Algorithms and Complexity (LNMB), Lecture 8 by Jesper Nederlof, adapted by Tom van der Zanden, 30/10/2023

Treewidth

Treewidth is a measure of how 'tree-like' a graph is. This will appear to be useful for two reasons: 1. Very many graphs problems can be solved fast on graphs of small treewidth, 2. many graphs have small treewidth.

8.1 Weighted Independent Set on Trees Revisited

Let's recall what we finished with previous time. Suppose we are given a rooted tree $T = (V, E)$ and a weight function $\omega : V \to \mathbb{N}$. The goal is to find an independent set $X \subseteq V$ maximizing $\sum_{e \in X} \omega(e)$. Denote $ch(v)$ to be the set of children of v (which is the empty set if v is a leaf of T). Define $A[v]$ to be the maximum weight of an independent set of $T[v]$, then we have that

$$A[v] = \begin{cases} \omega(v), & \text{if } T[v] \text{ is a single node,} \quad (8.1) \\ \max \left\{ \sum_{c \in ch(v)} A[c], \omega(v) + \sum_{c_1 \in ch(v)} \sum_{c_2 \in ch(c_1)} A[c_2] \right\}, & \text{otherwise.} \quad (8.2) \end{cases}$$

To see this, note that if $T[v]$ is a single node, the maximum independent set is to include v. Otherwise, an independent set may not include v, in which case it will induce an independent set in $T[c]$ for every child c of v, or it will include v, in which case it may not include any child of v so it will induce a maximum independent set in $T[c_2]$ for all 'grand-children' of v (e.g., children of children). It is easy to see that a naïve evaluation of 8.1 takes only $O(n)$ time since there are at most $O(n)$ 'child' and 'grandchild' relations.

What allowed us to solve this problem so fast? The main point is that once we made a decision on whether to include v, the subproblems become independent since v separates all subtrees from each other. The notion of treewidth indicates how treelike the graph is and how good the above strategy is going to work.

8.2 Separators

Given a graph $G = (V, E)$, a subset of the vertices S *separates* A, B if $V = S \cup A \cup B$ and there are no edges between A, B. In the previous section we used separators of trees (which happen to be separable by individual vertices) to solve Independent Set in polynomial time. Separators can

be used in many context to obtain fast algorithms. Consider for instance the planar separator theorem:

Theorem 8.1. *Let G be a planar graph and let $n = |V|$. Then there exists a partition $V = S \cup A \cup B$ such that $|S| \leq 3\sqrt{n}$ and $|A|, |B| \leq 2/3n$.*

The above separator can be found in polynomial time. Such a separator is often called a *balanced* separator, since A and B split the graph into two more-or-less equal halves. $|S|$ is referred to as the *size* of the separator.

Using this separator property, we can solve Independent Set in planar graphs in $2^{O(\sqrt{n})}$.

Algorithm `planaris`$(G = (V, E))$, Assumes G planar
Output: The maximum weight of an independent set of G.
1: Find a separator (S, A, B) of G using the planar separator theorem
2: Set $max = -\infty$.
3: **for all** $X \subseteq S$ **do**
4: **if** X is an independent set of $G[S]$ **then**
For $G = (V, E)$ and $X \subseteq V$, $G[X]$ denotes the graph $(X, \{e \in E : e \subseteq X\})$.
$Nb[X]$ denotes the set of vertices adjacent to some vertex in X.
4: Set $weight = w(S) +$ `planaris`$(G[A \setminus Nb(S)]) +$ `planaris`$(G[B \setminus Nb(S)])$
5: **if** $weight > max$ **then** set $max = weight$
6: **return** max

Algorithm 1: An algorithm using planar separators for solving Independent Set.

The running time of this algorithm satisfies $T(n) \leq 2^{3\sqrt{n}}(T(2n/3) + T(2n/3))$. This solves to $O^*(\text{const}^{\sqrt{n}})$.

The best known running time for solving Independent Set in general graphs is $2^{O(n)}$ and this is the best possible under the Exponential Time Hypothesis. Interestingly, $2^{O(\sqrt{n})}$ is also the best possible running time for solving Independent Set in planar graphs under the Exponential Time Hypothesis.

8.3 Definition of Treewidth

The definition of treewidth is always a bit hard to digest so examples are coming, but let's first just get it over with:

Definition 8.1. *A* tree decomposition *of an (undirected) graph $G = (V, E)$ is a pair (X, T) where $X = (X_1 \ldots, X_\ell)$ with $X_i \subseteq V$ and T is a tree with vertex set X such that*

1. $\bigcup_{i=1}^{l} X_i = V$,
2. $E \subseteq \bigcup_{i=1}^{l} X_i \times X_i$ *(i.e. if $(u, v) \in E$, then both u and v are simultaneously in one bag X_i for some i),*
3. *if $v \in X_i$ and $v \in X_j$, then $v \in X_{j'}$ for all j' on the path from i to j in T (i.e. the set of X_i containing v form a connected subtree of T).*

Definition 8.2. *The* width *of a tree decomposition* (X, T) *where* $X = \{X_1, \ldots, X_l\}$ *is* $\max_{i=1}^{l} |X_i| - 1$. *The* treewidth *of a graph* G *is the minimum width over all tree decompositions of* G.

In the above definition of width it might look strange that we subtract 1; this is since it is nice to have that trees and forests have treewidth 1 (we'll see this in Exercise 8.1).

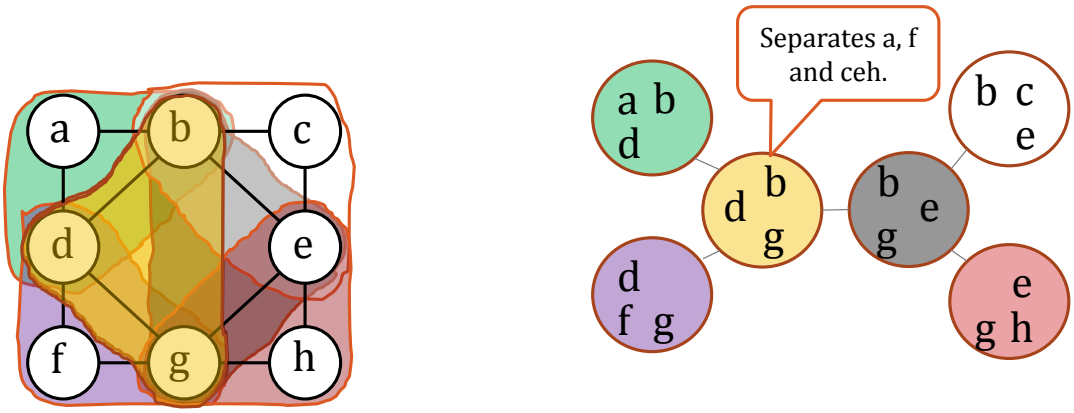

Figure 8.1: Example of a tree decomposition. The third property assures that for any bag X_i (in the example $X_i = \{b, d, g\}$), all vertices not in X_i occur in one of the connected components of the tree obtained by removing X_i from T. In the original graph, this means X_i is a separator.

It is important to know that the problem of determining whether the treewidth is at most w parameterized by w is FPT.

8.4 Optional: Cops and Robber Interpretation of Treewidth

Let $G = (V, E)$ be a graph and consider the following game: there is a robber r and w cops $c_1, \ldots, c_w$. One player controls the robber and the other player controls the cops. A position is described by vertices $v^r, v_1^c, \ldots, v_l^c \in V$, where $l \leq w$. In one turn, the cop-player removes a cop from the board or, if $l < w$ adds another one to an arbitrary position. During the move of the cop-player the robber may move freely around the edges of G as long as he does not visit a vertex occupied by a cop, he may use the knowledge of where the new cop will be added for this. The game starts with zero cops and the robber at a position of his choice, and ends when an added cop lands on the vertex where the robber is. See also the slides for examples.

Theorem 8.2. $w + 1$ *cops can win if and only if the graph has treewidth at most* w.

The proof of this theorem (in particular, the forward direction) is not that easy and will be omitted.

8.5 Nice Tree decompositions

Algorithms using tree decompositions are often presented using *nice tree decompositions*:

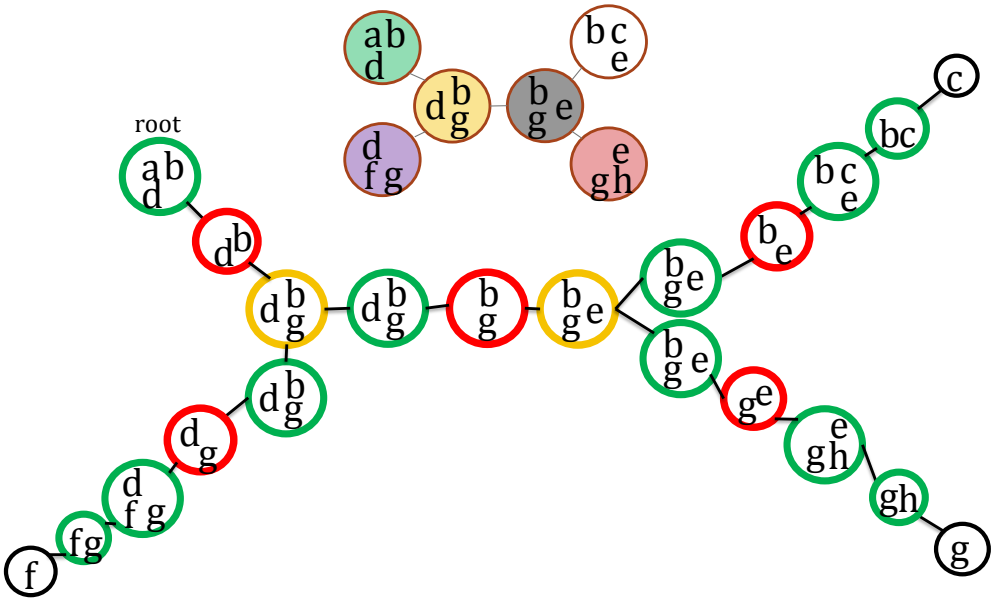

Figure 8.2: Example of a nice tree decomposition. Leaves are in black, introduce nodes in green, forget nodes in red and join nodes in yellow.

Definition 8.3. *A* nice tree decomposition *is a tree decomposition* (X, T) *where* $X = (X_1 \ldots, X_l)$, T *is a rooted tree and:*

- *every bag of* T *has at most two children,*
- *if a bag* X_i *has two children* $X_j, X_{j'}$*, then* $X_i = X_j = X_{j'}$*,*
- *if a bag* X_i *has one child* X_j*, then either*
  - $|X_i| = |X_j| + 1$ *and* $X_j \subseteq X_i$*; we say* X_i introduces v*, if* $X_i \setminus X_j = \{v\}$*, or*
  - $|X_i| = |X_j| - 1$ *and* $X_i \subseteq X_j$*; we say* X_i forgets v*, if* $X_j \setminus X_i = \{v\}$*.*

Lemma 8.1. *Given a graph* G *and a tree decomposition of* G *of width* w*, we can compute a nice tree decomposition of* G *of width* w *in polynomial time.*

Proof. Pick a root arbitrarily. Add bags under the leaves that introduce all vertices of the original leaves except one. If a bag X_i has a child X_j, add in the new tree decomposition bags between X_i and X_j where, starting from X_i we first have a series of introduce vertices that introduce $X_i \setminus X_j$ until we have a bag with set $X_{\cap} X_j$ and then a series of forget bags that forget vertices from $X_j \setminus X_i$. □

8.6 Weighted Independent Set on Graphs of Small Treewidth

Now let's see how to generalize the approach from Sections 8.1 and 8.2. First it is convenient to introduce some notation that is useful in general. Assume we are given a nice tree decomposition

(X, T). Since T is rooted, we may refer to children and ancestors etc. For a bag $i = 1, \ldots, l$, let $G_i = (V_i, E_i)$ be the subgraph of G induced by all vertices in bags that are descendants of bag i in T.

Theorem 8.3. *There exists an algorithm that given a tree decomposition of G of width w solves weighted Independent Set in $O^*(2^w)$ time.*

Proof. We first define the table entries of the dynamic programming algorithm: for every bag $i = 1, \ldots, l$ and subset $Y \subseteq X_i$ we define

$$A[i, Y] = \max\{\omega(W) : W \text{ is an independent set of } G_i \wedge W \cap X_i = Y\}.$$

Or less formally, $A[i, Y]$ is the maximum weight of an independent set of the graph G_i where we have to include from X_i exactly the vertices Y (and no more).

The algorithm computes $A[i, Y]$ for all bags $i = 1, \ldots, l$ and $Y \subseteq X_i$ in a bottom-up fashion (i.e. starting with the leaf bags and ending with the root bag). We now give the recurrence for $A[i, Y]$ that is used by the dynamic programming algorithm. In order to simplify notation let v be the vertex introduced and contained in an introduce or leaf bag, and let j, j' be the left and right child of i in T if present.

- **Leaf bag**:
$$A[i, \emptyset] = 0, \qquad A[i, \{v\}] = \omega(v)$$
- **Introduce vertex bag**:
$$\begin{aligned} A[i, Y] &= A[j, Y], && \text{if } v \notin Y, \\ A[i, Y \cup \{v\}] &= A[j, Y] + \omega(v), && \text{if } N(v) \cap Y = \emptyset, \\ A[i, Y \cup \{v\}] &= -\infty, && \text{if } N(v) \cap Y \neq \emptyset. \end{aligned}$$
If Y does not include v, v is not picked and hence not relevant and we can simply look the table entry up in $A[j, Y]$. If Y includes v we either account for it if no neighbor of it is picked as well (note that all neighbors of v in G_i need to be in X_i since the incident edges need to be covered somewhere and if it is not in $T[i]$ it is above and then the neighbor cannot be in $T[v]$ by the connectivity constraint), or return $-\infty$ if a neighbor is included since then we maximize over an empty set.
- **Forget bag**:
$$A[i, Y] = \max\{A[j, Y], A[j, Y \cup \{v\}]\}$$
In the child bag the vertex v can either be in the independent set or not. We account for its weight and whether this is a valid choice when it is introduced further down in the tree.
- **Join bag**:
$$A[i, Y] = A[j, Y] + A[j', Y] - \omega(Y)$$
The two independent sets of G_j and $G_{j'}$ together form an independent of G_i if they are consistent in the overlapping vertex set S, and we take care that vertices in S are accounted twice by subtracting this weight.

It is easy to see that using the above recurrences, $A[r, Y]$ can be computed in $O^*(2^k)$ time for every $Y \subseteq X_r$, if r is the root of T and it is follows by definition that $\max_{Y \subseteq X_r} A[r, Y]$ equals the maximum weight of an independent set. □

8.7 Minimum Weight Dominating Set

A *dominating set* of a graph $G = (V, E)$ is a subset $X \subseteq V$ such that for every $v \in V$, either $v \in X$ or $N(v) \cap X \neq \emptyset$. Equivalenty, for every $v \in V$ it holds that $N[v] \cap X \neq \emptyset$. In the minimum weight dominating set problem we are given a graph $G = (V, E)$ and weight function $\omega : V \to \mathbb{N}$ and need to find a dominating set X minimizing $\omega(X)$. We will now see an algorithm solving this in linear time on trees. As before we assume T is rooted and we use dynamic programming. The point of this exercise is that we need to distinguish three cases now; for independent set we only needed to know whether v is included or not to solve the subproblems independently, but for dominating set we also need to know whether the vertex is already dominated if it is not included (i.e. in which subproblem we pick a neighbor of v).

$$\begin{aligned} A_D[v] &= \text{ min weight of a dominating set of } T[v] \setminus v \\ A_I[v] &= \text{ min weight of dominating set of } T[v] \text{ that includes } v \\ A_E[v] &= \text{ min weight of dominating set of } T[v] \text{ that excludes } v. \end{aligned}$$

From the definition we see the minimum weight can be read off from $\min\{A_I[r], A_E[r]\}$, where r is the root of the tree. If v is a leaf we see that $A_E[v] = \infty$, $A_D[v] = 0$ and $A_I[v] = \omega(v)$. If v is not a leaf, we have that

$$\begin{aligned} A_D[v] &= \sum_{c \in ch(v)} \min\{A_I[c], A_E[c]\} \\ A_I[v] &= \omega(v) + \sum_{c \in ch(v)} \min\{A_D[c], A_I[c]\} \\ A_E[v] &= \min_{c \in ch(v)} \Big\{ A_I[c] + \sum_{c' \in ch(v) \setminus c} \min\{A_I[c'], A_E[c']\} \Big\}. \end{aligned}$$

To see that this is true, note that for $A_D[v]$ the problem splits into independent subproblems as before, for $A_I[v]$ we have that v is included so we account for its weight and solve the subproblems induced by the subtrees rooted at the children of v independently: we may decide whether to include each child or not and if we do not include it we have that it is already dominated to we refer to $A_D[c]$. For $A_E[c]$, v still needs to be dominated so we go over all possibilities of which child is picked and given such a picked vertex we go over all possibilities of the other children (which again induce independent subproblems)Note that a naïve evaluation of this recurrence would result in a quadratic time algorithm, it is easy to make it run in linear time but that is not really relevant for us.

8.8 k-path is FPT

In the k-path problem we are given an undirected graph $G = (V, E)$ and an integer k and need to determine whether there exists a simple path in G on at least k vertices. This problem is NP-hard since it reduces to Hamiltonian path in the special case $k = n$. Using a similar approach as in the previous sections, one can find an algorithm that given a graph G with tree decomposition of

width w and integer k, determines whether a k path exists in $O^*(w^{O(w)})$ time. We will explore this direction more in the exercises, but on a high level the intuition is that we need table entries for all possible ways a partial path can be connected. This is about equal to the number of matchings on w vertices and there are $w^{O(w)}$ matchings on w vertices which is the reason we arrive at the running time $O^*(w^{O(w)})$, informally.

Now we'll use this algorithm as a subroutine to solve the k-path problem in $O^*(k^k)$ time. Assume that the graph is connected. If not we can simply look for k-paths in all connected components independently. Pick a vertex s of the input graph arbitrarily and perform a DFS from s. Consider the DFS tree resulting from this. If it has height at least k, G clearly has a k-path since any path from the root to a leaf is a k-path. Otherwise, let $l_1, \ldots, l_p$ be the leaves of the DFS tree ordered as visited by the DFS and let L_i be all vertices on the path from s to l_i in the DFS tree.

We claim that (X, T) where $X = \{L_1, \ldots, L_p\}$ and T the path with edges (L_i, L_{i+1}) is a tree decomposition of G. To see this, note that clearly Property 1 is met since all vertices are on some path from s to l_i; for Property 2, all edges are indeed in some bag since by the property of a DFS tree all edges are between a vertex and one of his ancestors, so both lie on some path from a leaf to the root simultaneously. For the third property, note that a vertex v is contained in all bags L_i where l_i is a leaf of the subtree rooted at v and these leafs are consecutive numbered since they are visited consecutively in the DFS.

Thus we found a polynomial time algorithm that either finds a k-path or a tree decomposition of width at most k, and using the mentioned $O^*(w^{O(w)})$-time algorithm in the latter case we get an $O^*(k^{O(k)})$ time algorithm for k-path.

8.9 Approximation of Weighted Independent Set on Planar Graphs

Now we return to a problem already studied in the beginning of this course. Given a planar graph $G = (V, E)$ and weights $\omega : V \to \mathbb{N}$ we want to find an independent set $X \subseteq V$ such that $\omega(X) \geq (1 - \epsilon)OPT$ where OPT is the maximum of $\omega(X)$ over all independent sets X of G. The following theorem will be highly useful for that (and it also has other uses beyond approximation).

Theorem 8.4. *Given a planar graph $G = (V, E)$ and a spanning tree S of G of height*[1] *at most h, a tree decomposition of G of width at most $3h$ can be found in polynomial time.*

Proof sketch. First triangulate the graph (i.e. add edges until all faces are formed by three edges). If we find a tree decomposition of the new graph, it will also be a tree decomposition of the original graph since the added edges only make Property 2 more strict. Assign a root to S arbitrarily.

Then look at the dual graph but only have edges of the dual graph between two faces that share an edge *not* in S. Call this graph T. It is easy to see that T has no cycles since if it would it needs to cross S since S is spanning. We'll show in a second that T is connected, so it has to be a tree.

For every face i we have a bag X_i containing all vertices incident to that face and also all ancestors of these vertices in S. It is clear that all vertices and edges are in some bag, and for Property 3. it can be seen that the set of bags containing any vertex v induce a connected graph in T since we can walk around the faces incident to all edges of $S[v]$[2]

[1] The maximum number of edges on the path from the root to a leaf, where the maximum is taken over all leaves.

[2] This is not a rigorous argument, formally one should use induction here (where the leaves are the base case).

Since all bags contain the root of S, it follows that T is connected by the previous argument, so indeed (X, T) is a tree decomposition. The width is at most $3h$ since for any face the three incident vertices each have at most $3h + 1$ ancestors in S. □

Baker's layering approach Now let us see how Theorem 8.4 can be used for approximating the maximum weight independent set. We'll see an algorithm with running time $O^*(2^{3/\epsilon})$, which improves the algorithm taking $O^*(2^{O(1/\epsilon^2)})$ time algorithm we have seen in lecture 3/4.

The algorithm is as follows:

Algorithm `apxis`$(G = (V, E), \epsilon)$, Assumes G planar
Output: The weight of an independent set of weight $\geq (1 - \epsilon)OPT$, where OPT is maximum weight of an independent set of G.
1: Choose a vertex $s \in V$ arbitrarily
2: Perform a breadth-first search from s
3: For $i = 0, \ldots, n$, let L_i be all vertices at distance i from s (e.g., at depth i in the BFS tree)
4: For $i = 0, \ldots, k-1$, let V_i be the union of all sets L_j where $i \neq j \mod k$ (i.e., k is not a divisor of $i - j$)
5: Set $max = -\infty$.
6: **for** $l = 0$ **to** $k - 1$ **do**
7: Compute a tree decomposition of $G[V_i]$ of width $3k$
to find such tree decomposition, split $G[V_i]$ into connected components and find
tree decomposition of each connected component separately as follows:
add a vertex z on the place of s in the embedding and add an edge from z to every vertex
in L_a, where a is the smallest integer such that L_a is in the connected component.
A BFS tree from z will be of height at most k so we can use Theorem 8.4
to get a tree decomposition of the graph without the added vertex use Exercise 8.3
8: Compute a maximum weight independent set of $G[V_i]$ as outlined in Section 8.6
9: **if** found weight $> max$ **then** set $max =$ found weight
10: **return** max

Algorithm 2: Baker's layering approach for the Max Weight Independent Set problem.

Let us first see whether the output of `apxis` is indeed as promised. Denote $\overline{V_i} = V \setminus V_i$, i.e. $\overline{V_i}$ are all vertices in L_j such that $i \equiv j \mod k$). We see that the sets $\overline{V_0}, \ldots, \overline{V_{k-1}}$ partition V. Thus, if I is a maximum weight independent (i.e. $\omega(I) = OPT$), then for some $l = 1, \ldots, k$ we have that $\omega(I \cap \overline{V_i}) \leq OPT/k$ and thus $\omega(I \cap V_I) \geq OPT - OPT/k \geq OPT(1 - \epsilon)$. Since in iteration l of Line 6 we find the maximum weight independent set of $G[V_i]$ the algorithm finds $max \geq \omega(I \cap V_I)$, so $max \geq OPT(1 - \epsilon)$.

For the running time of the algorithm, note that Line 8 is the only part that does not take time polynomial in the input size, which takes $O^*(2^{3k})$ time so the algorithm takes $O^*(2^{3/\epsilon})$ time. To see that we indeed obtain a tree decomposition of $G[V_i]$ of width $3k$ in the way described, first note that L_b separates all vertices in L_a from all vertices in L_c for $a < b < c$ since if there would be an edge between a vertex in $u \in L_a$ and $v \in L_c$ then v would be at distance at most b from s, contradicting it is in L_c. Thus, $G[V_i]$ splits into separate connected components induced by

consecutive layers $L_a, L_{a+1} \ldots,$. It is easy to see that from tree decompositions of the connected components we can obtain a tree decomposition of same width of the whole graph by adding an empty bag connected the tree decompositions, so now it is sufficient to find tree decompositions of the connected components. To do this, consider one connected component and let a be the smallest integers such that it contains L_a. We see that if we add z at the place of s and add edges to all vertices in L_a we still have a planar graph since in the original graph there was a tree rooted at s and leaves at L_a with no vertices in L_b for $a > b$. Now performing a BFS from z in this graph gives a BFS tree of height at most k by definition of the layers so we may use Theorem 8.4 to obtain a tree decomposition of this connected component with z added of width at most $3k$. As asked in Exercise 8.3, this tree decomposition can easily be modified to obtain a tree decomposition of the connected component.

8.10 Graph Minors

For a graph $G = (V, E)$ and an edge $(u, v) \in E$, we can *contract* e as follows: delete vertices u, v from G and add a new vertex $w_{u,v}$ adjacent to $(N(u) \cup N(v))$. We say that a graph H is a *minor* of a graph G if H can be obtained from a subgraph of G using edge contractions. Or equivalenty, we can obtain H from G by removing vertices and removing and contracting edges. We call such a series of operations a *minor model*.

The $(l \times l)$-grid is the graph on vertices $v_{i,j}$ for $1 \leq i, j \leq l$ with edges $(v_{i,j}, v_{i+1,j})$ and $(v_{i,j}, v_{i+1,j})$ for all relevant i, j.

Theorem 8.5 (Grid Minor Theorem)**.** *For every integer l, every planar graph either has a $(l \times l)$-grid as a minor or treewidth at most $9l$. Moreover, there exists a polynomial time algorithm that either finds such a minor model or tree decomposition.*

Unfortunately, the proof of this theorem is beyond the scope of this course, but we describe some intuition here: we would either like to separate vertices in the 'west' from 'east' or vertices from 'south' to 'north'. If both are not possible, max flow min cut tells us there must be many disjoint paths from vertices in the west to east and south to north, and all these disjoint paths together can be used to find a grid minor.

A corollary of the Grid Minor Theorem is that planar graphs have treewidth $O(\sqrt{n})$. The ad-hoc result we found for independent set in planar graphs in the beginning can also be obtained by combining this observation with the $O^*(2^w)$-time algorithm for independent set in graphs of treewidth at most w.

8.11 Exercises

Exercise 8.1. Show how to build a tree decomposition of any tree of width 1.

Exercise 8.2. Explain why every n-vertex graph has treewidth at most $n - 1$, can you find an n-vertex graph with treewidth $n - 1$?

Exercise 8.3. Show that removing a vertex does not increase the treewidth.

Exercise 8.4. Give an algorithm that given a graph G and tree decomposition of G of width at most w, determines whether G is 3-colorable in $O^*(3^w)$ time.

Exercise 8.5. Show that any complete bipartite graph where both parts have n vertices has treewidth at most n.

Exercise 8.6. Show that the $l \times l$-grid has treewidth at most l.

Exercise 8.7. Give an algorithm that given a graph G and tree decomposition of G of width at most w, finds the minimum weight dominating set in $O^*(9^w)$ time.