CS577 Assignment 3

Liang Wang, Yan Zhai, Lei Kang

April 2, 2013

GREEDY ALGORITHMS

PROBLEM 1

Given a set of n jobs with a processing time t_i and a weight w_i for each job. We want to minimize the weighted sum of the completions times, $\sum_{i=1}^{n} w_i C_i$. We do the followings,

we get the value of weight over time by $\frac{w_i}{t_i}$, where $1 \le i \le n$. To select one job each time, we select job i with the largest $\frac{w_i}{t_i}$, which means for all i and j ($i < j$) in the job list, we have $\frac{w_i}{t_i} \ge \frac{w_j}{t_j}$, and recursively do this until all the jobs are done. This simple algorithm can lead to an optimized solution to this problem.

Proof. We prove the correctness of our algorithm by exchange argument. To do this, let's suppose that there is an optimal solution O that differs from our solution S. In other words, S consists of the weight to time fraction $\frac{w_i}{t_i}$ sorted in non-increasing order. So this optimal solution O must contain an inversion-that is, there must exist two neighbouring jobs i and j such that $\frac{w_i}{t_i} < \frac{w_j}{t_j}$.

We claim that by exchanging these two purchases, we can strictly improve our optimal solution, which contradicts the assumption that O was optimal.

Let us assume the previous time cost until job i is t_x and the weighted sum is N_x. So, the weighted sum of the optimal solution is $N_O = N_x + (t_i + t_x) * w_i + (t_i + t_j + t_x) * w_j$, and that of the exchanged algorithm is $N_S = N_x + (t_j + t_x) * w_j + (t_i + t_j + t_x) * w_i$.

To compare if the exchanged version is better or not, we have $N_S - N_O = t_j * w_i - t_i * w_j < 0$, which shows that the exchanged version has a smaller weighted sum and so is a better solution.

This concludes the proof of correctness. The running time of the algorithm is $O(nlogn)$, since the sorting takes that much time and the rest (outputting) is linear. So the overall running time is $O(nlogn)$

$\square$

PROBLEM 2

Given a connected graph $G = (V, E)$. Each edge e has a time varying edge cost given by function $f_e(t) = a_e * t^2 + b_e * t + c_e$, where $a_e > 0$. We want to get the minimum spanning tree at time t, where $0 \le t \le \infty$. Let m and n represents the number of edges and the number of nodes, respectively.

Suppose we get a spanning tree with $n-1$ edges, then the sum of the weights is $\sum_{i=1}^{n-1} (a_i t^2 + b_i t + c_i)$ The minimum value of this sum when $t = -2 * \frac{\sum_{i=1}^{n-1} b}{\sum_{i=1}^{n-1} a}$ if $\sum_{i=1}^{n-1} b < 0$ or $t = 0$ if $\sum_{i=1}^{n-1} b \ge 0$.

Now we discuss how to get the $n-1$ edges which has the minimum cost among all the edges. For a given time point, e.g., t_0, we can easily get the top $n-1$ edges by applying Prim's Algorithm. The real question is when any one of the edges among the top $n-1$ edges will be exchanged with any other edges during the change of time t. It only happens when the weight of one edge in the top $n-1$ edges is larger than that of any one edge among the rest. In other words, there is an intersection on these two curves of changing weight. There are totally $\frac{m(m-1)}{2}$ intersections, for each interval among these cross points, we calculate the minimum spanning tree correspondingly, and find the overall minimum weight sum among $\frac{m(m-1)}{2}$ intervals.

PROBLEM 3

Given $\rho \in [0,1]$, we want to find a set $I \subseteq \{1,...,n\}$ of size $n-k$ such that $\frac{\sum_{i \in I} s_i}{\sum_{i \in I} m_i} \geq \rho$, or we want to ensure the selected set after dropping is fulfil $\sum_{i \in I}(s_i - \rho * m_i) \geq 0$.

For each assignment score, with the changes of ρ values, we can get a curve by $y = s_i - \rho * m_i$. There will be $O(n^2)$ number of cross points, and each point represents a different ρ value. We get all the values into a vector P. We sort the vector in $O(n^2 log n)$ time. For each value in vector P (or we can do binary search in vector P in $O(log n)$), we sort all the assignment scores in ascending order according to the value of $s_i - \rho * m_i$, if the top $n-k$ values is not smaller than 0, then we get the final $n-k$ assignment scores.

Proof. The essential thing here is that the vector P contains all possible ρ values we need. Note that we only need the top $n-k$ values, the top $n-k$ assignments will be changed only when one curve represented by $y = s_i - \rho * m_i$ in the selected assignment result set goes below another curve that has not been selected yet. Therefore, the vector P contains all the intervals that the top $n-k$ values would be possibly swapped out by any of the rest k values.

For a given ρ, we can find if it meets our requirements by $O(n log n)$. There are n^2 possible ρ values, and we do binary search. it is $n(log^2 n)$. To find all possible ρ, we find all intersection points by $y = s - m * \rho$, so there are $O(n^2)$ points, we sort it in $O(n^2 log n)$. So, the time complexity is $O(n^2 log n)$.

$\square$

DYNAMIC PROGRAMMING

PROBLEM 1

Let r_i denotes the number of leftover trucks in month i, where $0 \leq r_i \leq S$ and $0 \leq i \leq n$, and $OPT(i, r_i)$ denotes the value of the optimal solution on month i when there are r_i leftover trucks. So $OPT(0, r_i) = 0$ a the start, and the problem we want to address is $OPT(n, 0)$.

In the first month, we need to order trucks anyway, the number of ordered trucks can be calculated by $d_i + r_i$, and the cost is $K + C * r_i$.

To get the solution, the sub problem we want to solve is $OPT(i, r_i) = OPT(i-1, r_{i-1}) + x * K + r_i * C$, where $x = 0$ if $r_i + d_i - r_{i-1} = 0$ and $x = 1$ if $r_i + d_i - r_{i-1} > 0$. By iteratively solving the sub problems over each month, we can get the smallest $OPT(n, r_n)$ as the optimal solution. We need to track all possible leftover trucks each month, so the time complexity is $O(nS)$.

One optimization we can do on this problem is reduce the number of leftover trucks need to check each month. A simple observation is if we need to re-order next month, when $r_i < d_{i+1}$, then there is no need to store the trucks.

To get an solution that is independent of S, we need to avoid checking all the possibilities of r_i in each month. To this end, we will not place an order as long as the storage cost is less than the ordering cost, $r_i * C \leq K$, and the number of leftover trucks can meet the requirements of the following month, $r_i \geq d_{i+1}$. Otherwise, we will place an order to meet the requirement of this month. If $r_i + d_i > S$, $OPT(i, r_i) = OPT(i-1, 0) + K$, and if else, $OPT(i, r_i) = min(OPT(i-1, r_i + d_i) + C * (r_i + d_i), OPT(i-1, 0) + K)$. In this case, the time complexity is $O(n)$.

PROBLEM 2

As the problem implied, Gerrymandering only happens when the total number of voters of one party is larger than the other. without losing of generality, let's assume party A has larger number of voters. Let $m_{i,A}$ and $m_{i,B}$ denote the number of voters in the i precincts that will vote for party A and B, respectively. So we have, each precincts have the same number of total voters and each voter supports one party only, $m_{i,B} + m_{i,A} = m$, and $\sum_{i=1}^{n} m_{i,A} > \frac{m*n}{2}$.

To keep track of all possible allocations when we start to check precinct i, we use a big binary matrix $M(i, j, x, y)$ with size $\frac{mn}{2} * \frac{mn}{2}$, where the matrix item is equal 1 when there are at least x and y voters have been allocated into set 0 and 1, respectively, and j precincts have been allocated to set 0, and it is equal to 0 otherwise. When we check the next precinct, the matrix item $M(i+1, j+1, x+m_{i+1}, y) = 1$ and $M(i+1, j, x, y+m_{i+1}) = 1$. Our interest is $M(n, \frac{n}{2}, x, y) = 1$ while $x > \frac{m*n}{4}$ and $y > \frac{m*n}{4}$. The time complexity to maintain and scan over the big binary matrix is $O(m^2 n^2)$.

PROBLEM 3

(a) Suppose we have an optimal solution J in which not all jobs are scheduled in increasing order of their deadline. By definition, all jobs in J can meet their deadline. Considering there are two neighbouring jobs in J with two deadlines d_i and d_{i+1}, we have $d_i > d_{i+1}$ and $s + t_i < d_i$ and $s + t_i + t_{i+1} < d_{i+1}$, where s is the start time of the first job of these two. By swapping these two jobs, job $i+1$ can meet its deadline as $s + t_{i+1} < d_{i+1}$, and job i can also meet its deadline as $s + t_i + t_{i+1} < d_{i+1} < d_i$. So, for any optimal solution, we can re-order it into an optimal solution in which all jobs are scheduled in increasing order of their deadline.

(b) Firstly, we sort all the jobs in the order of increasing deadlines d_i. For a sub problem, we want to know how long it lefts if we performed j jobs at deadline i, and $j \leq i$. We maintain a $n * n$ matrix for this, where each item should be not smaller than 0, $M(i, j) \geq 0$. The philosophy we use to solve this problem, is for a given number of tasks that can be finished so far, we want the left time is as large as possible. So, for each new job we traverse, we either ignore it or add it, the objective is to find the maximum left time for a given number of jobs j, and we have $M(i+1, j) = max(d_{i+1} - d_i + M(i, j), d_{i+1} - d_i + M(i, j-1) - t_{i+1})$. The solution we will get at the item $M(n, m) \geq 0$, where m has the largest value. The solution takes $O(n^2)$, as we need to check all possible j value for each deadline.