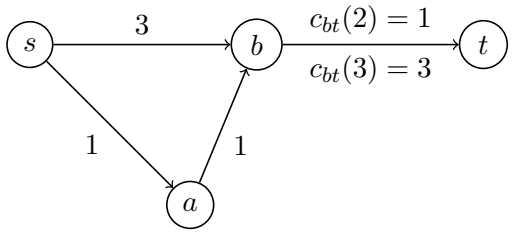
CS 577: Introduction to Algorithms 10/5/2015

Homework 4 Solutions (Review Problems)

Instructor: Dieter van Melkebeek TA: Bryce Sandlund

Problem 1

A counterexample is given in the following graph, where all edges are constant weight except for edge bt:



It can be seen that Dijkstra's algorithm will find the path $s \mapsto a \mapsto b \mapsto t$, of cost 5, but the optimal path from s to t is $s \mapsto b \mapsto t$, of cost 4.

The problem that Dijkstra's algorithm has with a graph of this type is that the cost going forward is dependent both on the cost of the edges used so far AND the number of edges used so far. Making the greedy choice that only the cost of the weights used so far matters results in an incorrect algorithm.

Problem 2

We can solve this problem using a greedy strategy. Our task is to place billboards so as to satisfy certain requirements, but keep the number of billboards that we place as small as possible. Think of the billboards as a resource we have access to, but whose use we wish to minimize. Our strategy is to delay making use of this resource as long as possible, only doing so when any further delay would cause us to break the constraints on billboard placement. More formally, we consider the billboard locations in the order they appear along the path; we place a billboard at location t if and only if failing to do so would make it impossible to ensure that all joggers see the required number of billboards, i.e., if and only if the collection consisting of the billboards we placed before t together with all billboards that come after t (but not the one at t), would violate the requirements for at least one jogger.

By construction, this strategy always produces a valid solution $\mathcal{G}$. We can use a "greedy stays ahead" argument to demonstrate that $\mathcal{G}$ is optimal.

Claim 1 *For every valid solution $\mathcal{S}$ and every location t, $\mathcal{G}$ places no more billboards up to t than $\mathcal{S}$ does.*

In particular, this claim implies that on the entire path, $\mathcal{G}$ places no more billboards than any other solution $\mathcal{S}$, i.e., $\mathcal{G}$ is optimal.

Our claim trivially holds for $t = 0$, since there are no locations where we could place a billboard before the first one. We now argue that if the claim holds for all t from 0 to i, then it must be true for $t = i + 1$ as well. If up to i the greedy solution $\mathcal{G}$ places strictly less billboards than $\mathcal{S}$ does or if $\mathcal{G}$ does not place a billboard at location $i + 1$, the claim definitely holds for $t = i + 1$. So, we only need to consider the case where $\mathcal{G}$ and $\mathcal{S}$ place the same number of billboards up to i and $\mathcal{G}$ places a billboard at $i + 1$. By construction $\mathcal{G}$ only does the latter if it is critical to ensuring that some jogger j sees the required number of billboards. That is, given the placement up to i, if we do not place a billboard at $i + 1$, it will be impossible to ensure that j sees enough billboards. Consider how the solution $\mathcal{S}$ places billboards along the portion of the path used by jogger j up to position i. Since we have that at each location up to i the number of billboards placed by $\mathcal{G}$ is no more than the number placed by $\mathcal{S}$, this is certainly true at the beginning of jogger j's interval. But then, since the number $\mathcal{G}$ and $\mathcal{S}$ place up to location i is equal, $\mathcal{S}$ can have placed no more billboards within j's interval up to position i than $\mathcal{G}$ did. So if it is critical in $\mathcal{G}$ that we place a billboard at location $i + 1$, it must also be critical that we do so in $\mathcal{S}$. Hence, $\mathcal{S}$ also places a billboard at position $i + 1$, which means our claim holds for $t = i + 1$.

In order to implement our greedy strategy we need to track when joggers become critical. Two observations lead to an initial implementation – that no jogger can possibly become critical until after his/her portion of the path begins, and that each time we add a billboard, we can push back the time when each jogger running past it becomes critical. Guided by this, we arrive at the following implementation. We can think of moving along the path from the start, placing billboards as we go. We process the joggers in order by their start positions, and maintain a list as we go of joggers whose portion of the path has begun, but who have not yet seen enough billboards. For each jogger j in the list, we keep track of when s/he becomes critical; that is, the first point c_j further along the path where we have to place a billboard if jogger j is to see the appropriate number. The value of c_j can be determined as follows. Suppose jogger j starts at location s_j, finishes at location f_j, and has seen b_j billboards so far. S/he needs to see $k_j = \min(k, f_j - s_j + 1)$ billboards in total so $c_j = f_j - k_j + 1 + b_j$. We maintain a priority queue of all active joggers j keyed on their critical time c_j.

We only need to consider positions along the path where some jogger starts or where a jogger is critical. In the former case, we first add every jogger j whose paths starts to our priority queue of active joggers using c_j as the key. Since j cannot have seen any billboards yet, we initialize c_j to $c_j = f_j - k_j + 1$. If a jogger has become critical in our list, we place a billboard at the current position, remove the joggers that are now fully satisfied, and update each remaining jogger in the list by increasing their critical value by 1 (as they have just seen a billboard). Note that this operation does not affect the ordering of the active joggers, and therefore can be done in time $O(1)$ per jogger. We then move on to the next location we need to consider.

As for the running time, the initial sort takes $O(n \log n)$ time. The only other major expense is outputting the billboards and maintaining our queue – we need to add and remove joggers, update the keys of the joggers in the list, and extract the minimum key value. Inserting a new jogger into the priority queue takes $O(\log n)$ time. As we already stated, the update cost when a jogger is critical is $O(1)$ per active jogger. As we remove a jogger after seeing at most k billboards, no jogger requires more than k updates. Hence, the cost for maintaining the list is $O(n \log n)$ for inserting the joggers into the priority queue, and $O(nk)$ for the other updates. Thus, the overall running time for this algorithm is $O(n \log n + nk)$.

To achieve a time bound of $O(n \log n)$, we need to avoid the nk term from our time bound.

Note that this is impossible if we output the billboards individually, as we may need as many as kn of them. However, once a jogger becomes critical, s/he remains critical for the rest of his/her portion of the path, so the greedy strategy places a billboard everywhere along that portion of the path; we can specify those billboards in the output as a range of locations rather than individual locations.

In order to improve our time bound, we still need to avoid updating all the relevant joggers in each step. Note that each time we update the critical positions in the list, we are increasing every one of them by the same amount. Instead of doing these updates for each jogger in the list separately, we can achieve the same effect by keeping track of a global offset and simply update that one value. In fact, since the offset increases by the same amount as the number b of billboards placed so far, we can use the latter quantity as the offset. That is, the key we use for an active jogger j in the priority queue is $c'_j = c_j - b$.

Our final algorithm works as follows. Sort the list of joggers in increasing order by the start of the interval on which their paths lie. Initialize $t = 0$, $b = 0$, and an empty priority queue for storing the joggers. Let p denote the value s_j of the next jogger j in the sorted list ($p = \infty$ if there is no such j) and c denote the value $c_j = c'_j + b$ where j denotes the jogger at the top of the priority queue ($c = \infty$ if the queue is empty). In each step we increase t to $\min(p, c)$.

- If $p \leq c$, we add the next jogger j in the sorted list to the priority queue, using the current value of b to compute his/her key $c'_j = f_j - k_j + 1 - b$.
- Otherwise, we consider the jogger j that is at the top of the priority queue. If $f_j < t$, this means j has already seen more billboards than required, and we can remove j from the priority queue and move on. Otherwise, we output the interval $[t, \min(f_j, p - 1)]$, and increase the value of b accordingly.

We continue this process until both the list and the priority queue are empty.

This algorithm yields the same result as our initial one, but avoids the nk term we saw in the time bound. Our initial sort of the joggers can be done in $O(n \log n)$ time. Since we can never have more than n elements in our priority queue, each time we add a jogger to it or remove one from it, we incur a cost of $O(\log n)$ time. Since each jogger is added to the priority queue (and removed from it) exactly once, all of the priority queue operations can be completed in $O(n \log n)$ time. Each time we output an interval $[t, \min(f_j, p - 1)]$, that operation can be uniquely attributed to one of the operations on the priority queue, namely the removal of j if $f_j \leq p - 1$, and the addition of a new jogger to the queue in the next step, otherwise. Since the additional computations we need to perform on each jogger upon addition to or removal from the priority queue require $O(1)$ time, this bound extends to the entire algorithm. Thus, our algorithm runs in $O(n \log n)$ time.