Poonam Gupta

COT 5407 – Introduction to Algorithms Homework Assignment #3

Problem 1

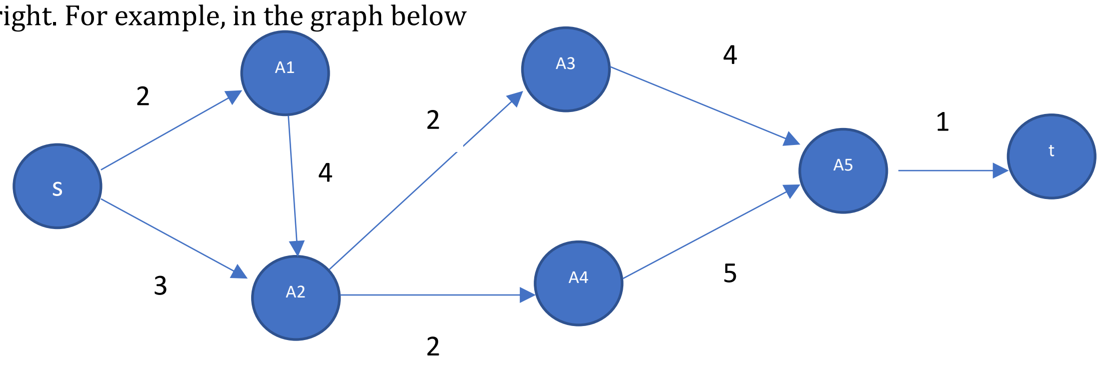
Given a directed acyclic graph G = (V, E) with node set V and weighted edge set E, where each edge $e \in E$ has a value $w(e)$ as its weight. Use dynamic programming to find a longest weighted simple path from node s to node t. You can assume the graph is represented using adjacent lists, i.e. for node u, $G.A$dj(u) contains all the nodes connected from u.

(1) Formulate the recursive relation of the optimal solution,

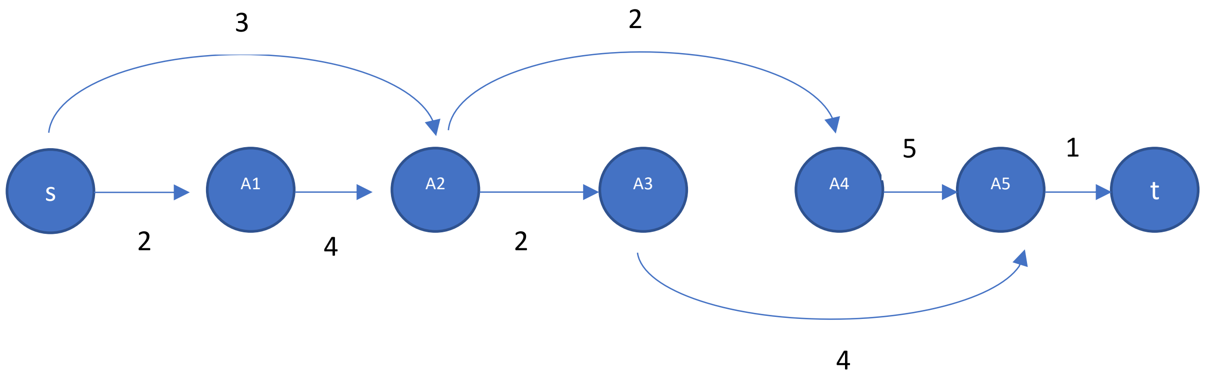
For us to find the longest weighted simple path from node s to node t, we need to identify if we can break down this problem into an optimal substructure (sub problems) in order to use dynamic programming.

In the given problem, we have a directed acyclic graph G = (V, E) with node set V and weighted edge set E and $w(e)$ in edge $e \in E$ as its weight. We also have the graph represented using adjacent lists $G.A$dj(u), given a node u.

We are assuming, there are no negative weights. DAG can be topologically sorted or linearized. This property allows us to traverse the nodes from node s(source) to node t(destination), from left to right. For example, in the graph below



Topographically sorted/ linearized representation



In the example above, if we have to calculate the longest weighted simple path from node(s) to node (A5), then we have to compute the longest weighted path from node(s) to node(A4) and node(s) to node(A3). Now that we know the path to these to nodes, we can calculate the longest weighted path to node(A5) by taking the larger of the longest weighted paths to these nodes after applying the respective weights for each edge to the predecessor.

dist(A5) = max{dist(A4) + w($e_{A5,A4}$), dist(A3) + w($e_{A5,A3}$)}

We can see that the subproblems here are the two previous nodes (dist(A3) and dist(A4)).

In this way, we can write the recurrence for the longest weighted path from node(s) to any node(v) in G, by first computing the longest path to all of v's previous nodes in the set.

dist(v) = $\max_{(u,v) \in E}$ {dist(u) + w($e_{u,v}$)}

(2) Design a bottom-up algorithm to calculate the longest weighted simple path,

For a weighted graph G = (V, E) with node set V and weighted edge set E, where each edge $e \in E$ has a value $w(e)$ as its weight, the algorithm is:

```
Longest-path(G)
Input: Weighted DAG $G$ = (V, $E$)
Output: Largest path cost in G
        Topologically sort G
        for each vertex v ∈ V in linearized order
                do dist(v) = max(u,v) ∈$E$ {dist(u) + w(u, v)}
        return maxv∈V {dist(v)}
```

(3) Analyze the complexity of your algorithm.

Time complexity of topological sorting is O(V+E). After finding topological order, the algorithm process all vertices and for every vertex, it runs a loop for all adjacent vertices. Total adjacent vertices in a graph is O(E). So, the inner loop runs O(V+E) times. Therefore, overall time complexity of this algorithm is O(V+E).

Problem 2
Given a set S= {$t1$, …, tn} of tasks, where ti requires pi units of processing time to finish once it has started. There is only one computer to run these tasks one at a time. Let ci be the completion time of task ti . The goal is to minimize the average completion time $1/n\sum_{i=1}^{n}$ ci. For example, two tasks $t1$ and $t2$ have processing times $p1$= 3 and $p2$ = 5 respectively; running $t1$ first results in the average completion time (3 + 8)/2 = 5.5, while running $t2$ first results in the average completion time (5 + 8)/2 = 6.5.

(1) Describe the greedy choice property and the optimal substructure in this problem,
As we can see in the above example, if the task with lower processing time is processed first, the average completion time is less than other possibilities. In order to ascertain if this greedy solution is

optimal, we consider that if we run the first task in an optimal solution, then we get an optimal solution by running the remaining tasks in a way that minimizes the average completion time. As we will see below, this can be achieved by running the tasks in ascending order of their processing times.

(2) Design a greedy algorithm to solve this problem,

Based on the property discussed above, we can establish that sorting the tasks based on its processing times and executing them in this order will result in the least average completion time. $c_{avg} = \frac{c_1 + c_2 .... + c_n}{n}$ or $[p_1 + [p_1 + p_2] + [p_1 + p_2 + p_3] .... + [p_1 + p_2 + p_3 + ... + p_n]] / n$. We can see here that p1 is used the most, then p2 and so on. Thus, ordering the tasks based on their processing times in ascending order and running them will minimize the average completion time.

(3) Analyze the complexity of your algorithm.

Let O be an optimal solution. Let t_1 be the task which has the smallest processing time and let t_2 be the first task run in O. Let G be the solution obtained by switching the order in which we run t_1 and t_2 in O. This amounts reducing the completion times of t_1 and the completion times of all tasks
in G between t_1 and t_2 by the difference in processing times of $t_1$and t_2. Since all other completion times remain the same, the average completion time of G is less than or equal to the average completion time of O, proving that the greedy solution gives an optimal solution. Since we must sort the elements first, the runtime is O(n lg n).

Problem 3

Binary search of a sorted array takes logarithmic search time, but the time to insert a new element is linear in the size of the array. The time for insertion can be improved by keeping several sorted arrays. Specifically, suppose that we wish to support SEARCH and INSERT on a set of n elements. Let $k = \lceil \log (n+1) \rceil$, and let the binary representation of n be $<n_{k-1}, n_{k-2}, ..., n_0>$. We have k sorted arrays $A_0, A_1, ..., A\ A_{k-1}$, where the length of array A_i is 2^i. Each array is either full or empty, depending on whether n_i=1 or 0, respectively. The total number of elements held in all k arrays is therefore $\sum_{i=0}^{k-1} n_i 2^i$ =n. For example, when n=10, its binary representation is <1,0,1,0>, A_0 and A_2 are empty, and A_1 and A_3 are full with 2 and 8 elements respectively. Although each individual array is sorted, elements in different arrays bear no particular relationship to each other.

(1) Describe how to perform the SEARCH operation for this data structure. Analyze its worst-case running time.

Answer:

In the above example no of elements (n) = 10

No of sorted arrays(k) = $\lceil \log (10+1) \rceil = 4$

Binary representation of n be $< A_3, A_2, A_1, A_0>$ or <1,0,1,0>

It means Array A_3 and array A_1 have 8 and 2 elements respectively However there is no relation between the elements in each array. This requires all arrays must be searched before the conclusion can be drawn.

e.g. in the scenario below, we have to perform a binary search on all arrays.

A3 $[e_7,e_6,e_5,e_4,e_3,e_2,e_1,e_0]$

A2[]

A1$[e_1,e_0]$

A0[]

Thus, the worst-case complexity is

$$\sum_{i=0}^{k} \lg(2^{i}) = \sum_{i=0}^{k} i = \frac{k(k+1)}{2} = O(k^2) = O(lg^2 n)$$

(2) Describe how to perform the INSERT operation. Analyze its worst-case and amortized running times.

To perform the insert operation, the new element is added to list A_0 and the following arrays are all updated in response. The worst-case scenario would have the algorithm merge all arrays A_0, A_1, …, A A_{m-1} into a new array A_m. As previously established, merging two sorted arrays is a linear over the total length of arrays. Therefore, the time this operation would take is $O(2^m)$. In this case, this takes time O(n) since m could be equal to k which means $O(2^m) = O(2^k) = O(n)$.

Now, when we consider using the accounting method for analysis of amortization cost, we see that cost of lg n for each insertion. This implies that each item carries lg n credit to pay for its later merges. Since an individual item can only be merged into a larger array and there are only lg n arrays, the credit pays for all future costs the item might incur. Thus, the amortized cost is O(logn).