Other Graph Topics

CS 311 Data Structures and Algorithms
Lecture Slides
Friday, December 11, 2009

Glenn G. Chappell
Department of Computer Science
University of Alaska Fairbanks
CHAPPELLG@member.ams.org

© 2005–2009 Glenn G. Chappell

The Rest of the Course
Overview

Two Topics

- ✓ External Data
  - Throughout this semester, we have dealt-with data stored in memory.
  - What if we store data on an external device, accessed via a (relatively) slow connection. How does this change the design of algorithms and data structures?
- (part) Graph Algorithms
  - A **graph** is a way of modeling relationships between pairs of objects.
  - This is a very general notion; thus, algorithms for graphs often have very general applicability.

Review
Introduction to Graphs [1/3]

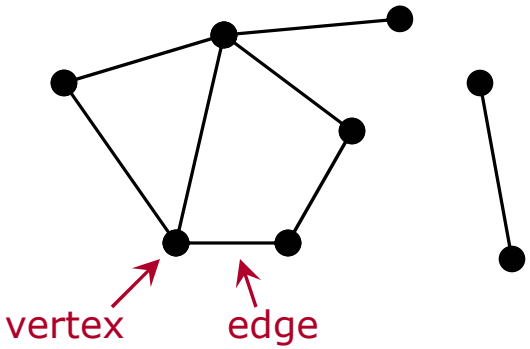
A **graph** is consists of **vertices** and **edges**. An edge joins two vertices.

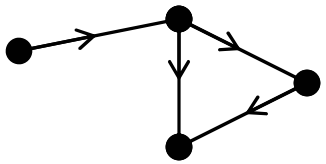
- An example of a graph is shown at right.

Sometimes we give each edge a direction.

- The result is a **directed graph** or **digraph.**

Graphs represent situations in which objects are related in pairs.





Review
Introduction to Graphs [2/3]

We use graphs to model:

- Networks
  - Vertices are nodes in network; edges are connections.
  - Examples
    - Communication
    - Transportation
    - Electrical
    - Web (edges are links)
- State Spaces
  - Vertices are states; edges are transitions between states.
  - See CS 451 for more info.
- Generally, situations in which objects are related in pairs:
  - Vertices are people, edges indicate relationships (friendship? common work?)
  - Vertices are events at a conference; edges join events that cannot be held simultaneously.
  - Vertices are data structure nodes; (directed) edges indicate (owning?) pointers.

Review
Introduction to Graphs [3/3]

Two common ways to represent graphs:

- **Adjacency matrix**. 2-D array of Boolean values. Entry (i, j) answers the question "Does edge (i, j) exist?"
  - Answer "does edge (i, j) exist?" in $O(1)$.
  - Finding all neighbors of a vertex can be slow for large, sparse graphs.
    - A **sparse** graph is one with relatively few edges.
  - Space used: $O(n^2)$, where n = number of vertices.
    - Note: For graphs in general, we cannot do better than this.
- **Adjacency list**. Array of lists. Entry i is a list of the neighbors of vertex i.
  - Answer "does edge (i, j) exist?" in $O(n)$.
  - Finding all neighbors of a vertex is fast.
  - Much better space usage for large, sparse graphs.

Both of these can be tweaked in obvious ways to handle digraphs.

Other graph representations are used.

Review
Spanning Trees — Introduction

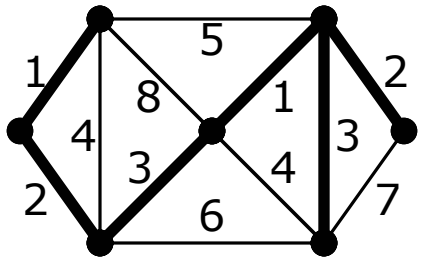
A **tree** is a graph that:

- Is **connected** (all one piece).
- Has no **cycles**.

Given a graph *G*, a **spanning tree** in *G* is a tree that:

- Uses only vertices and edges of *G*.
- Uses *all* vertices of *G*.

Fact. Every connected graph has a spanning tree.



An important problem is, given a weighted graph (weights on the edges), find a **minimum spanning tree** (a spanning tree of minimum total weight).

- There are several nice algorithms that solve this problem.

Review
Spanning Trees — Prim's Algorithm [1/4]

Given a connected weighted graph (weights on the edges), we can find a minimum spanning tree using a **greedy** algorithm.

Improve current state as much as possible at each step.

Here is one greedy algorithm for finding a minimum-weight spanning tree.

- Start with one vertex that we can reach.
- Repeatedly add the lowest weight edge joining a vertex we can reach to one we cannot (and now we can reach its other endpoint).

This is called **Prim's Algorithm**.

- R.C. Prim 1957 (also V. Jarnik 1930 & E. Dijkstra 1959).

Spanning Trees — Prim's Algorithm [2/4]

Prim's Algorithm more formally:

- Given: Connected graph with weights on the edges.
- Returns: Edge set of a minimum spanning tree.
- Procedure:
  - Mark all vertices as not-reached.
  - Set edge set of spanning tree to empty.
  - Choose a vertex. Mark it as reached.
  - Repeat while there exist not-reached vertices:
    - Find lowest weight edge joining a reached vertex to a not-reached vertex.
    - Add this edge to the edge set.
    - Mark the not-reached endpoint of this edge as reached.
  - Return edge set.

Review
Spanning Trees — Prim's Algorithm [3/4]

How can we specify a weighted graph (in a program)?

- Use something like an adjacency matrix, but instead of storing 0/1, store weights. If necessary, also have a value meaning "no edge".

How can we efficiently find the lowest cost edge between reached and a not-reached vertices?

- Use a Priority Queue.
- One way to do it:
  - PQ holds edges that, *at some point*, joined reached & not-reached vertices.
  - When getting an edge from the PQ, check to be sure it joins reached & not-reached vertices. If not, skip it.
  - When marking a vertex as reached, insert into the PQ all edges from this vertex to not-reached vertices.
  - When the PQ is empty, quit.

Spanning Trees — Prim's Algorithm [4/4]

TO DO

- Implement Prim's Algorithm.

Done. See **`prim.cpp`**, *on the web page.*

Note: Prim's Algorithm can be implemented somewhat more efficiently.

- Use adjacency lists instead of an adjacency matrix. Inserting in the PQ would become faster.
- Put vertices in the PQ, instead of edges, and be intelligent about figuring out which edge to use to get to that vertex.
- Use a Fibonacci Heap instead of a Binary Heap. A **Fibonacci Heap** is a data structure much like a Binary Heap, but: insert is faster [$O(1)$] at the expense of delete becoming *amortized* $O(\log n)$.

Spanning Trees — Other Algorithms

Kruskal's Algorithm (J. Kruskal 1956)

- Procedure: Repeatedly add the lowest weight edge joining two vertices that cannot be reached *from each other*.
- Need some way of determining efficiently whether a vertex can be reached from another vertex. (Coming soon!)

Reverse-Delete Algorithm (J. Kruskal 1956)

- Procedure: Start with all edges. Repeatedly remove the *highest* weight edge joining two vertices that *can* be reached from each other.
- (Same need as above.)

Other Graph Topics
Union-Find [1/6]

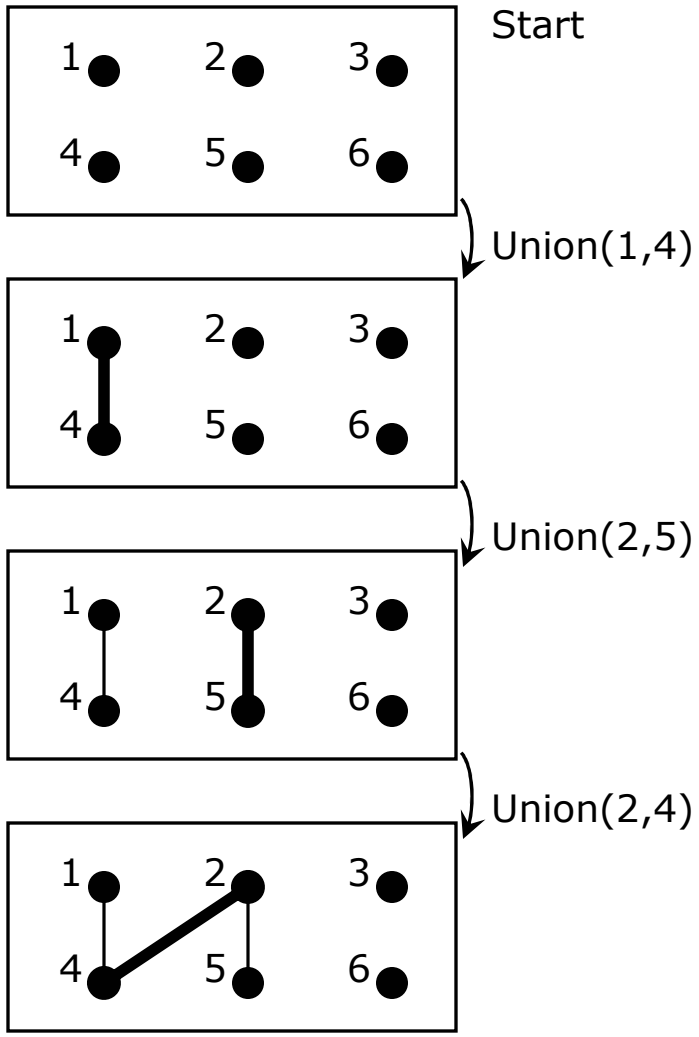
Some graph problems do not require a fully general graph representation.

One such example is the **Union-Find Problem**.

- We are given a set of vertices.
- We can do the following operations:
  - Union
    - Add an edge between two vertices.
  - Find
    - Given two vertices, determine whether there is a path from one to the other, using existing edges.

This is another ADT (right?).

Also, it is not really about graphs …



Other Graph Topics
Union-Find [2/6]

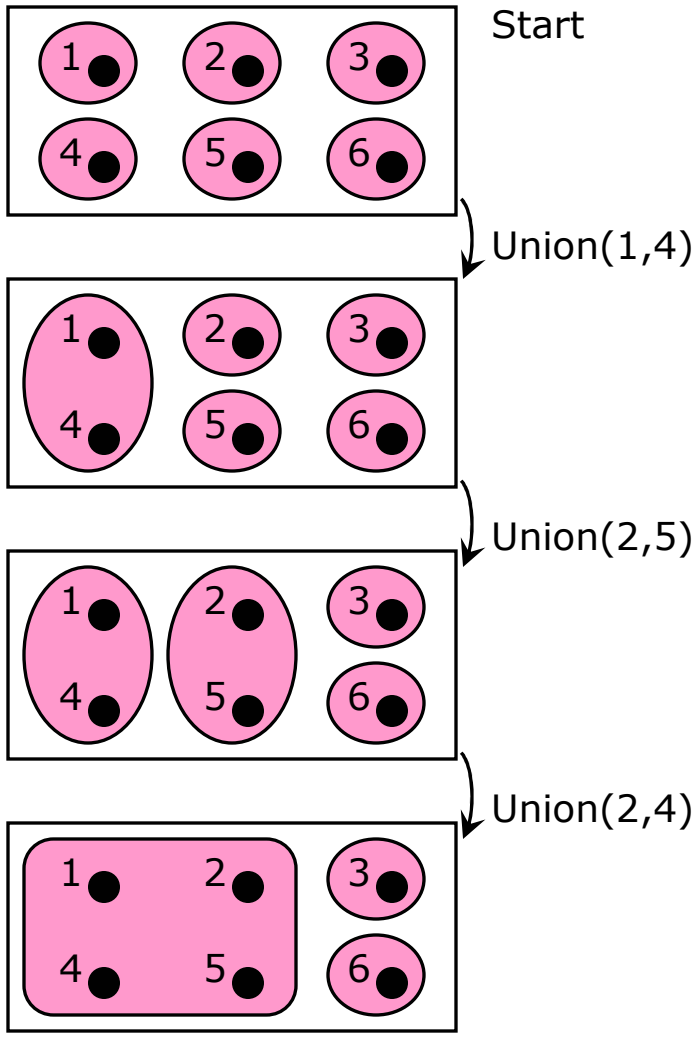
This is not really about graphs; it is about "blobs"*.

- Each vertex is in some blob.
- **Union** joins two blobs into one.
- **Find** determines whether two given vertices lie in the same blob.

Since all we care about are blobs, we do not actually need to keep track of edges.

- But then how do we determine whether two vertices lie in the same blob?

*In practice, blobs are called **sets**.



Union-Find [3/6]

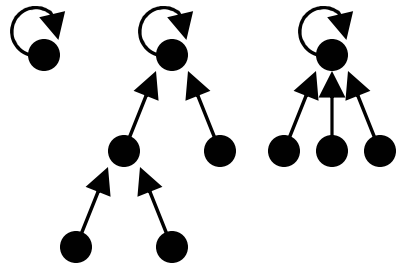
We usually do not deal with the Union-Find problem using a graph.

Rather, we use some kind of **disjoint-set structure**.

- Also called a *union-find structure* or *find-merge structure*.

The most efficient known is a **Disjoint-Set Forest** (B. Galler & M. Fischer 1964).

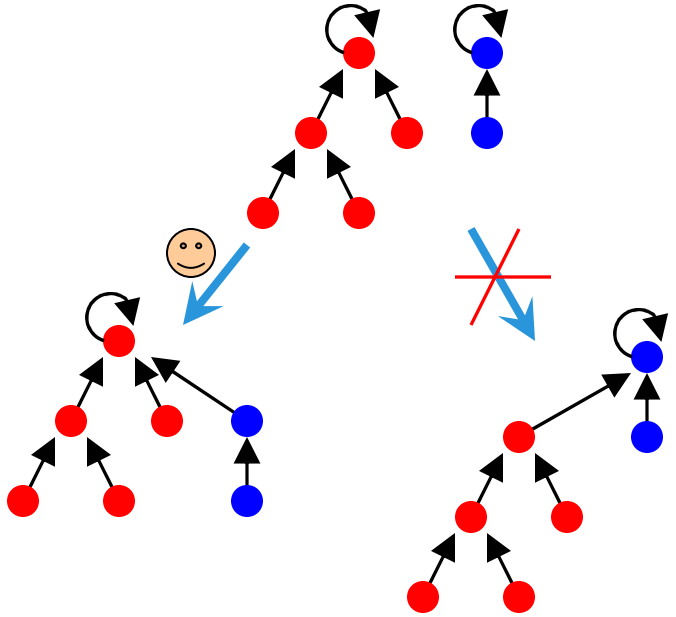
- Think of each set as forming a rooted tree.
- Each node has a pointer to its parent, or to itself if it is the root.
- Initially, each vertex is the root of its tree.
- When we do a Union, we point one set's root to the root of the other.
- When we do a Find, we figure out the roots of the vertices' trees (follow the pointer chain), and see if they are the same.



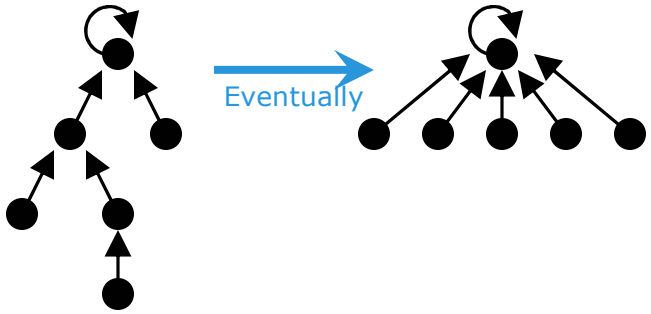
Other Graph Topics
Union-Find [4/6]

Good implementations of a Disjoint-Set Forest do two optimizations.

- First, when doing a Union, attach the smaller tree to the larger tree's root.

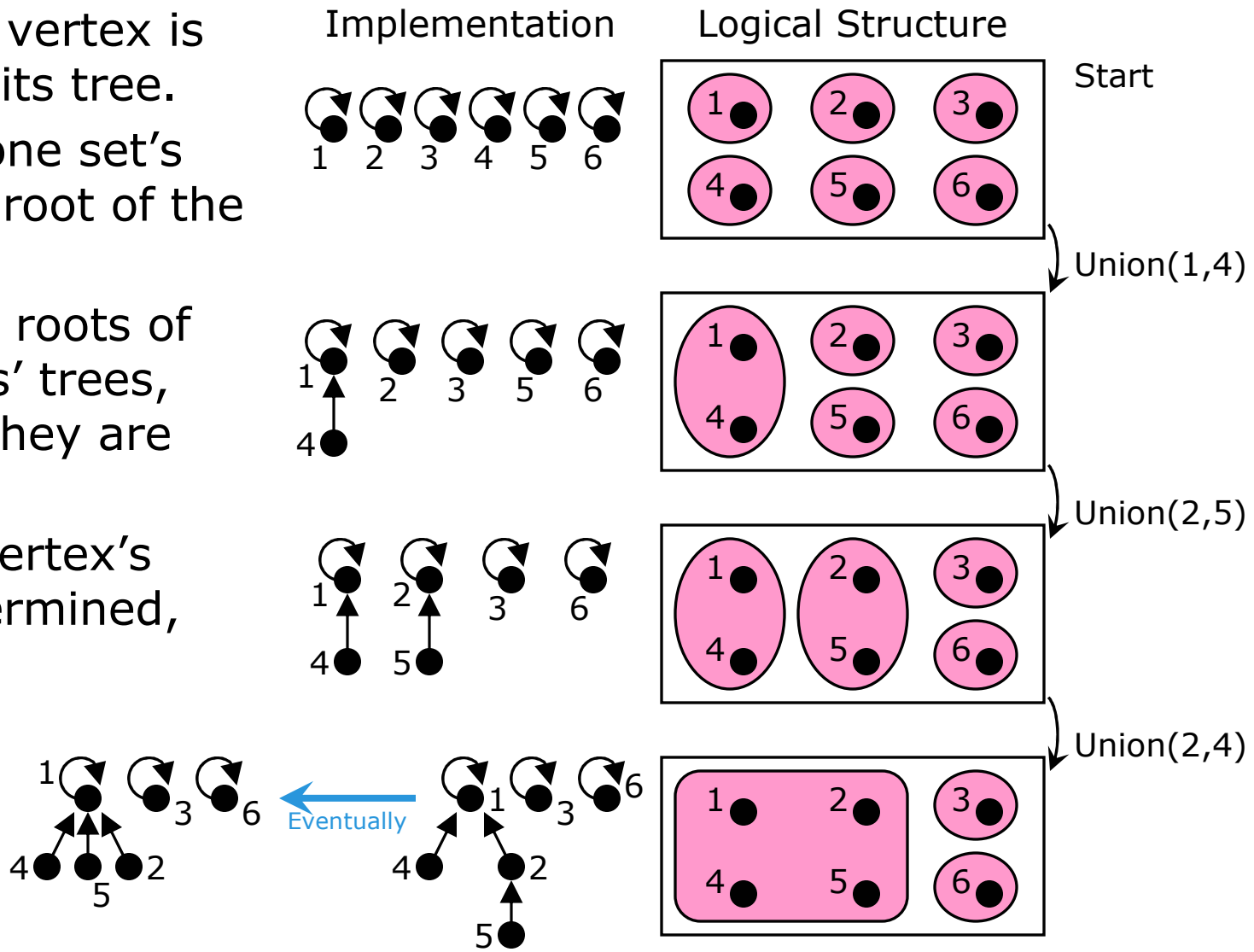


- Second, whenever a vertex's root is determined, point its pointer there.



Other Graph Topics
Union-Find [5/6]

- Initially, each vertex is the root of its tree.
- Union: Point one set's root to the root of the other.
- Find: Find the roots of the vertices' trees, and see if they are the same.
- Whenever a vertex's root is determined, point its pointer there.



Other Graph Topics
Union-Find [6/6]

The amortized time-per-operation for a Disjoint-Set Forest is $O(f(n))$, where $f(n)$ is the **extremely** slow-growing inverse Ackermann function.

- Blackboard time.
- Thus: About as near to amortized constant time as one can get, without actually being amortized constant time.

Note: Good implementations of Kruskal's Algorithm use a Disjoint-Set Forest.

Other Graph Topics
Hard Problems

One reason that the graph algorithms form an active research field is that many important practical problems are apparently very hard (computationally) to solve.

Examples

- Proper Vertex Coloring
  - Color the vertices of a graph, using the least possible number of colors, so that vertices joined by an edge get different colors.
  - This problem is **NP-hard**, meaning that it lies in a special class of problems for which no efficient algorithm is known.
    - See CS 411 for more info.
- Traveling Salesperson Problem
  - Given a graph with costs on the edges, find the lowest cost way to visit every vertex and return to the starting point.
  - This is another NP-hard problem.