Network Science	
Lecture 2: July 21, 2009	Lecturer: Maleq Khan
Scribe: Katherine Wendelsdorf	

Introduction to Networks: Graph Theory II

2.1 More Graph Definitions

2.1.1 Definitions of features of Graph *G*

Definition 1 *Complement Graph* (Figure 1)
A graph, $G'(V', E')$ is a **complement graph** of graph $G(V, E)$ only in the case that:
i) $V' \in V$
ii) $\{u, v\} \in E' \iff \{u, v\} \notin E$ for all node pairs $u, v \in V'$.
A complement graph of G is shown in Figure 2.1.

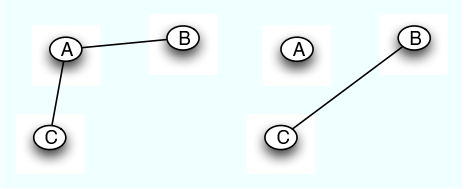

Figure 2.1: Graph G and its complement

Definition 2 *Subgraph* (Figure 2)
A graph, $G'(V', E')$ is a **subgraph** of graph $G(V, E)$ only in the case that:
i) $V' \in V$
ii) $E' \in E$
A subgraph of graph G is shown in Figure 2.2. Note that an edge that exists in G' must exist in G, but the converse is not true. An edge in G does not have to be present in G' for it to be a subgraph of G.

Definition 3 *Induced Graph*
A graph, $G'(V', E')$ is an subgraph of graph $G(V, E)$ **induced** by node subset $V' \in V$ only in the case that:
i) $V' \in V$
ii) $E' \in E$
iii) $u, v \in E' \iff \{u, v\} \in E$
Figure SUBGRAPHS shows an induced subgraph of G compared to a general subgraph. As mentioned in the definition of *subgraph*, an induced subgraph must contain all edges incident to all nodes

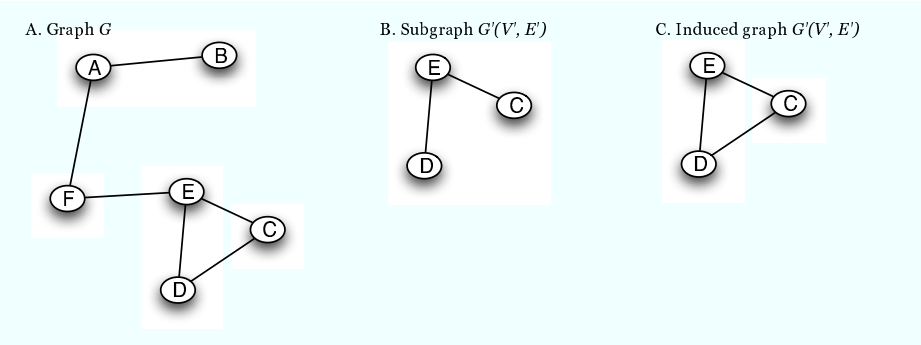

Figure 2.2: General Subgraph and Induced Subgraph

$v \in V'$ that are present in graph G. In other words, if there is a pair of nodes $u, v \in V'$ of graph G' and edge $\{u, v\} \in E$ of graph G then there must be an edge $\{u, v\} \in E'$ of the induced subgraph $G'[V']$. $G[V']$ is often written for the graph induced by node set $V' \in V$ on graph G to distinguish it from a general subgraph, $G'(V', E')$.

Definition 4 *Clique*
A **clique** is a subgraph of G that is complete. The **maximal clique** is the clique with the largest set of nodes $V' \in V$ of graph G.
Note: This definintion may be different across disciplines where a clique may be a subgraph with some maximized characteristic.

Review: A **connected graph** is one in which any node can be reached by a path from any other node. A **simple graph** is one that has no loops or parallel edges.

Definition 5 *Component*
A subgraph, C, of graph G is a **component** of G if:
i) C is an *induced* subgraph of G.
ii) C is connected
iii)C is not a subgraph of a larger connected subgraph of G.
For example, a *complete* graph has one component. A **giant component** may be defined as a component that consists of a large number of nodes.

2.2 Trees

Definition 6 *Tree*
A **tree** is a connected, acyclic, simple graph. It may be directed or undirected and has $n - 1$ edges. Figure 2.3 illustrates the anatomy of a tree. Terminology used to describe the components of a tree is described below.

Tree Terminology
Parent node: In a directed tree a parent node i is a node that points to another node called the

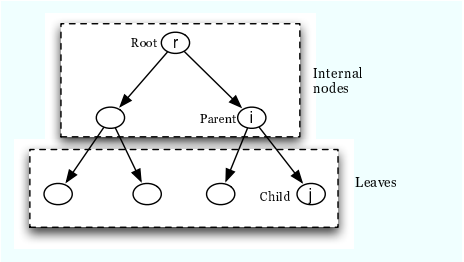

Figure 2.3: Terminology for tree anatomy

child node, j. More formally, node i is a parent node if $outdegree(i) \geq 1$. Such nodes may also be referred to as an *ancestor* of child node j.
Child node: In a directed tree, node j is a child node if $indegree(j) \geq 1$.
Root: In a directed tree, a root node is one that has no ancestors and points to other nodes. More formally, a node r is a root if, and only if: I) $indegree(r) = 0$ and ii) $outdegree(r) \geq 1$.
Leaves: A node is a *leaf* if it does not point to any other nodes. Node j is a leaf if $outdegree(j) = 0$
Internal nodes: An *internal* node that is not a leaf node. Node i is an internal node if $outdegree(i) \geq 1$.
Binary Tree: A binary tree is one in which each internal node has ≤ 2 child nodes.
Height: The *height* of a tree is the longest distance from the root to any leaf node.

Therefore, by definition:
• A *root* is a *parent* node.
• All *parent* nodes are *internal* nodes
• All *leaf* nodes are *child* nodes

2.3 Graph Data Structures

The set of nodes, V, and set of edges, E, that connect them on graph G may be stored as an **adjacency list** or **adjacency matrix**. the following examples represent graph G of Figure 2.4 using the two structures. Briefly discussed are the advantages and disadvantages of each.

Example 1: Adjacency matrix, A

	A	B	C	D
A	0	1	0	0
B	1	0	1	0
C	0	1	0	1
D	0	0	1	0

$A(i,j) = 1 \iff \{i,j\} \in E$

In an adjacency matrix representation of graph G a cell(i,j) is assigned a value of 1 when there is an edge $\{i,j\} \in E$ of G and 0 if the edge is not present. In the case of a weighted graph the weight

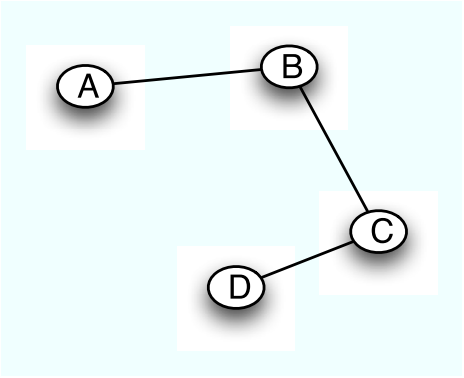

Figure 2.4: Example graph G

of the edge, w_{ij}, is typically assigned. An *advantage* of the adjacency matrix is that any specific edge can be found in constant time. A potential *disadvantage* is that for a graph with n nodes, the matrix requires n^2 space. This is too expensive in the case of a *sparse* graph which contains a large set of nodes V, but a small set of edges E such that $|V| >> |E|$.

Example 2: Adjacency List, A

A: B
B: A, C
C: B, D
D: C

In an adjacency list each node is associated with a list of its neighbors. The *advantage* of an adjacency list is that the structure takes up less space than a matrix. The *disadvantage* is that search for a specific edge $\{u, v\}$ takes $O(n)$ time and may be too expense in the case of a large set of nodes V.

2.4 Searching a Graph

There are multiple reasons to search a set of nodes and edges of a graph including a search for the shortest path between two specific nodes or a node (or edge) with a specific property. Similarly, there are many ways to carry out such a search. Here we discuss the Breadth First Search (BFS) strategy and a sample algorithm for imlementation.

2.4.1 Breadth First Search

Take the following scenario: You are an actor and you want to get noticed by a powerful studio executive, Mr. Big. As most people would agree that personal introduction is the most effective means of being noticed you wish to exploit your current social network to find someone who knows

someone whoknows someone...who knows Mr. Big. You may already see that this translates quite nicely in to a search for the shortest path betwen two nodes in a social network, the nodes being you, node v, and Mr. Big, node B. There are numerous strategies one may employ to find this path. Here we introduce **Breadth First Search** strategy.

Breadth First Search
The idea: You ask all your current contacts to list each of the people they can introduce you to. After the first round of introductions, you then ask all of those people who they can introduce you to. After this second round of introductions you ask all of *those* people who they can introduce you to and so on until you finally meet some one who can introduce you to Mr. Big.

In other words, to search for a specific node or edge in graph G, choose a node v and check each node u such that $u \in N(v)$. If you find the desired node, b, among $N(v)$ then you are done. Otherwise check the neighbors of neighbors of v, all $s \in N(u)$ for every $u \in N(v)$.

Example algorithm for BFS implementation:
Maintain a queue, Q, of nodes that are to be checked. A **queue** is a first-in-first-out (FIFO) data structure. Initially Q only consists of starting node v_0. If v_0 has desired property (it is node b) then the search is finished. Otherwise, add all neighbors of v_0 to queue Q. For each $u \in N(v_0)$, now in Q, if it is not the desired node, then add all nodes $\in N(u)$ to queue. Due to FIFO property of the queue data structure, these will be checked after all $u \in N(v_0)$.

$Q \leftarrow \{v_0\}$
while Q is not empty **do**
  $u \leftarrow$ dequeue Q {examine node at front of Q}
  if u is desired node **then**
    return u {search is done}
  else
    Enqueue all nodes $\in N(u)$
  end if
end while

Such a search will result in a **Breadth First Tree** that consists of the component of G, G', containing v and shortest paths $p\{u, v\}$, from v to all other nodes $u \in G'$. This can be done by storing each node removed from the queue and its neighbors in an adjacency list or matrix upon each iteration.