CSE 101 Class Notes

April 18, 2004

Today: Iterative Graph Algorithms

**Graphs: definitions, properties, representations**

A graph $G = (V,E)$ consists of a set of vertices (nodes) $V$ and a set of edges (pairs of vertices) $E$. Graphs can be represented in two ways:

- **Adjacency lists:** for each node, keep a list of nodes to which that node is connected.

- **Adjacency matrix:** for an $n$-node graph, keep an $n$-by-$n$ matrix where each entry $m_{ij}$ represents an edge between nodes $i$ and $j$.

**Definition:** a cycle is a list of at least 3 edges $(x_1, x_2), \ldots, (x_k, x_1)$, $k \geq 3$.

**Definition:** a forest is an undirected graph $F = (V,E)$ with no cycles.

**Finding cycles**

**Lemma 1:** If every $x \in V$ has degree $\geq 2$, $V$ is not a forest.

**Proof:** Let $x_1$ be any node, $x_2$ be any neighbor of $x_1$, and $x_3$ be a neighbor of $x_2$ other than $x_1$ (which must exist, since degree$(x_2) > 1$). Iteratively, having chosen nodes $x_{i-1}$ and $x_i$, where $x_i$ is a neighbor of $x_{i-1}$, let $x_{i+1}$ be any neighbor of $x_i$ other than $x_{i-1}$ for $i > 1$. (Since $x_i$ has at least two neighbors, such a choice is possible.) Then at some point, $x_j = x_i$ for $j < i - 1$, and therefore the graph has a cycle, $x_j, x_{j+1}, \ldots, x_{i-1}, x_i = x_j$.

**Lemma 2:** if $x$ has degree $\leq 1$, then $F - \{x\}$ is a forest $\iff$ if $F$ is a forest.

**Proof:** $(\Rightarrow)$ If $c$ is a cycle in $F - x$, $c$ is a cycle in $F$. Therefore if $F$ has no cycles, then $F - x$ has none.

$(\Leftarrow)$ If $c$ is a cycle in $F$, then every node in $c$ has degree $\geq 2$. So $c$ is also a cycle in $F - x$, since $deg(x) = 1$, so $x \not\in c$. Therefore if $F - x$ has no cycles, then $F$ has none.

These lemmas suggest an algorithm:

1. while there exists a node $x$ in $G$ with degree$(x) \leq 1$
2. \quad $G \leftarrow G - x$
3. if $G$ is empty
return true
else
  return false

How do we do this efficiently? What information do we need? The subset of undeleted nodes, and the degree of each node in this subset. How do we need to access this? We need to find any node with degree \( \leq 1 \), or know that no such node exists. How do we update it? By deleting \( x \), and by decrementing the degree of \( x \)'s neighbors.

What data structure could we use? A heap (i.e. a min-heap of (degree, node) pairs) is okay, and yields the following algorithm:

```
H <- heap of { (degree(x), x) | x in G }
A <- array of H-elements indexed by node name
while H not empty and degree(top(H)) <= 1
  for each neighbor y of top(H)
    if inheap(H, A[y])
      degree(A[y]) <- degree(A[y]) - 1
      adjust(H, A[y])
  pop(H)
if H is empty
  return true
else
  return false
```

Roughly, the loop at line (3) will be repeated \( n \) times, and a node can have as many as \( n \) neighbors, so the algorithm is \( O(n^2) \) even with constant-time heap operations. However, we can be more precise: if \( F \) is a forest with \( n \) nodes and \( m \) edges, then \( m < n \). So the loops at (3) and (4), equivalent to the sum

\[
\sum_{v \in V} \sum_{e \in \delta(v)} \text{lines5} - 7 = \sum_{e = (u,v) \in E} \sum_{u, v} \text{lines5} - 7
\]

which is \( O(2mt(\text{lines5} - 7)) = O(m \log n) \). For sparse graphs, this is \( O(n \log n) \), while for dense graphs, it is \( O(n^2 \log n) \). To avoid this dense-graph behavior, we can detect dense graphs quickly by replacing line (1) with the following:

```
for i = 1..n
  d <- degree(i)
  insert(H, (d, i))
  m <- m + d
  if m >= 2 * n
    then return false
```

by noting that if \( m > 2n \), then the graph must always have a cycle.

In this case, using a heap is overkill – it allows us to find the minimum-degree node, when all we care about is finding any node with degree \( \leq 1 \). Luckily, there's a better data structure. Let \( G \) be the set of undeleted nodes, \( L \) be the set with degree \( \leq 1 \). We require the following update operations:
1. delete $x$ from $L$

2. decrement degree of $x$’s neighbor $y$

3. possibly add $y$ to $L$.

A linked list for $L$ supports these operations easily. The algorithm then becomes

```plaintext
1  In <- { true | x in G }
2  n <- size(G)
3  D, L : array of integers
4  for i = 1..n  // initialize D, L
5      d <- degree(i)
6      D[i] <- d
7      m <- m + d
8          if m >= 2 * n
9              then return false
10             if d <= 1
11                 insert(L, i)
12             t <- 0
13             while L not empty
14                 x <- head(L); In(x) <- F
15                 L <- tail(L)
16                 For the at most one y in neighbor(x) with In(y)=T
17                     D[y] < D[i] - 1
18                     if D[y] <= 1
19                         insert(L, y)
20                 t <- t + 1
21             if t = n
22                 return true
23         else
24             return false
```

Now, if the graph has more than $2n$ edges, we will halt in line 9 after at most $O(n)$ work. So assume $m \leq 2n$. Note that line 16 takes time proportional to $deg(x)$, since we run through all neighbors $y$ of $x$ and see which one is still in the graph. So the total time for this line will be $\sum deg(x) = 2m = O(n)$ since $m \leq 2n$. The other lines are done once per node and take constant time, so are also $O(n)$. So this version has time complexity $O(n)$. Since we’ll run through each