# Lecture notes for Apr 17, 2023
# FPT and tree-width

Chun-Hung Liu

April 12, 2023

## 1 Search tree algorithms

Another trick to obtain a FPT algorithm is to consider a "search tree".

Assume that we have an increasing function $b$, constants $c_1$ and $c_2$, and an algorithm such that given an input $(I,k)$, we can produce $b(k)$ instances $(I_1,k_1),(I_2,k_2),...,(I_{b(k)},k_{b(k)})$ with $|I_i|\leq|I|$ and $k_i\leq k-1$ in time $|I|^{c_1}$ and we can determine whether $(I,k)$ is a positive instance or not in time $|I|^{c_2}$ by only using the information whether each $(I_i,k_i)$ is a positive instance or not. Then we can run the above algorithm again to each $(I_i,k_i)$ to further obtain $(I_{i,1},k_{i,1}),...,(I_{i,b(k_i)},k_{i,b(k_i)})$. Repeat this process until each $k_i$ becomes very small so that we can do brute force in time $n^{c_3}$ for some constant $c_3$. Then we can work backwards to know whether $(I,k)$ is positive or not by seeing whether each $(I_i,k_i)$ is positive or not.

What's the running time of this process? We can image that the above process creates a rooted tree $T$: The root of $T$ is $(I,k)$ and the children of $(I,k)$ are $(I_1,k_1),...,(I_{b(k)},k_{b(k)})$, and then each $(I_i,k_i)$ has $b(k_i)$ children, and so on. Every vertex of $T$ has at most $b(k)$ children, since $b$ is increasing and each $k_i$ is smaller than $k$. The height of $T$ is at most $k$, since each $k_i$ is strictly smaller than $k$. Hence $T$ has at most $(b(k))^k$ leaves. So $T$ has at most $2(b(k))^k$ vertices and edges. Since each $|I_i|\leq|I|$, it takes $|I|^{c_1}\cdot|E(T)|\leq 2(b(k))^k|I|^{c_1}$ time to construct $T$. And it takes $|I|^{c_3}$ time to decide whether a leaf instance is positive or not by brute force. So it takes $2(b(k))^k\cdot|I|^{c_3}$ time to decide the answers for all leaf instances. And the backward process take $|I|^{c_2}\cdot|E(T)|\leq 2(b(k))^k|I|^{c_2}$ time to decide the answer for the root instance $(I,k)$. So the total running time is at most $6(b(k))^k|I|^{\max\{c_1,c_2,c_3\}}$.

Hence this search tree algorithm gives an algorithm with running time $O(f(k)n^c)$, where $f(k) = (b(k))^k$ and $c = \max\{c_1, c_2, c_3\}$. So it is a FPT algorithm, and the function $f$ is determined by the function $b$. In fact, the above analysis works in the same way even we do not have the function $b$ to bound the degree of vertices in $T$. It works as long as the number of leaves of $T$ is bounded. More precisely, if $L(T)$ is the number of leaves of $T$, then the above search tree algorithm runs in time $O(L(T) \cdot n^c)$.

We first see an example that applies this trick to the vertex-cover problem.

**Proposition 1** *$k$-Vertex-Cover can be solved in time $O(2^k n)$.*

**Proof.** Assume an input $(G, k)$ is given. Then we pick an edge $uv$ of $G$. Let $(G_1, k_1) = (G - u, k - 1)$ and $(G_2, k_2) = (G - v, k - 1)$.

Note that if there exists a vertex-cover $S$ of $G$ with size at most $k$, and $S$ contains $u$ or $v$, so either $S - \{u\}$ is a vertex-cover of $G - u$ with size at most $k - 1$ or $S - \{v\}$ is a vertex-cover of $G - v$ with size at most $k - 1$. So if $(G, k)$ is a positive instance, then at least one of $(G_1, k_1)$ and $(G_2, k_2)$ is a positive instance. Conversely, if at least one of $(G_1, k_1)$ and $(G_2, k_2)$ is a positive instance, say $G_1$ has a vertex-cover $S_1$ of size at most $k_1 = k - 1$, then $G$ has a vertex-cover $S_1 \cup \{u\}$ with size at most $k$. Therefore, $(G, k)$ is a positive instance if and only if at least one of $(G_1, k_1)$ and $(G_2, k_2)$ is a positive instance.

We do brute force if $k = 0$. So the search tree algorithm is with the function $b = 2$ and constants $c = c' = 0$ and $c'' = 1$, and hence has running time $O(2^k n)$. ∎

Now we see another example for the search tree algorithm.

A *hole* in a graph $G$ is an induced subgraph $H$ such that $H$ is isomorphic to a cycle of length at least 4. A graph is *chordal* if it has no holes.

Chordal graphs are interesting for various reasons. For example, chordal graphs are typical examples of perfect graphs and are exactly the intersection graphs of subtrees of a tree.

We consider the following problem.

==========

**Chordal Completion Problem**
**Input:** A simple graph $G$ and an integer $k$.
**Output:** Determine whether it is possible to add at most $k$ edges to make $G$ become a chordal graph.

==========

The case when $G$ is a cycle is easy.

**Lemma 2** *Let $G$ be a cycle of length $\ell$ with $\ell \geq 3$. Then $(G, k)$ is a positive instance if and only if $k \geq \ell - 3$.*

**Proof.** It can be easily proved by induction on $\ell$. ■

Note that each way to add exactly $\ell - 3$ edges to make a cycle of length $\ell$ become a chordal graph is exactly a way to triangulate a convex $\ell$-side polygon by adding diagonals. The number of ways to triangulate a convex $\ell$-side polygon by adding diagonals is equal to the Catalan number. The proof of this fact can be found in most of standard combintorics courses, so we do not repeat it here. (In fact, we only need the "$\leq 4^{\ell-3}$" part in the following lemma. And this part can be easily proved by induction on $\ell$.)

**Lemma 3** *Let $G$ be a cycle of length $\ell$ with $\ell \geq 4$. Then there are exactly $\frac{1}{\ell-1}\binom{2\ell-4}{\ell-2} \leq 4^{\ell-3}$ ways to add exactly $\ell - 3$ edges to make $G$ become a chordal graphs.*

**Theorem 4** *The Chrodal Partition Problem can be solved in time $O(4^k n^c)$ for some constant $c$.*

**Proof.** We use a search tree algorithm. Let $(G, k)$ be the input.

We first determine whether $G$ has a hole, and if it does, find a hole in $G$. It is known that it can be done in polynomial time (but we will not describe how to do it here). If there is no hole in $G$, then $G$ is chordal and we answer "yes". So we may assume that we find a hole $C$ in $G$. By each way $W$ to add $|V(C)| - 3$ edges that makes $C$ chordal, we produce a child $(G', k - (|V(C)| - 3))$ of $(G, k)$, where $G'$ is obtained from $G$ by adding $|V(C)|-3$ edges in the way $W$. Note that $(G, k)$ has at most $4^{|V(C)|-3}$ children by Lemma 3. And $(G, k)$ is positive if and only if at least one child of $(G, k)$ is positive. And when $k = 0$, we can decide whether $(G, k)$ is a positive instance or not by testing whether $G$ has a hole and hence can be done in polynomial time.

We prove that the search tree $T$ created in this way has at most $4^k$ leaves. We prove this fact by induction on $k$. Note that $(G, k)$ has $4^{|V(C)|-3}$ children. For each children $h$ of $(G, k)$, it is an instance of the form $(G', k-|V(C)|+3)$.

Since $|V(C)| \geq 4$, for each child $h$ of $(G, k)$, by the induction hypothesis, the number of leaves of $T$ that are descendants of $h$ is at most $4^{k-|V(C)|+3}$. So $T$ has at most $4^{|V(C)|-3} \cdot 4^{k-|V(C)|+3} = 4^k$ leaves.

Therefore, the search tree algorithms runs in time $O(4^k n^c)$ for some constant $c$. ■

There are many other tricks to obtain FPT algorithms. One main research direction is to consider width parameter of graphs. We will give details for one such width parameter, called tree-width in the next section.

# 2 Tree-width

One way to get a parameterization of an algorithmic problem on graphs is to use properties of the input graphs. That is, we consider a measurement to measure the "complexity" of the input graph. For example, the chromatic number is such a measurement, as graphs with smaller chromatic number can be considered "simpler". But the chromatic number does not seem to be a useful measurement for parametrization of algorithmic problems. Graph width turns out to more effective. There are numerous different notions of width parameters extensively studied in the literature. In this section we focus on tree-width, which is arguably the most famous one.

Let $G$ be a graph. A *tree-decomposition* of $G$ is a pair $(T, \mathcal{X})$, where $T$ is a tree and $\mathcal{X} = \{X_t : t \in V(T)\}$ is a collection of subsets of $V(G)$ indexed by the vertices of $T$ such that

(TD1) $\bigcup_{t \in V(T)} X_t = V(G)$,

(TD2) for every edge $uv \in E(G)$, there exists $t \in V(T)$ such that $\{u, v\} \subseteq X_t$, and

(TD3) for every vertex $v \in V(G)$, the set $\{t \in V(T) : v \in X_t\}$ induces a connected subtree of $T$

(that is, if $t_1, t_2 \in V(T)$ with $v \in X_{t_1} \cap X_{t_2}$, then $v \in X_t$ for every vertex $t$ in the path in $T$ between $t_1$ and $t_2$).

Each set $X_t$ is called the *bag* at $t$.

Note that every graph has a "trivial" tree-decomposition, which is the tree-decomposition whose tree only has one vertex and the unique bag equals

$V(G)$. This tree-decomposition is useless, and we usually want a tree-decomposition whose every bag is small.

The *width* of the tree-decomposition $(T, \mathcal{X})$ is defined to be $\max_{t \in V(T)} |X_t| - 1$. The *tree-width* of $G$ is the minimum width of a tree-decomposition of $G$.

To get feelings about tree-decompositions, we first show some easy examples.

**Proposition 5** *Let $G$ be a graph. Then $G$ has tree-width 0 if and only if $G$ has no edge.*

**Proof.** ($\Rightarrow$) Clearly, if $G$ has at least one edge, then (TD2) implies that every tree-decomposition of $G$ has a bag with size at least two, so the tree-width is at least 1.

($\Leftarrow$) If $G$ has no edge, then by taking a tree $T$ with $V(T) = V(G)$ and defining $X_v = \{v\}$ for every $v \in V(T) = V(G)$, we obtain a tree-decomposition with width 0. ■

**Proposition 6** *Every tree with at least one edge has tree-width 1.*

**Proof.** Let $T$ be a tree with at least one edge. Let $T'$ be the tree obtained from $T$ by subdividing every edge of $T$ exactly once. So $V(T') = V(T) \cup E(T)$. That is, every vertex of $T'$ is either a vertex of $T$ or an edge of $T$.

For every $t \in V(T)$, let $X_t = \{t\}$. For every $uv \in E(T)$, let $X_{uv} = \{u, v\}$. Let $\mathcal{X} = \{X_t, X_{uv} : t \in V(T), uv \in E(T)\}$.

We show that $(T', \mathcal{X})$ is a tree-decomposition of $T$. Clearly (TD1) and (TD2) hold. Now we prove (TD3). Let $v \in V(T)$. Note that if $t \in V(T')$ such that $v \in X_t$, then either $t = v$ or $t$ is an edge of $T$ incident with $v$. So the subgraph of $T'$ induced by the set $\{t \in V(T') : v \in X_t\}$ is a star centered at $v$. Hence (TD3) holds.

So $(T', \mathcal{X})$ is a tree-decomposition of $T$. Clearly every bag has size at most 2, so the width of $(T', \mathcal{X})$ is at most 1. So the tree-width of $T$ is at most 1. And the tree-width of $T$ is at least 1 by Proposition 5. ■

**Proposition 7** *Every cycle on at least three vertices has tree-width at most 2.*

**Proof.** Let $C = v_1 v_2 ... v_n v_1$ be a cycle on $n \geq 3$ vertices. Let $T = t_1 t_2 ... t_{n-1}$ be a path. For every $i \in [n-1]$, define $X_{t_i} = \{v_i, v_n\}$. Then it is easy

to check (TD1)-(TD3) for $(T, \mathcal{X})$, so it is a tree-decomposition of width 2. Hence $C$ has tree-width at most 2. ■

In fact, it is not hard to show that every cycle on at least three vertices has tree-width exactly 2. Also, we know what graphs have large tree-width and what graphs have small tree-width. We will give more details about them later when we pay more attention on graph minors.