CS 170 HW 6

Due **2021-03-02, at 10:00 pm**

1 Study Group

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

2 2-SAT

In the 2SAT problem, you are given a set of clauses, where each clause is the disjunction (OR) of two literals (a literal is a Boolean variable or the negation of a Boolean variable). You are looking for a way to assign a value true or false to each of the variables so that all clauses are satisfied that is, there is at least one true literal in each clause. For example, heres an instance of 2SAT:

$$(x_1 \vee \overline{x_2}) \wedge (\overline{x_1} \vee \overline{x_3}) \wedge (x_1 \vee x_2) \wedge (\overline{x_3} \vee x_4) \wedge (\overline{x_1} \vee x_4)$$

This instance has a satisfying assignment: set x_1, x_2, x_3, and x_4 to `true, false, false, and true`, respectively.

The purpose of this problem is to lead you to a way of solving 2SAT efficiently by reducing it to the problem of finding the strongly connected components of a directed graph. Given an instance I of 2SAT with n variables and m clauses, construct a directed graph $G_I = (V, E)$ as follows.

- G_I has $2n$ nodes, one for each variable and its negation.
- G_I has $2m$ edges: for each clause $(\alpha \vee \beta)$ of I (where α, β are literals), G_I has an edge from from the negation of α to β, and one from the negation of β to α.

Note that the clause $(\alpha \vee \beta)$ is equivalent to either of the implications $\overline{\alpha} \implies \beta$ or $\overline{\beta} \implies \alpha$. In this sense, G_I records all implications in I.

(a) Show that if G_I has a strongly connected component containing both x and $\overline{x}$ for some variable x, then I has no satisfying assignment.

(b) Now show the converse of (a): namely, that if none of G_I's strongly connected components contain both a literal and its negation, then the instance I must be satisfiable. (*Hint*: Assign values to the variables as follows: repeatedly pick a sink strongly connected component of G_I . Assign value `true` to all literals in the sink, assign `false` to their negations, and delete all of these. Show that this ends up discovering a satisfying assignment.)

(c) Conclude that there is a linear-time algorithm for solving 2SAT.

Solution:

(a) Suppose 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.

(b) 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.

(c) Let φ be a formula acting on n literals $x_1, \ldots, x_n$. 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 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 $O(\log(n))$.

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 $\Pi = \{S_1, S_2 \dots S_j\}$ 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 $S_1 \dots S_j$ 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).