Computer Science 180, Homework 5

Michael Wu
UID: 404751542

February 15th, 2018

Chapter 4, Problem 25

Our set of points P and our distance function $d(p_i, p_j)$ forms a weighted, complete, undirected graph $G = (P, E)$ where the edge $e_{ij} = (p_i, p_j) \in E$ has the positive, non-zero weight $d(p_i, p_j)$. Then we can apply a form of Kruskal's Algorithm, where we cluster points with the smallest edge weights iteratively, to generate a hierarchical metric τ. We are motivated by the fact that if we were to add two points to τ with an edge weight that was not the smallest, we could potentially generate an inconsistent metric. This is because the parent of each node in τ must have a height that is greater than its children.

Begin our algorithm by having n leaf nodes v_i in τ, each assigned to the point p_i. We run Kruskal's Algorithm on G, and add the parent node v_{ab} that has v_a and v_b as its children when we add the first edge (p_a, p_b) in Kruskal's Algorithm. Let v_{ab} have the height $d(p_a, p_b)$. This node is the root of the cluster consisting of p_a connected to p_b. We continue with Kruskal's Algorithm, and each time when we join two clusters we create a parent node v that has the roots of the clusters as its children. We set the height of v to the length l of the edge that joins the clusters. Because Kruskal's Algorithm adds edges in increasing length order, this forms a metric τ that is consistent with d because the height of all children must be less than or equal to their parent. Additionally, the distance between the closest two points p_i in the cluster C_i and p_j in the cluster C_j that are joined together have the metric $\tau(p_i, p_j) = d(p_i, p_j)$, and every other pair of points between the clusters has a distance greater than this. Because Kruskal's Algorithm must generate a minimum spanning tree, we end up with a connected tree τ that forms our metric.

We prove that any other hierarchical metric τ' consistent with d has $\tau'(p_i, p_j) \leq \tau(p_i, p_j)$ for any p_i and p_j by contradiction. Assume that some τ' consistent with d exists such that $\tau'(p_i, p_j) > \tau(p_i, p_j)$ for some p_i and p_j. Then there is some least common ancestor v with children that are the subtrees C_i that contains p_i and and C_j that contains p_j. v has height $\tau'(p_i, p_j) = h_v$, and so the inequality $d(p_a, p_b) \geq h_v > \tau(p_i, p_j)$ holds for any pair of points $p_a \in C_i$ and $p_b \notin C_i$. This is because a common ancestor for p_a and p_b must have a height greater than or equal to h_v.

Consider a path from p_i to p_j in the minimum spanning tree of G. This path must go from some cluster containing p_i to some cluster containing p_j, which can be represented by the disjoint subtrees C_i and C_j, respectively. So this path must contain a point not in C_i. Let p be the first point along this path not in C_i, and p_C be the point in C_i that connects to p. Then $d(p_C, p) \geq h_v > \tau(p_i, p_j)$ since these two points can only share a common ancestor with height greater than or equal to h_v.

But in our formulation of τ, Kruskal's Algorithm adds edges in order of increasing length. Thus $\tau(p_i, p_j) \geq d(p_C, p)$ because each edge in the minimum spanning tree must be greater than any edge added before it. $d(p_C, p)$ is an edge in the path between p_i and p_j. So the edge (p_C, p) must be added before or during when we join the clusters containing p_i and p_j in Kruskal's Algorithm, so (p_C, p) must be smaller than or equal to $\tau(p_i, p_j)$. This is because $\tau(p_i, p_j)$ is the length of the longest edge along the path from p_i to p_j. But we also have $d(p_C, p) \geq h_v > \tau(p_i, p_j)$, which is a contradiction. Thus no such τ' exists.

Chapter 4, Problem 28

First we assign weights $X = 1$ and $Y = 2$ to each edge labeled X or Y, respectively. Then we run a minimum spanning tree algorithm that takes $O(n^2)$ time, which produces a spanning tree $T_{\max} = (V, E_{\max})$ that contains the maximum number of X edges. Let this maximum be a. Then we assign weights $X = 2$ and $Y = 1$ to each edge labeled X or Y, respectively, and run a minimum spanning tree algorithm to produce a spanning tree $T_{\min} = (V, E_{\min})$ that contains the minimum number of X edges. Let this minimum be b. Clearly if $k < b$ or $k > a$, no spanning tree with k edges labeled X can exist. We can end our algorithm here if that is the case.

If $b \leq k \leq a$, we can find a spanning tree with k edges. We do this as follows. Consider the set $E_{\max} - E_{\min}$. They differ by $d = |E_{\max} - E_{\min}|$ edges. We can create a new spanning tree $T_{\min+1} = (V, E_{\min+1})$ by choosing an edge $e \in E_{\max} - E_{\min}$ and adding it to $E' = E_{\min} \cup \{e\}$. Then the graph $G' = (V, E')$ has a cycle that contains an edge $e' \in E_{\min} - E_{\max}$, which we remove to create $E_{\min+1} = E_{\min} \cup \{e\} - \{e'\}$. Then $|E_{\max} - E_{\min+1}| = d - 1$, as $E_{\min+1}$ contains one more edge in $E_{\max}$ than $E_{\min}$. By induction, we can generate $T_{\min+i}$ by adding one edge in $E_{\max}$ and removing one in $E_{\min+i-1}$ until no more edges differ. Each step, the one edge that can differ either adds an X edge, or does not. Therefore this process generates at least one tree for every value between b and a number of X edges, since it must reach $T_{\max}$ from $T_{\min}$. So eventually this process must generate some T that has k edges labeled X, where $b \leq k \leq a$. We can return this tree T to finish our algorithm.

This last part is $O(n^2)$. This is because there are $n-1$ edges in a spanning tree with n nodes, and our algorithm requires $O(n)$ time to check for cycles and compute the differences between two sets of edges. Multiplying these together yields $O(n^2)$ time for the whole algorithm including the steps to find the minimum spanning trees, so we have an overall polynomial runtime.

Chapter 6, Problem 4

a) Let $M = 10$. Then the following operating cost table

	Month 1	Month 2
NY	1	3
SF	2	1

results in the given algorithm returning {NY, SF} for a cost of 12. The correct plan to minimize cost should be {SF, SF} for a cost of 3.

b) Let $M = 10$. Then the following operating cost table

	Month 1	Month 2	Month 3	Month 4
NY	1	100	1	100
SF	100	1	100	1

results in the only optimal plan {NY, SF, NY, SF} for a cost of 34. Every optimal plan must move at least 3 times, because the cost of moving is outweighed by the high cost of staying.

c) Let $C(a, b)$ be the cost of operating in city a during month b. Construct two arrays $N(i, j)$ and $S(i, j)$ that return the cost of operating in New York or San Francisco, respectively, from months i to j using the following algorithm.

```
for x from 1 to n
   for y from x to n
      if (y==x)
         N(x,y)=C(NY,y)
         S(x,y)=C(SF,y)
      else
         N(x,y)=N(x,y-1)+C(NY,y)
         S(x,y)=S(x,y-1)+C(SF,y)
return N, S
```

This step runs in $O(n^2)$ time, as it loops twice over n. Then we define a function $D(i)$ that gives the minimum operating cost after i months. We also define a function $E(i)$ that gives the end city after executing a minimum operating plan for i months. Because there may be multiple ways to plan optimally such that either cities may be the end cities, $E(i) \in \{\text{NY}, \text{SF}, \text{Either}\}$.

Let M be the cost of moving. Then we use the following algorithm to find the optimal cost after n months.

```
E(0)=Either
D(0)=0
for x from 1 to n
   minCost = infinity
   endCity = Either
   for y from 0 to x-1
      nyMoveCost=0
      sfMoveCost=0
      if (E(y)==NY)
         sfMoveCost=M
      else if (E(y)==SF)
         nyMoveCost=M
      if (D(y)+N(y+1,x)+nyMoveCost<minCost)
         minCost=D(y)+N(y+1,x)+nyMoveCost
         endCity=NY
      if (D(y)+S(y+1,x)+sfMoveCost<minCost)
         minCost=D(y)+S(y+1,x)+sfMoveCost
         endCity=SF
      if ((D(y)+N(y+1,x)+nyMoveCost==minCost && endCity==SF)||
          (D(y)+S(y+1,x)+sfMoveCost==minCost && endCity==NY))
         endCity=Either
   D(x)=minCost
   SE(x)=endCity
return D(n)
```

This algorithm is $O(n^2)$ because it loops over n twice. It is correct because it iteratively generates the next minimum cost for months $1, \ldots, n$ using a recurrence relation

$$D(x) = \begin{cases} 0 & x = 0 \\ \min\limits_{y=0}^{x-1}(D(y) + N(y+1,x) + M_N, D(y) + S(y+1,x) + M_S) & x \geq 1 \end{cases}$$

where M_N and M_S are equal to M if the optimal plan $D(y)$ requires the company to operate in SF or NY, respectively. This is valid because given optimal plan P of length n months, if there are $m \leq n$ months before the final move, the subset of P from months 1 to m must be an optimal plan P' for the first m months as well. If P' was not optimal, we could replace it with a lower cost plan, which is a contradiction as this means we can reduce the cost of P by rearranging it. Therefore it makes sense to only calculate the cost after a final move and minimize this value iteratively. We check if a move is necessary by keeping track of whether or not a minimum cost can be achieved while ending in a particular city using $E(x)$.

Chapter 6, Problem 6

First we wish to generate an array $S(i,j)$ of squared slacks for every line containing the words from word i to j. The slack of such a line is given by

$$L - \sum_{a=i}^{j-1}(c_a + 1) - c_j = L - j + i - \sum_{a=i}^{j} c_a$$

Let our array be

$$S(i,j) = \begin{cases} \infty & \text{if } i < j \\ \infty & \text{if } L - j + i - \sum_{a=i}^{j} c_i < 0 \\ (L - j + i - \sum_{a=i}^{j} c_a)^2 & \text{otherwise} \end{cases}$$

Let $C(n)$ be the character count c_n. We begin by calculating all $R(i,j) = \sum_{a=i}^{j} c_a$ as follows

```
for i from 1 to n
   for j from i to n
      if (j==i)
         R(i,j)=C(j)
      else
         R(i,j)=R(i,j-1)+C(j)
return R
```

This algorithm is $O(n^2)$ as it loops over n twice. Then we can generate $S(i,j)$ as follows

```
for i from 1 to n
   for j from i to n
      if (L-j+i-R(i,j)<0)
         S(i,j)=infinity
      else
         S(i,j)=(L-j+i-R(i,j))^2
return S
```

This is also $O(n^2)$ because it loops over n twice.

Now to minimize the squared slacks, consider the words in the last line from i to j. Given a breakpoint i for the first word of the last line, we can compute the minimum slack through the recurrence

$$\mathrm{OPT}(j) = \mathrm{OPT}(i-1) + S(i,j)$$

because the slack of the last line will be fixed. This means that we have a subproblem of size $i-1$ that we must minimize to obtain the optimal solution given this restraint. This leads to the overall recurrence

$$\mathrm{OPT}(j) = \min_{i=1}^{j}(\mathrm{OPT}(i-1) + S(i,j))$$

with $\mathrm{OPT}(0) = 0$. We compute $\mathrm{OPT}(n)$ to find the minimum square of slacks. We can keep track of the last word of each second to last line in order to print out a sequence of indices to partition W into, such that this partition minimizes the square of slacks. In full the algorithm is as follows.

```
opt(0)=0
K(0)=0
for j from 1 to n
   minCost = infinity
   previousLineEnd = 0
   for i from 1 to j
      if (opt(i-1)+S(i,j) <= minCost)
         minCost = opt(i-1)+S(i,j)
         previousLineEnd = i-1
   opt(j)=minCost
   K(j)=previousLineEnd
return opt(n)
```

This is $O(n^2)$ because it loops over n twice. It outputs the minimum squared slacks, and is correct because of the above recurrence relation. To output the partitions of W, do

```
pos = n
while (pos>0)
   println(pos)
   pos=K(pos)
```

This outputs the index of the last word of each line. It has a linear runtime, because each call of `pos=K(pos)` decrements `pos` by at least one. Therefore the whole algorithm is $O(n^2)$.

Chapter 6, Problem 12

Consider the location of the last file before the terminal file S_n. If it is $a \geq 1$ files before S_n, then the minimum cost OPT(n) given this restraint is $\mathrm{OPT}(n-a) + c_n + \sum_{i=0}^{a-1} i$. This is because the cost of the placements of the files on servers earlier to S_{n-a} is not affected by files placed afterwards. This forms a subproblem of size $n-a$ that we must minimize to obtain an optimal solution. This leads to the recurrence relation

$$\mathrm{OPT}(n) = \min_{1 \leq i \leq n} \left(\mathrm{OPT}(n-i) + c_n + \sum_{j=0}^{i-1} j \right)$$
$$\mathrm{OPT}(n) = \min_{1 \leq i \leq n} \left(\mathrm{OPT}(n-i) + c_n + \frac{(i-1)i}{2} \right)$$
$$\mathrm{OPT}(n) = \min_{0 \leq i \leq n-1} \left(\mathrm{OPT}(i) + c_n + \frac{(n-i-1)(n-i)}{2} \right)$$
$$\mathrm{OPT}(n) = \min_{0 \leq i \leq n-1} \left(\mathrm{OPT}(i) + c_n + \binom{n-i}{2} \right)$$

Initially we have $\mathrm{OPT}(0) = 0$. We can record the index i of the last file before the terminal file for each subproblem of size n in order to generate a configuration with minimum total cost. Let $C(n)$ be the placement cost c_n. Then in full the algorithm behaves as follows.

```
opt(0)=0
K(0)=0
for i from 1 to n
   minCost = infinity
   previousFileLocation = 0
   for j from 0 to i-1
      if (opt(j)+C(n)+(n-j)*(n-j-1)/2 <= minCost)
         minCost = opt(j)+C(n)+(n-j)*(n-j-1)/2
         previousFileLocation = j
   opt(i)=minCost
   K(i)=previousFileLocation
return opt(n)
```

This generates each OPT(i) according to the recurrence relation above. It loops twice over n, so it has a runtime of $O(n^2)$. It outputs the minimum cost $OPT(n)$. To output the indices of the servers where a copy of the file resides, simply do

```
pos = n
while (pos>0)
   println(pos)
   pos=K(pos)
```

which traverses the array of last files, generating a sequence that has the minimum cost $OPT(n)$ because `pos` must be decremented by at least one every iteration. Thus the entire time complexity of this algorithm is $O(n^2)$.