CP8201/CPS815 Advanced Algorithms

Assignment 1

Rohaan Ahmed
PhD Student
Department of Computer Science
rohaan.ahmed@ryerson.ca

October 10, 2020

Instructor: Dr. Javad (Jake) Doliskani
October 10, 2020

1 A greedy algorithm for finding a minimal subcover of an interval (a, b).

Inputs to the algorithm are:

- $X = (a, b)$: an interval.
- $C[1, ..., n]$: an array of intervals that represents a covering set for (a, b). The interval at index i is $(C[i].a, C[i].b)$.

Algorithm 1 Function: ***minimal_subcover(X = (a,b), C)***

```
Sort C by (C[i].b − a) in descending order, and re-number it (C[i].a, C[i].b)
U = C {Comment: U stores the unselected elements of C after each iteration}
current_start_point = a {Comment: current_start_point is the starting point of the still uncovered interval after each iteration}
R = empty {// R stores the subintervals selected as part of the optimal sub cover}
while U is not empty and current_start_point < b do
  for i ← 0 to length of U do
    Select the subinterval in U which satisfies the conditions:
    1. Includes the current_start_point
    2. Covers the largest portion of the remaining uncovered interval:
    max(U[1, ..., n].b − current_start_point).
    {// This is the 'Greedy' part of this Greedy Algorithm}
  end for
  Update current_start_point with the end of the covered interval
  Add the selected subinterval to optimal subcover R
  Delete the selected subinterval from U
end while
return R
```

In the above algorithm, we start find the minimal subcover of an interval $X = (a, b)$. We start at a and pick the interval covering the furthest right while containing a. We then set the end of the covered interval as the new start point, *current_start_point*, and repeat until our subcover inludes b.

Complexity Analysis

The algorithm above can be executed, in the worst case, in $O(n^2)$ time.

1. The while loop executes until the entire interval (a, b) is covered. In the worst case, this loop would take $O(n)$ time, where n is the total number of sets in C.
2. Inside the loop, we select the $C[i]$ which covers the largest portion of the remaining uncovered interval of (a, b) and includes the new *current_start_point*. This requires an iteration through the remaining C for comparison, which, in the worst case, would execute in $O(n)$ time, where n is the number of elements in C.

The sorting performed at the beginning of the algorithm may help improve the average case time-complexity, since C is sorted in order of largest to smallest with respect to their end-points from a. This sorting operation would take $O(nlogn)$ time. Once this is done, the comparison operation (Step 2) would take less time since the $C[i]$ which covers the largest portion of the remaining uncovered interval of (a, b) would be found much quicker, on average (i.e., $O(log(n))$). Thus, with pre-sorting, the *average* complexity of the above algorithm may be reduced to $\Theta(nlogn)$.

Note, this algorithm can also be written using Recursion, where, at the end of each selection step (Step 2), we call the function with the remaining uncovered interval, minimal_subcover(X = (*current_start_point*,b), C= U). We will use this fact to prove that this Algorothm is optimal.

Proof

We will use Proof by Induction:

Consider the set S_{all} of all the subintervals covering $X = (a, b)$. One of these sets, S_{OPT}, is the optimal minimal subcover of the interval.

Base Case:

S_{OPT} must then contain a subinterval $S_i = (a_i, b_i)$ which meets the following two conditions:

- Contains point a
- Its right end-point, b_i, is maximal in S_{all} with respct to a, i.e., reaches furthest to the from a compared to all other subintervals.

The remaining uncovered interval will therefore be $S_{OPT} - S_i = S_{OPT-i} = (b_i, b)$.

Base+1 Case:

Similarly, an optimal subcover of the interval (b_i, b) must contain a subinterval that also meets the two conditions:

- Contains point b_i
- Its right end-point, b_{i+1}, is maximal in S_{all} with respct to b_{i+1}.

Induction Case:

The remaining uncovered interval (b_{i+1}, b) is a subset of the remaining intervals within the optimal solution, $S_{OPT-(i+1)}$, and thus, can be covered with *no more* subintervals than the remaining uncovered intervals from the optimal solution, $S_{OPT-(i+1)}$.

Concluding Statement:

Therefore, any solution constructed by combining the optimal solution of $S_i = (a_i, b_i)$ and the optimal solution for the remaining interval S_{OPT-i} will also be optimal. In other words, any optimal subcover of interval $X = (a, b)$ must be a union of the the optimal subcovers of its sub-intervals, $S_{OPT} = S_{OPT-(i)} \cup S_{OPT-(i+1)} \cup ....$ This is precisely what the algorithm provided above does.

2.1 An algorithm that determines whether a given string of parentheses is balanced.

In this and the subsequent sections, it is assumed that the order in which the brackets appear matters for the purposes of balanced vs unbalanced.
For example: w = {{}} is balanced, and w = }}{{ is unbalanced.

Algorithm 2 Function: ***is_balanced(w)***

```
disbalance_counter ← 0 {// disbalance_counter will hold the number of parenthesis that are unbalanced}
for i ← 0 to |w| do
  if w[i] is { then
    disbalance_counter ← disbalance_counter + 1
  else if w[i] is } then
    disbalance_counter ← disbalance_counter − 1
  end if
  if disbalance_counter < 0 then
    Exit Loop
  end if
end for
return TRUE if disbalance_counter is 0
```

The above algorithm reads in a string w containing only two values, { and }, and returns $TRUE$ when the string is Balanced, and $FALSE$ otherwise.

We start by initializing a counting variable, *disbalance_counter* to 0. We then iterate through the length of the string w once, adding 1 to *disbalance_counter* when an open bracket is encountered, and subtracting 1 from *disbalance_counter* when a closed bracket is encountered. At the end of the loop, *disbalance_counter* holds the total number of brackets that do not have a corresponding bracket, i.e., number of unbalanced brackets in the string. Thus, if *disbalance_counter* = 0, then the string is Balanced.

Complexity Analysis
In the above algorithm, we iterate through the length of the string exactly once, counting up or down depending on its elements. Thus, this algorithm has $O(n)$ time complexity. Furthermore, since we only use a counter (rather than a stack as in section 2.2), it has a constant space complexity.

2.2 A greedy algorithm to compute the length of largest balanced substring of a string of parentheses.

Algorithm 3 Function: ***len_longest_balanced_substring(w)***

```
Initialize stack and push first index of w: stack = [1]
Initialize max_substring_length = 0 and max_substring = w
for i ← each index of w do
  if w[i] is { then
    Push i into stack {// If opening bracket, push index i into stack.  This starts
    a new valid substring}
  else if w[i] is } then
    Pop the last opening bracket's index
    if stack is not empty then
      substring_first_index ← last element of stack
      substring_last_index ← i
      {// If length of the current balanced substring is more than previous max
      balanced substring, then make this substring the new max.  This is the
      'Greedy' part of this Greedy Algorithm}
      if length of current balanced substring > max_substring_length then
        max_substring ← current balanced substring
        max_substring_length ← length of current balanced substring
      end if
    else
      Push i into stack {// If stack is empty, push i as base for next valid
      substring}
    end if
  end if
end for
Return max_substring, max_substring_length
```

The above algorithm reads in a string w containing only two values, { and }, and returns the length of the largest Balanced substring. As a bonus, we also return the the largest Balanced substring, which is a no-penalty benefit of using the stack-based implementation.

Complexity Analysis

In the above algorithm, we loop through the length of the string exactly once, adding and removing elements to the stack (which is a contstant-time operation). Thus, this algorithm has $O(n)$ time complexity. Furthermore, since our stack-size grows linearly with the length of the string, this algorithm also has $O(n)$ space complexity

Proof

By definition, a string w is called balanced if it satisfies the following recursive definition:

- w is the empty string, or

- w is of the form { x } where x is a balanced string, or
- w is of the form xy where x and y are balanced strings.

Let:
let $x \in \mathbb{N}_0 = \{0, 1, 2, ...\}$ = number of open parentheses in the string
let $y \in \mathbb{N}_0 = \{0, 1, 2, ...\}$ = number of closed parentheses in the string
Then:

$$|w| = x + y \tag{0.1}$$

where:
$|w|$ is the length of the input string w

The above definiton can then be re-writen as follows: a substring is **balanced** if and only if it meets the following two conditions:

Conditions

1. It is a **valid** string, i.e., a close bracket, }, is not encountered before a corresponding open bracket, {, is encountered. Namely,:

$$\forall i \in \{1, ..., |w|\}, y_{1,...,i} \leq x_{1,...,i} \tag{0.2}$$

2. It has an **equal** number of open and closed brackets:

$$x = y \tag{0.3}$$
$$\implies |w|_{balanced} = x + y = 2x = 2y \tag{0.4}$$

Proof of Condition 1:
In the algorithm presented above, a substring is only considered valid when there are 0 or more opening brackets, {, in the stack. Whenever a closed bracket is encountered while there are no open brackets in the stack, the previous valid substring is considered ended, and the loop continues. A new valid substring begins once an open bracket is encountered. Thus, the algorithm ensures that a substring is only valid when Equation 0.2 is true.

Proof of Condition 2:
An alternate way of writing Condition 2 is: a string or substring is **unbalanced** if $x > y$ and there exists $a > 0$ such that:

$$x = a + y \tag{0.5}$$

(Please note: It can never be that $y = a + x$ because it would violate Condition 1 ($y \not> x$), and thus, be disqualified from consideration as a balanced string)

Substituting 0.5 into 0.1, we get:

$$|w|_{unbalanced} = a + 2y \tag{0.6}$$

Comparing 0.6 to 0.4, we can see that:

$$\implies |w|_{balanced} = |w|_{unbalanced} - a \tag{0.7}$$

Equation 0.7 tells us that the length of the largest balanced substring, $|w|_{balanced}$, within an unbalanced substring, $|w|_{unbalanced}$, is given by $(|w|_{unbalanced} - a)$, where $a = x - y$ (from 0.5).

The algorithm given above returns a substring of length $|w|_{balanced} = |w|_{unbalanced} - a$ for every valid substring within a given string. In other words, for every substring within a string that is valid (meets Condition 1), the algorithm returns the largest balanced substring (meets Condition 2).

EXTRA: A greedy algorithm that returns the largest balanced substring of a string of parentheses (when the order of the brackets does not matter).

This section is presented here because the author initially misread the question in section (2.2) and developed this algorithm, rather than the one presented above. Rather than delete the entire section, the author has decided to keep it for discussion purposes.
In this section, it is assumed that the order in which the brackets appear does not matter for the purposes of balanced vs unbalanced.
For example: w = {{}} and w = }}{{ are both balanced.

Algorithm 4 Function: ***is_balanced2(w)***

```
disbalance_counter ← 0 {// disbalance_counter will hold the number of parenthesis that are unbalanced}
for i ← 0 to |w| do
  if w[i] is { then
    disbalance_counter ← disbalance_counter + 1
  else if w[i] is } then
    disbalance_counter ← disbalance_counter − 1
  end if
end for
return disbalance_counter
```

Algorithm 5 Function: *largest_substring(w)*

```
disbalance_counter ← is_balanced2(w)
if disbalance_counter is 0 then
  return w
else
  new_substring ← substring of w from 0 to (|w| − disbalance)
  new_disbalance ← is_balanced2(new_substring)
  if new_disbalance is 0 then
    return
  else
    {// Start sliding window on w}
    for k ← 1 to disbalance_counter do
      if new_substring[first_element] is { then
        new_disbalance ← new_disbalance − 1
      else if new_substring[first_element] is } then
        new_disbalance ← new_disbalance + 1
      end if
      new_substring ← substring of w from k to (|w| − disbalance + k) {// Slide window by
      1}
      if new_substring[last_element] is { then
        new_disbalance ← new_disbalance + 1
      else if new_substring[last_element] is } then
        new_disbalance ← new_disbalance − 1
      end if
      if new_disbalance is 0 then
        Exit Loop
      end if
    end for
  end if
end if
```

The above algorithm reads in a string w containing only two values, { and }, and returns the largest Balanced substring.

We start by calling the function defined in section (2.2), $is\_balanced2(w)$, which returns $disbalance\_counter$. We know that:

Length of Largest Balanced Substring: $substring\_length = |w| - disbalance\_counter$

To find the largest balanced substring, we iterate through the entire string use a sliding window of $substring\_length$. As we move the window, we update a variable $new\_disbalance$, which keeps track of the degree of disbalance (number of brackets without a corresponding bracket).

$new\_disbalance$ Update Rule:

We iterate through the length of the string, w, using a sliding window of $substring\_length$ by incrementally deleting the first element of the substring, and adding to the last element, until we reach the end of the string or find a balanced substring.
Similarly, we update $new\_disbalance$ by adding or subtracting 1 to it as we add and delete the elements. At the end of the loop, $new\_disbalance$ holds the total number of brackets that do not have a corresponding bracket in the substring, i.e., the degree of disbalance for the substring. When we reach $new\_disbalance = 0$, the largest Balanced substring has been found.

Complexity Analysis
In the above algorithm, in the worst case scenario, we loop through the length of string, w, exactly 3 times:

1. Evaluate the entire string using the function $is\_balanced2(w)$, which loops through the string exactly once - $O(n)$
2. Evaluate the first substring using the function $is\_balanced2(w)$, which loops through the substring exactly once - $O(n)$
3. Iterating through the string using a sliding window substring, which loops through the string exactly once - $O(n)$

Thus, the worst-case complexity of the entire algorithm is $O(n)$