Problem1: 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.

1. Prove that every tree is a bipartite graph.

To show that a graph is bipartite, we must divide the vertices into two sets L and R so that two vertices in the same set are adjacent.

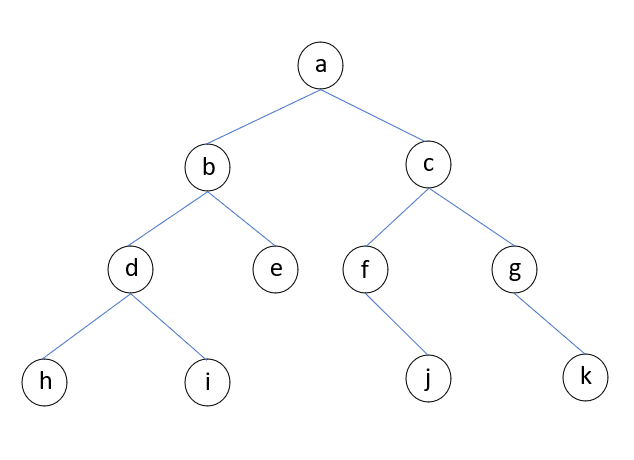
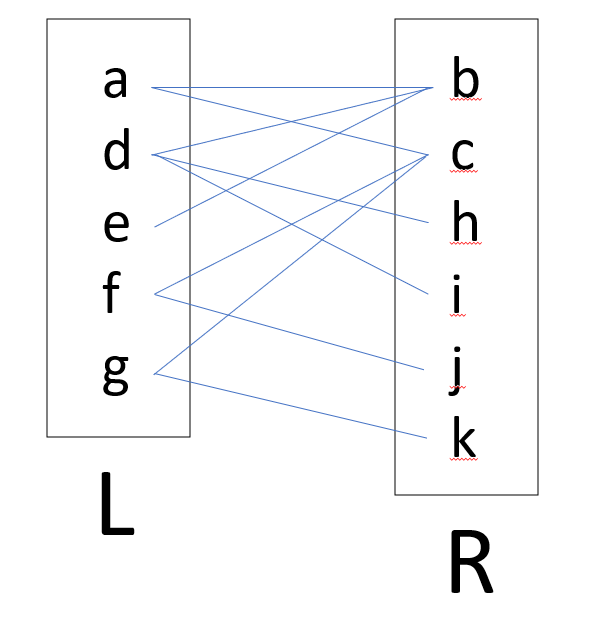
1. Designate any vertex as the root, and put the vertex in set L.

2. Put all of the children of the root in set R, None of these children are adjacent (they are siblings).

3. Put into L every child of every vertex in R (every grandchild of the root).

4. Keep going until all vertices have been assigned one of the sets, alternating between L and R every generation.

That is, a vertex is in set R if and only if it is the child of a vertex in set L.

fig.1 fig.2 (BFS)

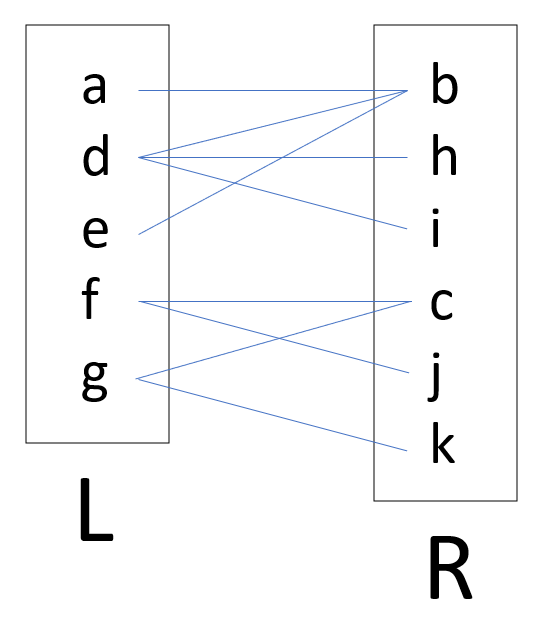
The key to how I partitioned the tree in fig.1 was to know which vertex to assign to a set next. I choose to visit all vertices in the same generation before any vertices of the next generation. This is usually called a Breadth First Search.

There could also be different orders to partition the tree.

1. Start with the root, put in L.

2. Then look for one child of the root to put in R.

3. Find a child of that vertex and put it into L, then find its child and put it into R. Until no children, restart to its parent and see if the parent has other children.

fig.3 (DFS)

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

I will prove problem 1(b) by contradiction.

First, Assuming there exists a graph G is not bipartite but every cycle in G has an even number of edges. (If the graph is not bipartite, then it must contain an odd cycle.)

If G is not bipartite, then there exists at least one cycle with an odd number of edges. This is because in a bipartite graph, all cycles must have an even number of edges. (According to its rule, and I have given the proof in problem1 (c).) Therefore, if G is not bipartite, then it must contain an odd cycle.

Because it is an if and only if problem, let's consider a graph G where the cycle has an even number of edges but is not bipartite. But if G is not bipartite, then it must contain an odd cycle.

That is a contradiction, as I assumed that every cycle in G has an even number of edges. Therefore, the assumption that G is not bipartite must be wrong.

Conversely, if a graph is bipartite, then by the proof, every cycle in the graph has an even number of edges. Then I have prove that "a graph is bipartite if and only if every cycle in the graph has an even number of edges" by contradiction.

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

The most common algorithm to check whether the graph is bipartite is the 2-coloring traversal approach.

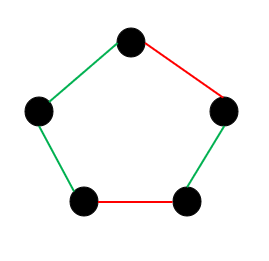
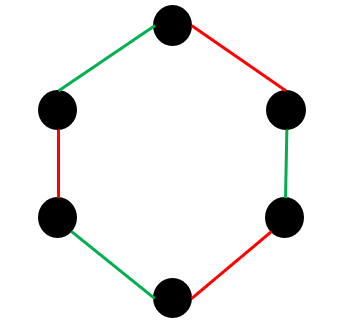
1. Assign color1 to the source vertex (putting into set U).

2. Color all the neighbors with color2 (putting into set V).

3. Color all neighbor's neighbors with color1 (putting into set U).

4. This way, assign color to all vertices such that it satisfies all the constraints of m way coloring problem where m = 2.

5. While assigning colors, if we find a neighbor that is colored with the same color as the current vertex, then the graph cannot be colored with 2 vertices (or the graph is not Bipartite)

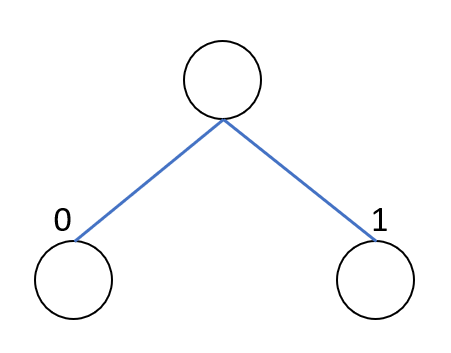
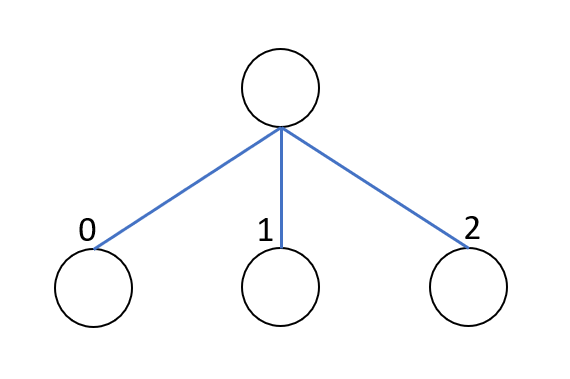
fig.4 (not bipartite) fig.5 (bipartite)

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

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

Problem2: Describe and analyze an algorithm to compute an optimal ternary prefix-free code for a given array of frequencies f [1. . . 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.

To answer this problem, I will need the concept of prefix-free code and the standard Huffman coding algorithm. Then I will modify the binary Huffman code to the ternary Huffman code. (fig.6 to fig.7)

fig.6 (binary)fig.7 (ternary)

This is the algorithm and what I have modified.

1. Initialization: Create a priority queue Q, which contains n nodes, and for each node, containing the symbols and their frequency in the input array f.

2. Build the Huffman Tree: while Q > 1,

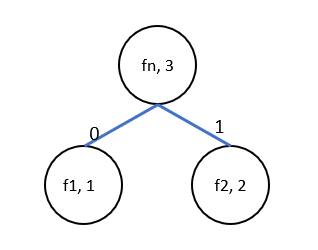
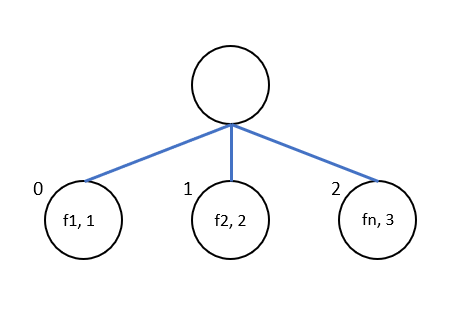
I). Remove the two nodes with the lowest frequencies f1 and f2 from Q.

II). Create a new internal node fn, which new node's frequency fn = f1 + f2.

III). Assign fn as the parent of f1 and f2.

IV). Insert fn back to Q.

(Final purpose like fig.8 to fig.9)

fig.8 fig.9

3. Assign the codewords.

I). Traversing the ternary Huffman tree from the root.

II). Adding "0", "1", and "2" for every child edges from left to right.

III). Record every path from the root as the symbols for codewords.

IV). Output the codewords, ex: "011", "012", "020"...

The correctness of this algorithm.

1. Optimality: My ternary Huffman algorithm has the same optimality as the binary Huffman algorithm because the binary Huffman algorithm always merges the two lowest frequency nodes. My algorithm also selects the two lowest nodes and adds them to be the new subtree, then inserts back them into the original tree, but the height will n - 1. (Same optimality concept with the binary Huffman algorithm)

2. Prefix-free property: The binary Huffman algorithm uses "0" and "1" to construct the Huffman binary tree. My algorithm has the same properties as the Huffman binary tree, I use the ternary tree "0", "1". and "2", which still do not exist on the same path in my tree.

3. Uniqueness: My ternary codewords will have one-only property, which is based on how the Huffman binary tree builds the Huffman codewords, we always merge the lowest frequency nodes, and every frequency is different, just like how we prove the uniqueness for the MST, we select the minimum value in every round to make sure there will not exist another answer.

Therefore, the algorithm correctly computes an optimal ternary prefix-free code for any given array of frequencies f[1...n].

Problem3: For each of the following statements, respond Ture, False, or Unknown.

1. If a problem is decidable then it is in P.

False. We can say that all P questions are decision questions, but not all decision questions are P questions. For example: The world-famous question "The Satisfiability Problem" is a decision question but its running time is non-polynomial.

1. For any decision problem there exists an algorithm with exponential running time.

False. As I prove in problem 3 (b), "The satisfiability Problem" is one of the decision questions, but the running time is unknown, it changes with the length of the formula and different formulas.

1. P = NP.

Unknown. P could be in NP and P also could be not in NP, we have to discuss four topics: P, NP, NP-complete, and NP-Hard. We use NP-complete to prove that P != NP, the concept is for every NP question, transforming NP into NP-complete, and finding the verification in polynomial time, then P will equal NP. So this question is still being resolved, because we also can not find the prove that no NP-complete exists.

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

False. NP-complete is a subset of NP questions, which is simplified from NP in polynomial time. However, we still don't know whether the NP question can be solved in polynomial time, for example: The Traveling Salesman Problem. But if the NP-complete question can be solved in polynomial time, which means all the NP questions can be solved in polynomial time, which brings us back to the P = NP question, which we don't know the answer.

1. If there is a reduction from a problem A to Circuit SAT then A is NP-hard.

True. If problem A is simplified from SAT, and SAT is proven to be an NP-Hard question, this means A is included in an NP-Hard domain.

1. If problem A can be solved in polynomial time then A is in NP.

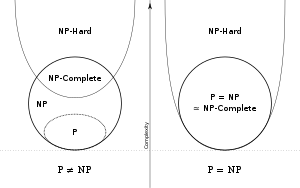
True. If problem A can be solved in polynomial time, which is the P question, it is a deterministic algorithm (doing one thing in a time), and NP non-deterministic algorithm needs a machine can solve many things in a time, so it can solve the P question. So problem A also can be an NP question.

1. If problem A is in NP then it is NP-complete.

False. Not all the NP questions belong to NP-complete. NP-complete is a subset of NP questions. The definition of NP-complete is the NP questions that can be simplified within polynomial time. So we can only say that NP-complete belongs to the NP question, but not all NP are NP-complete.

1. If problem A is in NP then there is no polynomial time algorithm for solving A.

False. NP questions contain using algorithms to verify the proposed solution in polynomial time, For example: people used brute-force algorithms to solve the chain matrix multiplication question in the past, in which time complexity was non-polynomial time. But we can use dynamic programming to prove it, and its time complexity will be O(n3).

fig.10