April 12 – Graph Traversal
Wade Fagen-Ulmschneider, Craig Zilles
Graph ADT

Data:
- Vertices
- Edges
- Some data structure maintaining the structure between vertices and edges.

Functions:
- insertVertex(K key);
- insertEdge(Vertex v1, Vertex v2, K key);
- removeVertex(Vertex v);
- removeEdge(Vertex v1, Vertex v2);
- incidentEdges(Vertex v);
- areAdjacent(Vertex v1, Vertex v2);
- origin(Edge e);
- destination(Edge e);
**Key Ideas:**
- Given a vertex, $O(1)$ lookup in vertex list
  - Implement w/ a hash table, etc
- All basic ADT operations runs in $O(m)$ time
Adjacency Matrix

Key Ideas:
- Given a vertex, $O(1)$ lookup in vertex list
- Given a pair of vertices (an edge), $O(1)$ lookup in the matrix
- Undirected graphs can use an upper triangular matrix
Graph Implementation: Edge List

![Graph Diagram]

- Vertices: u, v, w, z
- Edges: (u, v), (v, w), (u, w), (w, z), (u, c), (v, b), (w, d)
Adjacency List

u → a, c
v → b, d
w → b, c, d
z → d

u → v, a
v → w, b
w → u, c
w → z, d
Adjacency List
Adjacency List

insertVertex(K key):
Adjacency List

removeVertex(Vertex v):

u
  a
  c
  b
w
  d
z
v

a
  c
  b
  d

u
  v
  a

v
  w
  b

w
  c
  z
  d

z
  d=1

w
  d=3

v
  d=2

u
  d=2

b
  c
  d
  e

d
Adjacency List

incidentEdges(Vertex v):

- \( u \): \( a, c \)
- \( v \): \( a, b \)
- \( w \): \( b, c, d \)
- \( z \): \( d \)
Adjacency List

areAdjacent(Vertex v1, Vertex v2):

D = 2

D = 3

D = 1
Adjacency List

insertEdge(Vertex v1, Vertex v2, K key):
<table>
<thead>
<tr>
<th>Expressed as $O(f)$</th>
<th>Edge List</th>
<th>Adjacency Matrix</th>
<th>Adjacency List</th>
</tr>
</thead>
<tbody>
<tr>
<td>Space</td>
<td>$n+m$</td>
<td>$n^2$</td>
<td>$n+m$</td>
</tr>
<tr>
<td>$\text{insertVertex}(v)$</td>
<td>$1$</td>
<td>$n$</td>
<td>$1$</td>
</tr>
<tr>
<td>$\text{removeVertex}(v)$</td>
<td>$m$</td>
<td>$n$</td>
<td>$\deg(v)$</td>
</tr>
<tr>
<td>$\text{insertEdge}(v, w, k)$</td>
<td>$1$</td>
<td>$1$</td>
<td>$1$</td>
</tr>
<tr>
<td>$\text{removeEdge}(v, w)$</td>
<td>$1$</td>
<td>$1$</td>
<td>$1$</td>
</tr>
<tr>
<td>$\text{incidentEdges}(v)$</td>
<td>$m$</td>
<td>$n$</td>
<td>$\deg(v)$</td>
</tr>
<tr>
<td>$\text{areAdjacent}(v, w)$</td>
<td>$m$</td>
<td>$1$</td>
<td>$\min(\deg(v), \deg(w))$</td>
</tr>
</tbody>
</table>
Exam Programming C

• Two programming questions:
  • Max/min heap implementation, up tree implementation, B-Tree find
    • + some application code using the data structure
  • HashTable find, delete, and resize
    • Double hashing, linear probing, or separate chaining

• Potentially a code reading question
Traversal:

**Objective:** Visit every vertex and every edge in the graph.

**Purpose:** Search for interesting sub-structures in the graph.

We’ve seen traversal before ....but it’s different:

- Ordered
- Obvious Start
-
Traversal: BFS
Traversal: BFS

<table>
<thead>
<tr>
<th>v</th>
<th>d</th>
<th>P</th>
<th>Adjacent Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>H</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Traversal: BFS

<table>
<thead>
<tr>
<th>d</th>
<th>p</th>
<th>Adjacent Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
<td>A C B D</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>B A C E</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>C B A D E F</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>D A C F H</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
<td>E B C G</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
<td>F C D G</td>
</tr>
<tr>
<td>3</td>
<td>E</td>
<td>G E F H</td>
</tr>
<tr>
<td>2</td>
<td>D</td>
<td>H D G</td>
</tr>
</tbody>
</table>
BFS(G):
Input: Graph, G
Output: A labeling of the edges on G as discovery and cross edges

foreach (Vertex v : G.vertices()):
  setLabel(v, UNEXPLORED)

foreach (Edge e : G.edges()):
  setLabel(e, UNEXPLORED)

foreach (Vertex v : G.vertices()):
  if getLabel(v) == UNEXPLORED:
    BFS(G, v)

BFS(G, v):
Queue q
setLabel(v, VISITED)
q.enqueue(v)

while !q.empty():
  v = q.dequeue()
  foreach (Vertex w : G.adjacent(v)):
    if getLabel(w) == UNEXPLORED:
      setLabel(v, w, DISCOVERY)
      setLabel(w, VISITED)
      q.enqueue(w)
    elseif getLabel(v, w) == UNEXPLORED:
      setLabel(v, w, CROSS)
BFS Analysis

Q: Does our implementation handle disjoint graphs? If so, what code handles this?
   • *How do we use this to count components?*

Q: Does our implementation detect a cycle?
   • *How do we update our code to detect a cycle?*

Q: What is the running time?
Running time of BFS

While-loop at :19?

For-loop at :21?
BFS(G):
    Input: Graph, G
    Output: A labeling of the edges on G as discovery and cross edges

    foreach (Vertex v : G.vertices()):
        setLabel(v, UNEXPLORED)
    foreach (Edge e : G.edges()):
        setLabel(e, UNEXPLORED)
    foreach (Vertex v : G.vertices()):
        if getLabel(v) == UNEXPLORED:
            BFS(G, v)

BFS(G, v):
    Queue q
    setLabel(v, VISITED)
    q.enqueue(v)

    while !q.empty():
        v = q.dequeue()
        foreach (Vertex w : G.adjacent(v)):
            if getLabel(w) == UNEXPLORED:
                setLabel(v, w, DISCOVERY)
                setLabel(w, VISITED)
                q.enqueue(w)
            elseif getLabel(v, w) == UNEXPLORED:
                setLabel(v, w, CROSS)