Michael Inoue
405171527

Ling
Discussion 1C 2-3:50PM

CS 180

Homework 4

30 October 2019

1 Solution

Algorithm 1: Minimize the weighted sum of completion times

Input : List of customers $C_i \in C$ with completion times t_i and weighted importance w_i
Output: Ordering of list of customer jobs such that weighted sum of completion times is minimized

```
Each customer C_i ∈ C has completion times t_i and weighted importance w_i
Perform merge sort on C in descending order of C_i's w_i/t_i
return C
```

We first prove the correctness of our algorithm. Suppose we have some optimal algorithm O that does not follow the criteria of our algorithm A. Then, we have some sequence C_s and C_t such that C_t follows C_s but we have that

$$\frac{w_s}{t_s} < \frac{w_t}{t_t}$$

or equivalently,

$$w_s t_t < w_t t_s.$$

Now, consider swapping these elements such that C_s follows C_t, as our algorithm would. Now we wish to prove that this swap does not increase our weighted sum. Consider the weighted sum before the swap. We have our total weighted sum S,

$$S = y + w_s\left(x + t_s\right) + w_t(x + t_s + t_t)$$

where y is the weighted sum of the elements before C_s and C_t (if any), and x is the total time taken by each job preceding C_s and C_t. Now, consider the weighted sum after our swap. We have our total weighted sum S',

$$S' = y + w_t\left(x + t_t\right) + w_s(x + t_t + t_s).$$

Now, expanding S and S', we have:

$$S = y + w_s x + w_s t_s + w_t x + w_t t_s + w_t t_t$$

$$S' = y + w_t x + w_t t_t + w_s x + w_s t_t + w_s t_s$$

Now, we claim that $S' \leq S$, or equivalently,

$$y + w_t x + w_t t_t + w_s x + w_s t_t + w_s t_s \leq y + w_s x + w_s t_s + w_t x + w_t t_s + w_t t_t$$

Cancelling common terms on each side of the equality yields: $w_s t_t \leq w_t t_s$ But by assumption $w_s t_t < w_t t_s$, so the inequality holds. Thus, our swap does not increase the total weighted sum for any sequence C_s, C_t in our list.

Thus, we may swap every pair that does not meet our sorting criteria in the optimal ordering, obtaining our own solution without adding to the total weighted sum. Hence, we observe that our solution must also be optimal.

Complexity Analysis:

Now, our algorithm only performs merge sort on a list and then returns that list. Because we proved in lecture that merge sort runs in $O(n \log n)$ time, we conclude that our algorithm runs in $O(n \log n)$ time as well.

2 Solution

Algorithm 2: Maximize number of jobs in 24 hour schedule

Input : Intervals $I_i \in I$ with start times s_i and end times e_i

```
Perform merge sort on I such that each I_i in ascending order of end times e_i
Initially R and J are empty lists of intervals
Store I in R
foreach I_i ∈ R do
    if e_i < s_i (the interval 'wraps around' midnight) then
        Store I_i in J
    end
end
foreach I_j ∈ J do
    Initially integer localMax is 0
    Store I in R
    Store I_j in schedule A
    Delete I_j from R
    Increment localMax by one
    Delete all intervals from R that overlap with request I_j
    while R is not empty do
        Choose an interval I_i ∈ R that has the smallest e_i
        Add I_i to A
        Increment localMax by one
        Delete all intervals from R that are not compatible with interval I_i
    end
    if localMax > globalMax then
        Set B = A
        Set globalMax = localMax
    end
end
Store I in R
while R is not empty do
    Initially integer localMax is 0
    Choose an interval I_i ∈ R that has the smallest e_i and I_i ∉ J
    Add I_i to A
    Increment localMax by one
    Delete all intervals from R that are not compatible with interval I_i
    if localMax > globalMax then
        Set B = A
        Set globalMax = localMax
    end
end
```

First, we prove the correctness of the algorithm. Our algorithm seeks to find all the intervals that 'wrap around' midnight and stores them in list J. Now, the optimal solution will either contain one of these intervals that wraps around J (only one, as by definition, because each must 'wrap around' midnight, they all have some overlap), or it will not. Now, our algorithm adds each such interval $I_j \in J$ and performs the standard interval scheduling procedure we learned in class. If this particular I_j is in the final optimal solution, then our solution must be optimal, as our solution to the interval scheduling problem (choosing the interval with the first end time and deleting all the other intervals that conflict) was proven to be optimal in lecture. Our algorithm then updates the final optimal schedule according to each $I_j \in J$, that is, which I_j being in the solution yields the maximum number of jobs.

Finally, we must check the one solution that contains no such interval that wraps around midnight (indeed, it is possible that our optimal solution does not contain such an interval). We simply perform the standard interval scheduling procedure as before, with the caveat that no interval can be contained in J. If the schedule produced from this procedure contains more jobs than the maximum number of jobs with an interval $I_j \in J$, then this must be the optimal solution.

We have accounted for every possible solution, breaking down each case into a variant solution of the standard interval scheduling problem we covered in class. As our solution was proven to be optimal then, we conclude that our solution must also be optimal now.

Complexity Analysis:

Our algorithm first sorts our job intervals using merge sort, in ascending order of end times. Merge sort was proven to be $O(n \log n)$ in lecture, so this step is $O(n \log n)$.

Next, for each interval $I_j \in J$ (intervals that wrap around midnight), our algorithm performs the standard interval scheduling procedure covered in class. Such a procedure was proven to be $O(n)$ is lecture. Because we can have at most n intervals $I_j \in J$, this process in total is $O(n^2)$.

Finally our interval performs the standard interval scheduling procedure once more, with the caveat that each interval cannot be contained in J. Checking this for each interval takes n steps to iterate through J, and we iterate through n intervals while performing the standard interval scheduling procedure, so this step is $O(n^2)$.

(We also update the maximum list schedule accordingly if we find a schedule with a greater number of intervals, but this step is $O(1)$, and is thus inconsequential).

Thus, adding up our total time complexities, we have our time complexity, $T(n)$:

$$
\begin{aligned}
T(n) &\leq O(n \log n) + O(n^2) + O(n^2) && (1) \\
&\leq O(n^2) && (2)
\end{aligned}
$$

so our total time complexity is $O(n^2)$.

3 Solution

Algorithm 3: Determines if $> \frac{n}{2}$ cards belong to the same bank account

Input : n cards

Output: Determination if $> \frac{n}{2}$ cards belong to the same bank account

```
N is a list of all cards
Set list L = N
while L has more than one card do
    Pair up each remaining card arbitrarily (if n is odd, leave one card aside, keep it for last
     comparison)
    foreach pair p = (i, j) do
        if i and j are equivalent then
            Keep i, remove j from L
        else
            Remove both i and j from L
        end
    end
end
if L is empty then
    return no majority exists
else
    x is the last remaining element in L
    Initialize integer count to 1
    foreach card y ∈ N (and x is not the same card as y) do
        if x and y are equivalent then
            Increment count by 1
        end
    end
    if count > n/2 then
        return majority exists
    else
        return no majority exists
    end
end
```

We first prove the correctness of the algorithm. Our algorithm simply pairs up cards, removing pairs that are non-equivalent and keeping one card from pairs that are equivalent, repeating this pairing and deleting process until we are only left with one card remaining. Indeed, if such a majority card existed (i.e. there are more than $\frac{n}{2}$ equivalence relations for such a card), then this process should always leave us with a valid candidate for such a majority card. By the pigeonhole principle, with n possible bank accounts, only one can have more than $\frac{n}{2}$ equivalences.

However, this does not necessarily imply that our majority card candidate is indeed the majority card. As we did with the famous problem discussed in the first lecture, we must compare the candidate with every other card that we threw away, as we don't necessarily know the relationship between the majority candidate and every other card. If more than $\frac{n}{2}$ cards are equivalent to this majority candidate (including the candidate itself), than this indeed must be the majority card, and hence we return the fact that a majority exists. However, if there are only $\frac{n}{2}$ cards or less equivalent to this candidate (including the candidate itself), then we know, once more by the pigeonhole principle and our setup that no such card exists, and thus no majority exists. Our algorithm does precisely this, so we conclude that it yields correct results.

Complexity Analysis:

Our algorithm first pairs up the n cards, getting rid of cards or pairs entirely depending on if said pairs are equivalent or not. There can be at most n pairs, so there are at most $\frac{n}{2}$ comparisons.

Our while loop then runs again, this time with at most $\frac{n}{4}$ cards remaining in the worst case, as we have removed in the worst case half of the cards from each of the $\frac{n}{2}$ pairs, thus leaving us with $\frac{n}{4}$ pairs. We may repeat the same process as we did before, pairing each card up and comparing, now making $\frac{n}{4}$ comparisons.

We can repeat this process of halving and comparing until we only reach one card remaining in the worst case. Thus, our total number of comparisons is equivalent to

$$\frac{n}{2} + \frac{n}{4} + \frac{n}{8} + ... = n$$

comparisons in the worst case (as the sequence above is simply a geometric series that converges to n). Thus, this process runs in $O(n)$ time.

Next, in the worst case, we will have one remaining card left. We must compare this card with all other cards in our original list. Because there are n cards total, this process is $O(n)$ in the worst case.

Thus, adding up our total time complexities, we have our time complexity, $T(n)$:

$$\begin{aligned} T(n) &\leq O(n) + O(n) && (3) \\ &\leq O(n) && (4) \end{aligned}$$

We note that $T(n)$ is also $O(n \log n)$, as $O(n)$ is $O(n \log n)$.

4 Solution

Algorithm 4: Finds local minimum in G

Input : Graph G in an nxn grid
Output: Local minimum in G
Initially list of nodes B is every node on border
Set list of nodes X to every node in middle row or middle column of G that aren't in B
Set list of nodes Y to every node adjacent to B or X
Initially node *minbord* has max value
Initially node u is uninitialized
foreach *node v on border B* **do**
    if *v's value is < minbord's value* **then**
        $minbord = v$
    end
end
if *minbord's value < the value of all of its neighbors* **then**
    return v
else
    Consider each node in either X and Y
    Set u = smallest node compared to those in either X and Y
end
if $u \in X$ **then**
    return u
else
    Let H be one of the four sub-grids of G such that H contains u and is bordered by X and B
    recursively return Algorithm 4 on H
end

We first prove the correctness of our algorithm. Our algorithm first analyzes the nodes contained in the border B of our grid G. It finds the minimum of such nodes, and then checks if it is the minimum amongst all of its neighbors. If it is, then we are done, as by definition, this node must be a local minimum.

Now, if not, we now that our node is local to the interior of G. We construct two sets of nodes, X and Y. X is defined to be the node in the middle row and middle column of G. Y is defined to be every node adjacent to B or X. Indeed, if our set X and B were removed from our grid, we would be left with 4 subgraphs of G.

Now, we find the minimum node u in either X or Y. If u is contained in X, then it must be a local minimum, as u's neighbors are either contained in Y or X. Thus, our algorithm returns this node if this is the case. However, if u is contained in Y, we have no guarantee that its value is less than its neighbors, as it may be that there is some $t \notin X, Y$ such that t's value is less than u's. So, we take one of the aforementioned four subgrids, H, that contains u and recursively run our algorithm on H. We know that H must contain some interior local minimum, so our recursive call should eventually yield a correct result.

Thus, we conclude that our algorithm yields correct results.

Complexity Analysis:

Our algorithm first analyzes each node contained in the borders of our subgrid G to find a minimum, *minbord*. G is an nxn grid, so this should run in $O(n)$.

Next, our algorithm compares *minbord* to its neighbors. This comparison runs in $O(1)$.

Now, our algorithm analyzes each node in the middle row or column, X, of G that is not in our border and in the bordering nodes Y of our border B and X. Now, X contains at most $2n$ nodes and Y contains at most $4n$ nodes, so this runs in $O(n)$ as well. Thus, this process is $O(n)$.

Finally, our algorithm checks if u is in X or not. If it is, then it simply returns u, and if not, it will recursively run our algorithm on one of the four subgrids contained in G. H is an $\frac{n}{2}$x$\frac{n}{2}$ grid, so our number of probes, $T(n)$ can thus be given by:

$$T(n) = O(n) + T\left(\frac{n}{2}\right).$$

Solving this equation yields $T(n) = O(n)$. Thus, our total time complexity is $O(n)$.

5 Solution

Algorithm 5: Find rotation number k

Input : Rotated array A, lower index l, higher index h
Output: Rotation number k

Initially integer m is uninitialized
if $h = l$ **then**
  return h
end
if $h < l$ **then**
  return 0
end
Set $m = \frac{(l+h)}{2}$
if $m < h$ *and* $A[m+1] < A[m]$ **then**
  return m
end
if $m > l$ *and* $A[m] < A[m-1]$ **then**
  return m
end
if $A[h] > A[m]$ **then**
  return Algorithm 4 (A, l, $m-1$)
else
  return Algorithm 4 (A, $m+1$, h)
end

We first prove the correctness of our algorithm. Our algorithm utilizes three key variables: h, the upper index of the array we wish to analyze, l, the lower index of the array we wish to analyze, and m, the midpoint of h and l. It is assumed that when this algorithm is called on A, the initial parameters are the initial index, (0), for l, and the upper index (the index of the last element in the array) for h.

Now, we utilize one of the key properties of our rotation: the first element of the array (the minimum element) before rotation is the only element such that, after rotation, the element preceding it (if it exists) is larger, and the element following it (if it exists) is smaller. Indeed, if no element exists before this minimum element, than there must be no rotation (as our array is sorted), and if no element exists after this minimum element, then if our array has n elements, our rotation must be $n-1$ (the maximum, non-cyclical rotation count), as our smallest element is now in the last element position.

Now, keeping this in mind, our algorithm aims to find the displacement of our first element, as this will give us the rotation number. It first checks if $h = l$ (our first base case). Indeed, the index of our highest element in our array is equal to the last element of our array, then we are left with only one element in our array and thus we may simply return the index of said element in our array (either h or l).

Next, our algorithm checks if $h < l$. This would imply that our element was not found, and we have 'wrapped around' our array. By construction, we will see that this will only occur if there was no shift in our array. Thus, we return 0 in this case.

Next, we compare the midpoint m and the high index of our array h, as well as A$[m]$ and A$[m+1]$. If $m < h$ and A$[m+1] <$ A$[m]$, this implies that the index of our smallest element is at position m (recall our precondition stated in the first paragraph; there is only one such element that maintains this property, and its index should be the rotation factor). If this is the case, then we return m and we are done.

Next, we compare the midpoint m and the low index of our array l, as well as A$[m]$ and A$[m-1]$. If

$m > l$ and A$[m] <$ A$[m-1]$, this implies that the index of our smallest element is at position m (recall again our precondition stated in the first paragraph; there is only one such element that maintains this property, and its index should be the rotation factor). If this is the case, then we return m and we are done.

Now, if this condition doesn't hold, then we decide to either check the subarray either to the left or right of our midpoint. Now, if A$[h] >$ A$[m]$, then the right subarray is sorted, as it was before the rotation, implying that our minimum element is in the left subarray. If not, we check the right subarray. Our algorithm does just this in lines (15-18). Now, our base case holds and our recursive steps and logic are valid, so we conclude that our algorithm is correct.

Complexity Analysis:

Our algorithm first initializes variables and makes comparisons between array elements and m, l, and h. Such comparisons are done in constant time, so this runs in $O(1)$. Next, in the worst case, our algorithm makes a recursive call on either the right half or left half of our subarray, which eventually will terminate when we are left with one element in our array in the worst case. In total, we are thus left with the following recurrence relation for our time complexity, $T(n)$:

$$T(n) = T\left(\frac{n}{2}\right) + c$$

In class, we proved this can be solved as $T(n) = O(\log n)$, so our algorithm runs in $O(\log n)$ time.

6 Solution

Extracting Minimum Value:

Algorithm 6: Extract the minimum value

```
Input  : Heap h
Output: Minimum value
Let r be the root node
Let e be the node at the end of the heap
Set u = r
Swap values of r and e
Delete e
Decrease heap size by 1
while r's value > its children's value do
    Swap r's position with the lesser of its two children
    Relink nodes to signify swap has occurred (swap parent and child pointers)
end
return u
```

Insertion:

Algorithm 7: Insert a node

```
Input  : Heap h
Increase heap size by 1
Insert v at the end of the heap
while v's value is less than its parent's value do
    Swap v's position with it parents
    Relink nodes to signify swap has occurred (swap parent and child pointers)
end
```

Changing Value:

Algorithm 8: Change the value of a node

```
Input : Heap h with node v to be changed to have value x
Set v's value = x
while v's value is less than its parent's value or v's value is greater than its children's values do
    if v's value is less than its parent's value then
        Swap v's position with it parents
        Relink nodes to signify swap has occurred (swap parent and child pointers)
    end
    if v's value is greater than its children's values then
        Swap v's position with the lesser of its two children
        Relink nodes to signify swap has occurred (swap parent and child pointers)
    end
end
```

We first prove the correctness of our algorithms.

Extracting the minimum value works by swapping the values of the node at the end of the heap (the right-most node) and the root node. Now, we delete the node at the bottom of the heap, and we have effectively removed the minimum value. We now have to reheapify. This requires 'bubbling down' our new root node. We do this by swapping the root with the less of its two children, repeating this process until we have a valid heap structure once more. We have extracted the minimum value and have reheapified our heap, so our algorithm is valid.

Inserting a node works in a similar fashion, only opposite. Now, we add a node v to the end of a heap, and 'bubble' up v by comparing it with its parent node. If v is less than its parent node, we swap v and its parent, and repeat this process until we have a valid heap structure once more. We have inserted our node and have reheapified our heap, so our algorithm is valid.

Changing the value of a node utilizes strategies used in the previous two algorithms. It changes the value of a given node v, and either 'bubbles v up' or 'bubbles v down' depending on if v is less than its parent's value or greater than its children's values, respectively. We utilize the same methodology for bubbling up and bubbling down v, repeating this process until we have a valid heap structure once more. We have changed the value of our node and have reheapified our heap, so our algorithm is valid.

Time Complexity

Our extraction algorithm swaps values, deletes nodes, and decreases the heap size, which takes $O(1)$ time. It then bubbles down our root node r down the heap. In the worst case, it will traverse to the bottom of our heap. Because we have a balanced heap, we will traverse down at most the height of our heap. As $n = 2^h$, or equivalently, $h = \log n$, this will take $O(\log n)$ time.

Our insertion algorithm first increases the heap size, which takes $O(1)$ time. It then bubbles up our node v, starting at the bottom of the heap and traversing upwards. Because we have a balanced heap, we will traverse up at most the height of our heap. As $n = 2^h$, or equivalently, $h = \log n$, this will take $O(\log n)$ time.

Our value changing algorithm first sets our node v's value to our desired value x. This takes $O(1)$ time. It then swaps v's position with its parents or v's value with its children, continually doing this until we have a valid heap structure. Now, we will either traverse up the heap or down the heap, but not both, as v's value cannot be simultaneously be less than its parent's and greater than its children's, as that would violate the properties of a proper heap. Thus, in the worst case, v will have to bubble up from the bottom to the top or bubble down from the top to the bottom. Because we have a balanced heap, it will traverse up or down at most the height of our heap. As $n = 2^h$, or equivalently, $h = \log n$, this will take $O(\log n)$ time.