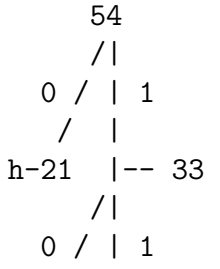
CS 3510 Honors Algorithms
Solutions : Homework 6

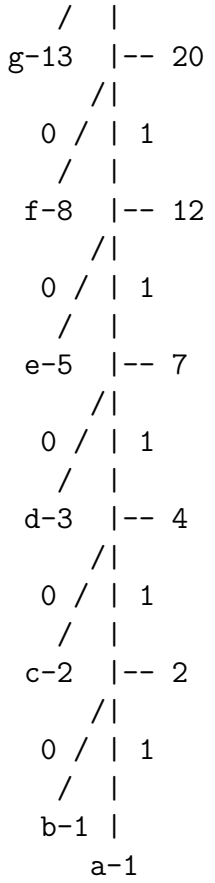
Problem 1: The algorithm is very similar to the Huffman code that we have seen in the class. Pick the smallest three frequencies, join them together and create a node with the frequency equal to the sum of the three. Repeat. However, notice that every contraction reduces the number of leaves by 2 – we remove 3 nodes and add 1 back. So to make sure that we end up with just one node, we have to have an odd number of nodes to start with. If not, add a dummy node with 0 frequency to start with.

Correctness : We shall show that any optimal tree has the lowest three frequencies at the lowest level. Suppose not. We could switch a leaf with a higher frequency from the lowest level with one of the lowest 3 leaves, and obtain a lower average length. Without any loss of generality, we can assume that all the three lowest frequencies are the children of the same node. (if they are at the same level, the average length does not change irrespective of where the frequencies are)

Now, observe that we can treat the contracted leaves as a new character with frequency equal to the sum of the frequencies of the three characters. By a similar reasoning to what was given in the class for the binary Huffman codes, we can see that the cost of the optimal tree is the sum of the tree with the three symbols contracted and the eliminated mini tree which had the nodes before contraction. Since it has been proved that the mini tree has to be present in the final optimal tree, we can optimize on the tree with the contracted node.

Problem 2: The optimal code for the given frequencies will be given by the following tree.





Verify that the above tree is correct (**Exercise**). To generalize for the first n Fibonacci numbers, notice that the sum of the first $n-2$ Fibonacci numbers is $F_n - 1$ where F_n denotes the nth Fibonacci number. (**Exercise:** Prove that $\sum_{i=0}^{n-2} F_i = F_n - 1$ using induction).

So the tree for the first n Fibonacci numbers will be of the above form, one long branch from the root to the lowest leaf, and branches of length 1 hanging from it. This is because, after we contract the first $n-2$ numbers into a node, the total frequency of that node will be $F_n - 1$ which is more than F_{n-1} (which is the least remaining) and less than F_n (the second least remaining). So this node will combine with F_{n-1} and a similar argument will follow by induction for all n.

Problem 3:

(a) The decision version of the problem is the following.

Given graph G, does there exist a simple path from u to v of length greater than or equal to k in G?

This decision problem is in **NP**. The proof is the path itself. Given a simple path from u to v of length greater than or equal to k, one can verify the decision problem in polynomial time. It is easy to test if the given path is a valid path (check for the validity of all the edges in the path), and if the path is simple (one can keep marking all the vertices in the path, and check if all the vertices appear at most once) in time $O(|V|)$. The length of the path can also be checked in $O(|V|)$ very easily. Thus there is a polynomial time verifier and thus the above problem is in **NP**.

(b) The decision problem is the following.

Given graph G, is there a way to color G with k or less colors such that no adjacent nodes have the same color?

The proof is the coloring itself. The verification process, given the coloring, will traverse each edge, and check if this is a valid coloring. This verifier will run in time $O(|E|)$ and hence is polynomial in the size of the input. So the problem is in **NP**.

Problem 4: By defintion, if a problem A is in **NP**, then there exists a polynomial time verification algorithm $V()$ and a polynomial $p()$, such that x is valid solution for A if and only if there exists a string y such that $|y| \leq p(|x|)$ and $V(x, y)$ accepts in polynomial time. If x is not a valid solution, then for any y, where $|y| \leq p(|x|)$, $V(x, y)$ does not accept.

```
Program A(x)
    for all strings y with |y| <= p(|x|)
        if ( V(x,y) = yes), return yes
    endfor
    return no if never returned yes
end
```

The above program calls V, $O(2^{p(|x|)})$ number of times. But V runs in polynomial time. Since the exponential term dominates the polynomial term

in the O-notation, the above program runs in time $O(2^{p(n)})$ where n is the length of the input.

Note : The class of problems which can be solved in time $O(2^{p(n)})$ where n is the length of the input, are said to be in the explonential complexity class, denoted by **EXP**.

Problem 5: This problem is **NP**-complete. Clearly MINIMUM-LEAF-SPANNING-TREE is in **NP**because given a spanning tree with at most k leaves, we can check in polynomial time if that is a spanning tree and if it has at most k leaves.

Now we shall show that this is **NP**-complete, by reduction from HAMILTONIAN CYCLE. Given graph G, for which we wish to check if a Hamiltonian cycle exists, we a construct a MINIMUM LEAF-SPANNING-TREE problem that asks if a spanning tree with $k = 2$ or fewer leaves exists in a modified graph G'. G' is constructed as follows : G' will have all the nodes and edges that G has, but we will also choose an arbitrary node v in G, and create a duplicate v'. v' is a duplicate of v in the sense that for every edge $\{v, u\}$ in G, $\{v', u\}$ will be an edge in G'. We also add two extra nodes w and w' with edges $\{w, v\}$ and $\{w', v'\}$ that connect them to v and v' respectively. Note that the reduction is polynomial time.

We have to show that G has a Hamiltonian cycle **if and only if** G' has a spanning tree with 2 or fewer leaves. To see the if part, notice that is G has a Hamiltonian cycle, then there exists a path from v to v that covers all the vertices. Since v' is a duplciate of v, this implies a path from v to v' in G', and we can add the two additional edges to make a path from w to w' that visits all the nodes in the graph G'. This path is a spanning tree with only 2 leaves, w and w'. So a Hamiltonian cycle in G implies a spanning tree in G' with 2 leaves.

To see the other direction, assume that G' has a spanning tree T with 2 or fewer leaves. Assume that (we will show later) T has a simple path from w to w' that covers all the vertices in G'. From this path, we can remove the starting edge $\{w, v\}$ and the ending edge $\{v', w'\}$, then we have a simple path from v to v'. Since v' is a duplicate of v, this implies a simple path which starts and ends in v and visits all the vertices in G. This is a Hamiltonian cycle in G.

The only thing that is left to be proven is that, if G' has a spanning tree T, then there must be a simple path from w to w' that visits all the vertices in G'. If T is a spanning tree, then there must be a unique path from w to

w' in T. We can show that if this path does not include all the vertices in G' then T must have atleast 3 leaves. Suppose that there is a node z that is not in the path from w to w' in T. The path from w to w' must diverge from the path from w to z in T. Let it diverge at x. Clearly, x has degree atleast 3 in T, and if we remove x, then we get 3 distinct components of T, and hence 3 leaves. This contradicts the fact that T has only 2 leaves, and hence there is no such z which the path from w to w' does not include.

Problem 6: Clearly, this problem is in **NP**, once given a valid set of k paths, we can verify in polynomial time that they are vertex disjoint, and are valid paths from each pair (s_i, t_i).

To show that this is **NP**-complete, we can do a reduction from 3-SAT. Given an arbitrary 3-SAT formula, we can construct a network routing problem, such that the formula is satisfiable if and only if the routing problem has a solution. For each clause C_i in the formula, we create a source/sink pair (s_i, t_i), and a node x_j^i for each literal x_j in the clause (for a negated literal $\bar{x_j}$, we add a node $\bar{x}_j^i$. We add an edge from s_i to each literal node x_j^i in the clause C_i, and an edge from each literal node x_j^i to t_i. In addition, we create source/sink pair (s_j, t_j) for each variable x_j, we get a total of $k = q+n$ source/sink pairs, where q is the number of clauses and n is the number of literals. For each (s_j, t_j), we will create two paths from s_j to t_j. One will travel through all nodes $\bar{x}_j^i$, which correspond to the positive literal of x_j, and another path which travels through all the literal nodes $\bar{x}_j^i$, which correspond to the negative literal of x_j. This routing problem can be constructed in polynomial time.

The idea is the following : to send a flow from s_i to t_i, it must travel through some literal node, this will correspond to a literal in the original 3-SAT formula that should be set to TRUE to satisfy the clause C_i. The pairs (s_j, t_j) for the variable x_j ensures that a literal and its negation are never both set to true.

We shall now show that if the formula is satisfiable, then there are $k = q + n$ disjoint paths that travel between source and sink pairs. Given the assignment that satisfies the original 3-SAT formula, we can construct the k paths as follows. For each (s_j, t_j) pair that representing a variable x_j, we choose the path that corresponds to the literal that is not set to true in the satisfying assignment (if x_j is set to TRUE, then we use the (s_j, t_j) path that travels through $\bar{x}_j^i$ nodes and if x_j is FALSE, then we use the (s_j, t_j) path that travels through x_j^i nodes). In this way, all the literal nodes, set

to TRUE will still be available for the q (s_i, t_i) paths, corresponding to each clause. Since we started off with a satisfying assignment, every (s_i, t_i) pair would be able to find a path through a literal that is set to TRUE. Thus we can cover all the $k = q + n$ source/sink pairs.

Suppose we have k disjoint paths between each source/sink pair, then both the (s_i, t_i) clause pairs and (s_j, t_j) variable pairs are to have disjoint paths between them. This will mean that the clause pairs never use the literal nodes that correspond to both a literal and a negation. If we look at the literals used by the (s_i, t_i) paths, and set all the corresponding literals TRUE, then we will have satisfied all the clauses. The source/sink pairs corresponding to each variable ensures that no literal and its negation are set too TRUE simultaneously. This means that we have a satisfying assignment for 3-SAT.

We now have shown both directions of equivalence, the original 3-SAT instance is satisfiable if and only if the routing problem has disjoint paths between the k source/sink pairs.

Problem 7: Clearly, given a set which is a kernel of the directed graph, we can verify in polynomial time if every vertex is in the kernel or has a directed edge from the kernel. We can also verify that no two vertices in the kernel share a directed edge. So the problem is in **NP**.

To prove that this is **NP**-complete, consider the following reduction from 3-SAT. Consider a 3-SAT instance. For each variable x_j in the 3-SAT instance, put down two vertices in the form of a 2-cycle, as in the question. The two vertices will correspond to the complemented and uncomplemented literals. Notice that of these two vertices, at most one will be in the kernel. If there are no edges directed into these two vertices, then exactly one of these two should be in the kernel.

Now, for every clause C_i, we put down three vertices in a 3-cycle similar to the picture in the question. The vertices correspond to the literals appearing in the clause. Notice that we can have atmost one vertex from every such 3-cycle in the kernel (if we pick more than one, then we will have a directed edge between 2 vertices). To help us cover all the three vertices, we add directed edges into the 3-cycle as follows. For every variable in the clause, we add a directed edge from the corresponding literal in the 2-cycle directed to the corresponding vertex in the 3-cycle. Note that this reduction is polynomial time.

Now, if the 3-SAT instance had a satisfying assignment, we pick the

vertices corresponding to the TRUE literals in each of the 2-cycles. So we pick exactly one vertex from each 2-cycle. Also, every clause has atleast one TRUE literal. For a 3-cycle of the form $x_k \rightarrow x_l \rightarrow x_m \rightarrow x_k$, if x_k is the TRUE literal, x_k would have an incoming edge from the picked vertex in the 2-cycle, and hence is covered. We just pick x_l so that both x_l and x_m are covered. This is a kernel for the graph. So if the 3-SAT instance has a solution, the constructed graph would have a kernel.

If the constructed graph has a kernel, then it is easy to see that there should be exactly one vertex from every 2-cycle, and exactly one vertex from every 3-cycle. The remaining vertex in the 3-cycle has to be covered using an incoming edge from a vertex in the 2-cycle, which is picked in the kernel. So look at the picked literals in the 2-cycle, and set the variables TRUE or FALSE correspondingly. Every variable is set TRUE or FALSE. And we have already noted that one vertex in the 3-cycle has to be covered using a picked vertex (which corresponds to a TRUE literal) in the 2-cycle. So every clause has a satisfying assignment as well. So if the graph has a kernel, then the 3-SAT instance is satisfiable.

So the 3-SAT instance is satisfiable if and only if the corresponding graph has a kernel. So the problem in deciding if a given directed graph has a kernel is **NP**-complete.