UCLA
Computer Science Department

CS180– Midterm
Algorithms & Complexity

10/30/2018

Name: Rohan Chitale 4-5:50 p.m RuiRui
UID: 704924773

This exam contains 7 pages (including this cover page) and 6 questions.

- Writing has to be legible.
- Express algorithms in bullet form, step by step.

Distribution of Marks

Question	Points	Score
1	20	10
2	20	2
3	20	14
4	10	10
5	20	5
6	10	6
Total:	100	65

+4
69

ame(last, first): Midterm 10/30/2018

20/20

2. (20 points) (a) Design an efficient algorithm that outputs the vertices of a DAG (Directed Acylic Graph), such that if there is an edge (x, y) then x is output before y.

(b) Analyze the run time of your algorithm.

a. We'll use topological sort:

- First, count number of incoming and outgoing edges for each node in the DAG ✓
- Output any node(s) that has no incoming edges as sources ✓
- Decrement incoming edges for all nodes directly adjacent to and reachable from sources (we're deleting these edges) ✓

repeat until all nodes are outputted

10/10

Proof: Assume by contradiction that topo sort does not output x before y if there is an edge (x, y)

- then topo sort would have to output y before x
- But this is a contradiction, because there's an incoming edge to y, and topo sort never outputs nodes with incoming edges

e = # edges n = # sources

B: Counting # incoming and outgoing edges → charge to edge, so O(e)

- If there are n nodes that are not connected by any edges, then we have O(n)
- So, first step → O(n+e) ✓

outputting all sources → n sources, so O(n) ✓

Changing incoming degree for all e edges → O(e) ✓

O(n+e) + O(n) + O(e) → Run time = O(e+n)

10/10

ame(last, first):

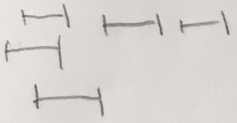
10

Midterm

10/30/2018

1. (20 points) Consider a set of intervals $I_1, I_2, \cdots, I_n$:
   (a) Design a linear time algorithm (assume that intervals are sorted in any manner you wish) that assigns the intervals to the minimum number of processors.
   (b) Prove the correctness of your algorithm.

a. - sort the intervals by finishing time such that interval with earliest finishing time comes first ✗ (no)
   - while there are still unvisited intervals, performing plane sweep $-\infty \to \infty$
     - each time we see a start point for interval, increment processor count
     - each time we see an interval end point, decrement processor count and mark interval as visited



b. Proof:

Proof by contradiction our algorithm is optimal

- at any time T, # processors taken up = number of intervals overlapping at that point
- So, in our algorithm, maximum # interval overlap = minimum # of processors we need (we think)
- Proving by contradiction, Assume we are at some time T where all processors (P) are taken up. P = max # interval overlap ✓
- Then we would have to assign T to a new processor, making our processor count P+1
- But this is a contradiction because the max # of interval overlap is not P+1 → we already defined it to be P, and the only way we add a new processor at T is if there's another interval overlapping at this time
- Therefore, our algorithm will always give the minimum # processors

Rohan Chitale
Discussion

Name(last, first):

Midterm 1φ

10/30/2018

3. (20 points) An undirected graph is said to have property X if you can start from a vertex, traverse all edges of the graph exactly once, without removing your pen from the paper.
(a) Classify the graphs that have property X?
(b) Design an efficient algorithm for generating a traversal of a graph that has property X.

a. These are Eulerian graphs - # odd degree vertices = 0 or 2, all other vertices have even degree ✓

Proof: - We can have at most 2 vertices with odd degree
- Each odd degree vertex must be either a start or end point → we have no way of returning back to these vertices once we traverse an adjacent edge
- Therefore, if we have more than 2 odd degree vertices, by Pigeonhole principle, we can't assign a "start" or "end" property to these vertices (n+2 vertices fitting into 2 holes → not possible)

Proof: Graph must either have 0 or 2 odd degree vertices
- already proved that 2 odd degree vertices → 2 pigeons, fit into 2 holes (start and end hole)
- Not possible to have graph with 1 odd degree vertex because $\sum_{i=1}^{n}$ degree $= 2n$ in a graph (sum of degrees of all vertices must be even), and an odd degree + even number of degree = odd number
- Graph with 0 odd degree vertices = cycle because each vertex has an x number of incoming ~~and~~ edges and an x number of outgoing edges, so each vertex can be visited without repeating edges

- Therefore, graphs with property X must have either 0 or 2 odd degree vertices, and the rest of the vertices are even

Algorithm:

- First, go through all e edges of the graph
- update degree of nodes adjacent on e by 1
- Look at each node N: if n has an odd degree, then stop: we still start at this node
- otherwise, we can start from any arbitrary node n
- use ~~BFS~~ starting at the start node,
- output unvisited nodes and ~~adjacent~~ edges every time we visit a node, ~~until~~ all our BFS ~~queue~~ is empty

- BFS ~~runtime~~: O(e+n)

- Starting at the start node, use DFS to visit all vertices. Everytime we visit a previously unvisited vertex, output that vertex

-6

DFS runtime: O(e+n)

Proof: DFS visits each edge twice for each vertex, ~~so it will output the~~ so that gives us O(2e) → O(e). If the graph has n connected components, then we get O(e+n)

Rohan Chitale

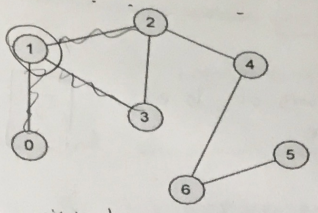
10

Name(last, first):

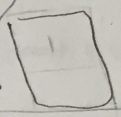
Midterm

10/30/2018

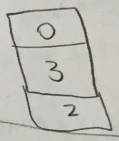
4. (10 points) Consider an unweighted graph G shown below:
   (a) Starting from vertex 1, show every step of BFS along with the corresponding FIFO next to it.



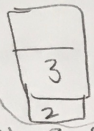
- step 1: Mark 1 as visited, push to front of queue
- step 2: Pop front of queue (1)



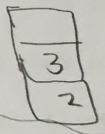
-Step 3: mark neighbors of 1 as visited, push neighbors to front of queue in arbitrary order →



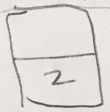
-step 4: pop front (0)



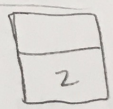
-step 5: 0 has no unvisited neighbors, so continue to next step



-step 6: pop front (3)

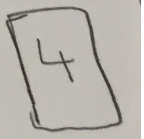


-step 7: 3 has no unvisited neighbors, so continue to next step

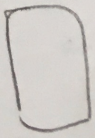


-step 8: pop front (2)

- step 9: mark all unvisited neighbors of 2 as visited, push to queue



-step 10: pop front (4)



step 11
- mark all unvisited neighbors of 4 as visited, push to queue

step 12
- pop front (6)

step 13
- mark all unvisited neighbors of 6 as visited, push to queue

step 14
- pop front (5)

step 15
- 5 has no unvisited neighbors, so continue

step 16:
queue is empty, so we return

Rohan Chitale
Discussion

Name(last, first):

Midterm

10/30/2018

5. (20 points) Consider an unsorted list of integers. You can find the minimum number in the list with $n-1$ comparisons. Similarly, you can find the maximum with $n-1$ comparisons. So you can find both the minimum and the maximum with about $2n-3$ comparisons. Design an algorithm that finds both the minimum and the maximum using about $\frac{3n}{2}$ comparisons.

10 5 8 7 4 3 12

$n-1 + \frac{1}{2}n$

10→5→8→7→4→3→12

4→3

- We can first find maximum by looking at all n numbers in the list, performing n-1 comparisions
- We can find minimum by perform $\frac{n}{2}$ comparisions to give us $n + \frac{n}{2} = \frac{3n}{2}$ comparisons total
- We'll use a modified version of bubble sort through one pass
  - start at the front of the list, set max and min = front
  - if the next node is greater than the current max, then set next node = current max, but delete this node from the list
  - continue until we reach the end of the list, deleting nodes greater than current max
- This will leave us with ~~also~~ almost $\frac{n}{2}$ nodes from n-1 comparisons for the max

- On this list of $\frac{n}{2}$ nodes, we perform $\frac{n}{2}-1$ comparisons, giving us $\frac{n}{2}-1+n-1 = \frac{3n}{2}-2 \approx \frac{3n}{2}$ comparisons.

Name(last, first): Midterm 10/30/2018

6. (10 points) Give an algorithm to color a graph with 2 colors (assuming it is 2-colorable). A proof of correctness is not necessary.

- algorithm:
- Use Breadth first search:
  - starting at some node s at Level 1, give s some color C_1
  - Keeping track of level, every time we encounter nodes in our BFS with an odd Level # → give those nodes color C_1
    - every time we encounter nodes with some even Level # → give those nodes color C_2
- return

4 Time complexity