CS 218, Spring 2017 Homework 5
Abhishek Kumar SRIVASTAVA
Student ID: 861307778
May 10, 2017

I certify that this submission represents my own original work

---

Solution 1. Algorithm:

Given: list of n jobs $j_1, j_2, \ldots, j_n$ each with a time $t(j)$ to perform a job and a weight $w(j)$.

Order jobs to minimize:

$$\sum_{i=1}^{n} \left(w(j_{\sigma(i)}) \sum_{k=1}^{i} t(j_{\sigma(k)}) \right)$$

To get the efficient greedy algorithm choose the highest weight-time ratio first strategy. First, calculate the $\frac{w(i)}{t(i)}$ ratio for each job and sort them in decreasing order. Pick the first element with the highest weight-to-time ratio and it can be obtained by calling $\sigma(1)$ which should return the corresponding index of the job and keep picking next item in the order given by $\sigma(i)$ function where $i = 2, 3 \ldots, n$. This should give the minimum weighted sum of the time each job has to wait before being executed.

---

function MinWeightedTimeSum($weight\_array, time\_array$) ▷ index $i = i^{th}$ job
  weight_to_time_array = []
  for $i = 1$ to n **do** ▷ Calculating Weight to Time Ratio
    weight_to_time_array[i] = weight_array[i]/time_array[i]
  end for
  Sort(weight_to_time_array) ▷ Sorting in decreasing order
  for $i = 1$ to n **do** ▷ Picking Order
    Pick $w(\sigma(i))$ & $t(\sigma(i))$ ▷ $\sigma(i)$ will return corresponding index
    ▷ to element in weight-to-time ratio.
  end for
end function

Time Complexity:

1. Sorting the weight-to-time ratio array will take $O(nlogn)$.

2. Selecting one-element at a time will give the time complexity of $O(n)$ assuming that $\sigma(i)$ returns the index in $O(1)$ which can easily done by using appropriate data structure such as hash.

$$T(n) = O(nlogn) + O(n)$$
$$\therefore T(n) = O(nlogn)$$

Optimality: We need to prove two property to proves the optimality of the algorithm suggested.

1. Greedy Choice Property: In this property we have to prove that local optimal choice will lead to the globally optimal choice as well.

Let j_i the element with highest weight-to-time ratio. Assuming there is an element j_k when picked first gives the minimum weighted time sum, but since j_i will be picked sometime after the j_k and being the highest weight-to-time ratio it will add more weighted time to the whole sum than it is being reduced by picking j_k as the first element which contradicts the assumption that picking j_i leads to min weight time sum. Therefore picking greedily the element with highest weight-to-time ratio each time will lead to the global optimal solution.

2. Optimal Substructure Property: In this property we need to prove that problem can be divided in optimal subproblems.

Once the greedy choice of the first job is made, the problem then reduces to finding optimal order of job which minimize the weighted time sum of the jobs which are now not selected.

If S set of selected jobs is optimal to J set of all the jobs, then S' is optimal to $J' = J - S$.

Solution 2. Consider edges in order of decreasing weight (breaking ties arbitrarily). When considering an edge e, delete e from E unless doing so would disconnect the current graph. The above described algorithm is:

---

function REVERSEDELETE(G,w)
  A ← E ▷ A is get of All the edges
  O ← Sorted(A) ▷ O is set of ordered weights in decreasing order
  while O is not empty **do**
    e = O.top() ▷ picks edge with highest weight and removed from set O
    if removing e will not create disjoint sets **then**
      A ← A - e
    end if
  end while
end function

Time Complexity:
1. Sorting all the edges of set A will take $O(ElogE)$ time complexity.
2. Checking whether the graph is disjoint or not can be done by using BFS and DFS which have time complexity of $O(V + E)$.
3. We are checking whether the graph is disjoint or not assuming e is deleted which is $O(E)$.

Thus, Total time complexity will be,

$$T(n) = O(E \cdot logE) + O(E \cdot (V + E))$$

$$\boxed{\therefore T(n) = O(E \cdot (V + E))}$$

Optimality:
1. Greedy Choice: If M is MST(V), by removing the largest weight between any two vertices maintains the property of light edge crossing $i.e$ minimum weight between an edge crossing cut. If we remove a edge with less weight between two vertices while not creating disjoint set then

it does not give the minimum spanning tree. That's why removing weights which does not lead to disjoint graph is a good greedy choice.
2. Optimal Substructure: One the chosen edge is removed from a set then the subproblem becomes to find the optimal MST for the remaining edges.

Solution 3. Dijkstra algorithm for the directed graph the algorithm will be the same as for undirected graph, the only thing which will change is because of the condition $w : E \rightarrow \{0, 1, \ldots, W\}$ to handle this we could use binary heap as our priority queue which stores all the vertices with distance d at a single node.

Building such heap for all vertices (n) with maximum of $W + 1$ nodes which is all possible weights of the edges. It will takes $O(n \cdot logW)$ time to construct the binary heap.Initially the priority queue (binary heap) will have maximum of $W + 1$ nodes.

During edge relaxation, every time we add a node u to the cloud with of the weight value d, the priority queue (binary heap) will be left with maximum of $W + 1$ distinct values in the range $d, d + 1, ..., d + W$. As we know from the Dijkstra algorithm edge relaxation takes $O(deg(V) \cdot logW)$ time which is equivalent to $m \cdot logW)$ since sum of all the degree is equivalent to edges. Therefore the total time complexity of the Dijkstra's algorithm in this scenario will be

$$\boxed{T(n) = O((n + m) \cdot logW)}$$