Compsci 330 Design and Analysis of Algorithms
Assignment 7, Spring 2024 Duke University

Example Solutions

Due Date: Thursday, March 21, 2024

How to Do Homework. We recommend the following three step process for homework to help you learn and prepare for exams.

1. Give yourself 15-20 minutes per problem to try to solve on your own, without help or external materials, as if you were taking an exam. Try to brainstorm and sketch the algorithm for applied problems. Don't try to type anything yet.
2. After a break, review your answers. Lookup resources or get help (from peers, office hours, Ed discussion, etc.) about problems you weren't sure about.
3. Rework the problems, fill in the details, and typeset your final solutions.

Typesetting and Submission. Your solutions should be typed and submitted as a single pdf on Gradescope. Handwritten solutions or pdf files that cannot be opened will not be graded. LaTeX[1] is preferred but not required. You must mark the locations of your solutions to individual problems on Gradescope as explained in the documentation. Any applied problems will request that you submit code separately on Gradescope to be autograded.

Writing Expectations. If you are asked to provide an algorithm, you should clearly and unambiguously define every step of the procedure as a combination of precise sentences in plain English or pseudocode. If you are asked to explain your algorithm, its runtime complexity, or argue for its correctness, your written answers should be clear, concise, and should show your work. Do not skip details but do not write paragraphs where a sentence suffices.

Collaboration and Internet. If you wish, you can work with a single partner (that is, in groups of 2), in which case you should submit a single solution as a group on gradescope. You can use the internet, but looking up solutions or using large language models is unlikely to help you prepare for exams. See the course policies webpage for more details.

Grading. Theory problems will be graded by TAs on an S/U scale (for each sub-problem). Applied problems typically have a separate autograder where you can see your score. The lowest scoring problem is dropped. See the course assignments webpage for more details.

[1] If you are new to LaTeX, you can download it for free at latex-project.org or you can use the popular and free (for a personal account) cloud-editor overleaf.com. We also recommend overleaf.com/learn for tutorials and reference.

Problem 1 (Interval Coloring). Let X be a set of n intervals on the real line labeled $1, \ldots, n$. A *proper coloring* of X assigns a color to each interval, so that any two intervals that overlap are assigned different colors.

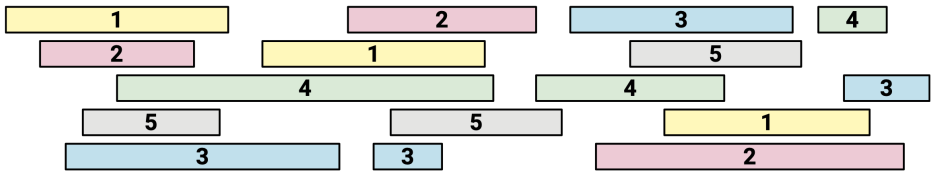

Figure 1: A set of intervals with a proper coloring using five colors.

Describe an $O(n \log(n))$ runtime algorithm to compute the minimum number of colors needed to properly color X. Assume that your input consists of two arrays $L[1], \ldots, L[n]$ and $R[1], \ldots, R[n]$ representing the left and right endpoints of the intervals. For simplicity you may assume that no endpoints are equal for different intervals. Prove the correctness of the algorithm and analyze its runtime complexity.

Solution 1. We use a greedy algorithm that scans the intervals from left to right. When a new interval is encountered at its left endpoint, we assign it an available color. Whenever we finish an interval at the right endpoint, we release its color for later use. To implement this strategy, we create a list of events, either left or right endpoints, and sort them by $L[i]$ or $R[i]$, also keeping track of the interval index and whether the event is a left or right endpoint. We also have an array to keep track of assigned colors and a queue to keep track of available colors, initially empty.

```
procedure COLOR($L, R, n$)
    Create an empty list $E$
    for $i = 1$ to $n$ do
        $E$.append(($i, L[i], T$))
        $E$.append(($i, R[i], F$))
    Sort $E$ from least to greatest by the second element of the tuples.
    Let $C$ be an empty length $n$ array
    Let $Q$ be an empty queue
    $nc = 0$
    for ($i, t, left$) in $E$ do
        if $left = T$ then
            if $Q$ is empty then
                $nc = nc + 1$
                $C[i] = nc$
            else
                $C[i] = Q$.remove()
        else
            $Q$.add($C[i]$)
    return $nc$
```

Correctness. To argue that the algorithm returns the *minimum* number of colors necessary, we argue by induction on the left endpoints as ordered in the algorithm from least to greatest.

We argue the the number of active colors (not in the queue) after processing the k'th left endpoint equals the number of intervals that overlap at that point. This implies that the maximum total number of colors used by the algorithm is the maximum number of overlapping intervals at any point, which is necessary for any solution by the proper coloring requirement.

In the base case, the algorithm uses 1 color for the leftmost endpoint. For the inductive hypothesis (IH), suppose the claim is true up to left endpoint k. At left endpoint $k+1$, the algorithm has released any colors from intervals that have ended prior to this point, and adds one for the beginning of the new interval.

Runtime. Creating and sorting the event list can be done in $O(n \log(n))$ time, for example with mergesort, since there are $2n$ entries. This dominates the runtime since the subsequent loop over events takes just linear time.

Problem 2 (Spanning Tree Modifications). Suppose we are given both an undirected graph $G = (V, E)$ with weighted edges (let $w(u, v)$ be the weight of the edge between vertices u and v) and a minimum spanning tree T of G. As usual, let $n = |V|$ and $m = |E|$ denote the number of vertices and edges respectively.

(a) Describe a linear runtime algorithm to update the minimum spanning tree when the weight of a single edge (u, v) not in T is decreased. Briefly explain why the algorithm is correct and analyze its runtime complexity.

(b) Describe a linear runtime algorithm to update the minimum spanning tree when the weight of a single edge (u, v) in T is increased. Briefly explain why the algorithm is correct and analyze its runtime complexity.

You may refer without proof to the optimality principle discussed in class: For any two disconnected components whose edges are in a mininmum spanning tree (MST), the least weight edge between them is also in a a MST.

Solution 2.

(a) *Algorithm.* Use a depth-first search on T to find the unique path $P_{u \to v}$ from u to v on T. $P_{u \to v}$ and the edge (u, v) form a cycle. Let e_{max} be the edge with the maximum weight in the cycle. If $e_{max} = (u, v)$ then T does not change. Otherwise, swap e_{max} out and include (u, v) in T.

Correctness. We argue that no MST could include a path between u and v that includes an edge (x, y) with $w(x, y) > w(u, v)$. Suppose for a contradiction it does. Then we could swap (x, y) for (u, v) and maintain connectivity but reduce total cost.

Therefore, e_{max} in our algorithm, if different from (u, v), certainly cannot be in the MST. After removing e_{max}, observe that the remaining two disconnected components are still optimal, so the overall MST can be found by adding the minimum weight edge connecting these components. Since $w(u, v) < w(e_{max})$ and e_{max} was originally in the MST, this must be the edge (u, v).

Runtime. Finding $P_{u \to v}$ with a DFS on T takes $O(n)$ time. We can find the maximum weight edge among the cycle in $O(n)$ time. The overall runtime is $O(n)$.

(b) *Algorithm.* Remove (u, v) from T, leaving two disconnected components; call the vertices of these components X and Y, each of which can be computed by a single depth-first search from u and v respectively. Search over all edges and identify the edge with minimum weight with one endpoint in X and one in Y. Add this edge in place of (u, v).

Correctness. The two components of T after removing (u, v) form a subset of the MST. This can be seen by contradiction: If not, then (u, v) plus the minimum spanning trees on X and Y would have had less total cost than T. Therefore, as we add the minimum weight edge between these components, the resulting spanning tree is minimum.

Runtime. X and Y can be computed with DFS on T after removing (u, v) in $O(n)$ time. Searching over all edges takes $O(m)$ time. The overall runtime is thus linear, $O(n + m)$.

Problem 3 (Interval Covers). Let X be a set of n intervals on the real line labeled $1, \dots, n$. We say that a subset of intervals $Y \subseteq X$ *covers* X if the union of all intervals in Y is equal to the union of all intervals in X. The *size* of a cover is just the number of intervals.

Describe an $O(n \log(n))$ runtime algorithm to compute the smallest cover of X. Assume that your input consists of two arrays $L[1], \dots, L[n]$ and $R[1], \dots, R[n]$ representing the left and right endpoints of the intervals. For simplicity you may assume that no endpoints are equal for different intervals. Prove the correctness of the algorithm and analyze its runtime complexity.

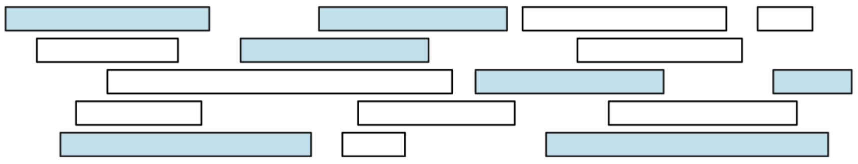

Figure 2: A set of intervals, with a cover (shaded) of size 7 (not necessarily the optimal cover).

Solution 3. We use a greedy algorithm that scans intervals from left to right. On encountering a left endpoint, if we have no chosen interval overlapping at this point, we must choose this interval. Otherwise, we search among all intervals with left endpoint before the right endpoint of the chosen interval overlapping this point and select such interval with farthest right endpoint.

```
procedure COVER($L, R, n$)
    Let $Q$ be a queue of $(L[i], R[i], i)$ tuples ordered from least to greatest $L[i]$
    Let $C$ be an empty list
    $r = -\infty$
    while $Q$ not empty do
        $(\ell_i, r_i, i) = Q$.remove()
        if $\ell_i > r$ then
            $C$.append($i$)
            $r = r_i$
        else if $r_i > r$ then
            while next in $Q$ with left endpoint at most $r$ do
                $(\ell_j, r_j, j) = Q$.remove()
                if $r_j > r_i$ then
                    $(\ell_i, r_i, i) = (\ell_j, r_j, j)$
            $C$.append($i$)
            $r = r_i$
        else
            Continue
    return $C$
```

Correctness. Let $x_1, x_2, \dots, x_m$ be the intervals chosen by the greedy algorithm sorted from least to greatest left endpoint. We will argue by induction that for all $1 \leq k \leq m$ there is an optimal schedule that chooses $x_1, x_2, \dots, x_k$.

For the base case, it is clearly necessary to choose the interval with the least left endpoint in order to cover the union of all intervals. For the inductive hypothesis (IH), suppose there is an optimal solution of the form $x_1, x_2, \dots, x_{k-1}, y_k, y_{k+1}, \dots, y_{m^*}$ where m^* is the number of intervals used in this optimal solution and again the intervals are ordered from least to greatest left endpoint. There are two cases.

(i) If $L[x_k] > R[x_{k-1}]$ then $y_k = x_k$, otherwise the optimal solution does not cover all of x_k using only intervals with greater left endpoints.

(ii) Otherwise $L[x_k] < R[x_{k-1}]$, and we can further observe that $R[x_{k-1}] < R[x_k]$ from line 10 of the algorithm. In order to cover x_k, it must be that $L[y_k] < R[x_{k-1}]$. The algorithm chooses the interval with the greatest R, so $R[x_k] > R[y_k]$. Then $x_1, x_2, \ldots, x_{k-1}, x_k, y_{k+1}, \ldots, y_{m^*}$ must also be a valid optimal solution.

Runtime. The algorithm takes $O(n \log(n))$ to sort the n elements initially with, for example, mergesort. Thereafter, every iteration of the (nested) while loop takes constant time and removes an element from Q which initially has n elements, so a total of $O(n)$ time. The runtime is therefore dominated by the sorting, $O(n \log(n))$.

Problem 4 (Applied). You are a network engineer and your company has a bunch of computers that need to be wired up to a network. There are also routers scattered around the compound, but you don't need to wire them to the network. They can be used if you desire. However, none of the cabling has been installed. There is a list of potential paths that you can install cables on. Some paths are between computers, some are between routers, and some are between computers and routers. Each path has a corresponding cost associated with it and can be thought of as an edge in a graph.

You will be provided with a list of potential paths (undirected edges) with corresponding non-negative costs and a list of vertices that are computers. Your algorithm should then **return a list of edges that creates a connected graph of all computers that may include as many routers as you wish and should use at most twice the cost of the optimal solution.** The cost is defined as the sum of the costs of the edges you output and our goal is to minimize the cost.

Your solution will be tested against a reference solution and should be efficient for sparse and dense graphs. We will test for a factor of 2 from the optimal solution on small examples, and for larger graphs you simply have to get with two times the cost of the reference solution.

Hint: Think about how you can combine the ideas from all-pairs shortest paths and minimum spanning trees in your solution.

Language-specific details follow. You can use whichever of Python or Java you prefer. You will receive automatic feedback when submitting, and you can resubmit as many times as you like up to the deadline. **NOTE:** Unlike the theory problems, the applied problem grade **is the raw score shown on Gradescope**. See the course assignments webpage for more details.

- **Python.** You should submit a file called `network.py` to the Gradescope item "Assignment 5 - Applied (Python)." The file should define (at least) a top level function `network` that looks like:
  - `def network(edge:  list[(s:int, d:int, c:int)], computer:  list[int])`

  where (s, d, t) is a potential path connecting machine s and d with non-negative cost c.
- **Java.** You should submit a file called `Network.java` to the Gradescope item "Assignment 5 - Applied (Java)." The file should define (at least) a top level function `network` that takes in a 2D int array `edge[][]` where `edge[i]` is the array $\{s, d, c\}$ or an edge connecting s to d (and d to s) with cost c. The method `network` looks like:
  - `public List<int[]> network(int[][] edges, int[] computer)`

  where `int[][] edges` is the undirected edges of the graph and `int[] computer` is the computers

Solution 4. We do not provide detailed code solutions, but we briefly explain the idea of the algorithm here. If we had to connect all of the computers and all of the routers then this would simply be the minimum spanning tree problem. Instead, if computers and routers are both vertices in the graph, we only need to ensure that we connect the subset of vertices corresponding to computers.

A greedy approach to this problem is to compute, for all pairs of computers, the shortest path between them. This can be done with several runs of Dijkstra's algorithm (preferable for a sparse network) or the Floyd-Warshall algorithm (preferable for a dense network). Then, define a new graph only on the computers with edges corresponding to the shortest path costs. Compute the minimum spanning tree on this graph and choose all of the edges from the corresponding shortest paths in the original graph.

Though you are not required to provide a proof, the above algorithm is actually guaranteed to find at most a 2-approximation. However, it is likely that you may be able to implement alternative greedy algorithms that also work to meet the approximation bound in this problem.