CS 170 HW 5

Due on 2018-02-26, at 11:59 pm

1 (★) Study Group

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

2 (★★) Minimum Spanning Trees (short answer)

(a) Given an undirected graph $G = (V, E)$ and a set $E' \subset E$ briefly describe how to update Kruskal's algorithm to find the minimum spanning tree that includes all edges from E'.

(b) Assume you are given a graph $G = (V, E)$ with positive and negative edge weights and an algorithm that can return a minimum spanning tree when given a graph with only positive edges. Describe a way to transform G into a new graph G' containing only positive edge weights so that the minimum spanning tree of G can be easily found from the minimum spanning tree of G'.

(c) Describe an algorithm to find a maximum spanning tree of a given graph.

Solution:

(a) Assuming E' doesn't have a cycle, add all edges from E' to the MST first, then sort $E \backslash E'$ and run Kruskal's as normal.

(b) We can use the attempted fix to Dijkstra's Algorithm described in problem 3 of discussion 3. We create G' by adding a large positive integer M to all the edge weights of G so that each edge weight is positive. The minimum spanning tree of G' is then the same as the minimum spanning tree of G.
Unlike Dijkstra's algorithm, which is finding minimum paths which may have different numbers of edges, all spanning trees of G must have precisely $|V| - 1$ edges, conserving the MST. So, if the minimum spanning tree of G has weight w, the minimum spanning tree of G' has weight $w + (|V| - 1)M$

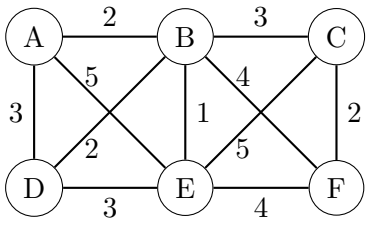
(c) Negate all edge weights and apply the algorithm from the previous part.

3 (★) Prim's Algorithm

A popular alternative to Kruskal's algorithm is Prim's algorithm, in which the intermediate set of edges X always forms a subtree, and S is chosen to be the set of this tree's vertices. We can think of Prim's algorithm as greedily processing one vertex at a time, adding it to S. The pseudocode below gives the basic outline of Prim's algorithm. See the book for a detailed example of a run of the algorithm.

```
S = {v}
X = {}
While S ≠ V:
    Choose t ∈ V\S, s ∈ S such that weight(s,t) is minimized
    X = X ∪ {(s,t)}
    S = S ∪ {t}
Return X
```

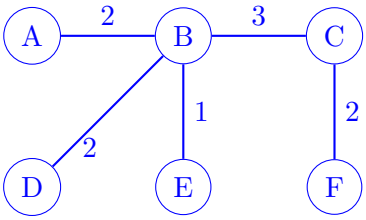
(a) Run Prim's algorithm on the following graph, starting from A, stating which node you processed and which edge you added at each step .



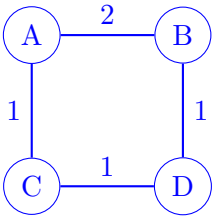
(b) Prim's algorithm is very similar to Dijkstra's in that a vertex is processed at each step which minimizes some cost function. These algorithms also produce similar outputs: the union of all shortest paths produced by a run of Dijkstra's algorithm forms a tree. However, the trees they produce aren't optimizing for the same thing. To see this, give an example of a graph for which different trees are produced by running Prim's algorithm and Dijkstra's algorithm. In other words, give a graph where there is a shortest path from a start vertex A using edges that don't appear in any minimum spanning tree.

Solution:

(a) In the form *vertex* : *edge*, the processing was [A, B : (A,B), E : (B,E), D : (D,E), C : (B,C), F : (C, F)]
The resulting MST is:



(b)



The MST is $\{(A,C),(C,D),(B,D)\}$, but the shortest path from A to B is through the weight 2 edge.

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_2, ... , 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) $mlog(n)$ 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

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

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

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

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

5 (★★) Horn Formulas

Describe a linear-time algorithm to find a satisfying assignment for a Horn formula, if it exists.

Solution:

Main Idea The only place where the algorithm from class does not seem efficient is in the loop where we repeatedly identify an unsatisfied implication and set a variable to true to satisfy it. More precisely, we identify an implication such that every variable in the conjunction on the LHS (left hand side) of the implication is set to true, but the variable on the RHS (right hand side) is false in the current truth assignment. To satisfy the implication, we

then set the RHS variable to be true. The challenge is to keep track of which new implications become unsatisfied as a result of setting this variable to true.

We note that a simple way to solve this problem is to keep a count, for each implication, of the number of false variables on the LHS. When we set a variable x to true, we decrement the count for every implication such that x occurs on the LHS of that implication. When the count for an implication decreases all the way to 0, we check whether the RHS is set to false. If so, the implication is unsatisfied.

Algorithm

In the algorithm below, $I_1, I_2, \ldots, I_m$ indicate the implications and $N_1, N_2, \ldots, N_l$ the negative clauses of a horn formula with variables $x_1, x_2, \ldots, x_n$.

HORN SOLVER($I_1, I_2, \ldots, I_m, N_1, N_2, \ldots, N_l$):

1. Initialize an array of linked lists, $Implications$, of length n
2. Initialize an array $Counts$ of length m
3. Set all variables to False and initialize an empty queue Q
4. For $i = 1$ to m:
5. Set $Counts[i] = 0$
6. For each variable x_j that appears in the left hand side of I_i:
7. Increment $Counts[i]$ and add i to $Implications[j]$
8. If $Counts[i] = 0$, add the variable on the right hand side of I_i to Q
9. While Q is not empty:
10. Remove x_j from Q
11. If x_j is False:
12. Set x_j to True
13. For each implication I_i in $Implications[j]$:
14. Decrement $Counts[i]$ and if $Counts[i] = 0$ add the variable on the right hand side of I_i to Q
15. If all negative clauses are satisfied, report the assignment
16. Else report "unsatisfiable"

Proof of Correctness We note that once a variable is set to true its truth value is never again changed. Now a simple proof by induction (on the number of times through the main loop) shows that the count associated with each implication is the number of false variables on the LHS in the current truth assignment. If follows that an implication is unsatisfied if and only if its count is 0 and the variable on its RHS is currently set to false, thus justifying this approach for identifying unsatisfied clauses.

If a more detailed, line by line proof of correctness is desired, we prove that for each iteration of the while loop starting on line 9, the algorithm either does nothing or picks some unsatisfied implication and sets the variable on its right hand side to True, and that the while loop does not end unless all implications are satisfied. This is accomplished by proving the following loop invariants for the while loop:

At the beginning of each iteration,

- For each $i \leq m$, exactly $Counts[i]$ of the variables on the left hand side of I_i are set to False
- If some implication I_i is unsatisfied, then the variable on its right hand side is in Q

- If a variable is in Q then either it is already set to True or it is on the right hand side of some unsatisfied implication

Note that all three of these conditions are met before the first iteration of the while loop. Assume for induction that they hold true at the beginning of some iteration of the while loop, and we will show that they still hold at the beginning of the next iteration. Let x_j be the variable removed from Q. If x_j is already set to True, then our job is easy because nothing changes except that x_j is removed from the queue (which does not violate the second invariant since no implication with x_j on its right hand side can be unsatisfied).

Suppose instead that x_j is set to False at the beginning of the iteration. Then the algorithm first sets x_j to True. This means that all formulas with x_j on their right hand side are now satisfied, so it is okay to remove x_j from Q. Also, all formulas with x_j on their left hand side should now have their counts decremented by one, which is just what the algorithm does. Finally, if an implication became unsatisfied when x_j was set to True then it must be the case that x_j was on its left hand side and that x_j was the only variable still set to False on its left hand side. In which case, by the first invariant the implication must have had a count of 1. So the algorithm must have set the count to 0 after setting x_j to True and thus added the variable on the right hand side of the implication to Q. Thus the second invariant is still satisfied. And since this is the only way that the algorithm adds variables to Q, the third invariant is still satsified as well.

By induction, all three invariants hold on every iteration of the while loop. So at the beginning of every while loop, whatever variable is removed from Q is either already set to True or is on the right hand side of some unsatisfied implication (by the third invariant). And if the while loop ends then by the second invarant, no implications are unsatisfied.

Running Time Building the arrays in the first portion of the algorithm takes time proportional to the length of the horn formula, since for each variable appearing in each clause we do a constant amount of work. Since a variable is never changed from True to False, a variable is only processed by the while loop at most once (it may be removed from the queue more than once, but every time after the first it is ignored). And when a variable is processed, the work done is proportional to the number of implications it appears in. So the total amount of work done to process variables in the while loop is at most proportional to the length of the formula. And since each implication adds at most one item to the Queue (since its count can only become 0 once and it only has one variable on its right hand side), the number of iterations of the while loop is at most the number of implications. Finally, checking all the negative clauses takes time at most proportional to the length of the horn formula. Thus the entire algorithm is linear in the length of the formula.

6 (★★) Money Changing Redux

During discussion section, we saw a simple greedy algorithm to try to find change that adds up to a given number. We saw that the greedy algorithm didn't find the optimal solution in all cases. In this problem, we will use our newly-found powers of computer science to fix this.

Recall that in the money-changing problem, we were given a fixed set of positive integers called *denominations* $x_1, x_2, \ldots, x_n$ (think of them as the integers 1, 5, 10, and 25). The problem you want to solve for these denominations is the following: Given an integer A,

express it as

$$A = \sum_{i=1}^{n} a_i x_i$$

for some nonnegative integers $a_1, \ldots, a_n \geq 0$.

(a) You might remember that we can represent any integer k in *unary* form by repeating k consecutive 1s (e.g., 3 is represented by 111 in unary). Assume you are given a positive integer A and a set of denominations $x_1, x_2, \ldots, x_n$ in unary form. Give a fast algorithm to solve the money-changing problem.

(b) If you are given A and $x_1, x_2, \ldots, x_n$ in binary, does your algorithm still run in polynomial time? Why or why not?

Solution:

(a) Main idea: We create a dynamic programming algorithm where, for each $n \leq A$, we will find the minimum number of coins needed to sum to n.

Pseudocode:

```
Denominations = [ x1, x2, ..., xk ]
min_coins(0) = 0
for n = 0 to A:
    run & store min_coins(n)
return min_coins(A)

min_coins(n){
    if n < 0:
     return infinity
    min = infinity
    for each x in Denominations:
        if min > min_coins(n - x) + 1:
            min = min_coins(n - x) + 1
    return min
}
```

Proof: We prove by strong induction on the size of A that the algorithm either returns the optimal solution (if it exists) or "infinity" (if it doesn't exist). Base: If $A == 0$, the algorithm returns 0. If $A < 0$, the algorithm returns "infinity". Inductive: Assume the proof holds for all $k < A$. If A doesn't have a solution, then for all $A - x_i$, the algorithm returns "infinity", by the induction hypothesis. If A has a solution, the optimal solution must contain some x_i. By the induction hypothesis, the algorithm returns the optimal solution on $A - x_i$ and thus the optimal solution on A. Thus the algorithm works for all A.

Runtime: This algorithm runs through each of k denominations once for all $n < A$, so the runtime is $O(kA)$.

(b) The algorithm becomes exponential. If A is given in binary form, then the value of A is exponential in the size of the binary representation of A. Since the solution has to solve the money-changing problem for at most every number less than A, this takes exponential time in the representation of A.