CS4234: Optimization Algorithms

Problem Set 1

Solution Sketches

S-1. You are given a graph $G = (V, E)$ with n nodes and m edges. (Perhaps the graph represents a telephone network.) Each edge is colored either blue or red. (Perhaps the blue edges are owned by Singtel and the red edges are owned by M1.) You are also given a parameter k as part of the input. (Perhaps your contract with the Singapore government specifies the precise value of k.)

Give an algorithm that finds a spanning tree of G with *exactly* k blue edges (and exactly $n-k-1$ red edges). Give the running time of your algorithm, and show that it is correct.

Solution

Executive summary: The basic idea is that we will build two spanning trees: one that has at least k blue edges, and one that has at least $n-k-1$ red edges. If this is not possible, then we report that no such spanning tree exists. Otherwise, we will then gradually switch one edge at at time from the second tree to the first tree until we have exactly k blue edges and exactly $n-k-1$ red edges. This algorithm will runs in time approximately $O(n^2 + m \log n)$. At the end, we will briefly describe a somewhat faster solution.

Algorithm description: The first step of the algorithm is to build a spanning tree B with at least k blue edges. There are many ways to accomplish this. The easiest is to give each blue edge a weight of 0, and each red edge a weight of 1. Now run a minimum spanning tree algorithm (e.g., Prim's Algorithm) to find a spanning tree B in $O(m \log n)$ time. If tree B has at least k blue edges, we continue. Otherwise, we report that no solution exists.

In the same way, we can also find a spanning tree R with at least $n-k-1$ red edges: set the weight of the red edges to 0 and the weight of the blue edges to 1 and run Prim's Algorithm. Again, if the resulting tree does not have at least $n-k-1$ red edges, then report that no solution exists.

The last step is to define a procedure that interpolates between the trees R and B. Consider the following procedure for morphing tree R into tree B: (1) take an edge e that is in tree B and not in tree R; (2) add edge e to tree R; (3) since R was a spanning tree, this creates a cycle, and there must be some edge on that cycle that is not in B (since B is also a tree)—remove that edge from R. Repeat these three steps until there are exactly k blue edges and exactly $n-k-1$ red edges in R. We refer to these three steps collectively as an iteration of the interpolation algorithm.

Explain why it works: We need to prove several facts in order to show that this algorithm works. First, we need to argue that we have correctly constructed trees R and B. It is immediately clear that they are both spanning trees. We need to argue:

Lemma 1 *If there exists a spanning tree B with at least k blue edges, then the MST algorithm*

returns a tree with at least k blue edges. Similarly, if there exists a spanning tree R with at least $n-k-1$ red edges, then the MST algorithm returns a tree with at least $n-k-1$ red edges.

Proof Assume that there exists a spanning tree with at least k blue edges. Then this spanning tree will have weight at most $n-k-1$. Thus any minimum spanning tree has weight at most $n-k-1$, and hence has at most $n-k-1$ red edges, as desired. The same holds for the red edges. □

This lemma ensures that if the algorithm reports that there is no possible spanning tree with k blue edges or with $n-k-1$ red edges, then it is correct.

Next, we need to analyze the interpolation between graphs R and B. We need a simple lemma which states that the interpolation steps work as expected:

Lemma 2 *As long as $R \neq B$, we can execute the three steps of the interpolation algorithm. When the three steps are complete, R is still a spanning tree.*

Proof First, if $R \neq B$, and both R and B are spanning trees, then there must be an edge in B that is not in R, and hence we can execute the first step. Second, since R was a spanning tree before we added the edge, that must create a cycle. Third, some edge on the cycle in R must not be in B: assume, for the sake of contradiction that every edge on the cycle is in B; then then there is a cycle in B which contradicts the fact that B is a tree. Finally, after the edge on the cycle is removed, R is again a tree, since it again has exactly $n-1$ edges and is still connected (since the only edge removed was on a cycle). □

We can now examine what would happen if we ran the interpolation steps indefinitely:

Lemma 3 *If R and B are each spanning trees where $R \neq B$, then after at most $n-1$ iterations of the interpolation algorithm, $R = B$.*

Proof In every iteration of the interpolation steps, we add an edge to R that is in B but not in R, and we never remove an edge from R that is in also B; thus the number of edge in B but not in R decreases at every iteration of the interpolation steps; hence eventually, after at most $n-1$ iterations of the interpolation steps, every edge in B is also in R. Since B is a spanning tree and R is also a tree, we conclude that $R = B$. □

Finally, we can show that at some point along the way, the algorithm discovers a proper spanning tree:

Theorem 4 *If R and B are each spanning trees, and initially R has at least $n-k-1$ red edges and B has at least B blue edges, and if we repeat the interpolation algorithm at most $n-1$ times, then after one of these iterations, the tree R has exactly k blue edges.*

Proof The proof follows by continuity. Initially, R has $\leq k$ blue edges. At the end, after $n-1$ iterations, R has $\geq k$ blue edges. At every step, the number of blue edges in R changes by at most 1. There are four cases:

- The edge added to R is blue and the edge removed is blue. The number of blue edges does not change.
- The edge added to R is blue and the edge removed is red. The number of blue edges increases by 1.
- The edge added to R is red and the edge removed is red. The number of blue edges does not change.
- The edge added to R is red and the edge removed is blue. The number of blue edges decreases by 1.

Thus, at some point between the beginning, when R has $\leq k$ blue edges, and the end, when R has $\geq k$ blue edges, there must be a point where R has exactly k blue edges. □

Running time: The first part of the algorithm requires running a minimum spanning tree algorithm twice, and hence runs in $O(m \log n)$ time. The second part of the algorithm repeats the interpolation algorithm at most $n-1$ times. Each iteration of the interpolation algorithm requires: (i) finding an edge to add to R, which takes at most n time; and (ii) finding an edge on the cycle to remove from R, which takes at most n time. Thus, the total time to run the interpolation steps is at most $O(n^2)$. (This could be optimized somewhat by keeping a list of all the edges in B but not in R.) Therefore the total running time of the algorithm is $O(n^2 + m \log n)$.

Faster solution: The slow part of this algorithm lies in finding cycles in the spanning tree as we interpolate one spanning tree in to another. Here is an approach that avoids that:

- First, identify the connected components in the graph with only the red edges. If there are $> k+1$ connected connected components, then it is impossible: there is no spanning tree with exactly k blue edges.
- Second, add blue edges connecting the red connected components. This can be done using a union-find data structure to check for connectivity among the red connected components, iterating through blue edges one at a time and adding the edge if it connects two previously unconnected components. If the graph is initially connected, this should be possible. Assume this adds x edges, and we need a further $k' = k - x$ blue edges.
- Third, add any other k' blue edges.

- Fourth, add red edges within each connected component until we have a spanning tree, i.e., all the nodes are connected. Since each red component was connected via red edges, there should exist a red path between any pair of nodes in a component, and so this should remain possible. Again, we can use union-find to check for connectivity.

Overall this should take approximately $O(m\alpha(n))$ time, where α is the inverse Ackerman function. (This additional cost comes from using the union-find data structure to check connectivity.)

S-2. You are given an undirected graph $G = (V, E)$. The goal of this problem is to remove a minimum number of edges such that the residual graph contains no triangles. (I.e., there is no set of three vertices (a, b, c) such that edges (a, b), (b, c), and (a, c) are all in the residual graph.)

Give a 3-approximation algorithm that runs in polynomial time. (That is, it should guarantee that there are no triangles, and that the number of edges removed is within a factor of 3 of optimal.)

Solution

Executive summary: The basic idea is to repeatedly find triangles in the graph G, and remove all three edges from the triangle. Since any optimal algorithm has to remove at least one edge from the triangle, we conclude that it is a 3-approximation algorithm.

Algorithm description: First, we need a sub-routine that, given a graph $G = (V, E)$, finds a triangle. For today's purposes, assume that the graph is stored as an adjacency matrix, and consider the simple algorithm that examines every $we\binom{n}{3}$ combinations of three nodes (u, v, w) and checks whether there are edges connecting them. In this way, it takes $O(n^3)$ time to find a triangle (or report that there are no triangles). Certainly further optimization is possible.

Second, the algorithm proceeds as follows: while there are any triangles left in the graph, execute the following steps:

- Let (u, v, w) be a triangle in G.
- Remove edges (u, v), (v, w), and (u, w) from G.

Explain why it works: First, it is easy to see that when the algorithm completes, there are no triangles. It remains to show that the algorithm is a 3-approximation algorithm. Define a ***triangle-set*** of G to be a set of disjoint triangles in G, i.e., a set of triangles such that no two triangles share an edge. (Triangles may share a node, however.) Notice that the algorithm produces a triangle-set, since after each iteration of the loop, it removes the edges from the previous triangle from the graph. Moreover, if the algorithm produces triangle set S, then the number of edges removed by the algorithm is exactly $3|S|$.

Let OPT be the optimal set of edges to remove from the graph G to ensure that it is triangle-free. Notice that for any triangle-set S, we can conclude that $|OPT| \geq |S|$, since OPT has to remove at least one edge in each triangle in S. Putting this together with the previous claim, we conclude that the number of edges removed by the algorithm is $3|S| \leq 3|OPT|$, and hence the algorithm is a 3-approximation of optimal.

Running time: The running time of the algorithm is at most $O(n^3 m)$: each iteration of triangle finding takes time $O(n^3)$, and there are at most m such iterations (since each iteration removes 3 edges).

S-3. Consider a set of n points in the Euclidean (2-dimensional) plane:

$$P = (x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n).$$

We say that a point (x_i, y_i) **dominates** a point (x_j, y_j) if it is bigger in both coordinates, i.e., if $(x_i > x_j)$ and $(y_i > y_j)$. A **maximal** point is not dominated by any other. In this problem, we will develop and analyze an algorithm for finding the set of maximal points.

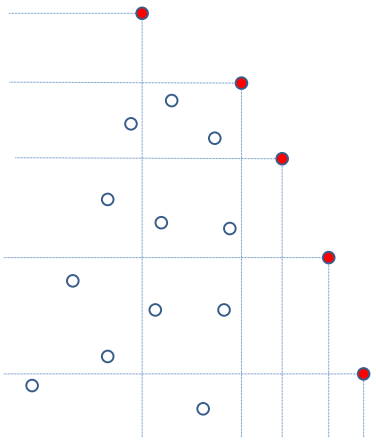

Figure 1: The solid circles are the maximal points, while the empty circles are dominated. The dashed lines indicate which points are dominated by a given solid circle.

Assume, for this problem, that all the x-coordinate and y-coordinate values are distinct. Consider the following solution, known as the Kirkpatrick-Seidel algorithm:

Problem 3.a. Prove that the `KirkpatrickSeidel` algorithm correctly returns the set of maximal points.

Solution

Executive summary. We need to argue that any point returned by `KirkpatrickSeidel` is a maximal point, and that `KirkpatrickSeidel` returns every maximal point. We do this in two steps, first establishing a set of basic properties about the points that remain at the end of every iteration, and then showing that the algorithm is correct.

Details. We first show the following:

Lemma 5 *At every iteration of the algorithm, the following four properties hold:*

1. *No point that remains in set P dominates the point being returned.*

```
1  Algorithm: KirkpatrickSeidel(P)
2  Procedure:
3      if P = ∅ then return ∅
4      if |P| = 1 then return P
5      Let x be the median x-coordinate of the points in P.
6      Let P_ℓ be all the points with x-coordinate ≤ x.
7      Let P_r be all the points with x-coordinate > x.
8      Let q be the point in P_r with the maximum y-coordinate.
9      Delete q from P_r.
10     Delete every point dominated by q in P_ℓ.
11     Delete every point dominated by q in P_r.
12     S_ℓ = KirkpatrickSeidel(P_ℓ)
13     S_r = KirkpatrickSeidel(P_r)
14     return S_ℓ ∪ S_r ∪ {q}
```

2. *No point that remains in the right half dominates any point in the left half.*

3. *No point that remains in the left half dominates any point in the right half.*

4. *Every point that is deleted is dominated (i.e., is not a maximal point).*

Proof The first property follows because when a point is returned, we delete all the points that it dominates.

The second property follows because q is chosen to be the point with maximum y-coordinate. That means that the only remaining points in P_r have y-coordinate $\leq q.y$. But every point in P_ℓ has x-coordinate less than q, so the only remaining points in P_ℓ have y-coordinate $> q.y$. Hence no remaining point in P_r dominates any points in P_ℓ.

The third property holds because every point in P_ℓ has x-coordinate $<$ every point ins P_r.

The fourth property holds because points are deleted at this iteration only when they are dominated by q. □

Given that these four properties hold at every iteration, we can conclude that the algorithm is correct.

Lemma 6 *Algorithm* `KirkpatrickSeidel` *is correct.*

Proof There are two ways by which the algorithm might fail: there is some maximal node that is not returned, and there is some node returned that is not maximal. We know that if a node is not returned, then it is dominated (by Property 4, Lemma 5), and hence it is not maximal. Thus the first type of error cannot occur.

We focus on the second type of error, i.e., some node is returned even though it is not maximal. Assume for the sake of contradiction, that x is a point that is returned, and that y is some other

point that dominates x. We now consider four cases, depending on when x and y were partitioned into different groups, or whether x or y were added/deleted first.

Case 1. Assume that points x and y at some point are divided into separate left and right sets, prior to x being returned or y being deleted or returned. We know, by Properties 2 and 3 (Lemma 5) that this cannot happen, as no remaining point in one half can dominate a point in the other half.

Case 2. Assume that points x and y are in the same group, and y is deleted. Notice that y is deleted due to the fact that it is dominated by some point z. This same point z would also dominate x, and hence x would also be deleted, and so this cannot happen.

Case 3. Assume that points x and y are in the same group, and y is added. Since y dominates x, point x would be deleted and hence it would not be returned.

Case 4. Assume that points x and y are in the same group, and x is added. This only occurs if x is the point with the highest y-coordinate in the group. But since y dominates x and is in the same group, this cannot happen.

Thus we conclude that there does not exist a node y that dominates x. □

Problem 3.b. Prove (briefly) that the `KirkpatrickSeidel` algorithm runs in time $O(n \log n)$.

Solution

We can describe the performance by a recurrence. Let $T(n)$ be the running time on n points. Then:

$$T(n) \leq 2T(n/2) + O(n)$$

That is, at each step, we divide the problem into two recursive pieces each of size *at most* $n/2$ (but maybe less). It costs $O(n)$ to do this partition, finding the median x-coordinate (in $O(n)$ time, using a linear time select), partitioning the list of points, finding the maximum y-coordinate, and deleting dominated points. Each of these steps can be done in $O(n)$ time. Solving the recurrence yields $T(n) = O(n \log n)$.

Problem 3.c. Assume that there are only h maximal points in the set P (but h is not known in advance). Show that the `KirkpatrickSeidel` algorithms runs in time $O(n \log h)$. (Hint: Beware that the recursion is not always balanced: for example, in one half, all the points may be deleted in line 10 or 11.)

Discussion: Notice that this is an example of parameterized analysis, where the running time depends on the parameter h. The `KirkpatrickSeidel` algorithm is, in fact, even better in that it is *instance optimal*, i.e., for any given input set P, there is no algorithm that is faster, in a sense.

Solution

Executive summary. There are several approaches, but one of the simplest is to divide the analysis into two cases, depending on how many points remain. Let N be the initial number of points, and let n be the number of points in a given recursive call. We will separately analyze the case where $n > N/h$ and the case where $n \leq N/h$. In the first case, a standard recurrence shows that the total cost is $O(N \log h)$. In the second case, a simple counting argument shows that the cost is at most $O(N)$.

Details. We begin by considering the cost of all the recursive calls where n is relatively large, i.e., $n \geq N/h$. You can think of this as looking at the first $\log h$ levels of the recursion tree.

Lemma 7 *The total cost of all recursive calls where the number of points $n \geq N/h$ is $O(N \log h)$.*

Proof Recall, we are solving the recurrence $T(n) = 2T(n/2) + O(n)$. For this part, we have a new base case: if $n \leq N/h$, then $T(n) = O(n)$, i.e., we are not going to count recursive calls on any smaller instances. Solving this recurrence in the standard way yields a cost $T(N) = O(N \log h)$.

One way to see this is to draw out the recurrence tree: it has at most $\log h$ levels, and each level has a cost that sums to $O(n)$.

Another way to analyze this is to divide the recursive calls up further, into sets I_k which includes all the recursive calls containing more than $n/2^{k+1}$ points but no more than $n/2^k$ points. Each of these sets I_k contains disjoint sets of points (because all the instances are approximately the same size) and hence contains $O(n)$ points, implying $O(n)$ work. Finally, there are $\log h$ such sets I_k. □

We can now look at the cost of small recursive calls where $n < N/h$.

Lemma 8 *The total cost of all iterations where $n < N/h$ is $O(n)$.*

Proof Notice that in total, there can be only h points output, and hence there are most h recursive calls where $n < N/h$. Each of these recursive calls has cost $O(N/h)$. Thus, the total cost of all of them is $O(N)$. □

Putting these two lemmas together yields our final claim, i.e., that the total cost is $O(n \log h)$.