Graph Traversals continued
Spanning Trees

CS 311 Data Structures and Algorithms

Lecture Slides

Wednesday, December 2, 2020

Glenn G. Chappell
Department of Computer Science
University of Alaska Fairbanks
ggchappell@alaska.edu

Some material contributed by Chris Hartman

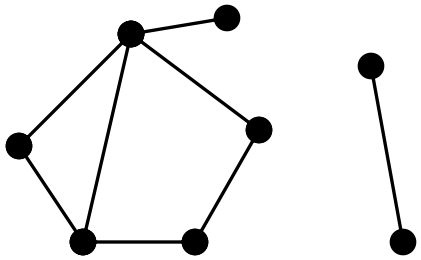
Review

The Rest of the Course Overview

Two 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 sizable 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.

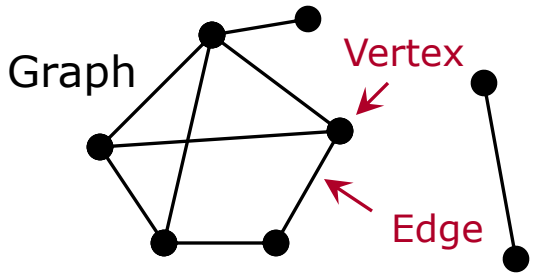


Drawing of a Graph

Review
Introduction to Graphs [1/3]

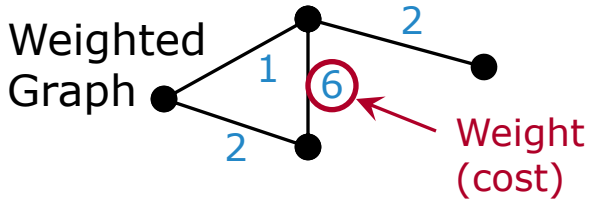
A **graph** consists of **vertices** and **edges**.

- 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.



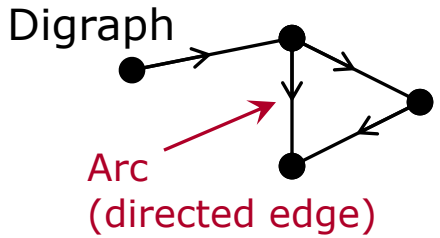
In a **weighted graph**, each edge has a **weight** (or **cost**).

- 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.



If we give each edge a direction, then we have a **directed graph**, or **digraph**.

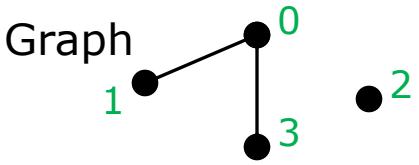
- Directed edges are called **arcs**.



Review
Introduction to Graphs [2/3]

Two common ways to represent graphs.

Both can be generalized to handle digraphs.



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

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

Adjacency Matrix

	0	1	2	3
0	0	1	0	1
1	1	0	0	0
2	0	0	0	0
3	1	0	0	0

Adjacency lists. List of lists (arrays?).
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.

Adjacency Lists

0: 1, 3
1: 0
2:
3: 0

Introduction to Graphs [3/3]

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
Graph Traversals — Introduction

To **traverse** means to visit each vertex (once).

- Traditionally, graph traversal is viewed in terms of a "search": visit each vertex searching for something.

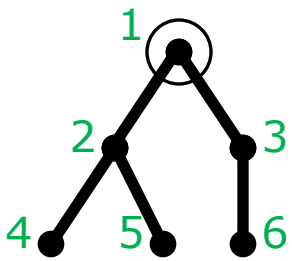
Two important graph traversals.

- **Depth-first search** (**DFS**)
  - Similar to a preorder Binary Tree traversal.
  - When we visit a vertex, give priority to visiting *its* unvisited neighbors (and when we visit one of them, we give priority to visiting *its* unvisited neighbors, etc.).
  - Result: we may proceed far from a vertex before visiting all its neighbors.
- **Breadth-first search** (**BFS**)
  - Visit all of a vertex's unvisited neighbors before visiting their neighbors.
  - Result: Vertices are visited in order of distance from start.

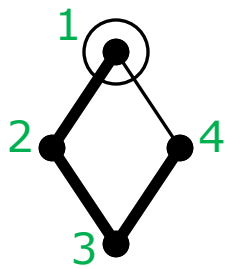
Graph Traversals — DFS [1/2]

DFS has a natural recursive formulation:

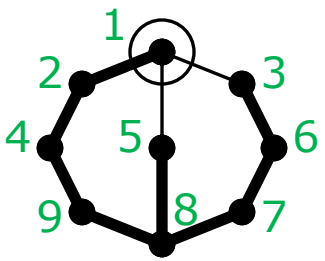
- Given a *start* vertex, visit it, and mark it as visited.
- For each of the start vertex's neighbors:
  - If this neighbor is unvisited, then do a DFS with this neighbor as the start vertex.



DFS: 1, 2, 4, 5, 3, 6



DFS: 1, 2, 3, 4



DFS: 1, 2, 4, 9, 8, 5, 7, 6, 3

Recursion can be eliminated with a Stack. And we can be more intelligent than the brute-force method.

Review
Graph Traversals — DFS [2/2]

Algorithm DFS

- Mark all vertices as unvisited.
- For each vertex:
  - Do algorithm DFS' with this vertex as *start*.

Algorithm DFS'

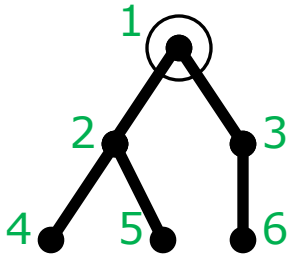
- Set Stack to empty.
- Push *start* vertex on Stack.
- Repeat while Stack is non-empty:
  - Pop top of Stack.
  - If this vertex is not visited, then:
    - Visit it.
    - Push its not-visited neighbors on the Stack.

This part is all we need, if the graph is **connected** (all one piece). The above is only required for **disconnected** graphs.

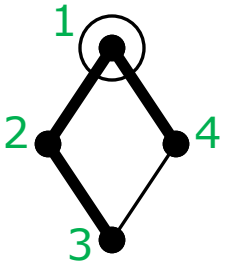
We wrote a non-recursive function to do a DFS on a graph, given adjacency lists.

See `graph_traverse.cpp`.

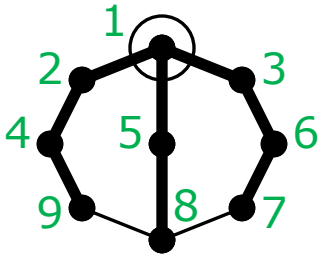
Review
Graph Traversals — BFS



BFS: 1, 2, 3, 4, 5, 6



BFS: 1, 2, 4, 3



BFS: 1, 2, 3, 5, 4, 6, 8, 9, 7

BFS is good for finding the shortest paths to other vertices.

We wrote a non-recursive function to do a DFS on a graph, given adjacency lists.

- For the non-recursive DFS algorithm, changing the Stack to a Queue makes it compute a BFS.
- There was also the detail of enqueueing vertices in forward order, while they were pushed on the Stack in backward order.

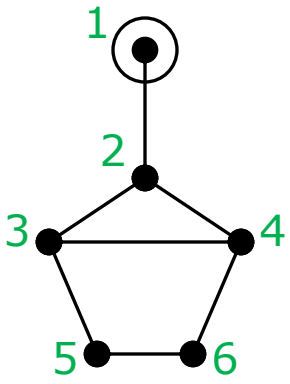
See `graph_traverse.cpp`.

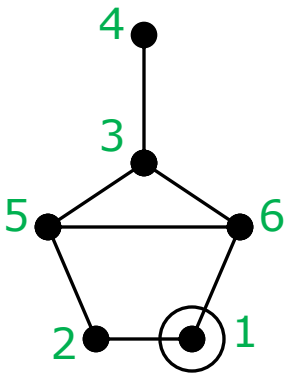
Graph Traversals

continued

Graph Traversals
Try It! [1/2]

For each graph below, write the order in which the vertices will be visited in a DFS and in a BFS.

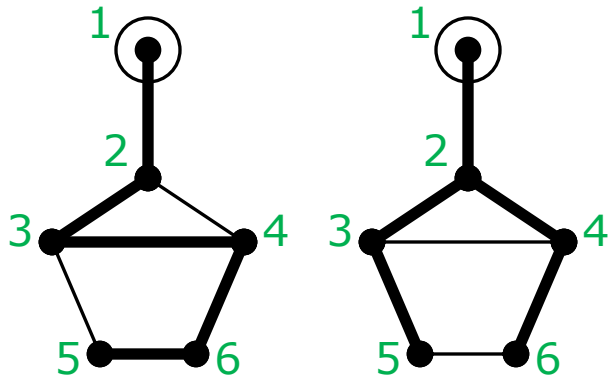


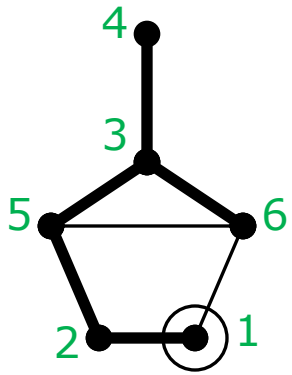


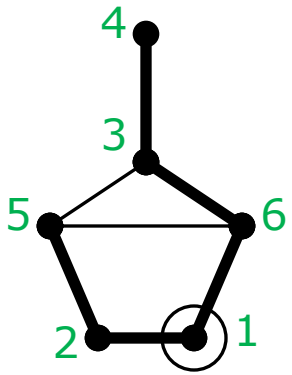
Answers on next slide.

Graph Traversals
Try It! [2/2]

For each graph below, write the order in which the vertices will be visited in a DFS and in a BFS.







Answers

DFS: 1, 2, 3, 4, 6, 5

BFS: 1, 2, 3, 4, 5, 6

DFS: 1, 2, 5, 3, 4, 6

BFS: 1, 2, 6, 5, 3, 4

Graph Traversals
Efficiency [1/2]

What is the order of our DFS & BFS algorithms, when given adjacency lists?

Assume *push*/*pop* & *enqueue*/*dequeue* are constant time.

- *Push* & *enqueue* may be amortized constant time, due to reallocate-and-copy, but since we are doing a lot of *push*/*enqueue* operations, they average out to constant time.

The concept of amortized constant time is very useful in reasoning of this kind.

We process each vertex.

- There are N of these.

We also do *push* and *pop* operations.

- Each time we *push*—or check if we should *push*—we are moving across an edge from one vertex to another. There are two directions to move across an edge: toward one endpoint or the other.
- So the number of *push* operations is no more than $2M$.
- The number of *pop* operations is the same.

Conclusion. Each algorithm is $\Theta(N + 2M) = \Theta(N + M)$.

Graph Traversals
Efficiency [2/2]

What is would the order of our DFS & BFS algorithms be, if they were given an adjacency matrix?

Abbreviated Argument

- We process each vertex. There are N vertices.
- When looking for vertices to push, we examine an entire row of the adjacency matrix.
- Eventually, we will examine every entry of every row: N^2.

Conclusion. Each algorithm is $\Theta(N + N^2) = \Theta(N^2)$.

Regardless of whether it is given adjacency lists or an adjacency matrix, the order of both algorithms in the same as the size of the input—which is the fastest efficiency category possible for an algorithm that reads all of its input.

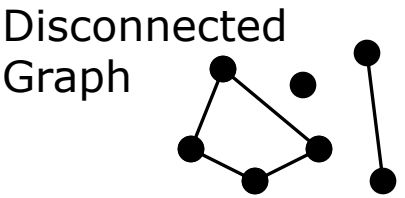
Spanning Trees

Spanning Trees
Introduction [1/3]

A **tree** is a graph that:

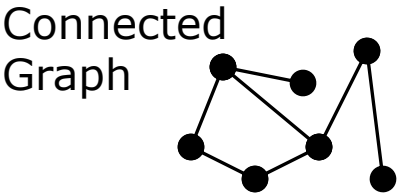
- Is **connected** (all one piece).
- Has no **cycles**.

Here, "tree" does *not* mean "rooted tree".

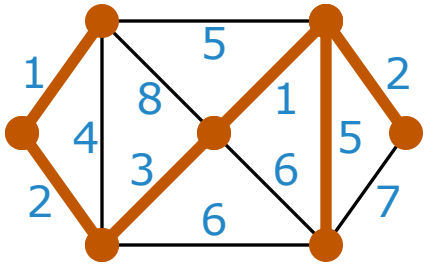


A **spanning tree** in a graph *G* is a tree that:

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



Fact. Every connected graph has a spanning tree.



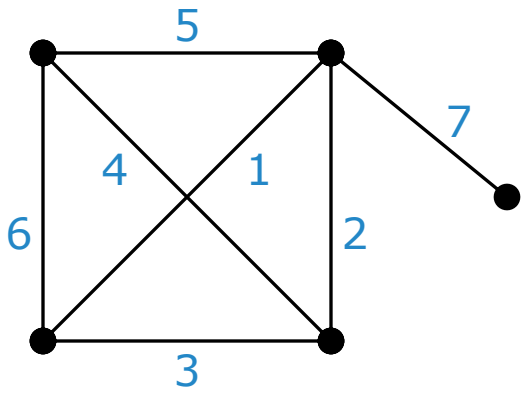
An important problem: given a weighted graph, find a **minimum spanning tree**—a spanning tree of minimum total weight.

There are several nice algorithms that solve this problem.

Spanning Trees
Introduction [2/3] (Try It!)

Try to find a minimum spanning tree in the following weighted graph. Draw it.

Blue numbers are edge weights.

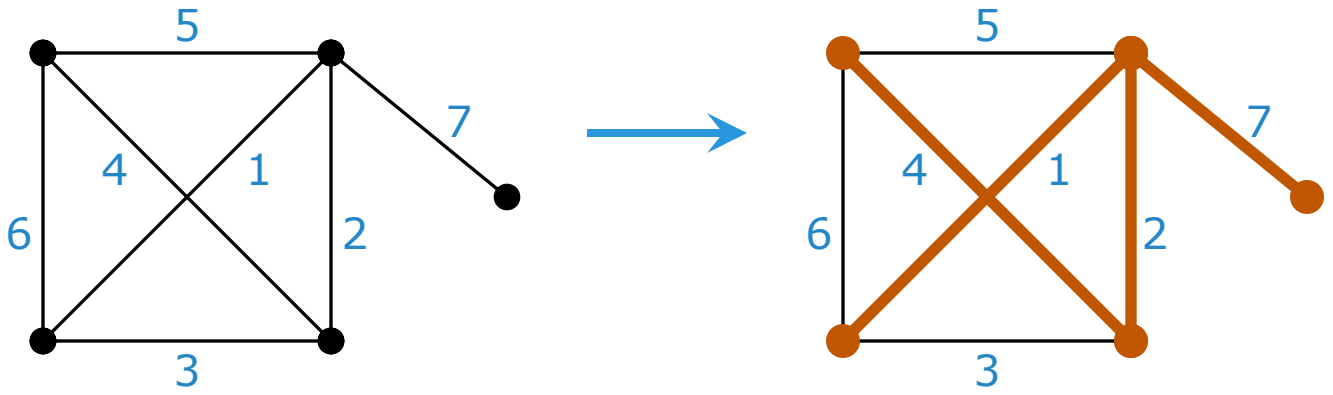


Answer on next slide.

Spanning Trees
Introduction [3/3] (Try It!)

Try to find a minimum spanning tree in the following weighted graph. Draw it.

Blue numbers are edge weights.



Total weight = 1 + 2 + 4 + 7 = 14

We can find a minimum spanning tree using a *greedy* algorithm.

A **greedy** method 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.

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*.

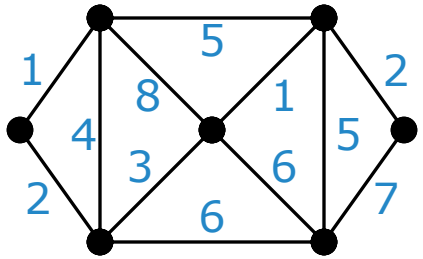
Greedy methods are not just for minimum spanning trees; there are many worthwhile greedy algorithms. *See CS 411.*

Prim's Algorithm — Idea

Here is an idea for a greedy algorithm to find a minimum spanning tree in a connected weighted graph.

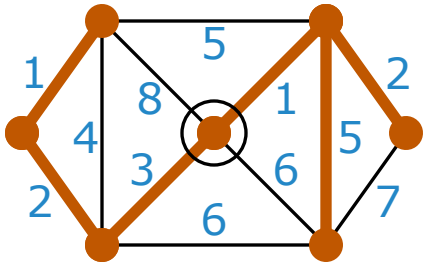
- One vertex is specified as *start*.
- As the algorithm proceeds, we add edges to a tree. Using these edges, we are able to reach more and more vertices from *start*.
- Procedure. Repeatedly add the lowest-weight edge from a reachable vertex to a non-reachable vertex, until all vertices are reachable.

This idea leads to a—correct!—greedy algorithm to find a minimum spanning tree: **Prim's Algorithm**, also called the **Prim-Jarník Algorithm**. [V. Jarník 1930, R. C. Prim 1957, E. Dijkstra 1959]



Prim's Algorithm

- Given: Connected graph, weights on the edges; one vertex is *start*.
- Returns: Edge set of a minimum spanning tree.
- Procedure:
  - Mark all vertices as not-reachable.
  - Set edge set of tree to empty.
  - Mark *start* vertex as reachable.
  - Repeat while there exist not-reachable vertices:
    - Find lowest weight edge joining a reachable vertex to a not-reachable vertex.
    - Add this edge to the tree.
    - Mark the not-reachable endpoint of this edge as reachable.
  - Return edge set of tree.



It is not obvious that Prim's Algorithm correctly finds a minimum spanning tree. *But it does!*

Prim's Algorithm — Issues [1/2]

How can we efficiently find the lowest-weight edge between reachable and a not-reachable vertices?

- Use a Priority Queue holding edges, ordered by weight and implemented as a Minheap. So we do getFront & delete on the edge of *least* weight.
- Insert edges that join reachable & not-reachable vertices.
- When to insert? When marking a vertex as reachable, insert into the PQ all edges from this vertex to not-reachable neighbors.
- This means that, for each edge in the PQ, *at some point*, it joined reachable & non-reachable vertices.
- When getting an edge from the PQ, check to be sure it *still* joins reachable & not-reachable vertices. If not, skip it.
- When the PQ is empty, quit.

Prim's Algorithm — Issues [2/2]

How can we represent a weighted graph?

- Use something like an adjacency matrix, but instead of storing 0/1, store weights.
- Also allow each entry in the matrix to have a special value meaning *no edge*.
- We may wish to have adjacency lists as well, for efficiency.

When the spanning tree is finished, the PQ may not be empty yet.

- Easy optimization: track the number of non-reachable vertices. Stop when this is zero.
- Equivalently, stop when the tree has $N-1$ edges, where N is the number of vertices in the graph.

Prim's Algorithm — CODE

TO DO

- Implement Prim's Algorithm.
  - Use `std::priority_queue` to find the lowest-weight edge.

Done. See `prim.cpp`.

Prim's Algorithm — Efficiency [1/3]

What is the order of our implementation of Prim's Algorithm?

- It does something with each vertex.
- It does something with each edge.
- The latter may involve insertion & deletion in a Binary Heap.

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

For a connected graph, we have $M \geq N - 1$.

So: $\Theta(M \log M)$.

Prim's Algorithm — Efficiency [2/3]

Our Prim's Algorithm implementation is $\Theta(M \log M)$.

But there are implementations that are a bit more efficient.

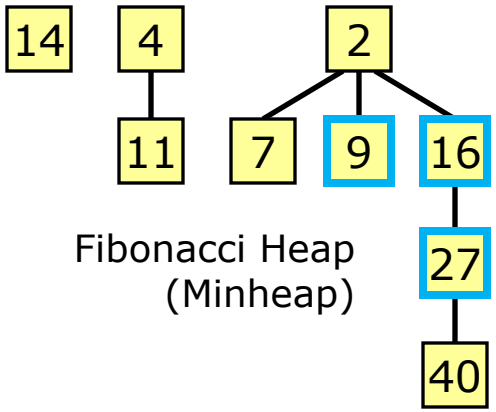
Idea #1. The Priority Queue holds vertices, ordered by cost, not edges.

- The cost of a non-reachable vertex is the cost of the least weight edge from it to a reachable vertex—or ∞, if there is no such edge.
- When we mark a vertex as reachable, we may need to update the cost of some of the vertices in the Priority Queue.
- So we need a structure with more operations than a Priority Queue.
- We can use a Binary Heap in which each vertex in the graph keeps track of where it is in the Heap. When we reduce the cost of a vertex, we do a sift-up on that vertex.
- This operation is called **decrease-key** (or **increase-key** for a Maxheap). It is logarithmic time for a Binary Heap.

Prim's Algorithm — Efficiency [3/3]

Idea #2 (used with Idea #1). Replace the Binary Heap with a *Fibonacci Heap*. [M. L. Fredman & R. E. Tarjan 1987]

- A **Fibonacci Heap** is a data structure similar to a Binary Heap, but:
  - Insert is Θ(1).
  - Decrease-key (increase-key for a Maxheap) is amortized Θ(1).
  - Delete is amortized Θ(log n).
  - No array representation is used.
- Prim's Algorithm goes through every vertex and every edge, so we can ignore the "amortized" when finding its running time.



Fibonacci Heap (Minheap)

- For each edge, we do a fixed number of operations that are all (possibly amortized) constant-time.
- For each vertex, we do a fixed number of operations that are all (possibly amortized) constant-time or logarithmic-time.

Result. Using Ideas #1 & #2, Prim's Algorithm is Θ(M + N log N).

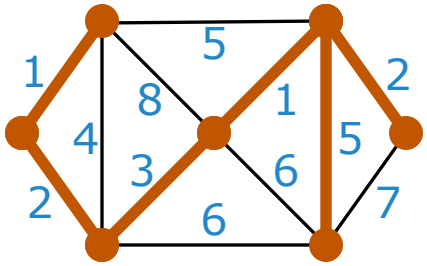
Our version: Θ(M log M).

Spanning Trees
Kruskal's Algorithm

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.



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!