Hyuntaek Oh

Amir Nayyeri

CS 325_002

7 Mar 2024

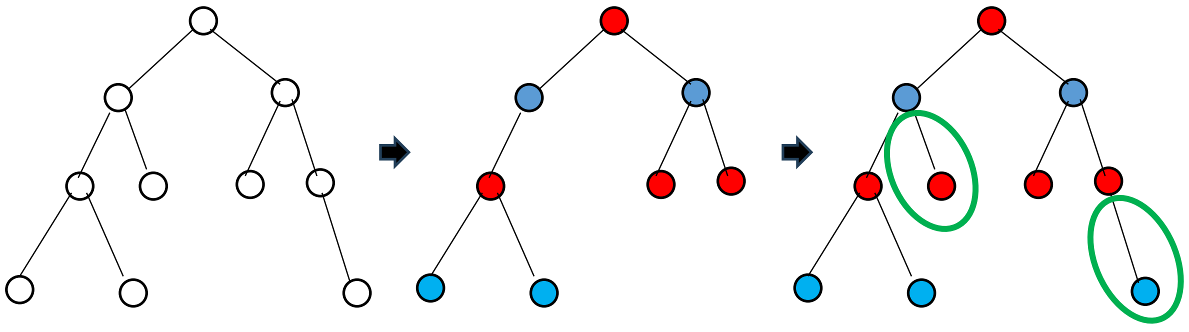
Practice Assignment 4

Problem 1. An undirected graph $G = (V, E)$ is bipartite if the vertices V can be partitioned into two subsets L and R, such that every edge has exactly one endpoint in L and one endpoint in R.

(a) Prove that every tree is a bipartite graph.

As shown in the lecture note, a tree is a connected graph with no cycles, which means the graph is a forest. In the Wikipedia, a bipartite graph is a graph whose vertices can be divided into two disjoint sets and independent sets U and V, that is, every edge connects a vertex in U to one in V.

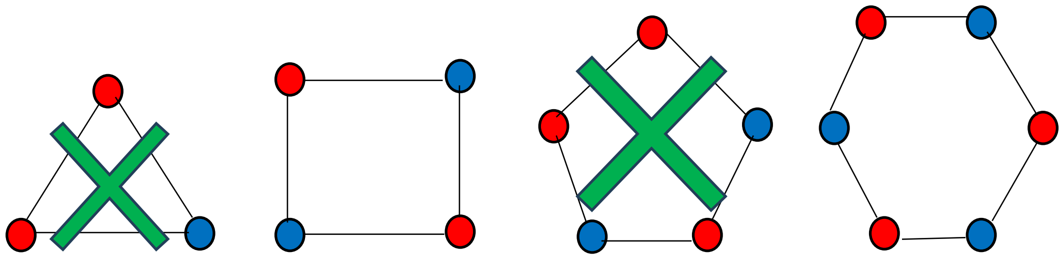
For proof of the problem, one method is to use induction. For example, a tree with 1 or 2 vertices is bipartite. For the inductive step, remove all of the vertices of degree 1. A smaller tree remains, which by the inductive hypothesis can be colored with 2 colors. Then, put all the degree 1 vertices back in, and color each of them the opposite of their neighbor's color.



< **Figure 1.** The proof of a tree that is a bipartite graph >

(b) Prove that a graph G is bipartite if and only if every cycle in G has an even number of edges.

To prove this problem, we can use contradiction. We need to examine that a graph G is not bipartite if and only if every cycle in G has an 'odd' number of edges.



< **Figure 2.** The cyclic graphs have an odd and even number of edges>

As shown in the Figure 2, a graph has an ‘odd’ number of edges is not bipartite since there are some vertices has same color with adjacent vertices. By using these cases above as contradictions, a graph G is bipartite if and only if every cycle in G has an even number of edges.

(c) Describe and analyze an efficient algorithm that determines whether a given undirected graph is bipartite.

We can use BFS algorithm to find out whether a given graph is bipartite or not. At the first stage, a vertex can be a starting point and be colored with red color, putting into set U. Then, the all the neighbor vertices near starting point are colored with blue color, putting into set V. After that, all the vertices that are near the neighbors are colored red again, putting into set U. While assigning colors, if we find a neighbor which is colored with same color as current vertex, then the graph cannot be bipartite.

This algorithm runs in O(V+E) time complexity, where V is the number of vertices and E is the number of edges. The BFS algorithm involves traversing the entire graph once.

Try to work on this problem as you read about graph search algorithms.

Problem 2. Describe and analyze an algorithm to compute an optimal ternary prefix-free code for a given array of frequencies $f[1 \ldots n]$. Don’t forget to prove that your algorithm is correct for all n. This is a good exercise to ensure that you understand Huffman codes. What you get here should be very similar to the Huffman code; try to modify each step/proof of Huffman codes to work for ternary codes instead of binary codes.

# Description and Analysis

For solving this problem, we can use a priority queue Q and initialize it with the frequences from the given array $f[1 \dots n]$. If Q is not empty, the lowest three nodes are removed from Q, and a new node for the sum of the frequencies of the three nodes removed is created. Then, this new node is inserted into the Q. This ternary Huffman tree can be traversed in a depth-first manner, assigning a code to each leaf node. Because of ternary tree, each node has three children, so assign 0, 1, and 2 as the codes for left, middle, and right child, respectively. From root to a specific leaf node can be unique ternary prefix-free code.

# Proof of Correctness

As mentioned above, ternary Huffman tree has prefix-free property, which means each symbol is unique, so we need to check optimality of it. We can use induction to prove the correctness of ternary Huffman tree. Let's suppose that T is ternary prefix-free tree generated by the algorithm above, and T' is any other ternary prefix-free tree, then the weighted path lengths of T are less than or equal to the weighted path lengths of T'.

The base case is when there are only three nodes, the algorithm always chooses the three nodes with the lowest frequencies because of the assumption above. In inductive steps, we will use extra depth to compare the weighted path lengths in T and T', the lowest frequencies nodes in T has smaller value than those in T'. It means that one of the nodes in T is smaller than one of the nodes in T'. We can guess the weighted path lengths of T are less than or equal to one of T'. Thus, like binary prefix-free tree, ternary prefix-free tree in the algorithm is optimal.

Problem 3. For each of the following statements, respond True, False, or Unknown.

(a) If a problem is decidable then it is in P.

In the textbook "Algorithms", P is the set of decision problems that can be solved in polynomial time. Intuitively, P is the set of problems that can be solved quickly. However, every decision problem in P is also in NP. It is not necessarily guaranteed that it is P if a problem is decidable.

$\therefore False$

(b) For any decision problem there exists an algorithm with exponential running time.

EXP (also called EXPTIME), which is significantly larger complexity class, is the set of decision problems that can be solved in exponential time, using at most 2^{n^c} steps for some constant c>0. Every problem in PSPACE, NP, and P is also in EXP.

$\therefore True$

(c) $P = NP$.

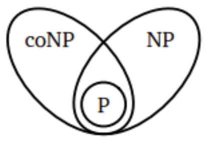


Figure 12.3. What we *think* the world looks like.

As shown in Figure 12.3, a part has a common field between P and NP whereas P and NP also have their own fields, which means it depends on the problem. If the problem is NP-hard and it can be solved in polynomial time, then P = NP.

∴ *Unknown*

(d) All NP-complete problems can be solved in polynomial time.

Every problem in NP is polynomially reducible to SAT, and SAT is reducible to every NP-hard problem. Since the set of NP-complete problems is a subset of NP, it follows that they are all solvable in polynomial time.

∴ *True*

(e) If there is a reduction from a problem A to $Circuit\ SAT$ then A is NP-hard.

If Circuit SAT (known NP-hard) could be reduced to problem A in polynomial time, then it would be proven NP-hard. The NP-hard problem must reduce to the problem we are solving. In this case, we do not know if problem A is NP-hard or not.

∴ *False*

(f) If problem A can be solved in polynomial time, then A is in NP.

If problem A can be solved in polynomial time, then it is a part of the P subset of NP. All items that are a part of P are also a part of NP.

∴ *True*

(g) If problem A is in NP, then it is NP-complete.

For a problem to be NP-complete, it has to be proven NP-hard and in the subset of NP. In this case, we know about problem A is in NP, but do not know about the problem A is NP-hard.

∴ *False*

(h) If problem A is in NP, then there is no polynomial time algorithm for solving A.

Problem A is in NP, which means that there is a possibility to solve the problem with polynomial time algorithm.

$\therefore False$