

# ORCO – Graphs and Discrete Structures

## October 27, 2021 – Lecture 5

### 1 Treewidth

#### 1.1 Definitions

Recall the following definition from Lecture 4. Given a graph  $G$ , a *subtree representation* of  $G$  is a tree  $\mathcal{T}$  together with subsets  $(B_t)_{t \in \mathcal{T}}$  of vertices of  $G$  (the set  $B_t$  is called the *bag* of  $t$ ), with the following properties.

1.  $u$  and  $v$  are adjacent in  $G$  if and only if there is a bag  $B_t$  containing  $u$  and  $v$  in the subtree representation, and
2. for any vertex  $v$ , the set of nodes  $t$  of  $\mathcal{T}$  whose bag  $B_t$  contains  $v$  forms a subtree of  $\mathcal{T}$  (equivalently, if  $v$  is in two bags  $B_t$  and  $B_s$ , then  $v$  lies in all the bags  $B_{t'}$  such that  $t'$  is on the path between  $t$  and  $s$  in  $\mathcal{T}$ ).

We defined *chordal graphs* as the graphs admitting a subtree representation.

We now define the notion of tree-decomposition, which is very similar to that of a subtree representation. The only difference is in the first item, where we do not require that vertices in the same bag are adjacent.

Given a graph  $G$ , a *tree decomposition* of  $G$  is a tree  $\mathcal{T}$  together with subsets  $(B_t)_{t \in \mathcal{T}}$  of vertices of  $G$  (the set  $B_t$  is called the *bag* of  $t$ ), with the following properties.

1. If  $u$  and  $v$  are adjacent in  $G$ , then there is a bag  $B_t$  containing  $u$  and  $v$  in the tree-decomposition, and
2. for any vertex  $v$ , the set of nodes  $t$  of  $\mathcal{T}$  whose bag  $B_t$  contains  $v$  forms a subtree of  $\mathcal{T}$  (equivalently, if  $v$  is in two bags  $B_t$  and  $B_s$ , then  $v$  lies in all the bags  $B_{t'}$  such that  $t'$  is on the path between  $t$  and  $s$  in  $\mathcal{T}$ ).

The *width* of the tree decomposition is defined as the maximum size of a bag  $B_t$  minus 1 (the  $-1$  is to make sure trees have treewidth 1, see below). The *treewidth* of a graph  $G$ , denoted by  $\text{tw}(G)$ , is the minimum  $k$  such that  $G$  has a tree decomposition of width at most  $k$ .

Using the results of the last lecture, we can define treewidth equivalently as follows. A graph has treewidth at most  $k$  if and only if it is a subgraph of a chordal graph of clique number at most  $k + 1$ .

A  $k$ -tree is a graph defined inductively as follows: it is either a complete graph on  $k + 1$  vertices, or obtained from a  $k$ -tree  $H$  by adding a vertex whose neighborhood is a clique of size  $k$  in  $H$ . Note that 1-trees are precisely trees. Using again the results of the last lecture (the existence of a vertex whose neighborhood is a clique, in every chordal graphs), we can also define treewidth as follows (this is equivalent to the previous definitions): a graph  $G$  has treewidth at most  $k$  if and only if it is a subgraph of a  $k$ -tree.

## 1.2 Algorithmic applications

It turns out that most combinatorial problems can be solved efficiently in graphs of bounded treewidth using dynamic programming. To illustrate that, we recall a simple algorithm to compute a MAXIMUM INDEPENDENT SET (MIS) in a tree, and explain briefly how to modify the algorithm to extend it to graphs of bounded treewidth. Recall that a subset  $S$  of vertices of a graph  $G$  is an *independent set* in  $G$  if no two vertices of  $S$  are adjacent.

Consider a tree  $T$ , and choose some root  $r$  (turning  $T$  into a rooted tree). For any vertex  $v$ , let  $T_v$  denote the subtree of  $T$  rooted in  $v$ . Our goal is to compute the following two quantities for each vertex  $v$ :  $I^+(v)$ , defined as the maximum size of an independent set of  $T_v$  that contains  $v$ , and  $I^-(v)$ , defined as the maximum size of an independent set of  $T_v$  that does not contain  $v$ . Note that  $T = T_r$  and thus the maximum size of an independent set in  $G$  is precisely  $\max(I^+(r), I^-(r))$ .

If  $v$  is a leaf, we simply define  $I^+(v) = 1$  and  $I^-(v) = 0$ . If  $v$  is not a leaf, we compute  $I^+(u)$  and  $I^-(u)$  for all children  $u$  of  $v$  and define  $I^+(v)$  as 1 plus the sum of  $I^-(u)$  for all children  $u$  of  $v$ . Moreover we define  $I^-(v)$  as the sum of  $\max(I^+(u), I^-(u))$  for all children  $u$  of  $v$ .

This takes  $O(n)$  steps, where  $n = |V(T)|$ , and allows to compute the size of a maximum independent set in  $T$ .

We now explain how to extend this approach to graphs of treewidth  $k$ , for fixed  $k$ . Let  $G$  be a graph of treewidth  $k$  and consider a tree decomposition of  $G$  of width  $k$ , with underlying tree  $\mathcal{T}$  and bags  $(B_t)_{t \in \mathcal{T}}$ . We root  $\mathcal{T}$  at some arbitrary vertex  $r$ . Given  $t \in \mathcal{T}$ , we denote again by  $\mathcal{T}_t$  the subtree of

$\mathcal{T}$  rooted in  $t$ , and we denote by  $D_t$  the union of all the bags  $B_s$ , for  $s \in \mathcal{T}_t$ . The goal will be to store, for every independent set  $J \in B_t$ , the number of independent sets  $I$  of  $G[D_t]$  (the subgraph of  $G$  induced by  $D_t$ ) such that  $I \cap B_t = J$ . There are at most  $2^{k+1}$  choices for  $J$ , as  $B_t$  has size at most  $k+1$ , and for each choice of  $J$  the relevant value can be computed from the values stored at the children  $s$  of  $t$ , as before.

We obtain a complexity of order  $2^{O(k)} \cdot |V(\mathcal{T})|$  for the computation of the size of a maximum independent set. We need to argue that a tree decomposition with at most  $|V(G)|$  bags exists (this is the case), and that it can be constructed efficiently. The latter is a bit more subtle, as computing the treewidth is NP-hard. But for fixed  $k$  a tree decomposition of width at most  $k$  can be constructed efficiently (if it exists): the problem is FPT parametrized by the treewidth, see the next section. Alternatively, a tree decomposition of width not much larger than the treewidth can be computed efficiently (with only a polynomial dependence in the treewidth).

### 1.3 Other algorithmic applications

In this section again we briefly explain some important applications of treewidth. It can be checked that a large grid or wall (see Figure 1) has large treewidth. A major result is that the converse is also true, in the following sense. A *subdivision* of a graph  $G$  is any graph obtained from  $G$  by replacing some edges of  $G$  by paths with the same ends (see Figure 1).

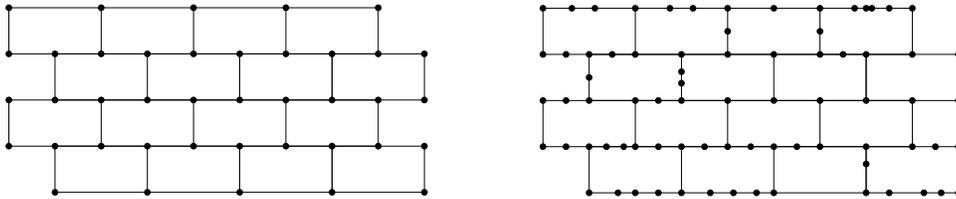


Figure 1: A  $4 \times 4$ -wall and a subdivided  $4 \times 4$ -wall.

**Theorem 1** (The grid minor theorem, Robertson and Seymour). *There is a function  $f$  such that for any  $k$ , if a graph has treewidth at least  $f(k)$ , then it contains a subdivision of a  $k \times k$ -wall as a subgraph.*

To see how this can be used in applications, consider the following problem.

2 DISJOINT ROOTED PATHS: *given a graph  $G$  with specified vertices  $s_1, s_2, t_1, t_2$ , is there a path from  $s_1$  to  $t_1$  and a path from  $s_2$  to  $t_2$  that are vertex-disjoint?*

There are two natural obstructions: the first is low connectivity (think of  $s_1, s_2$  in some component  $C_1$  and  $t_1, t_2$  in some component  $C_2$ , such that  $C_1$  and  $C_2$  have a single vertex in common). The second is topological: imagine that  $s_1, s_2, t_1, t_2$  lie in this order on the outerface of a planar graph (recall the Jordan curve theorem). Obviously these two obstructions can be combined in many ways, so it is not a priori clear that the problem above can be solved in polynomial time.

We now briefly sketch an approach to solve this problem (and the more general version with  $k$  disjoint paths instead of 2) in polynomial time, due to Robertson and Seymour. Given a graph  $G$  and  $s_1, s_2, t_1, t_2$ , we say that a vertex  $v$  is *irrelevant* if the following holds: there exist 2 disjoint rooted paths in  $G$  from  $s_1$  to  $t_1$  and  $s_2$  to  $t_2$  if and only there exist 2 disjoint rooted paths in  $G \setminus \{v\}$  from  $s_1$  to  $t_1$  and  $s_2$  to  $t_2$ . In other words,  $v$  is useless in the problem, it can be safely removed without affecting the answer.

The algorithm to solve the 2 disjoint rooted paths problem is now the following.

- As long as the treewidth of  $G$  is large, use the grid minor theorem to find a subdivision of a large wall, and use the wall to identify an irrelevant vertex (some vertex lying sufficiently deep in the wall, so that any paths going through it could be rerouted using others branches of the walls). Remove the irrelevant vertex.
- If no such irrelevant vertex exists then the treewidth must be small, so we can solve the problem efficiently using dynamic programming.

We end up with a polynomial time algorithm for the 2 disjoint rooted path problem, and more generally with an algorithm running in  $f(k) \cdot \text{poly}(n)$ , for the  $k$  disjoint rooted path problem.

## 2 Fixed Parameter Tractable problems

In the previous section we have seen problems that can be solved in time  $f(k) \cdot n^c$ , for some fixed constant  $c$  independent of  $k$ . These problems are

said to be *fixed parameter tractable (FPT)*, *parametrized by  $k$* . Note that this is much stronger than problems that are polynomial time solvable when  $k$  is fixed (where we can have a complexity of order  $O(n^{f(k)})$ ).

In the remainder we present a simple and powerful technique to obtain such algorithms, which is useful beyond graph theory. The idea is to reduce the original instance to an equivalent instance of size bounded by a function of  $k$ , and then to run a brute force (typically exponential) algorithm to solve the small instance.

Given an instance  $(G, k)$  of some graph problem parametrized by some integer  $k$ , a *kernel* is a polynomial time reduction from  $(G, k)$  to some  $(G', k')$  such that

- $k' \leq k$
- $(G, k)$  is equivalent to  $(G', k')$  (in other words,  $(G, k)$  a positive instance if and only if  $(G', k')$  is a positive instance).
- $|V(G')| \leq f(k)$ , for some function  $f$ .

The function  $f$  is called the size of the kernel, and the goal is usually to obtain linear or quadratic kernels.

We apply kernelization to a classical graph problem, MINIMUM VERTEX COVER (MVC). A *vertex cover* in a graph  $G$  is a subset  $S$  of vertices such that for every edge  $uv$ , at least one of  $u$  and  $v$  lies in  $S$ . Our problem is to decide whether a given graph  $G$  has a vertex cover of size at most  $k$  (we say that the problem is parametrized by the size of the solution).

To reduce the size of the instance while maintaining an equivalent instance (that is a graph  $G'$  and an integer  $k'$  such that  $G$  has a vertex cover of size at most  $k$  if and only if  $G'$  has a vertex cover of size at most  $k'$ ), we repeatedly apply the following rules, until they cannot be applied anymore.

1. If  $G$  contains an isolated vertex  $v$ , replace  $(G, k)$  by  $(G \setminus \{v\}, k)$ .
2. If  $G$  contains a vertex  $v$  of degree at least  $k + 1$ , replace  $(G, k)$  by  $(G \setminus \{v\}, k - 1)$ .

The validity of the first rule (the fact that it yields an equivalent instance) comes from the fact that isolated vertices are not part of any optimal vertex

cover. For the second rule, observe that if  $v$  is not part of the vertex cover, all its neighbors are, so if  $d(v) \geq k + 1$  then  $v$  has to be part of any vertex cover of size at most  $k$ .

Assume that none of the two rules can be applied. We claim that if  $G$  has a vertex cover of size at most  $k$ , then  $|V(G)| \leq k(k + 1)$ . This follows from the fact that since  $G$  has no isolated vertex, all vertices are either in the vertex cover, or adjacent to a vertex of the vertex cover (which contains at most  $k$  vertices, all of which have degree at most  $k$ ), so  $G$  contains at most  $k + k \cdot k = k(k + 1)$  vertices.

So if at this point,  $|V(G)| > k(k + 1)$ , then we can replace  $(G, k)$  by a trivial negative instance of bounded size, while if  $|V(G)| \leq k(k + 1)$  the instance has bounded size (quadratic in  $k$ ), as desired.

Once we have reduced the problem to an instance of size  $N \leq k(k + 1)$  we have several possibilities. We can simply check all  $\binom{N}{k}$   $k$ -element subsets of vertices of the graph and see if they form a vertex cover. A smarter option is to do the following: we take an edge  $uv$  such that neither  $u$  nor  $v$  is in the current vertex cover (starting with the empty set), and we check instances  $(G \setminus u, k - 1)$  and  $(G \setminus v, k - 1)$  (as at least one of  $u$  and  $v$  has to be in a vertex cover). The depth of the recursion tree is bounded by  $k$ , so the search runs in  $N^2 \cdot 2^k = 2^{O(k)}$  (since  $N \leq k(k + 1)$ ).

Overall, the algorithm for deciding whether a graph  $G$  on  $n$  vertices has a vertex cover of size at most  $k$  runs in time  $O(n^2) + 2^{O(k)}$ , so is fixed parameter tractable parametrized by the size of the solution (note that in this example we obtain a complexity of order  $O(f(k) + n^c)$ , which is even better than  $O(f(k) \cdot n^c)$ ).

*Homework.* Add a third rule related to vertices of degree 1. How does it affect the size of the kernel?