CS762: Graph-Theoretic Algorithms
Lecture 13: Trees and Friends
February 4, 2002

Scribe: Josh Lessard

Abstract

In this lecture we examine a special class of graphs called trees. Nearly all graph theory problems that are NP-hard in general can be solved efficiently when the graph in question is a tree. One such problem will be presented. Trees will then be generalized to k-trees.

1 Introduction

Computer scientists and mathematicians have spent much of the last few decades studying NP-hard problems, that is, problems for which a polynomial time algorithm is unlikely to exist. In particular, NP-hard graph theory problems such as Vertex Cover, Independent Set, and Maximum Cut have received much attention.

This paper examines a special class of graphs called trees. Most NP-hard problems can actually be solved efficiently when the graph in question is a tree. Specifically, almost any problem in a tree can be solved using dynamic programming.

We begin by introducing the necessary background material, including definitions for many of the aforementioned terms. We then present an algorithm for efficiently solving the Vertex Cover problem in a tree. Finally, we introduce the terms tree-decomposition, treewidth, k-tree, and partial k-tree, and prove that all trees are in fact 1-trees, that is, k-trees where $k = 1$.

2 Definitions

Definition 1 *A graph G is a tree if it is acyclic and connected.*

Definition 2 *A rooted tree is a tree in which one vertex is distinguished from the others. The distinguished vertex is called the root of the tree, and all other vertices become descendants of the root.*

Definition 3 *Dynamic programming is a recursive problem solving method in which an overall solution is obtained by combining the solutions to subproblems, which can be characterized with a constant number of parameters.*

Definition 4 *A vertex cover of an undirected graph $G = (V,E)$ is a subset $V' \subseteq V$ such that if $(u,v) \in E$, then $u \in V'$ or $v \in V'$ (or both).*

3 Dynamic Programming Algorithm for Vertex Cover in a Tree

In this section, an algorithm for finding a vertex cover in a tree is presented. Vertex Cover is, in general, an NP-hard problem. [CLRS01] However, this algorithm uses dynamic programming to find the vertex cover in linear time.

3.1 Preamble

Let T be a tree rooted at some vertex r.
For any $v \in V$, let T_v be the subtree of T rooted at v (see Figure 1).

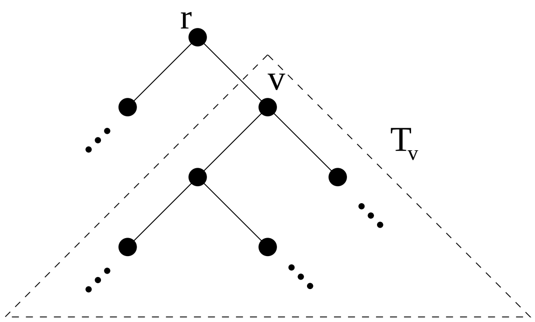

Figure 1: A tree rooted at r containing a subtree T_v rooted at v.

Define two functions:

- $f^+(v)$ = size of the smallest vertex cover in T_v that includes v
- $f^-(v)$ = size of the smallest vertex cover in T_v that excludes v

We want to find the smallest vertex cover in the tree. Keeping the above definitions in mind, we have two cases:

1. the smallest vertex cover contains the root, or
2. the smallest vertex cover does not contain the root.

Formally then, we want to compute $\min\{f^-(r), f^+(r)\}$.

3.2 Function Definition

The functions $f^-(v)$ and $f^+(v)$ are defined recursively, as is normally the case when using a dynamic programming algorithm.

Recursive definitions:

$$\begin{aligned} f^-(v) &= \sum_{w\ child\ of\ v} f^+(w) \\ f^+(v) &= 1 + \sum_{w\ child\ of\ v} min\{f^-(w), f^+(w)\} \end{aligned}$$

In the case of $f^-(v)$, v is not in the vertex cover so all its children must be in order to cover all edges incident to v. Hence, for v's subtrees, only vertex covers which include v's children are considered. For this reason, we must use $f^+(w)$ when making the recursive call.

However, in the case of $f^+(v)$, v *is* in the vertex cover. Thus, it makes no difference if v's children are included in the vertex cover or not; we simply want the smallest possible set. For this reason, both $f^-(w)$ *and* $f^+(w)$ must be considered when making the recursive call. Obviously, the function which returns the smallest value is used.

3.3 Algorithm

The algorithm itself is extremely simple given the above definitions. It is simply a matter of using a post order traversal of the tree to compute the desired function value. This means that for all vertices v in the tree, the value of the function is computed for the subtrees rooted at v's children before it is computed for v itself (ie a bottom-up approach).

```
for all vertices v in the tree T in post order {

   compute f+(v), f-(v) from the values of v's children

}

output min{f+(r), f-(r)}
```

Note that this algorithm only computes the size of the smallest vertex cover. Actually finding the vertex cover can be done with standard dynamic programming techniques.

3.4 Complexity Analysis

All vertices $v \in V$ must be kept in memory because we need to know who v's children are at each recursive call. For this reason, the cardinality of V is included in the complexity analysis formula. Similarly, for all vertices $v \in V$, $deg(v)$ recursive calls are made, which is why the summation is included as the second term in the formula. Simplifying the equation shows that this algorithm is linear in both time and space.

$$\begin{aligned} & O(\mid V \mid + \sum_{v \in V} deg(v)) \\ = \; & O(n + 2m) \\ = \; & O(m + n) \end{aligned}$$

The effectiveness of this algorithm is not limited to the Vertex Cover problem. The concept presented above also works for many other NP-hard problems (such as Independent Set) when the graph in question is a tree. Simply change the definitions of $f^+(v)$ and $f^-(v)$ so that the desired information is extracted from the graph.

The concept behind this algorithm is that a tree can be broken into smaller parts with only one vertex in common (see Figure 2 below).

As shown in the figure, there are no edges between the subtrees A and B. This means that a partial solution for subtree B does not overlap or have any effect on a partial solution for subtree

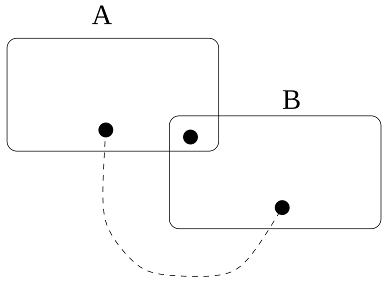

Figure 2: Subgraphs with only one vertex in common.

A. Thus, a divide-and-conquer method can be used in which solutions for subtrees can be "pasted" into the overall solution.

4 Expanding The Concept: k-Trees

We have considered the case where the intersection between two subgraphs contains one vertex. We now focus our attention on subgraphs whose intersections have $\leq k$ vertices. This means that 2^k functions are necessary, and this will typically lead to $O(2^k \cdot n)$ algorithms.

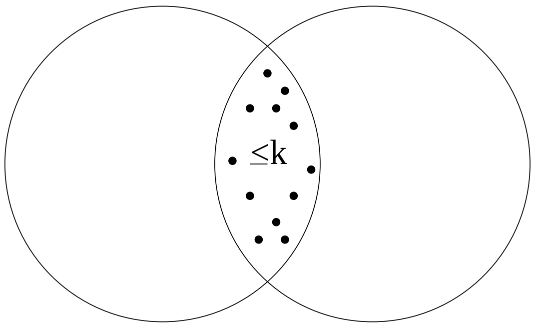

Figure 3: Subgraphs with $\leq$ k vertices in common.

To formalize this definition, we use the concept of a tree-decomposition.

4.1 Tree-Decompositions

Definition 5 *A tree-decomposition of a graph $G = (V, E)$ is a tree $T = (I, F)$ where each node $i \in I$ has a label $X_i \subseteq V$ such that:*

- $\bigcup_{i \in I} X_i = V$
- *for any edge (v, w), there exists an $i \in I$ with $v, w \in X_i$*
- *for any $i, k \in I$: if $v \in X_i$ and $v \in X_k$, then also $v \in X_j$ for any j on the path from i to k (the subtree containing j must be connected)*

Definition 6 *Given a tree-decomposition T=(I,F), an edge (u,v) is "allowed" if there exists an $i \in I$ such that $u, v \in X_i$.*

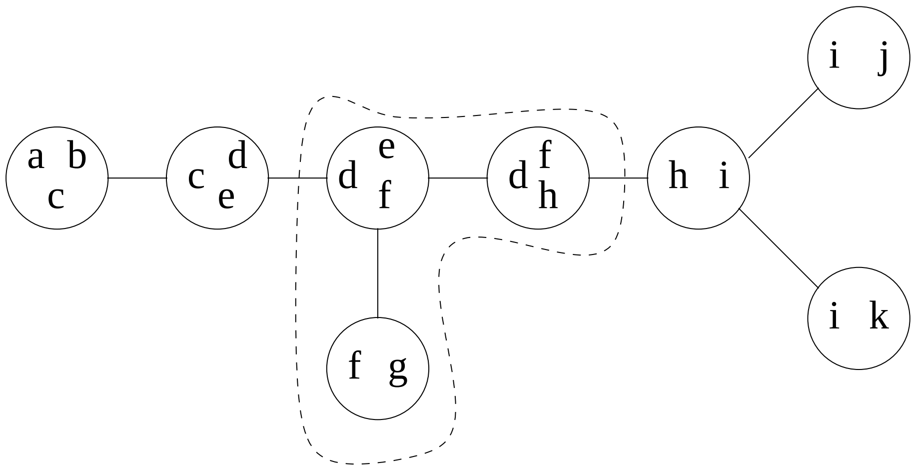

Figure 4: A tree-decomposition. Notice the subtree containing f is connected.

Given a tree-decomposition T, it is trivial to construct a graph of "allowed edges" (see Figure 5). Notice that a graph of all allowed edges of a tree-decomposition is chordal since it is the intersection graph of subtrees of a tree (refer to the lecture notes on subtree intersection graphs from January 21, 2002).

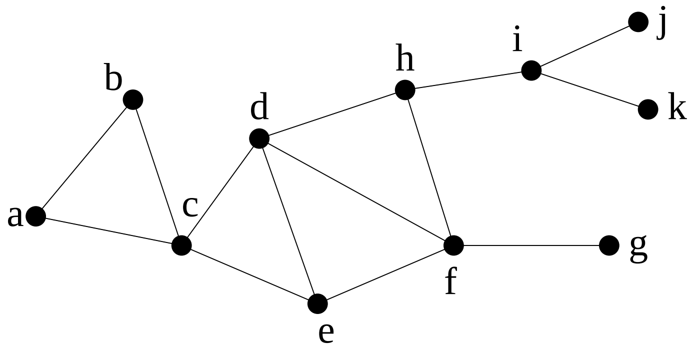

Figure 5: Allowed edges of the tree-decomposition in Figure 4.

4.2 Treewidth

Definition 7 *The treewidth of a tree-decomposition is* $\max_{i \in I} \mid X_i \mid -1$.

Definition 8 *The treewidth of a graph G is the minimum treewidth of a tree-decomposition of G.*

Equivalently, the treewidth of a graph G can be defined as the smallest k such that G admits a tree-decomposition into vertices with labels no bigger than $k+1$. [Die90]

4.3 k-trees

Definition 9 *A graph G is called a k-tree if G has a perfect elimination order such that indeg(v_i) = k for all i = k+1,...,n.*

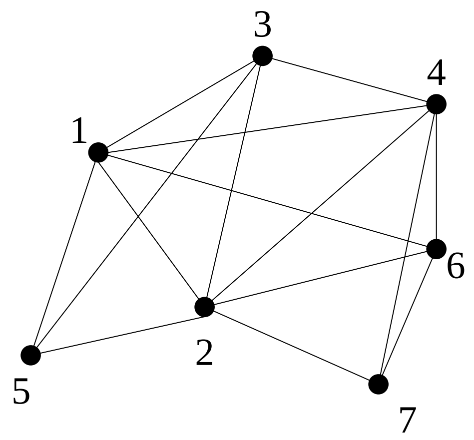

Figure 6: A k-tree with k = 3.

In any k-tree, $\{v_1, \ldots, v_k\}$ is a clique, and v_i for i $\geq$ k+1 is adjacent to a k-clique in G$[v_1, \ldots, v_{i-1}]$ (see Figure 6 for an example of a 3-tree). To illustrate this example, we give an easy observation.

Theorem 1 *Every tree is a 1-tree.*

Proof: Let $G = (V, E)$ be a tree. Choose an arbitrary vertex $r \in V$ and root G at r. Label all vertices in G in breadth-first order. The root r now has a label v_1, and every $v \in V - \{r\}$ has a label in {2,. . .,n}. We claim that $\{v_1, \ldots, v_n\}$ is a perfect elimination order with $indeg(v_i) = 1$ for all i = 2,. . .,n.

Every $v \in V - \{r\}$ has exactly one parent and an arbitrary number of children. Since the vertices were labelled in breadth-first order, the label on v's parent is smaller than v's label. Conversely, the labels of v's children (if any) are larger than v's label. This means that every $v \in V - \{r\}$ has exactly one predecessor. Therefore $v \cup Pred(v)$ is a clique and the order is a perfect elimination order.

As we just showed, $indeg(v_i) = 1$ for all $i = 2$,. . .,n, so all conditions of Definition 9 are satisfied. □

4.4 Number of Edges in a k-Tree

In this subsection, we will derive a simple formula for the number of edges in a k-tree. In order to understand the derivation, it is useful to illustrate how a k-tree with n vertices is constructed, and thus an algorithm for drawing one will now be outlined.

1. Draw a k-clique and label the vertices $v_1, \ldots, v_k$. These first k vertices in the perfect elimination order MUST form a clique since they will all be predecessors of the $k + 1^{st}$ vertex.

2. Draw a vertex and label it v_{max+1}.

3. Draw an edge between this new vertex and k other vertices that form a clique.

4. Repeat steps 2 and 3 until there are n vertices in the graph.

Notice that step 3 adds k edges to the graph during every iteration, and that step 3 is performed $n-k$ times. Therefore, the number of edges in a k-tree with n vertices is the number of edges in a k-clique, plus k edges for each of the other $n-k$ vertices.

$$\begin{aligned} number\ of\ edges\ in\ a\ k-tree &= \frac{k(k-1)}{2} + (n-k)\cdot k \\ &= k\cdot n - \frac{k(k+1)}{2} \end{aligned}$$

Notice that this total is linear if k is constant.

4.5 Partial k-trees

Definition 10 *A graph G is a partial k-tree if a k-tree can be obtained by adding edges to G.*

Note that unlike a k-tree, a partial k-tree is not necessarily a chordal graph. For example, C_4 is a partial 3-tree (add edges until it is K_4, which is a 3-tree), but it is not chordal.

References

[CLRS01] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition.* McGraw-Hill Book Company, 2001.

[Die90] Reinhard Diestel. *Graph Decompositions: A Study in Infinite Graph Theory.* Oxford Science Publications, 1990.