Mat 3770
Week 8

Spring 2014

Week 8 — Student Responsibilities

- Reading: Chapter 3.2 (Tucker), 10.3 (Rosen)
- Homework
- Attendance **re-FRESH-ingly** Encouraged

Section 3.2 Depth–First and Breadth–First Search

- **Tree-based searches** abound in applications, and are amenable to computerized solutions.
- Depending upon the application, we may wish to find
  1. **one** solution
  2. **all** solutions, or
  3. an **optimal** solution
- To organize the enumeration of possible solutions:
  1. let the sequential choices be **internal nodes** in a rooted tree
  2. the solutions and "dead ends" be the **leaves**
- The big challenge is to be sure to check that all possible ways to generate a solution are investigated: that the enumeration of possibilities is complete.

Two Basic approaches to Tree Enumeration

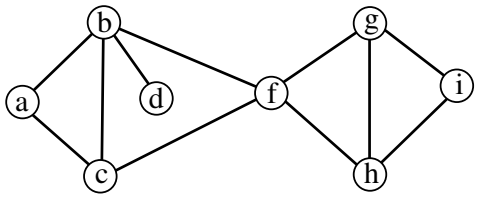
Graph **traversal** algorithms differ by the **order** in which nodes are "**visited**."

The two best known methods are:

1. **Depth–First Search** (or **Backtracking**)
2. **Breadth–First Search**

Example: DFS(G, f)

- **numbers** *will indicate visitation order*
- **dotted** *lines will represent unmarked edges*
- *use* **alphabetic order** *to choose among next nodes to visit*



DFS Algorithm

Given: An undirected graph $G = (V, E)$, and a vertex $v \in V$
Goal: Visit all vertices and edges of G starting with v.

```
Procedure DFS (G, v)
// PRE:  G is connected
// POST: every vertex and edge of G
//       is visited

begin
  Mark (v)
  Visit (v)  // do application processing
  for all unmarked edges <v, w> do
     if w is unmarked then
        MarkEdge (v, w)
        DFS (G, w)
     VisitEdge (v, w) // process if necessary
end
```

Lemma

- If G is connected, procedure DFS visits every vertex and every edge.
- Edges are visited or checked at most twice,
- DFS is called once for each vertex, and
- the marked edges form a tree (called the DFS tree).

Proof:

- Recall: $\sum \deg(v) = 2|E|$, i.e., the sum of the degrees of all nodes is twice the number of edges.
- For any v, DFS(G, v) is called exactly once — when v is unmarked.
- So, if $< v, w >$ is an edge, it's visited and marked at that time by DFS(G, v) *or* DFS(G, w), whichever node is visited first.
- Note also, this means the `FOR` loop is executed a **total** of $2|E|$ times over **all** calls of DFS.
- As for marked edges: **Claim**: DFS(G, v) marks edges which form a tree with root v. **[Proof by induction on the number of unmarked vertices in G.]**

Lemma

If T is the tree formed by DFS (G, v), then for each edge $e \in E_G$, either:

- $e \in T$, or
- e connects a vertex in T to an ancestor or descendant in T.

I.e., there are no **cross–edges** in T.

Proof.

Let $e = < u, W >$. if u is marked first, then w is marked during DFS (G, u).

So, by the previous lemma (vertices are visited only once), w is a descendant of u.

Complexity of DFS

- Assume *visit(v)* takes O(1) time
- How long to find next neighbor of v?
  - depends on implementation of G
  - let's assume O(1) time
- Then clearly each edge is visited at most twice and each vertex once, so time complexity is:

$$O(|V| + 2|E|) = O(|V| + |E|)$$

DFS Drawback

- Suppose we don't want to visit **all** the vertices — we merely want to find one with a certain property: i.e., we want to be able to **stop**.
- **Idea**: **non–recursive** version of DFS using a stack.
- We'll assume we're not interesting in visiting every edge.

Non–recursive DFS Algorithm

```
Procedure DFS2 (G, v)
// PRE:  G is connected or we only visit
//       subgraph in which v is found
// POST: every vertex of G is visited

begin
  CreateStack(S)
  Mark(v)
  Push(S, v)
  while not Empty(S)
     x = Pop(S)
     while x has unvisited neighbor (w)
        Mark(w)
        Push(S, w)
end
```

Application: Counting Connected Components

```
Procedure Components (G, cnt)
// PRE:  no vertex in G is marked
// POST: cnt is number of components in G

begin
  cnt = 0
  while there's an unmarked vertex v
     DFS(G, v)
     cnt = cnt + 1
end
```

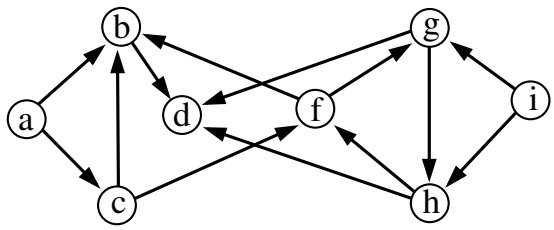

DFS in DiGraphs

Works the same as for undirected graphs, but there are differences with respect to the DFS tree — it may not be a tree, but a **forest**.

There are **four** possible types of edges in the DFS forest:

- the **tree** edges: of the DFS tree
- **back** edges: edges to ancestors in the tree
- **forward** edges: edges to descendants in the tree
- **cross** edges: all others

DiGraph Example: DFS(G, f)



Lemma

If $< u, w >$ is an edge of a directed graph G with DFS forest F and DFSnumber(u) $<$ DFSnumber(w), then w is a descendant of u in the DFS tree.

Proof:

- Vertex u was considered (visited) before w, so we would eventually process edge $< u, w >$ if no other path to w from u exists.
- Hence, cross–edges go from **higher** numbers to **lower** numbers.

Lemma

Suppose there is a directed path $v_1, v_2, \ldots, v_n$ in G such that v_1 has lowest DFSnumber in $\{v_1, \ldots, v_n\}$. Then, $\forall\ 1 < i \leq n$, v_i is a descendant of v_1.

Proof:

Note v_1 and all descendants have DFSnumbers in the interval [a..b], and DFSnumber(v_1) = a.

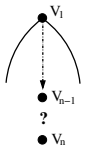
Proof will be by **induction on the number of nodes in the path**.

BC Let n = 2, and $v_1 \rightarrow v_2$. By the last lemma, v_2 is a descendant of v_1.

IH Assume for directed paths with $n - 1$ nodes where v_1 has lowest DFSnumber, that $v_2, \ldots, v_{n-1}$ are descendants of v_1.

IS Show lemma is true for paths with n nodes.

If v_n is not a descendant of v_1, and the DFSnumber(v_1) $>$ DFSnumber(v_n), then DFSnumber(v_n) $>$DFSnumber(v_{n-1}) since v_{n-1} is closer to v_1 in the path (and hence will be processed before v_n).



Consider the edge $v_{n-1} \rightarrow v_n$. Is it:

- in the tree? — assumed no, since not a descendant of v_1
- forward edge? — no, for same reason
- backward edge? — no, since DFSnumber(v_n) > DFSnumber(v_{n-1})
- cross edge? — no, since DFSnumber(v_{n-1}) < DFSnumber(v_n)

We reach a contradiction, and thus v_n **is** a descendant of v_1.

Application: Acyclic Digraphs

- Suppose a program consists of many procedures.
- Let us say procedure X **depends on** Procedure Y if there are procedures $X = X_1, X_2, \ldots, X_n = Y$ such that X_i calls X_{i+1} $\forall\ i < n$.
- Question: Does any procedure **depend on** itself?
- Why care?
  - As programmers, this is a sort of recursion
  - As compiler writers, we can allocate one big activation record for all the procedures if none depend on themselves.

Directed Cycles and Backedges

- Consider procedures as vertices of a directed graph, and X $\rightarrow$ X′ if X calls X′
- We want to know: does G contain any **directed cycles**?
- **Theorem**. let G = (V, E) be a directed graph, and let T be a DFS tree (forest) of G. Then G contains a directed cycle IFF T contains a back edge.

Proof

- certainly if there's a back edge from v_i to v_j, then there's a directed cycle.
- Suppose now there's a cycle and let v_i be the vertex on the cycle having the smallest DFSnumber.
- **Claim**: Every vertex on the cycle is a descendant of v_i — thus the last edge to v_i is a back edge.

- How do we check for a directed cycle?

  Answer: Look for a back edge!
- **Idea**: As we traverse the graph — along the DFS forest, mark every vertex on the path from the root to the current vertex.

  Then, when examining edges from the current vertex, if the other end of one is marked, there's a back edge

  When moving back up trees, unmark edges left behind — to *clean up*

Topological Sorting

- A **topological sort** of G = (V, E), a directed acyclic graph (dag), is a linear ordering of all its vertices such that if G contains an edge $< u, v >$, then u appears before v in the ordering.
- Obs: If G is not acyclic, then there is no linear ordering.
- Note: dag's have many applications, e.g., in problems where precedence among events must be indicated (such as in job scheduling).

In what order must the following articles of clothing be put on?

underwear **socks** **watch**

jeans **shoes** **toe–ring**

belt **Tshirt** **over–shirt**

jacket **cap**

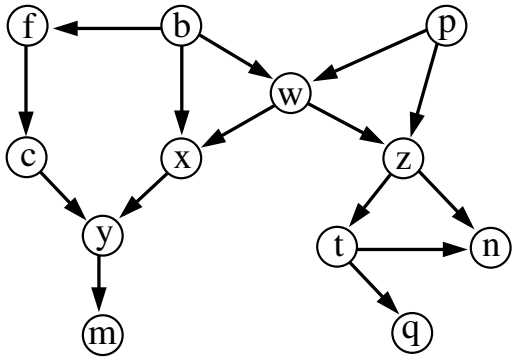
Which have indegree of 0?

Topological Sort Algorithm

```
Procedure TopoSort (G)
// PRE:  G is an acyclic directed graph
// POST: the vertices of G have been listed
//       in topological order
begin
  while vertices are left in V
     find a vertex, v, with indegree 0
     "output" v (i.e., add to the list)
     remove v (V = V - {v}), and all of its
        outgoing edges from the graph:
        for all <v, w> in E, E = E - {<v,w>}
     indegree(w)--
end
```

Question: What happens if G has cycles? Answer: The find will fail.

Find the Topological Sorting



Lemma

If G is acyclic (G has a vertex v of indegree 0), then we can find a linear ordering of G's vertices.

Proof:

BC $|V| = 1$, the linear ordering is the single vertex.

IH If G has n vertices and is acyclic, we can find an ordering

IS Suppose G has $n+1$ vertices and is acyclic.
Let v have indegree 0 (since acyclic, such a v exists), and let:

$$G' = (V - \{v\}, E - \{< v, u > | u \in V\})$$

I.e., remove v and its edges from G

Now, G' has n vertices and must also be acyclic (since G was), so we have an order $v_1, \ldots, v_n$ for G', and clearly $v, v_1, \ldots, v_n$ is an ordering for G.

Thus all dag's have a linear ordering.

Implementation

- Make an ordered container Indeg[1..n] to hold the indegrees of the vertices
- Indeg[ ] can be initialized in $O(|V| + |E|)$ time by traversing G.
- We can assume G is stored in a container V[1..n] of edge lists.
- The while loop in the algorithm eventually removes every edge (once and only once) so the total time for the loop is $O(|E|)$.

- How long does it take to find v with indegree zero?
  - **Naive**: O($|V|$) time, yielding O($|V|^2$) time overall for all $|V|$ finds.
  - **Better**: When Indeg[ ] is initialized, put vertices of indegree 0 in a **queue**.

    Whenever an edge $< v, u >$ is removed, check to see if u has indegree 0. If so, add it to the queue.

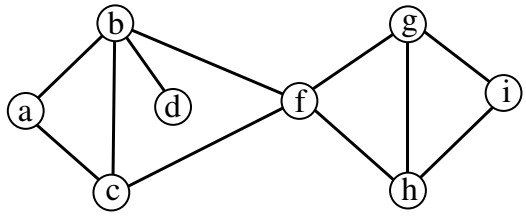
    Now, finding the next v takes O(1) time.
- So, overall, we can order the vertices with this topological sort in O($|V| + |E|$) to initialize Indeg[ ], and O($|E|$) to process.

Breadth First Search

Idea: visit all vertices and/or edges of G beginning with vertex v, then do the same for all neighbors, and their neighbors, and so on.

- The vertices of the BFS tree are visited in order of increasing depth
- non–BFS tree edges (the dotted ones) connect vertices which differ by at most 1 level.

Example: BFS(G, f)



BFS Algorithm

Given: An undirected graph $G = (V, E)$, and a vertex $v \in V$
Goal: Visit all vertices and edges of G starting with v.

```
Procedure BFS(G, v)
begin
  CreateQueue(Q)
  Mark (v)
  Enqueue(Q, v)
  while not Empty(Q)
     x = Dequeue(Q)
     Visit(x)
     for each neighbor w of x do
        if w is unmarked then
           Mark(w)
           Enqueue(Q, w)
        VisitEdge(x, w)
end
```


- **Claim**: If G is stored as an array of adjacency lists, then BFS takes time O($|E| + |V|$)
- Observation: BFS can be used instead of DFS in many situations.

  Usually the application determines which to use.

  E.g., Game trees (for chess, checkers, etc.) and possible moves.

BFS Application: Find the Center of a Graph

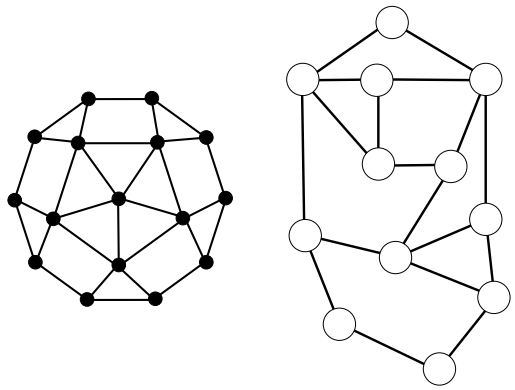
- Given a graph G, find a vertex v which minimizes the farthest distance to every node.

  I.e., find v such that $\max_{w \in V}$(distance(v, w)) is minimized.
- Such a vertex is called a **center** of G.

  Graphs may have one center, or many centers.
- Let us assume all edges have length 1.
- To find max distance:

  mark all neighbors
  for each neighbor
  mark all unmarked neighbors, etc.

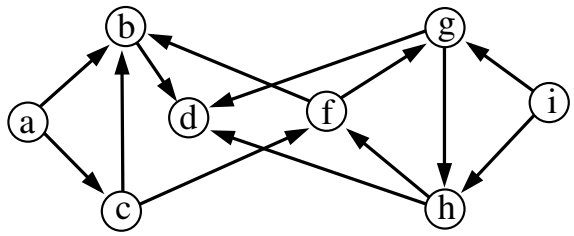
Examples: Finding Graph Centers



Complexity of Finding Graph Centers

- In BFS, how far can we go in one step? In 2 steps? Etc.
- For **each** $v \in G$, we can find the max distance to w in $O(|V| + |E|)$ time
- Do this for each vertex and choose the minimum
- Thus, $O(|V| * (|V| + |E|))$, or $O(|V|^2 + |V| * |E|)$ time

DiGraph: BFS(G, f)



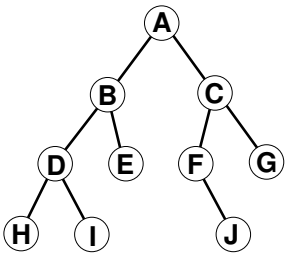
Binary Tree Traversals

Let **n** be a node, **L** be the node's left subtree, and **R** the right subtree.

There are three **systematic** approaches to traversing a binary tree — differing in when the node n is *visited*.

Pre–order	**nLR**
In–order	**LnR**
Post–order	**LRn**

Traversal Exercise 1

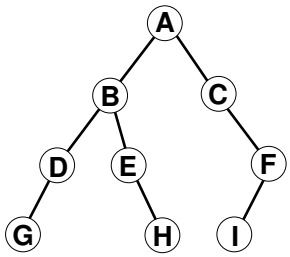


PRE: _____

IN: _____

POST: _____

Traversal Exercise 2



PRE: _____

IN: _____

POST: _____