CMSC 132: OBJECT-ORIENTED PROGRAMMING II

Graph Implementation

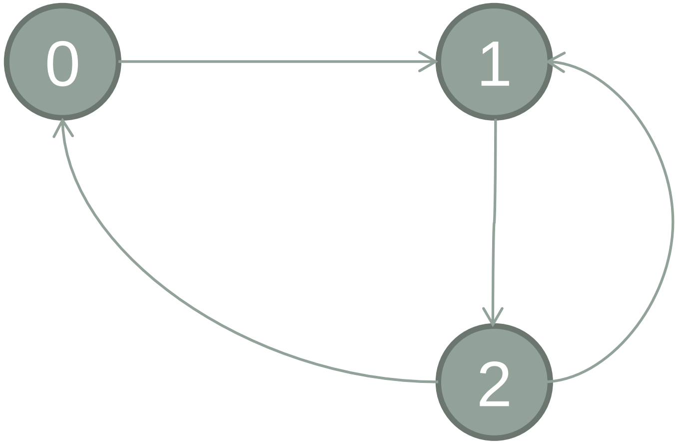
Department of Computer Science
University of Maryland, College Park

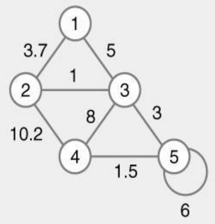
Graph Implementation

- How do we represent a graph?
- Two components
  - **Component #1** - Data each node stores about the system (e.g., if each node represents a computer, the number of users, memory capacity, etc.)
  - **Component #2** - How to represent each node and the adjacency properties (neighbors) of each one
- For **component #1** we could use a map where the key is the node'slabel and data an object with the node properties
- For **component #2** we could use
  - Adjacency matrix
    - 2D array of neighbors
  - Adjacency list/set/map
    - List/set/map of neighbors
- Which option for component #2 we use impacts efficiency/storage
- In this presentation we will discuss component #2.

Adjacency Matrix

- Single **two-dimensional array** for entire graph
- Directed Graph
  - **Unweighted graph**
    - Matrix elements → boolean
  - **Weighted graph**
    - Matrix elements → values
  - Let's see an example of each
- Undirected Graph
  - Let's see an example for unweighted graph
  - Let's see an example for weighted graph
  - For Undirected Graph
    - Only upper/lower triangle matrix needed
    - Since n_j, n_k implies n_k, n_j

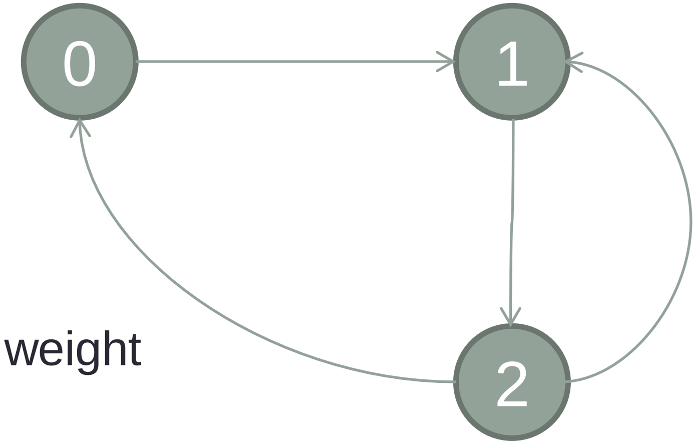




1	∘				
2	3.7	∘			
3	5.0	1.0	∘		
4	∘	10.2	8.0	∘	
5	∘	∘	3.0	1.5	6.0
	1	2	3	4	5

Adjacency List/Set/Map

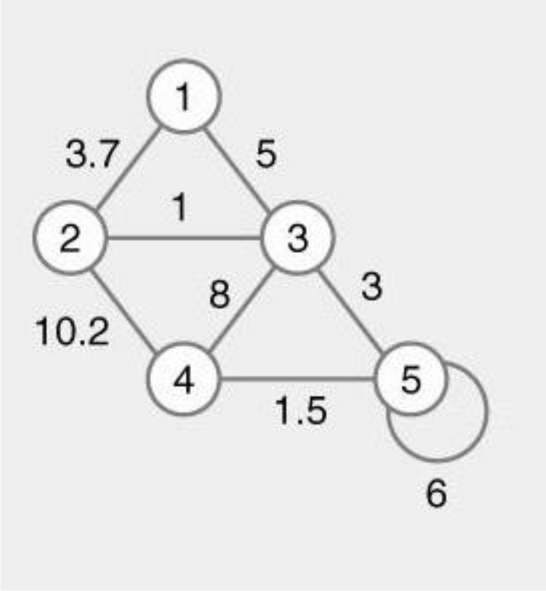
- For each node, **store neighbor information in a list, set, or map**
- The main structure can be a list, set, or map
- Directed Graph
  - **Unweighted Graph**
    - List or set of neighbors
  - **Weighted Graph**
    - Each entry keeps track of neighbor and weight
    - Easy to implement with maps
      - Maps of Maps (using HashMaps for efficiency)
  - Let's see an example of each
- Undirected Graph
  - Let's see an example for unweighted graph
  - Let's see an example for weighted graph



Additional Examples

- Examples
  - **Unweighted graph**

```
node 1: {2, 3}
node 2: {1, 3, 4}
node 3: {1, 2, 4, 5}
node 4: {2, 3, 5}
node 5: {3, 4, 5}
```



  - **Weighted graph**

```
node 1: {2=3.7, 3=5}
node 2: {1=3.7, 3=1, 4=10.2}
node 3: {1=5, 2=1, 4=8, 5=3}
node 4: {2=10.2, 3=8, 5=1.5}
node 5: {3=3, 4=1.5, 5=6}
```

Graph Properties

- Graph Density
  - Ratio edges to nodes (dense vs. sparse)
  - For adjacency matrix many empty entries for large, sparse graph
- Adjacency Matrix
  - Can find individual edge (a,b) quickly
  - Examine entry in array edge[a, b]
    - Constant time operation
- Adjacency list / set / map
  - Can find all edges for node (**a**) quickly
  - Iterate through collection of edges for node (**a)**
    - On average E / N edges per node

Complexity

- Average Complexity of Operations
  - For graph with N nodes, E edges

Operation	Adj Matrix	Adj List	Adj Set/Map
Find edge	**O(1)**	**O(E/N)**	**O(1)**
Insert edge	**O(1)**	**O(E/N)**	**O(1)**
Delete edge	**O(1)**	**O(E/N)**	**O(1)**
Enumerate edges for node	**O(N)**	**O(E/N)**	**O(E/N)**

Choosing Graph Implementations

- Factors to Consider
  - **Graph density**
  - **Graph algorithm**
    - Neighbor based

```
For each node X in graph
   For each neighbor Y of X  // adj list faster if sparse
      doWork( )
```

    - Connection based

```
For each node X in …
   For each node Y in …
      if (X,Y) is an edge  // adj matrix faster if dense
         doWork( )
```