# Spanning Trees

CS 311 Data Structures and Algorithms
Lecture Slides
Wednesday, December 9, 2009

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

# 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]

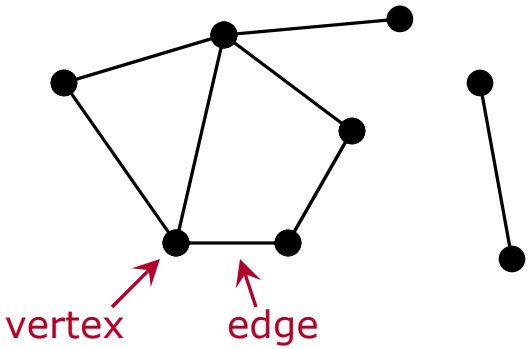
A **graph** is consists of **vertices** and **edges**. An edge joins two vertices.

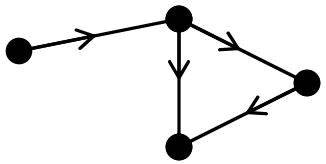
- An example of a graph is shown at right.

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
## Graph Traversals — Introduction

We have discussed traversals of Binary Trees.

- Preorder, inorder, postorder.

We traverse graphs as well.

- To "traverse" here means to visit each vertex once.
- Traditionally, graph traversal is viewed in terms of a "search".

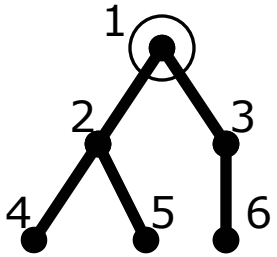
Two kinds of graph traversals.

- Depth-first search (DFS).
  - "Last visited, first explored."
  - Like preorder tree traversal.
  - When we visit a vertex, give priority to visiting its unvisited neighbors (and their unvisited neighbors, etc.).
- Breadth-first search (BFS).
  - "First visited, first explored."
  - Visit all of a vertex's unvisited neighbors before visiting their neighbors.

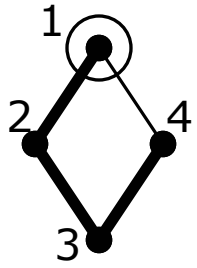
# Review
# Graph Traversals — DFS [1/2]

DFS has a nice 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, do a DFS with this neighbor as the start vertex.




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




DFS: 1, 2, 3, 4

We get a **DFS tree** (shown in bold above).

DFS is convenient if we think about traveling through the graph, minimizing the number of edges we cross.

# Graph Traversals — DFS [2/2]

We can, as usual eliminate recursion using a Stack.

- "Last visited, first explored."
- Then we have an iterative DFS algorithm using a local Stack.
  - And we can use it in a more intelligent way than the "brute-force" recursion elimination method.

Algorithm

- 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.

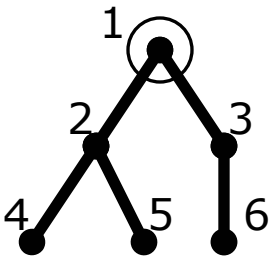
TO DO

- Write a non-recursive function to do a DFS on a graph, given an adjacency matrix.

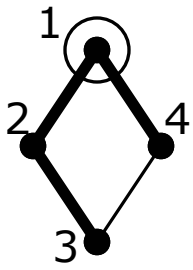
*Done. See* **`graphtraverse.cpp`**, *on the web page.*

## Review
## Graph Traversals — BFS

BFS is "first visited, first explored".




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




BFS: 1, 2, 4, 3

Thus: replace the Stack with a Queue.

BFS is not as nice in several ways.

- No elegant recursive formulation.
- Not a convenient way to travel around a graph.

But BFS is *useful*.

- BFS is good for finding the shortest paths to other vertices.
- Also, looking for things "nearby first".

TO DO

- Modify our DFS function to do BFS.

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

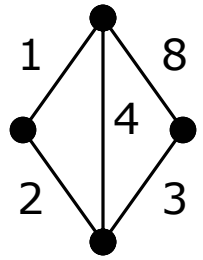
# Graph Traversals — Generalization: Shortest Path

Sometimes we place **costs** (or **weights**) on edges of a graph.

- Cost might represent distance, time, money, etc.
- In general, the "cost" of an edge is how much resource expenditure it takes to "use" the edge in some way.
- We want to do things in a way that minimizes the total cost.

Example: Shortest Path

- Use a variation on BFS called **Dijkstra's Algorithm**.
  - Edsger Dijkstra, 1959.
- Choose a start vertex.
- Label each vertex with the length of the shortest known path from the start to it.
  - So, for now, start gets 0, others get ∞ (no path is known).
- Repeat:
  - Among the unvisited vertices, find the one with the smallest label ("me" below).
  - For each neighbor, if my label plus the cost of the edge between us is less than its label, then replace its label with the new value.
  - Mark me as visited.



# Spanning Trees
## Introduction

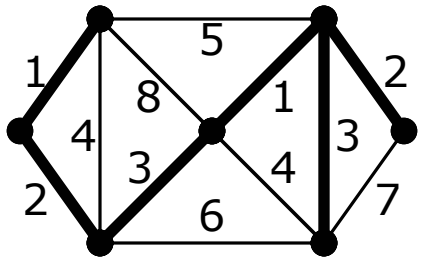
A **tree** is a graph that:

- 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.

# Spanning Trees
## Prim's Algorithm — Idea

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

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).

Improve current state as much as possible at each step.

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

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

## Prim's Algorithm — Description

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.

## Prim's Algorithm — Issues

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.

## Prim's Algorithm — Write It

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* $O(\log n)$.

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.)