CSE 548 Assignment 2

Amit Dharmadhikari (SBU ID: 112044244)

October 1, 2018

1 Question 1

Greedy algorithms have been taught in class. This problem can be solved with a greedy algorithm.

We see that the supercomputer operates in a serial fashion while the PCs operate in a parallel fashion. This means that the total working time of the supercomputer will be the same irrespective of the ordering of jobs in the schedule. Let $P = p_0 + p_1 + p_2 + ... + p_n$. P is the time at which the n'th job in the schedule will be handed over from the supercomputer to the PC, P is never going to change. Thus, the completion time of the schedule depends on the finishing time of the jobs, especially the last job in the schedule. Hence, we should order the jobs in the decreasing order of their finishing times.

Thus the algorithm is as follows:

Sort $J_1, J_2, ...J_n$ in the decreasing order of f_i

We can prove the correctness of this algorithm using an exchange argument. An inversion in this case would be defined as a pair of jobs in a schedule whose order does not comply with the decreasing order of their finishing times. Consider the schedule S_1 in which the jobs are not sorted in the decreasing order of f_i. S_1 is bound to have two jobs J_a and J_b such that $f_a < f_b$ and J_a is scheduled before J_b. Now consider another schedule S_2 which is obtained by swapping the places of J_a and J_b in S_1. As we discussed earlier, the working time of the supercomputer never changes. In S_2, the job J_a will be handed off from the supercomputer to the PC at the same time as that of J_b in S_1. However, since $f_a < f_b$, we see that S_2 does not have a greater completion time than S_1. Using a sequence of such swaps, any schedule can be transformed into our schedule given by the above algorithm. Thus, the schedule returned by the above algorithm is optimal.

2 Question 2

Each customer has a time t_i as well as a weight w_i associated with him/her. Though the importance of the customer to the business is apparent from the weight, it might not always be profitable for the business to serve a customer with a larger weight. This happens when the time t_i for a customer is too high. For example, let's consider two customers C_a and C_b. $w_a = 1, W_b = 2, t_a = 1, t_b = 10$. If customer C_j is served before customer C_i, then the weighted sum of the completion times, i.e. $w_a C_a + w_b C_b$ is $2 * 10 + 1 * 11 = 31$, whereas if it is done the other way round, it will evaluate to $1 * 1 + 2 * 11 = 23$. Thus, we see that serving a customer with a lower weight first was profitable for the business. It can be seen that the weight per unit time of the customer is an appropriate measure of the true importance of a customer. Thus, we should sort the jobs in the decreasing order of w_i/t_i.

The algorithm is as follows:

Sort the jobs in the decreasing order of w_i/t_i.

We can prove the correctness of this algorithm using an exchange argument. Let's consider a schedule which does not conform to our algorithm. Then there are bound to be two customers m and n such that $w_m/t_m < w_n/t_n$ and $C_m < C_n$. Let's suppose that the completion time before m and n is C. Then, the completion time of m is $C + t_m$ and that of n is $C + t_m + t_n$. Their total contribution to the weighted sum of the completion times is $w_m(C + t_m) + w_n(C + t_m + t_n)$. Now, if we swap the positions of these two tasks, then the contribution becomes $w_n(C + t_n) + w_m(C + t_n + t_m)$. If we subtract the latter from the former, we get $w_n t_m - w_m t_n$. We know that $w_m/t_m < w_n/t_n$, i.e. $w_m t_n < w_n t_m$. Thus, we see that the difference is positive. This means that the swapping of m and n decreases the weighted sum of the completion times. Any schedule can be converted to our schedule (the one returned by our algorithm) through a sequence of such swaps. Thus, our algorithm is optimal.

3 Question 3

The interval scheduling problem has been taught in class. This problem is a variation of that. With a bit of modification, we can convert this problem to an interval scheduling problem.

Each job must run daily from its start time to end time. A job which starts late at night and ends early in the morning may conflict with another job which starts early in the morning. So basically, these jobs operate in a circular fashion which is what makes this problem different from an Interval Scheduling problem, in which they operate in a linear fashion.

In order to convert this to an interval scheduling problem, we will choose a job j as a reference point and include it in our schedule. Then, we will eliminate all the other jobs which conflict with this job. Basically, we cut the time-line at the point j, thus effectively converting it to linear, like an interval scheduling problem. Now we solve this according to the optimal way taught in class, i.e. by choosing the job with the lowest finishing time in each iteration. If the jobs are sorted according to their finishing times, it will take $O(n)$ time. Otherwise, we'll need to sort the jobs, which can be done in $O(nlog(n))$ time. Now further, we will have to repeat this procedure for all the jobs by cutting the time-line at each job and solving the interval scheduling problem instance on the remaining jobs. Thus, our algorithm will take $O(n^2)$ time (assuming jobs are sorted, else $O(n^2log(n))$. We will return the solution of that instance which accepts the maximum number of jobs.

Thus, the algorithm is as follows:

Let $J_1, J_2, ..., J_n$ be the n jobs.

For i = 1 to n:
Choose J_i as the reference point.
Eliminate all the jobs that conflict with J_i
Solve the rest of the problem as an Interval Scheduling problem instance (By choosing the job with the lowest finishing time in each iteration)
Store the details of this schedule.

Return the schedule which accepts the maximum number of jobs.

4 Question 4

4.1 (a)

Let's suppose that the first conjecture is not true. Let's assume that the MST T is not a minimum-altitude connected sub-graph. Then for two nodes u and v, there will be a path of minimum altitude that is not contained in T. Let P be the path between u and v that passes through the MST T and $P^{'}$ be the path with minimum altitude that does not pass through T. Since P has a higher altitude than $P^{'}$, there exists an edge $(u^{'}, v^{'})$ in P that has the highest altitude out of all the edges in P and $P^{'}$ combined. We can find a path between $u^{'}$ and $v^{'}$ excluding the edge $(u^{'}, v^{'})$. We can do this by going from $u^{'}$ to u through P, u to v through $P^{'}$ and $v^{'}$ to v through P. But addition of these edges will form a cycle, which will violate the property that an MST cannot have a cycle. Thus, the minimum spanning tree of G is a minimum altitude connected sub-graph.

4.2 (b)

Let's assume that there exists a minimum altitude sub-graph H that does not contain all the edges of the MST T. In particular, let's assume that there exists an edge (u, v) such that (u, v) belongs to T but not to H. Now let's delete the edge (u, v) from T. This will partition T into two components, let's call them C_1 and C_2. H must have some way to go from C_1 to $C2$. However, it cannot use (u, v), and we know that the edge (u, v) is the edge with minimum altitude that connects the components C_1 and C_2. Thus, the path which H will use to connect C_1 and C_2 will not have the minimum weight, which contradicts our assumption that H is a minimum altitude sub-graph. Thus, we have proved that H is a minimum-altitude sub-graph if and only if it contains all the edges of T.

5 Question 5

5.1 (a)

The Kruskal's MST algorithm has been covered in class. We can solve this problem using this algorithm. Every time we add an edge of length l connecting two components C_i and C_j in the Kruskal's algorithm, we add a node to the tree T which is a parent to the sub-trees which represent the components C_i and C_j respectively. We assign this node a height of l, i.e. the length of the edge. For any two nodes u and v, $\tau(u, v)$ represents the length of the edge using which the components containing u and v were joined. $\tau(u, v) \leq d(u, v)$ must hold true because Kruskal's algorithm selects edges in the ascending order of their lengths and if the edge (u, v) is considered then u and v will belong to the same component from that point onward. Thus, τ is consistent with d.

5.2 (b)

Let's suppose there is another hierarchical metric $\tau^{'}$ such that $\tau^{'}(u, v) > \tau(u, v)$. Let $T^{'}$ be the tree associated with $\tau^{'}$, let $h^{'}_{v}$ be the height of a vertex v in tree $T^{'}$. $\tau^{'}(u, v) = h^{'}_{commonancestor} > \tau(u, v)$ Assuming that $\tau^{'}$ is consistent with d, we get, $\tau^{'}(u, v) \leq d(u, v)$. Since we assumed that $\tau^{'}(u, v) > \tau(u, v)$, we get $\tau(u, v) < \tau^{'}(u, v) \leq d(u, v)$, i.e. $\tau(u, v) < d(u, v)$. However, when the Kruskal's MST algorithm merges the components containing u and v, the height of the common ancestor can't be more than $d(u, v)$, as we reasoned before, but it can be equal to $d(u, v)$. Thus, we have a contradiction. Hence, we have proved that if $\tau^{'}$ is any other hierarchical metric consistent with d, then $\tau^{'}(p_i, p_j) \leq \tau(p_i, p_j)$.

6 Question 6

This problem can be solved using Kruskal's algorithm, which has been taught in class.

The function $f_e(t)$ is a polynomial of degree 2. Hence, the graph of $f_e(t)$ is a parabola. The Kruskal's algorithm selects the edge having the least weight in each iteration. A change in the absolute values of the edges will not make any difference to the algorithm, as long as the sorted order remains the same. Now, the sorted order for the edges will change only when two parabolas intersect each other. Any two parabolas can intersect at most twice. The total number of intersections will be $2\binom{m}{2}$, where m is the number of edges. $2\binom{m}{2}$ is upper bounded by m^2. Now, we will divide the time axis into these m^2 intervals. For each interval, we will find the minimum spanning tree using Kruskal's algorithm. Then, we will find the cost of this minimum spanning tree at all points in that interval, and take the one with minimum cost. After performing this for all intervals, we will find the point at which the cost of the minimum spanning tree is lowest over all intervals, using the minimum points which we just found out for each interval. This point at which the cost of the minimum spanning tree is lowest over all intervals is the desired answer.

7 Question 7

7.1 (a)

It has been mentioned that we add an edge (u, v) to the graph H if $3l_e < d_{uv}$, where l_e is the length of edge (u, v) and d_{uv} is the length of the path from u to v in H. Thus, we see that for any two points, the edge between them is not added to the graph H only if the graph H already has a path from u to v, the length of which is not more than three times that of the shortest path between u and v. Now, let's consider the case of two vertices a and b which do not have an edge in between. Let $e_1, e_2, ..., e_l$ be the edges in the shortest path from a to b in the original graph G. We know that for each edge e_i in this path, the graph H will either contain it or have a path that is not more than three times the length of that edge. We will apply this to all the edges $e_1, e_2, ..., e_l$ in the shortest path. Thus, we see that the length of this path in graph H will not be more than three times the length of this path in graph G. Hence, proved.

7.2 (b)

The graph H cannot have any short cycle (i.e. a cycle of length less than 4). This implies that the degree of each node in the graph H must be less than or equal to $\sqrt{n}$. In order to prove this, let's suppose that this is not the case. Let u be a node in the graph H. There can't be any edge joining the neighbors of node u, or that would form a cycle of length 3. Let I be the set of neighbors of node u and O be the set of nodes which are adjacent to some node in the set I. Each node in O will have one and only one neighbor in the set I. But since we assumed that the node u has a degree greater than $\sqrt{n}$ and each node in set I has greater than or equal to $\sqrt{n}$ neighbors apart from u, we come to

the conclusion that the number of nodes in set O is greater than n. This is a contradiction, which proves that our initial assumption was wrong. Thus, we have n nodes with a degree of less than or equal to $\sqrt{n}$ each. We can see that the total number of edges in the graph H is upper bounded by $n * \sqrt{n}$, i.e. $n^(3/2)$. Since $n^{3/2}$ is a function which grows asymptotically slower than n^2, $\lim_{n\to\infty} \frac{f(n)}{n^2} = 0$.