COMP 6651 Algorithm Design Techniques

Assignment 2 Answers

Name : Parsa Kamalipour , *StudentID* : 40310734

Exercise 1: (15 marks)

Consider a modification of the rod-cutting problem in which, in addition to a price p_i for each rod, each cut incurs a fixed cost of c. The revenue associated with a solution is now the sum of the prices of the pieces minus the costs of making the cuts. Give a dynamic-programming algorithm to solve this modified problem.

Answers to Exercise 1

So, I've been working on a modification of the rod-cutting problem where, in addition to a price p_i for each rod length i, each cut incurs a fixed cost c. The idea here is to figure out the best way to cut the rod so that we can maximize the revenue, which is now the sum of the prices of the pieces minus the total cost of making the cuts.

Recurrence Relation

I define $dp[i]$ as the maximum revenue I can get for a rod of length i. The recurrence relation looks like this:

$$dp[i] = \max_{1 \le j \le i} (p_j + dp[i-j] - c)$$

where p_j is the price of a piece of length j, and c is the fixed cost for making a cut. The base case is simple:

$$dp[0] = 0$$

Pseudocode

```
Modified-Rod-Cutting(p, n, c):
    let dp[0..n] be a new array
    dp[0] = 0
    for i = 1 to n:
        max_value = -infinity
        for j = 1 to i:
            max_value = max(max_value, p[j] + dp[i - j] - c)
        dp[i] = max_value
    return dp[n]
```

Explanation

The approach I used here is dynamic programming to solve the problem efficiently. The algorithm iterates over all possible lengths of the rod, and for each length, it looks at all possible first cuts to find the one that gives the highest revenue, considering the fixed cutting cost c.

1) The outer loop runs for each rod length i from 1 to n.
2) The inner loop considers each possible first cut length j from 1 to i, calculating the revenue by adding the price of the first piece p_j and the maximum revenue for the remaining length $dp[i-j]$, while subtracting the fixed cost c.

The value $dp[n]$ at the end of the algorithm gives the maximum revenue for a rod of length n.

Time Complexity

The time complexity of this approach is $O(n^2)$, since it involves two nested loops: one for the rod length and one for possible cuts.

Exercise 2: (15 marks)

Give pseudocode to reconstruct an LCS from the completed $c[0..m, 0..n]$ table and the original sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ in $O(m + n)$ time, without using the b table.

Answers to Exercise 2

Now, let's move on to a problem involving the reconstruction of the Longest Common Subsequence (LCS). Given the completed $c[0..m, 0..n]$ table and the original sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$, we need to reconstruct an LCS without using the b table, and we want to do this in $O(m + n)$ time.

Approach

To reconstruct the LCS, we will start from $c[m, n]$ and trace our way back to $c[0, 0]$. At each step, we will decide whether to include the current character in the LCS based on the values in the c table.

Pseudocode

```
Reconstruct-LCS(X, Y, c):
    i = length(X)
    j = length(Y)
    lcs = []
    while i > 0 and j > 0:
        if X[i] == Y[j]:
            lcs.append(X[i])
            i -= 1
            j -= 1
        elif c[i-1][j] >= c[i][j-1]:
            i -= 1
        else:
            j -= 1
    lcs.reverse()
    return lcs
```

Explanation

1) Start from the bottom-right corner of the c table ($c[m, n]$).
2) If $X[i] == Y[j]$, it means that the character $X[i]$ (or $Y[j]$) is part of the LCS. We add it to the LCS list and move diagonally up to $c[i - 1, j - 1]$.
3) If $X[i] \neq Y[j]$, we move in the direction of the larger value between $c[i - 1, j]$ and $c[i, j - 1]$ to continue the trace.
4) We continue this until we reach $c[0, 0]$.

Time Complexity

The time complexity of this approach is $O(m + n)$ because, in the worst case, we make a linear pass through both sequences.

Exercise 3: (20 marks)

Consider a modification to the activity-selection problem in which each activity a_i has, in addition to a start and finish time, a value v_i. The objective is no longer to maximize the number of activities scheduled, but instead to maximize the total value of the activities scheduled. That is, the goal is to choose a set A of compatible activities such that $\sum_{a_k \in A} v_k$ is maximized. Give a polynomial-time algorithm for this problem.

Answers to Exercise 3

So, I've been looking at a modification to the classic activity-selection problem where, instead of maximizing the number of activities, we're interested in maximizing the total value of the selected activities. Each activity a_i has a start time, a finish time, and a value v_i. The goal is to pick a set A of non-overlapping activities such that the sum of their values is maximized.

Approach

To solve this problem, I decided to use dynamic programming to figure out the maximum total value of the activities that can be scheduled. We need to consider all activities in increasing order of their finish times and use an array to store the optimal solutions for subproblems.

Dynamic Programming Solution

Let n be the number of activities, and let the activities be sorted in non-decreasing order of their finish times. Define $dp[i]$ as the maximum value obtainable by scheduling activities from 1 to i. The recurrence relation for $dp[i]$ is:

$$dp[i] = \max(v_i + dp[p(i)], dp[i-1])$$

where v_i is the value of the i-th activity, and $p(i)$ is the index of the last activity that does not overlap with activity i. If no such activity exists, $p(i) = 0$.

The base case is:

$$dp[0] = 0$$

Pseudocode

```
Activity-Selection-Value(activities):
    sort activities by finish time
    let dp[0..n] be a new array
    dp[0] = 0
    for i = 1 to n:
        p_i = find_previous_non_overlapping(i, activities)
        dp[i] = max(activities[i].value + dp[p_i], dp[i-1])
    return dp[n]

find_previous_non_overlapping(i, activities):
    for j = i-1 down to 1:
        if activities[j].finish <= activities[i].start:
            return j
    return 0
```

Explanation

The dynamic programming approach works like this:

1) Sort the activities by their finish times.
2) Use an array dp to store the maximum value that can be obtained by considering activities up to index i.
3) For each activity i, calculate $dp[i]$ as the maximum between including the current activity (adding its value to the optimal solution of the previous non-overlapping activity) and not including it (taking the value of $dp[i-1]$).

The function `find_previous_non_overlapping(i, activities)` helps determine the most recent activity that does not overlap with activity i. This allows us to include the current activity without any conflicts.

Time Complexity

The time complexity of this approach is $O(n \log n)$ due to the sorting step. The subsequent iteration and the search for the previous non-overlapping activity can be optimized using binary search, resulting in an overall time complexity of $O(n \log n)$.

Exercise 4: (20 marks)

Prove that the total cost $B(T)$ of a full binary tree T for a code equals the sum, over all internal nodes, of the combined frequencies of the two children of the node.

Answers to Exercise 4

So, in this question, I need to prove that the total cost $B(T)$ of a full binary tree T for a code equals the sum, over all internal nodes, of the combined frequencies of the two children of the node.

Problem Statement

Consider a full binary tree T used for a code, where each leaf node represents a symbol with a certain frequency. The goal here is to prove that the total cost $B(T)$ of the tree equals the sum, over all internal nodes, of the combined frequencies of the two children of each internal node.

Approach

To solve this problem, I'm going to consider a full binary tree T where every internal node has exactly two children. Let f_i represent the frequency of each leaf node i.

The cost $B(T)$ is defined as the sum of the product of the frequency of each symbol and its depth in the tree. That is:

$$B(T) = \sum_{i \in \text{leaves}} f_i \cdot d_i$$

where d_i is the depth of leaf i in the tree.

We need to prove that this cost is equal to the sum, over all internal nodes, of the combined frequencies of the two children of each internal node.

Proof by Induction

We will use induction on the number of leaves n in the full binary tree T to prove the statement.

Base Case:

For $n = 2$, the tree has only one internal node, which is the root, with two leaves x and y. Let the root be z. The cost $B(T)$ is:

$$B(T) = f(x) \cdot d_T(x) + f(y) \cdot d_T(y)$$

Since both leaves are at depth 1, we have:

$$B(T) = f(x) + f(y) = f(\text{child 1 of } z) + f(\text{child 2 of } z)$$

Thus, the base case holds.

Inductive Step:

Suppose the statement is true for all full binary trees with $n - 1$ leaves. Now consider a full binary tree T with n leaves. Let c_1 and c_2 be two sibling leaves in T with the same parent p. Let T' be the tree obtained by deleting c_1 and c_2.

By the induction hypothesis, the cost of T' is:

$$B(T') = \sum_{l' \in T'} f(l') \cdot d_T(l') = \sum_{i' \in T'} (f(\text{child 1 of } i') + f(\text{child 2 of } i'))$$

Now, adding back c_1 and c_2 to form T, the total cost $B(T)$ is:

$$B(T) = \sum_{l \in T} f(l) \cdot d_T(l)$$

Breaking this sum down:

$$B(T) = \sum_{l \neq c_1, c_2} f(l) \cdot d_T(l) + f(c_1) \cdot (d_T(c_1) - 1) + f(c_2) \cdot (d_T(c_2) - 1) + f(c_1) + f(c_2)$$

Using the induction hypothesis:

$$B(T) = \sum_{i' \in T'} (f(\text{child 1 of } i') + f(\text{child 2 of } i')) + f(c_1) + f(c_2)$$

$$B(T) = \sum_{i \in T} (f(\text{child 1 of } i) + f(\text{child 2 of } i))$$

Thus, by induction, the statement is true for all full binary trees.

Conclusion

The total cost $B(T)$ of a full binary tree T is equal to the sum, over all internal nodes, of the combined frequencies of the two children of each internal node. This result comes from the way the cost is distributed across the levels of the tree and how each internal node contributes to the overall cost.

Exercise 5: (10 marks)

You perform a sequence of PUSH and POP operations on a stack whose size never exceeds k. After every k operations, a copy of the entire stack is made automatically, for backup purposes. Show that the cost of n stack operations, including copying the stack, is $O(n)$ by assigning suitable amortized costs to the various stack operations.

Answers to Exercise 5

So, in this question, I need to show that the amortized cost of a sequence of n stack operations, including copying the stack after every k operations, is $O(n)$.

Problem Statement

I perform a sequence of PUSH and POP operations on a stack whose size never exceeds k. After every k operations, a copy of the entire stack is made automatically for backup purposes. We need to show that the total cost of n stack operations, including the copying of the stack, is $O(n)$ by assigning suitable amortized costs to the various stack operations.

Approach

I'm going to use the **amortized analysis** technique to determine the cost of stack operations, including the cost of copying the stack. Amortized analysis lets us spread out the cost of expensive operations over a sequence of operations so that the average cost per operation stays low.

Analysis

Consider the sequence of n operations, where each operation is either a PUSH or a POP, and every k operations we make a copy of the entire stack.

Assigning Costs:

1) The cost of a PUSH or POP operation is 1 unit.
2) The cost of copying the stack is proportional to its size, which is at most k. So, the cost of copying is k units.

We need to figure out the total cost of n operations, including the copying.
Amortized Cost Calculation:

1) I assign an **amortized cost** of 2 units to each PUSH operation.
2) 1 unit is for the actual PUSH operation.
3) The extra 1 unit is saved to pay for the future copying cost.
4) Every time I perform k operations, I make a copy of the stack, which costs k units. Since I've saved 1 unit per PUSH operation, after k operations I have k units saved, which is enough to pay for the copying cost.

So, the amortized cost per operation is 2 units, and the total amortized cost for n operations is $2n$, which is $O(n)$.

Conclusion

By using amortized analysis, I've shown that the total cost of n stack operations, including the copying of the stack every k operations, is $O(n)$. The key idea is to spread the cost of copying across all the operations, resulting in an amortized cost of 2 units per operation.

Exercise 6: (20 marks)

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. You can improve the time for insertion by keeping several sorted arrays.

Specifically, suppose that you wish to support SEARCH and INSERT on a set of n elements. Let $k = \lceil \log_2(n+1) \rceil$, and let the binary representation of n be $\langle n_{k-1}, n_{k-2}, \ldots, n_0 \rangle$. Maintain k sorted arrays $A_0, A_1, \ldots, A_{k-1}$, where for $i = 0, 1, \ldots, k-1$, the length of array A_i is 2^i. Each array is either full or empty, depending on whether $n_i = 1$ or $n_i = 0$, respectively. The total number of elements held in all k arrays is therefore

$$\sum_{i=0}^{k-1} n_i 2^i = n.$$

Although each individual array is sorted, elements in different arrays bear no particular relationship to each other.

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

(b) Describe how to perform the INSERT operation. Analyze its worst-case and amortized running times, assuming that the only operations are INSERT and SEARCH.

Answers to Exercise 6

In this question, I'm trying to figure out how I can efficiently support SEARCH and INSERT operations for a set of n elements by using multiple sorted arrays.

Problem Statement

Doing a binary search on a sorted array is efficient, it takes logarithmic time, but inserting a new element has a linear time cost because you have to shift elements. To make insertion faster, I decided to use multiple sorted arrays.

Specifically, I want to support SEARCH and INSERT for a set of n elements. Let $k = \lceil \log_2(n+1) \rceil$, and represent n in binary as $\langle n_{k-1}, n_{k-2}, \ldots, n_0 \rangle$. I keep k sorted arrays $A_0, A_1, \ldots, A_{k-1}$, where array A_i has a length of 2^i for $i = 0, 1, \ldots, k-1$. Each array can either be completely filled or empty, depending on whether $n_i = 1$ or $n_i = 0$. So, the total number of elements across all arrays is:

$$\sum_{i=0}^{k-1} n_i 2^i = n$$

While each individual array is sorted, there's no ordering relationship between elements in different arrays.

Operations

.

SEARCH Operation:
Since I don't know how the elements are distributed between arrays, I have to linearly go through each array and do a binary search in each one. In the worst-case scenario, I end up searching through all of the arrays. Array i has a size of 2^i, and since it's sorted, I can search it in $O(i)$ time. As i varies from 0 to $O(\log n)$, the total runtime for SEARCH becomes $O(\log^2 n)$.

INSERT Operation:
For inserting, I place the new element into array A_0 and then update the rest of the arrays as necessary. In the worst case, I need to merge arrays $A_0, A_1, \ldots, A_{m-1}$ into a new array A_m. Since merging two sorted arrays can be done in linear time relative to the combined length, the worst-case merge takes $O(2^m)$ time. In the worst case, this becomes $O(n)$ when m equals k.

To analyze the amortized cost, I use the accounting method. I assign a cost of $\log n$ to each insertion. This means that each inserted element carries enough credit to cover the costs of its future merges as other elements are added. Since an individual element can only be merged a maximum of $\log n$ times into increasingly larger arrays, this credit will pay for all its future costs. So, the amortized cost per insertion is $O(\log n)$.

Conclusion

By maintaining multiple sorted arrays and leveraging binary search for SEARCH operations and repeated merging for INSERT operations, I achieve an efficient way to manage the dynamic set of elements. The worst-case time complexity for SEARCH is $O(\log^2 n)$, and the amortized time complexity for INSERT is $O(\log n)$.

Acknowledgement

I would like to acknowledge that I used ChatGPT to help refine my grammar and sentence structure in this document. However, the solutions and ideas presented here are entirely my own.