References for Today’s Lecture

• Required reading
  – Sections 17.2-17.5

• References
  – AMO Sections 2.3
  – CLRS Section 22.1
Network Representation

• Our goal is to develop “efficient” algorithms → reasonable computation time.

• The main factors affecting efficiency are
  – The underlying algorithm
  – Data structure for storing the network

• The same algorithm may behave much differently with different graph data structure.

• What information do we need to store?
  – network topology (structure of nodes and arcs)
  – associated data (costs, capacities, supplies/demands)

• What are the important operations we might need to perform with a network data structure?
Common Representations

• Data structures
  – Node-Arc Incidence Matrix
  – Node-Node Adjacency Matrix
  – Adjacency List
  – Forward Star (Reverse Star)

• How do we evaluate a data structure?
Aside: Multiarcs and Loops

• *Multiarcs* are two or more arcs with the same tail and head nodes.

• A *loop* is an arc with the property that its tail and head nodes are the same.

• Generally we will assume that our networks do not contain parallel arcs or loops.

• The existence of such arcs can cause problems with standard data structures.
Example Graph

![Graph Image]
(Node-Arc) Incidence Matrix

- \( n \times m \) matrix denoted \( N \).
- One row for each node and one column for each arc.
- For each arc \((i, j)\), put +1 in row \(i\) and -1 in row \(j\).

\[
\begin{array}{cccccccc}
(1, 2) & (1, 3) & (2, 3) & (2, 4) & (3, 2) & (3, 4) & (3, 5) & (4, 5) \\
1 & 2 & 3 & 4 & 5
\end{array}
\]
(Node-Arc) Incidence Matrix

• What is the size of the matrix?
• How many entries are non-zero?
• What information do we get by reading across a row?
• Is this a space efficient representation?
• How about other operations?
(Node-Node) Adjacency Matrix

- $n \times n$ matrix denoted $H$
- one row for each node and one column for each node
- entry $h_{ij} = 1$ if arc $(i, j) \in A$ (0 otherwise)

\[
\begin{pmatrix}
1 & 2 & 3 & 4 & 5 \\
1 \\
2 \\
3 \\
4 \\
5
\end{pmatrix}
\]
(Node-Node) Adjacency Matrix

- What is the size of the matrix?
- How many entries are non-zero?
- What data structures might we use to store arc costs and capacities?
- Is this a space efficient representation?
- What operations are most efficient with this data structure?
Adjacency List

• Adjacency list of node $i$, $A(i)$, is a list of the nodes $j$ for which $(i, j) \in A$
• List stored as a \textit{linked list}.
• Need one linked list of length $|A(i)|$ for each node.
• Cell can store additional fields such as arc cost and capacity
• Is this a space efficient representation?
• What operations are most efficient with this data structure?
Forward Star

- Stores node adjacency list of each node in one large array
- Associates a unique sequence number with each arc using a specific order starting with arcs outgoing from node 1, then node 2, etc.
- Stores tail information about each arc in tail array, head information in head array, etc.
- Maintains a pointer for each node that indicates the smallest numbered arc in the arc list for that node.
- For consistency, set pointer(1) to 1 and pointer($n + 1$) to $m + 1$.
- What are the advantages of this representation?
Reverse Star

- Similar to a forward start except that arcs are sequenced starting with arcs incoming from node 1.
- The two representations can be maintained side-by-side if necessary.
Miscellaneous Issues

• Parallel Arcs
  – Why would we need parallel arcs?
  – Which representation(s) could accommodate them?

• Undirected Network
  – What needs to change?
    * Node-Arc Incidence Matrix
    * Node-Node Adjacency Matrix
    * Adjacency List
  – What needs to happen when we update \((i, j)\)?
# Summary of Representations

<table>
<thead>
<tr>
<th>Representation</th>
<th>Storage Space</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Incidence Matrix</td>
<td>$nm$</td>
<td>1. Space inefficient</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2. Expensive to manipulate</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3. MCFP constraint matrix</td>
</tr>
<tr>
<td>Adjacency Matrix</td>
<td>$kn^2$</td>
<td>1. Suited to dense networks</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2. Easy to implement</td>
</tr>
<tr>
<td>Adjacency List</td>
<td>$k_1n + k_2m$</td>
<td>1. Space efficient</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2. Efficient to manipulate</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3. Suited to dense and sparse</td>
</tr>
<tr>
<td>Forward Star</td>
<td>$k_3n + k_4m$</td>
<td>1. Space efficient</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2. Efficient to manipulate</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3. Suited to dense and sparse</td>
</tr>
</tbody>
</table>

Table 1: From Ahuja et al. Figure 2.25
Graph Interface Class: Adjacency Lists

class Edge:
    def __init__(self, i, j):
        self.source = i
        self.destination = j

    def get_end_points(self):
        return self.source, self.destination

class Vertex:
    def __init__(self, n):
        self.name = n
        self.out_neighbors = {}
        self.in_neighbors = {}

    def get_out_neighbors(self):
        return self.out_neighbors

    def get_in_neighbors(self):
        return self.in_neighbors
class Graph:
    def __init__(self, nodes, edges):
        self.nodes = {}

    def get_nodes(self):
        return self.nodes.keys()

    def add_node(self, n)
    def add_edge(self, e)
A Client Function for Printing a Graph

• Here’s an example of a standard way in which the graph interface class is used.

• Here, we print out a graph by enumerating all the edges incident to each vertex.

def print(G):
    for n in G.get_nodes():
        print n, ":",
        for i in n.get_out_neighbors():
            print i
    print