CS38
Introduction to Algorithms

Lecture 2

April 3, 2014

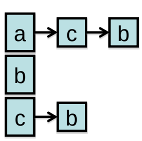
Outline

- graph traversals (BFS, DFS)
- connectivity
- topological sort
- strongly connected components

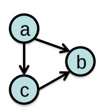
- heaps and heapsort
- greedy algorithms…

Graphs

- Graph G = (V, E)
  - directed or undirected
  - notation: $n = |V|$, $m = |E|$ (note: $m \leq n^2$)
  - adjacency list or adjacency matrix



	a	b	c
a	0	1	1
b	0	0	0
c	0	1	0



Graphs

- Graphs model many things…
  - physical networks (e.g. roads)
  - communication networks (e.g. internet)
  - information networks (e.g. the web)
  - social networks (e.g. friends)
  - dependency networks (e.g. topics in this course)

… so many fundamental algorithms operate on graphs

Graphs

- Graph terminology:
  - an undirected graph is connected if there is a path between each pair of vertices
  - a tree is a connected, undirected graph with no cycles; a forest is a collection of disjoint trees
  - a directed graph is strongly connected if there is a path from x to y and from y to x, $\forall\, x,y \in V$
  - a DAG is a Directed Acyclic Graph

Graph traversals

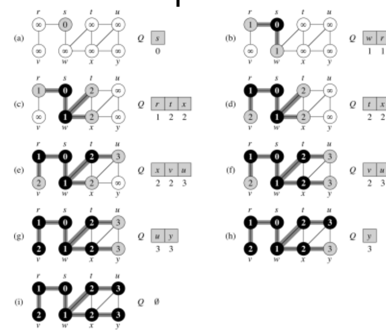
- Graph traversal algorithm: visit some or all of the nodes in a graph, labeling them with useful information
  - breadth-first: useful for undirected, yields connectivity and shortest-paths information
  - depth-first: useful for directed, yields numbering used for
    - topological sort
    - strongly-connected component decomposition

Breadth first search

BFS(undirected graph G, starting vertex s)
1. for each vertex v, v.color = white, v.dist = ∞, v.pred = nil
2. s.color = grey, s.dist = 0, s.pred = nil
3. Q = $\emptyset$; ENQUEUE(Q, s)
4. WHILE Q is not empty u = DEQUEUE(Q)
5. for each v adjacent to u
6. IF v.color = white THEN
7. v.color = grey, v.dist = u.dist + 1, v.pred = u
8. ENQUEUE(Q, v)
9. u.color = black

Lemma: BFS runs in time O(m + n), when G is represented by an adjacency list.

BFS example from CLRS

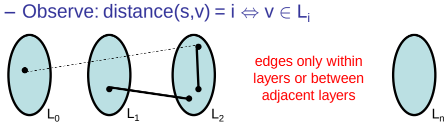


Breadth first search

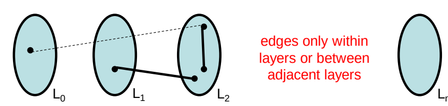
Lemma: for all $v \in V$, v.dist = distance(s, v), and a shortest path from s to v is a shortest path from s to v.pred followed by edge (v.pred,v)

Proof: partition V into levels

- $L_0 = \{s\}$
- $L_i = \{v : \exists u \in L_{i-1}$ such that $(u,v) \in E\}$
- Observe: distance(s,v) = i $\Leftrightarrow v \in L_i$



Breadth first search

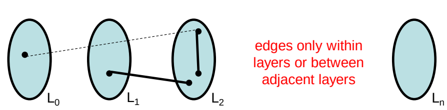


Claim: at any point in operation of algorithm:

1. black/grey vertices exactly $L_0, L_1, \ldots, L_i$ and part of L_{i+1}
2. $Q = (\underbrace{v_0, v_1, v_2}_{\text{level } i}, \underbrace{v_3, \ldots, v_k}_{\text{level } i+1})$ and all have v.dist = level of v

holds initially: s.color = grey, s.dist = 0, Q = (s)

Breadth first search



Claim: at any point in operation of algorithm:

1. black/grey vertices exactly $L_0, L_1, \ldots, L_i$ and part of L_{i+1}
2. $Q = (\underbrace{v_0, v_1, v_2}_{\text{level } i}, \underbrace{v_3, \ldots, v_k}_{\text{level } i+1})$ and all have v.dist = level of v

$\Rightarrow$ level $\geq$ i+1

$\Rightarrow$ level $\leq$ i+1

1 step: dequeue v_0; add white nbrs of v_0 w/ dist = v_0.dist + 1

Depth first search

DFS(directed graph G)
1. for each vertex v, v.color = white, v.pred = nil
2. time = 0
3. for each vertex u, IF u.color = white THEN **DFS-VISIT**(G, u)

DFS-VISIT(directed graph G, starting vertex u)
1. time = time +1, u.discovered = time, u.color = grey
2. for each v adjacent to u, IF v.color = white THEN
3. v.pred = u, **DFS-VISIT**(G, v)
4. u.color = black; time = time + 1; u.finished = time

Lemma: DFS runs in time O(m + n), when G is represented by an adjacency list.

Proof?

Depth first search

DFS(directed graph G)
1. for each vertex v, v.color = white, v.pred = nil
2. time = 0
3. for each vertex u, IF u.color = white THEN **DFS-VISIT**(G, u)

DFS-VISIT(directed graph G, starting vertex u)
1. time = time +1, u.discovered = time, u.color = grey
2. for each v adjacent to u, IF v.color = white THEN
3. v.pred = u, **DFS-VISIT**(G, v)
4. u.color = black; time = time + 1; u.finished = time

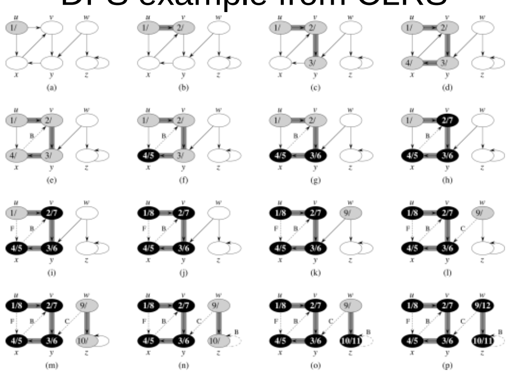
Lemma: DFS runs in time O(m + n), when G is represented by an adjacency list.

Proof: DFS-VISIT called for each vertex exactly once; its adj. list scanned once; O(1) work

Depth first search

- DFS yields a forest: "the DFS forest"
- each vertex labeled with discovery time and finishing time
- edges of G classified as
  - tree edges
  - back edges (point back to an ancestor)
  - forward edges (point forward to a descendant)
  - cross edges (all others)

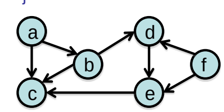
DFS example from CLRS

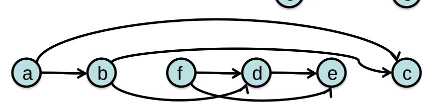


DFS application: topological sort

- Given DAG, list vertices $v_0, v_1, \ldots, v_n$ so that no edges from v_i to v_j ($j < i$)

example:





DFS application: topological sort

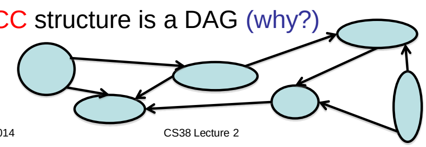
Theorem: listing vertices in reverse order of DFS finishing times yields a topological sort of DAG G

(can implement in linear time; how?)

Proof: claim for all $(u,v) \in E$, v.finish < u.finish
- when (u,v) explored, v not grey since then G would have a cycle [back-edge]
- v white ⇒ descendent of u so v finishes first
- v black ⇒ already done, so v.finish is set and u.finish will be set with a later time

Strongly connected components

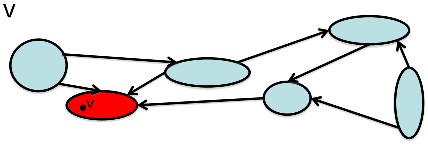
- say that $x \sim y$ if there is a directed path from x to y and from y to x in G
- equivalence relation, equivalence classes are strongly connected components of G
  - also, maximal strongly connected subsets
- SCC structure is a DAG (why?)



Strongly connected components

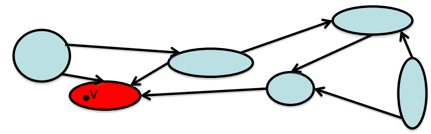
- DFS tree from v in G: all nodes reachable from v
- DFS tree from v in G^T: all nodes that can reach v

G with edges reversed



- **Key**: in sink SCC, this is exactly the SCC

Strongly connected components



- given v in a sink SCC, run DFS starting there, then move to next in reverse topological order…
  - DFS forest would give the SCCs
- **Key #2**: topological ordering consistent with SCC DAG structure! (why?)

Strongly connected components

SCC(directed graph G)
1. run DFS(G)
2. construct G^T from G
3. run DFS(G^T) but in line 3, consider vertices in decreasing order of finishing times from the first DFS

- running time O(n + m) if G in adj. list
  - note: step 2 can be done in O(m + n) time
- trees in DFS forest of the second DFS are the SCCs of G

Strongly connected components

SCC(directed graph G)
1. run DFS(G)
2. construct G^T from G
3. run DFS(G^T) but in line 3, consider vertices in decreasing order of finishing times from the first DFS

Correctness (sketch):
- first vertex is in sink SCC, DFS-VISIT colors black, effectively removes
- next unvisited vertex is in sink after removal
- and so on…

Summary

- O(m + n) time algorithms for
  - computing BFS tree from v in undirected G
  - finding shortest paths from v in undirected G
  - computing DFS forest in directed G
  - computing a topological ordering of a DAG
  - identifying the strongly connected components of a directed G

  (all assume G given in adjacency list format)

Heaps

- A basic data structure beyond stacks and queues: heap
  - array of n elt/key pairs in special order
  - min-heap or max-heap

    operations:

    INSERT(H, elt)

    INCREASE-KEY(H, i)

    EXTRACT-MAX(H)

Heaps

- A basic data structure beyond stacks and queues: heap
  - array of n elt/key pairs in special order
  - min-heap or max-heap

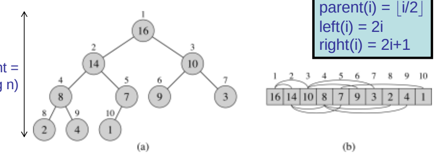
operations:	time:
INSERT(H, elt)	O(log n)
INCREASE-KEY(H, i)	O(log n)
EXTRACT-MAX(H)	O(log n)

Heaps

- array A represents a binary tree that is full except for possibly last "row"

parent(i) = $\lfloor i/2 \rfloor$
left(i) = 2i
right(i) = 2i+1

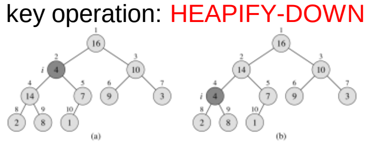
height = O(log n)

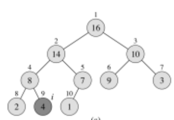


- heap property: A[parent(i)] $\geq$ A[i] for all i

Heaps

- key operation: HEAPIFY-DOWN(H, i)



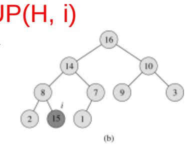


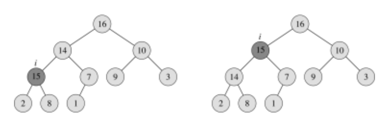
A[i] may violate heap property
- repeatedly swap with larger child
- running time?

Heaps

- key operation: HEAPIFY-UP(H, i)

A[i] may violate heap property
- repeatedly swap with larger child
- running time?





Heaps

- How do you implement

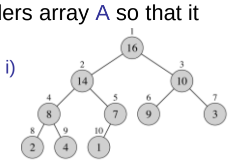
operations:	time:
INSERT(H, elt)	O(log n)
INCREASE-KEY(H, i)	O(log n)
EXTRACT-MAX(H)	O(log n)

using HEAPIFY-UP and HEAPIFY-DOWN?

- BUILD-HEAP(A): re-orders array A so that it satisfies heap property
  - how do we do this? running time?

Heaps

- BUILD-HEAP(A): re-orders array A so that it satisfies heap property
  - call HEAPIFY-DOWN(H, i) for i from n downto 1
  - running time O(n log n)



  - more careful analysis: O(n)

$$\sum_{h=0}^{\log n} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O(n) \cdot \sum_{h=0}^{\log n} \frac{h}{2^h} = O(n)$$

Heaps

$$\sum_{h=0}^{\log n} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O(n) \cdot \sum_{h=0}^{\log n} \frac{h}{2^h} = O(n)$$

- suffices to show $\sum_{h\geq 0} h/2^h = O(1)$
- note: $\sum_{h\geq 0} c^h = O(1)$ for $c < 1$
- observe: $(h+1)/2^{h+1} = h/(2^h) \cdot (1+1/h)/2$
- $(1+1/h)/2 < 1$ for $h > 1$

Heapsort

- Sorting n numbers using a heap
  - BUILD-HEAP(A) O(n)
  - repeatedly EXTRACT-MIN(H) n·O(log n)
  - total O(n log n)
- Can we do better? O(n)?
  - observe that only ever compare values
  - no decisions based on actual values of keys

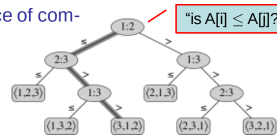
Sorting lower bound

comparison-based sort: only information about A used by algorithm comes from pairwise comparisons

- heapsort, mergesort, quicksort, …

visualize sequence of com-parisons in tree:

"is A[i] ≤ A[j]?"



- each root-leaf path consistent with 1 perm.
- maximum path length $\geq \log(n!) = \Omega(n \log n)$

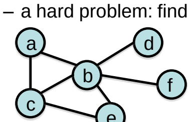
Greedy Algorithms

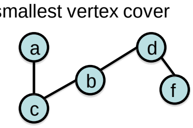
Greedy algorithms

- Greedy algorithm paradigm
  - build up a solution incrementally
  - at each step, make the "greedy" choice

Example: in undirected graph G = (V,E), a vertex cover is a subset of V that touches every edge

- a hard problem: find the smallest vertex cover





Dijkstra's algorithm

- given
  - directed graph G = (V,E) with non-negative edge weights
  - starting vertex $s \in V$
- find shortest paths from s to all nodes v
  - note: unweighted case solved by BFS

Dijkstra's algorithm

- shortest paths exhibit "optimal substructure" property
  - optimal solution contains within it optimal solutions to subproblems
  - a shortest path from x to y via z contains a shortest path from x to z
- shortest paths from s form a tree rooted at s
- Main idea:
  - maintain set $S \subseteq V$ with correct distances
  - add nbr u with smallest "distance estimate"