CSCI 136: Data Structures and Advanced Programming
Lecture 30
Graphs, part 2
Instructor: Dan Barowy Williams

Topics
Reachability
Connectedness
Graph operations
Graph representations

Your to-dos
1. Read before Wed: Bailey, Ch. 13.1.
2. Lab 10 (partner lab), due Tuesday 11/29 by 10pm.
3. Last quiz this Fri/Sat.

Announcements
Sean Barker ’09, Bowdoin College
Friday, Dec 2 @ 2:35pm
Computer Science Colloquium – Wege TCL 123
Smart Meters for Smart Cities: Data Analytics in Energy-Aware Buildings

The proliferation of smart energy meters has resulted in many opportunities for next-generation buildings. Energy-aware “smart buildings” may optimize their energy consumption and provide convenience and economic benefits through analysis of their meter data. However, storing and analyzing this data presents computational challenges, especially when conducted at scale. In this talk, I discuss our work on several problems in this space, focusing particularly on efficient compression of smart meter data and the disaggregation of building-wide consumption into individual device consumption. Our work in these areas aims to support the development of sustainable, energy-efficient smart cities and smart grids.
Graphs

Reachability and Connectedness

“Siri, can I drive from Boston to Hong Kong?”
“Siri, which places can I drive to?”

Reachability

A vertex $v$ in $G$ is \textbf{reachable} from vertex $u$ in $G$ if there is a \textbf{path} from $u$ to $v$.

For an \textbf{undirected} graph $G$, \textbf{v} is \textbf{reachable} from vertex $u$ iff \textbf{u} is \textbf{reachable} from vertex \textbf{v}.

Is $c$ \textbf{reachable} from $d$? Yes.

Connectedness

An undirected graph $G$ is \textbf{connected} if for every pair of vertices $u, v$ in $G$, $v$ is \textbf{reachable} from $u$.

The set of all \textbf{vertices reachable from v}, along with all \textbf{edges} of $G$ connecting any two of them, is called the \textbf{connected component of v}.

(note that the connected component is itself a graph)
Fundamental graph ADT operations

adjacent(a, d) = true
adjacent(a, b) = false
adjacent(a, c) = false

bool adjacent(Vertex u, Vextex v):
Given vertices u and v, are they adjacent?
(i.e., share an edge?)
Fundamental graph ADT operations

\[ \text{vertices}(1) = [a, b] \]
\[ \text{vertices}(2) = [d, b] \]

**vertices(1)** = \{a, b\}
**vertices(2)** = \{d, b\}

**Vertex[] vertices(Edge e):**
Given edge \( e \), what are its **end points**?

**int degree(Vertex v):**
Given vertex \( v \) how many vertices are **adjacent**?

\[ \text{degree}(a) = 2 \]
\[ \text{degree}(c) = 0 \]

**int degree(Vertex v):**
Given vertex \( v \) how many vertices are **adjacent**?

**Vertex[] neighbors(Vertex v):**
Given vertex \( v \) what other vertices are **adjacent**?

**Vertex[] neighbors(Vertex v):**
Given vertex \( v \) what other vertices are **adjacent**?

Reachability?

How might we implement the following method

\[ \text{bool reachable(Vertex u, Vertex v)} \]
using the fundamental operations just described?

\[ \text{bool adjacent(Vertex u, Vertex v)} \]
\[ \text{bool incident(Vertex v, Edge e)} \]
\[ \text{Vertex[] vertices(Edge e)} \]
\[ \text{int degree(Vertex v)} \]
\[ \text{Vertex[] neighbors(Vertex v)} \]
Graph data structures

An adjacency list is a data structure for representing a finite graph. It consists of a list of unordered lists.

```
[[c,d],[d,b],[a,b]]
```

Object-oriented adjacency list

There are many variants on adjacency lists. The most common is the **object-oriented adjacency list** that stores a list of adjacent vertices in each vertex object.

- a: [b]
- b: [a,d]
- c: [d]
- d: [b,c]

```
public class Vertex<T> {
    T label;
    List<Vertex<T>> neighbors = new SinglyLinkedList<>();
}
```

(Strictly speaking, c and d are references to Vertex objects)
Adjacency list

This latter version is especially thrifty for directed graphs.

```
  a: []
b: [a, d]
c: []
d: [b, c]
```

Adjacency matrix

An adjacency matrix is a data structure for representing a finite graph. It consists of a square matrix (usually implemented as an array of arrays). In the simplest case, the elements of the matrix indicate whether an edge is present. Elements on the diagonal are defined as zero.

```
ah  b  c  d
a  0  1  0  0
b  1  0  0  1
c  0  0  0  1
d  0  1  1  0
```

Adjacency matrix

In an undirected graph, the adjacency matrix is symmetric.

```
ah  b  c  d
a  0  1  0  0
b  1  0  0  1
c  0  0  0  1
d  0  1  1  0
```

Adjacency matrix

In an undirected graph, the adjacency matrix is symmetric.

```
ah  b  c  d
a  0  1  0  0
b  1  0  0  1
c  0  0  0  1
d  0  1  1  0
```
In an **undirected graph**, the adjacency matrix is **symmetric**.

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>c</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>d</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

In a **directed graph**, the adjacency matrix is **not symmetric** because edges are directed. A directed edge, from \( \rightarrow \) to, is conventionally encoded in **row-major** form, with from being on the **vertical** axis.

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>d</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Adjacency matrix

In a directed graph, the adjacency matrix is not symmetric because edges are directed. A directed edge, from \( \rightarrow \) to, is conventionally encoded in row-major form, with from being on the vertical axis.

Activity

Write down both adjacency matrix and adjacency list representations for this graph.

Recap & Next Class

Today:

Graph operations
Graph representations

Next class:

Connectedness algorithms