CMSC 27200 - Problem Set 2

Sohini Banerjee

January 19, 2024

0

Collaborators: Parnika Saxena
Sources: Written Lecture Notes, Professor and TA Ed Responses

1

We outline the following steps to find the shortest paths from s to t. dv is the shortest distance from s to v. The last edge is the most recent edge used to reconstruct the shortest path from a particular node. The priority queue has the format (x, y, d), which represents an edge from x to y with a distance of d from s to y using that edge.

Step	S	v	dv	Last Edge	Priority Queue	Select
0	s	s	ds=0		(s, e, 5), (s, c, 11), (s, a, 14)	(s, e, 5)
1	s, e	e	ds=0, de=5	e: (s, e)	(e, f, 8), (s, c, 11), (s, a, 14)	(e, f, 8)
2	s, e, f	f	ds=0, de=5, df=8	e: (s, e), f: (e, f)	(f, c, 10), (s, c, 11), (s, a, 14), (f, d, 19), (f, t, 20)	(f, c, 10)
3	s, e, f, c	c	ds=0, de=5, df=8, dc=10	e: (s, e), f: (e, f), c: (f, c)	(c, e, 11), (s, c, 11), (c, a, 12), (s, a, 14), (c, d, 19), (f, d, 19), (f, t, 20)	(c, e, 11)
4	s, e, f, c	e	ds=0, de=5, df=8, dc=10	e: (s, e), f: (e, f), c: (f, c)	(s, c, 11), (c, a, 12), (s, a, 14), (c, d, 19), (f, d, 19), (f, t, 20)	(s, c, 11)
5	s, e, f, c	c	ds=0, de=5, df=8, dc=10	e: (s, e), f: (e, f), c: (f, c)	(c, a, 12), (s, a, 14), (c, d, 19), (f, d, 19), (f, t, 20)	(c, a, 12)
6	s, e, f, c, a	a	ds=0, de=5, df=8, dc=10, da=12	e: (s, e), f: (e, f), c: (f, c), a: (c, a)	(s, a, 14), (a, b, 18), (c, d, 19), (f, d, 19), (f, t, 20)	(s, a, 14)
7	s, e, f, c, a	a	ds=0, de=5, df=8, dc=10, da=12	e: (s, e), f: (e, f), c: (f, c), a: (c, a)	(a, b, 18), (c, d, 19), (f, d, 19), (f, t, 20)	(a, b, 18)
8	s, e, f, c, a, b	b	ds=0, de=5, df=8, dc=10, da=12, db=18	e: (s, e), f: (e, f), c: (f, c), a: (c, a), b: (a, b)	(c, d, 19), (f, d, 19), (b, c, 20), (f, t, 20), (b, t, 20), (b, d, 21)	(c, d, 19)
9	s, e, f, c, a, b, d	d	ds=0, de=5, df=8, dc=10, da=12, db=18, dd=19	e: (s, e), f: (e, f), c: (f, c), a: (c, a), b: (a, b), d: (c, d)	(f, d, 19), (b, c, 20), (f, t, 20), (b, t, 20), (b, d, 21), (d, t, 21)	(f, d, 19)
10	s, e, f, c, a, b, d	d	ds=0, de=5, df=8, dc=10, da=12, db=18, dd=19	e: (s, e), f: (e, f), c: (f, c), a: (c, a), b: (a, b), d: (c, d) and (f, d)	(b, c, 20), (f, t, 20), (b, t, 20), (b, d, 21), (d, t, 21)	(b, c, 20)
11	s, e, f, c, a, b, d	c	ds=0, de=5, df=8, dc=10, da=12, db=18, dd=19	e: (s, e), f: (e, f), c: (f, c), a: (c, a), b: (a, b), d: (c, d) and (f, d)	(f, t, 20), (b, t, 20), (b, d, 21), (d, t, 21)	(f, t, 20)
12	s, e, f, c, a, b, d, t	t	ds=0, de=5, df=8, dc=10, da=12, db=18, dd=19, dt=20	e: (s, e), f: (e, f), c: (f, c), a: (c, a), b: (a, b), d: (c, d) and (f, d), t: (f, t)	(b, t, 20), (b, d, 21), (d, t, 21)	(b, t, 20)
13	s, e, f, c, a, b, d, t	t	ds=0, de=5, df=8, dc=10, da=12, db=18, dd=19, dt=20	e: (s, e), f: (e, f), c: (f, c), a: (c, a), b: (a, b), d: (c, d) and (f, d), t: (f, t) and (b, t)	(b, d, 21), (d, t, 21)	END

The length of the shortest path to vertex v from s is as follows: $d_s = 0$, $d_e = 5$, $d_f = 8$, $d_c = 10$, $d_a = 12$, $d_b = 18$, $d_d = 19$, $d_t = 20$

We see that the shortest distance from s to t is length 20. t has two last edges, (f, t) and (b, t), which we can use to reconstruct the shortest paths:

- If we take (f, t) as the last edge of t, we get $f \to t$. Then, we take (e, f) as the last edge of f and get $e \to f \to t$. Then, we take (s, e) as the last edge of e and get $s \to e \to f \to t$.
- Similarly, if we take (b, t) as the last edge of t, we get $b \to t$. Then, we take (a, b) as the last edge of b and get $a \to b \to t$. Then, we take (c, a) as the last edge of a and get $c \to a \to b \to t$. Then, we take (f, c) as the last edge of c and get $f \to c \to a \to b \to t$. Then, we take (e, f) as the last edge of f and get $e \to f \to c \to a \to b \to t$. Then, we take (s, e) as the last edge of e and get $s \to e \to f \to c \to a \to b \to t$.

Thus, the shortest paths paths from s to t have length 20 and are:

- $s \to e \to f \to t$
- $s \to e \to f \to c \to a \to b \to t$

2

We will apply a greedy algorithm to select the minimum number of guards. We assume that $T > 0$ (Professor Potechin's office hours).

Stored Data:

- Remaining intervals of guards I: intervals of guards that have not yet been selected by greedy algorithm
- Earliest uncovered time *uncovered*: earliest time in range $[0, T]$ that has not yet been covered by a guard selected by greedy algorithm
- Selected guards *res*: intervals of guards that have been selected by greedy algorithm

Initialization:

- I: initialize to list of intervals of guards given
- *uncovered*: 0 because 0 is the earliest time in range $[0, T]$ that has not been covered by a guard
- *res*: initialize to empty list

Iterative Step: From the remaining intervals of guards I, we pick the guard with interval $[a_i, b_i]$ that covers *uncovered* and ends the latest. Then, we add $[a_i, b_i]$ to *res* and update *uncovered* to be b_i. We also remove $[a_i, b_i]$ from I. We repeat this procedure until our earliest uncovered time *uncovered* is equal to T or there are no intervals left to choose from. Then, we terminate the iterative step and return *res*.

Runtime: The greedy algorithm runs in $O(n^2)$ where n is the number of guards. First, we know that at every iteration, we will traverse through at most n intervals of guards, which takes $O(n)$. Second, there will be a maximum of n iterations because each iteration adds a new interval to the result, and since we cannot continue the iterations unless there are intervals left, this step must be $O(n)$. Thus, because for each iteration we take $O(n)$ to check the guards in I and complete $O(n)$ iterations, the time complexity is $O(n^2)$, which is polynomial time. Note that other aspects of the algorithm, such as adding an interval to *res* can be done in $O(1)$ time when appending to the end of a list and thus, do not affect the runtime.

Lemma 1. The greedy algorithm covers the entire range [0, T].
Proof. Assume for contradiction that there exists an interval $(t, t + \epsilon)$ in $[0, T]$ such that the entire range $(t, t + \epsilon)$ is unguarded. Furthermore, suppose we have guards covering until time t. For the interval $(t, t + \epsilon)$ to be unguarded, the next guard chosen must have a starting time of at least $t + \epsilon$. However, since we only have guards until t, the next earliest uncovered time exists in the range $(t, t + \epsilon)$ and thus, the guard must have a starting time in $(t, t + \epsilon)$. This produces a contradiction and therefore, there cannot exist an interval in $[0, T]$ such that it is unguarded, meaning the greedy algorithm covers the entire range $[0, T]$.

A more intuitive explanation is as follows. The greedy algorithm covers the entire range $[0, T]$ because the iterative step continues until T becomes the earliest uncovered time or there are no intervals left to choose. However, the latter is not possible because if there is some uncovered time t, the problem notes that there exists an interval to cover t. We know that for every t, either it will be covered by a guard selected to cover a point less than t, or t becomes the earliest uncovered time at some point, and we find a guard to cover it.

Python Code :

```
def find_intervals(intervals, T):
    I = intervals
    res = []
    uncovered = 0
    while uncovered < T:
        current_max_ending = -float('inf')
        next_interval = -1
        for i in range(len(I)):
            if I[i][0] <= uncovered:
                if I[i][1] > current_max_ending:
                    current_max_ending = I[i][1]
                    next_interval = i
        uncovered = current_max_ending
        res.append(I[next_interval])
        del I[next_interval]
    return res
```

Lemma 2. We define $t_j(S)$ as the ending time of the jth guard for some solution S. Assume the jth guard is defined as the jth guard to be hired (i.e., the jth guard when intervals of guards are sorted by starting time). If $S_{greedy} = (i_1, ..., i_m)$ is the set of intervals of guards given by the greedy algorithm, then for any other valid set of intervals of guards $S' = (i'_1, ..., i'_{m'})$, for all $j \epsilon [m]$, $t_j(S_{greedy}) \geq t_j(S')$.

Proof. We prove this lemma by induction.

- Base case $j = 1$: The first interval in S_{greedy} and S' must both cover 0. By the nature of the greedy algorithm, S_{greedy} selects the guard that covers 0 and has the latest end time. So, $t_1(S_{greedy}) \geq t_1(S')$ because the first guard in S' must also be available to S_{greedy}.
- Inductive hypothesis: Assume that $t_{j-1}(S_{greedy}) \geq t_{j-1}(S')$ for $j < m$. After the $(j-1)$th guard is selected by the greedy algorithm, the next earliest uncovered time is $x = t_{j-1}(S_{greedy}) + \epsilon$. The greedy algorithm selects the guard whose shift ends latest and covers x. We know that the first $j-1$ guards of S_{greedy} did not cover x by definition, and due to the inductive hypothesis, the first $j-1$ guards of S' cover no later than the first $j-1$ guards of S_{greedy}, meaning the first $j-1$ guards of S' also could not have covered x. Now, assume for contradiction that $t_j(S_{greedy}) < t_j(S')$. This implies that the first j guards in S_{greedy} covered x and ends earlier than $t_j(S')$, meaning the first j guards in S' also covered x. However, since we established that the first $j-1$ guards of S' did not cover x due to inductive hypothesis, we know that the jth guard of S' covers x. In this case, the jth guard of S' ends later than the jth guard of S_{greedy} but would be available to S_{greedy}, so the greedy algorithm would have selected it, producing a contradiction. Thus, we know that $t_j(S_{greedy}) \geq t_j(S')$ for all $j \epsilon [m]$. For any $j > m$, our greedy algorithm is optimal because it did not need more than m guards, whereas S' did.

Therefore, if the greedy algorithm gives a valid set of guards $S_{greedy} = (i_1, ..., i_m)$, then for any other valid set of guards $S' = (i'_1, ..., i'_{m'})$, $m \leq m'$. This is because S_{greedy} needs m guards to cover $[0, T]$ and any other solution S' is no further than S_{greedy} for any jth guard up to m. This means, either S' is as optimal as the greedy solution S_{greedy} or S' needs more than m intervals to cover the range, in which case the greedy solution is more optimal.

3a

Let our algorithm solution $S_{greedy} = (i_1, ..., i_k)$ and optimal solution $S^* = (i'_1, ..., i'_{k'})$.

Lemma 1. Each job in S_{greedy} overlaps with at most two jobs in S^*.
Proof. Assume there exists some job m in S_{greedy} that overlaps with three jobs in S^*, denoted x, y, and z. A valid solution does not permit overlapping jobs, so we know that jobs x, y, and z do not intersect. Let jobs x, y, and z be in sorted order by starting time such that $a_x \leq b_x \leq a_y \leq b_y \leq a_z \leq b_z$. Since job m overlaps with intervals x and z, we know based on the intersection definition provided by the problem that:

- $b_x > a_m$ and $b_m > a_x$
- $b_z > a_m$ and $b_m > a_z$

Since $a_m < b_x$ and $b_x \leq a_y$, we know that $a_m < a_y$. Similarly, since $b_m > a_z$ and $a_z \geq b_y$, we know that $b_m > b_y$. However, if $a_y > a_m$ and $b_y < b_m$, then $|b_m - a_m| > |b_y - a_y|$. In other words, job y is contained in job m. In our greedy algorithm, we select the shortest intervals first, meaning job y would be selected before job m. Furthermore, after selecting job y, job m would have been removed from the set of available jobs I because they overlap. Thus, it is impossible for job m to contain job y with S_{greedy} and S^*. However, we previously claimed that job y must be contained within job m for job m to overlap with three jobs in S^*. This is a contradiction. Thus, each job in S_{greedy} overlaps with at most two jobs in S^*.

Lemma 2. Each job in S^* must overlap with at least one job in S_{greedy}.
Proof. Assume for contradiction that there exists a job p in S^* that does not overlap with a job in S_{greedy} and therefore cannot be in S_{greedy}. This means that for the addition of every new interval in S_{greedy}, job p was not removed from the list of available jobs I. Thus, at some point, job p must have become the shortest job available in I and added to S_{greedy}. This is a contradiction because we assumed that job p cannot be in S_{greedy}.

Lemma 3. The solution S_{greedy} provides a $\frac{1}{2}$-approximation to the optimal solution.
Proof. By Lemma 1, we know that each job in S_{greedy} can overlap with a maximum of two jobs in S^*. To demonstrate the approximation, we must minimize the ratio between $|S_{greedy}|$ and $|S^*|$ and find its lower bound. If we consider the worst case scenario where each interval in S_{greedy} overlaps with two intervals in S^*, we get that i_1 overlaps with two intervals in S^*, i_2 overlaps with two intervals in S^*, and so on. By Lemma 2, every interval in S^* must overlap with an interval in S_{greedy}, so to minimize the ratio between $|S_{greedy}|$ and $|S^*|$, we construct intervals where each interval in S_{greedy} overlaps with distinct sets of two intervals in S^*. In other words, i_1 overlaps with j_1 and j_2, i_2 overlaps with j_3 and j_4, and so on. This makes $\frac{|S_{greedy}|}{|S^*|} = \frac{1}{2}$.

3b

The following Python program produces a list of intervals for a given k. These are the outputs for some values of k:

- $k = 0 : []$
- $k = 1 : [[-3, 1], [0, 3], [2, 6]]$
- $k = 2 : [[-3, 1], [0, 3], [2, 6], [7, 11], [10, 13], [12, 16]]$

Python Code :

```
def find_intervals(k):
    intervals = []
    for i in range(k):
        a, b = 10 * i, 10 * i + 3
        c, d = 10 * i - 3, (10 * i - 3) + 4
        e, f = 10 * i + 2, (10 * i + 2) + 4
        intervals.append([c, d])
        intervals.append([a, b])
        intervals.append([e, f])
    return intervals
```

The intuition is that for each k, we add a set of three jobs to the output for $k-1$. The format of the three jobs can be explained as follows. If we sort the jobs by starting time, we get jobs x, y, and z. Job y overlaps with both job x and z. Job x and job z only overlap with job y. Also, job y has length 3 and job x and z have length 4, so job y will always be selected first, with x and z being discarded after since they overlap with y. For each new k, we add a set of jobs that do not overlap with the previous set, so this ratio of one job y to two jobs x and z remain.

4

We will have a min heap and max heap to determine the median of numbers. Retrieving the root of a min or max heap takes $O(1)$ time. Inserting into a min or max heap takes $O(\log n)$ time. Also, deleting the root of a min or max heap takes $O(\log n)$ time.

Stored Data: We will keep a min heap and max heap to store the numbers inserted into the data structure. The numbers added so far will be in one of the two heaps.

Initialization: We start with an empty min heap and max heap, both stored as arrays. We will use the implementation described in the lecture notes.

Operations:

- **Insert** - We can break this into two steps, adding x to one of the two heaps and balancing the heaps until their sizes differ by no more than one.
  - Add - We can break this into four cases:
    - min heap size $= 0$ and max heap size $= 0$: add x to min heap
    - min heap size $= 0$ and max heap size $\neq 0$: add x to max heap
    - min heap size $\neq 0$ and max heap size $= 0$: add x to min heap
    - min heap size $\neq 0$ and max heap size $\neq 0$:
      - $x \leq$ root of max heap: add x to max heap
      - $x \geq$ root of min heap: add x to min heap
      - root of max heap $< x <$ root of min heap: add x to min heap
  - Balance - We transfer elements between heaps until the size of the heaps do not differ by more than one.
    - while min heap size $>$ max heap size $+ 1$: add root of min heap to max heap and remove root of min heap
    - while max heap size $>$ min heap size $+ 1$: add root of max heap to min heap and remove root of max heap
- **Median** - We can break this into three cases:
  - min heap size $=$ max heap size: return the average of the root of the min heap and root of the max heap
  - min heap size $<$ max heap size: return the root of the max heap
  - min heap size $>$ max heap size: return the root of the min heap

To explain how the operations are implemented, we consider insert and median. For insert, if both heaps are empty, we simply add to the min heap. If one heap is empty and the other is not, we add to the non-empty heap and subsequently use the balance functionality at the end of insert to adjust the sizes. Finally, if both heaps are non-empty, we add to the max heap or min heap if the value is not between the roots of both heaps. However, if the value is in between, we simply add to the min heap and let the balance functionality take care of size adjustments. For the median operation, if the heap sizes differ, we know the median element is in the larger sized heap.

If we were to take all x inserted, the max heap would store the smallest half and min heap would store the largest half of numbers. We also ensure that the heaps are almost identical in size (allowed to differ by one when there are an odd number of elements inserted at some point). This allows us to access the median of the datasets by retrieving the largest element in the lower half and/or smallest element in the upper half at $O(1)$ time.