number of graph problems can be done in polynomial time on perfect graphs, though the algorithms to do so are quite involved and will not be reviewed in this class.

Theorem 6.4 *We can solve Clique, Colouring, and Independent Set in a perfect graph in polynomial time.*

6.2 Intersection Graphs

Interval graphs were defined as graphs that can be represented by intersecting intervals. We now generalize this concept to other types of objects to be intersected.

Definition 6.5 *A graph G is an intersection graph if there exists a universe $\mathcal{X}$ and for every vertex v a set $S_v \subseteq \mathcal{X}$ of objects of the universe such that edge (v, w) exists if and only if S_v and S_w intersect, i.e., $S_v \cap S_v \neq \emptyset$.*

It would be easy to show examples of graphs that are intersection graphs, but surprisingly enough, there are no graphs that are not intersection graphs.

Theorem 6.6 [Mar45] *Every graph is an intersection graph.*

Proof: Let the universe $\mathcal{X}$ be the set of edges, and for each vertex v, let S_v be the set of edges that is incident to v. Given two vertices v and w, then S_v and S_w intersect if and only if there exists an edge incident to both v and w, which holds if and only if v and w are adjacent. □

However, one can make the class of intersection graphs more interesting by imposing conditions on the universe, or the sets of the vertices. This gives rise to a huge number of graph classes. We will necessarily only study a small set of such classes here, and restrict the attention to classes that are similar to interval graphs.

6.2.1 Chordal Graphs

We first return to an old friend: chordal graphs. While neither of their equivalent definition (no induced k-cycle, has a perfect elimination order) hints at this, they can actually nicely be represented as intersection graphs.

Theorem 6.7 *A graph G is chordal if and only if it is the vertex-intersection graph of subtrees a free tree, i.e., if and only if there exists a tree T and a subtree T_v for every vertex in G such that (v, w) is an edge if and only if T_v and T_w have a vertex in common.*

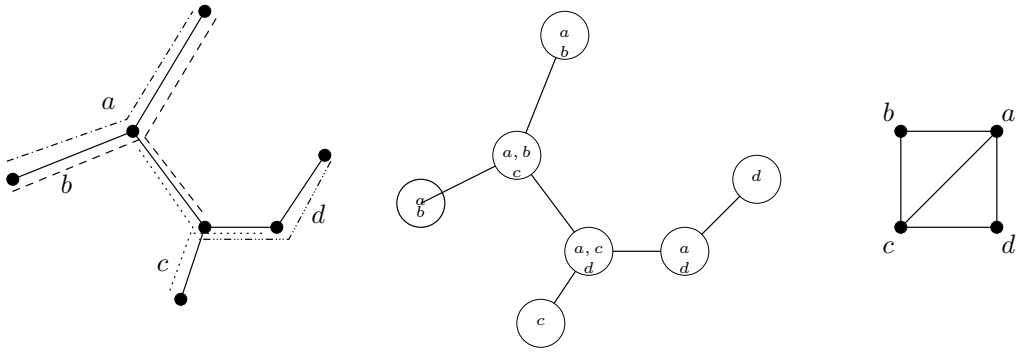


Figure 6.2: An intersection graph of subtrees of a tree. We depict the intersection graph in two different ways: once by drawing the subtrees with different line-styles, and once by writing onto each tree-node which vertices contain this tree-node in their subtrees.

The proof of this theorem is relatively straightforward and will be left as an exercise.

Note that the theorem nicely explains how chordal graphs generalize interval graphs. Interval graphs can be seen as graphs that are vertex-intersection graphs of sub-paths of a path. Chordal graphs then generalize interval graphs by allowing the path to split into different directions, and hence using a tree instead of a path.

6.2.2 Circular-arc Graphs

Interval graphs are graphs where vertices are represented as intervals on the real line. We now study graphs where the line has been replaced by a circle.

Definition 6.8 *G is a* circular-arc graph *if it is the intersection graph of arcs on a circle. Thus, we can assign to each vertex v an arc A_v on a circle such that (v, w) is an edge if and only if A_v and A_w intersect.*

Every interval graph is a circular-arc graph. Figure 6.3 shows a collection of arcs (solid lines) on a circle (dotted line) and its corresponding circular-arc graph (dashed lines). This graph is C_5, which proves that circular-arc graphs (as opposed to interval graphs) are not perfect.

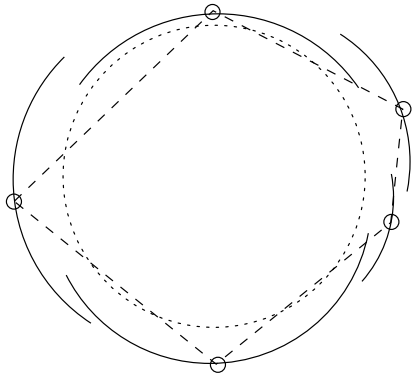


Figure 6.3: A circular-arc graph.

Circular-arc graphs can be recognized in linear time [McC03]. Some graph problems, such as Colouring, remain NP-hard on circular-arc graphs [GJMP78], whereas others, for example Clique, become polynomial-time for circular-arc graphs [Gav74].

6.2.3 Boxicity Graphs

Another way to generalize interval graphs is to move to higher dimensions.

Definition 6.9 *A graph is a boxicity-d graph if it is the intersection graph of d-dimensional intervals. Thus, we can assign an axis-aligned d-dimensional box B_v to every vertex such that (v, w) is an edge if and only if B_v and B_w intersect.*

Every interval graph is a boxicity-1 graph. Figure 6.4 shows that the 5-cycle C_5 and the complete bipartite graph $K_{3,6}$ are boxicity-2 graphs; both of them are not interval graphs. In particular, C_5 shows that boxicity-2 graphs (and boxicity-d graphs for $d \geq 2$) need not be perfect.

Every graph is a boxicity-d graph for sufficiently large d [Rob69]. Recognizing boxicity-2 graphs, and generally boxicity-d graph with $d \geq 2$ is NP-complete [Coz81, Yan82, Kra94].

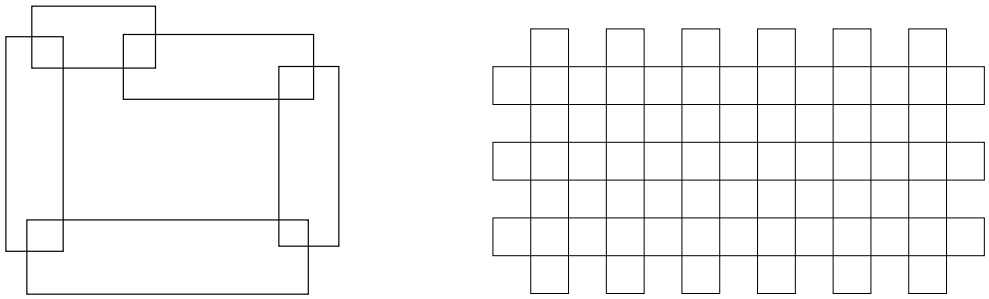


Figure 6.4: Boxicity-2 graphs: C_5 and $K_{3,6}$.

Many NP-hard graph problems remain NP-hard on boxicity-2 graphs, we will see this for Independent Set once we have studied planar graphs and visibility representations. On the other hand, Clique is polynomial on boxicity-2 graphs.

6.2.4 t-interval Graphs

In the final modification of interval graphs, we allow more than one interval for each vertex.

Definition 6.10 *A graph G is a t-interval graph if it is the intersection graph of sets of intervals of size at most t. Thus, every vertex is represented by a set of at most t intervals of the real line, and edge (v, w) exists if and only if some interval of v intersects some interval of w.*

The 1-interval graphs are exactly the interval graphs. Figure 6.5 shows that the 5-cycle C_5 and the graph obtained from subdividing every edge of the complete graph K_5 are 2-interval graphs; both of them are not interval graphs. In particular, C_5 shows that 2-interval graphs (and t-interval graphs for $t \geq 2$) need not be perfect.

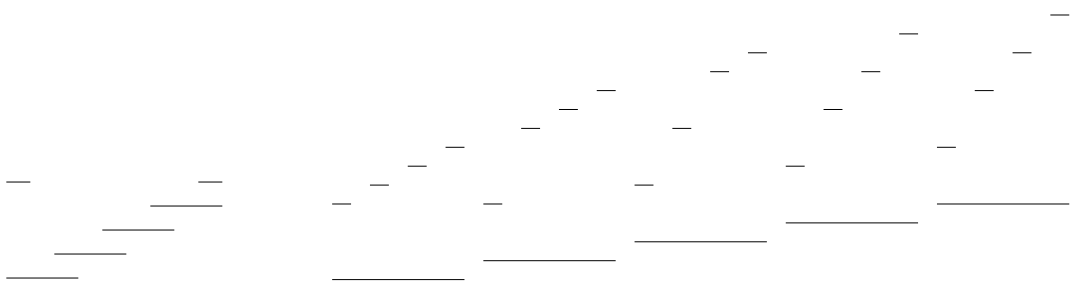


Figure 6.5: 2-interval graphs: C_5, and the graph obtained by subdividing every edge of K_5. In our representation, every vertex corresponds to one row.

Every graph G is a t-interval graph for sufficiently large t (one can show $t \leq \lceil(\Delta+1)/2\rceil$, where Δ is the maximum degree of G [GW80].) It is also not hard to show that every tree is a 2-interval graph. Recognizing t-interval graphs is NP-hard, even for $t = 2$ [WS84]. [1]

[1] What problems are NP-hard? What problems become polynomial?

Boxicity-d vs. t-interval

One might think that boxicity-d graphs should be related to t-interval graphs in some way. In both of them, vertices are represented by a set of intervals. However, there are a number of differences, which we summarize for $d = t = 2$ as follows:

- Each vertex v in a boxicity-2 graph is a *cross product* of two intervals I_v^x and I_v^y in x and y dimension respectively, that is $v = I_v^x \times I_v^y$. But for a 2-interval graph each vertex v is the *union* of two intervals I_v^1 and I_v^2, that is $v = I_v^1 \cup I_v^2$.

- For an edge (v, w) in a boxicity-2 graph *both* I_v^x, I_w^x intersect and I_v^y, I_w^y intersect, that is $I_v^x \cap I_w^x \neq \emptyset$ *and* $I_v^y \cap I_w^y \neq \emptyset$. But in a 2-interval graph *some* pair of (I_v^1, I_w^1), (I_v^1, I_w^2), (I_v^2, I_w^1) or (I_v^2, I_w^2) intersects, that is $I_v^1 \cap I_w^1 \neq \emptyset$ *or* $I_v^1 \cap I_w^2 \neq \emptyset$ *or* $I_v^2 \cap I_w^1 \neq \emptyset$ *or* $I_v^2 \cap I_w^2 \neq \emptyset$

In particular, neither graph class is a subset of the other. We showed earlier that $K_{3,6}$ is a boxicity-2 graph, and one can show that it is not a 2-interval graph. On the other hand, the graph obtained by subdividing every edge of K_5 is a 2-interval graph, but it is not a boxicity-2 graph (because otherwise we could find a planar drawing of K_5, which is impossible.)

Part II

Trees And Friends

Chapter 7

Trees and treewidth

We have already seen trees on a number of occasions. Trees are chordal, and trees are bipartite. For this reason, various graph problems on them are easy to solve in linear time. But as we will be sketching in this chapter (and seeing in more detail later on), just about any graph problem is solvable in linear time on trees.

This raises the question of generalization of trees for which a similar method works. Such generalizations exists and are known as graphs of bounded treewidth, which is what we study next. These graphs are also known as partial k-trees.

7.1 Trees

We study two kinds of trees: rooted trees, and trees where no root has been specified (also known as *unrooted trees* or *free trees.*) As usual, for any graph problem we assume that the input graph is undirected (hence a free tree), but as a first step to the algorithms to follow, we will convert it into a rooted tree. In particular, we will use the terminology for rooted trees, such as root, leaf, parent, child, ancestor, descendant, subtree (see also Appendix A).

Just about any graph problem is solvable in linear time on trees, by solving the problem recursively in the subtrees, and combining the solutions suitably. This leads to a simple recursive algorithm, which can be implemented in linear time. We illustrate this process by giving an algorithm to find a maximum matching in a tree. We define this problem first:

Definition 7.1 *Given an undirected graph $G = (V, E)$, a* matching *is a set of edges that does not have any endpoint in common. The* matching problem *is to find the maximum matching in a given graph.*

Figure 7.1 shows an example of a maximum matching in a tree. The matching problem is polynomial in any graph [Edm65], but the algorithm is neither particularly simple nor fast (the fastest known takes $O(\sqrt{n}m)$ time [MV80] .) Therefore, it does make sense to investigate how to find a maximum matching more efficiently in special graph classes such as trees.

Given a matching M, we call a vertex v *matched* if some incident edge of v belongs to M, and v *unmatched* otherwise. The example in Figure 7.1 has two unmatched vertices.

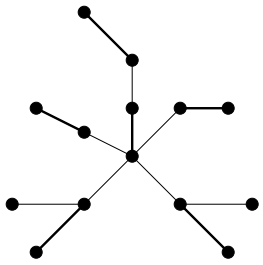


Figure 7.1: A maximum matching in a tree.

Maximum matching in a tree

To find a maximum matching in a given free tree T, we first root the tree at an arbitrary vertex, say r. For any vertex v, let T_v then be the subtree rooted at v. Then we define the following numbers:

$$\begin{aligned} f^+(T_v) &= \text{the size of the maximum matching of } T_v \text{ in which } v \text{ is matched} \\ f^-(T_v) &= \text{the size of the maximum matching of } T_v \text{ in which } v \text{ is unmatched} \\ f^m(T_v) &= \text{the size of the maximum matching of } T_v = \max\{f^+(T_v), f^-(T_v)\} \end{aligned}$$

Clearly, the size of the maximum matching in T is then $f^m(T_r)$, where r is the root of the tree. So this can be found in constant time once we have computed the functions.[1] Also, $f^m(T_v)$ can be computed in constant time once we have $f^+(T_v)$ and $f^-(T_v)$, so we will not be concerned with it from now on.

Base case

The idea is now that we can express $f^-(T_v)$ and $f^+(T_v)$ recursively. For the base case, assume that ℓ is a leaf, so T_ℓ contains only one vertex, ℓ. A graph without any edges has only one matching (the empty matching), which does not contain ℓ. So $f^-(T_\ell) = 0$. Note that $f^+(T_\ell)$ is not well-defined, since there exists no matching that contains ℓ. We could mark it as undefined, but as will be obvious from later formulas, it is also sufficient to set $f^+(T_\ell) = -\infty$. So we have

$$\begin{aligned} f^-(T_\ell) &= 0 \\ f^+(T_\ell) &= -\infty \end{aligned}$$

Recursive case

Now we consider the recursive case.

Lemma 7.2 *Assume vertex v has children $w_1, \ldots, w_k$, $k > 0$. Then*

$$f^+(T_v) = \max_{1 \le i \le k} \left\{ 1 + f^-(T_{w_i}) + \sum_{j \neq i} f^m(T_{w_j}) \right\}$$

[1] Note that this only finds the *size* of the maximum matching and not the matching itself. Recovering the actual matching to achieve this size can be done with standard techniques of dynamic programming, see for example [CLRS00].

Proof: Let M_v a matching that achieves $f^+(T_v)$, i.e., M_v is a matching in T_v for which v is matched, and $|M_v| = f^+(T_v)$. One edge in the matching is incident to v, say that this is edge $e = (v, w_i)$. Then $M_v - e$ contains only edges from $T_{w_1} \cup \cdots \cup T_{w_k}$, and thus can be split into matchings $M_1, \ldots, M_k$ for these k subtrees.

Since w_i is incident to e, and we removed edge e, matching M_i leaves w_i unmatched. So $|M_i| \leq f^-(T_{w_i})$. For all $j \neq i$, matching M_j may or may not have w_j matched, so we only know $|M_j| \leq f^m(T_{w_i})$. Altogether therefore,

$$\begin{aligned} f^+(T_v) &= |M| = |e \cup M_1 \cup \cdots \cup M_k| = 1 + |M_i| + \sum_{j \neq i} |M_j| \\ &\leq 1 + f^-(T_{w_i}) + \sum_{j \neq i} f^m(T_{w_j}) \leq 1 + \max_{1 \leq i \leq k} \left\{ f^-(T_{w_i}) + \sum_{j \neq i} f^m(T_{w_j}) \right\} \end{aligned}$$

which shows the "$\leq$" inequality. For "$\geq$", we proceed similarly by composing a matching for T_v from the matchings of the subtrees. Let i be such that $\max_{1 \leq i \leq k} \{ f^-(T_{w_i}) + \sum_{j \neq i} f^m(T_{w_j}) \}$ is maximized. Let M_i be a matching that achieves $f^-(T_{w_i})$, and for $j \neq i$ let M_j be a matching that achieves $f^m(T_{w_j})$. Set $M_v = (v, w_i) \cup M_i \cup \bigcup_{j \neq i} M_j$. One can easily verify that this is a matching in T_v that leaves v matched, so

$$\begin{aligned} f^+(T_v) &\geq |M| = |(v, w_i) \cup M_i \cup \bigcup_{j \neq i} M_j| = 1 + |M_i| + \sum_{j \neq i} |M_j| \\ &= 1 + f^-(T_{w_i}) + \sum_{j \neq i} f^m(T_{w_j}) = 1 + \max_{1 \leq i \leq k} \left\{ f^-(T_{w_i}) + \sum_{j \neq i} f^m(T_{w_j}) \right\} \end{aligned}$$

which proves the other inequality and hence the lemma. □

The formula for $f^-(T_v)$ is proved similarly; we leave this to the reader.

Lemma 7.3 *Let v be a vertex in T, and let $w_1, \ldots, w_k$ be the children of v in T. Then*

$$f^-(T_v) = \sum_{i=1}^{k} f^m(T_{w_i})$$

Algorithm

The algorithm itself is extremely simple given the above definitions. While doing a post-order traversal of the tree, compute the values of $f^+(T_v)$, $f^-(T_v)$ and $f^m(T_v)$ for all vertices v from the formula. Since we use a post-order, the values for the children have been computed already at this point, and hence can simply be looked up. So this takes $O(1 + \deg(v))$ time per vertex v, and the running time is $O(m + n)$, which is $O(n)$ for a tree.

root the tree at an arbitrary vertex r
for all vertices v in the tree T in post order
    if v is a leaf, set $f^+(T_v)$ and $f^-(T_v)$ as discussed in the base case.

else compute $f^+(T_v)$, $f^-(T_v)$ from the values of v's children
compute $f^m(T_v) = \max\{f^+(T_v), f^-(T_v)\}$
output $f^m(T_r)$

Actually, there are even simpler algorithm for maximum matching on a tree, but this algorithm has the advantage that the exact same algorithm (but with different recursive formulas) works for many graph problems, for example Independent Set, Vertex Cover, and Dominating Set. All these problems can hence be solved in linear time on trees.

7.2 Treewidth

The crucial idea behind the dynamic programming algorithm for trees is that a tree can be broken into smaller parts with only one vertex in common (see Figure 7.2).

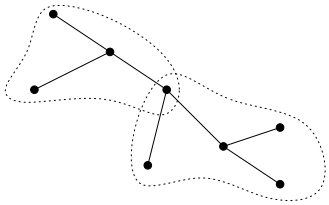


Figure 7.2: Subtrees of a tree have only one vertex in common.

The two subtrees have only one vertex in common, say vertex v. If we simply solve maximum matching in each subtree and return their union as an answer, then this might be incorrect, since v could be incident to two matching edges. But by computing maximum matchings while imposing constraints whether v is matched or not, and then considering all combinations of the two matchings which yield a correct matching will give us the maximum matching.

Now we generalize this idea to graphs with subgraphs whose intersections have at most k vertices. In a nutshell, therefore there will be only 2^k ways in which subgraphs can interact, and by storing them all and computing recursive functions for them all, we obtain an algorithm that solves various graph problems in $O(2^k \cdot n)$ time. This is linear as long as k is a constant.

7.2.1 Tree-Decompositions

To formalize this idea, we use the concept of a tree-decomposition.

Definition 7.4 *A tree-decomposition of a graph $G = (V, E)$ is a tree $T = (I, F)$ where each node $i \in I$ has a label $X_i \subseteq V$ such that:*

- $\bigcup_{i \in I} X_i = V$. *(We say that "all vertices are covered.")*
- *For any edge (v, w), there exists an $i \in I$ with $v, w \in X_i$. (We say that "all edges are covered.")*

- *for any $v \in V$, the nodes containing v in their label form a connected subtree of T. (We call this the "connectivity condition.")*

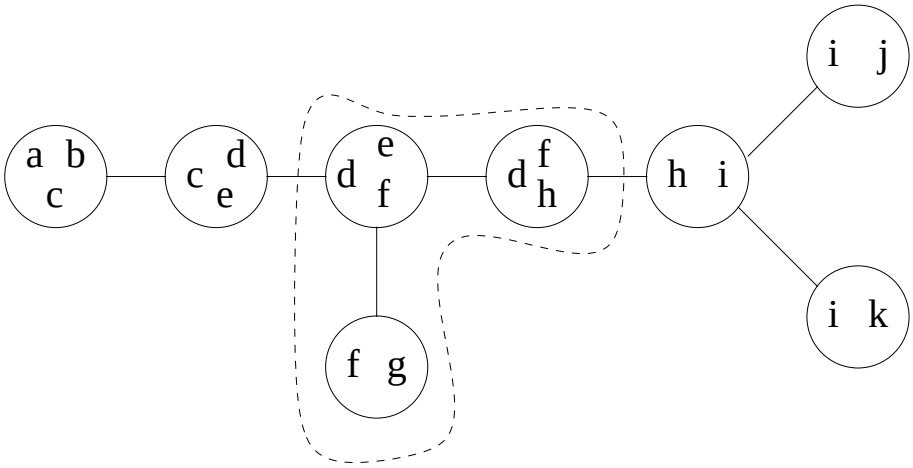


Figure 7.3: A tree-decomposition. We highlight the subset containing f, which is connected.

Note that the same tree decomposition can be used for a number of graphs. Let the *graph implied by a tree decomposition* be the graph obtained by adding all edges that are allowed in the tree decomposition, i.e., add an edge if and only if two vertices appear in a common label. As we have seen when studying intersection graphs, the graph implied by a tree decomposition is chordal. In fact, we can give an alternative definition of the same graph class via this observation; we will return to this in the next section.

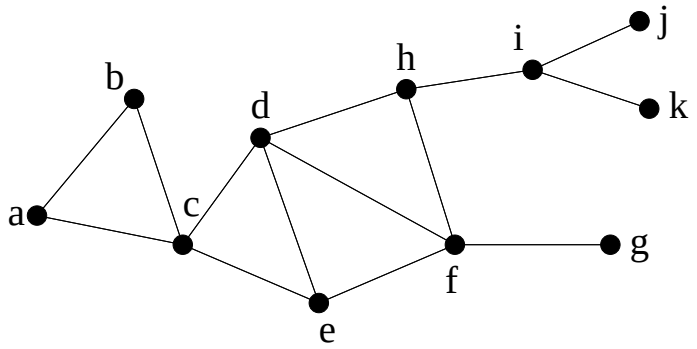


Figure 7.4: The chordal graph implied by the tree-decomposition in Figure 7.3.

A given tree decomposition can hence be used for any subgraph of the graph implied by it (though for a subgraph, a tree decomposition with fewer vertices per label is sometimes possible.)

Definition 7.5 *Given a tree-decomposition T=(I,F), the* width *(or* treewidth*) of the decomposition is* $\max_{i \in I} |X_i| - 1$.

Definition 7.6 *The* treewidth *of a graph G is the minimum k such that G has a tree-decomposition of width k.*

It is easy to see that every graph G has treewidth at most $n-1$, since a single node with all vertices in it is a tree decomposition for G and has width $n-1$. However, we are interested in graphs for which the treewidth is a small constant.

7.2.2 Closure properties

It is easy to see that if G is a graph of treewidth k, then every induced subgraph H has treewidth at most k. For if we take a tree decomposition of G, and simply delete all occurrences of all vertices in G that are not in H, then the resulting tree decomposition is a tree decomposition of H, and its treewidth has certainly not increased (and it might even have decreased.)

Lemma 7.7 *Graphs of treewidth at most k are closed under taking induced subgraphs.*

However, we can even extend this to arbitrary subgraphs. Recall again that having an edge (v, w) implies that v and w must be in a common node of the tree decomposition, but there is no need to have an edge if v and w are in a common node. Therefore, if we delete an edge from a graph of treewidth at most k, then the exact same tree decomposition serves as tree decomposition for the new graph, which hence also has treewidth at most k.

Lemma 7.8 *Graphs of treewidth at most k are closed under taking subgraphs.*

Graph of treewidth at most k are closed under another type of operation as well. Recall that contracting an edge (v, w) means deleting v and w and adding one new vertex x that is adjacent to all neighbours of both v and w.

Lemma 7.9 *If G has treewidth at most k, and H is obtained by contracting edge (v, w) in G, then H has treewidth at most k.*

Proof: Fix a tree decomposition of width k for G. Now replace every occurrence of either v or w by x. It is easy to verify that this tree decomposition covers all vertices and edges. To see why the subgraph for x is connected, note that the subgraphs for v and w each were connected, and that they had a node in common (the node for edge (v, w)), so the union of these two graphs is also connected. So this yields a tree decomposition for H of width k. □

A graph H is called a *minor* of G if H is obtained from G via some number of edge deletions and contractions. The above lemmas hence show:

Lemma 7.10 *Graphs of treewidth at most k are closed under taking minors.*

The topic of minors and graph classes that are closed under taking minors is vast, and some very exciting research has been done in it. We will return to this in Chapter 18.

7.3 Partial k-trees

Now we give a totally different characterization of the same graph class, using the earlier observation that the graph implied by a tree decomposition is chordal. We first study chordal graphs with a special property.

Definition 7.11 *A graph G is called a k-tree if G has a perfect elimination order such that $v_1, \ldots, v_k$ is a clique, and indeg$(v_i) = k$ for all $i \in \{k+1, \ldots, n\}$.*

See Figure 7.5 for an example of a 3-tree. Note that if G is a k-tree with at least $k+1$ vertices, then $\text{indeg}(v_{k+1}) = k$, which implies that v_{k+1} is incident to all of $v_1, \ldots, v_k$, which hence must form a clique since we have a perfect elimination order. So we could have dropped the first condition from the definition of a k-tree and obtain the same graph class.

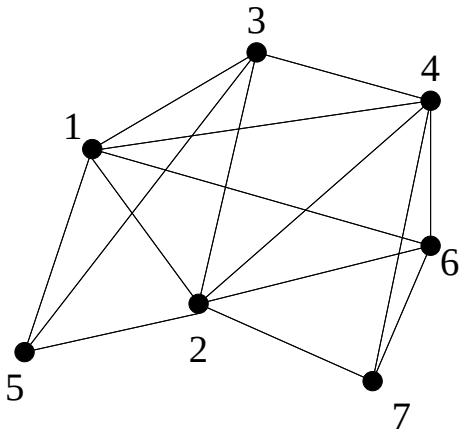


Figure 7.5: A k-tree with $k = 3$.

A k-tree with perfect elimination order $v_1, \ldots, v_n$ has $\binom{k}{2}$ edges in the clique $v_1, \ldots, v_k$, and $(n-k) \cdot k$ edges incident to $v_{k+1}, \ldots, v_n$. Therefore a k-tree has $k \cdot n - \frac{k^2}{2} - \frac{k}{2}$ edges, which in particular is linear if k is a constant.

Also note that a k-tree G has $\omega(G) \leq k+1$, since for a chordal graph G we have $\omega(G) = 1 + \max_v\{\text{indeg}(v)\}$. (In fact, we have $\omega(G) = k+1$ unless G has fewer than $k+1$ vertices.)

Definition 7.12 *A graph G is called a* partial k-tree *if G is a subgraph of a k-tree.*

Note that unlike a k-tree, a partial k-tree is not necessarily a chordal graph. For example, C_4 is a partial 3-tree (add edges until it is K_4, which is a 3-tree), but it is not chordal.

Clearly, if G is a partial k-tree, then it is a subgraph of a chordal graph G' with $\omega(G') \leq k+1$. The reverse direction also holds. To prove it, we need the following lemma, which we leave as an exercise.

Lemma 7.13 *Let G be a chordal graph with $\omega(G) = k+1$. Then we can add edges to G such that the resulting graph is a k-tree.*

Theorem 7.14 *The following statements are equivalent:*

i) G is a partial k-tree.

ii) G is a subgraph of a chordal graph G' with $\omega(G') \leq k+1$.

iii) G is a subgraph of a chordal graph G' with $\omega(G') = k+1$.

Proof: i) $\Rightarrow$ ii) follows immediately from the definition of a partial k-tree, since a k-tree is chordal and has clique-number at most $k+1$. iii) $\Rightarrow$ ii) is also trivial, for if G' has $\omega(G') < k+1$, then we can add vertices to G' until the maximum clique has size $k+1$. Finally, iii) $\Rightarrow$ i) holds because of Lemma 7.13, since we can add edges to G' to obtain a k-tree, thus any subgraph of G' is a partial k-tree. □

We will use the other two statements in Theorem 7.14 quite frequently as an alternative equivalent definition of partial k-trees that is often easier to handle than the original one.

As an illustration, we can use it to analyze partial 1-trees. By the theorem, these are the subgraphs of chordal graphs G with $\omega(G) = 2$. If $\omega(G) = 2$ for a chordal graph, then G cannot have any cycle (because the shortest-length cycle would have length 3 by chordality and give a 3-clique.) So $\omega(G) = 2$ implies that G is a forest. On the other hand, every forest is chordal and has $\omega(G) \leq 2$, so the partial 1-trees are exactly the forests.

7.4 Partial k-trees and treewidth at most k

Now we show that the two concepts of the previous two sections really defined the same graph class. To do so, we need to analyze intersection graphs of subtrees of a tree, and in particular, provide the proof that these are exactly the chordal graphs.

7.4.1 Partial k-trees to tree decomposition

Lemma 7.15 *If G is chordal, then G has a tree decomposition of width $\omega(G) - 1$.*

Proof: Let $v_1, \ldots, v_n$ be a perfect elimination order of G. We will build the tree decomposition for G by induction on i, and build one such that for every $j \leq i$ there exists a node in the tree decomposition whose label contains all vertices in $\{v_j\} \cup \mathit{Pred}(v_j)$.

Let $k = \omega(G) - 1$. The base case occurs for $i = k+1$. We add all of $v_1, \ldots, v_{k+1}$ into one node of the tree decomposition. Clearly this satisfies all conditions.

Now assume $i > k+1$, and we have a tree decomposition for the graph induced by $v_1, \ldots, v_{i-1}$ such that for all $j \leq i-1$ there exists a node in the tree decomposition whose label contains all vertices in $\{v_j\} \cup \mathit{Pred}(v_j)$. Consider vertex v_i, and let v_h be the last predecessor of v_i.

There exists a node a of the tree decomposition that contains all of $\mathit{Pred}(v_h) \cup \{v_h\}$. Add a new node b to the tree decomposition, make it adjacent to a, and give it label $\{v_i\} \cup \mathit{Pred}(v_i)$. See Figure 7.6. The new label has size $1 + \text{indeg}(v_i) \leq \omega(G)$.

Clearly, this tree decomposition covers all vertices and edges, has width at most $k = \omega(G) - 1$, and has a node that contains $\mathit{Pred}(v_j) \cup \{v_j\}$ for all $j \leq i$. Why does it satisfy the

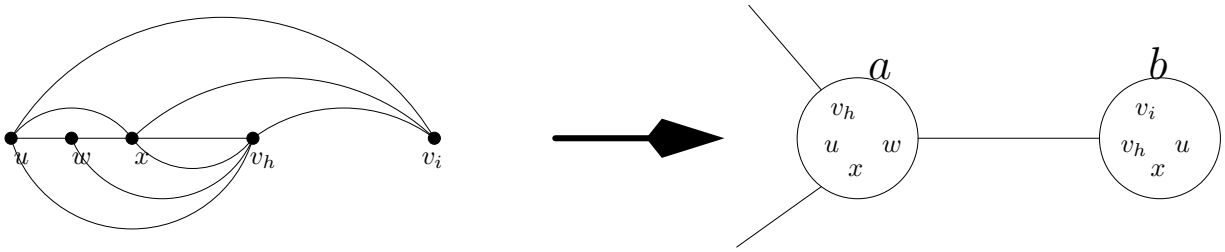


Figure 7.6: Adding a new node to the tree decomposition.

connectivity condition? Since v_h is the last predecessor of v_i, we have $Pred(v_i) \subseteq Pred(v_h) \cup \{v_h\}$. Therefore, the only vertices for which we have added a new node containing them are v_i (for which b is the only node containing it, so this is connected), or a vertex in $Pred(v_i)$, for which a also contains it, so its subtree continues to be connected. □

Corollary 7.16 *If G is a partial k-tree, then G has treewidth at most k.*

Proof: Since G is a partial k-tree, it is a subgraph of a chordal graph G' with $\omega(G') = k+1$. Build the tree decomposition of width $\omega(G') - 1 = k$ for G'; this is also a tree decomposition for G. □

For future reference, we want to analyze the tree decomposition obtained in Lemma 7.15 for k-trees.

Corollary 7.17 *If G is a k-tree, then there is a tree decomposition of G such that all nodes have a label of size $k+1$. Furthermore, if i and j are two adjacent nodes, then $|X_i \cap X_j| = k$.*

Proof: This follows directly from the tree decomposition built for a k-tree with the proof of Lemma 7.15. Let G be a k-tree with perfect elimination order $v_1, \ldots, v_n$, then $\text{indeg}(v_i) = k$ for all $i \geq k+1$. The proof of Lemma 7.15 adds a node for every vertex v_i, $i \geq k+1$ (the base case covers the vertex v_{k+1}), and gives it label $\{v_i\} \cup Pred(v_i)$ (this holds for the base case since G is a k-tree.) So all labels have size $k+1$.

Now assume i and j are two adjacent nodes, and assume that they have been added for vertices v_i and v_j for $i, j \geq k+1$. Assume $i < j$, then by construction v_i must be the last predecessor of v_j. Therefore $Pred(v_j) \subset \{v_i\} \cup Pred(v_i)$, which shows that

$$X_i \cap X_j = (v_i \cup Pred(v_i)) \cap (v_j \cup Pred(v_j) = (v_i \cup Pred(v_i)) \cap Pred(v_j) = Pred(v_j)$$

so $|X_i \cap X_j| = |Pred(v_j)| = \text{indeg}(v_j) = k$. □

7.4.2 Tree decomposition to partial k-trees

Now we prove the other direction.

Lemma 7.18 *Let G be the graph implied by a tree decomposition of width k. Then G has a simplicial vertex of degree at most k.*

Proof: We proceed by induction on the size of the tree decomposition. In the base case, the tree decomposition has only one node. Then the implied graph is the complete graph with at most $k+1$ vertices, for which all vertices are simplicial and have degree at most k.

Now assume that the tree decomposition has at least two nodes, and let i be a leaf of the tree, with unique neighbour j. We have two subcases. If $X_i \subseteq X_j$, then we can delete node i from the tree decomposition and keep the exact same implied graph. By induction, this graph has a simplicial vertex of degree at most k.

If, on the other hand, X_i is not a subset of X_j, then there exists a vertex v that is in X_i but not in X_j. See Figure 7.7. By the connectivity condition (and since j is the only neighbour of i), this is the only occurrence of v in the tree decomposition, and in particular all neighbours of v are in X_i. By $|X_i| \leq k+1$, therefore $\deg(v) \leq k$. Furthermore, in the implied graph the vertices of X_i form a clique, so the neighbours of v form a clique and v is simplicial. □

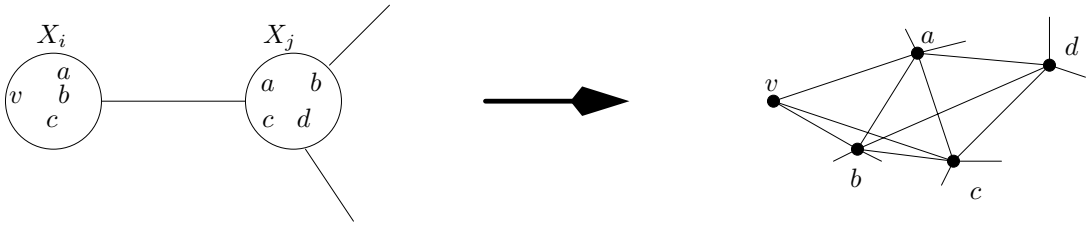


Figure 7.7: A vertex that appears only in a leaf of the tree decomposition is simplicial.

Corollary 7.19 *Every graph of treewidth k is a partial k-tree.*

Proof: Let G be a graph of treewidth k. Fix a tree decomposition of G that has width k, and let G' be its implied graph. G' has a simplicial vertex of degree at most k, call it v_n.

Now consider $G' - \{v_n\}$. Deleting v_n from the tree decomposition gives a tree decomposition for $G' - \{v_n\}$ of width at most k, so $G' - \{v_n\}$ has a simplicial vertex of degree at most k. Repeating this argument, we can obtain vertices $v_n, v_{n-1}, \ldots, v_1$ such that each v_i is simplicial and has degree at most k in the graph $G' - \{v_{i+1}, \ldots, v_n\}$. Then $v_1, \ldots, v_n$ is a perfect elimination order for which every vertex has indegree at most k, which shows that G' is chordal and has $\omega(G') \leq k+1$. Therefore G is a partial k-tree. □

Chapter 8

Series-parallel graphs

In this chapter we explore the structure of (partial) k-trees for small values of k. For $k = 1$ we have seen that partial 1-trees are exactly forests. For $k = 2$, partial 2-trees have an equivalent characterization that appears to have nothing to do with treewidth (and was known much before treewidth was ever studied), namely, they are series-parallel graphs.

8.1 2-terminal series-parallel graphs

Definition 8.1 *A* 2-terminal series-parallel graph *(also called* 2-terminal SP-graph*) is a graph with two distinguished vertices (the* terminals*) that is obtained as follows:*

- **Base case:** *A single edge (s, t) is a 2-terminal-SP-graph with terminals s and t.*
- **Step:** *If G_1 and G_2 are 2-terminal SP-graphs with terminals s_i and t_i $(i = 1, 2)$, then we can combine them in two possible ways:*
  - Combination in series: *The graph formed by identifying t_1 with s_2 is a 2-terminal SP-graph with terminals s_1 and t_2.*
  - Combination in parallel: *The graph formed by identifying s_1 with s_2, and t_1 with t_2, is a 2-terminal SP-graph with terminals $s_1 = s_2$ and $t_1 = t_2$.*

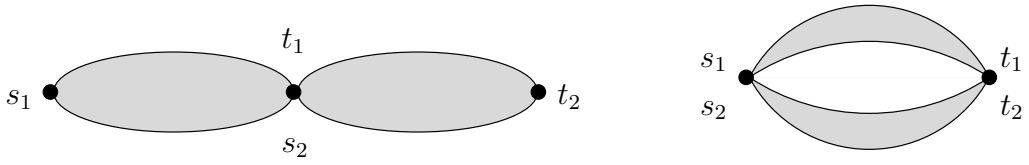


Figure 8.1: Combination in series and in parallel.

Note that 2-terminal SP-graphs need not be simple, since the parallel combination allows to create a multiple edge. See Figure 8.2 for an example of a 2-terminal SP-graphs.

The name "2-terminal" for these graphs stems from the fact that they can be oriented in a special way. Recall that for a directed graph, a *source* is a vertex without incoming edges,

Figure 8.2: A 2-terminal SP-graph, and the graph $K_{1,3}$, which is not a 2-terminal SP-graph.

and a *target* or *sink* is a vertex without outgoing edges. A *terminal* is a vertex that is either a source or a target. Every directed acyclic graph must have a source and a target, and for 2-terminal SP-graphs, we can find a direction such that there is only one source and one target.

Lemma 8.2 *Let G be a 2-terminal SP-graph with terminals s and t. Then there exists an acyclic edge orientation of G such that s is the only source and t is the only target.*

Proof: We proceed by induction on the number of combination steps. In the base case, G is an edge (s, t), and we can direct it from s to t and satisfy the conditions. In the recursive case, G is obtained from subgraphs G_1 and G_2. By induction find the acyclic edge orientation for G_1 and G_2; this edge orientation then satisfies all conditions. □

One use of this lemma is to show that some graphs are not 2-terminal SP-graphs. For example, consider $K_{1,3}$ (see Figure 8.2.) This graph cannot be a 2-terminal SP-graph, because in any edge orientation the three leaves must be either sources or targets.

This example is somewhat surprising for two reasons. First, it shows that not even all trees are 2-terminal SP-graphs. Secondly, it is easy to construct a 2-terminal SP-graph that has $K_{1,3}$ as an induced subgraph (e.g., the left graph in Figure 8.2 does), so 2-terminal SP-graphs are not even closed under taking induced subgraphs.

8.2 The SP-tree

Let G be a 2-terminal SP-graph. We can capture the sequence of combinations used to construct G in a tree T (called the *SP-tree*) defined as follows:

- **Base Case:** If G is just an edge (s, t), then T is a single node containing edge (s, t).
- **Step:** If G is the combination of G_1 and G_2, in series or in parallel, then T has a node labeled S or P; this node contains G, and its children are the SP-trees for G_1 and G_2.

Traditionally, with every node in the SP-tree, we also denote which vertices are the terminals fore its subgraph, and for an S-node, also which vertex was the common terminal of the two children. See Figure 8.3.

8.3 Equivalent characterizations

In the long term, we want to relate 2-terminal SP-graphs and partial 2-trees. However, the example above of a tree that is not a 2-terminal SP-graph shows that they are not directly

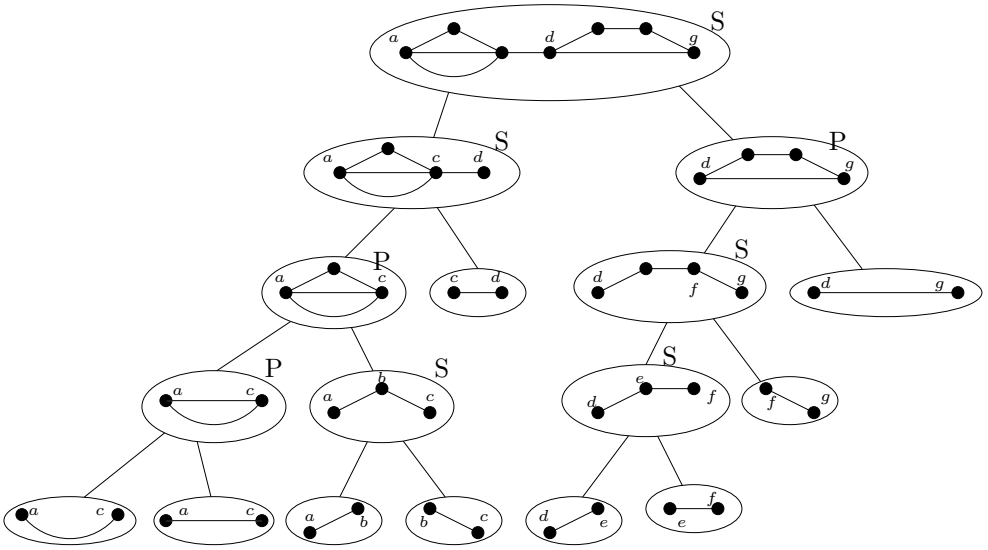


Figure 8.3: The SP-tree for the 2-terminal SP-graph in Figure 8.2.

related. We therefore first establish a relationship for those 2-terminal SP-graphs that are biconnected. A number of other equivalent characterizations can be proved along the way.

Theorem 8.3 *Let G be a biconnected graph. Then the following statements are equivalent:*

1. *G is a 2-terminal SP-graph.*
2. *G has treewidth at most 2.*
3. *G is a partial 2-tree.*
4. *G does not contain K_4 as a minor.*
5. *All triconnected components of G have at most 3 vertices.*[1]

Proof: We prove (1) $\Rightarrow$ (2) $\Rightarrow$ (3) $\Rightarrow$ (4) $\Rightarrow$ (5) $\Rightarrow$ (1) in a cyclical fashion.

- To see that (1)$\Rightarrow$(2), note that a tree decomposition of width 2 can be obtained directly from the SP-tree. Namely, label every node with the terminals of the graph it represents. For an S-node, also add the vertex which resulted from identifying the terminals of the children. See Figure 8.3, where the SP-tree has been labeled appropriately.

  Clearly every vertex and every edge is covered (already via the leaves.) The connectivity condition is also easily verified, since a vertex appears in two subtrees of a node a only if it was a terminal in both graphs of the children of a.

- (2)$\Rightarrow$(3) was proved in the previous chapter.

[1]For $n > 2$, this means that they are all triangles.

- (3)⇒(4) was also proved indirectly in the previous chapter. If K_4 were a minor of G, then K_4 would be a partial 2-tree, since partial 2-trees are closed under taking minors. But $\omega(.) \leq 3$ for any partial 2-tree, while $\omega(K_4) = 4$, so K_4 is not a partial 2-tree.
- (4)⇒(5) follows from Lemma B.6, which states that any triconnected graph with at least 4 vertices contains K_4 as a minor. So all triconnected components of G have at most 3 vertices. an edge or a triangle.
- (5)⇒(1) is proved by induction on the number of vertices. Since we actually need a slightly stronger statement, we phrase this as a separate lemma; see below.

□

For the missing directed (5)⇒(1), we need to briefly discuss triconnected components for graphs with double edges. While it may seem unintuitive, the proof below will work easiest if we define a multi edge to be a triconnected component as well. Thus, if (s,t) is an edge that occurs twice, then we consider $\{s,t\}$ to be a cutting pair, and replace the multi edge by one virtual edge; the removed multi edge (plus the virtual edge) is then one triconnected component. Also, in the lemma below, if (s,t) is a multiple edge, then $G-(s,t)$ denotes the graph from which one copy of (s,t) has been removed.

Lemma 8.4 *Let G be a biconnected graph (possibly with multiple edges) for which all triconnected components are triangles or multiple edges. Then for any edge (s,t) of G, the graph $G-(s,t)$ is a 2-terminal SP-graph with terminals s and t.*

Proof: We proceed by induction on the number of vertices. In the base case, $n = 2$, so G is triconnected, and hence must be a multiple edge (s,t). Therefore, $G-(s,t)$ is obtained by parallel combinations of edge (s,t), and the claim holds.

Now assume $n \geq 3$. We distinguish two cases. If $\{s,t\}$ is a cutting pair, then let $G_1, \ldots, G_k$ by the subgraphs defined by it. Each G_i contains (s,t) as a virtual edge and has fewer vertices, hence by induction $G_i-(s,t)$ is a 2-terminal SP-graph with terminals s and t. We can then obtain $G-(s,t)$ by repeated parallel combination of the G_i's. See Figure 8.4.

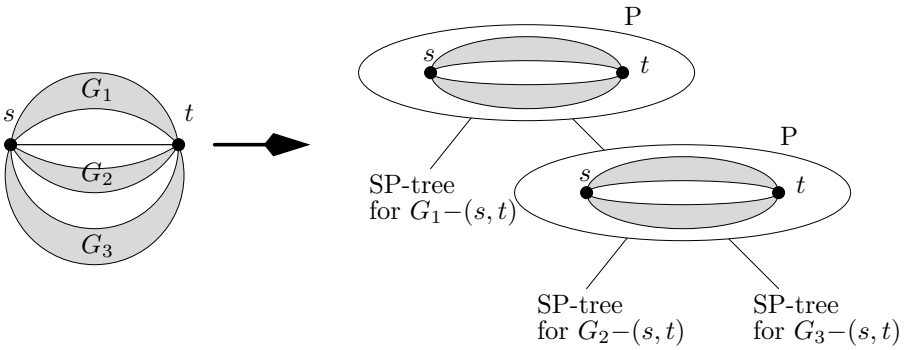


Figure 8.4: Parallel combinations if $\{s,t\}$ is a cutting pair.

Now assume that $\{s,t\}$ is not a cutting pair. By $n \geq 3$, the triconnected component that contains (s,t) then must be a triangle, say with third vertex x. We define two subgraphs. If

$\{s, x\}$ is a cutting pair, then let G_1 be the union of all subgraphs created by $\{s, x\}$ that do not contain t, and add to it the virtual edge (s, x). If $\{s, x\}$ is not a cutting pair, then (s, x) must be an edge by triconnectivity, and we set G_1 to be a double edge. Similarly define G_2 with respect to vertex pair $\{x, t\}$.

Note that (s, x) was added to G_1 as virtual edge, so by induction $G_1 - (s, x)$ is a 2-terminal SP-graph with terminals s and x. Likewise, $G_2 - (x, t)$ is a 2-terminal SP-graph with terminals x and t. We now obtain $G - (s, t)$ as a series combination of G_1 and G_2. See Figure 8.5. □

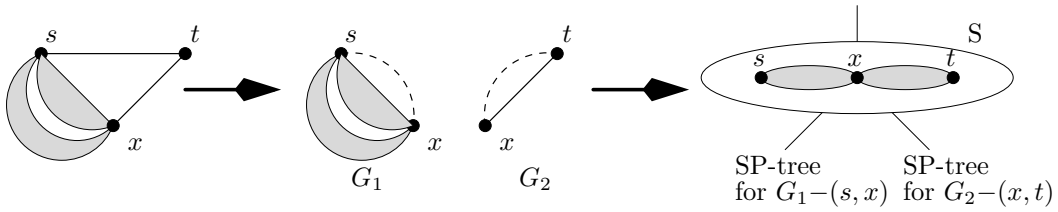


Figure 8.5: Series combinations if $\{s, t\}$ is not a cutting pair.

8.4 SP-graphs

Theorem 8.3 shows that 2-terminal SP-graphs are the same as partial 2-trees if they are biconnected. It is therefore worth studying graphs for which all biconnected components are 2-terminal SP-graphs.

Definition 8.5 *A graph G is called an* SP-graph *if every biconnected component of G is a 2-terminal SP-graph.*[2]

Note that with this definition, every tree is clearly an SP-graph, because every biconnected component of a tree is a single edge, which is a 2-terminal SP-graph.

Theorem 8.6 *A graph is an SP-graph if and only if it is a partial 2-tree.*

Proof: G is an SP-graph if and only if all biconnected components of G are 2-terminal SP-graphs. Each biconnected component of G is biconnected, so it is a 2-terminal SP-graph if and only if it is a partial 2-tree. It is easy to show that every graph is a partial k-tree if and only if all biconnected components are partial k-trees (this is left as an exercise), which proves the theorem. □

One implication of this theorem is a bound on the number of edges in an SP-graph. Since every k-tree has at most $kn - k^2/2 - k/2$ edges, an SP-graph has at most $2n - 3$ edges. Furthermore, the SP-tree of any 2-terminal SP-graph has at most $2n - 3$ leaves, and every internal node has two children, so the SP-tree has at most $4n - 5$ nodes.

[2]Warning! In the literature, often the distinction between SP-graphs and 2-terminal SP-graphs isn't done very clearly.

8.5 Recognizing SP-graphs

To test whether a given graph G is an SP-graph, we first compute all biconnected components of G in linear time. Then we test for each biconnected component whether it is a 2-terminal SP-graph.

So now let G be a biconnected graph, and we want to test whether G is a 2-terminal SP-graph. There are two approach; one is easy to explain (but hard to implement in linear time), and one is easier to implement.

First, recall from Theorem 8.3 that a biconnected graph is a 2-terminal SP-graph if and only if all triconnected components are triangles. So one easily-described algorithm is to compute all triconnected components and to test whether they are triangles. This takes linear time (theoretically), but is not easy to implement.

A second approach uses a different property of 2-terminal SP-graphs.

Lemma 8.7 *If G is a 2-terminal SP-graph with at least 2 edges, then G contains a multiple edge or a vertex of degree 2.*

Proof: Consider the lowest interior node in the SP-tree of G. If it is a P-node, then its two children form a double-edge. If it is an S-node, then its two children form a path of length 2, so the middle vertex in the path has degree 2. □

The algorithm to test for a 2-terminal SP-graph is now as follows:

Input: A biconnected graph G.

If G has one edge, then it is a 2-terminal SP-graph.
Else
  If G contains a vertex v of degree 2
    Contract v with one of its neighbours.
    Recursively test whether the resulting graph is a 2-terminal SP-graph.
  Else if G contains a multiple edge (v, w).
    Delete all but one copy of (v, w).
    Recursively test whether the resulting graph is a 2-terminal SP-graph.
  Else stop, G is not a 2-terminal SP-graph.

Lemma 8.8 *This algorithm correctly determines whether the graph is a 2-terminal SP-graph.*

Proof: We first show that if this algorithm stops prematurely, then indeed the graph is not a 2-terminal SP-graph. For if G is a 2-terminal SP-graph, then any subgraph G' created in the process is also a 2-terminal SP-graph. It is clear that G' is a minor of G, so it certainly has treewidth at most 2 if G does. Also, it is easy to show that both operations maintain biconnectivity, so G' is a biconnected graph of treewidth at most 2, which means by Theorem 8.3 that G' is indeed a 2-terminal SP-graph. Therefore, by Lemma 8.7, G' must have a vertex of degree 2 or a multiple edge.

Now if the algorithms runs its course until only one edge is left, then the graph is indeed a 2-terminal SP-graph. To see this, we show how to build the SP-tree by "undoing" the

operations. If G has only one edge, then clearly we can build an SP-tree for this. If G was obtained by replacing a vertex w of degree 2 by an edge (v, x) between its neighbours, then take the SP-tree for G, find the node for edge (v, x), and replace it by an S-node with two children for the edges (v, w) and (w, x). Likewise, if G was obtained by replacing a double edge between (v, w) by a single edge, then find the node of (v, w) in the SP-tree for G and replace it by a P-node with two children that both contain edge (v, w). Either way, this gives an SP-tree for the graph that G was obtained from, and by induction an SP-tree for the original graph. □

This algorithm can be made to work in linear time (the main difficulty here is identifying multiple edges in amortized constant time; we will not go into details of this.[3]) As a side note, we remark that by also allowing to remove vertices of degree 1 from the graph, the algorithm can be extended to test whether a graph is an SP-graph without having to compute biconnected components.

[3]I confess to not knowing them, but have heard the result quoted often enough to believe it.

Chapter 9

Algorithms for partial k-trees

In this chapter, we study algorithms for partial k-trees. In particular, we prove what was claimed earlier: Many graph problems are polynomial-time solvable for partial k-trees for which k is a constant. (We also call these graphs *graphs of bounded treewidth.*) The algorithm to do so is exponential in k, but linear in n.

9.1 Independent Set in 2-terminal SP-graphs

To "warm up", we will first study 2-terminal SP-graphs, which are partial 2-trees. We demonstrate the technique on the problem of Independent Set (IS), i.e., we want to find a maximal set V of vertices without an edge between them.

So let G be a 2-terminal SP-graph with terminals s and t. We define four functions:

$$\begin{aligned}
\alpha_{s,t}(G) &= \text{the size of the maximum IS in } G \text{ that contains both } s \text{ and } t \\
\alpha_{s}(G) &= \text{the size of the maximum IS in } G \text{ that contains } s, \text{ and does not contain } t \\
\alpha_{t}(G) &= \text{the size of the maximum IS in } G \text{ that contains } t, \text{ and does not contain } s \\
\alpha_{\emptyset}(G) &= \text{the size of the maximum IS in } G \text{ that contains neither } s \text{ nor } t
\end{aligned}$$

Clearly, the size of the maximum independent set in G is then

$$\alpha(G) = \max\{\alpha_{s,t}(G), \alpha_s(G), \alpha_t(G), \alpha_\emptyset(G)\},$$

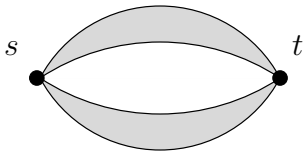
which we can compute in constant time once we know the other four numbers.

To compute the functions, we follow the recursive definition of G.

- In the **base case**, G is a single edge (s,t). The condition on the functions prescribe exactly what the independent set in G has to look like, and hence we know its size. A special case is the case $\alpha_{s,t}(G)$; since (s,t) is an edge, the set $\{s,t\}$ is not an independent set, so this value is undefined. We set this value to $-\infty$, which will therefore never be chosen when there is a better option, since we are maximizing the objective function. So we have

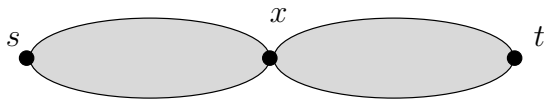
$$\begin{aligned}\alpha_\emptyset(G) &= 0\\ \alpha_s(G) &= 1\\ \alpha_t(G) &= 1\\ \alpha_{s,t}(G) &= -\infty\end{aligned}$$

- If G is a **parallel combination** of graphs G_1 and G_2, then we obtain the best independent set for G by combining the ones for G_1 and G_2 that satisfy the same conditions. However, we must be careful about double-counting. For example, in both $\alpha_s(G_1)$ and $\alpha_s(G_2)$ the vertex s contributes to the independent set. Thus to obtain the size of the independent set in G, we need to subtract one unit again to compensate for the double-counting of s. So we have:



$$\begin{aligned}\alpha_\emptyset(G) &= \alpha_\emptyset(G_1) + \alpha_\emptyset(G_2)\\ \alpha_s(G) &= \alpha_s(G_1) + \alpha_s(G_2) - 1\\ \alpha_t(G) &= \alpha_t(G_1) + \alpha_t(G_2) - 1\\ \alpha_{s,t}(G) &= \alpha_{s,t}(G_1) + \alpha_{s,t}(G_2) - 2\end{aligned}$$

- If G is a **series combination** of graphs G_1 and G_2, then the conditions on the independent set of G tell us only partially the condition for the independent sets of G_1 and G_2. In particular, if x is the common terminal of G_1 and G_2, then we don't know whether x is in the best independent set for G. Therefore, we try both possibilities, and use the one that yields the larger set. We also have to be careful about possibly double-counting x if it is to be in the independent set. So we have



$$\begin{aligned}\alpha_\emptyset(G) &= \max\{\alpha_\emptyset(G_1) + \alpha_\emptyset(G_2), \alpha_x(G_1) + \alpha_x(G_2) - 1\}\\ \alpha_s(G) &= \max\{\alpha_s(G_1) + \alpha_\emptyset(G_2), \alpha_{s,x}(G_1) + \alpha_x(G_2) - 1\}\\ \alpha_t(G) &= \max\{\alpha_\emptyset(G_1) + \alpha_t(G_2), \alpha_x(G_1) + \alpha_{x,t}(G_2) - 1\}\\ \alpha_{s,t}(G) &= \max\{\alpha_s(G_1) + \alpha_t(G_2), \alpha_{s,x}(G_1) + \alpha_{x,t}(G_2) - 1\}\end{aligned}$$

From the formulas, it is clear that we can compute $\alpha_{\dots}(G)$ from the values in its children in constant time. The time complexity is hence bound by the number of nodes in the SP-tree, which is linear.

9.2 Dynamic Programming in partial k-trees

Now we explain the dynamic programming approach for an arbitrary partial k-tree G. We again demonstrate it on the problem of finding the maximum independent set, though the same approach with different recursive functions works for many graphs problems.

9.2.1 Modifying the tree decomposition

As a first step, we create a tree decomposition of G in a special form.[1] Let G' be a supergraph of G that is a k-tree. Recall that G' (and hence also G) has a tree decomposition such that all labels have size $k+1$ and for all edges, the intersection of the labels has size k (Corollary 7.17).

Now for every edge (i, j) of the tree, subdivide the edge and let the label of the new node be $X_i \cap X_j$. Thus, the new node has a label of size k. Generally, now any edge connects a node whose label has size k with a node whose label has size $k+1$ and whose label is a superset of the other node.

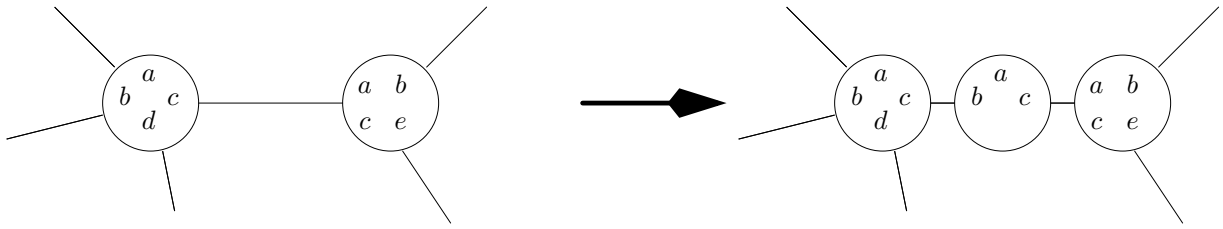


Figure 9.1: Subdividing an edge of the tree decomposition.

Pick an arbitrary node with a label of size $k+1$. Attach one new node to it and label it with some subset of size k of the label of its neighbour. Root the tree decomposition at this new node. Now we have the following properties:

- The root has a label of size k.
- Any node with a label of size k has exactly one child.
- For a node with a label of size k, the child has a label of size $k+1$ and is a superset.
- For a node with a label of size $k+1$, all children have a label of size k and are subsets.

9.2.2 Subgraphs of subtrees

Now that the tree decomposition is rooted, we can speak of the subgraph defined by the subtree rooted at node i. More precisely, for any node i, let Y_i be the set of all vertices that appear either in X_i or in X_j for some descendant j of i. Denote by $G[Y_i]$ the graph induced by the nodes in Y_i.

[1] This is actually not needed for the algorithm, but greatly simplifies the description of the formulas.

Note that if i is a node in the tree and j_1 and j_2 are two children, then Y_{j_1} and Y_{j_2} are disjoint except for vertices in X_i, i.e., $Y_{j_1} \cap Y_{j_2} \subseteq X_i$. In particular therefore, if an independent set in $G[Y_{j_1}]$ contains the same vertices of X_i as an independent set in $G[Y_{j_2}]$, then the union of the two independent sets is an independent set in $G[Y_i]$. This is the crucial idea of why the dynamic programming algorithm works.

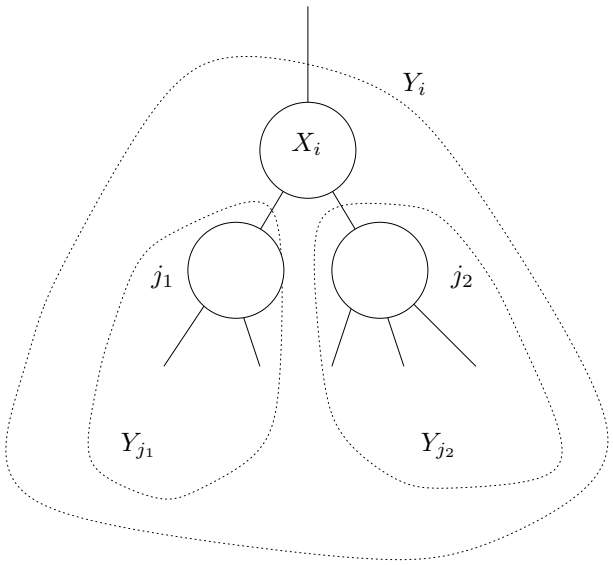


Figure 9.2: The set Y_i, Y_{j_1} and Y_{j_2}.

9.2.3 Recursive functions

Using this tree decomposition, we now define recursive functions, taking into account the configuration at the vertices that are in the label of each node. For any node i, and any subset $Z \subseteq X_i$, define

$$\alpha_i(Z) = \text{the size of a maximum independent set } I \text{ in } G[Y_i] \text{ that satisfies } I \cap X_i = Z$$

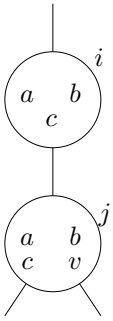
We develop recursive formulas for $\alpha_i(Z)$. In the **base case**, i is a leaf of the tree decomposition. Hence $Y_i = X_i$. We would like to know the size of the maximum independent set I in $G[Y_i]$ with $I \cap X_i = Z$. Since $X_i = Y_i$, there is only one choice: $I = Z$. If Z is not an independent set, then no suitable independent set I can exist. Therefore:

$$\alpha_i(Z) = \begin{cases} -\infty & \text{if } Z \text{ is not an independent set} \\ |Z| & \text{otherwise} \end{cases}$$

In the **recursive case**, node i has at least one child. We distinguish cases, depending on the size of the label of i and its number of children.

- Assume first that $|X_i| = k$, which implies that i has only one child, say j. By the conditions on the tree decomposition, $|X_j| = k + 1$ and $X_i \subset X_j$, so $X_i = X_j - \{v\}$ for some vertex v. Also, $Y_i = Y_j$, since X_i does not add any new vertices.

Consider the independent set I that attains $\alpha_i(Z)$. Then I is an independent in $G[Y_i]$ and hence also $G[Y_j]$. There are two cases depending on whether v is in I or not. If v is in I, then $|I| = \alpha_j(Z \cup \{v\})$, else $|I| = \alpha_j(Z)$. Therefore, we can compute $\alpha_i(Z)$ with the following formula:



$Z \subseteq \{a, b, c\}$

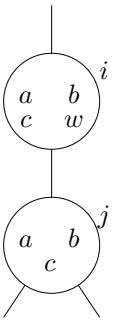
$$\alpha_i(Z) = \max\{\alpha_j(Z), \alpha_j(Z \cup \{v\})\}.$$

- Now assume that $|X_i| = k + 1$, and that i has only one child, say j. By the conditions on the tree decomposition, $|X_j| = k$ and $X_j \subset X_i$, so $X_j = X_i - \{w\}$ for some vertex w. Also, $Y_j = Y_i - \{w\}$.

  We want to compute $\alpha_i(Z)$ for some set $Z \subset X_i$. First, note that this is clearly $-\infty$ if Z is not an independent set.[2] If Z is an independent set, then we must distinguish further by whether w is contained in it, for w does not belong to the subgraph of the child.

  So assume first that $w \notin Z$. Let I be an independent set in $G[Y_i]$ that attains $\alpha_i(Z)$. Since $w \in X_i$ and $I \cap X_i = Z$, also $w \notin I$, so I is an independent set in $G[Y_j]$ as well, and hence is considered for $\alpha_j(Z)$. Also, the independent set that attains $\alpha_j(Z)$ is an independent set in $G[Y_i]$ and hence considered for $\alpha_i(Z)$, which proves $\alpha_i(Z) = \alpha_j(Z)$ if $w \notin Z$.

  If $w \in Z$, then we claim $\alpha_i(Z) = 1 + \alpha_j(Z - \{w\})$. To see this, note that if I attains $\alpha_i(Z)$, then $I - \{w\}$ is an independent set in $G[Y_j]$ whose intersection with X_j is exactly $Z - \{w\}$. Also, any such independent set can be extended to an independent set in $G[Y_i]$ by adding w to it (this is indeed an independent set since Z is an independent set). Putting it all together, we therefore have



$Z \subseteq \{a, b, c, w\}$

$$\alpha_i(Z) = \begin{cases} -\infty & \text{if } Z \text{ is not an IS} \\ \alpha_j(Z) & \text{if } Z \text{ is an IS and } w \notin Z \\ 1 + \alpha_j(Z - \{w\}) & \text{if } Z \text{ is an IS and } w \in Z \end{cases}$$

[2] We should have considered this special situation for the previous case as well, but this is not needed, since in the previous case the child will return $-\infty$ if Z is not an independent set.

For future reference, we rewrite this formula without using the specific name of vertex w, and instead expressing its membership in Z via $Z \cap X_j$.

$$\alpha_i(Z) = \begin{cases} -\infty & \text{if } Z \text{ is not an independent set} \\ \alpha_j(Z \cap X_j) + |Z| - |Z \cap X_j| & \text{otherwise} \end{cases}$$

- In the last case, $|X_i| = k+1$ and i has an arbitrary number of children, say $j_1, \ldots, j_\ell$.

  From the previous case, we know how the maximum independent set of $G[Y_i]$ relates to the one for each child. Now if we have two independent sets I_1 and I_2 for two children j_1 and j_2 such that $I_1 \cap X_i = Z = I_2 \cap X_i$, then their union is an independent set in Y_i, and its size is $|I_1| + |I_2| - |Z|$ (since vertices in Z are counted twice.) Generalizing this to an arbitrary number of children, we thus obtain the following formula:

$$\alpha_i(Z) = \begin{cases} -\infty & \text{if } Z \text{ is not an independent set} \\ \sum_{h=1}^{\ell} \left(\alpha_{j_h}(Z \cap X_{j_h}) - |Z \cap X_{j_h}|\right) + |Z| & \text{otherwise} \end{cases}$$

  Note that this formula contains as special cases both the formula for the previous case and the formula for the base case, so in an actual implementation only this formula and the formula for the case when $|X_i| = k$ is needed.

We note here without giving details that all formulas can be put together into one (very complicated looking) formula that is only applied to those nodes that have a label of size k. In particular therefore, we only need to store functions for 2^k subsets (rather than 2^{k+1} subsets as implied by our approach.)

The algorithm for compute the maximum independent set in a partial k-tree is now the same simple (?) linear-time algorithm as for trees: compute a post-order of the tree in the tree-decomposition, and for each node i, compute the values of $\alpha_i(Z)$ for all Z recursively from the values of the children. This takes $\deg(i)$ per subset Z, and hence $O(2^k \deg(i))$ per node i and $O(2^k n)$ overall.

Theorem 9.1 *We can find the maximum independent set in a partial k-tree in $O(2^k n)$ time.*

9.2.4 Fixed-parameter tractability

The above algorithm for independent set in partial k-trees is one example of a problem that is fixed-parameter tractable in some graphs. Fixed-parameter tractability is defined as follows: Assume you have a problem that has some parameters. One parameter is the input size of the instance (usually denoted n), but there is some other parameter as well (call it k.) We then say that the problem is *fixed-parameter tractable* if there exists an algorithm with run-time

$$O(f(k) \cdot p(k, n)),$$

where $p(k, n)$ is a polynomial in both k and n, whereas $f(k)$ can be an arbitrary function (exponential, super-exponential, it does not matter.) In particular, if a problem is fixed-parameter tractable, then it is polynomial as long as k is a constant.

There are two motivations why one should try to show fixed-parameter tractability for a problem. First, in practical settings k might be small. Second, showing fixed-parameter tractability shows what exactly makes the problem hard in general.

By the results in this section, we now know that Maximum Independent Set is fixed-parameter tractable in partial k-trees. The same result holds for a number of other graphs problems, for example vertex cover, dominating set, colouring, Hamiltonian Cycle, and Graph Isomorphism, to name just a few.

9.3 NP-hard problems in partial k-trees

However, not all graph problems are polynomial in partial k-trees. For starters, there are some graph problems which are not even polynomial in trees (there are such problems, for example the bandwidth problem and the Call Scheduling problem.) But even more interesting are problems that are solvable in trees, but not in partial k-trees for small k.

We give one example of such a problem here.

Definition 9.2 *Let $G = (V, E)$ be a graph with weights $w : E \to \mathbb{N}$ on the edges. A* weighted k-colouring *of G is an assignment of colours $c : V \to \{1, \ldots, k\}$ to the vertices such that*

$$|c(u) - c(v)| \geq w(e) \quad \text{for all edges } e = (v, w).$$

The Weighted-Colouring problem *is then to find the minimum number k such that G has a weighted k-colouring.*

The weighted colouring problem is trivially solvable in any bipartite graph, and in particular therefore in any tree. Let W be the maximum weight assigned to an edge, say it is assigned to edge (u, v). Clearly, we must have at least $W + 1$ colours in any weighted colouring, because if (say) $c(u) = 1$, then $c(v) \geq W + 1$.

But we can easily get a weighted $(W + 1)$-colouring, by assigning colour 1 to all vertices on one side of the bipartite graph, and colour $W + 1$ to all vertices on the other side of the bipartite graph. So this is an optimal weighted colouring.

Theorem 9.3 *The weighted colouring problem can be solved in linear time on trees.*

On the other hand, the weighted colouring problem is NP-hard for partial k-trees for $k \geq 3$ [MR01]. The complexity of this problem is open for partial 2-trees (i.e., SP-graphs), and this is an interesting topic for research.

9.4 Recognizing partial k-trees

Another question that we have not yet addressed here is how easy it is to recognize whether a given graph is actually a partial k-tree, and to find the tree decomposition if it is.

The answer to this depends on what exactly is meant by "recognize". Given a graph G and a constant k, if we want to know whether G has treewidth at most k, then there exists

an algorithm that is polynomial in n to test this. However, doing so is not pretty. There is an algorithm with time complexity $O(n^{k+2})$ [ACP87]. Also, it follows from Minor Theory that there exists some $O(n^2)$ algorithm for this problem (with a constant that is huge relative to k); we will return to this in Chapter 18. Neither algorithm is really practical. An algorithm that (apparently) is usable in practice was developed by Bodlaender and Kloks [BK96].

One might wonder whether there exists an algorithm that is polynomial in both k and n for testing treewidth. This is (likely) not the case, since the problem of finding the treewidth is NP-hard if k is part of the input [ACP87].

However, for many applications we don't really care to find the best possible k. All we care about is whether a graph has bounded treewidth, i.e., whether it is a partial k-tree for a fairly small k. This problem can be solved much more efficiently. Namely, there are algorithms that, for a given k, either prove that G does not have treewidth at most k, or give a tree decomposition of width at most $4k - 2$ [Ree92]. Further improvements to this result have been done, both with respect to improving the time complexity and with respect to reducing the amount by which the tree decomposition is "off". See [Bod97] for an overview.

Chapter 10

Friends of partial k-trees

Now we will study some classes that are related to partial k-trees, and in particular, define various other parameters for graphs similar to the treewidth. We will mostly restrict ourselves to defining what these parameters are, but omit the many results on how they relate to each other; see for example [Bod97].

10.1 Pathwidth

For a graph of treewidth k, we had a tree decomposition, i.e., a tree with labels of size $k+1$ that satisfied certain properties. We now study a graph class where we replace "tree" by "path". We thus define a graph G to have *pathwidth bounded by* k if G has a tree decomposition T of width k such that T is a path. (This is also called a *path decomposition of width* k. See Figure 10.1.

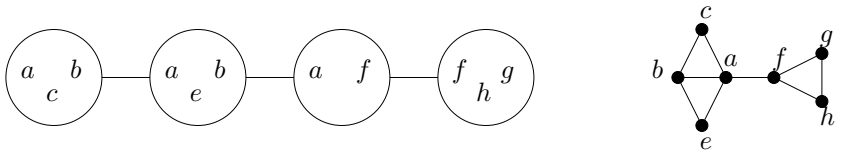


Figure 10.1: This graph (and any subgraph of it) has pathwidth at most 2.

Note that trees do not necessarily have pathwidth 1, see for example the tree in Figure 10.2. For assume this tree had a path decomposition of width 2. Then all nodes of the path have a label of size 2, which in particular means we must have a node for every edge. There are three nodes which contain c, and one of them must be between the other two in the path decomposition, say this is the one with label $\{c, d\}$. But then there is no place to attach the node labeled $\{d, e\}$ without either creating a node of degree 3 in the (supposed) path or violating the connectivity condition. So this tree does not have pathwidth 1 (but it can easily be shown to have pathwidth 2.)

Computing the pathwidth is NP-hard in general [ACP87], but for a given constant k, testing whether G has pathwidth $\leq k$ can be done in linear time [BK96].

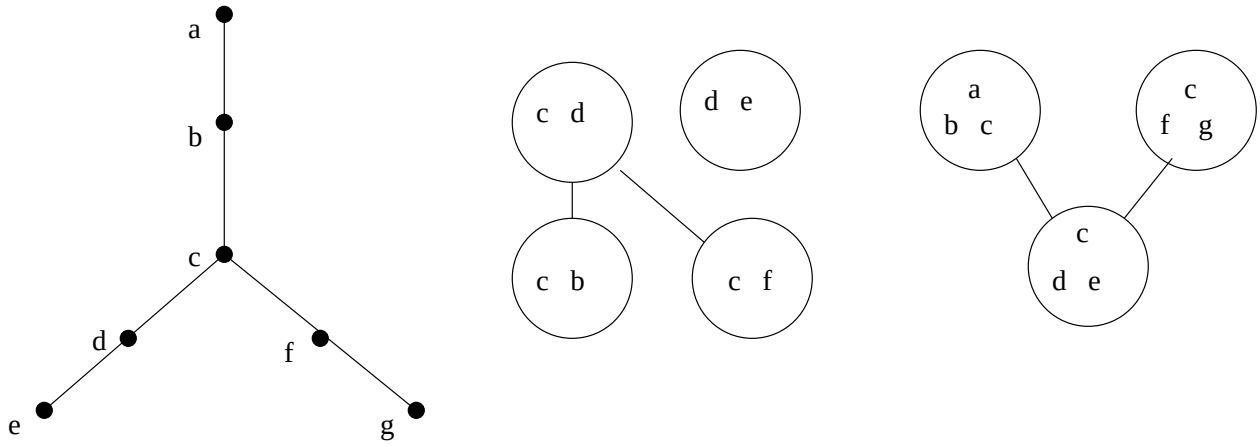


Figure 10.2: A tree with pathwidth 2.

10.2 Bandwidth

We define a graph G to have bandwidth at most k, if we can permute the vertices in such a way that all the entries in the resulting adjacency matrix are within k positions of the diagonal. See Figure 10.3.

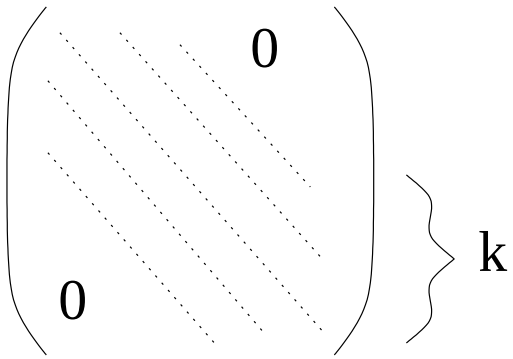


Figure 10.3: An adjacency matrix where all non-zero entries have horizontal and vertical distance at most k from the diagonal.

We can phrase this differently as follows: There exists an ordering of vertices $v_1, .., v_n$ such that for all $(v_i, v_j) \in E$ we have $|i - j| \leq k$.

Yet another way to think of this is to imagine the vertices to be placed on the real line with x-coordinates $1, 2, \ldots$. Then add all edges that span a distance of at most k. A graph has bandwidth at most k if and only if it is a subgraph of this graph. See Figure 10.4.

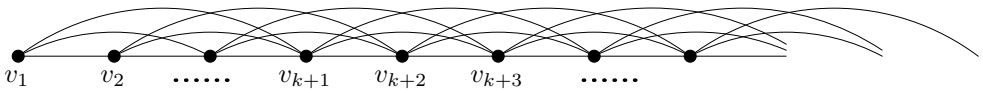


Figure 10.4: The graphs of bandwidth at most k are exactly the subgraphs of this graph.

Finally, we can relate bandwidth to pathwidth. We can create a path decomposition for the graph in Figure 10.4 by adding any $k+1$ consecutive vertices into the label of a node. See Figure 10.5. In particular therefore, any graph of bandwidth at most k also has pathwidth at most k. The reverse direction does *not* hold, for example $K_{1,3}$ has pathwidth 1, but bandwidth 2.

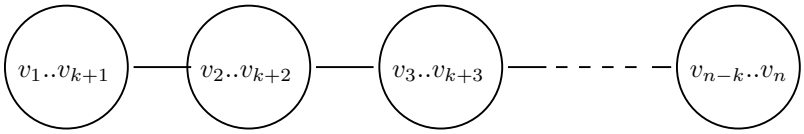


Figure 10.5: A path decomposition of the graph depicted in Figure 10.4.

Note that in the path decomposition above, in each next label on the path we always remove the vertex that has been in the label the longest. Put differently, all labels have size $k+1$ and every vertex belongs to at most $k+1$ labels. Graphs that have a path decomposition with this property are sometimes called *graphs of proper pathwidth at most* k, and it is relatively straightforward to show that these are exactly the graph of bandwidth at most k. It is easy to show that the graphs of proper pathwidth at most 1 are exactly those graphs that are a union of paths.

10.3 Domino width

We just saw that proper pathwidth can be described by putting a limit on the number of labels in which a vertex is allowed to appear. This idea has been applied to other descriptions as well. For example, a graph of *domino width* at most k is a graph that has a tree decomposition of width k such that every vertex appears in at most two nodes of the tree decomposition.

The domino treewidth can be found in polynomial time for trees [BE97], but the same paper also shows that is NP-hard to compute in general.

10.4 Cutwidth

Recall that one way to define graphs of bandwidth at most k was via ordering the vertices on a line and then putting restrictions on the edges within this ordering. A related concept are graphs of bounded cutwidth. A graph is said to have *cutwidth* at most k if its vertices can be ordered as $v_1, \ldots, v_n$ such that every cut in this order crosses at most k edges. More precisely, for any $1 \leq j < n$, there can be at most k edges (v_i, v_l) such that $i \leq j < l$. See Figure 10.6.

Computing the cutwidth is NP-hard in general, but testing whether a give graph has cutwidth at most k can be done in time linear in n (but exponential in k) [TSB00].

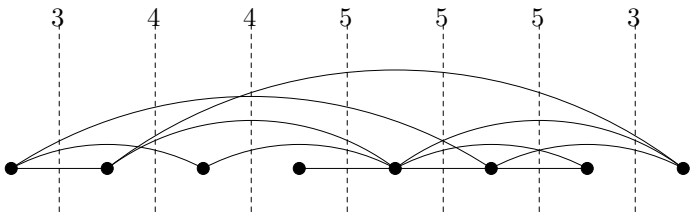


Figure 10.6: A graph with cutwidth at most 5. The lines show the cuts considered; we also indicate the number of edges in each cut.

10.5 Branchwidth

Tree decompositions were defined by labeling nodes of a tree with vertices. One more graph parameter, the branchwidth, is defined quite differently: the leaves are labeled with edges, and the edges of the tree (not the nodes) receive labels.

A *branch decomposition* of a graph G is a binary tree B such that every leaf of B is labeled with an edge of G and every edge of B is labeled with a set of vertices of G such that the following holds:

- For every edge e in G, there is exactly one leaf of G that is labeled with e.
- An edge e of B contains v in its label if and only if v appears on both sides of e, i.e., in a leaf of both subtrees that result from deleting e.

The width of such a decomposition is the size of the largest edge-label in B, and the branchwidth of G is the minimum width of a branch decomposition of G. See Figure 10.7 for an example.

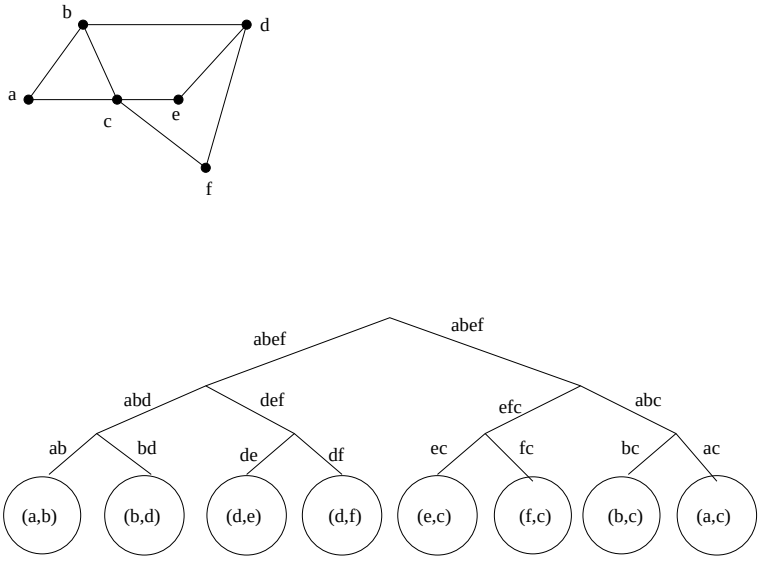


Figure 10.7: A graph G and a branch decomposition of G of width 4.

Computing the branchwidth is NP-hard in general, but testing whether a give graph has branchwidth at most k can be done in time linear in n [BT97].

Part III

Planar Graphs And Friends

Chapter 11

Planar Graphs

This chapter introduces planar graphs and some of their properties.

11.1 Definitions

A graph is *planar* if it can be drawn in the plane so that no two edges cross. Not all graphs are planar, for example, we will show soon that K_5 and $K_{3,3}$ are not planar.

Planar graphs arise naturally in many applications. For example, in networks, and in any other application that derives from the way objects are placed and how they interact with each other, the geometry of the objects is often such that the resulting graph is planar (or has only very few crossings.) For this reason, we will study planar graphs extensively.

11.1.1 Combinatorial embeddings

The definition of a planar graph, while easy to understand in theory, is hard to capture for algorithmic purposes. In the current definition, we allow drawings with arbitrary curves for edges. How would we store this in a computer, i.e., represent it in a discrete way? One idea would be to store vertices as points and declare edges as straight lines, but to be allowed to do this, we would first need to prove that all planar graphs have such a straight-line drawing. (They do, but this is not at all an easy result – see Chapter 16.)

Instead, we will represent planar graphs in a very different way, via what is called a *combinatorial embedding*. This concept actually exists for all graphs, whether planar or not, but is equivalent to planar graphs in a special case.

Definition 11.1 *Let G be a graph. A* combinatorial embedding *of G is a set of orderings π_v for each vertex $v \in V$, where π_v specifies a cyclic ordering of edges incident to v.*

If we are given a drawing of a graph (with crossings or without), then this always implies a combinatorial embedding, by taking the clockwise order of the edges incident to each vertex. On the other hand, if we are given a combinatorial embedding, then it is easy to create a drawing (with crossings, possibly) such that the combinatorial embedding exactly corresponds to the clockwise order of edges at each vertex. Figure 11.1 shows an example of a combinatorial embedding.

a: $\{e_1\}$
b: $\{e_1, e_2, e_3, e_2, e_5, e_7\}$
c: $\{e_3, e_4\}$
d: $\{e_4, e_6, e_5\}$
e: $\{e_6, e_7\}$

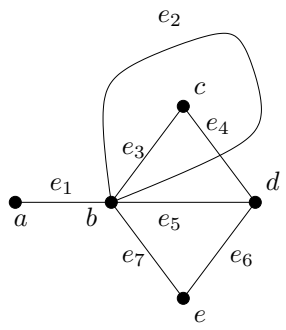


Figure 11.1: A combinatorial embedding of a graph, and a drawing that respects it.

Given a combinatorial embedding, we can define for each edge a *next edge* by following the combinatorial embedding. To make this precise, we will for a little while replace each edge (v, w) by two directed edges $v \to w$ and $w \to v$, and replace the entry of (v, w) in the list of v by $v \to w$, followed by $w \to v$.[1] Then for edge $v \to w$, the *next edge after* $v \to w$ is the edge $w \to u$ that immediately follows $v \to w$ in the cyclic order of edges around vertex w.

Using the notion of the next edge, we can define a *face* as the equivalence classes of edges that can reach each other via the next operation. More precisely, this is defined as follows. Start at an arbitrary edge $v_1 \to v_2$. Let $v_2 \to v_3$ be the next edge after $v_1 \to v_2$. Iterate, i.e. for $i > 0$, let $v_i \to v_{i+1}$ be the next edge after $v_{i-1} \to v_i$. Continue this process until at some point we repeat an edge (which must happen since the graph is finite.) This repeated edge actually must be $v_1 \to v_2$, since every edge has only one edge for which it is the next. The resulting circuit

$$v_1 \to v_2 \to \cdots \to v_k \to v_1$$

is called a *facial circuit*, or *face*. The number k is the *degree* of the face, and is also denoted $\deg(F)$ for face F. Figure 11.2 illustrates one face of the combinatorial embedding of the graph in Figure 11.1.

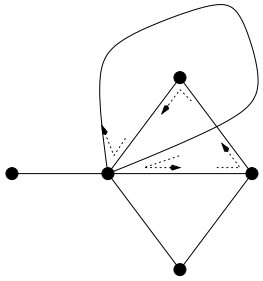


Figure 11.2: A face of the graph in Figure 11.1.

A graph with a fixed combinatorial embedding thus naturally splits into a number of faces; each direction of each edge belongs to exactly one face (though the two directions might belong to the same face twice.) In particular therefore, $\sum_{F \text{ face}} \deg(F) = 2m$.

[1]Some data structures, for example LEDA [LED], actually store planar drawings of graphs that way. We will only do this for the purpose of defining faces and then forget about it again.

Note that none of these definitions need an actual drawing of the graph; they only rely on the combinatorial embedding. However, there is a deep connection between the number of vertices, edges and faces and the type of drawing that have this combinatorial embedding.

Theorem 11.2 *Let G be a graph with a combinatorial embedding ϕ, and let f be the number of faces of this combinatorial embedding. Let g be such that*

$$n - m + f = 2 - 2g,$$

i.e., $g = \frac{1}{2}\{2 - n + m - f\}$. Then there exists a drawing of G on a surface of genus g that has no crossing between edges.

For a proof of this theorem, see for example [MT01]. The other direction of this theorem also holds (and is much easier to prove), i.e., if G has a crossing-free drawing on a surface of genus g, then $n - m + f = 2 - 2g$. We will return to this for the case $g = 0$ (i.e., drawings in the plane) shortly.

11.1.2 Combinatorial embeddings of planar graphs

While the above definitions and theorem holds for arbitrary combinatorial embeddings, the concepts are easier to visualize for a planar graph. Assume for a moment that someone gives us a drawing of a planar graph G that has no crossing. Then use as combinatorial embedding the one induced by this drawing, i.e., order the edges as they appear in clockwise order around each vertex.

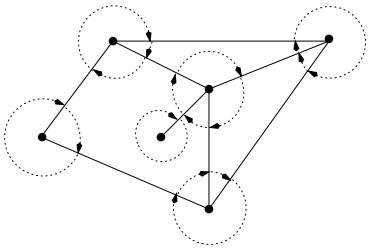


Figure 11.3: The combinatorial embedding defined by a planar drawing.

We will call a combinatorial embedding a *planar embedding* if it corresponds to a crossing-free drawing in the plane, i.e., if its genus (computed as $\frac{1}{2}(2 - n + m - f)$) is 0.

Assume we are given a planar embedding of a connected graph, and fix an arbitrary planar drawing that respects the planar embedding. It is then easy to verify that the *faces* exactly correspond to the connected pieces of the plane after the drawing is removed. More precisely, draw the graph on a piece of paper. Now cut the paper along every edge of the graph. The resulting pieces of paper each constitute one face. In this approach, there is one special face, which is the one that includes the boundary of the paper (or in general, includes infinity). This face is called the *outer-face*. The outer-face depends on the specific drawing, and is not defined from the combinatorial embedding alone.

A *plane graph* G is a planar graph with a *fixed* combinatorial embedding. In particular, by Theorem 11.2, this means that if we are given a plane graph, then we may assume that

we have a drawing of this graph G in the plane such that the combinatorial embedding is respected. (Actually finding such a drawing is a topic on its own; we will return to this in Section 16.)

Figure 11.4 provides an example of different planar drawings of the same graph. Here, the leftmost drawing is truly different from the middle drawing: the leftmost drawing has a face of degree 6 (the outer-face), while the middle drawing has no such face.

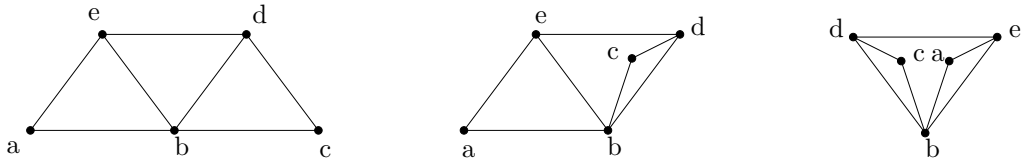


Figure 11.4: Different planar drawings of the same planar graph.

The difference between these two drawings results from having "flipped" the subgraph at the cutting pair $\{a, c\}$. One can show that such flips at cutting vertices or cutting pairs are the only way to modify a planar embedding, and all possible planar embeddings can be stored (implicitly) via the blocktree and the SPQR-tree of triconnected components. See [BT96] for details. In particular, if a planar graph is triconnected, then the planar embedding is unique.

Theorem 11.3 (Whitney) *If G is a 3-connected planar graph, then G has a unique planar embedding.*[2]

Proof: Let Φ_1 and Φ_2 be two planar embeddings of G. Let C be a facial cycle in Φ_1. Since G is 3-connected, $G - C$ must be connected. (This follows from Menger's theorem: for any two vertices v and w not in C, there must be three disjoint paths from v to w, and only two of those can intersect C by planarity.) See Figure 11.5.

Now consider the location in C in Φ_2. Since $G - C$ is connected, there cannot be vertices both inside and outside C. Thus, C is again a face in Φ_2, except if C has a chord (which in particular implies that C has length at least 4.) But C cannot have a chord either, since such a chord would give a cutting pair in embedding Φ_1. □

Now let us return to the planar graph with different planar drawings in Figure 11.4. Close inspection reveals that the leftmost drawing is not so different from the rightmost drawing, because the planar embedding is the same in both, and only the outer-face has changed. It turns out that for any planar drawing, we can always change the outer-face without changing the planar embedding.

Lemma 11.4 *Let G be a connected planar graph with a planar drawing Γ, and let F be one of its faces. Then there exists a planar drawing of G with the same planar embedding as Γ such that F is the outer-face.*

[2]By "unique", we mean that a face in one embedding is also a face in the other embedding. This means that the graph has two planar embeddings, since we can always obtain a different one by "flipping" the picture.

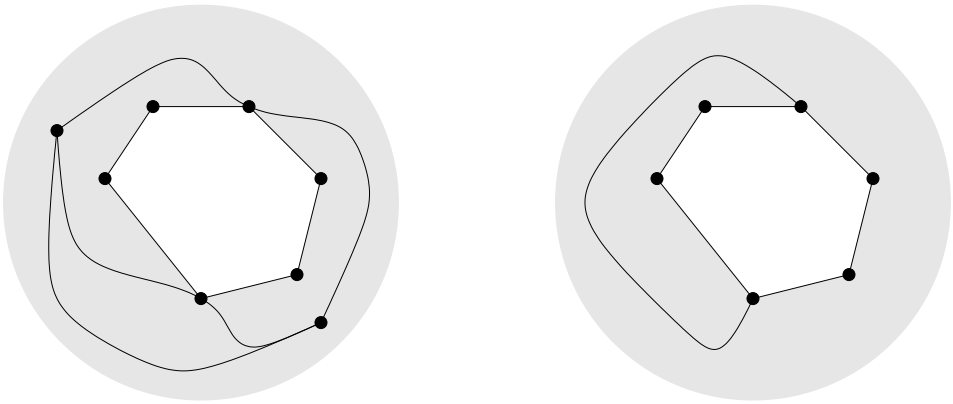


Figure 11.5: If C is a facial cycle of a 3-connected graph, then $G - C$ is connected. Also, C cannot have a chord.

Proof: A terse proof would be to say the following: map the drawing onto a sphere, and then map the sphere back into the plane in such a way that a point inside F becomes infinity. The resulting drawing has the same planar embedding and F is the outer-face.

In case this was too terse, here is a long and more algorithmically suited proof. Let p be any point inside face F in Γ, and let r be a ray emanating from p that does not cross any vertex. (Since there are only finitely many vertices, there exists such a ray.)

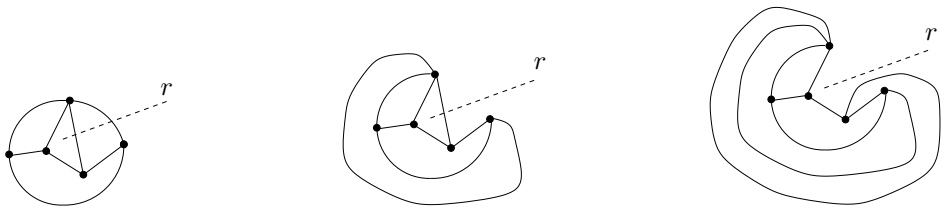


Figure 11.6: Making a face the outer-face.

The proof is now by induction on the number of edges that r crosses. If it crosses none, then F is already the infinite face, i.e., the outer-face, and we are done. If ray r crosses some edges, then it must also cross an edge $e = (u, w)$ on the outer-face of the current drawing. Reroute the edge (u, w) such that it does not cross ray r (see Figure 11.6). In this way we get planar drawing Γ' in which ray r crosses fewer edges, and are done by induction. □

11.1.3 Data structures for planar graphs

From now on, we will often assume that we are given a planar graph and a fixed planar embedding, i.e., a plane graph. (Finding this is a different matter; we will return to this in Chapter 15.) In particular, the planar embedding gives an ordering of the edges for each vertex, and we assume that the data structure to store the graph is the usual adjacency lists, but the entries in each adjacency lists have been sorted as to reflect the planar embedding.

Whenever needed, we will also assume that we are given the outer-face (which, together with the planar embedding, then uniquely identifies a drawing of the planar graphs, up to

deformations of the plane.) This can be done by specifying one directed edge; the outer-face is then the facial cycle defined by this directed edge.

11.2 Dual graphs

For every graph with a fixed planar embedding, we can define a dual graph that captures (in essence) the planar embedding in a graph structure. More formally, if G is a connected graph with a fixed planar embedding, the *dual graph* G^* of a graph G is built by defining a vertex v_F for every face F of G, and then adding an edge $e^* = (v_F, v_{F'})$ to G^* for every edge e in G that is incident to the two faces F and F'.[3] Edge e^* is called the *dual edge* of edge e.

Note that the definition of dual graph does not depend on planarity, and can be extended to any graph with a fixed combinatorial embedding, though we will use it only for planar graphs. We can visualize it as follows: Each face corresponds to a region in the plane. Place a vertex inside the region, and connect two vertices if and only if the two regions have a common boundary. (If they have more than one common boundary, connect them repeatedly.) It follows trivially that for a planar graph the dual graph G^* is planar, and that the clockwise order of the dual edges around vertex v_F is the same as the clockwise order of the corresponding edges around face F.

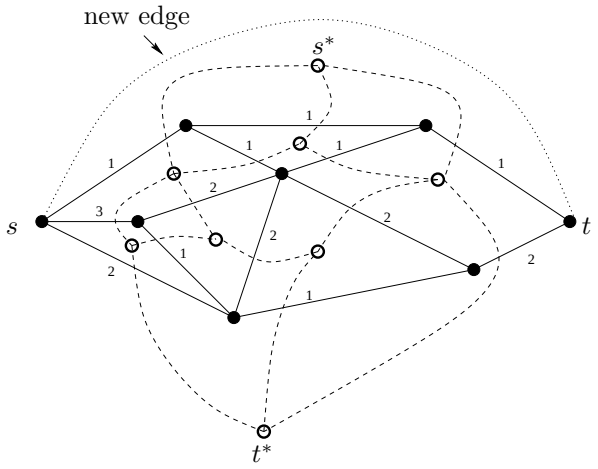


Figure 11.7: A planar graph G and its dual graph G^*. Vertices of the dual graph are stars; dual edges are dashed and drawn such that they intersect their corresponding edge in the original graph.

Even if G is simple, the dual graph need not be simple, see the example in Figure 11.7.

The same planar graph may have different dual graphs, depending on the embedding. Consider for example Figure 11.4: In the left drawing, the dual graph has a vertex of degree 6, which is not the case in the middle drawing. However, if two drawings of a planar graph have the same planar embedding, then the dual graph is the same.

[3]More precisely, if $e = (v, w)$, then $v \to w$ is incident to F and $w \to v$ is incident to F'. But we will from now on "forget" about the edge directions imposed .

The dual graph has a vertex for every face of the original graph. But what becomes of the vertices of the original graph? If v was a vertex, with incident edges $e_0 = (v, w_0), \ldots, e_{k-1} = (v, w_{k-1})$ in clockwise order around it, then we now have dual edges $e_0^*, \ldots, e_{k-1}^*$. For any $i = 0, \ldots, k-1$, the dual edges e_i^* and e_{i+1}^* have a common endpoint, namely, the vertex v_{F_i} that corresponds to the face bounded by e_i and e_{i+1} (all additions modulo k). Also, e_i^* and e_{i+1}^* are consecutive at v_{F_i}. It follows that $e_0^*, \ldots, e_{k-1}^*$ form a face. Thus, the dual graph has a face F_v for every vertex v in the original graph. Using this observation, it is very easy to show:

Lemma 11.5 *If G is a connected plane graph, then the dual of the dual graph of G is again G, or more precisely, $(G^*)^* = G$.*

The appropriate definition of what the dual graph is for a graph that is not connected is somewhat unclear. The first approach is to let G^* be the union of the dual graphs of each of the connected components of G. This has a minor flaw: the outer-face is represented by as many vertices as there are connected components, and thus we cannot really talk of "the outer-face" anymore. The second approach is to take the union, as in the first approach, but then to merge all the vertices representing the outer-faces into one vertex. Now the outer-face is represented by one vertex, but it no longer holds that $(G^*)^* = G$, because G^* and therefore $(G^*)^*$ is connected while G is not. So neither solution is perfect.[4]

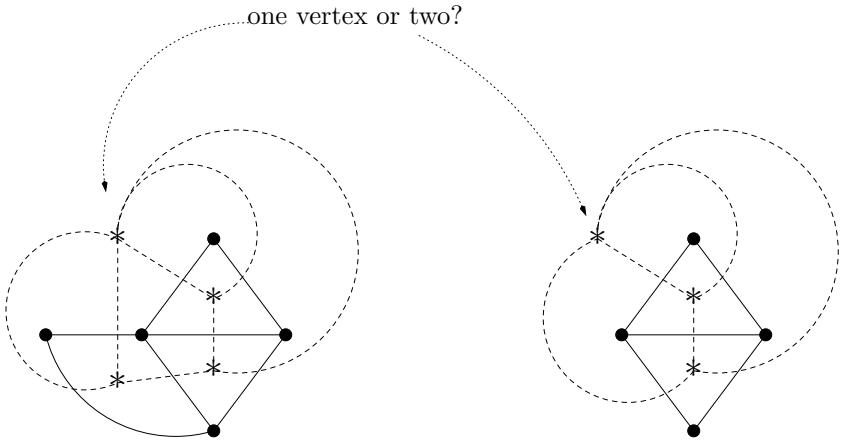


Figure 11.8: How to define a dual graph for a disconnected graph?

11.2.1 Data structures for dual graphs

Given a fixed planar embedding, the dual graph can be computed in $O(m + n)$ time. The idea to do so is to follow the definition of facial cycles. We define a dual edge for every edge, and then, following each facial cycle, combine the endpoints of dual edges into one vertex for each face. Details are omitted.

[4]The first solution is generally the more accepted one, since it corresponds to the formal definition of faces. But this really depends on the application, so read papers carefully.

It is often convenient to store both the original graph and the dual graph in one data structure. We then have a list of vertices, a list of faces, and a list of edges. Each vertex has a list of incident edges (ordered by the planar embedding), each face has a list of incident edges (ordered by the order in the facial cycle that defined the face), and each edge refers to both entries in both lists (i.e., four entries total.) This data structure can also be computed in $O(m+n)$ time, presuming that we have the planar embedding of the original graph.

11.3 Closure properties

Recall that we had some standard graph operations, such as adding or deleting vertices or edges. What happens if we perform an operation in a planar graph G? Is the resulting graph planar?

- If we delete a vertex or an edge from G to obtain G', then simply deleting the drawing of the vertex/edge in a planar drawing of G will give a planar drawing of G'. Thus, any subgraph of a planar graph is again planar. We will call this drawing of G' the *planar drawing of G' induced by the planar drawing of G.*
- If we add an edge $e = (v, w)$ to G to obtain G', then G' may or may not be planar. But, if we know that v and w are on one face, then G' is planar, because we can route the new edge by going through the face.
- If we contract v and w in G to obtain G', then G' may or may not be planar. But, if v and w are on one face, then G' is planar.
- If we contract edge (v, w) in G to obtain G', then G is planar, because v and w are on one face.

In particular, note that if G is a planar graph, then any minor of G is also a planar graph.

Another interesting question is what happens to the dual graph G^* when we perform an operation in the graph G (referred to in the following as the *primal graph*).

Lemma 11.6 *An edge deletion in the primal graph corresponds to an edge contraction in the dual graph, as long as all graphs are connected. More precisely, if G is a connected graph and e is an edge in G that is not a bridge, then $(G - e)^* = G^* \backslash e^*$.*

Proof: When we delete e, the two faces F, F' incident to e become one face. (These were different faces because e is not a bridge.) Thus in the dual graph, vertices v_F and $v_{F'}$ become one vertex, which is a contraction. See Figure 11.9 for an example. □

11.4 Euler's formula

Recall that Theorem 11.2 states that if $n - m + f = 2 - 2g$, then G has a drawing on a surface of genus g. The reverse direction also holds, and we prove it here, though only for genus $g = 0$, i.e., for planar graphs.

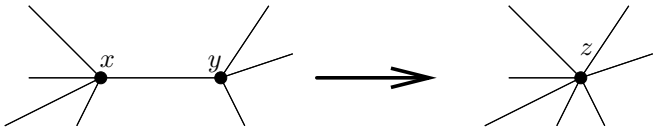


Figure 11.9: An edge deletion in the primal graph corresponds to an edge contraction in the dual graph.

Theorem 11.7 (Euler's formula) *Let G be a connected, not necessarily simple, plane graph. Then $n - m + f = 2$ where f is the number of faces.*

Proof: We will use induction on the number of faces. In the base case there is only one face. This implies that G contains no cycle, because each cycle divides the plane into an inner and an outer part which cannot belong to the same face. Graph G is therefore a forest, and because it is connected, it is a tree. A tree with n vertices has exactly $n - 1$ edges, so $n - m + f = n - (n - 1) + 1 = 2$.

Now assume that the graph has at least 2 faces and that the theorem holds for all graphs with fewer faces. Because there are at least two faces, there must be distinct faces that have a common boundary, say edge e is incident to two different faces F and F'. See Figure 11.10.

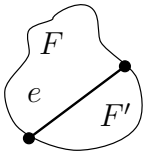


Figure 11.10: There must be an edge e incident to two different faces.

Let G' be the graph obtained by deleting the edge e from graph G. Graph G' is still connected because an edge of a planar graph is a bridge if and only if it is incident to only one face. Moreover G' has only $f - 1$ faces because faces F and F' became one face. We have obtained a graph with $n' = n$ vertices, $m' = m - 1$ edges and $f' = f - 1$ faces. Using the induction hypothesis for G' we get $2 = n' - m' + f' = n - (m - 1) + (f - 1) = n - m + f$, which proves the claim. □

A number of useful results follow from Euler's formula. We start with a simple observation.

Corollary 11.8 *Let G be a planar graph. Then any planar drawing of G has the same number of faces.*

Proof: The number of faces is $m - n + 2$ by Euler's formula. Since m and n are both independent of the planar drawing, the result follows. □

Another corollary of Euler's formula is quite important: any simple planar graph has $O(n)$ edges.

Lemma 11.9 *Every simple connected planar graph with at least 3 vertices has at most* $3n-6$ *edges.*

Proof: We will prove this statement by double-counting the edge-face incidences in some planar embedding of the graph. We will make a list of faces of a graph in one column and a list of edges in the other column. Then we draw a line from an edge to a face if they are incident. We draw two lines between one pair if the edge is incident to this face twice. See Figure 11.11.

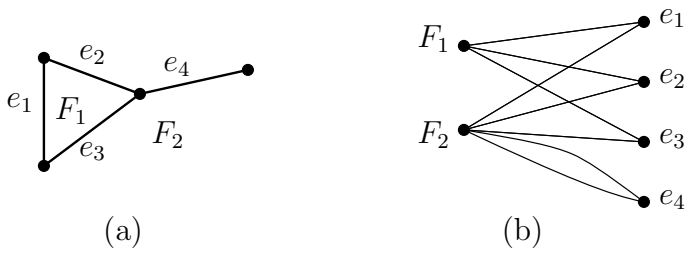


Figure 11.11: The method of double-counting.

Now let us count the number L of lines twice. We know that each edge is incident to two faces (not necessarily different) and thus there are two lines leading from each edge. Therefore $L = 2m$.

We can prove that each face is incident to at least three edges. If there were a face incident to only one edge, then it would be enclosed in this edge, i.e., the edge would be a loop, which contradicts that the graph is simple. If there were a face incident to only one edge twice, this edge would be a connected component by itself, which contradicts that the graph has at least 3 vertices and is connected. If a face were incident to exactly two different edges, then these edges would form a cycle, i.e., they are multiple edges, which contradicts that the graph is simple.

So each face is incident to at least three edges, therefore at least three lines leave from each face and $L \geq 3f$. We have counted the number of lines L twice and if we put the results together we get $3f \leq 2m$.

To finish the proof we multiply Euler's formula by 3 and plug the inequality $3f \leq 2m$ into it to obtain $6 = 3n - 3m + 3f \leq 3n - 3m + 2m = 3n - m$. □

Observe that the bound of at most $3n-6$ edges does not hold for graphs with one or two vertices. On the other hand, the bound of at most $3n-3$ edges holds for any simple planar graph, even if it is not connected, which one can show easily by induction on the number of connected components. Thus, every planar simple graph has $O(n)$ edges. On the other hand, no such bound holds for planar graphs that are not simple: we could have 2 vertices, and arbitrarily many edges between them.

Corollary 11.10 *Every simple planar graph has a vertex of degree at most 5.*

Proof: Any graph with $n \leq 2$ vertices has a vertex of degree at most 1 by simplicity. If $n \geq 3$, then by Lemma 11.9 we have $6n-12 \geq 2m = \sum_{v \in V} \deg(v)$. If each vertex in a graph

had degree higher than 5, then the sum of vertex degrees would be at least $6n$ which is a contradiction. □

We can similar, but even stronger, results for planar bipartite graphs.

Lemma 11.11 *Any simple planar bipartite graph with at least 3 vertices has at most* $2n-4$ *edges.*

Proof: The proof is almost identical to the proof of Lemma 11.9, except for the following observation: If a graph G is bipartite, then all cycles in G have even length, which means in particular that G has no triangle. Therefore, every face must be incident to at least four edges. Using again double-counting, we obtain $4f \leq 2m$, which in conjunction with Euler's formula yields the result. □

Lemmas 11.9 and 11.11 can be used to show that some graphs are not planar.

Corollary 11.12 *K_5 and $K_{3,3}$ are not planar graphs.*

Proof: Recall that K_5 is a complete graph with 5 vertices and $K_{3,3}$ is a complete bipartite graph with 3 vertices in each partition.

K_5 has $\binom{5}{2} = 10$ edges. According to Lemma 11.9, a planar graph with 5 vertices can have at most $3 \cdot 5 - 6 = 9$ edges, so K_5 is not planar.

$K_{3,3}$ has $3 \cdot 3 = 9$ edges. According to Lemma 11.11 a planar bipartite graph with 6 vertices can have at most $2 \cdot 6 - 4 = 8$ edges, so $K_{3,3}$ is not planar. □

Since planar graphs are closed under taking minors, any graph that contains K_5 or $K_{3,3}$ as a minor therefore is also not planar. Kuratowski's famous theorem states that these are the only graphs that are not planar. We will not prove this theorem here, see for example [Gib85].

Theorem 11.13 (Kuratowski) *Graph G is not planar if and only if it contains K_5 or $K_{3,3}$ as a minor.*

Chapter 12

Problems in planar graphs

In this chapter, we study the computational complexity of problems in planar graphs.

12.1 NP-hard problems on planar graphs

We first give some problems that remain NP-hard even in planar graphs. There are quite a few of these, and we only give a small selection here.

NP-hardness proofs for a problem P in planar graphs come in two possible flavours:

1. Develop a "crossing gadget" and use it to show that some problem P reduces to problem P in planar graphs.

   A crossing gadget here is some small subgraph by which we can replace every crossing in a (non-planar) drawing of a graph without changing the solution to the problem. We will illustrate this for 3-colouring below.

2. Look at the original NP-hardness proof for P, say it was done by reducing some NP-hard problem Q to P. If Q is also NP-hard for planar graphs, and the reduction from Q to P preserves planarity, then P is also NP-hard for planar graphs. We will see this for Independent Set below.

12.1.1 Coloring planar graphs

Recall that a colouring of a graph G with k colours is an assignment of labels (or colours) in $\{1, \ldots, k\}$ to the vertices such that for every edge the two endpoints have different colours. The k-colouring problem is the problem where we are given a graph and we want to know whether it has a colouring with k colours.

So we will now show that 3-colouring is NP-hard, even in planar graphs. To do this, we use the crossing-gadget shown in Figure 12.1. It has four special vertices, labeled a, a', b and b'.

Lemma 12.1 *In any 3-colouring of the gadget, $c(a) = c(a')$ and $c(b) = c(b')$, where $c(v)$ denotes the colour assigned to vertex v. Moreover, there exists a 3-colouring of the gadget where $c(a) = c(b)$, and there exists a 3-colouring of the gadget where $c(a) \neq c(b)$.*

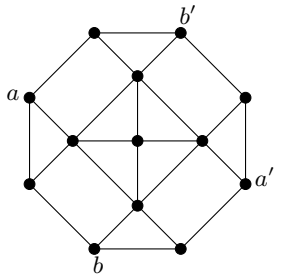


Figure 12.1: The crossing-gadget for 3-colouring.

Proof: After possible renaming of colours, we may assume that the central vertex is coloured with colour 1. Its neighbours must be coloured alternatingly with 2 and 3, and hence after possible renaming of colours we may assume that the five vertices not on the outer-face are coloured as illustrated in Figure 12.1.

Now there are two cases. If a is coloured with 1, then going counter-clockwise around the outer-face, the colours of all remaining vertices is forced, and we get the left colouring in Figure 12.1. If a is coloured with 2, then going clockwise around the outer-face, the colours of all remaining vertices is forced, and we get the right colouring in Figure 12.1. a cannot be coloured with 3 since it has a neighbour coloured 3. So the two colourings in Figure 12.1 are the only two possible colourings (up to renumbering of colours), and one quickly verifies the claims of the lemma. □

Now we are ready for the formal proof that 3-colouring is NP-hard for planar graphs. The proof is by reduction from 3-colouring in general graphs. So let G be an instance of general 3-colouring; G is not necessarily planar. Create a simple drawing (with crossings) of G by drawing the vertices on a horizontal line in an arbitrary order, and drawing edges as partial circles.[1]

Now create a new graph G_1 as follows. Consider the crossing s with the smallest x-coordinates of all crossings. Say the crossing edges are edges (a, c) and (b, d). Since all edges are drawn monotonically increasing in x-direction, and since we picked the leftmost crossing, there can be no more crossings between s and the left endpoint of each of the edges; assume that these endpoints are a and b, respectively. We obtain G_1 by inserting the crossing gadget at s, identifying the two vertices labeled a, identifying the two vertices labeled b, and changes edges (a, c) and (b, d) to become (a', c) and (b', d) instead. See Figure 12.2 for an example.

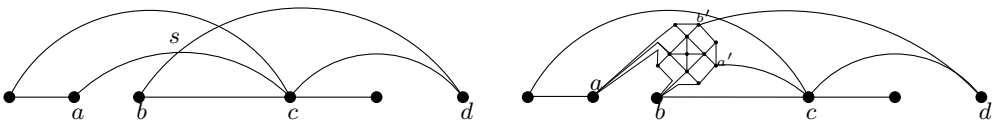


Figure 12.2: Replacing a crossing by the crossing gadget.

[1]The type of drawing here is really quite irrelevant; the only thing that matters is that we guarantee that there always is a crossing "at" two endpoints of the crossing edges and that there are a polynomial number of crossings.

Claim 12.2 *G is 3-colourable if and only if G_1 is 3-colourable.*

Proof: Assume first that we have a 3-colouring of G. To obtain a 3-colouring of G_1, set $c(a') = c(a)$ and $c(b') = c(b)$, and 3-colour the rest of the gadget appropriately (we know that this is feasible regardless of whether $c(a) = c(b)$ or not).

For the other direction, assume we have a 3-colouring of G_1. Then we must have $c(a) = c(a')$ and $c(b) = c(b')$. By simply taking the same colouring for G, we obtain a 3-colouring for G. For the only edges where there might possibly be a violation of the colouring rules are (a, c) and (b, c), but edges (a', c) had differently coloured endpoints in G_1, and $c(a) = c(a')$, so (a, c) has differently coloured endpoints in G. Similarly (b, d) has differently coloured endpoints in G, so this is indeed a legal 3-colouring of G. □

Now we are almost done. Note that G_1 has one fewer crossing that G and all edges that are involved in crossings are still drawn x-monotone. By repeating the argument k times, where k is the number of crossings in the original drawing of G, we obtain a graph G_k that is planar and that has a 3-colouring if and only if G has one. Also, since there were at most $m^2 \in O(n^4)$ crossings, the size of G_k is polynomial in the size of G. Therefore, there is a planar graph that is 3-colourable if and only if G is 3-colourable, and 3-colouring reduces polynomially to 3-colouring in planar graphs. Therefore 3-colouring is NP-hard even in planar graphs.

12.1.2 Planar 3-SAT

Now we look at the 'other' way of proving a problem NP-hard in planar graph, i.e., by inspecting the reduction and showing that it preserves planarity. The most commonly used problem for proving NP-hardness (used for example for Independent Set, Vertex Cover and Hamiltonian Cycle) is 3-SAT. Recall that 3-SAT is the problem where we are given n boolean variables $x_1, \ldots, x_n$ and m clauses $c_1, \ldots, c_m$, where each clause consists of at most three literals. We want to know whether there is an assignment of values TRUE and FALSE to the variables such that all clauses are satisfied.

This problem is not a graph problem, so how can we talk about the reduction preserving planarity? One can define a graph out of an instance of 3-SAT as follows:[2]

- Create one vertex for every literal x_i and $\overline{x_i}$.
- Create one vertex for every clause c_j.
- Create an edge (ℓ_i, c_j) if and only if clause c_j contains literal ℓ_i.
- Create an edge $(x_i, \overline{x_i})$ for every variable.
- Create a cycle $x_1 - x_2 - x_3 - \cdots - x_n - x_1$ of edges.

[2]There are various definitions of the graph defined by 3-SAT in the literature. For example, in some papers there is only one vertex per variable, and it represents both x_i and $\overline{x_i}$. In other variants, the last type of edges is not required. We use here the most general variant possible; since this already implies NP-hardness if the corresponding graph is planar, then for the other variants the problem is NP-hard as well.

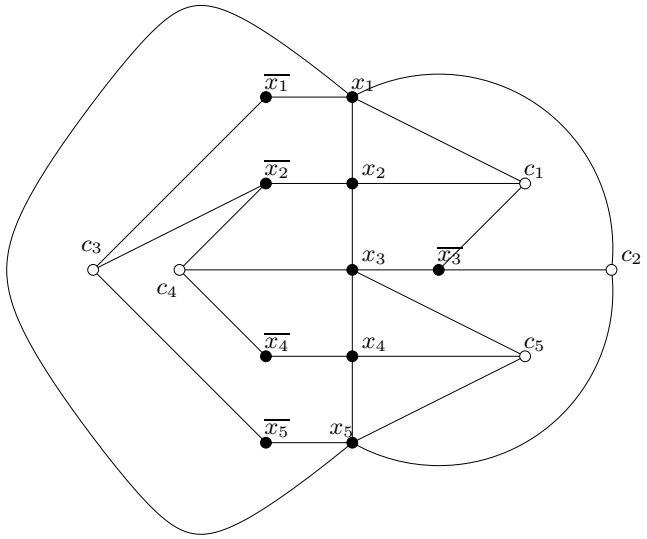


Figure 12.3: An instance of Planar 3-SAT.

See Figure 12.3 for an example. An instance of the problem of Planar 3-SAT is an instance of 3-SAT for which (after suitable renumbering of the variables, if needed) the associated graph is planar. With an idea similar as to the one for 3-colouring (again using a crossing gadget), one can show the following:

Theorem 12.3 *3-SAT reduces to Planar 3-SAT, and in particular Planar 3-SAT is NP-hard.*

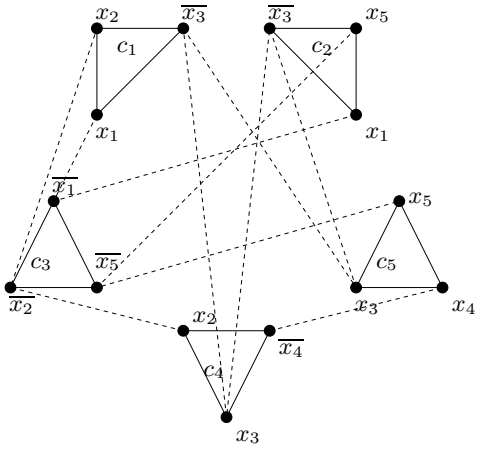
We will not even give the crossing gadget here; it is relatively complicated to draw and even more complicated to explain its correctness. See [Lic82] for details of the gadget and the formal proof of this theorem.

12.1.3 Independent Set

From the NP-hardness of Planar 3-SAT, it is now straightforward to prove that VertexCover, Independent Set and Hamiltonian Cycle are all NP-hard in planar graphs, by inspecting the original reduction from 3-SAT to the problem and showing that it preserves planarity (or can be easily modified to do so.) We illustrate this for Independent Set here. We first review the original reduction form 3-SAT to Independent Set.

Theorem 12.4 *3-SAT reduces polynomially to Independent Set.*

Proof: Let an instance of 3-SAT with n variables and m clauses be given. Define a graph as follows: For every clause c_j, define a *clause-triangle* consisting of three vertices that form a triangle. Label the three vertices with the three literals in c_j. Then for any variable x_i, add all edges between a vertex labeled x_i and a vertex labeled $\overline{x_i}$, i.e., add the complete bipartite graph between the set of vertices labeled x_i and the set of vertices labeled $\overline{x_i}$. See Figure 12.4. It is clear that the size of G is polynomial in n and m.



$c_1 = x_1 \vee x_2 \vee \overline{x_3}$
$c_2 = x_1 \vee \overline{x_3} \vee x_5$
$c_3 = \overline{x_1} \vee \overline{x_2} \vee \overline{x_5}$
$c_4 = x_2 \vee x_3 \vee \overline{x_4}$
$c_5 = x_3 \vee x_4 \vee x_5$

Figure 12.4: The reduction from 3-SAT to Independent Set. Edges for the clause-triangles are solid, edges between vertices labeled in the opposite way are dashed.

Observe that any independent set I in G contains at most one vertex per clause triangle. Since every vertex belongs to a clause-triangle, therefore $|I| \leq m$. Also, I cannot contain both a vertex marked x_i and a vertex marked $\overline{x_i}$. Using these two observations, it follows that the 3-SAT instance is satisfiable if and only if G contains an independent set of size m, by letting the TRUE literals be the ones for which the vertices are in such an independent set. □

The graph constructed in this proof is not planar, even for an instance of planar 3-SAT. The principle obstacles are the edges in the complete bipartite graph between vertices labeled x_i and $\overline{x_i}$, since those may well form a $K_{3,3}$. But by replacing this part of a graph with a slightly different construction, we can achieve planarity.

Theorem 12.5 *Planar 3-SAT reduces polynomially to Independent Set in planar graphs.*

Proof: Let an instance of 3-SAT with n variables and m clauses be given. Define a graph as follows: As before, create a clause-triangle for every clause. For every variable x_i, define two adjacent vertices t_i and f_i. Connect t_i to all vertices labeled $\overline{x_i}$, and f_i to all vertices labeled x_i. See Figure 12.5 for the difference between the two reductions. Clearly the resulting graph is polynomial in size.

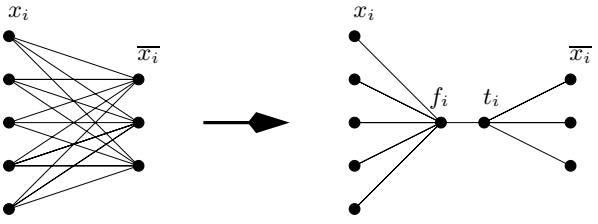


Figure 12.5: Instead of a complete bipartite graph, we add two more vertices per variable.

Note that any independent set I in G contains at most one vertex per clause-triangle and at most one vertex per edge (t_i, f_i). Therefore, $|I| \leq m+n$. Also, if $t_i \in I$, then no vertex $\overline{x_i}$

can be in I, and if $f_i \in I$, then no vertex labeled x_i can be in I. Using these two observations it follows that the 3-SAT instance is satisfiable if and only if G contains an independent set of size $m + n$, by setting x_i to be true iff t_i is in the independent set.

We claim that if the graph G of the 3-SAT instance is planar, then the resulting graph H is also planar. To see this, note that a super-graph of H can be obtained from G as follows:

- Subdivide all edges from a clause-vertex c_j to a vertex of a literal.
- For each clause-vertex c_j, add a cycle among its three neighbours.

See Figure 12.6. Neither of these operations can destroy planarity, so the super-graph of H (and hence also H) is planar if G is. □

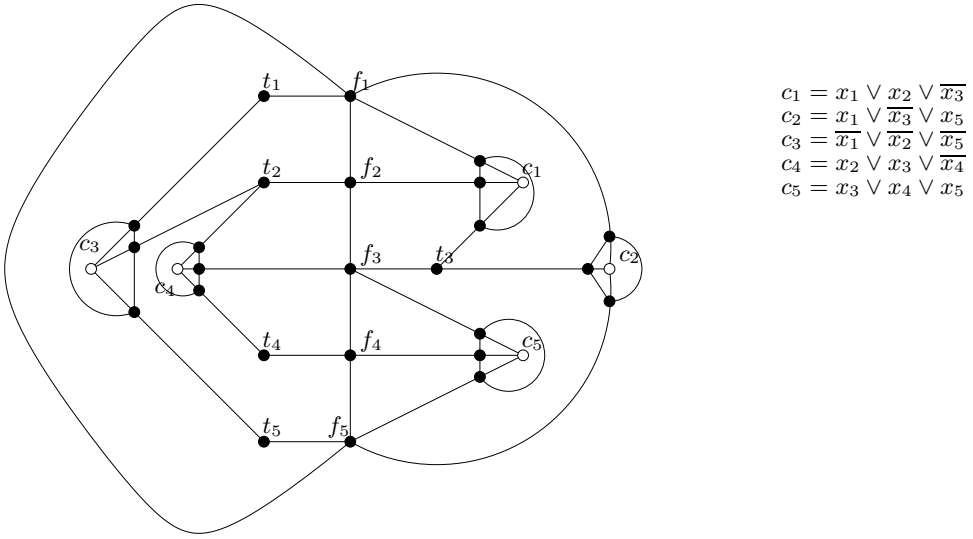


Figure 12.6: Converting the planar graph of 3-SAT into a planar graph for Independent Set.

Note that any independent set I in G contains at most one vertex per

12.2 Problems that are polynomial in planar graphs

Some problems that are ordinarily NP-hard become polynomial-time solvable for planar graphs, which is what we will study now.

12.2.1 Clique

Recall that Clique is the problem of finding the largest set of vertices that form a clique, i.e., have all possible edges between them. In general, this problem is NP-hard. However, for planar graphs it is easily solvable in polynomial time. Recall that K_5 is not planar, so no planar graph can have a clique of size 5 or larger. Finding the maximum clique is therefore a simple matter of testing all subsets of size 4 or less for whether they are a clique, and returning the largest set that is. This can be done in $O(n^4)$ time since there are only

$O(n^4)$ such sets. (In fact, with a more sophisticated approach this can be done in linear time [PY81].)

12.2.2 Coloring once more

The k-colouring problem behaves strangely in planar graphs. For $k = 2$, the problem is polynomial (not only in planar graphs, but in all graphs, since this amounts to testing bipartiteness.) For $k = 3$, we saw that the problem is NP-hard even in planar graphs. However, for $k = 4$ the problem is solvable in planar graphs (and the answer is always "yes"!) The reason is a famous theorem:

Theorem 12.6 (Four-Colour Theorem) *Every planar graph has a vertex-colouring with at most 4 colours.*

Whole books have been devoted to the (very lengthy and computer-aided) proof of this theorem; see [AH77, AHK77] for the original proof, [Aig84] for a good overview, and [RSST97] for a 're-proof' of the theorem. We will obviously not prove this theorem here, but we will briefly touch on the 6-colour and 5-colour theorem.

It is quite easy to see that every planar graph has a 6-colouring. First, multiple edges and loops do not change the existence of a colouring, so we can remove those and only study simple planar graphs. Now, every simple planar graph has a vertex of degree at most 5. By computing a minimum-degree order, we can hence obtain an ordering such that $\text{outdeg}(v) \leq 5$ for all vertices. By running the greedy algorithm for colouring with the reverse of this ordering, we obtain a colouring with $\max\{\text{outdeg}(v) + 1\} \leq 6$ colours. In fact, this algorithm can easily be implemented in linear time.

It is not too hard to see that in fact 5 colours suffice. Whenever in the above approach we have a vertex that has $\text{outdeg}(v) = 5$, we need to modify the greedy-algorithm slightly to contract two of its non-adjacent neighbors, and we will then be able to use only 5 colours throughout. See [CNS81] for more on this argument, as well as a linear time algorithm to find a 5-colouring of a planar graph.

Chapter 13

Maximum Cut

In this chapter, we study another problem that is NP-hard in general, but becomes polynomial in planar graphs: Maximum Cut.

A *cut* of a graph $G = (V, E)$ is a partition of the vertices V into C and $\overline{C} = V - C$. A *cut-edge* is an edge (v, w) with $v \in C$ and $w \in \overline{C}$ or $v \in \overline{C}$ and $w \in C$; the number of cut-edges is called the *size* of the cut. A *minimum/maximum cut* is a cut with minimum/maximum size among all non-trivial cuts (i.e., cuts where both C and $\overline{C}$ are non-empty.)

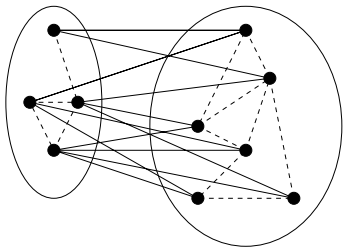


Figure 13.1: Example of a cut. Cut-edges are solid, other edges are dashed.

The minimum cut of any graph can be found in polynomial time with maximum-flow algorithms, see for example [AMO93]. This makes is somewhat surprising that the Maximum Cut problem (which is to find the maximum cut) is NP-hard, which we prove first. Then we show that Maximum Cut becomes polynomial in planar graphs.

13.1 NP-hardness of Maximum Cut

We first show that Maximum Cut is NP-hard. The reduction is from 3-SAT, except that we use a slight variant of 3-SAT which is known as Not-all-Equal 3-SAT (or NAE-3-SAT), and can easily shown to be NP-hard as well [GJ79].

Definition 13.1 *An instance of NAE-3-SAT is a set X of variables and a collection C of clauses over X such that each clause has three literals. Is there a boolean assignment for X such that each clause in C has at least one true literal and at least one false literal?*

Thus, NAE-3-SAT is the same as 3-SAT, except that we do not allow all three literals in a clause to be false.

Theorem 13.2 *Maximum Cut is NP-hard.*

Proof: We show that NAE-3-SAT reduces to Maximum Cut. Given an instance of NAE-3-SAT we construct a graph G as follows (see also Figure 13.2). We add an edge $(x_i, \overline{x_i})$ for every variable x_i. For every clause c_j, we connect the three literals of this clause in a triangle. Note that this creates a multiple edge if the same pair of literals is in more than one clause.[1]

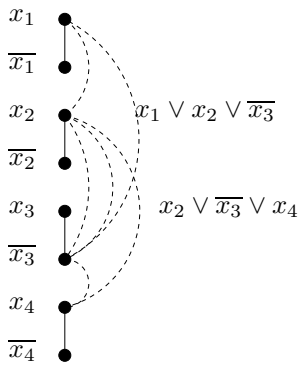


Figure 13.2: The graph constructed for reducing NAE-3-SAT to Maximum Cut.

Call the resulting graph G. Note that G has $n + 3m$ edges. Also note that any cut can have at most 2 out of the 3 edges of a triangle as cut-edge, therefore any cut in G can have at most $n+2m$ cut-edges. We claim that the instance of NAE-3-SAT is satisfiable if and only if G has a cut with $n+2m$ cut-edges. For assume G has such a cut $(C, \overline{C})$. Then necessarily all edges $(x_i, \overline{x_i})$ must be in the cut. Also, for any clause-triangle, at least one literal must be in C and at least one literal must be in $\overline{C}$. So we get a solution to the instance of NAE-3-SAT by setting x_i to be TRUE if and only if x_i belongs to C; this sets at least one literal and at most two literals per clause to be TRUE.

The other direction is nearly the same: given a solution of the NAE-3-SAT instance, we let C be all those literals which are TRUE. Since every clause is satisfied, two edges in each clause-triangle are in the cut. Since only one of x_i and $\overline{x_i}$ is TRUE, also all edge $(x_i, \overline{x_i})$ are in the cut, so the cut has size $2m + n$ as desired. □

13.2 Maximum Cut in Planar Graphs

Amazingly so, a polynomial time algorithm exists for Maximum Cut in planar graphs. We will present a scheme due to Hadlock [Had75]. The algorithm works by performing a series of transformations to cast the maximum cut problem into a series of equivalent problems. After the final change we are required to find a minimum-weight perfect matching. The maximum

[1] Multiple edges can be avoided with a modified construction.

cut is recovered by reversing the previous transformations. Since each step requires at most polynomial time, the entire algorithm has polynomial complexity.

13.2.1 Minimum odd circuit cover

Recall that a bipartite graph G is formed from two disjoint vertex sets, V_1 and V_2, such that every edge of G joins a vertex of V_1 to a vertex of V_2. We observe that given a cut $(C, \overline{C})$ with cut-edges E_c, the graph (V, E_c) is bipartite. In other words, we can make a graph G bipartite by deleting the edges contained in $E - E_c$. With this observation we can cast the maximum cut problem in another form:

Transformation 1 *Finding a maximum cut in G is equivalent to finding a minimum cardinality edge-set whose removal makes G bipartite.*

An edge set E' whose removal makes G bipartite will also be called an *odd circuit cover*, since for every odd cycle, at least one edge must be in E'. [An *odd cycle cover* may be a better name, but circuit appears in Hadlock's original paper.]

13.2.2 Minimum odd vertex cover

Up to this point all statements have been valid for general graphs. But for planar graphs, we can reformulate the problem using duality. We need an observation.

Lemma 13.3 *Let G be a plane graph. Then G is bipartite if and only if G^* is Eulerian, i.e., all vertices in G^* have even degree.*

Proof: If G is bipartite, then all cycles have even length, and in particular, all facial circuits have even length and therefore even degree. Every vertex in the dual graph corresponds to a face in G and therefore also has even degree, so G^* is Eulerian.

The other direction is not quite as trivial. Assume that G^* is Eulerian, and let C be a cycle in G. We claim that C must have even length, which shows that G is bipartite. Let $F_1, \ldots, F_k$ be the faces that are inside C, and let E_c be the edges of G that are strictly inside C. ("Inside" is defined relative to a fixed planar drawing of G; it does not matter which one we take.) Then $\sum \deg(F_i)$ counts every edge on C once and every edge in E_c twice, therefore

$$\sum_{i=1}^{k} \deg(F_i) = |C| + 2|E_c|.$$

But every face has even degree (since G^* is Eulerian) and $2|E_c|$ is even as well, which proves that $|C|$ is even. □

Recall that deleting an edge in a primal graph G contracts an edge in the dual graph G^*. The only exception occurs when we delete a bridge of G, but we would never include a bridge in a minimum odd vertex cover, since it does not belong to any cycle. Hence, making G bipartite by deleting edges is the same as making G^* Eulerian by contracting edges.

Remark: Hadlock originally said to delete the edge instead of contract. This mistake was later fixed by Aoshima and Iri [AI77].

Transformation 2 *Finding an odd circuit cover is equivalent to finding a minimum set of edges in G^* such that contracting the edges makes the graph Eulerian.*

We call a set E' of edges an *odd vertex cover* in G^* if contracting the edges of E' makes the graph Eulerian. The name comes from the term *odd vertex* for a vertex of odd degree, which we will use occasionally in the following.

13.2.3 Minimum odd vertex pairing

A crucial observation is now that an odd vertex cover can be characterized.

Lemma 13.4 *A minimum odd vertex cover P has the form $P = P_1 \cup \cdots \cup P_k$, where for $i = 1, \ldots, k$, P_i is a path of edges connecting two odd vertices in graph G. Moreover, the $P_i's$ are edge-disjoint, and no P_i and P_j have the same endpoint.*

The proof of this lemma is omitted for now (and will hopefully be provided later.)
So from the lemma, we can obtain yet another reformulation of the problem.

Transformation 3 *Finding an odd vertex cover is equivalent to finding a pairing $\{v_1, w_1\}$, $\ldots, \{v_k, w_k\}$ of all odd vertices and a set of edge-disjoint paths $P_1, \ldots, P_k$, where P_i connects v_i and w_i such that $\sum |P_i|$ is minimized.*

We will call such a set an *odd vertex pairing.*

13.2.4 Finding a minimum odd-vertex pairing

Clearly, if we have the pairing $\{v_1, w_1\}, \ldots, \{v_k, w_k\}$ of all odd vertices, then it would makes sense to let P_i be the shortest path from v_i to w_i. The set of these shortest paths is not necessarily edge-disjoint, but one can show that if $\{v_1, w_1\}, \ldots, \{v_k, w_k\}$ is the *minimum* odd-vertex pairing, then the shortest paths between them must be edge-disjoint.

Lemma 13.5 *Assume $\{v_1, w_1\}, \ldots, \{v_k, w_k\}$ is a minimum odd vertex pairing. Let P_i' be the shortest path from v_i to w_i. Then $P_1', \ldots, P_k'$ are edge-disjoint.*

Proof: Assume that P_1' and P_2' have a common edge (x, y), as shown in Figure 13.3. Let P' and P'' be the two shorter paths that do not use (x, y), (from, say w_1 to w_2 and v_1 to v_2), they are shown with dashed lines. Then the combined length of P' and P'' is less than combined length of P_1' and P_2', because edge (x, y) is not contained in the first set of paths. Thus pairing w_1 with w_2 and v_1 with v_2 and using P' and P'' would give an odd vertex pairing with fewer edges contradicting the minimality. □

This lemma shows actually that for any odd vertex pairing with paths $P_1, \ldots, P_k$, the paths must have been a system of shortest paths, because if they were not, then by using shortest paths we still get an odd vertex pairing, and it has fewer edges.

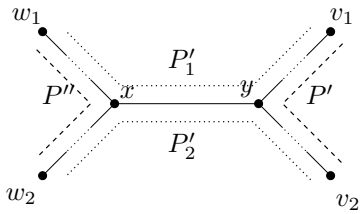


Figure 13.3: If P_1' and P_i are not edge-disjoint, then we can find a smaller odd vertex pairing.

13.2.5 Finding a minimum-weight matching

Finding a shortest path between a given pair of vertices is easy. Thus, in order to find a minimum odd vertex pairing, we need to consider all pairings $\{v_i, w_i\}$ and chose the one for which the total lengths of the shortest paths is minimized. This turns out to be not too hard either, by using matching techniques.

Recall that a *perfect matching* is a subset M of the edges such that no two edges in M have a common endpoint, and every vertex in G has exactly one incident edge in M. If edge e has weight $w(e)$ assigned to it, then a *minimum weighted matching* is the perfect matching with the minimum weight, where the weight of a matching is defined by $\text{weight}(M) = \sum_{e \in M} \text{weight}(e)$.

The "best" pairing of odd vertices can now be found by computing a minimum-weight perfect matching as follows: Define a complete graph K_{2k} which has as many vertices as there are odd vertices in G. For each odd vertices v in G, we thus have a vertex in K_{2k}, which we also denote by v.

If (u, v) is an edge in K_{2k}, then set $w(u, v)$ to be the length of the shortest path between the odd vertices u and v in G. For example, consider Figure 13.4(a). All odd degree vertices are denoted by open circles and are labeled with letters. Even vertices are filled circles. The complete graph corresponding to the odd vertices is given in Figure 13.4(b).

Directly from the definition of a minimum-weight perfect matching, it follows that the "best" pairing of odd vertices corresponds to a minimum-weight perfect matching in K_{2k}.

Transformation 4 *Finding the minimum shortest-path odd-vertex pairing is equivalent to finding a minimum-weight perfect matching in K_{2k}.*

Finding a minimum-weight perfect matching is not exactly trivial, but it can be done in polynomial time. Namely, finding a minimum-weight perfect matching is the same as finding a maximum-weight perfect matching in the graph where all costs have been set to the negative. The problem of finding maximum-weight perfect matching has been studied before. Edmonds [Edm65] showed how to find it in $O(n^4)$ time, and his algorithm can actually be implemented in $O(n^3)$ time [Law76].

13.2.6 Transforming back

Now we need to show how to obtain the maximum cut, by undoing the transformations done above.

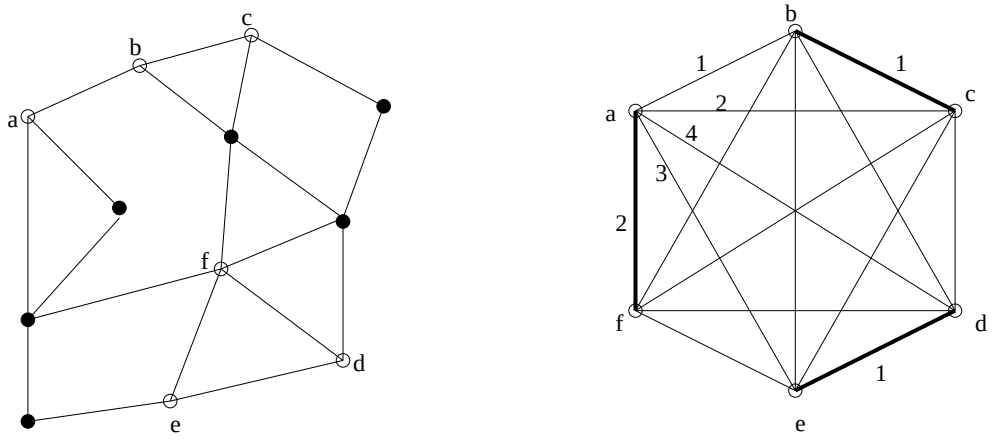


Figure 13.4: Part (a) is the original graph. Part (b) is the complete graph corresponding to the odd vertices. The numbers are the (minimal) distances between the vertices in the original graph (for clarity, not all numbers are included). The thick edges represent a minimum weighted matching.

- First compute the weighted graph K_{2k}. To compute the weights, we solve the all-pair shortest paths problems, which can be done with the algorithm by Floyd and Warshall in $O(n^3)$ time (see for example [CLRS00].)

- Next we find a minimum-weight perfect matching in K_{2k}. This takes $O(n^3)$ time.

- The minimum-weight perfect matching tells us the "best" pairing of odd vertices. Let P be the union of the shortest paths between the pairs.

- It was shown that P is a minimum set of edges whose contraction would make G^* Eulerian.

- Take the dual edges of P, and denote them by E_C. Then E_C is a minimum set of edges whose deletion would make G bipartite.

- Take the vertex partition $(C, \overline{C})$ of the bipartite graph $(V, E - E_C)$; this is then the maximum cut.

Theorem 13.6 *Maximum Cut can be solved in $O(n^3)$ time on a planar graph.*

The time complexity can be improved to $O(n^{1.5} \log n)$, and the same algorithm even works for a weighted version of Maximum Cut [SWK90].

13.3 Satisfiability once more

The fact that Maximum Cut is polynomial time solvable in planar graphs should give rise to some thinking. Inspecting the reduction for Maximum Cut again, one sees that the graph introduced during that reduction is almost exactly the graph of a satisfiability problem, and hence if this graph is planar then so is the graph for Maximum Cut. So why does this not show that 3-SAT is polynomial for planar graphs?

The difference is subtle, but important. The reduction for Maximum Cut was not from 3-SAT, but from NAE-3-SAT. The two problems seem very similar, but the small change makes a big difference. So our reduction from NAE-3-SAT to Maximum Cut, which works (without change) as a reduction from Planar NAE-3-SAT to Maximum Cut in planar graphs, shows the following theorem:

Theorem 13.7 *NAE-3-SAT is polynomial time solvable if the graph defined by the satisfiability instance is planar.*

This theorem is seems to have been pointed out explicitly for the first time in [KT00], though it likely was known before then.

Chapter 14

Maximum Flow

Network flow problems are central problems in operations research, computer science, and engineering and they arise in many real world applications. Starting with early work in linear programming and spurred by the classic book of Ford and Fulkerson [FF62], the study of such problems has led to continuing improvement in the efficiency of network flow algorithms. In this chapter a very efficient algorithm for finding a maximum flow in an st-planar graph will be introduced.

14.1 Background

A *network* is a simple graph G with two distinct vertices s and t, which we call *source* and *sink*, and a function $c : E \to R^+$, which assigns to each edge (i, j) the *capacity* c_{ij} of the edge. We assume that the network is undirected, i.e., $c_{ij} = c_{ji}$ throughout this chapter. See Figure 14.1 for an example.

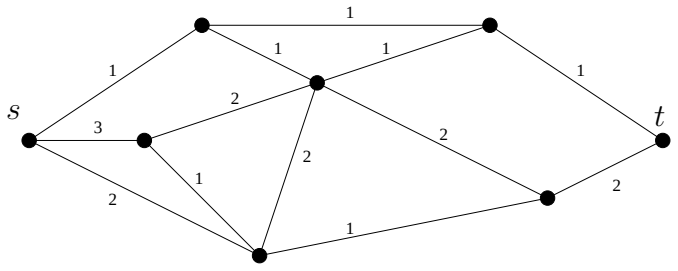


Figure 14.1: A sample network. The number next to each edge shows the capacity of that edge.

A *network flow* is a function $x : V \times V \to R^+$ which satisfies constraints outlined in the following. Usually, we write x_{ij} rather than $x(i, j)$. Also, note that $x_{ij} \neq x_{ji}$ is perfectly feasible. The constraints are as follows:

1. Capacity constraint: $0 \leq x_{ij} \leq c_{ij} \quad \forall (i, j) \in E$.

2. Balance constraint:

$$\sum_{(i,j)\in E} x_{ij} = \sum_{(j,k)\in E} x_{jk} \quad \forall j \in V, j \neq s, t.$$

3. $x_{ij} = 0$ if $(i,j) \notin E$.

The *maximum flow* is a flow x that maximizes $\text{value}(x) = \sum_{(s,i)\in E} x_{si} - \sum_{(j,s)\in E} x_{js}$; this is called the *value of the maximum flow*. Intuitively, it can be imagined as sending the maximum possible number of units from s to t.

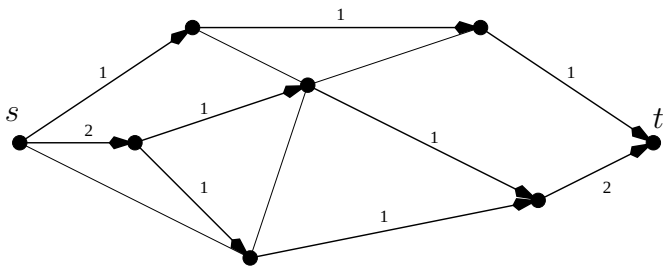


Figure 14.2: Maximum flow for the network in Figure 14.1. The value of the maximum flow is 3.

A *cut separating the source and the sink* is a partition of the vertices of the network into two disjoint sets $S, \bar{S} = V - S$ such that $s \in S$ and $t \in \bar{S}$ (Figure 14.3). For the rest of this chapter, we will shorten "a cut separating the source and the sink" to "a cut". The *value of a cut* is defined as $\text{value}(S, \bar{S}) = \sum c_{ij}$, where the sum is over all those edges (i,j) with $i \in S$ and $j \in \bar{S}$. A *minimum cut* is a cut with minimum value.

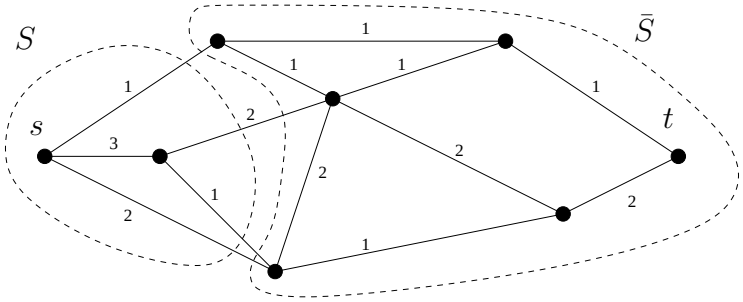


Figure 14.3: A cut $(S, \bar{S})$ with $value(S, \bar{S}) = 6$ in a graph.

It is easy to show (using the balance constraint) that if x is a flow and $(S, \bar{S})$ is a cut, then

$$\text{value}(x) = \sum_{\substack{(i,j)\in E \\ i\in S \\ j\in \bar{S}}} x_{ij} - \sum_{\substack{(i,j)\in E \\ i\in \bar{S} \\ j\in S}} x_{ij} \leq \sum_{\substack{(i,j)\in E \\ i\in S \\ j\in \bar{S}}} c_{ij} = \text{value}(S, \bar{S}). \tag{14.1}$$

In particular, this inequality also holds if we take the maximum flow and the minimum cut. In the famous Max-Flow-Min-Cut-theorem, it is shown that this inequality is actually an equality for the maximum flow and the minimum cut. This was first shown by Ford and Fulkerson [FF62]; a precise proof of this theorem can be found in [HY69].

Theorem 14.1 (*Max-Flow Min-Cut Theorem)* *For any network the value of the maximum flow from source to sink is equal to the value of a minimum cut separating the source and sink.*

Thus, if we want to know only the value of the maximum flow, we might as well find a minimum cut (which is easier for the graphs we study later). However, finding the maximum flow from the minimum cut then becomes a separate problem, which is not always trivial.

14.2 History

The maximum flow problem has been studied for over forty years. The first classical method for solving this problem was the *Ford-Fulkerson augmenting path method* [FF62]. This algorithm is based on the fact that a flow is a maximum flow if and only if there is no *augmenting path*, i.e., a path from s to t, such that for every edge (i, j) in the path, $c_{ij} - x_{ij} > 0$. The algorithm repeatedly finds an augmenting path and augments along it, until no augmenting path exists. This simple generic method need not terminate if the network capacities are irrational numbers, and it could take exponential time if the capacities are integers represented in binary [FF62].

Other classical method for the maximum flow problem are the *blocking flow method* of Dinic [Din70] (which runs in $O(mn)$ time), the *push-relabel method* of Goldberg [Gol85] (which runs in $O(n^3)$ time), and an improvement of the push-relabel method developed by Goldberg and Tarjan [GT88] (which runs in $O(nm \log(n^2/m))$ time). Cheriyan and Maheshwari showed that a variant of the algorithm introduced by Goldberg and Tarjan runs in $O(n^2\sqrt{m})$ time [CM89].

In this chapter we specially are interested in finding maximum flows when the underlying graph of the network is planar. The first efficient algorithm for solving the maximum network flow in planar graphs was developed by Itai and Shiloach [IS79]. The time-complexity of this algorithm was $O(n^2 \log n)$ and required $O(n)$ space. They also developed an algorithm that runs in $O(n \log n)$ time when the source and the sink of the network are on the same face of the graph. However, the latter algorithm only finds the value of the maximum flow, not the flow itself. It was shown by Hassin [Has81] how to extract the maximum flow. The reader can refer to [AMO93] for more details about these algorithms and algorithms for variants of the maximum flow problem.

14.3 Maximum flows in st-planar graphs

We will review here the algorithm by Hassin [Has81], which shows how to compute the maximum flow in a planar network where the source and sink are on the same face. Such a

graph is called an *st-planar graph*. We may assume that the face containing s and t is the outer-face (Lemma 11.4).

In an st-planar graph G, suppose that a curve passes through the graph from top to bottom such that the curve intersects only edges and does not pass through vertices. See Figure 14.4.

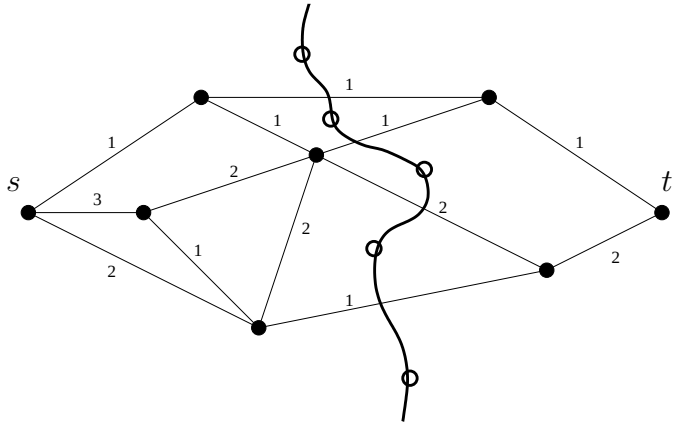


Figure 14.4: A curve cutting through edges of G.

Trivially, this curve induces a cut in graph G by separating its vertices into two disjoint sets. If the dual graph of G had two vertices for the outer-face (one for "above" and one for "below" G), then we could consider this curve as a path in the dual graph G^*. Formally we can define a dual network as follows:

1. Add an edge between s and t in primal network.
2. Take the dual graph; this will be the dual of the primal network.
3. Delete the dual edge between the two faces s^* and t^* incident to the new edge (s, t).

See Figure 14.5 for an illustration.

By definition, the value of the cut induced by the curve is the sum of the capacities of edges crosses by the curve. Thus, this value is the length of the path in the dual network from s^* to t^*, where we assume that for each edge e^* in G^* that is the dual edge of e, $cost(e^*) = capacity(e)$. Hence, to find the minimum cut induced by such a curve, we should compute the shortest paths from s^* to t^* (with this cost-function) in the dual network. This shortest path can be computed, for example with Dijkstra's algorithm, in $O(m \log n) = O(n \log n)$ time (see e.g. [CLRS00]).

Claim 14.2 *The length of the shortest path from s^* to t^* is the value of maximum flow in the network.*

To prove this claim, we will find the flow itself, and then show that it is a maximum flow in the network. We assume that we use a shortest path algorithm that finds shortest

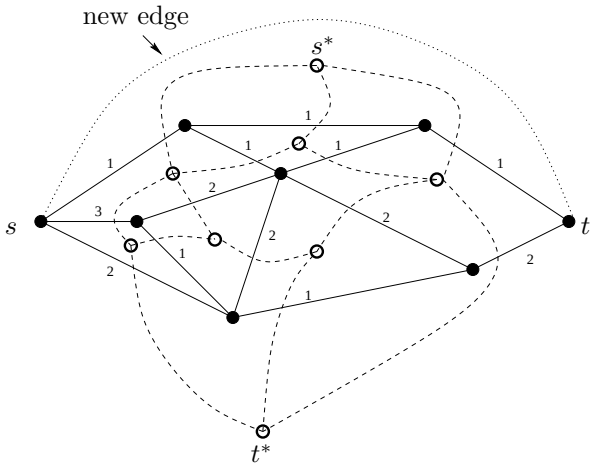


Figure 14.5: Dual of the network show in Figure 14.1. Dashed lines show the dual edges.

paths from s^* not only to t^*, but also to all other vertices (Dijkstra's algorithm satisfies this assumption). Denote for every vertex j^* (in the dual graph G^*) by $d(j^*)$ the distance of j^* from s^*.

Assume that (i, j) is an edge in the primal network. Let i^* be the face that is to the left of (i, j) (when walking from i to j), and let j^* be the face that is to the right of (i, j). Then define the flow for edge (i, j) by

$$\begin{aligned} x_{ij} &= \max\{0, d(i^*) - d(j^*)\}, \text{ and} \\ x_{ji} &= \max\{0, d(j^*) - d(i^*)\}. \end{aligned}$$

Also, set $x_{ij} = 0$ if (i, j) is not an edge.

To show that function x is a network flow we just need to show that it satisfies both constraints of a network flow:

1. Capacity constraints: The capacity constraints for x_{ij} clearly hold if (i, j) is not an edge, so let (i, j) be an edge. By definition of the flow we have $x_{ij} \geq 0$. Let i^* and j^* be the faces to the left and right of (i, j). Since $d(.)$ denotes the length of a shortest path from s^*, we have $d(j^*) \leq d(i^*) + \text{cost}(i^*, j^*)$. (This is a well-known inequality that holds for all shortest paths; see Figure 14.6 for an illustration). Therefore $d(j^*) - d(i^*) \leq c_{ij}$, which means that $x_{ij} \leq c_{ij}$.

2. Balance constraints: Let $v \neq s, t$ be a vertex, and let i^* be a face incident to v. We will show that the contribution of i^* is the same to both the incoming and the outgoing flow at v. Since all flow at v is defined via some face, and since this holds for all faces, this proves the balance constraint.

   So let i^* be a face incident to v, and let let j^* and k^* be the face before and after i^* in the clockwise order around v. Let e_1 be the edge common to j^* and i^* at v, and let e_2 be the edge common to i^* and k^* at v. See Figure 14.7. Note that i^* influences the flow-value on e_1 and e_2, but not on any other edge incident to v.

   We have four cases, depending on whether e_1 and e_2 are incoming or outgoing at v.

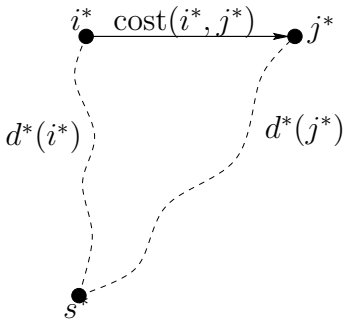


Figure 14.6: Dashed lines indicate the shortest path from s^* to i^* and j^*. If $d(j^*) > d(i^*) + \text{cost}(i^*, j^*)$, then we could find a shorter path to j^* by going first to i^* and then along (i^*, j^*).

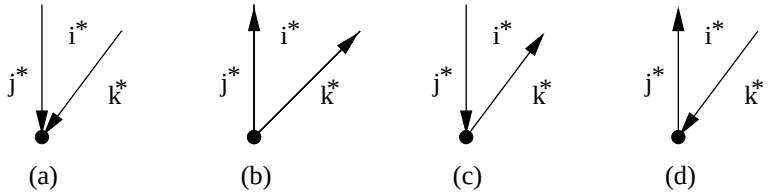


Figure 14.7: Four different cases of two consecutive edges of a vertex v.

- Both e_1 and e_2 are incoming at v.
  In this case the amount of incoming flow contains the term $d(i^*) - d(j^*) + d(k^*) - d(i^*) = d(k^*) - d(j^*)$. So i^* has no contribution to incoming flow and of course has no contribution to the flow of outgoing edges.
- Both e_1 and e_2 are outgoing at v.
  In this case the amount of outgoing flow contains the term $d(j^*) - d(i^*) + d(i^*) - d(k^*) = d(j^*) - d(k^*)$. So again i^* has no contribution to outgoing flow and of course has no contribution to the flow of incoming edges.
- e_1 is incoming, e_2 is outgoing at v.
  The incoming flow in this case contains the term $d(i^*) - d(j^*)$ and the outgoing flow contains the term $d(i^*) - d(k^*)$. So i^* has the same contribution to both.
- e_1 is outgoing, e_2 is incoming at v.
  The incoming flow for this case contains the term $d(k^*) - d(i^*)$ and the outgoing flow contains the term $d(j^*) - d(i^*)$. So i^* has the same contribution to both.

This discussion does not hold for s and t, because the outer-face (which is really two faces, s^* and t^*) does not contribute the same to the incoming and outgoing flow at s and t. In fact, it is very easy to show that the value of the outgoing flow of s (which is the value of the overall flow) is $d(t^*)$.

Now we want to show that flow x is a maximum flow. Let P^* be the shortest path from s^* to t^*, which we presume to be directed from s^* to t^*. For each edge $i^* \to j^*$ in P^*, direct

the dual edge from the face to the left of $i^* \to j^*$ (when walking from i^* to j^*) to the other face. Denote the collection of the directed edges that are dual to edges in P^* by P.

For every edge $i \to j$ on P^*, by property of a shortest path $d(j^*) = d(i^*) + \text{cost}(i^*, j^*)$. Therefore for any edge $i \to j$ in P, we have $x_{ij} = c_{ij}$ and $x_{ji} = 0$.

Define S to be all vertices that can be reached from source s without using an edge in P. By this definition, any edge between a vertex i in S and a vertex j not in S belongs to P. Therefore, $x_{ij} = c_{ij}$ and $x_{ji} = 0$ for any edge (i, j) with $i \in S$ and $j \in \bar{S}$. By Equation 14.1, we therefore have for this particular flow and cut

$$\text{value}(x) = \sum_{\substack{(i,j)\in E \\ i\in S \\ j\in\bar{S}}} x_{ij} - \sum_{\substack{(i,j)\in E \\ i\in\bar{S} \\ j\in S}} x_{ij} = \sum_{\substack{(i,j)\in E \\ i\in S \\ j\in\bar{S}}} c_{ij} = \text{value}(S, \bar{S}),$$

Thus, equality holds for this flow and cut in Equation 14.1, which means that this must be the maximum flow and the minimum cut. (Note that we did not use the max-flow-min-cut theorem to prove that this flow is maximum.)

To find the maximum flow, we compute the dual network, which can be done in linear time, and then run a shortest path algorithm in the dual network, which takes $O(n \log(n))$ time. So the whole running time of this algorithm is $O(n \log(n))$.

Theorem 14.3 *A maximum flow in an st-planar graph can be found in $O(n \log n)$ time.*

We note here that the time complexity can be improved to $O(n)$ time by using the (rather complicated) algorithm to find shortest paths in planar graphs in $O(n)$ time [KRRS97].

Chapter 15

Planarity Testing

In this chapter we discuss linear-time planarity testing algorithms. First, we give a brief history of this topic. Then we discuss several simplifications commonly used for preprocessing input in planarity testers. Finally we present the algorithm by Lempel, Even and Cederbaum.

15.1 History

The history of planarity testing is long and involved, and the story has not necessarily ended. Here are some of the more important steps:

- The history starts in 1961 with the paper by Auslander and Parter [AP61]. They were the first to show that planarity testing is in fact polynomial, though the time complexity of this algorithm is $O(n^3)$, hence not particularly fast.

- During the next decade, various attempts were made to decrease the time complexity. One notable one among those is the algorithm by Demoucron, Malgrange and Partuiset [DMP64], which is probably one of the simplest planarity testing algorithms known (a number of textbooks cover it, see for example [BM76, Gib85]). But unfortunately its time complexity is still relatively high at $O(n^2)$.

- One relatively simple algorithm that proves to be very useful later was presented in 1966 by Lempel, Even and Cederbaum [LEC67]. We will give some details of it below.

- In 1974, a significant breakthrough was achieved in that Hopcroft and Tarjan presented the algorithm to test planarity in linear time [HT74]. However, this algorithm was not entirely satisfactory on various accounts. First, it is relatively hard to understand, and even harder to implement so that its running time actually is linear time. Second, the algorithm only returns whether a graph is planar, but it does not actually find the planar embedding.

- In 1976, two independent results showed together that the algorithm by Lempel, Even and Cederbaum mentioned above actually also can be implemented in $O(n)$ time. First, Even and Tarjan [ET76] showed that an *st*-order (one of the vital ingredients for

the algorithm) can be found in linear time. Then, Booth and Lueker introduced the PQ-tree [BL76], with which some of the conditions of the algorithm can be tested in linear time. Hence, the results of [LEC67, ET76, BL76] together yields a linear-time planarity test.

Unfortunately, this still is not an entirely satisfying algorithm. The concept of the algorithm is simple enough, and finding the *st*-order in linear time is not difficult either, but the details of the linear-time implementation of PQ-trees are relatively complicated. Also, this algorithm yet again does not yield the planar embedding in linear time (though it can be obtained in $O(n^2)$ time.)

- Nothing happened for a decade. Then Chiba et al. [CNAO85] showed how to find the embedding during the Lempel/Even/Cederbaum algorithm in linear time. Then nothing happened for a decade. Then interest in planarity testing algorithms was revived, probably because people started to build libraries of graph algorithms and data structures, and in particular, packages that did graph drawing, especially for planar graphs. In 1996, Mehlhorn and Mutzel implemented Hopcroft and Tarjan's algorithm as part of the LEDA program package, and in the process, clarified the algorithm and explained how to find the planar embedding in linear time as well [MM96].

- Also in 1996, Di Battista and Tamassia developed an on-line planarity testing algorithm [BT96]. Here, the question is to decide whether the current graph is planar, under a sequence of changes to the graph that may or may not destroy or add planarity.

- In 1999, Boyer and Myrvold presented a paper at SODA [BM99] in which they gave an entirely new approach to planarity testing, based on doing a depth-first search with appropriate bookkeeping. (As a matter of fact, this approach wasn't so entirely new: in apparently independent work, Hsu and Shih [SH92] have developed an algorithm that appears to be the same one, though the details of this paper are not entirely clear).

  The conference paper by Boyer and Myrvold leaves out some details, especially as far as the linear time complexity of implementing the algorithm is concerned, and a journal version does not appear to be in sight yet. So there are still some doubts about this algorithm. In this scribe's opinion, there are no doubts that the algorithm works correctly, and it seems likely that it can be implemented in linear time, though this may be less obvious than the authors make it seem in the paper.

In summary, a truly simple linear-time planarity testing algorithm that could be explained in, say, a lecture at most is still waiting to be found. At this point in time, the best approach to planarity testing may be to either give up on linear time (and implement the algorithm by Demoucron et al.) or to use an exiting program package (for example LEDA) that has a linear-time planarity testing algorithm built in.

15.2 Assumptions on the input graph

So assume that we are given a graph G, and we want to know whether it is planar. If it is planar, we would like to get its planar drawing (represented by a combinatorial embedding).

If it is not planar we want to have some proof of its non-planarity.

We will first introduce a series of simple transformations which are common for most planarity testing algorithms. Later we can then discuss planarity testing omitting these simple issues and focus on the core of the algorithm.

More precisely, starting with the next section we will assume the following properties of the graph G:

- Graph G is simple.
- Graph G has at least 5 vertices.
- Graph G has at most $3n - 6$ edges.
- Graph G is connected.
- Graph G is biconnected.

In the rest of this section we will discuss each of these issues separately.

Graph G is simple

Assume G has loops and/or multiple edges. If the underlying simple graph G_s of G is planar, then it is easy to obtain a planar drawing of G as well: add any loop very close to its vertex, and any multiple edge close to the drawing of the single edge representing it in G_s. On the other hand, if G is planar, then so is G_s which is a subgraph. Hence G is planar if and only if G_s is.

Therefore to test the planarity of G we can first compute the underlying simple graph G_s of the graph G in $O(n+m)$ time (see Section A.3.1). Then we test whether G_s is planar; we know that G is planar if and only if G_s is planar.

Graph G has at least 5 vertices

Any graph which has at most 4 vertices is planar, because its underlying simple graph is a subgraph of K_4, which is planar. So if G has fewer than 5 vertices, then we can always output "yes" and stop.

Graph G has at most $3n - 6$ edges

We know now that G is simple and has at least 5 vertices. If G is planar, then it has at most $3n - 6$ edges by Lemma 11.9. Therefore if G has more than $3n - 6$ edges, this is proof that G is not planar.

Graph G is connected

If G has at least two connected components, then we can test planarity of each of its components separately. Therefore we compute the connected components of the graph in $O(n+m)$ time (see Section B.1.2). We test planarity of each connected component. Graph G is planar if and only if each connected component is planar.

Graph G is biconnected

It suffices to test planarity for each biconnected component of the graph by the following lemma:

Lemma 15.1 *A simple graph G is planar if and only if all its biconnected components are planar.*

Proof: If G is planar, then any biconnected component, which is a subgraph of G, is also planar.

We will prove the other direction by induction on the number of cut-vertices. In the base case, the graph has no cut-vertices and is biconnected, hence the statement holds trivially.

Now assume that G has a cut-vertex v, and the statement holds for all graphs with fewer cut-vertices. Let $G'_1, \ldots, G'_k$ be the connected components of $G - v$, and let G_i be the graph induced by the vertices of G'_i and v, $i = 1, \ldots, k$. Vertex v is not a cut-vertex for G_i since $G_i - v = G'_i$ is connected by definition. So we can apply induction on G_i, $i = 1, \ldots, k$.

Assume that all biconnected components of G (and therefore all biconnected components of $G_1, \ldots, G_k$) are planar. By the induction hypothesis all of $G_1, \ldots, G_k$ are planar. Take arbitrary planar drawings of $G_1, \ldots, G_k$. For each of these planar drawings, pick an arbitrary face F incident to vertex v, and modify the drawing such that face F becomes the outer-face of the drawing (Lemma 11.4). Then we can merge these drawings through the vertex v and we get a planar drawing of G (see Figure 15.1). Therefore G is also planar. □

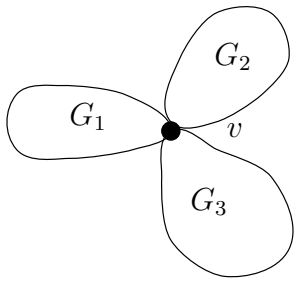


Figure 15.1: Merging planar drawings

Therefore, to test planarity, we compute the biconnected components of G in $O(m + n)$ time. We then test planarity of each biconnected component. Graph G is planar if and only if each biconnected component is planar.

15.3 *st*-order

Now we turn to the algorithm by Lempel, Even and Cederbaum. They define the so-called *st*-order and prove its existence for biconnected graphs.

Definition 15.2 *A vertex order $v_1, \ldots, v_n$ is called an* st-order *if (v_1, v_n) is an edge,*[1] *and if every vertex v_i, $1 < i < n$ has at least one predecessor and at least one successor.*

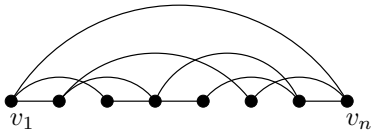


Figure 15.2: Example of an *st*-order.

Thus, for every vertex v in the order, we want one neighbour of v *before* v and one neighbour of v *after* v. Of course this isn't feasible for the first and last vertex of the order, but we want this for all other vertices. The name *st*-order comes from the observation that if the graph has an *st*-order, then we have an acyclic edge-direction with one source v_1 and one target v_n (proof left as an exercise.) This may sound familiar – we have seen a similar concept for 2-terminal SP-graphs – and in fact, one can easily show that every 2-terminal SP-graph for which the terminals are adjacent has an *st*-order with the terminals as first and last vertex.

It is quite easy to show that if G is biconnected, then it has an *st*-order for any choice of edge (v_1, v_n), and the proof gives a straightforwards $O(mn)$ algorithm to find it.

Lemma 15.3 *Let G be a biconnected graph with an edge (s, t). Then G has an st-order $v_1, \ldots, v_n$ with $v_1 = s$ and $v_n = t$.*

Proof: The proof builds up the *st*-order, starting with just the vertices v_1 and v_n, and adding a suitable set of vertices to it. Thus, the proof is by induction on the number of vertices that have not been added to the *st*-order yet. We start with the order $\{s, t\}$, which is easily verified to be an *st*-order. Now assume not all vertices have been included in the *st*-order. Find a vertex v that is in the current *st*-order, but one neighbour w of v has not been included.

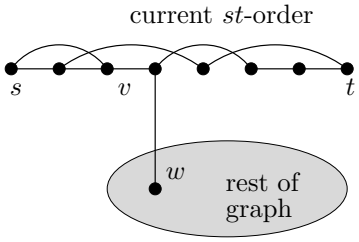


Figure 15.3: Find an edge (v, w) where v is in the *st*-order already and w is not.

Since G is biconnected, $G - v$ is connected, so there must exist a path P from w to some other vertex u that is already in the *st*-order. Assume that v comes before u in the current *st*-order, the other case is similar. Then add path P directly after v (and hence before u) to

[1] In some applications of *st*-order, (v_1, v_n) need not be an edge, but this is needed for planarity testing.

the *st*-order. See Figure 15.4. All vertices on P have at least one predecessor and at least one successor. All vertices $\neq s, t$ not in P had at least one predecessor and at least one successor by induction, so the resulting order is an *st*-order. We have also added at least one vertex (namely, w), so we are done by induction. □

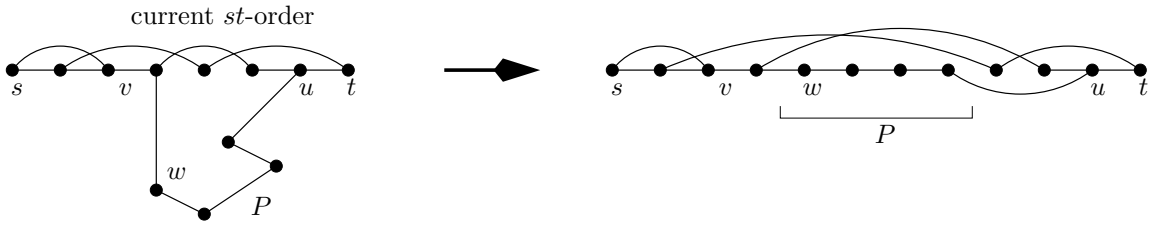


Figure 15.4: Adding a path that contains w to the *st*-order.

The other direction of this lemma also holds: if G has an *st*-order, then G is biconnected. We leave proving this as an exercise.

The proof of this lemma clearly yields an algorithm that takes at most $O(nm)$ time: we can run a depth-first search to find path P, and have to do this at most $n-2$ times until all vertices are included. Using the right data structure, and re-using some of the information from previous depth-first search runs, this running time can in fact be decreased to $O(m)$, see [ET76].

15.4 The algorithm by Lempel, Even and Cederbaum

Now assume that G is a plane graph with an *st*-order $v_1, \ldots, v_n$, and edge (v_1, v_n) is on the outer-face. This tells us a lot about the location of other vertices. Let G_i be the graph induced by $v_1, \ldots, v_i$, and use for G_i the planar embedding induced by G. Since v_n is on the outer-face, for each G_i, $i < n$, vertex v_n is located inside the outer-face of G_i. But even more can be said. For any v_j, $j > i$, there exists a path from v_j to v_n that only uses vertices not in G_i (this holds because we can get from v_j to a successor, to a successor, to a successor, and so on, until we must finally stop at v_n.) Therefore, this whole path must in the outer-face of G_i, and in particular, all v_j, $j > i$ must be in the outer-face of G_i.

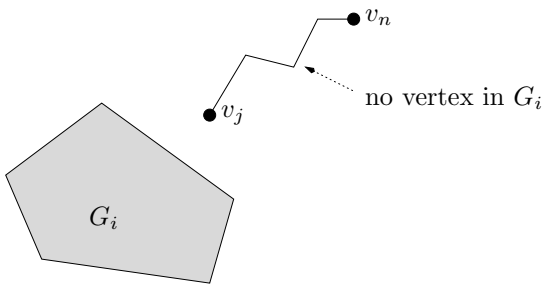


Figure 15.5: Each v_j, $j > i$ must be in the outer-face of G_i.

Now assume that G is a graph with an st-order $v_1, \ldots, v_n$ and we want to test whether G is planar. The idea is to build planar embeddings of G_i in such a way that $v_{i+1}, \ldots, v_n$ can be in the outer-face of G_i.

To do so, we define a slightly modified graph called the *bush form* B_i. B_i contains all vertices of G_i. Also, for each edge (v_h, v_j) of G with exactly one endpoint in G_i (say $h \leq i < j$), we add a vertex of degree 1 that is connected to v_j and labeled with j. See Figure 15.6.

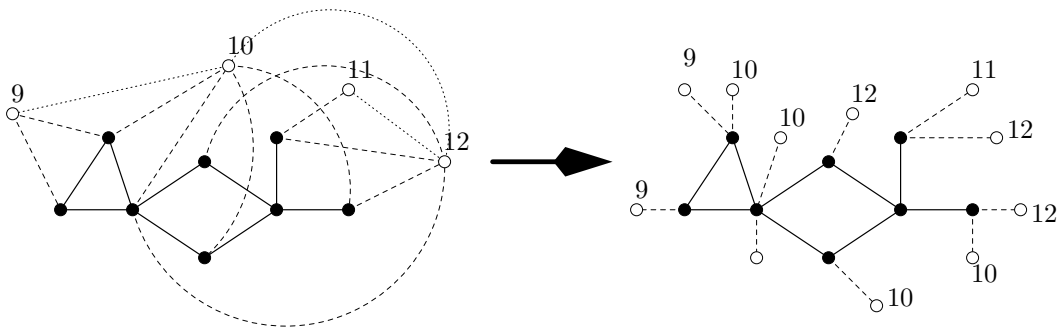


Figure 15.6: A graph (which is planar, but not in this embedding) and its 8th bush form.

The objective is to build embeddings of the ith bush form such that all degree-1 vertices are on the outer-face. This is straightforward for the first bush form, since it is a tree (in fact, a star.) In order to go from the $(i-1)$st bush form to the ith bush form, we need to rearrange the embedding of the graph such that the degree-1 vertices labeled i become consecutive.

Thus, we need to essentially try all embeddings of the $(i-1)$st bush form. This may seem like a daunting task, but is actually quite easy. One can show that there are only two ways in which planar embeddings can change while keeping all degree-1 vertices on the outer-face:

- If there is a cutvertex, then the order of the biconnected components can be permuted arbitrarily.
- Also, each biconnected component can be flipped to the reverse order.

These two operations should sound familiar: we can either permute all elements, or flip them to the reverse. We have seen this when we studied PQ-trees in Chapter 5, and it hence comes as no surprise that we can store all possible orders of degree-1 vertices of a bush form in a PQ-tree. Using the operations of the PQ-tree, we can then test whether we can make the degree-1 vertices labeled i consecutively. If we can, then we can update the PQ-tree by adding the constraint that these have to stay consecutively.

Similar as for interval-graph recognition, this is easy to implement in quadratic time. Using the ideas of Booth and Lueker [BL76], this can actually be reduced to $O(m)$ time, which is $O(n)$ time for planarity testing since we assumed that the graph has at most $3n-6$ edges.

Chapter 16

Triangulated graphs

16.1 Definitions

Recall that every planar simple graph with $n \geq 3$ vertices has at most $3n - 6$ edges. In this chapter, we now study planar simple graphs where this is tight. In particular, these graphs are maximal planar in the sense that we cannot add an edge to them without either destroying simplicity or planarity. A more commonly used term for them, however, is *triangulated planar graph*, which is what we study first.

Definition 16.1 *A triangulated graph is a plane graph where every face is a triangle.*

In Figure 16.1 we see a graph that is not triangulated because its outer-face is not a triangle, and a graph that is triangulated but is not simple.

(a) (b)

Figure 16.1: Examples concerning the definition of triangulated graphs.

Lemma 16.2 *Let G be a simple connected planar graph with at least 3 vertices. Then $m = 3n - 6$ if and only if G is a triangulated graph.*

Proof: Examine the proof of Lemma 11.9, where we proved that $m \leq 3n - 6$. In order to obtain equality $m = 3n-6$, we need to have number of lines L in the double-counting method equal to $3m$ which means that each face is incident to exactly three edges. Conversely, if every face is incident to exactly three edges, then equality holds and $m = 3n - 6$. □

From Euler's formula, which says $n - m + f = 2$, one can show easily that every simple triangulated graphs with at least 3 vertices has exactly $2n - 4$ faces.

Finally, we can show that triangulated graphs have a high connectivity.

Lemma 16.3 *Any simple triangulated graph is 3-connected.*

Proof: Assume that a graph G is simple and triangulated. The claim holds if $n \leq 2$, so assume $n \geq 3$. We will show that unless G is 3-connected, we can add an edge to G without destroying simplicity or planarity. The new graph then has $3n-5$ edges by Lemma 16.2 and is planar and simple, which contradicts Lemma 11.9.

Assume G is not connected, say it has components $G_1, \ldots, G_k$. Then some face must contain vertices from two components, say the outer-face contains $v_1 \in G_1$ and $v_2 \in G_2$. Then we can add edge (v_1, v_2); this will not destroy planarity because both vertices are on one face, and this edge didn't exist before because the endpoints were in different connected components.

Assume next that G is connected, but not biconnected, say it has a cut-vertex v which belongs to the biconnected components $G_1, \ldots, G_k$, $k \geq 2$. Scan the edges incident to v in clockwise order. At some point, there must be a transition point from edges in one biconnected component to edges in another biconnected component, say there are two consecutive edges (v, v_1) and (v, v_2) with $v_1 \in G_1$ and $v_2 \in G_2$. Then we can add edge (v_1, v_2); this will not destroy planarity because v_1 and v_2 are consecutive neighbors of v, and this edge didn't exist before because v_1 and v_2 belong to different biconnected components.

Finally assume that G is biconnected, but not triconnected, say it has a cutting pair $\{v, w\}$. Let $G'_1, \ldots, G'_k$ be the connected components of $G - \{v, w\}$, and let G_i be the subgraph of $G - (v, w)$ induced by v, w and the vertices of G'_i. Any edge $\neq (v, w)$ belongs to one of the graphs $G_1, \ldots, G_k$.

Scan the edges incident to v in clockwise order starting at (v, w) (if it exists) and at an arbitrary edge otherwise. The next edge belongs to one of the above subgraphs, say G_1. Keep scanning the edges until for the first time we encountered an edge (v, v_2) not in G_1. This edge cannot be (v, w), because we haven't seen any edge from G_2 yet, and we would be done scanning otherwise. So (v, v_2) belongs to some subgraph other than G_1, say G_2. Let (v, v_1) be the edge just before (v, v_2) in clockwise order.

Then we can add edge (v_1, v_2); this will not destroy planarity because v_1 and v_2 are consecutive neighbors of v. Also, this edge didn't exist before, because otherwise v_1 and v_2 would be in the same component of $G - \{v, w\}$, and thus in the same subgraph. □

See Figure 16.2 for an illustration of this proof. In particular, this lemma implies that every triangulated graph has a unique planar embedding by Whitney's theorem (Theorem 11.3).

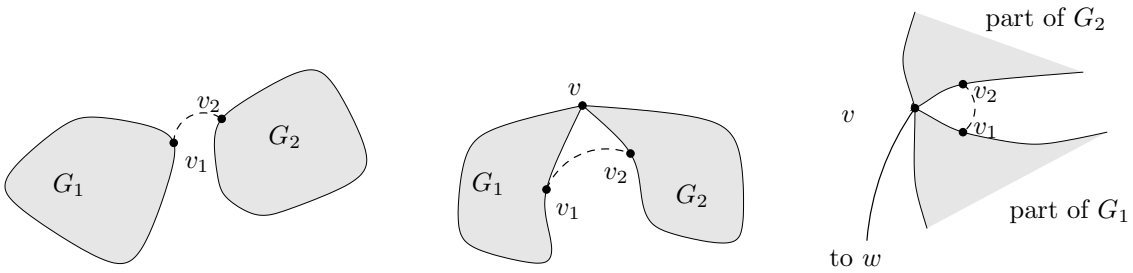


Figure 16.2: A triangulated graph is 3-connected, otherwise we can add an edge.

A word of warning: The name "triangulated graph" is heavily over-used in the literature.

- We use the term for planar graphs where all faces are triangles. Whenever possible, we'll write "planar triangulated graph" to make this clear.

- Some references (for example [Gol80]) refer to chordal graphs as triangulated graphs, mostly because for every cycle we have a chord, and therefore the shortest induced cycle must be a triangle.

  There is absolutely no connection between chordal graphs and planar triangulated graphs: there are many graphs that are chordal but not planar triangulated (e.g. any tree), and there are many graphs that are planar triangulated but not chordal (e.g. a double-pyramid.)

- A third use of "triangulated" (or more precisely, "triangulation") comes from computational geometry, graphics, and numerical simulation. Here, a triangulation refers to a set of points in $\mathbb{R}^2$ and straight-line edges between them such that all interior faces are triangles. This is quite closely related to planar triangulated graphs, but we do not require that the outer-face is a triangle. (In particular, triangulations need not be 3-connected.) We will sometimes refer to such graphs as *internally triangulated planar graphs.* Also, a triangulation naturally has a drawing associated with it, whereas a planar triangulated graph is specified only via its planar embedding.

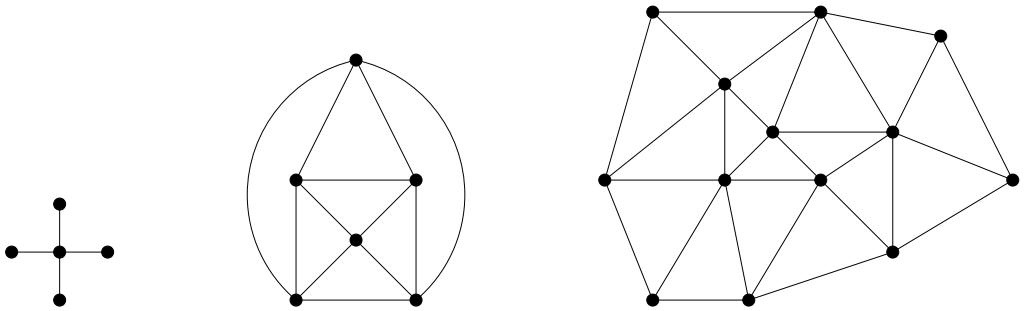


Figure 16.3: A chordal graph that is not a triangulated planar graph; a triangulated planar graph that is not a chordal graph; and a triangulation.

16.2 Making Graphs Triangulated

For some applications (e.g. bounds on the arboricity and graph drawing – see later), it will be useful to know how to make a planar graph triangulated. There are various graph operations one could use to make a graph triangulated:

- Add edges only,

- Add edges and vertices, or

- Add vertices and edges incident to at least one new vertex only.

Here we give an outline for the first way, which of course implies the second way. The third way is left as an exercise.

We proceed in steps. First we make the graph connected by adding edges; it should be obvious how to do this without destroying planarity (add edges between vertices on the outer-faces of the connected components). Next we make it biconnected by adding edges, and then finally we make it triangulated (and hence triconnected.) All our operations will maintain simplicity, i.e., will not add loops or multiple edges.

Theorem 16.4 *Let G be a simple connected plane graph with $n \geq 3$. Then we can find a set of $O(n)$ edges E' such that $(V, E \cup E')$ is a simple biconnected planar graph.*

Proof: We proceed by the number of biconnected components of G; if G has only one then G is biconnected and we are done.

So assume G has a cutvertex v. Fix an arbitrary planar embedding of G. While scanning edges around v, we must at some point have edges in one biconnected component B_1, and then edges in another biconnected component B_2. Let (v, v_1) be the last edge (in counter-clockwise order) in B_1, and let (v, v_2) be the next edge in counter-clockwise order; then (v, v_2) belongs to B_2. Adding edge (v_1, v_2) does not violate planarity, since v_1 and v_2 are consecutive neighbours of v. Also, this edge cannot have existed before, since B_1 and B_2 belonged to different connected components of $G - v$. Finally, adding this edge unifies the two biconnected component into one, so that the graph now has fewer biconnected components, and we are done by induction. □

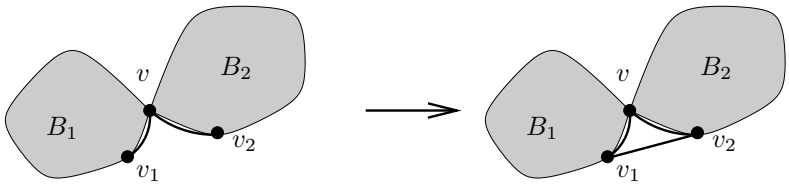


Figure 16.4: Joining two biconnected components by adding an edge.

The algorithm implicit in the proof of this theorem can be implemented in $O(m)$ time.

Before we can do similar (but more complicated) methods for making graphs triangulated, we need a simple observation.

Lemma 16.5 *Let G be a biconnected graph. Then every facial circuit is a cycle, i.e., no vertex appears on a face more than once.*

Proof: (Sketch) It is not hard to verify that if v appears on a facial circuit twice, then v is a cutvertex. See also Figure 16.5. □

Theorem 16.6 *Let G be a simple biconnected plane graph with $n \geq 3$. Then we can find a set of $O(n)$ edges E' such that $(V, E \cup E')$ is a simple, triangulated and planar graph.*

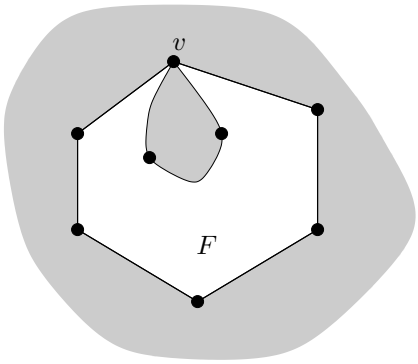


Figure 16.5: If a vertex appears more than once on a face, then it is a cut vertex.

Proof: Let F be a face of degree ≥ 4 with vertices $v_1, v_2, ..., v_k$ (see Figure 16.6(a)). The simplest idea to make the face F triangulated is to add edges $(v_1, v_3), (v_1, v_4), ..., (v_1, v_{k-1})$ (dashed lines in Figure 16.6(b)). But this does not always work, because there may be an edge (v_1, v_j) for $3 \leq j \leq k-1$ already, and hence we would add a multiple edge (see the dotted line in Figure 16.6(b)).

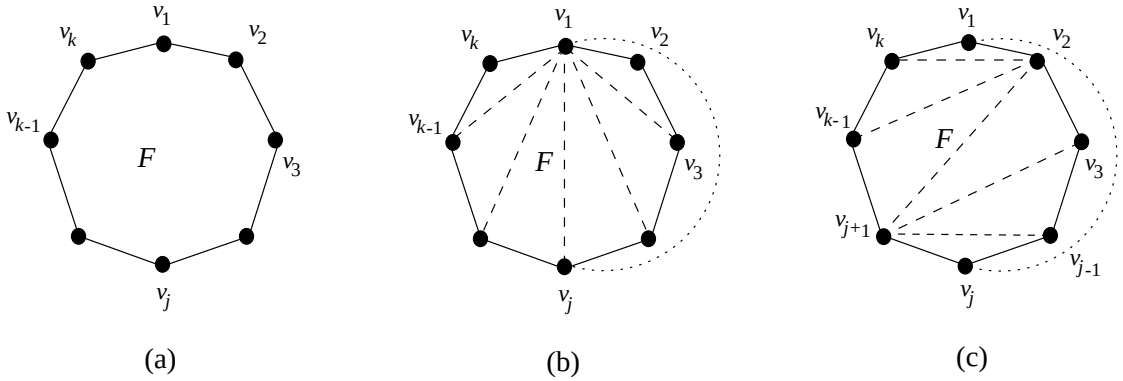


Figure 16.6: Making a planar graph triangulated.

Assume we have an edge (v_1, v_j) for $3 \leq j \leq k-1$ in the graph. This then implies that (v_2, v_k) is not an edge, for we could not route it without violating planarity or destroying face F. (A more precise argument is the following: assume F is a face, and (v_1, v_j) and (v_2, v_k) are both edges. Add a new vertex v inside F and connect it to v_1, v_2, v_j, v_k; this does not destroy planarity since F is a face. Now the facial circuit, plus vertex v, plus the edges (v_1, v_j) and (v_2, v_k), form a subdivision of K_5, which is a contradiction in a planar graph. This is indeed a K_5, because we know that none of the vertices v_1, v_2, v_j, v_k can be the same vertex repeatedly, since the graph is biconnected.)

Similarly one can show that none of the edges $(v_2, v_{k-1}), (v_2, v_{k-2}), ..., (v_2, v_{j+1})$, and $(v_{j+1}, v_{j-1}), (v_{j+1}, v_{j-2}), ..., (v_{j+1}, v_3)$ can exist. We add all these edges inside F (dashed lines in Figure 16.6(c)). This divides F into triangles.

We take another face with degree ≥ 4 and follow the same procedure. In this way, by induction, the whole graph becomes triangulated. □

Algorithm 1 Triangulate a Planar Graph G

```
1: fix an arbitrary embedding of a given graph G_i
2: for all faces F in this embedding do
3:   if deg(F) ≥ 4 then
4:     Let v_1 be a vertex of minimum degree in F
5:     Let v_2, v_3, ..., v_k be the remaining vertices of F in clockwise order
6:     Mark all neighbors of v_1 in the graph
7:     if none of v_3, ..., v_{k-1} is marked then
8:       Add edges (v_1, v_3), (v_1, v_4), ..., (c_1, v_{k-1})
9:     else
10:      {(say v_j is marked)}
11:      Add edges (v_2, v_{j+1}), (v_2, v_{j+2}), ..., (v_2, v_k)
12:      Add edges (v_3, v_{j+1}), (v_4, v_{j+1}), ..., (v_{j-1}, v_{j+1})
13:    end if
14:    Unmark all neighbors of v_1
15:  end if
16: end for
```

Above we give pseudo-code of the algorithm to make a planar graph triangulated. It differs from the one implicit in the proof in Theorem 16.6 in a small, but important, way. Previously, we let v_1 (Line 4) be an arbitrary vertex on the face, but now we pick a vertex of the minimum degree instead of an arbitrary one. This is needed for the improved time complexity.

Our goal is to show that this algorithm takes $O(n)$ time, which is not obvious at all. Finding all faces can be done in $O(n)$ time. Also, for each face F we need to find the vertex of minimum degree, which can be done in $O(\deg(F))$ time. Now for each face F, we need to mark and unmark the neighbors of a vertex of minimum degree. This costs $O(\deg(v))$ time, where v is the vertex in F with the minimum degree. Potentially, $\deg(v) \in \theta(n)$, so we will have to work harder to show that over all faces this does not become too large. All other operations per face take $O(\deg(F))$ time, so the total running time is proportional to

$$\sum_{F \text{ face}} \left\{ \deg(F) + \min_{v \in F} \{\deg(v)\} \right\}$$

Since $\sum_{F \text{ face}} \deg(F) = 2m$, we only need to be worried about the second term in this running time, and in particular, need to find a bound for

$$\sum_{F \text{ face}} \min_{v \in F} \{\deg(v)\}.$$

This will be done by repeatedly applying upper bounds. As a first step, note that

$$\min_{v \in F} \{\deg(v)\} = \min_{(v,w) \in F} \min\{\deg(v), \deg(w)\} \leq \sum_{(v,w) \in F} \min\{\deg(v), \deg(w)\}.$$

Since every edge belongs to exactly two faces, therefore

$$\sum_{F \text{ face}} \min_{v \in F}\{\deg(v)\} \leq \sum_{F \text{ face}} \sum_{(v,w)\in F} \min\{\deg(v), \deg(w)\} = 2 \sum_{(v,w)\in E} \min\{\deg(v), \deg(w)\}.$$

Now we have an expression that does not depend on planarity anymore. For general graphs, this expression might well be cubic (e.g., for the complete graph), but for planar graphs, one can show that it is at most linear. To do so, we need the concept of arboricity.

Definition 16.7 *Let G be a graph. We say that G has* arboricity $a(G) \leq k$ *if there exist k forests $F_1, F_2, \ldots, F_k$ on the vertices of G such that $E(G) \subseteq E(F_1) \cup E(F_2) \cup \cdots \cup E(F_k)$.*

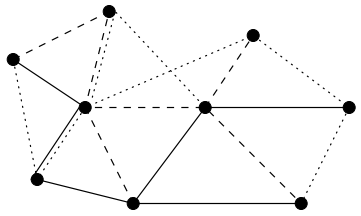


Figure 16.7: A graph with arboricity at most 3. The three forests are marked with dashed, dotted and solid lines.

The arboricity can be used to bound the running time of the algorithm to triangulate the graph because of the following result:

Lemma 16.8 *For any graph G,*

$$\sum_{(v,w)\in E} \min\{\deg(v), \deg(w)\} \leq a(G) \cdot 2m$$

Proof: Let F_i $(1 \leq i \leq a(G))$ be the edge-disjoint forests of G such that $E \subseteq \cup_{1\leq i\leq a(G)} E(F_i)$. We can orient the edges of each F_i such that any vertex v has at most one incoming edge in F_i, i.e., $\text{indeg}_{F_i}(v) \leq 1$. (In what follows, $\deg(v)$ denotes the degree of a vertex v in graph G.) Now we first analyze the sum for each edge in one forest F_i. We blatantly throw away the minimum and replace it by the head of the directed edge, and thus get

$$\sum_{(v,w)\in E(F_i)} \min\{\deg(v), \deg(w)\} = \sum_{v\to w\in E(F_i)} \min\{\deg(v), \deg(w)\} \leq \sum_{v\to w\in E(F_i)} \deg(w).$$

How often does $\deg(w)$ appear in this sum? As often as it has incoming edges in F_i. But w has at most one such incoming edge! Therefore,

$$\sum_{(v,w)\in E(F_i)} \min\{\deg(v), \deg(w)\} = \sum_{w\in V(F_i)} \text{indeg}_{F_i}(w) \deg(w) \leq \sum_{w\in V(F_i)} \deg(w) \leq 2m.$$

Since every edge belongs to one forest, we can now get the sum for all edges by summing up over the forests, and get

$$\sum_{(v,w)\in E(G)} \min\{\deg(v),\deg(w)\} = \sum_{i=1}^{a(G)} \sum_{(v,w)\in E(F_i)} \min\{\deg(v),\deg(w)\} \leq \sum_{i=1}^{a(G)} 2m = 2m \cdot a(G)$$

as desired. □

Now we return to the triangulation algorithm.

Theorem 16.9 *Any simple planar graph can be made triangulated by adding edges. A set of such edges can be found in $O(n)$ time.*

Proof: We gave the algorithm earlier and analyzed its running time to be proportional to $2m+\sum_{(v,w)\in E}\min\{\deg(v),\deg(w)\}$, which we know by Lemma 16.8 to be at most $2m+2m\cdot a(G)$. This still is not linear in general (for example, the complete graph has arboricity $n/2$), but as we will see soon, planar graphs have arboricity at most 3, and hence this evaluates to $O(m) = O(n)$. □

16.3 Canonical Ordering

Now we introduce the canonical ordering, which is a useful tool for graph drawing and other applications. In the following, whenever we speak of a planar triangulated graph, we assume that the planar embedding is fixed (there is only one since the graph is triconnected), and an outer-face has been chosen.

Definition 16.10 *A canonical ordering is a vertex order $v_1, v_2, \ldots, v_n$ of a triangulated planar graph such that v_1, v_2, v_n is the outer-face, and for all k with $3 \leq k \leq n-1$, the graph G_k induced by $v_1, v_2, \ldots, v_k$ is 2-connected, and v_{k+1} is in the outer face of G_k,*

Figure 16.8 gives an example of a canonical order, and illustrates how v_{k+1} is added to graph G_k.

The original definition of canonical orders (see [FPP90]) stipulates more conditions on the order, but all of those follow from the ones listed above as follows:

Lemma 16.11 *Let G be a planar triangulated graph with canonical order $v_1, \ldots, v_n$. Then for any $3 \leq k \leq n-1$, the following holds:*

1. *The outer-face of G_k contains edge (v_1, v_2).*
2. *v_k has at least two predecessors.*
3. *Every $v_j, j > k$ is in the outer-face of G_k.*
4. *Every internal face of G_k is a face of G. In particular, G_k is internally triangulated.*

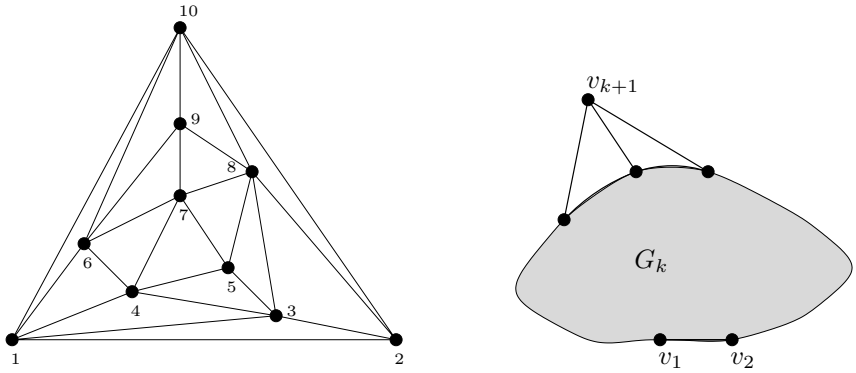


Figure 16.8: An example of a canonical ordering, and how v_{k+1} is added to G_k.

5. *The neighbours of v_{k+1} form an interval (i.e., a path) on the outer-face of G_k.*

6. *v_k has at least one successor.*

Proof: (1) follows immediately because the outer-face is $\{v_1, v_2, v_n\}$. (2) is also straightforward, because G_k is 2-connected and has $k \geq 3$ vertices, so v_k must have at least two neighbours in G_k, and these are predecessors of v_k.

(3) is proved by induction. v_j is in the outer-face of G_{j-1}, which is part of the outer-face of G_{j-2} since v_{j-1} is in the outer-face of G_{j-2}. Using induction the outer-face of G_{j-1} is part of the outer-face of G_k, and hence v_j is in it.

(4) follows easily from (3). For since all vertices in $G - G_k$ belong to the outer-face of G_k, all internal faces of G_k are unchanged in G. Since G is triangulated, all internal faces of G_k are hence also triangulated.

(5) follows from (4). Since v_{k+1} is in the outer-face of G_k, all of its neighbours in G_k must be on the outer-face of G_k. When adding edges from v_k to its neighbours in G_k, we are creating some internal faces of G_{k+1}. These must be triangles by (4), and hence any consecutive neighbours of v_{k+1} must be adjacent, and the neighbours form a path on the outer-face of G_k.

(6) follows from (5). Since $3 \leq k \leq n-1$, vertex v_k does not belong to the outer-face $\{v_1, v_2, v_n\}$ of G, but it belongs to the outer-face of G_k. So at some point v_k must disappear from the outer-face, say when adding vertex v_j, $j > k$. By (5), vertex v_j is adjacent to v_k and hence a successor of v_k. □

Note in particular that the canonical order is an *st*-order, since every vertex has a predecessor and a successor, and (v_1, v_n) is an edge. But the canonical order is more, since actually every vertex (except the first two) has *two* predecessors. There are further generalizations of this idea (for 4-connected triangulated graphs), but we will not go into this; see [KH97].

We first need to establish that such canonical orders actually exist. They do, regardless of how the outer-face is chosen.

Theorem 16.12 *Let G be a triangulated planar graph, and let $\{a, b, t\}$ be the outer-face of G. Then there exists a canonical order of G such that $v_1 = a$, $v_2 = b$ and $v_n = t$.*

Proof: We show this by reverse induction, i.e., we show how to choose for $k = n, n-1, \ldots$ a vertex v_k that satisfies the conditions.

Set $v_n = t$, then $G_{n-1} = G - v_n$. Since v_n is on the outer-face of G, it is in the outer-face of G_{n-1}. Also, G_{n-1} is 2-connected since G is triangulated and therefore 3-connected. So the conditions hold. Also note that (a, b) is on the outer-face of G_{n-1}; we will maintain this for all graphs G_k that we build.

Assume $k \geq 3$, and we have chosen $v_{k+1}, \ldots, v_n$ such that $G_k = G - \{v_{k+1}, \ldots, v_n\}$ is 2-connected and has (a, b) on the outer-face. We also chose $v_{k+1}, \ldots, v_n$ such that they are in the outer-face of G_k, which implies that G_k is internally triangulated as in Lemma 16.11. Now, we want to pick v_k on the outer face of G_k such that $v_k \neq a, b$ and $G_k - v_k$ is 2-connected. Not any vertex on the outer-face of G_k can be taken; see Figure 16.9.

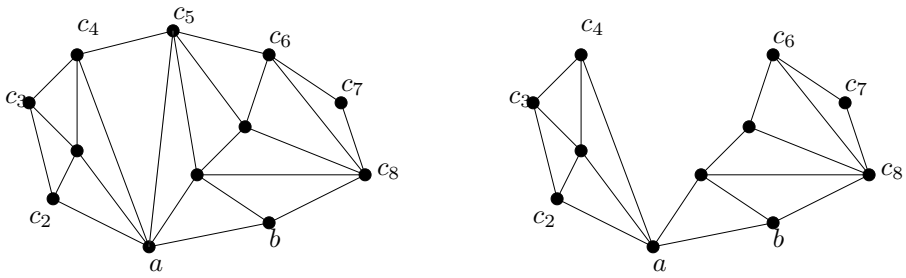


Figure 16.9: Picking $v_k = c_5$ would cause $G_k - v_k$ to become not 2-connected.

In order to ensure $G_k - v_k$ is 2-connected, we need to choose v_k as a vertex on the outer-face that is not a or b and also is not incident to a chord. We claim that there always is such a vertex. This is obvious if there are no chords by $k \geq 3$. If there are chords, let the outer-face be $c_1 = a, c_2, c_3, \ldots, c_p = b$. Let (c_i, c_j) be the chord that minimizes $j - i$ (In Figure 16.9, we have $i = 6$ and $j = 8$.) Now consider vertex c_{i+1}. Since (c_i, c_j) is the chord that minimizes $j - i$ we know there is no other chord within the vertices $\{c_i, c_{i+1}, \ldots, c_j\}$. Since the graph is planar there cannot be a chord between c_{i+1} and a vertex outside of interval c_i, c_j. Since $1 \leq i < i + 1 < j \leq p$, also $c_{i+1} \neq a, b$, so c_{i+1} can be chosen as vertex v_k.

In this manner, we can find all vertex $v_3, \ldots, v_n$. Finally, we are left with two vertices (which by choice are a and b), and we simply set $v_1 = a$ and $v_2 = b$. □

16.4 Applications of the canonical order

Canonical orders have many applications. They were developed primarily for results in graph drawing [FPP90, Sch90, Kan96], but have also applications for encoding planar graphs [HKL99], and give an easy proof for the arboricity of a planar graph. We start with this last result.

16.4.1 Arboricity

Recall that the arboricity of a graph is the number of forests needed to cover all edges. Assume that we have the canonical ordering $\{v_1, v_2, \ldots, v_n\}$ on a planar triangulated graph. Then label edge (v_1, v_2) with "1" and label all other edges with $\{1, 2, 3\}$ as illustrated in Figure 16.10. More precisely, for edges from G_k to vertex v_{k+1}, assign "1" to the leftmost edge, "2" to the rightmost edge and "3" to the edges in-between. Recall that v_{k+1} has at least two neighbours in G_k, so there must be an edge labeled 1 and an edge labeled 2, but there need not be an edge labeled 3.

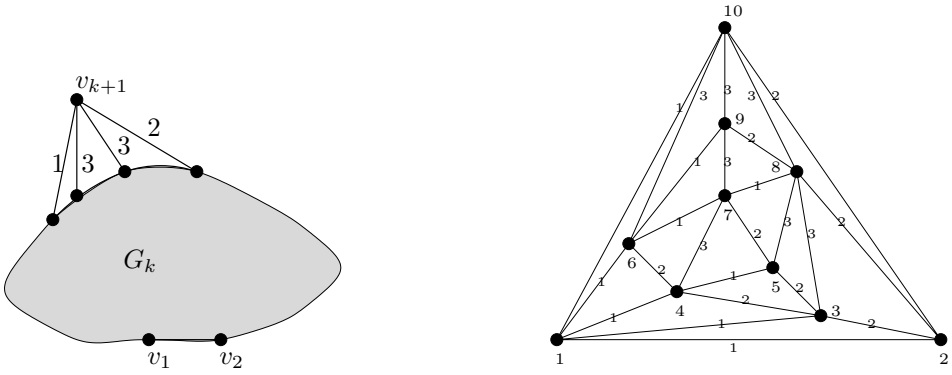


Figure 16.10: Label the directed edges with 1, 2 or 3.

We claim that for each $i \in \{1, 2, 3\}$, the edges labeled i form a forest. Since every edge is labeled, this proves that the arboricity of G is at most 3.

To see why the edges labeled i form a forest, temporarily direct each edge from the lower-numbered to the higher-numbered endpoints; this is an acyclic orientation. Now note that each vertex has at most one incoming edge labeled 1, which together with acyclic-ness implies that the edges labeled 1 are a forest. Similarly the edges labeled 2 are a forest. A vertex may have many incoming edges labeled 3, but it has at most one outgoing edge labeled 3, because it gets one only if it disappears from the outer-face. Therefore all edges labeled "3" also form a forest.

One can show that all three forests are in fact trees. Therefore any planar triangulated graph can be split into three trees, a result also known as Schnyder's tree decomposition, which was found independently (and at the same time) as the canonical order [Sch90], and has some other interesting implications (for example, every partially ordered set for which the Hasse diagram is planar has dimension at most 3... but we *really* cannot get into this.)

Theorem 16.13 *Every planar graph has arboricity at most 3.*

Proof: Let G be a planar graph. As in Section 16.2, add edges to G until we have a triangulated planar graph G'. From Schnyder's tree decomposition, we know that $a(G') \leq 3$. Since deleting edges cannot possibly increase the arboricity, therefore $a(G) \leq a(G') \leq 3$. □

In particular therefore, we now really know how to make a planar graph triangulated in linear time by adding edges. (The fact that we're using "making graphs triangulated" to prove Theorem 16.13 is not a contradiction, since we need the bound on the arboricity only for the bound on the running time, not for the ability to do it.)

16.4.2 Visibility representations

A *visibility representation* of a graph is a drawing of the graph using horizontal line segments for vertices and vertical line segments for edges such that the edges do no cross vertex. (For ease of reading, line segments for vertices are often fattened into axis-parallel boxes.). A graph has a *strong visibility representation* if all visibility lines between the segments representing vertices are also edges; if only some of these are edges then the representation ion is a called a *weak visibility representation* (see Figure 16.11).

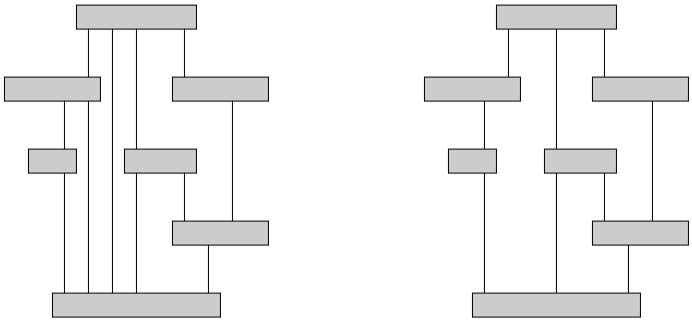


Figure 16.11: An example of a strong and a weak visibility representation.

In 1978, Otten and Van Wijk [OvW78] showed that every planar graph admits a weak visibility representation and in 1986 Rosenstiehl and Tarjan [RT86] and independently, Tamassia and Tollis [TT86] gave a linear time algorithm for constructing this representation. Their approach uses the *st*-order that we saw in Chapter 15. We will give the result here for triangulated graphs, where we actually get a strong visibility representation.

Theorem 16.14 *Every planar triangulated graph has a strong visibility representation and every planar graph has a weak visibility representation.*

Proof: Let $\{v_1, v_2, \ldots, v_n\}$ be the canonical ordering of the graph G. We start with a single edge (v_1, v_2) and represent it as in Figure 16.12. Above the visibility representation, we have also labeled the visibility interval of each vertex from above. These intervals are created by scanning a vertical ray (whose source is above the graph) from left to right and determining which vertex the ray hits first. We will maintain the invariant that for $k \geq 2$ these intervals reflect exactly the vertices on the outer-face of G_k, and are ordered while going clockwise around the outer-face from v_1 to v_2.

So assume G_k has been drawn such that this invariant holds. To add v_{k+1}, we create a new segment above the current visibility representation. Let c_i and c_j be the leftmost and rightmost neighbour of v_{k+1} in G_k (with respect to the order on the outer-face.) Place the

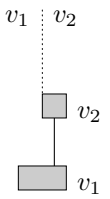


Figure 16.12: The base case for creating a visibility representation from a canonical order.

box for v_{k+1} such that it covers partially the intervals for c_i and c_j (and fully all intervals in-between.) By Lemma 16.11(5), v_{k+1} is adjacent to all vertices between c_i and c_j on the outer-face of G_k (and to none else by choice of c_i and c_j), so all visibility lines from v_{k+1} correspond to edges. Also, the invariant continues to hold, since c_i and c_j are still on the outer-face. See Figure 16.13.

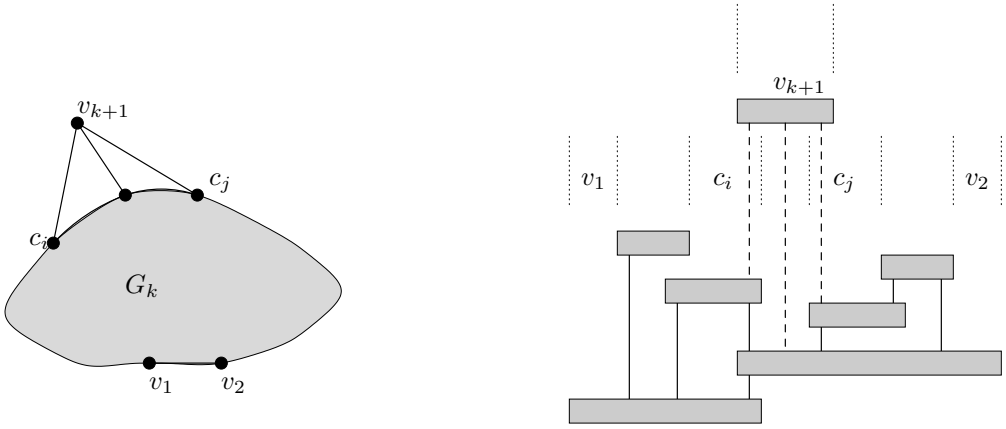


Figure 16.13: Update the visibility intervals as a new vertex is added.

Thus, we have shown how to build a strong visibility representation for planar triangulated graphs from a canonical order. Since every planar graph can be made triangulated by adding edges, this also gives a (weak) visibility representation for all planar graphs. □

Note that visibility representations resemble grid graphs. We can use them to show that every planar graph in fact is a minor of a grid graph.

Theorem 16.15 *Every planar graph is the minor of a $k \times k$-grid, for k sufficiently large.*

Proof: Take a visibility representation of the planar graph G. It is not hard to see that there exists a visibility representation such that all segments of vertices and edges have integer coordinates at the endpoints (in fact, one can show that an $(2n-4) \times n$-grid is enough.) Then this visibility representation is actually a subgraph of the grid. See Figure 16.14. We can get G from the visibility representation by contracting the paths which represent each vertex, and contracting all but one edge on each path that represents an edge. □

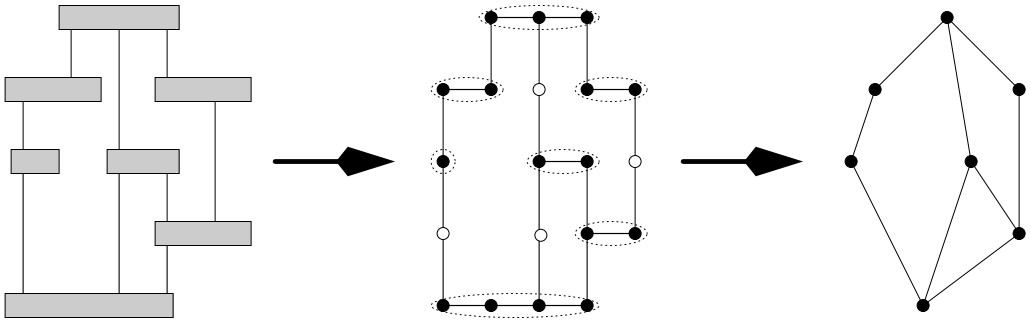


Figure 16.14: Converting a visibility representation to a grid graph.

16.4.3 Straight-line planar drawings

We now study the application that inspired the concept of a canonical ordering, which is straight-line drawings of planar graphs.

Recall that a planar drawing of a graph is a drawing in the plane such that no two edges cross. A *planar straight-line drawing* is a special case of a planar drawing: all edges must be drawn as straight-line segments between their endpoints. Thus, vertices are assigned to points in the plane, and each edge is represented by the straight-line segment between its endpoints. No two edges should cross. Also, no edge should be going "through" a non-incident vertex. In a more formal description, the interior of the straight-line segment of every edge should not intersect any other element of the drawing.

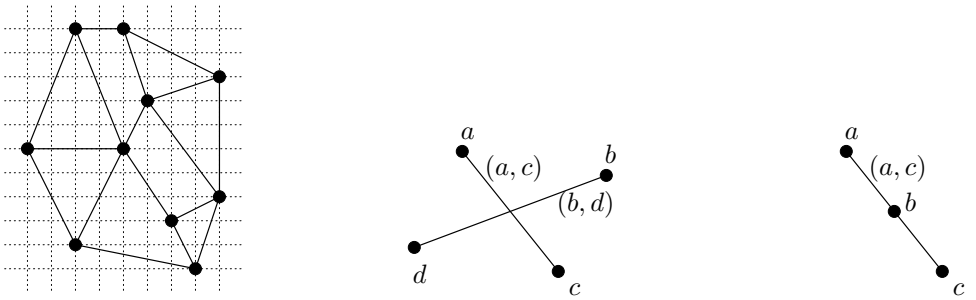


Figure 16.15: A planar straight-line drawing (in fact, a planar straight-line grid-drawing), and the two forbidden configurations for a planar straight-line drawing.

While by definition every planar graph has a planar drawing, it is not clear at all whether every planar graph has a planar straight-line drawing. However, this indeed is the case, and this result was among the first results in the area of graph drawing. It was proved independently by a number of people, among them Wagner [Wag36], Fáry [Fár48] and Stein [Ste51].

Theorem 16.16 *Every planar simple graph has a planar straight-line drawing.*

The proof that we give here is none of the above proofs, but instead uses the canonical ordering, and is based loosely on the work by de Fraysseix, Pach and Pollack [FPP90] for *planar straight-line grid-drawings*. In such drawings, we want a planar straight-line drawing

such that additionally all vertices have integral coordinates and such that the area of the used grid is small.

So assume that $\{v_1, \ldots, v_n\}$ is a canonical ordering of a planar triangulated graph. (Recall that we can make every planar graph triangulated by adding edges, so if we can show that every triangulated graph has a planar straight-line drawing, then every planar graph has a straight-line drawing as well.) For $i = 3, \ldots, n$, we will build a straight-line drawing of the graph induced by $v_1, \ldots, v_i$ using induction. To be able to add v_i efficiently, we will keep the following invariant:

Invariant 16.17 *Let G_k be the graph induced by $v_1, \ldots, v_k$, and assume that its outer-face consists of $v_1 = c_1, c_2, \ldots, c_p = v_2$ in clockwise order. Then G_k has a straight-line drawing such that $x(c_1) < x(c_2) < \cdots < x(c_p)$, where $x(v)$ denotes the x-coordinate or vertex v.*

This invariant clearly holds for $k = 2$, by drawing (v_1, v_2) as a horizontal line segment. Now assume that we have such a drawing for G_k; we will show how to add v_{k+1} to it. Assume that v_{k+1} is adjacent to $c_i, \ldots, c_j$ on the outer-face of G_k, with $i < j$. Let $x(v_{k+1})$ be some value with $x(c_i) < x(v_{k+1}) < x(c_j)$; if we place v_{k+1} with this x-coordinate then the invariant will be satisfied.

We must be a little careful in choosing the y-coordinate for v_{k+1} to avoid introducing crossings or overlap. But it is clear that some suitable y-coordinate must exist. For each of $c_i, \ldots, c_j$ (which are the neighbors of v_{k+1}) can see upwards towards infinity by the invariant. This means that at infinity, the upward lines from these points intersect the line $x = x(v_{k+1})$, and thus at $y = \infty$, the line from c_i to $(x(v_{k+1}), y)$ does not intersect any other element of the drawing. But in fact, since all lines in the drawing have finite slope, already for some finite value of y, the line from c_i to $(x(v_{k+1}), y)$ does not intersect any other element of the drawing.[1] Setting $y(v_{k+1})$ to be the maximum over all these restrictions on y implied by the neighbors of v_{k+1} will yield a feasible y-coordinate. See Figure 16.16 for an illustration.

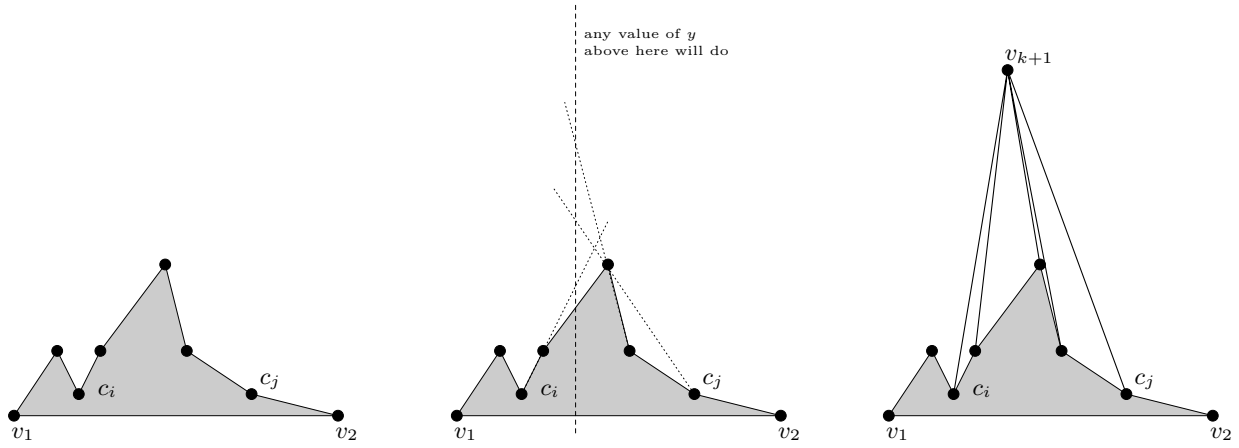


Figure 16.16: Adding to the straight-line drawing.

[1]The precise value of y that we can use depends on the other coordinates and is of no importance for the general proof; for detailed proofs that also involve bounds on the grid size, we use stronger invariants that allow for specific values of y.

Since we can repeatedly add vertices until we have the whole graph, this proves that every planar graph has a straight-line drawing. But in fact, with a slight strengthening of the invariant, we can achieve a grid-drawing. If we demand that all slopes of edges on the outer-face are either $+1$ or -1 (with the exception of the edge (v_1, v_2)), then it is easy to see that the edges (c_i, v_{k+1}) and (v_{k+1}, c_j) only need slopes $+1$ and -1, respectively. However, to avoid overlap, we then must start to modify the drawing of G_{i-1} and "shift" some vertices; see [FPP90] for details. In total, this then leads to a width of $2n - 4$ and a height of $n - 2$; in particular the area of the resulting grid-drawing is $O(n^2)$.

Chapter 17

Friends of planar graphs

We now turn to other graph classes related to planar graphs. We will only study subclasses of planar graphs, namely, SP-graphs, outer-planar graphs, and k-outerplanar graphs. There are some interesting super-classes as well (for example, toroidal graphs, i.e., graphs that can be drawn without crossing on a torus), but space and time constraints do not permit to study these in detail.

17.1 SP-graphs

Recall that a 2-terminal series-parallel graph was defined recursively by combining smaller 2-terminal SP-graphs either in series or in parallel. Also, an SP-graph is a graph for which all biconnected components are 2-terminal SP-graphs. One can easily show that every SP-graph is a planar graph.

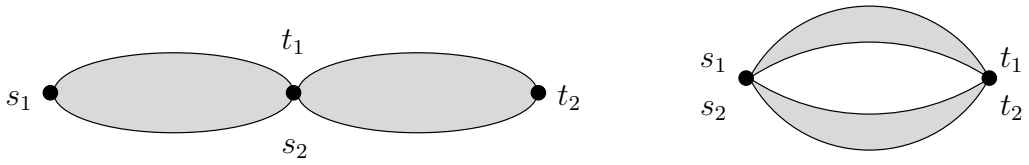


Figure 17.1: Combination in series and in parallel.

Lemma 17.1 *Every SP-graph is planar.*

Proof: Since a graph is planar if and only if all biconnected components are planar, it suffices to show that 2-terminal series-parallel graphs are planar. We will show a slightly stronger statement: a 2-terminal SP-graph with terminals s and t has a planar embedding such that both s and t are on the outer-face.

This clearly holds for an edge (s, t). Now assume that a 2-terminal SP-graph has been obtained as serial or parallel combination. Recursively find planar embedding of the subgraphs such that the terminals are on the outer-face. By combining these two embeddings in the obvious way (see Figure 17.1), we get the desired embedding for the graph. □

Not every planar graph is an SP-graph, for example K_4 is not an SP-graph.

17.2 Outerplanar graphs

A planar graph G is called an *outer-planar* graph if there is some planar embedding of G such that all vertices are on the outer-face. Figure 17.2 shows some examples of outer-planar graphs.

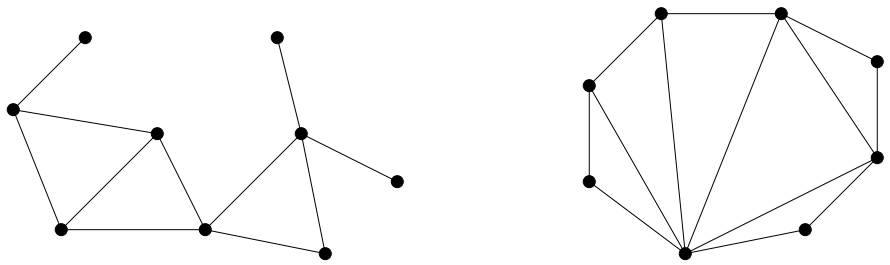


Figure 17.2: Examples of outer-planar graphs.

An example of a planar graph that is not an outer-planar graph is K_4. This graph is triangulated, and hence has only one possible planar embedding, and in this embedding, one vertex is not on the outer-face. Generally, triangulated graphs are not outer-planar graphs (except for the trivial case of a 3-cycle), since the outer-face by definition contains only 3 vertices.

However, internally triangulated graphs can be outer-planar, as the third example in Figure 17.2 shows. This graph has another special property: we cannot add another edge to it and stay outer-planar. We will need a name for this: A simple graph G is called *maximal outerplanar* if G is outer-planar, and no edge can be added to G without destroying simplicity or creating a graph that is not outer-planar.

Lemma 17.2 *Let G be a maximal outer-planar graph. Then G contains a Hamiltonian cycle $v_1, \ldots, v_n$ and is internally triangulated. In particular, G can be viewed as the triangulation of a convex polygon.*

Proof: First note that G necessarily must be biconnected. For if G contains a cutvertex v, then v must appear twice on the outer-face. We can then add an edge between two consecutive neighbours of v (as in Theorem 16.4) to connect two biconnected components incident to v. This removes one incidence of v to the outer-face, but keeps the other one intact, so the resulting graph is still outer-planar, contradicting the maximality of G.

So G is biconnected. By Lemma 16.5, the outer-face then is a cycle and contains every vertex at most once. By definition of outer-planar, the outer-face contains every vertex at least once. So the outer-face contains every vertex exactly once, and is a Hamiltonian cycle.

If any interior face of G were not a triangle, then we could add edges to it without destroying simplicity, which contradicts the maximality of G. So G is internally triangulated. □

From the proof of this lemma, it also follows that every outerplanar graph can be made into a maximal outerplanar graph by adding edges (first to make it biconnected and then to make it internally triangulated). This can be done in $O(n)$ time.

The reverse of this lemma does not hold, i.e., there exist graphs that have a Hamiltonian cycle and that are internally triangulated but that are not outer-planar. K_4 is an example of such a graph.

Finally, we can obtain a bound on the number of edges of outerplanar graphs.

Lemma 17.3 *A simple maximal outer-planar graph has $2n-3$ edges, and every simple outer-planar graph has at most $2n-3$ edges.*

Proof: A simple maximal outer-planar graph has n edges in the Hamiltonian cycle. Also, we need $n-3$ chords to triangulate a polygon with n vertices, so in total we have $2n-3$ edges. The second claim holds because every outerplanar graph can be made maximal outerplanar by adding edges. □

It is interesting to study the dual graph of an outer-planar graph, which has a special structure, especially for maximal outer-planar graphs.

Lemma 17.4 *Let G be a maximal outer-planar graph. Then $G^* = \{v^*\} \cup T$, where v^* is the vertex representing the outer-face, and T is a tree with maximum degree 3.*

Proof: Recall that every vertex of G is incident to the outer-face. This means that in G^*, every face is incident to v^*, the vertex corresponding to the outer-face. Let C be an arbitrary cycle in G^*. Then there is some face F inside C (after picking the outer-face of G^* arbitrarily), which means that v^* is either inside or on C. Likewise there is some face F outside C, which means that v^* is either outside or on C. This is possible only if v^* is on C. So for every cycle C, vertex v^* is on it. Therefore, $G^* - v^*$ is a forest.

G has $2n-3$ edges, hence $n-1$ faces by Euler's formula. G^* hence has $n-1$ vertices and $2n-3$ edges of which n are incident to v^*. So the forest has $n-2$ vertices and $n-3$ edges, which proves that it is in fact a tree. Finally, since all faces except the outer-face are triangles, all vertices except v^* have degree 3, so all vertices in the tree have degree at most 3. □

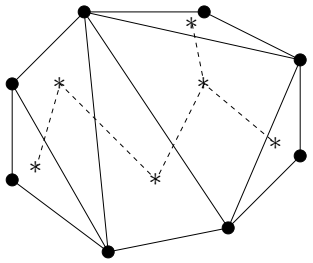


Figure 17.3: The tree in the dual of a maximal outerplanar graph.

For outerplanar graphs that are not maximal outerplanar, we can analyze the dual graph similarly. If we delete the vertex corresponding to the outer-face from the dual graph, then we again must get a forest. If the primal graph is biconnected, then the forest must be a tree (though it may have vertices of higher degree.)

Now we relate outer-planar graphs to some graph classes we've seen.

Lemma 17.5 *Every maximal outer-planar graph G with $n \geq 3$ vertices has a vertex v of degree 2. Moreover, the neighbours of v are adjacent.*

Proof: The claim clearly holds for $n = 3$, since then G is a triangle. So let $n \geq 4$, consider the dual graph of G, and write it as $T \cup v^*$ with T a tree. By $n \geq 4$, T has at least two vertices. So T contains a leaf, i.e., a node of degree 1. This node corresponds to an interior face of G for which two incident edges belong to the outer-face. Let v be the vertex between these two edges. Then $\deg(v) = 2$, and the neighbours of v must be adjacent since this is the third edge of the face. See Figure 17.4. □

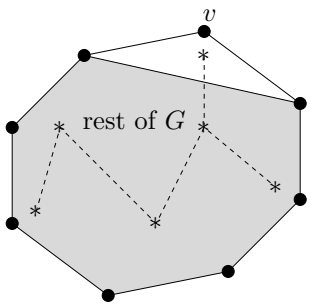


Figure 17.4: A leaf of the dual tree corresponds to a vertex of degree 2.

So if we have a maximal outerplanar graph, and v is such a vertex of degree 2, then after removing v from G we still have an outer-planar graph, and it must be a maximal outer-planar graph, since we have removed one vertex and two edges, and hence $m' = 2n' - 3$ in the new graph as well. So we can find another such vertex. And another such vertex. And so. This gives an interesting vertex order (where the vertex removed first is placed last.)

Lemma 17.6 *Every maximal outer-planar graph has a vertex order $v_1, \ldots, v_n$ such that for each $i \geq 3$, vertex v_i has exactly two predecessors, and they are adjacent.*

We have actually seen this kind of vertex order: It is a perfect elimination order, since the predecessors of each vertex are adjacent. (This also holds for v_1, v_2 trivially.) Moreover, setting $k = 2$, we have $\text{indeg}(v_i) = k$ for all $i > k$. We had a special name for graphs that have a perfect elimination order such that $\text{indeg}(v_i) = k$ for all $i > k$: they are the k-trees. So every maximal outerplanar graph is a 2-tree. Every outerplanar graph therefore is a partial 2-tree (since it can be made maximal outerplanar by adding edges.) And partial 2-trees are exactly SP-graphs, so every outerplanar graph is an SP-graph.

Theorem 17.7 *Every outerplanar graph is an SP-graph, and in particular, has treewidth at most 2.*

The reverse does not hold: there are SP-graphs that are not outerplanar. One example is $K_{2,3}$, see Figure 17.5. Assume $K_{2,3}$ is outerplanar, and fix some embedding in which all vertices are on the outer-face. Then we could add a universal vertex (i.e., a vertex connected

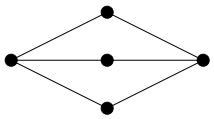


Figure 17.5: $K_{2,3}$ is an SP-graph, but not outerplanar.

to all other vertex) in the outer-face, and the resulting graph would still be planar. But then we get a planar drawing of $K_{3,3}$, a contradiction.

The above argument holds in general: G is outer-planar if and only if we can add a universal vertex to G and the remaining graph is planar. This yields immediately an algorithm to test whether a given graph is outer-planar, and its runtime is the same as the runtime for planarity testing, i.e., linear.

17.3 k-outerplanar graphs

The class of outer-planar graphs is a very restricted class, since all vertices must be on the outer-face. We now relax this, and only demand that all vertices should be "close" to the outer-face.

Definition 17.8 *Let G be a planar graph with a fixed planar embedding and outer-face. Then the* ith onion peel *is defined recursively as follows:*

- *The first onion peel L_1 is the set of all vertices on the outer-face of G.*
- *For $i > 1$, the ith onion peel is the set of vertices on the outer-face of $G - L_1 - \cdots - L_{i-1}$ in the planar embedding and outer-face induced by G.*

The name "onion peel" is motivated by the idea of peeling the vertices of G off in layers like one would peel an onion. A graph G is called *k-outerplanar* if for some planar embedding and choice of outer-face of G, all its vertices are contained within the first k onion peels. Figure 17.6 illustrates this concept.

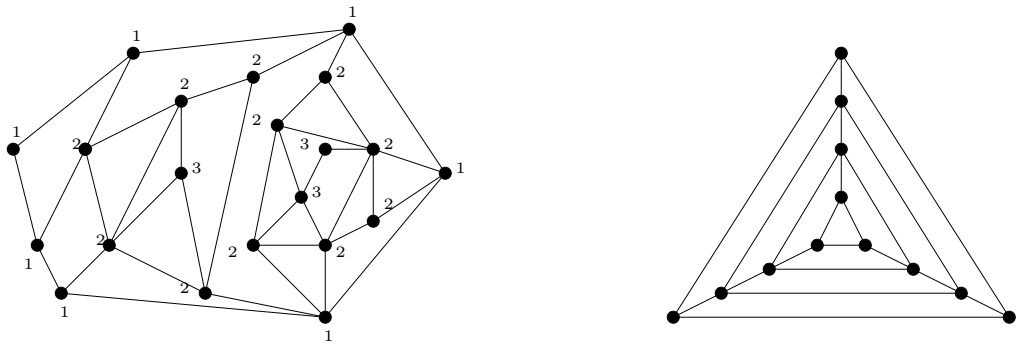


Figure 17.6: Example of a graph with 3 onion peels and a graph with $n/3$ onion peels (in this planar embedding.)

Since the outer-face always contains at least 3 vertices, every onion peel contains at least 3 vertices, and any planar graph is at most $n/3$-outerplanar. It is not known whether this is tight for some planar graphs. The right graph in Figure 17.6 seems to come close, but if we choose the outer-face differently, then it is only $n/6$-outerplanar. (One can show that we cannot do better than $n/6$ for this graph.)

17.3.1 Recognizing k-outerplanar graphs

Testing whether a graph is k-outer-planar is more difficult than testing whether a graph is outerplanar. First, recall that the definition of k-outer-planar states that among all possible planar embeddings, there must be one that has at most k onion peels. For a given planar embedding and outer-face, this is very easy to test – just compute all onion peels and count how many there are. But if the planar embedding is not given, then we effectively must try all planar embeddings. This is not as daunting as it may sound, because there exist good data structures to store all planar embeddings, by only storing the places where a planar embedding can be changed. In particular, testing whether a graph is k-outerplanar can be done in $O(k^3n^2)$ time [BM90].

17.3.2 Relationship to other graphs classes

By definition any outerplanar graph is 1-outerplanar, and hence belongs to the class of k-outerplanar graphs for some constant k. There is no relationships between k-outerplanar graphs (for constant k) and SP-graphs. K_4 is a 2-outerplanar graph, but not an SP-graph. On the other hand, one can construct a graph class $H_0, H_1, H_2, \ldots$ such that H_i is a 2-terminal SP-graph, has $2 \cdot 3^i - 1$ vertices and is i-outerplanar, but not $(i-1)$-outerplanar. See Figure 17.7. Since $i = \log_3((n+1)/2)$, therefore H_i is not k-outerplanar for constant k.

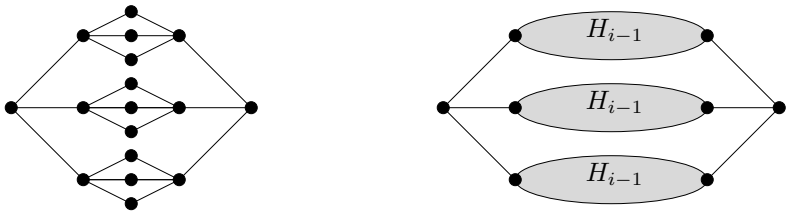


Figure 17.7: A 2-terminal SP-graph that is not k-outerplanar for constant k. We show H_2, and the general construction of H_i using three copies of H_{i-1}.

So we get the following relationships between subclasses of planar graphs. All containments are strict.

17.3.3 More on onion peels

The concept of onion peels is also useful for graphs that are not k-outerplanar, both for making them k-outerplanar and for approximation algorithms. We first need an easy observation:

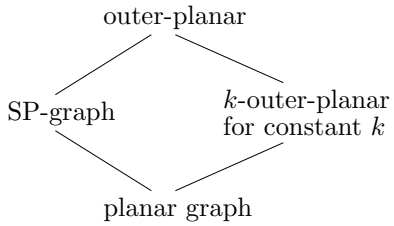


Figure 17.8: Relationships between subclasses of planar graphs.

Lemma 17.9 *Any edge of a planar graph connects two vertices that are either in the same onion peel or in two adjacent onion peels.*

Proof: Assume (v, w) is an edge for which not both v and w are in the same onion peel. Assume v is removed first, say $v \in L_i$. This must bring w to the outer-face of $G - L_1 - \cdots - L_i$ by planarity, so $w \in L_{i+1}$. □

Now we bundle the onion peels into k groups. More precisely, for $1 \leq i \leq k$, set

$$V_i = L_i \cup L_{(k+1)+i} \cup L_{(2k+2)+i} \cup \ldots,$$

i.e., V_i consists of all those onion peels for which the index equals i modulo $k + 1$. Clearly, V is the disjoint union of $V_1, \ldots, V_k$.

Set $G_i = V - V_i$, i.e., remove every $(k + 1)$st onion peel from G. By Lemma 17.9, G_i thus splits into connected components, each of which resides within at most k onion peels. Therefore, each connected component of G_i is k-outer-planar, and so is G_i.

Theorem 17.10 *Any planar graph can be made k-outerplanar by removing at most n/k vertices.*

Proof: Let i be such that $|V_i|$ is minimal; then $|V_i| \leq n/k$ since V is the disjoint union of $V_1, \ldots, V_k$. Removing the vertices of V_i gives G_i, which is k-outerplanar. □

17.3.4 Approximation algorithms

Onion peels can also be used to obtain approximation algorithms for NP-hard problems in planar graphs. (This was in fact the motivation for studying k-outerplanar graphs, see Baker [Bak94].) To see this, we first need to show that such problems are polynomial in k-outerplanar graphs. Baker did this by giving direct (and neither pretty nor fast) dynamic programming algorithms. However, it was discovered later that an easier argument can be provided, by reducing k-outerplanar graphs to some old friends of ours.

Theorem 17.11 *Every k-outerplanar graph G has treewidth at most $3k - 1$.*

The proof of this theorem is lengthy and delayed to Section 17.4. Using this, we now obtain approximation algorithms. We demonstrate this for independent set.

Theorem 17.12 *Let G be a planar graph and let k be a constant. Then we can find in time $O(k \cdot f(k)n)$ an independent set of size at least $\frac{k-1}{k}\alpha(G)$, where $f(k)$ is a function that depends on k but not on n.*

Proof: Define V_i and $G_i = G - V_i$ as before; then G_i is a k-outerplanar graph and has treewidth at most $3k-1$, so we can find the maximum independent set in G_i in $O(f(k)n)$ time for some function k. Let I_i be the maximum independent set of G_i and let I be the largest independent set among $I_1, \ldots, I_k$.

We claim that I is the desired independent set. To see this, let I^* be a maximum independent set of G, so $|I^*| = \alpha(G)$. For $i = 1, \ldots, k$, let $I_i^* = I^* \cap V_i$, so I^* is the disjoint union of $I_1^*, \ldots, I_k^*$. Let i be such that $|I_i^*|$ is minimized, then $|I_i^*| \leq |I^*|/k$.

But $I^* - I_i^*$ is an independent set in $G_i = G - V_i$, so $|I^* - I_i^*| \leq |I_i| \leq |I|$. Therefore

$$|I| \geq |I_i| \geq |I^* - I_i^*| \geq |I^*| - \frac{1}{k}|I^*| = \frac{k-1}{k}|I^*| = \frac{k-1}{k}\alpha(G)$$

as desired. □

In particular, this theorem implies that we can "almost" solve independent set in planar graphs. More precisely, by setting k sufficiently large, we can get arbitrarily close to the maximum independent set. (For those that know these terms: we have an ε-approximation scheme for the Independent Set problem in planar graphs.)

The one given here (which was first presented by Baker [Bak94]) was actually not the first such approximation scheme. Lipton and Tarjan showed already in 1979 that every planar graph has a weighted separator (remember weighted separators?) of size at most $2\sqrt{2}\sqrt{n}$, and proved that this can be used for another approximation scheme for independent set in planar graphs [LT79]. However, their algorithm is both more difficult to explain and slower than Baker's algorithm, so we will not give details here.

17.4 Proof of Theorem 17.11

In this section, we prove Theorem 17.11, i.e., we show that every k-outerplanar graph has treewidth at most $3k-1$. To keep things simple, we will show a slightly weaker statement, and only prove that the treewidth is at most $4k+2$. The proof of this theorem breaks down into three steps.

- Reduce the problem to graphs of maximum degree 3. Thus, show that there exists a k-outerplanar graph G' that has maximum degree 3 such that treewidth$(G) \leq$treewidth(G').
- Show that every k-outerplanar graph with maximum degree 3 has a spanning tree with load at most $2k$. (We will define load formally later.)
- Show that if a graph with maximum degree 3 has a spanning tree with load ℓ, then it has treewidth at most $2\ell + 2$.

17.4.1 Achieving maximum degree 3

So the first step is to modify G into a graph G' that has maximum degree 3, is k-outerplanar and treewidth$(G) \leq$treewidth(G').

We will first show how to modify one vertex of high degree. Assume that G is a k-outerplanar graph, and v is a vertex in G that has $\deg(v) \geq 4$. Then we replace v by a chain $C(v)$ of $\deg(v) - 2$ vertices. Each endpoint of the chain is incident to two edges that used to be incident to v, and all other vertices of the chain are incident to one edge that used to be incident to v. See Figure 17.9 for an illustration.

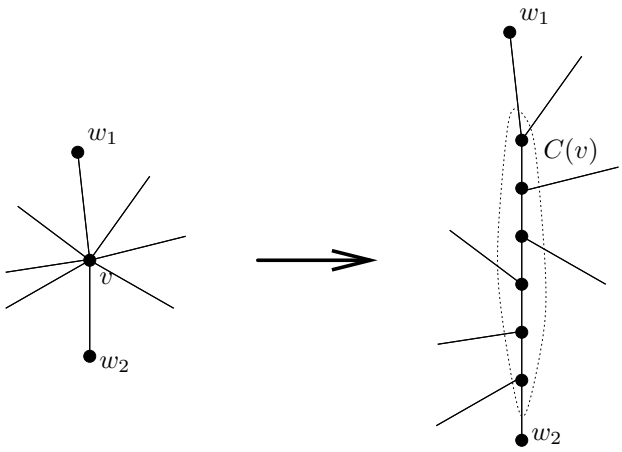


Figure 17.9: Replacing v by a chain $C(v)$ of vertices.

Incidentally, this replacement of a vertex of high degree by a chain of vertices of degree 3 is a common technique, also used in Graph Drawing, or for showing that problems remain NP-complete even in graphs with small maximum degree. In order for it to do the job for k-outerplanar graphs, we need to be careful in which edges we assign where along the chain. We need the following two rules:

- Since the graph was planar, there was a fixed ordering of the edges around vertex v. Maintain this ordering while splitting vertex v into a chain of vertices.
- Choose edges incident to the ends of the chain in a special way. More precisely, consider the time when we had peeled away just enough layers from the graph such that v is on the outer-face. Then let w_1 and w_2 be two neighbours of v on the outer-face of the current graph. Assign the edges to the chain in such a way that the edges (w_1, v) and (w_2, v) are incident to the two ends of the chain.

Let G' be the graph that results from G by replacing all vertices of degree at least four in such a fashion. Clearly G' has maximum degree 3. It is also not hard to see that G' is again k-outerplanar, because the layer that used to contain v now contains $C(v)$ (by our choice of w_1 and w_2 at the ends of the chain), and hence all vertices of $C(v)$ are deleted at the same time that we would have deleted v.

So all that remains to show is that treewidth$(G) \leq$treewidth(G'). To show this, observe that G can be obtained from G' by contracting all edges between vertices in $C(v)$ (for every vertex v). Since contracting edges does not decrease the treewidth, the result follows.

17.4.2 Spanning trees of small load

Assume that G is a connected graph, and T is a spanning tree of G. Let e be an edge in the tree. Then the *load* of the tree-edge e is the number of non-tree edges (x, y) in G such that the unique path between x and y in T uses edge e. See Figure 17.10 for an example. (In some references, the load is also called the *remember number*.) The *load* of tree T is the maximum load of an edge.

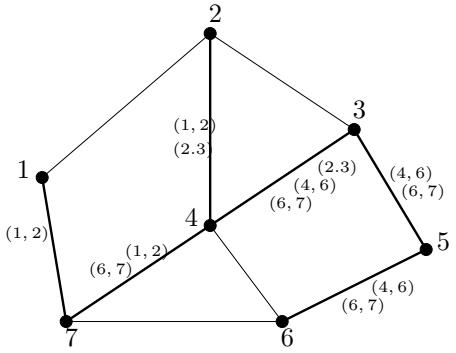


Figure 17.10: A spanning tree with load 3. We indicate the non-tree edges that contribute to the load of each tree-edge at the tree-edge.

For the induction hypothesis to work out properly in the proof to come, we need to allow the graph to become disconnected. For this reasons, we will be working with spanning forests (i.e., a forest that is a spanning tree in each connected component). The definition of load is the same for spanning forests.

Lemma 17.13 *Every k-outerplanar graph G with maximum degree at most 3 has a spanning forest of load at most $2k$.*

We prove this lemma only if G is connected; if G is not connected then we can obtain such a spanning forest by combining the spanning trees for all connected components; this does not change the load.

So assume from now on that G is connected. We proceed by induction on k. If $k = 1$, then G is outerplanar. Since G has maximum degree 3, no vertex v can be incident to more than one interior edge, because if v is incident to any interior edge, then it must be incident to at least two vertices on the outer-face. Therefore the interior edges form a matching, and in particular a forest. Let T be a spanning tree that contains all interior edges, as well as enough edges on the outer-face to make it connected. We claim that this tree has load at most 2. Since we will use this result again in the induction step, we state it as a separate claim.

Claim 17.14 *Let G be a planar graph with a fixed planar embedding, and let T be a spanning tree of G such that all non-tree edges are on the outer-face of G. Then T has load 2.*

Proof: First observe that every interior face F of G can contain at most one non-tree edge. For if it contained two non-tree edges, then F would have at least two edges (u, w) and

(x, y) in common with the outer-face ($w = x$ is possible). In consequence, removing edges (u, w) and (x, y) from the graph would disconnect the graph. Since T is a spanning tree, it therefore must contain one of these two edges, which contradicts that they are non-tree edges. See also Figure 17.11.

Now let (v, w) be a non-tree edge, and let F be the interior face incident to (v, w). (There must be such a face; otherwise (v, w) would be a bridge of the graph and any bridge must be in a spanning tree.) Consider the simple path from v to w along the boundary of F that doesn't use (v, w). This path cannot contain a non-tree edge (for (v, w) is the only non-tree edge on F), and hence is a path from x to y in T. Since there is only one such path, edge (v, w) contributes at most one to the load of every tree-edge on F.

Now consider an arbitrary tree-edge e. It is incident to at most two interior faces. Each interior face is incident to at most one non-tree edge, so each interior face incident to e contributes at most one to the load of e. So the load of e is at most two as desired. □

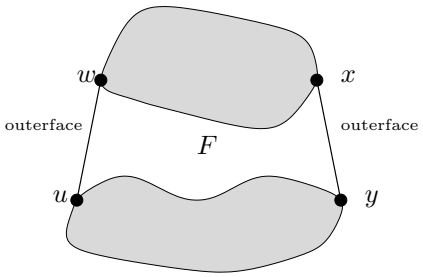


Figure 17.11: Two non-tree edges on one interior face would cut the graph into two disjoint parts.

Using this claim, we are done with the base case $k = 1$, since we found a spanning tree that has load at most $2 = 2k$.

Now for the induction step. Let G be a k-outerplanar graph, $k \geq 2$, and let G' be the graph that results from G by deleting all edges on the outer-face. G' need not be connected. We claim that G' is $(k-1)$-outerplanar. Note that if v is a vertex on the outer-face of G, then as before v is incident to at most one interior edge of G, and therefore has degree zero or one in G'. Thus, G' consists of a $(k-1)$-outerplanar graph (the graph that would have resulted from deleting all *vertices* (as opposed to edges) from the outer-face of G) plus a collection of vertices of degree zero or one. One can easily see that such a graph is $(k-1)$-outerplanar.

By induction, G' contains a spanning forest T' that has load at most $2(k-1) = 2k-2$. We will now show how to expand T' into a spanning tree T of G that has at most 2 more units of load.

Let H be the graph formed by taking all edges in T' and adding the edges on the outer-face of G to it. Since G was connected and T' is a spanning forest, H is also connected. Let T be a spanning tree of H that contains all edges of T', i.e., add edges on the outer-face of G to T' until it is connected.

Notice that T is a spanning tree of graph H for which all non-tree edges are on the outer-face. By Claim 17.14, therefore the load of T (with respect to the non-tree edges in H) is at most 2. But T contains T', and all edges in T' had load (with respect to the edges

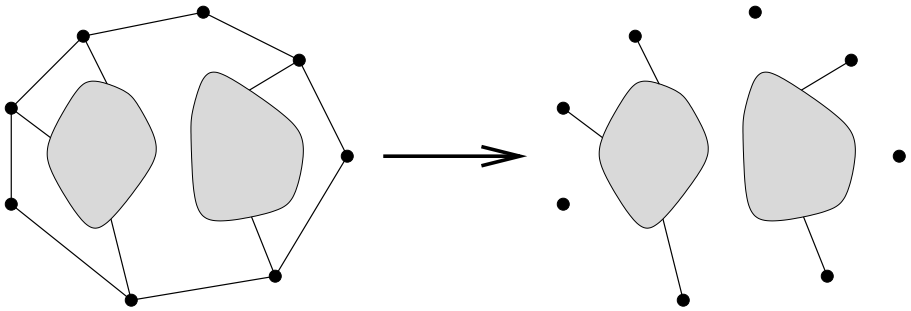


Figure 17.12: Obtaining G' from G.

in G') at most $2k - 2$. Therefore, all edges in T have load at most $2 + 2k - 2 = 2k$ with respect to the edges in $G = G' \cup H$, which proves the claim, and therefore Lemma 17.13 and the second step.

17.4.3 From load to treewidth

The last step is to convert a spanning tree with small load into a tree decomposition of a graph. More precisely, we have the following result:

Lemma 17.15 *Let G be a graph with maximum degree three, and let T be a spanning tree of G that has load ℓ. Then G has treewidth at most $2\ell + 2$.*

The proof of this lemma (which also completes the proof of Theorem 17.11) was left as a homework.

Chapter 18

Hereditary properties and the Minor Theorem

As our final topic for the book, we study a topic that brings us back to many of the graph classes that we have seen in this course: hereditary properties. We first give examples of this, and then give a deep and amazing theorem that characterizes graph classes that are minor hereditary, the so-called Graph Minor Theorem.

18.1 Hereditary properties

Often throughout these lecture notes, we have given statements such as "Every induced subgraph of an interval graph is an interval graph", "Every minor of a partial k-tree is a partial k-tree", "Every subgraph of a planar graph is a planar graph", and so on. These statements can be unified in saying that some graph class is closed under taking subgraphs (for some sense of "subgraph"). Another term for this is to say that the graph class has the *hereditary property*.

There are different ways of defining "subgraph", and depending on this we get different meanings of hereditary properties.[1] Let G be a graph and let G' be another graph. We consider three ways of how G' could be obtained from G:

- We delete vertices from G to get G'. Then G' is an *induced subgraph* of G.

  A graph class is called *induced hereditary* if it is closed under taking induced subgraphs.

- We delete vertices and edges from G to get G'. Then G' is a *subgraph* of G.

  A graph class is called *hereditary* if it is closed under taking induced subgraphs.

- We delete vertices and edges from G, and contract edges from G, to get G'. Then G' is a *minor* of G.[2]

[1] The following definitions are not standard; most references simply use "hereditary" and it is clear from the context which meaning of "hereditary" should be used.

[2] Assuming that G is connected, we could omit the operation of deleting vertices and get the same sets of graphs as minors.

A graph class is called *minor hereditary* if it is closed under taking minors.

Any minor hereditary graph class is also hereditary, and any hereditary graph class is also induced hereditary. Almost all graph classes that we have seen are induced hereditary (the sole exception are 2-terminal SP-graphs), and many are minor hereditary as well. For example:

- Interval graphs are induced hereditary, since deleting a vertex amounts to deleting its corresponding interval, and the remaining intervals form a representation for the remaining subgraph.

  Interval graphs are not hereditary. To see this, note that K_n is an interval graph for any n. If interval graphs were hereditary, then therefore *all* simple graphs would be interval graphs, which is clearly not the case. In general, any graph class that contains K_n for all n is not hereditary (unless it's the trivial graph class of "all graphs", which is not very interesting.)

  Since interval graphs are not hereditary, they are also not minor hereditary. However, note that interval graphs are actually closed under contracting edges (to contract edge (v, w), unify the intervals of v and w.)[3]

- Any intersection graph of some type of object is induced hereditary, since deleting a vertex corresponds to removing its object. In particular, chordal graphs, permutation graphs, circular-arc graphs, boxicity-k graphs, t-interval graphs, and various others are all induced hereditary.

- Some other friends of interval graphs, for example comparability graphs and perfect graphs, are also easily shown to be induced hereditary.

- Partial k-trees are closed under taking minors, and hence are minor hereditary. The same also holds for graphs of pathwidth at most k or proper pathwidth at most k, but not [4] for some of the other related concepts (e.g. cutwidth, domino treewidth, branchwidth.)

- Planar graphs are minor hereditary, and so are SP-graphs (but not 2-terminal SP-graphs!), outer-planar graphs and k-outer-planar graphs.

18.2 Minimal forbidden subgraphs

Assume graph class $\mathcal{C}$ is hereditary (for one of the above definitions of hereditary). Then it makes sense to look at the "smallest" graphs that do not belong to $\mathcal{C}$.

Definition 18.1 *Let $\mathcal{C}$ be α-hereditary, for $\alpha \in \{\emptyset$, induced, minor$\}$. Then a* minimal forbidden α-subgraph *of $\mathcal{C}$ is a graph G such that $G \notin \mathcal{C}$, but every α-subgraph of G belongs to $\mathcal{C}$.*

[3] There appears to be no name for subgraphs obtained by deleting vertices and contracting edges only.

[4] Or at least not obviously - I'm not sure whether it does.

On can think of the minimal forbidden α-subgraphs as follows: let $\mathcal{F}$ be the complement of $\mathcal{C}$, i.e., take all graphs that are not in $\mathcal{C}$. (This is usually an infinite set of graphs, but that's ok.) Now for each graph G in $\mathcal{F}$, check whether some α-subgraph H of G is also in $\mathcal{F}$. If it is, then delete G from $\mathcal{F}$.

The main use of this set is that it gives us an easy way of characterizing when a graph is *not* in a graph class. The following lemma follows almost immediately from the definition of minimally forbidden α-subgraph, taking into account that $\mathcal{C}$ is closed under taking α-subgraphs.

Lemma 18.2 *Let $\mathcal{C}$ be α-hereditary, for $\alpha \in \{\emptyset$, induced, minor$\}$, and let $\mathcal{F}$ be the minimal forbidden subgraphs of $\mathcal{C}$. Then G does not belong to $\mathcal{C}$ if and only if there exists some $H \in \mathcal{F}$ which is an α-subgraph of G.*

We will give some examples to illustrate this concept:

- A bipartite graph is a graph that does not contain any odd cycles. Thus, the minimal forbidden subgraphs for bipartite graphs are $C_3, C_5, C_7, \ldots,$.
- A tree is a graph that does not contain any cycles. Thus the minimal forbidden subgraphs for trees are $C_3, C_4, C_5, \ldots$.

  But actually, trees are closed under taking minors, so we can also consider minimal forbidden minors for trees. Since every cycle contains C_3, trees contain only one minimal forbidden minor: C_3.
- Chordal graphs are graphs that do not contain an induced cycle of length at least 4. Thus, the minimal forbidden induced subgraphs are $C_4, C_5, C_6, \ldots$.

  Note that chordal graphs look at lot like trees here. But there is a difference: For trees the cycles are minimal forbidden *subgraphs*, whereas for chordal graphs they are minimal forbidden *induced subgraphs*. This makes a big difference, since there are many chordal graphs that are not trees.
- By the strong perfect graph theorem, perfect graphs are exactly those graphs that do not contain an odd hole or an odd anti-hole as induced subgraph, so the odd holes and odd anti-holes are minimal forbidden induced subgraphs for perfect graphs.
- Partial 2-trees are the same as SP-graphs, and we have seen that these are the same as graphs that do not contain K_4 as a minor. So K_4 is the unique minimal forbidden minor for partial 2-trees.
- Planar graphs, by Kuratowski's theorem, are the graphs that do not contain K_5 or $K_{3,3}$ as forbidden minors. So K_5 and $K_{3,3}$ are the minimal forbidden minors for planar graphs.

Notice that in the above examples, all graph classes that are minor hereditary had a finite number of minimal forbidden minors. This is not a coincidence; we will return to this in the next section.

So far, we have looked at graph classes that we know, and found the minimal forbidden subgraphs for them. But we can actually go the other way: Given a set of graphs, we can define a graph class for which they are minimal forbidden subgraphs.

Definition 18.3 *Given a (possibly infinite) set $\mathcal{F}$ of graphs, and $\alpha \in \{\emptyset$, induced, minor$\}$, we can define a graph class $\mathcal{C}$ as all those graphs that do not contain any graph in $\mathcal{F}$ as α-subgraph. Class $\mathcal{C}$ is denoted* $\mathcal{F}$-free graphs *for induced subgraphs, $Forb_{\leq}(\mathcal{F})$ for subgraphs, and $Forb_{\preceq}(\mathcal{F})$ for minors.*

So for example, one can define *claw-free graphs* to be the graphs that do not contain $K_{1,3}$ as an induced subgraph ($K_{1,3}$ is sometimes known as a "claw".) Similarly one can define many other graph classes, for example AT-free graphs, and House-Hole-Domino-free graphs. Many of these graphs have been studied, especially with regards to their relationship to interval graphs and chordal graphs, and how to recognize them. See [Spi03] for examples.

Another result in this area that is worth mentioning relates treewidth and planarity in a very surprising way. We now study minors, and in particular, $Forb_{\preceq}(\{H\})$ for some graph H; we will often only write $Forb_{\preceq}(H)$ for this to simplify notation.

Theorem 18.4 *If H is planar, then there exists some constant k (depending on H) such that all graphs in $Forb_{\preceq}(H)$ are partial k-trees.*

Proof: (Sketch) Recall that H is a minor of the $r \times r$-grid, for some integer r that depends on H. Let k be a sufficiently huge constant depending on r. ($k = r^{4r^2(r+2)}$ should do.) One can show (but this is not at all an easy proof!) that any graph of treewidth at least k contains the $r \times r$-grid as a minor. Therefore, any graph of treewidth at least k contains also H as a minor. So all graphs in $Forb_{\preceq}(H)$ have treewidth less than k. □

This theorem has some far-reaching implications. Observe that $Forb_{\preceq}(H_1, H_2, \dots) = \bigcap_i Forb(H_i)$. So if *any* of the forbidden minors of a graph class is planar, then the graph class is a subclass of the partial k-trees for some constant k.

18.3 The Graph Minor Theorem

Next we come to the crown jewel of results regarding hereditary properties.

Theorem 18.5 (Graph Minor Theorem) *Every minor-hereditary graph class has a finite set of forbidden minors.*

This is really quite an amazing theorem. To see what it says on a specific example, consider toroidal graphs (i.e., graphs that can be drawn without crossing on the surface of a torus, or in other words, graphs that have a combinatorial embedding with genus at most 1.) One can easily show that toroidal graphs are closed under taking minors. The theorem says that there are some graphs $H_1, H_2, \dots, H_l$ such that toroidal graphs are exactly those graphs that don't contain $H_1, H_2, \dots, H_l$ as minors. So the theorem gives us a completely different way of characterizing any graph class that is closed under taking minors.

We will not prove the theorem here. The original proof (by Robertson & Seymour) was done in a number of journal articles over the span of many years and using over 500 pages (see [RS83] and [RS03] for the first and the 18th paper – I'm not sure whether the 18th paper is the last one.) Since then, many simplifications have appeared, and (apparently) the proof could now be presented in a book chapter or two (see for example [MT01].)

So why is this theorem so amazing? There are two reasons. One is its proof, which created many other interesting results and concepts in its process. For example, while the concept of treewidth actually had been known a lot longer (under different names, such as the node-searching number or pursuit-evasion [Par78]), its relevance for forbidden minors (see Theorem 18.4, which is an important ingredient in the proof of the minor theorem), and its usefulness in developing polynomial algorithms was not recognized until Robertson and Seymour "re-surfaced" the concept.

The second reason why this theorem is so amazing is at the same time very wonderful and very frustrating. As part of the proof of the graph minor theorem, one can get the following result:

Theorem 18.6 *Let H be a graph. For any n-vertex graph G, we can test whether $H \preceq G$ in $O(n^3)$ time.*

Therefore, if $\mathcal{C}$ is a graph classes that is closed under minors, then by the graph minor theorem, there is a finite set of graphs $H_1, \ldots, H_l$ such that $\mathcal{C} = \mathit{Forb}(H_1, \ldots, H_l)$. So to test whether a graph G belongs to $\mathcal{C}$, we should test for each H_i whether $H_i \preceq G$; then G belongs to $\mathcal{C}$ if and only if the answer is negative for all $i = 1, \ldots, l$.

For each H_i, this test takes $O(n^3)$ time. Therefore, this algorithm runs in $O(ln^3)$ time total. Note that l, the number of forbidden minors, depends on the graph class and *not* on the number of vertices of the graph G to be tested. Therefore, we can treat l as a (very very large) constant, and there exists an $O(n^3)$ algorithm to test whether a given graph G belongs to graph class $\mathcal{C}$.

Corollary 18.7 *Let $\mathcal{C}$ be a minor-hereditary graph class. Then testing whether a graph G belongs to $\mathcal{C}$ can be done in $O(n^3)$ time.*

Thus, the graph minor theorem shows that for *any* minor-hereditary graph class, recognizing graphs that belong to it is polynomial. That's the wonderful part. The frustrating part is that we don't have a clue as to what this algorithm actually is. In particular, for most graph classes the set (or even the number) of forbidden minors is unknown. So while we do know that a polynomial time algorithm exists, we cannot actually execute it. Annoying, isn't it?

Conclusion

So, where are we going from here? Is this all about graph algorithms? Or at least planar graph algorithms?

Far from it. I could easily fill another graduate course (and I might, in about two years) with similar topics. Hamiltonian cycles in planar 4-connected graphs. Circle packing theorem. Edge coloring in planar graphs. Triconnectivity test. Soooo many things come to mind that I would love to present, but didn't have the time for.

But, these lecture notes are not supposed to be more than 200 pages, and at some point I should do something else than writing them, so I'll stop here.

Appendix A

Background

In this chapter, we review basic graph definitions. Then we recall algorithmic aspects, in particular how to store graphs and how to find vertices of small degrees efficiently. The material in this chapter is taken from a variety of sources, among others [CLRS00, Gib85, NC88].

A.1 Graph definitions

A *graph* $G = (V, E)$ consists of a set V of *vertices* and and a set E of *edges*. Each edge e is a tuple from $V \times V$, i.e., $e = (v, w)$ with $v, w, \in V$.

In terms of notation, the variables v, w, u, x, and y are usually used for vertices. Edges are usually represented by $e, e', e_1, \ldots$. The number of vertices is $|V| = n$, and $|E| = m$ is the number of edges. If the graph in question is not clear from the context, we write $V(G), E(G), n(G)$ and $m(G)$ instead.

Cycles and paths

A *path* in a graph is a sequence $v_1, e_1, v_2, e_2, \ldots, v_k$ of alternatingly vertices and edges, beginning and ending at vertices, such that the endpoints of any edge in the sequence are exactly the vertex before and after it. Whenever the used edges are clear from the vertices alone, we will omit the edges, and just denote the path as $v_1, v_2, \ldots, v_k$. We say that the path *connects* the vertices v_1 and v_k. The *length* of a path is the number of edges on it, which is $k - 1$ in this example.

A path is called a *simple path* if no vertex appears on it twice. It is easy to show that if there is a path connecting v and w, $v \neq w$, then there is also a simple path connecting v and w.

A *cycle* is a path that begins and ends at the same vertex. Sometimes we write *k-cycle* for a cycle with k edges on it. A 3-cycle is also called a *triangle*.

A *simple cycle* is a cycle on which every vertex appears only once (we consider the begin and endpoint, which are the same, as only one appearance). If we want to emphasize that a cycle may or may not be simple, we will use the term *circuit*, rather than cycle.

Multigraphs

An edge e may be a *loop*, which means $e = (v, v)$, $v \in V$. Multiple edges are also allowed: an edge $e = (v, w)$ is a *multi-edge* if there exists some other edge $e' \in E$ such that $e' = (v, w)$. If there are exactly k edges (v, w), then (v, w) is called a *k-fold edges*. For $k = 1, 2, 3$, (v, w) is called a *simple, double* and *triple edge*, respectively. A *simple* graph is a graph that contains no loops or multiple edges. If we want to emphasize that a graph G may or may not be simple, we will use the term *multigraph* for it.

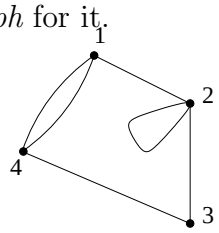


Figure A.1: A graph, with loops and multiple edges.

There are times when multiple edges and loops unnecessarily complicate matters. A straightforward solution, often employed, is to use the *underlying simple graph* G', which is obtained by deleting every edge from G that makes G non-simple. So remove all loops, and replace each k-fold edge (v, w) with $k > 1$ by exactly one edge (v, w). We will show how to do this efficiently in Section A.3.1.

Degrees

The degree of a vertex v is the number of times that v is an endpoint of an edge; it is denoted as *deg(v)*. Every loop counts twice towards the degree of its vertex. For example, every vertex in Figure A.1 has degree 4. We write $\deg_G(v)$ for the degree of vertex v in graph G if the graph is not clear from the context. A vertex is called *even* if it has an even degree, and *odd* otherwise. A graph is called *k-regular* if every vertex has degree k.

Various things can be said about the degrees of vertices; two of the most useful observations are stated here.

Lemma A.1 *For any graph,* $2m = \sum_{v \in V} \deg(v)$.

Proof: Every edge has two endpoints, and thus counts twice to the degree of some vertex. □

Lemma A.2 *For any graph, the number of odd vertices is even.*

Proof: Consider that

$$2m = \sum_{v \in V} deg(v) = \sum_{\substack{v \in V \\ v \text{ even}}} deg(v) + \sum_{\substack{v \in V \\ v \text{ odd}}} (deg(v) - 1) + \#\{\text{odd vertices}\}$$

Since the left-hand side is an even number, and all entries in the sums on the right-hand side are even numbers, the number of odd vertices must also be even. □

Special graph classes

The following special graph classes will be needed occasionally.

- A *forest* is a graph without a cycle.
- A *tree* is a graph that is a forest and that has exactly $n-1$ edges. Many properties of trees are known, we will come back to this in Section B.1.3. If we want to emphasize that the tree is not rooted (see later), we call it a *free tree*.
- A *Eulerian* graph is a graph that has a *Eulerian circuit*, i.e., a cycle (not necessarily simple) that passes through every edge exactly once. It is known that a graph is Eulerian if and only if all vertices have even degree (see for example [PS82]).
- The *complete graph* with n vertices, denoted K_n, is the graph with n vertices and all possible edges that can exist such that the graph remains simple. Thus, any pair of vertices is connected by exactly one edge, which in particular means that K_n has $\binom{n}{2}$ edges.
- A *bipartite graph* is a graph for which the vertices can be partitioned into sets A and B (i.e., $V = A \cup B$) such that any edge connects a vertex in A with a vertex in B. Since any cycle in a bipartite graph must alternate between vertices in A and B, any cycle in a bipartite graph has even length. The converse also holds: any graph for which all cycles have even length is bipartite.
- The *complete bipartite graph* on n_1+n_2 vertices, denoted K_{n_1,n_2}, is the simple bipartite graph with $|A| = n_1$ and $|B| = n_2$ that has as many edges as possible. Thus, any vertex in A is connected to any vertex in B by exactly one edge, which in particular means that K_{n_1,n_2} has $n_1 \cdot n_2$ edges.

Directed graphs

Sometimes, the edges have a sense of where they are coming from and where they are going to. We then call the graph a *directed graph* or *digraph*. A directed edge is denoted as $e = v \to w$, where w is called the *head* and v the *tail* of e. We denote by $\text{indeg}(v)$ the number of edges for which v is the head; this is called the *in-degree*, and these edges are called *incoming at* v. We denote by $\text{outdeg}(v)$ the number of edges for which v is the tail; this is called the *out-degree*, and these edges are called *outgoing at* v. We write $\text{indeg}_G(v)$ and $\text{outdeg}_G(v)$ if the graph is not clear from the context.

Lemma A.3 *For any digraph,* $m = \sum_{v \in V} \text{indeg}(v) = \sum_{v \in V} \text{outdeg}(v)$.

Proof: Every edge has exactly one head and exactly one tail, hence counts exactly once to an in-degree, and exactly once to an out-degree. □

A.1.1 Rooted trees

One special class of directed graphs are rooted trees, which is a tree in which one vertex (the *root*) is distinguished from the others, and all edges are directed away from the root.

A directed edge then leads from a *parent* to the *child*. Vertex v is called an *ancestor* of vertex w (and vertex w is called a *descendant* of vertex v) if there exists a directed path from v to w.

Given a vertex v in a rooted tree, the *subtree rooted at* v is the tree induced by v and all descendants of v.

A.2 Operations on graphs

Subgraphs

Let $G = (V, E)$ be a graph. A *subgraph* of G is a graph $G' = (V', E')$ such that $V' \subseteq V$ and $E' \subseteq E$. Since G' is a graph, all edges in E' must have both endpoints in V'.

An *induced subgraph* of G is a subgraph where all possible edges are taken, so a subgraph $G' = (V', E')$ where E' are all those edges in G for which both endpoints are in V'. Another way to look at it is to say that an induced subgraph is one that has been obtained from the original graph by deleting some vertices (and their incident edges.) An induced subgraph can be specified by just giving its vertex set, and we say that G' is the graph *induced by* V'.

We can also specify a subgraph by giving an edge set. If E' is a set of edges, then the graph *formed by* E' is the graph whose vertices are the endpoints of edges in E', and whose edges are exactly E'.

Deletion, contraction, subdivision

Various operations can be performed on a graph $G = (V, E)$ that result in another graph $G' = (V', E')$. We mentioned a few of them here:

- Deletion of an edge e: This simply means that e is removed from the edge set.

  If G' results from G by deleting the edges in E', then we denote $G' = G - E'$.

- Deletion of a vertex v: This means that first we delete all incident edges of v, and then the vertex v itself.

  If G' results from G by deleting the vertices in V', then we denote $G' = G - V'$.

- Contraction of two vertices v and w: This means that we create one new vertex x. Then, for any edge (u, v) incident to v, we delete this edge, and add a new edge (u, x). For any edge (w, y) incident to w, we delete this edge and add a new edge (x, y). Finally, we delete vertices v and w.

  If G' results from G by contracting the vertices in V', then we denote $G' = G \backslash V'$. Even if G was simple, G' might very well have loops and multiple edges. See Figure A.2.

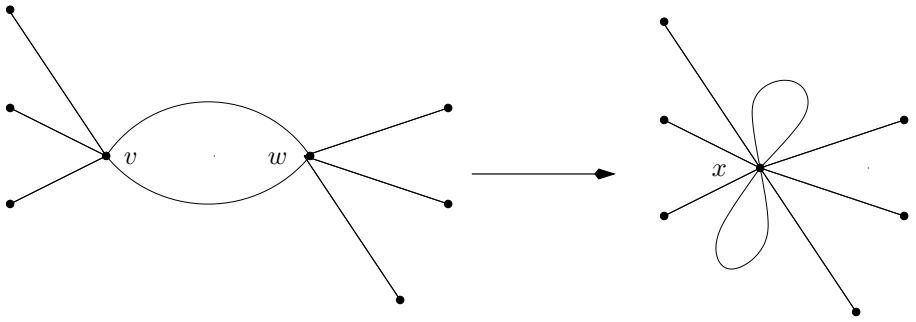


Figure A.2: The contraction of $\{v, w\}$ creates multiple edges and loops.

- Contraction of an edge $e = (v, w)$: This means that we delete the edge e, and then contract v and w as described above.

  If G' results from G by contracting the edges in E', then we denote $G' = G \backslash E'$.

- Removing a vertex v of degree 2: Let (u, v) and (v, w) be the two incident edges of v. This operation means that we delete (u, v) and (v, w), and add the edge (u, w). Then we delete v.

- Subdividing an edge $e = (u, w)$: This operation is the reverse of the previous operation. We add a new vertex v, add edges (u, v) and (v, w), and delete the edge (u, w).

- A graph G is called a *subdivision* of a graph H, if G can be obtained from H by a number of edge subdivisions.

A.3 Algorithmic aspects of graphs

There is a variety of ways how graphs can be stored. We review here only one, which is probably the most complicated, but in exchange also the most versatile, in that it also supports multiple edges and loops, makes it easily possible to store extra information with vertices and edges, allows fast graph updates and uses only linear space. Namely, we use incidence lists, i.e., every vertex has a list of incident edges. More precisely:

- Every vertex has a list of incident edges. (Without further mentioning, we assume that any list is a doubly-linked lists with pointers to the first and last element.)

- Every vertex v knows its degree $\deg(v)$.

- Every edge $e = (v, w)$ knows its endpoints $v, w \in V$, and where e is referred to in the incidence lists of v and w.

See Figure A.3 for an example of this data structure.
With this data structures, the following holds:

- Insertion or deletion of vertices or edges can be done in constant ($O(1)$) time.

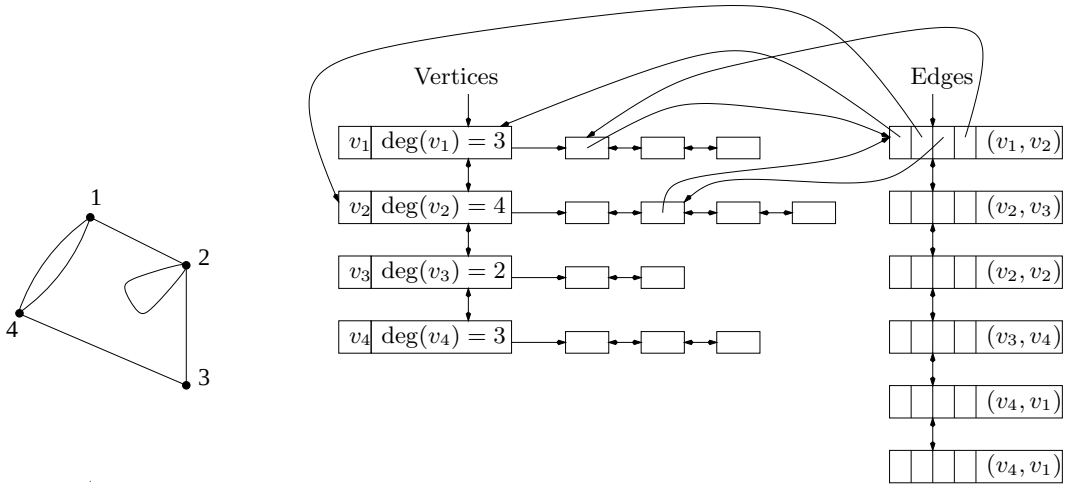


Figure A.3: A very simple graph, and excerpts of the data structure to store it.

- The adjacency query "is v adjacent to w" can be answered in $O(\min\{deg(v), deg(w)\})$ time, by performing a linear search through the incidence list of whichever of the two vertices has the smaller degree.
- Contracting two vertices v, w takes $O(\min\{deg(v), deg(w)\})$ time, by re-connecting the edges at the vertex with the smaller degree to the other vertex.

Some not-so-obvious claims will be shown in the next subsections.

A.3.1 Computing the underlying simple graph

We claim that the underlying simple graph can be computed in linear ($O(n+m)$) time. This is obvious for removing loops: just scan the list of edges, and remove any edge that has the same endpoint twice. This takes $O(m)$ time.

To remove multiple edges, we have to work a little harder. The crucial idea is to sort the edges in such a way that any two edges that have the form (v, w) are guaranteed to be consecutive. This can be achieved with a double bucket sort (also known as *radix sort*) as follows:

Assume that loops have been removed, the vertices are numbered $1, \ldots, n$, and each edge (i, j) is stored such that $i < j$. Sort the edges with a bucket sort by their second endpoint; this takes $O(n + m)$ time because there are n endpoints. Sort the resulting list of edges again, but this time by the first endpoint. In this second sorting, we have to be careful that in case of a tie, the sorting does not change the order of the elements. This can be done with a bucket sort during which new entries are always added at the end of the bucket. This takes again $O(n + m)$ time.

The edges are now sorted by the first (the one with the smaller index) endpoint, and in case of a tie, by the second endpoint. In a list thus sorted, multiple edges must be consecutive. Thus, we scan the list, and whenever two consecutive edges connect the same pair of vertices, we remove one of them. The total time is thus $O(n + m)$.

A.3.2 Finding a vertex of minimum degree

Assume that we have a simple graph G. Finding a vertex of minimum degree in G can be done in $O(n)$, and generally not in less time. However, if we need to find the minimum-degree vertex repeatedly, then we can do better in average, even when we add or delete edges or vertices.

We proceed as follows:

- We create an array of n buckets $B[0] \dots B[n-1]$. Bucket $B[d]$ will store vertices of degree d in a list. No vertex can have degree n or more in a simple graph, so all vertices belong to a bucket.
- We have a "master" list Q, which contains the first element of each $B[i]$ that is not empty. Each element in $B[0], \dots, B[n-1]$ knows whether it belongs to Q, and if so, where it is in Q.
- Each vertex knows its entry in $B[i]$.

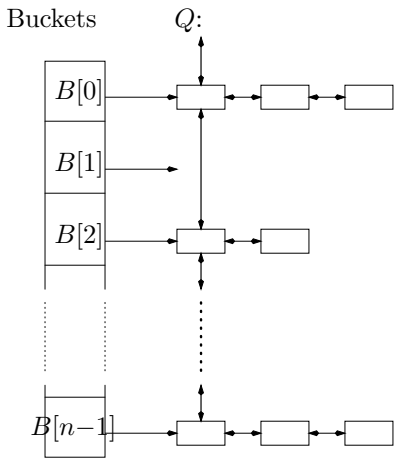


Figure A.4: The data structure to find vertices of minimum degree in constant time (in average).

Building this data structure (which happens only once) involves

- Enter vertex v in $B[\deg(v)]$. This takes $O(1)$ per vertex, $O(n)$ total.
- Go through lists $B[0] \dots B[n-1]$ to link the first elements of those buckets that are not empty into Q. This takes $O(n)$ time.

So the total time taken to initialize this data structure is linear.

Now, we can find the vertex of minimum degree in $O(1)$ time, because it is the first vertex in the master list Q. But we need to ensure that, with each change in the graph, the maintenance of this new data structure does not take more time to achieve than the change does.

- Removing a vertex v (we are not counting the time here that might be needed to remove incident edges; for those we must count an edge deletion as explained below).
  - Let d be the degree of v. Vertex v knows where it is in $B[d]$.
  - If v is not the first element of $B[d]$, simply remove v from $B[d]$.
  - If v is the first, but not the only element of $B[d]$, remove v from $B[d]$, and replace v in Q with the next element of $B[d]$.
  - If v is the first and only element of $B[d]$, remove v from both $B[d]$ and Q.

  This takes $O(1)$ time.

- Deletion of an edge $e = (v, w)$.
  - Let d be the degree of v before deletion of the edge. Vertex v knows where it is in $B[d]$.
  - Let p be the predecessor in Q of the first element of $B[d]$. (This may be be undefined.)
  - Remove v from $B[d]$ as described above.
  - If $B[d-1]$ is non-empty, insert v into $B[d-1]$ after the first element.
  - If $B[d-1]$ is empty, insert v as first element of $B[d-1]$, and insert v after p in Q (if p is defined), respectively at the beginning of Q.
  - Repeat for w.

  This takes $O(1)$ time.

- Insertion of an edge $e = (v, w)$.

  This takes $O(1)$ time using a similar reasoning as for deletion of an edge.

So the deletion of a vertex in this data structure takes $O(1)$ time and the deletion or insertion of an edge takes $O(1)$ time. Since these operations must take at least $O(1)$ time to update the graph data structure, we can look at the time taken to update our data structure as *overhead*, i.e., it doesn't change the worst-case bound, it only changes the constant of that bound.

A.3.3 Testing the existence of edges

Assume that we have a simple graph G. Testing whether an edge exists between vertex v and vertex w can be done in $O(\min\{\deg(v), \deg(w)\})$ time, and generally not in less time. However, if we need to test the existence of edges repeatedly, then we can do better in average. So assume that we are given a multi-set T of vertex-pairs. We will now show how to test whether each pair in T is indeed an edge in time $O(m + n + |T|)$.

The crucial idea is that since there are n vertices in the graph, we can label them with numbers $1, \ldots, n$, and then use radix sort to sort T in $O(n + |T|)$ time. This sorting enables us to test quickly whether each pair in T is also in the set of edges E.

The precise pseudo-code is as follows:

- Assume the vertices are number $1, \ldots, n$. This takes $O(n)$ time.
- For each edge (i, j) with $i < j$ in E, add an ordered pair $\{i, j\}$ to a list L_E. This takes $O(m)$ time.
- For each (unordered) vertex pair $\{i, j\}$ with $i < j$ in T, add ad ordered pair $\{i, j\}$ to a list L_T. This takes $O(|T|)$ time.
- Sort L_E with radix sort in $O(n + m)$ time, and L_T with radix sort in $O(n + |T|)$ time.
- Check for each element in L_T whether it is in L_E. Since both lists are sorted, this can be done in $O(m + |T|)$ time by walking through both of them in parallel.

Appendix B

Graph k-Connectivity

In this chapter, we introduce the concept of connectivity of graphs, and show how to find connected and biconnected components in linear time.

B.1 Connected graphs and components

B.1.1 Definitions

A graph is *connected* if there exists a path from any vertex to any other vertex in the graph. A *connected component* of a graph G is a maximal subgraph G' of G that is connected.

No vertex v can belong to two connected components C_1 and C_2, for otherwise if w_1, w_2 are two vertices in $C_1 \backslash C_2$ and $C_2 \backslash C_1$, then there would be a path from w_1 to v and a path from v to w_2. Combining these two paths we get a path from w_1 to w_2, so w_1 and w_2 belong to the same connected component.

Taking the union of all the connected components of a graph will result in the original graph.

B.1.2 Computing connected components

The connected components can be found easily with any graph traversal method, for example depth-first search or breadth-first search. Here, by a *graph traversal*, we mean any method that starts at an arbitrary vertex r (called the *root* of the traversal), and proceeds to visit all vertices that can be reached along a path from r. Therefore, the graph traversal will find the connected component containing r.

Presuming the graph traversal works in linear time, we can find all connected components in linear time. Namely, we scan all vertices of the graph. If the vertex in consideration has been already visited during a graph traversal, we ignore it and proceed with the next vertex in the list. If the vertex v in consideration has not yet been visited during a graph traversal, then we start a new graph traversal with v as the root. This will visit all vertices in the connected component C that contains v, and takes $O(n(C) + m(C))$ time. We can store this connected component, for example, by keeping track of the number of found components,

storing with each vertex v this number when we find v; then two vertices are in the same connected components if and only if they have the same number.

The overall time for this algorithm is $O(n)$ for scanning the vertex list, and $\sum(n(C) + m(C))$ for the graph traversals, where the sum is over all connected components. Since every vertex and edge belongs to exactly one connected component, the total time is $O(n+m)$ as desired.

B.1.3 Properties of connected graphs and trees

If a graph is connected, then it cannot have too few edges.

Lemma B.1 *Any connected graph G with $n \geq 2$ vertices has at least $n-1$ edges.*

Proof: Let r be an arbitrary vertex of the graph. For each vertex $v \neq r$, there exists a path P_v connecting r and v by connectivity. If there is more than one path, pick an arbitrary one. For the purpose of the following description, it will be convenient to think of P_v as directed from r to v.

We construct a subgraph G_T by scanning the vertices $v \neq r$. We initialize G_T as just vertex r. When we look at a vertex $v \neq r$, then we first look whether v already belongs to G_T; if so, do nothing. If v does not yet belong to G_T, then let x be the last vertex on P_v that does belong to G_T. Add all edges and vertices on P_v between x and v to G_T.

By induction on the number of vertices in G_T, one can verify the following the following properties of G_T:

- For any vertex $v \neq r$ in G_T, there exists exactly one path from r to v in G_T.
- v has exactly one incoming edge in G_T.
- r has no incoming edge in G_T.

Also, eventually all vertices will belong to G_T. Since every edge in G_T belongs (in its undirected version) also to G, we have

$$m(G) \geq m(G_T) = \sum_{v \in V} \text{indeg}_{G_T}(v) = n - 1,$$

which proves the claim. □

A lot of interesting results follow from this lemma, and in particular its proof. For example, recall that a tree is a graph without cycles with $n-1$ edges. The re-occurrence of the bound of $n-1$ in Lemma B.1 suggests that maybe equality holds if and only if G is a tree? This indeed is the case. In fact, the following characterization of tree holds:

Lemma B.2 *Any tree has the following three properties:*

(a) It has $n-1$ edges.

(b) It has no cycle.

(c) It is connected.

Conversely, any graph that satisfies any two of the above properties is a tree.

For a proof, see for example [PS82].

Now recall the graph G_T that we computed in the proof of Lemma B.1. We showed that any vertex $v \neq r$ can be reached from r in G_T. Therefore, G_T is connected, because we can get from any v to any w by going from v to r, and then from r to w. We also showed that $m(G_T) = n-1$. This two observations together with the above characterization of trees says that G_T is a tree.

Thus, any connected graph G has a subgraph G_T that contains all vertices and is a tree; such a subgraph is sometimes also called a *spanning tree*.

Now let T be a tree, and let v be an arbitrary vertex. Apply the proof of Lemma B.1 to T, setting $r = v$. The spanning tree that we obtain has $n-1$ edges, and therefore must be again T. For any vertex $w \neq v$, we have a unique path from v to w in $G_T = T$. This proves that for any tree T and any two vertices v, w, there exists a unique path between v and w in T.

Trees have another useful property:

Lemma B.3 *Any tree with $n \geq 2$ vertices has at least two* leaves*, i.e., vertices with degree 1.*

Proof: Let T be a tree with $n \geq 2$ vertices, and let w be a vertex of minimum degree in T. Because T is connected and $n \geq 2$, vertex w must have at least one neighbor, so $\deg(w) \geq 1$. Assume now that all vertex $\neq w$ have degree ≥ 2, then

$$2m = \sum_{v \in V} \deg(v) \geq 2(n-1) + 1.$$

But on the other hand, $2m = 2(n-1)$ because T is a tree, so this implies $0 \geq 1$, which is impossible. Therefore there must be at least one vertex $x \neq w$ with $\deg(x) \leq 1$. As above, one argues $\deg(x) \geq 1$, so $\deg(x) = 1$. Since w was the vertex of minimum degree, this means $\deg(w) = 1$ as well, which proves the claim. □

B.2 Higher connectivity

B.2.1 Definitions

A *cutting k-set* is a set V_k of k vertices such that $G - V_k$ has strictly more connected components than G. When $k = 1$, the vertex in V_k is known as a *cut-vertex* (sometimes also called *articulation point*), and when $k = 2$, the pair of vertices in V_k is known as a *cutting pair* (sometimes also called *separation pair*). A graph is *k-connected* if it is connected and there is no cutting set with fewer than k vertices. In Figure B.1 the leftmost graph is 1-connected, since the circled vertex is a cut-vertex. The middle graph is 2-connected (also known as *biconnected*), while the rightmost graph is 3-connected (also known as *triconnected*). There are no fancy names for graphs that are k-connected, for $k > 3$.

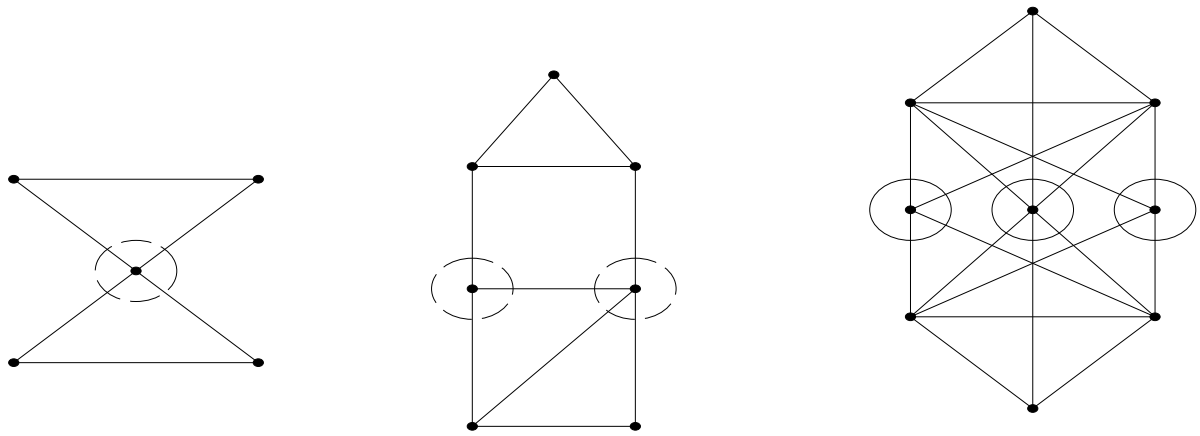


Figure B.1: 1-connected, 2-connected, and 3-connected

B.2.2 Non-simple graphs

Most source in the literature treat connectivity in graphs that are not simple (i.e., that have multiple edges and/or loops) by considering the underlying simple graph, i.e., simply forgetting about loops and multiple edges. However, in some applications (for example for the 2-terminal SP-graphs, which we saw in Chapter 8), it makes sense to give a more stringent definition of k-connected graphs.

So for biconnectivity, we say that a graph is *biconnected* if it is connected, has no cutvertex, and *does not have a loop.* (Note that if the graph has a loop at vertex v, then v in some sense could be considered a cutvertex, since it separates the edge from the rest of the graph.) For triconnectivity, we say that a graph is *triconnected* if it is biconnected, has no cutting pair, and *does not have multiple edges.* (If (v, w) is a multiple edge, then we consider the pair $\{v, w\}$ to be a cutting pair that separates the multiple edge from the rest of the graph.)

B.2.3 Menger's theorem

If G is k-connected and $n \geq k+1$, then any vertex must have degree $\geq k$. For if $\deg(v) = l \leq k-1$, then the $\leq l$ neighbors of v separate v from the rest of the graph, which is non-empty by $n \geq k+1$. But even more can be said:

Theorem B.4 (Menger's theorem) *A graph G with $n \geq k+1$ vertices is k-connected if and only if for any 2 distinct vertices v, w in G, there exist k paths $P_1, \ldots, P_k$ in the underlying simple graph of G that are* internally vertex-disjoint*, i.e., that have no vertex other than v and w in common.*

For a proof of Menger's theorem, refer for example to [Gib85]. In particular, we can apply Menger's theorem for $k = 2$; then the two paths form a simple cycle.

Corollary B.5 *A graph G with $n \geq 3$ vertices is biconnected if and only if for any two vertices v, w there exists a simple cycle in the underlying simple graph of G containing both v and w.*

Another application of Menger's theorem analyzes properties of a triconnected graph. Recall that a *minor* is a subgraph obtained by contracting or deleting edges; in particular, to show that a graph H is a minor of G, it suffices to show that a graph obtained from H by subdividing edges is a subgraph of G.

Lemma B.6 *If G is triconnected and $n \geq 4$, then G contains K_4 as a minor.*

Proof: Let (v, w) be an edge of G. Let P_1 and P_2 be two vertex-disjoint paths from v to w; these exist by Menger's theorem. Let x_1 and x_2 be the two neighbours of v on these paths. See also Figure B.2(a).

Since $\{v, w\}$ is not a cutting pair, there exists a path from x_1 to x_2 that does not use either v or w. Call this path P. Then P begins at a vertex in P_1 and ends at a vertex in P_2, and does not contain v or w. See Figure B.2(b).

Let P^* be the shortest path that begins at a vertex in P_1, ends at a vertex in P_2, and does not contain either v or w. Clearly P^* does not contain any vertices from P_1 or P_2 (except its endpoints), otherwise there would be a shorter path. Thus, the four paths $(v, w), P_1, P_2, P^*$ form a subdivision of K_4, which proves the claim. See Figure B.2(c). □

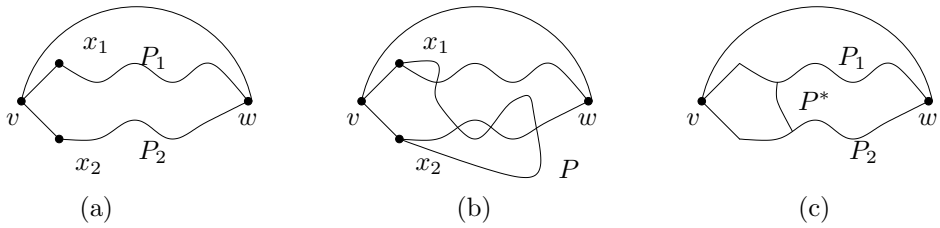


Figure B.2: A triconnected graph contains K_4 as a minor.

B.2.4 k-connected components

If a graph is not k-connected, then many algorithms that need k-connectivity handle such a graph by breaking it into components that are k-connected and dealing with each such component separately. Thus we need the concept of a k-connected component. This is *not* just a maximal subgraph that is k-connected (though this definition is equivalent for $k = 2$), but instead they are defined through a recursive algorithmic process as follows:

Definition B.7 *The* k-connected components *of a graph G are defined recursively as follows:*

- *If G is k-connected, then it is the unique k-connected component of itself.*

  For $k = 2$, we also stop if G is a loop, and for $k = 3$ we stop if G is a multiple edge.

- *Else, assume G has a cutting $(k-1)$-set V'.*

  - *Let $G_1, \ldots, G_l$ be the connected components of $G - V'$.*

- *For each $i \in \{1, \ldots, l\}$, let G_i^* be the graph induced by $V(G_i) \cup V'$, and add to it all edges between vertices in V'. (These edges are called* virtual edges*, and depicted dashed in all our examples.)*
- *The k-connected components of G are then the union of the k-connected components of $G_1^*, \ldots, G_l^*$.*[1]

Given the set of k-connected components, we can define a tree as follows: We have a node for every k-connected component, and a node for every cutting k-set that has been used to compute the k-connected components. Each node of a cutting k-set stores all edges between them as well as all virtual edges created by this cutting k-set. We connect a cutting k-set-node to a k-connected component node if and only if the cutting k-set is part of the k-connected component. See Figure B.3.[2]

That this is indeed a tree follows from the way that k-connected components are defined recursively. For $k = 2$, this tree is also known as the *blocktree*. For $k = 3$, this tree is closely related to the *SPQR-tree*, a data structure to store triconnected components [DBT96].

Biconnected components can be found efficiently in linear time by doing a depth-first search and keeping track of some additional information. See for example [BG00]. The block tree can then be built easily in linear time as well.

Triconnected components can be found in linear time as well, though the algorithm is anything but simple ([HT73], see [GM01] for corrections to this algorithm.) The SPQR-tree can then also be built in linear time [DBT96].

[1] With this definition, the k-connected components depend for $k \geq 3$ on the order in which cutting sets were chosen. This can be avoided by merging k-connected components back together. We will not do this here, but see for example [HT73] and [DBT96] for how to do this for $k = 3$.

[2] We sometimes omit nodes of cutting k-sets that have degree 2, i.e., that created only two k-connected components, since all the information about them can be inferred from their neighbours. Such nodes can be replaced by an edge while maintaining a tree.

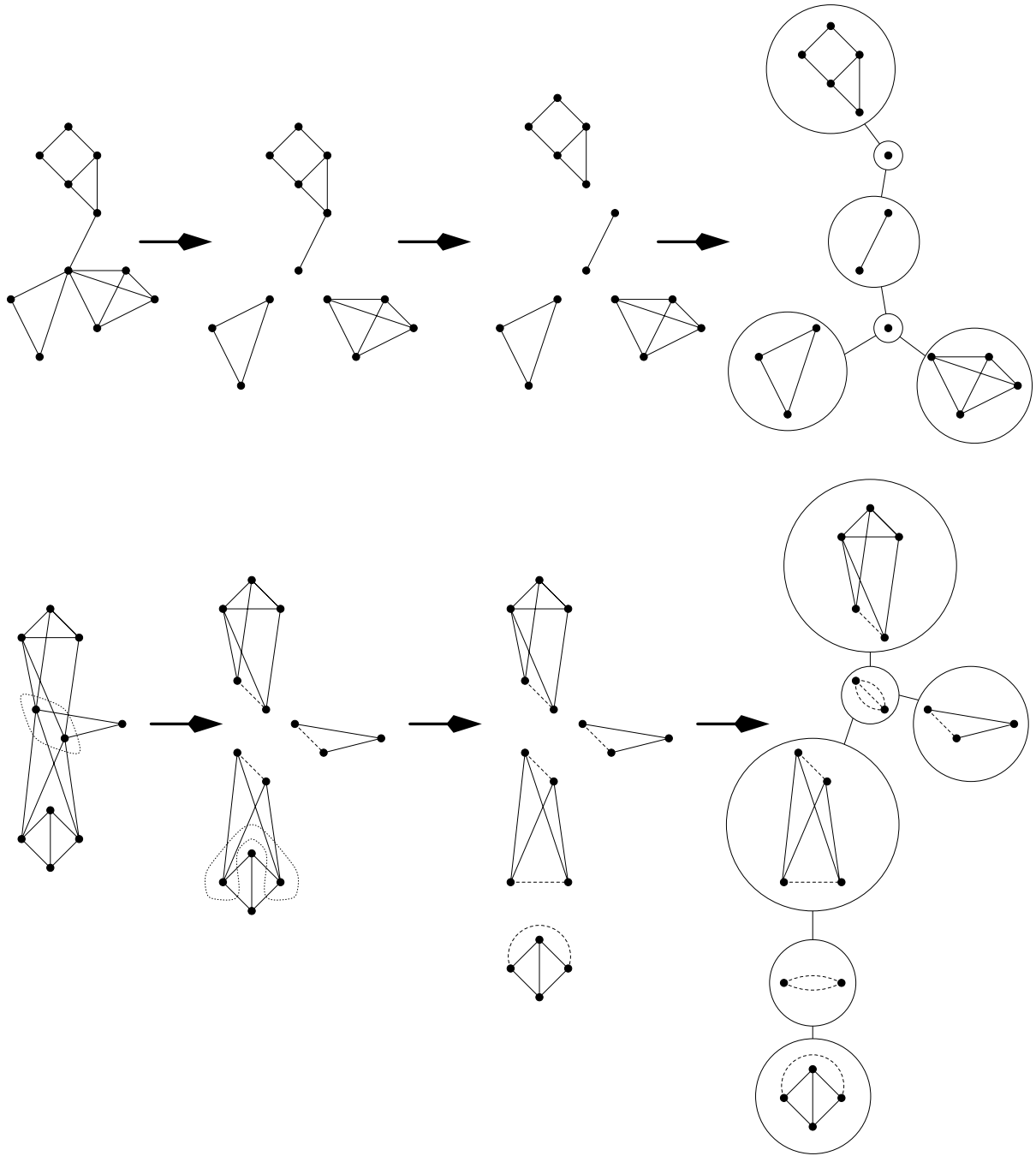


Figure B.3: The k-connected components and the tree defined by them, for $k = 2$ and $k = 3$.

Appendix C

Some Common Graph Problems

This section lists some common graph problems that are studied in this course:

- **Clique**:

  A *clique* of a graph $G = (V, E)$ is a set C of vertices such that all pairs of vertices in C are adjacent. The (Maximum) Clique problem is the problem of finding a maximum clique in a given graph. The size of a maximum clique is denoted $\omega(G)$.

  It is well-known that Clique is NP-hard for general graphs.

- **Independent Set**:

  An *independent set* of a graph $G = (V, E)$ is a subset I of the vertices such that no edge has both endpoints in I. The (Maximum) Independent Set problem (sometimes referred to as MIS or IS) is the problem of finding a maximum independent set in a given graph. The size of this set is denoted $\alpha(G)$.

  One can easily see that $\alpha(G) = \omega(\overline{G})$, since a clique in a graph becomes an independent set in the complement graph and vice versa.

  It is well-known that independent set is NP-hard for general graphs.

- **Coloring**:

  A *k-coloring* of a graph $G = (V, E)$ is an assignment of integers in $\{1, \ldots, k\}$ to the vertices such that for any edge the two endpoints have a different color. The k-Coloring problem consists of testing whether a given graph has a k-coloring. The Colouring problem consists of finding the smallest number k such that a given graph G has a k-coloring. This number is also denoted $\chi(G)$.

  k-Coloring is NP-hard for $k \geq 3$, but is polynomial for $k = 2$, since 2-coloring is the same as testing whether a graph is bipartite, which can be done with a depth-first search traversal in linear time.

  If a graph contains a clique of size k, then we need at least k colors to color the vertices in that clique. Hence, $\chi(G) \geq \omega(G)$ for all graphs. Equality does not necessarily hold (for example for a 5-cycle), but does hold for perfect graphs.

- **Longest Path**: Given a graph $G = (V, E)$, find the longest simple path in it. This problem also exists in a weighted version, where each edge has a weight $w(e)$ attached to it. We then want to find the simple path P that maximizes $\sum_{e \in P} w(e)$.

  This problem is NP-hard, since the Hamiltonian Path problem is a special case of it.

- **Maximum Cut**: A *cut* in a graph $G = (V, E)$ is a partition of the vertices into two subsets $V = A \cup B$. More precisely, given such a partition, the cut is the set of edges with exactly one endpoint each in A and B. The Maximum Cut problem is the problem of finding a partition with the maximum number of edges in the cut.

  This problem is similar to the Minimum Cut problem, where we want to find the cut with the minimum number of edges. The Minimum Cut problem is well-known to be solvable in polynomial time (via Maximum Flow, see for example [AMO93]). On the other hand, the Maximum Cut problem is known to be NP-hard.

Bibliography

[ACP87] S. Arnborg, D.G. Corneil, and A. Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM J. Alg. Disc. Meth.*, 8:277–284, 1987.

[AH77] K. Appel and W. Haken. Every planar map is four colorable. I. Discharging. *Illinois J. Math.*, 21(3):429–490, 1977.

[AHK77] K. Appel, W. Haken, and J. Koch. Every planar map is four colorable. II. Reducibility. *Illinois J. Math.*, 21(3):491–567, 1977.

[AI77] K. Aoshima and M. Iri. Comments on F. Hadlock's paper: "Finding a maximum cut of a planar graph in polynomial time". *SIAM J. Computing*, 6(1):86–87, 1977.

[Aig84] M. Aigner. *Graphentheorie: Eine Entwicklung aus dem 4-Farben Problem*. Teubner Studienbücher, 1984. English translation: *Graph Theory: A Development from the 4-Color Problem*, 1987.

[AMO93] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993.

[AP61] L. Auslander and S. V. Parter. On imbedding graphs in the sphere. *Journal of Mathematics and Mechanics*, 10(3):517–523, 1961.

[Bak94] B. Baker. Approximation algorithms for NP-complete problems on planar graphs. *J. ACM*, 41(1):153–180, 1994.

[BBD+04] T. Biedl, B. Brejová, E. Demaine, A. Hamel, A. López-Ortiz, and T. Vinař. Finding hidden independent sets in interval graphs. *Theoretical Computer Science*, 310(1-3):287–307, January 2004.

[BE97] Hans L. Bodlaender and Joost Engelfriet. Domino treewidth. *J. Algorithms*, 24(1):94–123, 1997.

[BG00] Sara Baase and Allen Van Gelder. *Computer Algorithms: Introduction to Design and Analysis, 3rd edition*. Addison Lewley, 2000.

[BK96] Hans L. Bodlaender and Ton Kloks. Efficient and constructive algorithms for the pathwidth and treewidth of graphs. *J. Algorithms*, 21(2):358–402, 1996.

[BL76] K. Booth and G. Lueker. Testing for the consecutive ones property, interval graphs and graph planarity using PQ-tree algorithms. *Journal of Computing and System Sciences*, 13:335–379, 1976.

[BM76] J. Bondy and U. Murty. *Graph Theory and Applications.* Amerian Elsevier Publishing Co., 1976.

[BM90] D. Bienstock and C. Monma. On the comlexity of embedding planar graphs to minimize certain distance measures. *Algorithmica*, 5(1):93–109, 1990.

[BM99] J. Boyer and W. Myrvold. Stop minding your P's and Q's: A simplified $O(n)$ planar embedding algorithm. In *SIAM-ACM Symposium on Discrete Algorithms*, pages 140–146, 1999.

[Bod93] Hans Bodlaender. A tourist guide through treewidth. *Acta Cybernetica*, 11:1–21, 1993.

[Bod97] Hans L. Bodlaender. Treewidth: algorithmic techniques and results. In *Mathematical foundations of computer science 1997 (Bratislava)*, volume 1295 of *Lecture Notes in Comput. Sci.*, pages 19–36. Springer, Berlin, 1997.

[BT96] G. Di Battista and R. Tamassia. On-line planarity testing. *SIAM J. Computing*, 25(5), 1996.

[BT97] Hans L. Bodlaender and Dimitrios M. Thilikos. Constructive linear time algorithms for branchwidth. In *Automata, languages and programming (Bologna, 1997)*, volume 1256 of *Lecture Notes in Comput. Sci.*, pages 627–637. Springer, Berlin, 1997.

[CLRS00] T. H. Cormen, C. E. Leiserson, Ronald L. Rivest, and C. Stein. *Introduction to Algorithms, 2nd edition.* MIT Press, McGraw-Hill Book Company, 2000.

[CLV03] G. Cornuéjols, X. Li, and K. Viskovic. A polynomial algorithm for recognizing perfect graphs. In *44th Annuel IEEE Symposium on Foundations in Computer Science (FOCS'03)*, pages 20–27, 2003.

[CM89] J. Cheriyan and S.N. Maheshwari. Analysis of preflow push algorithms for maximum network flow. *SIAM J. on Computing*, 18:1057–1086, 1989.

[CNAO85] N. Chiba, T. Nishizeki, S. Abe, and T. Ozawa. A linear algorithm for embedding planar graphs using PQ-trees. *J. Comput. System Sci.*, 30:54–76, 1985.

[CNS81] N. Chiba, T. Nishizeki, and N. Saito. A linear 5-coloring algorithm of planar graphs. *J. Algorithms*, 2:317–327, 1981.

[COS98] Derek G. Corneil, Stephan Olariu, and Lorna Stewart. The ultimate interval graph recognition algorithm? (extended abstract). In *Symposium on Discrete Algorithms*, pages 175–180, 1998.

[Coz81] M.B. Cozzens. *Higher and multi-dimensional analogues of interval graphs*. PhD thesis, Computer Science, 1981.

[CRST03] Maria Chudnovsky, Neil Robertson, P. D. Seymour, and Robin Thomas. Progress on perfect graphs. *Math. Program.*, 97(1-2, Ser. B):405–422, 2003. ISMP, 2003 (Copenhagen).

[CS04] Maria Chudnovsky and P.D. Seymour. Recognizing berge graphs, 2004. submitted.

[DBT96] G. Di Battista and R. Tamassia. On-line maintenace of triconnected components with SPQR-trees. *Algorithmica*, 15(4):302–318, 1996.

[Din70] E.A. Dinic. Algorithm for solution of a problem of maximum flow in networks with power estimation. *Soviet Math. Dokl.*, 11:1277–1280, 1970.

[DMP64] G. Demoucron, Y. Malgrange, and R. Pertuiset. Graphes planaires: reconnaissance et construction de représentations planaires topologiques. *Rev. Francaise Recherche Opérationnelle*, 8:33–47, 1964.

[Edm65] Jack Edmonds. Maximum matching and a polyhedron with $0,1$-vertices. *J. Res. Nat. Bur. Standards Sect. B*, 69B:125–130, 1965.

[ET76] S. Even and R. E. Tarjan. Computing an *st*-numbering. *Theoretical Computer Science*, 2:436–441, 1976.

[Fár48] I. Fáry. On straight line representation of planar graphs. *Acta. Sci. Math. Szeged*, 11:229–233, 1948.

[FF62] L. Ford and D. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.

[FPP90] H. de Fraysseix, J. Pach, and R. Pollack. How to draw a planar graph on a grid. *Combinatorica*, 10:41–51, 1990.

[Gav74] F. Gavril. Algorithms on circular-arc graphs. *Networks*, 4:357–369, 1974.

[Gib85] A. Gibbons. *Algorithmic Graph Theory*. Cambridge University Press, 1985.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, 1979.

[GJMP78] M.R. Garey, D.S. Johnson, G.L. Miller, and C.H. Papadimitriou, 1978. Unpublished results.

[GM01] Carsten Gutwenger and Petra Mutzel. A linear time implementation of spqr-trees. In J. Marks and K. Ryall, editors, *Symposium on Graph Drawing 2000*, volume 1984 of *Lecture Notes in Computer Science*, pages 77–90, 2001.

[Gol77] M.C. Golumbic. The complexity of comparability graph recognition and colouring. *Computing*, 18:199–208, 1977.

[Gol80] Martin Charles Golumbic. *Algorithmic graph theory and perfect graphs*. Academic Press, New York, 1980.

[Gol85] A.V. Goldberg. A new max-flow algorithm. Technical Report MIT/LCS/TM-291, MIT, Laboratory for Computer Science, 1985.

[GT88] A.V. Goldberg and R.E. Tarjan. A new approach to the maximum flow problem. *J. Association Computing Machinery*, 35:921–940, 1988.

[GW80] Jerrold R. Griggs and Douglas B. West. Extremal values of the interval number of a graph. *SIAM J. Algebraic Discrete Methods*, 1(1):1–7, 1980.

[Had75] F. Hadlock. Finding a maximum cut of a planar graph in polynomial time. *SIAM J. Computing*, 4:221–225, 1975.

[Has81] M. Hassin. Maximum flow in (s, t)-planar networks. *Information Processing Letters*, 13:107, 1981.

[HKL99] Xin He, Ming-Yang Kao, and Hsueh-I Lu. Linear-time succinct encodings of planar graphs via canonical orderings. *SIAM J. Discrete Math.*, 12(3):317–325 (electronic), 1999.

[HM91] Wen-Lian Hsu and Tze-Heng Ma. Substitution decomposition on chordal graphs and applications. In Wen-Lian Hsu and Richard C. T. Lee, editors, *ISA '91 Algorithms, 2nd International Symposium on Algorithms, Taipei, Republic of China, December 16-18, 1991, Proceedings*, volume 557 of *Lecture Notes in Computer Science*, pages 52–60. Springer, 1991.

[Hsu93] W. L. Hsu. A new planarity test. In *International Workshop on Discrete Math. and Algorithms*, pages 110–122, 1993.

[HT73] J. E. Hopcroft and R. E. Tarjan. Dividing a graph into triconnected components. *SIAM J. Computing*, 2(3):135–158, 1973.

[HT74] J. E. Hopcroft and R. E. Tarjan. Efficient planarity testing. *J. Association Computing Machinery*, 21(4):549–568, 1974.

[HY69] T. C. Hu and R. D. Young. *Integer programming and network flows*. Addison-Wesley Publishing Co., Reading, Mass.-London-Don Mills, Ont., 1969.

[IS79] A. Itai and Y. Shiloach. Maximum flows in planar networks. *SIAM J. Computing*, 8:135–150, 1979.

[Kan96] G. Kant. Drawing planar graphs using the canonical ordering. *Algorithmica*, 16:4–32, 1996.

[KH97] Goos Kant and Xin He. Regular edge labeling of 4-connected plane graphs and its applications in graph drawing problems. *Theoret. Comput. Sci.*, 172(1-2):175–193, 1997.

[KM89] N. Korte and R.H. Möhring. An incremental linear-time algorithm for recognizing interval graphs. *SIAM J. Computing*, 18(1):68–81, 1989.

[Kra94] Jan Kratochvíl. A special planar satisfiability problem and a consequence of its NP-completeness. *Discrete Appl. Math.*, 52(3):233–252, 1994.

[KRRS97] Philip Klein, Satish Rao, Monika Rauch, and Sairam Subramanian. Faster shortest-path algorithms for planar graphs. *J. Computer System Sciences*, 55:3–23, 1997.

[KT00] J. Kratochvil and Zs. Tuza. On the complexity of bicoloring clique hypergraphs of graphs. In *11th Annual Symposium on Discrete Algorithms (SODA 2000)*, pages 40–41. ACM, 2000.

[Law76] Eugene Lawler. *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart and Winston, 1976.

[LEC67] A. Lempel, S. Even, and I. Cederbaum. An algorithm for planarity testing of graphs. In *Theory of Graphs, International Symposium Rome 1966*, pages 215–232. Gordon and Breach, 1967.

[LED] LEDA: Library of efficient data structures and algorithms. See http://www.mpi-sb.mpg.de/LEDA.

[Lic82] David Lichtenstein. Planar formulae and their uses. *SIAM J. Comput.*, 11(2):329–343, 1982.

[LT79] R. Lipton and R. Tarjan. A separator theorem for planar graphs. *SIAM J. Appl. Math.*, 36(2):177–189, 1979.

[Mar45] E. Marczewski. Sur deux propriétés des classes d'ensembles. *Fundamenta Methematicae*, 33:303–307, 1945.

[McC03] Ross M. McConnell. Linear-time recognition of circular-arc graphs. *Algorithmica*, 37(2):93–147, 2003.

[MM96] K. Mehlhorn and P. Mutzel. On the embedding phase of the Hopcroft and Tarjan planarity testing algorithm. *Algorithmica*, 16:233–242, 1996.

[MR01] Colin McDiarmid and Bruce Reed. Channel assignment on nearly bipartite and bounded treewidth graphs. In Jaroslav Nesetril, Marc Noy, and Oriol Serra, editors, *Electronic Notes in Discrete Mathematics*, volume 10. Elsevier, 2001.

[MS89] J.H. Muller and J.P. Spinrad. Incremental modular decomposistion. *J. Assoc. Comput. Mach.*, 36:1–19, 1989.

[MT01] Bojan Mohar and Carsten Thomassen. *Graphs on surfaces*. Johns Hopkins Studies in the Mathematical Sciences. Johns Hopkins University Press, Baltimore, MD, 2001.

[MV80] S. Micali and V. V. Vazirani. An $O(\sqrt{|V|} \cdot |E|)$ algorithm for finding maximum matching in general graphs. In *21st Annual Symposium on Foundations of Computer Science*, pages 17–27, New York, 1980. Institute of Electrical and Electronics Engineers Inc. (IEEE).

[NC88] T. Nishizeki and N. Chiba. *Planar Graphs: Theory and Algorithms*. North-Holland, 1988.

[OvW78] R Otten and J van Wijck. Graph representations in interactive layout design. In *IEEE International Symposium on Circuits and Systems*, pages 914–918, New York, 1978.

[Par78] T. D. Parsons. Pursuit-evasion in a graph. In *Theory and applications of graphs (Proc. Internat. Conf., Western Mich. Univ., Kalamazoo, Mich., 1976)*, pages 426–441. Lecture Notes in Math., Vol. 642. Springer, Berlin, 1978.

[PS82] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice Hall, Inc., Englewood Cliffs, NJ, 1982. ISBN 0-13-152462-3.

[PY81] C. Papadimitriou and M Yannakakis. The clique problem for planar graphs. *Information Processing Letters*, 13:131–133, 1981.

[Ree92] Bruce A. Reed. Finding approximate separators and computing tree width quickly. In *Annual ACM Symposium on Theory of Computing (STOC'92)*, pages 221–228. ACM, 1992.

[Rob69] Fred S. Roberts. On the boxicity and cubicity of a graph. In W. T. Tutte, editor, *Recent Progress in Combinatorics*, pages 301–310. Academic Press, 1969.

[Rob76] Fred S. Roberts. *Discrete mathematical models with application to social, biological and ecological problems*. Prentice-Hall, Englewood Cliffs, NJ, 1976.

[RS83] Neil Robertson and P. D. Seymour. Graph minors. I. Excluding a forest. *J. Combin. Theory Ser. B*, 35(1):39–61, 1983.

[RS03] Neil Robertson and Paul Seymour. Graph minors. XVIII. Tree-decompositions and well-quasi-ordering. *J. Combin. Theory Ser. B*, 89(1):77–108, 2003.

[RSST97] N. Robertson, D. Sanders, P. Seymour, and R. Thomas. The four-colour theorem. *J. Combin. Theory Ser. B*, 70(1):2–44, 1997.

[RT75] Donald J. Rose and R. Endre Tarjan. Algorithmic aspects of vertex elimination. In *Seventh Annual ACM Symposium on Theory of Computing (Albuquerque, N. M., 1975)*, pages 245–254. Assoc. Comput. Mach., New York, 1975.

[RT86] P. Rosenstiehl and R. E. Tarjan. Rectilinear planar layouts and bipolar orientation of planar graphs. *Discrete Computational Geometry*, 1:343–353, 1986.

[RTL76] Donald J. Rose, R. Endre Tarjan, and George S. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM J. Comput.*, 5(2):266–283, 1976.

[Sch90] W. Schnyder. Embedding planar graphs on the grid. In *1st Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 138–148, 1990.

[SH92] W.-K. Shih and W.-L. Hsu. A simple test for planar graphs. In *Proceedings of the Sixth Workshop on Discrete Mathematics and Theory of Computation*, pages 35–42, 1992.

[Spi03] Jeremy Spinrad. *Efficient graph representations*. American Mathematical Society, 2003.

[Ste51] S. Stein. Convex maps. In *Amer. Math. Soc.*, volume 2, pages 464–466, 1951.

[SWK90] Wei-Kuan Shih, Kuo Wu, and Y.S. Kuo. Unifying maximum cut and minimum cut of a planar graph. *IEEE Transactions on Computers*, 39:694–597, 1990.

[TSB00] Dimitrios M. Thilikos, Maria J. Serna, and Hans L. Bodlaender. Constructive linear time algorithms for small cutwidth and carving-width. In *Algorithms and computation (Taipei, 2000)*, volume 1969 of *Lecture Notes in Comput. Sci.*, pages 192–203. Springer, Berlin, 2000.

[TT86] R. Tamassia and I. Tollis. A unified approach to visibility representations of planar graphs. *Discrete Computational Geometry*, 1:321–341, 1986.

[Wag36] K. Wagner. Bemerkungen zum Vierfarbenproblem. *Jahresbericht der Deutschen Mathematiker-Vereinigung*, 46:26–32, 1936.

[WG86] M.S. Waterman and J.R. Griggs. Interval graphs and maps of DNA. *Bulletin Math. Biol.*, 48(2):189–195, 1986.

[WS84] Douglas B. West and David B. Shmoys. Recognizing graphs with fixed interval number is NP-complete. *Discrete Appl. Math.*, 8(3):295–305, 1984.

[Yan82] Mihalis Yannakakis. The complexity of the partial order dimension problem. *SIAM J. Algebraic Discrete Methods*, 3(3):351–358, 1982.