CS403: Algorithm Design and Analysis

Feb-Jun 2017

Indian Institute of Technology Mandi

Assignment 2

Priyansh Saxena

B14118

Computer Science and Engineering

CS 403 – Algorithm Design and Analysis

Assignment 2

Problem 1: Give the prefix-codes for an alphabet using Huffman's Algorithm, given their frequencies. To construct a prefix code for an alphabet S , with given frequencies, use the following algorithm:

If S has two letters then

Encode one letter using 0 and the other letter using 1

Else

Let y and z* be the two lowest-frequency letters*

Form a new alphabet S' by deleting y and z* and replacing them with a new letter ω of frequency* $f_{y*} + f_{z*}$

Recursively construct a prefix code γ' for S' , with tree T'

Define a prefix code for S as follows:

Start with T'

Take the leaf labeled ω and add two children below it labeled y and z**

Endif

The algorithm constructs a binary tree in the following manner: take the two lowest-frequency letters of the alphabet and merge them into a single letter with a frequency equal to the sum of the frequencies of the merged letters, till only two letters are left in the alphabet.

After the tree is constructed, the letters are assigned codes by appending 0 and 1 to the prefix-code of the current node and giving them to each of the two children-nodes. Following this appraoch recursively upto the leaves, we arrive at an optimal prefix-coding.

The space complexity of the algorithm is O(n), because of construction of the binary tree, where n is the size of the alphabet.

The time-complexity of the algorithm depends upon the data-structure used to find the two least-frequency alphabets. The two different implementations of the problem are discussed separately.

Implementation A: Using Priority-Queue to store the alphabet during construction of binary tree.

In this approach, we initially construct a priority-queue, where the top-element is the one with the lowest frequency. Then, for (n-1) steps, we pick the top two elements from this queue and merge them into a single letter and add it to the priority-queue again.

The insertion and deletion operations – O(n) in number – are each of complexity O(logn). Therefore, time-complexity of this approach is O(n.logn).

Implementation B: Using arrays to store the alphabet during construction of binary tree.

In this approach, we store the elements in an array. For (n-1) iterations, we find the two least-frequency letters by applying two iterations of Selection-Sort. At the i-th iteration, we add the new alphabet at the (n-i)-th position in the array and thereafter consider only (n-i) elements of the array.

The insertion and deletion operations – O(n) in number – are each of complexity O(n), because there are two iterations of selection-sort, each of O(n) complexity. Therefore, time-complexity of this approach is $O(n^2)$.

Problem 2: Verify the statement for an incidence matrix M: "If a set of columns of M is linearly independent, then the corresponding set of edges does not contain a directed cycle."

Proof of correctness of statement: Let S be a subset of E, such that there is a directed cycle formed by the elements of S. Now, this would mean that for this subset, any node has as many outgoing edges as incoming edges. So, for all columns of the incidence matrix consisting only of edges in S, the sum is 0, and therefore these columns are not linearly independent.

"If there exists a subset of E with a directed cycle, then the corresponding subset of columns of M is not linearly independent" - is the contrapositive statement of the statement to be proved, and has been shown to be true. Therefore the statement "If a set of columns of M is linearly independent, then the corresponding set of edges does not contain a directed cycle." is true, by property of contrapositive statements.

Verification: The verification of the statement can be done by taking an arbitrary set S of columns of M and verifying that the corresponding set of edges do not form a cycle if elements of S are linearly independent. Independence of elements of S can be checked by using the Gauss elimination method to find the row-echelon form of the matrix M' formed by S.

The space-complexity of the algorithm is O(n). Only O(n) extra space is required to store if each vertex is visited, and to detect cycles.

The time-complexity of the algorithm is $O(m^2n)$. this can be shwon with the following arguments.

1. The time-complexity of DFS is O(n+m), used to detect cycle.

2. The Gauss-elemination method used nested loops, for a complexity of $O(m^2n)$.

Therefore the running-time of the algorithm is $O(m^2n)$.

Problem 3: For an undirected graph G = (V,E) with non-negative edge-weights, find an acyclic subset of E of maximum total weight.

The algorithm used to solve the problem is a modification of the Minimum Spanning Tree algorithm. Instead of taking minimum-weight edges at each step, we take maximum-weight edges. The algorithm used is the appropriately modified version of the Kruskal Algorithm.

Let T be an empty set.

Sort E by weights, in descending order

Let each node v in V be in its own disjoint set.

For each edge e in E

Let e join nodes u and v

If u and v are in different sets

Add e to T.

Add v to the set of u.

The returned tree T is the maximum-weight acyclic subset of E. This can be shown by the following arguments:

1. Since T is a tree, it does not have a cycle.
2. Since the algorithm chooses the maximum-weight edge at every iteration, it forms a tree with maximum-possible weight, without cycles.

The space-complexity of the algorithm is O(n+m) – O(n) for storing the set-identifier of every node and O(m) for the edges of T.

The time-complexity of the standard Kruskal Algorithm is O(m.logn). This is because sorting of edges is done in O(m.logm) = O(m.logn) time, and then the edges are traversed in O(m) time.

The modified algorithm just reverses the order in which the sorting is done. Therefore, the algorithm has a time-complexity of O(m.logn).

Problem 4: Find a schedule for n jobs to minimize the average-completion time.

We need to minimize $((\sum c_i)/n) = (\sum (s_i + t_i))/n$. Since t_i = time taken by the i-th job is constant for all i's, we need to minimize the start-time s_i for all jobs in order to minimize the average completion-time.

The start-time of the i-th job is minimum if all the (i-1) jobs before it have their durations $t_j < t_i$, because if there was a job k before i with $t_k > t_i$, then we could swap k and i to reduce the completion-times of all jobs from k-th position till i-th position.

Therefore, the optimal scedule is the one in which the start-times are arranged in ascending order. The algorithm is as follows:

> *Let S be the given set of activities.*
>
> *Sort S based on start-time to get S'*
>
> *S' represents the schedule, in order, for the minimum-average time.*

The time-complexity of the algorithm is O(n.logn), because the n activities are required to be sorted by start-time.

Problem 5: Schedule n activities using as few classrooms as possible, given their start-time and end-time.

This is an interval-partioning problem. To solve this problem, we can keep track of the maximum number of classrooms allotted after time t, and the classrooms which are available at time t.

To accomplish this, we notice that the start and end of an activity are two separate events. Whenever an event of type 'start' comes, we look for a free resource. If a free resource exists, we allocate that resource to the activity associated with this event. Otherwise, we allocate a new resource to this activity and increase our count of maximum resources used by one. Also, whenever an event of type 'end' comes, we add the resource allocated to the activity associated with this event to the list of free resources.

The outline of the algorithm is follows:

Let maxResources be 0 and freeResources be an empty queue.

For each activity i, add ($start_i$,'start') and (end_i, 'end') to a list L.

Sort L by the time of the events.

For each event e in L

If e_{type} = 'start'

If freeResources is empty

add 1 to maxResources

add resource number 'maxResources' to freeResources

Get a free resource from freeResources and assign it to the activity associated with e.

Else

add resource allocated to the activity associated with event e to freeResources

The n activities break up into 2n events. The 2n events get sorted in O(n.logn) time. All the events are traversed in O(n) time, and lookup for a free resource and allocation/deallocation of a resource for every event is done in O(1) time. Therefore, the time-complexity of the algorithm is O(n.logn).

Problem 6: Let T be a minimum spanning tree for a graph G = (V,E) with distinct positive weights. Let T' be a minimum spanning tree for a graph G' = (V,E'), where each edge e' in E' has a weight equal to the square of the weight of the corresponding edge e in E. Is T identical to T' ?

Assumption: All edge-weights are natural numbers.

Since all the edge-weights in G are distinct and positive, we can define an order of the edges in terms of their weights. Now, if the edge-weights are all positive and distinct, then the order of edges by weight is identical to their order by square of the weight, because (a < b) for natural numbers a,b implies ($a^2 < b^2$). Hence, the MST T and T' will be identical.

Since the edges have distinct weights, the algorithm cannot pick an arbitrary element at any point of execution to resolve ties. Therefore T will always be identical to T' in given conditions.

The verification is done by running the MST algorithm for both G and G' and verifying that T = T'.