Graph algorithms lecture of Florent Foucaud (Master ICS, ISIMA)

Notes by Lucas Lorieau

December 2024

1 Dominating Set for Intervals

Suppose G is an interval graph with vertex set V corresponding to a set of intervals $\mathcal{I}$. We prove that the following algorithm solves Dominating Set in polynomial time on G:

- Sort intervals by increasing end time, *i.e.* $\mathcal{I} = \{I_i \mid i \leq n\}$. We denote by v_i the vertex of G associated to I_i.
- While there exist an interval not covered by the current solution, consider the first interval I_i that is not already covered. Pick in the solution the last interval in the considered ordering that covers I_i. (In the graph G, this corresponds to picking the last vertex adjacent to v_i).

Proof. Consider D^A the dominating set obtained by the algorithm above and D^* an optimal solution with the largest common prefix with D^A (with respect to the ordering considered in the algorithm). We build a new solution D^{**} that will be optimal and share a bigger prefix with D^A, thus creating a contradiction. Consider $I_k^* \neq I_k^A$ the first intervals different in both solutions. We define $D^{**} = D^* \setminus I_k^* \cup I_k^A$. This set of intervals is also a solution : indeed by definition of I_k^A, any interval that is not covered by the common prefix of both solution but covered by I_k^* is also covered by I_k^A. Furthermore, this new solution is of the size of D^*, thus it is an optimal one with a bigger common prefix with D^A. □

This proof technique based on maximum common prefix is very useful and can be applied to a variety of problems on interval graphs.

2 Trees and Treewidth

2.1 Independent Set on trees

Consider a r-rooted tree T of depth 1. Intuitively, choosing leaves for an independent set will be better than choosing only the root r (which is by definition adjacent to all other vertices of the tree). This intuition is the foundation of an algorithm based on dynamic programming that solves the problem on trees:

- $S = \emptyset$
- $M = \emptyset$
- While $M \cup S \neq V(T)$:

- Pick a leaf in $T \setminus (M \cup S)$ and add it to S
- Mark its parent by adding it to M

The proof for this algorithm is fairly simple, and will not be described here since the next algorithm solves a generalization of this problem.

Consider now the weighted version of Independent Set : for some weight function $w : V \to \mathbb{N}$, compute the independent set S of maximum weight in a given graph (here, we keep focusing on trees).

The following observation will be used to define a similar dynamic programming algorithm solving this generalised version of the algorithm. Consider some vertex v of the tree and denote by T_v the subtree rooted in v. Either it is a part of the solution, and none of its children will be in the solution, or it is not a part of the solution and its children may or not be in the solution. Based on this, we define two inductive functions that will be useful to define the algorithm:

$$i(v,1) = \text{maximum weight of an Independant Set of } T_v \text{ with } v \notin S$$
$$i(v,0) = \text{maximum weight of an Independant Set of } T_v$$

The goal of the algorithm is then to compute $v(r,0)$. To do so, we can consider the following rules:

- if v is a leaf:
  - $i(v,1) = 0$
  - $i(v,0) = w(v)$
- if v is not a leaf, let $v_1, \ldots, v_k$ be the children of v. Then
  - i(v,1) $= \sum_{j=1}^{k} i(v_j, 0)$
  - $i(v,0) = \max\left\{w(v) + \sum_{j=1}^{k} i(v_j,1),\ i(v,1)\right\}$

The algorithm is then fairly simple: we compute the two values $i(v,0)$ and $i(v,1)$ for each vertex in a bottom-up fashion, which yields an algorithm with complexity $\mathcal{O}(n)$.

Proof of correctness. By induction on the height of T_v:

- if v is a leaf, the base cases are clearly correct
- if v has children $v_1, \ldots, v_k$, by induction the values $i(v_j, \text{out})$ for any $j \in [k]$ and out $\in [0,1]$ are correct. Let us now prove the correctness of formula $i(v,1)$. In each subtree T_{v_j}, consider a maximum weight S_j with value $i(v_j,0)$. Construct $S = \bigcup_{j=1}^{k} S_j$ and remark that $w(S) = \sum_{j=1}^{k} i(v_j,0)$. This means that $i(v,1) \geq \sum_{j=0}^{k} i(v,1)$. On the other hand, consider now S^* an optimal independent set of T_v and consider the sets $S_j^* = S^* \cap T_{v_j}$. By iduction, $w(S_j^*) \leq i(v_i,0)$, so $w(S^*) = \sum_{j=1}^{k} w(S_j^*) \leq \sum_{j=1}^{k} i(v_j,0)$. The same type of reasoning can be applied to the second formula by constructing two independent set $S^1 = \{v\} \cup \bigcup_{j=1}^{k} S_j^1$ where S_j^1 is a maximum weight independent set of T_{v_j} not containing v_j and $S^2 = \bigcup_{j=1}^{k} S_j^2$ where S_j^2 is a maximum weight independent set of T_{v_j} to prove that $i(v,0) \geq \max\left\{w(S^1), w(S^2)\right\}$. The remaining inequality is proven by also taking an optimal independent set S^* of T_v and using a similar case analysis on whether v belongs to S^* or not. □

2.2 Tree decompositions and Tree Width

We have seen that Dynamic Programming can be very efficient to design algorithm on trees. A natural question that one can ask from the previous observation is the following : is it possible to adapt and generalize DP algorithms to graphs that are "similar" to trees? We will in the following describe some notions of "similarity to trees" and see how to adapt DP techniques with respect to said notions.

Definition 1 (k-tree)**.** *Let $k \geq 1$ be an integer.*

- *The complete graph K_k is a k-tree*
- *a k-tree can be obtained by another k-tree by selecting a k-clique C of G and by adding a new vertex adjacent to all vertices of C*

As a remark, 1-trees are exactly trees (thus the notion of "similarity" to trees). In fact, these types of graphs look like a "tree of cliques": by contracting the cliques that were selected during the inductive construction of the graph, one obtains a tree. Furthermore, every k-tree is chordal since it has a simplicial elimination scheme.

Definition 2 (partial k-tree)**.** *A graph is a partial k-tree if it is a subgraph of a k-tree.*

For instance, forests are partial 1-trees, cycles are partial 2-trees, and in general every graph of order n is a partial n-tree.

Definition 3 (Tree decomposition (Robertson-Seymour 1984))**.** *Let G be a graph. A* tree decomposition *of G is a couple $(T, \mathcal{B})$ composed of a tree T and a set of vertex bags $\mathcal{B}$ containing a bag X_v for each node $v \in V(T)$ so that:*

- *(every vertex of G belongs to some bag of $\mathcal{B}$)*
- *for every two vertices $a, b \in V(G)$ adjacent in G, there exists a node v of T so that a and b belong to X_v*
- *the set of nodes of T whose bag contains a given vertex a of G forms a connected subtree of T.*

Furthermore, we call the width *of a tree decomposition the size of the largest bag of the decomposition* minus one*. The* tree-width *of the graph is the minimum width of a tree decomposition of G.*

Few properties / theorems concerning tree decompositions:

Theorem 1. *G has treewidth at most k if and only if G is a partial k-tree.*

Proof sketch. If G is a partial k-tree, we get the tree decomposition by considering the order of construction of the graph.

For the converse, one can use induction on the size of the tree decomposition, remove a leaf node, and proceed. In the k-tree of which G is a subgraph, every bag is a clique. □

Proposition 1. *Every clique of a graph G is contained in a bag of any tree decomposition of G.*

Proof sketch. Take three vertices forming a triangle, and look at their corresponding subtrees of the tree decomposition T. One can check that the three subtrees must intersect in one node of T (this property is called Helly's property for subtrees of a tree). By induction, all the subtrees corresponding to the vertices of a clique can be shown to intersect in a same node of T: its bag contains the clique. □

2.3 Dynamic programming on a tree decomposition

Given a tree decomposition of a graph G of width w (bags of size at most $w+1$), the goal is to design an algorithm to solve the independent set problem on general graphs using a similar technique to the one used for trees. We will obtain through the dynamic programming technique an FPT algorithm (algorithm with complexity $\mathcal{O}(f(w)\cdot n^c)$ for f an increasing function and c a constant), *i.e.* an algorithm that is polynomial if we bound a certain parameter of the input, here the width of the decomposition.

The idea is to use the tree structure given by the tree decomposition of G to obtain such an algorithm. Let us root the tree of the decomposition of G in some node r. Consider a node v of the tree decomposition $\mathcal{T}$ and let us denote by G_v the subgraph of G induced by bags associated with the subtree of $\mathcal{T}$ rooted in v. We will apply a dynamic procedure depending on the current bag X_v and a $S \subset X_v$ a subset of the current bag: we will compute the best independent set of G_v containing the vertices of S.

Let us define $i(v,S)$ = largest size of an independent set of G_v so that the intersection of this independent set with X_v is exactly S. By convention, if S is not an independent set, then $i(v,S) = -\infty$. Then, the goal of the algorithm is to compute $\max_{S\subseteq X_r}\{i(r,S)\}$. To do so, we use the following rules:

- if v is a leaf node of $\mathcal{T}$, then $i(v,S) = |S|$ if S is a independent set and $-\infty$ otherwise by convention.
- if v is an internal node with children $v_1,\dots,v_k$:

$$i(v,S) = |S| + \sum_{j=1}^{k} \max_{\substack{S'\subseteq X_{v_j} \\ S'\cap X_v = S\cap X_{v_j}}} \{i(v_j,S') - |S\cap S'|\}$$

  Running time: computing $i(v,S)$ for any subset S of X_v takes $\mathcal{O}(2^{w+1}\cdot|V(T)|) = \mathcal{O}(2^w\cdot n)$ so total running time is $\mathcal{O}(2^w\cdot n^2)$.

  Proof of correctness. We will use the same type of proof that was used for the algorithm on trees.

  - For leaves, the correctness is direct
  - By induction on the height of the considered tree decomposition, suppose the root v currently considered has children $v_1,\dots,v_k$. As for the algorithm on trees, we will show that $i(v,S) \geq |S| + \sum_{j=1}^{k} \max_{\substack{S'\subseteq X_{v_j} \\ S'\cap X_v = S\cap X_{v_j}}} \{i(v_j,S') - |S\cap S'|\}$ first, and then prove the other inequality to obtain the equality.
    - The "at most" part is as before a constructive part. Let S_j^* be an optimal independent set of G_{v_j} so that $S_j^*\cap X_v = S\cap X_{v_j}$. By induction, $|S_j^*| = i(v_j, S_j^*\cap X_{v_j})$. Take $S^* = \bigcup_{j=1}^{k} S_j^*$: it is an independent set (because X_v is a separator in T and all S_j^*'s agree on X_v) of size

$$\begin{aligned}
|S^*| &= |S| + \sum_{j=1}^{k} (|S_j^*| - |S_j^*\cap S|) \\
&= |S| + \sum_{j=1}^{k} (i(v_j, S_j^*\cap X_{v_j}) - |S_j^*\cap S|) \\
&= |S| + \sum_{j=1}^{k} \max_{\substack{S'\subseteq X_{v_j} \\ S'\cap X_v = S\cap X_{v_j}}} \{i(v_j,S') - |S\cap S'|\}
\end{aligned}$$

Because S^* is an independent set of G_v with $S^* \cap X_v = S$, we have that $i(v,S) \geq |S^*| = |S| + \sum_{j=1}^{k} \max_{\substack{S' \subseteq X_{v_j} \\ S' \cap X_v = S \cap X_{v_j}}} \{i(v_j, S') - |S \cap S'|\}$.

* The "at least" part is proven by taking an optimal independent set $S^*_{G_v}$ of G_v so that $S^*_{G_v} \cap X_v = S$. Consider then the intersections of this independent set with the graphs G_{v_j} and bound those as it was made in the proof for the algorithm on trees. □