Graph ADT

- Data
  - Vertices and edges
- Methods
  - `endVertices(e)`: an array of the two end vertices of `e`
  - `opposite(v, e)`: the vertex opposite of `v` on `e`
  - `areAdjacent(v, w)`: true iff `v` and `w` are adjacent
  - `getAdjacent(v)`: return a list of the vertices adjacent to vertex `v`
  - `insertVertex(o)`: insert a vertex storing element `o`
  - `insertEdge(v, w, o)`: insert an edge `(v, w)` storing element `o`
  - `removeVertex(v)`: remove vertex `v` (and its incident edges)
  - `removeEdge(e)`: remove edge `e`
  - `incidentEdges(v)`: edges incident to `v`
  - `getDegree(v)`: Returns the degree of vertex `v`

Representation of Graphs

- Graphs should be able to be represented by a logical method.
- Information to be stored
  - Vertices: number and labels
  - Edges: connectivity and weights
- Methods
  - Adjacency matrix
  - Adjacency list

Adjacency Matrix

- Representation by matrix
  - Uses a matrix of size `|V| \times |V|`
  - Unweighted graph: Each entry `(i, j)` is true if there is an edge from vertex `i` to vertex `j`, otherwise false.
  - Weighted graph: Each entry stores weights.
- Characteristics
  - Very simple, but large space requirement: \( O(|V|^2) \)
  - Appropriate if the graph is dense
Adjacency Matrix

- Undirected and unweighted graph

```
1 2 3 4 5
1 0 1 0 0 1
2 1 0 1 1 0
3 0 1 0 1 0
4 0 1 1 0 1
5 1 0 0 1 0
```

- Directed and unweighted graph

```
1 2 3 4 5
1 0 1 0 0 0
2 0 0 0 1 0
3 0 1 0 0 0
4 0 0 1 0 1
5 1 0 0 1 0
```

Adjacency Matrix

- Directed and weighted graph

```
1 2 3 4 5
1 0 2 0 0 0
2 0 0 0 6 0
3 0 7 0 0 0
4 0 0 3 0 2
5 8 0 0 5 0
```

Adjacency List

- Representation by a vector of lists
  - Motivation: Maintaining adjacent matrix is inefficient if graph is sparse.
  - A vector of lists of vertices
  - The $i$th element of the vector is a list of vertices adjacent to $v_i$.
- Complexity
  - If graph is sparse: $O(|E| + |V|)$
  - If graph is dense: $O(|V|^2)$
### Adjacency List

- **Undirected graph**

```
1  2  3  4  5
2  3  4  5  -
3  4  5  -  -
4  5  -  -  -
5  -  -  -  -
```

- **Directed graph**

```
1  2  3  4  5
2  3  -  4  5
3  4  -  5  -
4  5  -  -  -
5  -  -  -  -
```

### Performance

<table>
<thead>
<tr>
<th>Operation</th>
<th>Adjacency List</th>
<th>Adjacency Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Space</td>
<td>$n + m$</td>
<td>$n^2$</td>
</tr>
<tr>
<td>incidentEdges($v$)</td>
<td>$\deg(v)$</td>
<td>$n$</td>
</tr>
<tr>
<td>areAdjacent($v$, $w$)</td>
<td>$\min(\deg(v), \deg(w))$</td>
<td>$1$</td>
</tr>
<tr>
<td>insertVertex($o$)</td>
<td>$1$</td>
<td>$n$</td>
</tr>
<tr>
<td>insertEdge($v$, $w$,$o$)</td>
<td>$1$</td>
<td>$1$</td>
</tr>
<tr>
<td>removeVertex($v$)</td>
<td>$\deg(v)$ (undirected)</td>
<td>$n$</td>
</tr>
<tr>
<td>removeEdge($e$)</td>
<td>$1$</td>
<td>$1$</td>
</tr>
</tbody>
</table>

- $n$ vertices, $m$ edges
- no multiple edges and no self-loops

### Graph Traversal

- The most common operation is to visit all the vertices in a systematic way.
  - A traversal starts at a vertex $v$ and visits all the vertices $u$ such that a path exists from $v$ to $u$.
  - Similar to tree traversal
  - Unlike trees, we need to specifically guard against repeating a path from a cycle and considering the same vertices multiple times

- **Methods**
  - Depth First Search (DFS): based on stack
  - Breadth First Search (BFS): based on queue
Depth-First Search

- DFS traversal of a graph
  - Searches all the way down a path before backing up to explore alternatives
  - Recursive
  - Stack-based
  - Time complexity: $O(|E| + |V|)$

- DFS can be further extended to solve other graph problems:
  - Find and report a path between two given vertices
  - Find a cycle in the graph

Properties of DFS

- Properties
  - $\text{DFS}(G, v)$ visits all the vertices and edges in the connected component of $v$.
  - The discovery edges labeled by $\text{DFS}(G, v)$ form a spanning tree of the connected component of $v$. 
DFS Algorithm

Algorithm $DFS(G)$
Input graph $G$
Output labeling of the edges of $G$ as discovery edges and back edges
for all $u \in G.\text{vertices}()$
  setLabel$(u, \text{UNEXPLORED})$
for all $e \in G.\text{edges}()$
  setLabel$(e, \text{UNEXPLORED})$
for all $v \in G.\text{vertices}()$
  if getLabel$(v) = \text{UNEXPLORED}$
    $DFS(G, v)$

Analysis of DFS

- Methods
  - setLabel() for a vertex and edge
    - $O(1)$ time
    - Each vertex is labeled twice: UNEXPLORED and VISITED
    - Each edge is labeled twice: UNEXPLORED and (DISCOVERY or BACK)
  - getLabel() for a vertex and edge
    - $O(1)$ time
    - The label of each vertex and edge is checked a constant number of time.
  - incidentEdges() 
    - $O(|E|)$ time for all vertices
    - Called once for each vertex
- Time complexity of DFS
  - $O(|E| + |V|)$ provided the graph is represented by the adjacency list
  - Recall that $\sum_v \deg(v) = 2|E|$. 

Path Finding

- Algorithm
  - We use a stack $S$ to keep track of the path between the start vertex and the current vertex.
  - As soon as destination vertex $z$ is encountered, we return the path as the contents of the stack.

Algorithm $pathDFS(G, v, z)$
setLabel$(v, \text{VISITED})$
$S.\text{push}()$
if $v = z$
  return $S.\text{elements}()$
for all $e \in G.\text{incidentEdges}(v)$
  if getLabel$(e) = \text{UNEXPLORED}$
    $w \leftarrow \text{opposite}(v,e)$
  if getLabel$(w) = \text{UNEXPLORED}$
    setLabel$(e, \text{DISCOVERY})$
    $DFS(G, w)$
  else
    setLabel$(e, \text{BACK})$
$S.\text{pop}()$
return

Cycle Finding

- Algorithm
  - We use a stack $S$ to keep track of the path between the start vertex and the current vertex.
  - As soon as a back edge $(v, w)$ is encountered, we return the cycle as the portion of the stack from the top to vertex $w$.

Algorithm $cycleDFS(G, v, z)$
setLabel$(v, \text{VISITED})$
$S.\text{push}()$
for all $e \in G.\text{incidentEdges}(v)$
  if getLabel$(e) = \text{UNEXPLORED}$
    $w \leftarrow \text{opposite}(v,e)$
  if getLabel$(w) = \text{UNEXPLORED}$
    setLabel$(e, \text{DISCOVERY})$
    $pathDFS(G, w, z)$
  else
    $T \leftarrow \text{new empty stack}$
    repeat
      $o \leftarrow S.\text{pop}()$
      $T.\text{push}()$
    until $o = w$
  return $T.\text{elements}()$
$S.\text{pop}()$
return
Breadth-First Search

- BFS traversal of a graph
  - Traverse the graph level by level
  - Queue-based
  - Iterative
  - Time complexity: $O(|E| + |V|)$
- BFS can be further extended to solve other graph problems
  - Find and report a path with the minimum number of edges between two given vertices
  - Find a simple cycle, if there is one

**Example**

- Discovery edge
- Cross edge
- Unexplored vertex
- Visited vertex
- Unexplored edge
- Discovery edge
- Cross edge
Properties

- **Properties**
  - BFS\((G, s)\) visits all the vertices and edges of \(G\).
  - The discovery edges labeled by BFS\((G, s)\) form a spanning tree.
  - Level represents the number of edges to reach the node in the level from the initial node.

BFS Algorithm

Algorithm **BFS**\((G)\)
- **Input** graph \(G\)
- **Output** labeling of the edges and partition of the vertices of \(G\)
  - for all \(u \in G\).vertices()
    - setLabel\((u, \text{UNEXPLORED})\)
  - for all \(e \in G\).edges()
    - setLabel\((e, \text{UNEXPLORED})\)
  - for all \(v \in G\).vertices()
    - if getLabel\((v) = \text{UNEXPLORED}\)
      - setLabel\((v, \text{VISITED})\)
      - Q.enqueue\((v)\)
  - while \(Q\).isEmpty()
    - for all \(v \in Q\).dequeue()
      - for all \(e \in G\).incidentEdges\((v)\)
        - if getLabel\((e) = \text{UNEXPLORED}\)
          - w <- opposite\((v,e)\)
          - if getLabel\((w) = \text{UNEXPLORED}\)
            - setLabel\((e, \text{DISCOVERY})\)
            - setLabel\((w, \text{VISITED})\)
            - Q.enqueue\((w)\)
          - else
            - setLabel\((e, \text{CROSS})\)
  - 

Analysis of BFS

- **Methods**
  - setLabel\((\cdot)\) for a vertex and edge
    - \(O(1)\) time
    - Each vertex is labeled twice: UNEXPLORED and VISITED
    - Each edge is labeled twice: UNEXPLORED and (DISCOVERY or BACK)
  - getLabel\((\cdot)\) for a vertex and edge
    - \(O(1)\) time
    - The label of each vertex and edge is checked a constant number of time.
  - incidentEdges\((\cdot)\)
    - \(O(|E|)\) time for all vertices
    - Called once for each vertex
- **Time complexity of DFS**
  - \(O(|E| + |V|)\) provided the graph is represented by the adjacency list
  - Recall that \(\sum_v \text{deg}(v) = 2|E|\).
Applications

• Find the connected components
• Find a spanning forest
• Find a simple cycle in G, or report that G is a tree or forest
• Given two vertices, find a path in G between them with the minimum number of edges, or report that no such path exists

**Time complexity:** $O(|E| + |V|)$

---

**DFS vs. BFS**

<table>
<thead>
<tr>
<th>Applications</th>
<th>DFS</th>
<th>BFS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spanning forest, connected components, paths, cycles</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Shortest paths</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Biconnected components</td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

**Back edge** $(v,w)$

- $w$ is an ancestor of $v$ in the tree of discovery edges

**Cross edge** $(v,w)$

- $w$ is in the same level as $v$ or in the next level