CS 170 HW 6

Due on 2019-03-04, at 10:00 pm

1 Study Group

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

2 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.

3 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 $x_0, x_1, \ldots, x_{n-1}$ (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 $x_0, x_1, \ldots, x_{n-2}$ 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 $O(\log(n))$.

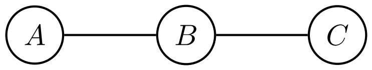
4 Graph Game

Given an undirected, unweighted graph G, with each node v having a value $\ell(v) \geq 0$, consider the following game.

1. All nodes are initially *unmarked* and your score is 0.

2. Choose an unmarked node u. Let $M(u)$ be the *marked* neighbours of u. Add $\sum_{v \in M(u)} \ell(v)$ to your score. Then mark u.

3. Repeat the last step for as many turns as you like, or until all the nodes are marked.



For instance, suppose we had the graph:
with $\ell(A) = 3$, $\ell(B) = 2$, $\ell(C) = 3$. Then, an optimal strategy is to mark A then C then B giving you a score of $0 + 0 + 6$. We can check that no other order will give us a better score.

(a) Is it ever better to leave a node unmarked? Briefly justify your answer.

(b) Give a greedy algorithm to find the order which gives the best score. Describe your algorithm, prove its optimality, and analyse its running time.

(c) Now suppose that $\ell(v)$ can be negative. Give an example where your algorithm fails.

(d) Your friend suggests the following modified algorithm: delete all v with $\ell(v) < 0$, then run your greedy algorithm on the resulting graph. Give an example where this algorithm fails.

Solution:

(a) Marking a node can only ever increase your score, since all values are positive.

(b) Sorting the nodes by value largest-first (breaking ties arbitrarily), and taking them in that order will lead to an optimal solution. We show this by an exchange argument. Suppose that we have an ordering $v_1, \ldots, v_n$ which is not sorted by decreasing value, and let i be such that $\ell(v_i) < \ell(v_{i+1})$. Note first that if we swap the order of these nodes, only the scores that we obtain in steps i and $i + 1$ could change, since in all other steps the set of marked nodes is unaffected. There are two cases.

   (a) Case 1: v_i and v_{i+1} do not have an edge between them. Then the score when marking v_i is not affected by whether v_{i+1} is marked, and vice versa.

   (b) Case 2: π_i and π_{i+1} do have an edge between them. Then if we swap them, v_{i+1}'s score decreases by $\ell(v_i)$, and v_i's score increases by $\ell(v_{i+1})$. Since $\ell(v_{i+1}) > \ell(v_i)$, the total score increases.

   Thus, by the exchange argument, sorting the nodes will give us an optimal solution. The running time of this algorithm is $O(|V| \log |V|)$, since we just sort the vertices.

(c) We can take the example graph and set $\ell(A) = 1$, $\ell(B) = -1$, $\ell(C) = -2$. Then the greedy algorithm gives A, B, C with value 0. The optimum is A, B with value 1.

(d) Again we take the example graph and set $\ell(A) = \ell(C) = 3$, $\ell(B) = -1$. The modified algorithm gives A, C which has value 0. The optimum is A, C, B with value 6.

5 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.

6 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 $O(|V| + |E|)$.