Problem Set 2 Solutions

Calvin Walker

Problem 1:

Let P be the priority queue used by Dijkstra's algorithim, where the entries of P are in increacing order of distance from s.

1. We start at vertex s, $P = [(5,e),(11,c),(14,a)]$
2. Next is vertex e, the length from s to e is 5, $P = [(8,f),(11,c),(14,a)]$
3. Next is vertex f, the length from s to f is 8, $P = [(10,c),(11,c),(14,a),(19,d),(20,t)]$
4. Next is vertex c, the length from s to c is 10, $P = [(6,e),(11,c),(12,a),(14,a),(19,d),(19,d),(20,t),(20,d)]$
5. We skip e as it has already been seen previously. $P = [(11,c),(12,a),(14,a),(19,d),(19,d),(20,t),(20,d)]$
6. We skip c as it has already been seen previously. $P = [(12,a),(14,a),(19,d),(19,d),(20,t),(20,d)]$
7. Next is vertex a, the length from s to a is 12, $P = [(14,a),(18,b),(19,d),(19,d),(20,t),(20,d),(20,b)]$
8. We skip a as it has already been seen previously. $P = [(18,b),(19,d),(19,d),(20,t),(20,d),(20,b)]$
9. Next is vertex b, the length from s to b is 18, $P = [(19,d),(19,d),(20,t),(20,t),(20,d),(20,b)]$
10. Next is vertex d, the length from s to d is 19, $P = [(19,d),(20,t),(20,t),(20,d),(20,b),(21,t)]$
11. We skip d as it has already been seen previously. $P = [(20,t),(20,t),(20,d),(20,b),(21,t)]$
12. Finally we reach vertex t, the length from s to t is 20 and the algorithm terminates.

The final shortest path from s to t is s,e,f,t. There is another shortest path of length 20: s,e,f,c,a,b,t.

Problem 2:

A Greedy algorithim to miminize the number of guards would be starting at time $t = 0$, to choose a guard who can work at time t from the set $\{i \in [n] \mid a_i \leq t \leq b_i\}$ such that b_i is maximized. Then set t equal to b_i (the start of the next ungaurded interval), and repeat the procedure for the new t until $b_i = T$.

Runtime: An ideal implementation would initially sort the guards in increacing order of their starting times in $O(n \log n)$ time. Then, a variable could be used to track the current position in the sorted array, and for each iteration, progress forward in the array until the optimal guard is found for unguarded time t. Since the array of intervals would only be iterated over once after it is sorted, the runtime is $O(n \log n)$.

Proof: Let $S = (i_1, \ldots, i_m)$ be the schedule given by the greedy algorithm, and define $t_j(S) = b_{i_j}$ to be the time at which the j'th job finishes. If $S' = (i_1, \ldots, i_{m'})$ is another valid schedule, we will first prove that for all $j \in \mathbb{N}$, $t_j(S) \geq t_j(S')$.

By induction on j. The base case $j = 1$ is trivial as the greedy algorithm always chooses the guard such that b_i is maximized, so $t_1(S) = b_{i_1} \geq b_{i'_1} = t_1(S')$. For the inductive step, assume that $t_k(S) \geq t_k(S')$. Observe that since both schedules must cover the interval $[0,T]$, $a_{i'_{k+1}} \leq b_{i'_k} \leq b_{i_k}$, so job i'_{k+1} is available to S. Since S always chooses the job with the latest completion time out of the available jobs, $t_{k+1}(S) = b_{i_{k+1}} \geq b_{i'_{k+1}} = t_{k+1}(S')$, and the inductive case holds.

Since S is ahead of S' for all $j \in \mathbb{N}$, $m \leq m'$, and the correctness of the greedy algorithim is proven.

Problem 3:

a) Let S be the schedule given by the greedy algorithm, and S' be a valid schedule. Consider a job $k \in S$ on the interval $[a_k, b_k]$. Since k is the shortest possible job within the interval $[a_k, b_k]$, as any shorter job would've been chosen first by the Greedy algorithm, it can overlap with at most two other jobs in S'. So S has at least half as many jobs as S'.

b) $I = \{[j, j+1] : j \in [2k]\} \cup \{[2j - \frac{1}{3}, 2j + \frac{1}{3}] : j \in [k]\}$, so the greedy algorithm will choose the k intervals from the latter set while the optimal solution is $2k$ intervals from the former.

Problem 4:

I would use two heaps: a max heap to store the smaller half of the data points, and a min heap to store the larger half of the data points. For simplicity, if there are an odd number of data points, we will store the extra one in the max heap.

1. `Insert(x)`:
   - Add element to max heap

- Push the top element of the max heap to the min heap
- If the size of the min heap is greater than the size of the max heap, push the top of the min heap to the max heap

This procedure will keep the heaps balanced using a constant number of $O(\log n)$ insertions and removals.

2. `Find Median`:
    - If the size of the max heap is larger, return the top element. Otherwise, return the average of both heaps. Since at most we just need the top of each heap, this operation takes $O(1)$ time.