CS173 Lecture B, November 17, 2015

Tandy Warnow

November 17, 2015

CS 173, Lecture B
November 17, 2015
Tandy Warnow

Today's material

- Adjacency matrices and adjacency lists (and a class exercise)
- Depth-first and Breadth-first search trees (and a class exercise)
- Two theorems about trees with their proofs (comment about induction on trees)
- More theorems about trees (no proofs)
- Minimum spanning tree algorithms (and a class exercise)

Adjacency Matrices

We can represent a graph $G = (V, E)$, where $V = \{v_1, v_2, \ldots, v_n\}$, in several ways. Here we describe one of the two most popular methods, **adjacency matrices**.

Adjacency Matrices for simple undirected graphs

We define a matrix M where

- $M[i,j] = 1$ if $(v_i, v_j) \in E$
- $M[i,j] = 0$ if $(v_i, v_j) \notin E$

Note $M[i,i] = 0$ for all $i = 1, 2, \ldots, n$, and $M[i,j] = M[j,i]$ for all i, j.

Adjacency Matrices of Directed Graphs

For directed graphs, we distinguish between edges from v_i to v_j and from v_j to v_i; hence, we can get asymmetric matrices.

Adjacency Matrices

Note that this representation inherently requires $\Theta(n^2)$ space, even for graphs that don't have many edges. But checking if an edge is present is $O(1)$ time.
See https://en.wikipedia.org/wiki/Adjacency_matrix.

Adjacency Lists

An alternative representation of graphs that is sometimes more useful is an Adjacency List.
In an adjacency list, for each vertex in the graph, you have a list of its neighbors. If the graph is undirected, the list for vertex v is the set of all x such that $(v, x) \in E$. For directed graphs, you list only those x such that $v \rightarrow x \in E$.
Note that an adjacency list requires $\Theta(m)$ space, where m is the number of edges. This can be much more space efficient than an adjacency matrix for sparse graphs, but some operations take more time (e.g., checking if an edge is present).
See https://en.wikipedia.org/wiki/Adjacency_list.

Adjacency Matrices and Lists

Depth-first and breadth-first search

Depth-first search (DFS) and breadth-first search (BFS) are two ways of exploring a connected component of a graph.
Each exploration can produce a **tree** that spans the connected component.
You'll learn about this in later classes... here you should just get the overall idea.

Breadth-First Search (BFS)

Trees: connected acyclic graphs

More about trees

- Every tree $T = (V, E)$ with at least two vertices has at least two nodes that have degree 1 (hint: consider a longest path in the tree)
- If a tree $T = (V, E)$ has n vertices, then it has $n - 1$ edges

Every tree with at least two vertices has at least two leaves

The **leaves** of a tree are the nodes with degree 1; all other nodes are **internal nodes**.
Theorem: Every tree T with at least two vertices has at least two leaves.
Proof: Consider a longest path P in T.

Since T is finite, the path begins at some node v and ends at some node w.

We will prove that both endpoints of P are leaves.

Proving every tree has at least two leaves

P is a longest path in a tree T; we prove its endpoints are leaves.

Suppose v is not a leaf; then v has at least two neighbors, x and y, and one of them (say x) is is not in P. (Otherwise we have a cycle.)

Let P' be the path that begins at x followed by P. This is a longer path than P, contradicting our assumption.

Every tree with n vertices has exactly $n-1$ edges

Theorem: Every tree with n vertices has exactly $n-1$ edges.
Proof: By induction on n.
Base case: If $n = 1$, then T has no edges, and the base case holds.

The inductive hypothesis is that $\exists K \geq 1$ such that for all $n, 1 \leq n \leq K$, if tree T has n vertices then T has $n-1$ edges.

Now assume T has $K+1 \geq 2$ vertices; we want to prove T has K edges.

Proof that every tree with n vertices has $n-1$ edges

Since T is a tree, T has at least two leaves.

Let v be a leaf in T, and let w be its single neighbor.

Let T' be the graph created by deleting v.

Note that T' is a tree with K vertices, because:

- T' has one less vertex than T.
- T' is connected and acyclic

By the inductive hypothesis, T' has $K-1$ edges.

Recall that T' has one less edge than T.

Hence T has K edges. (q.e.d.)

Important: We started with a tree on $K+1$ vertices and removed a leaf to get a tree on K vertices. We did not go the reverse direction!

More about trees

What NP-hard problems can we solve efficiently on trees?

- Chromatic number?
- Max Clique?
- Maximum Independent Set?
- Minimum Dominating Set?
- Minimum Vertex Cover?

Some results on trees

- Chromatic number of any tree is at most 2.
- The max clique size of any tree is at most 2.
- If X is a maximum independent set in T, we can assume X contains all the leaves of T. (Why?)
- If X is a minimum vertex cover in T and T has more than two vertices, we can assume X does not contain any leaves. (Why?)

Class exercise

Do one or more of the following:

1. Make adjacency matrix and adjacency list for the wheel graph W_4 on vertices v_1, v_2, v_3, v_4, x, with x being the node in the middle.

2. Write down a BFS tree for the wheel graph in Problem 1, starting at x.
3. Write down a BFS tree for the wheel graph in Problem 1, starting at v_1.
4. Figure out why every tree can be properly 2-colored.
5. Figure out why every tree with at least 3 vertices has a minimum dominating set that does not contain any leaves.

Spanning Trees

A **spanning tree** of a connected graph $G = (V, E)$ is a subgraph that includes all the vertices and is a tree.

Minimum Spanning Trees (MST)

- Input: Connected graph $G = (V, E)$ and positive edge weights $w : E \rightarrow Z^+$
- Output: Spanning tree $T = (V, E')$ of G that has minimum cost, where $cost(T) = \Sigma_{e \in E'} w(e)$

Finding MSTs

Try one of the greedy algorithms on the complete bipartite graph $K_{3,5}$ with $w(v_i, w_j) = i + j$:

- Keep adding the least weight edges (don't include those that create cycles) - Kruskal's algorithm
- Keep deleting the most costly edges (don't delete bridges)
- Grow a spanning tree, adding least costly edge to an unvisited vertex - Prim's algorithm

Finding MSTs

Running Kruskal's algorithm on $K_{3,5}$ with weight $w(v_i, w_j) = i + j$:

Finding MSTs

Why did that method work?