CSE 241 Class 18

Jeremy Buhler

November 2, 2015

1 Graphs: a Spiffy Abstraction

- A **graph** is a collection of nodes (vertices) connected by links (edges)
- **Example**:

- may not be fully connected
- above graph is *directed*: each edge has a direction indicated by arrow.
- we say, e.g., that edge goes from 4 to 1.
- graph may contain *cycles*

A little more formally...

- A directed graph (digraph) G is a pair (V, E)
- V is a finite set of vertices
- E is a set of edges (a binary relation on V)
- Example from diagram: **What is V? What is E?**
- $V = \{1, 2, 3, 4, 5, 6\}$

- $E = \{(1,2)(2,2)(2,4)(2,5)(4,1)(4,5)(5,4)(6,3)\}$

Graphs can be *undirected* as well. **Example**:

- In an undirected graph, edges go both ways
- Example from diagram: **What is V? What is E?**
- $V = \{1,2,3,4,5\}$
- $E = \{(1,2)(2,3)(2,5)(3,5)\}$
- Edge $(2,1)$ identical to $(1,2)$ (not true in digraph!)

2 Who Cares About Graphs?

Graphs abstract the idea of **relationships** between objects: dependencies, networks (roads, computers), compatibility, etc. Tons of applications!
Example 1: compilation dependencies

- When compiling big programs, must perform some operations (e.g. compilation, library construction) before others (e.g. linking)
- *Dependencies*: describe what other files must exist before each file can be created (a makefile)
- *Abstraction*: dependency graph
- vertices represent files (source code, object code)
- directed edge (u, v) means that file u must exist before file v can be built
- graph must not contain circular dependencies – we say it is a **DAG** (directed acyclic graph)

- **Problem**: given dependency graph, find order in which all files can be built without violating dependencies
- Solution is **topological sort** (coming later)

Example 2: shortest paths

- What's the fastest way from point A to point B, given a restricted network of roads?
- *Abstraction*: build graph describing road network
- vertices represent locations (e.g. cities)
- edges represent roads (e.g. highways)
- each edge is **weighted** with the distance between its endpoints (e.g. mileage)
- edges may be directed (one-way) or undirected (two-way)
- **Problem**: what is shortest path (fastest travel route) between two vertices (two cities?)
- (ever asked Google Maps for driving directions?)

Example 3: telephone networks

- In the late 19th and early 20th century, cities were connected by long-distance telephone and railroad networks.
- *Abstraction*: graph of potential networks
- vertices represent cities
- edges represent possible network links (undirected)
- edge weight = cost of building the link
- **Problem**: what set of links will connect every city on the map at the lowest cost?
- this is the **minimum spanning tree** problem!

Not considered: max flow / maximum matching, some interesting comp bio problems, etc etc etc.

3 Representations of Graphs

Given a graph $G = (V, E)$, how to represent it in a computer?

- Assume $V = \{1 \dots n\}$ for some n
- **Defn**: vertices i, j are **adjacent** if $(i, j) \in E$

Option 1: adjacency list

- Create an array **Adj**[] with one slot per vertex of G
- Adj[u] contains list of each vertex v for which $(u, v) \in E$
- For undirected graphs, every edge (i, j) has both $j \in \text{Adj}[i]$ and $i \in \text{Adj}[j]$

Example (for first and second graphs):

How much space does adjacency list take up?

- Let $m = |E|$ (# of edges)
- Let $n = |V|$ (# of vertices)
- Adjacency list requires **how much space?**
- $\Theta(n + m)$ – one slot per vertex, one (two) list element(s) per edge

Extension: we can have multiple edges between any pair of vertices. This is called a *multigraph*.

- Why do this? Sometimes, abstraction wants it!
- *Example*: to get from South Grand to Wash U., can take
  1. Grand to 40 to Clayton to Big Bend, or
  2. Grand to Chippewa to Laclede Station to Big Bend, or
  3. Grand to Lindell to Skinker
- All these options connect the two endpoints. We can assign them *weights* reflecting relative length, amount of traffic etc.
- How do we represent a multigraph in an adjacency list?
- For each vertex, think of storing *edges*, not vertices, adjacent to it.
- (Edges have "other" endpoint and also weight)

Option 2: adjacency matrix

- Again, assume vertices are numbered $1 \dots n$

- Create $n \times n$ matrix Adj such that

$$\text{Adj}[u,v] = \begin{cases} 1 & \text{if } (u,v) \in E \\ 0 & \text{otherwise} \end{cases}$$

- For an undirected graph, the matrix Adj is always **what?** symmetric
- takes $\Theta(n^2)$ space

Example (for first graph):

4 List vs Matrix Representation

Is one graph representation better than the other? Depends on application, **density** of graph

- In any graph, $m \leq n^2$
- *Dense graph*: $m = \Omega(n^2)$ (lots of edges)
- *Sparse graph*: $m = O(n)$ (few edges)

Running times of algorithms can depend on both m and n – some better for sparse, some for dense graphs

- If graph is dense, adj list and adj matrix take about same amount of space
- If graph is sparse, adj list takes much less space
- But how fast can we check whether two vertices u, v are adjacent in G?
- Adj matrix: can look up $\text{Adj}[u,v]$ in constant time
- Adj list: must traverse list for u looking for v
- List could be **how long?** ... n (edge from u to each vertex in G)
- Hence, lookup could be $\Theta(n)$