UCLA
Computer Science Department

CS180– Midterm
Algorithms & Complexity

10/30/2018

Dis: 1G

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	11
2	20	20
3	20	8
4	10	10
5	20	20
6	10	6
Total:	100	75

+4

11

Name(last, first

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.

Algorithm

- Assume the intervals are sorted from earliest end time to latest end time ✗
- While the list of interval's isn't empty
- Pick the earliest ending interval, x, and assign it to processor 1
- Remove all intervals conflicting with x and add them to list 2
- End while

- While list 2 isn't empty
- Pick the earliest ending interval, x, and assign it to processor 2
- Remove all intervals conflicting with x, and add them to list 3
- End While

- Repeat until list n is empty at the start of its while loop (all intervals have been assigned)

Time Complexity
- O(n) to do a plane sweep of the sorted intervals
- linear time
- O(1) to assign to a processor
- Overall: O(n)

b. Proof

- Our algorithm picks the intervals with the earliest end time in order to the maximize the amount of intervals per processor. Say ours fits n intervals in a given processor.
- Say there's an algorithm, X', that fits m > n intervals in the same processor
- Say both algorithms have the same k intervals to begin
- We can replace X''s k+1th element with our k+1 elements since our algorithm greedily picks the earliest end time possible.
- Now, inductively do this until the first n intervals match
- Our algorithm would pick up any non-conflicting interval left, so it is impossible for X' to have m > n intervals → contradiction. Therefore, our algorithm is optimal.

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

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. Topological Sort Algorithm

- Calculate the in-degree (# of incoming edges) of each vertex ✓
- While every vertex hasn't been removed from the graph ✓
  - find a source, s (source = vertex with in-degree of zero)
  - output s
  - remove s from the graph ✓
  - decrement the in-degree of all of the neighbors of s by 1
- End while

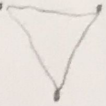
10/10

b. Runtime: O(n+e), where n is the number of vertices, and e is the number of edges

- O(e) to calculate the in-degree of each vertex (have to look at all the edges)
- O(n) to find the source
- O(e) to update the in-degrees of each edge
- Overall: O(n+e)

10/10

Cases
0
1
2
8 odd cycle



Name(last, first

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. The graphs that have property X have the following properties:
- connected
- all vertices have an even degree

-5

Proof
- Say a graph is disconnected and has property X.
- there's no way to go from one component to other w/o lifting your pencil - contradiction

- Say a graph has an odd number of verties of odd-degree
- this is impossible because this graph would have an odd # of total degrees but the total degree of a graph must be even

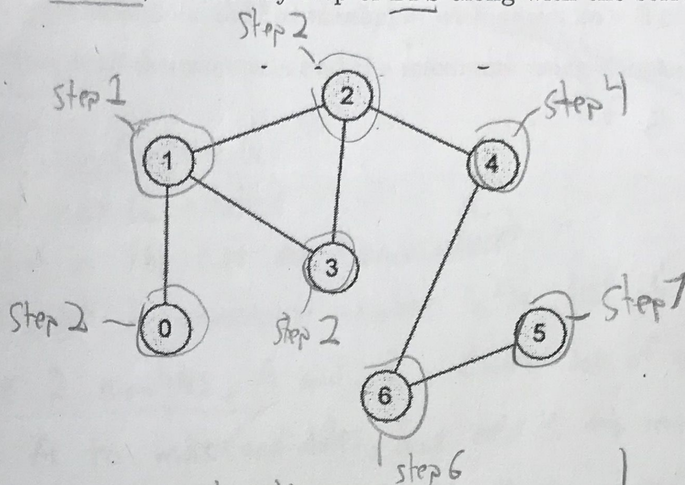
- Say a graph has an even amount of vertices of odd-degree
- You will get stuck because each vertex needs an even number of adjacent vertices
  - half to go that vertex, half to go away from that vertex

b. DFS, but with edges
- initialize each edge as unvisited
- pick an arbitrary edge S, and push it to a stack, and mark it as visited
- While the stack isn't empty
  - pop the top edge, e, off the stack
  - add all of e's unvisited ✓ adjacent edges to the stack, and mark them as visited
- End While

-7

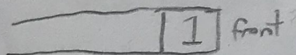
4. (10 points) Consider an unweighted graph G shown below:

(a) Starting from vertex 1, show every step of BFS along with the corresponding FIFO next to it.



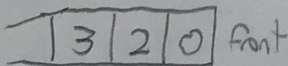
Step 1

- add 1 to the queue and mark as visited



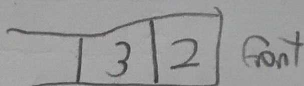
Step 2

- pop 1 from the queue
- add it's unvisited neighbors, 0, 2, and 3, to the queue, and mark them as visited



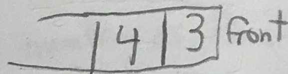
Step 3

- pop 0 from the queue
- check for unvisited neighbors - we have none



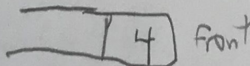
Step 4

- pop 2 from the queue
- add it's unvisited neighbor, 4, to the queue and mark it as visited



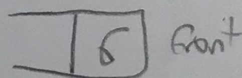
Step 5

- pop 3 from the queue
- check for unvisited neighbors - we have none



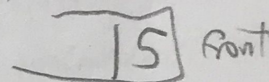
Step 6

- pop 4 from the queue
- add it's unvisited neighbor, 6, to the queue, and mark it as visited



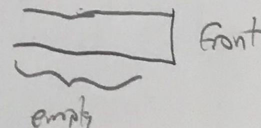
Step 7

- pop 6 from the queue
- add it's unvisited neighbors 5, to the queue, and mark it as visited



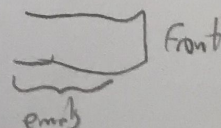
Step 8

- pop 5 from the queue
- check for unvisited neighbors - we have none



Step 9

- the queue is empty, so terminate BFS



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. 1.5n

Algorithm

- make a list called min Candidates
- make a list called max Candidates
- initialize each number in the list as "unassigned"
- While there's at least 1 unassigned number in the list
  - Arbitrarily choose 2 numbers, A and B (w/o loss of generality)
  - if A > B, add A to maxCandidates, and add B to minCandidates
  - if A < B, add A to minCandidates, and add B to maxCandidates
  - if A == B, add A to both minCandidates and maxCandidates
  - mark both A and B as "assigned"
  - if there's only 1 unassigned number left, choose that as A. And arbitrarily choose one assigned number as B. Don't add B to minCandidates or maxCandidates as instructed because it has already been added.
- End While
- While there's more than 1 value in min Candidates
  - Arbitrarily pick 2 values, A and B, (w/o loss of generality)
  - if A < B, remove B from minCandidates
  - if B < A, remove A from minCandidates
  - if A == B, remove B from minCandidates
- End While
- there is one value left in minCandidates - store that in absoluteMin

(back side)

- While there's more than 1 value in maxCandidates
  - Arbitrarily pick 2 values A and B (w/o loss of generality)
  - if A>B, remove B from maxCandidates
  - if A<B, remove A from maxCandidates
  - if A==B, remove B from maxCandidates
- End While
- there is one value left in maxCandidates - store that in absoluteMax
- absoluteMin is the minimum, and absoluteMax is the maximum

Time Complexity Proof

- $n/2$ comparisons to pair up each number, and assign one to minCandidates, and one to maxCandidates
- $\approx n/2 - 1$ comparison in minCandidates
  - the size of minCandidates is $\approx n/2$, and it takes "size-1" comparisons to eliminate all numbers besides the min
- $\approx n/2 - 1$ comparisons in maxCandidates
  - the size of maxCandidates is $\approx n/2$, and it takes "size-1" comparisons to eliminate all numbers besides the max
- Overall: $\approx \frac{n}{2} + (\frac{n}{2} - 1) + (\frac{n}{2} - 1) \approx \frac{3n}{2} - 2 \approx$ $\boxed{\frac{3n}{2} \text{ comparisons}}$ ✓

- The initial splitting is $\approx n/2$ operations because we split up n items into $\approx n/2$ pairs, and do 1 operation per pair
- The min and max calculation both take $\approx \frac{n}{2} - 1$ steps because after comparisons you can eliminate one number, and you eliminate/compare $\approx \frac{n}{2} - 1$ times each

---

Also, say there's a solution where our algorithm doesn't put the max in maxCandidates. This is a contradiction because if any number $\geq$ anything, we put it in maxCandidates,

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

- run a BFS on the graph
- if a vertex is in an odd-numbered level in the BFS tree, color it "Color 1"
- if a vertex is in an even-numbered level in the BFS tree, color it "Color 2"

-4 Time complexity?