CS 180 Homework 4

Jiaping Zeng

11/14/2020

4.14 (a) **Algorithm**:

1. Sort the list of sensitive processes by finish times.
2. While there is processes left on the list, do the following:
   - Invoke `status_check` at the finish time of the first unchecked process.
   - Remove the process from the list as well as other processes that were running during the check.

Proof of correctness: By contradiction. Suppose that the algorithm produce the correct status check schedule, then it must either missed at least one sensitive process or produced a schedule that is not optimal. The first case is not possible as we are only removing processes from the sorted list when there is a `status_check` that occurs while the process is running. If the second case were to happen, there must exist another schedule that checks all sensitive processes while invoking `status_check` fewer times. Then there must exist a time where the optimal schedule invokes `status_check` that checked more processes than the one produced by our algorithm. However, since our algorithm sorts the processes by finish times, `status_check` already checks all the overlapping processes there are. Therefore our algorithm does always provide the optimal schedule.

Time complexity: $O(n \log n)$; sorting takes $O(n \log n)$ and traversing through the list of processes takes $O(n)$, so the algorithm runs in $O(n \log n)$ time overall.

(b) **Answer**: The claim is true; by construction of our algorithm, we reomve overlapping processes from the list if `status_check` is invoked at the end of a different process. So after we sort the list of processes by finish time, if we ignore the overlapped processes, then `status_check` is invoked at the end of each process, i.e. it will be invoked $k*$ times. Since our algorithm provides the optimal schedule, it is not possible to have another schedule that invokes `status_check` fewer than $k*$ times.

4.18 **Algorithm**: we will modify Dijkstra's algorithm as follows:

1. Let $S = \{s\}$ be the set of explored nodes and $d(u)$ be the distance from s to u, to be calculated for each node $u \in S$. Let $d(s) = 0$. In addition let $p(u)$ be the previous node on the shortest path to u.
2. While $S \neq V$, perform the following:

- Select a node $v \notin S$ with at least one edge from S for which $d'(v) = \min_{e=(u,v):u \in S} f_e(d(u))$ is as small as possible.
- Add v to S, define $d(v) = d'(v)$ and $p(v) = u$.

3. Using the stored $p(u)$ values, retrieve the shortest path by backtracking from the destination to the starting point. Return the shortest path.

Proof of correctness: By contradiction; suppose the algorithm returns a path that is not the shortest path. Then there must exist another path with a shorter distance than the returned path. However, since Dijkstra's algorithm always provides the shortest distance between the starting node and destination node, it is not possible to have a path with a shorter distance. Now suppose that $l(u)$ does not give us the shortest path to u; however, since $l(u)$ is only stored when we have a shortest path to u, backtracking using $l(u)$ will always provide us the shortest path. Therefore by contradiction the algorithm always returns the shortest path.
Time complexity: $O(n^2)$; this algorithm is the same as Dijkstra's algorithm ($O(n^2)$) with an extra backtrack step ($O(n)$) to retrieve the shortest path, so it is overall $O(n^2)$.

5.5 **Algorithm**:

1. Merge sort the lines by the slope with one modification: when comparing the slopes of two lines, also store the intersection point of the two lines. Store the intersections for each line in a separate list.
2. Now that we have a list of lines sorted by slope and a list of intersections for each line, initialize an empty list to store the visible lines.
3. Add the line with the least slope to the list of visible lines.
4. Traverse through the list of intersections for the list, find the point with the smallest x-value.
5. Repeat steps 3-4 for the line intersected by the current line. Continue until the line with the greatest slope is reached.
6. Return the list of visible lines.

Proof of correctness: By contradiction. Suppose the algorithm does not produce the correct list of visible lines. Then there is either a line in the list that is not visible, or that there is a visible line that is not included. We will examine the two scenarios separately:

- Suppose there is a line in the list that is not actually visible. By construction of our algorithm, it finds the lines that intersect with the previous line first. So if a line on the list is not actually visible, it would mean that either the previous line is not visible or the intersection is not the first one. Since we are starting from the line with the least slope, which is always visible, the first case is not possible. The second case is also not possible as we are specifically looking for the smallest x-values in the intersections.
- Now suppose that there is a visible line that is not included in the returned list. Again by construction of our algorithm, since we are always starting from the line with the least slope,

which is always visible, the only way for us to miss a line is to have used an intersection point that is not actually the first. By the same logic as the previous case, this is not possible.

Since the above two scenarios are not possible, the algorithm does indeed produce the correct list of visible lines.

Time complexity: $O(n \log n)$; merge sort takes $O(n \log n)$ and traversing through intersections take $O(n)$, so overall the complexity is $O(n \log n)$.

5.6 **Algorithm**:

1. Start from the root of the tree, compare the root with its two children. If the root has no children, return it as the local minimum.
2. If the root is smaller than both its children, return it as a local minimum.
3. Otherwise, repeat steps 1-2 on the child with a smaller value.

Proof of correctness: By contradiction. Suppose our algorithm does not return a local minimum, then the returned node must have a neighboring node that has a smaller value. Since we are using a binary tree, the neighboring node must be either the parent of the returned node or one of its children, if they exist. If the smaller node is the parent of the returned node, then our algorithm must have found it in step 1 where the two nodes were compared when the algorithm travelled to the parent. If the smaller node is a children of the returned node, our algorithm must have compared it again in step 1 and continued without returning the current node. Therefore neither scenario is possible and our algorithm does indeed return a local minimum.

Time complexity: $O(\log n)$ since we are halving our search area each iteration.

P5 Suppose you are given an array of sorted integers that has been circularly shifted k positions to the right. For example, taking $1, 3, 4, 5, 7$ and circularly shifting it 2 positions to the right you get $5, 7, 1, 3, 4$. Design an efficient algorithm for finding k.

Algorithm: Assuming there is no duplicate elements, we will use a modified binary search method as follows:

1. Go to the element in the middle of the list $arr[i]$, $i = \lfloor \frac{n}{2} \rfloor$ and compare it to its neighbors. Return 0 if there is only one element in the list.
   - If $arr[i-1] > arr[i]$: $arr[i]$ is the minimum. Return i as the shifted positions.
   - Else if $arr[i] > arr[i+1]$: $arr[i+1]$ is the minimum. Return $i+1$ as the shifted positions.
   - Continue otherwise.
2. Divide the list in half into two sublists. Repeat step 1 for the left list if $arr[i] < arr[n-1]$ where n is the length of the list, repeat step 1 for the right list otherwise.

Proof of correctness: By induction on the length of the list.

Base case: $n = 1$: There is only one element in the list, so there is no shift, i.e. $k = 0$. Our algorithm returns 0 at step 1 as expected.

Inductive step: Suppose the algorithm works for a list of n elements or fewer, we want to show that it will also work for a list of $n + 1$ elements.

- n is odd: Our algorithm will divide the list of $n+1$ elements into two sublists of smaller sizes, we simply need to make sure it checks the correct sublist. By step 2, if $arr[i] < arr[n-1]$,the the right sublist must be sorted. Therefore the minimum must be in the left sublist. We can proceed analogously otherwise.
- n is even: Let i_n denote the center index of a list with n elements; since $i_{n+1} = \lfloor \frac{n+1}{2} \rfloor = \lfloor \frac{n}{2} \rfloor = i_n$ here, our algorithm will proceed identically as the it would on a list with n elements. Therefore by inductive hypothesis the result is correct.

Therefore the algorithm is correct by induction.
Time complexity: $O(\log n)$ since binary search is $O(\log n)$.

P6 Consider d sorted array of integers each containing $n_1, n_2, \ldots, n_d$ numbers. The numbers n_i are arbitrary. The total number of all elements is n (sum of all n_i) Design an $O(n \log d)$ algorithm that merges all arrays into one sorted list. You may wish to use a data structure that we have discussed in class.
Algorithm:

1. For each two adjacent lists, combine them as follows. If d is odd, leave the last list untouched until the next iteration.
   - Create an empty list to store the combined list.
   - While at least one list is not empty, compare the first element of both lists.
   - Add the smaller element to the combined list and remove it from the original list.
2. Repeat until all the lists are combined.

Proof of correctness: By induction on the number of lists. We will show that the algorithm works for a list of length 2 as the base case, and by the inductive step that it will work for longer lists as well.
Base case: $d = 2$, the algorithm combines the two lists using step 1 exactly once. Since we are iterating through both lists simutaneously and placing the elements in the new list in order, the new list must be sorted.
Inductive step: Suppose the algorithm works for d lists, we will show that it will also work for $d+1$ lists. We can examine the following two case:

- d is odd ($d+1$ is even): Since the algorithm works for d lists, we can simply place a new list at the end and combine it with the lone list in the first iteration. The iterations after the first are identical to the iterations in combining d lists so by inductive hypothesis we will have the correct result.
- d is even ($d+1$ is odd): The new list would be placed by itself at the end, then combined with the rest in the second iteration and beyond. There must be an iteration i where $\frac{d+1}{2^i}$ becomes even, so it will work as shown in the previous case.

Therefore this algorithm is correct by induction.
Time complexity: $O(n \log d)$; combining two lists (sorting not needed) takes $O(n)$ time, then since we are combining them $(d/2 + d/4 + \ldots)$ times the complexity is $O(n \log d)$.