Algorithms in Systems Engineering
IE170

Lecture 13

Dr. Ted Ralphs

References for Today's Lecture

- Required reading
  - CLRS Chapter 21 and 22
- References
  - R. Sedgewick, *Algorithms in C++* (Third Edition), 1998.

Connectivity Relations

- So far, we have only considered sets of items that are related to each other through some kind of ordering (if at all).
- In other words, two items x and y are only related by their relative positions in the ordered list.
- We will now generalize this idea by considering additional *connectivity relationships* between items.
- To do so, we will specify that there is a direct link between certain pairs of items.
- This will allow us to ask questions such as

Graphs

- A *graph* is an abstract object used to model such connectivity relations.
- A graph consists of a list of items, along with a set of connections between the items.
- The study of such graphs and their properties, called *graph theory*, is hundreds of years old.
- Graphs can be visualized easily by creating a physical manifestation.
- There are several variations on this theme.
  - The connections in the graph may or may not have an *orientation* or a *direction*.
  - We may not allow more than one connection between a pair of items.
  - We may not allow an item to be connected to itself.
- For now, we consider graphs that are
  - *undirected*, i.e., the connections do not have an orientation, and
  - *simple*, i.e., we allow only one connection between each pair of items and no connections from an item to itself.

Applications of Graphs

Graph Terminology and Notation

- In an undirected graph, the "items" are usually called *vertices* (sometimes also called *nodes*).
- The set of vertices is denoted V and the vertices are indexed from 0 to $n-1$, where $n = |V|$.
- The connections between the vertices are unordered pairs called *edges*.
- The set of edges is denoted E and $m = |E| \leq n(n-1)/2$.
- An undirected graph $G = (V, E)$ is then composed of a set of vertices V and a set of edges $E \subseteq V \times V$.
- If $e = \{i, j\} \in E$, then
  - i and j are called the *endpoints* of e,
  - e is said to be *incident* to i and j, and
  - i and j are said to be *adjacent* vertices.

More Terminology

- Let $G = (V, E)$ be an undirected graph.
- A *subgraph* of G is a graph composed of an edge set $E' \subseteq E$ along with all incident vertices.
- A subset V' of V, along with all incident edges is called an *induced subgraph*.
- A *path* in G is a sequence of vertices such that each vertex is adjacent to the vertex preceding it in the sequence.
- A path is *simple* if no vertex occurs more than once in the sequence.
- A *cycle* is a path that is simple except that the first and last vertices are the same.
- A *tour* is a cycle that includes all the vertices.

Connectivity in Graphs

- An undirected graph is said to be *connected* if there is a path between any two vertices in the graph.
- A graph that is not connected consists of a set of *connected components* that are the maximal connected subgraphs.
- Given a graph, one of the most basic questions one can ask is whether vertices i and j are in the same component.
- In other words, is there a path from i to j?
- Such questions might arise in the design of a network or circuit.
- They may not be that easy to answer!
- One approach is to use a data structure for storing *disjoint sets*.

Disjoint Set Data Structure

- A disjoint set data structure is used to store a set of items that consists of a number of disjoint subsets.
- There are two basic operations we'd like to be able to perform.

- This type of data structure is sometimes called a *union-find* data structure.
- How could we use such a data structure to determine the connected components of a graph?

Connected Components Algorithm

- Suppose the graph is specified simply as a a list of edges.
- Algorithm
  - Start with each vertex in its own subset.
  - While there are still edges left on the list,

- After reading in all the edges, a call to `find(i, j)` will determine whether `i` and `j` are in the same connected component.
- The advantage of this method is that we never have to actually store the list of edges.
- We will also consider more efficient methods that require storing the edges.

Quick Find Implementation of Union-Find

- The simplest implementation involves an array of length n.
- We will maintain the array such that two items are in the same subset if and only if the array entries are equal.
- This makes the `find(i, j)` constant time, so we call this implementation *quick find*.
- How do we implement `union(i, j)`?
- What is the running time?
- Note that this could also be implemented using linked lists, as described in CLRS.

Quick Union Implementation of Union-Find

- To speed up the union operation, we maintain the array in a different fashion.
- We will consider the i^{th} entry of the array to be a pointer to some other item.
- We start with the i^{th} entry of the array pointing to item i, i.e., all items pointing to themselves.
- To perform `find(i, j)`,
  - Follow the pointers from nodes i and j until reaching a node that points to itself, called the *representative*
  - If the same representative is reached from both nodes i and j, then they are in the same subset.
- To perform `union(i, j)`, perform the find operation and then point the representative for i to the representative for j.
- What is the performance now?

Weighted Quick Union

- Note that the quick union algorithm essentially builds a tree out of the nodes in each component, with the root being the representative.
- As in binary search, the running time of the find operation depends on the depth of the trees.
- Each union operation essentially connects two trees together by pointing the root of one tree to the root of the other.
- One way to limit the depth of the tree is to always point the smaller tree to the larger one.
- This ensures that each find takes less than $\lg n$ steps.
- Note that we must now keep track of the number of nodes in each tree, but that's easy to do.
- Another approach is to keep track of the height of each tree and always point the shorter tree to the taller one.

Path Compression

- Ideally, we would like each item to point directly to the representative of its subset.
- One possibility is to simply keep track of all the nodes encountered in the path to the root.
- After reaching the root, set all the nodes on the path to point to the root.
- This is easy to implement recursively and doesn't change the asymptotic running time.
- An easier method to implement is *compression by halving*, which is setting each node to point to its grandparent.
- Combining weighted quick union with path compression yields a total running time for connected components of approximately $O(m)$.