Other Graph Topics Course Wrap-Up

CS 311 Data Structures and Algorithms Lecture Slides Friday, December 6, 2024

Glenn G. Chappell Department of Computer Science University of Alaska Fairbanks ggchappell@alaska.edu © 2005-2024 Glenn G. Chappell Some material contributed by Chris Hartman

The Rest of the Course Overview

Final Topics

External Data

- Previously, we dealt only with data stored in memory.
- Suppose, instead, that we wish to deal with data stored on an external device, accessed via a relatively slow connection and available in chunks (data on a disk, for example).
- How does this affect the design of algorithms and data structures?
- (part) Graph Algorithms
 - A graph models relationships between pairs of objects.
 - This is a very general notion. Algorithms for graphs often have very general applicability.

This usage of "graph" has nothing to do with the graph of a function. It is a different definition of the word.



Topics

- ✓ Introduction to Graphs
- ✓ Graph Traversals
- ✓ Spanning Trees
 - Other Graph Topics

Review

A graph consists of vertices and edges.

- An edge joins two vertices: its **endpoints**.
- 1 **vertex**, 2 vertices (Latin plural).
- Two vertices joined by an edge are adjacent; each is a neighbor of the other.



In a **weighted graph**, each edge has a **weight** (or **cost**).

- The weight is the resource expenditure required to use that edge.
- We typically choose edges to minimize the total weight of some kind of collection.



Review Introduction to Graphs [2/3]

Two common ways to represent graphs.

Adjacency matrix. 2-D array of 0/1 values.

- "Are vertices *i*, *j* adjacent?" in $\Theta(1)$ time.
- Finding all neighbors of a vertex is slow for large, sparse graphs.

Adjacency lists. List of lists (arrays?). Adjacency Lists List *i* holds neighbors of vertex *i*. "Are vertices i, j adjacent?" in Θ(log N) time if lists are sorted arrays; Θ(N) if not. 🔨 Finding all neighbors can be faster. *N*: the number of vertices.





0:1,3

1:0

3:0

2:

When we analyze the efficiency of graph algorithms, we consider *both* the number of vertices and the number of edges.

- N = number of vertices
- *M* = number of edges

When analyzing efficiency, we consider adjacency matrices & adjacency lists separately.

The *total* size of the input is:

- For an adjacency matrix: N^2 . So $\Theta(N^2)$.
- For adjacency lists: N + 2M. So $\Theta(N + M)$.

Some particular algorithm might have order (say) $\Theta(N + M \log N)$.

Review Spanning Trees — Introduction

A **tree** is a graph that:

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

A **spanning tree** in a graph *G* is a tree that:

- Includes only vertices and edges of G.
- Includes *all* vertices of *G*.
- Fact. Every connected graph has a spanning tree.

An important problem: given a weighted graph, find a **minimum spanning tree**—a spanning tree of minimum total weight. There are several nice algorithms that solve this problem.



Here, "tree"

does *not* mean

"rooted tree".





- A **greedy** algorithm is "shortsighted". It proceeds in a series of choices, each based on *what is known at the time*. Choices are:
 - Feasible. Each makes sense.
 - Locally optimal. Best possible based on current information.
 - Irrevocable. Once a choice is made, it is permanent. A greedy algorithm never backtracks.

Being greedy is usually *not* a good way to get correct answers.However, in the cases when being greedy gives correct results, it tends to be *very fast*.

Prim's Algorithm is a greedy algorithm to find a minimum spanning tree in a connected weighted graph.

Idea

- One vertex is specified as *start*.
- As the algorithm proceeds, we add edges to a tree. Using these edges, we are able to reach more and more vertices from *start*.
- Procedure. Repeatedly add the lowest-weight edge from a reachable vertex to a non-reachable vertex, until all vertices are reachable.



It is not obvious that Prim's Algorithm correctly finds a minimum spanning tree. But it does! Finding the lowest-weight edge from reachable to not-reachable:

- Use a Priority Queue holding edges, ordered by weight and implemented as a Minheap. So getFront & delete the edge of *least* weight.
- Insert edges that join reachable & not-reachable vertices: when marking a vertex as reachable, insert into the PQ all edges from this vertex to not-reachable neighbors.
- When getting an edge from the PQ, check to be sure it still joins reachable & not-reachable vertices. If not, skip it.

Represent a weighted graph as usual + matrix of edge weights.

Our Prim's Algorithm implementation uses std::priority_queue to find the lowest-weight edge.

See prim.cpp.

How efficient is our implementation of Prim's Algorithm?

- It does something with each vertex.
- It does something with each edge, which may involve insertion & deletion in a Priority Queue—implemented using a Binary Heap.
- We do a lot of these, so we can ignore the "amortized" in the time required for the Priority Queue insertion.

9

Result: $\Theta(N + M \log M)$.

For a connected graph, we have $M \ge N - 1$. So: $\Theta(M \log M)$.

Prim's Algorithm can be written in an even faster form.

- Put vertices in the Priority Queue, and allow for increase-key (decrease-key in a Minheap): Fibonacci Heap updating a vertex priority as the algorithm proceeds.
- Implement the Priority Queue using a **Fibonacci Heap**. Result: $\Theta(M + N \log N)$.

4

2

Another greedy algorithm to find a minimum spanning tree: Kruskal's Algorithm [J. Kruskal 1956].

Procedure

- Set edge set of tree to empty.
- Repeat:
 - Add the least-weight edge joining two vertices that cannot be reached from each other using edges added so far.
- Return edge set of tree.



Same spanning tree as Prim's Algorithm, but constructed in a different order.

To implement Kruskal's Algorithm well, we need an efficient way to check whether a vertex can be reached from another vertex. *We cover a solution to this problem soon!*

Other Graph Topics

Shortest Path Problem. Given a weighted graph with two vertices specified, find the shortest path from one to the other (the path with the lowest total weight).



7 + 1 + 2 + 2 = 12

Dijkstra's Algorithm [E. Dijkstra 1959]. Given a *start* vertex, this algorithm finds, for each vertex in the weighted graph, the shortest path from *start* to that vertex.

Vertex label = length of shortest path from *start* to this vertex that has been found *so far*.

Procedure:

- Label each vertex. *Start* gets 0. Others get ∞ .
- Mark all vertices as unvisited.
- Repeat while there are unvisited vertices:
 - Find the unvisited vertex with the smallest label. Call it x.
 - Mark x as visited.
 - For each unvisited neighbor of *x*:
 - Let newLabel = (label of x) + (weight of edge from x to neighbor).
 - If newLabel < (label of neighbor), then set (label of neighbor) = newLabel.</p>
- Done. Meaning of each vertex label when done: length of shortest path from *start* to this vertex, or ∞ if there is no path.



Efficiency issues for Dijkstra's Algorithm are similar to those for Prim's Algorithm.

We put vertices in a Minheap and allow for decrease-key.



Then, for a connected graph, using a Binary Heap gives $\Theta(M \log N)$, while a Fibonacci Heap gives $\Theta(M + N \log N)$, just as for Prim's Algorithm.

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

An example of this is given by the operations in ADT called **Union-Find** (a.k.a. **Find-Merge**).

Data

• A graph.

Operations

- MakeSet. Create a new vertex.
- **Union**. Given two vertices, add an edge between them.



 Find. Given a vertex, determine which component (connected chunk) of the graph it lies in.

The names of these operations should make more sense shortly. Using *MakeSet* and *Union*, we can construct any graph.

Using *Find*, we can determine whether there is a path joining two vertices: this is true if they lie in the same component.

Other Graph Topics Union-Find [2/7]

Union-Find operations:

- MakeSet. Create a new vertex.
- Union. Given two vertices, add an edge between them.
- **Find**. Given a vertex, determine which component of the graph it lies in.
- But Union-Find is not really about graphs ...



Other Graph Topics Union-Find [3/7]

Union-Find is not really about *graphs*; it is about *blobs**.

- Each vertex lies in some blob.
- MakeSet creates a new singlevertex blob.
- Union merges the blobs containing two given vertices into one blob.
- Find determines which blob a given vertex lies in.

Since we only care about blobs, we do not need to keep track of edges.

We must keep track of *something*—but what? We answer this shortly.

Blobs* are actually called **sets. Thus the names of the operations.



Union-Find operations usually do not use a graph representation. Rather, the are done on some kind of **disjoint-set structure** (also called a *union-find structure* or *find-merge structure*.).

A commonly used—and very efficient—example is a **Disjoint-Set Forest** [B. Galler & M. Fischer 1964].

- Node-based. Each node represents one vertex.
- A set (blob) forms a rooted tree.
- Each node has a pointer. This points to its parent, or to itself if it is the root of a tree.

Doing the Union-Find Operations

- MakeSet. Create a new node, pointing at itself.
- **Find**. Start at the given vertex, and follow the pointer chain to the root. Return identifier for this root—perhaps its index in an array?
 - So the root—or its index—serves as the identifier for a set.
- Union. Do a *Find* on each of the two given vertices. Point one root at the other, forming a single tree.



Other Graph Topics Union-Find [5/7]

Two optimizations greatly speed up Union-Find operations on a Disjoint-Set Forest.

Union by Rank. When doing a *Union*, if the two trees have different heights, then attach the tree with *smaller* height to the root of the other tree.

 To make this efficient, track the height of each tree in its root. (This is easy to do.)

Path Compression. After following a pointer chain—which happens during a *Find*—point each node in the chain at the root.



Other Graph Topics Union-Find [6/7]

MakeSet. Create node pointing at itself.

- **Union**. 2 *Finds*. Point root of small-height tree at other root.
- **Find**. Follow pointer chain to the root. Point each vertex in chain at the root. Return the root.



CS 311 Fall 2024

Amortized time per operation for a Disjoint-Set Forest, with the two optimizations discussed, is known to be $O(\alpha(n))$, where $\alpha(n)$ is the *extremely* slow-growing **inverse Ackermann function**.

A web search finds info about this easily.

Thus, about as near to amortized constant time as one can get, without actually being amortized constant time.

Implementations of Kruskal's Algorithm typically use a Disjoint-Set Forest.

Lastly, a Disjoint-Set Forest is easy to implement well.

Module Overview Graph Algorithms



The Rest of the Course Overview



Course Wrap-Up

Upon successful completion of CS 311, you should:

- Have experience writing and documenting high-quality code.
- Understand proper error handling, enabling software components to support robust, reliable applications.
- Be able to perform basic analyses of algorithmic efficiency, including use of big-O and related notation.
- Be familiar with various standard algorithms, including those for searching and sorting.
- Understand what data abstraction is, and how it relates to software design.
- Be familiar with standard container data structures, including implementations and relevant trade-offs.

Course Wrap-Up From the First Day of Class: Course Overview — Topics



CS 311 Fall 2024

Scalability matters!

The big winners in the modern world are those who design and produce scalable systems.

Remember, we generally look for:

- Constant or logarithmic time for single-item operations.
- Linear or log-linear time (or faster) for whole-dataset operations.
- Q. Are we talking about worst-case behavior? Is average-case acceptable, with worst-case being slower? What about amortized time?
- A. It depends on the requirements of the project.

We measure time by counting basic operations. These operations may vary. **It matters what we count!** Trade-offs matter!

- Q. Why not just give everyone a quick-reference sheet listing the best solutions to various problems?
- A. Because many problems have no single best overall solution. Understand trade-offs; find solutions that meet your needs.
- Some examples from this semester:
 - Finding: Sequential Search vs. Binary Search.
 - Sequences: Array vs. Linked List vs. Sequence in a Table, etc.
 - Finding again: Binary Search on sorted array vs. Table.
 - Tables: Red-Black Tree vs. Hash Table vs. Prefix Tree vs. *other*.
 - And then there is the effect of slow connections (external methods).
 - Taking time (& money!) to optimize vs. slow code.
 - Using already written data structure/algorithm vs. writing our own.

Error handling, robustness, and reliability *matter!*

Software systems manage sensitive personal data, financial transactions, military equipment, and medical devices. They drive cars, planes, trains, and ships. They run security systems.

The days when we could get away with saying, "Aw, that hardly ever happens; don't worry about it," are largely gone.

- Responsibilities like those above require very low failure rates.
- The increasing size of systems can make "rare" events common.
- Thanks to the Web, malicious users are an important concern.

But *some* projects can get away with ignoring some of these ideas. Understand the needs of the project you are working on!

