UCLA
Computer Science Department

CS180– Midterm
Algorithms & Complexity

10/30/2018

Name: _____
UID: _____

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	20
2	20	17
3	20	14
4	10	10
5	20	20
6	10	6
Total:	100	87

Name(last, first): 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)
- start points and end points of intervals are sorted.
- Initialize a count to zero and a max to zero
- Traverse the start points and end points and add one if we encounter a start point and subtract one if we encounter an end point. Update the max to count if count > max.
- Max is the minimum number of processors we need
- Create max number of processors and set them to available
- Traverse the start and end points a second time. When we encounter an endpoint, set the processor that has the value of the corresponding interval to available. When we encounter a start point, set the interval to the first available processor, and set that processor to the value of that interval.

- In a nutshell, we are putting intervals in the first available processor when we encounter the start of that interval.

b) Proof by contradiction:

Case 1: The algorithm produces an assignment to the minimum number of processors

Case 2: There is an assignment that uses less processors

- This is not possible, as we see because if there is an assignment of less processors, two tasks must simultaneously be assigned to the same processor. We know that there is a place in time where max number of our intervals overlap, and at that time our algorithm uses all processors and at all other times it uses less than all processors. There is no way to use less than all processors when max tasks overlap. We know our algorithm uses the first processor that is available, so we don't use an extra processor when we have an available one.

More on back

~~It is also~~

Case 2 is also possible if, ~~there~~ in our algorithm, there is an empty processor when max ~~processors~~ intervals overlap. We see this is impossible because our algorithm uses the first available processor, so there will not be an extra available processor when max intervals overlap

14/20

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) This is topological sort

- Find a source and output the source(vertice w/ only outgoing edges)
- Remove the source from the graph along with all of its outgoing edges.
- Repeat the previous two steps until the graph is empty

10/10

b) O(n+e)

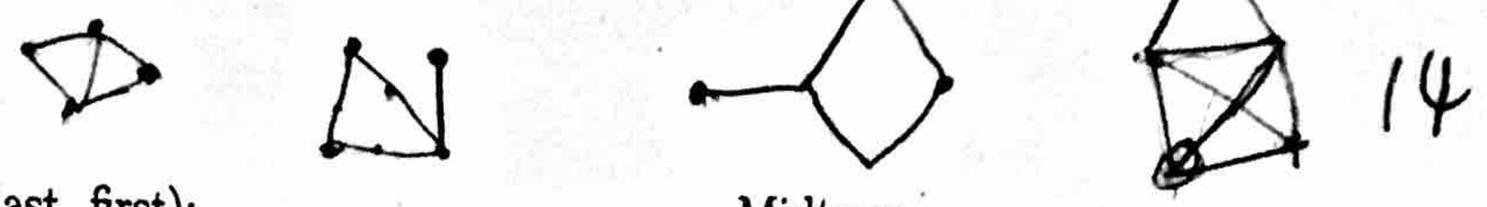
because

O(n): every vertice must be visited and removed from the graph O(n+n)

O(e): every edge must be removed from the graph O(e)

find source? O(n+e)

7/7



14

Name(last, first): Midterm 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) A graph has property X if and only if

AND { OR { • there are no vertices with odd degree
OR { • there are two vertices with odd degree ✓
• The graph is connected

b) • If there are vertices of odd degree we must start with an arbitrary vertice of odd degree
• Run a modified DFS from the odd vertice w/ these modifications:
- Do not track visited vertices
- Track visited edges BUT
- When backtracking, remove visited edges from visited list
- Record failed paths and do not repeat them

-b

10

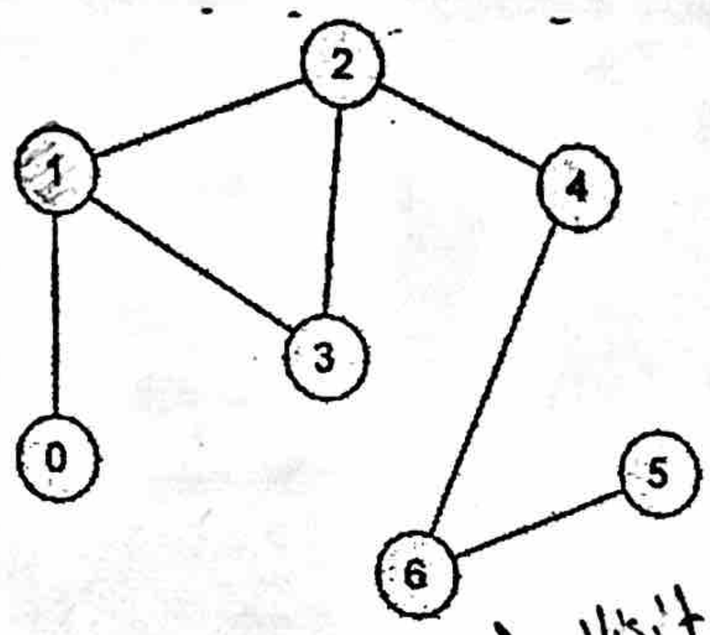
Name(last, first):

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.



1) push 1 in queue

← 1 ←

queue

Visited

0 1 2 3 4 5 6

2) Visit 1, add neighbors

← 0 2 3 ←

queue

Visited

0 X 2 3 4 5 6

3) Visit 0, add neighbors

← 2 3 ←

queue

Visited

Ø X 2 3 4 5 6

4) Visit 2, add neighbors

← 3 4 ←

queue

Visited

Ø X X 3 4 5 6

5) Visit 3, add neighbors

← 4 ←

queue

Visited

Ø X X X 4 5 6

5) Visit 4, add neighbors

← 6 ←

queue

Visited

Ø X X X X 5 6

6) Visit 6, add neighbors

← 5 ←

queue

Visited

Ø X X X X X 6

7) Visit 5, add neighbors

←

queue

Visited

Ø X X X X X X

8) Queue is empty, check if all vertices are visited

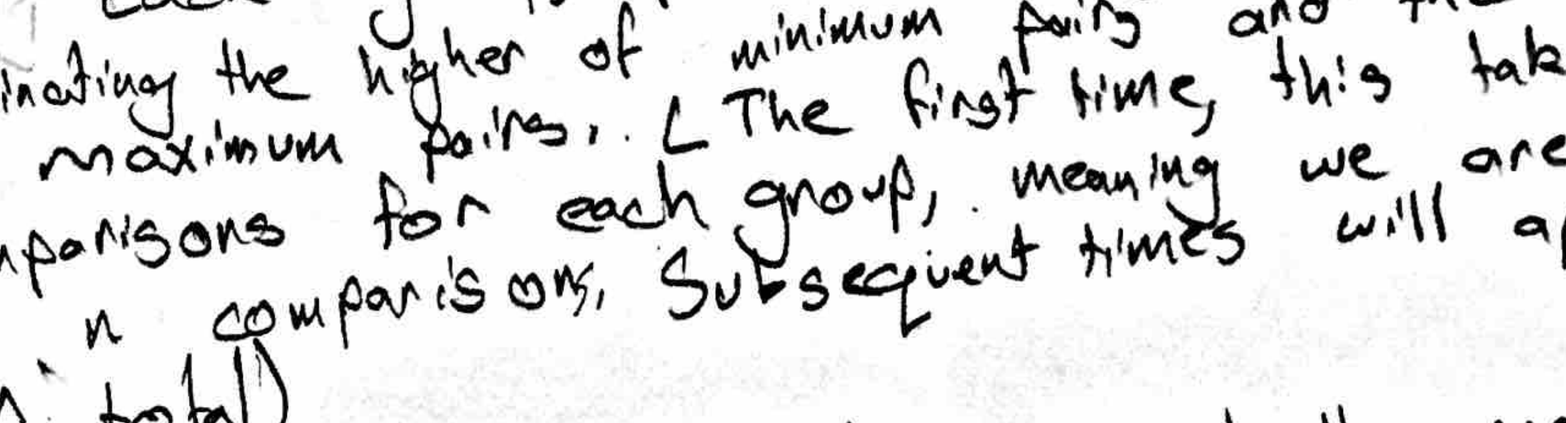
9) They are, so you're done!

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.

n (2n-3) (2(n-1))-3 2n-2-3 = 2n-5 $\frac{n}{2}$

$\frac{2n-1}{2}$ $\frac{2n-3}{2}$

This going to require splitting the list up and comparing minimums of parts of the list.



Ab

- Split the list of unsorted integers into pairs which is $\frac{n}{2}$ comparisons)
  - Now we have half the list as contenders for the minimum and the other half as contenders for the maximum,
- In each group, pair them again and compare, eliminating the higher of minimum pairs and the lower of maximum pairs. (The first time, this takes $\frac{n}{4}$ comparisons for each group, meaning we are now at n comparisons. Subsequent times will approach $\frac{3n}{2}$ total)
- With the remaining contenders, repeat the previous step until we've eliminated all but one maximum and all but one minimum.

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.

- Start at an arbitrary vertex and color it color 1 and mark this vertice as visited
- Put all of its unvisited neighbors into a FIFO queue and attempt to color ALL neighbors the opposite color of the current vertex (color 2 in the first case, but if we are visiting neighbors of a color 2, then color 1)
  - If you attempt to color an already colored vertice a different color than it is colored, break out of the algorithm and return false; the graph is not bipartite
- Pop the queue and repeat the previous step but with the vertice we popped, then mark this vertice as visited.
- Repeat steps 2 and 3 until the stack is empty. If we reach the end without breaking, the graph is bipartite.

-4 Time complexity