CS 225
Data Structures

November 16 – Graph Implementations and Traversals
G Carl Evans
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 with a hash table, etc
- All basic ADT operations run 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: Adjacency List
Adjacency List

- **u**: a, c
- **v**: a, b
- **w**: b, c, d
- **z**: d

- **d=2**: u
- **d=2**: v
- **d=3**: w
- **d=1**: z
Adjacency List

insertVertex(K key):

![Diagram](image_url)
Adjacency List

removeVertex(Vertex v):

u
\begin{array}{c}
\text{u} \\
\text{a} \\
\text{v} \\
\text{w} \\
\text{z} \\
\end{array}
\begin{array}{c}
\text{b} \\
\text{c} \\
\text{d} \\
\text{u} \\
\text{v} \\
\end{array}
\begin{array}{c}
\text{w} \\
\text{z} \\
\text{d} \\
\text{u} \\
\text{v} \\
\end{array}
\begin{array}{c}
\text{a} \\
\text{c} \\
\text{b} \\
\text{v} \\
\text{w} \\
\end{array}
\begin{array}{c}
\text{d} \\
\text{d} \\
\text{d} \\
\text{d} \\
\text{d} \\
\end{array}

\text{a}
\text{c}
\text{b}
\text{d}
\text{u}
\text{v}
\text{w}
\text{z}
\text{u}
\text{v}
\text{a}
\text{w}
\text{b}
\text{c}
\text{d}
\text{u}
\text{v}
\text{a}
\text{w}
\text{c}
\text{w}
\text{z}
\text{d}

\text{u} \quad \text{v} \quad \text{a}
\text{v} \quad \text{w} \quad \text{b}
\text{u} \quad \text{w} \quad \text{c}
\text{w} \quad \text{z} \quad \text{d}
Adjacency List

areAdjacent(Vertex v1, Vertex v2):

- The diagram shows a graph with vertices connected by edges.
- Vertices and their connections are labeled with letters.
- Edges are indicated by arrows connecting the vertices.
- The text defines the `areAdjacent` function for two vertices.
- The function checks if there is a direct or indirect path between two vertices.
- The graph includes a triangle and a line connecting different vertices.
- The vertices have labels indicating their degree, such as `d=2` or `d=3`.
Adjacency List

insertEdge(Vertex v1, Vertex v2, K key):
<table>
<thead>
<tr>
<th></th>
<th>Edge List</th>
<th>Adjacency Matrix</th>
<th>Adjacency List</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Expressed as O(f)</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Space</strong></td>
<td>n+m</td>
<td>n^2</td>
<td>n+m</td>
</tr>
<tr>
<td><strong>insertVertex(v)</strong></td>
<td>1</td>
<td>n</td>
<td>1</td>
</tr>
<tr>
<td><strong>removeVertex(v)</strong></td>
<td>m</td>
<td>n</td>
<td>deg(v)</td>
</tr>
<tr>
<td><strong>insertEdge(v, w, k)</strong></td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><strong>removeEdge(v, w)</strong></td>
<td>1*</td>
<td>1</td>
<td>1*</td>
</tr>
<tr>
<td><strong>incidentEdges(v)</strong></td>
<td>m</td>
<td>n</td>
<td>deg(v)</td>
</tr>
<tr>
<td><strong>areAdjacent(v, w)</strong></td>
<td>m</td>
<td>1</td>
<td>min( deg(v), deg(w) )</td>
</tr>
</tbody>
</table>
mp_traversals and mp_mazes

• mp_traversals
  • Vertex Set: The pixels are the vertices
  • Edge Set: There is an edge between every n/s/e/w pixel unless the color change exceeds the tolerance
  • There are several graphs here depending on the tolerance

• mp_mazes
  • Vertex Set: The squares in the maze are the vertices
  • Edge Set: There is an edge between two vertices if canTravel() returns true
  • Once the maze is made this graph is a spanning tree of the graph with canTravel() returning true.
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
Traversals: 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 CBD</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>B ACE</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>C BADEF</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>D ACFH</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
<td>E BCG</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
<td>F CDG</td>
</tr>
<tr>
<td>3</td>
<td>E</td>
<td>G EFH</td>
</tr>
<tr>
<td>2</td>
<td>D</td>
<td>H DG</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)
BFS Observations

Q: What is a shortest path from A to H?

Q: What is a shortest path from E to H?

Q: How does a cross edge relate to d?

Q: What structure is made from discovery edges?
BFS Observations

**Obs. 1:** Traversals can be used to count components.

**Obs. 2:** Traversals can be used to detect cycles.

**Obs. 3:** In BFS, \(d\) provides the shortest distance to every vertex.

**Obs. 4:** In BFS, the endpoints of a cross edge never differ in distance, \(d\), by more than 1:

\[ |d(u) - d(v)| = 1 \]
Traversal: DFS

Diagram of a graph with nodes A, B, C, D, E, F, G, H, J, and K, illustrating Depth-First Search (DFS).
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)
DFS(G):
  Input: Graph, G
  Output: A labeling of the edges on G as discovery and back 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:
      DFS(G, v)

DFS(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)
        DFS(G, w)
      elseif getLabel(v, w) == UNEXPLORED:
        setLabel(v, w, BACK)
Running time of DFS

Labeling:
• Vertex:
• Edge:

Queries:
• Vertex:
• Edge: