CS 170 HW 6

Due **2020-03-02, at 10:00 pm**

1 Study Group

List the names and SIDs of the members in your study group. If you have no collaborators, you must explicitly write none.

2 2-SAT

Please provide solutions to parts (d), (e) and (f) of Question 3.28 from `http://algorithmics.lsi.upc.edu/docs/Dasgupta-Papadimitriou-Vazirani.pdf`.

Solution:

(d) Supoose there is a SCC containing both x and $\overline{x}$. Notice that the edges of the graph are necessary implications. Thus, if some x and $\overline{x}$ are in the same component, there is a chain of implications which is equivalent to $x \to \overline{x}$ and a different chain which is equivalent to $\overline{x} \to x$, i.e. there is a contradiction in the set of clauses.

(e) Take any sink component, and assign variables so all the literals in this component are True. Because of how we define the graph, there is a corresponding source component which has the negations of all literals in this component. Remove this source/sink component pair, and repeat the process until the graph is empty. Since we set components to true in reverse topological order, there is no implication from a true literal to a false literal. Since no literal and its negation are in the same SCC, we never try to set a variable to be both true and false. So this produces an assignment satisfying all clauses.

(f) Let φ be a formula acting on n literals $x_1, \dots, x_n$. Construct a graph with $2n$ vertices representing the set of literals and their negations. For each clause $(a \vee b)$ of φ add the edges $\overline{a} \Rightarrow b$ and $\overline{b} \Rightarrow a$. Use the strongly connected components algorithm and for each i, check if there is a SCC containing both x_i and $\overline{x_i}$. If any such component is found, report unsatisfiable. Otherwise, report satisfiable.

(Note: A common mistake is to report unsatisfiable if there is a path from x_i to $\overline{x_i}$ in this graph, even if there is no path from $\overline{x_i}$ to x_i. Even if there is a series of implications which combined give $x_i \to \overline{x_i}$, unless we also know $\overline{x_i} \to x_i$ we could set x_i to False and still possibly satisfy the clauses. For example, consider the 2-SAT formula $(\overline{a} \vee b) \wedge (\overline{a} \vee \overline{b})$. These clauses are equivalent to $a \to b, b \to \overline{a}$, which implies $a \to \overline{a}$, but this 2-SAT formula is still easily satisfiable.)

3 Perfect Matching on Trees

A *perfect matching* in an undirected graph $G = (V, E)$ is a set of edges $E' \subseteq E$ such that for every vertex $v \in V$, there is exactly one edge in E' which is incident to v.

Give an algorithm which finds a perfect matching *in a tree*, or reports that no such matching exists. Describe your algorithm, prove that it is correct and analyse its running time.

Solution: Let v be a leaf vertex in T. Since v has only one incident edge $e = (u, v)$, e must be in every perfect matching in T. Add e to E' and remove u, v from T. Repeat until there are no edges remaining. If there are no vertices remaining, return E', otherwise output 'no matching'.
One point to note: removing u, v might cause the graph to become disconnected. In that case we just run the algorithm on each connected component.
The running time of this algorithm is $O(|V|)$. We visit every node at most twice: once when we search into its subtree, and once when we remove it from T after it's been matched.

4 Encoding Emergencies

The basic intuition behind Huffman's algorithm (i.e. that frequent words have short encodings and infrequent words have long encodings) is at work in the English language. The words "I", "me", "you", "he", and "she" are all short, while the word "velociraptor" is quite long. However, words like "fire!","danger!", or "high-voltage!" are not short because they are frequent. Instead, time is precious in situations when these words are used.
To make things theoretical, suppose we have a file containing words numbered $1, \dots, m$ occurring with frequencies $f(1), \dots, f(m)$ and suppose that for each word i, the cost per bit of encoding of the word is $c(i)$, so that if we find a prefix code where word i has encoding length $l(i)$, the total cost of our encoding will be $\Sigma_i f(i) \cdot c(i) \cdot l(i)$.
Show how to modify Huffman's algorithm in order to compute the prefix code of minimum total cost.

Solution: The "modified frequencies" $f(i) \cdot c(i)$ play precisely the same role in the statement of this problem as the frequencies $f(i)$ did in the original problem. So we need only run Huffman's algorithm, but letting the "frequency" of each character k be the sum, over all words w in the file, of the number of occurrences of k in w times the cost per bit for w.

5 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 k connected components. For example, consider the graph $G(V, E)$ with vertices $V = \{A, B, C, D, E\}$ and all possible edges. One spanning 2-forest of this graph is $F = \{(A, C), (B, D), (D, E)\}$, because the graph with vertices V and edges F has components $\{A, C\}, \{B, D, E\}$.
The minimum spanning k-forest is defined as the spanning k-forest with the minimum total edge weight. (Note that when $k = 1$, this is equivalent to the minimum spanning tree). In this problem, you will design an algorithm to find the minimum spanning k-forest. For simplicity, you may assume that all edges in G have distinct weights.

(a) Define a j-partition of a graph G to be a partition of the vertices V into j (non-empty) sets. That is, a j-partition is a list of j sets of vertices $\Pi = \{S_1, S_2 \dots S_j\}$ such that every S_i includes at least one vertex, and every vertex in G appears in exactly one S_i.

For example, if the vertices of the graph are $\{A, B, C, D, E\}$, one 3-partition is to split the vertices into the sets $\Pi = \{\{A, B\}, \{C\}, \{D, E\}\}$.

Define an edge (u, v) to be crossing a j-partition $\Pi = \{S_1, S_2 \ldots S_j\}$ if the set in Π containing u and the set in Π containing v are different sets. For example, for the 3-partition $\Pi = \{\{A, B\}, \{C\}, \{D, E\}\}$, an edge from A to C would cross Π.

Show that for any j-partition Π of a graph G, if $j > k$ 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.

Please give a 3-part solution.

Solution:

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

For some j-partition Π where $j > k$, 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 j-partition and $j > k$, 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 $S_1 \ldots S_j$ be the components defined by the solution Kruskal's arrived at prior to adding e. $S_1 \ldots S_j$ 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).

6 A Divide and Conquer Algorithm for MST

Is the following algorithm correct? If so, prove it. Otherwise, give a counterexample and explain why it doesn't work.

```
procedure FindMST($G$: graph on $n$ vertices)
    If $n = 1$ return the empty set
    $T_1 \leftarrow$ FindMST($G_1$: subgraph of $G$ induced on vertices $\{1, \dots, n/2\}$)
    $T_2 \leftarrow$ FindMST($G_2$: subgraph of $G$ induced on vertices $\{n/2+1, \dots, n\}$)
    $e \leftarrow$ cheapest edge across the cut $\{1, \dots, \frac{n}{2}\}$ and $\{\frac{n}{2}+1, \dots, n\}$.
    return $T_1 \cup T_2 \cup \{e\}$.
```

Solution: This algorithm does not work; multiple edges of the MST could cross this particular cut. Another way to see this is that the MSTs of the subgraph needn't also be part of the MST of the whole graph.
As a concrete counterexample, consider a wide rectangle and the horizontal cut between the top two vertices and the bottom two. Both edges on this cut should be in the MST.

7 Picking a Favorite MST

Consider an undirected, weighted graph for which multiple MSTs are possible (we know this means the edge weights cannot be unique). You have a favorite MST, F. Are you guaranteed that F is a possible output of Kruskal's algorithm on this graph? How about Prim's? In other words, is it always possible to "force" the MST algorithms to output F without changing the weights of the given graph? Justify your answer. **Solution:** Yes; for both MST algorithms,

it's possible to ensure they output F, provided it is indeed an MST.
First, consider Kruskal's algorithm. Make sure that the edges of F are always before any other equally-weighted edges after the sort. Now, it will add all such edges as early as possible. Consider towards a contradiction the case where at any point Kruskal's declines to add an edge e in F to the MST. This means that e would have created a cycle with other equally-weighted edges in F, and/or lighter edges (possibly in F). (Before this point, Kruskal's may have added edges not in F to the MST.) But then it's impossible that both e and all of the other equally-weighted edges in F in this cycle can be in any MST, as one of them is the heaviest edge in some cycle of the graph, and such edges cannot be in any MST.
Since we assumed that F is an MST, this is a contradiction. Therefore we conclude that Kruskal's algorithm will add all edges in F to the MST. And since all MSTs have the same number of edges, this means it cannot add any edges not in F.

Now, consider Prim's algorithm. As it expands the fringe, have it only choose edges in F (so when there are multiple lightest edges to choose, choose a lightest edge in F). If this strategy fails, there must have been some cut across which none of the lightest edges were in F. But if this is the case, F cannot have been an MST (one of the lightest edges across any cut must be in any MST). Given that F must be an MST, this strategy will work.