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);
**Edge List**

**Vertex List**
- u
- v
- w
- z

**Edge List**
- u v a
- v w b
- u w c
- w z d

**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
Adjacency List

- v
  - a
  - b
  - w
    - c
    - d
  - z
    - d=2
    - w
      - d=3
      - z
        - d=1

- u
  - a
  - c

- d

- u
  - v
    - a
  - w
    - b
  - c
  - z
    - d
Key Ideas:
- O(1) lookup in vertex list
- Vertex list contains a doubly-linked adjacency list
  - O(1) access to the adjacent vertex’s node in adjacency list (via the edge list)
- Vertex list maintains a count of incident edges, or $\text{deg}(v)$
- Many operations run in $O(\text{deg}(v))$, and $\text{deg}(v) \leq n-1$, $O(n)$. 

Adjacency List

### Key Ideas:
- O(1) lookup in vertex list
- Vertex list contains a doubly-linked adjacency list
- O(1) access to the adjacent vertex’s node in adjacency list (via the edge list)
- Vertex list maintains a count of incident edges, or $\text{deg}(v)$
- Many operations run in $O(\text{deg}(v))$, and $\text{deg}(v) \leq n-1$, $O(n)$. 

#### Adjacency List

- **O(1) lookup in vertex list**
- **Vertex list contains a doubly-linked adjacency list**
- **O(1) access to the adjacent vertex’s node in adjacency list (via the edge list)**
- **Vertex list maintains a count of incident edges, or $\text{deg}(v)$**
- **Many operations run in $O(\text{deg}(v))$, and $\text{deg}(v) \leq n-1$, $O(n)$**.
<table>
<thead>
<tr>
<th>Expressed as big-O</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>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>deg(v)</td>
</tr>
<tr>
<td>areAdjacent(v, w)</td>
<td>m</td>
<td>1</td>
<td>min( deg(v), deg(w) )</td>
</tr>
</tbody>
</table>
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

[Diagram of a tree structure]

[Diagram of a more complex graph structure]
Traversal: BFS
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>ACBD</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>ABCE</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>BCDEF</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>ACFH</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
<td>BCG</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
<td>CDFG</td>
</tr>
<tr>
<td>3</td>
<td>E</td>
<td>EFGH</td>
</tr>
<tr>
<td>2</td>
<td>D</td>
<td>DFG</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
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: