DISCRETE MATHEMATICS

lecture notes

Gašper Fijavž
Faculty of Computer and Information Science

Ljubljana, November 2014

CIP - Kataložni zapis o publikaciji
Narodna in univerzitetna knjižnica, Ljubljana

51(075.8)(0.034.2)

FIJAVŽ, Gašper
Discrete mathematics [Elektronski vir] : lecture notes / Gašper Fijavž. - El. knjiga. - Ljubljana : Fakulteta za računalništvo in informatiko, 2014

Način dostopa (URL): http://matematika.fri.uni-lj.si/discrete_mathematics.pdf

ISBN 978-961-6209-84-7 (pdf)

277297152

Contents

Uvod

Predmet *Diskretna matematika* študenta popelje na področje zahtevnejših grafovskih algoritmov, ki jih obravnavamo delno z matematičnega in delno z računalniškega vidika.

Zdi se, da so algoritmični problemi na grafih ali zelo enostavni, takšne algoritme srečamo pri tečaju iz algoritmov in podatkovnih struktur, ali pa NP-težki. Pri diskretni matematiki bomo poskusili najti srednjo pot, večinoma se bomo držali v bazenu polinomsko rešljivih problemov, ki presegajo osnovno algoritmično šolo.

Za uspešen študij takšnih problemov pa bo potrebno nekaj novih matematičnih znanj.

Predmet *Diskretna matematika* izvajamo v angleškem jeziku, pričujoči zapiski predavanj pa so dostopni tudi na spletnem naslovu

`matematika.fri.uni-lj.si/discrete_mathematics.pdf`

Introduction

The *Discrete mathematics* course tackles a selection of graph algorithms, which are studied from both the mathematical and computational point of view.

We often have the impression that graph algorithmic problems are either very basic, and as such taught in an introductory algorithms course, or NP-hard. This course tries to steer in between. We shall mostly study problems that are computationally easy—solvable in polynomial time, yet difficult enough to surpass the collection of elementary algorithms.

We shall need quite a lot of discrete mathematical background to successfully deal with these types of problems, and the details are provided herein.

These lecture notes are available at

`matematika.fri.uni-lj.si/discrete_mathematics.pdf`

1 Graph Searching

1.1 Definitions

Let G be a graph. We say that vertex u is *reachable* from a vertex v, $u \rightsquigarrow v$, if there exists a path P_{uv} (equivalently a walk) starting at u and ending at v. In case G is a directed graph also the path P_{uv} is supposed to be a directed one.

Reachability is trivially — using paths of length 0 — a reflexive relation on the set $V(G)$. If G is undirected, then reachability is also symmetric and transitive.

Let G temporarily denote an undirected graph. Reachability, being an equivalence relation, partitions the set $V(G)$ into equivalence sets $V_1, V_2, V_3, \ldots, V_k$, and the induced graphs $C_1 = G[V_1], C_2 = G[V_2], C_3 = G[V_3], \ldots, C_k = G[V_k]$ are called *connected components* of G. In case G has a single connected component we call G a *connected* graph.

A component C_i of G is a maximal connected induced subgraph of G.

The story is somewhat different in the case of directed graphs. If $\overrightarrow{G}$ is a directed graph it might happen that a vertex is reachable from another but not vice-versa, $u \rightsquigarrow v$ and $v \not\rightsquigarrow u$, the relation not being symmetric. Yet the relation of *mutual reachability*, $u \rightsquigarrow v$ **and** $v \rightsquigarrow u$, is an equivalence relation, and similarly as above, decomposes the graph $\overrightarrow{G}$ into *strongly connected components (s.c. components)* $\overrightarrow{C}_1, \overrightarrow{C}_2, \ldots, \overrightarrow{C}_\ell$. As above, a strongly connected component is a maximal strongly connected component of $\overrightarrow{G}$.

If $\overrightarrow{G}$ is a directed graph, then its underlying graph $\overline{G}$ is obtained by removing orientations of edges of $\overrightarrow{G}$, keeping the same vertex set, and suppressing possible parallel edges obtained by deleting orientations of a pair of counter oriented edges. Clearly, if $\overrightarrow{G}$ is strongly connected, then $\overline{G}$ is connected, but the reverse may not hold. We call $\overrightarrow{G}$ *weakly connected* if its underlying counterpart $\overline{G}$ is connected. Note that weak connectivity of $\overrightarrow{G}$ does not imply that for arbitrary vertices $u, v \in V(\overrightarrow{G})$ at least one is reachable from the other.

In order to keep our results tidier we shall also say that a connected undirected graph G is strongly connected as well.

Let us now define distance between vertices in G: the *distance from x to y*, $\operatorname{dist}(x, y)$, is the length of a shortest $x \rightarrow y$-path in $\overrightarrow{G}$. Note that in a directed graph $\overrightarrow{G}$ the distances $\operatorname{dist}(x, y)$ and $\operatorname{dist}(y, x)$ may be different, yet in an undirected graph distance is symmetric.

We call $\overrightarrow{G}$ a *directed acyclic graph* or *dag* (for short) if $\overrightarrow{G}$ has no *directed* cycles — a pair of counter oriented edges is considered a cycle of length 2. Observe that its underlying graph $\overline{G}$ may contain cycles. A *topological ordering* of vertices of $\overrightarrow{G}$ is a linear ordering $v_1, v_2, v_3, \ldots, v_n$ (numbering of vertices), so that if $v_i v_j \in E(\overrightarrow{G})$ then $i < j$. In other words, with respect to the topological ordering, every edge is pointing to the right.

If $\overrightarrow{G}$ is not a dag then its vertices cannot be topologically ordered. Irrespective of the numbering of vertices, traversing a directed cycle necessarily makes at least one step to the left. Does the reverse implication also hold? If $\overrightarrow{G}$ is a dag, do its vertices admit a topological ordering? The easiest argument is inductive: if $\overrightarrow{G}$ is a dag, then there exists a vertex v satisfying $\operatorname{outdeg}(v) = 0$ (otherwise every walk can be extended by an additional step, $\overrightarrow{G}$ admits walks with repeated vertices, and the shortest walk with repeated vertices is a directed cycle). Now v can be put as the last vertex in the topological ordering, and vertices of $\overrightarrow{G} - v$ can be ordered recursively.

1.2 Graph searching, general schema

We can picture graph searching like a disease infecting vertices which is spreading along (directed) edges. Initially, vertices are healthy, and in the beginning a vertex, often called the *root*, gets infected. The process stabilizes when no new infections are possible, and it might happen that not all vertices are infected. In this case we may restart by infecting another vertex.

We shall rather use colors for indicating whether a vertex has been visited with a search algorithm. A vertex is *white* if it has not been discovered with a searching procedure, and infection turns the vertex *black* (like plague or black death, right). Upon discovery a vertex is put in a data structure D. Initially all vertices are colored white, D is empty, and an auxiliary structure *search forest T* is trivial.

```
   GraphSearch($G$)
1  while exists a white vertex $v$ do
2      GraphSearch($(G, v)$)

   GraphSearch($G, v$)
1  add $s$ to $D$ and color it black;
2  while $D \neq \emptyset$ do
3      remove a vertex $v$ from $D$;
4      foreach $u \in N(v)$ do
5          if $u$ is white then
6              add $u$ to $D$ and color it black;
7              $T = T + uv$
```

Algorithm 1.1: GraphSearch(G, s) and its wrapper GraphSearch(G), top.

The edges of T store the information of the disease spread. If a vertex u has acquired the disease from vertex v, then the (directed) edge vu is put in the search forest T.

Note that every call of GraphSearch(G, s) produces a single component of the search forest. In case we call the searching procedure GraphSearch(G, s) without a prior call of the wrapper GraphSearch(G) we name T the *search tree*. The search tree T is rooted at s with edges oriented away from the root s.

A search forest T partitions the edges of G into four classes. An edge $uv \in E(G)$ is

- a *tree edge* if $uv \in E(T)$,
- a *forward edge* if v is a *descendant* of u in T,
- a *backward edge* if v is an *ancestor* of u in T, and
- a *cross edge* in all the remaining cases.

Proposition 1.1 *If G is strongly connected, then the search forest T has a single component (i.e. is indeed a tree).*

Proof. Assume to the contrary that at the end not every vertex is black. Let x and y be a black and a white vertex, respectively, so that the shortest directed $x \to y$-path is as short as possible. An interior vertex of any color is in contradiction with the minimal length of the path, hence $xy \in E(G)$. As x is black at some point x is added to D. Now observe line 3 of Algorithm 1.1 when vertex x is removed from D. The next line adds all white neighbors of x, including y, to D, which is a contradiction to the color of y. □

Corollary 1.2 *Let T be a search forest of G. If G is undirected, then T has the same number of components as G. If G is a directed graph, then the number of components of T is at most the number of strongly connected components of G.*

Proof. The second statement is an immediate consequence of Proposition 1.1. The first one follows as there are no edges between components in an undirected graph. □

Let us compute the time complexity of the GraphSearch(G) algorithm. Let us assume that G is a directed graph. We shall implicitly assume that data structure operations take constant amount of time. Preprocessing is done in $O(n)$ time; namely we assign every vertex a color. We proceed by amortized analysis. Every vertex enters and leaves D at most once. Upon v leaving D, we check all edges whose tail equals v, on line 3. In total, each edge is checked at most once whether or not its head u is colored white. Summing vertex outdegrees over all vertices equals the number of edges in G. Hence the total time complexity is $O(n + m)$.

Theorem 1.3 *Graph searching takes $O(n + m)$ time.*

1.3 Breadth first search

Breadth first search is a variant of graph searching where the goal is to spread out the search as evenly as possible using a queue Q, a first-in-first-out Fifo data structure.

```
BFS($G$)
1 while exists a white vertex $v$ do
2     BFS ($G, v$)
```

```
BFS($G, s$)
1 EnQueue ($s, Q$);
2 while $Q \neq \emptyset$ do
3     $v$=DeQueue (Q);
4     foreach $u \in N(v)$ do
5         if $u$ is white then
6             EnQueue (u,Q) and color $u$ black;
7             add edge $vu$ to search forest $T$
```

Algorithm 1.2: BFS(G, s) and its wrapper BFS(G), top.

Breadth first search is typically the preferred choice for computing connected components of an undirected graph.

Let us begin by observing that at every time of the run of BFS(G, s), the vertices in Q are ordered by their distances from s, and even more:

Proposition 1.4 *Let $x_1, x_2, x_3, \ldots, x_k$ is the sequence of vertices in Q at an arbitrary instant of the run of* BFS(G, s)*. Then*

(a) $dist(s, x_1) \leq dist(s, x_2) \leq dist(s, x_3) \leq \ldots \leq dist(s, x_k)$*, and*

(b) $dist(s, x_k) \leq dist(s, x_1) + 1$.

Proof. We omit this proof □

We can nonetheless exploit the spread of searching in BFS(G, s) to compute distances from a fixed vertex s to the other vertices in G. Next, an $s \to x$ path in T is also a shortest $s \to x$ path in G. Let x be an ancestor of y in T. Then let $\text{dist}_T(x, y)$ denote the length of the $x \to y$-path in T. Obviously $\text{dist}_T(x, y) \geq \text{dist}(x, y)$. A little less obvious is the converse:

Proposition 1.5 *Let x be an ancestor of y in T. Then $dist_T(x, y) = dist(x, y)$.*

Proof. It is enough to prove the result for the case $x = s$ is the root of T. Let $s = x_0, x_1, x_2, \ldots, x_{k-1}, x_k = y$ be the sequence of vertices along the shortest and only $s \to y$-path in T, so that $k = \text{dist}_T(s, y) > \text{dist}(s, y)$. This implies that $\text{dist}(s, y) = \text{dist}(s, x_{k-1}) = k - 1$. Let y' be a vertex adjacent to y on some shortest $s \to y$-path, in particular $y' \neq x_{k-1}$. By Proposition 1.4 y' has entered Q before x_{k-1}. As $y'y$ is not a tree edge, y was already colored black by the time y' has left Q. This is impossible, as x_{k-1}, the predecessor of y, did not leave Q before y', again by

Proposition 1.4. □

Now Proposition 1.5 has an immediate consequence.

Theorem 1.6 *If T is a BFS-tree of G with root s, then T contains shortest path from s to every vertex v of G which is reachable from s, and both distances from s and instances of shortest $s{\to}v$ paths.*

Proof. Distances from s can be computed by a simple modification of BFS(G, s). Let us first set $d(v) = \infty$ for every $v \in V(G) \setminus \{s\}$, and set $d(s) = 0$, and run BFS(G, s) with an additional line

```
8            $d(u) = d(v) + 1$
```

□

1.4 Depth first search

Depth first search DFS can be implemented in an analogous way as BFS, using a stack instead of a queue. Initially stack S is empty, all vertices are colored white. The only technical issue — or difference compared to BFS — is that we color vertices black immediately *after* they get popped from S.

```
DFS($G$)
1 while exists a white vertex $v$ do
2   DFS ($G, v$)

DFS($G, s$)
1 Push ($s, S$);
2 while $S \neq \emptyset$ do
3   $v$=Pop (S);
4   if $v$ is white then
5     color $v$ black;
6     foreach $u \in N(v)$ do
7       if $u$ is white then
8         Push ($u, S$);
9         $T = T + uv$
```

Algorithm 1.3: DFS(G, s) and its wrapper DFS(G), top.

It is however most convenient to implement DFS recursively, again starting with all vertices colored white and a trivial search forest T.

Instead of using a stack as a data structure for storing vertices, the process uses the execution stack (call stack) storing active calls of DFSr. Let us enumerate call

```
DFSr(G)
1 while exists a white vertex $v$ do
2     DFSr($G, v$)

DFSr($G, s$)
1 color $u$ black;
2 foreach $u \in N(v)$ do
3     if $u$ is white then
4         $T = T + uv$;
5         DFSr($G, v$)
```

Algorithm 1.4: DFSr, a recursive version of depth first search.

stack operations pushing and popping calls of DFSr with consecutive integers ≥ 1. The *starting* and *finishing time* of a vertex v, $\mathrm{start}(v)$ and $\mathrm{finish}(v)$, respectively, are defined as indices of call stack operations marking the call of DFSr(G, v) and the end of its execution. The *active time* of a vertex v is the interval between starting and finishing times, $\mathrm{time}(v) = [\mathrm{start}(v), \mathrm{finish}(v)]$.

The call stack is, as its name suggests, a last-in-first-out Lifo structure, hence.

Proposition 1.7 *Let x, y be vertices, and $\mathrm{time}(x)$ and $\mathrm{time}(y)$ their respective active times. Then either $\mathrm{time}(x) \cap \mathrm{time}(y) = \emptyset$ or $\mathrm{time}(x)$ and $\mathrm{time}(y)$ are nested (one is a strict subset of the other).*

Nested active times of vertices can be further characterized.

Theorem 1.8 *Let T be a search forest/tree produced by depth first search. The following statements are equivalent for every pair of vertices x, y:*

(P1) $\mathrm{time}(y) \subset \mathrm{time}(x)$.

(P2) y is a descendant of x in T.

(P3) At the time $\mathrm{start}(x)$ there exists a directed $x \to y$ path whose vertices are all white.

Proof. Let us at this point stress that the stack in the next arguments is not the stack storing vertices from DFS but the call stack storing recursive calls of DFSr: a vertex v is in stack if the procedure DFSr(G, v) was recursively called and has not yet terminated.

(P1 $\Rightarrow$ P2 & P3) Observe the call stack at time $\mathrm{start}(y)$; vertex y has just been pushed to the top of the call stack. As $\mathrm{time}(x) \subset \mathrm{time}(y)$, vertex x is in the call stack and at time $\mathrm{start}(y)$ we can denote the sequence of vertices near the top of the call stack with $x = x_0, x_1, x_2, \ldots, x_{k-1}, x_k = y$. As for every $i \in \{0, \ldots, k-1\}$ the edge $x_i x_{i+1}$ is a tree edge, y is a descendant of x. Further, at time $\mathrm{start}(x)$ the very same sequence is a white path.

(P2 $\Rightarrow$ P1) The call of DFSR(G, x) is not yet finished at the start of the call DFSR(G, y). Hence the inclusion of active times.

(P3 $\Rightarrow$ P2) Assume that there exists a pair of vertices x, y, a path $P_{x,y}$ defined by $x = x_0, x_1, x_2, \dots, x_{k-1}, x_k = y$ which is white at start(x), and let us also assume that y is not a descendant of x in T. We may assume that the pair x, y is chosen so that $P_{x,y}$ is as short as possible. This implies that vertices $x_1, \dots, x_{k-1}$ are all descendants of x, which in turn implies time$(x_{k-1}) \subseteq$ time(x) (by (P1) applied to x and x_{k-1} taking into account that k might be 1). As being a descendant is a transitive relation we may assume that y is not a descendant of x_{k-1}. Hence time$(x_k) \cap$ time$(y) = \emptyset$. Now start$(y) >$ finish(x_{k-1}) contradicts the definition of BFSR, and finish$(y) <$ start$(x_{k-1})(<$ finish$(x_{k-1}) <$ finish$(x))$ implies that time$(y) \not\subset$ time(x) as y is not a descendant of x nor finish$(y) <$ start(x) as y is white at time start(x). This is a contradiction. □

1.5 Applications of DFS: topological sort and strongly connected components

In this section we will show how to use DFS to topologically sort vertices of a dag and how to compute strongly connected components of a directed graph using two runs of DFS.

Proposition 1.9 *$\overrightarrow{G}$ admits a topological ordering if and only if $\overrightarrow{G}$ is a directed acyclic graph.*

□

Let us start with a DFS-based computation of a topological ordering. Let L be a list, which is initially empty. TopologicalSort can be defined with a single (albeit not simple) line of pseudocode.

```
TopologicalSort($\overrightarrow{G}$)
1 run DFSr($\overrightarrow{G}$) and at the finish of each DFSr($\overrightarrow{G}, v$) prepend vertex $v$ to $L$
```

Algorithm 1.5: TopologicalSort

Observe that at the end of the run the list L contains vertices of G sorted according to descending finishing time. The following proposition establishes correctness of Algorithm 1.5

Proposition 1.10 *Let $\overrightarrow{G}$ be a dag, and let* DFSR *compute the finishing times of its vertices. If $xy \in E(\overrightarrow{G})$ then* finish$(x) >$ finish(y).

Proof. If $xy \in E(\overrightarrow{G})$ then $y \not\rightsquigarrow x$, as $\overrightarrow{G}$ is a dag. Hence, at no time there exists a white $y \rightarrow x$-path, and by Theorem 1.8 x is not a descendant of y, or equivalently

$\text{time}(x) \not\subset \text{time}(y)$. Now $\text{start}(y) > \text{finish}(x)$ contradicts the run of DFSR, hence on one hand $\text{start}(y) < \text{finish}(x)$. Together with $\text{time}(x) \not\subset \text{time}(y)$ this implies $\text{finish}(y) < \text{finish}(x)$. □

Let $\overrightarrow{G}$ be a directed graph. A *component graph* of $\overrightarrow{G}$, also called *condensation* of $\overrightarrow{G}$, is a directed graph $\overrightarrow{G}^{\text{cond}}$ defined in the following way: vertices of $\overrightarrow{G}^{\text{cond}}$ are strongly connected components $C_1, C_2, \ldots, C_\ell$ of $\overrightarrow{G}$, and $C_iC_j \in E(\overrightarrow{G}^{\text{cond}})$ if there exists an edge $x_ix_j \in E(\overrightarrow{G})$, so that $x_i \in V(C_i)$ and $x_j \in V(C_j)$.

Let $\overrightarrow{G}$ be a directed graph. Its *reverse graph* $\overleftarrow{G}$ is obtained by reversing orientations of all edges of $\overrightarrow{G}$. Observe that strongly connected components of both $\overrightarrow{G}$ and $\overleftarrow{G}$ are the same. Namely, if a pair of vertices x, y are mutually reachable, then they are also mutually reachable if we reverse all edge orientations.

We can compute strongly connected components in two consecutive runs of depth first search. Let L be a list, which is initially empty.

```
StronglyConnectedComponents($\overrightarrow{G}$)
1 run DFSr($\overrightarrow{G}$) and at the finish of each DFSr($\overrightarrow{G}, v$) prepend vertex $v$ to $L$
  reverse edges to compute $\overleftarrow{G}$;
2 run DFSr($\overleftarrow{G}$) in the order as the vertices appear in $L$ and compute the
  search forest $T$;
3 vertex-sets of components of $T$ induce strongly connected components of $\overrightarrow{G}$
```

Algorithm 1.6: StronglyConnecteComponents

Before proving correctness of the above algorithm we need to establish some technical results.

Proposition 1.11 *Let C_1 and C_2 be distinct strongly connected components of $\overrightarrow{G}$. If there exists an edge $x_1x_2 \in E(\overrightarrow{G})$, then no vertex of C_1 is reachable from a vertex of C_2.*

Proof. Assume that $y_2 \rightsquigarrow y_1$ for some $y_1 \in V(C_1)$ and $y_2 \in V(C_2)$. As $x_2 \rightsquigarrow y_2$ and $y_1 \rightsquigarrow x_1$, vertices x_1 and x_2 are mutually reachable. This contradicts the fact that they lie in different strongly connected component. □

An immediate consequence of Proposition 1.11 is the next corrolary which we state without its proof.

Corollary 1.12 *Every component graph $\overrightarrow{G}^{\text{cond}}$ is a dag.*

Let us first extend starting and finishing times to subgraphs of $\overrightarrow{G}$. If $\overrightarrow{H} \subseteq \overrightarrow{G}$, then its *starting time* $\text{start}(\overrightarrow{H})$ is defined as $\text{start}(\overrightarrow{H}) = \min\{\text{start}(v) \mid v \in V(\overrightarrow{H})\}$.

Similarly we define its *finishing time* $\text{finish}(\overrightarrow{H})$ as $\text{finish}(\overrightarrow{H}) = \max\{\text{finish}(v) \mid v \in V(\overrightarrow{H})\}$.

We shall first compare finishing times of adjacent strongly connected components.

Proposition 1.13 *Let C_1 and C_2 be distinct strongly connected components of $\overrightarrow{G}$. If there exists an edge $x_1x_2 \in E(\overrightarrow{G})$ with $x_1 \in V(C_1)$ and $x_2 \in V(C_2)$, then* $\text{finish}(C_2) < \text{finish}(C_1)$.

Proof. Let $x \in V(C_1) \cup V(C_2)$ be the vertex satisfying $\text{start}(x) = \min\{\text{start}(C_1), \text{start}(C_2)\}$. If $x \in V(C_1)$ then for every vertex $y \in V(C_2)$ there exists a directed $x \to y$-path whose vertices are at time $\text{start}(x)$ all white. Hence every vertex of C_2 is a descendant of x, and $\text{finish}(C_2) < \text{finish}(y) \le \text{finish}(C_1)$.

Next assume that $x \in V(C_2)$. As above, for every vertex $y \in V(C_2)$ there exists a directed $x \to y$-path whose vertices are at time $\text{start}(x)$ all white. Hence, by nesting of active times we have $\text{finish}(C_2) = \text{finish}(x)$. Let x_1 be an arbitrary vertex of C_1. By the choice of x we have $\text{start}(x) < \text{start}(x_1)$, and since x_1 is not a descendant of x (by Theorem 1.8 and Proposition 1.11, as $x \not\to x_1$) we have $\text{finish}(x) < \text{finish}(x_1)$, and the proof is finished. $\square$

We are now in the position to prove correctness of Algorithm 1.6.

Theorem 1.14 $\textsc{StronglyConnectedComponents}(\overrightarrow{G})$ *correctly computes strongly connected components of $\overrightarrow{G}$.*

Proof. Let $C_1, C_2, C_3, \ldots, C_\ell$ be the strongly connected components sorted by decreasing finishing time implicitly computed on line 1 of Algorithm 1.6. Proposition 1.13 states that they are indeed topologically ordered. As the routine $\textsc{DFSr}(\overleftarrow{G}, x)$ cannot visit a proper nonempty subset of vertices of some strongly connected component we shall inductively assume that the call $\textsc{StronglyConnectedComponents}(\overrightarrow{G})$ has at some point correctly identified strongly connected components $C_1, \ldots, C_{k-1}$, and did not visit vertices outside these components. The induction basis is trivial.

As an inductive step we only need to see that the search tree T_k produced by the call $\textsc{DFSr}(\overleftarrow{G}, v_k)$ (where v_k is an arbitrary vertex of C_k) satisfies $V(T_k) = V(C_k)$.

Clearly $V(C_k) \subseteq V(T_k)$, as all vertices of C_k are reachable from v_k and are also white at time $\text{start}(v_k)$. Let $xy \in \overleftarrow{G}$ so that $x \in V(C_k)$ and $y \in V(C_j)$ for some $j \ne k$. By Proposition 1.13 we have $\text{finish}(C_k) < \text{finish}(C_j)$ or equivalently $j < k$. This implies that y is black. Hence at time $\text{start}(v_k)$ every white path starting at v_k ends in a vertex of C_k. Theorem 1.8 implies that $V(T_k) \subseteq V(C_k)$, which finishes the proof. $\square$

2 Paths, flows, and connectivity

Imagine that we want to transfer information originating in a vertex u to a distant vertex v. As information can only be passed along edges, a single $u \to u'$-path suffices for the task. On the other hand there might be an attacker trying to stop the information transfer by either deleting edges or deleting vertices. If we can find several independent ways to transfer information then an attacker possible of compromising only a few structures in our graph cannot stop the information flow.

We will try to look at the problem from both parties' ways. On one hand we want to find as many independent $u \to u'$-paths in G, on the other we shall look for a structure as small as possible, whose removal inhibits the spread of information.

2.1 Menger theorems

Let G be a (directed) graph, and let u and u' be different vertices of G. We say that $u \to u'$ paths P_1 and P_2 are *edge disjoint* if $E(P_1) \cap E(P_2) = \emptyset$, and P_1 and P_2 are *internally disjoint* if $V(P_1) \cap V(P_2) = \{u, u'\}$. In other words, internally disjoint $u \to v$-paths do not share vertices nor edges, apart from their endvertices.

Assume that we want to find a collection $\mathcal{P}$ of edge-disjoint $u \to u'$-paths whose cardinality is as large as possible. Is there an easy upper bound on $|\mathcal{P}|$? Clearly $|\mathcal{P}| \leq \text{outdeg}(u)$ and also $|\mathcal{P}| \leq \text{indeg}(u')$. More generally, let us partition vertices of G into two parts, one containing u and the other containing u'. The number of edges from the first part to the second is also an upper bound for the number of paths in $\mathcal{P}$.

To be precise given vertices u and u', an *u, u'-cut* is an edge set $E(U, U') = \{xy \in E(G) \mid x \in U \text{ and } y \in U'\}$, for which U, U' is a partition of $V(G)$ ($U \cup U' = V(G)$, $U \cap U' = \emptyset$) so that $u \in U$ and $u' \in U'$.

Theorem 2.1 (Menger, 1.0) *Let G be a (directed) graph, u and u' distinct vertices. The maximal number of edge disjoint $u \to u'$-paths is equal to the size of the smallest u, u'-cut.*

We shall postpone the proof of Theorem 2.1 until later. But let us at this point stress the duality aspect: for every collection of edge-disjoint $u \to u'$-paths $\mathcal{P}$ and every u, u'-cut $\mathcal{C}$ we have $|\mathcal{P}| \leq |\mathcal{C}|$. Now an equality implies optimality of both, not only that $\mathcal{P}$ is as large as possible, but also that $\mathcal{C}$ is as small as possible.

But the story goes on. Let G be an undirected graph and assume that $S \subseteq V(G - u - u')$ is a set of vertices so that vertices u and u' lie in different components of $G - S$. This implies that every $u - u'$-path of G uses a vertex of S. We call such a vertex set S a *u, v-separator*. As two internally disjoint $u - u'$-paths cannot share a vertex from S, $|S|$ is clearly an upper bound on the number of internally disjoint $u - u'$-paths. Menger has proven yet another version of the theorem (and its proof we shall again postpone until later):

Theorem 2.2 (Menger, 2.0) *Let G be an undirected graph and u and u' distinct nonadjacent vertices. The maximal number of internally disjoint $u - u'$-paths is equal to the order of the smallest u, u'-separator.*

As above, the importance lies in the duality. Clearly the number of internally disjoint $u - u'$-paths cannot exceed the size of any u, u'-separator. The essence of Theorem 2.2 lies in the equality in the extremal cases.

Is nonadjacency really necessary? Well, if u and u' were adjacent, they cannot be separated by deleting vertices only.

But what if one would like to have truly disjoint paths? Let A, B be vertex sets and let us look for collections of disjoint $A - B$-paths in an undirected graph G. Clearly both $|A|$ and $|B|$ are upper bounds on the cardinality of such a collection.

Note that in case A and B are not disjoint, the singleton paths from $A \cap B$ form a collection of $A - B$-paths, and for the rest we can look for $A \setminus B - B \setminus A$-paths between disjoint vertex sets in $G - (A \cap B)$.

Menger has an appropriate result in this case as well, and we apologize once more for postponing the proof.

Theorem 2.3 (Menger, 3.0) *Let A, B be vertex sets of an undirected graph G. The maximal number of disjoint $A - B$-paths is equal to the order of smallest vertex set S, so that $G - S$ contains no $A - B$-paths.*

Note that in Theorem 2.3 we allow S to contain vertices from $A \cup B$.

2.2 Flows

The above Menger theorems (there are more to come), though dealing with different types of graph structure, appear to be similar enough so a common approach should take care of all their proofs. This is indeed the case, but we will have to make a detour into the class of weighted graphs.

Again let us use the transfer of information example. Two links between nodes in a network may have different bandwidths. Two pipes in a water supply network may have different diameters. The amount of information or water flowing on a connection may differ from one place to another, but in principle can be anything between zero and the connection's capacity.

Hence, weighted graphs. A *weighted graph G^w* , is a directed graph together with a mapping

$$w : E(\overrightarrow{G}) \to \mathbb{R}^+.$$

The *weight* of a directed edge uv is its w-value $w(uv)$.

We say that weights are *symmetric* if $w(uv) = w(vu)$ for every directed edge. In this case we shall talk about *weighted undirected graphs*. Similarly, an unweighted graph (directed or undirected) may be modeled with *0/1 weights*. We shall call weights of edges also *capacities*, and regard an edge uv with $w(uv) = 0$ as a nonedge in G^w.

Let G^w be a weighted graph and s, t vertices, called *source* and *sink* respectively. An *s-t-flow f* is a mapping

$$f : E(G^w) \to \mathbb{R}^+$$

satisfying

(F1) $0 \le f(uv) \le w(uv)$ and

(F2) $\sum_{x \in N(v)} f(vx) - \sum_{v \in N(y)} f(yv) = 0$ for every vertex $v \ne s, t$.

Condition (F1) states that edge-flow is nonnegative and bounded from above by edge-weight, and conditions (F2) are also called *Kirchhoff's laws*, the inflow at every vertex v (other than the source and the sink) is equal to the outflow.

The *value* of the flow f, $|f|$, is defined as the outflow at the source: $|f| = \sum_{x \in N(s)} f(sx)$, with an additional assumption that the inflow at s is equal to 0.

Clearly the *zero flow* satisfies both (F1) and (F2) but we will be looking for the other extreme, maximizing the value $|f|$. We will implicitly assume $f(uv) \cdot f(vu) = 0$ for every pair of counter oriented edges uv and vu. If $\delta = \min\{f(uv), f(vu)\} > 0$, then decreasing the flow f on both uv and vu by δ does not change its value and the resulting flow still satisfies both (F1) and (F2).

The dual concept of a flow is a *cut*. Let G^w be a weighted graph and s, t vertices. An *s, t-cut* is an edge set $E(U, U')$ for some partition of vertices $\{U, U'\}$ of $V(G)$ so that $s \in U$, $t \in U'$.

The *capacity of a cut $E(U, U')$* is the sum of capacities of edges in $E(U, U')$.

$$w(E(U, U')) = \sum_{uv \in E(U,U')} w(uv)$$

The capacity of a cut limits the amount of flow going from U to U':

Proposition 2.4 *Let $E(U, U')$ be an s, t-cut and let f be an $s - t$-flow. Then*

$$|f| \le w(E(U, U'))$$

Our alternative goal is to minimize $w(E(U, U'))$, as by Proposition 2.4 the capacity of a cut is an upper bound for the value of a flow, and the smaller capacity the better the upper bound.

Let $E(U, U')$ be an s, t-cut. The *flow across $E(U, U')$* is defined as

$$f(E(U, U')) = \sum_{uv \in E(U,U')} f(uv) - \sum_{vu \in E(U',U)} f(vu)$$

Proposition 2.5 *Let G^w be a weighted graph, and f an arbitrary $s-t$-flow. Then every s,t-cut $E(U,U')$ satisfies*

$$f(E(U,U')) = |f|.$$

Proof. Let us compute

$$\sum_{v\in U}\left(\sum_{x\in N(v)} f(vx) - \sum_{v\in N(y)} f(yv)\right) \tag{2.1}$$

On one hand the above sum (2.1) equals

$$\sum_{uv\in E(U,U')} f(uv) - \sum_{vu\in E(U',U)} f(vu) = f(E(U,U'))$$

as the contribution of an edge xy whose both endvertices x and y lie in U cancels out, xy carries inflow to y and outflow from x. On the other hand (2.1) is equal to

$$\sum_{x\in N(s)} f(sx) - \sum_{s\in N(y)} f(ys) = |f| - 0 = |f|,$$

as Kirchhoff laws apply at every vertex $v \in U \setminus \{s\}$. □

We are now ready to state the main result.

Theorem 2.6 (Ford-Fulkerson) *Let G^w be a weighted graph, s,t vertices. Then*

$$\max|f| = \min w(E(U,U')),$$

where the max *ranges over all $s-t$-flows an* min *ranges over all s,t-cuts.*

Rather than giving a mathematical proof we shall describe an algorithm whose output will be both a flow f^* and a cut $E(U^*,U^{*\prime})$, so that $|f^*| = w(E(U^*,U^{*\prime}))$ implying that both f^* and $E(U^*,U^{*\prime})$ are optimal.

Given an $s-t$-flow f in a weighted graph G^w let us define the *residual graph* $\mathrm{Res}(G,f)$, describing the amount of unused capacities of edges. $\mathrm{Res}(G,f)$ is a

(R0) weighted graph on the same vertex set as G^w with weights w_{Res},

(R1) if $f(uv) > 0$ then $w_{\mathrm{Res}}(uv) = w(uv) - f(uv)$ and
$w_{\mathrm{Res}}(vu) = w(vu) + f(uv)$,

(R2) if $f(uv) = 0$ and $f(vu) = 0$ then $w_{\mathrm{Res}}(uv) = w(uv)$.

Residual graph indicates by how much and in which direction we may alter the flow. If $w_{\mathrm{Res}}(uv) = 0$ we shall implicitly assume that $uv \notin E(\mathrm{Res}(G,f))$.

Now $\mathrm{Res}(G,f)$ can be computed using G^w and f, but also vice versa. The flow f can be determined from G^w and $\mathrm{Res}(G,f)$:

$$f(uv) = \max\{w(uv) - w_{\mathrm{Res}}(uv), 0\}$$

The Ford-Fulkerson algorithm 2.1 tries to push additional flow along a directed $s \to t$-path in the residual graph $\mathrm{Res}(G,f)$ starting with the zero flow. If $\mathrm{Res}(G)$ contains no directed $s \to t$-path, then the set of vertices which are reachable from s in $\mathrm{Res}(G,f)$ and its relative complement determine the minimal cut.

```
   FordFulkerson($G^w, s, t$)
1  $\mathrm{Res}(G,f) = G^w$;
2  $maximal$ := $False$;
3  repeat
4      $T \leftarrow$ BFS($\mathrm{Res}(G,f), s$);
5      if $t \in V(T)$ then
6          UpdateFlow($\mathrm{Res}(G,f), T, s, t$)
7      else
8          $maximal$ := $True$
9  until $maximal = True$;
10 return $\max\{G^w - \mathrm{Res}(G,f), 0\}$, $E(V(T), V(G^w - T))$
```

```
   UpdateFlow($\mathrm{Res}(G,f), T, s, t$)
1  let $P$ be the $s \to t$-path in $T$;
2  $\delta = \min\{w_{\mathrm{Res}}(uv) \mid uv \in E(P)\}$;
3  foreach $uv \in E(P)$ do
4      $w_{\mathrm{Res}}(uv) = w_{\mathrm{Res}}(uv) - \delta$;
5      $w_{\mathrm{Res}}(vu) = w_{\mathrm{Res}}(vu) + \delta$;
```

Algorithm 2.1: Ford-Fulkerson algorithm for computing maximal flow and minimal cut, and a subroutine.

Theorem 2.7 *Upon finishing* FordFulkerson *returns a maximal flow and a minimum cut.*

Proof. Let $V(T)$ denote the vertex set the algorithm FordFulkeson outputs at line 10. $V(T)$ is exactly the set of all vertices reachable from s in the final residual graph $\mathrm{Res}(G,f)$. Hence $\mathrm{Res}(G,f)$ contains no edge from $V(\mathrm{Res}(G,f)-T)$ to $V(T)$, or more formally, if $v \in V(\mathrm{Res}(G,f) - T)$ and $u \in V(T)$, then $w_{\mathrm{Res}}(uv) = 0$. This implies that $f(uv) = w(uv)$, and consequently

$$\begin{aligned} w(E(V(T), V(\mathrm{Res}(G,f) - T))) &= \sum_{uv \in E(V(T), V(\mathrm{Res}(G) - T))} w(uv) \\ &= \sum_{uv \in E(V(T), V(\mathrm{Res}(G,f) - T))} f(uv) = |f|, \end{aligned}$$

by observing that a flow across an arbitrary cut is equal to $|f|$, see Proposition 2.5 □

Theorem 2.8 (flow integrality) *Let G^w be an integral weighted graph. Then* FordFulkerson *computes an integral maximal flow.*

Proof. This follows immediately by observing that integral weights of $\mathrm{Res}(G)$ at the start of the algorithm imply that these weights remain integral, as in each run δ computed on line 2 of UpdatePath is an integer. $\square$

Theorem 2.9 *Let G^w be an integral weighted graph and let $k = \max\{|f|\}$ be the optimal flow value. Then* FordFulkerson *runs in*

$$O(k(n+m))$$

time.

Proof. The return loop on line 3 is repeated at most k times, as each run increases $|f|$ by at least 1. As computing the search tree T as well as the call of UpdatePath takes $O(n+m)$ time, the total time used is $O(k(n+m))$. $\square$

2.3 Back to Menger theorems and connectivity

Let us now take care of the proofs of Theorems 2.1, 2.2, and 2.3 by using flow results on carefully constructed graphs.

Proof.[of Theorem 2.1] We compute the maximal set of edge-disjoint $u \to v$-paths in a directed graph $\overrightarrow{G}$ by solving the $u-v$-flow problem in $\overrightarrow{G}$, where we consider $\overrightarrow{G}$ as a weighted graph with 0/1 weights. A maximal flow f is by Theorem 2.8 integral and can be, as nontrivial eights are equal to 1, interpreted as an edge set $F = \{e \in E(\overrightarrow{G}) \mid f(e) = 1\}$.

The edge disjoint paths can be constructed inductively using only edges of F: each additional path P deletes $E(P)$ from F. $\square$

Proof.[of Theorem 2.2] Let $\overrightarrow{G}$ be a directed graph, u and v nonadjacent vertices. Let G' be the graph obtained by the following procedure:

1. delete all incoming edges to u and rename u to u_{out},
2. delete all outgoing edges from v and rename v to v_{in},
3. replace each vertex $x \neq u, v$ with a pair of vertices x_{in}, x_{out},
4. add an edge $x_{in}x_{out}$ of weight 1 between each pair of vertices x_{in}, x_{out},

5. replace each original edge xy with an edge $x_{out}y_{in}$ and set its weight to 2

Let f' be a maximal $u_{out} - v_{in}$ flow in G'. By 2.8 we may assume that f' is integral. As each vertex $x \neq u_{out}, v_{in}$ has either only one outgoing or only one ingoing edge whose weight is equal to 1, the flow f' on a single edge cannot exceed 1. As in the above proof f' can be interpreted as a subset of edges of G', and edge disjoint paths $P'_1, P'_2, \dots, P'_k$ can be found inductively.

These paths can be lifted to $u \to v$-paths $P_1, P_2, \dots, P_k$ in G, and are by construction edge-disjoint. If P_i and P_j share a common internal vertex x, then P'_i and P'_j both use the $x_{in}x_{out}$ edge, which is impossible.

In fact the minimum u_{out}, v_{in}-cut in G' contains only edges of the form $x_{in}x_{out}$, and this cut lifts to a u, v-separator in G. □

Proof.[of Theorem 2.3] Let G^{++} be a graph obtained from G by adding a pair of new vertices a and b, so that a is adjacent to every vertex of A, and every vertex of B is adjacent to b. The internally disjoint $a - b$-paths in G^{++} correspond to disjoint $A - B$-paths in G. □

Let G be, for the rest of this section, an undirected graph. Let us for technical reasons also assume that G is connected. Without a reference to a pair of fixed vertices we can define *edge-* and *vertex-connectivity* of a graph.

A *cut* in G is a subset of edges F so that $G - F$ is disconnected (in case G is disconnected a cut is a set of edges whose removal increases the number of connected components, but at this point we do not want to enter such technical details). A cut F in G is *minimal* if no proper subset of F is a cut, and a cut F is a *smallest* cut in G if G contains no cut F' with $|F'| < |F|$. Clearly a smallest cut is also a minimal one, but the converse might not be true.

Observe also that if F is a minimal cut in G, then $G - F$ has exactly two connected components, and that the only connected graph without a cut is a singleton graph K_1.

A *separator* in a connected graph G is a vertex set S so that $G - S$ is disconnected. A separator S in G is *minimal* if no proper subset of S is a separator, and a separator S is a smallest one, if G contains no separator S' with $|S'| < |S|$. As above every smallest separator is also a minimal one, but the converse might no be true.

If S is a minimal separator, then every vertex $s \in S$ has a neighbor in every component of $G - S$. Note also that complete graphs contain no separators.

Let G be a graph on at least 2 vertices. We say that G is *k-edge-connected* if G contains no cuts of size $< k$. *Edge-connectivity of G, $\lambda(G)$*, is the largest integer ℓ so that G is ℓ-edge-connected.

Theorem 2.10 (Menger, 4.0) *Let G be a graph on at least 2 vertices. G is k-edge connected if and only if for every pair of vertices u, v G contains at least k*

edge-disjoint $u - v$-paths.

Let G be an undirected graph. We say that G is *k-vertex-connected* or just *k-connected* if G contains at least $k+1$ vertices and G contains no separators of $< k$ vertices. *(Vertex)-connectivity of G, $\kappa(G)$*, is the largest integer k so that G is k-(vertex)-connected.

Theorem 2.11 (Menger, 5.0) *Let G be an undirected graph on at least 2 vertices. G is k-connected if and only if for every pair of vertices u, v G contains at least k internally-disjoint $u - v$-paths.*

Where is the condition on $k+1$ vertices in a k-connected graph? Well, if x and y are distinct vertices of G (mind you, such vertices exist as G contains at least two vertices) then the k internally-disjoint $x - y$-paths contain together with x and y at least $k+1$ vertices (one of the paths may be a single edge).

2.4 Biconnectivity and blocks

Let G be a connected graph. What are the possible reasons that G is not 2-connected? Either G is too small, G has only 1 or 2 vertices, or G contains a separator of order 1. A single vertex s for which $G - s$ is disconnected is also called a *cutvertex* (and a single edge e, so that $G - e$ is disconnected is called a *cutedge*).

It is natural to define *blocks* of G as maximal subgraphs of G without cutvertices. Hence, a block B of G can be either

(B0) an isolated vertex $B \equiv K_1$, or

(B1) a cutedge together with its endvertices $B \equiv K_2$, or

(B2) a maximal 2-connected subgraph of G (which we may sometimes call a *proper block*).

Let G be a connected graph. The *block tree* of G, *$B(G)$*, is a graph with

(BT1) vertices of $B(G)$ being cutvertices and blocks of G, and

(BT2) a cutvertex x and a block B are adjacent in $B(G)$ if and only if $x \in V(B)$.

By construction a block tree is a bipartite graph. As the name suggests it is also a tree.

Proposition 2.12 *If G is a connected graph then $B(G)$ is a tree.*

Proof. Assume that $B_0x_0B_1x_1\dots x_{k-1}B_0$ is a cycle in $B(G)$. Let b_0, b_1 be neighbors of x_0 from B_0 and B_1 respectively. As x_0 is a cutvertex in G vertices b_0 and b_1 lie in different components of $G - x_0$. On the other hand b_0 is reachable from b_1 following a walk along C avoiding x_0, which is a contradiction. □

There is an alternative way of defining blocks of G, if we forget about isolated vertices. Let us define a relation R on the edges-set of G with

$$eRf \iff e = f \text{ or } e \text{ and } f \text{ lie on a common cycle.}$$

Now R is an equivalence relation on $E(G)$ (transitivity takes some effort, let us postpone it until the next time) and partitions the edge set into equivalence classes of edges. A singleton equivalence class consists of an edge e, that lies on no cycle, such an edge is a cutedge. Now a block is a subgraph of G induced by endvertices of edges in from a single equivalence class.

Our next computational task is to compute cutvertices, and consequently blocks of G, in linear time. We shall do this by computing an additional vertex parameter low(.) using depth first search.

Let G be a connected undirected graph, and let T be its DFS tree with root r. Every edge $uv \in E(G)$ is either a *tree edge* (one of u, v is the son of the other in T) or a *backward edge* (equivalently a forward edge, one of u, v is an ancestor (not immediate) of the other). There can be no cross edges in an undirected graph G.

Assume that we have computed start times of vertices of G: low(v) is the smallest start(x) over all vertices x which can be reached from v using tree edges away from the root r with a possible single backward edge at the end.

We can compute low(.) recursively. When computing low(v) we shall recursively assume that we have determined low(x) for every descendant x of v.

```
   DFSLow($G, v$)
1  color $v$ black;
2  start($v$) = low($v$) = $step$;
   // step is global and initially set to 1
3  $step$++;
4  foreach $u \in N(v)$ do
5      if $u$ is white then
6          $T = T + uv$;
7          DFSLow($G, v$)
8  foreach $u \in N(v)$ do
9      if $vu \in E(T)$ and $u$ is a son of $v$ then
10         low($v$) = $\min\{\text{low}(v), \text{low}(u)\}$
11     else
12         low($v$) = $\min\{\text{low}(v), \text{start}(u)\}$
```

Algorithm 2.2: DFSLow recursively computes low(v) for every vertex v.

Proposition 2.13 *Algorithm 2.2* DFSLow *correctly computes* low(v) *for every vertex* $v \in V(G)$.

Proof. If v is a leaf of T then v is incident with no tree edges away from the root, hence $\text{low}(v)$ equals the smallest $\text{start}(x)$ over its backward neighbors, which is correctly computed in line 12. Note that in this case line 10 does not apply as no vertex is a son of a leaf v in T.

Let us now argue that DFSLow correctly computes $\text{low}(v)$ for every nonleaf vertex v. Recursively we may assume that $\text{low}(x)$ is correctly computed for every descendant of v, in particular for every son of v. Now a nonstationary path along tree edges starting from v necessarily goes through a son of v, whose $\text{low}(v)$ is by assumption correctly computed. Hence unless $\text{low}(v) = \text{start}(x)$ it is either equal to the minimal $\text{low}(u)$ over all sons of v, this is computed on line 10, or is the smallest $\text{start}(x)$ over backward edges emanating from v, which is computed on line 12. □

Let us finish with the characterization of cutvertices, which can be computed using DFSLow.

Theorem 2.14 *Let G be a connected undirected graph, and let r be the* root *of its BFS tree T:*

- *r is a cutvertex if it is incident with ≥ 2 tree edges, and*
- *a nonroot vertex v is a cutvertex if v has a son y so that $\text{low}(y) \geq \text{start}(v)$.*

Proof. We know that G contains no cross edges with respect to T. Hence if x and y are different sons of r in T, then every $x - y$-path in G uses r, which makes r a cutvertex. Now if r has only one son in T, then $T - r$ is a spanning tree of $G - r$. Hence $G - r$ is connected and r is not a cutvertex.

Let us turn to a nonroot vertex $v \neq r$. Assume first that there exists a vertex y, which is a son of v, so that $\text{low}(y) \geq \text{start}(v)$. Let Y be the set of descendants of y (including y itself). We claim that there is no edge between vertices of Y and vertices of $G - v - Y$. Assume to the contrary that there exists a $y'x$ edge for some $y' \in Y$ and x in $G - v - Y$. As G has no cross edges and $y'x$ is not a tree edge we conclude that $y'x$ is a backward edge. Now $x \notin Y$ implies that x is an ancestor of v, and also an ancestor of y. Hence $\text{low}(y) \leq \text{start}(x) < \text{start}(v)$ which is a contradiction, proving that v is indeed a cutvertex.

Fix a vertex v and let C be an arbitrary component of $G - x$. As T is a spanning tree there exists a vertex $y \in V(C)$ so that either $yv \in E(T)$ or $vy \in E(T)$, in other words v is a son of y or vice versa. Assume first that y is a son of v, and that $\text{low}(y) < \text{start}(x)$. This implies that a descendant of y (or y itself) has a neighbor among ancestors of x, and hence $r \in C$. If v is a son of y then also C contains r. This shows that if $\text{low}(y) < \text{start}(v)$ for every y which is a son of v, then every component of $G - v$ contains r. Hence v is not a cutvertex. □

At the very end, let us state, without proof (this should be obvious as we know the running time of DFS).

Theorem 2.15 *Let G be an undirected graph. We can compute cutvertices of G in linear time $O(n + m)$.*

3 Constructing 2-connected graphs

3.1 Biconnectivity augmentation problem

Let G be an undirected graph (we shall only discuss undirected graphs in this section). If G is not connected and $C_1, C_2, C_3, \ldots, C_k$ are its components, then we can make G connected by adding a set of $k-1$ edges to G. An edge between two vertices from different components decreases the component count by exactly one. This also implies that $k-1$ is an optimal (minimal) number of edges one needs to add to G in order to make it connected.

The problem of making a graph 2-connected (or *biconnected*, this term is more often used in this setting) is a more difficult one:

Biconnectivity augmentation problem or *BAP*

input: undirected connected graph G.

output: a set of undirected edges F, so that $G+F$ is 2-connected (also biconnected) and $|F|$ is minimal.

The optimal edge set F shall also be called an *augmenting edge set*.

Traditionally one can observe the problem in a larger class of not-necessarily-connected graphs, but for our purpose we shall limit ourselves to connected input graphs.

Our first result indicates that when inserting a single edge e we need not to be very picky about its endvertices.

Proposition 3.1 *Let G be a connected graph, let B and $\bar{B}$ be blocks, and let $x, y \in V(B)$ and $\bar{x}, \bar{y} \in V(\bar{B})$ be* non-cutvertices *(in fact it is enough to require that $x, \bar{x}, y, \bar{y}$ are not contained on a $B-\bar{B}$ path in $B(G)$). Then the graphs $G+x\bar{x}$ and $G+y\bar{y}$ have the same block structure. More precisely, let $B, c_2, B_2, c_3, B_3, \ldots, B_{k-1}, c_{k-1}, \bar{B}$ be the unique $B-\bar{B}$-path in $B(G)$. Then the addition of both $x\bar{x}$ or $y\bar{y}$ to G constructs a graph G' which contains*

1. *a new block B' with $V(B') = V(B) \cup V(B_2) \cup \ldots \cup V(B_{k-1}) \cup V(\bar{B})$ and*
2. *a vertex c_i is a cutvertex in G' if and only if $\deg_{B(G)}(c_i) \geq 3$ (and in this case $\deg_{B(G')} = \deg_{B(G)}(c_i) - 1$).*

Proof. We shall only do a sketch. Let $u, v \in V(B')$ be different vertices. Then u and v lie on a cycle which uses a newly added edge ($x\bar{x}$ or $y\bar{y}$), and consequently in the same block. If c_i is a cutvertex in G and $d = \deg_{B(G)}(c_i) \geq 3$, then c_i is adjacent to exactly $d-2$ blocks apart from B_{i-1} and B_i in $B(G)$. Now c_i is adjacent to B^* and the very same set of $d-2$ blocks in $B(G')$. □

We would like to make the block tree of the resulting graph as small as possible by adding a single edge. One possible measure is the number of blocks in the resulting graph. This implies that the $B-\bar{B}$-path between blocks should be (if not as long as possible) maximal in terms of containment. This can be achieved by choosing the blocks B and $\bar{B}$ among leaf blocks of $B(G)$.

Proposition 3.2 *Assume that G can be made 2-connected by adding a set of edges $e_1, e_2, \ldots, e_k$. Then G can be turned into a 2-connected graph by adding the same number of edges $f_1, f_2, \ldots, f_k$, so that for every $i \in \{1, \ldots, k\}$ the edge f_i connects vertices from two leaf blocks of $G + e_i + \cdots + e_{i-1}$.*

The text right above the Proposition serves as its proof. Let us note that we when looking for an optimal set of edges (to add in order to obtain a 2-connected graph G) we shall always act according to Proposition 3.2 by only adding edges between pairs of leaf blocks.

There is another consequence of Proposition 3.1: it shows that BAP is indeed a problem on the block tree of the graph $B(G)$, rather than the graph G itself. By taking Proposition 3.2 into account we shall with each leaf block $B \in V(B(G))$ store a vertex $v \in V(G)$ which is not one of cutvertices. We shall denote v as $\mathrm{vx}(B)$.

In view of this observation we shall skip the graph G altogether and will have only $B(G)$ in mind. Now vertices of $B(G)$ come in two flavors, blocks and cutvestices of the original graph. We shall also call them *b-vertices* and *c-vertices*, respectively. If v is a b- od c-vertex of $B(G)$ then by $\deg(v)$ we shall denote the *degree of v as a vertex in $B(G)$*. and not its degree as a possible vertex in G. (!!!)

Do b-and c-vertices behave differently with respect to the BAP? Imagine a graph G with a single cutvertex v, which is adjacent to six blocks $B_1, \ldots, B_6$. It is easy to see that we need to add at least 5 edges to G to make it 2-connected. On the other hand let G' be a graph having a central block B'_0, which is adjacent to six different blocks $B'_1, \ldots, B'_6$ via six different cutvertices of degree 2 (their degree in $B(G)$!!). Then adding a suitable set of 3 edges can make the resulting graph 2-connected.

The above analysis yields lower bound on size of the augmenting edge set $|F|$. If $B(G)$ contains a c-vertex v with $\deg(v) = d$, then $|F| \geq d-1$, as adding a single edge to G reduces $\deg(v)$ by at most 1. On the other hand let ℓ be the number of leaf blocks in G. Then $|F| \geq \lceil \ell/2 \rceil$, as adding a single edge may reduce the number of leaf blocks by at most 2.

It is surprising that the lower bound is in fact the exact one.

Theorem 3.3 (Eswaran, Tarjan) *Let G be a connected graph, ℓ the number of leaves in $B(G)$ and d the maximal degree of a c-vertex. Then there exists an edge set F of size*

$$\max\{d-1, \lceil \ell/2 \rceil\}$$

so that $G+F$ is a 2-connected graph, and no edge set of strictly fewer edges has this property.

We shall devote the rest of the section for the proof of the above theorem. In fact we shall present an efficient algorithm which will, given G or equivalently its block tree $B(G)$, find an augmenting set F of size $|F| = \max\{d-1, \lceil \frac{\ell}{2} \rceil\}$ in linear time.

Let ℓ denote the number of leaves in $B(G)$, and let d denote the maximal degree of a c-vertex.

Let us first consider some small cases, where the number of leaves ℓ in $B(G)$ is at most 4. If $\ell = 2$, then $B(G)$ is a path, and the degree of every c-vertex in $B(G)$ is equal to 2. By adding an edge between two leaf blocks we obtain a graph having a single block, see Proposition 3.1, which is 2-connected.

If $\ell = 3$ then $B(G)$ contains exactly one vertex of degree 3, and let B_1, B_2, B_3 be the three leaf blocks. In this case $d \leq 3$. The addition of two edges $\mathrm{vx}(B_1)\mathrm{vx}(B_2)$ and $\mathrm{vx}(B_2)\mathrm{vx}(B_3)$ turns G into a 2-connected graph.

The story shows a first complication if $\ell = 4$. Let B_1, B_2, B_3, B_4 be the leaf blocks. If $d = 4$ then the addition of edges $\mathrm{vx}(B_1)\mathrm{vx}(B_2)$, $\mathrm{vx}(B_2)\mathrm{vx}(B_3)$ and $\mathrm{vx}(B_3)\mathrm{vx}(B_4)$ makes G 2-connected. Otherwise let B_1 and B_2 be blocks so that the $B_1 - B_2$-path in $B(G)$ contains all vertices of degree ≥ 3 (there exist at most two such vertices). Now the addition of edges $\mathrm{vx}(B_1)\mathrm{vx}(B_2)$ and $\mathrm{vx}(B_3)\mathrm{vx}(B_4)$ makes G 2-connected.

Observe, that we have in all above *small cases* constructed an augmenting set of exactly $\max\{d-1, \lceil \frac{\ell}{2} \rceil\}$ edges.

Let us first state a graph theoretic tool. The number of leaves in a tree can be computed from set of vertices of higher degree.

Proposition 3.4 *Let T be a tree, and let ℓ be the number of its leaves. Then*

$$\ell = 2 + \sum_{\deg(v) \neq 1} (\deg(v) - 2) \tag{3.1}$$

Proof. A funny fact of the above formula is that it is true also in the trivial case, where T is having just one vertex. Such a tree has zero leaves and the expression on the right hand side also evaluates to zero.

It is trivial to observe that the above formula is valid in case T is a path, and also in the case T has exactly one vertex of degree ≥ 3.

Let us inductively assume that Proposition 3.4 holds for every tree with at most k vertices of degree ≥ 3. Let T be a tree with $k+1$ vertices of large degree and let e be an edge on a path between two such vertices. The edge $e = v_1v_2$, as it is not contained in a cycle, is a cutedge in T. Let T_1, T_2 be the components of $T - v_1v_2$, so that $v_1 \in V(T_1)$ and $v_2 \in V(T_2)$, an let us attach the edge v_1v_2 to both T_1 and T_2. (As if we would cut the edge e in half, treating both *halfedges* as pendant edges in each respective component.) Now $2 + 2 + \sum_{\deg(v)\neq 1}(\deg(v) - 2)$ is exactly the number of leaves in T_1 and T_2 together, and is on the other hand equal to $\ell + 2$. $\square$

Let us call a c-vertex v *massive* if $\deg(v) > \lceil \frac{\ell}{2} \rceil + 1$, and let v be *critical* if $\deg(v) = \lceil \frac{\ell}{2} \rceil + 1$. A *chain* in $B(G)$ is a path with one endvertex of degree 1 whose internal vertices are of degree 2. Let v be a vertex of degree ≥ 3. A *v-chain* is a chain with v as an endvertex, the other endvertex of degree 1 is also called a *v-chain leaf*.

Using Proposition 3.4 we can show that there cannot be too many massive or critical c-vertices in $B(G)$.

(1) *Let $\ell \geq 3$ be the number of leaves and let us also assume that the maximal degree of a c-vertex $d \geq 3$. Then*

- *if $B(G)$ contains a massive c-vertex v, then no other c-vertex can be either massive or critical,*
- *if $B(G)$ contains a critical vertex v, then at most one other c-vertex is critical, and*
- *if $B(G)$ contains two critical c-vertices u and v, then every other vertex of $B(G)$ has degree ≤ 2.*

The condition $d \geq 3$ is a natural one. If $B(G)$ is a path and contains 2 leaves, then a c-vertex is critical if it has degree 2. There can be a lot of such vertices in $B(G)$. On the other hand the definition itself implies that a massive vertex has degree at least 4.

Let us apply equation (3.1). Let v_1 and v_2 be two critical or massive vertices of $B(G)$ of degree ≥ 3.

$$\begin{aligned}
\ell &= 2 + \sum_{\deg(v)\neq 1} (\deg(v) - 2) \\
&= 2 + \sum_{\deg(v)\geq 2} (\deg(v) - 2) \\
&\geq 2 + (\deg(v_1) - 2) + (\deg(v_2) - 2) \\
&\geq 2 + \left\lceil \frac{\ell}{2} \right\rceil - 1 + \left\lceil \frac{\ell}{2} \right\rceil - 1 \\
&= 2 \cdot \left\lceil \frac{\ell}{2} \right\rceil \geq \ell
\end{aligned}$$

All of the above inequalities are indeed equalities. From this we infer that (1) there are exactly two vertices in $B(G)$ of degree ≥ 3, both (2) are critical and not massive and consequently (3) in this case the number of leaves is an even number. ♢

Next we shall show that a massive vertex v determines a sufficient number of v-chains.

(2) *Let v be a massive vertex in $B(G)$. Then there exist at least four v-chains.*

Let k be the number of v-chains, and let us count the number of leaves ℓ in $B(G)$. Each v-chain contains exactly one leaf, and the remaining $d-k$ subtrees of $B(G)-v$ contain at least $2(d-k)$ leaves. Hence

$$\ell \geq k + 2(d-k) = 2d - k \geq 2(\lceil \ell/2 \rceil + 2) - k \geq \ell + 4 - k \geq 4.$$

♢

(3) *Let v be a massive vertex of degree d in a block tree $B(G)$ with ℓ leaves. By inductively adding edges between pairs of v-chain leaves we can transform v to a critical vertex.*

If v is a massive vertex, then $d > \lceil \ell/2 \rceil + 1$. Observe the change of the expression $d - (\lceil \ell/2 \rceil + 1)$ when adding a single edge between v-chain leaves of $B(G)$. As both d and ℓ drop by exactly one, its value decreases by either 1 or zero, depending on the parity of ℓ. Hence $d - (\lceil \ell/2 \rceil + 1)$ will eventually be equal to 0, making v a critical vertex. $\Diamond$

Let us call $B(G)$ without massive vertices a *balanced* tree. We shall see that if $B(G)$ is a balanced tree with $\ell \geq 4$ leaves, we can find a pair of leaf blocks B_1 and B_2, so that the addition of the edge $\mathrm{vx}(B_1)\mathrm{vx}(B_2)$ to G reduces the number of leaves on $B(G)$ by 2 and does not introduce a massive vertex. We shall say that such blocks B_1 and B_2 satisfy the *leaf connecting condition*.

(4) *Let $B(G)$ be a balanced block tree of G with $\ell \geq 4$. The leaf blocks B_1 and B_2 satisfy the leaf connecting condition if the $B_1 - B_2$-path P contains all critical c-vertices, and P either contains two vertices of degree ≥ 3 or a b-vertex of degree ≥ 4.*

The newly added edge $\mathrm{vx}(B_1)\mathrm{vx}(B_2)$ creates a new block B'. If P contains two vertices of degree ≥ 3 or a c-vertex of degree ≥ 4, then B' will contain at least two cutvertices and will not become a leaf block itself. As the former leaf blocks B_1 and B_2 get merged into B' the number of leaf blocks decreases by 2. If v is a critical c-vertex in $B(G)$ which is not contained in B', equivalently it is not a vertex of P, then its degree does not drop, and v becomes a massive one. $\Diamond$

In order to give a full description of the algorithm and prove its efficiency let us describe the auxiliary data structures that we shall use.

1. $B(G)$ is a rooted tree, having a b-vertex B_0 as its root (let us denote the subtree of $B(G)$ rooted at v with $B(G)_v$),
2. an ordered list $\mathcal{C}$ of c-vertices sorted according to decreasing degrees,
3. a pair of sets $\mathcal{B}_{3+}$ and $\mathcal{B}_{2-}$ storing b-vertices of degree ≥ 3 and ≤ 2, respectively,
4. for every vertex $v \in V(B(G))$ and every son u of v let $\ell_{v,u}$ denote the number of leaves of $B(G)$ in $B(G)_u$,
5. for every vertex $v \in V(B(G))$ the number of leaves ℓ_v^- which are descendants of v.
6. for every $v \in V(B(G))$ the sets of S_{2+}^v and S_1^v containing the sons of v having ≥ 2 or exactly 1 leaf of $B(G)$ in their respective subtrees.
7. for every leaf block $B \in V(G(B))$ a vertex $\mathrm{vx}(B) \in V(G)$ which is not a cutvertex of G,

We shall compute the above initial structures in the preprocessing stage. We can compute $\ell_{v,u}$ using the bottom-up approach. We shall not require that the sons of a vertex v are sorted according to the number of leaves in their respective subtrees, yet we shall assume that we can in constant time pick a son u of v, so that the number of leaves in $B(G)_u$ is ≥ 2 or decide that such a son u does not exist. Similarly we can initially sort $\mathcal{C}$ in linear time, as the keys are integers between 1 and n.

```
BAP($B(G)$)
$L = \emptyset$;
while $B(G)$ contains a massive vertex $v$ do
    $P, B_1, B_2$ = FindPathMassive($v$);
    AddEdge($P, B_1, B_2$)
while $V(B(G)) \geq 2$ do
    $P, B_1, B_2$ = FindPathBalanced();
    AddEdge($P, B_1, B_2$)
return $L$
```

```
FindPathMassive($v$)
let $P_1$ and $P_2$ be $v$-chains and let $B_1$ and $B_2$ be the corresponding $v$-chain
leaves;
return $P_1 \cup P_2, B_1, B_2$
```

```
FindPathBalanced()
let $v$ be a $c$-vertex of maximal degree;
case deg($v$) $\geq 3$
    $x = v$
case $\mathcal{B}_{3+} \neq \emptyset$
    choose $x \in \mathcal{B}_{3+}$
otherwise
    $x = v$
if deg($x$) $\geq 3$ then
    if $B(G)$ contains $\geq 2$ vertices of degree $\geq 3$ then
        if $x$ has a descendant $y$ of degree $\geq 3$ then
            let $P_0$ be a $x - y$ path in $B(G)$;
            let $P_1$ be a $y - B_2$ path where $B_2$ is a leaf;
            let $P_2$ be a path from $x$ to root an (possibly, if its degree$\geq$ 1)
            towards another leaf $B_1$
        else
            let $P_2$ be a $x - B_2$ path where $B_2$ is a leaf descendant of $x$;
            let $P_0$ be a path from $x - y$ path where $y$ possibility lies on a
            path towards root;
            let $P_1$ be the path from $y$ towards a leaf $B_1$ which contains the
            root if $P_0$ does not
    else
        let $P_0 = \{x\}$;
        let $P_1$ and $P_2$ be $x$-chains and let $B_1$ and $B_2$ be the corresponding
        $x$-chain leaves;
if deg($x$) $= 2$ then
    $P_0 = \{x\}$;
    let $P_1$ and $P_2$ be $x$-chains and let $B_1$ and $B_2$ be the corresponding
    $x$-chain leaves;
return $P_1 \cup P_0 \cup P_2, B_1, B_2$
```

```
AddEdge($P, B_1, B_2$)
compute the new block $B'$;
update data structures;
$L = L \cup \{\text{vx}(B_1)\text{vx}(B_2)\}$
```

Algorithm 3.1: BAP and necessary subroutines.

Let us first FindPathBalanced. Note that x on line 17 is a critical vertex, if a critical vertex of degree ≥ 3 exists in $B(G)$. If $\deg(x) = 2$, then $B(G)$ does not contain vertices of degree ≥ 3 and, in particular, has exactly two leaves.

Now let us turn to line 18, and let us denote the root of $B(G)$ with r. Assume first that $x \neq r$. If $\deg(r) \geq 3$ then we can choose P_0 to be the $B_0 - x$ path. If $\deg(B_0) \leq 2$ and both sons y_1 and y_2 satisfy $\ell_{r,y_1} \geq 2$ and $\ell_{r,y_2} \geq 2$. Then we can route P_0 from x up to r, and then down in the direction of the (other) subtree having ≥ 2 leaves. This will eventually hit in a vertex of degree ≥ 3.

Otherwise we know that the root r does not lie on a path between two vertices of degree ≥ 3. Does x have an ancestor of degree ≥ 3? We can check this in constant time, comparing ℓ and ℓ_v^-. If $\ell - \ell_v^- \geq 2$, then $B(G)$ contains a vertex of degree ≥ 3 which lies on the path from x to the root r. Finally if $\ell - \ell_v^- = 1$, then $B(G)$ contains another vertex of degree ≥ 3 if and only if $S_{2+}^v \neq \emptyset$, which can again be decided in constant time.

The case $x = r$ is even easier. $B(G)$ has another vertex of degree ≥ 3 if and only if $S_{2+}^r \neq \emptyset$.

To put it all together: we can in constant time decide in which direction we should look for P_0. Clearly the amount of time needed to actually find and construct $P_0 \cup P_1 \cup P_2$ is proportional to the length of the resulting path.

Let G' be the graph $G + \mathrm{vx}(B_1)\mathrm{vx}(B_2)$. How does its block graph $B(G')$ depend on $B(G)$? We compute the new block B' on line 30, its vertex set is the union of sets of vertices of blocks which lie on P. If a c-vertex v lies on P and $\deg(v) \geq 3$, then v is also a c-vertex of $B(G')$ whose new degree has decreased by exactly 1. If $\deg(v) = 2$ then v is no longer a c-vertex of $B(G')$. Finally as P contains r, then the newly constructed block B' serves as the root of $B(G')$.

We need to refresh auxiliary data structures on line 31 only for vertices of $B(G')$ which are descendants of B^*. Now if $v \notin V(P)$ then its auxiliary data is left unchanged. If v is a c-vertex in $V(B(G')) \cap V(P)$, then the corresponding auxiliary data can be computed in constant time. The data for B' on the other hand takes time which is proportional to the length of P, which is up to a constant term proportional to the difference $|V(B(G)) - V(B(G'))|$. This implies that we can solve the BAP problem in linear time.

Theorem 3.5 *Let G be an undirected, and let $B(G)$ be its block tree. We can solve the* Biconnectivity Augmentation Problem *in $O(n+m)$ time.*
Even more, if $B(G)$ is precomputed and has n' vertices, then the algorithm BAP *computes an augmenting set of edges F in $O(n')$ time.*

Proof. As computing $B(G)$ takes $O(n+m)$ time, and since $n' = O(n)$, it is enough to show the latter statement.

Both FindPathMassive and FindPathBalanced take time proportional to the length P, their output. Also the routine AddEdge takes time, that is proportional to the length of the input path P. Then there exist positive constants a and b, so that the running time of FindPathBalanced, FindPathMassive, and AddEdge takes at most $a(|P|-1)+b$ time.

What is the cumulative running time of BAP on an input graph $G = G_0$? Let $G_0, G_1, G_2, \dots, G_k$ be the sequence of graphs obtained by inductively adding edges between b-vertices of paths $P_1, P_2, \dots, P_k$, resulting in a 2-connected graph G_k. The total running time is $O((|P_1|-1)+(|P_2|-1)+\cdots+(|P_k|-1)+k)$, which is since $k = O(n')$ equal to $O(n')$. $\square$

3.2 Structure of 2-connected graphs

Let us now turn our attention to 2-connected graphs. An *ear decomposition* of a graph G is a sequence of graphs $G_1, G_2, G_3, \dots, G_k$, so that

(ED1) G_1 is a cycle,

(ED2) $G_k = G$, and

(ED3) G_i is a graph obtained from G_{i-1} by attaching a path P_i between two vertices x and x' of G_{i-1}.

The added path P_i is also called an *ear*, and can also be a single edge.

Theorem 3.6 *Let G be an undirected graph. Then G admits an ear decomposition if and only if G is 2-connected.*

Proof. Assume that G admits an ear decomposition: 2-connectivity of G can be shown by induction. Let $G_1, G_2, G_3, \dots, G_{k-1}, G_k = G$ be the ear decomposition of G. Inductively we may assume that G_{k-1} is 2-connected, as its ear decomposition is shorter. If G is not 2-connected, then G contains a cutvertex v. As G_{k-1} is 2-connected $G_{k-1} - v$ is a connected graph. Hence also $G_{k-1} - v \cup P_k$. This implies that v is an internal vertex of P_k. This is also not possible, as $G - v$ can be obtained by attaching a pair of pendant paths to a connected graph G_k.

As G is 2-connected, the minimal degree of a vertex is at least 2, which implies that G contains a cycle subgraph G_1. Assume that we have already constructed the sequence $G_1, G_2, \dots, G_j$. Assume there exists a vertex $v \in V(G) \setminus V(G_j)$. As G is a 2-connected graph there exist $v - V(G_j)$ paths Q, Q' which share vertex v but are otherwise disjoint (and only meet $V(G_j)$ in their endvertices). Their union $Q \cup Q'$ can be used as the next ear P_{j+1}. Hence we may assume that G_j is a spanning subgraph of G. Now the missing edges from $E(G) \setminus E(G_j)$ serve at the final ears in the ear decomposition of G. $\square$

Let G be an undirected graph. An *st-labeling* (or also *st-ordering*) of G is a linear ordering of its vertices $v_1, v_2, \dots, v_n$, so that for every vertex v_i, $1 < i < n$, there exist indices $j < i$ and $k > i$, so that v_i is adjacent to both v_j and v_k.

We can picture an st-ordering by arranging vertices of G on a real line, so that every vertex, except the leftmost and the rightmost one, has a vertex both to the left and to the right of itself.

The origin of the name st-labeling comes from directed graphs. If we orient every edge of G from a vertex of the lower label towards the vertex of the higher label, we obtain a dag with a single source $s = v_1$ and a single sink $t = v_n$.

Let us first state a nice property of an st-labeling of a graph G.

Proposition 3.7 *Let $v_1, v_2, \ldots, v_n$ be an st-labeling of G. Then for every $i \in \{1, \ldots, n-1\}$ both induced subgraphs $G[v_1, \ldots, v_i]$ and $G[v_{i+1}, \ldots, v_n]$ are connected.*

Proof. Fix $i \in \{1, \ldots, n-1\}$ and let $j \leq i$. Inductively can show that v_1 and v_j lie in the same component: this is obviously true if $j = 1$, and is also true for bigger values of j, as v_j is adjacent to a vertex with a smaller index. The proof follows by symmetry. $\square$

Is there a relation between an st-labeling and an ear decomposition? Indeed, every ear decomposition can be transformed to an st-labeling, as we shall see in the proof of Theorem 3.8.

Does every graph admit an st-labeling? Clearly disconnected graphs do not admit st-labelings. Also not every connected graph admits an st labeling. Let G be a connected graph and assume that $B(G)$ contains three leaf blocks B_1, B_2, B_3. Now, being leaf blocks, B_i, $i = 1, 2, 3$, attaches to the rest of the graph through a cutvertex b_i. Assume that G admits an st-labeling $v_1, \ldots, v_n$. We may without loss of generality assume that neither v_1 nor v_n are vertices of B_1. Now let $I = \{i \mid v_i \in V(B_1)\}$ and let $i_{\min}$ and $i_{\max}$ be the smallest and largest indices in I, respectively. As b_1 can only be identical to one of $v_{i_{\min}}$ or $v_{i_{\max}}$, either $v_{i_{\min}}$ has no neighbor to its left or $v_{i_{\max}}$ has no neighbor to its right.

If G is a connected graph so that $B(G)$ contains exactly 2 leaf blocks, then G can be transformed into a 2-connected graph with an addition of a single edge. These graphs do admit an st-labeling.

Theorem 3.8 *Let G be a 2-connected graph and let $xy \in E(G)$. Then G admits an st-labeling so that $x = v_1$ and $y = v_n$.*

Proof. Let $G_1, G_2, \ldots, G_k = G$ be a fixed ear decomposition of G, so that $xy \in E(G_1)$. We shall inductively construct linear orders of vertices of graphs in the decomposition, and compute labels of vertices only with the final ordering L.

The $x - y$-Hamilton path represents the initial order of $V(G_1)$. Inductively assume that we have constructed an ordering of G_i. Let $x_i, y_i \in V(G_i)$ be the endvertices of the next ear P_{i+1}. We can insert the internal vertices of P_{i+1} between L so that every internal vertex of P_{i+1} lies in L between its neighbors in P_{i+1} (not necessarily continuously). $\square$

Let us finish with an algorithm for computing an st-ordering of a 2-connected graph using Algorithm 2.2 DFSLow. Let G be a 2-connected graph, and xy a fixed edge. In order to construct an st-labeling with x the initial vertex and y the terminal one let us first compute both the starting times and low points $\mathrm{start}(v)$ and $\mathrm{low}(v)$ for every vertex $v \in V(G)$, assuming the search starts at x moving to y next, i.e. $\mathrm{start}(x) = 1$ and $\mathrm{start}(y) = 2$. Let us denote the start time of the father of v by $\mathrm{pred}(v)$. As G is a 2 connected graph we have

(5) *for every vertex $v \neq x, y$ we have* $\mathrm{pred}(v) > \mathrm{low}(v)$,

as if $\mathrm{pred}(v) \leq \mathrm{low}(v)$ implies that $\mathrm{pred}(v)$ is a cutvertex in G since his son v has its lowpoint $\mathrm{low}(v)$ at least as large as $\mathrm{pred}(v)$. Let us consider vertices according to their start times. Choose $v \neq x, y$ and let us assume that the partial st-ordering L contains every vertex whose start time is strictly smaller than $\mathrm{start}(v)$. Then let us

extend L by putting (1) v between $\mathrm{pred}(v)$ and $\mathrm{low}(v)$ and also (2) next to $\mathrm{pred}(v)$. (3.2)

Let $v \neq s, t$. A *v-lowpath* is a $v - \mathrm{low}(v)$-path following tree edges with a possible final back edge. We claim that in the end (3.2) produces an st-labeling of the whole graph G. We have to show that every vertex $v \neq x, y$ has a neighbor both to its left and to its right, and we will do it by induction on the length of the v-lowpath P. If $|P| = 1$ then v is adjacent to both $\mathrm{pred}(v)$ and $\mathrm{low}(v)$, and thus has a left and a right neighbor in the final ordering. If $|P| > 2$ let v' be adjacent to v along P. This implies that v' is a son of v, $\mathrm{low}(v') = \mathrm{low}(v)$, and v'-lowpath P' is strictly shorter than P. As v' lies between $v = \mathrm{pred}(v')$ and $\mathrm{low}(v') = \mathrm{low}(v)$ in the final ordering, both v' and $\mathrm{low}(v)$ lie to the same side relative to v, and to the other side as $\mathrm{pred}(v)$. Hence also v has both a neighbor to the left and a neighbor to the right.

There is an algorithmic caveat. In order to be able to add elements immediately next to a fixed element, the data structure containing the temporary linear ordering should be a doubly linked list. But given two elements from a doubly linked list $\mathrm{low}(v)$ and $\mathrm{pred}(v)$, how can we quickly (in constant time) decide which lies to the left of the other?

There is a solution to the above problem by using signs of vertices: $\mathrm{sign}(v) = +$ (or $-$) indicates that a vertex u whose $\mathrm{low}(u) = v$ should be put in L before (or after)

v, respectively. A sign of a vertex may change in time, though.

```
    STLabel$(G, s, t)$
1   run DFSLow to compute start$(v)$, low$(v)$,
2     and pred$(v)$, so that start$(s) = 1$ and start$(t) = 2$;
3   set sign$(s) = -$ and $L = [s, t]$;
4   **for** $v \in V(G)$ *in the increasing* start(.) *order* **do**
5       **if** sign(low$(v)$) $= +$ **then**
6           insert $v$ immediately after pred$(v)$ in $L$;
7           set sign(pred$(v)$) $:= -$
8       **if** sign(low$(v)$) $= -$ **then**
9           insert $v$ just before pred$(v)$ in $L$;
10          set sign(pred$(v)$) $:= +$
11      **return** $L$
```

Algorithm 3.2: *st*-labeling algorithm

Theorem 3.9 *The algorithm* STLabel *correctly computes L and does so in time* $O(n + m)$.

Proof. Time complexity is easy, as DFSLow runs in $O(n + m)$ time, and the rest can be done in $O(n)$ time, provided we store L in a doubly linked list.

Let v be the current vertex on line 4 of STLabel. We claim that upon inserting v in L every proper ancestor u of v has a sign describing its relative positions to v: if u is before v in L then sign$(u) = -$, and u lies after v then sign$(u) = +$.

Let $v^* =$ pred(v). Upon inserting v in L we have only adjusted the sign of v^*. Hence for every ancestor u of v^* its sign sign(u) describes its relative position to v^* as well: sign$(u) = -$ if and only if u lies before v^* in L. Now low(v) is a proper ancestor of v^* and lies to the left of v^* if and only if sign(low(v)) $= -$. In this case line 8 puts v immediately to the left of v^* which in turn implies that (1) low(v) is to the left of v and (2) sign(v^*) indicates the relative position of v^* with respect to v, as claimed.

By symmetry the proof is finished. □

4 Planar graphs

4.1 Embedding as a drawing

Choose a graph G. We would like to draw G in the plane *without crossings*. But let us first tackle the simpler question on how to draw vertices and edges.

An *arc* in Σ is an embedding (injective continuous mapping) of an interval $[0, 1]$ into S, a *simple closed curve* is an embedding of S^1 in Σ. We shall implicitly assume that mappings are *nice* — they do not contain topological pathologies. We may for example assume that an embedding is piecewise linear (provided Σ is equipped with a linear-space-like structure) or differentiable (if Σ is smooth).

A *drawing* of G in Σ is a mapping

$$D : V(G) \cup E(G) \to \Sigma \tag{4.1}$$

satisfying the following conditions:

(D1) for different vertices $u, v \in V(G)$ we have $D(u) \neq D(v)$, the restriction of D to $V(G)$ is injective,

(D2) for every edge $e = uv \in E(G)$ its image $D(e)$ is an *arc* whose endpoints are $D(u)$ and $D(v)$, and

(D3) there are no unnecessary intersections — if u is not an endvertex of e then $D(u)$ and $D(v)$ are disjoint, and the only possible intersection of $D(e)$ and $D(f)$ is the image of their common endvertex.

We shall first focus on *planar drawings* where $\Sigma = \mathbb{R}^2$, and minutes later establish a connection with drawings in the sphere.

Graph G is *planar* if G admits a *planar drawing* D. A pair (G, D) is called a *plane graph*. Let us define *$D(G)$* as the union of images of vertices and edges of G and call $D(G)$ the *drawing of G*.

Given a fixed drawing, we shall informally identify G with its drawing and speak of the *plane graph G*.

A *face* of (G, D) is a connected component of $\Sigma \setminus D(G)$. Informally faces can be obtained by cutting the surface Σ with scissors running along drawings of edges. In a planar drawing, exactly one of the faces is unbounded, we shall call it the *infinite face*. A face can be described with a *facial walk* — a sequence of vertices that we encounter by travelling along its boundary.

Let S^2 denote the 2-sphere, set of points in 3-space at unit distance from the origin, and let N denote the *north pole* of S^2 (the point $(0, 0, 1)$). We know that the stereographic projection φ maps $S^2 \setminus N$ bijectively to $\mathbb{R}^2$ (the xy plane in 3-space). Let r be a one-way infinite ray originating in the north pole. If $v_s \in S^2 \setminus N$, $v_r \in \mathbb{R}^2$, and both lie in r, then $\varphi(v_s) = v_r$.

If (G, D) is a plane graph, then the mapping $\varphi^{-1} \circ D$ is a drawing of G in the sphere (which misses N). Similarly if D' is a drawing of G in the punctured sphere $S^2 \setminus N$, then $\varphi \circ D'$ is a planar drawing of G.

Now let f^∞ be the infinite face of (G, D). Now $\varphi^{-1} \circ D$ is a drawing of G in the sphere, and $\varphi^{-1}(f^\infty)$ is a face in the spherical drawing containing N. Let R be an arbitrary rotation of the sphere, and consider the mapping $\varphi \circ R \circ \varphi^{-1} \circ D$. We may slightly perturb R (if necessary at all) so that the north pole is not the image of a vertex or an edge. The set of facial walks has stayed the same, but the facial walk of f^∞ may now describe a bounded face.

The above argument implies that the infinite face of a planar drawing f^∞ is just an *inconvenience* (as every face can be made the infinite one if we rotate the spherical projection of the planar drawing). On the other it can be a *feature* of the drawing: if in some case we would like to pick a particular face from a planar drawing we can always name it *the infinite* face.

4.2 Combinatorial embeddings

Let G be a plane graph, and let us choose an orientation — say clockwise — in the plane. For every vertex v let π_v denote the cyclic ordering of neighbors of v in the clockwise (i.e. chosen orientation) ordering around v. The collection of all cyclic orderings $\{\pi_v \mid v \in V(G)\}$ is called *rotation system* of the drawing of G.

If $uv \in V(G)$, then $u' = \pi_v(u)$ is called the *successor of u around v* and symmetrically u is the *predecessor of u' around v*. The ordered triple $[u', v, u]$ is then also called an *angle*. These terms have the following geometric interpretation. Choose a point p on $D(vu)$ very close to $D(v)$. Rotating p (clockwise) around u will first intersect $D(G)$ in a point of $D(vu')$.

Let v_0v_1 be an edge of G. The sequence $v_0v_1v_2 \dots v_{d-1}$ is a *facial walk* if

(F1) for every $i = 1 \in \{0, \dots, d-1\}$ the triple $[v_i, v_{i+1}, v_{i+2}]$ is an angle (indices are taken $\mod d$) and

(F2) no angle is repeated, for $i \neq j$ we have $[v_i, v_{i+1}, v_{i+2}] \neq [v_j, v_{j+1}, v_{j+2}]$.

Condition (F1) implies that we follow a face along consecutive angles, and (F2) prevents of going around the face twice.

Let us look at the the cube graph Q_3, with vertices $V(Q_3) = \{1, \dots, 8\}$. The edge set will implicitly defined by the rotation system

$$\begin{aligned}
\pi_{v_1} &= (v_2, v_3, v_5)\\
\pi_{v_2} &= (v_1, v_7, v_4)\\
\pi_{v_3} &= (v_1, v_4, v_5)\\
\pi_{v_4} &= (v_2, v_6, v_3)\\
\pi_{v_5} &= (v_3, v_6, v_8)\\
\pi_{v_6} &= (v_4, v_7, v_5)\\
\pi_{v_7} &= (v_2, v_8, v_6)\\
\pi_{v_8} &= (v_1, v_5, v_7)
\end{aligned}$$

In order to compute the facial walks (which in turn determine faces) let us follow angles starting from every possible direction of an edge. Choose, for example v_1v_2. Now v_4 is the predecessor of v_1 around v_2, as $\pi_{v_2}(v_4) = v_1$, hence $[v_1, v_2, v_4]$ is an angle. The following three angles are $[v_2, v_4, v_3]$, $[v_4, v_3, v_1]$, and $[v_3, v_1, v_2]$, and the next angle $[v_1, v_2, v_4]$ is the repetition of the first one. Hence we obtain a facial walk $v_1v_2v_3v_4$.

The remaining facial walks are $v_2v_1v_8v_7$, $v_4v_2v_7v_6$, $v_4v_6v_5v_3$, $v_1v_3v_5v_8$, and $v_5v_6v_7v_8$. Observe that every edge appears exactly twice in the facial walks, once in each direction.

Similarly, given a collection of facial walks (a collection of walks in which every edge exactly twice, once in each of the directions) determines the rotation system: on one hand we can determine the lists of neighbors of each vertex v, on the other hand we can for each neighbor u of v determine its successor in the rotation system — we only need to find the vertex u' which immediately precedes v, u in the collection of facial walks.

It seems that the original choice of the orientation (clockwise or counter-clockwise) produces a different rotation system, and consequently a different set of faces. We shall however declare these choices being equivalent: by reversing all cyclic orderings in a rotation system has a consequence that the constructed facial walks all run in different directions.

This motivates us to define the *combinatorial embedding* of G as every structure that uniquely determines both the rotation system and the set of facial walks in a drawing of G (up to a possible reversal).

Now a combinatorial embedding may not produce a planar drawing. Fortunately, there is only one criterion to meet. Provided that the combinatorial embedding determines the correct number of faces (see Theorem 4.9) then it also determines a drawing in the plane.

Let us finish this subsection with a definition. Let f be a face of a drawing of a planar graph G. The *length of f*, $\deg(f)$, is defined as the length of its facial walk.

4.3 Results from topology

Let us state a couple of mathematical (indeed topological) theorems. These results are surprising in the following way — they seem to be obviously true. Yet they require deep (i.e. this is not freshman-year-mathematics) and technical proofs which are way above our reach.

Theorem 4.1 (Jordan Curve Theorem) *Let C be a simple closed curve in $\mathbb{R}^2$. Then $\mathbb{R}^2 \setminus C$ has exactly two connected components: the interior of C is the bounded component and the exterior of C is unbounded. C is the boundary of both the interior and exterior components.*

Theorem 4.2 (Three arcs theorem) *Let P_1, P_2, P_3 be three arcs with common endpoints x and y, and otherwise disjoint in $\mathbb{R}^2$. Let z_1 and z_2 be interior points of P_1 and P_2, respectively. Let P be an arbitrary $z_1 - z_2$-arc, which lies in the same connected component of $\mathbb{R}^2 \setminus (P_1 \cup P_2)$ as P_3. Then $P \cap P_3 \neq \emptyset$.*

4.4 Connectivity and planar graphs

Let G be a disconnected graph. Then G is planar if and only if every component of G is planar, as disjoint components can be drawn in the plane independently, far apart. Hence we shall in the sequel limit ourselves to connected graphs.

Let G be a graph which is not 2-connected, and contains at least 3 vertices. Then G contains at least one cutvertex, and possibly also a cutedge. These play a particular role in case G is planar.

Proposition 4.3 *Let G be a plane graph.*
A vertex v is a cutvertex in G if and only if there exists a face f so that v appears at least twice on its facial walk.
An edge e is a cutedge in G if and only if there exists a face f' so that e is traversed exactly twice on a facial walk of f'. In this case each direction of e is traversed exactly once.

Proof. Assume first that v is a cutvertex and let G_1 be a component of G. Let us color all edges of $G_1 + v$ in red, and the rest of edges green. Now the local rotation π_v determines a mixed angle (v_0, v, v_1), an angle in which the edge v_0v is green and vv_1 is a red edge. Let f be the face containing the angle (v_0, v, v_1) and observe the colors of edges along its facial walk. They must change at least twice, and this can only happen in v, as v is the only vertex incident with edges of both colors.

Assume that the facial walk of f enters v twice. Let (v_0, v, v_1) and (u_0, v, u_1) be the corresponding angles. We can construct a simple closed curve C entering v between vv_0 and vv_1 and leaving between vu_0 and vu_1, which is otherwise disjoint with G. As no $v_0 - v_1$-path can pass through $C \setminus \{v\}$, vertices v_0 and v_1 lie in different components of $G - v$.

The edge-part of the proof can be done analogously. □

As 2-connected planar graphs do not contain cutvertices, we have the following easy corollary.

Corollary 4.4 *Let G be a 2-connected planar graph, and let D be an arbitrary drawing of G. Then every facial walk of $(D(G)$ is a cycle.*

Corollary 4.4 has itself a rather nice corollary (see, again). Facial cycles of $D(G)$ can be interpreted as subgraphs of G. On the other hand it is not true that the graph G itself determines the set of facial cycles. Even a 2-connected graph can have several drawings in the plane, which in turn determine different collections of facial cycles in each of the embeddings.

Consider the following scenario. Let G be a plane graph, and let x, y be a 2-separator in G. This implies that we can find a simple closed curve C, passing through both x and y and otherwise disjoint from G, so that both the interior and exterior of C contain vertices of G, respectively. We can now reflect the interior of C, keeping x and y fixed, in order to obtain a different embedding of G. This procedure is called a *Whitney flip*.

Theorem 4.5 (Whitney) *Let D_1 and D_2 be combinatorial embeddings of a 2-connected planar graph G. Then D_2 can be obtained from D_1 by a series of Whitney flips.*

Now a 3-connected graph contains no 2-separators, so no Whitney flips are possible in an embedding of a 3-connected planar graph. This implies the following corollary of Theorem 4.5.

Theorem 4.6 *Let G be a 3-connected planar graph. Then G has a unique combinatorial embedding in the plane.*

Now Theorem 4.6 implies that in a 3-connected planar graph the collection of facial walks (these are indeed cycles by Corollary 4.4) is uniquely determined. It is not surprising that these can be described in a purely combinatorial way.

Theorem 4.7 *Let G be a 3-connected planar graph. Then the collection of faces of G is exactly the set of nonseparating induced cycles of G.*

Proof. Assume G is drawn in the plane, and let C be a face of G (we shall also assume that C is not drawn as the infinite face so that its interior is empty). Let u, v be nonconsecutive vertices along C which are adjacent. Choose an $u - v$-arc in the interior of C. Together with the edge uv it forms a simple closed curve which separates vertices of the two segments of $C - v - u$. This is not possible as G is a 3-connected graph. Therefore C is induced.

Now let $x, y \notin V(C)$. As G is 3-connected there exist three internally disjoint $x - y$-paths P_1, P_2, and P_3. Now C lies in one of the regions bounded by two among these paths not containing the third, without loss of generality $P_1 \cup P_2$. Hence C and P_3 have no vertices in common and C does not separate x form y. As x, y were arbitrary this implies that C is not separating.

For the other direction observe that a nonseparating induced cycle C cannot contain vertices of G both in its interior and its exterior. Hence we may without loss of generality assume that its interior contains no vertices of G. As C is an induced cycle its interior cannot contain edges of G either. Hence C is a face. □

A *triangulation* is a plane graph G in which every face is of length 3 (recall that G can have no parallel edges).

Proposition 4.8 *Let T be a triangulation. Then T is 3-connected. Moreover if S is a minimal separator in T, then S induces a cycle.*

Proof. Let S be a minimal separator in T, and let T_1 be a component of $T - S$. Let us replicate the trick by coloring edges. Let us color edges in T_1 as well as edges between S and T_1 red, edges between vertices of S black, and remaining edges green. If $|S| \leq 2$ then there exists a face f with an angle (u, s, x) where us is a green edge and sx is a red edge. As x and u are not adjacent, $|f| \geq 4$, which is a contradiction.

Let S be a minimal separator in T. Now Proposition 4.15 implies that $T - S$ contains exactly two components T_1 and T_2. Let us color edges of T in three colors red, green, and black as above, and let us again use the fact that no angle contains a red and a green edge. Hence each vertex $s \in S$ is incident with at least two black edges. This implies that $G[S]$ contains a cycle. Let C_1 be the shortest cycle of $G[S]$ so that both its interior and exterior contains a vertex of G. If $V(C_1) \neq S$, then we have a contradiction with minimality of S. Let $ss' \in E(G)$ be an edge between two nonconsecutive vertices of C_1. Now $C_1 + ss'$ divides $\mathbb{R}^2$ into three regions so that (i) at least two contain vertices of G (as C_1 is a separating cycle) and at (ii) least two are empty as the both short cycles of $C_1 + ss'$ are nonseparating, which is a contradiction. □

4.5 Euler formula

Let G be a plane graph. Let us with $F(G)$ denote the set of faces of G.

Theorem 4.9 (Euler formula) *Let G be a connected plane graph. Then*

$$|V(G)| - |E(G)| + |F(G)| = 2$$

Proof. Choose a spanning tree T of G. Now $V(T) = V(G)$, $E(T) = V(T) - 1 = V(G) - 1$ and $F(T) = 1$, hence

$$|V(T)| - |E(T)| + |F(T)| = |V(G)| - (|V(G)| - 1) + 1 = 2, \tag{4.2}$$

in other words, Euler formula holds for the spanning tree T. Now adding an edge of $E(G) \setminus E(T)$ has the following effect: number of vertices does obviously not change, and the number of edges increases by one. An addition of a new edge (it is not a cutedge as together with T it forms a cycle) divides an existing face into a pair of new faces, and thus increases the number of faces by one as well. Hence Euler formula holds by induction on the number of edges in $E(G) \setminus E(T)$ for the original graph G as well. □

Now Euler formula effectively bounds the number of edges in a planar graph:

Proposition 4.10 *Let G be a (planar) plane graph with at least 3 vertices. Then*

1. $|E(G)| \leq 3|V(G)| - 6$, *and*
2. $|E(G)| = 3|V(G)| - 6$ *if and only if G is a triangulation.*

Proof. Assume G is drawn in the plane, and let us sum the lengths of its faces.

$$2|E(G) = \sum_{f \in F(G)} \deg(f) \geq 3|F(G)|$$

The first equality follows as every edge appears exactly twice along facial walks, and the second inequality as the length of each face is at least 3. Note that equality arises if and only if G is a triangulation, both in the above and consequently below computation.

$$3|V(G)| - 6 = 3|E(G)| - 3|F(G)| \geq 3|E(G)| - 2|E(G)| = |E(G)|$$

□

Let us continue with a technical lemma.

Lemma 4.11 *Let G be a plane graph. Then G is bipartite if and only if all facial walks are of even length.*

Proof. Clearly no closed walk in a bipartite graph can be of odd length. For the converse, let us choose a cycle C in G. Now $|C|$ has the same parity as the sum of parities of lengths of faces, which are contained in the interior of C. This follows easily by induction on the number of faces in the interior of C. □

Proposition 4.12 *Let G be a bipartite (planar) plane graph with at least 3 vertices. Then*

1. $|E(G)| \leq 2|V(G)| - 4$, *and*
2. $|E(G)| = 2|V(G)| - 4$ *if and only if G is a* quadrangulation.

A *quadrangulation* is a plane graph whose faces are all of length 4. The proof of Proposition 4.12 can be settled in the very same manner as the proof of Proposition 4.10. The only difference is that every face in a bipartite plane graph has to be longer than 4.

We have nevertheless devised the necessary tools to find first examples of nonplanar graphs.

Proposition 4.13 *K_5 and $K_{3,3}$ are not planar graphs.*

Proof. K_5 has 5 vertices and 10 edges, and $K_{3,3}$ is a bipartite graph with 6 vertices and 9 edges. By Proposition 4.10 an Proposition 4.12 their numbers of edges are too big for them to be planar. □

Let G be a planar graph. Clearly the following three operations preserve planarity: deleting vertices, deleting edges, and subdividing edges (replacing a single edge with a longer path). All these operations can namely be carried out not only on the graph G itself, but also on its drawings.

Hence the class of all planar graphs is closed under taking subgraphs and also under taking subdivisions.

There is another operation which preserves planarity, the *edge contraction*. Let $e = uv$ be an edge in G. Let $N_{u,v}$ be the set of vertices of $G - u - v$ which are adjacent to at least one of u, v. The graph G/e is obtained form $G - u - w$ by adding a new vertex x_{uv} and making it adjacent to every vertex of $N_{u,v}$. We say that G/e is obtained from G *by contracting e*.

If G is a plane graph, then the drawing of G/e can be obtained by drawing x_{uv} on a midpoint of the edge uv. The $x_{uv} - N_{u,v}$ edges can be drawn simultaneously without crossings in the vicinity of uv and edges incident with either u or v.

We call a graph H a *minor* of G, if H can be obtained from G by a series of vertex deletions, edge deletions, and edge contractions.

By above description the class of planar graphs is also *minor closed*.

It is therefore not surprising that we can characterize planar graphs both in terms of forbidden subdivisions and also forbidden minors. What is surprising is the fact that the sets of obstructions is the same in both cases, and that K_5 and $K_{3,3}$ are indeed the smallest nonplanar graphs.

Theorem 4.14 (Kuratowski, Wagner) *G is planar if and only if*

1. *G does not contain a subdivision of K_5 or $K_{3,3}$ (or equivalently)*
2. *G does not contain a K_5 or $K_{3,3}$ as a minor.*

We shall skip the proof.

Finally let us state and prove a proposition on the number of components relative to a small separator.

Proposition 4.15 *Let G be a 3-connected planar graph, and let S be a minimal separator. Then $G - S$ contains exactly two connected components.*

Proof. Assume that $G - S$ contains three connected components G_1, G_2, and G_3. Let us contract G_1, G_2, and G_3 to single vertices g_1, g_2, g_3, respectively (we can do that inductively, as contracting a single edge reduces the vertex-count by one). As S is a minimal separator, every vertex $s_i \in S$ has a neighbor in each of the components of $G - S$. Hence every vertex of S is adjacent to g_1, g_2, and g_3 in the contracted graph. As $|S| \geq 3$ we obtain a $K_{3,3}$-minor in G, which is absurd. $\square$

4.6 Dual graphs

Let G be a (connected) plane graph. We can construct the *dual graph* G^* in the following way:

1. for every face $f \in F(G)$ put a vertex $f^* \in V(G^*)$ in the interior of f, and
2. for every edge $e \in E(G)$ so that faces f_1 and f_2 contain e in their respective boundaries, the vertices f_1^* and f_2^* are adjacent via an edge e^* in G^*.

The edge e^* is also called the dual edge of e, which also means that there exists a bijective correspondence between edges of G and those of G^*.

Dual graph G^* appears to be a purely geometric concept. Yet it turns out to be of a different, combinatorial intact, flavor. Minimal edge-cuts in G correspond to cycles of G^* and vice-versa. Cycles in G correspond to minimal edge cuts of G^*. This is due to Jordan Curve Theorem 4.1. A cycle C in G is a simple closed curve in the plane (sphere) and separates $\mathbb{R}^2$ into exactly two connected components, the interior and exterior of C. Let V_1^* be the set of vertices of G^* (i.e. faces of G) which lie in the interior of C, and V_2^* be the set of vertices of G^* which lie in exterior of C. Now every $V_1^* - V_2^*$-path in G^* crosses C, and hence uses an edge e^* which is a dual edge of some edge $e \in E(C)$. Hence the collection of *dual edges* of $E(C)$ is a cut in G^*. It is indeed a minimal cut, as the end vertices of each edge $e^* \in E^*(C)$ lie in different components of $\mathbb{R}^2 \setminus C$.

It is easy to argue that $G^{**} = G$, if G is a connected plane graph.

How are deletion and contraction of an edge in G and its dual G^* related? The deletion of an edge e in $E(G)$ merge the faces f_1, f_2 on either side of G into a single face $f_{1,2}$. The same effect can be produced in G^* by *contracting* its dual edge $e^* = f_1^* f_2^*$. And conversely, contracting an edge in G has the same effect as deleting its dual edge e^* in G^*.

Hence contraction and deletion of edges form a pair of dual operations. The execution of one operation in one of the graphs has exactly the same effect (as effect on the pair of graphs, the original and its dual) as executing the other operation in the dual graph.

5 Discharging technique

5.1 Consequences of Euler formula

Let G be a connected plane graph. Euler formula (Theorem 4.9) implies several structural results for planar graphs.

Proposition 5.1 *Let G be a plane graph.*

- *G contains a vertex of degree ≤ 5,*
- *if $\delta(G) = 5$ then G contains at lest 12 vertices of degree 5.*

Proof. If $\delta(G) \geq 6$, then summing vertex degrees implies that $2|E(G)| = \sum_{v\in V(G)} \deg(v) \geq 6|V(G)|$ which in turn contradicts the fact that $|E(G) \leq 3|V(G)| - 6$, see Proposition 4.10.

Further we may assume that G is a maximal planar graph (i.e. a trinagulation) as addition of edges cannot increase the number of vertices of small degree, and also cannot decrease the minimum degree. Let n_5 denote the number of vertices of degree 5 in G, and let n_{6+} denote the number of vertices of larger degree in G.

Then $2|E(G)| = \sum_{v\in V(G)} \deg(v) \geq 5n_5 + 6n_{6+}$. By Proposition 4.10 the expression $2|E(G)| \leq 6|V(G)| - 12 = 6n_5 + 6n_{6+} - 12$. This implies that $n_5 \geq 12$. □

A graph G is called *k-degenerate* if we can obtain an empty graph by repetitively deleting vertices of degree $\leq k$ starting from G. Equivalently stated, every subgraph of G has a vertex of degree $\leq k$. An easy consequence of Proposition 5.1 is that planar graphs are 5-degenerate. Starting with a planar graph, deleting vertices of degree ≤ 5 will ultimately lead to the empty graph.

Proposition 5.2 *Let G be a planar graph with n vertices. Then*

$$\sum_{e=uv\in E(G)} \min\{\deg(u), \deg(v)\} \leq 10|E(G)| = O(n)$$

Proof. Let us number vertices $v_1, v_2, v_3, \ldots, v_n$, so that v_i has smallest degree (which is ≤ 5) in $G[v_i, \ldots, v_n]$. Now

$$\begin{aligned}
&\sum_{e=v_kv_\ell\in E(G)} \min\{\deg(v_k), \deg(v_\ell)\} \\
\leq &\sum_{e=v_kv_\ell\in E(G)} \deg(v_{\min\{k,\ell\}}) \\
= &\sum_{i=1}^{n} \deg(v_i) \cdot |\{j \mid j > i \text{ and } v_iv_j \in E(G)\}| \\
\leq &\sum_{i=1}^{n} 5\deg(v_i) = 10|E(G)| = O(n),
\end{aligned}$$

as the equality between second and third lines follows by counting the number of edges, so that v_i is their endvertex with the smaller index. □

5.2 Triangulating a plane graph

One of the basic problems is turning a planar/plane graph G into a triangulation by adding edges. Let G be a planar/plane graph with $n \geq 3$ vertices and m edges. If $m < 3n - 6$ then at least one of the faces f in an arbitrary drawing of G (in case G is not actually a plane graph, but merely a planar one) has length ≥ 4. Pick a drawing of G and let f be a long face of $\deg(f) = k \geq 4$. Let $v_1 v_2 v_3 \dots v_k$ be its facial walk. By adding an edge $v_1 v_3$ in the interior of face f we have increased the total number of edges by one, which is a single step towards making all faces of length 3. But there is a catch, the edge $v_1 v_3$ might have already been present in the original graph, lying in the exterior of face f. As we do not want parallel edges we have to be a bit more careful.

```
TRIANGULATE($G$)
1  while $G$ contains a face $f$ of length $\deg(f) = k \geq 4$ do
2      let $v_0$ be a vertex of smallest degree along $f$ ;
3      if $v_0$ has a nonconsecutive neighbor $v_i$ along $f$ then
4          foreach $j \in \{i+1, i+2, \dots, k-1\}$ do
5              add edge $v_1 v_j$
6          foreach $j \in \{2, 3, \dots, i-1\}$ do
7              add edge $v_{i+1} v_j$
8      else
9          foreach $j \in \{2, 3, \dots, k-2\}$ do
10             add edge $v_0 v_j$
```

Algorithm 5.1: TRIANGULATE(G)

Proposition 5.3 *Let G be a 2-connected plane graph. Then Algorithm 5.1 correctly computes a plane triangulation T_G so that G is a spanning subgraph of T_G.*

Proof. It is easy to see that in either of if-cases on line 3 the algorithm triangulates the long face f, and the outer while loop takes care that every long face is triangulated upon finishing.

We only need to argue that no parallel edges are inserted in the run. This is clear if v_0 is not adjacent to another vertex of f, apart from the consecutive neighbors along f. On the other hand if $v_0 v_i$ is an edge and $2 \leq i \leq k-2$ then by Theorem 4.2 no pair of vertices x, y, which lie on different segments of $f - v_0 - v_i$ can be adjacent. Hence the inserted edges on lines 5 and 7 do not form parallel pairs with existing edges. □

Can we triangulate a plane graph G which is not 2-connected in the first place? There are some additional technical difficulties as facial walks in a plane graph which is not 2-connected can have repeated vertices. Rather than specifying a more general version of Algorithm 5.1 which would work on a larger class of input graphs let us describe a tool which works in a plane graph G having cutvertices.

Let G be a plane graph and v its cutvertex. Let π_v be the local rotation of neighbors around v in G. Assume that v_1 is a predecessor of v_2 around v, $\pi_v(v_1) = v_2$. Then adding the edge v_1v_2 to G, adjusting π_{v_1} and π_{v_2}, so that v is the predecessor of v_1 around v_2 and that v_2 is the predecessor of v around v_1 produces a plane graph G' with one block less in its block-tree. By applying the analogous operation on all pairs of predecessor-successor from different blocks, and iteratively over all cutvertices of G we obtain a plane 2-connected supergraph G' of G, so that $V(G') = V(G)$.

What we shall care next is the time complexity of Algorithm 5.1. Observe that for a connected planar graph we have $|V(G)| - 1 \leq |E(G)| \leq 3|V(G)| - 6$. This implies that the number of vertices and the number of edges are of the same order of magnitude and hence $O(n) = O(m)$ if we use n and m for the numbers of vertices and edges, respectively.

Theorem 5.4 *Let G be a 2-connected plane graph. Then Algorithm 5.1 runs in linear time.*

Proof. Let $f^1, f^2, \ldots, f^k$ be the list of long faces in $G = G_0$. Let G_i denote the graph obtained after triangulating the faces $f^1, \ldots, f^i$ by Algorithm 5.1.

In order to find the vertex v_0 of smallest degree along f^ℓ we have to traverse all vertices of f^ℓ which takes $O(|f^\ell|)$ time. In this process we also flag every vertex of f^ℓ, so that we can check for possible neighbors of v_0 along f^ℓ in $O(\deg_{G_{\ell-1}}(v_0))$ time. Adding the edges in order to triangulate f^ℓ takes time $O(|f^\ell|)$ and so does removing the flags of vertices along f^ℓ at the end.

Hence the total time complexity is equal to

$$O\left(\sum_{i=1}^{k} \deg(f^i)\right) + O\left(\sum_{i=1}^{k} \min_{v \in f^i}\{\deg_{G_{i-1}}(v)\}\right). \tag{5.1}$$

Let us estimate the terms separately. On one hand

$$\sum_{i=1}^{k} \deg(f^i) \leq \sum_{f \in F(G)} \deg(f) = 2|E(G)|.$$

The other sum proves to be more troublesome:

$$\begin{aligned}
&\sum_{i=1}^{k} \min_{v\in f^i}\{\deg_{G_{i-1}}(v)\} \\
\leq &\sum_{i=1}^{k} \min_{v\in f^i}\{\deg_{G_k}(v)\} \\
\leq &\sum_{i=1}^{k} \frac{1}{4} \sum_{e=uv\in E(f^i)} \min\{\deg_{G_k}(u), \deg_{G_k}(v)\} \\
\leq &\sum_{e=uv\in E(G_k)} \min\{\deg_{G_k}(u), \deg_{G_k}(v)\} = O(|V(G_k)|)
\end{aligned}$$

The first inequality holds as degrees of vertices grow as we triangulate faces, the second as each face f^i is of length ≥ 4 (note that f^i is a cycle in G_k). The last inequality follows as every edge $e \in E(G_k)$ belongs to at most two long faces of G, finally we apply Proposition 5.2.

Hence (5.1) is the sum of two terms of order $O(n)$ and the proof is complete.
$\square$

5.3 Discharging method

Our first results in this writeup have established the fact that every planar graph contains a vertex of bounded degree (at most 5 was the best possible bound). Does every planar graph contain an edge so that the sum of degrees of its endvertices is bounded as well? This turns out to be false, as every edge in $K_{2,n}$ (which is a planar graph) is adjacent with a vertex of large degree. However, if one limits to planar graph without vertices of degree ≤ 2, then we have an appropriate result.

Theorem 5.5 (Kotzig) *Let G be a planar graph with $\delta(G) \geq 3$. Then G contains an edge $e = uv$ so that* $\deg(u) + \deg(v) \leq 13$. *Moreover if both* $\deg(u) \geq 4$ *and* $\deg(v) \geq 4$, *then also* $\deg(u) + \deg(v) \leq 11$.

Proof. Let $e = uv \in E(G)$ and let us assume that $\deg(u) \leq \deg(v)$. Let us call $e = uv$ *light* if $\deg(u) = 3$ and $\deg(v) \leq 10$ or $\deg(u) \geq 4$ and $\deg(v) \leq 11 - \deg(u)$.

Assume that the theorem is false, and let G be a *minimal counterexample*: G is a planar graph without light edges, that has the smallest possible number of vertices and biggest possible number of edges. Let us draw G in the plane.

We claim that every face of G is a triangle. Otherwise G has a face f of length ≥ 4. Now f contains two nonconsecutive vertices u, v of degree ≥ 6 (otherwise two consecutive vertices have degree ≤ 5 which yields a light edge). Now $G + uv$ is a plane graph without light edges as the newly added edge is not light, and contradicts the minimality of G. (Note that the added edge might produce a graph with parallel edges, but $G+uv$ still has a drawing without faces of length 2, which in turn implies $|E(G+uv)| \leq 3|V(G+uv)| - 6$.)

Now let us apply the *discharging argument*. We shall assign *initial charge* to vertices and faces of G. This charge will be redistributed in the graph, but the total charge shall remain the same. In the end we shall obtain a contradiction: the structure of the graph G (in our case the lack of light edges) shall contradict the final distribution of charge.

Let $v \in V(G)$. Its initial charge $c_0(v) = \deg(v) - 6$. If f is a face of G, then let $c_0(f) = 2\deg(f) - 6 = 2 \cdot 3 - 6 = 0$. Now the total initial charge equals

$$\begin{aligned} &\sum_{v \in V(G)} c_0(v) + \sum_{f \in F(G)} c_0(f) \qquad (5.2)\\ &= \sum_{v \in V(G)} (\deg(v) - 6) + \sum_{f \in F(G)} (2\deg(f) - 6)\\ &= \sum_{v \in V(G)} \deg(v) - 6|V(G)| + 2\sum_{f \in F(G)} \deg(f) - 6|F(G)|\\ &= 2|E(G)| - 6|V(G)| + 4|E(G)| - 6|F(G)|\\ &= -6(|V(G)| - |E(G)| + |F(G)|) = -12. \end{aligned}$$

Let us call u a vertex of *small* degree if $\deg(u) \leq 5$, and let us call vertices of degree ≥ 7 vertices of *large* degree. We shall redistribute charge in the discharging process according to the following rule.

Rule: Let $uv \in E(G)$ and let u be a vertex of small degree. Then v sends v charge $\frac{6-\deg(u)}{\deg(u)}$.

After applying the Rule let $c_1(V)$ denote the final charge of a vertex. If f is a face then $c_1(f) = c_0(f)$ as the rule only transfer charges between vertices.

Observe first, as no edge is light, that a vertex of small degree u can only be adjacent to vertices of large degree. Hence u receives $\frac{6-\deg(u)}{\deg(u)}$ from each of its neighbors, and

$$c_1(u) = c_0(u) + \deg(u) \cdot \frac{6 - \deg(u)}{\deg(u)} = (\deg(u) - 6) + (6 - \deg(u)) = 0.$$

If $\deg(v) = 6$, then $c_1(v) = c_0(v) = 0$, as vertices of degree 6 do not play a role in discharging.

Now let w be a vertex of large degree k. As every face of G is a triangle, no two consecutive (around w) neighbors of w can be of small degree. Hence w can only send charge to at most $\lfloor \frac{k}{2} \rfloor$ neighbors.

If $\deg(w) = 7$, then every neighbor of w has degree ≥ 5. Therefore $c_1(w) \geq c_0(w) - \lfloor \frac{7}{2} \rfloor \frac{1}{5} = 1 - \frac{3}{5} \geq 0$.

If $\deg(w) = k \in \{8, 9, 10\}$ then every neighbor of w has degree ≥ 4. This implies that $c_1(w) \geq c_0(w) - \lfloor \frac{k}{2} \rfloor \frac{1}{2} = (k - 6) - \lfloor \frac{k}{2} \rfloor \frac{1}{2} \geq 0$.

Finally if $\deg(w) = k \geq 11$ then $c_1(w) \geq c_0(w) - \lfloor \frac{k}{2} \rfloor = (k-6) - \lfloor \frac{k}{2} \rfloor \geq 0$

Now the total final charge is on one hand equal to -12 and is nonnegative in the other. This contradicts the fact that G has no light edges. □

5.4 Colorings of planar graphs

A *(vertex)-coloring* of G is a mapping

$$c : V(G) \to \mathbb{N},$$

so that for every pair of adjacent vertices u, v we have $c(u) \neq c(v)$. A coloring should necessarily color neighboring vertices with different colors. A *k-coloring* is a coloring into $\{1, 2, \dots, k\}$, and the smallest k for which there exists a k-coloring of G is called the *chromatic number* of G, $\chi(G)$.

The notorious *Four color theorem* states that we can color vertices of an arbitrary planar graph with no more than 4 colors. Let us state it without proof.

Theorem 5.6 (Four color theorem) *Every planar graph G satisfies $\chi(G) \leq 4$.*

Four color theorem is the optimal result, K_4 is a planar graph and $\chi(K_4) = 4$. It is however possible to give short proofs of weaker results.

It is easy to see that $\chi(G) \leq 6$ for every planar graph G. Namely, every planar graph is 5-degenerate. This implies that we can construct an ordering $v_1, v_2, \dots, v_n$ of vertices of G so that v_i has degree ≤ 5 in $G[v_i, v_{i+1}, \dots, v_n]$, see Proposition 5.2. By refreshing the set of vertices of degree ≤ 5 in the remaining graph we can construct the sequence in linear time. Finally we color vertices of G greedily in the reverse ordering $v_n, v_{n-1}, \dots, v_1$, by coloring v_i by the smallest color from $\{1, 2, 3, 4, 5, 6\}$ which is not already used on colored neighbors of v_i.

Constructing a 5-coloring of a planar graph is a bit more technical. At least three different approaches exist. Historically, using *Kempe chain* argument predates both coloring by contraction and a list coloring argument of Thomassen 6.6.

Let us briefly explain the former two approaches, and let us leave Thomassen's list coloring idea for the next section, where it will be studied in greater detail.

Both Kempe-chain and contraction argument choose a vertex of smallest degree $v \in G$, and first recursively construct a coloring of $G - v$. If $\deg(V) \leq 4$, then the 5-coloring c' of $G - v$ can be extended to a 5-coloring of v. Hence we may assume that $\deg(v) = 5$, and that the neighbors of v in G are v_1, v_2, v_3, v_4, v_5, with indices matching the local rotation around v.

Now if the coloring c' does not use five different colors for $v_1, \dots, v_5$, again c' can be extended to v. Hence we may (without loss of generality) assume that $c'(v_i) = i$. Observe the component $C_{1,3}$ of G induced by vertices of colors 1 and 3 which contains v_1. We can recolor vertices of $C_{1,3}$ by changing colors 1 and 3 within $C_{1,3}$, this procedure is called a *Kempe change*. If we are lucky and $v_3 \notin C_{1,3}$ then the change results in exactly four colors $2, 3, 4, 5$ being used on neighbors of v. Otherwise there is a $v_1 - v_3$ path consisting entirely of vertices colored with colors 1 and 3 in $G - v$, this is a *Kempe chain*. Now a Kempe-chain between v_1 and v_3 implies that vertices v_2 and v_4 do not lie on a path consisting of vertices of colors 2 and 4 only. Hence we can recolor the component $C_{2,4}$ containing vertex v_2, and again construct a 5-coloring of $G - v$ using 4 colors on neighbors of v.

The contraction argument on the other hand preemptively constructs a 5-coloring c' of $G - v$ which only uses ≤ 4 colors on $v_1, \ldots, v_5$. Observe that it is not possible that both $v_1v_3 \in E(G)$ and $v_2v_4 \in E(G)$. We can assume that $v_1v_3 \notin E(G)$. Now $G' = G - v + v_1v_3/v_1v_3$ is a planar graph in which vertices v_1 and v_3 are identified. A 5-coloring c' of G' can be identified with a 5-coloring of $G - v$ in which vertices v_1 and v_3 are assigned the same color. Again this implies that $v_1, \ldots, v_5$ use at most 4 different colors, so c' can be extended to a 5-coloring of G.

However simple the above arguments are, they cannot be turned into a linear-time algorithm. The Kempe chain approach needs a possible recoloring of sizable portions of G, which can only be proven to take quadratic time altogether. The contraction argument can be turned into a linear-time algorithm by increasing the number of different contraction types — preformed not only at vertices of degree ≤ 5, but also at vertices of degree 6.

If we restrict ourselves to smaller classes of planar graphs, we can prove better bounds. On one hand chromatic number drops by one if we forbid triangles.

Theorem 5.7 (Grötzsch) *Let G be a planar graph without triangles. Then $\chi(G) \leq 3$.*

On the other hand we achieve the same bound by keeping triangles and forbidding short cycles of several other lengths.

Theorem 5.8 *Let G be a planar graph without cycles of lengths 4,5,6,7,8,9. Then $\chi(G) \leq 3$.*

Proof. Let us start with a minimal counterexample: let G be a planar graph without cycles of lengths 4,5,6,7,8,9, let $\chi(G) > 3$, and let G have the smallest possible number of vertices (we are not concerned with the number of edges).

As G is a minimal counterexample we may assume that every proper induced subgraph of G is 3-colorable. Let us first state some easy properties of G:

1. G has no vertices of degree ≤ 2.

Assume to the contrary that G has a vertex v of degree ≤ 2. By minimality $G - v$ admits a 3-coloring. Now neighbors of v might use at most $\deg(v) \leq 2$ different colors, hence there exists a free color for v, and a 3-coloring of $G - v$ can be extended to the whole G.

2. G is 2-connected.

Assume that v is a cutvertex of G and let G_1 be a component of $G-v$. By minimality there exist 3-colorings of both G_1+v and $G-G_1$, and these 3-colorings can be merged to a 3-coloring of G, after possibly shifting colors of G_1.

Now let us start with discharging. Let v be a vertex of G, its initial charge $c_0(v)$ is equal to $\deg(v)-6$. Similarly let $c_0(f) = 2\deg(f) - 6$ for every face $f \in F(G)$. Initial charges of vertices are compiled in the following table:

$\deg(v)$	3	4	5	6	7	8	9	...
$c_0(v)$	-3	-2	-1	0	1	2	3	...

Recycling the computation of (5.2) we have

$$\begin{aligned}&\sum_{v\in V(G)} c_0(v) + \sum_{f\in F(G)} c_0(f)\\ &= \sum_{v\in V(G)} (\deg(v)-6) + \sum_{f\in F(G)} (2\deg(f)-6) = -12.\end{aligned}$$

Observe that there does not exist a pair of triangles which share a common edge, as these would give rise to a cycle of length 4.

We shall redistribute charge from positively charged faces to vertices of small degree according to the following rules:

Rule 1a: Let v be a vertex of degree 3. If v is incident with three long faces then each face sends charge 1 to v.

Rule 1b: Let v be a vertex of degree 3. If v is incident with one triangle then each of the remaining two long faces sends charge $\frac{3}{2}$ to v.

Rule 2a: Let v be a vertex of degree 4. If v is incident with four long faces then each face sends $\frac{1}{2}$ to v.

Rule 2b: Let v be a vertex of degree 4. If v is incident with two triangles then each of the remaining two long faces sends charge 1 to v.

Rule 2c: Let v be a vertex of degree 4 which is incident with exactly one triangle t. Then the two long faces which are adjacent to t send v charge $\frac{1}{2}$. The long face antipodal to t sends v charge 1.

Rule 3: Let v be a vertex of degree 5 and f a long face incident with v. Then f sends charge $\frac{1}{2}$ to v.

Let us first describe a *reducible configuration*, a graph which cannot appear in a minimal counterexample G.

Let G_R be a graph on vertex set $\{u_i, v_i, z_i \mid 0 \le i \le 4\}$ so that z_i is adjacent to u_i and v_i, and v_i is also adjacent to u_{i-1} and u_i (all indices are taken modulo 5).

The vertices $\{u_i, v_i \mid 1 \leq i \leq 5\}$ induce the so called *inner cycle* of length 10, and vertices z_i are pasted as tips of triangles on alternate edges of the 10-cycle. The set $\{z_0, \ldots, z_4\}$ is the set of vertices of *attachment* of G_R.

We claim that

- G does not contain G_R as a subgraph, so that vertices of the inner cycle have degree 3 in G(we allow additional edges emanating from attachment vertices).

Assume that G contains G_R as a subgraph so that the vertices of the inner ring all have degree 3 in G. Let G' be obtained from G by deleting vertices of the inner cycle. As G is a minimal counterexample, there exists a 3-coloring c' of G'. We will show hot to extend c' to vertices of the inner ring.

Assume first that c' uses at most two colors on attachment vertices $z_0, \ldots, z_4$, without loss of generality these colors are 1 and 2. Let us use color 3 for every vertex $u_i, 0 \leq i \leq 4$, and finally let us color every vertex v_i with color $3 - c'(z_i)$.

If, on the other hand, c' uses all three colors on all attachment vertices, then we may without loss of generality assume that z_0 is the only attachment vertex colored with color 3. Now let us color vertices v_1, v_2, v_3, v_4 with color 3. Next we color vertices u_1, u_2, u_3 with color 1 or 2 which is not present at z_1, z_2, z_3, respectively. Finally let us color vertices u_4, v_0, u_0 in this order with colors 1 and 2. In this order the neighbors of a vertex currently being colored use exactly two colors, leaving one of the colors free.

Finally let us analyze the discharging procedure. Let $c_1 : V(G) \cup F(G) \to \mathbb{R}$ denote the final charge of vertices and faces of G.

Let v be a vertex. If $\deg(v) \geq 6$, then $c_1(v) = c_0(v) \geq 0$. If $\deg(v) \leq 4$, then $c_1(v) = 0$, as Rules 1a, 1b, 2a, 2b, and 2c effectively discharge a vertex of degree ≤ 4 and also cover every possible arrangement of long faces around v (note that no two triangles share a common edge).

If $\deg(v) = 5$, then v lies on at least three faces of length ≥ 5, hence its final charge $c_1(v)$ is at least $\frac{1}{2}$.

Let us now estimate the final charges of faces. If $\deg(f) = 3$, then $c_1(f) = c_0(f) = 0$. As G is 2-connected every face of G is a cycle, and since G has no cycles of lengths $4, \ldots, 9$, every face of G has length ≥ 10.

Now a face can send at most $\frac{3}{2}$ to each vertex incident with f. If $\deg(f) \geq 12$, then $c_1(v) \geq c_0(v) - \frac{3}{2}\deg(v) = 2\deg(v) - 6 - \frac{3}{2}\deg(v) \geq 0$.

Let $\deg(f) = 11$. Now $c_0(f) = 16 = 10 \cdot \frac{3}{2} + 1$. Now $c_1(f) < 0$ implies that f has sent $\frac{3}{2}$ to every vertex along f using Rule 1b. This in turn implies that (1) all vertices along f have degree 3, and (2) every vertex $v \in f$ is incident with a triangular face. Hence every second edge along f is incident with a triangle, which is absurd, as f has odd length.

Let $\deg(f) = 10$. Now $c_0(f) = 14 = 8 \cdot \frac{3}{2} + 2 \cdot 1 = 9 \cdot \frac{3}{2} + \frac{1}{2}$. We shall study all possibilities for which $c_1(f) < 0$. Let k_1 be the number of vertices of degree ≥ 4

along f. If $k_1 \geq 2$, then $c_1(f) \geq 0$, as there exist at least two vertices that receive at most 1 from f.

Assume next that f is incident with a vertex v of degree $d \geq 4$. If $d \geq 5$, then f sends at most $9 \cdot \frac{3}{2} + \frac{1}{2}$ charge away, which implies $c_1(f) \geq 0$. Hence v has degree 4 and v receives charge 1 from f. Let e_1 and e_2 be edges incident to both v and f. The faces adjacent to f along e_1 and e_2 are either both triangles or none is a triangle, as v receives charge 1 form f using either Rule 2b or Rule 2c. As all other vertices along f have degree 3 the parity of f implies that some vertex u along f is not adjacent to a triangle. Now f sends charge 1 to u as well and $c_1(f) \geq 0$.

As the last instance let us consider the case where all vertices along f have degree 3. If f is adjacent to ≤ 4 triangles, then f sends charge 1 to at least two vertices. As f cannot be consecutively adjacent to a pair of triangles (these would make a 4-cycle) the number of triangle neighbors of f is bounded above by 5. But this is impossible as 5 triangles adjacent to a face of length 10 whose vertices are of degree 3 forms a reducible configuration G_R. We infer that $c_1(f) \geq 0$ in all cases.

Thus we have arrived in contradiction as the total terminal charge is negative.
□

6 List coloring of planar graphs

Ordinary vertex coloring allows for every vertex v the same palette of colors, from where the color of v can be chosen from. In a more general setting we could have chosen palettes of colors for each vertex individually, and requiring that each vertex is assigned a color from its own palette — and still keeping the condition that adjacent vertices are assigned different colors.

Formally, let G be a graph. We shall call a mapping

$$L : V(G) \to \mathcal{P}\mathbb{N}$$

a *list assignment*, and call $L(v)$ the list of *admissible colors* for v.

Given L we say that G is *L-list-colorable*, if there exists a coloring c of G so that

$$c(v) \in L(v)$$

for every $v \in V(G)$. In other words, as c is a coloring adjacent vertices have to get different colors, and each vertex has to be assigned a color from its own list.

Now if G is L-list-colorable for every list assignment L satisfying $|L(v)| \geq k$ for every $v \in V(G)$, then we say that G is *k-choosable*. The *choice number of G*, $\mathrm{ch}(G)$, is the smallest k for which G is k-choosable.

6.1 Basic properties

Consider the list assignment which is constant — every vertex v of G is assigned the same list $L(v)$. Clearly the critical information is the order of $|L(v)|$, and an L-list-coloring can be identified with a $|L(G)|$-coloring. Hence the list-coloring-problem is a generalization of the ordinary (vertex) coloring problem, and we also have the following relation.

Proposition 6.1 *For every G we have $\chi(G) \leq \mathrm{ch}(G)$.*

But is there a difference between these two types of coloring? Consider the 6-cycle $v_1, v_2, v_3, v_4, v_5, v_6$, with the additional edge (main diagonal) v_1v_4, together with the list assignment $L(v_1) = L(v_4) = \{a, b\}$, $L(v_2) = L(v_5) = \{a, c\}$, and $L(v_3) = L(v_6) = \{b, c\}$. Let us temporarily disregard the diagonal edge. Observe that choosing color a for v_1 forces a unique extension of colors to the remaining vertices. Similarly, setting $c(v_1) = b$ forces a unique extension to the other vertices as well, the argument follows the other direction of the cycle. In either of cases we infer that $c(v_1) = c(v_4)$ are the same (and by analogy the same relations hold for colors of other pairs of antipodal vertices). As v_1 and v_4 are adjacent we infer that our graph is not L-list-colorable and consequently not 2-choosable. On the other hand it is bipartite.

Form this discussion it follows that $\mathrm{ch}(G)$ can be strictly larger than $\chi(G)$. In fact, the difference can be arbitrarily large.

Proposition 6.2 *For every k there exists a graph G so that* $\mathrm{ch}(G) - \chi(G) \geq k$.

Proof. Take the highly nonbalanced complete bipartite graph K_{k,k^k}. Let $v_1, \ldots, v_k$ be the vertices from the smaller color class, and let $L(v_i) = \{(i-1) \cdot k+1, (i-1) \cdot k+2, \ldots, i \cdot k\}$. There exists exactly k^k different k-sets of colors where exactly one color is chosen from each $L(v_i), i = 1, \ldots, k$, which are used as lists of admissible colors for the vertices of the larger color class. Now every choice of colors for vertices $v_1, \ldots, v_k$ appears as a list of admissible colors for one of the remaining vertices. □

Proposition 6.3 *If G is k-degenerate then* $\mathrm{ch}(G) \leq k + 1$.

Proof. We can color vertices greedily in the reverse order obtained by chipping vertices of smallest degree from the remaining graph, starting from G. When coloring v, it has at most k-colored neighbors, and at least one color from its own list $L(v)$ is free. □

We shall use Proposition 6.3 to partially prove the characterization of 2-choosable graphs.

Let G be a graph. A *1-core* of G is a minimal subgraph of G which can be obtained by repetitively deleting vertices of degree 1, starting from G. Observe that G is 2-choosable if and only if its 1-core is 2-choosable.

A *theta graph* $\Theta_{a,b,c}$ is the union of three internally disjoint paths of respective lengths a, b, and c between a pair of vertices.

Theorem 6.4 *Let G be a connected graph. G is 2-choosable if and only if its 1-core is either K_1 or $\Theta_{2,2,2k}$ for some integer $k \geq 1$.*

Proof. We shall only prove the reverse implication. Disregarding K_1 as a trivial case, we need to show that $\Theta_{2,2,2k}$ is a 2-choosable graph. To fix notation, let us assume that vertices $v_0, v_1, \ldots, v_{2k}$ form the path P_{2k} of length $2k$, and let u, w be the remaining two vertices adjacent to both v_0 and v_{2k}. Fix a 2-uniform list assignment L.

Let us first assume that vertices of P_{2k} are all assigned the same list of two admissible colors. This implies that we can two color P_{2k} in such a way, that v_0 and v_{2k} are assigned the same color. Then we can extend the coloring to u and w, as both their neighbors are colored the same.

Assume next that for some i the lists $L(v_i)$ and $L(v_{i+1})$ are not the same. Hence there is a color $c_i \in L(v_i) \setminus L(v_{i+1})$ and also a color $c_{i+1} \in L(v_{i+1}) \setminus L(v_i)$. Let us set $c(v_i) = c_i$ and let us first extend the coloring to $v_{i-1}, \ldots, v_1, v_0$ (this can be done as at each point a vertex has a single colored neighbor), setting $c(v_0) = c_0$. The only case where the choice of $c(v_0) = c_0$ cannot be extended to vertices u, w, and v_{2k} is the following: $c_0 \in L(u) \cap L(w)$ and $L(v_{2k}) = (L(u) \cup L(w)) \setminus \{c_0\}$. However in this case we may construct an alternative coloring c' by first setting $c'(v_{i+1}) = c_{i+1}$, extending it along $v_{i+2}, \dots, v_{2k}$, setting $c'(u) = c'(v) = c_0$, and proceeding along the remaining vertices of the P_{2k} towards v_i. $\square$

6.2 List coloring of planar graphs

If G is a planar graph then $\chi(G) \leq 4$. As G is 5-degenerate we have $\mathrm{ch}(G) \leq 6$ by Proposition 6.3. However it is not true that every planar graph is 4-choosable.

Proposition 6.5 (Voigt) *There exists a planar graph G which is not 4-choosable.*

Now an amazingly simple argument by Thomassen shows that lists of length 5 are sufficient.

Theorem 6.6 (Thomassen) *Every planar graph G is 5-choosable.*

Proof. A *near triangulation* is a plane graph G_C with a distinguished face f_C of length ≥ 3, all other faces being triangles. Let us denote with C the facial cycle of f_C. It is convenient to picture f_C as the outer face of G_C.

It is enough to prove Theorem 6.6 for near triangulations (a triangulation is also a near triangulation) as adding edges cannot make the coloring problem easier.

Let G_C be a near triangulation. We shall assume that

(1) there exists precolored consecutive vertices v_1, v_2 along C, whose colors are $c(v_1)$ and $c(v_2)$,

(2) for the remaining vertices $v_3, v_4, \dots, v_r$ along C we have $|L(v_i)| = 3$, and

(3) for every vertex x in the interior of C we have $|L(x)| = 5$.

and inductively claim that every smaller near triangulation satisfying above conditions admits a list coloring which extends the precoloring of v_1, v_2.

The induction basis is easy. If $|V(G_C)| = 3$, then v_1, v_2, v_3 are the only vertices of G_C. As $L(v_3) = 3$ there exists a color different from both $c(v_1)$ and $c(v_2)$ for v_3.

The induction step comes in two flavors.

Assume first that C has a chord, a pair of two nonconsecutive vertices v_i and v_j along C which are adjacent. Then let $C_1 = v_1, v_2, \dots v_i, v_j, v_{j+1}, \dots, v_r$, and $C_2 = v_i, v_{i+1}, \dots, v_{j-1}, v_j$ be the two cycles. Now let G_{C_1} and G_{C_2} be the two near triangulations induced by $\mathrm{Int}(C_1)$ and $\mathrm{Int}(C_2)$, *closed interiors* of C_1 and C_2, respectively.

Let c_1 be a coloring of G_{C_1}, its existence is guaranteed by induction. Now G_{C_1} and G_{C_2} share a pair of vertices v_i and v_j, which lie consecutively along C_2 and are already assigned colors $c_1(v_i)$ and $c_1(v_j)$. Inductively we can extend c_1 to the whole G_{C_2}.

If G_C has no chord, then let us focus on neighbors of v_r — vertices $v_{r-1}, v'_r, v'_{r+1}, \ldots, v'_{r+\ell}, v_1$ in this order around v_r. As G_C has no diagonals none of $v'_r, \ldots, v'_{r+\ell}$ lie on C. Let $G'_{C'}$ be the near triangulation $G_C - v_r$, whose outer face $f'_{C'}$ is determined by the facial cycle $C' = v_1, v_2, \ldots, v_{r-1}, v'_r, v'_{r+1}, \ldots, v'_{r+\ell}$. Let us construct an appropriate list assignment L' for vertices $v'_r, \ldots, v'_{r+\ell}$ to keep the induction going.

Let a, b be the colors in $L(v_r)$ different form $c(v_1)$. For every vertex v'_{r+i} let $L'(v'_{r+i})$ be obtained by removing colors a, b from $L'(v'_{r+1})$ (and possible additional colors to keep $L'(v'_{r+1})$ of length exactly 3 — this is not entirely necessary as additional options for colors do not make a coloring problem more difficult, but it simplifies the arguments).

Let us color $G_C - v_r$ inductively. Now vertices $v'_r, v'_{r+1}, \ldots, v'_{r+\ell}$ use neither a nor b. Also $c(v_1)$ is different from a and b. Hence we only need to take a look at $c(v_{r-1})$, and at least one of a, b is an appropriate choice for $c(v_r)$. $\square$

Let us note that by a similar argument Thomassen was able to give a simple proof of Grötzsch theorem (Theorem 5.7). Limited to planar graphs with girth ≥ 5 the induction asks for a precoloring of a path or a cycle of length ≤ 6 lying along the outer face of G, with lists of admissible colors of length ≥ 3, except for an independent set of vertices from the outer face, where lists of length 2 are allowed.

6.3 5-list coloring algorithm

In this subsection we shall try to describe a linear-time 5-list-coloring algorithm. Let us naively estimate the time complexity of Thomassen's reductions. Finding a possible chord in G_C takes time which is proportional to the sum of vertex degrees of vertices along C. This can be large on one hand and what is worse: it may not lead to a reduction, as we are unable to find a diagonal. Cutting away a vertex v_r takes time which is proportional to $\deg(v_r)$.

Fortunately we can remove a vertex v_r even if C contains chords, as long as no chord contains v_r.

In the reduction process let us start looking for a chord incident with v_r by checking the neighbors of v_r in the following order $v'_{r+\ell}, v'_{r+\ell-1}, \ldots, v'_{r+1}, v'_r$ (we are recycling notation from the proof of Theorem 6.6). If none of $v'_{r+\ell}, v'_{r+\ell-1}, \ldots, v'_{r+1}, v'_r$ lies on C we remove v_r and proceed recursively.

On the other hand let $v_{r-1}v'_{r+j} = v_{r-1}v_i$ be *the first* C-chord we find. We split G_C along $v_{r-1}v_i$ to obtain near triangulations G_{C_1} and G_{C_2}, where G_{C_1} contains both v_1 and v_2 along C_1. In this case v_{r-1} immediately precedes v_1 along C_1 and what is more important no chord of C_1 has an endvertex in v_{r-1}. Hence we first recursively find a 5-coloring of $G_{C_1} - v_{r-1}$, extend the coloring to v_{r-1}, and finally color G_{C_2} with v_{r-1} and $v'_{r+j} = v_i$ as the pair of precolored vertices.

We shall implicitly use notation from the proof of Theorem 6.6 in Algorithm 6.1. We shall treat the list assignment L as a global variable and assume that if $|L(v)| = 1$ the vertex v is colored with the only admissible color form its list $L(v)$.

```
   5-LIST-COLOR$(G, C, v_1, v_2)$
1  *chord-not-found* := true ;
2  $P := v_1$ ;
3  $i := \deg(v_r) - 3$;
4  **while** *chord-not-found* & $i \geq 0$ **do**
5      **if** $v'_{r+i} \notin C$ **then**
6          $P := v'_{r+i} + P$;
7          $i{-}{-}$
8      **else**
9          *chord-not-found* := false;
10         $P{:=}v'_{r+i} + P$;
11         $C'_1 = C_{v_1, v'_{r+i}} + P$;
12         $C_2 = C_{v'_{r+i}, v_r} + v_r v'_{r+i}$
13 let $a, b$ be colors from $L(v_{r-1})$ different from $c(v_1)$;
14 **for** $j := \deg(v_{r-1}) - 3$ *to* $i$ **do**
15     $L(v'_j) = L(v'_j) \setminus \{a, b\}$;
16 **if** *chord-not-found* **then**
17     **if** $\deg(v_3) = 2$ & $|C| = 3$ **then**
18         choose a color from $L(v_r) \setminus \{c(v_1), c(v_2)\}$ for $c(v_3)$
19     **else**
20         5-list-color$(G, C_{v_1, v_{r-2}+P}, v_1, v_2)$;
21         choose a color from $L(v_{r-1}) \setminus \{c(v_1), c(v_{r-1})\}$ for $c(v_r)$
22 **else**
23     5-list-color$(G, C'_1, v_1, v_2)$;
24     choose a color from $L(v_{r-1}) \setminus \{c(v_1), c(v'_{r+i})\}$ for $c(v_r)$;
25     5-list-color$(G, C_2, v'_{r+1}, v_{r-1})$
```

Algorithm 6.1: 5-LIST-COLOR computes a 5-list coloring of a planar near triangulation.

Theorem 6.7 *Let G be a triangulation, C a cycle in G, and v_1, v_2 two consecutive vertices along G. Assume that v_1 and v_2 are assigned different colors from their lists of admissible colors. Assume also that for every vertex $v \in V(C)$ we have $L(v) \geq 3$, and $L(u) \geq 5$ otherwise. Then* 5-LIST-COLOR(G, C, v_1, v_2) *computes an L-list-coloring of $G \cap \mathrm{Int}(C)$ in linear time i.e. in time proportional to the size of $G \cap \mathrm{Int}(C)$.*

Proof. The correctness of the resulting L-list-coloring follows from proof of Theorem 6.6.

Let us first explain the notation uses in the pseudocode. Line 2 initializes the path P, which will be used together with a segment of C in order to produce an outer cycle of the recursively smaller problem. P is a path along neighbors of v_r along their embedding order. Note that v_r is adjacent to v_{r-1}, v_1, and also to the vertices $v'_r, v'_{r+1}, \ldots, v'_{r+1}$. Hence $\deg(v_r)$ is really equal to $i+3$.

As C is a cycle (with its orientation implicit by notation — in the increasing order of indices) then $C_{x,y}$ denotes the subpath of C starting at vertex x an ending in vertex y, following the same orientation as C. This notation is for example used on line 11 and the next one as well.

Now 5-LIST-COLOR is a recursive algorithm. Let $m = |E(G \cap \mathrm{Int}(C))|$. We claim that if 5-LIST-COLOR(G, C, v_1, v_2) is the original call, then the procedure 5-LIST-COLOR has been recursively called at most m times. Let us charge each call of 5-LIST-COLOR with parameters G', C', v'_1, v'_2 to the edge immediately preceding $v'_1 v'_2$ along C'. Now each edge of $G \cap \mathrm{Int}(C)$ is charged with at most one call of 5-LIST-COLOR, and the claim is established.

Let us estimate (up to an additive constant) the time needed to execute a single call 5-LIST-COLOR(G, C, v_1, v_2) if we *disregard* the time needed for recursive calls. The while loop in line 4 and also the for loop on line 14 both take time which is proportional to the number of edges incident with v_r which are tested for being chords of C, let us denote this number with m_0.

If a chord was not found, then the recursion on line 20 proceeds with a near triangulation which has exactly $m_0 + 1$ triangular faces less than $G \cap \mathrm{Int}(C)$. If a chord was found, then the both graphs, upon which recursion is called on in lines 23 and 25 together have exactly m_0 triangular faces less that $G \cap \mathrm{Int}(C)$.

To put it all together. The procedure 5-LIST-COLOR is recursively called $O(n)$ times. Now the time used within a single call of 5-LIST-COLOR is proportional to the reduction of size of the problem. Hence the total running time is bounded above by the size of $G \cap \mathrm{Int}(C)$. □

7 Chordal graphs

Coloring vertices of a graph is considered as one of the most important combinatorial problems — this problems generalizes a vast collection of problems in combinatorial optimization, and the 4-color conjecture (now 4-Color Theorem) has served as the driving force for the whole mathematical discipline in much of the 20th century.

The divide-and-conquer approach to graph coloring would require us to split the graph G in half (or more generally in more parts) into parts G_1 and G_2 sharing only a handful of vertices S. We would like to merge respective colorings of G_1 and G_2 into a unified coloring of G, but this may not always be the case. Imagine that $G_1 \cap G_2$ contains vertices u, v, and that coloring of G_1 requires u and v to receive different colors, yet the coloring of G_2 would want to see u and v colored with the same color. We could save the day by increasing the number of available colors, but this is generally not the strategy we want to apply.

The class of *chordal graphs* (formally defined below) introduces a class of graphs, on which merging colorings should not pose a problem. We shall see they are interesting in their own right.

7.1 k-trees

Let us first look at a class of graphs which directly generalizes trees. We shall call a complete subgraph of G a *clique*, and if a clique contains exactly k vertices it will be called a *k-clique*.

We can construct trees inductively starting form K_1 by attaching pedant edges to an already constructed graph — a pendant edge is a *new* vertex joined to a complete graph (of order one) in the original graph.

The class of *k-trees* is defined inductively:

(T1) K_k is a k-tree,

(T2) if S is a k-clique in a k-tree G, then adding a *new* vertex v making it adjacent to every vertex of S produces a k-tree G'.

A tree is a 1-tree, and 2-trees can be obtained by pasting triangles along edges — cliques of order 2.

Let us state without proof some properties of k-trees. Let G be a k-tree, then

1. G is a connected graph,
2. G contains a k-clique,
3. G does not contain a $(k+2)$-clique,
4. $|E(G)| = k|V(G)| - \frac{1}{2}k(k+1)$,
5. every minimal separator in G is a k-clique,

6. if S is a minimal separator and (G_1, G_2) the corresponding separation, then both G_1 and G_2 are k-trees.

Now the last pair of properties effectively enables a divide-and-conquer approach to run on k-trees. Partial solutions on G_1 and G_2 can be recombined along a clique separator S into a solution on the whole G.

7.2 Chordal graphs

Let us begin with a definition. A graph G is *chordal* if G does not contain induced cycles of length ≥ 4. Equivalently, if every cycle C of length ≥ 4 in G contains a *chord*.

Proposition 7.1 *Every induced subgraph of a chordal graph is chordal itself. The class of chordal graphs is closed under induced subgraphs.*

Proof. Let $G_1 \leq_i G_2$, and let C be an induced subgraph in G_1. As C is also induced in G_2, C has a chord. □

The above proposition was effectively only taking into account that an induced subgraph of an induced subgraph is itself induced. The below one is more technical.

Proposition 7.2 *G is a chordal graphs if and only if every minimal separator in G is a clique.*

Proof. ($\Rightarrow$) Let G be a chordal graph, and let S be a minimal separator in G, so that $x, y \in S$. We would like to see that $xy \in E(G)$. Assume this is not the case.

Let (G_1, G_2) be a corresponding separation — a pair of graphs so that $G_1 \cup G_2 = G$ and $G_1 \cap G_2 = S$. Let $H_1 = G_1 - S + x + y$, and symmetrically $H_2 = G_2 - S + x + y$. Let P_1 be a shortest $x - y$-path in H_1 and P_2 be a shortest $x - y$-path in P_2. As P_1 is a shortest path it is induced, and a similar observation holds for P_2 as well. Now $P_1 \cup P_2$ is a cycle of length ≥ 4, and no internal vertex of P_1 can be adjacent to an internal vertex of P_2, as S is a separator. The edge xy is then the only possible chord of $P_1 \cup P_2$.

($\Leftarrow$) Assume now that every minimal separator in G is a clique. Let $C = v_0 v_1 v_2 v_3 \dots v_{k-1} v_0$ be a cycle of length $k \geq 4$. If $v_0 v_2 \in E(G)$, then C has a chord. Hence we may assume that v_0 and v_2 are not adjacent. Let S be a minimal separator separating v_0 from v_2. By construction S contains v_1 and at least one of vertices $v_3, \dots, v_{k-1}$. As S is a clique, C contains an edge between v_1 and at least one of $v_3, \dots, v_{k-1}$, which is a chord in C. □

We have an immediate corollary.

Corollary 7.3 *Every k-tree is a chordal graph.*

Let us call $v \in V(G)$ a *simplicial vertex* if its neighborhood $N(v)$ is a clique.

Lemma 7.4 *Let G be a chordal graph. Then G is either a complete graph or G contains at least two nonadjacent simplicial vertices.*

Proof. Let us perform induction on $n = |V(G)|$. If $n = 1, 2$ then the result is clear.

Let us assume that $n > 2$, that G is not a complete graph, and also that every chordal graph on less than n vertices satisfies the desired condition. Let a and b be a pair of nonadjacent vertices in G, and let S be a minimal separator separating a and b. Take the corresponding separation (G_a, G_b) for which $a \in V(G_a)$ and $b \in V(G_b)$. Inductively, G_a contains a pair of simplicial vertices which are either nonadjacent or G_a is a complete graph. G_b shares this same property.

Hence G_a contains a simplicial vertex $v_a \notin S$, and similarly let v_b be a simplicial vertex in G_b which is not contained in S. Now v_a and v_b are nonadjacent and are also simplicial vertices in G. $\square$

7.3 Recognizing chordal graphs

A *recognition problem* of a graph class $\mathcal{G}$ can be described as the following decision problem:

input: graph G.

output: does $G \in \mathcal{G}$?

The recognition problem for the class of chordal graphs can be naively solved according to the following procedure:
Keep deleting simplicial vertices, until no longer exist. If the final graph is empty, then the original graph G is chordal, otherwise G is not chordal.

The naive procedure is polynomial, but its time complexity may be as bad as $\Theta(n^4)$ in the general case. We should look for a more efficient algorithm.

Let G be a graph. A *perfect elimination ordering* or *PEO* is an ordering $v_1, v_2, v_3, \ldots, v_n$ of vertices of G, so that v_i is a simplicial vertex in $G[v_i, v_{i+1}, \ldots, v_n]$.

Theorem 7.5 *G is a chordal graph if and only if G admits a perfect elimination ordering.*

Proof. ($\Rightarrow$) Let G be a chordal graph, and let us inductively assume that every chordal graph with fewer vertices has a PEO. Let v be a simplicial vertex of G, and let $G' = G - v$. Assume that the sequence

$$v'_1, v'_2, \ldots, v'_{n-1}$$

is a PEO for G'. Then

$$v, v'_1, v'_2, \dots, v'_{n-1}$$

is a PEO for G.

($\Leftarrow$) Let G be a graph and

$$v_1, v_2, \dots, v_n$$

its PEO. Assume that C is a cycle in G of length ≥ 4. Let $v_i \in V(C)$ with the smallest possible index, and let v_j, v_k be neighbors of v_i along C. As v_i is simplicial in $G[v_i, \dots, v_n]$ vertices v_j and v_k are adjacent. Hence C has a chord. □

Let us start with a lemma characterizing perfect elimination orderings.

Lemma 7.6 *Let $\mathcal{O} = v_1, v_2, \dots, v_n$ be an ordering of vertices of a graph G. Then $\mathcal{O}$ is not a perfect elimination ordering if and only if there exists an induced path $v_{i_0}v_{i_1}v_{i_2}\dots v_{i_{k-1}}v_{i_k}$ of length $k \geq 2$, so that for every $j \in \{1, \dots, k-1\}$ we have $i_j < i_0$ and $i_j < i_k$.*

Proof. ($\Rightarrow$) Assume that $\mathcal{O}$ is not a PEO. Then there exists a vertex v_i so that its neighbors with higher indices do not induce a clique: there exists nonadjacent vertices v_{i_1} and v_{i_2} so that $v_iv_{i_1} \in E(G)$, $v_iv_{i_2} \in E(G)$, and also $i_1 > i$ and $i_2 > i$. Hence $v_{i_1}v_iv_{i_2}$ is the desired path.

($\Leftarrow$) Let P be an induced path so that its endvertices have bigger indices than any of the internal vertices. Let $v_j \in V(P)$ be a vertex with the smallest index j along P. Its neighbors along P have bigger indices and are not adjacent. Hence $\mathcal{O}$ is not a PEO. □

Is there an efficient way to find a PEO of a chordal graph? Indeed there is. The procedure *maximum cardinality search* inductively selects vertices, that have maximal number of neighbors among already selected vertices.

```
  MaximumCardinalitySearch($G$)
1 $PEO = \emptyset$;
2 label all vertices $white$;
3 for $i = n$ to $1$ do
4     let $v_i$ be a white vertex with maximal number of neighbors among black
      vertices $v_{i+1}, \dots v_n$;
5     $PEO = v_i + PEO$;
6     label $v_i$ $black$
7 return $PEO$
```

Algorithm 7.1: MaximumCardinalitySearch computes a PEO of a chordal graph G.

Proposition 7.7 *If G is a chordal graph, then* MaximalCardinalitySearch(G) *computes a PEO of G in time $O(n+m)$.*

Proof. Let us for each white vertex v with r_v denote the number of neighbors of v among black vertices. In the beginning $r_v = 0$ for every vertex v.

We can choose the next vertex v_i to enter PEO in constant time, but we have to refresh the numbers of black neighbors for every white vertex. This takes at most time proportional to $\deg(v_i)$ time. Hence the total time complexity is equal to

$$O(n) + O(\sum_{v \in V(G)} \deg(v)) = O(n+m).$$

We have yet to show that the computed sequence

$$\mathcal{O} = v_1, v_2, \ldots, v_n$$

of vertices is indeed the perfect elimination ordering.

Assume this is not the case. By Lemma 7.6 there exists an induced path $P = v_{i_0} v_{i_1} \ldots v_{i_{k-1}} v_{i_k}$ of length $k \geq 2$, so that for every $j \in \{1, \ldots, k-1\}$ we have $i_j < i_0$ and $i_j < i_k$. We shall also assume that $i_0 < i_k$ and also that i_0 is the biggest possible index allowing a path with this desired property.

Let $N^+(v_i) = N(v) \cap \{v_j \mid j > i\}$. Let v_x be a neighbor of v_{i_0} so that $x > i_0$. By maximality of i_0 the path $P_x = v_x v_{i_0} \ldots v_{i_{k-1}} v_{i_k}$ has a chord, which necessarily attaches to v_x, since P is induced. Let j be maximal possible subscript $j \in \{1, \ldots, k\}$ so that $v_x v_{i_j} \in E(G)$. If $j < k$ then $v_x v_{i_j} \ldots v_{i_k}$ contradicts the choice of P and minimality of i_0. Hence v_x is adjacent to v_{i_k}.

This implies that $C_x = v_x v_{i_0} \ldots v_{i_{k-1}} v_{i_k} v_x$ is a *cycle* of length ≥ 4, so it must have a chord. As argued above, every chord of C_x is incident with v_x. Let us again choose a maximal possible subscript $j \in \{1, \ldots, k-1\}$ so that $v_x v_{i_j} \in E(G)$. If $j < k-1$ then the cycle $v_x v_{i_j} \ldots v_{i_{k-1}} v_{i_k} v_x$ is a chordless cycle of length ≥ 4, which is absurd. Hence v_x is a neighbor of $v_{i_{k-1}}$.

Hence every vertex $v_x \in N^+(v_{i_0})$ is also a neighbor of $v_{i_{k-1}}$. As v_{i_k} is also a neighbor of $v_{i_{k-1}}$ and not a neighbor of v_{i_0} we infer that $v_{i_{k-1}}$ has strictly more neighbors in $\{v_{i_0+1}, \ldots, v_n\}$ than v_{i_0}. This is a contradiction as the ordering $\mathcal{O}$ was produced by maximum cardinality search. □

Running Algorithm 7.1 on an input graph G, which is not chordal, returns an ordering $\mathcal{O}$ of vertices, which is not a perfect elimination ordering. We shall describe an algorithm TestChordal which decides on input G and $\mathcal{O}$, whether $\mathcal{O} = v_1, v_2, \ldots, v_n$

is a perfect elimination ordering of G.

```
TestChordal($G, \mathcal{O}$)
1 for $i = 1$ to $n$ do
2     let $v_j$ be a neighbor of $v_i$ with $j > i$ and $j$ as small as possible;
3     foreach $v_k$ a neighbor of $v_i$ with $k > j$ do
4         if $v_j v_k \notin E(G)$ then
5             return False
6 return True
```

Algorithm 7.2: TestChordal($G, \mathcal{O}$) test whether $\mathcal{O}$ is a PEO for G.

Theorem 7.8 TestChordal($G, \mathcal{O}$) *correctly determines if $\mathcal{O}$ is a PEO of G.*

Proof. Assume that $\mathcal{O} = v_1, \dots, v_n$ is not a PEO for G. Then there exists a vertex v_ℓ, and a set $N^+_{v_\ell} = \{v_{\ell_1}, v_{\ell_2}, \dots, v_{\ell_r}\}$ which does not induce a clique and so that $\ell < \ell_1 < \cdots < \ell_r$. Let us choose ℓ as large as possible. If a pair of vertices from $\{v_{\ell_1}, v_{\ell_2}, \dots, v_{\ell_r}\}$ are not adjacent the maximality of ℓ implies that v_{ℓ_1} is not adjacent with v_{ℓ_s} for some $s \in \{2, \dots, r\}$. Now TestChordal($G, \mathcal{O}$) returns $False$ on line 5 with v_i, v_j, v_k equal to $v_\ell, v_{\ell_1}, v_{\ell_s}$.

Conversely, if $\mathcal{O}$ is a PEO then every vertex v_ℓ is simplicial in $G[v_\ell, v_{\ell+1}, \dots, v_n]$ and hence TestChordal($G, \mathcal{O}$) returns $True$. □

7.4 Hard problems on chordal graphs may be easy

Consider the following two problems on the class of general graphs.

MaxClique	OptColoring
input: graph G.	**input:** graph G.
output: maximal clique of G.	**output:** $\chi(G)$ coloring of G.

It is known that both MaxClique and OptColoring are NP-hard combinatorial problems on the class of general graphs. On the other hand PEO makes these problems easy if G is chordal.

Theorem 7.9 *Let G be a chordal graph. Then we can compute both the maximal clique and an optimal vertex coloring in linear time.*

Proof. Let $v_1, v_2, \dots, v_n$ be a PEO of G, and let $\deg^+(v_j) = |N^+(v_j)|$. Let K be a maximal clique in G, and $v_i \in V(K)$ a vertex of smallest index in K. Now by maximality of K we have $1 + \deg^+(v_i) \le |V(K)|$ and by minimality of i we have the reverse as $V(K) \subseteq \{v_i\} \cup N^+(v_i)$.

Let us greedily color vertices in the reverse of PEO $v_n, v_{n-1}, \dots, v_1$. The number of colors needed is equal to $1 + \max\{\deg^+(v_j) \mid v_j \in V(G)\}$ which is by above argument the size of the maximal clique in G. Hence the greedy coloring is optimal. □

7.5 Intersection graphs and Helly property

Let $\mathcal{A} = \{A_1, \dots, A_n\}$ be a collection of sets, which defines an *intersection graph $G_{\mathcal{A}}$* in a natural way: vertices of $G_{\mathcal{A}}$ are elements of $\mathcal{A}$, where A_i and A_j are adjacent if and only if their intersection is nonempty, $A_i \cap A_j \neq \emptyset$.

It is not difficult to see that every graph is an intersection graph of an appropriate collection of sets (hint: for each vertex v consider the set containing v and the edges incident with v).

A collection of sets $\mathcal{A} = \{A_1, \dots, A_n\}$ satisfies the *Helly property* if for every nonempty index set $I \subseteq \{1, \dots, n\}$ the following implication holds:

$$(\forall j, k \in I : A_j \cap A_k \neq \emptyset) \Longrightarrow \bigcap_{i \in I} A_i \neq \emptyset \tag{7.1}$$

In other words, if for an index subset T all pairwise intersections are nonempty, then also the intersection over all indices in T is nonempty.

Lemma 7.10 *Let T be a tree, and $\mathcal{T} = \{T_1, \dots, T_m\}$ a collection of subtrees. Then $\mathcal{T}$ satisfies the Helly property.*

Proof. Let us start with three vertices a, b, c, and let P_{ab}, P_{bc}, P_{ac} be three subpaths of T with endvertices a, b and b, c and a, c, respectively. As T is a tree and contains no cycles, $P_{ab} \cap P_{bc} \cap P_{ca}$ is nonempty, and consequently these paths contain a common vertex x.

Let $T_1, \dots, T_m$ be a collection of subtrees so that $\forall j, k \in \{1, \dots, m\}$ the intersection $T_j \cap T_k$ is nonempty. We shall prove that for every nonempty index set $I \subseteq \{1, \dots, m\}$ the intersection

$$\bigcap_{i \in I} T_i \neq \emptyset, \tag{7.2}$$

using an inductive argument on $|I|$.

Obviously the cases $|I| \leq 2$ are valid. Let us take an index subset $I' = \{i_1, i_2, \dots, i_r\}$ of order $r \geq 3$, an let us assume that (7.2) holds for every index subset I of order $< r$. By assumption, $T_{i_1} \cap T_{i_2} \cap \dots \cap T_{i_{r-1}}$, $T_{i_2} \cap T_{i_3} \cap \dots \cap T_{i_r}$, and $T_{i_1} \cap T_{i_m}$ are nonempty, and contain vertices a, b, and c, respectively. Hence each of $T_{i_1}, \dots, T_{i_r}$ contains at least two of a, b, c, and hence at least one of P_{ab}, P_{bc}, P_{ac}. As $P_{ab} \cap P_{bc} \cap P_{ca}$ is nonempty we also have

$$\bigcap_{i \in I'} T_i \neq \emptyset.$$

This completes the proof. □

8 Tree decomposition

Let P be a hard problem on the class of graphs: imagine that we want to decide whether G has a vertex-cover of small order, or we want to construct a hamilton cycle, or maybe a 3-vertex coloring. These three problems are difficult, more precisely — using the language of computational complexity theory — NP-hard.

On the other hand if our input graph is a tree, these problems turn out to be easily solvable. Of course a tree T has no hamilton cycle, and of course T is 3-vertex colorable, but even the minimal vertex cover can be constructed without much fuss (how exactly?).

It should come to no surprise if (most common) problems, which are hard on the class of general graphs, tend to become easy if the input graph is not too far from a tree.

It is the concept of *similarity to trees* that we shall study in the sequel.

8.1 Definition

Let G be a graph. A *tree decomposition* of G is a pair $(T, \mathcal{B})$, where T is a tree, and $\mathcal{B} = \{B_t \mid t \in V(T)\}$ is a collection of vertex sets of G, called *bags*, indexed by nodes of T satisfying

(T1) $\bigcup_{t \in V(T)} B_t = V(G)$

(T2) for every edge $uv \in E(G)$ there exists a bag B_t so that $u, v \in B_t$, and

(T3) for all $t, t', t'' \in V(T)$ for which t' lies between t and t'' we have $B_t \cap B_{t''} \subseteq B_{t'}$.

Let us first comment on the properties of tree width. (T1) requires that every vertex of G belongs to a bag, it may belong to several bags, though. (T2) represents containment of edges in bags — formally bags contain vertices, but if a pair of vertices are adjacent, then they both have to appear in at least one of the bags together. They do not need to appear together in every bag. The last property (T3) is best explained from a vertex' point of view: a vertex v may belong to several bags. But if v belongs to a pair of bags B_t and $B_{t''}$ then v belongs to every bag $B_{t'}$ which is indexed by a node t' which lies between t and t'' in T.

Let us denote by T_v the subgraph of T induced by the nodes of T that contain v in their bags. Then (T3) is equivalent to (T3')

(T3') for every vertex $v \in V(G)$ the subgraph T_v is a tree (= is connected).

A single graph may admit several tree decompositions. One of the options is to take a trivial tree T of order 1, and place every vertex of a graph in a single bag. This clearly satisfies the axioms (T1—T3), but may not be an option we are looking for. We would namely like to have bags as small as possible:

The *width* of a tree decomposition $(T, \mathcal{B})$ is equal to $\max_t\{|B_t|\} - 1$, and the *tree-width* of G, $\mathrm{tw}(G)$, is the smallest width of a tree decomposition of G.

The structure of G forces that some of the bags are large.

Proposition 8.1 *Let K be a clique in G, and let $(T, \mathcal{B})$ be a tree decomposition of G. Then there exists a bag B_t so that $V(K) \subseteq B_t$.*

Proof. Let $\{v_1, v_2, \ldots, v_k\}$ be the vertices of K, and let $T_{v_1}, \ldots, T_{v_k}$ be trees induced by nodes of T containing $v_1, \ldots, v_k$ in their bags, respectively. For $i, j \in \{1, \ldots, k\}$ we have $v_i \sim v_j$, and hence $T_{v_i} \cap T_{v_j} \neq \emptyset$. By Helly property (Lemma 7.10) the trees $T_{v_1}, \ldots, T_{v_k}$ share a common node t. Hence B_t contains all vertices of a clique K. □

Proposition 8.2 *Tree width is* minor monotone, *if $H \leq_m G$ then $\mathrm{tw}(H) \leq \mathrm{tw}(G)$.*

Proof. Let $(T, \mathcal{B})$ be an optimal tree decomposition of G, i.e. its width is equal to $\mathrm{tw}(G)$. We shall construct tree decompositions of width at most $\mathrm{tw}(G)$ for $G - v$, $G - e$, and G/e, respectively. The result shall follow as every minor of G is obtained by a series of vertex and edge deletions, and edge contractions starting from G.

Removing v from every bag of $\mathcal{B}$ results in a tree decomposition of $G - v$, and a tree decomposition of G is also a tree decomposition of $G - e$. Assume that $e = uv$ and let x_{uv} be the new vertex obtained by contracting $e = uv$ in G/e. Replacing vertices u, v by a single vertex x_{uv} in every bag of $\mathcal{B}$ results in a tree decomposition of G/e, whose width may even drop by one. This completes the proof. □

Proposition 8.3 *Let G be a graph, and let (G_1, G_2) be a separation of G so that $G_1 \cap G_2$ is a clique. Then*

$$\mathrm{tw}(G) = \max\{\mathrm{tw}(G_1), \mathrm{tw}(G_2)\}.$$

Proof. Let $(T_i, \mathcal{B}_i)$, $i = 1, 2$, be optimal tree decompositions of G_1 and G_2, and let $K = G_1 \cap G_2$. As K is a clique both a bag $B_{t_1} \in \mathcal{B}_1$ and a bag $B_{t_2} \in \mathcal{B}_2$ contain all vertices of K. Joining T_1 and T_2 in a single tree by adding an edge t_1t_2 creates a tree decomposition $(T, \mathcal{B}_1 \cup \mathcal{B}_2)$ of G of the appropriate width. □

An immediate consequence of Proposition 8.3 is the following.

Proposition 8.4 *Let G be a graph and $G_1, \ldots, G_k$ its blocks. Then*

$$\mathrm{tw}(G) = \max_{i \in \{1, \ldots, k\}} \{\mathrm{tw}(G_i)\}.$$

8.2 Examples

If G is a singleton graph then its sole vertex can be put in a singleton bag of a tree with a single node. This is a tree decomposition of width 0, and we clearly cannot do better.

Let G be a tree. Let T be a total subdivision of G, obtained by inserting one additional vertex of degree 2 on every edge of G. Now $V(T)$ splits into $V_1 \cup V_2$, where nodes of V_1 represent vertices of G, and nodes in V_2 correspond to edges of G. Let the bags be built correspondingly: for $t \in V_1$ let B_t be the singleton $\{v\}$ so that t corresponds to v. If $t' \in V_2$ then $B_{t'} = \{u', v'\}$ so that t' corresponds to the edge $u'v'$. We have thus constructed a tree decomposition of width 1.

Let us note at this point that somehow artificial definition of width of a decomposition (subtracting 1 from the size of bags) is in place with the sole reason that trees have tree-width equal to 1. By 8.4 we infer:

(6) $\mathrm{tw}(G) = 0$ *if and only if* G *is edgeless,* $\mathrm{tw}(G) \leq 1$ *if and only if* G *is a forest.*

Let G be a clique on n vertices. Choosing a single-node tree and putting the whole vertex-set $V(G)$ in its bag is a decomposition of width $n-1$. On the other side by Proposition 8.1 every tree decomposition of G has width $\geq n-1$.

If G contains a pair of nonadjacent vertices x and y, then we can construct a tree decomposition with bags $V(G-x), V(G-y)$, which has width $n-2$. To sum it together:

(7) *Let* G *be a graph on* n *vertices. Then* $\mathrm{tw}(G) = n-1$ *if and only if* G *is complete.*

Next, take a cycle C_n. Contracting a single edge in C_n results in a cycle of length $n-1$. Repeatedly contracting edges will get us to $C_3 = K_3$. As tree-width might have decreased in the process we have $\mathrm{tw}(C_n) \geq \mathrm{tw}(K_3) = 2$.

In order to construct a decomposition of C_n of width 2 let us first pick a pair of vertices at maximal distance on C_n and label these with v_1, v_n. The rest of C_n can be seen as a pair of internally disjoint $v_1 - v_n$ paths, and let us label internal vertices along these with even $v_2, v_4, \ldots, v_{2\lfloor\frac{n-1}{2}\rfloor}$ and odd $v_3, v_5, \ldots, v_{2\lfloor\frac{n-1}{2}\rfloor+1}$ indices. The tree decomposition of width 2 of C_n contains bags $\{\{v_1, v_2, v_3\}, \{v_2, v_3, v_4\}, \ldots, \{v_{n-2}, v_{n-1}, v_n\}\}$ strung along a path.

(8) *If* C_n *is a cycle then* $\mathrm{tw}(C_n) = 2$

Finally let us construct a tree decomposition of width n of a planar quadrangular $n \times n$ grid $P_n \Box P_n$. Vertices of $P_n \Box P_n$ are pairs of integers (x, y), where $x, y \in \{1, \ldots, n\}$ and pairs of integers (x_1, y_1) and (x_2, y_2) are adjacent if and only if $|x_1 - x_2| + |y_1 - y_2| = 1$.

For every *vertical* edge $\{(x_0, y_0), (x_0, y_0+1\}$ let us construct a bag $B_{x_0,y_0} = \{(x, y_0) \mid x \leq x_0\} \cup \{(x, y_0+1) \mid x \geq x_0\}$, and let us make bags $B_{x,y}$ and $B_{x',y'}$ adjacent in the tree (to be precise their corresponding nodes) if the pair (y', x') is the successor of (y, x) in the lexicographic ordering (note the transposition of coordinates). Hence:

(9) $\mathrm{tw}(P_n \Box P_n) \leq n$

8.3 Properties of tree decomposition

Let us first argue that a tree decomposition $(T, \mathcal{B})$ encompasses separation properties of the graph G. Let t_1t_2 be an edge of T. Deleting t_1t_2 splits T into a pair of trees T_1, T_2 (we adjust notation so that $t_i \in V(T_i)$), and let G_i, $i = 1, 2$, be the subgraph of G induced by the bags of T_i.

Proposition 8.5 *With notation as above, if $(T, \mathcal{B})$ is a tree decomposition of G and $t_1t_2 \in E(T)$ then $B_{t_1} \cap B_{t_2}$ separates $V(G_1)$ from $V(G_2)$.*

Proof. We need to show that every $V(G_1) - V(G_2)$-path P in G contains a vertex of $S = B_{t_1}\mathcal{B}_{\sqcup_{\in}}$. Assume to the contrary that there exists a $V(G_1) - V(G_2)$ path P avoiding S. This implies that a pair of consecutive vertices u_1, u_2 along P belong to $V(G_1) \setminus S$ and $V(G_2) \setminus S$, respectively. As u_1u_2 is an edge in G there exists a node $t \in V(T)$ so that both $u_1, u_2 \in B_t$. By symmetry we may assume $t \in T_1$. As $u_2 \in V(G_2) \setminus S$ there is a node $t' \in V(T_2)$ so that $u_2 \in B_{t'}$.

Now by $(T3')$ the vertex u_2 belongs to every bag B_{t^*} where t^* lies between t and t'. As both t_1 and t_2 lie between t and t' we have a contradiction as $u_2 \notin B_{t_1} \cap B_{t_2} = S$. $\square$

We say that a tree decomposition $(T, \mathcal{B})$ of G is *linked* if for every pair of bags B_t and $B_{t'}$ and every integer k there exist k disjoint $B_t - B_{t'}$ paths or a bag $B_{t''}$ of order $< k$, where t'' is a node of T lying between t and t'.

Linked tree-decompositions model connectivity properties in an exact manner, and linked tree-decompositions exist, as the next proposition states.

Proposition 8.6 *If G admits a tree decomposition of width w then G also admits a linked tree decomposition of the same width.*

Let us first argue that we can assume that the bags of a decomposition are incomparable.

Proposition 8.7 *Let $(T, \mathcal{B})$ be a tree decomposition of width w of graph G. Then there exists a tree decomposition $(T', \mathcal{B}')$ of G having incomparable bags.*

Proof. Assume the tree decomposition contains a pair of bags B_t, B_{t^*}, so that $B_{t^*} \subseteq B_t$. Let P be a $t^* - t$-path in T, let t^*t_0 be the first edge along P, and let $t_1, t_2, \ldots, t_k$ be the remaining (apart from t_0) neighbors of t^* in T.

Let us construct an alternative tree decomposition of G. Let T' be the tree obtained by deleting the edge t^*t_0, and adding edges t_it, for $i \in \{1, \ldots, k\}$. Let $\mathcal{B}' = \mathcal{B} \setminus \{B_{t^*}\}$, the nodes of T' (viewed as nodes of T as well) retain their respective bags. As $B_{t^*} \subseteq B_t$ both (T1) and (T2) clearly hold.

Assume, contrary to (T3'), that there exist nodes $\tau_1, \tau_2 \neq t^*$ and a vertex v, so that T' does not contain a $\tau_1 - \tau_2$-path, yet there is a $\tau_1 - \tau_2$-path P_τ in T. P_τ contains the edge t^*t_0, and consequently also vertices t_0 and t_j for some $j \in \{1, \ldots, k\}$.

Now the subpaths of $P_\tau - t^*$ can be joined in T using t, as both t_0 and t_j are adjacent to t. This is a contradiction terminating the proof. $\square$

The order of T in a tree decomposition can be arbitrarily large compared to the order of G, for example if many nodes of T share the same bag. On the other hand if bags are incomparable the order of G effectively bounds the number of nodes in T.

Proposition 8.8 *Assume that the bags of a decomposition* $(T, \mathcal{B})$ *are pairwise incomparable. Then* $|V(T)| \leq |V(G)|$.

Proof. We can prove the inequality by induction on the number of vertices of G. The basis where G has a single vertex holds.

Let t be a leaf of T, and t' its unique neighbor. Let $U = B_t \setminus B_{t'}$. The assumption tells us that $U \neq \emptyset$, and for every vertex $u \in U$ the bag B_t is the sole bag containing u, by (T3). Now $(T - t, \mathcal{B} \setminus \{B_t\})$ is a tree decomposition of $G - U$ whose bags are pairwise incomparable. Inductively $|V(G - U)| \geq |V(T - t)|$ and consequently $|V(G)| \geq V(T)$. $\square$

We can, if we want, impose additional properties on the tree decomposition.

Proposition 8.9 *Let* $(T, \mathcal{B})$ *be a tree decomposition of width* w *of graph* G. *Then there exists a tree decomposition* $(T', \mathcal{B}')$ *of the same width having the following properties:*

(N1) *for every edge* t_1t_2 *in* $E(T)$ *we have either* $B_{t_1} \subseteq B_{t_2}$ *or* $B_{t_2} \subseteq B_{t_1}$,

(N2) *neighboring bags differ by at most one element,*

(N3) T *is subcubic.*

Proof. We shall only do a sketch.For every pair of neighboring nodes t_1, t_2 with incomparable bags let us subdivide edge t_1t_2 with a new note t_{12} and set $B_{T_{12}} =$

$B_{t_1} \cap B_{t_2}$. Next if for some edge t_3t_4 the difference $B_{t_4} \setminus B_{t_3} = U$ with $|U| \geq 2$, we can subdivide the edge t_3t_4 with additional $|U| - 1$ vertices, and let the orders of bags raise by one on the $t_3 - t_4$-path.

Finally if t is a node with neighbors $t_0, t_1, \ldots, t_{k+1}$ where $k \geq 2$ we can replace t with a path of k vertices all carrying the same bag B_t. □

8.4 Nice tree decomposition

A rooted tree decomposition $(T, \mathcal{B})$ with root r is *nice* if apart from (N1), (N2), and (N3) it also satisfies the the following two properties:

(N4) If t has two sons t_1, t_2 then $B_{t_1} = B_{t_2} = B_t$.

(N5) Every leaf t satisfies $|B_t| = 1$.

Proposition 8.10 *Let G be a graph and $(T, \mathcal{B})$ a tree decomposition of width w and order $O(n)$. Then we can in $O(wn)$ time construct a nice tree decomposition $(T_n, \mathcal{B}_n)$ of G having the same width w.*

Proof.[sketch] We have to take into account that $|V(T)|$, $|E(T)|$, and the number of leaves of T are all $O(n)$. Subdividing every edge (if needed to achieve (N1)) takes time $O(n)$, and increases the number of nodes and edge by a constant factor. Splitting all high degree vertices takes time proportional to $O(n)$, and yields a tree which still has $O(n)$ edges, let us also split r so that its degree is at most 2. Now for every edge the orders of bags differ by at most $w + 1$, hence in $O(wn)$ time we can satisfy (N2).

Finally let us hang the resulting tree by its root r, and observe that every vertex has at most two sons. If t has two sons and a bag $B_{t'}$ differs from B_t, let us subdivide the edge tt' with an additional vertex t'', and let us set $B_{t''} = B_t$. This takes time proportional to the number of edges in T, hence the total time spent is $O(wn)$.
□

Let us associate names to nodes of a nice tree decomposition $(T_n, \mathcal{B}_n)$. If t is a leaf, then t is also called the START node. A node with two sons (who share bags with their parent) is called a JOIN node. If a node t has exactly one son then it is either a FORGET node or an INTRODUCE node, depending whether its bag has one vertex less or one vertex more, respectively, than its son's.

8.5 Computing a tree decomposition

A natural question arises. Why one needs a tree decomposition and how could one compute one. We shall answer the first question in the next subsection, and try to give a satisfying answer for the second one right here right now.

Unfortunately, computing tree-width (or computing an optimal tree decomposition) is a difficult problem.

Theorem 8.11 (Arnborg, Corneil, Proskurowski) *Deciding whether* $\mathrm{tw}(G) \le k$ *is NP-complete.*

On the other hand let us decide that k is not part of the input, but rather a constant. Then, in theory, thighs brighten up.

Theorem 8.12 (Bodlaender) *Fix* k*. There exists an algorithm that either constructs a tree decomposition of* G *of width* $\le k$ *or correctly decides that* $\mathrm{tw}(G) > k$ *in time* $O(n)$*.*

However, the linear algorithm of Bodlaender is not a practical one. The constants hidden in $O(n)$ notation a huge and depend heavily on k. We shall see in the following section how one can efficiently compute a tree decomposition of G of small width, provided that $\mathrm{tw}(G)$ is small.

8.6 Dynamic programming on a tree decomposition

Let G be a graph. A 3-(vertex)-coloring is a mapping

$$c : V(G) \to \{1, 2, 3\},$$

so that for every edge uv we have $c(u) \neq c(v)$. We know that not every graph admits a 3-coloring, and in some cases it is easy to show so. Yet in the general case deciding whether G admits a 3-coloring, equivalently whether its chromatic number $\chi(G)$ is at most 3, is a difficult problem. More precisely, the problem

3-COLORING
input: Graph G.
output: Is $\chi(G) \le 3$?

is NP-complete. Constructing a 3-coloring, if one exists, an optimization version of the abovementioned decision problem, is a NP-hard problem. It may come as a surprise that, given a tree decomposition $(T, \mathcal{B})$ of G of width w we will be able to either construct a 3-coloring or show that $\chi(G) \ge 4$ in $O(f(w)n)$ time. If we only consider graphs of bounded tree-width we can compute an existing 3-coloring in *linear* time.

Assume that we are given a nice tree decomposition $(T, \mathcal{B})$ of G whose width is w. Now each bag contains at most $w + 1$ vertices. If $t \in V(T)$ then let G_t be the graph induced by bags of descendants of t (including t itself). We shall name G_t the *lower graph* or *t-lower graph* if we really want to be precise. Note that B_t separates the lower graph from $V(G - G_t)$.

Let $t \in V(T)$ and $S \subseteq B_t$. A *characteristic* is a pair (γ, δ) where

$$\gamma : B_t \to \{1, 2, 3\}$$

and

$$\delta_S = \begin{cases} 1, \text{ if mapping } \gamma \text{ extends to a 3-coloring of } G_t, \\ 0, \text{otherwise.} \end{cases}$$

A *list of characteristics* (at a node t) is a collection of characteristics for every possible mapping of B_t into $\{1, 2, 3\}$. We can think of a characteristic as a *fingerprint* of a restricted (its intersection with the bag is prescribed) 3-coloring of the lower graph or a piece of information that a mapping of B_t cannot be extended to a 3-coloring.

We shall in a bottom-up approach construct lists of characteristics at every node t. Whether $\chi(G) \leq 3$ or not can be easily retrieved from list of characteristic at the root, as $G_r = G$.

What we shall need is a collection of four algorithms for each of the node types explaining how to compute lists of characteristics at node t, provided we know the characteristics of its sons.

START: Let us first observe a start node t, and let v be the sole vertex of B_t. As G_t is an edgeless graph, every characteristic describing a possible choice of color for v (as a mapping of B_t into $\{1, 2, 3\}$) has its second coordinate equal to 1.

JOIN: Assume that t_1 and t_2 are sons of t, and $B_t = B_{t_1} = B_{t_2}$. Let (γ, δ_1) and (γ, δ_2) be characteristics at t_1 and t_2, respectively. As B_t separates G_{t_1} and G_{t_2} the coloring of B_t extends to G_t if and only if it extends to both G_{t_1} and G_{t_2}. Hence the corresponding characteristic at t is equal to $(\gamma, \delta_1 \cdot \delta_2)$.

FORGET: Let t' be a son of t and let $v \in B_{t'} \setminus B_t$. Choose a mapping $\gamma : B_t \to \{1, 2, 3\}$, which extends three ways to a mapping $\gamma'_i : B_{t'} \to \{1, 2, 3\}$, $i = 1, 2, 3$, by setting $\gamma'_i(v) = i$. Let $(\gamma'_1, \delta'_1), (\gamma'_2, \delta'_2), (\gamma'_3, \delta'_3)$ be the three characteristics at t'. Now we compute a characteristic at t by $(\gamma, \max\{\delta'_1, \delta'_2, \delta'_3\}$.

INTRODUCE: Assume now that $v \in B_t \setminus B_{t'}$, where t' is the only son of t. A characteristic (γ', δ') be a characteristic at t', and let $\gamma_1, \gamma_2, \gamma_3$ be extensions of γ' to B_t, where $\gamma_i(v) = i$, for $i \in \{1, 2, 3\}$. For every $i \in \{1, 2, 3\}$ let us set a characteristic (γ_i, δ_i) where $\delta_i = 1$ if and only if both $\delta' = 1$ and v has no neighbor in B_t of the same color.

How much time do the above algorithms require? The list of characteristics in a node t has length $\leq 3^{w+1}$, and computing the list of characteristics of t, given characteristics of its sons, takes $O(q(w)3^w)$ time where $q(w)$ is a low degree polynomial. As a nice tree decomposition of a n-vertex graph G has $O(wn)$ nodes we can compute lists of characteristics of every node of T in time $O(p(w)3^w n)$ for some polynomial p. Consequently we can in this time decide whether G admits a 3-coloring or not.

If yes, constructing an actual 3-coloring requires some additional bookkeeping. Let us for every characteristic (γ, δ) at a nonstart node t for which $\delta = 1$ remember its predecessor characteristics (γ', δ') for every son t' of t – the one which enables us to extend the 3-coloring of $B_{t'}$ to B_t. Hence by tracnig predecessors we can in a top down manner transform a yes answer of a decision problem to an actual solution of the optimization problem.

We can compile the above analysis into a pair of meta-theorems describing dynamic programming on a tree decomposition.

Dynamic programming on a tree decomposition can be roughly described with the following recipe:

(1) Construct suitable characteristics for a problem.

(2) Prove that the number of characteristics is bounded if the width of the decomposition is bounded.

(3) Construct four algorithms (of constant time complexity — the constant relies on the width, though) for each type (START, INTRODUCE, FORGET, JOIN) of node t:

  input: B_t, the set of introduced/forgotten vertices, families of characteristics of sons

  output: characteristics of node t

(4) Prove *correctness* and *completeness.*

Theorem 8.13 *Let P be a decision problem for which (1)–(4) hold. Then there exists a linear time algorithm for P.*

In order to be able to solve the optimization version of P one additional ingredient is needed:

(5) four polynomial algorithms for each type (START, INTRODUCE, FORGET, JOIN) of node t:

  input B_t, the set of introduced/forgotten vertices, families of characteristics of sons, for every characteristic C_t and for every son t' a pair $(C_{t'}, S_{t'})$ which induces C_t

  output: partial solution S_t with characteristic C_t, which for each son t' restricts to $S_{t'}$ on (lower graph) $G_{t'}$

Theorem 8.14 *Let P be an optimization problem for which (1)–(5) hold. Then there exists a polynomial algorithm for P.*

If the decision variant of P can be solved in time $O(n)$ and algorithms for (5) run in constant time then solving P takes $O(n)$.

9 Tree decomposition lower bounds

Let G be a graph. If its tree-width is small, then dynamic programming on a tree decomposition enables us to solve hard problems efficiently. We can in *linear-time* compute its minimal-vertex cover, a possible 3-coloring or a hamilton cycle, or a maximal independent set. These problems are generally NP-hard. There are two caveats. On one hand, there is no easy way to compute a tree-decomposition of optimal width.

The other possible problem is that a graph might have large-tree width in the first place. The width of a tree-decomposition is an upper bound to tree-width, and is a decomposition of small width is therefore a certificate of small tree-width of a graph. The quest for the structure implying that tree-with is large is our next task.

9.1 Brambles

Let G be a graph. A family of vertex-subsets $\mathcal{M} = \{M_1, M_2, \ldots, M_k\}$ is a *bramble* if:

(M1) $M_i \subseteq V(G)$ for every $i \in \{1, \ldots, k\}$,

(M2) M_i induces a connected graph, $G[M_i]$ is connected, for every $i \in \{1, \ldots, k\}$,

(M3) for every pair $i, j \in \{1, \ldots, k\}$ the sets M_i and M_j *touch*, which means that they either intersect, $M_i \cap M_j \neq \emptyset$, or alternatively there exists vertices $v_i \in M_i$ and $v_j \in M_j$, so that v_i and v_j are adjacent.

We shall call M_i a *block* of a bramble $\mathcal{M}$.

A subset $U \subseteq V(G)$ is a *cover* (sometimes also called a *hitting set*) of a bramble $\mathcal{M} = \{M_1, M_2, \ldots, M_k\}$, if U intersects every bag of $\mathcal{M}$, $U \cap M_i \neq \emptyset$ for every $i \in \{1, \ldots, k\}$.

The *order* of a bramble is the minimal size of its cover.

The tree-width duality theorem connects the concept of a tree-decomposition with that of a bramble.

Theorem 9.1 (tree-width duality) *Let $k \geq 0$. Then $\mathrm{tw}(G) \geq k$ if and only if G admits a bramble of order $> k$.*

Equivalently formulated,

for every G and integer k exactly one of below holds

- $\mathrm{tw}(G) < k$
- *G admits a bramble of order $> k$.*

We shall technically omit the proof of the above theorem, yet will in the next section describe a close connection with a combinatorial game of cops and robbers in a graph.

Fixing a graph, we are looking for a bramble with maximal possible order. Clearly the order of a bramble is bounded from above with the number of bags (and also the number of vertices of a graph), so as an approximation a bramble with as many bags as possible in a legitimate intermediate goal.

Let us take a look at a planar quadrangular $n \times n$ grid $P_n \Box P_n$. A *cross* is a union of a *row* and a *column*. There are exactly n^2 different crosses, and they do form a bramble: (M1) is clear, as a cross induces a graph which is a union of two paths sharing a common vertex we also have (M2). Now given a pair of crosses C and C', a *row* of the former intersects the *column* of the other, hence also (M3).

If U contains at most $n-1$ vertices, then U misses a both a row and a column. Their union is a cross disjoint from U. Hence the size of a minimal cover is at least n. On the other hand the set of n-vertices from, say, the first column intersect every cross, and hence forms a cover. By Theorem 9.1 we infer $\mathrm{tw}(P_n \Box P_n) \geq n-1$.

We have to play it a bit more careful to get the exact bound.

Proposition 9.2 $\mathrm{tw}(P_n \Box P_n) = n$.

Proof. In the previous section we have constructed a tree-decomposition of $P_n \Box P_n$ of width n (with bags being "broken rows" with exactly $n+1$ vertices). By Theorem 9.1 it is enough to construct a bramble of order $n+1$.

Let us label vertices of $V(P_n \Box P_n) = \{(i,j) \mid 1 \leq i,j \leq n\}$. A *short cross* $C_{k,\ell}$, $k, \ell \in \{1, \dots, n-1\}$ is the set $\{(k,i) \mid 1 \leq i \leq n-1\} \cup \{(j,\ell) \mid 1 \leq j \leq n-1\}$. Intuitively a short cross lacks a vertex from the bottom row and the right column. Let $R = \{(i,n) \mid 1 \leq i \leq n-1\}$ and $B = \{(n,j) \mid 1 \leq j \leq n\}$; R denotes the rightmost column (without the bottom vertex), and B is the bottom row.

Let us construct a bramble $\mathcal{M} = \{C_{k,\ell} \mid 1 \leq k, \ell \leq n-1\} \cup \{R, B\}$. The above discussion indicates that $\mathcal{M}$ satisfies (M2), and also (M1). In order to show (M3), the short crosses clearly intersect, we need to observe that every short cross has a vertex in both the penultimate row and the second-to-right column.

Now a cover U of $\mathcal{M}$ needs at least $n-1$ vertices to cover all the short crosses, and an additional pair of vertices to cover the disjoint sets R and B. Hence the order of $\mathcal{M}$ is at least $n+1$ (exactly $n+1$, indeed). $\Box$

9.2 Cops and robbers

A lack of the proof of Theorem9.1 still leaves us with a bitter aftertaste. So let us play a different game. We shall exhibit a connection with a combinatorial *cops-and-robbers game* (game? again) that will serve our purpose.

Searching an *sweeping* in graphs are a pair of related concepts, where either a set of agents would like to find an intruder hiding in a graph, or alternatively would like to purge a graph of a disease that spreads along edges. The dynamic of the agents, intruder, and the spread of disease may vary in different problems, but the underlying graph structure serves as the basic movement frame.

Let us describe the rules of the *cops-and-robbers* game that will encompass both tree-decompositions and brambles. Cops-and-robbers is a 2-player game, the first player $P1$ controls the squad of k cops, the second player $P2$ controls the movement of a single robber.

(game start) At the start of the game player $P1$ distributes cops in vertices of G. Several cops may occupy the same vertex in G, and also a cop may be temporarily removed from the game. However, the objective to catch the robber will push player P1 to distribute k cops on k different vertices in G. Next, player P2 chooses robber's position in one of the cop-free vertices.

(cop's move) A single move of P1 consists of flying a single cop c from one vertex to another. The movement of a cop in not constrained to the underlying graph. However we can think of cops flying around in helicopters — these machines cannot land very fast, and a robber observing an approaching cop can escape before touch down.

(robber's move) A robber moves around on a speedy motorcycle traveling at great speed, yet a robber is bound to stay at vertices of G. If C is the subset of $V(G)$ containing the cops, and the robber stays at $v \notin C$, then the robber may choose to move to any vertex of the component $G - C$ containing v.

(objective) Cops would like to catch the robber, but given that the robber is in v they can only do so (as a robber can escape a landing cop) by first occupying every neighbor of v, and in the last move landing on the top of robber's head. The robber would like to escape the cops indefinitely long.

Proposition 9.3 *Let G be a graph. If $\mathrm{tw}(G) < k$ then k cops have a winning strategy of catching a robber.*

Proof. Let $(T, \mathcal{B})$ be a tree decomposition of G of width $< k$, the bags of $\mathcal{B}$ have size $\leq k$. Let us assume that $\mathcal{B}$ satisfies the property that (i) $B_t \neq B_{t'}$ for every pair of nodes $t, t' \in V(T)$, and (ii) for every edge $tt' \in E(T)$ we have either $B_t \subseteq B_{t'}$ or $B_{t'} \subseteq B_t$.

A decomposition satisfying these properties may be constructed by first taking a decomposition with incomparable bags, and second subdividing every edge of T and assigning the intersection bag at each new node.

Let us put the whole lot of cops in a single bag B_t of maximal order. Now B_t is a separator in G, and there exists a component R of $G - B_t$, so that the robber is in R. Let T' be the subtree of T so that $R \subseteq \bigcup_{\tau \in V(T')} B_\tau$, and let t' be a neighbor of t in T'. We might say that the robber hides in T'.

Let us "move" the cops from B_t to $B_{t'}$: now if $B_{t'} \subseteq B_t$ then no additional helicopter flights are needed, the cops are already in $B_{t'}$. If, on the other hand, $B_t \subseteq B_{t'}$, then at least $|B_{t'} \setminus B_t|$ cops are free to fly to B'_t so that at all times all vertices of B_t are filled with cops.

Now this move results in the robber hiding in a smaller tree. In a finite number of steps the robber will have no place to hide, and this will lead to its capture. $\square$

Proposition 9.4 *Let G be a graph. If G admits a bramble of order $> k$, then a robber has a winning strategy agains k cops.*

Proof. Let $\mathcal{M} = \{M_1, M_2, \ldots, M_\ell\}$ be a bramble of order $> k$. This implies that no set of $\leq k$ vertices cover $\mathcal{M}$. Let C be the set of vertices occupied with cops. The robber r follows the strategy of *staying in a bag of $\mathcal{M}$ which is not covered with C.*

Assume the robber resides in M_j and let player P1 choose to land a cop c in a vertex of M_j. As the number of cops is strictly smaller than the order of $\mathcal{M}$ some other block $M_{j'}$ will not be covered with a cop after c lands. So the robber can immediately prior to c's landing escape to $M_{j'}$ — this is possible as M_j and $M_{j'}$ touch (M3) and as M_j induces a connected graph (M2). $\square$

Now clearly a simultaneous winning strategy for both the cops and the robber is out of the question. This implies that no graph admits both a tree decomposition of width $< k$ and a bramble of order $> k$. The other direction will be left unsettled (how does a winning strategy for, say, cops imply a tree decomposition of width $< k$).

10 Matchings

A *matching* in a graph G is a set of edges $M \subseteq E(G)$, so that no two edges $e, e' \in M$ share an endvertex. The trivial examples of matchings include both the *empty matching* $\emptyset$, and if e is an arbitrary edge of G, the *singleton matching* $\{e\}$. We shall be interested in the other end of the spectrum. We would like our matching to have as many edges as possible.

Given a matching M, we call a vertex v *M-matched* (or just *matched* if the matching is obvious from the context) if v is an endvertex of an edge from M, otherwise v is called *M-free* (or just *free*). A matching M is *inclusion-wise maximal* if no pair of adjacent vertices are both M-free, equivalently if $G - V(M)$ is edgeless, where $V(M)$ denotes the set of endvertices of edges from M.

Greedy approach is a possible strategy to construct a large matching: pick an arbitrary edge $e_1 = u_1v_1$, put it in the bag of a greedily growing matching, and iterate the process on $G - u_1 - v_1$. We finish with a matching M, and the remainder of the original graph is a collection of isolated vertices, an edgeless graph.

An inclusion-wise maximal matching can be constructed using the above mentioned greedy approach.

Constructing the matching greedily might not finish in a matching with the maximal possible number of edges — a matching M is *maximal* (in G) if no matching M' satisfies $|M'| > |M|$. Clearly a maximal matching is inclusion-wise maximal, yet the converse might not hold. Take a path P of length 3. Its edge set $E(P)$ decomposes in a pair of inclusion-wise maximal matchings, the one containing the middle edge and the other containing the two end edges. The latter matching is also a maximal one, and the former consequently is not.

Later on we shall find a criterion proving a matching is maximal. At this point let us opt for an easier optimality property — a matching M is *perfect* if every vertex of G is M-matched. Clearly a perfect matching M satisfies $|M| = |V(G)|/2$ and as already mentioned a perfect matching is a maximal one.

Yet a perfect matching may be sour grapes — not every graph may admit a perfect matching. The easiest examples include cycles of odd length and paths of even length.

10.1 Matching and vertex-covers

A *vertex-cover* is a set of vertices $U \subseteq V(G)$ so that every edge has an endvertex (or both) in U. The vertex-set $V(G)$ itself is a vertex-cover, and omitting a single vertex v we still are left with a vertex-cover $V(G - v)$. We are however interested in vertex-covers of small order.

There is an easy relation comparing orders of a matching and that of a vertex-cover.

Proposition 10.1 *Let $G = (V, E)$ be a graph and choose an arbitrary vertex-cover*

$U \subseteq V(G)$ and an arbitrary matching $M \subseteq E(G)$. Then

$$|M| \leq |U| \tag{10.1}$$

Proof. Choose an edge $uv \in M$. A vertex-cover U must contain at least one of u, v. Consequently $|U| \geq |M|$, as at least one of the endvertices of every edge from M (recall, they do not share endvertices) has to be included in U. □

The following is an easy consequence of the above proposition. We can, by optimizing both sides of (10.1), infer:

Corollary 10.2 *Let $G = (V, E)$ be a graph. If $U^* \subseteq V(G)$ is a minimal vertex-cover and $M^* \subseteq E(G)$ is a maximal matching then*

$$|M^*| \leq |U^*|$$

Let G be a graph, and assume that by some chance we find a pair, a matching M and a vertex-cover U having the same order $|M| = |U|$. Then we immediately know that both are optimal, M is a maximal matching and also U is a minimal vertex-cover.

The converse might not hold. In an odd cycle C_{2k+1} the maximal matching has size k, and the minimal vertex-cover has order $k + 1$.

We know that searching for a minimal vertex-cover on the class of all graphs is a NP-hard problem. We shall learn in the sequel that the quest for maximal matchings is a much easier one.

10.2 Augmenting paths

Let G be a graph and let M be a matching. Let $P = v_0v_1v_2 \ldots v_{k-1}v_k$ be a path in G, and let us denote the edges of P with $e_1, e_2, \ldots, e_k$, so that $e_i = v_{i-1}v_i$. We say that P is an *alternating path* if every other edge along P is in M, in other words either $M \cap E(P) = \{e_1, e_3, e_5, \ldots\}$ or $M \cap E(P) = \{e_2, e_4, e_6, \ldots\}$.

An alternating path P is an *M-augmenting path* (or just *augmenting path* if there is no doubt on the matching in question) if its endvertices v_0 and v_k are both M-free. Clearly every augmenting path is of odd length.

Given an M-augmenting path P we can *augment the matching M along P*, by exchanging the edges along P with respect to the matching by setting

$$M' = M + E(P)$$

where $+$ denotes the symmetric difference of sets. In other words we remove the edges of P previously in M, and insert those edges of P that were previously not contained in M. As $|M'| = |M| + 1$ we have *augmented* the matching in the process.

The existence of an M-augmenting path proves that M is not a maximal matching. The implication in the other direction is valid as well.

Theorem 10.3 *Let G be a graph and M a matching. Then G is a maximal matching if and only if there exists no M-augmenting paths in G.*

Proof. ($\Rightarrow$) is clear.
For ($\Leftarrow$) it is enough to see the following. If M and M' are matchings and $|M| < |M'|$, then there exists an M-augmenting path in G. Let us imagine that edges in M are given green color, and edges in M' receive red color. Let us observe $G' = G[M \cup M']$ the spanning subgraph of G spanned by colored edges. Now every vertex of G' has degree at most 2 and connected components of G' are either (i) isolated vertices, (ii) cycles, and (iii) paths.

An edge e of G' may receive both colors, red and green, let us call such an edge a 2-cycle (we may think of both red and green instance of e as a pair of parallel edges). Next a (longer) cycle C in G' is necessarily of even length, as edges of each color alternate along C.

Let P be a path component of G' and let v be an endvertex of P. By maximality of P (a component is a maximal connected subgraph) if v is incident with an edge from M then it is also M'-free, and vice versa.

Globally, as $|M'| > |M|$, there exists a component K of G, so that $|E(K) \cap M'| > |E(K) \cap M|$. Now K is neither a 2-cycle, neither a (proper) cycle of greater length, nor a path of even length. All these types of possible subgraphs are balanced with respect to the number of green and red edges. Hence K is a path of odd length, and both its endvertices u and v are incident with edges from M'. Hence both u and v are M-free.

This makes K an M-augmenting path in G. □

Now Theorem 10.3 provides us with the template approach for computing a maximal matching.

```
MaximalMatching($G$)
1 $M = \emptyset$;
2 while $G$ contains an $M$-augmenting path $P$ do
3     $M = M + E(P)$
4 return $M$
```

Algorithm 10.1: MaximalMatching algorithm template.

The only *problem* is to find an augmenting path.

We shall finish the section with a technical lemma. Its proof is straightforward but we will use the exact wording in the last subsection.

Lemma 10.4 *Let G be a graph, and let M_1 and M_2 be matchings of the same size. Then G contains an M_1-augmenting path if and only if G contains an M_2-augmenting path.*

Proof. As M_1 and M_2 are of the same size they are either both maximal or none of them is maximal. In the former case no path can be augmenting, and in the latter case both M_1- and M_2-augmenting paths exist. This follows from Theorem 10.3.

$\square$

10.3 Matchings in bipartite graphs

We shall devote this section to the problem of finding maximal matchings in bipartite graphs. Why limiting ourselves on bipartite graphs, one might ask.

Often enough we study problems on objects of different types, for example relations between teachers and students, trains and train stations, points and lines, to name a few. In these cases the graphs representing relations between objects of different types by definition turn out to be bipartite.

König's theorem ties matchings and vertex-covers in bipartite graphs. If computing minimal vertex-covers is NP-hard, restricted to bipartite graphs it becomes a polynomially solvable problem.

Theorem 10.5 *Let $G = (V, E)$ be a bipartite graph. If $U^* \subseteq V(G)$ is a* minimal *vertex-cover and $M^* \subseteq E(G)$ is a* maximal *matching then*

$$|M^*| = |U^*|$$

We shall postpone the proof of the above theorem.

Let G be a bipartite graph. We shall construct a related weighted graph G^+, so that solving the max-flow-min-cut problem on G^+ will enable us to extract both a maximal matching an a minimal vertex-cover.

Let $A \cup B = V(G)$ be the bipartition of vertices of G. The auxiliary weighted graph G^+ is constructed following these steps:

- add a pair of new vertices s and t, $V(G^+) = V(G) \cup \{s, t\}$,
- for every vertex $a \in A$ add an edge sa, orient it away from s, and set its weight $w(sa) = 1$,
- for every vertex $b \in B$ add an edge bt, orient it towards t, and set its weight $w(bt) = 1$,
- orient every original edge from A towards B, and set its weight to 2.

Proposition 10.6 *Let f be an $s - t$-flow in G^+.*

(a) If f is integral *then the set of $A - B$ edges $M_f = \{e \mid f(e) = 1\}$ is a matching in G.*

(b) *If f is also maximal, then M_f is a maximal matching. The set of vertices $\{v \in A \cup B \mid v$ is incident with an edge e so that $f(e) = w(e) = 1\}$ is a minimal vertex-cut.*

Proof. Kirchhoff's conditions imply that $f(e) \leq 1$ for every edge e. Hence f can be interpreted as a collection of $|f|$ edge-disjoint $s-t$-paths $\mathcal{P}_f$. No two paths from $\mathcal{P}_f$ can share a vertex in either A or B, as the inflow into an arbitrary vertex $a \in A$ is bounded from above by 1, and so is the outflow from every $b \in B$. Hence the paths from $\mathcal{P}_f$ are internally disjoint. This implies that the $A-B$ edges from paths in $\mathcal{P}_f$ form a matching.

Let f be a maximal flow, and $E(U, U')$ a minimal cut. By above arguments $E(U, U')$ contains no $A-B$ edges. By Ford-Fulkerson theorem (Theorem 2.6) $|f| = w(E(U, U'))$. On one hand the $A-B$ edges from paths of $\mathcal{P}_f$ form a matching of size $|f|$.

On the other hand let $e \in E(U, U')$. As e is not a $A-B$ edge e has exactly on endvertex $v_e \in A \cup B$. We claim that $S = \{v_e \mid e \in E(U, U')\}$ is a vertex-cover of order $w(E(U, U'))$ in G. If $ab \in E(G)$ has no endvertex in S, then $f(sa) = 0$, $f(bt) = 0$, and consequently also $f(ab) = 0$, contradicting the optimality of f. This completes the proof. □

We have thus reduced the problem of finding optimal matchings and vertex-covers to computing maximal flows and minimal cuts. And, duality between cuts and flows also implies Theorem 10.5.

Let us finish the section with the classical Hall's theorem. Let G be a bipartite graph, and let $A \cup B = V(G)$ be the bipartition of its vertices. In several applications we only need to find a matching M in which every vertex $a \in A$ is M-matched, and not worry about vertices from B.

An obvious obstruction to such a matching is a subset $S \subseteq A$ having too few neighbors in B; $|N(S)| < |S|$. It is however surprising that this is the only possible obstruction.

Theorem 10.7 (Hall) *Let G be a bipartite graph and let $A \cup B = V(G)$ be the bipartition of its vertices. Then G contains a matching M so that every $a \in A$ is M-matched if and only if*

$$\forall S \subseteq A : |S| \leq |N(S)| \tag{10.2}$$

Proof. ($\Rightarrow$) is clear.
Now ($\Leftarrow$) is an immediate consequence of Theorem 10.5. Let U be an optimal vertex cover of G. We only need to consider the case $|U| < |A|$. In this case let $S = A \setminus U$. Clearly $S \neq \emptyset$ and let us assume that $|S| \leq |N(S)|$. As U is a vertex cover $N(S) \subseteq U$. Hence $|U| \geq |A \cap U| + |N(S)| \geq |A \cap U| + |S| = |A \cap U| + |A \setminus U| = |A|$. This is a contradiction finishing the proof. □

Corollary 10.8 *Let G be a regular bipartite graph. Then G contains a perfect matching.*

Proof. Let $V(G) = A \cup B$, and let d be the degree of a (and hence of every) vertex of G. By double counting of edges we infer that $d|A| = d|B|$ implying that G is balanced, $|A| = |B|$. If M is a matching so that every vertex of A will be M-matched, then also every vertex $b \in B$ is M-matched, as $|A| = |B|$.

By Theorem 10.7 it is enough to establish the Hall condition (10.2). Pick a subset $S \subseteq A$, and assume that $|N(S)| < |S|$. Let F be the set of edges incident with vertices of $N(S)$. Clearly $|F| = d|N(S)|$, and as every edge emanating from S has an endvertex in $N(S)$ also $|F| \geq d|S|$. Now this implies $|N(S)| \geq |S|$ establishing Hall's condition and finishing the proof. □

A *k-factor* in G is a spanning k-regular subgraph. A 1-factor is therefore exactly a perfect matching. A *1-factorization* of G is a partition of $E(G)$ into perfect matchings. Clearly every graph admitting a 1-factorization must be a regular one.

By repeatedly applying Corollary 10.8 in a d-regular bipartite graph G we infer:

Corollary 10.9 *Let G be a d-regular bipartite graph. Then G admits a 1-factorization with d blocks, i.e. the edges of G can be partitioned into d perfect matchings.*

10.4 Matchings in general graphs

We shall devote the last section to describe an algorithm for computing maximal matchings in general graphs. According to Algorithm template 10.1 we shall only need to find an M-augmenting path, or give a certificate that no M-augmenting paths exist.

Let G be a graph and M a matching. A *M-blossom* (or just a *blossom*) is an odd cycle C satisfying the following properties:

(B1) there exists exactly one vertex $v \in V(C)$ so that v is not incident with an edge in $E(C) \cap M$, implying that if C has length $2k+1$ then $|E(C) \cap M| = k$, and

(B2) v is an endvertex of an alternating path P of even length whose other endvertex v' is M-free.

We shall call P and v the *stem* and the *base* of a blossom C, respectively. Observe that if v is M-free then the stem of C has length 0.

Let C be an M-blossom in G. *Contracting a blossom C* results in a graph G' obtained by contracting vertices of C into a single vertex v_C. Now in G' the new vertex v_C is adjacent to every vertex of $G - V(C)$ which has a neighbor in C. The corresponding matching equals $M' = M \setminus E(C)$.

The following theorem is a cornerstone observation.

Theorem 10.10 *Let C be an M-blossom in G, and let G' be a graph and M' a matching obtained by contracting C, respectively. Then G contains an M-augmenting path if and only if G' contains an M'-augmenting path.*

Proof. Let C be an M-blossom with base v, and let us first assume that v is M-free. Let G' be the graph obtained by contracting the blossom C, with M' the corresponding matching. If P is an M-augmenting path disjoint from C then P is also an M'-augmenting path in G. Otherwise P intersects C, yet P has an endvertex $u \notin V(C)$, as C contains exactly one M-free vertex. Let P_u be the initial segment of P starting at u and just touching C. As v_C is an M'-free vertex P_u is an M'-augmenting path in G'.

If v is not M-free and P_v is a nontrivial stem of C then let $M_1 = M + E(P_v)$. Now C is both an M-blossom and also an M_1-blossom. Similarly, contracting C produces a pair of matchings M' and M_1' in these respective cases. Assume that G admits an M-augmenting path. By Lemma 10.4 G admits an M'-augmenting path, and by above argument also G' admits an M_1'-augmenting path. As M_1' and M' are of the same size there exists an M'-augmenting path as well.

For the converse implication let us first choose an M'-augmenting path P' in G'. If $v_C \notin V(P')$ then P' is also an M-augmenting path in G. Now if P' contains v_C, then let us split P' into internally disjoint subpaths P_1' and P_2' sharing v_C as their common endvertex. Let P_1 and P_2 be their lifts in G — paths in G spanned by $E(P_1')$ and $E(P_2')$ — and let v_1 and v_2 be their endvertices on C, respectively. There is exactly one $v_1 - v_2$-path P^* along C of *even* length (generally v_1 and v_2 split C into a pair of paths of lengths of different parities, but if $v_1 = v_2$ then we can take as P^* the trivial $v_1 - v_2$-path). Now the concatenation of P_1, P^*, and P_2 is an M-augmenting path in G. □

In the sequel we shall exhibit a recursive algorithm `FindAugmentingPath` (Algorithm 10.2) for computing an augmenting path in G with respect to a matching M. We may be lucky an find an M-augmenting path by applying a search procedure in G. Yet a search may fail to find an M-augmenting path even if one exists. In this case we will settle for a blossom C, whose contraction produces G' and the corresponding matching M', and will try to find an M'-augmenting path in G' recursively. By Theorem 10.10 an M'-augmenting path P' in G' can be lifted to an M-augmenting path P in G. We shall call P the *lift* of P'.

Let us comment on the data structures needed in the run of Algorithm 10.2. We shall iteratively grow a forest F, so that each tree of F is rooted in an M-free vertex. In the run a vertex or an edge x may become flagged, by setting *Flag(x):=True*. For every discovered vertex v we shall also note the subtree of F containing v by setting the corresponding entry in the *Root* table. Further we shall note the parity of the

$v - \mathit{Root}(v)$-path in F.

```
FindAugmentingPath($G, M$)
foreach $x \in V(G) \cup E(G)$ do
    $\mathit{Flag}(x)$ :=$\mathit{False}$
$F$ = forest all of $M$-free vertices of $G$;
foreach $v \in V(F)$ do
    $\mathit{Parity}(v)$ :=$\mathit{Even}$;
    $\mathit{Root}(v) := v$
foreach $v \in V(F)$ satisfying $\mathit{Flag}(v)$ =$\mathit{False}$ and $\mathit{Parity}(v)$ =$\mathit{Even}$ do
    foreach edge $vw \in E(G) \setminus M$ satisfying $\mathit{Flag}(vw)$ =$\mathit{False}$ do
        case $\mathit{Flag}(w)$ =$\mathit{False}$
            $\mathit{Parity}(w)$ :=$\mathit{Odd}$;
            $\mathit{Flag}(w)$ :=$\mathit{True}$;
            let $z$ be the only vertex so that $wz \in M$;
            $\mathit{Parity}(z)$ :=$\mathit{Even}$;
            $F := F + w + z + vw + wz$
        case $\mathit{Flag}(w)$ = $\mathit{True}$ and $\mathit{Parity}(w)$=$\mathit{Odd}$
            ignore this case;
        case $\mathit{Flag}(w)$ = $\mathit{True}$ and $\mathit{Parity}(w)$=$\mathit{Even}$ and $\mathit{Root}(w) \neq \mathit{Root}(v)$
            return ($\mathit{Root}(v)$-$v$-$w$-$\mathit{Root}(w)$ path)
        case $\mathit{Flag}(w)$ = $\mathit{True}$ and $\mathit{Parity}(w)$=$\mathit{Even}$ and $\mathit{Root}(w) = \mathit{Root}(v)$
            $C$ := cycle in $F + vw$;
            $(G', M')$=ContractBlossom($C$);
            $P'$ :=FindAugmentingPath ($G', M'$);
            return $\mathit{lift}(P')$
        Flag($vu$):=True;
    Flag($v$):=True;
return ($\emptyset$)
```

Algorithm 10.2: FindAugmentingPath finds a possible augmenting path with respect to G and M.

Theorem 10.11 *Let G be a graph, and M a matching. Then the call* `FindAugmentingPath`(G, M) *in Algorithm 10.2 finds an M-augmenting path in G or correctly reports that no M-augmenting path exists.*

Moreover, the running time of `FindAugmentingPath`(G, M) *is $O(n^3)$, where n denotes the number of vertices in G.*

Proof. Let us first prove the former statement. We proceed by induction on the number of edges in G. The basis $E(G) = \emptyset$ trivially holds.

Let G be a graph, M its matching, and let us inductively assume that `FindAugmentingPath` produces results as stated for every input graph (and its matching) on fewer vertices. Assume first that M is a maximal matching, i.e. there exists no M-augmenting paths in G. There are two possibilities for a call

`FindAugmentingPath`(G, M) to return a path. Either the algorithm outputs $\emptyset$ (an indication that there exists no M-augmenting path) on Line 26 or outputs a *lift(P')* on Line 23, where P' is a path returned by the call `FindAugmentingPath`(G', M'). As the contracted graph G' contains no M'-augmenting path by Theorem 10.10, the inductive hypothesis implies $P' = \emptyset$. Hence also *lift*$(P') = \emptyset$.

Assume next that there exists an M-augmenting path $P = v_0 v_1 v_2 \dots v_k$ in G. In order to achieve a contradiction let us assume that the call `FindAugmentingPath`(G, M) returns $\emptyset$. Inductively we may assume that the primary routine `FindAugmentingPath`(G, M) does not recursively call `FindAugmentingPath`(G', M'), and finishes by returning $\emptyset$ on line 26. This implies that every vertex v having even parity is ultimately flagged.

Let us inductively prove that for every vertex v_i of P its parity equals the parity of i. This clearly is true for $i = 0$. Now consider an edge $v_i v_{i+1}$ of P. If $v_i v_{i+1} \notin M$ then v_i and v_{i+1} have different parities, as inductively v_i is of even parity and as otherwise `FindAugmentingPath`(G, M) returns an augmenting path or finds a blossom. If $v_i v_{i+1} \in M$ then we may by induction assume that v_i is of odd parity. Now if parity of v_i was set before v_{i+1} then v_{i+1} is the only neighbor of v_i along an edge in M and immediately after Line 12 the parity of v_{i+1} is set to even. On the other hand if parity of v_{i+1} is odd then at the moment the parity of v_{i+1} was set the vertex v_i had not been determined its parity. That would make v_i of even parity which contradicts the inductive assumption.

Hence v_i and i are of the same parity along all P. This is clearly a contradiction as k is odd, and as v_k is M-free, its parity is even.

For the time complexity it is easy to see that a single call of `FindAugmentingPath`(G, M), excluding recursion, takes $O(m) = O(n^2)$ time, where m denotes the number of edges in G. As G' contains fewer edges, the total time complexity of `FindAugmentingPath`(G, M) including the recursive calls is bounded above by $O(\sum_{i=1}^{n} i^2) = O(n^3)$. □

Now let us finish with a couple of comments. Using the template Algorithm 10.1 we infer that we can compute maximal matchings in general graphs in time $O(n^4)$.

If the input graph G is bipartite then the case on line 19 never occurs. This implies that `FindAugmentingPath` can be used also for computing maximal matchings in bipartite graph. A single call of `FindAugmentingPath` takes $O(m)$ which implies that we can compute maximal matchings in bipartite graphs in time $O(nm)$.

Finally let us argue that looking for blossoms in indeed necessary. If M is not a maximal matching in G, then G contains an M-augmenting path, but we may not find it with the search approach employed in Algorithm 10.2. The catch lies in case on line 15. An edge vw is in this case ignored, but may itself be an edge of an M-augmenting path P, as P may use a number of not yet discovered vertices.