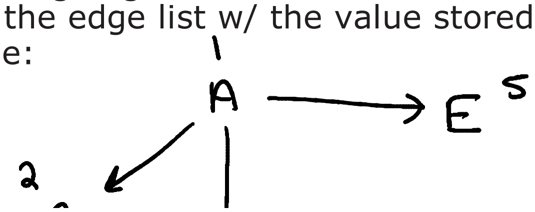
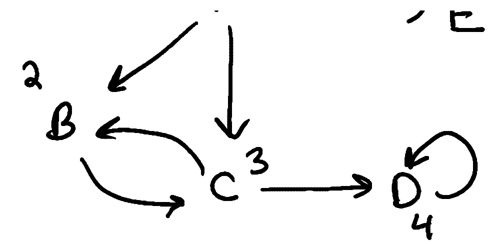
Graphs & Digraphs

Tuesday, November 06, 2007
10:34 PM

16.1 Directed Graphs (digraphs)

- like a tree but w/ no root node & no guarantee of paths between nodes
- consists of:
  - nodes/vertices - a set of elements
  - directed edges/arcs - a set of connections between nodes
    - incoming edges & outgoing edges
      - in-degree - number of incoming edges for a node
      - out-degree - number of outgoing edges for a node
- cyclic vs acyclic
- many applications
  - networks
  - dependencies
  - routes
- Digraph ADT
  - Data: set of nodes & set of edges
  - Operations
    - construct an empty digraph
    - check if empty
    - destructor
    - insert a node
    - insert an edge
    - delete a node & all its incoming & outgoing edges
    - delete an edge
    - search for a value starting at a green node
- Representing the data
  - Adjacency-Matrix representation
    - number the nodes 1 to n
    - have an n x n matrix of ints
      - [row i, col j] = 1 for edge from i to j
      - [row i, col j] = 0 for no edge
    - can have a weighted digraph by using weight instead of 1
    - can determine in-degree & out-degree easily
      - in-degree for node m is sum of set edges in m-th column
      - out-degree for node m is sum of set edges in m-th row
    - need a second 1D array of size n to store the values in each node
    - issue: wasted space when graph is sparse (few edges)
  - Adjacency-list representation
    - less wasted space for sparse graphs
    - use an array or list for each node that represents the outgoing edges
    - pair the edge list w/ the value stored in the node
  - Example:





Matrix

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

List

A → 2 → 3 → 5
B → 3
C → 2 → 4
D → 4
E

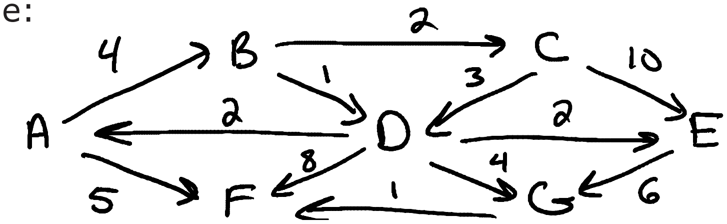
16.2 Searching & Traversing Digraphs

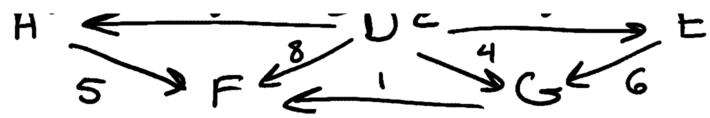
- tree traversals are easier because all nodes reachable from root
  - no such guarantees w/ digraphs
- may not be able to reach all nodes from any starting node
  - how to still visit each node once?
- two methods for searching
  - depth-first search
    - go until a "leaf" is reached then backtrack
  - breadth-first search
    - visit all children of a node first then children's children
- Depth-First Search
  - backtracking only possible if we can know which paths have already been taken
    - mark nodes as processed
  - when backtracking, go back to previous node & see if it has any unprocessed children
    - continue this check recursively until unprocessed child found
    - then process that child & any unprocessed nodes it reaches
  - a "leaf" is a node that has no unprocessed children
  - after processing all reachable node from given starting node, some nodes may be unprocessed
    - unreachable nodes from that starting node
  - Pseudocode
    - visit the starting node v
    - mark v as processed
    - for each node w that is adjacent to v
      - if w is unprocessed
        - call depth-first search w/ w as starting node
- Breadth-First Search
  - visit all children & then process each child's children in order
    - outputs a tree level by level
  - again, some nodes may be unreachable
  - Pseudocode

- visit the start vertex
- mark start vertex as processed
- put start vertex in a queue
- while the queue is not empty
  - remove vertex v from queue
  - for all vertices w that are adjacent to v
    - if w is unprocessed
      - visit w
      - mark w as processed
      - put w in the queue

Traversals
- repeatedly call one search method until all nodes are processed
- Pseudocode
  - initialize processed array w/ false for each node
  - while nodes are unprocessed
    - select a starting node from unprocessed nodes
    - call one of the searches w/ starting node

Shortest Path
- find shortest path between any two nodes
- Dijkstra's algorithm commonly used to find shortest path
  - book's method is for unweighted digraphs
    - visit start & label w/ 0 & mark
    - initialize distance to 0
    - add start to a queue
    - while destination is not processed and queue is not empty
      - remove v from queue
      - if label of v > distance
        - increment distance
      - for each node w that is adjacent to v
        - if w has not been processed
          - visit w & mark
          - label w w/ distance+1
          - add w to queue
      - end for
    - end while
    - if destination is not processed
      - issue "unreachable" error
    - else find path p[0]... by
      - initialize p[distance] to destination
      - for each k from distance-1 to 0
        - find a node p[k] that is adjacent to p[k+1] & has label k
  - for weighted graphs:
    - find closest child to start
    - see if adding a child of the child is still less than using another child of start
    - continue this sort of search until destination is reached
    - Example:





Find path from B to F

Initial

v	known	d	prev	
A	F	∞	0	
B	T	0	0	start
C	F	∞	0	
D	F	∞	0	
E	F	∞	0	
F	F	∞	0	destination
G	F	∞	0	

Now fill d for nodes reachable from B

C	F	2	B
D	F	1	B

Select smallest d & mark as known

D	T	1	B

Fill d in closer via D

A	F	3	D	
B	T	0	0	start
C	F	2	B	
D	T	1	B	
E	F	3	D	
F	F	9	D	destination
G	F	5	D	

C is smallest d, mark as known & update d for path via C

A	F	3	D	
B	T	0	0	start
C	T	2	B	
D	T	1	B	

E	F	3	D	
F	F	9	D	dest
G	F	5	D	

Select A & E & update d

A	T	3	D	
B	T	0	0	start
C	T	2	B	
D	T	1	B	
E	T	3	D	
F	F	8	A	dest
G	F	5	D	

Select G

F	F	6	G
G	T	5	D

Select F, done

F reached in distance 6

via F - G - D - B

16.3 Graphs

- Undirected graph -edges are bidirectional
- No edges to self allowed like in digraph
- Graph ADT
  - Data: set of nodes & set of edges between two distinct nodes
  - Operations
    - Construct empty
    - check if empty
    - destructor
    - insert a node
    - insert an edge
    - delete a node & associated edges
    - delete an edge
    - Search from a given node
- Representation
  - Adjacency matrix is symmetric
    - edge i to j means also edge j to i
    - inefficient representation
  - Adjacency list also has each edge twice

- Edge-List Representation
  - have an edge node
    - contains the two vertices
    - an optional label or weight
    - two pointers to other edges
      - pointer 1 to another edge for node 1
      - pointer 2 to another edge for node 2
- Connectedness
  - a connected graph has a path to all other nodes from a given node
  - can be checked by doing a search from any node
    - if all nodes processed, graph is connected
    - works because all edges bidirectional