The Bridges of Königsberg

Each bridge is an edge
Each part of town is a vertex
Is there a path that crosses each edge exactly once?

The (Graffitied) Bridges of Königsberg

Can we walk around Königsberg, crossing each bridge exactly once?

Graph… ADT?

Graphs are a formalism useful for representing relationships between things

\[ G = (V, E) \]

- \( V \) is a set of vertices: \( v_1, v_2, \ldots, v_n \)
- \( E \) is a set of edges: \( e_1, e_2, \ldots, e_m \) where each \( e_i \) connects two vertices \( (v_{i1}, v_{i2}) \)

Operations include:
- Iterating over vertices
- Iterating over edges
- Iterating over vertices adjacent to a specific vertex
- Asking whether two vertices are connected via an edge

\[ V = \{ \text{Han, Leia, Luke} \} \]
\[ E = \{ (\text{Luke, Leia}), (\text{Han, Leia}), (\text{Leia, Han}) \} \]
Graph Definitions

In **directed** graphs, edges have a specific direction:

![Directed Graph Example]

In **undirected** graphs, they don’t (edges are two-way):

![Undirected Graph Example]

Vertices $u$ and $v$ are **adjacent** if $(u, v) \in E$

More Definitions:

**Simple Paths and Cycles**

A simple path repeats no vertices (except that the first can be the last):

$p = \{\text{Seattle, Salt Lake City, San Francisco, Dallas}\}$

$p = \{\text{Seattle, Salt Lake City, Dallas, San Francisco, Seattle}\}$

A cycle is a path that starts and ends at the same node:

$p = \{\text{Seattle, Salt Lake City, Dallas, San Francisco, Seattle}\}$

A simple cycle is a cycle that repeats no vertices except that the first vertex is also the last (in undirected graphs, no edge can be repeated)

Trees as Graphs

- Every tree is a graph with some restrictions:
  - The tree is **directed**
  - There are no cycles (directed or undirected)
  - There is a directed path from the root to every node

![Bad Tree Example]

Directed Acyclic Graphs (DAGs)

DAGs are directed graphs with no cycles.

*If program call-graph is a DAG, then all procedure calls can be in-lined*

Trees $\subset$ DAGs $\subset$ Graphs

Graph Representations

- List of vertices + list of edges

- 2-D matrix of vertices (marking edges in the cells) “adjacency matrix”

- List of vertices each with a list of adjacent vertices “adjacency list”

Representation 1: Adjacency Matrix

A $|V| \times |V|$ array in which an element $(u, v)$ is true if and only if there is an edge from $u$ to $v$

![Adjacency Matrix Example]

runtime: 

space requirements:
Representation 2: Adjacency List

A $|V|$-ary list (array) in which each entry stores a list (linked list) of all adjacent vertices.

Some Applications: Moving Around Washington

What’s the fastest way from Seattle to Spokane?

Some Applications: Communication in Washington

What’s the cheapest inter-city network?

Some Applications: Reliability of Communication

If we lose Wenatchee, can Seattle still talk to Spokane?

Some Applications: Bus Routes in Downtown Seattle

If we’re at 3rd and Pine, can we get to 1st and Union?

Some Applications: Orderings and Determining Dependancies

Okay, everybody, get up and stretch!
Total Ordering on Graphs

1. Does it make sense to define an ordering on an undirected graph?

Partial Order: Taking a Break in Class

Okay, everybody, stand up and stretch!

Some Applications:
Topological Sort

Given a graph, $G = (V, E)$, output all the vertices in $V$ such that no vertex is output before any other vertex with an edge to it.

Topo-Sort (Take One)

Label each vertex’s in-degree (# of inbound edges)
While there are vertices remaining
Pick a vertex with in-degree of zero and output it
Reduce the in-degree of all vertices adjacent to it
Remove it from the list of vertices

Runtime:

Topo-Sort (Take Two)

Label each vertex’s in-degree
Initialize a queue to contain all in-degree zero vertices
While there are vertices remaining in the queue
Pick a vertex $v$ with in-degree of zero and output it
Reduce the in-degree of all vertices adjacent to $v$
Put any of these with new in-degree zero on the queue
Remove $v$ from the queue

Runtime:

Other Graph Applications?
Mazes == Graphs

- Cells are vertices
- Edges are doors from one cell to another

Breadth-First Search

BFS characteristics:
- Nodes being worked on maintained in a FIFO Queue, not a stack (like DFS)
- Iterative style procedures sometimes easier to design than recursive procedures

- Put root in a Queue
- Repeat until Queue is empty:
  - Dequeue a node
  - Process it
  - Add its children to queue

BFS, Graphically

Explore vertices in order of distance from start

More BFS pictures

Using BFS

How do we …
- Determine if $G$ is connected?
- Find the distance from the root to a node?
- Determine if $G$ has any cycles?
- Determine if $G$ is a tree?
- Find a path from the root to a node?

Introducing the BFS tree
Graph Traversals

• Breadth-first search (and depth-first search) work for arbitrary (directed or undirected) graphs - not just mazes!
  – Must mark visited vertices so you do not go into an infinite loop!
• Either can be used to determine connectivity:
  – Is there a path between two given vertices?
  – Is the graph (weakly) connected?
• Important difference: Breadth-first search always finds a shortest path from the start vertex to any other (for unweighted graphs)
  – Depth first search may not!

“Weakly connected”: A detour into connectivity

Undirected graphs are connected if there is a path between any two vertices

Directed graphs are strongly connected if there is a path from any one vertex to any other

Directed graphs are weakly connected if there is a path between any two vertices, ignoring direction

A complete graph has an edge between every pair of vertices