CS170 Discussion Section 6: *3/1*

1 Minimum Spanning Trees

For each of the following statements, either prove or supply a counterexample. Always assume $G = (V, E)$ is undirected and connected. Do not assume the edge weights are distinct unless specifically stated.

(a) Let e be any edge of minimum weight in G. Then e must be part of some MST.

Solution: True, e will belong to the MST produced by Kruskal.

(b) If the lightest edge in a graph is unique, then it must be part of every MST.

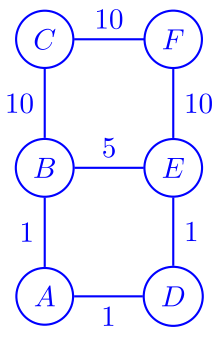
Solution: True, if e is not part of some MST, there exists a cycle connecting the two endpoints of e, so adding e and removing another edge of the cycle, produces a lightest tree. If e is not on a cycle, it bridges two components and is part of any spanning tree to ensure connectivity.

(c) If e is part of some MST of G, then it must be a lightest edge across some cut of G.

Solution: True, suppose (u, v) is the edge. Let one side of the cut be everything reachable in the MST from u without using the edge (u, v). If this cut has an edge lighter than (u, v) then we could add this edge to the MST and remove (u, v). We know this edge is not already in the MST because otherwise both its endpoints would be reachable from u without using (u, v).

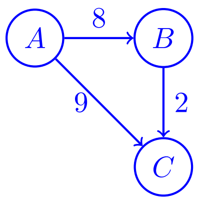
(d) If G has a cycle with a unique lightest edge e, then e must be part of every MST.

Solution: False. Let e be also the heaviest edge of a different cycle; then, we know that e can't be part of the MST. Concretely, in the following graph, edge (B, E) satisfies this condition, but will not be added to the graph.



(e) The shortest path tree computed by Djikstra's algorithm is necessarily part of some MST.

Solution: False. In the following graph, the MST has edges (A, B) and (B, C) while Dijkstra's algorithm from A gives $(A, B), (A, C)$.



(f) Prim's algorithm works correctly when there are negative edges.

Solution: True. At each stage, the new edge that contributes the least weight to the tree is added. The order in which vertices are added to the MST depends only on the ordering of the edge weights; whether the edges are negative has no effect on this.

(g) For any $r > 0$, define an r-path to be a path whose edges all have weight less than r. If G contains an r-path from s to t, then every MST of G must also contain an r-path from s to t.

Solution: True. Let $v_1, v_2, \ldots, v_n$ denote the r path in G from $s = v_1$ to $t = v_n$. Consider the greatest i such that the MST has an r path from s to v_i and assume for contradiction that $i < n$. Then the edge (v_i, v_{i+1}) is not in the MST. Adding this edge to the MST forms a cycle, which must have edges of length less than r since otherwise we could replace one of them with (v_i, v_{i+1}). Thus there is an r path from v_i to v_{i+1} and hence from s to v_{i+1}, contradicting our initial assumption.

2 Approximating Vertex Cover

Given an undirected graph $G = (V, E)$, nodes $S \subseteq V$ form a *vertex cover* of G if every edge has at least one of its endpoints in S. In other words, $\forall (i, j) \in E$, we have $i \in S$ or $j \in S$.

It turns out that finding the smallest set of vertices that covers all the edges—the *minimum vertex cover*—is a very hard problem. Fortunately, it is easy find a vertex cover not too much larger than the true minimum. Consider the following greedy algorithm:

```
$S \leftarrow \emptyset$
**for** $(i, j) \in E$ **do**
    **if** $(i, j)$ is not covered **then** add both $i$ and $j$ to $S$
**return** $S$
```

Show that this algorithm is guaranteed to find a vertex cover of size at most $2 \cdot \textsc{opt}$, where $\textsc{opt}$ is the size of the minimum vertex cover.

Solution: Observe that S is just the set of endpoints of the edges visited by the algorithm. Moreover, these edges do not have any endpoints in common, since an edge is visited only if both of its endpoints are not currently in S.

Thus the number of edges visited is $|S|/2$, and any vertex cover must have size at least $|S|/2$ by above. The conclusion follows since the algorithm outputs a vertex cover of size $|S|$.

3 Service scheduling

A server has n customers waiting to be served. Customer i requires t_i minutes to be served. If, for example, the customers were served in the order $t_1, t_2, t_3, \ldots$, then the ith customer would wait for $t_1 + t_2 + \cdots + t_i$ minutes.

We want to minimize the total waiting time

$$T = \sum_{i=1}^{n} (\text{time spent waiting by customer } i)$$

Given the list of t_i, give an efficient algorithm for computing the optimal order in which to process the customers.

Solution: We simply proceed by a greedy strategy, by sorting the customers in the increasing order of service times and servicing them in this order. The running time is $O(n \log n)$. To prove the correctness, for any ordering of the customers, let $s(j)$ denote the jth customer in the ordering. Then

$$T = \sum_{i=1}^{n} \sum_{j=1}^{i-1} t_{s(j)} = \sum_{i=1}^{n} (n-i) t_{s(i)}$$

For any ordering, if $t_{s(i)} > t_{s(j)}$ for $i < j$, then swapping the positions of the two customers gives a better ordering. Since, we can generate all possible orderings by swaps, an ordering which has the property that $t_{s(1)} \leq \ldots \leq t_{s(n)}$ must be the global optimum. However, this is exactly the ordering we output.

4 Huffman Encoding

1. Under a Huffman encoding of n symbols with frequencies $f_1, f_2, \ldots, f_n$, what is the longest a codeword could possibly be? Give an example set of frequencies that would produce this case, and argue that it is the longest possible.

   Solution: The longest codeword can be of length $n-1$. An encoding of n symbols with $n-2$ of them having probabilities $1/2, 1/4, \ldots, 1/2^{n-2}$ and two of them having probability $1/2^{n-1}$ achieves this value. No codeword can ever by longer than length $n-1$. To see why, we consider a prefix tree of the code. If a codeword has length n or greater, then the prefix tree would have height n or greater, so it would have at least $n+1$ leaves. Our alphabet is of size n, so the prefix tree has exactly n leaves.

2. Prove that if all characters occur with frequency less than $1/3$, there is guaranteed to be no codeword of length 1.

   Solution: Suppose this was not the case. Let x be a node corresponding to a single character with $p(x) < 1/3$ such that the encoding of x is of length 1. Then x must not merge with any other node until the end. Consider the stage when there are only three leaves x, y, z left in the tree. At the last stage, y, z must merge to form another node so that x still corresponds to a codeword of length 1. But $p(x) + p(y) + p(z) = 1$, and $p(x) < 1/3$ implies $p(y) + p(z) > 2/3$. Hence, at least one of $p(y)$ or $p(z)$, say $p(z)$ must be greater than $1/3$. But these two can not merge since $p(x)$ and $p(y)$ would be minimum. This is a contradiction.

3. Prove that if some character occurs with frequency more than $\frac{2}{5}$, there is guaranteed to be a codeword of length 1.

   Solution: Let s be the symbol with highest frequency ($p(s) > 2/5$) and suppose that it merges with some other node x during the process of constructing the tree and hence does not correspond to a codeword of length 1. Since s, x must be the two with minimum frequencies, there was at least one other node y such that $p(y) > p(s)$ and $p(y) > p(x)$. Thus, $p(y) > 2/5$ and hence $p(x) < 1/5$. Now, y cannot itself represent a character because it would be a codeword of length 1, and therefore y must have been formed by merging two nodes z, w. Since $p(y) > 2/5$, either z, w must have probability greater than $1/5$. But this is a contradiction: $p(z), p(w)$ could not have been the minimum since $p(x) < 1/5$.