College of Computer & Information Science
Northeastern University
CS5800: Algorithms

Spring 2011
Handout 23
10 April 2011

Sample Solution to Problem Set 5

1. (7 + 7 = 14 points) Project management

Suppose you are a high-level manager in a software firm and you are managing n software projects. You are asked to assign m of the programmers in your firm among these n projects. Assume that all of the programmers are equally competent.

After some careful thought, you have figured out how much benefit i programmers will bring to project j. View this benefit as a number. Formally put, for each project j, you have computed an array $A_j[0..m]$ where $A_j[i]$ is the benefit obtained by assigning i programmers to project j. Assume that $A_j[i]$ is nondecreasing with increasing i.

(a) Nonincreasing marginal benefits: Make the economically sound assumption that the marginal benefit obtained by assigning an ith programmer to a project is nonincreasing as i increases. Thus, for all j and $i \geq 1$, $A_j[i+1] - A_j[i] \leq A_j[i] - A_j[i-1]$.

Design a greedy algorithm to determine how many programmers you will assign to each project such that the total benefit obtained over all projects is maximized. Justify the correctness of your algorithm and analyze its running time.

Answer: We assign programmers one at a time, from 1 to m. At the ith step, we greedily assign the ith programmer to the project to which the programmer will bring the most marginal benefit. Note that the marginal benefit that the ith programmer brings to a project j depends on the particular assignment of the previous $i-1$ programmers; it may not be the same as $A_j[i]$.

We now prove that the above greedy choice leads to an optimal solution using a standard "swapping argument". Our main claim is this: for all i, the assignment obtained after the ith step is the subset of an optimal assignment. We prove the claim by induction.

The induction basis is trivial since we start with an empty assignment which is clearly a subset of an optimal assignment. As the induction hypothesis, we assume that the claim holds after step $i-1$. That is, the assignment obtained after step $i-1$ is a subset of an optimal assignment, say S. We now consider the state after step i. Let us suppose that the greedy choice assigns the ith programmer to project j, thus resulting in k programmers for project j. If S has assigned at least k programmers to project j, then there is nothing to prove. (Note that by the induction hypothesis, S has assigned at least $k-1$ programmers to project j.) Consider the case in which S has assigned only $k-1$ programmers. This implies that there must be some other project j' such that S has assigned more programmers than the greedy assignment has assigned thus far. By the definition of our greedy choice and the assumption about marginal benefits, it follows that if we modify S by removing one programmer from project j' and move it to j, the change in total benefit cannot decrease. Therefore, the new assignment thus obtained by modifying S is optimal. It also contains the current greedy assignment, thus establishing the induction step.

We now consider the running time of the greedy algorithm. We have m iterations and in each iteration we are determining the project for which the marginal benefit will be maximum. Thus, a naive n-element comparison to determine the maximum will take $O(n)$ time, leading to a runtime of $\Theta(nm)$. One can do better by maintaining a priority queue in which we store an element for each project, representing the marginal benefit the next programmer will bring to the project. Initially, we add n entries to a priority queue, one for each of the projects, representing the marginal benefit that the first programmer brings to the project. Using heaps, we can implement this in $O(n)$ time. Each iteration then translates to one "extract-max" operation (to find the project to which the programmer will be assigned) and an insert operation to insert a new element giving the marginal benefit the next programmer will bring to the chosen project. Each iteration thus takes $O(\log n)$ time, thus leading to a total runtime of $O(n + m \log n)$.

(b) General benefits: Now *remove* the assumption made above that the marginal benefit obtained by assigning an ith programmer to a project is nonincreasing as i increases. Design a dynamic programming algorithm to determine how many programmers you will assign to each project such that the total benefit obtained over all projects is maximized. Analyze its running time.

Answer: Let $C[i, j]$ denote the total benefit obtained in the optimal assignment i programmers among projects 1 through j. Let $B[i, j]$ denote the number of programmers assigned to the jth project in this optimal solution. We can establish the following recurrences for $C[i, j]$ and $B[i, j]$. We set $C[0, j]$ to 0 for all j. Furthermore, for all i, $C[i, 1]$ equals $A_1[i]$ and $B[i, 1]$ equals i. For $i > 0$ and $j > 1$, we have the following. Let k be such that $A_j[k]+C[i-k, j-1] = \max_{0\le\ell\le i}\{A_j[\ell]+C[i-\ell, j-1]\}$. Then, $C[i, j] = A_j[k]+C[i-k, j-1]$ and $B[i, j] = k$.

The algorithm proceeds by iteratively computing $C[i, j]$ and $B[i, j]$, i going from 1 through m and j going from 1 through n. The final assignment can be computed using the B array. In each iteration, we need to take the maximum of at most m entries. Therefore, there are at most nm iterations, each taking $O(m)$ time, leading to a runtime of $O(m^2n)$.

Note that in the above argument, we do not use the property about marginal benefits that was used in the first part of the problem. The property that the array A_j is nondecreasing is implicitly used by the fact that we try to get as much benefit as we can by using all of our programmers.

2. (8 points) Stacking stones

Consider a set of n rectangular paving stones where the ith stone is l_i units long and w_i units wide, $l_i \ge w_i \ge 1$. Assume that paving stone i can be stacked on top of paving stone j iff $l_i \le l_j$ and $w_i \le w_j$. Give an efficient dynamic programming algorithm for computing the maximum number of paving stones that can be stacked together. Briefly justify its correctness and analyze the asymptotic running time of your algorithm.

Answer:

1) First sort these n rectangle paving stones by their areas, which we denote as a_i, i.e. l_i times w_i, by decreasing order. Thus we have a_{i-1} is no less than a_i for every i.

2) Then we define $H(i)$ is the maximum number of stacked stones, where stone i is on the top. We have the following equation of dynamic programming:

$$H(j) = \max_{i<j, l_i>l_j, w_i>w_j} \{H(i)\} + 1$$

The above equation means that, when we consider the maximum number of stacked stones where j is on the top, we will try every i less than j, if stone i is compatible with j, where $l_i > l_j, w_i > w_j$, we have a possible solution with j on top of i. Thus we select the maximum number of such possible solutions.

3) Finally $\max\{H(j)\}$ is our result, since we don't know which stone is on top in the final solution. We have to check every $H(j)$ to find a maximum value.

The complexity for step 1 is $O(n \lg n)$; and to calculate each $H(j)$, our complexity is $O(n)$, the total complexity of step 2 should be $O(n^2)$. The complexity of step 3 is $O(n)$.

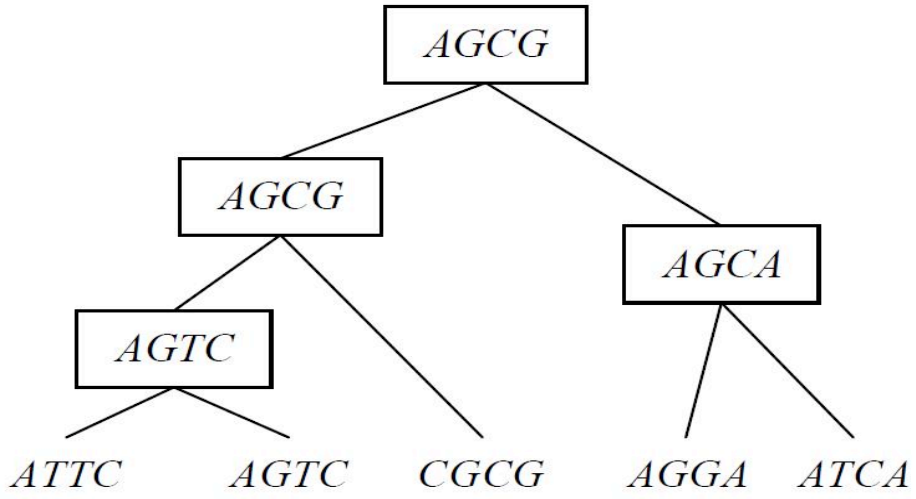
Thus the total complexity of this algorithm is $O(n^2)$.

3. (2 + 8 = 10 points) Evolutionary trees

Problem 6.30 of text.

Answer:

(a) See the graph below:



(b) According to the hint, we define a state $S(T, c)$, in which $c \in \{A, C, G, T\}$, representing the minimum of score of subtree T, and c is the root of T.

Then we have the following equations:

$$\begin{aligned}
\delta(i,j) &= 1(\text{if } i = j), \delta(i,j) = 0(\text{if } i \neq j) \\
L &= \text{left-subtree}(T) \\
R &= \text{right-subtree}(T) \\
S(T,c) &= \min_{a \in \sum, b \in \sum} \{(S(L,a) + S(R,b) + \delta(c,a) + \delta(c,b))\}
\end{aligned}$$

For the base case, we set $S(T, c)$ to be 0, for any character c, if T is a leaf. Since every inner node has two children, so we have the number of inner nodes is equal to the number of leaf nodes minus 1.

Thus we have the number of states $s = O(n)$. Moreover the complexity of transition between two states is $O(1)$. The total complexity will be $O(nk)$.

4. (8 points) Finding an optimal seating arrangement

Several families go out to dinner together. To increase their social interaction, they would like to sit at tables so that no two members of the same family are at the same table. Show how to formulate the problem of finding a seating arrangement that meets this objective as a maximum flow problem.

Assume that the dinner contingent has n families and the ith family has a_i members. Also assume that m tables are available and the jth table has a seating capacity of b_j.

Answer:

The capacitated graph $G(V, E)$ is defined as:

$$\begin{aligned} V &= \{s, t\} \bigcup \{u_i : 1 \leq i \leq n\} \bigcup \{v_j : 1 \leq j \leq m\} \\ E &= \{(s, u_i) : 1 \leq i \leq m\} \bigcup \{(u_i, v_i) : 1 \leq i \leq n, 1 \leq j \leq m\} \bigcup \{(v_j, t) : 1 \leq j \leq m\} \end{aligned}$$

We also have the following equations:

$$\begin{aligned} cap(s, u_i) &= a_i \\ cap(u_i, v_j) &= 1 \\ cap(v_j, t) &= b_j \end{aligned}$$

After we represent the problem as the above, an integral solution to this max flow problem gives an assignment of families to tables. A flow from u_i to v_j is either zero or one, with one meaning that a member of family i sits in table j. Thus the maximum flow maximizes the number of people that can be seated under the stated constraint.