CS570

Analysis of Algorithms

Summer 2015

Exam III

Name: ______________________

Student ID: __________________

Email Address:________________

_____Check if DEN Student

	Maximum	Received
Problem 1	20	
Problem 2	16	
Problem 3	16	
Problem 4	16	
Problem 5	16	
Problem 6	16	
Total	100	

Instructions:

1. This is a 2-hr exam. Closed book and notes
2. If a description to an algorithm or a proof is required please limit your description or proof to within 150 words, preferably not exceeding the space allotted for that question.
3. No space other than the pages in the exam booklet will be scanned for grading.
4. If you require an additional page for a question, you can use the extra page provided within this booklet. However please indicate clearly that you are continuing the solution on the additional page.

1) 20 pts
Mark the following statements as **TRUE** or **FALSE**. No need to provide any justification.

[TRUE/]
If all edge capacities in a flow network are integer multiples of 7, then the maximum value of flow is a multiple of 7.

[/FALSE]
If P = NP, then all NP-Hard problems can be solved in Polynomial time.

[/FALSE]
Let T be a complete binary tree with n nodes. Finding a path from the root of T to a given vertex v ∈ T using breadth-first search takes O(log n) time. $O(n)$

[TRUE/]
Halting Problem is an NP-Hard problem.

[TRUE/]
Every decision problem in P has a polynomial time certifier.

[/FALSE]
In a flow network, if we increase the capacity of an edge that happens to be on a minimum cut, this will increase the max flow in the network.

[/FALSE]
If the capacity of every arc is odd, then there is a maximum flow in which the flow on each arc is odd.

[TRUE/]
If the edge weights of a weighted graph are doubled, then the number of minimum spanning trees of the graph remains unchanged

[/FALSE]
The linear programming solution to the shortest path problem discussed in class can fail in the presence of negative cost edges.

[/FALSE]
In a divide and conquer solution, the sub-problems are disjoint and are of the same size.

Sort jobs on ↑ duration $j_1 \cdot\cdot j_n$ opt. sol. have no gap

Schedule jobs one by one with no gaps.

$f_{j_1} \le f_{o_1}$

2) 16 pts

You are given n jobs of known duration $t_1, t_2, \ldots, t_n$ for execution on a single processor. All the jobs are given to you at the start and they can be executed in any order, one job at a time. We want to find a schedule that minimizes **the total time spent by all the jobs** in this system. The **time spent** by one job is the **sum of the time spent on waiting plus the time spent on its execution**. In other words, the total time spent by all jobs is the total sum of their finish times. Give an efficient solution for this problem. Analyze the complexity of your solution and prove that it is optimal.

Solution: We can use a greedy approach to solve this problem. Sort the jobs in increasing order of duration and schedule the jobs based on the sorted order, first job in the sorted order is scheduled first. Complexity of the approach is O(nlogn) algorithm. To prove that the algorithm is optimal, we can use proof by exchange technique.

Assume that there is optimal way to schedule jobs other than our greedy approach. Assign jobs a rank which is the index in the increasingly sorted array. If the array is not increasingly sorted then we can find adjacent jobs, j and j+1 such that rank of j is greater than rank of j+1. Let S be the starting time of job j, then $S + t_j$ is the finishing time of job j and $S + t_j + t_{\{j+1\}}$ is the finishing time of job j+1. When the jobs j and j+1 are only swapped, we can observe that the finishing times of other jobs are not affected. Also the total time spent by the jobs after swapping is not worse than the before as $S + t_{j+1} + S + t_{\{j+1\}} + t_j \le S + t_j + S + t_j + t_{\{j+1\}}$, since $t_j \ge t_{\{j+1\}}$.

We can keep on swapping such jobs until we reach our greedy solution. So our greedy solution is no worse than the optimal solution, so our greedy solution is also optimal.

3) 16 pts
For bit strings X = $x_1 \dots x_m$, Y = $y_1 \dots y_n$ and Z = $z_1 \dots z_{m+n}$, we say that Z is an interleaving of X and Y if it can be obtained by interleaving the bits in X and Y in a way that maintains the left-to-right order of the bits in X and Y. For example if X = 101 and Y = 01 then $x_1x_2y_1x_3y_2$ = 10011 is an interleaving of X and Y, whereas 11010 is not. Give the most efficient algorithm you can to determine if Z is an interleaving of X and Y. Prove your algorithm is correct and analyze its time complexity as a function m = |X| and n = |Y|.

Solution

The general form of the subproblem we solve will be: determine if Z = $z_1 \dots z_{i+j}$ is an interleaving of $x_1 \dots x_i$ and $y_1 \dots y_j$ for $0 \le i \le m$ and $0 \le j \le n$. Let c[i; j] be true if and only if $z_1 \dots z_{i+j}$ is an interleaving of $x_1 \dots x_i$ and $y_1 \dots y_j$. We use the convention that if i = 0 then $x_i = \lambda$ (the empty string) and if j = 0 then $y_j = \lambda$. The subproblem c[i, j] can be recursively defined as shown (where c[m, n] gives the answer to the original problem):

$$c[i; j] = \begin{cases} \text{true} & \text{if } i = j = 0 \\ \text{false} & \text{if } x_i \neq z_{i+j} \text{ and } y_j \neq z_{i+j} \\ c[i - 1, j] & \text{if } x_i = z_{i+j} \text{ and } y_j \neq z_{i+j} \\ c[i, j - 1] & \text{if } x_i \neq z_{i+j} \text{ and } y_j = z_{i+j} \\ c[i - 1, j] \text{ OR } c[i, j - 1] & \text{if } xi = y_j = z_{i+j} \end{cases}$$

We now argue this recursive definition is correct. First the case where i = j = 0 is when both X and Y are empty and then by definition Z (which is also empty) is a valid interleaving of X and Y . If $x_i \neq z_{i+j}$ and $y_j = z_{i+j}$ then there could only be a valid interleaving in which xi appears last in the interleaving, and hence c[i; j] is true exactly when $z_1 \dots z_{i+j-1}$ is a valid interleaving of $x_1 \dots x_{i-1}$ and $y_1 \dots y_j$ which is given by c[i – 1, j]. Similarly, when $x_i \neq z_{i+j}$ and $y_j = z_{i+j}$ then c[i, j] = c[i - 1; j]. Finally, consider when $x_i = y_j = z_{i+j}$. In this case the interleaving (if it exists) must either end with xi (in which case c[i - 1, j] is true) or must end with y_i (in which case c[i, j - 1] is true). Thus returning c[i – 1, j] OR c[i, j - 1] gives the correct answer. Finally, since in all cases the value of c[i, j] comes directly from the answer to one of the subproblems, we have the optimal substructure property.
The time complexity is clearly O(nm) since there are n.m subproblems each of which is solved in constant time. Finally, the c[i, j] matrix can be computed in row major order.

4) 16 pts

You are given two decision problems L1 and L2 in NP. You are given that L1 $\le_p$ L2. For each of the following statements, state whether it is true, false or an open question. Justify your answers.

(a) If L1 $\in$ P, then L2 $\in$ P.

(b) If L1 $\in$ NP-complete, then L2 $\in$ NP-complete.

(c) If L2 $\in$ P, then L1 $\in$ P.

(d) Suppose L2 is solvable in O(n). Then, L1 is also solvable in O(n).

Solution:

(a) **Open question**. If P = NP, then L2 $\in$ P. Otherwise, L2 can be P, NP or NP-complete.

(b) **True**. L2 is NP-complete, since we can convert any problem in NP to problem regarding L1 in polynomial time. Therefore, if we can reduce problems of L1 to problems in L2 in polynomial time, we can reduce all problems in NP to problems in L2. This is the technique we use to prove NP-completeness – by reducing an unknown problem to a known NP-complete problem.

(c) **True**. We can solve problems in L1 by first reducing it to problems in L2 in polynomial time, and then use the polynomial time algorithm for L2 solve the reduce problem. This gives a polynomial time algorithm for L1.

(d) **False**. The reduction L1 $\le_p$ L2 may take more than linear time, therefore, we are not guaranteed a linear time algorithm for L1. We are guaranteed that L1 $\in$ P can be solved in polynomial time – time to reduce L1 to L2 plus the time to solve problem in L2.

5) 16 pts
USC Admissions Center needs your help in planning paths for Campus tours given to prospective students or interested groups. Let USC campus be modeled as a weighted, directed graph G containing locations V connected by one-way roads E. On a busy day, let *k* be the number of campus tours that have to be done at the same time. It is required that the paths of campus tours do not use the same roads. Let the tour have k starting locations $A = \{a_1, a_2, . . . , a_k\} \subset V$. From the starting locations the groups are taken by a guide on a path through G to some ending location in $B = \{b_1, b_2, . . . , b_k\} \subset V$. Your goal is to find a path for each group *i* from the starting location, a_i , to any ending location b_j such that no two paths share any edges, and no two groups end in the same location bj .

(a) Formulate this as a flow problem. Find an algorithm (if any) to find *k* paths ai -> bj that start and end at different vertices and that share no edges, and briefly justify the correctness of your algorithm. (8 pts)

(b) It is still possible that some paths share a vertex. Modify your algorithm to find *k* paths ai -> bj, that start and end in different locations and that share neither vertices nor edges. (8 pts)

Solution:

(a) The complete algorithm is as follows:

1. Create a flow network G' containing all vertices in V, all directed edges in E with capacity 1, and additionally a source vertex s and a sink vertex t. Connect the source to each starting location with a directed edge (s, ai) and each ending location to the sink with a directed edge (bi, t), all with capacity 1.

2. Run Ford-Fulkerson on this network to get a maximum flow f on this network. If |f| = k, then there is a solution; if |f| < k, then there is no solution, so we return FALSE. We will later show that it is always the case that |f| <= k.

3. To extract the paths from ai to bj (as well as which starting location ultimately connects to which ending location), run a depth-first search on the returned max flow f starting from s, tracing a path to t. Remove these edges and repeat k times until we have the k disjoint paths.

To **justify** correctness, any argument conveying that there is a flow of size k if and only if there are k disjoint paths is correct.

(b) Duplicate each vertex *v* into two vertices v_{in} and v_{out}, with a directed edge between them. All edges (u, v) now become (u, v_{in}); all edges (v,w) now become (v_{out},w). Assign the edge $(v_{in}, vout)$ capacity 1. With this transformation, we now have a graph in which there is a single edge corresponding to each vertex, and thus any paths that formerly shared vertices would be forced to share this edge. Now, we can use the same algorithm as in part (a) on the modified graph to find k disjoint paths sharing neither edges nor vertices, if they exist.

6) 16 pts

A company makes three products and has 4 available manufacturing plants. The production time (in minutes) per unit produced varies from plant to plant as shown below:

		Manufacturing Plant			
		1	2	3	4
Product	1	5	7	4	10
	2	6	12	8	15
	3	13	14	9	17

Similarly the profit ($) contribution per unit varies from plant to plant as below:

		Manufacturing Plant			
		1	2	3	4
Product	1	10	8	6	9
	2	18	20	15	17
	3	15	16	13	17

If, one week, there are 35 working hours available at each manufacturing plant how much of each product should be produced given that we need at least 100 units of product 1, 150 units of product 2 and 100 units of product 3. Formulate this problem as a linear program. You do not have to solve the resulting LP.

Solution

Variables:

At first sight we are trying to decide how much of each product to make. However on closer inspection it is clear that we need to decide how much of each product to make at each plant. Hence let

xij = amount of product i (i=1,2,3) made at plant j (j=1,2,3,4) per week.

Although (strictly) all the xij variables should be integer they are likely to be quite large and so we let them take fractional values and ignore any fractional parts in the numerical solution. Note too that the question explicitly asks us to formulate the problem as an LP rather than as an IP.

Constraints:

We first formulate each constraint in words and then in a mathematical way.

- Limit on the number of minutes available each week for each workstation

$5x11 + 6x21 + 13x31 <= 35(60)$

$7x12 + 12x22 + 14x32 <= 35(60)$

$4x13 + 8x23 + 9x33 <= 35(60)$

$10x14 + 15x24 + 17x34 <= 35(60)$

- Lower limit on the total amount of each product produced

$x11 + x12 + x13 + x14 >= 100$

$x21 + x22 + x23 + x24 >= 150$

$x31 + x32 + x33 + x34 >= 100$

All variables are greater than equal to zero.

Objective: Maximize total profit.

Maximize

$10x11 + 8x12 + 6x13 + 9x14 + 18x21 + 20x22 + 15x23 + 17x24 + 15x31 + 16x32 + 13x33 + 17x34$

Additional Space