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 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: Adjacency List

- Adjacency list:
  - u: v, a
  - v: w
  - w: u, v, c
  - z: w, d
Adjacency List

The diagram shows a graph with nodes labeled as follows:

- Node `u` has connections to nodes `a`, `c`, and `b`.
- Node `v` has connections to nodes `a` and `b`.
- Node `w` has connections to nodes `b`, `c`, and `d`.
- Node `z` has a connection to node `d`.

The adjacency list is represented as follows:

- Node `u`: `a`, `c`, `b`
- Node `v`: `a`, `b`
- Node `w`: `b`, `c`, `d`
- Node `z`: `d`

The diagram also includes labels for the degree of each node:

- Node `u` has a degree of 3.
- Node `v` has a degree of 2.
- Node `w` has a degree of 3.
- Node `z` has a degree of 1.

The adjacency list is given in a matrix form:

```
<table>
<thead>
<tr>
<th></th>
<th>u</th>
<th>v</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>u</td>
<td></td>
<td>u</td>
<td>v</td>
</tr>
<tr>
<td>v</td>
<td>u</td>
<td></td>
<td>u</td>
</tr>
<tr>
<td>w</td>
<td>u</td>
<td>w</td>
<td>c</td>
</tr>
<tr>
<td>z</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Adjacency List
Adjacency List

insertVertex(K key):

- Insert vertex K with key K.
- Update adjacency lists accordingly.
- Maintain the order of vertices in the list.

Graph representation:

- Vertices: u, v, w, z
- Edges: u-a, u-c, v-b, w-c, w-d, z-d
- Vertex degrees: u=2, v=2, w=3, z=1

Diagram shows the adjacencies and degrees for each vertex.
Adjacency List

removeVertex(Vertex v):

```
removeVertex(Vertex v):
```

```
(1) d = 1
(2) d = 2
(3) d = 3
```

```
<table>
<thead>
<tr>
<th>d</th>
<th>Vertex</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>u</td>
</tr>
<tr>
<td>2</td>
<td>v</td>
</tr>
<tr>
<td>3</td>
<td>w</td>
</tr>
<tr>
<td>1</td>
<td>z</td>
</tr>
</tbody>
</table>
```
Adjacency List

incidentEdges(Vertex v):

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

areAdjacent(Vertex v1, Vertex v2):
Adjacency List

insertEdge(Vertex v1, Vertex v2, K key):

- **u** with degree 2
- **v** with degree 2
- **w** with degree 3
- **z** with degree 1
<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>insertVertex($v$)</td>
<td>$1$</td>
<td>$n$</td>
<td>$1$</td>
</tr>
<tr>
<td>removeVertex($v$)</td>
<td>$m$</td>
<td>$n$</td>
<td>$\text{deg}(v)$</td>
</tr>
<tr>
<td>insertEdge($v$, $w$, $k$)</td>
<td>$1$</td>
<td>$1$</td>
<td>$1$</td>
</tr>
<tr>
<td>removeEdge($v$, $w$)</td>
<td>$1$</td>
<td>$1$</td>
<td>$1$</td>
</tr>
<tr>
<td>incidentEdges($v$)</td>
<td>$m$</td>
<td>$n$</td>
<td>$\text{deg}(v)$</td>
</tr>
<tr>
<td>areAdjacent($v$, $w$)</td>
<td>$m$</td>
<td>$1$</td>
<td>$\min(\text{deg}(v), \text{deg}(w))$</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

---

![Graph Diagrams]

![Graph Diagrams]

- •
- •
Traversal: BFS
Traversal: BFS

Adjacent Edges

<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 B A DEF</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>D A CF H</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
<td>E B CG</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
<td>F CD G</td>
</tr>
<tr>
<td>3</td>
<td>E</td>
<td>G EF 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)
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?

<table>
<thead>
<tr>
<th>d</th>
<th>p</th>
<th>v</th>
<th>Adjacent</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
<td>A</td>
<td>A B D</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>B</td>
<td>A C E</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>C</td>
<td>B A D E F</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>D</td>
<td>A C F H</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
<td>E</td>
<td>B C G</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
<td>F</td>
<td>C D G</td>
</tr>
<tr>
<td>3</td>
<td>E</td>
<td>G</td>
<td>E F H</td>
</tr>
<tr>
<td>2</td>
<td>D</td>
<td>H</td>
<td>D G</td>
</tr>
</tbody>
</table>

![Diagram showing BFS observations](image)

**Diagram:**
- **A** connects to **B**, **C**, and **D**.
- **B** connects to **C** and **D**.
- **C** connects to **D** and **E**.
- **D** connects to **F** and **G**.
- **E** connects to **F** and **G**.
- **F** connects to **G** and **H**.
- **G** connects to **H**.

The diagram represents the structure of the graph with nodes **A** through **H** and edges connecting them.
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
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: