Module 1 Part D - Elementary Graph Algorithms

Athens State University

January 5, 2016

Outline

Contents

1 Terminology

Directed and undirected graphs

Directed graph A *directed graph* G is a pair $< V, E >$ where V is a finite set and E is a binary relation on V

- The set V is the *vertex set* of G
- The set E is the *edge set* of G. An element of E is called an *edge* on G.

Undirected graph An *undirected graph* $G =< V, E >$ is a graph where the edge set E consisted of unordered pairs of vertexes.

Thought question: Is a tree a directed or undirected graph?

Note that we will use the terms *vertex* and *node* interchangeably.

Directed and undirected graphs: more terms

Adjacent If (u, v) is an edge of G, then v is *adjacent* to u.

Degree The *degree* of a vertex is the number of edges entering that vertex

- We have concepts of *in-degree* and *out-degree*

Path A *path* of *length* k from vertex u to $u^{'}$ in a graph G is the sequence of vertexes you travel to get from u to $u^{'}$.

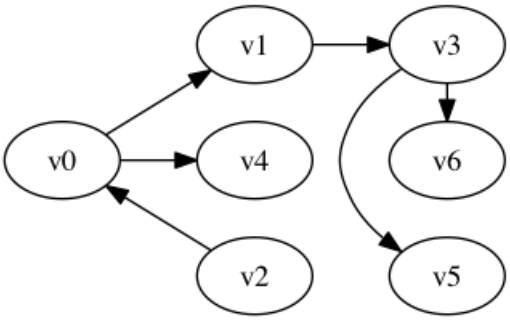
- We say that the path *contains* the vertices and edges in the path
- If there is a path p from vertex a to vertex b, then b is *reachable* from a.
- A path is *simple* if all paths in a vertex are distinct.

Directed and undirected graphs: more terms

Cycle A path $< v_0, v_1, \ldots, v_k >$ forms a *cycle* if $v_0 = v_k$

- A cycle is simple if nodes are distinct.
- A graph with no cycles is *acyclic*.

Applying the definitions of a graph



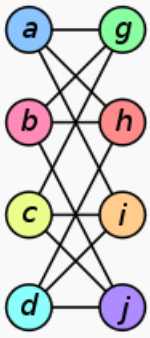
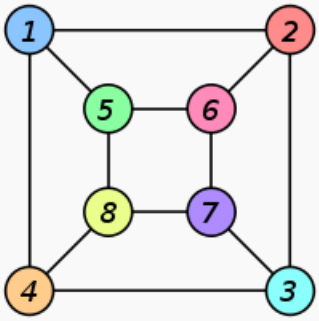
Directed and undirected graphs: more terms

- An undirected graph is *connected* if every pair of vertices is connected by a path.
- A directed graph is *strongly connected* if every two vertices are reachable from each other. The *strongly connected components* of a graph are those vertices that are mutually reachable

Directed and undirected graphs: more terms

- Two graphs G and G' are *isomorphic* if we can relabel the vertices of G to be the vertices in G' with the same edge structure.
- In a directed graph, a *neighbor* of a vertex if for vertices u and v, both $(u,v) and (v, u) are in E.

Graph Isomorphism Example

Graph G	Graph H	An isomorphism between G and H
		f(a) = 1 f(b) = 6 f(c) = 8 f(d) = 3 f(g) = 5 f(h) = 2 f(i) = 4 f(j) = 7

Determining of two graphs are isomorphic is important because it gives us a way to determine if two graphs are equivalent. Equivalency is a difficult problem because we need to separate properties inherent from the structure of the graph from properties of the graph representation.

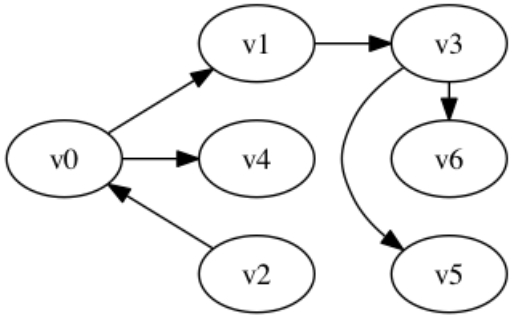


Figure 1: A Sample Graph

Directed and undirected graphs: special types

- A *complete graph* is an undirected graph in which every pair of vertices is adjacent.
- A *forest* is an acyclic, undirected graph and a connected, acyclic, and undirected graph is a *free tree.*
- A *dag* is a directed, acyclic graph.

2 Graph Traversal

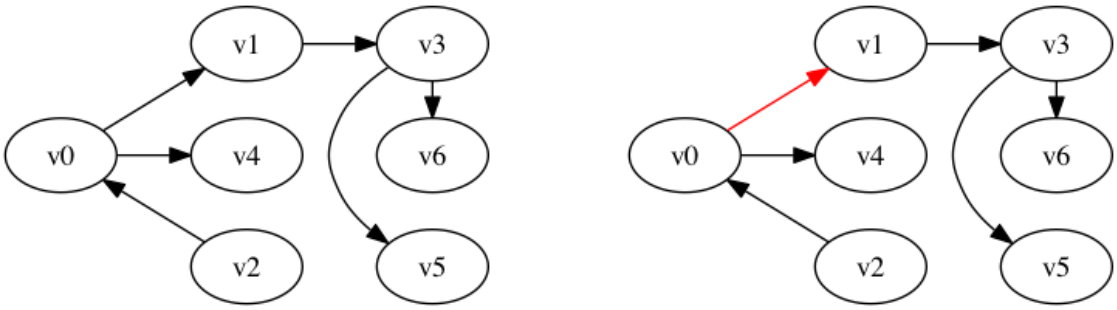
Graph Traversal vs. Tree Traversal

- For binary trees, traversal of a tree was defined as starting at the root node and visiting all nodes in the tree
  - Two types: depth-first and breath-first
- There is a similar idea for graphs: walk through the graph and visit all nodes that can be found from some starting point
  - We will have the idea of both depth-first and breath-first traversals
  - We have the additional complication of avoiding cycles

Consider the graph in Figure 1. The nodes $v0$, $v1$, and $v3$ form a cycle. If we don't do anything special to detect the existence of a cycle, then any attempt to traverse the graph will result in an infinite loop. What must be done is that we need to keep track of the nodes that we have visited and not revisit any nodes that are so marked. How we keep track of visited nodes will be what separates a depth-first traversal from a breath-first traversal.

Depth-First Graph Traversal

- In this case, we will use a stack to keep track of un-visited nodes
- We will mark each node we visited (for instance, by keeping an extra field in the node structure)
  - In general, we will never move forward to a marked node
- Most implementations will be recursive



(a) t = 1

(b) t = 2

(c) t = 3

(d) t = 4

(e) t = 5

(f) t = 6

Figure 2: Depth-First Traversal of a Graph

Breath-First Traversal

- In this case, we place discovered nodes into a queue, beginning with our starting node
- The traversal is a two step process that repeats until the queue is emptied:
  - Remove a node from the queue
  - For each unmarked neighbor of the node, process and mark the neighbor and place the neighbor into the queue

3 Ways to represent graphs

Ways to represent graphs

Two ways to represent a graph

- Adjacency list
- Adjancency matrix

Ways to represent graphs : Adjacency lists

For a graph $G = (V, E)$,

- Start with an array Adj of list, one for each vertex of V
  - For each element u in G, $Adj[u]$ contains pointers to all the vertices in a list s.t. (u, v) is in E
- If G is a directed graph, what is the sum of lengths of each element of Adj?
- If G is an undirected graph, what is the sum of lengths of each element of Adj?
- Given your answers, what is the space efficiency (Big-Oh) of adjacency lists?

Ways to represent graphs : Adjacency matrix

For a graph $G = (V, E)$

- Number each of the vertices in the graph starting from 1
- Create a $|V| * |V|$ matrix $A = (a_{ij})$ such that $a_{ij} = 1$ when (i, j) is in E and 0 other wise.
- Note the symmetry that occurs in the matrix A
  - A is its own transpose
  - Means in some cases can store only half the array

Ways to represent graphs : Adjacency matrix

- No quick way to find out if (u, v) is present in the graph if you're using adj. lists.
- Adj. matrices easier to use, but graph can't be too large
- In many cases, want to do bit-wise storage of adj. information in adj. matrix.

4 Key Points

Key Points

- Definition of a graph
- Key terms and concepts
- Type of graphs
- Ways to represent graphs in memory