UCLA
Computer Science Department

CS180– Midterm
Algorithms & Complexity

10/30/2018

Name: Shunning Ma Discussion 1B
UID: 204996819

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	19
3	20	20
4	10	10
5	20	20
6	10	6+4
Total:	100	95

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) First assume the intervals are sorted by starting time from earliest starting time to latest.
Algorithm:
for each interval I_i that is sorted in above manner start from the earliest starting one.
label the processors numbers 1, 2, 3, ...
assign an interval to the lowest number labeled processor that has no processing interval at the starting time of I_i.
After all the intervals are processed, the highest label number of processor being used is the minimum number of processors.

b) Let the earliest starting point to the latest ending point of all given intervals T_{total}. for every time t in T_{total}, there should be at lease number of processors n equal to the number of intervals that contain time t.
We want to show that the algorithm above find the minimum number of processor equal to the maximum n in T_{total}.
Suppose the above algorithm find number of processors bigger than maximum number of n. since we assign a interval to the lowest labeled processor #n+1, there should be at least n interval that happen at the same time in the given time point. and the one going to be assigned must be at least the (n+1)th, which contradicts with the fact that there are n interval at most happen at one time in T_{total}, so it is correct

19/20

Name(last, first): Midterm 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.

we first assign each vertex in the graph 2 properties: 2 numbers n-in and n-out, and initially those numbers are all zero

a) run topological sort on such a graph the output of the topological sort is the desired output for this problem.

b) in the topological sort, then, we trace through all the edges (i, j) within the graph, for each (i, j), we add one to n-out of vertex i, add one to n-in of vertex j. Since this step take constant time for each edge, and repeat the number of edges time, this is O(e), then we find one vertex that has n-in of 0, and output it, this is totally O(n) for n output vertices, then we remove every edges that start from this vertex to any other vertex, and for those vertices that are connect by such edges, minus one of their n-in property. repeat this step until all the vertices are outputed, since this take constant time for each vertex, and removing edges happens at most once for each of edge, so this is O(n+e) add all the steps above, the algorithm is totally O(n+e)

19/20

20

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) Since for every vertex, when travel through, 2 edges connected to it will be consumed. So for odd-number degree vertices, they should either be starting point or ending point, according to pigeonhole principle, there should be at most 2 such vertices. And since every edge add $\sum$degree(n) with 2, there cannot be odd number of odd-number degree vertices

So, in conclusion, graph has property X classified as following:

1. graph having 0 odd-number degreed vertex
2. graph having 2 odd-number degreed vertices.

(all undirected and connected)

b) choose an arbitrary vertex and run DFS start from it with the following additional rules:

start from the starting vertex, do DFS search, and record every vertex in a list until 1 of 2 condition happens: 1. for a given point, when searching one vertex's neighbors, if search a vertex already in the list, this is a cycle, save the current list and start recording in a new list and continue do DFS

2. for a given point a vertex has no unreached neighbors, in this case, restart the DFS start from this point (which means this is a odd-degreed vertex).

Continue the algorithm until all the vertices are searched.

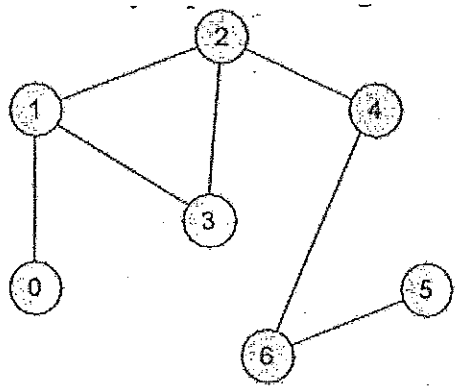
searching from the already searched neighbor to it once again to record a cycle, call it a cycle list

if there's a previous part of the list before the interval stated above, save it in another list, call it a non-cycle list

Then, for each of two cycle list, if found their point in common, combine them to one cycle list, after all cycle list are combined, find the final cycle list with the rest of the list common point(s), when found, the traversal should be start from the non-cycle list first (if exist), attach to the cycle list, then another non-cycle list (if exist) (attach the lists from the common points)

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.



first, see every vertices unreached — queue: (start from left) (denote this queue as S)

S is empty, push 1. — see them reached — 1

push every 1's unreached neighbor — see them reached — 1 0 3 2

output front of S, pop it — 0 3 2

push every 0's unreached neighbor — see them reached — 0 3 2

output 0 and pop it — 3 2

push 3's unreached neighbors, set reached — 3 2

output 3 and pop — 2

push 2's unreached neighbors, set reached — 2 4

output 2 and pop — 4

push 4's unreached neighbors, set reached — 4 6

output 4 and pop — 6

push 6's unreached neighbors, set reached — 6 5

output 6 and pop — 5

push 5's unreached neighbors, set reached — 5

output 5 and pop — empty

S is empty, end of 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.

for n unsorted integers in one list.
take every two of the n integers as a pair (if n is odd then there is one last integer it itself is a pair), which totally $\lceil \frac{n}{2} \rceil$ pairs.
inside each of the pair, compare the two numbers, in total there should be $\lceil \frac{n}{2} \rceil$ comparison.
put the larger integer of each pair in List_max,
put the smaller integer of each pair in List_min.
then, do the pairing with the same manner above for List_max and List_min, and within each pairs, do comparison again, totally $\lceil \frac{n}{4} \rceil + \lceil \frac{n}{4} \rceil = \lceil \frac{n}{2} \rceil$ comparisons.
then, ~~put the larger integers from the pairs in List_max~~
remove the smaller integers from the pairs in List_max
remove the larger integers from the pairs in List_min.
and for the rest of the two lists, do the pairing in same manner again, and do comparisons within pairs again, this time totally $\lceil \frac{n}{8} \rceil + \lceil \frac{n}{8} \rceil = \lceil \frac{n}{4} \rceil$ comparisons.
repeat this process until there are only one number in List_max and only one number in List_min. and the last number in List_max is the maximum, the last number in List_min is the minimal
this takes in total $\lceil \frac{n}{2} \rceil + \lceil \frac{n}{2} \rceil + \lceil \frac{n}{4} \rceil + \lceil \frac{n}{8} \rceil + \cdots + 2 + 1 \approx \frac{3n}{2}$ comparisons.

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.

Suppose the 2 colors are C_1, C_2, respectively.
Start from arbitrary vertex. color this vertex C_1.
then: run BFS ~~with the following additional rules:~~
~~we see that for level 0 of the BFS~~
and output each of the lists of different levels.
for ~~each number~~ each of i-th level,
if i is odd, then color all the vertices in this level as C_2
if i is even, then color all the vertices in this level as C_1

-4 Time complexity