CS 170 HW 5

Due on 2018-09-30, at 9:59 pm

1 (★) Study Group

List the names and SIDs of the members in your study group.

2 (★★) Updating a MST

You are given a graph $G = (V, E)$ with positive edge weights, and a minimum spanning tree $T = (V, E')$ with respect to these weights; you may assume G and T are given as adjacency lists. Now suppose the weight of a particular edge $e \in E$ is modified from $w(e)$ to a new value $\hat{w}(e)$. You wish to quickly update the minimum spanning tree T to reflect this change, without recomputing the entire tree from scratch. There are four cases. In each, give a description of an algorithm for updating T, a proof of correctness, and a runtime analysis for the algorithm. Note that for some of the cases these may be quite brief.

(a) $e \notin E'$ and $\hat{w}(e) > w(e)$

(b) $e \notin E'$ and $\hat{w}(e) < w(e)$

(c) $e \in E'$ and $\hat{w}(e) < w(e)$

(d) $e \in E'$ and $\hat{w}(e) > w(e)$

Solution:

(a) **Main Idea:** Do nothing.

Correctness: T's weight does not increase, and any other spanning tree's weight either stays the same or increases, so T must still be an MST.

Runtime: Doing nothing takes $O(1)$ time.

(b) **Main Idea:** Add e to T. Use DFS to find the cycle that now exists in T. Remove the heaviest edge in the cycle from T.

Correctness: Any edge not in an MST must be the heaviest edge in some cycle in G. For any any edge not in T except for e, decreasing e's weight does not change that it is the heaviest edge in the cycle, so it is safe to exclude from the MST. By adding e to T and then removing the heaviest edge in the cycle in T, we remove an edge that is also safe to exclude from the MST. Thus after this update, all edges outside of T are safe to exclude from the MST.

Runtime: This takes $O(|V|)$ time since T has $|V|$ edges after adding e, so the DFS runs in $O(|V|)$ time.

(c) **Main Idea:** Do nothing.

Correctness: T's weight decreases by $w(e) - \hat{w}(e)$, and any other spanning tree's weight either stays the same or also decreases by this much, so T must still be an MST.

Runtime: Doing nothing takes $O(1)$ time.

(d) **Main Idea:** Delete e from T. Now T has two components, A and B. Find the lightest edge with one endpoint in each of A and B, and add this edge to T.

Correctness: Every edge besides e in the MST is a lightest edge in some cut prior to changing e's weight, and increasing e's weight cannot affect this property. So all edges besides e are safe to keep in the MST. Then, whatever edge we add is also the lightest edge in the cut (A, B) and is thus also safe to include in the MST.

Runtime: This takes $O(|V| + |E|)$ time since we may have to scan all the edges in the graph.

3 (★★) Finding MSTs by Deleting Edges

Consider the following algorithm to find the minimum spanning tree of an undirected, weighted graph $G(V, E)$. For simplicity, you may assume that no two edges in G have the same weight.

procedure FINDMST($G(V, E)$)
  $E' \leftarrow E$
  for Each edge e in E in decreasing weight order **do**
    if $G(V, E' - e)$ is connected **then**
      $E' \leftarrow E' - e$
  return E'

Show that this algorithm outputs a minimum spanning tree of G.

Solution:

There are several solutions. One is to note that, anytime the algorithm chooses not to delete an edge e, that edge must be a lightest edge across some cut. In particular, the cut is the two components of the disconnected graph $E' - e$ (using the value of E' at the start of the iteration where the algorithm looks at e). The algorithm is also guaranteed to output E' containing no cycles, so by applying the cut property we get that it outputs a minimum spanning tree.

Other proofs include the use of the cycle property. As a review, the cycle property claims that the heaviest edge in any cycle in G cannot appear in the (unique) minimum spanning tree. To prove the cycle property, suppose e is the heaviest edge in some cycle C and is in the minimum spanning tree T^*. Consider deleting e from the minimum spanning tree to get $T^* - e$. $T^* - e$ has two components, and some edge e' in C other than e must connect the two components. So $T^* - e + e'$ is a spanning tree, and costs less than T^* since e' costs less than e, a contradiction.

The algorithm is guaranteed to output a spanning tree T. Suppose that the MST $T*$ is not T, and let $e \in T * -T$. Then $T \cup e$ contains a cycle; denote by e' its heaviest edge, and

note that $e \neq e'$ by the cycle property. When we considered e', all edges in $T \cup e$ were still there, and so we should have deleted e' since removing it would leave the graph connected.

An alternative proof is to show that every edge we delete is the heaviest edge in some cycle. This is because whenever we delete an edge e, it is part of some cycle in the remaining edges in E' since E' remains connected after deleting e. No other edge in this cycle can be heavier than e, otherwise we would have deleted that edge first. So by the cycle property we have only deleted edges that do not appear in the minimum spanning tree. Furthermore, note that this algorithm will eliminate all cycles from T. So we know the final solution is a tree, and thus must be the minimum spanning tree.

4 (★★) Minimum Spanning k-Forest

Given a graph $G(V, E)$ with nonnegative weights, a spanning k-forest is a cycle-free collection of edges $F \subseteq E$ such that the graph with the same vertices as G but only the edges in F has at most k connected components. The weight of a k-forest F is the total weight of all edges in F. The minimum spanning k-forest is defined as the spanning k-forest of minimum weight. Note that when $k = 1$, this is equivalent to the minimum spanning tree. For simplicity, in this problem you may assume that all edges in G have distinct weights.

(a) Define a j-partition of a graph G to be a list of j sets of vertices $\Pi = \{S_1, S_2 \dots S_j\}$ such that every vertex in G appears in exactly one of these sets. Define an edge (u, v) to be crossing a j-partition $\Pi = \{S_1, S_2 \dots S_j\}$ if the set in Π containing u and the set in Π containing v are different sets. For example, one 3-partition of the graph with vertices $\{A, B, C, D, E\}$ is $\Pi = \{\{A, B\}, \{C\}, \{D, E\}\}$, and an edge from A to C would cross this partition.

Show that for any k'-partition Π of a graph G, if $k' > k$ then the lightest edge crossing Π must be in the minimum spanning k-forest of G.

(b) Give an efficient algorithm for finding the minimum spanning k-forest. Your solution should include the algorithm description, proof of correctness, and runtime analysis.

Solution:

(a) It helps to note that when $k = 1, k' = 2$ this is exactly the cut property. A similar argument lets us prove this claim.

For some k'-partition Π where $k' > k$, suppose that e is the lightest edge crossing Π but e is not in the minimum spanning k-forest. Let F be the minimum spanning k-forest. Now, consider adding e to F. One of two cases occurs:

- $F + e$ contains a cycle. In this case, some edge e' in this cycle besides e must cross Π. This means $F + e - e'$ is a spanning k-forest, since deleting an edge in a cycle cannot increase the number of components. Since e by definition is cheaper than e', the forest $F + e - e'$ is cheaper than F, which contradicts F being a minimum spanning k-forest.

- $F + e$ does not contain a cycle. In this case, the endpoints of e are in two different components of F, so $F + e$ has $k - 1$ components. Since Π is a k'-partition and $k' > k$, some edge e' in F must cross Π. Deleting an edge from a forest increases the number of components in the forest by only 1, so $F + e - e'$ has k components, i.e. is a k-forest. Since e by definition is cheaper than e', the forest $F + e - e'$ is cheaper than F, which contradicts F being a minimum spanning k-forest.

In either case, we arrive at a contradiction and have thus proven the claim.

(b) There are multiple solutions, we recommend the following one because its proof of correctness follows immediately from part a:

Main Idea: The algorithm is to run Kruskal's, but stop when $n - k$ edges are bought, i.e. the solution is a spanning k-forest.

Correctness: Any time the algorithm adds an edge e, let $S_1 \dots S_j$ be the components defined by the solution Kruskal's arrived at prior to adding e. $S_1 \dots S_j$ form a j-partition and by definition of the algorithm, $j > k$. e is the cheapest edge crossing this j-partition, so by part a e must be in the (unique) minimum spanning k-forest. Since every edge we add is in the minimum spanning k-forest, our final solution must be the minimum spanning k-forest.

Runtime Analysis: This is just modified Kruskal's so the runtime is $O(|E| \log |V|)$ (Kruskal's runtime is dominated by the edge sorting, so the fact that we may make less calls to the disjoint sets data structure because the algorithm terminates early does not affect our asymptotic runtime).

5 (★★) Parallelizable Prim's

Given an undirected, weighted graph $G(V, E)$, consider the following algorithm to find the minimum spanning tree. This algorithm is similar to Prim's, except rather than grow out a spanning tree from one vertex, it tries to grow out the spanning tree from every vertex at the same time.

procedure FINDMST($G(V, E)$)
  $T \leftarrow \emptyset$
  while T is not a spanning tree **do**
    Let $S_1, S_2 \dots S_k$ be the components of the graph with vertices V and edges T
    For each i, let e_i be the minimum-weight edge with exactly one endpoint in S_i
    $T \leftarrow T \cup \{e_1, e_2, \dots e_k\}$
  return T

For simplicity, in the following parts you may assume that no two edges in G have the same weight.

(a) Show that this algorithm finds a minimum spanning tree.

(b) Give an exact upper bound (that is, an upper bound without using Big-O notation) on the worst-case number of iterations of the while loop in one run of the algorithm.

(c) Using your answer to the previous part, give an upper bound on the runtime of this algorithm.

Solution:

This algorithm is known as Boruvka's algorithm. Despite perhaps appearing complicated for an MST algorithm, it was discovered in 1926, predating both Prim's and Kruskal's. As the title of the problem hints, this algorithm is still of interest due to being easily parallelizable, among other reasons.

(a) If we add an edge e_i to T, it is because it is the cheapest edge with exactly one endpoint in S_i. So applying the cut property to the cut $(S_i, V - S_i)$ we see that e_i must be in the (unique) minimum spanning tree. So every edge we add must be in the minimum spanning tree, i.e. this algorithm finds exactly the minimum spanning tree.

(b) The number of components in the graph with vertices V and edges T starts out at $|V|$, with one component for each vertex. If there are k components at the start of an iteration, we must add at least $k/2$ edges in that iteration: every component S_i contributes some e_i to the set of edges we're adding, and each edge we add can only have been contributed by up to two components. This means the number of components decreases by at least a factor of 2 in every iteration. Thus, the while loop runs for at most $\log |V|$ iterations.

(c) The previous part shows that at most $\log |V|$ iterations are needed. Each iteration can be performed in $O(|E|)$ time (e.g. you can compute the components in $O(|E|)$ time, and then initialize a table which stores the cheapest edge with exactly one endpoint in each component, and then using a linear scan over all edges fill out this table) giving a runtime bound of $O(|E| \log |V|)$.

6 (★) Huffman Coding

In this question we will consider how much Huffman coding can compress a file F of m characters taken from an alphabet of $n = 2^k$ characters x_0, x_1, ... , x_{n-1} (each character appears at least once).

(a) Let $S(F)$ represent the number of bits it takes to store F without using Huffman coding (i.e., using the same number of bits for each character). Represent $S(F)$ in terms of m and n.

(b) Let $H(F)$ represent the number of bits used in the optimal Huffman coding of F. We define the *efficiency* $E(F)$ of a Huffman coding on F as $E(F) := S(F)/H(F)$. For each m and n describe a file F for which $E(F)$ is as small as possible.

(c) For each m and n describe a file F for which $E(F)$ is as large as possible. How does the largest possible efficiency increase as a function of n? Give you answer in big-O notation.

Solution:

(a) $m \log(n)$ bits.

(b) The efficiency is smallest when all characters appear with equal frequency. In this case, $E(F) \approx 1$.

(c) Let F be x_0, x_1, ..., x_{n-2} followed by $m - (n-1)$ instances of x_{n-1}. This file has efficiency

$$\frac{m \log(n)}{(m-(n-1)) \cdot 1 + (n-1) \cdot \log(n)}$$

This efficiency is best when m is very large, and thus $(m-(n-1)) \cdot 1$ far outweighs $(n-1) \cdot \log(n)$. This results in the equation

$$\frac{m \log(n)}{(m-(n-1)) \cdot 1 + (n-1) \cdot \log(n)} \approx \frac{m \log(n)}{(m-(n-1)) \cdot 1} \approx \log(n)$$

So the efficiency is $O(\log(n))$ (just stating the efficiency is enough for full-credit).

7 (★★) Sum of Products

This question guides you through writing a proof of correctness for a greedy algorithm. You have n computing jobs to perform, with job i requiring t_i units of CPU time to complete. You also have access to n machines that you can assign these jobs to. Since the machines are in high demand, you can only assign one job to any machine. The jth machine costs c_j dollars for each unit of CPU time it spends running a job, so assigning job i to machine j will cost you $t_i \cdot c_j$ dollars (each job takes the same amount of CPU time to complete, regardless of which machine is used). Your goal is to find an assignment of jobs to machines that minimizes the total cost.

Assume the jobs and machines are sorted and have distinct runtimes/costs, i.e. $t_1 > t_2 > \ldots > t_n$ and $c_1 > c_2 > \ldots > c_n$.

(a) What assignment of jobs to machines minimizes the total cost? (Hint: What machine should we assign the longest job to? What machine should we assign the second longest job to? It might help to solve a small example by hand first.)

(b) Given an assignment of jobs to machines, consider the following modification: If there is a pair of jobs i, j such that job i is assigned to machine i', job j is assigned to machine j', and $t_i > t_j$ and $c_{i'} > c_{j'}$, instead assign job i to machine j' and job j to machine i'. Show that this modification decreases the total cost of an assignment.

(c) Use part b to show that the assignment you chose in part a has the minimum total cost (Hint: Show that for any assignment other than the one you chose in part a, you can apply the modification in part b. Conclude that the assignment you chose in part a is the optimal assignment.)

Solution:

(a) Assign the longest job (job 1) to the cheapest machine (machine n), the second longest job (job 2) to the second cheapest machine (machine $n-1$), etc.

(b) The cost of computing jobs other than i and j is unaffected. The old cost of computing jobs i and j is $t_i c_{i'} + t_j c_{j'}$. The new cost of computing jobs i and j is $t_i c_{j'} + t_j c_{i'}$. The decrease in cost is then $t_i c_{i'} + t_j c_{j'} - t_i c_{j'} - t_j c_{i'} = (t_i - t_j)(c_{i'} - c_{j'})$, which is positive because $t_i > t_j$ and $c_{i'} > c_{j'}$.

(c) Suppose for an assignment of jobs to machines that no modification of the type suggested in part b is possible. Then job 1 must be assigned to machine n in this assignment - otherwise job 1 and whatever job is assigned to machine n satisfy the conditions in part b. But if we know job 1 is assigned to job n, then we can similarly conclude that job 2 must be assigned to machine $n-1$, and so on. So this assignment must be exactly the assignment we chose in part a.

Then for every assignment except the assignment we found in part a, there is another assignment that has lower total cost, i.e. the former is not optimal. This implies that the assignment from part a is the optimal assignment.

8 (★★★) Preventing Conflict

A group of n guests shows up to a house for a party, but some pairs of guests are enemies. There are two rooms in the house, and the host wants to distribute guests among the rooms, breaking up as many pairs of enemies as possible. The guests are all waiting outside the house and are impatient to get in, so the host needs to assign them to the two rooms quickly, even if this means that it's not the best possible solution. Come up with a linear-time algorithm that breaks up at least half as many pairs of enemies as the best possible solution. Provide a proof of correctness and a runtime analysis as well.

You can treat the input as an undirected graph $G = (V, E)$ where each vertex in V represents a guest and (u, v) is in E if u and v are enemies.

Solution:

Main Idea: Let guests be nodes in a graph $G = (V, E)$ and there is an edge between each pair of enemies. The number of conflicts prevented after partitioning nodes into two non-intersecting sets A and B (also called a cut) is the number of edges between A and B. We assign nodes to the sets one by one in any order. A node v not yet assigned would have edges to nodes already in sets A or B. We greedily assign it to the set where it has a smaller total number of edges to. This cuts at least half of the total edges.

Pseudocode:

1. Initialize empty sets A and B
2. For each node $v \in V$:
3. Initialize $n_A := 0$, $n_B := 0$
4. For each $\{v, w\} \in E$:
5. Increment n_A or n_B if $w \in A$ or $w \in B$, respectively
6. Add v to set A or B with lower n_A or n_B, break ties arbitrarily
7. Output sets A and B

Proof of Correctness: In each iteration, when we consider a vertex v, its edges are connected to other vertices already in these three disjoint sets: A, B, or the set of not-yet-assigned vertices. Let the number of edges in each case be n_A, n_B and n_X respectively.

Suppose we add v to A, then n_B edges will be cut, but n_A edges can never be cut. Since $\max(n_A, n_B) \geq \frac{n_A+n_b}{2}$, each iteration will cut at least $\frac{n_A+n_b}{2}$ edges, and in total at least $\frac{|E|}{2}$ of the edges will be cut. Let $\texttt{greedycut}(G)$ be the number of edges cut by our algorithm, and $\texttt{maxcut}(G)$ be the best possible number of edges cut. Thus

$$\frac{\texttt{greedycut}(G)}{\texttt{maxcut}(G)} \geq \frac{\texttt{greedycut}(G)}{|E|} \geq \frac{|E|/2}{|E|} \geq \frac{1}{2}$$

Running Time: Each vertex is iterated through once, and each edge is iterated over at most twice, thus the runtime is $O(|V| + |E|)$.