97

Name(last, first): [redacted]

Dis. 1A

UCLA Computer Science Department

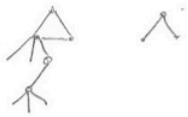
CS 180 **Algorithms & Complexity** **ID :** [redacted]

Midterm **Total Time: 1.5 hours** **October 27, 2016**

Each problem has 20 points.

All algorithm should be described in English, bullet-by-bullet

1 **a.** Describe Depth First Search on an undirected and un-connected graph.
b. Analyze the time complexity of DFS when there are no cycles.



a). Let Graph $G = (V, E)$. with k components $(C_1, C_2, \ldots, C_k)$

1. Pick an arbitrary component C_i (not-visited already)
2. Pick an arbitrary node $n \in V$ & $n \in C_i$
3. Examine n's children $(n_1', n_2', \ldots n_n')$
4. If a child n' has already been visited, don't add to visited set of vertices.
5. If there is at least one child, return to step 2 with new node n' that is a child of n. & list node n as explored.
6. return to step 1 & pick another arbitrary component (& list node n as totally explored)
7. continue steps $1 \to 5$ until all components explored.

+15

b) Step 2 & 3 collectively visits V nodes total
To examine, must traverse down height of components (visit each edge)

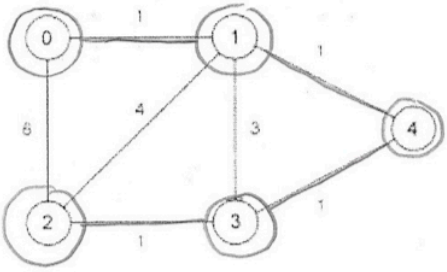
$O(E + V)$

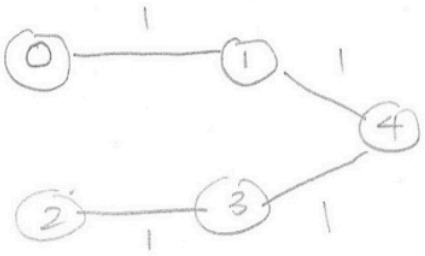
+5

20

Name(last, first) [redacted]

2 . a. Use Prim's MST algorithm to find an MST in this graph. Show each step on the graph shown below.
b. If some edges were negative would the algorithm still find an MST. Why?





a) Pick an arbitrary node (will use ∅).

1. $\ell(0) = 0$ (add 0)
   ↳ edges available 01, 02
   $\ell(01) = 1$ $\ell(02) = 5$
   ↳ min $\ell(01)$. & no cycle

2. $\ell(01) = 1$ (add 1)
   ↳ edges available 02, 12, 13, 14
   $\ell(02) = 5$ $\ell(13) = 3$
   $\ell(12) = 4$ $\ell(14) = 1$
   ↳ min $\ell(14)$ & no cycle

3. $\ell(14) = 1$ (add 4)
   ↳ edges available 02, 12, 13, 34
   $\ell(02) = 5$ $\ell(13) = 3$
   $\ell(12) = 4$ $\ell(34) = 1$
   ↳ min $\ell(34)$ & no cycle

4. $\ell(34) = 1$ (add 3).
   ↳ edges available 02, 12, 13, 23
   $\ell(02) = 5$ $\ell(13) = 3$
   $\ell(12) = 4$ $\ell(23) = 1$
   ↳ min $\ell(23)$ & no cycle

5. $\ell(23) = 1$ (add 2)
   ↳ edges available 02, 12, 13
   ↳ any edge added would create a cycle

Stop
NO more nodes. stop.

b) Yes, since we are not adding edge weights to find a path (just picking minimum edge weights alow) negative edges would still produce an MST, Here the smallest edges would be negative.

+5

20

-15

Name(last, first): [redacted]

20

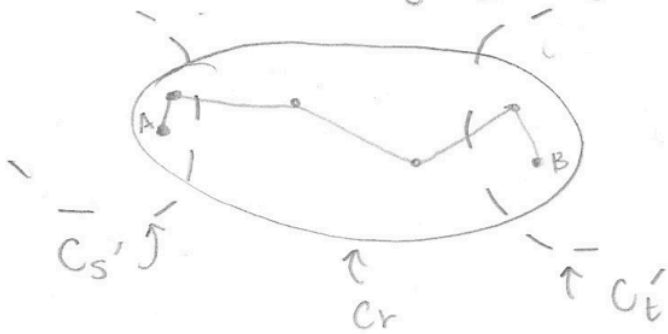
3. Prove that solving the k-clustering problem (described in class and in the book) using Kruskal's MST algorithm, produces an optimal clustering. That is, it will produce an optimal set of clusters C1, C2, …, Ck with Maximum cluster distances. (Use a figure to better describe your proof: as was done in class / book).

Proof : Assume that we have some k-clustering $C_1, C_2, \dots, C_k$. (with kruskal's)

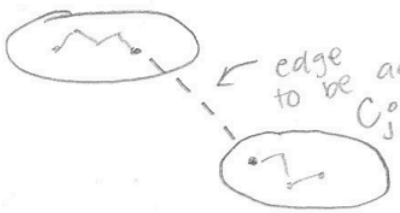
Assume there is an optimal algorithm with some k-clustering $C_1', C_2', \dots, C_k'$.

if $C_1 = C_1'$, $C_2 = C_2'$, $\dots$, $C_k = C_k'$ we're done kruskal's method optimal.

If not, then there is some C_r in our clustering that has nodes in optimal algorithm's 2 clusters C_s' & C_t'.



Now we examine the path between nodes A & B in C_r.



An edge to be added between C_r & C_r must be more than any edge already in C_r (by MST theorem).

If we were to break an edge in C_r (as done to get C_s' & C_t') this edge is less than the next edge to be added, which is a contradiction to assuming optimal clustering with max distances!

∴ our algorithm is optimal

Name(last, first): [redacted]

20

4. Consider a sequence of n real numbers X = (x1, x2, …, xn).
a. Design an algortihm to partition the numbers into n/2 pairs. We call the sum of each pair S1, S2, …, Sn/2. The algorithm should find the partitioning that minimizes the maximum sum.
b. Analyze the time comlpexity of your algorithm.

cont. on back ex. 4 2 0 6 $n/2 = 6$
S_i (4,2)
S_2 (0,6)

1 4 2 6 $n/2 = 6$
S_1 (1,6)
S_2 (2,4)

a) wants each pairs sum to be minimal.

$1 \leq i \leq n/2$ for S_i

↳ pair up min & max.

First thoughts: sort sequence then pair up min max.

~~Brute force is $O(n^2)$, try each pair.~~ How ?

Algorithm:

1. Divide sequence of numbers X into 2 halves.
2. If each half size of 1 element, x_i & x_j
   if $x_i \leq x_j$
   merge into $\{x_i, x_j\}$
   return to last recursive call
   if $x_i > x_j$
   merge into $\{x_j, x_i\}$
3. If halfs not 1 element,
   return to step 1. for left half
   return to step 1 for right half
   merge them.

apply a merge sort

4. once sequence sorted via merge algorithm,
   set pointers to first & last elements of sequence (pointer at x_i & pointer at x_j) initially here $i=1$ $j=n$
5. Set $S_1 = x_i + x_j$
6. Move pointer on the left forward one to x_{i+1}
7. Move pointer on the right back one to x_{j-1}
8. return to step 5. with new elements x_{i+1} & x_{j-1}
9. If $i=j$, or $i > j$
   stop. pairs complete.

10

b. Time complexity: Merge sort takes $O(n \log n)$, traversing to get pairs iterates $n/2$ times

b) cont. since merge $\Rightarrow O(n \log n)$
pairs $\Rightarrow O(n/2)$

5

$$O(n \log n + n/2) = O(n \log n)$$

proof: Assume there is some optimal pairing S' compared to my pairing S. Read x_1 smallest elem to x_n largest.

For S_1' and S_1, if same, great we move on to subproblem of pairings $n/2 - 1$

If $S_1' \neq S_1$, then there is some $S_1' = x_i + x_j$ & mine is $S_1 = x_1 + x_n$

↳ if $i \neq 1$, then since my algorithm sorts in non decreasing order

$x_i > x_1$ & $S_1' > S_1$ so our algorithm is better.

5

↳ if $j \neq n$, then $x_j < x_n$. However since $x_j \neq x_n$, there is some x_k in S' where $x_k = x_n$. $S_i' = x_k + x_m$ with this, then that avg sum between S_i' & S_i' will either be larger than or same avg-sum in S. Then the sums will not be minimized because S_i' will be either as good or worse than S_i. so our algorithm is at least as good.

$S_i = x_0 + x_m$
$x_0 \leq x_k = x_m$
so $S_i \leq S_i'$

If you do this with the subproblem with sequence $\{x_2, x_3, \ldots, x_{n-1}\}$ this holds true also, so inductively the algorithm holds

(17)

Name(last, first): [redacted]

5. Let G be a DAG and let K be the maximal number of edges in a path in G.
a. Design an algorithm that divides the vertices into at most k+1 groups such that for each two vertices v and w in the same group there is no path from v to w and there is no path from w to v.
b. Analyze the time complexity of your algorithm. (proof on back)

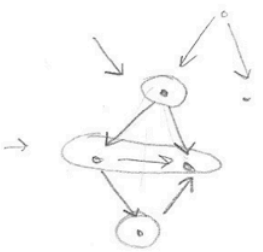
a) (1)

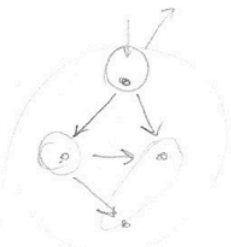
since a DAG consider making use of topological sort
(like using kruskals for K-clustering)
↳ need priority for direction & group by level.

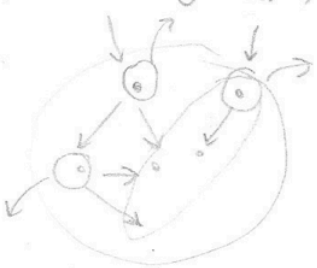
1. set all vertices into one group initially.
2. start at arbitrary vertex n in G.
3. perform a topological ✓sort from vertex n.
4. If n has no vertices connected to it, do nothing & return to step 2 w/ a new non-visited vertex.
5. If n has at least 1 vertex connected.
   ↳ break node n from group
   ↳ create a new group i for node n.
6. If n has vertices connected,
   ↳ once visited all connected vertices from n, continue w/ topological sort on next vertex n' connected from n.
7. ✓ once all vertices explored, done.

b) (5)

Topological sort uses O(V+E), since we only visit each node once due to sort & decide if it stays or leaves group, O(V+E).







5. Proof:

(2)

Best case scenario, topological sort shows that none of the vertices are connected, my algorithm in step 1) puts them into 1 group.

Assume DAG G has some connected components, show each group has no $v \to w$ & $w \to v$.

Assume there is a valid grouping with a group where $v \to w$. $v, w \in V$ in $G(V, E)$ By my algorithm, if $v \to w$, the edge would have been traversed by the topological sort. If traversed, then v would have been ejected from the group that held v & w, which is a contradiction to my assumption.

If edge not traversed $v \to w$, then by my algorithm there is no edge (since all edges explored), which is a contradiction. This can be done for $w \to v$ since v & w are arbitrary.

true in best case only

Therefore this unstable grouping cannot occur & my algorithm is correct.

prob is at most $k+1$?