Prelim coverage

There's a prelim in class on March 6.

- The review session is March 5, 7 PM, Upson 207

You're responsible for everything we've cover up to the end of this lecture:

- Big-O, Θ (Chapter 2.1)
- Solving recurrences using the Master Theorem
- Stacks, queues, and linked lists
- Hashing
- Binary Search Trees
- Priority Queues and Heaps
- Skip List + Union-Find
- Intro to Graph Algorithms (up to BFS)

You need to know

- advantages/disadvantages of various methods:
  - e.g., hashing with chaining vs. open addressing
- how to implement basic operations (insert, delete, search, etc.) on standard data structures.

Graph Algorithms

Review Section 5.4 (pp. 86–91).

Recall a graph G consists of vertices V and edges E

- We write $G = (V, E)$ or $G(V, E)$

I will presume you know about:

- directed graphs vs. undirected graphs
- the degree (indegree, outdegree) of a vertex
- the length of a path
- reachability
- connected components
- subgraph (induced by V')
- complete graph

Will now consider some basic graph algorithms

- will deal data structure and representation issues much more than in CS280

Sparse vs. Dense Graphs

Note that if $G = (V, E)$, then $0 \leq |E| \leq |V|^2$.

- a graph is *dense* if $|E| = \Omega(|V|^2)$
- a graph is *sparse* if $|E| \ll |V|^2$ (typically $O(|V|)$)

Representing Graphs

What's the best way of representing a graph?

- depends on whether the graph is sparse or dense

There are two standard ways of representing graphs.

1. *adjacency-list representation*:
   - Use an array Adj of $|V|$ lists
   - list $Adj[u]$ consist of all v such that $(u, v) \in E$
   - $|Adj[u]| =$ (out)degree(u)
   - $\Sigma_{u \in V} |Adj[u]| = |E|$ for directed graphs
   - $\Sigma_{u \in V} |Adj[u]| = 2|E|$ for undirected graphs
   - memory required $= O(\max(V, E)) = O(V + E)$
   - can easily represent weighted graphs
2. adjacency-matrix representation
   - assume vertices are numbered 1, ..., $|V|$
   - use a $|V| \times |V|$ matrix $A = (a_{ij})$

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \\ 0 & \text{otherwise} \end{cases}$$

   - can also easily represent weighted graphs
   - requires $O(|V|^2)$ *bits* of storage
     - vs. $O(|V| + |E|)$ *words* for adjacency list

Breadth-First Search

Idea: starting at a vertex s (the *source*) in $G(V, E)$, systematically explore G:

- start with vertices closest to s and work out
- the search produces a "breadth-first tree", with s at the root
- if v is reachable from s, the path from s to v in the tree is the shortest path from s to v in G

If we don't reach the whole graph starting from s, then start over at another vertex.

Breadth-First Search Algorithm

Idea of the algorithm:

- Start at some vertex s
- Vertices are colored:
  - white vertices – not yet "discovered"
  - gray vertices – discovered, neighbors not checked
  - black – discovered + neighbors checked
- algorithm uses a (FIFO) queue Q to manage the gray vertices
- initially only s gray
- For each gray vertex v
  - visit all its neighbors
  - if they were white, color them gray
  - then color v black
- array *color* is used to keep track of the color
- for later applications, keep track of
  - $d[u]$ – distance from u to s
  - $\pi[u]$ – parent of u in breadth-first tree

```
BFS(G)

for each vertex u ∈ V(G)
    do color[u] ← WHITE
       π[u] ← NIL
for each vertex u ∈ V(G)
    if color[u] = WHITE
       then BFS-SEARCH(u)
```

```
BFS-SEARCH(s)
color[s] ← GRAY
d[s] = 0
Q ← {s}
while Q ≠ ∅
    do u ← head[Q]
       for each v ∈ Adj[u]
       (Adj[u] = {v : (u, v) ∈ E}
           if color[v] = WHITE
              then color[v] ← GRAY
                   d[v] ← d[u] + 1
                   π[v] ← u
                   ENQUEUE(Q, v)
       DEQUEUE(Q)
       color[u] ← BLACK
```

Running Time of BFS

Initialization (lines 1–4) takes time $O(|V|)$

- must initialize *color*, d, π for all vertices

Each vertex gets ENQUEUEd at most once

- only vertices that have just changed from white to gray get ENQUEUEd
- once a vertex becomes gray, it never changes back to white
  - it can't get ENQUEUEd again

Each vertex gets DEQUEUEd at most once.

Each edge (u, v) is processed at most twice at line 6 of BFS-SEARCH:

- once for u, once for v

Running time is $O(|V|+|E|)$ *using the adjacency-list representation.*

Properties of BFS

Let $\delta_G(u, v)$ be the *shortest-path* distance from u to v in G:

- minimum number of edges on a path from u to v

Theorem: After running BFS-SEARCH(s), for every vertex v reachable from s is visited and $d[v] = \delta(s, v)$; for $v \neq s$, $\pi[v]$ is the predecessor of v on a shortest path from s to v.

- This is true for both directed and undirected graphs.

This seems almost obvious from the construction of the algorithm, but we need to be careful when we do a formal proof ...

Lemma 1: Suppose at some point in BFS-SEARCH$[s]$, $Q = [v_0, \ldots, v_k]$. Then there is some i, j such that

$$d[v_0] = \cdots = d[v_j] = i,\ d[v_{j+1}] = \cdots = d[v_k] = i + 1.$$

Proof: This is true initially (when $Q = \{s\}$).

The property is maintained after each pass through the loop:

- when we process v_0, we add white neighbors u of v_0 to the end of Q, with $d[u] = d[v_0] + 1$.

Lemma 2: If we enqueue $v_1, v_2, v_3, \ldots, v_k$ (in that order), then $d[v_1] \leq d[v_2] \leq \ldots$.

Proof: Immediate from Lemma 1.

Lemma 3: Every vertex that is "discovered" (colored gray in line 8) is reachable from s.

Proof: Show that this property is maintained on each iteration of the loop. (Formally, by induction on the k, show property holds on kth iteration of loop.)

Proof of Theorem: By Lemma 3, if $\delta(s, v) = \infty$, then v is not discovered.

If $\delta(s, v) = k < \infty$, then we prove by induction on k then there is a point in BFS-Search(s) when we

- color v gray
- set $d[v] = k$
- put v into Q
- if $s \neq v$, then $(\pi[v], v) \in E$ and $d[\pi[v]] = k - 1$

Base case: $v = s$ — OK.

Inductive step: Suppose $\delta(s, v) = k + 1$.

- Exists u such that $\delta(s, u) = k$ and $(u, v) \in E$.
- If $\delta(s, u') < k$, then $(u', v) \notin E$.

Induction assumption $\Rightarrow$ u is Enqueued, $d[u] = k$. We must discover v while processing u, if we haven't discovered it already.

Suppose we discover v while processing u'.

- Either $u' = u$ or we process u' before u
- By Lemma 2, $d[u'] \leq d[u]$ ($\Rightarrow d[u'] \leq k$)
- Since $\delta(s, v) = k + 1$, can't have $\delta(s, u') < k$.
- Thus, $d(u') = k$, $d(v) = k + 1$, $\pi(v) = u'$.

Breadth-First Trees

Let $E_\pi = \{(\pi[v], v) : v \in V,\ \pi[v] \neq \text{NIL}\}$

- $E_\pi \subseteq E$

Proposition: BFS(G) constructs π so that $G_\pi = (V, E_\pi)$ is a forest (set of disjoint trees), whose roots are the vertices s for which we call BFS-Search(s). Moreover, if s is the root of a tree, then v is in the tree iff v is reachable from s, and the path from s to v in the tree is a minimal length path from s to v in G.

Note that this gives us another way of computing the connected components of G if G is undirected.

Depth-First Search

This time we search a graph by following a path as long as possible, then backtracking.

- We use a stack instead of a queue to keep track of gray edges

As we discover vertex u, we timestamp it:

- We timestamp twice:
  - once when we first discover v: $d[v]$
  - again when we're done with v's adjacency list: $f[v]$
  - v is white before $d[v]$, gray between $d[v]$ and $f[v]$, black after $f[v]$

DFS(G)

```
for each vertex u ∈ V(G)
    do color[u] ← WHITE
       π[u] ← NIL
time ← 1
for each vertex u ∈ V(G)
    do if color[u] WHITE
          then DFS-Visit(u)
```

DFS-Visit(u)

```
color[u] ← GRAY
d[u] ← time
time ← time + 1
for each v ∈ Adj[u]
    do if color[v] = WHITE
          then π[v] ← u
               DFS-Visit[v]
color[u] ← BLACK
f[u] ← time
time ← time + 1
```

Running Time of DFS

Initialization (lines 1–3) takes time $O(|V|)$.

We call DFS-VISIT at most once for each $u \in V$.

- We call DFS-VISIT$[u]$ only when u is white
- u is colored gray as soon as we call DFS-VISIT$[u]$

The total cost of lines 2–5 of DFS-VISIT$[u]$ is $O(|Adj[u]|)$.

The total cost of lines 2–5 of all calls of DFS-VISIT is

$$\sum_{u \in V} O(|Adj[u]|) = O(|E|).$$

Total cost of DFS is $O(|V| + |E|)$ (for the adjacency-list representation).

- it would be $O(|V|^2)$ for the adjacency-matrix representation

Parenthesis Structure

Proposition: DFS(G) constructs π so that $G_\pi = (V, E_\pi)$ is a forest whose roots are the vertices s for which we call DFS-Visit(s).

The start times and finish times for vertices u form a *parenthesis structure*

- either $[[d[u], f[u]]$ is contained in $[[d[v], f[v]]$, or they are disjoint.

Parenthesis Theorem: After running DFS(G), for any vertices u and v in $V(G)$, either

- $[d[u], f[u]]$ and $[d[v], f[v]]$ are disjoint
  - $[d[u], f[u]] \cap [d[v], f[v]] = \emptyset$
- $[d[u], f[u]] \subset [d[v], f[v]]$ and u is a descendant of v in some tree of the depth-first forest
- $[d[v], f[v]] \subset [d[u], f[u]]$ and v is a descendant of u in some tree of the depth-first forest

Proof: Can't have $d[u] = d[v]$

- whichever one is discovered first must have smaller start time

Suppose $d[u] < d[v]$

- if $d[v] < f[u]$, v is discovered while u is still gray
  - must be running DFS-VISIT(u)
  - v is a descendant of u
  - $f(v) < f(u)$
- if $d(v) > f(u)$, intervals must be disjoint

Similar argument if $d[v] < d[u]$.

Corollary: v is a descendant of u in the depth-first forest iff $d[u] < d[v] < f[v] < f[u]$.

White Path Theorem: v is a descendant of u in the depth-first forest iff when u is discovered, there is a path from u to v consisting of only white vertices.

Proof: If v is a descendant of u, let w be any vertex on the path from u to v in the depth-first forest.

- By previous corollary, $d[w] > d[u]$.
- So w must be white at $d[u]$ (w turns gray at $d[w]$).

So there is a path of white vertices from u to v at time $d[u]$.

Conversely, if there is a path from u to v consisting of only white vertices of length k, we prove by induction on k that v is a descendant of u. If $k = 1$:

- Algorithm guarantees that we must discover v before $f[u]$.
- $d[u] < d[v] < f[v] < f[u]$ (Parenthesis Theorem)
- By Corollary, v is a descendant of u.

If $k = k' + 1$, consider predecessor w of v on the path.

- w is a descendant of u (induction hyp.)
- By Corollary, $d[u] < d[w] < f[w] < f[u]$.
- Must have $d[v] < f[w]$.
- By Parenthesis Theorem, $f[v] < f[w] < f[u]$.
- By Corollary, v is a descendant of u.

Topological Sort

A *dag* (directed acyclic graph) is a directed graph with no cycles.

A *topological sort* of a dag $G = (V, E)$ is a linear ordering of the vertices in V such that if $(u, v) \in E$, then $u < v$.

- can't do this if G has a cycle

Suppose the dag G describes a precedence ordering of events

- $(u, v) \in E$ means that u must be done before v

Then a topological sort of G describes one way in which the events can be performed.

- There may be several possible topological sorts of a dag.

Using DFS for Topological Sort

We can use the finishing times of DFS to topologically sort

- vertices with earlier finishing times come later in the list

TOPOLOGICAL-SORT(G)

call DFS(G)
each time a vertex is finished,
  insert it onto the front of a linked list
return the linked list

Note: if there are n vertices, it may be better to return an array $T[1..n]$

- put vertices onto array starting at end
- $T[i]$ is ith vertex in the topological sort

Theorem: TOPOLOGICAL-SORT(G) produces a topological sort of G.

Proof: Must show that $(u, v) \in E \Rightarrow f(v) < f(u)$.

Case 1: We turn u gray before v.

- then we discover v while we are running DFS-VISIT(u)
- we finish v before we finish u
- $f(v) < f(u)$

Case 2: We turn v gray before u

- then we don't discover u before we finish v
- otherwise u is a descendant of v in G and we have a cycle
- so $f[v] < d[u] < f[u]$.