1.5em 0pt

Submission for Problem Set 4

Name: Ajeya Kempegowda

Problem 1. (6 points) Total Weighted Finish Time

A scheduler needs to determine an order in which a set of n processes will be assigned to a processor. The i th process requires t_i units of time and has weight w_i.

For a given schedule, define the finish time of process i to be the time at which process i is completed by the processor. (Assume that the processor starts processing the tasks at time 0.)
For example, consider 4 processes with $t_1 = 5, t_2 = 2, t_3 = 7, t_4 = 4$. And weights $w_1 = 1, w_2 = 3, w_3 = 2, and w_4 = 2$. Consider the schedule 1, 3, 2, 4. The finish time for process 1 is 5, for process 2 is $t_1 + t_3 + t_2 = 5 + 7 + 2 = 14$, for process 3 is $t_1 + t_3 = 5 + 7 = 12$, and for process 4 is $t_1 + t_3 + t_2 + t_4 = 5 + 7 + 2 + 4 = 18$. The total weighted finish time of the schedule is then $5 \times 1 + 14 \times 3 + 12 \times 2 + 18 \times 2 = 5 + 42 + 24 + 36 = 107$.

Give a greedy algorithm to determine a schedule that minimizes the total weighted finish time. Prove the correctness of your algorithm, and analyze its worst-case running time.
Answer:

For simplicity consider two tasks - task 1 and task 3. The finish time for process 1 would be t_1 and $t_1 + t_3$ according to the above example. Now, if the process is pipe lined as $process1 \rightarrow process2$ then the weighted finish time would be $w_1 t_1 + w_3(t_1 + t_3)$. For vice-versa pipeline, the weighted time finish would be $w_3 t_3 + w_1(t_1 + t_3)$. Here, the minimized total weighted finish depends on the value of $w_3 t_1$ and $w_1 t_3$, after eliminating the common terms. task 1 would be selected ahead of task 3 if,

$$w_3 t_1 < w_1 t_3 \tag{1}$$

$$\frac{t_1}{w_1} < \frac{t_3}{w_3} \tag{2}$$

The time complexity of the algorithm is just the time needed for sorting, which is $\theta(nlogn)$, where n is the number of processes.

The proof of correctness is done by contradiction. If there's a better scheduler which can achieve better total weighted finish time such that for two subsequent tasks a and b such that $t_a/w_a < t_b/w_b$, i.e a is scheduled just before b. For the contradiction if we interchange the schedules, then the weighted finish time of before and after a, b of every task remains the same. Further the difference in the weighted finish times before and after the swap of a and b processes is $w_a t_b$ - $w_b t_a < 0$. So the total weighted finish time of the new schedule is less than that of the old schedule. Hence, a contradiction.

Algorithm 1 Scheduler

1: **procedure** SCHEDULER($t[], w[]$)
2: Compute t_i/w_i for ith processes
3: Merge sort the above values in increasing order ▷ O(ilogi)
4: return Sorted values(Schedule the tasks according to these values)
5: **end procedure**

Lets, make use of list data structure that can hold weights and time separately. The algorithm to attain the optimal scheduler would be as stated above.

Problem 2. (6 points) Subsequences

Comparing two sequences of characters arises in several applications including the life sciences and information retrieval. Suppose you are given two sequences of characters drawn from a common alphabet: S of length m and T of length n, each possibly containing a character more than once. We say that S is a *subsequence* of T if S can be obtained from T by deleting some characters from T. For instance, the sequence A, T, C, T, G is a subsequence of $G, \underline{A}, A, A, \underline{T}, \underline{C}, G, G, \underline{T}, T, \underline{G}$. (Positions indicating the subsequence property are underlined.)

Give an algorithm that takes as input two sequences S and T and determines whether S is a sub sequence of T. Prove the correctness of your algorithm and state its time complexity.

Answer:

The problem can be solved using recursion paradigm. If both the characters from the strings are the same, the character position of the first string is decremented by one position and the recursive call is made until one of the strings are completely exhausted. If the characters from the strings are dissimilar then the character positions are decremented by one on both the the strings and the recursion call is made until the above condition is satisfied.

Algorithm 2 Common sub-sequence

```
procedure CSS(S, T, m, n)
    if m == 0 or S == T then
        return True
    end if
    if n == 0 then
        return False
    end if
    if S[n] == T[m] then
        return CSS(S, T, m-1, n-1)
    end if
    return CSS(S, T, m, n-1)
end procedure
```

The time complexity of the above algorithm is linear (length of the T) i.e $O(n)$

The proof of correctness can be proved by induction:

- Base case: If s and t are characters instead of strings, then base cases would be s== t and s!=t. If s==t then the first recursion call would be called which returns true as m==0 condition yields true. On the other hand when s!=t the second recursion call would be invoked which finally yields False.
- Inductive hypothesis: For the inductive step, assume the claim is true for strings of k-1 lengths. If at this point S=T, then the algorithm terminates by returning True. If S $\neq$ T, the algorithm goes for the next iteration. The recursive algorithm calls itself on the sub-problems. Now if the hypothesis is true for k-1 lengths, then according to the induction hypothesis we need to prove for the +1th state. For this let's pad a character to S and T so that the length is now (k-1)+1. Now, to be S to be a sub sequence of T, the padded character has to be the same. Which yields a true value. If the padded character is not the same then the result would be false. So, either cases, the Algorithm terminates and is optimal. This completes the induction.

Problem 3. (7 points) Optimal tank capacity

You are given a transportation network as an undirected graph $G = (V, E)$, where V is a set of cities and E is a set of roads. Each road $e \in E$ connects two of the cities, and you know its length ℓ_e in kms. You want to get from city s to city t. There's one problem: your car can only hold enough gas to cover L kms. There are gas stations in each city, but not between cities. Therefore, you can only take a route if every one of its edges has length $l_e \leq L$.

You are now planning to buy a new car, and you want to know the minimum fuel tank capacity that is needed to travel from s to t. Give an efficient algorithm to determine this. Prove the correctness and optimality of the algorithm, and determine its worst-case time complexity.

Answer:

The optimal tank capacity can be found out by applying the Reverse-delete algorithm to find out the minimum spanning tree. The gist of the algorithm can be summarized as follows:

1. Start with graph G, which contains a list of edges E.
2. Go through E in decreasing order of edge weights.
3. Check if deleting current edge will further disconnect graph.
4. If G is not further disconnected, delete the edge.

Algorithm 3 Tank capacity

```
1: procedure TANK CAPACITY(edges E [], source s, sink t)
2:     Initialize tank capacity = 0
3:     if E is empty then
4:         return tank capacity
5:     else if size(E) == 1 then
6:         return weight of edge E
7:     else
8:         Merge sort E in decreasing order                          ▷ O(ElogE)
9:         Initialize index = 0
10:        while index < size(E) do                                  ▷ E
11:            current largest ← E[index]
12:            delete E[i]
13:            Run DFS to check the connectivity of the network from s to t   ▷ O(|V'| + |E|)
14:            if graph is not connected then
15:                tank capacity = current largest
16:            end if
17:        end while
18:        return tank capacity
19:    end if
20: end procedure
```

Proof of correctness is done by contradiction: Let the critical path length of G be the length of the longest path in G. Clearly, the car would need to have the tank capacity equivalent to the length of the critical path. We now claim that our algorithm outputs the length of the critical path. i.e only unsafe edges are removed. If there exists another critical path, greater than or less than the existing value, we've two cases. If the critical path is greater than the existing one, then this invalidates the reverse delete algorithm as it deletes the largest values. Hence, this can't be true. Now, there can't be a critical lesser than the existing one as we're sorting the values of edges in the initial step, and the algorithm wouldn't bypass any steps to arrive at this lower critical value. Hence we have a contradiction on both cases.

The proof of optimality follows proof of correctness and the Reverse delete greedy algorithm are optimal in working. We can formally establish that the above algorithm produces an optimal result. The time complexity of the algorithm is $O(Elog(E))) + O(|V| + |E|)$

Problem 4. (6 points) Minimum-cost Spanning Tree

Let $G = (V, E)$ be an (undirected) graph with costs $c_e \geq 0$ on the edges $e \in E$. Assume you are given a minimum-cost spanning tree T in G. Now assume that a new edge is added to G, connecting two nodes $v, w \in V$ with cost c.

Give an efficient algorithm to test if T remains the minimum-cost spanning tree with the new edge added to G (but not to the tree T). Analyze the worst-case time complexity of your algorithm. Please note any assumptions you make about what data structure is used to represent the tree $T$and the graph G.

Answer:

Given a graph G=(V,E) and it's minimal spanning tree T, the given problem can be solved using the following key point:

- If the newly added edge is the most expensive edge in T then T remains the MST
- The Graph G, is assumed to be stored in an Adjacency list(or a matrix would also work)

Algorithm 4 is MST

```
1: procedure isMST(T, v, w)
2:     Initialize source ← v, sink ← w in T
3:     Traverse paths v to w reached, storing the path traversed and marking the nodes as visited.
4:     Once w is reached, we obtain the shortest path in T from v to w
5:     if e1 is the most expensive edge then
6:         return True
7:     else
8:         return False
9:     end if
10: end procedure
```

Since the traversal of paths follows the principles of BFS algorithm the time complexity of the algorithm is O($|V| + |E|$). But, in a tree $|E|$ is one less than $|V|$ so we can boil down the equation to O($2|V|$). Finally, the time complexity would be O($|V|$)